LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

StarT-Voyager: Hardware Engineering Specification (Version 4.0)

Computation Structures Group Memo 385
March 11, 1999

Boon S. Ang and Derek Chiou

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

StarT-Voyager: Hardware Engineering Specification (Version 4.0)

Boon S. Ang and Derek Chiou

March 11, 1999

Contents

Change Log

Date	Description	Author
Changes in Version 0.1		
May 3, 96	Changed the description of actions for "sP Serviced Space" and "Local Global Memory Space" in Section 3.6. There was a mistake where Capture for Notification was not handling the case of full MemQIn or full MemQDataIn properly. Transaction retry is needed in such cases.	Boon Ang
May 3, 96	Changed the description of actions for "sBIU Special ShTx Space" in Section 3.6. Added the requirement that Addr[1] has to be forced to #b0 (to indicate Short Tx Format) before writing it into SRAM.	Boon Ang
May 3, 96	Added "Message Formats", and "NES Function Blocks Interface Signal Descriptions" (Section A) from Technical Specs into this document.	Boon Ang
May 6, 96	Corrected mistakes in the aBIU and sBIU specification of QPtr Space access format. It should be Addr[26] and not Addr[18] which specifies whether to perform update. The new value should be in Addr[19:25] and not Addr[20:26].	Boon Ang
May 6, 96	Corrected mistakes in the sBIU specification of ImmdCmd. The Arctic Acks request ptr should be 8b and not 12b. The aSRAM associated with approval for pending SM op should be 12b, in sPAddr[17:28].	Boon Ang
May 6, 96	Removed sPAddrP from sBIU interface. sBIU is never a bus master and therefore does not require to output addresses. Since we are not checking address parity, there is no need for these signals.	Boon Ang
May 6, 96	Corrected mistakes in sBIU specification, where I referred to the aSRAM-tmp address as sSRAM-tmp address. The convention is to name this address according to the SRAM that it is serving as temporary buffers. A similar confusion and mistake was also corrected.	Boon Ang
May 6, 96	Also put in some comments about specification conventions, *e.g.*, the prefix to signals under Internal Organization. These should make the document more readable.	Boon Ang
May 27, 96	Changed the organization of Communication Group (ComGp) encoding	Derek Chiou
May 29, 96	Changed the QPtr address space organization for sP to reduce the number of required pages. Moved VAS/VAS* differentiation bit to bit 26 and shifted everything else up.	Derek Chiou
Jun 24, 96	Changed the QPtr address space organization for aP to match sP.	Derek Chiou

Date	Description	Author
Changes from Version 0.1 to Version 1.0		
Jul 6, 96	Fairly extensive changes are made to the address spaces. The NES slave space is pruned down to 1GB, a restriction imposed by the MPC105. The interleaving of QPtr space with SRAM space is removed. This allows the sP to map as much of the NES space with as few BATs/pages as possible. Overall functionality is of the NES is not changed. Some details of the changes follow:	Boon Ang
Jul 10, 96	CompGp encoding is changed. First, the field has been shifted to Addr[6:9]. While CTRL's resources are still kept to contiguous locations, the arrangement of queues within that space is changed. This is motivated by the new encoding for User Space/ShTx Space.	Boon Ang
Jul 10, 96	NES slave space is prunned to 1GB on the sP-bus. NES slave space remains at 1.5GB on the aP-bus. Notice that the "User Space" and "Command Space" on the two sides are now offset by 1GB. This is to allow sP Serviced Space to continue to be a contiguous, aligned 1GB space.	Boon Ang
Oct 15, 96	Modified the aBIU section on how it responds to Snooped Space transactions.	Boon Ang
Changes from Version 1.0 to Version 2.0		
Oct 31, 96	Added CTRL SysReg information.	Boon Ang
Nov 19, 96	The NES Mastered aP-bus Op's SRAM Addr field is shifted left by 1b to bits[4:13] instead of the previous bits[5:14]. This is necessary as bit 14 is fixed as 0 for this operation. There was an inconsistency in the previous specification. This changed is reflected in Figure 19 and accompanying text.	Boon Ang
Nov 19, 96	Modified specification of ClsSRAM Update in Seciton 2.2.3. sP can now request a ClsSRAM update with one bus transaction, while maintaining FIFO ordering with commands in MemQOut. aBIU now has to decode bits 14, 16 and 18 to determine what command it is getting from MemQOut. (bit 18 is new)	Boon Ang
Nov 19, 96	Added a "CompGp" for Overflow queue. 1110 was still available, so that was used. User level access to status pointer through QPtr space is not available with this CompGp, only access is through Cmd Space Config Q State Access (or SysReg). To accomodate this, the space dedicated to status pointers is doubled, i.e. qptr-base is 1 bit smalled, and Addr[6] is now concatenated to form the address as well. These changes are reflected in Sections 2.5.3, 2.5.8, 3.6.3, 3.6.6.	Boon Ang
Nov 19, 96	While making the previous change, I found a mistake in Section 2.5.8. Previously, sPAddr[6,16] were not included in the check to determine which are the states with dedicated shadow copies in the NES SRAM's. Corresponding mistake in Section 3.6.6 is also amended.	Boon Ang

Date	Description	Author
Nov 19, 96	Added a secondary opcode (#b110, the last one with the current number of bits) which toggles locking of aP-bus. See Table 66.	Boon Ang
Nov 24, 96	Deleted a paragraph from the description of Short Tx Format in Section 5.2. This was old description that became inconsistent (and was superseded by other description) with the change in Tag-on size.	Boon Ang
Nov 25, 96	Added more detailed description of EPRAM and QRAM states in Sections 4.1.1 and 4.1.2.	Boon Ang
Nov 26, 96	Changed DMA data packet format, moving the BusOps to the end of the packet.	Boon Ang
Nov 26, 96	Added description of short message format when it is placed into the Overflow queue. See Section 5.3	Boon Ang
Changes from Version 2.0 to Version 3.0		
Jan 14, 97	Added ImmdCmd for clearing ApprovalReg and moved the "opcode" for ImmdCmd for resetting DMA RxData queue ptr in aBIU. See Table 15.	Boon Ang
Jan 16, 97	Changed the way SRAM address for PCP queues and Pas/Vas Tag-On queues are computed.	Boon Ang
Jan 16, 97	Changed ComGp encoding, basically moving what used to be in bit 7 back to 9, and shifting what used to be in [8:9] to [7:8]. Changes are reflected in Tables 24 and 5. This change is done in order to "undo" the non-linearity in external addresses (aPAddr and sPAddr) and internal addresses (CtrlAddr, aBIUAddr and sBIUAddr). This undoing is reflected in Table 46. The change causes other Tables to be updated too, namely, Tables 12, 13, 15, 30, 31, 33.	Boon Ang
Jan 16, 97	Extensively changed aBIU and sBIU internal state addressing. The previous addressing had some problems and wouldn't have worked.	Boon Ang
Jan 17, 97	Incrementing aBIU's DMA-DRx queue pointer is now explicitly indicated in the commands received by the aBIU over the internal bus. Bit[0] is used to indicate this. #b1 indicates increment.	Boon Ang
Jan 17, 97	Augmented DMA Rx Mastered Op Cmd to allow option of clearing ApprovalReg. bit[10] indicates whether to clear. See Figure 21.	Boon Ang
Jan 17, 97	Clarified that "done" for ApprovalReg "Give Approval" Cmd. This first turns the ApprovalReg state to "Ready", and is not considered done until this state changes to "Free".	Boon Ang
Changes from Version 3.0 to Version 3.1		
Jan 22, 97	Removed "Queue disable" from the list of ImmdCmds that go to CTRL. Tables 15 and 33 are updated to reflect this.	Boon Ang
Jan 25, 97	Added section on hardwired SRAM addresses. See Section 4.4.	Boon Ang

Date	Description	Author
Jan 25, 97	Changed ApprovalReg SRAM address to be 10b (See Section 5.6.1), and clarified the alignment of address within 32B SRAM buffers used by the aBIU to capture or supply data to aP-bus transactions. (See Section 3.3.3)	Boon Ang
Jan 25, 97	Added Ack format. See Section 5.6.3.	Boon Ang
Jan 25, 97	Added specification for unit of allocation (increment) for MemQDIn: 64B, with data occupying the first 32B.	Boon Ang
Changes from Version 3.1 to Version 3.2		
Feb 10, 97	Changed the region of aBIU Snooped Space (aka Local Global Memory Space). See Tables 20 and 29. (Note that the latter originally had an error, which is now rectified.)	Boon Ang
Feb 10, 97	Changed the ClsSRAM Update Cmd so that it only updates one cache-line's state at a time. See Table 10. This also frees up bits, so that it is now possible to specify whether the command is to be acknowledged. Consequently, SysReg 9 in sBIU is no longer needed. That is removed from the listing in Section 4.2.2.	Boon Ang
Feb 10, 97	Added ability for sP to interrupt aP, and while we are at it, we also added the ability for aP to interrupt sP. These are added to the ImmdCmd that goes to CTRL, since CTRL is currently the only one generating interrupts. See change in Tables 15 and 33.	Boon Ang
Feb 10, 97	Changed format of Captured aP-bus Transaction. See Table 69 and Figure 24.	Boon Ang
Feb 10, 97	Corrected error in the way Overflow queue status is read. Overflow queue status is automatically updated in the SRAMs but its address is not configuous with those the normal queues. The modification is reflected in: Section 3.6.6 and 2.5.8.	Boon Ang
Feb 10, 97	Changed the way BIU SysReg is read. Value of 4 SysReg is packed into a 4B word. See Sections 4.2.2 and 4.3.2. Also added HAL-Statistics register. See registers 12-15 in Section4.3.2. See Sections3.3.2 and 3.6.10 to see when HAL-Statistics register is updated.	Boon Ang
Jun 7, 97	Found an error in ComGp encoding in Tables 5 and tab:abiu-com-gp-std-enc. Encoding for "aBIU's copy of Pas/Vas-3 ComGp;" should be 1011 and not 1101.	Boon Ang
Changes from Version 3.2 to Version 4.0		
Sep 2, 98	Fixed a number of typographical errors. Updated for changes to the DMA Read and Send formats. Several Ctrl SysReg's specifications were inaccurate. These were fixed. In general, effort to maintain this document has been restricted to Sections 4 and 5 which contain information critical to users of the machine. Details of implementations, in particular, are mostly out of date.	Boon Ang

Please note that parts of this specification are out dated. We started writing this specification at the beginning of the project as a specification for the hardware implementers. The StarT-Voyager NES designed evolved over time, and the hardware implementers made many design changes to the internal functional partitioning and internal interfaces. Only the externally visible components of this specification have been kept up to date.

Chapter 1 is an accurate overview of the message passing support. It is missing description of the shared memory support; please refer to CSG Memo 388[2] for that.

Only the first three sections of both Chapters 2 and 3 are kept up to date. The remaining parts of these chapters are implementation suggestions that served as design starting points but are not accurate accounts of the final design.

Chapters 4 and 5 are kept up to date.

Chapter 6 is empty; for information on CTRL, please refer to Rosenband's Master's thesis[8].

My PhD thesis[1] provides a more abstract description of the StarT-Voyager NES design.

– Boon S Ang, March 11, 1999.

1 Introduction

StarT-Voyager is a parallel system designed to support both message passing and coherent distributed shared memory. StarT-Voyager is constructed from a collection of commercial SMPs connected by a fast network which interfaces to each SMP's coherent memory bus via a Network Endpoint Subsystem (NES) card. Each SMP, referred to as a *site*, is a desktop-class, commercial, dual-PowerPC 604 SMP. It uses a stock PC/workstation class motherboard which can accommodate two processor cards. Each processor card normally contains one 604 processor and an in-line L2 cache. For StarT-Voyager, we replace one of the two processor cards in each SMP site with an NES card, making each site into a uniprocessor system[1] This 604 processor, referred to as the application processor, or *aP*, runs a copy of AIX, augmented with a parallel layer to coordinate parallel job execution.

Each NES card is attached to the Arctic network, a Fat-tree [4] network constructed with Arctic router chips[3]. Both the Arctic router chip and the network boards are designed and implemented at / as part of the StarT-Voyager project. Each Arctic router chip is a 4x4 packet-switched router with support for variable length packets, virtual-cut-through, FIFO ordering, and two logical networks. The Arctic network employs hardware link-level flow-control to provide a lossless packet routing service. Each link is 16 bit-wide and runs at 80MHz to achieve 160 MBytes/sec bandwidth. With two links, one in each direction, between each NES card and the network, each site has a peak network communication bandwidth of 320 MBytes/sec. A fast router together with the Fat-tree topology allows the Arctic network to provide redundancy[2], the ability to exploit network locality and very high bisection bandwidth – over 5 GBytes/sec in a 32 node machine.

Figure 1 shows the organization of a StarT-Voyager site, with details of the NES. The NES manages several regions of memory, each providing a different functionality. The NES is designed for both speed and flexibility. The NES Core provides dedicated control hardware and data paths for the most common operations. Actions that are expected to occur very frequently are handled *completely* in hardware and are thus very fast. An embedded processor, referred to as the service processor (*sP*) provides flexibility and extensibility. The sP is organized with its own sub-system comprising of off-the-shelf parts: a PowerPC 604 processor, a memory controller, and DRAM. Design complexity is further reduced by the symmetry of the organization: from the NES Core's perspective, the sP-subsystem is almost identical to the aP complex. The sP subsystem, however, has additional functionality. The sP observes all aP memory operations to one of the NES address regions called the *sP Serviced Space*, and is capable of initiating transactions on the aP-bus, controlling any NES state, and handling all exceptional situations. These features are extremely flexible and can be used to extend the functionality of StarT-Voyager. They will be used to implement coherent shared memory and *non-resident message queues* (see Section 1.1.5). Both the aP and sP have access to the same message passing mechanisms, but each has its own set of resources. Two banks of dual-ported SRAM, the aSRAM and sSRAM, provide storage space for resident message queues.

The NESCtrl, a small ASIC, implements all the hardware control functionality. It also holds all the control state, including the queue pointers of message queues. The I-bus, the central data path of the NES core over which all messages flow, has a 64 bit-wide data path running at 50MHz. This roughly matches both the bandwidth of the 60X bus which runs at the same speed and has the same data path width, and the 320 MBytes/s aggregate bandwidth between an NES and the

[1] We desired multiple processors per site, but currently no such PowerPC 604 motherboard exists at a reasonable price.

[2] Multiple paths through a Fat-tree network between most pairs of source-destination sites provide redundancy and hence fault tolerance.

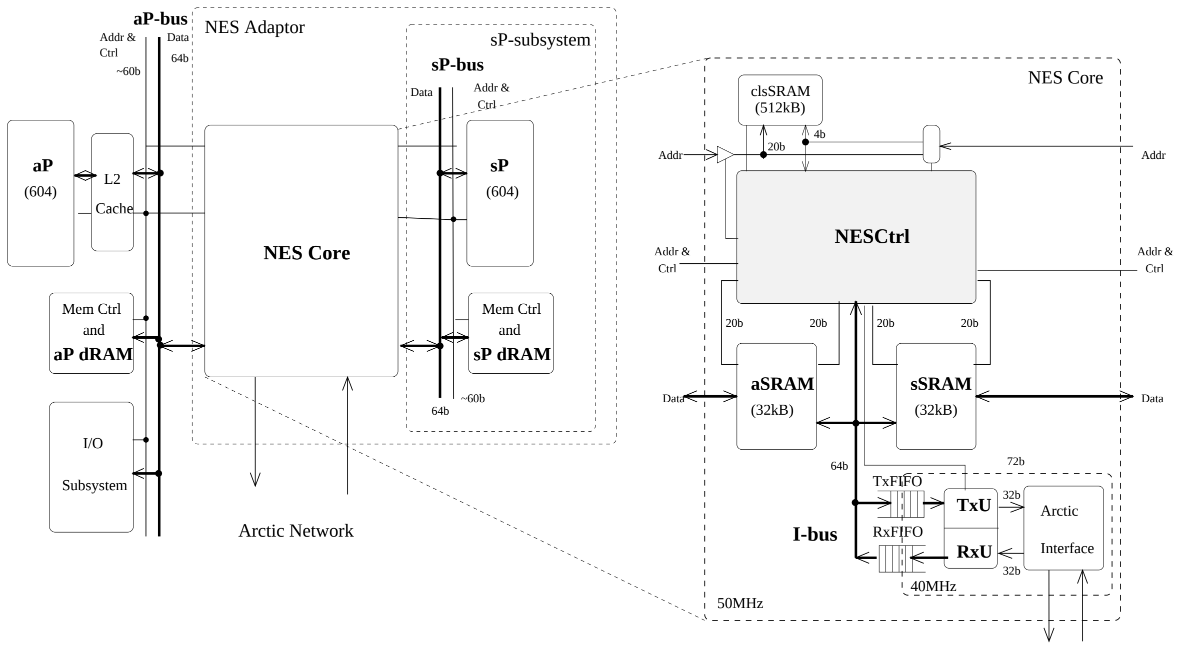

Figure 1: A StarT-Voyager site.

Arctic network. Outgoing messages undergo destination address translation (See Section 1.1.5) and are tagged with a source identifier in the Transmit Unit (TxU), which also computes and appends a CRC to the message packet. This CRC is checked at each Arctic stage, and again in the Receive Unit (RxU) at the destination NES. The TxU is also responsible for the expansion of an Express message (Section 1.1.2) into an Arctic packet, while the RxU performs the compression. Finally, the Arctic Interface handles low-level signaling and analog electrical conversions (TTL to ECL) between the NES and the Arctic Network itself. The clsSRAM, which stores cache-line state bits in coherent distributed shared memory implementations, is not involved in message passing.

1.1 Message Passing Support

Messages of a variety of types occur naturally in parallel and distributed applications. The most obvious variation is the amount of the data transferred in a message. The data size is likely to be small for control messages amounting to no more than a few bytes, *e.g.* when conveying a simple request or an acknowledgment. On the other hand, block data transfers on the order of many kilobytes of data are also common, especially with coarse-grain parallel applications or when supporting a parallel file system. Another variation concerns the location of the message data. In general, for the smallest sized messages, the processor registers are the source and desired destination. As message size increases, caches are the source and desired destination. For really large messages, main memory DRAM becomes the most likely repository of source data, and the desired destination. The most significant performance measure also varies with message size. Low per-message processor overhead and communication latency are significant for small messages. As message size increases, maximizing bandwidth and minimizing amortized processor overhead become increasingly important. Thus, message size affects engineering design tradeoffs.

Cache-coherent distributed shared memory (CCDSM) protocols are examples of parallel programs that require a variety of message types. Experimental flexibility in protocol implementation is important because protocols are still an active area of research. Consequently, many CCDSM machines implement the protocols in software on embedded processors which communicate via message passing. In CCDSM protocols, requests for a cache-line, invalidations and acknowledgments are small register-based messages, containing only an address, and a message type identifier. A cache-line data transfer from one node to another involves medium sized messages. Entire pages of data may be migrated as program locality changes, giving rise to large messages.

While few will disagree that a spectrum of message passing sizes is needed to support general parallel processing, most available parallel machines today provide only a single message passing mechanism. For messages that are larger than the native message size, software has to packetize, send and reassemble the message, thereby increasing processor overhead and latency. Messages that are smaller than the native message size are often aggregated into larger messages in order to amortize the overhead; such aggregation, however, increases the granularity of parallelism. Though a single message passing mechanism can emulate message passing of all sizes, it is clear that *a single mechanism cannot do so efficiently.*

StarT-Voyager is designed to support efficiently a complete spectrum of message passing requirements by providing four message passing mechanisms – Basic, Express, Tag-On, and DMA messages. The design facilitates protected communication among multiple users and provides the underlying support for coherent distributed shared memory.

1.1.1 Basic Messages

StarT-Voyager's Basic Message mechanism provides protected access to Arctic's native packet transfer service. With a 32-bit header specifying the logical destination queue and other options, and a variable payload of between four and twenty-two 4-byte words, a Basic Message is ideal for communicating an RPC request, or performing a small scatter/gather transfer. Microprocessors are optimized for communicating data to/from registers and caches.

The Basic Message interface consists of separate transmit and receive queues, each with a cacheable message buffer region, and *uncached* producer and consumer pointers for exchanging control information between the processor and the NES. The message buffer region is arranged as a circular FIFO with the whole queue visible to application software, enabling concurrent access to multiple messages. The producer and consumer pointers are mapped into uncached space to reduce latency. An uncached write pointer update immediately triggers NES processing. We do not implement cached producer and consumer pointers because of attendant design complexity and limited benefits.

The four steps of composing a Basic Message are shown in Figure 2. The Basic Message transmit code first checks to see if there is sufficient buffer space to send the message. The software maintains a copy of the producer and consumer pointers (P, C in top half of figure) to minimize reading them from the NES. This copy of the consumer pointer is updated again only when it indicates that the queue is full. By then, space may have freed up since the pointer was last read from the NES. The NES may move the consumer pointer at any time if it launches any messages, an event which occurrs in our example between steps 1 and 2. If the transmit code finds that enough buffer space is available, the message is written into the cached buffer space, with the write effectively going to the processor cache (step 2). Next, the processor issues `clean` instructions to write the modified cache-lines to the corresponding NES buffer locations (step 3). Due to PowerPC's weak memory model, a barrier instruction is required after the `clean` instructions and before the producer pointer is updated via an uncached write. This write (step 4) prompts the NES to launch the message, after which the NES frees the buffer space by incrementing the consumer pointer. With NES Reclaim,

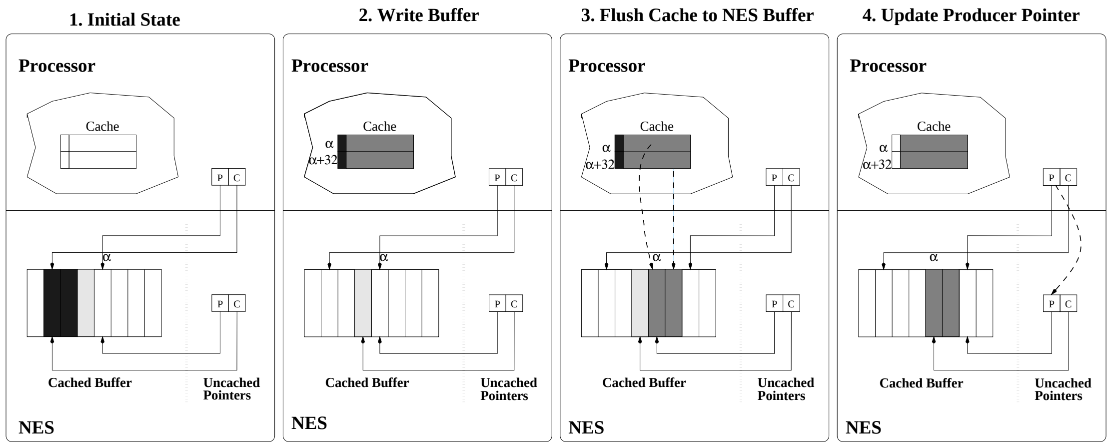

Figure 2: Sending a Basic Message

the NES issues `clean` bus operations to maintain coherence between the processor cache and the NES buffers, eliminating the need for the processor to issue `clean` instructions (but not the `sync`) in step 3. In this case, the pointer update will cause the NES to reclaim the message, then launch it.

The interface is designed primarily for message reception by polling, although an application can request for an interrupt upon message arrival. This choice is available to a receiver on a per receive queue basis, or to a sender on a per message basis. When polling for messages, an application compares the producer and consumer pointers to determine if there is any message. Messages are read out directly from the message buffer region. Coherence maintenance is again needed so that the application does not read a cached old message. This can be done either explicitly by the processor with `flush` instructions or by NES Reclaim.

A unique aspect of the Basic Message buffer queue is its memory allocation scheme. Buffer space in this queue is allocated in cache-line granularity and both the producer and consumer pointers are cache-line address pointers. Allocation in smaller granularity is undesirable because of the coherence problem caused by multiple messages sharing a cache-line. The other obvious choice of allocating maximum-sized buffers was rejected because it does not work well with either software prefetching of received messages, or NES Reclaim. The main problem is not knowing the size of a message until the header is read. Therefore, both prefetching and a simple implementation of NES Reclaim must either first read the header, and then decide how many more cache-lines of data to read, or blindly read all three cache-lines. The former introduces latency while the latter wastes bandwidth. With cache-line granularity allocation, every cache-line contains useful data, and data that is fetched will either be used for the current message, or subsequent ones. Better buffer space utilization is another benefit of this choice.

1.1.2 Express Messages

Messages with a minimal amount of data are common in many applications for synchronization or communicating a simple request or reply. Basic Messages, with a cached message buffer space, is a bad match because the bandwidth of burst transfer is not needed while the overhead of coherence maintenance, weak memory model, and explicit handshake remain. Express Messages are introduced in StarT-Voyager to cater to such small payloads by utilizing a single uncached access

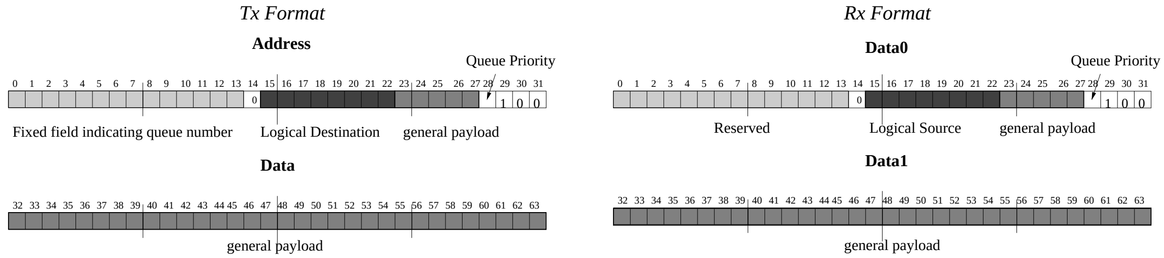

Figure 3: Express Message Formats

to transfer all the data of a message and thus avoid the overheads of Basic Messages.

A major challenge of the Express Message design is to maximize the amount of data transported by a message while keeping each compose and launch to a single, uncached memory access. The Express Message Mechanism packs the transmit queue specifier, message destination and 5 bits of data into the *address* of an uncached write. The NES automatically transforms the information contained in the address into a message header and appends the data from the uncached write to form an Arctic message. A diagram showing a simplified format for sending and receiving Express Messages is given Figure 3. Additional address bits can be used to convey more information but consumes larger (virtual and physical) address space and can also have a detrimental effect on TLB if the information encoded into the address bits do not exhibit "good locality". (Alternate translation mechanisms such as PowerPC's block-address translation mechanism[6] may be employed to mitigate this problem but depends on both processor architecture and OS support.)

Unlike Express Message transmits, address bits cannot be used to convey message data to the processor when receiving an Express Message. To reduce the data read by a receive handler, the NES reformats a received Express Message packet into a 64 bit value as illustrated in Figure 3. Like an Express Message transmit, an Express Message receive can be accomplished with a 64 bit uncached read into an FPR. The data is subsequently move into GPRs via (cache) memory[3]. Alternatively, two 32 bit loads into GPRs can be issued to receive the message.

Unlike Basic Messages, the addresses for accessing Express Message queues do not specify specific queue entries. Instead, the NES provides a FIFO push/pop interface to transmit and receive Express Messages. When a processor performs a write to enqueue the message, the NES provides the necessary buffer address to put that message into NES SRAM. Likewise, when the processor performs a read to receive a message; the NES provides the necessary SRAM address from which to read the message. Speculative loads from Express Message receive regions are disabled by setting the page attributes appropriately[4]. When an application attempts to receive a message from an empty receive queue, a special Empty Express Message, whose content is programmable by system code, is returned. If message handler information is encoded in the message, such as in Active Messages[9], the Empty Message can be treated as a legitimate message with a "no action" message handler. Unlike for receiving, the use of producer and consumer pointers is required for transmit queues to ensure that they do not overflow.

Express Message transmit and receive queues appear as hardware FIFO with application software addressing a queue, but not a particular queue buffer location. Uncached writes to a transmit queue has the effect of pushing data into the queue, *i.e.* the producer pointer is implicitly in-

[3]PowerPC does not support 8 Byte load into a pair of GPRs nor direct data transfer between a FPR and a GPR.
[4]This involves asserting the guarded bit in PowerPC architecture.

cremented. Similarly, uncached reads pop data off a receive queue, implicitly incrementing its consumer pointer.

1.1.3 Tag-On Messages

The Tag-On Message mechanism extends the Express Message mechanism to allow additional data from NES SRAM to be appended to an out-going message. The Tag-On Message mechanism was designed to eliminate a copy if message data was already in NES SRAM's. It is especially useful for implementing coherent shared memory protocol, and for multi-casting a medium sized message. As composed by an application, a Tag-On Message looks similar to an Express Message with the addition that several previously unused address bits now specify the SRAM location where the additional 32 Bytes (or 64 Bytes) of message data can be found. At the destination NES, a Tag-On Message is partitioned into two parts that are placed into two separate queues. The first part is its header which is delivered like an Express Message, via a queue that appears to be a hardware FIFO. The second part, made up of the data that is "tagged on", is placed in a separate buffer similar to a Basic Message receive queue which utilizes explicit buffer deallocation.

Tag-On Messages have the advantage of decoupling the header from the message data, allowing them to be located in non-contiguous addresses. This is useful in coherence protocol when shipping a cache-line of data from one site to another. Suppose the sP is responding to another site's request for data. In StarT-Voyager, this is achieved by the sP first issuing a command to move the data from aP-DRAM into aSRAM, followed by a Tag-On Message that ships the data to the requester. In addition to the cache-line of data, a Tag-On Message inherits the 37 bit payload of Express Messages which can be used in this instance to identify the message type and the address of the cache-line that is shipped. Cache-line data may also be brought into the NES without the sP asking for it, for example the aP's cache may initiated a write-back of dirty data. In such cases, Tag-On Message's ability to decouple message header and data allows the data to be shipped out without further copying.

Tag-On Messages are also useful for multi-casting. To multi-cast some data, an application first moves it into NES SRAM. Once that is done, the application can economically send it to multiple destinations using a Tag-On Message for each one. Thus, data is moved over the system memory bus only once at the source site, and the incremental cost for each destination is an uncached write to indicate a Tag-On Message.

1.1.4 One-poll

In order to minimize the overhead of polling from multiple receive queues, StarT-Voyager introduces a novel mechanism, called *One-poll*, which allows one polling action to poll simultaneously from a number of Express Message receive queues as well as Basic Message receive queues. A single uncached read specifies in its address the queues to poll from and obtains the highest priority Express Message. If the highest priority non-empty queue is a Basic Message queue, a special Express Message that includes the Basic Message queue identifier and its queue pointers is returned. If there are no messages in any of the polled queues, a special Empty Express Message is returned. One-poll is useful to user applications, most of which are expected to have four receive queues: high priority Basic Message, low priority Basic Message, high priority Express and Tag-On Message, and low priority Express and Tag-On Message. The sP has nine queues to poll. It is clear that the one-poll mechanism will dramatically reduce polling costs.

TxQ	Logical Dest 0	Logical Dest 1	...	Logical Dest n
0	$\langle$site$_A$, RxQ$_\beta$, source$_i\rangle$	$\langle$site$_B$, RxQ$_\delta$, source$_j\rangle$	...	$\langle$site$_C$, RxQ$_\zeta$, source$_k\rangle$
1	$\langle$site$_A$, RxQ$_\theta$, source$_l\rangle$	$\langle$site$_E$, RxQ$_\iota$, source$_m\rangle$	...	$\langle$site$_F$, RxQ$_\lambda$, source$_n\rangle$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	$\langle$site$_G$, RxQ$_\nu$, source$_o\rangle$	$\langle$site$_H$, RxQ$_\pi$, source$_p\rangle$	...	$\langle$site$_I$, RxQ$_\sigma$, source$_q\rangle$

Figure 4: Destination Table: each row contains the logical destination to $\langle$site, RxQ, source$\rangle$ mapping for a transmit queue.

1.1.5 Protection and Multiuser Concerns

StarT-Voyager was designed to support a multiuser, multitasking, loosely gang-scheduled environment. To achieve this goal, StarT-Voyager employs a combination of protection and translation mechanisms and provides multiple message queues at each site.

Specifically, each StarT-Voyager site has 512 pairs of transmit/receive queues. In a standard allocation scheme each process of a parallel job is assigned its own unique set of message passing queues[5]. No queue is assigned to two independent processes simultaneously. Access to a local queue is controlled by the virtual memory mapping of the processor, *i.e.*, only specific queues are mapped to a process's virtual memory space. Furthermore, a process is allowed to send messages to other processes (i.e., enqueue messages in remote receive queues) only if it is so specified in a predetermined *Destination Table*.

A message packet header must specify both the destination site and destination queue at the site. Rather than addressing physical sites and queues, the application code uses a logical destination specifier, which identifies an entry in the Destination Table (see Figure 4). The NES translates logical destination specifiers to the appropriate site/queue pair. Since the translation is also a function of the local transmit queue, only specific transmit queues can send messages to specific receive queues[6].

The NES also attaches a source identifier to each outgoing message. The destination process can reply to a message using the source identifier. For symmetry in translation the source identifier must look like a logical destination specifier at the destination. Such a specifier is also kept in the Destination Table to facilitate attaching of the source identifier (see Figure 4). The logical destination specifiers should have the same meaning across all of the processes in a single parallel job, allowing the passing of logical destination specifiers to other processes. The Destination Table is configured through a separate NES address space and should be accessible only to system software. Thus an application must be granted permission to access remote queues prior to initiating any communication.

With this support, efficient protected multitasking can be implemented easily. A running parallel job can be suspended and "swapped out" by suspending and swapping out each of its constituent processes independently. Since each receive queue is assigned to at most one process, messages can arrive for a process that is swapped out; swapping out a parallel process requires no global coordination, though it may impact performance. Such freedom dramatically reduces the parallel job swapping time needed by traditional MPP's. Traditional MPP's such as the CM-5[5] support only

[5] The smallest allocation unit is two transmit and two receive queues.

[6] An inverse table could be associated with each receive queue so that only messages from a specified set of sources will be accepted. Such additional hardware would protect against untrusted OS or sP code at a source. StarT-Voyager does not implement this additional hardware.

a single set of network queues, forcing the queues as well as the network to be entirely swapped out at the same time a parallel job is being swapped out.

Logical destination translation also enables processes to migrate from one site to another and from one set of receive queues to another. Moving processes to other sites and receive queues is useful if a parallel job must be moved to a smaller space partition, say to adapt to the failure of a site. Several processes of the same job can be mapped to the same site, each with its own set of message passing queues.

Other machines use a Parallel Job Identifier (PJID) which is added automatically to every outgoing message and verified at the destination[7]. Such schemes define monolithic protection domains in which every process can communicate with every other process in the domain. While suitable for parallel applications, it does not offer enough flexibility in a distributed environment, where a server application may want to communicate with a number of client applications without allowing a client to communicate directly with every other client. StarT-Voyager's logical destination mechanism supports such non-uniform communication patterns in a protected fashion.

The large number of queues at each site are far too many to be supported directly in hardware. Each NES supports 16 pairs of transmit and receive queues (8 Express/Tag-On and 8 Basic) directly in hardware. The hardware queues act as a software-managed cache for the 512 pairs of transmit and receive queues. The queues mapped to hardware queues are called *resident* while the others are called *non-resident*. When a message packet arrives at an NES, the NESCtrl determines if the destination queue specified in the message header is resident and if it has enough room for the message. If both conditions are true, the NESCtrl enqueues the message accordingly . Otherwise, the NES enqueues the message in the *miss queue* serviced by the sP. There are several ways to implement non-resident queues. One method is to map the buffer space into the aP's DRAM and the producer-consumer pointers into sP service space. Messages can continue to be received into and sent from non-resident queues with the sP's assistance. The sP implements the transition of a queue between resident and nonresident status when directed by the OS.

1.2 Shared Memory Support

For a description of shared memory support in Start-voyager, please see CSG Memo 388[2].

2 sBIU

The aBIU and sBIU share many common parts. We have, nevertheless, documented them separately so that the specification of each can be read independently. Readers who read both specifications will find some repeated information.

2.1 sBIU Functionality Briefs

The sBIU provides the interface between the sP-bus and the rest of the NES. Unlike the aBIU, the sBIU is only *a slave* on the sP-bus, and *never a master nor a snooper.* Through its slave space, the sBIU provides the following functionalities to the sP.

1. sSRAM read/write.
2. aSRAM read/write.
3. MsgQPtr Status read and Control update. (If possible, we will place the control for "Msg Rx by interrupt" in this space, thus allowing user code to directly set or clear it.)
4. ShTx writes, including to MemQOut-0 and MemQOut-1.
5. ShRx reads, including from MemQIn.
6. ImmdCmd and Config Space read/update.
7. Cls write.

These functionalities are described in greater detail when we discuss address space in Section 2.2.

2.2 sP-bus Address Space

The sP-bus address space is divided into three major regions:

Size	From Address	To Address	Description
1GB	`#x00000000`	`#x3FFFFFFF`	sP Host Memory (dRAM) Space
1GB	`#x40000000`	`#x7FFFFFFF`	sBIU (slave) Space
2GB	`#x80000000`	`#xFFFFFFFF`	PCI/ISA/ROM/Flash ROM Space (Probably not used, except for ROM/Flash ROM)

Although we are not having a PCI bus, we nevertheless leave the PCI space unused by other functionalities. This design is imposed by the MPC105 PCI Bridge cum Memory Controller used in the sP subsystem which constrains any local bus slave like the NES in our design to the 1GB of space specified above.

The sBIU slave space is divided into several regions to provide the functionalities itemized in Section 2.1. The division does not always achieve contiguous address ranges due to the need to enforce protection, *i.e.* functions that should be protected separately must be on different pages. An easy way to understand the address space division is to view the addresses in the sBIU slave space as "instruction encoding". Using this view, the Primary Opcode is in sPAddr[0:3] and has the encoding as shown in Table 1 [7].

[7] A reminder to anyone changing the encoding: keep in mind which of the sPAddr bits are unprotection. The 12b in sPAddr[20:31] is always unprotected as they lie within a page (page size is 4kB). If we are using a BAT for translation, at least 17b (sPAddr[15:31]) are unprotected as the minimum BAT size is 128kB. Bit fields that differentiate between different protection domain must lie outside unprotected sPAddr bits.

Width (bits)	Bit field	Content	Description
4	[0:3]	00XX	sP dRAM Space.
		010X	sBIU Special ShTx Space.
		0110	sBIU User Space.
		0111	sBIU Cmd Space.
		1XXX	PCI Space.

Table 1: sBIU sP-bus address space Primary Opcode encoding.

2.2.1 sBIU User Space

The sBIU User Space uses a 3b wide Secondary Opcode, in sPAddr[4:6] and has the encoding as shown in Table 2. One can think of sPAddr[4] as distinguishing between SRAM Space (cacheable) and the other spaces (uncacheable), sPAddr[5] differentiates QPtr Space from the ShTx and ShRx spaces, and sPAddr[6] differentiates between ShTx and ShRx Spaces.

Width (bits)	Bit field	Content	Description
3	[4:6]	0XX	SRAM Space.
		10X	QPtr Space.
		110	ShTx Space.
		111	ShRx Space.

Table 2: sBIU sP-bus User Space Secondary Opcode encoding.

SRAM Spaces

Width (bits)	Bit field	Content	Description
4	[0:3]	0110	==== Fixed Field ====
1	[4]	0	==== Fixed Field ====
11	[5:15]	X	** Unused **
1	[16]	0	sSRAM Space.
		1	aSRAM Space.
13	[17:29]	X	bits[0:12] of SRAM address.
2	[30:31]	00	==== Fixed Field ====

Table 3: SRAM Space sPAddr encoding.

The complete address encoding for accesses to aSRAM or sSRAM space is shown in Table 3. This table, which is an sPAddr encoding table that is used in many other parts of this document, has four types of entries:

- A "==== Fixed Field ====" is one where the bit-pattern is fixed at the value specified under the "Content" column.

- A "** Reserved **" field is one whose value is not used by the specified address space, but is utilized by other spaces that have to work togather with the space concerned. These fields are not available for arbitrary usage.
- In contrast, "** Unused **" fields are not currently used in the address space, and can be used to encode information should the need to include further information arises.
- The last category are fields which are used by the address space to encode variable information. The meaning of these fields are described in the table itself.

In the case of accesses to SRAM Space, sPAddr[16] distinguishes between accesses to sSRAM (#b1) and aSRAM, while sPAddr[17:29] specifies the address within the SRAM for a particular transaction.

QPtr Space

The goal of the QPtr Space is to allow user level access to Pcp and Pas/Vas queues' pointer state in CTRL. The chosen interface encoding allows it to be implemented with almost the same hardware as that used for system level access to NESCtrl chip's state (See Table 12 in Section 2.2.3). *QPtr Space provides a small optimization over the Cmd Space's Config Q State Access capability: a Read bus transaction to update Control Pointers can at the same time read the Status Pointers.* (Control Pointers are: TxQ's P-ptr and RxQ's C-ptr; Status Pointers are: TxQ's C-ptr and RxQ's P-ptr.)[8]

Table 4 summarizes the encoding for QPtr Space. The sPAddr[6:9] field encodes the ComGp that the QPtr access is refering; the actual encoding is described in Table 5. Since QPtr Space is only interested in providing access to state in CTRL, and only to Pcp and Pas/Vas queues, sPAddr[6] is constrained to be #b0. The Fixed Fields aPAddr[10:11,14:16] are constrained to select the correct fields as defined by the queue state map in Section 4.

For Pcp comp gp, a 4B odd word (sPAddr[29] == 1) Read to the QPtr Space returns the Status Pointers associated with the specified ComGp: TxL's C-ptrs in data[1:7], TxH's C-ptr in data[9:15], RxL's P-ptrs in data[17:23], RxH's P-ptr in data[25:31]. (See Table 48 in Section 4.)

In addition, the other address bits may encode a request to update one of the control pointers. sPAddr[26] indicates whether there is an update associated with an access (4B Read or Write). If there is, other fields come into play. sPAddr[19:25] specifies the new pointer value. sPAddr[27:28] indicates which queue's pointer to update, and in conjunction with sPAddr[17] indicates which pointer to update. Note that sPAddr[27:28] and sPAddr[17] play no part in the data read, only in the update. A 4B even word (sPAddr[29] == 0) Read returns data that contains no information that should be of interest to user code. The functionality is similar for Pas/Vas CompGp's state in CTRL, but in addition, a 4B even word (sPAddr[29] == 0) Read returns the RxL* P-ptr in data[17:23] and the RxH* P-ptr in data[25:31].

Not all combinations for identifying the pointer to update are valid, *e.g.*, sPAddr[17] == 1 for a Pas TxQ is not valid. It is an error on the part of software to specify updates to such addresses. Nevertheless, *the hardware has to be designed in such a way that such errors will neither cause damage to the hardware, nor place the NES hardware into an unrecoverable state that requires reset and/or power-cycle.*

Note the following unusual semantics of accesses to the QPtr space: (a) the data part of writes to the QPtr space is discarded; (b) both read and write can update pointers and all the information needed to achieve this are encoded in sPAddr.

ShTx Space

[8] Table 25 shows where the queue state for each CompGp resides.

Width (bits)	Bit field	Content	Description
4	[0:3]	0110	==== Fixed Field ====
2	[4:5]	10	==== Fixed Field ====
1	[6]	0	==== Fixed Field ====
3	[7:9]	X	ComGp.
2	[10:11]	01	==== Fixed Field ====
2	[12:13]	X	** Unused **.
3	[14:16]	100	==== Fixed Field ====
1	[17]	0	In the case of a Pcp TxQ, update P-ptr; for Pcp RxQ, update C-ptr; for Pas TxQ, update P-ptr; for Vas RxQ, update C-ptr.
		1	In the case of a Pcp TxQ or RxQ, update R ptr; for Vas RxQ, update Data C-ptr. This is undefined for a Pas TxQ.
1	[18]	X	** Reserved **.(To maintain compatibility with state access in Cmd Space.)
7	[19:25]	X	New ptr value.
1	[26]	0	No Update.
		1	Update.
1	[27]	0	Tx queue.
		1	Rx queue.
1	[28]	0	L Priority queue.
		1	H Priority queue.
1	[29]	X	For 32b reads, specifies which word is read.
2	[30:31]	00	==== Fixed Field ====

Table 4: QPtr Space sPAddr encoding.

Width (bits)	Bit field	Content	Description
4	[6:9]	`0000`	CTRL's copy of Pas/Vas-0 ComGp;
		`0001`	CTRL's copy of Pas/Vas-1 ComGp;
		`0010`	CTRL's copy of Pas/Vas-2 ComGp;
		`0011`	CTRL's copy of Pas/Vas-3 ComGp;
		`0100`	CTRL's copy of Pcp-0 ComGp.
		`0101`	CTRL's copy of Pcp-1 ComGp.
		`0110`	CTRL's copy of Pcp-2 ComGp.
		`0111`	CTRL's copy of Pcp-3 ComGp.
		`1000`	sBIU's copy of Pas/Vas-0 ComGp; used by sP.
		`1001`	sBIU's copy of Pas/Vas-1 ComGp; used by sP.
		`1010`	aBIU's copy of Pas/Vas-2 ComGp; used by aP.
		`1011`	aBIU's copy of Pas/Vas-3 ComGp; used by aP.
		`1100`	sBIU's copy of MemQ ComGp.
		`1101`	CTRL's copy of Overflow.
		`1110`	aBIU's copy of MemQ ComGp.
		`1111`	aBIU's copy of DMA ComGp.

Table 5: Encoding for ComGp field, bit[6:9] of sPAddr. sPAddr[6] differentiates between state in CTRL (sPAddr[6] == 0) and in one of the BIUs. sPAddr[7] differentiates between Pas/Vas (sPAddr[7] == 0) and Pcp queue. The exceptions to this encoding rules are the MemQ and Overflow ComGp's, which are special ComGp's. sPAddr[8] (roughly) distinguishes between resources devoted to the aP (sPAddr[8] == 1), and those used by the sP. sPAddr[9] distinguishes between even and odd resources.

CompGp	Queue	P-ptr Location	C-ptr Location	P-ptr* Location	C-ptr* Location
Pcp[0-3]	Tx	CTRL (U_w, S_r, S_w)	CTRL (U_r, S_r, S_w)	—	—
	Rx	CTRL (U_r, S_r, S_w)	CTRL (U_w, S_r, S_w)	—	—
Pas/Vas[0-3]	Tx	CTRL (U_w, S_r, S_w) xBIU (U_w, S_w)	CTRL (U_r, S_r, S_w)	—	—
	Rx	CTRL (U_r, S_r, S_w) xBIU (S_w)	CTRL (U_w, S_r, S_w) xBIU (U_w, S_w)	CTRL (U_r, S_r, S_w)	CTRL (U_w, S_r, S_w)
MemQ	MemQOut[0-1]	CTRL (S_r, S_w) sBIU (S_w)	CTRL (S_r, S_w)	—	—
	MemQIn	aBIU (S_w) sBIU (S_w)	aBIU (S_w) sBIU (S_w)	—	—
	MemQDIn	aBIU (S_w)	aBIU (S_w)	—	—
DMA	TxQ	CTRL (S_w)	CTRL (S_w)	—	—
	RxCmdQ	CTRL (S_w)	CTRL (S_w)	—	—
	RxDQ	CTRL (S_w) aBIU (S_w)	CTRL (S_w) aBIU (S_w)	—	—

Table 6: The location of Q-state information, and accessibility from the processors. (S_r: System Read, S_w: System Write; U_r: User Read, U_w: User Write.)

Width (bits)	Bit field	Content	Description
4	[0:3]	`0110`	==== Fixed Field ====
3	[4:6]	`110`	==== Fixed Field ====
1	[7]	`0`	Pas-0
		`1`	Pas-1
20	[8:27]	`X`	Message content.
1	[28]	`1`	H Priority queue.
		`0`	L Priority queue.
3	[29:31]	`100`	==== Fixed Field ====

Table 7: ShTx Space sPAddr encoding.

The encoding for ShTx Space is summarized in Table 7. Because only a subset of the ComGp supports ShTx operations, specifically, the Pas/Vas-X ComGps, and these are statically partitioned between the sP and the aP, the encoding therefore uses only 1b, sPAddr[7] to specify the CompGp.

sP code uses a 32b or 64b uncached write to access this space. In the former case, the address of an uncached 32b write to ShTx space is concatenated with its 32b data to form a 64b quantity, with the address occupying 0:31, and data from 32:63. This 64b quantity is treated by the NES hardware as having the Short Tx Format shown in Figure 12 in Section 5.2. In the case of 64b uncached write, the 64b data is taken as the message specification and can either have the Short Tx Format, or the Fully Specified Format (aka Raw Header). See Section 5.2 for more details of how short messages are transformed by the NES during message send and reception.

ShRx Space

Width (bits)	Bit field	Content	Description
4	[0:3]	`0110`	==== Fixed Field ====
3	[4:6]	`111`	==== Fixed Field ====
8	[7:14]	`X`	** Unused **.
1	[15]	`0`	VasR-0 ComGp: do not poll.
		`1`	VasR-0 ComGp: poll.
1	[16]	`0`	PcpR-0 ComGp: do not poll.
		`1`	PcpR-0 ComGp: poll.
1	[17]	`0`	VasR-1 ComGp: do not poll.
		`1`	VasR-1 ComGp: poll.
1	[18]	`0`	PcpR-1 ComGp: do not poll.
		`1`	PcpR-1 ComGp: poll.
1	[19]	`0`	MemQIn ComGp: do not poll.
		`1`	MemQIn ComGp: poll.
1	[20]	`0`	Overflow: do not poll.
		`1`	Overflow: poll.
6	[21:26]	`X`	** Unused **.
1	[27]	`0`	H priority queue: not included in poll.
		`1`	H priority queue: included in poll.
1	[28]	`0`	L priority queue: not included in poll.
		`1`	L priority queue: included in poll.
1	[29]	`X`	When using 32b read, this specifies either the odd or the even word.
2	[30:31]	`00`	==== Fixed Field ====

Table 8: ShRx Space sPAddr encoding.

Table 8 summarizes the encoding for ShRx Space. Notice that ShRx queues to poll from is specified with a bit-vector for ComGp and another for H/L priority. The poll returns the short message (in the case of a Pcp queue, a pseudo-message) associated with the highest priority, non-empty queue. Prioritization among the 9 queues in the 5 communication groups is static, and the order, *in descending priority* is:

Queue Name	Expected Usage
VasR-0 H	Flow-control
PcpR-0 H	SM-RxH
VasR-1 H	SM-ShRxH
MemQIn	aBIU "messages"
Overflow	Overflow/Miss
PcpR-1 H	** Unused **
VasR-0 L	** Unused **
PcpR-0 L	SM-RxL
VasR-1 L	SM-ShRxL
PcpR-1 L	** Unused **

2.2.2 sBIU Special ShTx Space

Width (bits)	Bit field	Content	Description
3	[0:2]	`010`	==== Fixed Field ====
2	[3:4]	`00`	Pas/Vas-0 ComGp.
		`X1`	MemQOut ComGp; when sPAddr[3] is 0, this is an internal cmd, as opposed to msg to send out to network when sPAddr[3] is 1.
		`10`	Pas/Vas-1 ComGp.
23	[5:27]	`X`	Message content.
1	[28]	`1`	H Priority queue.
		`0`	L Priority queue.
3	[29:31]	`100`	==== Fixed Field ====

Table 9: Special ShTx Space sPAddr encoding.

The sBIU Special ShTx Space is very similar to the ShTx Space in sBIU User Space. The main differences are: (i) MemQOut is included as a target ComGp; (ii) a wider field is devoted to "Message content". Table 9 summarizes the encoding for this Space. Notice that the ComGp is specified with only 2b in sPAddr[3:4].

Similar to the case of ShTx Space, sP code uses a 32b or 64b uncached write to access this space. In the former case, the address of an uncached 32b write to ShTx space is concatenated with its 32b data to form a 64b quantity, with the address occupying 0:31, and data from 32:63. For the Pas/Vas queues, this 64b quantity is treated by the NES hardware as having the Short Tx Format shown in Figure 12 in Section 5.2. In the case of 64b uncached write, the 64b data is taken as the message specification and can either have the Short Tx Format, or the Fully Specified Format (aka Raw Header). See Section 5.2 for more details of how short messages are transformed by the NES during message send and reception.

The situation is more complex for MemQOut queues which in addition to message packets, support commands which either go to the aBIU, the Data Motion unit in CTRL, or the DMA Read/Send Engines in CTRL. See Section 5.6.1.

2.2.3 sBIU Cmd Space

The sBIU Cmd Space uses sPAddr[4:5] to distinguish between three major classes of commands: Config State Access, Cls Access and ImmdCmd.

Width (bits)	Bit field	Content	Description
2	[4:5]	`0X`	Cls Access Space.
		`10`	ImmdCmd Space.
		`11`	Config State Space.

ClsSRAM Update

Table 10 summarizes the encoding for Cls update. The data of the bus transaction should have the encoding shown in Table 11. The sBIU composes an appropriate MemQOut command into the identified MemQOut queue, which will cause the update to occur when the command is at the head of the command queue. sPAddr[28] indicates whether this MemQOut cmd should be acknowledged. (See Section 5.6.1 for the format of the command composed by sBIU.)

Width (bits)	Bit field	Content	Description
4	[0:3]	`0111`	==== Fixed Field ====
1	[4]	`0`	==== Fixed Field ====
22	[5:26]	`X`	** unused **
1	[27]	`X`	Ack_P
1	[28]	`0`	Use MemQOut-0
		`1`	Use MemQOut-1
3	[29:31]	`X`	** unused **

Table 10: sPAddr encoding for Cmd Space used for Cls Access.

Width (bits)	Bit field	Content	Description
2	[0:1]	`X`	** unused **
1	[4]	`X`	extra state bit (currently unused)
3	[5:26]	`X`	cache line state value
21	[27]	`X`	cache line addr
5	[28]	`X`	** unused **

Table 11: 4 Byte data encoding for Cls Update.

Cmd Space Config State Access

Accesses to Config State Space is similar to QPtr space accesses in that state update is specified in the transaction address only, with the new state value (7b) specified in sPAddr[19:25]. In addition, whether update is done is qualified by sPAddr[26]. The data part of a Write transaction is discarded. Unlike the User QPtr Space, however state update is *not* supported with Read transactions; an attempt to do this results in undefined results with no guarantee on either the data returned or the effect of the update attempted.

Width (bits)	Bit field	Content	Description
4	[0:3]	`0111`	==== Fixed Field ====
2	[4:5]	`11`	==== Fixed Field ====
4	[6:9]	`X`	ComGp.
			sPAddr[6] determines whether accessing CTRL (0) or xBIU (1) state.
			sPAddr[7] → CtrlAddr[2], xBIUAddr[2]
			sPAddr[8] → CtrlAddr[3], xBIUAddr[3]
			sPAddr[9] → CtrlAddr[4], xBIUAddr[4]
1	[10]	`0`	==== Fixed Field ==== (sPAddr[10] → CtrlAddr[0], xBIUAddr[0])
1	[11]	`X`	sPAddr[11] → CtrlAddr[1], xBIUAddr[1]
2	[12:13]	`X`	** Unused **.
4	[14:17]	`X`	sPAddr[14] → CtrlAddr[5], xBIUAddr[5]
			sPAddr[15] → CtrlAddr[8], xBIUAddr[8]
			sPAddr[16] → CtrlAddr[9], xBIUAddr[9]
			sPAddr[17] → CtrlAddr[10], xBIUAddr[10]
8	[18:25]	`X`	New value.
1	[26]	`0`	No Update.
		`1`	Update.
1	[27]		sPAddr[27] → CtrlAddr[6], xBIUAddr[6]
1	[28]		sPAddr[28] → CtrlAddr[7], xBIUAddr[7]
1	[29]	`X`	For 32b reads, specifies which word is read.
2	[30:31]	`00`	==== Fixed Field ====

Table 12: sPAddr encoding for Cmd Space used for Config Q State Access.

Table 12 summarizes the encoding for accessing the Q State part (sPAddr[10] == 0) of Config State Space; it also gives the correspondence between aPAddr and CtrlAddr and xBIUAddr, which are internal addresses used by CTRL and the two BIUs respectively. Table 13 summarizes the encoding for for accessing the SysState part (sPAddr[10] == 1) of Config State Space. The correspondence to CtrlAddr and xBIUAddr is similar for both tables.

In order to know what state is kept at different state addresses, see Section 4, the address map of state registers.

Width (bits)	Bit field	Content	Description
4	[0:3]	`0111`	==== Fixed Field ====
2	[4:5]	`11`	==== Fixed Field ====
4	[6:9]	`X`	sPAddr[6] determines whether accessing CTRL (0) or xBIU (1) state.
			sPAddr[7] → CtrlAddr[2], xBIUAddr[2]
			sPAddr[8] → CtrlAddr[3], xBIUAddr[3]
			sPAddr[9] → CtrlAddr[4], xBIUAddr[4]
1	[10]	`1`	==== Fixed Field ==== (sPAddr[10] → CtrlAddr[0], xBIUAddr[0])
1	[11]	`0`	==== Fixed Field ==== sPAddr[11] → CtrlAddr[1], xBIUAddr[1]
2	[12:13]	`X`	** Unused **.
4	[14:17]	`X`	sPAddr[14] → CtrlAddr[5], xBIUAddr[5]
			sPAddr[15] → CtrlAddr[8], xBIUAddr[8]
			sPAddr[16] → CtrlAddr[9], xBIUAddr[9]
			sPAddr[17] → CtrlAddr[10], xBIUAddr[10]
8	[18:25]	`X`	New value.
1	[26]	`0`	No Update.
		`1`	Update.
1	[27]		sPAddr[27] → CtrlAddr[6], xBIUAddr[6]
1	[28]		sPAddr[28] → CtrlAddr[7], xBIUAddr[7]
1	[29]	`X`	For 32b reads, specifies which word is read.
2	[30:31]	`00`	==== Fixed Field ====

Table 13: sPAddr encoding for Cmd Space used for Config SysState Access.

Cmd Space: ImmdCmd

ImmdCmd's are subdivided into several category by its Primary Opcode field, shown in Table 14. (See Appendix B for how the ImmdCmd's in this section are used.) Table 15 summarizes the encoding for ImmdCmd's that go to CTRL. Table 16 summarizes the encoding for ImmdCmd's that go to aBIU.

2.3 Bus Transactions Supported for sBIU Address Spaces

The sBIU supports only a subset of 60X bus transactions to its address space. The actual subset supported is further dependent on the address sub-space within the sBIU space. For ease

Width (bits)	Bit field	Content	Description
1	[12]	0	Goes to CTRL for final actions.
		1	Approval Mechanism related; goes to aBIU for final actions.

Table 14: ImmdCmd Primary Opcode encoding.

Width (bits)	Bit field	Content	Description
4	[0:3]	0111	==== Fixed Field ====
2	[4:5]	10	==== Fixed Field ====
1	[6]	X	** Unused **
3	[7:9]		** Unused **
2	[10:11]	11	==== Fixed Field ====
1	[12]	0	==== Fixed Field ====
1	[13]	X	** Reserved **
1	[14]	X	** Unused **
3	[15:17]	000	NES Reset
		001	** Unused **
		010	Arctic Ack
		011	Clear DMA state and DMA Cmd queue in CTRL.
		100	** Unused **
		101	** Unused **
		110	Interrupt aP.
		111	Interrupt sP.
3	[18:20]	X	** Unused **
5	[21:25]	X	Arctic Ack Count; Used only when (sPAddr[15:17] == 010)
3	[26:28]	X	** Unused **
3	[29:31]	100	==== Fixed Field ====

Table 15: Encoding for ImmdCmds that go to CTRL. sPAddr[15:17] is the secondary opcode field.

Width (bits)	Bit field	Content	Description
4	[0:3]	0111	==== Fixed Field ====
2	[4:5]	10	==== Fixed Field ====
4	[6:9]	X	** Unused **
2	[10:11]	11	==== Fixed Field ====
1	[12]	1	==== Fixed Field ====
2	[13:14]	X	** Unused **
3	[15:17]	000	** Unused **
		001	** Unused **
		010	Reset (to 0) DMA RxData queue C-ptr in aBIU.
		011	Clear ApprovalReg. (Change ApprovalReg state to Free regardless of current state.
		100	** Unused **
		101	** Unused **
		110	** Unused **
		111	** Unused **
12	[18:29]	X	** Unused **
2	[30:31]	00	==== Fixed Field ====

Table 16: Encoding for ImmdCmds that go to aBIU. sPAddr[15:17] is the secondary opcode field.

of reference, we categorizes the 60X bus transactions of interest to this document into several categories listed in Table 17. Using this categorization, Table 18 lists the transactions that are supported for each sub-space of the sBIU address space.

Dealing with Unsupported Transactions

When a transaction that is not supported for an address space is detected, the sBIU must still complete the required bus protocols so that the transaction completes on the sP-bus. The effect of writes is undefined, although the hardware implementation must ensure that protection is adequate. A simple way to achieve this is to make ensure that no write is performed to the SRAM when it is due to an unsupported transaction. In the case of reads, there is no guarentee on what data is returned. Detection of an unsupported 60X transaction is an error that is reported in the Err part of the NES MSR . A programmable interrupt mask allows this error to cause an interrupt if desired.

2.4 sBIU External Interfaces

The external interface of sBIU can be grouped into the following categories:

Trans Cat	TSiz	Tbst Asserted	TT	TT Description
4B Aligned Read (4B-AR)	#b100	No	#b01010	Read
			#b11110	Read-atomic
			#b01011	RWNITC
4B Aligned Write (4B-AW)	#b100	No	#b00010	write-w-flush
			#b00110	write-w-kill
			#b10010	Write-w-flush-atomic
8B Aligned Read (8B-AR)	#b000	No	#b01010	Read
			#b11110	Read-atomic
			#b01011	RWNITC
8B Aligned Write (8B-AW)	#b000	No	#b00010	write-w-flush
			#b00110	write-w-kill
			#b10010	Write-w-flush-atomic
32B Read (32B-R)	#b010	Yes	#b01010	Read
			#b11110	Read-atomic
			#b01011	RWNITC
			#b01110	RWITM
			#b11010	RWITM-atomic
32B Write (32B-W)	#b010	Yes	#b00010	write-w-flush
			#b00110	write-w-kill
			#b10010	Write-w-flush-atomic

Table 17: These are the 60X transactions that the sBIU supports as a slave device. They are divided into several categories. Table 18 gives further details of which categories are supported for a particular sBIU sP-bus address space. Bus transactions not listed in this table are not supported by the sBIU.

NES sP Space	Trans Cat Supported
sBIU User Space/xSRAM Space	4B-AR 4B-AW 8B-AR 8B-AW 32B-R 32B-W
sBIU User Space/QPtr Space	4B-AR 4B-AW
sBIU User Space/ShTx Space	4B-AW 8B-AW
sBIU User Space/ShRx Space	4B-AR 8B-AR
sBIU Cmd Space	4B-AR 4B-AW 8B-AR 8B-AW
sBIU Special ShTx Space	4B-AW 8B-AW

Table 18: This table specifies the type of transaction supported for each sP-bus NES address space.

Interface	Interfacing with Functional Block
sP-bus	sP memory bus
s-port	sSRAM
Shared Memory	aBIU
SC-bus	CTRL
CS-bus	CTRL
QPtr Status	CTRL
ShTx/ShRx	CTRL
MemQOut	CTRL
Data Motion(inter-SRAM transfer)	CTRL
J-bus	CTRL

2.4.1 sBIU sP-bus Interface

Signal direction is described with respect to the sBIU, *i.e.* “I” means an input to the sBIU.

Name	I/O	# bits	Description
SPBAddr	I	32b	address
_SPBTS	I	1b	Transaction start; start of address transaction
SPBTT	I	5b	Transfer Type
_SPBTBst	I	1b	Tbst; Transfer Burst
SPBTSiz	I	3b	TSiz; Transfer Size
_SPBAAck	O	1b	AAck; Address Ack
_SPBARtry	O	1b	ARTRY; Address Retry
_SPBShd	O	1b	Shared
_sPDbg	I	1b	Data bus grant
_SPBDBB	I	1b	Data bus busy
_SPBTA	O	1b	Data Ack; slave indicating to master data is valid/consumed.
_SPBHReset	I	1b	HRESET; Hard Reset
_SPBSReset	I	1b	SRESET; Soft Reset
_SPBInt	O	1b	INT; Interrupts sP

The following signals are on the 604 bus, but are not needed by the NES on the sP side:

Name	I/O	# bits	Description
sPAddrP	IO	4b	address parity; sBIU is never a master on the sP bus. sBIU also does not check for parity.
_BR	O	1b	Bus request; sBIU is never a master on the sP bus
_BG	I	1b	Address bus grant; sBIU is never a master on the sP bus
_ABB	IO	1b	Address bus busy; sBIU is never a master on the sP bus
_GBL	IO	1b	Global
_CI	IO	1b	Cache Inhibit
_WT	IO	1b	Write through
TC	IO	3b	Transaction code
_DBWO	I	1b	Data Bus Write Only; Not supported by the MPC105.
_XATS	I	1b	Extended Address Transfer start, indicates direct-store operation, to I/O space. sP does not use any direct-store operations.
_APE		3b	Address parity error
_CSE		3b	Cache set element
_DPE		1b	Data Parity Error
_DRTRY		1b	Data Retry
_TEA		1b	Transfer Error Acknowledge
_DBDIS		1b	
L2_INT		1b	
_Ckstp_in		1b	Checkstop input (wrt 604)
_MCP		1b	Machine check interrupt
_SMI		1b	System Management interrupt
_Ckstp_out		1b	Checkstop output (wrt 604)
Run		1b	Run
Halted		1b	Halted

2.4.2 sBIU s-port Interface

Signal direction is described with respect to the sBIU, *i.e.* "I" means an input to the sBIU.

Name	I/O	# bits	Description
SSRSAddr	O	12b	sSRAM sP-bus port address
SSRSRW	O	1b	sSRAM R/_W; both banks share a common _OE signal.
_SSRSOE	O	1b	sSRAM _OE; both banks share a common _OE signal.
_SSRSCE	O	2b	sSRAM odd 32b and even 32b _CE;

2.4.3 sBIU Shared Memory Interface

Signal direction is described with respect to the aBIU, *i.e.* "I" means an input to the aBIU.

Name	I/O	# bits	Description
AcdOpV	I	1b	whether AcdOp is valid.
AcdOp	I	3b	sBIU convey shared memory related requests from sP
AcdApprAddr	I	12b	aSRAM address where data can be found/placed if the captured pending request is to be handled as Approval-w-Data.
AcdClsSync	I	1b	When AcdOpV, this indicates whether a synchronous ClsWrite enable is also requested.
AcdOpDone	O	1b	Notification from aBIU to sBIU that AcdOp is completed.
ClsUpdateReq	I	1b	Post sP clSRAM Update Request.
ClsSyncUpdate	I	1b	Indicates whether the requested update is a Synchronized one.
ClsUpdateDone	O	1b	Notification from aBIU to sBIU that clSRAM update is completed.

Signal direction is described with respect to the sBIU, *i.e.* "I" means an input to the sBIU.

Name	I/O	# bits	Description
clssPAddrInEn	O	1b	sP bus HAL address register input enable

2.4.4 sBIU SC-bus Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
ScBusPassThru	I	1b	When ScBusV is asserted, ScBusPassThru indicates whether the requested operation is just passing through CTRL, destined for aBIU.
SCBusAddr	I	11b	When ScBusV is asserted, ScBusAddr is the address of the state that is accessed.
SCBusData	I	7b	When ScBusV is asserted, and ScBusRW indicates a write, ScBusData contains the data to be written.
SCBusOp	I	1b	When ScBusV is asserted, ScBusOp indicates the requested operation.
SCBusFree	O	1b	SC bus available.
SCBusDone	O	1b	Pulsed for one cycle to indicate that the SC bus operation requested by aBIU has been completed.

2.4.5 sBIU CS-bus Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
CSBusAddr	O	8b	When CSBusValid is asserted, CsBusAddr is the address of the state that is accessed.
CSBusData	O	7b	When CSBusValid is asserted, and CsBusRW indicates a write, CsBusData contains the data to be written.
CSBusValid	O	1b	CS bus signals valid; when asserted, this indicates that there is valid data on the CS bus.

2.4.6 sBIU ShTx/ShRx Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
ShTxComp	I	1b	sBIU short message compose I-bus request
CompFree	O	1b	Compose Buffer Available
RxLateAck	I	1b	sBIU asking that a late ack be returned to Arctic.
RxQEmpty	O	9b	Indicates whether a PcpR queue is empty. There are 8 different PcpR queues plus the Overflow queue.

2.4.7 sBIU MemQOut Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
MemQOutOp	I	2b	MemQOut OpType (Passed to CTRL by sBIU to indicates whether the new MemQOut cmd is a short messages going out to Arctic network, a cmd going to the aBIU, a DMA Read or Send cmd, or a Data Motion cmd. See Section 5.6.1 for the encoding.)
MemQOut0Comp	I	1b	Requesting to put a cmd into MemQOut0.
MemQOut1Comp	I	1b	Requesting to put a cmd into MemQOut1.

2.4.8 sBIU Data Motion (inter-SRAM transfer) Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
DMValid	I	1b	sBIU Data Motion I-bus Request
DMFree	O	1b	sBIU Data Motion Available
DMDone	O	1b	sBIU Data Motion done

2.4.9 sBIU J-bus Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
JBusData	I	32b	J-bus data; correspond to the even (i.e. 0th) 32b-word.
JBusAddr	I	13b	J-bus address

2.5 sBIU Actions

This section describes the abstract functionality of the sBIU in response to sP-bus transactions to various sBIU spaces. Determination of the space that an sP-bus transaction is accessing is done by using the decoding specified in Tables 1 and 2. *In the case of write-like transactions that are not supported, write-enable for s-Port is never asserted.* The description below sometimes introduces symbolic variables, and assign them values. This is to make descriptions clearer and more precise. The assignment has a C-like semantics of treating the assignment as an expression that returns the assigned value. sPortAddr is a 8B aligned address. For 4B accesses, sPAddr[29] is used to select the word read/written. The symbol "++" is used to indicate concatenation of bit fields.

2.5.1 sBIU User Space: sSRAM Space

Accesses to this space conforms to the encoding in Table 3, with the further constraint that sPAddr[16] is #b1.

◇ **sPortAddr:** sPAddr[17:28]

◇ sBIU asserted the expected control signals on the sP-bus, and s-Port to complete the read or write. For 32B write, sPortAddr has to be incremented for the 4 beats. Reminder: 32B-R requires critical word first, while 32B-W is zeroth word first.

◇ No involvement of any other NES component.

2.5.2 sBIU User Space: aSRAM Space

Accesses to this space also conforms to the encoding in Table 3, but with sPAddr[16] set to #b0. (See Section 5.6.1 for format of Data-Motion command.)

◇ **sPortAddr:** `aSRAM-tmp` ++ sPAddr[27:28]

◇ For write (4B-AW, 8B-AW, 32B-W),
- After data from sP-bus is ready (potentially pipelinable), arrange with Data-Motion unit in CTRL for appropriate granularity transfer from aSRAM-tmp to final aSRAM location (as specified by aSRAM address sPAddr[17:29]).

◇ For read (4B-AR, 8B-AR, 32B-R),
- First issue a Data-Motion command to move data from aSRAM location (sPAddr[17:29]) to be transfered to aSRAM-tmp buffer in sSRAM. (Critical word first for Data-Motion will be helpful.)
- When transfer is far along enough (pipeline if possible), start asserting _sPTa so that data is supplied to sP-bus.

2.5.3 sBIU User Space: QPtr Space

Accesses to this space conforms to the encoding in Table 4. See Section 4 to see how requests are forwarded to CTRL with CtrlAddr and used locally as sBIUAddr.

◇ **sPortAddr:** `qptr-base` ++ sPAddr[7:9,29]

◇ If read,
just assert _sPTa as required data is already in sSRAM.

- For both read and write,
  if (sPAddr[26] == Update)
  if (Updating QPtr state in sBIU, *i.e.* (sPAddr[6,9] == #b10))
  update sBIU QPtr state
  else
  forward the update request to CTRL over the SC bus.
- For write, there is a special requirement that no write to sSRAM should occur, *i.e.* sPortR/_W should be R, and _sPortOE should be deasserted.

2.5.4 sBIU User Space: ShRx Space

Accesses to this space conforms to the encoding in Table 8. AccessedOnce is a persistent state variable.

- **sPortAddr:** ShRx-likeAddrGen[sPAddr[15:19,27:28]]

- For read, ShRx-likeAddrGen determines the address to poll from as follows:
  If ((sPTSiz[0] == #b1 /* 4B */) AND (AccessedOnce)
  AccessedOnce = False
  return (One-poll-addr)
  Else
  One-poll-addr =
  (determine among the polled queues, specified by sPAddr[15:19,27:28], the highest priority non-empty queue. If the queue is a VasR queue or MemQIn, ShRx-likeAddrGen returns the C-ptr. If the queue is a Pcp queue, the corresponding ContAddr is returned. If all the queues are empty, the Empty-Msg's ContAddr is returned.)
  If a C-ptr, increment the C-ptr
  (Incrementing C-ptr requires updating the sBIU local copy, and then the CTRL's copy.)
  If (chosen queue uses Late_Ack)
  pulse SbiuLateAck signal to indicate need to provide one ack to Arctic.
  If (sPTSiz[0] == #b1) /* 4B */
  AccessedOnce = True
  return (One-poll-addr)
- There should be no write to this space. As in the case of QPtr Space, we require that no write to sSRAM occurs in the event of a sP-bus write transaction to this space.

Implementor's Notes: SbiuLateAck is pulsed possibly before the second 32b of the message is read. This appears to prematurely release the buffer but is okay because the circular buffer with P-ptr and C-ptr used in the implementation always leaves one entry empty.

2.5.5 sBIU User Space: ShTx Space

Accesses to this space conforms to the encoding in Table 7.

- ◇ **sPortAddr:** ShTx-likeAddrGen[sPAddr[7],0,sPAddr[28]]

- ◇ ShTx-likeAddrGen returns the P-ptr of the TxQ specified by its input parameter (sPAddr[7,28]).
- ◇ For write,
  - If ((queue is in 32b-ShTx mode) and (sPTSize[0] == #b1 /* 4B */)),
    arrange to write sPAddr into sSRAM at the higher word (bank 0) of address used for sPortAddr.
  - if (((sPAddr[29] == #b0) AND (sPTSiz[0] == #b0 /* 8B */))
    OR ((sPAddr[29] == #b1) AND (sPTSiz[0] == #b1)))
    Increment P-ptr
    (This requires first updating the sBIU local copy, and then the CTRL's copy when it is known that any write of sPAddr in sSRAM has been completed, and the data part of the write bus transaction has completed. The timing has to be coordinated such that when CTRL starts pushing this packet out to TxU, it will read the correct data from the sSRAM.)

Implementor's Notes: The implementation is a little tricky here because several things have to be coordinated. The sBIU has to coordinate all writes to SRAM before updating CTRL's copy of the P-ptr. CTRL is free to implement further pipelining of the pointer update and SRAM write, but has to guarantee that either both paths go in synchrony, or that SRAM write always go ahead of pointer update.

2.5.6 sBIU Special ShTx Space

Accesses to this space conforms to the encoding in Table 9.

- ◇ **sPortAddr:** ShTx-likeAddrGen[sPAddr[3:4,28]]

- ◇ ShTx-likeAddrGen returns the P-ptr of the TxQ specified by its input parameter(sPAddr[3:4,28]).
- ◇ For write,
  If ((queue is in 32b-ShTx mode) and (sPTSize[0] == #b1 /* 4B */)),
  arrange to write sPAddr into sSRAM at the higher word (bank 0) of address used for sPortAddr.
  if (((sPAddr[29] == #b0) AND (sPTSiz[0] == #b0 /* 8B */))
  OR ((sPAddr[29] == #b1) AND (sPTSiz[0] == #b1)))
  Increment P-ptr
  (This requires first updating the sBIU local copy, and then the CTRL's copy when it is known that any write of sPAddr in sSRAM has been completed, and the data part of the write bus transaction has completed. The timing has to be coordinated such that when CTRL starts pushing this packet out to TxU, it will read the correct data from the sSRAM.)
  If (sPAddr[4] == 1) /* compose is to MemQOut */
  pass to CTRL:
  sPAddr[3,15:16] as SbiuMqOp;
  sPAddr[28] as SbiuMqOId;

2.5.7 sBIU Cmd Space: Cls Access

Accesses to this space conforms to the encoding in Table 10.

- ◇ **sPortAddr:** `sBIU-qconfig-tmp`

- ◇ The sBIU composes an appropriate MemQOut command into the identified MemQOut queue, which will cause the update to occur when the command is at the head of the command queue. sPAddr[28] indicates whether to use MemQOut 0 or 1. sPAddr[27] indicates whether the cmd should be acknowledged. (See Section 5.6.1 for the format of the command composed by sBIU. See also Table 67)

2.5.8 sBIU Cmd Space: Config State Access

Accesses to this space conforms to the encoding in Tables 12 and 13. See Section 4 to see how requests are forwarded to CTRL with CtrlAddr and used locally as sBIUAddr.

- ◇ **sPortAddr:**

  if (Read AND (sPAddr[10:11] == #b00)
          AND ((sPAddr[6] == #b0) OR (sPAddr[6:9] == #b1101)))
      If (aPAddr[6] == #b0)
          `qptr-base` ++ sPAddr[7:9,29]
      else
          `Overflow-queue-statue`
  else
      `sBIU-qconfig-tmp`

- ◇ If read,

  if ((sPAddr[10:11,16] != #b001)
          AND ((sPAddr[6] == #b0) OR (sPAddr[6:9] == #b1101)))
      arrange with CTRL to read value into sBIU-qconfig-tmp in sSRAM before asserting _sPTa.
  else
      just assert _sPTa as required data is already in sSRAM. is returned from

- ◇ Else

  if (sPAddr[26] == Update)
      if (Updating QPtr state in sBIU, *i.e.* (sPAddr[6,9] == #b10))
          update sBIU QPtr state
      else
          forward the update request to CTRL over the SC bus.

- ◇ For write, there is a special requirement that no write to sSRAM should occur, *i.e.* sPortR/_W should be 1, and _sPortOE should not be asserted.

2.5.9 sBIU Cmd Space: ImmdCmd

Accesses to this space conforms to the encoding described in Tables 15 and 16.

- ◇ **sPortAddr:** `sBIU-qconfig-tmp`

- ◇ Decode and forward the request to CTRL.
- ◇ Data part of the transaction has no meaning to NES.
- ◇ Switch (ImmdCmdPrimaryOp = sPAddr[12])

```
Case (#b0 /* Goes to CTRL */):
    supply CtrlAddr[0:1] with sPAddr[10:11] /* #b11 */
    supply CtrlAddr[8:10] with sPAddr[15:17]
    supply CtrlAddr[2:4] with sPAddr[7:9]
    supply CtrlAddr[6:7] with sPAddr[27:28]
    supply CtrlData[2:6] with sPAddr[21:25]
    break;
Case (#b1 /* Goes to aBIU */):
    supply AcdOp with sPAddr[15:17]
    supply AcdApprAddr with sPAddr[18:29]
    break;
```

2.5.10 Request from CTRL over the CS-bus Interface

The sBIU receives config state access request from CTRL over the CS-bus Interface, for state that resides in the sBIU. A request has arrived if CsBusV is asserted. When the sBIU has performed the requested operation, it asserts the CsBusDone for one cycle.

2.6 sBIU State

This section describes the state in sBIU, categorizing them into three groups:

Config Queue State	Queue related state, that is configurable by software on the sP or aP.
Config SysState	Config state that is not queue related, usually only reable but not writeable.
Internal State	Internal state of the sBIU, not directly accessible by software.

2.6.1 sBIU Configurable Queue State

sBIU contains the following QState: (One-poll-addr and UsedOnce must be both readable and writeable by system code to support context switching.)

- ShTx-like QState:
  - ○ 6 x {P-ptr (7b) + Base (9b) + Bound (5b)}

  These 6 are used for:
    - ◇ PasT-0L
    - ◇ PasT-0H
    - ◇ PasT-1L
    - ◇ PasT-1H
    - ◇ MemQOut-0
    - ◇ MemQOut-1

- ShRx-like QState:
    - 5 x {C-ptr (7b) + P-ptr (7b) + Base (9b) + Bound (5b) + LateAck (1b)}
    - 5 x emptyP (1b)

    These 5 are used for:
        - VasR-0L
        - VasR-0H
        - VasR-1L
        - VasR-1H
        - MemQIn
    - 3 x PcpContAddr (12b)

    These 3 are used for:
        - PcpR-0
        - PcpR-1
        - Empty-Msg

    This is used for one-poll:
        - One-poll-addr (12b)
        - AccessedOnce (1b)
- General sPortAddr State:
    - 1 x sBIU-qconfig-tmp (12b)
    - 1 x qptr-base (8b)
    - 1 x aSRAM-tmp (10b)

2.6.2 sBIU Configurable SysState

- sBIU MSR:
    - 5? x 1b

    These are used to indicate internal resource utilization:
        - Tx Addr Write Pending
        - aSRAM Write Pending
        - State Update Pending
        - clSRAM write Pending
        - clSRAM read Pending

comment: *This has to be re-visited. There can be 1 pending Tx Addr Write, up to 2 pending aSRAM write (1 queued in JI and another being done by DM-unit), and multiple (2 or more) State Update pending.*

2.6.3 sBIU Internal State

Functional Block	Functionality
sBIU-MC (MC)	Overall control of sBIU: runs main state machine and tracks sBIU functional block busy/avail.
sP-bus Interface (SI)	Handles sP-bus (604, MPC105) protocol, responding to and generating control signals.
	Generals control signals for s-Port.
s-Port Addr Gen (SAG)	generate sPortAddr for sP-bus transactions to sBIU slave space.
	Generate request to update CTRL Q State.
	Read sBIU Q State.
	Update sBIU Q State.
	Update sBIU Q State and re-evaluate empty bit.
	Increment sBIU VasR Q State, and set corresponding empty bit.
State Access/ImmdCmd (SAI)	State access request from sP-bus.
	State access request from CTRL.
	State update request from SAG.
	State access request to SAG.
	State access request to MC.
J-bus Interface (JI)	J-bus Write.

Table 19: Internal Organization of the sBIU.

- In sP-bus Interface:
  - 1 x sPAddrIn (32b)
  - 1 x sPCtrlIn (8b)

    Latches sPAddr and other control signals for further reference. sPCtrIn Control signals include:
    - sPTT (5b)
    - sPTSiz (3b)
    - _sPTbst (1b)
  - 1 x s-PortAddrOut (12b)
- In SAI:
  - 1 x ConfCmdIn (20b)
  - 1 x ConfCmdOut (20b)
- In JI:
  - 1 x JData (32b)
  - 1 x JAddr (13b)

2.7 sBIU Internal Organization

The sBIU is organized into several functional blocks as described in Table 19. Figure 5 is a

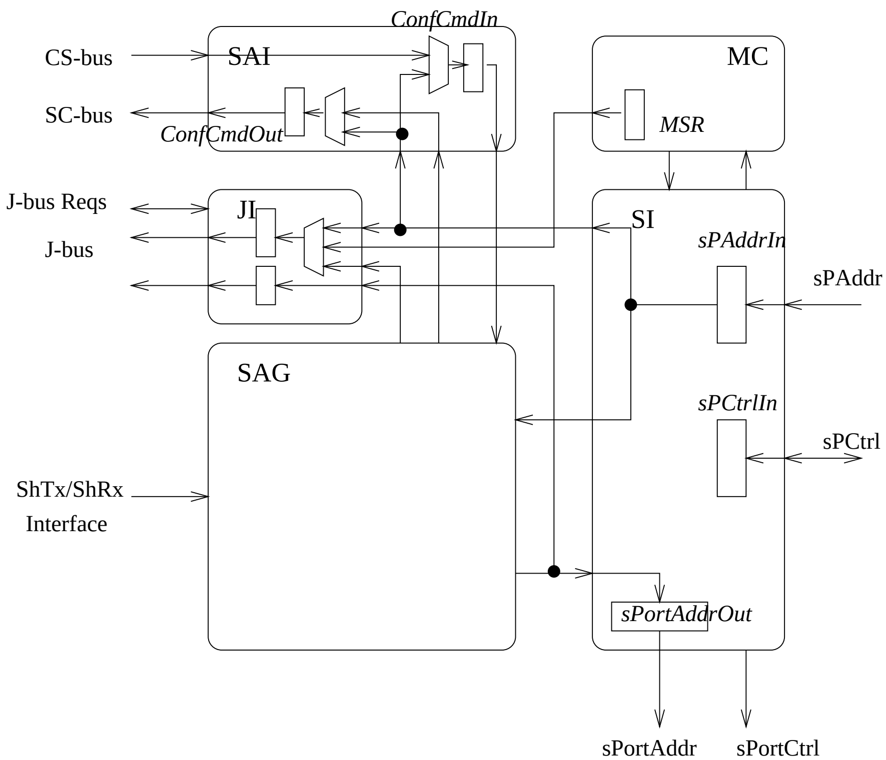

Figure 5: Internal Organization of sBIU.

diagram illustrating this organization. Each block is described in greater detail below. Signal descriptions are usually preceded by a two char prefix: ii, io, xi or xo. If the first character is 'x' (external), the signals are from/to functional blocks external to the sBIU. Otherwise, it is 'i' (internal). The second char indicates whether the signal is an input ('i') or an output ('o') with respect to the functional block being described. Sometimes the second character is not specified when we are refering to a logical group of signals which include both inputs and outputs.

2.7.1 sBIU-Master Control (MC)

- External functionalities:
  - sP-bus transaction, to sBIU slave space.
    ii SiMcAccSp[0:2]
    xi _sPTs
    xo McSiBusy (sPAddrIn occupied)
- Internal functionalities:
  - generates and coordinates DM request.
  - coordinates ShTx.
  - forwards and coordinates Config State Access.

  - tracks any on-going transaction. These include:
    (reflected in sBIU MSR)
    - ⋄ Tx (Addr) write (up to 1 outstanding, queued in JI)
    - ⋄ aSRAM write DM (up to 2 outstanding, 1 queued in JI 1 being done)
    - ⋄ Config State write (up to ? outstanding, 1 queued in JI ?-1 being done)
    - ⋄ QPtr State write (up to ? outstanding, 1 queued in JI ?-1 being done)
    - ⋄ clSRAM write (up to 1 outstanding, queued in external register)
    - ⋄ clSRAM read (up to 1 outstanding, queued in external register)

    (Not reflected in sBIU MSR)
    - ⋄ sPAddrIn occupied.
- State in this functional block:
  - sBIU MSR[0:4?]
  - sPAddrInOccupied (1b)
- Scheduling:
  - There is no internal scheduling to take care of here. The appropriate sub-state machine runs during an sP-bus transaction. All further action needed is also set up properly with the appropriate functional block that will perform them (or in turn arrange to perform them). Further pending actions are tracked with the aid of the sBIU MSR.

2.7.2 sP-bus Interface (SI)

- External functionalities:
  - Address decode; identifies which sBIU address space, if any, a transaction is referring to.
    xi sPAddr[0:3], sPAddr[4:6]
    io SiMcAccSp[0:2]
  - MemQOut Opcode decode
    xi sPAddr[3,15:16]
    xo SiMemQOp[0:2]
  - Generates control signals for s-Port.
    xo sPortR/_W[0:1], sPortOE[0:1], sPortCE[0:1]
    ii SagSiAddrRdy
    io SiSagAddrDone
- Internal functionalities:
  - TT decode
    xi sPTt
    Ii SiMcAccSp[0:2]
    Io TTSupported
  - TSize decode
    xi sPTT, _sPTbst
    Ii SiMcAccSp[0:2]
    Io SizeSupported

- Responds to and generates sP-bus controls.
  xi _sPTs, sPBg, sPBb, sPDbg, sPDbb
  ii McSiBusy
  xo sPArtry, _sPAAck(?Depends on memory controller. Check MPC105 to see what it expects.), sPTa

- State in this functional block:
  - sPAddrIn[0:31]
  - sPCtrlIn[0:31?]

2.7.3 s-Port Addr Gen (SAG)

- External functionalities:
  - generate sPortAddr for sP-bus transactions to sBIU slave space.
    xi _sPTs (indicates request initiated)
    xi _sPAAck (indicates request committed)
    io SagSiAddrRdy
    io SagSiAddr
    ii SiSagAddrDone
    (master select)
    xi sPAddr[1:3], sPAddr[4:6,16]
    (ShTx-like Addr Gen)
    xi sPAddr[3:4], sPAddr[7], sPAddr[28]
    (ShRx-like Addr Gen)
    xi sPAddr[15:19], sPAddr[27:28]
    ii CsPcpREp[0:3]
  - generate request to update CTRL Q State.
    io SagSaiReq, SagSaiVal[0:7], SagSaiRegAddr[0:10?]
    ii SagSaiDone
  - read from sBIU Q State.
    ii SaiSagReq, SaiSagRegAddr[0:10?], SaiSagRW
    io SagSaiDone
    xo SagJiData[0:31]
  - update to sBIU Q State.
    ii SaiSagReq, SaiSagVal[0:7], SaiSagRegAddr[0:10?], SaiSagRW
    io SaiSagDone
  - update to sBIU Q State requiring re-evaluation of empty bit.
    ii SaiSagReq, SaiSagVal[0:7], SaiSagRegAddr[0:10?], SaiSagRW
    io SaiSagDone
  - increment sBIU VasR P-ptr, and set corresponding empty bit.
    xi SbiuVasRP-PtrInc[0:3]
    Note: we have to perform the increment in time to accept one request every 2 cycles. If not, flow-control feedback is needed.
- Internal functionalities:
  - Maintain BCP for ShRxQ and ShTxQ, incrementing them when queue is accessed, and re-evaluating associated empty bit. Also generate request to update CTRL Q State as a result.

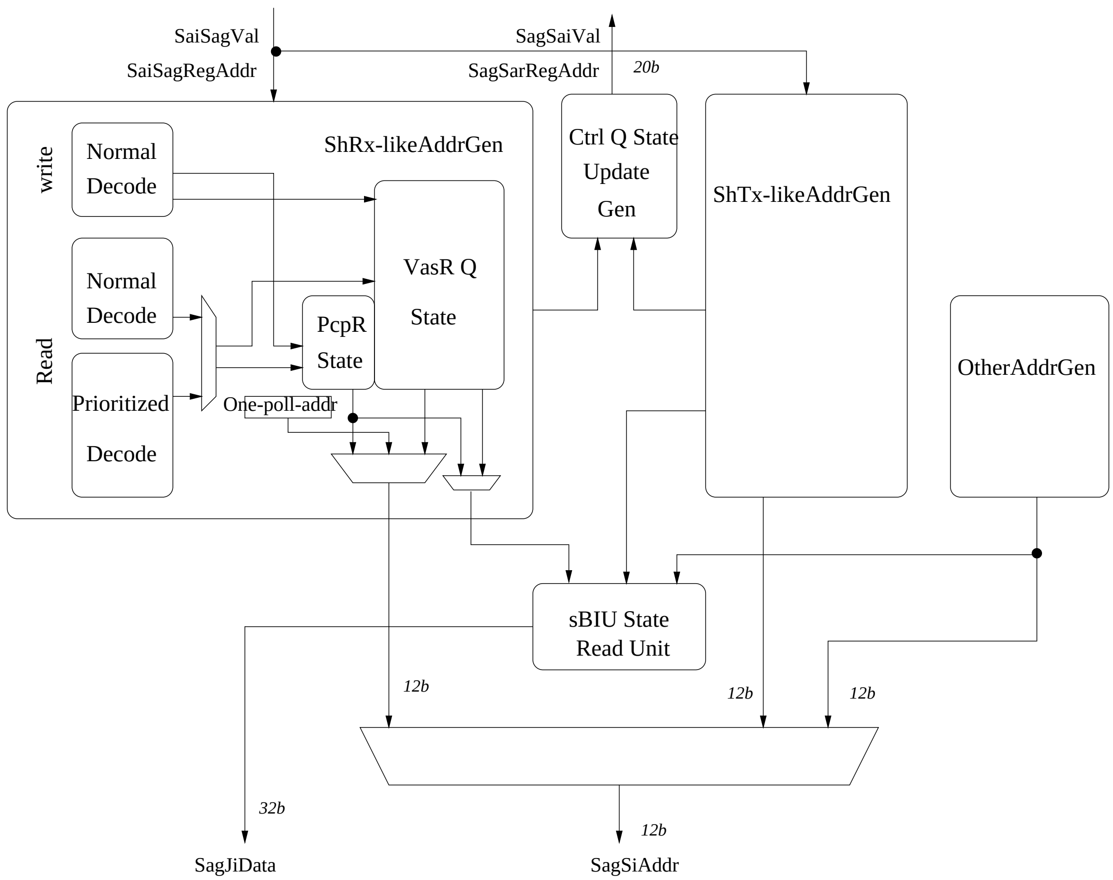

Figure 6: Organization of s-Port Addr Gen.

  - ○ Prioritized ShRx-like addr selection.
- State in this functional block:
  - ○ ShRx-like Qstate
  - ○ ShTx-like Qstate
  - ○ General sPortAddrState
- Scheduling:
  - ○ Use raw _sPTs to coordinate scheduling. _sPTs asserted means next read cycle is for sP-bus request.

The s-Port Addr Gen is more complex than the other functional blocks. Figure 6 shows a possible organization. OtherAddrGen is responsible for address generating with/for aSRAM, aSRAM-tmp, qptr-base, and sBIU-qconfig-tmp.

2.7.4 State Access and ImmdCmd (SAI)

- External functionalities:

- ◦ State access and ImmdCmd request from sP-bus.
  - xi sPAddr[6:9,10:11,14:17,27:28], sPAddr[18:25]
  - xi sPAddr[4:5], to distinguish between the 2 Config State spaces and ImmdCmd spaces.
  - xo ACD interface.
- ◦ State access request from CTRL.
  - x sBIU SC-bus Interface
- ◦ State update request from SAG.
  - i SagSai interface (Sag to Sai req)
  - ii SagSaiReq, SagSaiVal[0:7], SagSaiRegAddr[0:10?]
  - io SagSaiDone
- ◦ State access request to SAG.
  - i SaiSag interface (Sai to Sag req)
  - io SaiSagReq, SaiSagVal[0:7], SaiSagRegAddr[0:10?], SaiSagRW
  - ii SaiSagDone
- ◦ State access request to MC.
  - i SaiMc interface (Sai to Mc req)
  - io SaiMcReq, SaiMcVal[0:7], SaiMcRegAddr[0:10?], SaiMcRW
  - ii SaiMcDone

- State in this functional block:
  - ConfCmdIn[0:19]
  - ConfCmdOut[0:19]
- Scheduling:
  - all write in order.
  - all read in order
  - but read can overtake write.
- Suggestion:
  - Bypass capabilities will improve performance.

2.7.5 J-bus Interface (JI)

- External functionalities:
  - J-bus Write.
    - ii SagJiAddr[0:13], SagJiData[0:31], SiJiData[0:31], SagJiReqType[0:2], SiJiReqType[0:2].
    - xo JBusData[0:63], JBusAddr[0:12].
    - xo SbiuShReq, SbiuMqReq, SbiuDmReq.
    - xi SbiuShGnt, SbiuMqGnt, SbiuDmGnt.
- State in this functional block:
  - JAddr[0:13], JData[0:32]
- Suggestion:
  - Bypass capabilities will improve performance.

2.8 sBIU Timing Constraints

comment: *We will fill this in as we discover needs.*

Three categories of timing constraints:

- 60X bus specific timing constraints.
- Time to completion of command, particularly where real time constraints are needed.
  - Request to increment sBIU VasR P-ptr currently has real time requirement as there is no acknowledgement. Need to coordinate with CTRL designer what is the fastest allowed rate of request. Something like 1 every 2 cycles, or 1 every 3 cycles is probably reasonable. If it is too difficult to meet this requirement, change the interface to have flow-control feedback.
- Ordering between different operations accepted by sBIU. Some operations need to be processed in FIFO order.

2.9 sBIU Miscellaneous Issues

2.9.1 Rejected Proposed sBIU Functionalities

1. It was proposed that we support SYNC operation, with the semantics that the following operations, which may take some time, have to be completed before the SYNC is allowed to complete:

   (a) Cls write

   (b) MsgQPtr Control update

   (c) Config space update

   (d) ImmdCmd

   (e) ShTx write of addr bits

   An alternative/additional suggestion was that we support lower cost ways of detecting completion of common ops, e.g. Cls write.

   We have chosen a design where there is a real time bound (possibly large but bounded in some cases). For certain actions, because the design allows reading most of the state in CTRL and sBIU, it is possible through that mechanism to determine if certain actions have been performed, *e.g.* to determine if a ShTx write has finished, check the queue's P-ptr in CTRL to see if it has been updated to the expected value which can be read from the sBIU's copy of the same P-ptr.

3 aBIU

The aBIU and sBIU share many common parts. We have, nevertheless, documented them separately so that the specification of each can be read independently. Readers who read both specifications will find some repeated information.

3.1 aBIU Functionality Briefs

The aBIU provides the interface between the aP-bus and the rest of the NES. It is *a slave, a master and a snooper* on the aP-bus. Through its slave space, the aBIU provides the following functionalities to the aP.

1. aSRAM read/write.
2. sSRAM read/write.
3. MsgQPtr Status read and Control update. (If possible, we will place the control for "Msg Rx by interrupt" in this space, thus allowing user code to directly set or clear it.)
4. ShTx writes.
5. ShRx reads.
6. ImmdCmd and Config Space read/update.
7. sP Serviced Space access. (Includes Global Shared Memory)
8. sP space access.

Through its snooped space, the aBIU provides the following functionalities to the aP.

1. Local Global Memory Space access.

Finally, the aBIU provides the following functionalities to the NES, some as an aP-bus master.

1. NES Mastered aP-bus operations.
2. DMA Rx Facilities.
3. Memory Emulation Facilities (Approval and Notification to sP).

These functionalities are described in greater detail when we discuss address space in Section 3.2.

3.2 aP-bus Address Space

The aP-bus address space is divided into three major regions:

Size	From Address	To Address	Description
.5GB	`#x00000000`	`#x1FFFFFFF`	aP Host Memory (dRAM) Space
1.5GB	`#x20000000`	`#x7FFFFFFF`	aBIU (slave) Space
2GB	`#x80000000`	`#xFFFFFFFF`	PCI/ISA/ROM/Flash ROM Space

The aBIU slave space is divided into several regions to provide the functionalities itemized in Section 3.1. The division does not always achieve contiguous address ranges due to the need to enforce protection, *i.e.* functions that should be protected separately must be on different pages. An easy way to understand the address space division is to view the addresses in the aBIU slave space as "instruction encoding". Using this view, the Primary Opcode is in sPAddr[0:5] and has the encoding as shown in Table 20 [9].

Width (bits)	Bit field	Content	Description

6	[0:5]	`0000 00`
		`0000 1X`
		`0001 XX`
			aP dRAM Space. (Not snooped by aBIU.)

		`0000 01`	aBIU Snooped aP dRAM
		`001X XX` `010X XX`	sP serviced Space.
		`0110 XX`	aBIU User Space.
		`0111 XX`	aBIU Cmd Space.
		`1XXX XX`	PCI Space.

Table 20: aBIU aP-bus address space Primary Opcode encoding.

3.2.1 aBIU User Space

The aBIU User Space uses a 3b wide Secondary Opcode, in sPAddr[4:6] and has the encoding as shown in Table 21. One can think of sPAddr[4] as distinguishing between SRAM Space (cacheable) and the other spaces (uncacheable), sPAddr[5] differentiates QPtr Space from the ShTx and ShRx spaces, and sPAddr[6] differentiates between ShTx and ShRx Spaces.

Width (bits)	Bit field	Content	Description
3	[4:6]	`0XX`	SRAM Space.
		`10X`	QPtr Space.
		`110`	ShTx Space.
		`111`	ShRx Space.

Table 21: aBIU aP-bus User Space Secondary Opcode encoding.

SRAM Spaces

The complete address encoding for accesses to sSRAM or aSRAM space is shown in Table 22. This table, which is an aPAddr encoding table that is used in many other parts of this document, has four types of entries:

[9] A reminder to anyone changing the encoding: keep in mind which of the sPAddr bits are unprotection. The 12b in sPAddr[20:31] is always unprotected as they lie within a page (page size is 4kB). If we are using a BAT for translation, at least 17b (sPAddr[15:31]) are unprotected as the minimum BAT size is 128kB. Bit fields that differentiate between different protection domain must lie outside unprotected sPAddr bits.

Width (bits)	Bit field	Content	Description
4	[0:3]	`0110`	==== Fixed Field ====
1	[4]	`0`	==== Fixed Field ====
11	[5:15]	`X`	** Unused **
1	[16]	`0`	sSRAM Space.
		`1`	aSRAM Space.
13	[17:29]	`X`	bits[0:12] of SRAM address.
2	[30:31]	`00`	==== Fixed Field ====

Table 22: SRAM Space aPAddr encoding.

- A "==== Fixed Field ====" is one where the bit-pattern is fixed at the value specified under the "Content" column.
- A "** Reserved **" field is one whose value is not used by the specified address space, but is utilized by other spaces that have to work togather with the space concerned. These fields are not available for arbitrary usage.
- In contrast, "** Unused **" fields are not currently used in the address space, and can be used to encode information should the need to include further information arises.
- The last category are fields which are used by the address space to encode variable information. The meaning of these fields are described in the table itself.

In the case of accesses to SRAM Space, sPAddr[16] distinguishes between accesses to sSRAM (#b1) and aSRAM, while sPAddr[17:29] specifies the address within the SRAM for a particular transaction.

QPtr Space

The goal of the QPtr Space is to allow user level access to Pcp and Pas/Vas queues' state in CTRL. The chosen interface encoding allows it to be implemented with almost the same hardware as that used for system level access to NESCtrl chip's state (See Section 3.2.4). QPtr Space provides a small optimization over the Cmd Space's Config Q State Access capability: a Read bus transaction to update Control Pointers can at the same time read the Status Pointers. (Control Pointers are: TxQ's P-ptr and RxQ's C-ptr; Status Pointers are: TxQ's C-ptr and RxQ's P-ptr.) (Table 25 shows where the queue state for each CompGp resides.)

Table 23 summarizes the encoding for QPtr Space. The aPAddr[6:9] field encodes the ComGp that the QPtr access is refering; the actual encoding is described in Table 24. Since QPtr Space is only interested in providing access to state in CTRL, and only to Pcp and Pas/Vas queues, aPAddr[6] is constrained to be #b0. The Fixed Fields aPAddr[10:11,14:16] are constrained to select the correct fields as defined by the queue state map in Section 4.

For Pcp comp gp, a 4B odd word (aPAddr[29] == 1) Read to the QPtr Space returns the Status Pointers associated with the specified ComGp: TxL's C-ptrs in data[1:7], TxH's C-ptr in data[9:15], RxL's P-ptrs in data[17:23], RxH's P-ptr in data[25:31]. (See Table 48 in Section 4.)

In addition, the other address bits may encode a request to update one of the control pointers. aPAddr[26] indicates whether there is an update associated with an access (4B Read or Write). If there is, other fields come into play. aPAddr[19:25] specifies the new pointer value. aPAddr[27:28]

Width (bits)	Bit field	Content	Description
4	[0:3]	0110	==== Fixed Field ====
2	[4:5]	10	==== Fixed Field ====
1	[6]	0	==== Fixed Field ====
3	[7:9]	X	ComGp.
2	[10:11]	01	==== Fixed Field ====
2	[12:13]	X	** Unused **.
3	[14:16]	100	==== Fixed Field ====
1	[17]	0	In the case of a Pcp TxQ, update P-ptr; for Pcp RxQ, update C-ptr; for Pas TxQ, update P-ptr; for Vas RxQ, update C-ptr.
		1	In the case of a Pcp TxQ or RxQ, update R ptr; for Vas RxQ, update Data C-ptr. This is undefined for a Pas TxQ.
1	[18]	X	** Reserved **.(To maintain compatibility with state access in Cmd Space.)
7	[19:25]	X	New ptr value.
1	[26]	0	No Update.
		1	Update.
1	[27]	0	Tx queue.
		1	Rx queue.
1	[28]	0	L Priority queue.
		1	H Priority queue.
1	[29]	X	For 32b reads, specifies which word is read.
2	[30:31]	00	==== Fixed Field ====

Table 23: QPtr Space aPAddr encoding.

indicates which queue's pointer to update, and in conjunction with aPAddr[17] indicates which pointer to update. Note that aPAddr[27:28] and aPAddr[17] play no part in the data read, only in the update. A 4B even word (aPAddr[29] == 0) Read returns data that contains no information that should be of interest to user code. The functionality is similar for Pas/Vas CompGp's state in CTRL, but in addition, a 4B even word (aPAddr[29] == 0) Read returns the RxL* P-ptr in data[17:23] and the RxH* P-ptr in data[25:31].

Not all combinations for identifying the pointer to update are valid, *e.g.*, aPAddr[17] == 1 for a Pas TxQ is not valid. It is an error on the part of software to specify updates to such addresses. Nevertheless, *the hardware has to be designed in such a way that such errors will neither cause damage to the hardware, nor place the NES hardware into an unrecoverable state that requires reset and/or power-cycle.*

Width (bits)	Bit field	Content	Description
4	[6:9]	`0000`	CTRL's copy of Pas/Vas-0 ComGp;
		`0001`	CTRL's copy of Pas/Vas-1 ComGp;
		`0010`	CTRL's copy of Pas/Vas-2 ComGp;
		`0011`	CTRL's copy of Pas/Vas-3 ComGp;
		`0100`	CTRL's copy of Pcp-0 ComGp.
		`0101`	CTRL's copy of Pcp-1 ComGp.
		`0110`	CTRL's copy of Pcp-2 ComGp.
		`0111`	CTRL's copy of Pcp-3 ComGp.
		`1000`	sBIU's copy of Pas/Vas-0 ComGp; used by sP.
		`1001`	sBIU's copy of Pas/Vas-1 ComGp; used by sP.
		`1010`	aBIU's copy of Pas/Vas-2 ComGp; used by aP.
		`1011`	aBIU's copy of Pas/Vas-3 ComGp; used by aP.
		`1100`	sBIU's copy of MemQ ComGp.
		`1101`	CTRL's copy of Overflow.
		`1110`	aBIU's copy of MemQ ComGp.
		`1111`	aBIU's copy of DMA ComGp.

Table 24: Encoding for ComGp field, bit[6:9] of aPAddr. aPAddr[6] differentiates between state in CTRL (aPAddr[6] == 0) and in one of the BIUs. aPAddr[7] differentiates between Pas/Vas (aPAddr[7] == 0) and Pcp queue. The exception to this encoding rules are the MemQ and Overflow ComGp's, which are special ComGp's. aPAddr[8] (roughly) distinguishes between resources devoted to the aP (aPAddr[8] == 1), and those used by the sP. aPAddr[9] distinguishes between even and odd resources.

Note the following unusual semantics of accesses to the QPtr space: (a) the data part of writes to the QPtr space is discarded; (b) both read and write can update pointers and all the information needed to achieve this are encoded in aPAddr.

ShTx Space

The encoding for ShTx Space is summarized in Table 26. Because only a subset of the ComGp

CompGp	Queue	P-ptr Location	C-ptr Location	P-ptr* Location	C-ptr* Location
Pcp[0-3]	Tx	CTRL (U_w, S_r, S_w)	CTRL (U_r, S_r, S_w)	—	—
	Rx	CTRL (U_r, S_r, S_w)	CTRL (U_w, S_r, S_w)	—	—
Pas/Vas[0-3]	Tx	CTRL (U_w, S_r, S_w) xBIU (U_w, S_w)	CTRL (U_r, S_r, S_w)	—	—
	Rx	CTRL (U_r, S_r, S_w) xBIU (S_w)	CTRL (U_w, S_r, S_w) xBIU (U_w, S_w)	CTRL (U_r, S_r, S_w)	CTRL (U_w, S_r, S_w)
MemQ	MemQOut[0-1]	CTRL (S_r, S_w) sBIU (S_w)	CTRL (S_r, S_w)	—	—
	MemQIn	aBIU (S_w) sBIU (S_w)	aBIU (S_w) sBIU (S_w)	—	—
	MemQDIn	aBIU (S_w)	aBIU (S_w)	—	—
DMA	TxQ	CTRL (S_w)	CTRL (S_w)	—	—
	RxCmdQ	CTRL (S_w)	CTRL (S_w)	—	—
	RxDQ	CTRL (S_w) aBIU (S_w)	CTRL (S_w) aBIU (S_w)	—	—

Table 25: The location of Q-state information, and accessibility from the processors. (S_r: System Read, S_w: System Write; U_r: User Read, U_w: User Write.)

Width (bits)	Bit field	Content	Description
4	[0:3]	`0110`	==== Fixed Field ====
3	[4:6]	`110`	==== Fixed Field ====
1	[7]	`0`	Pas-2
		`1`	Pas-3
20	[8:27]	`X`	Message content.
1	[28]	`1`	H Priority queue.
		`0`	L Priority queue.
3	[29:31]	`100`	==== Fixed Field ====

Table 26: ShTx Space aPAddr encoding.

supports ShTx operations, specifically, the Pas/Vas-X ComGps, and these are statically partitioned between the sP and the aP, the encoding therefore uses only 1b, aPAddr[7] to specify the CompGp.

ShRx Space

Width (bits)	Bit field	Content	Description
4	[0:3]	`0110`	==== Fixed Field ====
3	[4:6]	`111`	==== Fixed Field ====
8	[7:14]	`X`	** Unused **.
1	[15]	`0`	VasR-2 ComGp: do not poll.
		`1`	VasR-2 ComGp: poll.
1	[16]	`0`	PcpR-2 ComGp: do not poll.
		`1`	PcpR-2 ComGp: poll.
1	[17]	`0`	VasR-3 ComGp: do not poll.
		`1`	VasR-3 ComGp: poll.
1	[18]	`0`	PcpR-3 ComGp: do not poll.
		`1`	PcpR-3 ComGp: poll.
1	[19]	`X`	** Reserved **.
7	[20:26]	`X`	** Unused **.
1	[27]	`0`	H priority queue: not included in poll.
		`1`	H priority queue: included in poll.
1	[28]	`0`	L priority queue: not included in poll.
		`1`	L priority queue: included in poll.
1	[29]	`X`	When using 32b read, this specifies either the odd or the even word.
2	[30:31]	`00`	==== Fixed Field ====

Table 27: ShRx Space aPAddr encoding.

Table 27 summarizes the encoding for ShRx Space. Notice that ShRx queues to poll from is specified with bit-vectors: one for ComGp and another for H/L priority. The poll returns the short message (in the case of a Pcp queue, a pseudo-message) associated with the highest priority, non-empty queue. Prioritization among the 8 queues in the 4 communication groups is static, and the order, *in descending priority* is:

Queue Name	Expected Usage
VasR-2 H	user 0
PcpR-2 H	user 0
VasR-3 H	user 1
PcpR-3 H	user 1
VasR-2 L	user 0
PcpR-2 L	user 0
VasR-3 L	user 1
PcpR-3 L	user 1

3.2.2 sP Serviced Space

This space is used to support Global Shared Memory, and other communication to the sP. Expected usage of the latter category include:

1. aP posting DMA requests to sP.
2. Accesses to sP-dRAM. (provision for 32MB)
3. (In some usage of the machine) aP posting split-phase synchronization requests to sP.

There is little semantics associated with the address encoding of this space aside from fixed fields needed to specify this space. Table 28 shows the restrictions. The more interesting aspects of this space is how the aBIU react to different aP-bus transactions to this space. This is described in greater detail in Section 3.3, under "sP Serviced Space".

Width (bits)	Bit field	Content	Description
3	[0:2]	`001`	==== Fixed Field ====
		`010`	==== Fixed Field ====
29	[3:31]	`X`	Arbitrary bit values, subjected to sP's interpretation.

Table 28: sP Serviced Space aPAddr encoding.

3.2.3 aBIU Snooped aP dRAM Space (Local Global Memory Space)

This region of memory is snooped by the aBIU. Control over which aP-bus transactions to this space, if any at all, should be snooped is programmable. Details of this, including hardware support for fast snooping responsed, is described in Section 3.3, under "Local Global Space". There is little semantics associated with the address encoding of this space, aside from fixed fields needed to specify this space. Table 29 shows the restrictions. *This design assumes that there is more than 64MB of aP dRAM memory. Otherwise, no memory will be snooped.* Cls memory, used in hardware implemented snooping, is provided for address in the second 64MB.

Width (bits)	Bit field	Content	Description
6	[0:5]	`000001`	==== Fixed Field ====
26	[6:31]	`X`	Arbitrary bit values, subjected to sP's interpretation.

Table 29: Local Global Memory (Snooped aP dRAM) Space aPAddr encoding.

Important Note: *snooping by aBIU is* not *done when the aBIU itself is the master of the transaction.*

3.2.4 aBIU Cmd Space

The aBIU Cmd Space uses aPAddr[4:5] to distinguish between two major classes of commands: Config State Access Access and ImmdCmd. (A third class of command, Cls Access is only supported by the sBIU but not he aBIU.)

Width (bits)	Bit field	Content	Description
2	[4:5]	`XX`	** Reserved **. (Used to support Cls Access on the sP side.)
		`10`	ImmdCmd Space.
		`11`	Config State Space.

Cmd Space Config State Access

Accesses to Config State Space is similar to QPtr space accesses in that state update is specified in the transaction address only, with the new state value (7b) specified in aPAddr[19:25]. In addition, whether update is done is qualified by aPAddr[26]. The data part of a Write transaction is discarded. Unlike the User QPtr Space, however, state update is *not* supported with Read transactions; an attempt to do this results in undefined results with no guarantee on either the data returned or the effect of the attempted update.

Table 30 summarizes the encoding for accessing the Q State part (aPAddr[10] == 0) of Config State Space; it also gives the correspondence between aPAddr and CtrlAddr and xBIUAddr, which are internal addresses used by CTRL and the two BIUs respectively. Table 31 summarizes the encoding for accessing the SysState part (aPAddr[10] == 1) of Config State Space. The correspondence to CtrlAddr and xBIUAddr is similar for both tables. See Section 4 for address map of state registers.

Cmd Space: ImmdCmd

The aBIU supports a subset of the ImmdCmds supported by the sBIU. In particular, those ImmdCmds supported on the sP-bus which are directed to aBIU are not supported on the aP-bus. For uniformity, however, the same encoding is used for ImmdCmds on both the aP-bus and sP-bus. Thus, the ImmdCmd still has Primary Opcode field in aPAddr[12:13] (Table 32) but because there is only one category that is supported, this field can be ignored by the implementation. (See Appendix B for how the ImmdCmd's in this section are used.) Table 33 summarizes the encoding for ImmdCmd's that go to CTRL.

3.3 Bus Transactions Supported by aBIU

In describing how the aBIU reacts to aP-bus transaction, we divide bus transactions based on its Transaction Type (TT) into two groups. For one group, the AddrIndep TTs, the aBIU treats each transaction the same way regardless of the address associated with it. In fact, some transactions in this category have no address associated with them. Transactions in the first group are listed in Table 34. (We will discuss shortly what it means for aBIU Capture behavior to be "Configurable".)

The aBIU's response to the second group of bus transactions, the AddrDep TTs, depends on the address space that the transaction specifies. For this purpose, the aBIU supports three address spaces shown in Table 35. The bus Transaction Types in the second category are listed in Table 36. Transactions to "Other Spaces" is ignored by the aBIU. Those to the aBIU's "General Slave Space" are discussed in Section 3.3.4 below. As shown in Table 36, NES response to some transaction types addressed to either the "sP Serviced Space" or "aBIU Snooped Space" are configurable.

Width (bits)	Bit field	Content	Description
4	[0:3]	`0111`	==== Fixed Field ====
2	[4:5]	`11`	==== Fixed Field ====
4	[6:9]	`X`	ComGp.
			aPAddr[6] determines whether accessing CTRL (0) or xBIU (1) state.
			aPAddr[7] → CtrlAddr[2], xBIUAddr[2]
			aPAddr[8] → CtrlAddr[3], xBIUAddr[3]
			aPAddr[9] → CtrlAddr[4], xBIUAddr[4]
1	[10]	`0`	==== Fixed Field ==== (aPAddr[10] → CtrlAddr[0], xBIUAddr[0])
1	[11]	`0`	aPAddr[11] → CtrlAddr[1], xBIUAddr[1]
2	[12:13]	`X`	** Unused **.
4	[14:17]	`X`	aPAddr[14] → CtrlAddr[5], xBIUAddr[5]
			aPAddr[15] → CtrlAddr[8], xBIUAddr[8]
			aPAddr[16] → CtrlAddr[9], xBIUAddr[9]
			aPAddr[17] → CtrlAddr[10], xBIUAddr[10]
8	[18:25]	`X`	New value.
1	[26]	`0`	No Update.
		`1`	Update.
1	[27]		aPAddr[27] → CtrlAddr[6], xBIUAddr[6]
1	[28]		aPAddr[28] → CtrlAddr[7], xBIUAddr[7]
1	[29]	`X`	For 32b reads, specifies which word is read.
2	[30:31]	`00`	==== Fixed Field ====

Table 30: aPAddr encoding for Cmd Space used for Config Q State Access.

Width (bits)	Bit field	Content	Description
4	[0:3]	`0111`	==== Fixed Field ====
2	[4:5]	`11`	==== Fixed Field ====
4	[6:9]	`X`	aPAddr[6] determines whether accessing CTRL (0) or xBIU (1) state.
			aPAddr[7] → CtrlAddr[2], xBIUAddr[2]
			aPAddr[8] → CtrlAddr[3], xBIUAddr[3]
			aPAddr[9] → CtrlAddr[4], xBIUAddr[4]
1	[10]	`1`	==== Fixed Field ==== (aPAddr[10] → CtrlAddr[0], xBIUAddr[0])
1	[11]	`0`	==== Fixed Field ==== aPAddr[11] → CtrlAddr[1], xBIUAddr[1]
2	[12:13]	`X`	** Unused **.
4	[14:17]	`X`	aPAddr[14] → CtrlAddr[5], xBIUAddr[5]
			aPAddr[15] → CtrlAddr[8], xBIUAddr[8]
			aPAddr[16] → CtrlAddr[9], xBIUAddr[9]
			aPAddr[17] → CtrlAddr[10], xBIUAddr[10]
8	[18:25]	`X`	New value.
1	[26]	`0`	No Update.
		`1`	Update.
1	[27]		aPAddr[27] → CtrlAddr[6], xBIUAddr[6]
1	[28]		aPAddr[28] → CtrlAddr[7], xBIUAddr[7]
1	[29]	`X`	For 32b reads, specifies which word is read.
2	[30:31]	`00`	==== Fixed Field ====

Table 31: aPAddr encoding for Cmd Space used for Config SysState Access.

Width (bits)	Bit field	Content	Description
2	[12:13]	`0X`	Goes to CTRL for final actions.
		`1X`	** Unused **.

Table 32: ImmdCmd Primary Opcode encoding.

Width (bits)	Bit field	Content	Description
4	[0:3]	`0111`	==== Fixed Field ====
2	[4:5]	`10`	==== Fixed Field ====
1	[6]	`X`	** Unused **
3	[7:9]		** Unused **
2	[10:11]	`11`	==== Fixed Field ====
1	[12]	`0`	==== Fixed Field ====
1	[13]	`X`	** Reserved **
1	[14]	`X`	** Unused **
3	[15:17]	`000`	NES Reset
		`001`	** Unused **
		`010`	Arctic Ack
		`011`	Clear DMA state and DMA Cmd queue in CTRL.
		`100`	** Unused **
		`101`	** Unused **
		`110`	Interrupt aP.
		`111`	Interrupt sP.
11	[18:28]	`X`	** Unused **
3	[29:31]	`100`	==== Fixed Field ====

Table 33: Encoding for ImmdCmds that go to CTRL. aPAddr[15:17] is the secondary opcode field.

604 Bus Transaction Type	Has Address?	aBIU Captures?
EIEIO	No	Ignored
ECOWX	Yes	Ignored
ECIWX	Yes	Ignored
ICBI	Yes	Ignored
Sync	No	Configurable
Tlbsync	No	Configurable
Tlb Invalidate	Yes	Configurable

Table 34: aP-bus Transactions that are treated by the aBIU in the same way regardless of the address, if any, associated with the transaction.

Width (bits)	Bit field	Content	Description
5	[0:2,5]	`001 X` `010 X`	aBIU Slave Space: sP serviced Space
		`011 X`	aBIU General Slave Space: all aBIU Slave Space other than sP Serviced Space
		`000 1`	aBIU Snooped aP dRAM Space
		`000 0` `1XX X`	Other Spaces

Table 35: aBIU's classification of aP-bus Address Spaces.

604 Bus Transaction Type	sP serviced Space Capture?	aBIU Snooped Space Capture?	Other Slave Spaces Response
LWARX	Configurable	HAL-config	Ignore
Clean	Configurable	HAL-config	Ignore
Flush	Configurable	HAL-config	Ignore
Write-w-flush-atomic	Configurable	HAL-config	Maybe(Size dep)
write-w-flush	Configurable	HAL-config	Maybe(Size dep)
write-w-kill	Configurable	HAL-config	Ignore
Kill	Configurable	HAL-config	Maybe(Size dep)
Read	Configurable	HAL-config	Maybe(Size dep)
RWITM	Configurable	HAL-config	Maybe(Size dep)
Read-atomic	Configurable	HAL-config	Maybe(Size dep)
RWITM-atomic	Configurable	HAL-config	Maybe(Size dep)
RWNITC	Configurable	HAL-config	Maybe(Size dep)

Table 36: aP-bus Transactions that are treated by the aBIU in a way dependent on the address associated with the transaction.

Encoding	Capture Type	Semantics
00	Ignore; allow completion	The NES does not intervene in the bus transaction. NES does not retry the transaction; sP is not informed of the transaction.
01	Notification Only	NES does not retry bus transaction. If the bus transaction completes, *i.e.* no other device retries it, aBIU informs the sP by capturing the transaction and placing it in MemQIn and MemQDataIn (if there is data). (For transactions to sP Serviced Space which is reading data, the NES returns default data from DefaultDataAddr in aSRAM.)
10	Retry Indefinitely	This is only useful as a transient configuration. The sP must use this with care as it will lead to deadlock otherwise.
11	Approval	The bus transaction is retried and sP is informed of the transaction via MemQIn if the ApprovalReg is available to store the pending transaction. Retry of the pending transaction is stopped only when Approval is given. A match between a bus transaction and the captured transaction in the ApprovalReg requires matching the Addr only.

Table 37: NES's four different responses to aP-bus transactions to sP Serviced Space.

3.3.1 Configuration of aBIU Response to sP Serviced Space Transactions

Configuration of aBIU response to bus transaction is achieved using 2 programmable bits for each configurable transaction type. The encoding and semantics of the 2b are shown in Table 37. Bit[0] specifies whether the transaction should be retried, while Bit[1] specifies whether sP should be informed of the transaction. If the transaction should be retried *and* sP should be informed, it is considered as requiring *approval*. This encoding applies to all bus transactions with configurable aBIU response.

To support approval by the sP, the aBIU keeps track of one such pending transaction in a register (*ApprovalReg*) in the aBIU itself, which can be in one of four states (See Figure 7). This design is necessary as the decision of whether approval has been obtained has to be made very soon after an address bus transaction starts. (See Section 3.9 for timing requirement.)

When a transaction is captured, the aBIU places the address & control information into the MemQIn queue in the sSRAM. If data is transfered back to the NES as part of the transaction, it is placed into the MemQDataIn queue in aSRAM. Sometime later, the sP examines the captured transaction and carries out appropriate action. When capturing a transaction, the aBIU has to check for space availability in MemQIn and MemQDataIn (if that is used). For Approval transac-

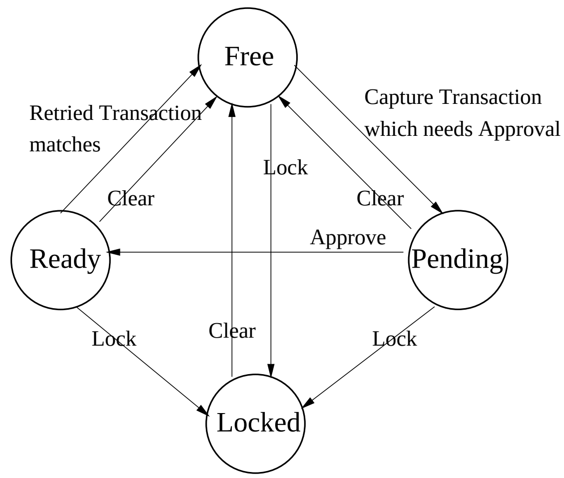

Figure 7: State transition diagram of the ApprovalReg.

604 Bus Transaction Type
Read
Read-atomic

Table 38: The aBIU asserts the _SHD signal for these bus transaction if the address falls into the sP Serviced Space or aBIU Snooped Space.

tions, the ApprovalReg in the aBIU must be available (*i.e.* in Free state) too. If there is insufficient space, the transaction is retried until space is available. A transaction requiring approval is retried until Approval is given by the sP. The latter involves a match between a bus transaction and that stored in the ApprovalReg (matching Addr only) with ApprovalReg in the Ready state. If data is to be supplied to the bus master (such as for a Read), the ApprovalReg's SRAM address field (supplied by the sP when it provides approval) is used to locate the data.

For transactions with data phase, only 1B, 4B-aligned, 8B-aligned, and 32B transactions are supported. The _SHD signal matters only for the bus transactions listed in Table 38, both of which are Approval type transaction. When giving Approval, the sP also specifies the _SHD signal to use when the transaction is allowed to complete.

3.3.2 Configuration of aBIU Response to Local Global Space (aka Snooped Space) Transactions

The sBIU reacts to each transaction to the Local Global Space in one of the four ways described in Table 37, but uses a more sophisticated way of determining the response which takes into account the cache-line address, Transaction Type, and the content of the ApprovalReg.

Associated with each of the twelve Transaction Types in Table 36 is a configurable table of eight entries, each containing 2b specifying the NES response to the transaction. For a bus transaction event, the actual row used is picked by a 3b cache-line state (Cls) kept by the aBIU for each (32B) cache-line. The Cls is kept in fast, single ported SRAM writen by the sP and read by the aBIU, with the aBIU as the arbitrator controling access to the single port of Cls memory. The outcome

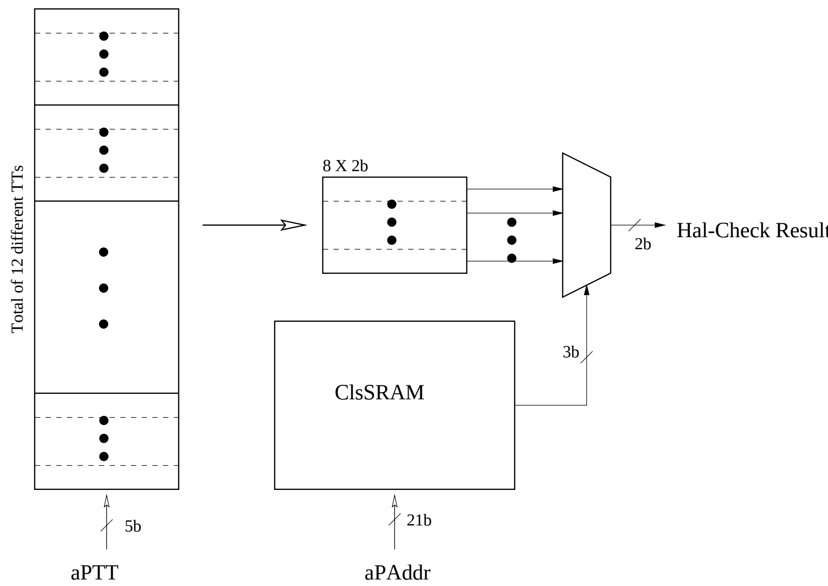

Figure 8: The HAL mechanism for the Snooped, Local Global Space in StarT-Voyager.

of this check, called the HAL Check Result, is a 2b quantity with the encoding shown in Table 37. (See Figure 8 for an illustration of this check.) The HAL-check result has to be combined with the outcome of the ApprovalReg check described next.

The ApprovalReg is used to store a transaction that require approval. (This is described for sP Serviced Space transactions in the previous Section.) The ApprovalReg is said to be Pending (*i.e.* have a pending transaction) after it has captured a transaction. The sP can remove a pending transaction by Clearing the ApprovalReg, or by giving Approval to the pending transaction. In the latter case, the ApprovalReg is said to be Ready. Subsequently, when the transaction which has been waiting for the approval is retried by its master, it is allowed to complete, and the ApprovalReg is implicitly Freed. Figure 7 shows the state transition of the ApprovalReg.

The two bits specifying the NES response is obtained as follows:

```
if ((ApprovalReg.state != Free)
    && (ApprovalReg.state != Locked)
    && (ApprovalReg's address matches bus transaction address)) {
    /* response comes from ApprovalReg */
  if (ApprovalReg.state == Pending) {
    retry, does not inform sP
  } else {  /* ApprovalReg.state == Ready) */
    response is: Ignore; allow completion
  }
} else {
  response comes from HAL
  if (HaL-Check Result) is ``No Retry and No Notification'',
    increment HAL-Statistics Reg.
}
```

For transactions with data phase, only 1B, 4B-aligned, 8B-aligned, and 32B transactions are supported. As in the case of sP Serviced Space, the _SHD signal is meaningful only for the bus transactions listed in Table 38. The value to use is picked by the transaction's 3b Cls from a configurable, eight entries, single-bit/entry, table.

Note that for transactions to Local Global Space that write-back data, notifying the sP without retrying the transaction includes capturing the data into MemQDIn. This is similar to the case for sP Serviced Space, with the slight difference that the NES is not the slave in this case, and therefore is not asserting _TA to control the rate at which data is arriving.

3.3.3 Alginment of Data Captured or Supplied by aBIU

When capturing data from or supplying data to sP Serviced Space and Snooped Space transactions, the aBIU uses 10b, 32B aligned base addresses internally. These addresses all refer to a 32B buffer in the aSRAM. Sources of these addresses are: MemQDIn P-ptr, ApprovalReg SRAM address, NES Mastered aP-bus Op Cmd, and DMA RxData Q C-ptr.

Data is located in the 32B buffer according to the offset implied by its aP-bus address. For example, for a 4B aP bus transaction with aPAddr[28:31] = #b1001, its location in the SRAM buffer with base address X is formed by concatenating X with #b1001. This applies regardless of whether the NES is supplying or accepting data.

In the case of burst transfer, data again resides within the 32B buffer at the offset implied by its aPAddr[28:31]. Implementors must take note of this, particularly in view of the critical-word-first requirement for Read-like bus transactions, but no critical-word-first for Write-like bus transactions.

Please also take note that each entry in MemQIn is 64b, while each entry in MemQDIn is 64B, with data occupying only the first 32B. (The reason for allocating the extra 32B is to allow the sP to add information there, and send both the captured data, and the added information with a Tag-on message.)

3.3.4 aBIU General Slave Space

The aBIU supports only a subset of 60X bus transactions to its slave address space. The actual subset supported is further dependent on the address sub-space within the aBIU slave space. For ease of reference, we categorizes the 60X bus transactions of interest to this document into several categories listed in Table 39. Using this categorization, Table 40 lists the transactions that are supported for each sub-space of the aBIU General Slave Space.

3.3.5 Dealing with Unsupported Transactions

When a transaction that is not supported for an address space is detected, the aBIU must still complete the required bus protocols so that the transaction completes on the aP-bus. The effect of writes is undefined, although the hardware implementation must ensure that protection is adequate. A simple way to achieve this is to make ensure that no write is performed to the SRAM when it is due to an unsupported transaction. In the case of reads, there is no guarentee on what data is returned. Detection of an unsupported 60X transaction is an error that is reported in the Err part of the NES MSR . A programmable interrupt mask allows this error to cause an interrupt if desired.

3.4 aBIU K-bus Services

Four kinds of requests to the aBIU arrive on the K-bus:

1. NES Mastered aP-bus Op Cmd (format in Table 41)

Trans Cat	TSiz	Tbst Asserted	TT	TT Description
1B Read (1B-R)	#b001	No	#b01010 #b11110 #b01011	Read Read-atomic RWNITC
1B Write (1B-W)	#b001	No	#b00010 #b00110 #b10010	write-w-flush write-w-kill Write-w-flush-atomic
4B Aligned Read (4B-AR)	#b100	No	#b01010 #b11110 #b01011	Read Read-atomic RWNITC
4B Aligned Write (4B-AW)	#b100	No	#b00010 #b00110 #b10010	write-w-flush write-w-kill Write-w-flush-atomic
8B Aligned Read (8B-AR)	#b000	No	#b01010 #b11110 #b01011	Read Read-atomic RWNITC
8B Aligned Write (8B-AW)	#b000	No	#b00010 #b00110 #b10010	write-w-flush write-w-kill Write-w-flush-atomic
32B Read (32B-R)	#b010	Yes	#b01010 #b11110 #b01011 #b01110 #b11010	Read Read-atomic RWNITC RWITM RWITM-atomic
32B Write (32B-W)	#b010	Yes	#b00010 #b00110 #b10010	write-w-flush write-w-kill Write-w-flush-atomic

Table 39: These are the 60X transactions that the aBIU supports as a slave device. They are divided into several categories. Table 40 gives further details of which categories are supported for a particular aBIU aP-bus address space.

NES sP Space	Trans Cat Supported
aBIU User Space/xSRAM Space	4B-AR 4B-AW 8B-AR 8B-AW 32B-R 32B-W
aBIU User Space/QPtr Space	4B-AR 4B-AW
aBIU User Space/ShTx Space	4B-AW 8B-AW
aBIU User Space/ShRx Space	4B-AR 8B-AR
aBIU Cmd Space	4B-AR 4B-AW 8B-AR 8B-AW

Table 40: This table specifies the type of transaction supported for each aP-bus NES address space.

2. DMA Rx Mastered Op Cmd (format in Table 42)
3. DMA Receive Cmd (format shown in Table 43)
4. ApprovalReg/Notification related Memory Emulation functions (format shown in Table 44)

The aBIU distinguishes between them using Bit[14,16], with the encoding shown below:

Width (bits)	Bit field	Content	Description
1	[14,16,18]	`00X`	DMA Receive Cmd
		`01X`	NES Mastered aP-bus Op Cmd
		`10X`	DMA Rx Mastered Op Cmd
		`110`	ApprovalReg/Memory Emulation Cmds
		`111`	ClsSRAM Update

3.5 aBIU External Interfaces

The external interface of aBIU can be grouped into the following categories:

Width (bits)	Bit field	Content
1	[0]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
10	[4:13]	32B aligned aSRAM address associated with any transfers to/from NES. (** [4] is constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write. This limits addressibility to half of the memory. This can be overcome with the use of 64b uncached writes.)
3	[14:16]	== Fixed field ==: must have the value #b001, indicating that this is an NES Mastered aP-bus Op cmd.
1	[17]	Ack_P (Whether this request should be acknowledged.)
1	[18]	_604WT
1	[19]	_604CI
1	[20]	_604GBL
1	[21]	_604TBST
1	[22]	604Tsiz[2]
5	[23:27]	604TT (Transaction Type).
1	[28]	Priority
3	[29:31]	(Forced to #b100 if composed by sP with short message style uncached write;) bit[29] ** Reserved ** bit[30:31] used for 604Addr[30:31].
30	[32:61]	604Addr[0:31]
2	[62:63]	604Tsiz[0:1]

Table 41: NES Mastered aP-bus Op Cmd format

Width (bits)	Bit field	Content
1	[0:2]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	General payload (** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
5	[5:9]	** unused **
1	[10]	Whether to clear ApprovalReg. Clear if 1. The clearing must only occur after the requested bus transaction has completed.
3	[11:13]	DMA Channel number.
1	[14]	== Fixed field ==: must have the value #b1. Distinguish between a DMA Receive Cmd and a DMA Rx Mastered Op Cmd.
1	[15:16]	== Fixed field ==: must have the value #b00, indicating that this is an NES Mastered aP-bus Op cmd.
1	[17]	Ack_P (Whether this request should be acknowledged.)
1	[18]	_604WT
1	[19]	_604CI
1	[20]	_604GBL
1	[21]	_604TBST
1	[22]	604Tsiz[2]
5	[23:27]	604TT (Transaction Type);
1	[28]	Priority
3	[29:31]	(Forced to #b100 if composed by sP with short message style uncached write; bit[30:31] used for 604Addr[30:31].)
30	[32:61]	604Addr[0:31]
2	[62:63]	604Tsiz[0:1]

Table 42: DMA Rx Mastered Op Cmd format

Width (bits)	Bit field	Content
1	[0]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	(** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
6	[5:9]	** unused **
1	[10]	== Fixed field ==: must have the value #b0. (Has to be 0 because 1 is used to indicate clearing ApprovalReg in DMA Rx Mastered Op Cmd.)
3	[11:13]	DMA Channel Number.
1	[14]	== Fixed field ==: must have the value #b0. Distinguish between a DMA Receive Cmd and a DMA Rx Mastered Op Cmd.
2	[15:16]	== Fixed field ==: must have the value #b00, indicating that this is a DMA cmd serviced by aBIU.
1	[17]	Ack_P (Whether this request should be acknowledged.)
7	[18:24]	** unused **
3	[25:27]	** unused **
1	[28]	MemQOut Priority.
3	[29:31]	** unused ** (Forced to #b100 if composed by sP with short message style uncached write.)
16	[32:47]	** unused **
16	[48:63]	Pending Cnt (of 32B Cache-line).

Table 43: DMA Receive Cmd format

Width (bits)	Bit field	Content
1	[0]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	(** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
9	[5:13]	** unused **
1	[14]	== Fixed field ==: must have the value #b1.
2	[15:16]	== Fixed field ==: must have the value #b01, indicating that this is an an Approval Cmd serviced by aBIU.
1	[17]	Ack_p.
1	[18]	== Fixed field ==: must have the value #b0
9	[19:27]	** unused **
1	[28]	MemQOut priority.
3	[29:31]	** unused ** (if request composed by 32b uncached write, this is constrained to have the value #b01
13	[32:44]	** unused **
1	[45]	Used only for ([47:49] == 000) when this indicates the _SHD signal to use.
1	[46]	** unused **
3	[47:49]	Secondary Opcode.
10	[50:59]	aSRAM address for any associated data.
4	[60:63]	** unused **

Width (bits)	Bit field	Content	Description
3	[47:49]	`000`	Approval for pending bus transaction in ApprovalReg
		`001`	Lock/reserver ApprovalReg
		`010`	Reset (to 0) DMA RxData queue C-ptr in aBIU.
		`011`	Clear/free ApprovalReg; effectively discarding any captured transaction or previous lock/reservation.
		`100`	Nop. This allows issuing a synchronouse Clswrite enable only command.
		`101`	Lock Notification resource, i.e. any transaction that needs notification will be retried until this lock is cleared.
		`110`	** Unused **
		`111`	Enable Notification resource, undoing any previous lock.

Table 44: ApprovalReg/Memory Emulation Cmd format

Interface	Interfacing with Functional Block
aP-bus	aP memory bus
a-port	aSRAM
Shared Memory	sBIU
AC-bus	CTRL
CA-bus	CTRL
QPtr Status	CTRL
ShTx/ShRx	CTRL
MemQIn	CTRL
Data Motion(inter-SRAM transfer)	CTRL
K-bus	CTRL
NES Mastered Ops	CTRL

3.5.1 aBIU aP-bus Interface

Signal direction is described with respect to the aBIU, *i.e.* "I" means an input to the aBIU.

Name	I/O	# bits	Description
aPSysClk	I	1b	Sysclk; bus clock
aPAddr	I/O	32b	address
_aPNesBr	O	1b	Bus request
_aPNesBg	I	1b	Address bus grant
_aPAbb	I/O	1b	Address bus busy
_aPTs	I/O	1b	Transaction start; start of address transaction
_aPXats	I	1b	Extended Address Transfer start (_XATS), indicates direct-store operation, to I/O space. (May not be needed.)
aPTT	I/O	5b	Transfer Type
aPTc[0]	I	1b	Transaction code; This is not among the signals on Doral's processor card connector but is output from Mamba, and we really want to see this signal.
_aPTbst	I/O	1b	Tbst
aPTSiz	I/O	3b	TSiz
_aPGbl	I/O	1b	Global
_aPCi	I/O	1b	Cache Inhibit
_aPWt	I/O	1b	Write through
_aPAAck	I	1b	AAck; Address Acknowledgement
_aPL2Hit	O	1b	L2Hit; use this to indicate to memory controller that it is a transaction to NES Space.
_aPArtry	I/O	1b	ARTRY; Address Retry
_aPShd	I/O	1b	Shared
_aPDbg	I	1b	Data bus grant
_aPDbb	I/O	1b	Data bus busy
_aPTa	I/O	1b	Transaction acknowledge; slave indicating to master data is valid/consumed.
_aPHReset	I	1b	HRESET; Hard Reset
_aPSReset	I	1b	SRESET; Soft Reset
_aPInt	I	1b	INT; Interrupt

The following signals are on the 604 bus, but are not needed by the NES on the aP side:

Name	I/O	# bits	Description
aPAddrP	I/O	4b	address parity
aPTc[1:2]		2b	Transaction code; We believe this is not needed as they are not among the signals on Doral's processor card connector.
_APE		3b	Address parity error
_CSE		3b	Cache set element
_DPE		1b	Data Parity Error
_DRTRY		1b	Data Retry
_TEA		1b	Transfer Error Acknowledge
_DBDIS		1b	
_DBWO		1b	Data Bus Write Only; Not sure if this is needed; depends on the motherboard.
L2_INT		1b	
_Ckstp_in		1b	Checkstop input
_MCP		1b	Machine check interrupt
_SMI		1b	System Management interrupt

3.5.2 aBIU a-port Interface

Signal direction is described with respect to the aBIU, *i.e.* "I" means an input to the aBIU.

Name	I/O	# bits	Description
aPortAddr	O	12b	aSRAM aP-bus port address
aPortR/_W	O	2b	aSRAM odd 32b and even 32b R/_W;
_aPortOE	O	2b	aSRAM odd 32b and even 32b _OE;
_aPortCE	O	2b	aSRAM odd 32b and even 32b _CE;

3.5.3 aBIU Shared Memory Interface

Signal direction is described with respect to the aBIU, *i.e.* "I" means an input to the aBIU.

Name	I/O	# bits	Description
AcdOpV	I	1b	whether AcdOp is valid.
AcdOp	I	3b	sBIU convey shared memory related requests from sP
AcdApprAddr	I	12b	aSRAM address where data can be found/placed if the captured pending request is to be handled as Approval-w-Data.
AcdClsSync	I	1b	When AcdOpV, this indicates whether a synchronous ClsWrite enable is also requested.
AcdOpDone	O	1b	Notification from aBIU to sBIU that AcdOp is completed.
ClsUpdateReq	I	1b	Post sP clSRAM Update Request.
ClsSyncUpdate	I	1b	Indicates whether the requested update is a Synchronized one.
ClsUpdateDone	O	1b	Notification from aBIU to sBIU that clSRAM update is completed.

Signal direction is described with respect to the aBIU, *i.e.* "I" means an input to the aBIU.

Name	I/O	# bits	Description
clsData	I	4b	cache-line state data
clsRSel	O	2b	cache-line state R/W
clsCS	O	2b	cache-line state chip select
clsaPEn	O	1b	aP bus address tri-state enable
clssPAddrOutEn	O	1b	sP bus HAL address register output enable

3.5.4 aBIU AC-bus Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
AcBusPassThru	I	1b	When AcBusV is asserted, AcBusPassThru indicates whether the requested operation is just passing through CTRL, destined for aBIU.
AcBusAddr	I	11b	When AcBusV is asserted, AcBusAddr is the address of the state that is accessed.
AcBusData	I	7b	When AcBusV is asserted, and AcBusRW indicates a write, AcBusData contains the data to be written.
AcBusRW	I	1b	When AcBusV is asserted, AcBusRW indicates whether the requested operation is state read, or state write
AcBusV	I	1b	AC bus signals valid; when asserted, this indicates that there is valid data on the AC bus.
AcBusDone	O	1b	Pulsed for one cycle to indicate that the AC bus operation requested by aBIU has been completed.

3.5.5 aBIU CA-bus Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
CaBusAddr	O	8b	When CaBusV is asserted, CaBusAddr is the address of the state that is accessed.
CaBusData	O	7b	When CaBusV is asserted, and CaBusRW indicates a write, CaBusData contains the data to be written.
CaBusRW	O	1b	When CaBusV is asserted, CaBusRW indicates whether the requested operation is state read, or state write.
CaBusV	O	1b	CA bus signals valid; when asserted, this indicates that there is valid data on the CA bus.
CaBusDone	I	1b	Pulsed for one cycle to indicate that the CA bus operation requested by aBIU has been completed.

3.5.6 aBIU ShTx/ShRx Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
AbiuShReq	I	1b	aBIU short message I-bus request
AbiuShGnt	O	1b	aBIU short message I-bus grant
AbiuLateAck	I	1b	aBIU asking that a late ack be returned to Arctic.
AbiuPcpEmpty	O	4b	Indicates whether a PcpR queue is empty. There are 4 different PcpR queues, so 1b for each queue.

3.5.7 aBIU MemQIn Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
AcdIBusReq	I	1b	Shared memory I-bus reqeust
AcdIBusGnt	O	1b	CTRL granting aBIU usage of I-bus for transfer of shared memory request address and control information into MemQIn.

3.5.8 aBIU Data Motion (inter-SRAM transfer) Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
AbiuDmReq	I	1b	aBIU Data Motion request
AbiuDmGnt	O	1b	aBIU Data Motion grant
SbiuDmDone	O	1b	aBIU Data Motion done

3.5.9 aBIU K-bus Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
KBusDataAC	I	32b	K-bus data from aBIU to CTRL
KBusDataCA	O	64b	K-bus data from CTRL to aBIU
KBusAddr	I	13b	K-bus address

3.5.10 NES Mastered aP-Bus Transaction Interface

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
aPBusReq	O	1b	Internal request for aBIU to perform a mastered operation on aP-bus
aPBusDone	I	1b	The previous internal (from MemQOut, Reclaim Engine, Dma Read Engine, for Dma Rx Cmd) request to aBIU is done

3.6 aBIU Actions

This section describes the abstract functionality of the aBIU in response to aP-bus transactions to various aBIU spaces, and internal requests coming from other parts of the NES. Determination of the space that an aP-bus transaction is accessing is done by using the decoding specified in Tables 20 and 21. *In the case of write-like transactions that are not supported, write-enable for a-Port is never asserted.* The description below sometimes introduces "symbolic variables", and assign them values. This is to make descriptions clearer and more precise. The assignment has a C-like semantics of treating the assignment as an expression that returns the assigned value.

3.6.1 aBIU User Space: aSRAM Space

Accesses to this space conforms to the encoding in Table 22, with the further constraint that aPAddr[16] is #b0.

- ◇ **aPortAddr:** aPAddr[17:28]

- ◇ aBIU asserted the expected control signals on the aP-bus, and a-Port to complete the read or write. For 32B write, sPortAddr has to be incremented for the 4 beats. Reminder: 32B-R requires critical word first, while 32B-W is zeroth word first.
- ◇ No involvement of any other NES component.

3.6.2 aBIU User Space: sSRAM Space

Accesses to this space also conforms to the encoding in Table 22, but with aPAddr[16] of #b1. (See Section 5.6.1 for format of Data-Motion command.)

- ◇ **aPortAddr:** sSRAM-tmp ++ aPAddr[27:28]

- ◇ For write (4B-AW, 8B-AW, 32B-W),
  - After data from sP-bus is ready (potentially pipelinable), arrange with Data-Motion unit in CTRL for appropriate granularity transfer from sSRAM-tmp to final sSRAM location (as specified by sSRAM address in aPAddr[17:29]).
- ◇ For read (4B-AR, 8B-AR, 32B-R),
  - First issue a Data-Motion command to move data from sSRAM location (aPAddr[17:29]) to be transfered to sSRAM-tmp buffer in aSRAM. (Critical word first for Data-Motion will be helpful.)
  - When transfer is far along enough (pipeline if possible), start asserting _sPTa so that data is supplied to sP-bus.

3.6.3 aBIU User Space: QPtr Space

Accesses to this space conforms to the encoding in Table 23. See Section 4 to see how requests are forwarded to CTRL with CtrlAddr and used locally as abiuAddr.

- ◇ **aPortAddr:** qptr-base ++ aPAddr[7:9,29]

- ◇ If read,

  just assert _sPTa as required data is already in aSRAM.

- For both read and write,
  if (aPAddr[26] == Update)
  if (Updating QPtr state in aBIU, *i.e.* (aPAddr[6,9] == #b11))
  update aBIU QPtr state
  else
  forward the update request to CTRL over the AC bus.
- For write, there is a special requirement that no write to aSRAM should occur, *i.e.* aPortR/_W should be R, and _aPortOE should be deasserted.

3.6.4 aBIU User Space: ShRx Space

Accesses to this space conforms to the encoding in Table 27.

- **aPortAddr:** ShRx-likeAddrGen[aPAddr[15:19,27:28]]

- For read, ShRx-likeAddrGen determines the address to poll from as follows:
  If ((aPTSiz[0] == #b1 /* 4B */) AND (AccessedOnce))
  AccessedOnce = False
  return (One-poll-addr)
  Else
  One-poll-addr =
  (determine among the polled queues, specified by aPAddr[15:19,27:28], the highest priority non-empty queue. If the queue is a VasR queue or MemQIn, ShRx-likeAddrGen returns the C-ptr. If the queue is a Pcp queue, the corresponding ContAddr is returned. If all the queues are empty, the Empty-Msg's ContAddr is returned.)
  If a C-ptr, increment the C-ptr
  (Incrementing C-ptr requires updating the aBIU local copy, and then the CTRL's copy.)
  If (chosen queue uses Late_Ack)
  pulse AbiuLateAck signal to indicate need to provide one ack to Arctic.
  AccessedOnce = True
  return (One-poll-addr)
- There should be no write to this space. As in the case of QPtr Space, we require that no write to sSRAM occurs in the event of a aP-bus write transaction to this space.

Implementor's Notes: AbiuLateAck is pulsed possibly before the second 32b of the message is read. This appears to prematurely release the buffer but is okay because the circular buffer with P-ptr and C-ptr used in the implementation always leaves one entry empty.

3.6.5 aBIU User Space: ShTx Space

Accesses to this space conforms to the encoding in Table 26.

- **aPortAddr:** ShTx-likeAddrGen[aPAddr[3:4,28]]

- ShTx-likeAddrGen returns the P-ptr of the TxQ specified by its input parameter(aPAddr[3:4,28]).
- For write,
  If ((queue is in 32b-ShTx mode) and (aPTSize[0] == #b1 /* 4B */)),
    arrange to write aPAddr into sSRAM at the higher word (bank 0) of address used for aPortAddr.
  if (((aPAddr[29] == #b0) AND (aPTSiz[0] == #b0 /* 8B */))
      OR ((aPAddr[29] == #b1) AND (aPTSiz[0] == #b1)))
    Increment P-ptr
    (This requires first updating the aBIU local copy, and then the CTRL's copy when it is known that any write of aPAddr in sSRAM has been completed, and the data part of the write bus transaction has completed. The timing has to be coordinated such that when CTRL starts pushing this packet out to TxU, it will read the correct data from the sSRAM.)

3.6.6 aBIU Cmd Space: Config State Access

Accesses to this space conforms to the encoding in Tables 30 and 31. See Section 4 to see how requests are forwarded to CTRL with CtrlAddr and used locally as aBIUAddr.

- **aPortAddr:**
  if (Read AND (aPAddr[10:11] == #b00)
      AND ((aPAddr[6] == #b0) OR (aPAddr[6:9] == #b1101)))
    If (aPAddr[6] == #b0)
      `qptr-base` ++ aPAddr[7:9,29]
    else
      `Overflow-queue-statue`
  else
    aBIU-qconfig-tmp
- If read,
  if ((aPAddr[10:11,16] != #b001)
      AND ((aPAddr[6] == #b0) OR (aPAddr[6:9] == #b1110)))
    arrange with CTRL to read value into aBIU-qconfig-tmp in sSRAM before asserting _aPTa.
  else
    just assert _aPTa as required data is already in sSRAM. is returned from
- Else
  if (aPAddr[26] == Update)
    if (Updating QPtr state in aBIU, *i.e.* (aPAddr[6,9] == #b11))
      update aBIU QPtr state
    else
      forward the update request to CTRL over the AC bus.
- For write, there is a special requirement that no write to sSRAM should occur, *i.e.* aPortR/_W should be 1, and _aPortOE should not be asserted.

3.6.7 aBIU Cmd Space: ImmdCmd

Accesses to this space conforms to the encoding described in Tables 33.

- ◇ **sPortAddr:** `aBIU-qconfig-tmp`

- ◇ Decode and forward the request to CTRL.
- ◇ Data part of the transaction has no meaning to NES.
- ◇ Switch (ImmdCmdPrimaryOp = aPAddr[12])

    Case (#b0 /* Goes to CTRL */):

        supply CtrlAddr[0:1] with aPAddr[10:11] /* #b11 */
        supply CtrlAddr[8:10] with aPAddr[15:17]
        supply CtrlAddr[2:4] with aPAddr[7:9]
        supply CtrlAddr[6:7] with aPAddr[27:28]
        supply CtrlData[2:6] with aPAddr[21:25]
        break;

    Case (#b1 /* Goes to aBIU */):

        ** Error **
        break;

3.6.8 Request from CTRL over the CA-bus Interface

The aBIU receives config state access request from CTRL over the CA-bus Interface, for state that resides in the aBIU. A request has arrived if CaBusV is asserted. When the aBIU has performed the requested operation, it asserts the CaBusDone for one cycle.

3.6.9 sP Serviced Space

This command conforms to the encoding in Table 28.

- ◇ **aPortAddr:** DataInP(aPTT) ?
  BcpAddrGen(MemQDataIn.P-ptr)
  : ApprovalSramAddr

```
◇ Switch (sSsAddrDepResp.lookup(aPTT)):
    Case (Ignore):
        Allow transaction to complete. (No retry)
        break;
    Case (Notification Only):
        If (Not (BcpFullP(MemQIn)
                OR (BcpFullP(MemQDataIn) AND DataInP(aPTT))
                OR NotificationLocked ))
            Allow transaction to complete(No retry); Assert _SHD
            If (other bus devices do not retry the transaction)
                Place the transaction address and control information into
                BcpAddrGen(MemQIn.P-ptr).
                If (DataInP(aPTT)),
                    place the captured data into MemQDataIn.
                Else if (DataOutP(aPTT)),
                    supply data from MissPatternAddr.
        Else
            aBIU Retries transaction.
        break;
    Case (Approval):
        If ((ApprovalReg.state == Ready)
                AND ApprovalReg.Match(aPAddr)
            /* Approval available */
                AND (Not (DataInP(aPTT) AND BcpFullP(MemQDataIn))))
            /* If need to capture data, check space available */
            Do not retry transaction;
            _SHD signal taken from ApprovalReg.SHD.
            If (no other bus device retries transaction)
                supply/accept data if necessary;
                ApprovalReg.Clear();
        Else
            If ((ApprovalReg.state() == Free)
                    AND (Not (BcpFullP(MemQIn))))
                ApprovalReg.Capture(aPAddr);
                Place the captured transaction address and control information
                into MemQIn.
                ApprovalReg.state() = Pending;
            aBIU Retries transaction.
        break;
    Case (Retry Indefinitely):
        aBIU Retries transaction.
        break;
```

Implementor's Notes: Transactions in the "Notification Only" category must not wait for ApprovalReg to be Free; otherwise, there is danger of deadlock.

Implementor's Notes: Unit of allocation of MemQDIn is 64B, with data occupying only the first 32B.

3.6.10 Local Global Memory Space (aka Snooped Space)

This command conforms to the encoding in Table 29.

```
◇ aPortAddr:      (DataInP(aPTT) AND
                      (Not ((ApprovalReg.AddrMatch(aPAddr))))) ?
                  BcpAddrGen(MemQDataIn.P-ptr)
                  : ApprovalSramAddr
◇ If ((ApprovalReg.AddrMatch(aPAddr))
          AND ((ApprovalReg.state() == Ready)
                  OR (ApprovalReg.state() == Pending)))
      Switch (ApprovalReg.state())
          Case (Ready):
              Nes-resp = Ignore;
              Nes-shd = ApprovalReg.SHD;
              break;
          Case (Pending):
              Nes-resp = Retry Indefinitely;
              Nes-shd = ApprovalReg.SHD; /* don't care */
              break;
◇ Else
      Nes-resp =
          Entry-sel(SsTTHalResp(aPTT),
                  ClsSRAM.read(aPAddr[6:26]));
      Nes-shd =
          If ((aPTT == Read OR Read-atomic))
              Entry-sel(SsTTShdResp(aPTT),
                  ClsSRAM.read(aPAddr[6:26]));
          Else
                  deassert; /* don't care */
```

```
◇ Switch (Nes-resp)
      Case (Ignore):
          Allow transaction to complete (No retry), i.e.
          If (ApprovalReg.state() == Ready)
              If (DataOutP(aPTT))
                  Assert L2 claim, supply data from ApprovalReg.Sram;
              Else if (DataInP(aPTT))
                  Assert L2 claim, write data into ApprovalReg.Sram;
          Else /* HAL says okay */
              simply allow data to come from/go to aP-DRAM.
              Increment HAL-Statistics Reg.
          break;
      Case (Notification Only):
          If (Not (BcpFullP(MemQIn)
                  OR (BcpFullP(MemQDataIn) AND DataInP(aPTT))
                  OR NotificationLocked))
              Allow transaction to complete(No retry)
              If (other bus devices do not retry the transaction)
                  Place aPAddr and control info into MemQIn.
                  /* L2 claim is not asserted; this is to allow */
                  /* data to be written to aP-DRAM as well.  aBIU */
                  /* does not supply _TA, but listens to it as */
                  /* though it is a bus master performing Read.  */
              If (DataInP(aPTT))
                  place the captured data into MemQDataIn.
                      /* MemQDataIn allocates space in 32B chunks; */
                      /* Smaller unit writes are done to the */
                      /* offset specified by aPAddr[27:31].  */
              Else if (DataOutP(aPTT))
                  supply data from MissPatternAddr.
          Else
              aBIU retries transaction.
          break;
      Case (Approval):
          /* Approval needed, this must have come from HAL */
          If ((ApprovalReg.state() == Free)
                  AND (Not (BcpFullP(MemQIn))))
              ApprovalReg.Capture(aPAddr);
              Place the sPAddr and control info into MemQIn.
              ApprovalReg.state() = Pending;
          aBIU Retries transaction.
          break;
      Case (Retry Indefinitely):
          aBIU Retries transaction.
          break;
```

Implementor's Notes: Transactions in the "Notification Only" category must not wait for ApprovalReg to be Free; otherwise, there is danger of deadlock.

Implementor's Notes: Unit of allocation of MemQDIn is 64B, with data occupying only the first 32B.

3.6.11 NES Mastered aP-bus Op

This command conforms to the encoding in Table 41.

- **aPortAddr:** aPAddr[5:14] ++ aPAddr[59:61] ++ aPAddr[30:31]
- **aPAddr:** aPAddr[32:61] ++ aPAddr[30:31]
- **aPTT:** aPAddr[23:27]
- **aPTSiz:** aPAddr[62:63] ++ aPAddr[22]
- **aPTbst:** aPAddr[21]
- **aPWt:** aPAddr[18]
- **aPCi:** aPAddr[19]
- **aPGbl:** aPAddr[20]

- Arbitrate for aP-bus.
- When bus is granted, perform the bus transaction. The exact action depends on aPTT and aPTSiz.

  Normally, if the transaction is retired, the aBIU again arbitrates for aP bus. The aBIU keeps count of the number of times the transaction is retried. If it exceeds MOpRetryLimit, stop retrying and consider the transaction as having failed.
- While waiting for the bus to be granted,

  ** as an "optimization" **

  We check the pending request against aP-bus transactions initiated by other bus-masters that occur during the wait. First, compare aPAddr and aPTSiz, aPTbst. If they match, some special cases for short-circuited "completion" of command are:
  - The pending transaction is "clean", and the transaction on the bus is a "write-w-kill".
  - The pending transaction is "Flush" and the transaction on the bus is a "write-w-kill" and (TC[0] == 1).
  - The pending transaction is "Read" and the transaction on the bus is a "write-w-kill" and (TC[0] == 1). (NB: (TC[0] == 0) alone is not okay because the write-w-kill may have been caused by a dcbst which means that the L1/L2 caches still has an E copy; unfortunately, a snoop-push triggered by a Read has (TC[0] == 0). To take advantage of this, we also need (TC[1] == 1).) We can grab the data as it is written back to aP-dRAM, and consider this NES mastered op done.
  - The pending transaction is "RWITM", and the transaction on the bus is a "write-w-kill" and (TC[0] == 1). We can grab the data as it is written back to aP-dRAM, and consider this NES mastered op done.
- If transaction failed,

  place the command itself, with success_P bit (aPAddr[29]) set to indicate failure, into the MemQIn queue as acknowledgement.

- ⋄ Else If (Ack_P = aPAddr[17])
  place the command itself (with success_P bit (aPAddr[29]) set to indicate success) into the MemQIn queue as acknowledgement.

comment: *There is currently no special provision to deal with access to a wild address that no device will respond to. What happens in such cases is the Union chip time-outs after 8ms or so. (Need to confirm this information, which was in one of the emails that Ek forwarded.) This will cause a machine-check or some other unrecoverable error.*

3.6.12 DMA Rx Mastered Op

This command conforms to the encoding in Table 42. The action described below is similar to that for NES Mastered aP-bus Op, except that aPortAddr is computed differently, and DmaRxPendingCnt for the appropriate Dma Channel is decremented.

- ⋄ **aPortAddr:** BcpAddrGen(DmaRxDataQ.C-ptr)
- ⋄ **aPAddr:** aPAddr[32:61] ++ aPAddr[30:31]
- ⋄ **aPTT:** aPAddr[23:27]
- ⋄ **aPTSiz:** aPAddr[62:63] ++ aPAddr[22]
- ⋄ **aPTbst:** aPAddr[21]
- ⋄ **aPWt:** aPAddr[18]
- ⋄ **aPCi:** aPAddr[19]
- ⋄ **aPGbl:** aPAddr[20]

- ⋄ Arbitrate for aP-bus.
- ⋄ When bus is granted, perform the bus transaction. The exact action depends on aPTT and aPTSiz.
  Normally, if the transaction is retired, the aBIU goes back to arbitrating for the bus. The aBIU keeps count of the number of times the transaction is retried. If it exceeds MOpRetryLimit, stop retrying and consider the transaction as having failed.
- ⋄ (DmaRxPending[(DmaChannelNum = aPAddr[11:13])]) − = 1
- ⋄ If (((DmaRxPending[DmaChannelNum]) == 0)
  AND (DmaRxAck_P[DmaChannelNum]))
  place a DMA Receive Cmd acknowledgement (only the DMA Channel matters) into the MemQIn queue.
- ⋄ If transaction failed,
  place the command itself, with success_P bit (aPAddr[29]) set to indicate failure, into the MemQIn queue as acknowledgement.
- ⋄ Else if (Ack_P = aPAddr[17])
  place the command itself (with success_P bit (aPAddr[29]) set to indicate success) into the MemQIn queue as acknowledgement.
- ⋄ If (IncDMAPtr = aPAddr[0])
  (DmaRxDataQ.C-ptr)++
- ⋄ If (FreeApprovalReg = aPAddr[10])
  ApprovalReg.state = Free

3.6.13 DMA Receive Cmd

This command conforms to the encoding in Table 43.

◇ DmaRxPendingCnt[(DmaChannelNum = KBusDataCA[11:13])]
  ← (PendingCnt = KBusDataCA[48:63])

◇ If (Ack_P = KBusDataCA[17])
  place the command itself (with success_P bit (KBusDataCA[29]) appropriately set) into the MemQIn queue as acknowledgement.

3.6.14 Request from sBIU over the (ACD) Shared Memory Interface

The commands in this section are used to implement memory emulation, including shared memory. Commands are issued by the sP, mostly via MemQOut. Cmd Space ImmdCmds currently supports only one of the commands: that to Clear the ApprovalReg. Command from MemQOut conform to the encoding in Table 44.

◇ Switch (Secondary-opcode = KBusDataCA[47:49]):
  Case (000): `/* Approval for pending Transx */`
    ApprovalSramAddr = KBusDataCA[50:61];
    ApprovalReg.state = Ready;
    break;
  Case (001): `/* Lock/reserve ApprovalReg */`
    ApprovalReg.state = Locked;
    break;
  Case (010): `/* Reset DMA RxDataQ C-ptr in aBIU */`
    DmaRxDataQ.C-ptr = 0;
    break;
  Case (011): `/* Clear/free ApprovalReg */`
    ApprovalReg.state = Free;
    break;
  Case (100): `/* Nop */`
    break;
  Case (101): `/* Lock Notification resource */`
    NotificationLocked = TRUE;
    break;
  Case (110): `/* ** Unused ** */`
    ** Error **;
    break;
  Case (111): `/* Enable Notification resource */`
    NotificationLocked = FALSE;
    break;

◇ If (Ack_p = KBusDataCA[17])
  If (Secondary-opcode == 000)
      `/* Approval for pending Transx */`
    wait until ApprovalReg.state == Free.
        `/* i.e.  Approval has been used.  */`
  place the command itself (with success_P bit (KBusDataCA[29]) appropriately set) into the MemQIn queue as acknowledgement.

3.7 aBIU State

This section describes the state in aBIU, categorizing them into three groups:

Config Queue State	Queue related state, that is configurable by software on the sP or aP.
Config SysState	Config state that is not queue related, usually only reable but not writeable.
Internal State	Internal state of the aBIU, not directly accessible by software.

3.7.1 aBIU Configurable Queue State

aBIU contains the following QState: (One-poll-addr and UsedOnce must be both readable and writeable by system code to support context switching.)

- ShTx-like QState:
  - 4 x {P-ptr (7b) + Base (9b) + Bound (5b)}

  These 4 are used for:
    - PasT-2L
    - PasT-2H
    - PasT-3L
    - PasT-3H
- ShRx-like QState:
  - 4 x {C-ptr (7b) + P-ptr (7b) + Base (9b) + Bound (5b) + LateAck (1b)}
  - 4 x emptyP (1b)

  These 4 are used for:
    - VasR-2L
    - VasR-2H
    - VasR-3L
    - VasR-3H
  - 3 x PcpContAddr (13b)

  These 3 are used for:
    - PcpR-2
    - PcpR-3
    - Empty-Msg

  This is used for one-poll:
    - One-poll-addr (12b)
    - AccessedOnce (1b)
- MemQDataIn QState:
  - 1 x {C-ptr (7b) + P-ptr (7b) + Base (9b) + Bound (5b)}
  - 1 x FullP (1b)
- MemQIn QState:
  - 1 x {C-ptr (7b) + P-ptr (7b) + Base (9b) + Bound (5b)}
  - 1 x FullP (1b)

- General aPortAddr State:
  - 1 x aBIU-qconfig-tmp (12b)
  - 1 x qptr-base (8b)
  - 1 x sSRAM-tmp (10b)
- DmaRx QState:
  - 1 x DmaRxDataQ {C-ptr (7b) + Base (9b) + Bound (5b)}
- Shared Memory support state:
  - 3 x AddrIndepResp (2b)
  - 12 x sSsAddrDepResp (2b) (For sP Serviced Space)
  - 12 x aSsHalDecode (8b) (For aBIU Snooped Space)

3.7.2 aBIU Configurable SysState

- aBIU MSR:
  - 4? x 1b

    These are used to indicate internal resource utilization:
    - Tx Addr Write Pending
    - sSRAM Write Pending
    - State Update Pending
    - aBIU Mastered Op Ack Pending

comment: *The aBIU MSR state has to be re-visited. There can be 1 pending Tx Addr Write, up to 2 pending sSRAM write (1 queued in KI and another being done by DM-unit), and multiple (2 or more) State Update pending.*

3.7.3 aBIU Internal State

- In MC:
- In aP-bus Interface:
  - 1 x aPAddrIn (32b)
  - 1 x aPCtrlIn (8b)

    Latches aPAddr and other control signals for further reference. aPCtrIn Control signals include:
    - aPTT (5b)
    - aPTSiz (3b)
    - _aPTbst (1b)
  - 1 x a-PortAddrOut (12b)
- In AAG:
  - 1 x ApprovalSramAddr(12b)
- In SAI:
  - 1 x ConfCmdIn (20b)
  - 1 x ConfCmdOut (20b)
- In SM:

(Shared Memory Capture and Approval State)
  - 1 x ApprovalReg {Addr (32b); TT (5b) ; TSiz (3b) ; Tbst (1b) + state (2b) }
  - 1 x NotificationLocked (1b)

- In AMO:

(NES Mastered aP-bus Op)
  - 1 x TransReg {Addr (32b); TT (5b) ; TSiz (3b) ; Tbst (1b) + Gbl (1b) + CI (1b) + (WT) + valid-p (1b) }

(DMA Rx Channel)
  - 8 x DmaRxPendingCnt (16b)
  - 8 x DmaRxAck_P (1b)
- In KI:
  - 1 x KDataAC (64b)
  - 1 x KDataCA (64b)
  - 1 x KAddr (13b)

3.8 aBIU Internal Organization

The aBIU is organized into several functional blocks as described in Table 45. Figure 9 is a diagram illustrating this organization. Each block is described in greater detail below. Signal descriptions are usually preceded by a two char prefix: ii, io, xi or xo. If the first character is 'x' (external), the signals are from/to functional blocks external to the aBIU. Otherwise, it is 'i' (internal). The second char indicates whether the signal is an input ('i') or an output ('o') with respect to the functional block being described. Sometimes the second character is not specified when we are refering to a logical group of signals which include both inputs and outputs.

3.8.1 aBIU-Master Control (MC)

- External functionalities:
  - aP-bus transaction, to aBIU slave space.
    ii AiMcAccSp[0:2]
    xi _aPTs
    xo McAiBusy (aPAddrIn occupied)
- Internal functionalities:
  - generates and coordinates DM request.
  - generates and coordinates QPtr status refresh.
  - coordinates ShTx.
  - forwards and coordinates Config State Access.
  - tracks any on-going transaction. These include:
    (reflected in aBIU MSR)
    - Tx (Addr) write (up to 1 outstanding, queued in KI)
    - sSRAM write DM (up to 2 outstanding, 1 queued in KI 1 being done)
    - Config State write (up to ? outstanding, 1 queued in KI ?-1 being done)
    - QPtr State write (up to ? outstanding, 1 queued in KI ?-1 being done)
    - aBIU Mastered Op Ack (up to 1 outstanding, queued in AMO)

    (Not reflected in aBIU MSR)
    - aPAddrIn occupied.
- State in this functional block:

Functional Block	Functionality
aBIU-MC (MC)	Overall control of aBIU: runs main state machine and tracks aBIU functional block busy/avail.
aP-bus Interface (AI)	Handles aP-bus (604, MPC105) protocol, responding to and generating control signals.
	Generals control signals for a-Port.
a-Port Addr Gen (AAG)	generate aPortAddr for aP-bus transactions to aBIU slave space.
	Generate request to update CTRL Q State.
	Read aBIU Q State.
	Update aBIU Q State.
	Update aBIU Q State and re-evaluate empty bit.
	Increment aBIU VasR Q State, and set corresponding empty bit.
State Access/ImmdCmd (SAI)	State access request from aP-bus.
	State access request from CTRL.
	State update request from AAG.
	State access request to AAG.
	State access request to MC.
Shared Memory (SM)	HAL check.
	sP HAL write request.
	sP HAL read request.
	Read from SM config state.
	Update to SM config state.
	New SM bus transaction processing.
	SM approval from sP.
	SM discard captured.
aBIU Mastered Op (AMO)	Mastered Op on aP-bus.
	Dma Receive Cmd.
K-bus Interface (KI)	K-bus Write.

Table 45: Internal Organization of the aBIU.

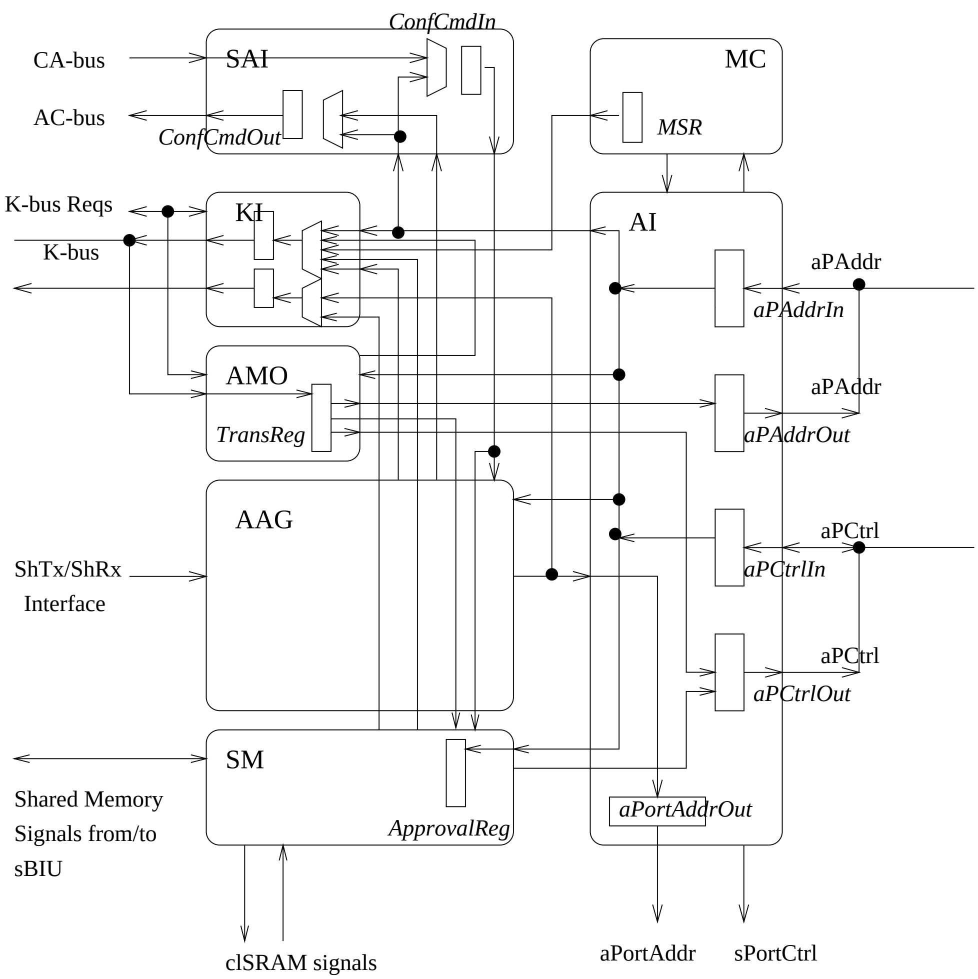

Figure 9: Internal Organization of aBIU.

   - aBIU MSR[0:4?]
   - aPAddrInOccupied (1b)
- Scheduling:
   - There is no internal scheduling to take care of here. The appropriate sub-state machine runs during an aP-bus transaction. All further action needed is also set up properly with the appropriate functional block that will perform them (or in turn arrange to perform them). Further pending actions are tracked with the aid of the aBIU MSR.

3.8.2 aP-bus Interface (AI)

- External functionalities:
   - Address decode; identifies which aBIU address space, if any, a transaction is referring to.
     xi aPAddr[0:3], aPAddr[4:6]
     io AiMcAccSp[0:2]
   - Generates control signals for a-Port.
     xo aPortR/_W[0:1], aPortOE[0:1], aPortCE[0:1]
     ii AagAiAddrRdy
     io AiAagAddrDone
   - aP-bus Mastered Transaction Request
     xi AiAmoMopReq, AmoAiAddr, AmoAiTT, AmoAiTSiz, AmoAiTbst, AmoAiWt, AmoAiCi, AmoAiGbl
     xo AiAmoMopGnt, AiAmoArtry
- Internal functionalities:
   - TT decode
     xi aPTt
     Ii AiMcAccSp[0:2]
     Io TTSupported
   - TSize decode
     xi aPTT, _aPTbst
     Ii AiMcAccSp[0:2]
     Io SizeSupported
   - Responds to and generates aP-bus controls.
     xi _aPTs, aPBg, aPBb, aPDbg, aPDbb
     ii McAiBusy
     xo aPArtry, _aPAAck(?Depends on memory controller. Check MPC105 to see what it expects.), aPTa
- State in this functional block:
   - aPAddrIn[0:31]
   - aPCtrlIn[0:31?]

3.8.3 a-Port Addr Gen (AAG)

- External functionalities:

- generate aPortAddr for aP-bus transactions to aBIU slave or snoop space.
  xi _aPTs (indicates request initiated)
  xi _aPAAck (indicates request committed)
  io AagAiAddrRdy
  io AagAiAddr
  ii AiAagAddrDone
  (— master select —)
  xi aPAddr[1:3], aPAddr[4:6,16]
  (— ShTx-like Addr Gen —)
  xi aPAddr[7], aPAddr[28]
  (— ShRx-like Addr Gen —)
  xi aPAddr[15:18], aPAddr[27:28] (Note: unlike sBIU, aBIU does not need aPAddr[19])
  ii CsPcpREp[0:3]
  (— Mastered Op Addr Gen —)
  xi KBusDataCA[59:61,30:31]
- generate request to update CTRL Q State.
  io AagSaiReq, AagSaiVal[0:7], AagSaiRegAddr[0:10?]
  ii AagSaiDone
- read from aBIU Q State.
  ii SaiAagReq, SaiAagRegAddr[0:10?], SaiAagRW
  io AagSaiDone
  xo AagKiData[0:31]
- update to aBIU Q State.
  ii SaiAagReq, SaiAagVal[0:7], SaiAagRegAddr[0:10?], SaiAagRW
  io SaiAagDone
- update to aBIU Q State requiring re-evaluation of empty bit.
  ii SaiAagReq, SaiAagVal[0:7], SaiAagRegAddr[0:10?], SaiAagRW
  io SaiAagDone
- increment aBIU VasR P-ptr, and set corresponding empty bit.
  xi AbiuVasRP-PtrInc[0:3]
  Note: we have to perform the increment in time to accept one request every 2 cycles. If not, flow-control feedback is needed.

- Internal functionalities:
  - Maintain BCP for ShRxQ and ShTxQ, incrementing them when queue is accessed, and re-evaluating associated empty bit. Also generate request to update CTRL Q State as a result.
  - Prioritized ShRx-like addr selection.
- State in this functional block:
  - ShRx-like Qstate
  - ShTx-like Qstate
  - General aPortAddrState
  - ApprovalSramAddr
  - MemQDataIn Qstate
- Scheduling:

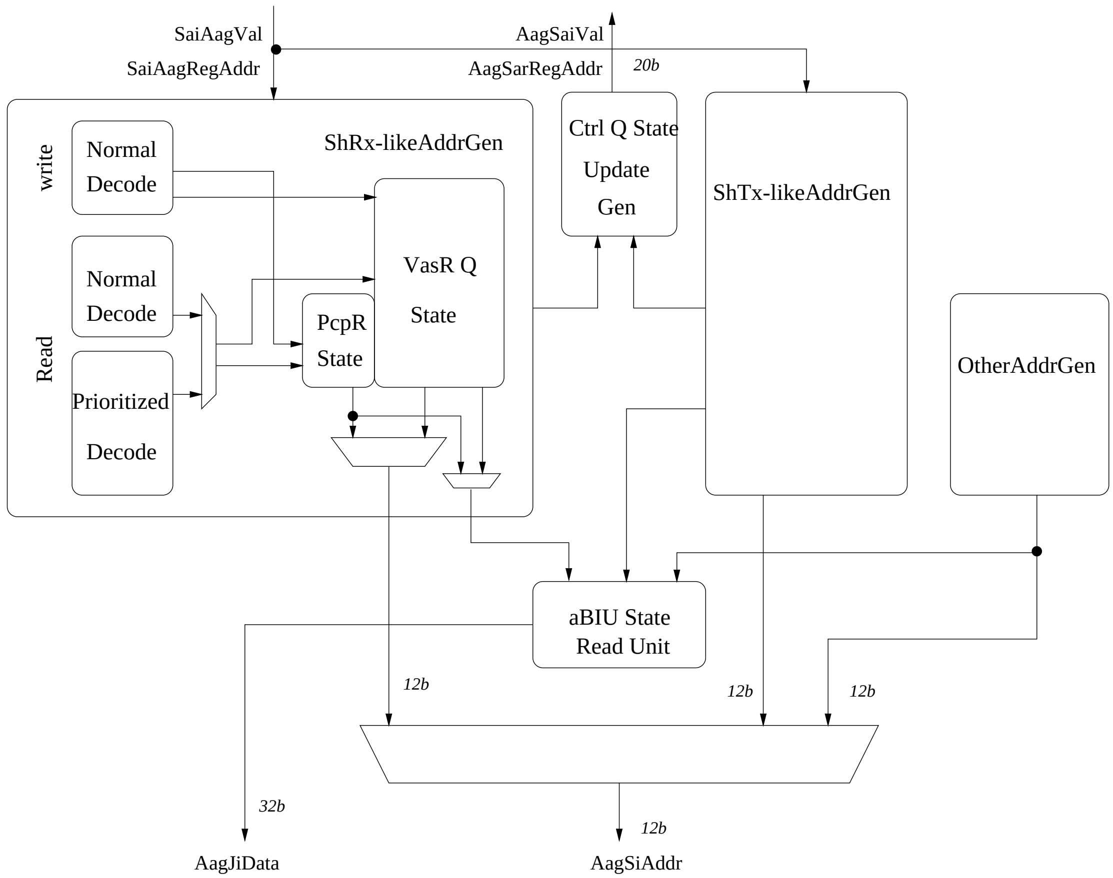

Figure 10: Organization of a-Port Addr Gen.

- Use raw _aPTs to coordinate scheduling. _aPTs asserted means next read cycle is for aP-bus request.

The a-Port Addr Gen is more complex than the other functional blocks. Figure 10 shows a possible organization. OtherAddrGen is responsible for address generating with/for sSRAM, sSRAM-tmp, qptr-base, aBIU-qconfig-tmp, MemQDataIn, ApprovalSramAddr.

3.8.4 State Access and ImmdCmd (SAI)

- External functionalities:
  - State access and ImmdCmd request from aP-bus.
    xi aPAddr[6:9,10:11,14:17,27:28], aPAddr[18:25]
    xi sPAddr[4:5], to distinguish between the 2 Config State spaces and ImmdCmd spaces.
    xo ACD interface.
  - State access request from CTRL.
    x aBIU AC-bus Interface

  - State update request from AAG.
    i AagSai interface (Aag to Sai req)
    ii AagSaiReq, AagSaiVal[0:7], AagSaiRegAddr[0:10?]
    io AagSaiDone
  - State access request to AAG.
    i SaiAag interface (Sai to Aag req)
    io SaiAagReq, SaiAagVal[0:7], SaiAagRegAddr[0:10?], SaiAagRW
    ii SaiAagDone
  - State access request to MC.
    i SaiMc interface (Sai to Mc req)
    io SaiMcReq, SaiMcVal[0:7], SaiMcRegAddr[0:10?], SaiMcRW
    ii SaiMcDone
- State in this functional block:
  - ConfCmdIn[0:19]
  - ConfCmdOut[0:19]
- Scheduling:
  - all write in order.
  - all read in order
  - but read can overtake write.
- Suggestion:
  - Bypass capabilities will improve performance.

3.8.5 Shared Memory (SM)

comment: *We should consider merging AMO, AI and SM because of the wide path between them.*

- External functionalities:
  - HAL check
    xi aPAddr[7:26], aPTT, clsData,
    xo clsRSel, clsCS, clsaPEn
    io SmHalApprovOk
  - sP HAL write request
    xi ClsWriteReq
    xo clssPToClsOutEn, clssPAddrOutEn, ClsWriteDone
    Note: This has to be performed to the clSRAM as soon as it is next available. It is only delayed temporarily if this request comes while the clSRAM is used by the aP side. No subsequent new request from the aP side is allowed to use the clSRAM before the write is performed.
  - sP HAL read request (?? Not sure we are supporting this ??
    ii ClsReadReq
    xo clsClsTosPInEn, clssPAddrOutEn, ClsReadDone
  - Read SM config state
    ii SaiSmReq, SaiSmVal[0:7], SaiSmRegAddr[0:10?], SaiSmRW
    io SmSaiDone
    xo SmKiData[0:31]

- Update SM config state
  ii SaiSmReq, SaiSmVal[0:7], SaiSmRegAddr[0:10?], SaiSmRW
  io SmSaiDone
- New SM bus transaction processing
  xi aPTT, aPAddr[0:31], aPTSiz[0:2], aPTbst
  ii SmHalApprovOk
  xo SmAiArtry
- SM Approval from sP
  xi AcdApproval, AcdApprAddr[0:12]
- SM discard captured
  xi ClrApprovalReg
- Increment MemQIn consumer pointer
  xi MemQInCons
  Note: we have to perform the increment in time to accept one request every 2 cycles. If not, flow-control feedback is needed.

- State in this functional block:
  - 1 x ApprovalReg {Addr (32b); TT (5b) ; TSiz (3b) ; Tbst (1b) + valid-p (1b) }
  - 1 x ApprovalGiven (1b)
  - 3 x AddrIndepResp (2b)
  - 12 x sSsAddrDepResp (2b) (For sP Serviced Space)
  - 12 x aSsAddrDepResp (2b) (For aBIU Snooped Space)
  - 12 x aSsHalCode (4b) (For aBIU Snooped Space)

3.8.6 aBIU Mastered Op (AMO)

comment: *We should consider merging AMO, AI and SM because of the wide path between them.*

- External functionalities:
  - Mastered Op on aP-bus
    xi aPBusReq, KiAmoData[0:63]
    xi AiAmoMopGnt, AiAmoArtry
    xo AmoAiMopReq, AmoAiAddr, AmoAiTT, AmoAiTSiz, AmoAiTbst, AmoAiWt, AmoAiCi, AmoAiGbl
    xo AmoMcMemQInAckReq
    xi AmoMcMemQInAckGnt
    xo aPBusDone
  - Dma Receive Cmd
    xi aPBusReq, KiAmoData[0:63] (only a subset used)
    xo AmoMcMemQInAckReq
    xi AmoMcMemQInAckGnt
    xo aPBusDone
  - Short-circuit optimization check
    xi aPAddr, aPTT, aPTSiz, aPTbst
    io ShortCircuitHit
- State in this functional block:

  - 1 x TransReg {Addr (32b); TT (5b) ; TSiz (3b) ; Tbst (1b) + Gbl (1b) + CI (1b) + (WT) + valid-p (1b) }
  - 8 x DmaRxPendingCnt (7b)
  - 8 x DmaRxAck_P (1b)
- Suggestion:
  - Bypass (TransReg) capabilities will improve performance.

3.8.7 K-bus Interface (KI)

- External functionalities:
  - K-bus Write.

    ii AagKiAddr[0:13], AagKiData[0:31], AiKiData[0:31], AagKiReqType[0:2], AiKiReqType[0:2].
    xo KBusDataOut[0:32], KBusAddr[0:12].
    xi KBusDataIn[0:63], KBusAddr[0:12].
    xo AbiuShReq, AbiuMqReq, AbiuDmReq.
    xi AbiuShGnt, AbiuMqGnt, AbiuDmGnt.
- State in this functional block:
  - KAddr[0:13], KData[0:63]
- Suggestion:
  - Bypass capabilities will improve performance.

3.9 aBIU Timing Constraints

comment: *We will fill this in as we discover needs.*

Three categories of timing constraints:

- 60X bus specific timing constraints.
  - aPAack is generated by the Union Chip (aP-bus Controller) either 1 or 2 cycles (programmable) after aPTs is asserted. aPArtry and aPShd will have to follow the cycle after aPAack. HAL, and Approval logic will have to be fast enough
  - When the aBIU is an sP-bus transaction master, it must be able to accept/supply data as fast as the slave can take, starting immediately after qualified data bus grant.
- Ordering between different operations accepted by aBIU. Some operations need to be processed in FIFO order.

3.10 aBIU Miscellaneous Issues

3.10.1 Rejected Proposed aBIU Functionalities

1. It was proposed that we support SYNC operation, with the semantics that the following operations, which may take some time, have to be completed before the SYNC is allowed to complete:
   (a) MsgQPtr Control update
   (b) Config space update

(c) ImmdCmd

(d) ShTx write of addr bits

An alternative/additional suggestion was that we support lower cost ways of detecting completion of common ops, e.g. Cls write.

We have chosen a design where there is a real time bound (possibly large but bounded in some cases). For certain actions, because the design allows reading most of the state in CTRL and sBIU, it is possible through that mechanism to determine if certain actions have been performed, *e.g.* to determine if a ShTx write has finished, check the queue's P-ptr in CTRL to see if it has been updated to the expected value which can be read from the sBIU's copy of the same P-ptr.

3.10.2 No Separate Global Shared Memory and sP Serviced Space

Unlike previous version of the specification, we are no longer having two separate regions for Global Shared Memory and sP Serviced Space. Both space are treated the same way by software in the previous design. Having them separate was to make provision for a larger Global Shared Memory space. Given our emphasis on Local Global Memory space for implementing shared memory, and the difficulty in finding space for the sP Serviced Space, we have collapsed them into one, and call it the sP Serviced Space. What used to be Global Shared Memory will be part of the sP Serviced Space.

sPAddr bit	CtrlAddr bit	xBIUAddr bit
6	X (value 0)	X (value 1)
7	2	2
8	3	3
9	4	4
10	0	0
11	1	1
14	5	5
15	8	8
16	9	9
17	10	10
27	6	6
28	7	7

Table 46: Correspondence between sPAddr and CtrlAddr and xBIUAddr for NESCtrl state access.

4 NESCtrl Chip State Address Map

Accessible NESCtrl Chip state is addressed internally using CtrlAddr for state in CTRL, sBIUAddr for state in sBIU, and aBIUAddr for state in aBIU. Externally, they are accessed from the aP-bus and sP-bus Cmd Spaces (See Sections 3.2.4 and 2.2.3), and to a lesser degree, their QPtr Spaces (See Sections 3.2.1 and 2.2.1). Table 46 gives the correspondence between the bits in sPAddr and these other addresses (xBIUAddr represents both aBIUAddr and sBIUAddr).

4.1 CtrlAddr

Width (bits)	Bit field	Content	Description
2	[0:1]	`00`	End-point state (EPRAM)
		`01`	Queue state (QRAM)
		`10`	System register (SysReg)
		`11`	** Reserved ** (ImmdCmd)

Table 47: CtrlAddr Primary Opcode encoding.

CtrlAddr is the address used by CTRL for referring to state elements. When forwarding requests to CTRL, the sBIU and aBIU also use this addressing. State in CTRL is separated into 3 categories: EPRAM, QRAM, and SysReg. In addressing CTRL state, CtrlAddr[0:1] is the primary opcode which distinguishes between these 3 categories, using the encoding shown in Table 47.

4.1.1 EPRAM State

Access to EPRAM space uses the encoding shown in Table 48. For reads, the 8B quantity is obtained by ignoring CtrlAddr[5:7]. The following gives a more detailed description of the fields in Table 48, separating them out into the two different types of ComGp's. The bit fields are numbered in a Big Endian fashion, assuming the 8B register that one sees by ignoring CtrlAddr[5:7]. In the following description, "++" indicates concatenation of bit-fields. NES SRAM Address is a 13b

Width (bits)	Bit field	Content	Description
2	[0:1]	`00`	==== Fixed Field ====
3	[2:4]	`000`	CompGp Pas/Vas-0
		`001`	CompGp Pas/Vas-1
		`010`	CompGp Pas/Vas-2
		`011`	CompGp Pas/Vas-3
		`100`	CompGp Pcp-0
		`101`	CompGp Pcp-1
		`110`	CompGp Pcp-2
		`111`	CompGp Pcp-3
3	[5:7]	`000`	TransBase in Bb[1:5], TransAND[0] in Bb[7]
		`001`	TransAND[1:7] in Bb[1:7]
		`010`	For Pas/Vas CompGp TagOnRxL P-ptr in Bb[1:7] For Pcp CompGp As programmed by software.
		`011`	For Pas/Vas CompGp TagOnRxH P-ptr in Bb[1:7] For Pcp CompGp As programmed by software.
		`100`	TxL C-ptr in Bb[1:7]
		`101`	TxH C-ptr in Bb[1:7]
		`110`	RxL P-ptr in Bb[1:7]
		`111`	RxH P-ptr in Bb[1:7]
3	[8:10]	`X`	** unused **

Table 48: Addressing of CTRL's EPRAM state. The address bits refer to the internal CtrlAddr that aBIU and sBIU use when interacting with CTRL.

quantity, with bit 0 selecting between aSRAM(#b1) and sSRAM(#b0), and the other 12 bits an 8B aligned address into the SRAM concerned.

Pcp ComGp:

Size	Field	Name	Description
Byte 0			
1	[0]	** Unused **	
5	[1:5]	TransBase	This is used together with TransAND, and the logical destination (LDest[0:7]) address to compute the NES SRAM address of the Destination Translation Table entry used to form the header of a out-going message. NES SRAM Addr[0:12] = TransBase[0:4] ++ (LDest[0:7] AND TransAND[0:7]);
1	[6]	Software Defined	As written by software; not used by Ctrl or the NES
1	[7]	TransAND[0]	This is part of the TransAND value, used together with TransAND, and the logical destination address to compute the NES SRAM address of the Destination Translation Table entry used to form the header of a out-going message. See description for TransBase above for how NES SRAM Addr is computed.
Byte 1			
1	[8]	** Unused **	
7	[9:15]	TransAND[1:7]	This is part of the TransAND value, used together with TransAND, and the logical destination address to compute the NES SRAM address of the Destination Translation Table entry used to form the header of a out-going message. See description for TransBase above for how NES SRAM Addr is computed.
Byte 2			
1	[16]	** Unused **	
3	[17:19]	Software Defined	As written by software; not used by Ctrl or the NES
3	[20:22]	ComGp Number	(?? Written by CTRL??)
1	[23]	Software Defined	As written by software; not used by Ctrl or the NES; Probably used for storing TT[0].
Byte 3			
4	[24:27]	Software Defined	As written by software; not used by Ctrl or the NES; Probably used for storing TT[1:4].
4	[28:31]	Software Defined	As written by software; not used by Ctrl or the NES

Byte 4			
1	[32]	** Unused **	
7	[33:39]	TxL Consumer Pointer	The Consumer ptr's actual SRAM address is computed using Base defined in the TxL queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR TxLConsumerPtr[0:4]) ++ TxLConsumerPtr[5:6] ++ #b00;
Byte 5			
1	[40]	** Unused **	
7	[41:47]	TxH Consumer Pointer	The Consumer ptr's actual SRAM address is computed using Base defined in the TxH queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR TxHConsumerPtr[0:4]) ++ TxHConsumerPtr[5:6] ++ #b00;
Byte 6			
1	[48]	** Unused **	
7	[49:55]	RxL Producer Pointer	The Producer ptr's actual SRAM address is computed using Base defined in the RxL queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR RxLProducerPtr[0:4]) ++ RxLProducerPtr[5:6] ++ #b00;
Byte 7			
1	[56]	** Unused **	
7	[57:63]	RxH Producer Pointer	The Producer ptr's actual SRAM address is computed using Base defined in the RxH queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR RxHProducerPtr[0:4]) ++ RxHProducerPtr[5:6] ++ #b00;

Pas/Vas (Short) ComGp:

Size	Field	Name	Description
Byte 0			
1	[0]	** Unused **	
5	[1:5]	TransBase[0:4]	This is used together with TransAND, and the logical destination (LDest[0:7]) address to compute the NES SRAM address of the Destination Translation Table entry used to form the header of a out-going message. NES SRAM Addr[0:12] = TransBase[0:4] ++ (LDest[0:7] AND TransAND[0:7])
1	[6]	Software Defined	As written by software; not used by Ctrl or the NES
1	[7]	TransAND[0]	This is part of the TransAND value, used together with TransAND, and the logical destination address to compute the NES SRAM address of the Destination Translation Table entry used to form the header of a out-going message. See description for TransBase above for how NES SRAM Addr is computed.
Byte 1			
1	[8]	** Unused **	
7	[9:15]	TransAND[1:7]	This is part of the TransAND value, used together with TransAND, and the logical destination address to compute the NES SRAM address of the Destination Translation Table entry used to form the header of a out-going message. See description for TransBase above for how NES SRAM Addr is computed.
Byte 2			
1	[16]	** Unused **	
7	[17:23]	TagOnRxL Producer Pointer	The Producer ptr's actual SRAM address is computed using TagOnBase defined in the RxL queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = TagOnBase[0:3] ++ (TagOnBase[4:8] OR TagOnRxLProducerPtr[0:4]) ++ TagOnRxLProducerPtr[5:6] ++ #b00;
Byte 3			
1	[24]	** Unused **	
7	[25:31]	TagOnRxH Producer Pointer	The Producer ptr's actual SRAM address is computed using TagOnBase defined in the RxH queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = TagOnBase[0:3] ++ (TagOnBase[4:8] OR TagOnRxHProducerPtr[0:4]) ++ TagOnRxHProducerPtr[5:6];

Byte 4			
1	[32]	** Unused **	
7	[33:39]	TxL Consumer Pointer	The Consumer ptr's actual SRAM address is computed using Base defined in the TxL queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:5] ++ (Base[6:8] OR TxLConsumerPtr[0:2]) ++ TxLConsumerPtr[3:6];
Byte 5			
1	[40]	** Unused **	
7	[41:47]	TxH Consumer Pointer	The Consumer ptr's actual SRAM address is computed using Base defined in the TxH queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:5] ++ (Base[6:8] OR TxHConsumerPtr[0:2]) ++ TxHConsumerPtr[3:6];
Byte 6			
1	[48]	** Unused **	
7	[49:55]	RxL Producer Pointer	The Producer ptr's actual SRAM address is computed using Base defined in the RxL queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:5] ++ (Base[6:8] OR RxLProducerPtr[0:2]) ++ RxLProducerPtr[3:6];
Byte 7			
1	[56]	** Unused **	
7	[57:63]	RxH Producer Pointer	The Producer ptr's actual SRAM address is computed using Base defined in the RxH queue's QRAM state (see Section 4.1.2): NES SRAM Addr[0:12] = Base[0:5] ++ (Base[6:8] OR RxHProducerPtr[0:2]) ++ RxHProducerPtr[3:6];

4.1.2 QRAM State

Access to QRAM space uses the encoding shown in Table 49. For reads, the 8B quantity is obtained by ignoring CtrlAddr[8:10]. The following gives a more detailed description of the fields in Table 49, separating them out into the four different types of queues. The bit fields are numbered in a Big Endian fashion, assuming the 8B register that one sees by ignoring CtrlAddr[8:10]. In the following description, "++" indicates concatenation of bit-fields. NES SRAM Address is a 13b quantity, with bit 0 selecting between aSRAM(#b1) and sSRAM(#b0), and the other 12 bits an 8B aligned address into the SRAM concerned.

Pcp Tx Queue:

Width (bits)	Bit field	Content	Description
2	[0:1]	01	==== Fixed Field ====
3	[2:4]	000	CompGp Pas/Vas-0
		001	CompGp Pas/Vas-1
		010	CompGp Pas/Vas-2
		011	CompGp Pas/Vas-3
		100	CompGp Pcp-0
		101	CompGp Pcp-1
		110	CompGp Pcp-2
		111	CompGp Pcp-3
1	[5]	X	** Unused **
1	[6]	0	Tx
		1	Rx
1	[7]	0	L priority
		1	H priority
3	[8:10]	000	In Bb[1:7] C-ptr for Rx, P-ptr for Tx.
		001	In Bb[1:7] R-ptr for Pcp (both Tx and Rx), TagOnC-ptr (tag-on data) for Vas Rx.
		010	Base[0:6] in Bb[1:7]
		011	Base[7:8] in Bb[1:2]; Bound[0:4] in Bb[3:7].
		100	For Tx: Bb[1]: Allow Tag-on. Bb[2]: Allow UpRoute. Bb[3]: Allow interrupt. Bb[4]: Arctic priority. Bb[5:7]: RQID[6:8]. For Rx: Bb[1]: Allow interrupt. Bb[2]: Always interrupt. Bb[3]: (?) which processor to interrupt. Bb[4]: Mirror. Bb[5]: Lazy Ack. Bb[6]: Enable Overflow.
		101	For Tx: Bb[1]: Allow raw.
		110	(Not used for Pcp RxQ). For Vas Rx: TagOnBase[0:6] in Bb[1:7]. For all Tx: TagOnBase[0:5] in Bb[2:7].
		111	(Not used for Pcp RxQ). For Vas Rx: TagOnBase[7:8] in Bb[1:2]; TagOnBound[0:4] in Bb[3:7] For all Tx: TagOnMask[0:3] in Bb[4:7]

Table 49: Addressing of CTRL's QRAM state. The address bits refer to the internal CtrlAddr that aBIU and sBIU use when interacting with CTRL.

Size	Field	Name	Description
Byte 0			
1	[0]	** Unused **	
7	[1:7]	Producer Ptr	The Producer ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR ProducerPtr[0:4]) ++ ProducerPtr[5:6] ++ #b00;
Byte 1			
1	[8]	** Unused **	
7	[9:15]	Reclaim Ptr	When NES Reclaim is enabled, application code on aP does not update the Producer Ptr directly. Instead it updates the Reclaim Ptr. This causes the NES to initiate aP-bus translaction to extract the processor/L2 cache copies of message buffer in the region between the current Producer Ptr and the Reclaim Ptr, after which the Producer Ptr is then updated by the NES. The Reclaim ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR ReclaimPtr[0:4]) ++ ReclaimPtr[5:6] ++ #b00;
Byte 2			
1	[16]	** Unused **	
7	[17:23]	Base[0:6]	This together with Base[7:8] forms the Base address of the queue's msg buffer space. See description of the pointers to see how Base address is used.
Byte 3			
1	[24]	** Unused **	
2	[25:26]	Base[7:8]	This together with Base[0:6] forms the Base address of the queue's msg buffer space. See description of the pointers to see how Base address is used.

5	[27:31]	Bound	Bound is used to delimit the extent of the queue's msg buffer space. When computing a new pointer,
			if (NewPtr[0:4] == Bound),
			set NewPtr[0:6] = #b0000000;

Byte 4			
1	[32]	** Unused **	
1	[33]	Allow TagOn	Indicates whether TagOn is allowed from this queue.
1	[34]	Allow Uproute	Indicates whether message specified Uproute is allowed for packets launched from this queue.
1	[35]	Allow Interrupt	Indicates whether message specified Interrupt-on-message arrival is allowed for packets launched from this queue.
1	[36]	Arctic Priority	The Arctic Priority level to use for packets launched from this queue.
1	[37:39]	RQID[6:8]	This specifies part of the RQID. The rest of the RQID comes from the Destination Translation Table. RQID[6] has to be set to 1 for a Pcp queue. This is not used for packets with Fully Specified headers (also referred to as Raw header).
Byte 5			
1	[40]	** Unused **	
1	[41]	Allow Raw	Whether Fully Specified headers (also referred to as Raw header) are allowed for packets from this queue.
6	[42:47]	** Unused **	
Byte 6			
2	[48:49]	** Unused **	
6	[50:55]	TagOn Base[0:5]	When TagOn is allowed, this with TagOn Mask and the TagOn address offset specified in the message are used to compute the starting NES SRAM address to obtain the tagged-on data from. (TagOnOffset is a 9b field taken from bits[5:13] of the message header. See Section 5.2.) NES SRAM Addr[0] = (TagOnOffset[0] AND TagOnMask[0]) OR TagOnBase[0]; NES SRAM Addr[1:2] = TagOnBase[1:2]; NES SRAM Addr[3:5] = (TagOnOffset[1:3] AND TagOnMask[1:3]) OR TagOnBase[3:5]; NES SRAM Addr[6:10] = TagOnOffset[4:8]; NES SRAM Addr[11:12] = #b00;
Byte 7			
4	[56:59]	** Unused **	
4	[60:63]	TagOn Mask[0:3]	When TagOn is allowed, this with TagOn Base and the TagOn address offset specified in the message are used to compute the NES SRAM address to obtain the tagged-on data from. See the description above, for TagOn Base for information on how TagOn data's NES SRAM Addr is generated using this field.

Pcp Rx Queue:

Size	Field	Name	Description
Byte 0			
1	[0]	** Unused **	
7	[1:7]	Consumer Ptr	The Consumer ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = Base[0:3] ++ (Base[5:8] OR ConsumerPtr[0:4]) ++ ConsumerPtr[5:6] ++ #b00;
Byte 1			
1	[8]	** Unused **	
7	[9:15]	Reclaim Ptr	When NES Reclaim is enabled, application code on aP does not update the Consumer Ptr directly. Instead it updates the Reclaim Ptr. This causes the NES to initiate aP-bus translaction to flush the processor/L2 cache copies of message buffer in the region between the current Producer Ptr and the Reclaim Ptr, after which the Producer Ptr is then updated by the NES. The Reclaim ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = Base[0:3] ++ (Base[4:8] OR ReclaimPtr[0:4]) ++ ReclaimPtr[5:6] ++ #b00;
Byte 2			
1	[16]	** Unused **	
7	[17:23]	Base[0:6]	Together with Base[7:8], this forms the Base address of the queue's msg buffer space. See description of the pointers to see how Base address is used.
Byte 3			
1	[24]	** Unused **	
2	[25:26]	Base[7:8]	Together with Base[0:6], this forms the Base address of the queue's msg buffer space. See description of the pointers to see how Base address is used.

5	[27:31]	Bound	Bound is used to delimit the extent of the queue's msg buffer space. When computing a new pointer,
			if (NewPtr[0:4] == Bound),
			set NewPtr[0:6] = #b0000000;

Byte 4			
1	[32]	** Unused **	
1	[33]	Allow Interrupt	Indicates whether packets arrival is allowed to raise an interrupt. Packets arrival may want to raise an interrupt because its MsgOp specifies so, or that the RxQ is configured to Always Interrupt on message arrival.
1	[34]	Always Interrupt	Indicates whether packets arrival always raises an interrupt.
1	[35]	Interrupt Processor	The processor to interrupt. sP is 0, aP is 1.
1	[36]	Mirror	Whether the receive queue is mirrored. If so, any incoming packet is written to both SRAMs, at the same offset. The NES SRAM Addr[0] is basically ignored.
1	[37]	Lazy Acknowledge	Whether flow-control acknowledgement back to Arctic is done in a lazy fashion (1 for lazy). If this acknowledgement is *not* lazy, the NES generates an acknowledgement to Arctic once the packet is moved from the RxFIFO into the message queue in NES SRAM.
1	[38]	en-overflow	Indicates whether packets that arrive to find this RxQ full are placed into the Overflow queue. This feature is only available for queues with buffer space in aSRAM and if the queue is not Mirror-ed (see Mirror, bit[36] above).
1	[39]	** Unused **	
Byte 5			
8	[40:47]	** Unused **	
Byte 6			
8	[48:55]	** Unused **	
Byte 7			
8	[56:63]	** Unused **	

Pas (Short) Tx Queue:

Size	Field	Name	Description
Byte 0			
1	[0]	** Unused **	
7	[1:7]	Producer Ptr	The Producer ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = Base[0:5] ++ (Base[6:8] OR ProducerPtr[0:2]) ++ ProducerPtr[3:6];
Byte 1			
7	[8:15]	** Unused **	

Byte 2			
1	[16]	** Unused **	
7	[17:23]	Base[0:6]	This together with Base[7:8] forms the Base address of the queue's msg buffer space. See description of the pointers to see how Base address is used.
Byte 3			
1	[24]	** Unused **	
2	[25:26]	Base[7:8]	This together with Base[0:6] forms the Base address of the queue's msg buffer space. See description of the pointers to see how Base address is used.

5	[27:31]	Bound	Bound is used to delimit the extent of the queue's msg buffer space. When computing a new pointer,
			if (NewPtr[0:4] == Bound),
			set NewPtr[0:6] = #b0000000;

Byte 4			
1	[32]	** Unused **	
1	[33]	Allow TagOn	Indicates whether TagOn is allowed from this queue.
1	[34]	Allow Uproute	Indicates whether message specified Uproute is allowed for packets launched from this queue.
1	[35]	Allow Interrupt	Indicates whether message specified Interrupt-on-message arrival is allowed for packets launched from this queue.
1	[36]	Arctic Priority	The Arctic Priority level to use for packets launched from this queue.
1	[37:39]	RQID[6:8]	This specifies part of the RQID. RQID[6] has to be set to 0 for a Pas (Short) Rx queue. The rest of the RQID comes from the Destination Translation Table. This is not used for packets with Fully Specified headers (also referred to as Raw header).
Byte 5			
1	[40]	** Unused **	
1	[41]	Allow Raw	Whether Fully Specified headers (also referred to as Raw header) are allowed for packets from this queue.
6	[42:47]	** Unused **	

Byte 6			
2	[48:49]	** Unused **	
6	[50:55]	TagOn Base	When TagOn is allowed, this with TagOn Mask and the TagOn address offset specified in the message are used to compute the starting NES SRAM address to obtain the tagged-on data from. (TagOnOffset is a 9b field taken from bits[5:13] of the message header. See Section 5.2.) NES SRAM Addr[0] = (TagOnOffset[0] AND TagOnMask[0]) OR TagOnBase[0]; NES SRAM Addr[1:2] = TagOnBase[1:2]; NES SRAM Addr[3:5] = (TagOnOffset[1:3] AND TagOnMask[1:3]) OR TagOnBase[3:5]; NES SRAM Addr[6:10] = TagOnOffset[4:8]; NES SRAM Addr[11:12] = #b00;
Byte 7			
4	[56:59]	** Unused **	
4	[60:63]	TagOn Mask	When TagOn is allowed, this with TagOn Base and the TagOn address offset specified in the message are used to compute the NES SRAM address to obtain the tagged-on data from. (TagOnOffset is a 9b field taken from bits[5:13] of the message header. See Section 5.2.) NES SRAM Addr[0] = (TagOnOffset[0] AND TagOnMask[0]) OR TagOnBase[0]; NES SRAM Addr[1:2] = TagOnBase[1:2]; NES SRAM Addr[3:5] = (TagOnOffset[1:3] AND TagOnMask[1:3]) OR TagOnBase[3:5]; NES SRAM Addr[6:10] = TagOnOffset[4:8]; NES SRAM Addr[11:12] = #b00;

Vas (Short) Rx Queue: A Short Rx queue actually maintains two queues: (i) a queue for the non tag-on (or short) part of the message; (ii) a queue for the tag-on data.

Size	Field	Name	Description
Byte 0			
1	[0]	** Unused **	
7	[1:7]	Consumer Ptr	The Consumer ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = Base[0:5] ++ (Base[6:8] OR ConsumerPtr[0:2]) ++ ConsumerPtr[3:6];

Byte 1			
1	[8]	** Unused **	
7	[9:15]	TagOn Consumer Ptr	The TagOnConsumer ptr's actual SRAM address is computed using Base: NES SRAM Addr[0:12] = TagOnBase[0:3] ++ (TagOnBase[4:8] OR TagOnConsumerPtr[0:4]) ++ TagOnConsumerPtr[5:6] ++ #b00;
Byte 2			
1	[16]	** Unused **	
7	[17:23]	Base[0:6]	This together with Base[7:8] forms the Base address of the queue's (short) msg buffer space. See description of the Consumer pointer to see how Base address is used.
Byte 3			
1	[24]	** Unused **	
2	[25:26]	Base[7:8]	This together with Base[0:6] forms the Base address of the queue's (short) msg buffer space. See description of the Consumer pointer to see how Base address is used.

5	[27:31]	Bound	Bound is used to delimit the extent of the queue's (short) msg buffer space. When computing a new pointer,
			if (NewPtr[0:4] == Bound),
			set NewPtr[0:6] = #b0000000;

Byte 4			
1	[32]	** Unused **	
1	[33]	Allow Interrupt	Indicates whether packets arrival is allowed to raise an interrupt. Packets arrival may want to raise an interrupt because its MsgOp specifies so, or that the RxQ is configured to Always Interrupt on message arrival.
1	[34]	Always Interrupt	Indicates whether packets arrival always raises an interrupt.
1	[35]	Interrupt Processor	The processor to interrupt. sP is 0, aP is 1.
1	[36]	Mirror	Whether the receive queue is mirrored. If so, any incoming packet is written to both SRAMs, at the same offset. The NES SRAM Addr[0] is basically ignored.
1	[37]	Lazy Acknowledge	Whether flow-control acknowledgement back to Arctic is done in a lazy fashion (1 for lazy). If this acknowledgement is *not* lazy, the NES generates an acknowledgement to Arctic once the packet is moved from the RxFIFO into the message queue in NES SRAM.
1	[38]	en-overflow	Indicates whether packets that arrive to find this RxQ full are placed into the Overflow queue. This feature is only available for queues with buffer space in aSRAM and if the queue is not Mirror-ed (see Mirror, bit[36] above).
1	[39]	** Unused **	

Byte 5			
8	[40:47]	** Unused **	
Byte 6			
1	[48]	** Unused **	
7	[49:55]	TagOn Base[0:6]	This together with TagOnBase[7:8] forms the TagOnBase address of the queue's TagOn data buffer space. See description of the TagOnConsumer pointer to see how TagOnBase address is used.
Byte 7			
1	[56]	** Unused **	
2	[57:58]	TagOn Base[7:8]	This together with TagOnBase[0:6] forms the TagOnBase address of the queue's TagOn data buffer space. See description of the TagOnConsumer pointer to see how TagOnBase address is used.

5	[59:63]	TagOn Bound[0:4]	Bound is used to delimit the extent of the queue's (short) msg buffer space. When computing a new pointer,
			if (NewPtr[0:4] == TagOnBound[0:4]),
			set NewPtr[0:6] = #b0000000;

4.1.3 Ctrl SysReg State

Width (bits)	Bit field	Content	Description
2	[0:1]	`10`	==== Fixed Field ====
5	[2:6]	`X`	SysReg Number
4	[7:10]	`X`	Offset into SysReg (Currently 10 is not used)

Table 50: Addressing of CTRL's SysReg state. The address bits refer to the internal CtrlAddr that aBIU and sBIU use when interacting with CTRL.

Access to SysReg space uses the encoding shown in Table 50. CtrlAddr[7:10] is ignored when reading SysReg. Details of the registers are described below, including how to read and update the registers, and the initial values upon reset.

SysReg No.:	0
SysReg Name:	TagOnLengthCL
Usage:	1b value used to indicate size of Tagon; #b1 indicates 2.5 cache-lines (80B); #b0 indicates 1.5 cache-lines (48B)
Read details:	Value is returned in bit 63.
Update details:	Value is written when CtrlAddr[7:9] is 0; data is taken from DataIn[6].
Reset Value:	Undefined.
Comments:	

SysReg No.:	1
SysReg Name:	ErrorReg
Usage:	This register is used to record any errors detected by CTRL.
Read details:	Status[59] records ErrorEPRAMWrite; (updated a part of EPRAM that is not used) Status[60] records ErrorQRAMWrite; (updated a part of QRAM that is not used) Status[61] records ErrorSystemRegRead; (attempted to read an invalid SysReeg) Status[62] records ErrorSystemRegWrite; (attempted to write an invalid SysReeg) Status[63] records EELongTxError; (EELong detected an error in a message specification, *e.g.* an invalid message length was specified.)
Update details:	regardless of the value of CtrlAddr[7:9], an update to this register clears it. This is done in a way which latches in any errors raised at the cycle of clear.
Reset Value:	Cleared (all 0) after reset.
Comments:	May change.

SysReg No.:	2
SysReg Name:	TxQEmpty
Usage:	Indicates which TxQs are empty. This is used internally by the scheduler in CTRL to track which TxQs have messages to send.
Read details:	Status[48:63]; 16 bit value addressed using (comgrp, priority)
Update details:	This register should not be updated. Any attempt to do so leads to undefined behaviour.
Reset Value:	Cleared (all 1) upon reset.
Comments:	

SysReg No.:	3
SysReg Name:	TxQPriorityVector
Usage:	This register encodes the send priority for the TxQ's. The 64b register is divided into 4b fields, each of which contains the number of a TxQ. Field[0:3] has the TxQ number of the highest priority TxQ, Field[4:7] has the TxQ number of the next priority TxQ, and so on.
Read details:	A Read of this register returns Status[0:63], with queue number for top priority TxQ in [0:3], the queue number with the next highest priority in [4:7] and so on.
Update details:	Update is performed to 4b of the register using value in DataIn[3:6], and CtrlAddr[7:10] to identify the specific field.
Reset Value:	Undefined.
Comments:	Design not changing unless significant timing problems, in which case priorities will become static.

SysReg No.:	4
SysReg Name:	TxQEnable
Usage:	16b of this register (in [48:63]) tracks/indicates which TxQ's are enabled. Messages are only transmitted from TxQ's that are enabled. Software on the aP or sP can shut down a queue by writing a 0 into the bit representating the queue concerned. A TxQ is not shut down immediately when its enable bit is cleared. Any transmission in progress will complete correctly, before the queue is shut down. Currently, aP/sP code has to wait ?? bclks before it can assume that the queue is shut down.
Read details:	A Read returns 8B where Status[48:63] contains the actual enabled status.
Update details:	Update uses CtrlAddr[7:10] to indicate which TxQ's enable status is being written. The written value comes from DataIn[6].
Reset Value:	Cleared (all 0) upon reset.
Comments:	

SysReg No.:	5
SysReg Name:	RxShortTags
Usage:	This register keeps the Short RxQ's "cache tags", used to track the RQID of the currently resident Short RxQ's. There are 8 entries, each with a 7b tag value corresponding to RQID[0:6]. Each entry has a fixed correspond to one of the resident RxQ's. RQID[0] has to be #b0 for any of the tags to match. A value of #b1 in RQID[0] indicates that the message should go into a Med RxQ, and can only match entries in SysReg 6, RxMedTags. The RxQ's are essentially divided into two sets, RxQ 0-3 (sP side), and RxQ 4-7 (aP side). The former can be used onlywhen RQID[1] is #b0, and the latter, only when RQID[1] is #b1. RxQ-0 corresponds to Pas/Vas-0 L priority; RxQ-1 corresponds to Pas/Vas-0 H priority; RxQ-2 corresponds to Pas/Vas-1 L priority; RxQ-3 corresponds to Pas/Vas-1 H priority; and so on.
Read details:	Upon a Read, the 8B value contains the tags in Status[8:63]. Tag-0 is in Status[8:14], Tag-1 is in Status[15:21] and so on.
Update details:	Update is achieved using CtrlAddr[8:10] to indicate the Tag field being updated, with value coming form DataIn[0:6].
Reset Value:	Undefined upon reset
Comments:	

SysReg No.:	6
SysReg Name:	RxMedTags
Usage:	This register is similar to RxShortTags. It keeps the Medium RxQ's "cache tags" for tracking the RQID of the currently resident Medium RxQ's. There are 8 entries, each with a 7b tag value corresponding to RQID[0:6]. Each entry has a fixed correspond to one of the resident TxQ's. RQID[0] has to be #b1 for any of the tags to match. A value of #b0 in RQID[0] indicates that the message should go into a Short RxQ, and can only match entries in SysReg 5, RxShortTags. The RxQ's are essentially divided into two sets, RxQ 0-3, and RxQ 4-7. The former can be used only when RQID[1] is #b0, and the latter, only when RQID[1] is #b1. RxQ-0 corresponds to Pcp-0 L priority; RxQ-1 corresponds to Pcp-0 H priority; RxQ-2 corresponds to Pcp-1 L priority; RxQ-3 corresponds to Pcp-1 H priority; and so on.
Read details:	Upon a Read, the 8B value contains the tags in Status[8:63]. Tag-0 is in Status[8:14], Tag-1 is in Status[15:21] and so on.
Update details:	Update is achieved using CtrlAddr[8:10] to indicate the Tag field being updated, with value coming form DataIn[0:6].
Reset Value:	Undefined upon reset
Comments:	

SysReg No.:	7
SysReg Name:	OverflowQState
Usage:	This regiser keeps track of the OverflowQ's state. The state values are kept in bits [36:63] and holds the following states: In Reg[36:42]: OverflowCPtr[0:6]; In Reg[43:49]: OverflowPPtr[0:6]; In Reg[50:58]: OverflowBase[0:8]; In Reg[59:63]: OverflowBound[0:4]; The semantics of these state variables are similar to those of normal message Rx queue.
Read details:	A Read of this register returns an 8B value with relevant state in Status[36:63]
Update details:	Update is achieved 7b at a time, using data in DataIn[0:6]. The address bits CtrlAddr[9:10] indicates the 7b slice that is updated each time.
Reset Value:	Undefined upon reset
Comments:	Overflow queue perform wrap-around just like ordinary PCP queues. The unit of allocaiton in the Overflow queue is 32B. Each short msg is allocated 32B in the Overflow queue, even though they occupy only the first 16B (128b). *Overflow queue is always mirrored in both SRAMs, at the same offset.* Dedicated shadow locations for this register is provided in the NES SRAMs. The content of this register is automatically written out to those locations as a 4B quantity occupying the odd word (bits[32:63]) after the PPtr has been updated.

SysReg No.:	8
SysReg Name:	RxQEmpty
Usage:	This keeps track of the which Medium RxQ's, including Overflow queue, are empty. 9b ([55:63]) are used, and correspond to the 8 Medium queues, followed by the Overflow queue.
Read details:	Upon a Read, an 8B value is returned, with the useful data in Status[55:63].
Update details:	This register should not be updated. Any attempt to do so leads to undefined behaviour.
Reset Value:	Cleared (all 1) upon reset.
Comments:	

SysReg No.:	9
SysReg Name:	RxQEnable
Usage:	Similar to TxQEnable, this register tracks/indicates which RxQ's are enabled. Messages are only received into RxQ's that are enabled. Aside from software on aP and sP writing to this register to enable or disable a queue, an enabled queue with overflow capability turned on is automatically disabled by hardware when an overflow occurs. Similar to the case of TxQEnable, aP/sP has to wait ?? bclks after disabling a RxQ before it can assume that the queue is shut down. This is to give time for reception in progress to complete.
Read details:	A Read returns 8B where Status[48:63] contains the actual enabled status.
Update details:	Update uses CtrlAddr[7:10] to indicate which TxQ's enable status is being written. The written value comes from DataIn[6].
Reset Value:	Cleared (all 0) upon reset.
Comments:	

SysReg No.:	10
SysReg Name:	MemQOut0State
Usage:	Queue state of MemQOut0 queue. 50b of state is stored here, in [14:63] of the 64b register.
Read details:	An 8B read returns the state in Status[14:63], with the following fields: In Reg[14:20]: PPtr[0:6] In Reg[21:27]: CPtr[0:6] In Reg[28:36]: Base[0:8] In Reg[37:41]: Bound[0:4] In Reg[42:47]: Permissions In Reg[48:52]: TransBASE[0:4] In Reg[53:59]: TagonBASE[0:6] In Reg[60:63]: TagonMask[0:3]
Update details:	To Update this register, CtrlAddr[8:10] is used to pick the byte field as follows (DataIn refers to the 7b data supplied). 0–PPtr[0:6] 1–CPtr[0:6] 2–Base[0:6] 3–Base[7:8], Bound[0:4]; 4–Permissions 5–TransBASE DataIn[2:6] 6–TagonBASE DataIn[1:6] 7–TagonMask DataIn[3:6]
Reset Value:	Cleared (all 0) upon reset.
Comments:	will probably change (remove all permissions but Arctic priority, RQID offset)

SysReg No.:	11
SysReg Name:	MemQOut1State
Usage:	Queue state of MemQOut1 queue. 50b of state is stored here, in [14:63] of the 64b register.
Read details:	An 8B read returns the state in Status[14:63], with the following fields: In Reg[14:20]: PPtr[0:6] In Reg[21:27]: CPtr[0:6] In Reg[28:36]: Base[0:8] In Reg[37:41]: Bound[0:4] In Reg[42:47]: Permissions In Reg[48:52]: TransBASE[0:4] In Reg[53:59]: TagonBASE[0:6] In Reg[60:63]: TagonMask[0:3]
Update details:	To Update this register, CtrlAddr[8:10] is used to pick the byte field as follows (DataIn refers to the 7b data supplied). 0–PPtr[0:6] 1–CPtr[0:6] 2–Base[0:6] 3–Base[7:8], Bound[0:4]; 4–Permissions 5–TransBASE DataIn[2:6] 6–TagonBASE DataIn[1:6] 7–TagonMask DataIn[3:6]
Reset Value:	Cleared (all 0) upon reset.
Comments:	will probably change (remove all permissions but Arctic priority, RQID offset)

SysReg No.:	12
SysReg Name:	DMATxQState
Usage:	Queue state of DMATxQ. 21b of state is stored here, in [43:63] of the 64b register.
Read details:	An 8B read returns the state in Status[43:63], with the following fields: In Reg[43:49]: DMATxBase[0:6] In Reg[50:55]: DMATxBound[0:5] In Reg[56:63]: DMATxLogSrcNum[0:7] (Logical Source Node Number)
Update details:	Update to these fields are achieved with CtrlAddr[9:10] specifying the field: 0 – DataIn[0:6] specifies DMATxBase[0:6] 1 – DataIn[0:5] specifies DMATxBound[0:5] DataIn[6] specifies DMATxLogSrcNum[0] 2 – DataIn[0:6] specifies DMATxLogSrcNum[1:7]
Reset Value:	Undefined upon reset
Comments:	Will probably add one or more fields for BusOps, i.e. TT, etc. The DMATxBase and DMATxBound values are shared by both the DMA Read and DMA Send Engines. The SRAM addresses are generated with: SRAMAddr[0:3] = Base[0:3] SRAMAddr[4:6] = Base[4:6] OR Ptr[0:2] SRAMAddr[7:10] = Ptr[3:6] SRAMAddr[11:12] = #b00 DMATxBound is 6b, and is compared to Ptr[0:5], and only matches if (Ptr[6] == #b1). This allows the entire queue to be used.

SysReg No.:	13
SysReg Name:	ReclaimEnable
Usage:	This register keeps track of which Med TxQs and RxQs have NES Reclaim feature enabled. 16b of state is kept in [48:63], corresponding to state for TxQs 8-15 and RxQs 8-15.
Read details:	An 8B Read returns the state in Status[48:63].
Update details:	Update is achieved with DataIn[6] specifying the new enable value, and CtrlAddr[7:10] specifying the queue (0 refers to TxQ 8, 1 refers to TxQ 9, and so on).
Reset Value:	Cleared (all 0) upon reset.
Comments:	

SysReg No.:	14
SysReg Name:	DMARxQState
Usage:	Queue state of DMARxQ's: both the Data queue and the Bus Operation (or sometimes referred to as the Cmd) queue. 26b of state is stored here, in [36:63] of the 64b register.
Read details:	An 8B Read returns the state in Status[36:63].
Update details:	In an update, CtrlAddr[8:10] picks the field to and the value in the 7b value in DataIn[0:6] is used unless otherwise specified: 0–DataBase[0:6] 1–DataBase[7:8](from DataIn[0:1]), (** no longer used; see comments **) DataBound[0:4](from DataIn[2:6]) 2–BusBase[0:6] 3–BusBase[7:8], BusBound[0:4]
Reset Value:	Undefined upon reset
Comments:	DataBound is not used; instead, bound is determined by BusBound. The Data and Bus (Cmd) queues move in lock-step; Cmd in units of (2 * 64) bits, Data in units of (2 * 32) Bytes.

SysReg No.:	15
SysReg Name:	RxQInterruput
Usage:	This register keeps track of whether an interrupt has been raised for a RxQ. Interrupts are normally raised upon the arrival of an interrupt type packet, or if the RxQ has be set to interrupt-on-arrival. 16b of state holds the interrupt status for the 16 normal RxQ's 0-15.
Read details:	8B Read returns this state in Status[48:63].
Update details:	Each update clears a particular bit, chosen with CtrlAddr[7:10].
Reset Value:	Cleared (all 0) upon reset.
Comments:	

SysReg No.:	16
SysReg Name:	APInterruptMask
Usage:	This is the interrupt Mask for the aP, which controls whether an interrupt is raised with the aP when any bit in the RxQInterrupt register or ErrorReg register is set. Currently, there are 21b of state in this registers. [48:63] is used to mask RxQInterrupt, while [43:47] is used to mask ErrorReg.
Read details:	read: Status[43:63] = mask for RxQInterrupt, ErrorReg
Update details:	Update is achieved using data in DataIn[0:6] and CtrlAddr[9:10] to select the 7b to update: 0 : [43:49] 1 : [50:56] 2 : [57:63]
Reset Value:	Cleared (all 0) upon reset.
Comments:	will change as size of ErrorReg changes

SysReg No.:	17
SysReg Name:	SPInterruptMask
Usage:	This is the interrupt Mask for the sP, which controls whether an interrupt is raised with the sP when any bit in the RxQInterrupt register or ErrorReg register is set. Currently, there are 21b of state in this registers. [48:63] is used to mask RxQInterrupt, while [43:47] is used to mask ErrorReg.
Read details:	read: Status[43:63] = mask for RxQInterrupt, ErrorReg
Update details:	Update is achieved using data in DataIn[0:6] and CtrlAddr[9:10] to select the 7b to update: 0 : [43:49] 1 : [50:56] 2 : [57:63]
Reset Value:	Cleared (all 0) upon reset.
Comments:	will change as size of ErrorReg changes

4.2 sBIUAddr

Two categories of visible states in the sBIU: Queue state and SysReg state.

4.2.1 sBIU Queue State

sBIU state Q state (Table 51) can be update from aP and sP, but currently can only be read from the sP. Each update can only modify a 7b register. Each read extracts the value of 2 (contiguous and aligned) registers which is returned in bits [17:23] and [25:31] of the odd word.

Actual sSRAM address of queue buffer is computed as follows: (" ++" indicates concatenation.)

```
sSRAM Addr[0:11] = Base[1:5]
                 ++ (Base[6:8] OR QPtr[0:2])
                 ++ QPtr[3:6];
```

Bound is used to delimit the extent of the queue's msg buffer space. When computing a new pointer,

```
if (NewPtr[0:4] == Bound),
         set NewPtr[0:6] = #b0000000;
```

4.2.2 sBIU SysReg State

SysReg state in sBIU are 7b each, with the addressing structure shown in (Table 52). Each read by the sP extracts the value of 2 (contiguous and aligned) registers which is returned in bits [17:23] and [25:31] of the odd word. sBIU SysReg state can be updated from both aP and sP. Currently, it can only be read from sP. (We may put in support for reading it from aP later.)

Width (bits)	Bit field	Content	Description
2	[0:1]	`01`	==== Fixed Field ====
3	[2:4]	`000`	CompGp Pas/Vas-0
		`001`	CompGp Pas/Vas-1
		`010`	** reserved **
		`011`	** reserved **
		`100`	CompGp MemQ (MemQIn corresponds to MemQ-Rx, L priority; MemQOut corresponds to MemQ-Tx)
		`101`	** reserved **
		`110`	** reserved **
		`111`	** reserved **
1	[5]	`0`	** unused **
1	[6]	`0`	Tx
		`1`	Rx
1	[7]	`0`	L priority
		`1`	H priority
3	[8:10]	`x00`	In Bb[1:7] C-ptr for Rx, P-ptr for Tx.
		`x01`	In Bb[1:7] P-ptr for Rx, unused for Tx.
		`x10`	Base[0:6] in Bb[1:7]
		`x11`	Base[7:8] in Bb[1:2]; Bound[0:4] in Bb[3:7].

Table 51: Addressing of sBIU's Q state. The address bits refer to the internal sBIUAddr that sBIU use when addressing the physical state elements.

Width (bits)	Bit field	Content	Description
2	[0:1]	`10`	==== Fixed Field ====
2	[2:3]	`00`	==== Fixed Field ====
3	[4:6]	`000`	==== Fixed Field ====
4	[7:10]	`X`	SysReg Number

Table 52: Addressing of sBIU SysReg state. The address bits refer to the internal sBIUAddr that sBIU use when addressing the physical state elements.

SysReg No.:	0
SysReg Name:	
Usage:	Together with SysReg 1, this register stores intermediate state of OnePoll implementation. (See Section 2.5.4) Access to this is needed to implement transparent context switching by allowing this intermediate state to be saved and restored. The fields in this register are: [0] OnePoll? (whether in the middle of OnePoll?) [1:6] OnePollAddr[0:5]
Reset Value:	
Comments:	

SysReg No.:	1
SysReg Name:	
Usage:	Together with SysReg 0, this register stores intermediate state of OnePoll implementation. Access to this is needed to implement transparent context switching by allowing this intermediate state to be saved and restored. This register contains OnePollAddr[6:11]
Reset Value:	
Comments:	

SysReg No.:	2
SysReg Name:	
Usage:	Bits [0:3] contains the ArcticLateAck for the Vas (aks short) Rx queue. Bit[0] : Vas-0, L priority. Bit[1] : Vas-0, H priority. Bit[2] : Vas-1, L priority. Bit[3] : Vas-1, H priority.
Reset Value:	
Comments:	

SysReg No.:	3
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

SysReg No.:	4
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

SysReg No.:	5
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

SysReg No.:	6
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

SysReg No.:	7
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

4.3 aBIUAddr

Two categories of visible states in the aBIU: Queue state and SysReg state.

4.3.1 aBIU Queue State

aBIU state Q state (Table 53) can only be written but not read out. can be update from aP and sP, but currently can only be read from the aP. Each update can only modify a 7b register. Each read extracts the value of 2 (contiguous and aligned) registers which is returned in bits [17:23] and [25:31] of the odd word.

Except for MemQIn, MemQDIn and DMA-RxDQ, actual aSRAM address is computed as follows: (" ++" indicates concatenation.)

```
aSRAM Addr[0:11] = Base[1:5]
                 ++ (Base[6:8] OR QPtr[0:2])
                 ++ QPtr[3:6];
```

For MemQIn, the address is actually a sSRAM address. It is computed in a manner similar to the above:

```
sSRAM Addr[0:11] = Base[1:5]
                 ++ (Base[6:8] OR QPtr[0:2])
                 ++ QPtr[3:6];
```

For MemQDIn and DMA-RxDQ, actual aSRAM address is computed as follows: (" ++" indicates concatenation.)

Width (bits)	Bit field	Content	Description
2	[0:1]	01	==== Fixed Field ====
3	[2:4]	000	** reserved **
		001	** reserved **
		010	CompGp Pas/Vas-2
		011	CompGp Pas/Vas-3
		100	** reserved **
		101	** reserved **
		110	CompGp MemQ (MemQIn corresponds to MemQ-Tx, L priority for all of its state, except C-ptr. TxQ's do not have C-ptr state in BIU's. MemQIn's C-ptr state in aBIU is mapped to MemQ-Rx, L priority's P-ptr. The rest of MemQ-Rx state is not used, nor is MemQ-Tx, H priority.
		111	CompGp DMA (DMA-RxCmdQ corresponds to DMA-Rx, L priority; DMA-RxDQ corresponds to MemQ-Rx, H priority. DMA-Tx is not used.
1	[5]	0	** unused **
1	[6]	0	Tx
		1	Rx
1	[7]	0	L priority
		1	H priority
3	[8:10]	000	In Bb[1:7] C-ptr for Rx, P-ptr for Tx.
		001	In Bb[1:7] P-ptr for Rx, unused for Tx.
		010	Base[0:6] in Bb[1:7]
		011	Base[7:8] in Bb[1:2]; Bound[0:4] in Bb[3:7].
		1XX	** unused **

Table 53: Addressing of aBIU's Q state. The address bits refer to the internal aBIUAddr that aBIU use when addressing the physical state elements.

```
aSRAM Addr[0:11] = Base[0:3]
                 ++ (Base[4:8] OR QPtr[0:4])
                 ++ QPtr[5:6]
                 ++ #b00;
```

Bound is used to delimit the extent of the queue's msg buffer space. When computing a new pointer,

```
if (NewPtr[0:4] == Bound),
        set NewPtr[0:6] = #b0000000;
```

4.3.2 aBIU SysReg State

Width (bits)	Bit field	Content	Description
2	[0:1]	`10`	==== Fixed Field ====
2	[2:3]	`01`	==== Fixed Field ====
3	[4:6]	`000`	==== Fixed Field ====
4	[7:10]	`X`	SysReg Number

Table 54: Addressing of aBIU SysReg state. The address bits refer to the internal aBIUAddr that aBIU use when addressing the physical state elements.

SysReg state in sBIU are 7b each, with the addressing structure shown in (Table 52). Each read by the sP extracts the value of 2 (contiguous and aligned) registers which is returned in bits [17:23] and [25:31] of the odd word. sBIU SysReg state can be updated from both aP and sP. Currently, it can only be read from aP. (We may put in support for reading it from sP later.)

SysReg No.:	0
SysReg Name:	
Usage:	Together with SysReg 1, this register stores intermediate state of OnePoll implementation. (See Section 3.6.4) Access to this is needed to implement transparent context switching by allowing this intermediate state to be saved and restored. The fields in this register are: [0] OnePoll? (whether in the middle of OnePoll?) [1:6] OnePollAddr[0:5]
Reset Value:	
Comments:	

SysReg No.:	1
SysReg Name:	
Usage:	Together with SysReg 0, this register stores intermediate state of OnePoll implementation. Access to this is needed to implement transparent context switching by allowing this intermediate state to be saved and restored. This register contains: Bit[0:5] : OnePollAddr[6:11] Bit[6] : nesSmEnabled (Enable Shared Memory)
Reset Value:	
Comments:	

SysReg No.:	2
SysReg Name:	
Usage:	Bits [0:3] contains the ArcticLateAck for the Vas (aks short) Rx queue. Bit[0] : Vas-0, L priority. Bit[1] : Vas-0, H priority. Bit[2] : Vas-1, L priority. Bit[3] : Vas-1, H priority.
Reset Value:	
Comments:	

SysReg No.:	3
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

SysReg No.:	4 – 7
SysReg Name:	
Usage:	HAL-Statistics Reg, 32b distributed over 4 registers of 8b each. This register is not meant to be updated. Upon reset, it will be zero-ed. Software should sample it frequently enough so there is at most one "roll-over" between sampling. Software should keep track of the last value (starting with 0), and compute the number of events counted.
Reset Value:	0
Comments:	

SysReg No.:	8
SysReg Name:	
Usage:	MemQDIn, P-ptr
Reset Value:	
Comments:	

SysReg No.:	9
SysReg Name:	
Usage:	MemQDIn, C-ptr
Reset Value:	
Comments:	

SysReg No.:	10
SysReg Name:	
Usage:	MemQDIn Base[0:6]
Reset Value:	
Comments:	

SysReg No.:	11
SysReg Name:	
Usage:	MemQDIn Base[7:8] and MemQDIn Bound[0:4].
Reset Value:	
Comments:	

SysReg No.:	12
SysReg Name:	
Usage:	** unused **
Reset Value:	
Comments:	

SysReg No.:	13
SysReg Name:	
Usage:	DmaRxD, C-ptr
Reset Value:	
Comments:	

SysReg No.:	14
SysReg Name:	
Usage:	DmaRxD Base[0:6]
Reset Value:	
Comments:	

SysReg No.:	15
SysReg Name:	
Usage:	DmaRxD Base[7:8] and DmaRxD Bound[0:4].
Reset Value:	
Comments:	

4.3.3 aBIU Serviced Space Response State

Width (bits)	Bit field	Content	Description
2	[0:1]	`10`	==== Fixed Field ====
2	[2:3]	`01`	==== Fixed Field ====
2	[4:5]	`01`	==== Fixed Field ====
5	[6:10]	`X`	TT[0:4]

Table 55: Addressing of aBIU SysReg state containing Serviced Space response table.

This set of registers is used to program the NES's response to various bus transactions to the sP Serviced Space, and for transactions that are not address specific. Each register keeps track of the response for a Transaction Type:

			Reset Value
Bb[1]	==	SSRetry[TT]	0 (No Retry)
Bb[2]	==	SSNotify[TT]	0 (No Notify)

4.3.4 aBIU Snooped Space Response State

Width (bits)	Bit field	Content	Description
2	[0:1]	`11`	==== Fixed Field ====
1	[2]	`X`	HAL (ClsSRAM) Data[0]
1	[3]	`1`	==== Fixed Field ====
2	[4:5]	`X`	HAL (ClsSRAM) Data[1:2]
5	[6:10]	`X`	TT[0:4]

Table 56: Addressing of aBIU SysReg state containing Snooped Space response table.

This set of registers is used to program the HAL (aka ClsSRAM) check responses, given a transaction type TT[0:4], and HalData[0:2]. Each register keeps track of the response:

Address (xPAddr[16:28])	size	Location name
#x0FE7	8B	aBIU-qconfig-tmp
#x0FE8 to #x0FEB	40B	** unused **
#x0FEC to #x0FEF	32B	aSRAM-tmp
#x0FF0 to #x0FF3	32B	ExpRawData
#x0FF4	8B	Overflow-queue-status
#x0FF5	8B	Express Empty Msg
#x0FF6	8B	sBIU-qconfig-tmp
#x0FF7	8B	CTRL-qconfig-tmp
#x0FF8 to #x0FFF	64B	qptr-base

Table 57: Hardwired Special Locations in sSRAM.

Address (xPAddr[16:28])	size	Location name
#x1FE8 to #x1FEB	32B	SM Miss pattern
#x1FEC to #x1FEF	32B	sSRAM-tmp
#x1FF0 to #x1FF3	32B	ExpRawData
#x1FF4	8B	Overflow-queue-status
#x1FF5	8B	Express Empty Msg
#x1FF6	8B	sBIU-qconfig-tmp
#x1FF7	8B	aBIU-qconfig-tmp
#x1FF8 to #x1FFF	64B	qptr-base

Table 58: Hardwired Special Locations in aSRAM.

			Reset Value
Bb[1]	==	HALRetry[TT,HAL]	0 (No Retry)
Bb[2]	==	HALNotify[TT,HAL]	0 (No Notify)
Bb[3]	==	HALShared[TT,HAL]	0? (Not Shared)

4.4 Special SRAM Addresses

This section lists the hardwired SRAM locations allocated for special uses. Locations are 8Byte Word addressable. Each address is listed in a unified 13bit, 8Byte-Word address which can address both aSRAM and sSRAM with no ambiquity. The left most bit selects between aSRAM (#b0) and sSRAM (#b1). Table 57 lists the hardwired sSRAM addresses, while Table 58 lists the hardwired aSRAM addresses.

5 Message Formats

5.1 Message Format as Required by Arctic

This section describes the message format expected by the Arctic network. It is provided here mostly for reference as the NES performs some reformatting of software composed message, so the message format as indicated below need not be directly exposed to application software.

1. Header:

Width (bits)	Bit field	Content
1	[0]	** ignored **
1	[1]	Priority
14	[2:15]	UpRoute (programmable to use only a subfield)
16	[16:31]	DownRoute (programmable to use only a subfield)
11	[32:42]	** ignored **
5	[43:47]	Length (in units of 4B words)

   The actual packet format that StarT-Voyager uses is more restricted, with 10b UpRoute and 10b DownRoute. (See Section 5.2.)

2. Body: between 2 to 22 (inclusive) 4B words. Arctic does not interpret them.

3. Tail: last (aligned) 16b is CRC

Figure 11 shows the header format in two views. The first is a straight-forward view, while the second is offset by 16b. Because the Arctic header is 48b, it is more convenient for software to view it as starting at bit 16, so that the body of a message appears to software as aligned to a 32b boundary.

5.2 Message Formats Accepted by NES's Tx Unit

The Tx Unit receive packets from various Tx queues in the NES for sending out to the Arctic network. The different formats are described below, and illustrated in Figure 12.

The UpRoute for a packet can either be randomly generated by the TxU, specified in the message itself(not available for Short Tx Format with Tag-on data because not enough bits are avilable), or supplied by the Destination Translation Table (not available with Fully Specified Format since no Destination Translation Table is involved). A per-TxQ fixed UpRoute takes precedence over everything else. If this is not enabled, but the queue allows user specified per-message UpRoute, and a message specifies an UpRoute, that will be adopted. Randomly generated UpRoute has the lowest precedence.

1. *Fully Specified Format*, used by *trusted* code on aP and sP. Access to this message type has to be protected as it allows message forgery. This is also the format of packets that are actually launched into the Arctic network (except for the first 16 bits, which are dropped just before the packet is launched into the Arctic network).

   (a) Header: First 64 bits.

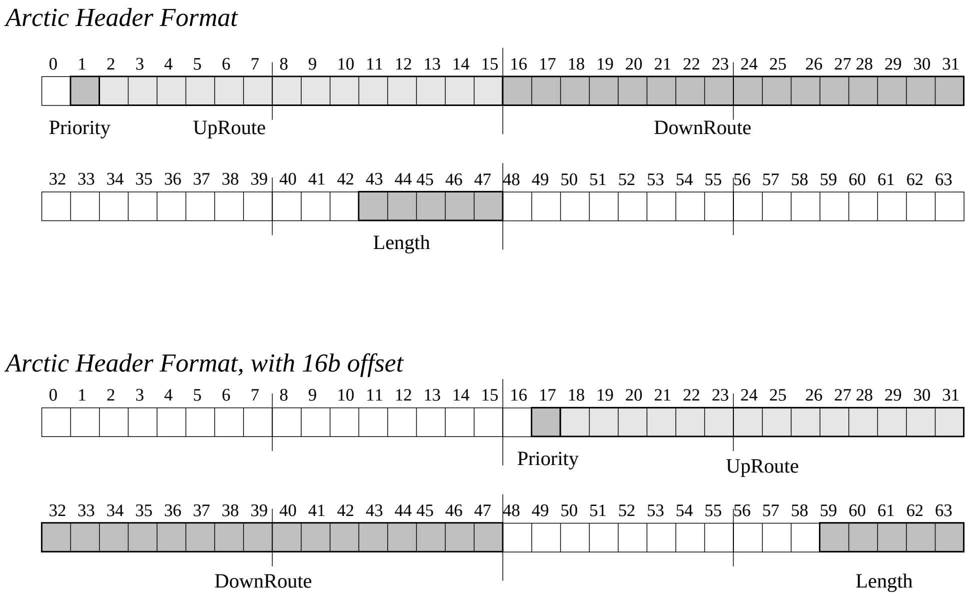

Figure 11: Arctic header format.

Width (bits)	Bit field	Content
1	[0]	Must be 1; identifies this as a long-header msg.
1	[1]	Short Message? 0 ⇒ Standard Msg. 1 ⇒ Short Msg.
9	[5:13]	tag-on-data SRAM address offset. If tag-on-data is not enabled, [12] is used to indicate whether application is specifying UpRoute? 1 ⇒ yes.
1	[14]	tag-on-data? 0 ⇒ No. 1 ⇒ Yes. [5:13] indicates tag-on-data SRAM address offset.
2	[15:16]	** ignored **
1	[17]	Arctic Network Priority
10	[18:27]	UpRoute (If specify UpRoute is “No”, this is replaced at the TxU with randomly generated bits.)
8	[28:35]	logical source node number
1	[36]	interrupt on arrival? 0 ⇒ No.
10	[37:46]	physical destination node number (DownRoute) (Software should not use this to identify its physical node number.)
3	[47:49]	Message Opcode (MsgOp)
9	[50:58]	Receive Queue ID (RQID)
5	[59:63]	Length (in units of 4B words)

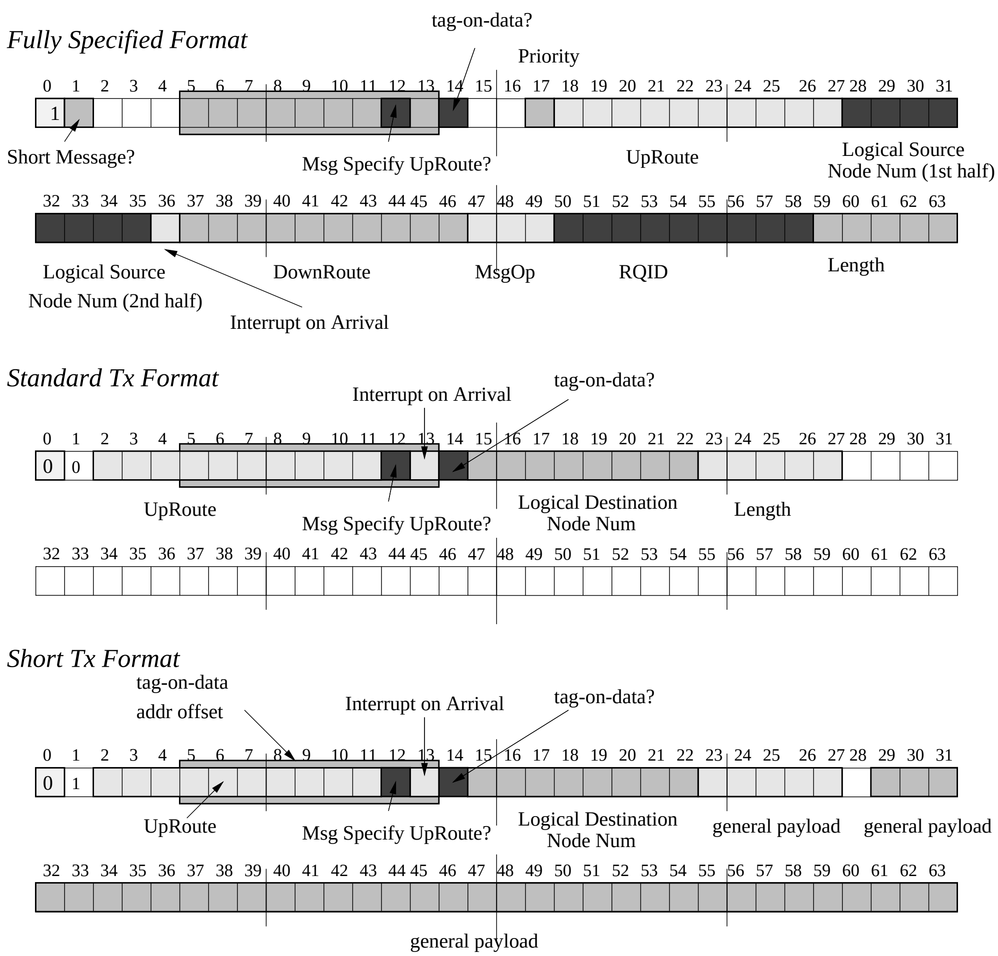

Figure 12: The three different Tx (header) formats.

(b) Body: between 2 to 22 (4B) words.

2. *Standard Tx Format*, used by code on aP or sP to compose medium size messages.

(a) Header: The header is specified as occupying 64 bits, but only the first 32b are used:

Width (bits)	Bit field	Content
1	[0]	Must be 0; identifies this as a short-header msg.
1	[1]	Must be 0; identifies this as a Standard Tx Format as opposed to Short Tx Format.
3	[2:4]	Part of UpRoute specification if bit[14] indicates no tag-on-data and bit[12] indicates no random UpRoute generation.
9	[5:13]	tag-on-data SRAM address offset. If tag-on-data is not enabled, [13] is used to indicate whether interrupt on arrival, $1 \Rightarrow$ yes. [12] is used to indicate whether application is specifying UpRoute? $1 \Rightarrow$ yes. [2:11] UpRoute (if UpRoute specified) The field [2:11] can be used as general payload when tag-on-data is disabled, and random UpRoute generation is enabled.
1	[14]	tag-on-data? $0 \Rightarrow$ No. $1 \Rightarrow$ Yes. [5:13] indicates tag-on-data SRAM address offset.
8	[15:22]	logical destination node number (DownRoute)
5	[23:27]	Length (in units of 4B word, including 2 words of header specifier)
1	[28]	** reserved **
3	[29:31]	** ignored **

(b) Body: between 2 to 22 (4B) words.

Implementor's Notes: We have chosen this format to be close to that of Short Message Format. This will allow sharing of some datapaths for reformatting the header. Complete sharing is not possible because the message length is explicitly specified here, but not in the short message case.

3. *Short Tx Format*; packets in this format are 64b and come from aP or sP sending short message via 32b uncached write.

Width (bits)	Bit field	Content
1	[0]	Must be 0; identifies this as a short-header msg.
1	[1]	Must be 1; identifies this as a Short Tx Format as opposed to Standard Tx Format.
3	[2:4]	Part of UpRoute specification if bit[14] indicates no tag-on-data and bit[12] indicates no random UpRoute generation.
9	[5:13]	tag-on-data SRAM address offset. If tag-on-data is not enabled, [13] is used to indicate whether interrupt on arrival, 1 ⇒ yes. [12] is used to indicate whether application is specifying UpRoute? 1 ⇒ yes. [2:11] UpRoute (if UpRoute specified) The field [2:11] can be used as general payload when tag-on-data is disabled, and random UpRoute generation is enabled.
1	[14]	tag-on-data? 0 ⇒ No. 1 ⇒ Yes. [5:13] indicates tag-on-data SRAM address offset.
8	[15:22]	logical destination node number (DownRoute)
5	[23:27]	General payload.
1	[28]	** reserved **
3	[29:31]	** general payload **
32	[32:63]	** general payload **

The NES has to transform packets with Standard Tx Format and Short Tx Format into Arctic Format before transmission. Figures 13 and 14 illustrate this process. Each entry in the Destination Translation Table is 8B and has the format shown in Table 59. Here are several points regarding the NES's treatment of the different message formats:

- **fixed vs random UpRoute:** The NES hardware is capable of generating a random UpRoute, using a Linear Feedback Shift Register, LFSR). When fifo ordering of messages between a pair of source and destination sites is not important, this allows the multiple paths between the pair of sites in the Arctic Fatree to be exploited, without the application overhead of generating a random number.

  When ordering matters, the application code can specify an UpRoute. This can be done in one of two ways. On a per-message basis, if tag-on is not used, bit[12] indicates if UpRoute for the packet should be taken from the UpRoute field in the message. The second method of picking a non-random UpRoute is to specify it on a per Destination Translation Table entry basis. This method means that all packets translated with that entry will use the same UpRoute.

  > **comment:** *Currently, a message with Fully Specified Format which also uses Tag-on is unable to specify a fixed UpRoute. We propose to change things so that by default, Fully Specified Format does not use Random UpRoute generation.*

- **Interrupt message delivery:** Messages delivery can be by interrupt. There are two ways to specify this. One is on a per receive queue basis. A receive queue can be put into a "raise interrupt when packet arrives" mode. The second method is on a per-message basis, and is specified by the sender. The per-message specification only overides the per-queue specification when the queue specifies non-interrupt, and the message requests for interrupt. Note that the latter method is only available when Tag-on is not used.

- **Tag-on message support:** This allows a certain amount of data from the NES SRAMs to be appended to a out-going message. The size of appended data is either 1.5 (32B+16B) or 2.5 cache-lines, with the choice specified in a system register. We expect a system to be running with all NESes configured to the same tag-on size.

  When Tag-on is used with Fully Specified or Standard Tx Formats, a variable amount of payload can come from the first part of the message which includes the header. The "length" field specifies the full Arctic Packet size, i.e. 2 (for header) + x (for payload following the header) + Tag-on-size. Buffer space allocation in a Pcp queue for such a message is RoundUpTo32B(length - Tag-on-size).

  A Fully Specified Short message is only useful when used with Tag-on. The Arctic packet sent out contains 48b Arctic header, following by 64b of unspecified data, the Tag-on data, and then 16b of CRC.

Width (bits)	Bit field	Content	Description
1	[0]	X	Fixed UpRoute?
10	[1:10]	X	UpRoute
8	[11:18]	X	Logical Source
10	[19:28]	X	DownRoute
2	[29:30]	X	** Unused **
3	[31:33]	X	MsgOp
6	[34:39]	X	RQID
1	[40]	X	Valid?
23	[41:63]	X	** Unused **

Table 59: Format of each entry (8B) in the Destination Translation Table.

5.3 Message Formats as Delivered by NES's Rx

Software sees two packet formats described below, and illustrated in Figure 15.

1. Medium Rx Format:

   (a) The first 64 bits are the header:

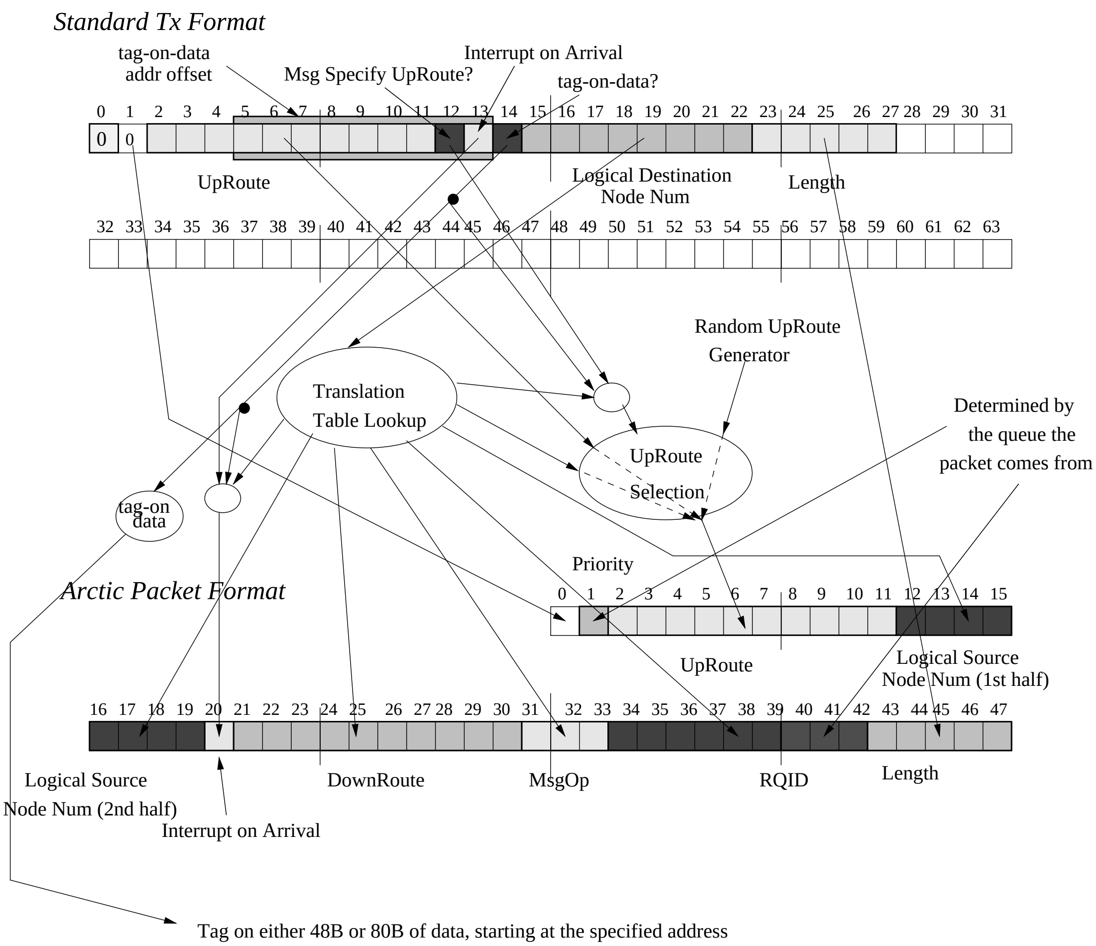

Figure 13: Reformatting out-going packets in Standard Tx Format into Arctic Format.

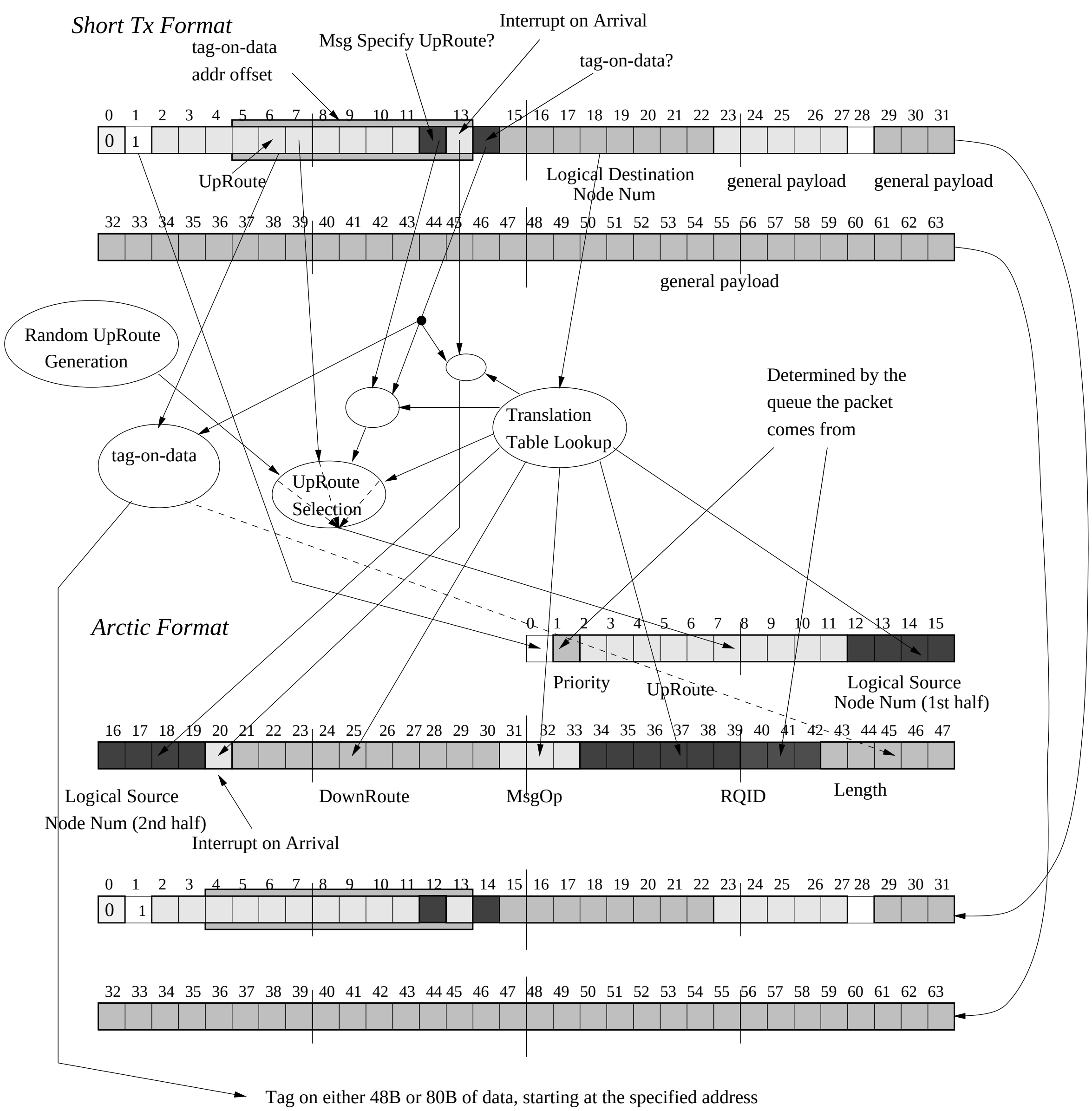

Figure 14: Reformatting out-going packets in Short Tx Format into Arctic Format.

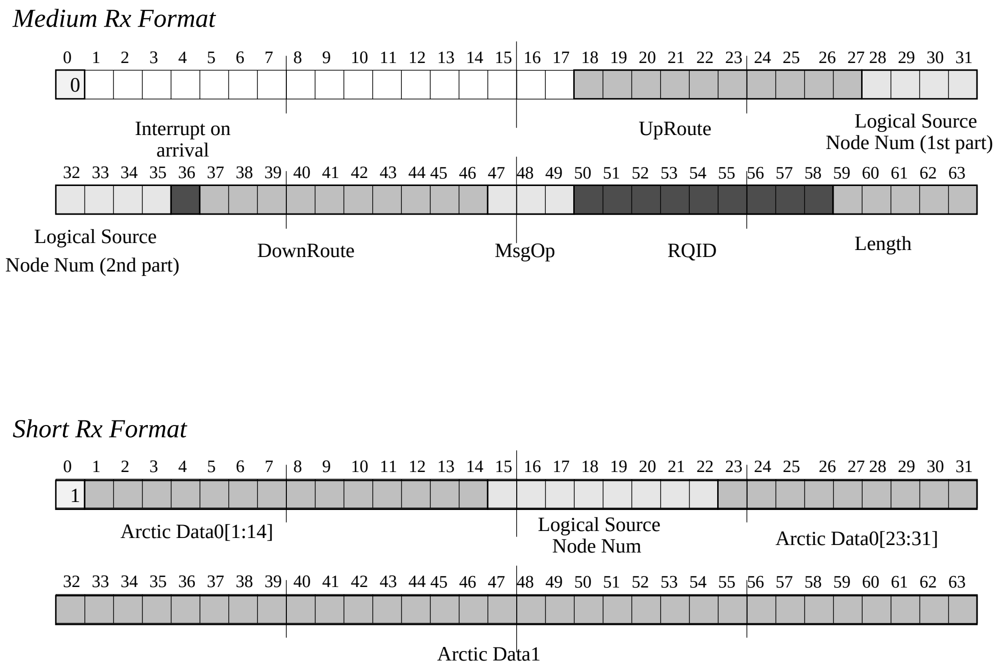

Figure 15: The two different Rx (header) formats.

Width (bits)	Bit field	Content
1	[0]	Must be 1; identifies this as a normal msg.
15	[1:15]	** ignored **
2	[16:17]	** ignored **
10	[18:27]	UpRoute
8	[28:35]	logical source node number
1	[36]	interrupt on arrival? 0 $\Rightarrow$ No.
10	[37:46]	physical destination node number (DownRoute) (Software should not use this to identify its physical node number.)
3	[47:49]	Message Opcode (MsgOp)
9	[50:58]	Receive Queue ID (RQID)
5	[59:63]	Length (in units of 4B words)

Currently, these are the defined Message Opcode types:

- Standard Message
- DMA Data
- DMA Cmd
- Drop on blocked Rx queue.

(b) Body: between 2 to 22 (4B) words.

2. Short Rx Format:

Width (bits)	Bit field	Content
1	[0]	Must be 0; identifies this as a short msg.
14	[1:14]	Arctic packet data0[1:14].
8	[15:22]	Arctic packet Header's Source node number field, headr[12:19].
9	[23:31]	Arctic packet data0[23:31].
32	[32:63]	Arctic packet data1.

3. Short Rx Format when Overflow occurs:

   When en-overflow is set for an RxQ and the queue is full, a packet destined for that queue will be channeled to the Overflow queue, which is handled by the sP. Overflowing medium messages are simply placed in the Overflow queue. (Space allocation in the Overflow queue is in 32B granularity.) Short messages however requires some additional special treatment. The problem here is that by the time overflow is detected, the Short message no longer contains RQID, information that is needed for the sP to deal with the message. Thus, overflowing short messages are expanded into a 128b quanity, the first 64b only contains a bit indicating that this is a short message (bit[0] is set to 1), and the RQID (in bits[50:59]), while the second 64b contains the compressed short message. If there is additional data tagged-on to the message, they follow after these 16B.

Medium Rx Format requires no reformatting of the in-coming packet by the NES. Short Rx Format requires the reformating illustrated in Figure 16.

5.4 Software View of Message Formats

The short message format seen by the TxU is used for several different purposes, and the software running on the aP and the sP see different aspects of the format as described below. The NES can operate in one of two modes. In the ShAddrContent mode, the 64b value that makes up a short message in the Short Tx Format is specified by software on the aP or sP via 32b uncached writes. Two pieces of information are associated with the uncached write: (i) 32b of data; (ii) 32b of address. The address provides the first 32b of the 64b short message, while the data provides the second 32b, *i.e.* the aBIU/sBIU writes the 32b address into [0:31] field of a 64b aligned buffer, and the 32b of data associated with the uncached write into the [32:63] field.[10]

In the second mode, the ShExpContent mode, all 64b value is specified by the aP or sP software, using either a single 64b write, or two 32b writes. In the latter case, the write to the second 32b has to occur before that to the first 32b.

The first 32b has slightly different semantics depending on whether the short message send is from the sP or aP side. This is particularly so when the NES is in ShAddrContent mode.

1. For the aP side, the format is:

[10] In this description, we use big-endian convention to view the 64b quantity. "First" refers to the more significant 4B, and "second", the less significant 4B.

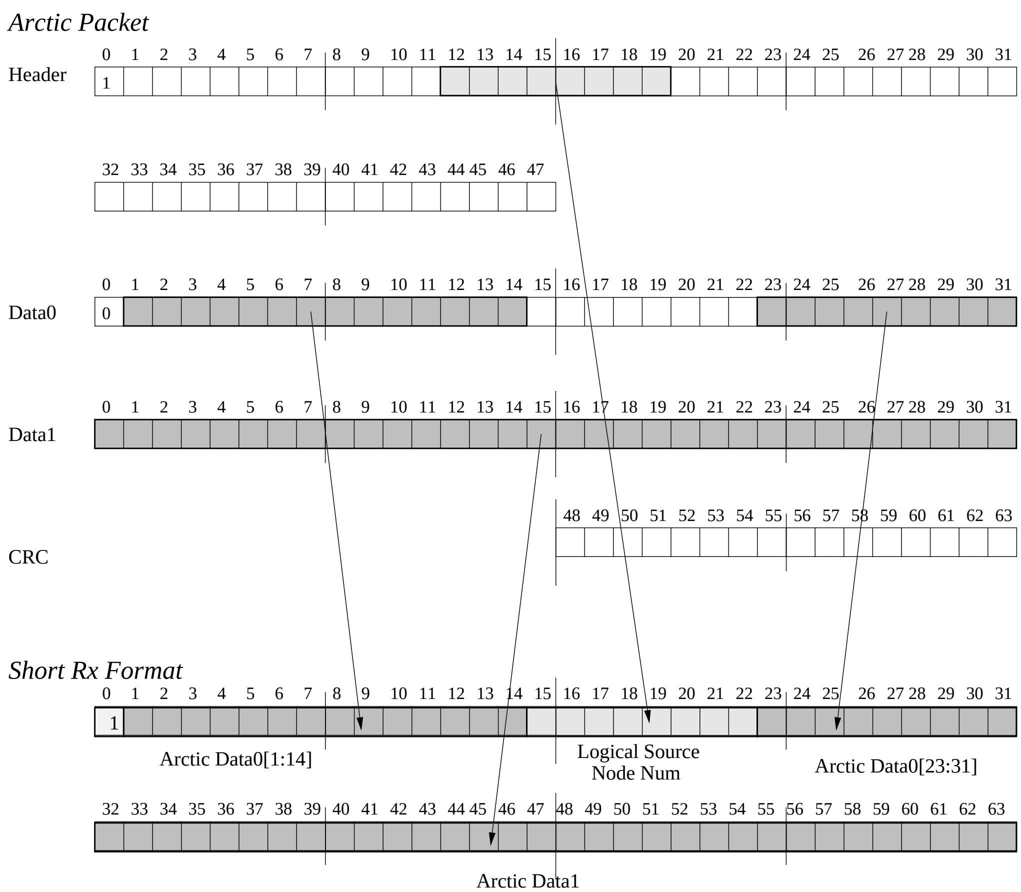

Figure 16: Reformatting in-coming packet (Arctic Format) into Short Rx Format.

Width (bits)	Bit field	Content
4	[0:3]	** constrained to specify short message (ShTx) space **
3	[4:6]	** constrained to specify short message (ShTx) space **
1	[7]	** constrained to specify short message ComGp. **
6	[8:13]	tag-on-data SRAM address offset. If tag-on-data is not enabled, [13] is used to indicate whether interrupt on arrival, 1 $\Rightarrow$ yes. [12] is used to indicate whether to specify the UpRoute? 1 $\Rightarrow$ yes. For the case where tag-on-data is enabled, a fixed UpRoute can still be specified, but only on a per-TxQ basis. [2:11] UpRoute (if UpRoute Specified) Notice that a large parts of this field is constrained so that there is freedom to choose only the last 3b of UpRoute. We are short of bits and just have to live with the contraint.
1	[14]	tag-on-data? 0 $\Rightarrow$ No. 1 $\Rightarrow$ Yes. [8:13] indicates tag-on-data SRAM address offset.
8	[15:22]	logical destination node number
5	[23:27]	general payload;
1	[28]	priority
3	[29:31]	** ignored **, always #b100

2. For the sP side, the format is:

Width (bits)	Bit field	Content
3	[0:2]	** constrained to specify special short message space **
2	[3:4]	** constrained to specify specific short queue **
9	[5:13]	tag-on-data SRAM address offset. If tag-on-data is not enabled, [13] is used to indicate whether interrupt on arrival, 1 $\Rightarrow$ yes. [12] indicates whether UpRoute is specified? 1 $\Rightarrow$ yes. For the case where tag-on-data is enabled, whether to use randomly generated UpRoute can still be controlled, but only on a per-TxQ basis. [2:11] UpRoute (if UpRoute Specified) Notice that part of this field is constrained so that there is freedom to choose only the last 8b of UpRoute. We are short of bits and just have to live with this constrain. The field [5:11] can be used as general payload when tag-on-data is disabled, and random UpRoute generation is enabled.
1	[14]	tag-on-data? 0 $\Rightarrow$ No.
8	[15:22]	logical destination node number
5	[23:27]	General payload.
1	[28]	priority
3	[29:31]	** ignored **, always #b100

The main reason for the difference between the aP side and the sP side uncached write message send's address semantics is that on the sP side, the short message send window has to be 256MB to accomodate all the bits needed, particularly to specify the sRAM address. This is too large a window onto the address space for the aP side.

5.5 DMA Message Formats

The DMA Data Packets generated by the DMA Send Engine has the format shown in Figure 17. The DMA Receive Engine ignores the header of a DMA Data Packet, places the next 2 *times* 32B into the DMARx Data queue, and the following 2 *times* 64b into the DMARx Cmd queue. Because the DMA Receive Engine does not see the actual Arctic packet but one that has already been processed by the RxU, it is possible to fabricate a DMA Data Packet using an Express Message with TagOn of size 2.5 cache-lines.

5.6 NES Internal Message Formats

This section describes the formats of "messages" that are used internal to the NES. These include commands to the NES from the local sP which are issued through the MemQOut queues. The MemQOut queues also allows some types of messages to be launched into Arctic network. These are all described in Section 5.6.1. Section 5.6.3 describes the format of entries in the MemQIn

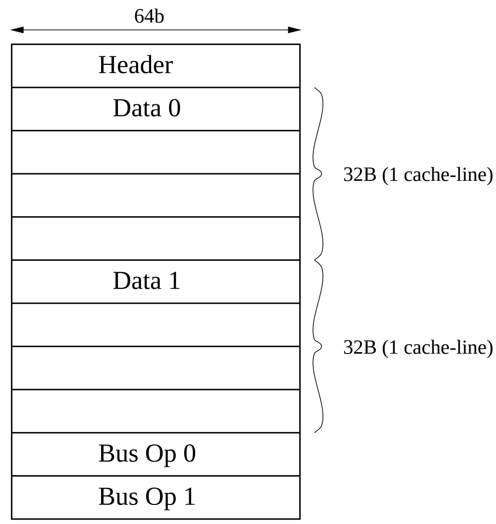

Figure 17: Format of a DMA Data packet.

queue, which includes captured aP-bus transactions, and acknowledgements to commands that have been completed by the NES. Finally, Section 5.6.2 describes the format of commands in the DMA Cmd queue.

5.6.1 MemQOut Command/Message Formats

Among the NES Internal Message Formats, 5 are launched by the sP via the MemQOut queues, and the encodings chosen have to distinguish between them. The 5 types of "commands" and the bits and encoding used to differentiate them are shown below.

Width (bits)	Bit field	Content	OpType (to CTRL)	Description
3	[3,15:16]	1XX	00	sP Message Launch via MemQOut queues (Msg)
		011	11	DM (Data Motion) Cmd (DM)
		010	10	DMA Read/Send Cmds (DmaRdSnd)
		001	01	NES Mastered aP-bus Op Cmd (MOp) and Approval
		000	01	DMA Receive Cmd (DmaRcv)

The encoding uses bit[3] to differentiate between Msg (#b1) and the other commands, bit[15] distinguishes between commands serviced by the aBIU (#b0) and CTRL, and bit[16] identifies those serviced by the DM Unit (#b1) and Dma Read/Send Engine.

Another requirement of the encoding listed below is that the aBIU has to be able to distinguish between NES Mastered aP-bus Op, DMA Rx Mastered Op Cmd, DMA Receive Cmd and ApprovalReg/Memory Emulation Cmds, and ClsSRAM Update Cmd. Bit[14,16,18] are used for this purpose, with the encoding shown below:

Width (bits)	Bit field	Content	Description
1	[14,16,18]	`00X`	DMA Receive Cmd
		`01X`	NES Mastered aP-bus Op Cmd
		`10X`	DMA Rx Mastered Op Cmd
		`110`	ApprovalReg/Memory Emulation Cmds
		`111`	ClsSRAM Update

Finally, the sP needs to distinguish between an aBIU Captured aP-bus Transaction and aBIU Acknowledgement. This is done with bit[17]:

Width (bits)	Bit field	Content	Description
1	[17]	`1`	aBIU Acknowledgement
		`0`	aBIU Captured aP-bus Transaction

1. *sP Message Launched via MemQOut queues*

   The sP can launch message from the MemQOut queues. The format, described in Table 60, is illustrated in Figure 18. If the message is composed using 32b uncached write, certain fields become constrained. These fields can still be used if the request is composed using 64b uncache write.

2. *NES Mastered aP-bus Op Cmd*

   This command is either specified by the sP, or by the DMA Read Engine and Reclaim Engine. When initiated by the sP, certain fields are constrained if the command is issued using 32b uncached write. These fields can still be used if the request is composed using 64b uncache write. The format, described in Table 61, is illustrated in Figure 19.

3. *DM (Data Motion) Cmd:*

   This is used by hardware (aBIU and sBIU) and sP (via MemQOut queues) to request inter-SRAM transfers. Table 62 describes the format, which is also illustrated in Figure 20. The reason for chosing this encoding is to exploit data-paths that are already in existent for other functionalities as much as possible.

4. *DMA Read/Send Cmds:*

   This is used by the sP to set up DMA Read Engine and DMA Send Engine. The format, shown in Table 63, is also illustrated in Figure 21 where the format of other DMA related cmds are shown.

5. *DMA Receive Cmd:*

   This is used by the sP, both local and remote, to set up DMA Receive Engine. The format, in Table 64, is also illustrated in Figure 21 where the format of other DMA related cmds are shown.

6. *ApprovalReg/Memory Emulation Cmds via MemQOut:*

   Approval for a pending transaction that has been captured in the ApprovalReg can be given either via an Cmd Space ImmdCmd or via MemQOut. The format of this command when given via MemQOut is shown in Tables 65 and 66 below and illustrated in Figure 22. *An ApprovalReg Cmd which puts the ApprovalReg into the Ready state is not considered done until the ApprovalReg state changes from the Ready state to Free state. (See Figure 7 for state transition of ApprovalReg.)*

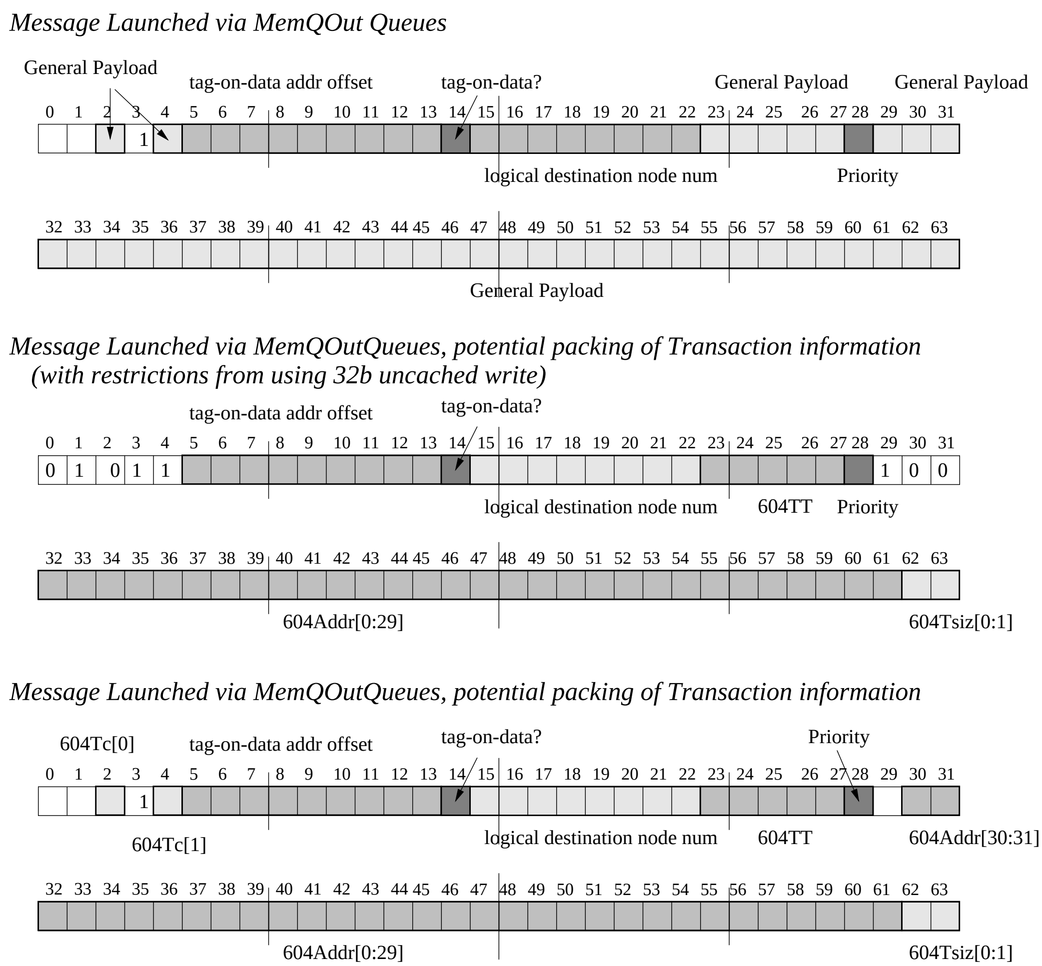

Figure 18: Format of sP messages launched via MemQOut queues.

Width (bits)	Bit field	Content
3	[0:2]	** unused ** (** constrained to specify Special ShTx Space, #b010 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 1, indicating that this is a Msg to be launched into the Arctic network, as opposed to Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM (Data Motion) cmd.).
1	[4]	General payload (** constrained to specify MemQOut, #b1 ** when request is composed using 32b uncached write. Where needed by Shared Memory communication, this may carry: 604Tc[2])
9	[5:13]	tag-on-data SRAM address offset. If tag-on-data is not enabled, [13] is used to indicate whether interrupt on arrival, 1 $\Rightarrow$ yes. [12] is used to indicate whether application is specifying UpRoute? 1 $\Rightarrow$ yes, and [2:11] is the UpRoute. Notice that part of this field is constrained so that there is freedom to choose only the last 8b of UpRoute. We are short of bits and just have to live with this constrain. The field [5:11] can be used as general payload when tag-on-data is disabled, and random UpRoute generation is enabled.
1	[14]	tag-on-data? 0 $\Rightarrow$ No. 1 $\Rightarrow$ Yes.
8	[15:22]	logical destination node number
5	[23:27]	IP-offset, usually 604TT (Transaction Type).
1	[28]	Priority
3	[29:31]	General payload (Forced to #b100 if composed by sP with short message style uncached write.)
30	[32:61]	General payload (For a typical Shared Memory transaction, we expect it to carry: 604Addr[0:29])
2	[62:63]	General payload (For a typical Shared Memory transaction, we expect it to carry: 604Tsiz[0:1])

Table 60: Format of sP messages launch via MemQOut queues.

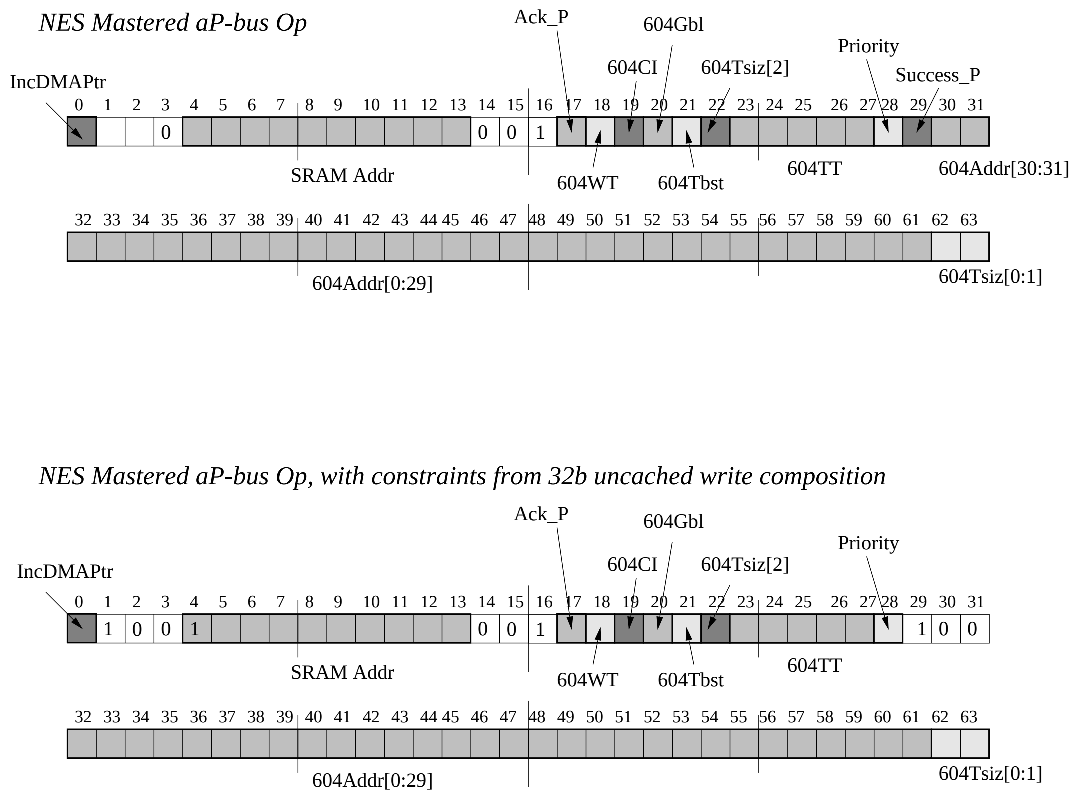

Figure 19: NES Mastered aP-bus Op Cmd format.

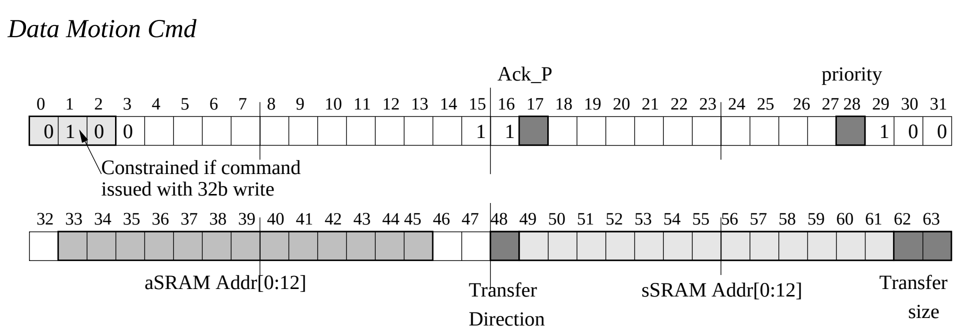

Figure 20: Data Motion (DM) Cmd format.

Width (bits)	Bit field	Content
1	[0]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
10	[4:13]	32B aligned aSRAM address associated with any transfers to/from NES. (** [4] is constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write. This limits addressibility to half of the memory. This can be overcome with the use of 64b uncached writes.)
3	[14:16]	== Fixed field ==: must have the value #b001, indicating that this is an NES Mastered aP-bus Op cmd.
1	[17]	Ack_P (Whether this request should be acknowledged.)
1	[18]	_604WT
1	[19]	_604CI
1	[20]	_604GBL
1	[21]	_604TBST
1	[22]	604Tsiz[2]
5	[23:27]	604TT (Transaction Type).
1	[28]	Priority
3	[29:31]	(Forced to #b100 if composed by sP with short message style uncached write;) bit[29] ** Reserved ** bit[30:31] used for 604Addr[30:31].
30	[32:61]	604Addr[0:31]
2	[62:63]	604Tsiz[0:1]

Table 61: NES Mastered aP-bus Op Cmd format

Width (bits)	Bit field	Content
3	[0:2]	** unused ** (** constrained to specify Special ShTx Space, #b010 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM (Data Motion) cmd.).
1	[4]	(** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
6	[5:10]	** unused **
3	[11:13]	** unused **
1	[14]	** unused **
2	[15:16]	== Fixed field ==: must have the value #b11, indicating that this is a DM cmd.
1	[17]	Ack_p.
12	[18:27]	** unused **
1	[28]	MemQOut priority.
2	[29:31]	** unused ** (Forced to #b100 if composed by sP with short message style uncached write.)
1	[32]	** unused **
13	[33:45]	aSRAM Addr[0:11]. Note that aSRAM Addr[12] is not explicitly specified because Data Motion requires that it be exactly the same as sSRAM Addr[12].
2	[46:47]	** unused **
1	[48]	Direction of transfer: aSRAM to sSRAM (#b0). sSRAM to aSRAM (#b1).
13	[49:61]	sSRAM Addr[0:12]
2	[62:63]	Transfer size. (4B:#b10, 8B:#b00, 32B:#b01; note that this encoding corresponds to 604TSiz[0:1].)

Table 62: DM (Data Motion) Cmd format.

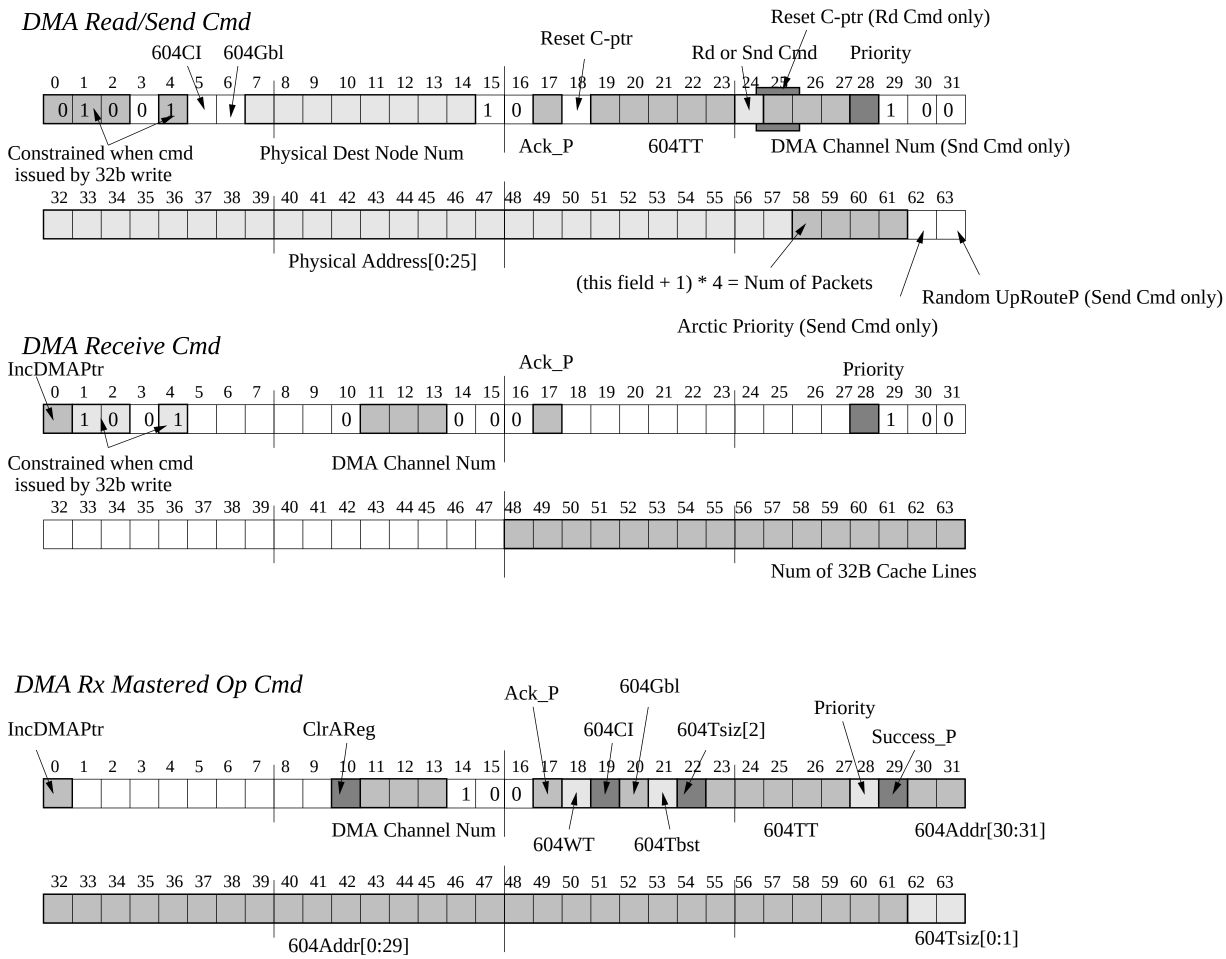

Figure 21: DMA Cmds format.

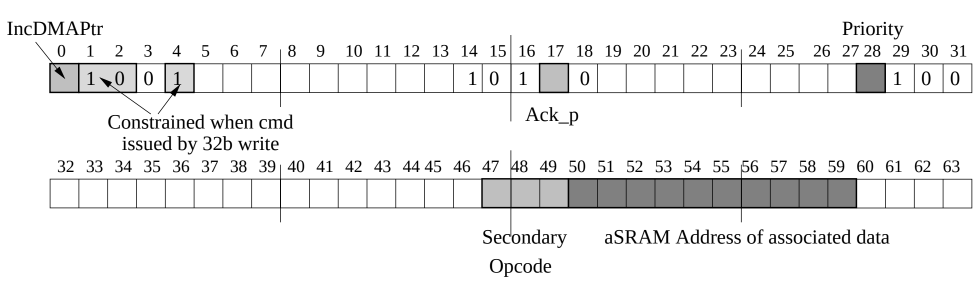

Figure 22: Format of ApprovalReg/Memory Emulation Cmds issued via MemQOut.

Width (bits)	Bit field	Content
3	[0:2]	** unused ** (** constrained to specify Special ShTx Space, #b010 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	(** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
1	[5]	_ci of bus transaction.
1	[6]	_gbl of bus transaction.
8	[7:14]	Physical destination node number. (Only used for Send cmd.)
1	[15:16]	== Fixed field ==: must have the value #b10, indicating that this is a DMA cmd, serviced by CTRL.
1	[17]	Ack_P (Whether this request should be acknowledged.)
1	[18]	whether to reset C-ptr.
5	[19:23]	TT (60X bus transaction type) of bus transaction.
1	[24]	DMA Read Cmd (#b0) vs DMA Send Cmd (#b1).
3	[25:27]	For Send Cmd, DMA Channel Number. For Read Cmd, bit 25 indicates whether to reset P-ptr.
1	[28]	MemQOut priority.
3	[29:31]	** unused ** (Forced to #b100 if composed by sP with short message style uncached write.)
26	[32:57]	64B Aligned Physical Addr[0:25] (Only used for Read Cmd)
6	[58:63]	For Read cmd, all 6 bits are used to specify the number of CL's (bus operations) involved. The actual number is ((this field + 1) * 2). For Send cmd, bits[58:63] are used to specify the number of packets. (this field + 1)*4. (This makes it possible to specify up to 64 packets, each with 64 byte data.) The remaining two bits are used to select Arctic network priority (bit[62]), and whether to use randomized up-Route (bit[63].

Table 63: DMA Read/Send Cmd format.

Width (bits)	Bit field	Content
1	[0]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	(** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
6	[5:9]	** unused **
1	[10]	== Fixed field ==: must have the value #b0. (Has to be 0 because 1 is used to indicate clearing ApprovalReg in DMA Rx Mastered Op Cmd.)
3	[11:13]	DMA Channel Number.
1	[14]	== Fixed field ==: must have the value #b0. Distinguish between a DMA Receive Cmd and a DMA Rx Mastered Op Cmd.
2	[15:16]	== Fixed field ==: must have the value #b00, indicating that this is a DMA cmd serviced by aBIU.
1	[17]	Ack_P (Whether this request should be acknowledged.)
7	[18:24]	** unused **
3	[25:27]	** unused **
1	[28]	MemQOut Priority.
3	[29:31]	** unused ** (Forced to #b100 if composed by sP with short message style uncached write.)
16	[32:47]	** unused **
16	[48:63]	Pending Cnt (of 32B Cache-line).

Table 64: DMA Receive Cmd format.

Width (bits)	Bit field	Content
1	[0]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	(** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
9	[5:13]	** unused **
1	[14]	== Fixed field ==: must have the value #b1.
2	[15:16]	== Fixed field ==: must have the value #b01, indicating that this is an an Approval Cmd serviced by aBIU.
1	[17]	Ack_p.
1	[18]	== Fixed field ==: must have the value #b0
9	[19:27]	** unused **
1	[28]	MemQOut priority.
3	[29:31]	** unused ** (if request composed by 32b uncached write, this is constrained to have the value #b01
13	[32:44]	** unused **
1	[45]	Used only for ([47:49] == 000) when this indicates the _SHD signal to use.
1	[46]	** unused **
3	[47:49]	Secondary Opcode.
10	[50:59]	aSRAM address for any associated data.
4	[60:63]	** unused **

Table 65: ApprovalReg/Memory Emulation Cmds Format.

Width (bits)	Bit field	Content	Description
3	[47:49]	`000`	Approval for pending bus transaction in ApprovalReg
		`001`	Lock/reserver ApprovalReg
		`010`	Reset (to 0) DMA RxData queue C-ptr in aBIU.
		`011`	Clear/free ApprovalReg; effectively discarding any captured transaction or previous lock/reservation.
		`100`	Nop. This allows issuing a synchronouse Clswrite enable only command.
		`101`	Lock Notification resource, i.e. any transaction that needs notification will be retried until this lock is cleared.
		`110`	LockaPBus toggle
		`111`	Enable Notification resource, undoing any previous lock.

Table 66: Secondary opcode encoding for ApprovalReg/Memory Emulation Cmds issued via MemQOut. The encoding is similar to the corresponding ImmdCmds.

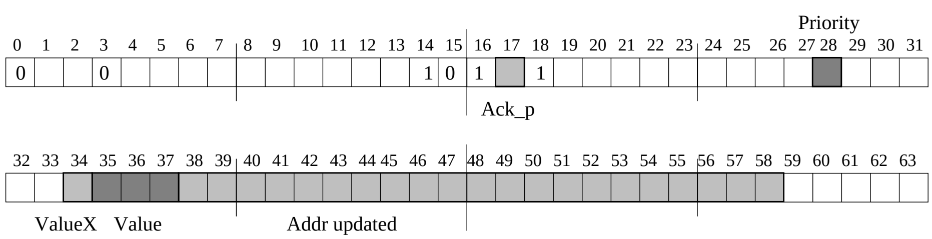

Figure 23: ClsSRAM Update Cmd format. See also Table 67.

7. *ClsSRAM Update Cmd:*

   The ClsSRAM Update Cmd is issued into a MemQOut queue by the sBIU when it receives the ClsSRAM update command. (See 2.2.3.) Figure 23 illustrates the format of this command, which is also described in Table 67.

5.6.2 DMA Cmd Queue Formats

DMA Rx Mastered Op Cmd:

This command is composed by the DMA Send Engine at the sender site, and serviced by the DMA Receive Engine at the receiver site. It is similar to NES Mastered aP-bus Op launched by sP, differing in that a DMA Rx Mastered Op Cmd has to specify the DMA Channel involved, and the actual SRAM address is provided by an aBIU internal pointers into the DMA-DRx queue. The format below is illustrated in Figure 21.

Width (bits)	Bit field	Content
1	[0]	== Fixed field ==: must have the value 0, indicating to aBIU that it should not increment DMA-DRx Q pointer.
2	[1:2]	Has the value #b00 if composed by sBIU, or #10 if directly composed to one of the MemQOut queues using Special ShTx.
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	** unused ** (Will have value 1 if this is directly issued using 32b Special ShTx into one of the MemQOut queues.)
9	[5:13]	** unused **
1	[14]	== Fixed field ==: must have the value #b1.
2	[15:16]	== Fixed field ==: must have the value #b01, indicating that this is a Cmd serviced by aBIU.
1	[17]	Ack_p.
1	[18]	== Fixed field ==: must have the value #b1
9	[19:27]	** unused **
1	[28]	MemQOut priority.
3	[29:31]	** unused **
2	[32:33]	** unused **
1	[34]	ValueX (extra, currently unused)
3	[35:37]	Value
21	[38:58]	Update Addr
5	[59:63]	** unused **

Table 67: ClsSRAM Update Cmd format. See also Figure 23

Width (bits)	Bit field	Content
1	[0:2]	Whether to increment aBIU's DMA-DRx Q pointer. Increments if #b1. (** constrained to specify Special ShTx Space, #b0 ** when request composed using 32b uncached write.)
2	[1:2]	** unused ** (** constrained to specify Special ShTx Space, #b10 ** when request composed using 32b uncached write.)
1	[3]	== Fixed field ==: must have the value 0, indicating that this is an Internal cmd (Internal cmd includes NES Mastered aP-bus Op, DMA cmd, and DM cmd.).
1	[4]	General payload (** constrained to specify MemQOut, #b1 ** when request composed using 32b uncached write.)
5	[5:9]	** unused **
1	[10]	Whether to clear ApprovalReg. Clear if 1. The clearing must only occur after the requested bus transaction has completed.
3	[11:13]	DMA Channel number.
1	[14]	== Fixed field ==: must have the value #b1. Distinguish between a DMA Receive Cmd and a DMA Rx Mastered Op Cmd.
1	[15:16]	== Fixed field ==: must have the value #b00, indicating that this is an NES Mastered aP-bus Op cmd.
1	[17]	Ack_P (Whether this request should be acknowledged.)
1	[18]	_604WT
1	[19]	_604CI
1	[20]	_604GBL
1	[21]	_604TBST
1	[22]	604Tsiz[2]
5	[23:27]	604TT (Transaction Type);
1	[28]	Priority
3	[29:31]	(Forced to #b100 if composed by sP with short message style uncached write; bit[30:31] used for 604Addr[30:31].)
30	[32:61]	604Addr[0:31]
2	[62:63]	604Tsiz[0:1]

Table 68: DMA Rx Mastered Op Cmd Format

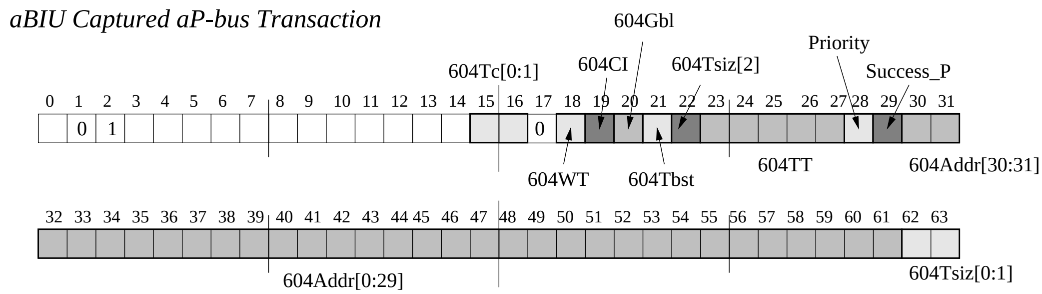

Figure 24: Format of aP-bus transactions captured by aBIU, and placed in MemQIn queue.

5.6.3 MemQIn Formats

1. *aBIU Captured aP-bus Transactions:*

   When the aBIU captures a bus operation, it packs the information in the way described below and forwards it to the sP, via the MemQueueIn queue. The format, shown in Table 69, is also illustrated in Figure 24.

2. *Acknowledgement formats:*

   These are the formats of acknowledgements to commands issued by the local sP via MemQOut, and remote sP via DMA Cmd Queue.

   (a) *NES Mastered aP-bus Op Ack:* As per issued cmd, except bits[1:3] are forced to #b000, and bit[29] is set to indicate success (#b1) or failure (#b0).

   (b) *DM (Data Motion) Ack:*

[0]	same as in issued cmd
[1:2]	forced to #b00.
[3:31]	same as in issued cmd
[32]	same as in issued cmd
[33:45]	source of DM at end of cmd
[46]	** undefined **
[47:48]	same as bits [46:47] of cmd
[49]	#b0
[50:61]	destination of DM at end of cmd
[62:63]	#b00

   (c) *DMA Read Ack:*

[0]	same as in issued cmd
[1:2]	forced to #b00.
[3:31]	same as in issued cmd
[32:51]	DMATxCmdPageAddr
[52:57]	PageOffset of last addr read + 1
[58:63]	#b000000

   (d) *DMA Send Ack:*

Width (bits)	Bit field	Content
1	[0]	** reserved **
2	[1:2]	== Fixed field == : must have the value #b01; used for distinguishing between captured aP bus transaction, incoming short messages from Arctic network (or Empty messages), and MemQOut Cmd Acks when One-Poll is used.
12	[3:14]	** reserved **
2	[15:16]	604Tc[0:1]
1	[17]	== Fixed field ==, Ack_P: must have the value 0; indicates that this is a newly captured transaction, as opposed to an acknowledgement to request serviced by aBIU).
1	[18]	_604WT
1	[19]	_604CI
1	[20]	_604GBL
1	[21]	_604TBST
1	[22]	604Tsiz[2]
5	[23:27]	604TT (Transaction Type); conveniently, #b11111 is not a valid 604TT, as required by Short Tx Format.
1	[28]	Priority
1	[29]	Success_P (meaningful only when Ack_P is #b1)
2	[30:31]	604Addr[30:31]
29	[32:61]	604Addr[0:29]
2	[62:63]	604Tsiz[0:1]

Table 69: Format of Captured aP-bus Transactions

[0]	same as in issued cmd
[1:2]	forced to #b00.
[3:31]	same as in issued cmd
[32:51]	DMARdCmdPageAddr
[52:57]	PageOffset of last addr read + 1
[58:63]	#b000000

(e) *DMA Receive Ack:* As per issued cmd, except bits[1:2] are forced to #b00.

(f) *DMA Channel Done Ack:*

[0]	** undefined **
[1:2]	#b00
[3]	#b0
[4:9]	** undefined **
[10]	#b0
[11:13]	DMA Channel Number
[14:16]	** undefined **
[17]	#b1, indicating this is an Ack
[18:63]	** undefined **

(g) *ApprovalReg Ack:* As per issued cmd, except bits[1:3] are forced to #b000, bit[15] is #b0 and bits[62:63] are not defined.

(h) *ClsSRAM Update Ack:* As per issued cmd, except bits[1:3] are forced to #b000, bit[15] is #b0 and bits[62:63] are not defined.

(i) *DMA Rx Mastered Op Ack:* As per issued cmd, except bits[1:3] are forced to #b000, and bit[29] is set to indicate success (#b1) or failure (#b0).

6 CTRL

comment: *Please refer to Daniel Rosenband's Master's thesis[8].*

A NES Function Blocks Interface Signal Descriptions

A.1 Interfaces Between NES Functional Units

The major function blocks in the NES are:

- NESchip
- sSRAM
- aSRAM
- clSRAM
- TxU/RxU

The NESchip is further divided into the following major blocks:

- aBIU
- sBIU
- CTRL

A.1.1 NESchip and aSRAM

Signal direction is described with respect to the NESchip, *i.e.* "I" means an input to the NESchip.

Name	I/O	# bits	Description
a-port			
aPortAddr	O	12b	aSRAM aP-bus port address
aPortR/_W	O	2b	aSRAM odd 32b and even 32b R/_W;
_aPortOE	O	2b	aSRAM odd 32b and even 32b _OE;
_aPortCE	O	2b	aSRAM odd 32b and even 32b _CE;
I-bus			
iBusAddr	O	12b	aSRAM I-bus port address
isPortR/_W	O	2b	aSRAM I-bus port R/_W 2b;
isPortOE	O	2b	aSRAM I-bus port OE 2b
isPortCE	O	2b	aSRAM I-bus port CS 2b
IbusData	I/O	64b	aSRAM I-bus data

A.1.2 NESchip and sSRAM

Signal direction is described with respect to the NESchip, *i.e.* "I" means an input to the NESchip.

Name	I/O	# bits	Description
s-port			
SSRSAddr	O	12b	sSRAM sP-bus port address
SSRSRW	O	1b	sSRAM R/_W; both banks share a common _OE signal.
_SSRSOE	O	1b	sSRAM _OE; both banks share a common _OE signal.
_SSRSCE	O	2b	sSRAM odd 32b and even 32b _CE;
I-bus			
iBusAddr	O	12b	sSRAM I-bus port address
iaPortR/_W	O	2b	sSRAM I-bus port R/_W 2b;
iaPortOE	O	2b	sSRAM I-bus port OE 2b
iaPortCE	O	2b	sSRAM I-bus port CS 2b
IbusData	I/O	64b	sSRAM I-bus data

A.1.3 NESchip and TxU/RxU

Signal direction is described with respect to the NESchip, *i.e.* "I" means an input to the NESchip.

Name	I/O	# bits	Description
RxFIFOEn	O	1b	RxFIFO Enable
RxRQID	I	7b	RQID of received packet
RxMsgOp	I	3b	MsgOp of received packet
RxFIFOaE	I	1b	RxFIFO almost empty
RxFIFOoR	I	1b	RxFIFO output ready
RxArcticAck	O	1b	Ctrl requesting that an Ack be returned to the Arctic output port connected to the NES.
TxFIFOEn	O	1b	TxFIFO enable
TxOK	O	1b	??
TxTagonSize	O	1b	Size of tag-on-data, either 32B or 64B.
TxPriority	O	1b	Tx priority
TxRQIDOffset	O	3b	Tx RQID Offset
TxArcticAck	I	1b	Arctic input port connect to NES has indicated buffer free.

A.1.4 CTRL and aBIU

Signal direction is described with respect to the CTRL, *i.e.* "I" means an input to the CTRL.

Name	I/O	# bits	Description
AC (aBIU, CTRL) Bus			
AcBusPassThru	I	1b	When AcBusV is asserted, AcBusPassThru indicates whether the requested operation is just passing through CTRL, destined for aBIU.
AcBusAddr	I	11b	When AcBusV is asserted, AcBusAddr is the address of the state that is accessed.
AcBusData	I	7b	When AcBusV is asserted, and AcBusRW indicates a write, AcBusData contains the data to be written.
AcBusRW	I	1b	When AcBusV is asserted, AcBusRW indicates whether the requested operation is state read, or state write
AcBusV	I	1b	AC bus signals valid; when asserted, this indicates that there is valid data on the AC bus.
AcBusDone	O	1b	Pulsed for one cycle to indicate that the AC bus operation requested by aBIU has been completed.
CA (aBIU, CTRL) Bus			
CaBusAddr	O	8b	When CaBusV is asserted, CaBusAddr is the address of the state that is accessed.
CaBusData	O	7b	When CaBusV is asserted, and CaBusRW indicates a write, CaBusData contains the data to be written.
CaBusRW	O	1b	When CaBusV is asserted, CaBusRW indicates whether the requested operation is state read, or state write.
CaBusV	O	1b	CA bus signals valid; when asserted, this indicates that there is valid data on the CA bus.
CaBusDone	I	1b	Pulsed for one cycle to indicate that the CA bus operation requested by aBIU has been completed.

Name	I/O	# bits	Description
Short message signals			
AbiuShReq	I	1b	aBIU short message I-bus request
AbiuShGnt	O	1b	aBIU short message I-bus grant
AbiuLateAck	I	1b	aBIU asking that a late ack be returned to Arctic.
AbiuPcpEmpty	O	4b	Indicates whether a PcpR queue is empty. There are 4 different PcpR queues, so 1b for each queue.
Global shared memory signals			
AcdIBusReq	I	1b	Shared memory I-bus reqeust
AcdIBusGnt	O	1b	CTRL granting aBIU usage of I-bus for transfer of shared memory request address and control information into MemQIn.
NES Mastered aP-Bus Transaction Signals			
aPBusReq	O	1b	Internal request for aBIU to perform a mastered operation on aP-bus
aPBusDone	I	1b	The previous internal (from MemQOut, Reclaim Engine, Dma Read Engine, for Dma Rx Cmd) request to aBIU is done
(Data-motion) Inter-sRAM data transfer signals			
AbiuDmReq	I	1b	aBIU Data Motion request
AbiuDmGnt	O	1b	aBIU Data Motion grant
SbiuDmDone	O	1b	aBIU Data Motion done
K-Bus Signals			
KBusDataAC	I	32b	K-bus data from aBIU to CTRL
KBusDataCA	O	64b	K-bus data from CTRL to aBIU
KBusAddr	I	13b	K-bus address

A.1.5 CTRL and sBIU

Signal direction is described with respect to the CTRL, *i.e.* “I” means an input to the CTRL.

Name	I/O	# bits	Description
SC (sBIU, CTRL) Bus			
ScBusPassThru	I	1b	When ScBusV is asserted, ScBusPassThru indicates whether the requested operation is just passing through CTRL, destined for aBIU.
SCBusAddr	I	11b	When ScBusV is asserted, ScBusAddr is the address of the state that is accessed.
SCBusData	I	7b	When ScBusV is asserted, and ScBusRW indicates a write, ScBusData contains the data to be written.
SCBusOp	I	1b	When ScBusV is asserted, ScBusOp indicates the requested operation.
SCBusFree	O	1b	SC bus available.
SCBusDone	O	1b	Pulsed for one cycle to indicate that the SC bus operation requested by aBIU has been completed.
CS (sBIU, CTRL) Bus			
CSBusAddr	O	8b	When CSBusValid is asserted, CsBusAddr is the address of the state that is accessed.
CSBusData	O	7b	When CSBusValid is asserted, and CsBusRW indicates a write, CsBusData contains the data to be written.
CSBusValid	O	1b	CS bus signals valid; when asserted, this indicates that there is valid data on the CS bus.
Short message signals			

Name	I/O	# bits	Description
ShTxComp	I	1b	sBIU short message compose I-bus request
CompFree	O	1b	Compose Buffer Available
RxLateAck	I	1b	sBIU asking that a late ack be returned to Arctic.
RxQEmpty	O	9b	Indicates whether a PcpR queue is empty. There are 8 different PcpR queues plus the Overflow queue.
MemQOut signals			
MemQOutOp	I	2b	MemQOut OpType (Passed to CTRL by sBIU to indicates whether the new MemQOut cmd is a short messages going out to Arctic network, a cmd going to the aBIU, a DMA Read or Send cmd, or a Data Motion cmd. See Section 5.6.1 for the encoding.)
MemQOut0Comp	I	1b	Requesting to put a cmd into MemQOut0.
MemQOut1Comp	I	1b	Requesting to put a cmd into MemQOut1.
(Data-motion) Inter-sRAM data transfer signals			
DMValid	I	1b	sBIU Data Motion I-bus Request
DMFree	O	1b	sBIU Data Motion Available
DMDone	O	1b	sBIU Data Motion done
J-Bus Signals			
JBusData	I	32b	J-bus data; correspond to the even (i.e. 0th) 32b-word.
JBusAddr	I	13b	J-bus address

A.1.6 aBIU and sBIU

Signal direction is described with respect to the aBIU, *i.e.* "I" means an input to the aBIU.

Name	I/O	# bits	Description
AcdOpV	I	1b	whether AcdOp is valid.
AcdOp	I	3b	sBIU convey shared memory related requests from sP
AcdApprAddr	I	12b	aSRAM address where data can be found/placed if the captured pending request is to be handled as Approval-w-Data.
AcdClsSync	I	1b	When AcdOpV, this indicates whether a synchronous ClsWrite enable is also requested.
AcdOpDone	O	1b	Notification from aBIU to sBIU that AcdOp is completed.
ClsUpdateReq	I	1b	Post sP clSRAM Update Request.
ClsSyncUpdate	I	1b	Indicates whether the requested update is a Synchronized one.
ClsUpdateDone	O	1b	Notification from aBIU to sBIU that clSRAM update is completed.

A.2 External Interfaces

A.2.1 NESchip and aP-Bus

Signal direction is described with respect to the NESchip, *i.e.* "I" means an input to the NESchip.

Name	I/O	# bits	Description
aPSysClk	I	1b	Sysclk; bus clock
aPAddr	I/O	32b	address
_aPNesBr	O	1b	Bus request
_aPNesBg	I	1b	Address bus grant
_aPAbb	I/O	1b	Address bus busy
_aPTs	I/O	1b	Transaction start; start of address transaction
_aPXats	I	1b	Extended Address Transfer start (_XATS), indicates direct-store operation, to I/O space. (May not be needed.)
aPTT	I/O	5b	Transfer Type
aPTc[0]	I	1b	Transaction code; This is not among the signals on Doral's processor card connector but is output from Mamba, and we really want to see this signal.
_aPTbst	I/O	1b	Tbst
aPTSiz	I/O	3b	TSiz
_aPGbl	I/O	1b	Global
_aPCi	I/O	1b	Cache Inhibit
_aPWt	I/O	1b	Write through
_aPAAck	I	1b	AAck; Address Acknowledgement
_aPL2Hit	O	1b	L2Hit; use this to indicate to memory controller that it is a transaction to NES Space.
_aPArtry	I/O	1b	ARTRY; Address Retry
_aPShd	I/O	1b	Shared
_aPDbg	I	1b	Data bus grant
_aPDbb	I/O	1b	Data bus busy
_aPTa	I/O	1b	Transaction acknowledge; slave indicating to master data is valid/consumed.
_aPHReset	I	1b	HRESET; Hard Reset
_aPSReset	I	1b	SRESET; Soft Reset
_aPInt	I	1b	INT; Interrupt

The following signals are on the 604 bus, but are not needed by the NES on the aP side:

Name	I/O	# bits	Description
aPAddrP	I/O	4b	address parity
aPTc[1:2]		2b	Transaction code; We believe this is not needed as they are not among the signals on Doral's processor card connector.
_APE		3b	Address parity error
_CSE		3b	Cache set element
_DPE		1b	Data Parity Error
_DRTRY		1b	Data Retry
_TEA		1b	Transfer Error Acknowledge
_DBDIS		1b	
_DBWO		1b	Data Bus Write Only; Not sure if this is needed; depends on the motherboard.
L2_INT		1b	
_Ckstp_in		1b	Checkstop input
_MCP		1b	Machine check interrupt
_SMI		1b	System Management interrupt

A.2.2 aSRAM and aP-Bus

Signal direction is described with respect to the aSRAM, *i.e.* "I" means an input to the aSRAM.

Name	I/O	# bits	Description
aPData	I/O	64b	data;

A.2.3 clSRAM and aP-Bus

Signal direction is described with respect to the clSRAM, *i.e.* "I" means an input to the clSRAM.

Name	I/O	# bits	Description
clsReadAddr	I	20b	address

A.2.4 NESchip and sP-Bus

Signal direction is described with respect to the NESchip, *i.e.* "I" means an input to the NESchip.

Name	I/O	# bits	Description
SPBAddr	I	32b	address
_SPBTS	I	1b	Transaction start; start of address transaction
SPBTT	I	5b	Transfer Type
_SPBTBst	I	1b	Tbst; Transfer Burst
SPBTSiz	I	3b	TSiz; Transfer Size
_SPBAAck	O	1b	AAck; Address Ack
_SPBARtry	O	1b	ARTRY; Address Retry
_SPBShd	O	1b	Shared
_sPDbg	I	1b	Data bus grant
_SPBDBB	I	1b	Data bus busy
_SPBTA	O	1b	Data Ack; slave indicating to master data is valid/consumed.
_SPBHReset	I	1b	HRESET; Hard Reset
_SPBSReset	I	1b	SRESET; Soft Reset
_SPBInt	O	1b	INT; Interrupts sP

The following signals are on the 604 bus, but are not needed by the NES on the sP side:

Name	I/O	# bits	Description
sPAddrP	IO	4b	address parity; sBIU is never a master on the sP bus. sBIU also does not check for parity.
_BR	O	1b	Bus request; sBIU is never a master on the sP bus
_BG	I	1b	Address bus grant; sBIU is never a master on the sP bus
_ABB	IO	1b	Address bus busy; sBIU is never a master on the sP bus
_GBL	IO	1b	Global
_CI	IO	1b	Cache Inhibit
_WT	IO	1b	Write through
TC	IO	3b	Transaction code
_DBWO	I	1b	Data Bus Write Only; Not supported by the MPC105.
_XATS	I	1b	Extended Address Transfer start, indicates direct-store operation, to I/O space. sP does not use any direct-store operations.
_APE		3b	Address parity error
_CSE		3b	Cache set element
_DPE		1b	Data Parity Error
_DRTRY		1b	Data Retry
_TEA		1b	Transfer Error Acknowledge
_DBDIS		1b	
L2_INT		1b	
_Ckstp_in		1b	Checkstop input (wrt 604)
_MCP		1b	Machine check interrupt
_SMI		1b	System Management interrupt
_Ckstp_out		1b	Checkstop output (wrt 604)
Run		1b	Run
Halted		1b	Halted

A.2.5 sSRAM and sP-Bus

Signal direction is described with respect to the sSRAM, *i.e.* "I" means an input to the sSRAM.

Name	I/O	# bits	Description
sPData	I/O	64b	data;

A.2.6 clSRAM and sP-Bus

Signal direction is described with respect to the clSRAM, *i.e.* "I" means an input to the clSRAM.

Name	I/O	# bits	Description
clsUpdateAddr	I	20b	clSRAM address; part of address on sP-port.
clsUpdateData	I	4b	clSRAM data; part of address on sP-port

A.2.7 NESchip and clSRAM

Signal direction is described with respect to the NESchip, *i.e.* "I" means an input to the NESchip.

Name	I/O	# bits	Description
clsData	I	4b	cache-line state data
clsRSel	O	2b	cache-line state R/W
clsCS	O	2b	cache-line state chip select
clsaPEn	O	1b	aP bus address tri-state enable
clssPAddrOutEn	O	1b	sP bus HAL address register output enable
clssPAddrInEn	O	1b	sP bus HAL address register input enable

B Command and Configuration

Control and configuration of the NES is achieved through a Cmd and Config Space that is accessible by both the sP and the aP, over the sP-bus and the aP-bus respectively. This interface is (almost) identical on both buses. Most of the control and command is achieved by writing NES Configuration State (Config State) directly. A few special cases require further coordination among different parts of the NES. Our design differentiates the two, with those in the first category supported via Config State Access capability, while those in the latter category via the ImmdCmd mechanism.

Table 71 lists all the control and configuration functionalities, and indicate whether they are achieved via Config State Access ("Conf") or ImmdCmd, indicated with "Cmd-X", where X is either 'R' or 'C'. ImmdCmd have special support from the NES hardware. Those under the Cmd-R category are guaranteed to complete in a fixed number of clock cycles after they are received by the NES, *i.e.* there is real-time guarantee of time-to-completion. Cmd-C commands do not have any real-time guarantee of completion. Instead, completion is indicated by some state bit in the NES, sBIU or aBIU MSRs.

Command Category	Description
General	
Cmd-R	NES reset
Msg Passing Related, Arctic Oriented	
Conf	NES Arctic Interface Tx enable; starts generating idle.
Conf	NES Arctic Interface Rx enable; starts checking for idle pattern and incoming packets.
Conf	NES Arctic Interface Tx disable; stops generating idle pattern; non-recoverable.
Conf	NES Arctic Interface Rx disable; stops checking for idle pattern.
Msg Passing Related, NES oriented	
Cmd-R	NES clear TxFIFO; This is an asynchronous clear.
Cmd-R	NES clears RxFIFO; This is an asynchronous clear.
Conf	NES Tx master enable; enables pushing packets into TxFIFO.
Conf	NES Rx master enable; enables taking packets from RxFIFO.
Cmd-C	NES Tx master disable; recoverable disabling, i.e. do not stop in the middle of a packet transmission; Completion detected by polling NES MSR.
Cmd-C	NES Rx master disable; stop moving in-coming packet out of Rx-FIFO; recoverable, i.e. do not stop while in the middle of moving a packet out of RxFIFO. Completion detected by polling NES MSR.
Cmd-C	Arctic Ack. Use a count based scheme to indicate number of Late Acks to return to Arctic router attached to the NES.
Conf	Read aBIU's One-poll-addr and AccessedOnce registers.
Conf	Write aBIU's One-poll-addr and AccessedOnce registers.
Conf	Read sBIU's One-poll-addr and AccessedOnce registers.
Conf	Write sBIU's One-poll-addr and AccessedOnce registers.

Table 70a: Control and configuration of the NES is achieved using the commands listed in this table.

Command Category	Description
Msg Passing Related, Per queue specific	
Conf	PcpQ-X Reclaim enable.
Conf	PcpQ-X Reclaim disable.
Conf	TxQ-X enable. Enable pushing packets into TxFIFO.
Cmd-C	TxQ-X disable; Q status enable bit only cleared when the queue is actually inactive. This matters when "disable" command is given while a packet is being transmitted from the queue. For PasT queues, disabling of queue also has to ensure that the BIU concerned does not have outstanding operation on the queue.
Conf	RxQ-X enable. Enable pulling packets from RxFIFO. Note that the enable can be cleared by the NES automatically, if the queue has Early Ack enabled, and it becomes full.
Cmd-C	RxQ-X disable; Q status enable bit only cleared when the queue is actually inactive. This matters when "disable" command is given while a packet is being received into the queue. For VasR queues, disabling of queue also has to ensure that the BIU concerned does not have outstanding operation on the queue.
Conf	Resident Rx table entry read.
Conf	Resident Rx table entry write.
Msg passing, per queue state access	
Conf	Q state (excluding status) read.
Conf	Q state (excluding status) write.
Conf	Q status register read.
Conf	Q status register write; Note that higher level software has to coordinate this write, so that it does not happen while packets are moving in or out of the queue. This can be achieved by first shutting down the queue, which requires first issuing a disable command, and then polling till it is confirmed that the queue has been disabled.
DMA control	
Conf	Dma Snd enable.
Cmd-C	Dma Snd disable; stop after finishing the current DMA Snd cmd. Discard all the remaining commands, and ignore new DMA Snd cmd.

Table 70b: Control and configuration of the NES is achieved using the commands listed in this table.

Command Category	Description
Shared memory related	
Conf	enable Global Memory Space.
Conf	disable Global Memory Space; higher level software must ensure that no global memory space transaction is in progress.
Conf	enable Local Global Space.
Conf	disable Local Global Space; higher level software must ensure that no local global memory transaction is in progress.
Conf	enable HAL check.
Conf	disable HAL check; higher level software must ensure that no local global memory transaction is in progress.
Cmd-R	approval for pending SM op.
Cmd-R	aBIU discard op pending approval. (Used when Kill is turned into a RWITM).
Cmd-R	Lock aBIU's ApprovalReg. (Unlock with a "aBIU discard op pending approval")
Cmd-R	Lock aBIU's Notification capability.
Cmd-R	Unlock aBIU's Notification capability.
Cmd-R	cls write.
Interrupt/exception related	
Conf	read aP/sP interrupt mask.
Conf	write aP/sP interrupt masks.
Conf	read NES ErrReg.
Conf	write NES ErrReg.
Conf	read PcpR IntReg.
Conf	write PcpR IntReg.
Conf	read VasR IntReg.
Conf	write VasR IntReg.
Conf	read NES MSR-enable.
Conf	write NES MSR-enable; not all bits are writeable. NES hardware has to enforce the requirement that certain MSR-enable bits are not directly changeable via Config Space write of MSR-enable.

Table 70c: Control and configuration of the NES is achieved using the commands listed in this table.

Command Category	Description
General	
Cmd-R	NES reset
Msg Passing Related, NES oriented	
Cmd-R	NES clear TxFIFO; This is an asynchronous clear.
Cmd-R	NES clears RxFIFO; This is an asynchronous clear.
Cmd-C	NES Tx master disable; recoverable disabling, i.e. do not stop in the middle of a packet transmission; Completion detected by polling NES MSR.
Cmd-C	NES Rx master disable; stop moving in-coming packet out of Rx-FIFO; recoverable, i.e. do not stop while in the middle of moving a packet out of RxFIFO. Completion detected by polling NES MSR.
Msg Passing Related, Per queue specific	
Cmd-C	TxQ-X disable; Q status enable bit only cleared when the queue is actually inactive. This matters when "disable" command is given while a packet is being transmitted from the queue. For PasT queues, disabling of queue also has to ensure that the BIU concerned does not have outstanding operation on the queue.
Cmd-C	RxQ-X disable; Q status enable bit only cleared when the queue is actually inactive. This matters when "disable" command is given while a packet is being received into the queue. For VasR queues, disabling of queue also has to ensure that the BIU concerned does not have outstanding operation on the queue.
DMA control	
Cmd-C	Dma Snd disable; stop after finishing the current DMA Snd cmd. Discard all the remaining commands, and ignore new DMA Snd cmd.
Shared memory related	
Cmd-R	approval for pending SM op.
Cmd-R	aBIU discard op pending approval. (Used when Kill is turned into a RWITM.)
Cmd-R	Lock aBIU's ApprovalReg. (Unlock with a "aBIU discard op pending approval")
Cmd-R	Lock aBIU's Notification capability.
Cmd-R	Unlock aBIU's Notification capability.
Cmd-R	cls write.
Cmd-R	Clear CTRL's MemQOut-X Request Category queue.

Table 71: This table lists all the ImmdCmds supported.

B.1 Higher Level Command and Control Actions

B.1.1 Booting up the NES

This is a sketch of the sequence, in chronological order, for booting up the NES, possibly with rebooting of the Arctic Network. The host may or may not be rebooted at the same time. It assumes that after a reset, the NES come up with MemQIn, MemQOut and the ApprovalReg mechanism functioning.

1. If the Arctic network is not rebooted, the Arctic input port connected to the Rx of the NES that is being rebooted has to be shut down first, so that it will not be expecting Idle Patterns.
2. *Reset NES* . All message queues come up disabled. All error registers cleared upon reset. MemQIn, MemQOut and ApprovalReg mechanism comes up working after reset. There is real time guarantee of how long the NES takes to reset. The sP comes up polling MemQIn, ready for startup code loading request from the aP. The aP will essentially load program into the sP by writing it into sSRAM and then informing the sP to load the code. It may take several installments to load all the code into the sP.
3. *NES clear TxFIFO*
4. *NES Arctic Interface Tx enable*
5. If rebooting Arctic network as well, enable Tx on all Arctic switches after the Arctic switches have been reset.
6. Barrier synchronize across the entire machine.
7. *NES clear RxFIFO*
8. *NES Arctic Interface Rx enable*
9. If rebooting Arctic network as well, enable Rx on all Arctic switches. Otherwise, just enable Rx on the Arctic input port connected to the NES that is being rebooted.
10. Barrier synchronize across the entire machine. During this time, each site can initialize its queues.
11. *NES Rx master enable*
12. *NES Tx master enable*

B.1.2 Handling RxQ Overflow

The following is the sequence of events following an RxQ overflow. Note that only RxQ that uses Early Ack can overflow.

1. When an RxQ that is configured to use Early Ack overflows, CTRL automatically disables the RxQ. It then channels the in-coming packet to the Overflow Queue.

   Disabling the queue means that no more in-coming packet will go into the RxQ from RxFIFO, but software can continue to read messages out of it, and update the User Control Registers.
2. During the time when an RxQ is disabled, all packets destined for it are placed into the Overflow Queue, which is handled by the sP. The sP will probably place these packets into either sP dSRAM or aP dSRAM.

3. Upon first detecting overflow of an RxQ, the sP software may also choose to request all sender queues to stop transmitting. Each sP that receives such a "stop-transmit" request will disable the appropriate transmit queue(s) by using the *TxQ-X disable* command.

   In this disabled state, a TxQ will not push packets into TxFIFO (effectively into the Arctic network), but software can continue to compose messages into it if there is space. Software will also detect back-pressure when the TxQ runs out of buffer space.

4. Periodically, the sP at the site that has the overflowed RxQ polls the pointers of the RxQ to find if the queue has freed up space. When buffer space becomes available, depending on software policy, the sP may place the overflowed messages back into NES SRAM, and update the RxQ's P-ptr.

5. When there is no more overflow messages in dRAM, the sP will re-enable the RxQ by using the *TxQ-X enable* command. If it has requested disabling of sender queues, it will now multicast a re-enable sender queue message. Upon receiving such a request, an sP uses the *TxQ-X enable* command to re-enable the TxQ concerned.

6. An alternate policy is to have overflowed messages received as though the RxQ were non-resident. This involves having the aP VMM to remap message buffer and control space appropriately when the resident RxQ is empty.

   Once the remapping is done, the resident RxQ can be re-enabled, so that new packets from Arctic can be channelled into it. An implementation may, however, still choose to delay the enabling of the RxQ until the dRAM based queue containing the overflowed messages is empty. This avoid the situation where the re-enabled queue starts overflowing again before the old overflowed messages have been completely serviced. All these are software choices.

   comment: *A helpful feature is to have the NES provide some form of notification, either via interrupt or MemQIn, when the RxQ becomes empty. It is unlikely that we will support this.*

B.1.3 Resident/Non-resident switching PcpT and PcpR Queues

1. *disable TxQ* or *disable RxQ* . Poll relevant CTRL SysReg repeatedly until queue has been disabled. Also make sure that no processor is composing into or receiving from the queues (*e.g.* if aP is the user, it has to be interrupted before queue switching is done; the sP cannot unilaterally do it.).

2. Poll QEnable register to make sure that the queue(s) concerned are disabled. Disabled means NES hardware will no longer be moving packets in/out of the queues.

3. Make sure no Q State update, either initiated by software or by xBIU to CTRL, or from CTRL to xBIU are in progress. The latter two cases apply only to Pas/Vas queues. To deal with them, compare the xBIU and CTRL copies to make sure that no master-to-shadow pointer copy update is happening. The technique to make sure that no software initiated update is in progress relies on the FIFO ordering of state updates and reads. By requesting a read from the bus with the processor that uses the queue that is shut down, we know that no previous updates are outstanding when the read returns.

4. Higher level software has to ensure that no messages are being written into the queues simultaneously. This only means that one must not have the aP using the resident queues at the

same time when the sP is moving the queue. It is okay for interrupted aP/sP software to be composing/reading message half way, as long as appropriate address mapping is done so that the switch between Resident and Non-resident queues is transparent to the interrupted software.

5. Swap the contents of the queues between NES SRAM and aP dRAM or sP dRAM. Do this using SRAM Reads, SRAM Writes, and possibly MemQOut mechanism (if performing this from the sP).

6. Swap the contents of Q State, using Config Q State Space accesses.

7. To make sure that Q State Space Writes have completed, read out Q State from the same bus that the writes are done from. To make sure that SRAM Writes have completed, just read the last written value. Because reads/writes are handled in order, the fact that the read has occurred is sufficient to indicate completion.

8. Finally, *enable TxQ* or *enable RxQ* .

Notice that the design does not allow the sP to swap a queue between resident and non-resident without making sure that aP is not writing to the queue at that very moment. It is too difficult and risky to lock the queue with NES hardware mechanism. In general, address mapping needs to be changed anyway, so the aP has to be interrupted.

B.1.4 Resident/Non-resident switching PasT and VasR Queues

This is similar to swapping Pcp queues. Because the *disabling TxQ-X* and *disabling RxQ-X* commands *do not* make sure that there is no outstanding pointer update operation from aBIU or sBIU to CTRL and vice-versa, the switching code has to read the pointers from both the xBIU and CTRL, compare them to make sure they agree which is an indication that there is no outstanding update.

It is also necessary to context switch the One-poll-addr register, and the register containing the boolean Accessed in the appropriate BIU.

B.1.5 MemQOut Frozen due to Failed NES Mastered aP-bus Op

When the aBIU performs an NES Mastered aP-bus Op, it tries it n times. If the transaction is retried more than n times, the aBIU stops trying it, the two MemQOut's are frozen, and an error bit in NES MSR ErrReg is set. A negative acknowledgement for the request is also sent back to the sP, which can then take appropriate action. Often time, that requires changing some of the commands that are already in the MemQOut queues. This involves:

1. sP reads, via Cmd Space/Config Q State access, the relevent MemQOut ptrs.

2. sP poll aBIU and/or sBIU to make sure that no update to MemQOut is in-progress, before proceeding to the next step.

3. Read out queue content of MemQOut, using direct SRAM Reads.

4. Write MemQOut ptrs to empty the MemQOut(s) that need changes to the commands that have already been issued.

Because CTRL keeps a queue of the entry types for commands that are already in MemQOut queues, these have to be cleared too if the MemQOut queue is "flushed". Use *Clear CTRL's MemQOut Request Category queue.*

5. Commands that are re-issued into the queue goes through the usual ShTx or Special ShTx interface for short message composition into MemQOut queue.

B.1.6 Changing Destination Translation Table Entry

Software on the sP or aP has to coordinate such a change. Generally, it only makes sense to change a destination translation table entry when all previously composed messages have been sent out. It is possible that the change should take effect immediately, though that opens up the possibility of many race situations that higher level software has to take care of.

1. If preventing race between an on-going translation and the change is desired, the TxQs concerned can be disabled (using *disable TxQ*) before the change is made.
2. If emptying the queue first is desired, the sP can poll periodically to determine when the TxQs are empty. The code requesting the Destination Translation Table entry change will have to be blocked during this time to prevent it from composing more messages into the qeueus.
3. The change itself is simple, requiring only a write to the appropriate location in the SRAM location where the table entry resides.
4. Poll sBIU or aBIU MSR state to make sure that the write has completed all the way to SRAM. (This is only necessary if using aSRAM since accesses to sSRAM complete at the same time as the sP-bus operation that causes it.)

B.1.7 Multicast Using ShTx Mechanism

The ShTx with data tag-on facility is useful for efficiently implementing multi-cast. We assume that the software using a ShTx queue already knows the queue's C-ptr and P-ptr as these are needed to determine if there is space to compose a short message.

1. Check that there is sufficient space for the multicast.
2. If there is tag-on data, make sure those are already written to SRAM.
3. Write the multicast Short messages using SRAM write, directly to where the ShTx buffers are in SRAM.
4. Make sure write have been done. (SYNC; in the unusual case where the further SRAM is used, the application code needs to make sure that the writes have completed. This can be done by reading from the further SRAM after the SYNC; the NES implementation guarantees FIFO ordering, so by the time the read completes, the writes must have completed.
5. Update ShTxQ's P-ptr, using user level QPtr Space update. This must be done to the sBIU/aBIU copy of the P-ptr *and* CTRL's copy.

   May need to SYNC after this. We assume that there is no need to check that the QPtr state update is already done. This is based on the assumption that all Config State update occur in FIFO order, i.e. in the sBIU/aBIU, there is no overtaking of Config State update based on say whether it is user initiated or sBIU or aBIU hardware initiated.

Space used for tag-on data cannot be reused until it is established that the message has been transmitted into the Arctic network. The user code can detect this by polling the C-ptr, and wait for the C-ptr to advance past the buffer that contains the tag-on short message.

References

[1] B. S. Ang. *Design and Implementation of a Multi-purpose Cluster System Network Interface Unit*. PhD thesis, Massachussetts Institute of Technology, Feb. 1999. (Also available as CSG Memo 420.).

[2] B. S. Ang. Shared Memory Support on the StarT-Voyager Cluster System. CSG Memo 388, MIT Laboratory for Computer Science, Jan. 1999.

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

[4] C. E. Leiserson. Fat-trees: Universal Networks for Hardware-efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10), Oct. 1985.

[5] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[6] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman Publishers, Inc., San Francisco, CA, second edition, May 1994.

[7] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Proceedings of Supercomputing '93, Portland, Oregon*, pages 624–635, Nov. 1993.

[8] D. L. Rosenband. Design of the control logic for StarT-Voyager. Master's thesis, Massachusetts Institute of Technology, May 1998.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture Conference Proceedings, Gold Coast, Australia*, pages 256 – 266, 1992.