1 Runway Bus Interface Block

1.1 Overview

The Runway Bus Interface Block (RBIB) in Astro is responsible for connecting up to four PCXW processor caches, Astro's I/O controller, and Astro's memory controller together in a manner that preserves cache coherency and transaction ordering. The major tasks performed by the RBIB are:

- Route Runway transactions to the memory controller or the I/O controller or both
- Route DMA and peer-to-peer I/O transactions from the I/O controller to Runway
- Route read return transactions from the memory controller to Runway
- Track transactions that are pending in the memory controller
- Track coherency status of pending coherent transactions
- Detect and resolve memory ordering conflicts for transactions pending in the memory controller
- Perform system level flow control of coherency checks, memory transaction buffers, and I/O transaction buffers

The following documents provide useful background for understanding some of the topics discussed in the rest of this chapter:

- *Runway Bus Specification Rev 1.3*
- *PA-RISC 2.0 Architecture*

1.2 Runway vs. Astro Bit and Byte Ordering

The Runway Bus uses the convention that the most significant bit of a data field is bit 0 and the most significant byte of a data field is byte 0. (i.e. Big-endian bit ordering and big-endian byte ordering.) The Runway pads in Astro follow the same convention to make it easy for board designers to connect Astro to the Runway Bus.

	ADDR_DATA[0:63]							
Bit	Most Sig Bit 0							Least Sig Bit 63
Byte	0	1	2	3	4	5	6	7

Table 1: Runway Bus Byte Ordering (Non-Turbo-Mode)

		ADDR_DATA[0:63]							
Time	**Bit**	Most Sig Bit 0							Least Sig Bit 63
↓	**Byte**	0	1	2	3	4	5	6	7
	Byte	8	9	10	11	12	13	14	15

Table 2: Runway Bus Byte Ordering (Turbo-Mode)

The PCI Bus, AGP Bus, Astro internal busses, and Astro registers use the convention that the least significant bit of a data field is bit 0 and the least significant byte of a data field is byte 0. (i.e. Little-endian bit ordering and little-endian byte ordering.)

	REGISTER[63:0]							
Bit	Most Sig Bit 63							Least Sig Bit 0
Byte	7	6	5	4	3	2	1	0

Table 3: Astro Byte Ordering (64-bit Internal Register)

	addr_data[127:0]															
Bit	Most Sig Bit 127															Least Sig Bit 0
Byte	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 4: Astro Byte Ordering (128-bit Internal Bus)

Astro converts the big-endian Runway bus to little-endian at the Runway pads in an address invariant manner. (Or in DEW terminology, Bytes-is-bytes.) This means that the address of each byte is preserved. (Byte N on the Runway bus is mapped to Byte N on PCI, AGP, Astro internal busses, and Astro internal registers.) Within any byte, bit n on Runway is mapped to bit 7-n in Astro. This strategy has the following properties:

- If a processor is in big-endian mode, then any data larger than the size of a byte (i.e. half-word, word, or double-word) will appear byte-swapped to a little-endian PCI device, AGP device, or Astro internal register and vice-versa. For example, assume a 64-bit register in Astro contains the value 0x0123456789ABCDEF. If a processor in big-endian mode reads the register, then the processor would read 0xEFCDAB8967452301. The processor must do an 8-byte byte-swap to interpret the data correctly.

- If a processor is in little-endian mode, then data of all sizes will appear the same to a little-endian PCI device, AGP device, or Astro internal register and vice-versa. For example, assume a 64-bit register in Astro contains the value 0x0123456789ABCDEF. If a processor in big-endian mode reads the register, then the processor would read 0x0123456789ABCDEF.

This method is also consistent with the way PCI is implemented on HP-PA platforms with the Dino GSC to PCI bus bridge.

Note for Astro design and verification engineers: Due to the address invariant mapping of the Runway pads, the 40-bit physical address field and the 10-bit virtual index field that are found on the Runway **ADDR_DATA[0:63]** lines during address cycles will be mapped to the following bit positions on Astro internal busses (assume that the internal bus is named **addr_data[127:0]**):

- **Address[39:0]** = **addr_data[31:24,39:32,47:40,55:48,63:56]**
- **Virt_Index[9:0]** = **addr_data[19:18,11:8,23:20]**

1.3 Basic Transaction Flow

The RBIB routes data between the following three blocks:

- Runway Interface (which acts as an interface to the processors)
- Coherency Map (which acts as an interface to the memory controller)
- I/O controller (which acts as an interface to the Rope Bus and PCI host bridges)

The three major blocks in the RBIB are connected together by two internal busses, the Runway Input Bus and the Runway Output Bus, as show in **Figure 1: Basic Runway Bus Interface Block Diagram**. The Runway Input Bus routes in-bound Runway transactions from the Runway interface to the coherency map or the I/O controller or both. The Runway Output Bus routes out-bound transactions from the I/O controller to the Runway interface or from the coherency map to the Runway interface.

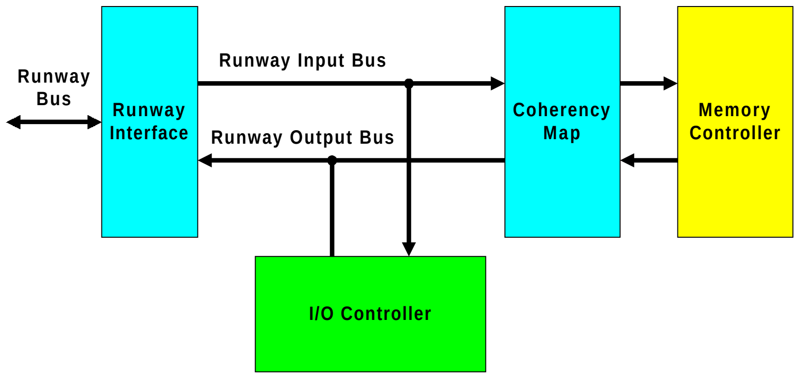

Figure 1: Basic Runway Bus Interface Block Diagram

Table 5 defines all possible transactions that take place in the RBIB. (Note: A processor-to-processor cache-to-cache copy is equivalent to a processor write.)

Transaction	**Description**
Processor Write	Processor write to memory space.
Processor Non-Coherent Read	Processor non-coherent read from memory space.
Processor Coherent Read	Processor coherent read from memory space.
Processor Read Return	Processor memory space read return from memory controller.
Processor I/O Write	Processor write to I/O space.
Processor I/O Read	Processor read to I/O space.
Processor I/O Read Return	Processor I/O space read return from I/O controller.
Processor Flush/Purge/Sync	Processor flush, purge, or sync transaction.
Processor Purge/Sync Done	Done transaction for a processor purge or sync transaction.
DMA Write	I/O controller write to memory space.
DMA Coherent Read	I/O controller coherent read from memory space.
DMA Read Return	I/O controller memory space read return from memory controller.
DMA Cache-to-Cache Copy	I/O controller memory space read return from processor.
I/O Write to Processor	I/O controller write to I/O space in processor.
I/O Read from Processor	I/O controller read from I/O space in processor.
I/O Read Return from Processor	I/O controller I/O space read return from processor.
Peer-to-Peer I/O Write	I/O controller write to I/O space.
Peer-to-Peer I/O Read	I/O controller read from I/O space.
Peer-to-Peer I/O Read Return	I/O controller I/O space read return from I/O controller.

Table 5: Runway Bus Interface Block Transactions

One transaction that appears to be missing from **Table 5** is an I/O controller-to-processor cache-to-cache copy. This transaction is not generated by the I/O controller. Instead, the I/O controller will write all private-dirty cache lines that are hit by a coherent read back to memory before sending the coherency response for the read.

1.3.1 Processor Write, Processor Non-coherent Read Transaction Flow

This transaction starts in the processor and goes to the Runway interface via the Runway bus. The Runway interface sends the transaction to the coherency map via the Runway Input Bus. The coherency map sends the transaction to the memory controller.

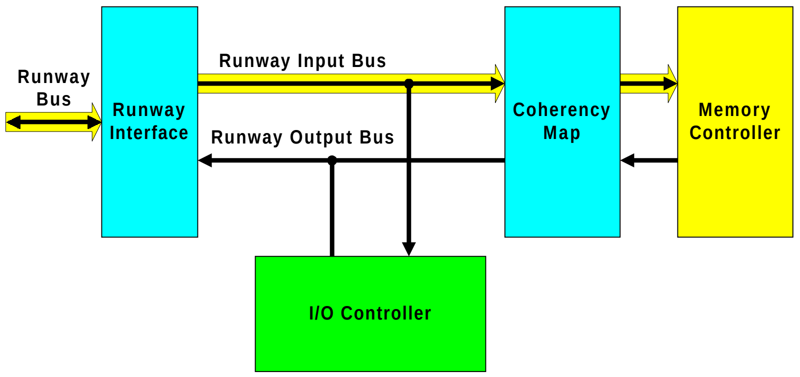

Figure 2: Processor Write, Processor Non-coherent Read Transaction Flow

1.3.2 Processor Coherent Read Transaction Flow

This transaction starts in the processor and goes the to Runway interface via the Runway bus. The Runway interface sends the transaction to the I/O controller and coherency map via the Runway Input Bus. The I/O controller will snoop the transaction. The coherency map sends the transaction to the memory controller.

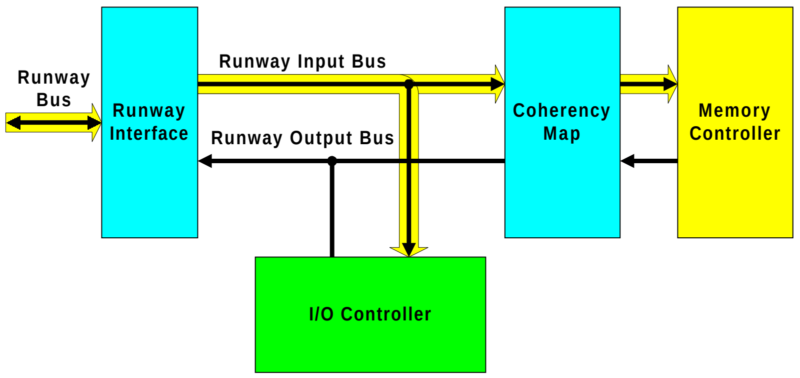

Figure 3: Processor Coherent Read Transaction Flow

1.3.3 Processor Read Return Transaction Flow

This transaction starts in the memory controller and goes to the coherency map. The coherency map send the transaction to the Runway interface via the Runway Output Bus. The Runway interface sends the transaction to the processor via the Runway bus.

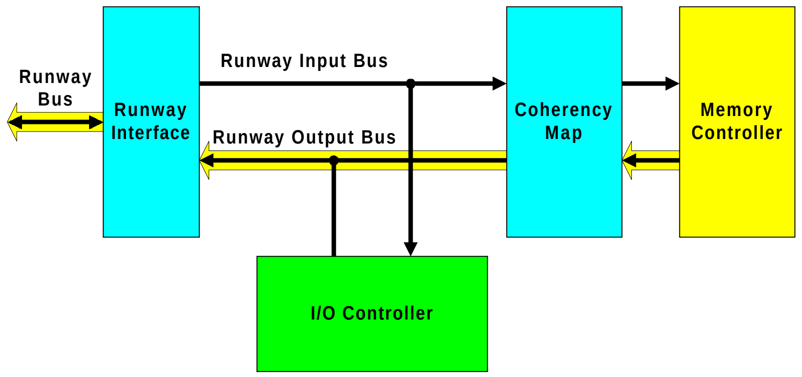

Figure 4: Processor Read Return Transaction Flow

1.3.4 Processor I/O Write, Processor I/O Read, I/O Read Return from Processor Transaction Flow

This transaction starts in the processor and goes the to Runway interface via the Runway bus. The Runway interface sends the transaction to the I/O controller via the Runway Input Bus.

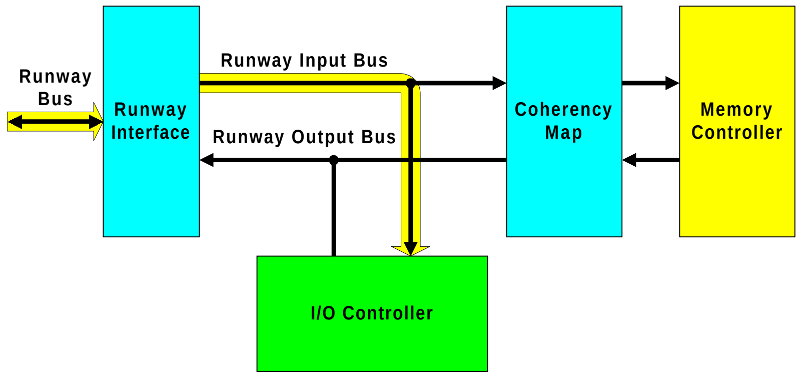

Figure 5: Processor I/O Write, Processor I/O Read, I/O Read Return from Processor Transaction Flow

1.3.5 Processor I/O Read Return, I/O Write to Processor, I/O Read from Processor Transaction Flow

This transaction starts in the I/O controller and goes to the Runway interface via the Runway Output Bus. The Runway interface sends the transaction to the processor via the Runway bus.

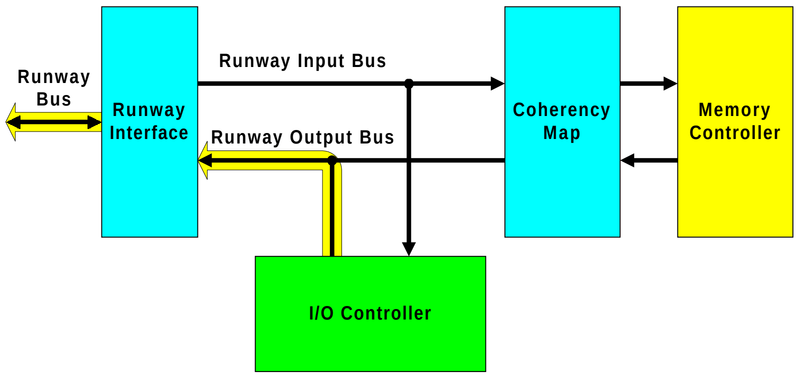

Figure 6: Processor I/O Read Return, I/O Write to Processor, I/O Read from Processor Transaction Flow

1.3.6 Processor Flush/Purge/Sync Transaction Flow

This transaction starts in the processor and goes the to Runway interface via the Runway bus. The Runway interface sends the transaction to the I/O controller and coherency map via the Runway Input Bus. The I/O controller will snoop the transaction.

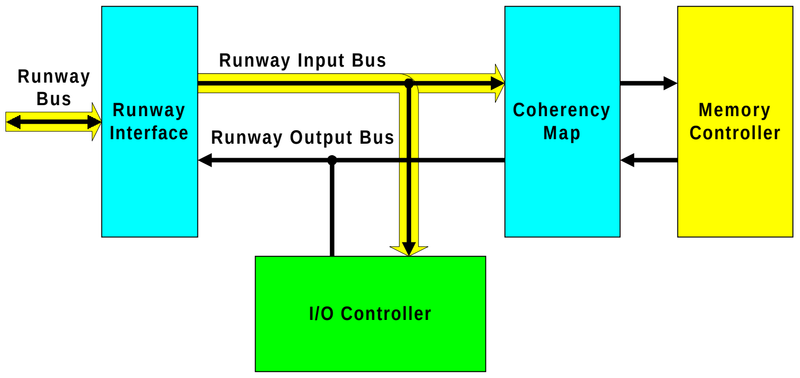

Figure 7: Processor Flush/Purge/Sync Transaction Flow

1.3.7 Processor Purge/Sync Done Transaction Flow

This transaction starts in the coherency map and goes to the Runway interface via the Runway Output Bus. The Runway interface sends the transaction to the processor via the Runway bus.

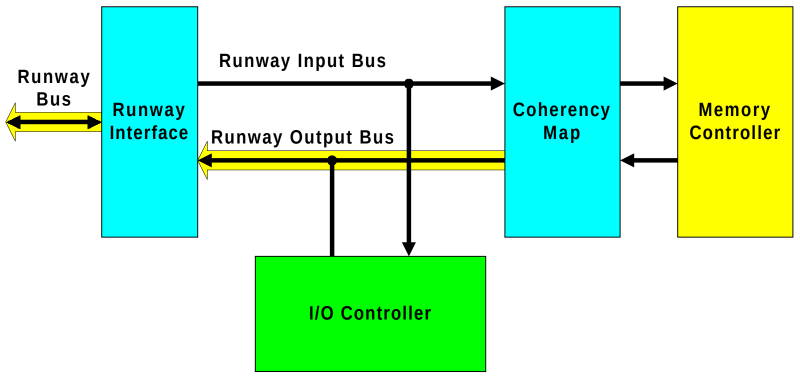

Figure 8: Processor Purge/Sync Done Transaction Flow

1.3.8 DMA Write Transaction Flow

This transaction starts in the I/O controller and goes the to the Runway interface via the Runway Output Bus. The Runway interface reflects the transaction on the Runway bus and sends it to the coherency map via the Runway Input Bus. The coherency map sends the transaction to the memory controller.

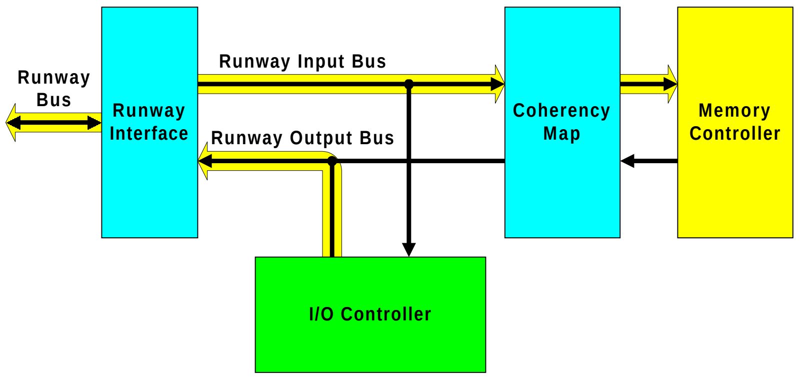

Figure 9: DMA Write Transaction Flow

1.3.9 DMA Coherent Read Transaction Flow

This transaction starts in the I/O controller and goes to the Runway interface via the Runway Output Bus. The Runway interface reflects the transaction on the Runway bus and sends it to the I/O controller and coherency map via the Runway Input Bus. The transaction is reflected back to the I/O controller so it can snoop the transaction and know when to take cache line ownership. The coherency map sends the transaction to the memory controller.

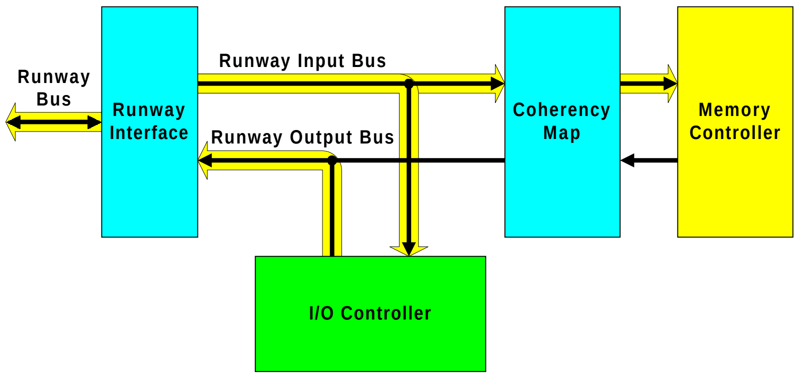

Figure 10: DMA Coherent Read Transaction Flow

1.3.10 DMA Read Return Transaction Flow

This transaction starts in the memory controller and goes to the coherency map. The coherency map send the transaction to the Runway interface via the Runway Output Bus. The Runway interface reflects the transaction on the Runway bus and sends it to the I/O controller via the Runway Input Bus.

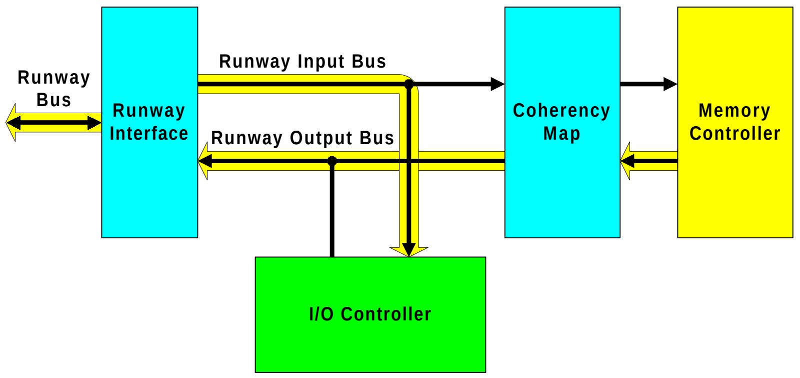

Figure 11: DMA Read Return Transaction Flow

1.3.11 DMA Cache-to-Cache Copy Transaction Flow

This transaction starts in the processor and goes the to Runway interface via the Runway bus. The Runway interface sends the transaction to the I/O controller as a DMA read return and to the coherency map as a write via the Runway Input Bus. The coherency map sends the transaction to the memory controller.

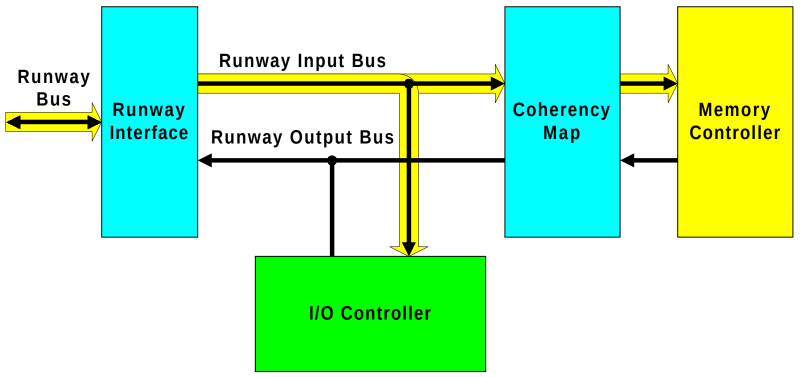

Figure 12: DMA Cache-to-Cache Copy Transaction Flow

1.3.12 Peer-to-Peer I/O Write, Peer-to-Peer I/O Read, Peer-to-Peer I/O Read Return Transaction Flow

This transaction starts in the I/O controller and goes to the Runway interface via the Runway Output Bus. The Runway interface reflects the transaction on the Runway bus and sends it back to the I/O controller via the Runway Input Bus.

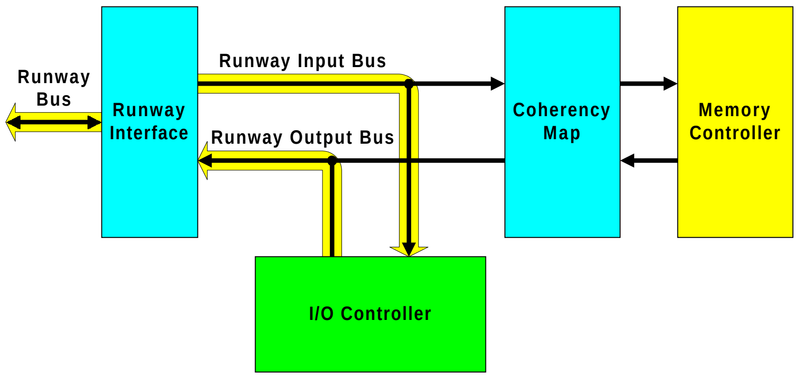

Figure 13: Peer-to-Peer I/O Write, Peer-to-Peer I/O Read, Peer-to-Peer I/O Read Return Transaction Flow

1.4 Preserving Memory Coherency Ordering

This section describes the ordering rules for memory transactions. (IO read and write transactions are not covered by this section. See **Appendix G: Memory Ordering Model** in *PA-RISC 2.0 Architecture* for more information.)

Astro preserves the order of Runway memory transactions. This does not mean that the physical order of Runway transactions is the same as the physical order of transactions to the SDRAM. The memory system in Astro (i.e. the coherency map and memory controller) is allowed to reorder Runway transactions going to memory to reduce read latency or improve memory bandwidth provided the reordering of memory transactions does not violate any Runway ordering rules. As long as none of the Runway ordering rules are violated, then a cache agent on Runway can not tell if the memory system reordered Runway transactions going to SDRAM or not.

1.4.1 Runway Transaction Ordering Rules

Runway transactions must obey the following ordering rules:

1. All *coherent transactions without coherency checks* (e.g. WRITE_BACK, FLUSH_BACK, and C2C_WRITE) to the same cache line are ordered with respect to each other in the order they are issued on the Runway bus.

2. All *coherent transactions with coherency checks* (e.g. READ_SHAR_OR_PRIV, READ_CURRENT, READ_PRIV, DFLUSH, IFLUSH, DPURGE, DPURGE_ALLOC, CACHE_SYNC, and DMA_SYNC) to the same cache line are ordered with respect to each other in the order they are issued on the Runway bus.

3. All *coherent transactions without coherency checks* issued on the Runway bus before the completion of the coherency response phase for a *coherent transaction with coherency checks* to the same cache line are logically ordered in front of the *coherent transaction with coherency checks*.

4. There are no ordering restrictions on *non-coherent transactions* (e.g. READ, WRITE). Software is responsible for managing any ordering restrictions through the use of strongly ordered sync transactions.

Rules #1, #2, and #4 are straightforward. Rule #3 can benefit from an example. Assume that two processors, named **A** and **B**, are on a Runway bus, and that **A** issues a coherent read of a cache line that is private dirty in **B**'s cache. Assume that just before **A** reads the cache line, **B** decides to move the cache line from cache back to memory with a write-back transaction. Under these conditions, one possible sequence of Runway transactions is:

State	Master	Transaction	Coh Response	Notes
0	A	READ_PRIV		A starts private read
1			A:OK	A signs-off on READ_PRIV in state #0
2	B	WRITE_BACK		B starts coherent write
3	B	DATA0		Data for WRITE_BACK in state #2
4	B	DATA1		Data for WRITE_BACK in state #2
5	B	DATA2		Data for WRITE_BACK in state #2
6	B	DATA3		Data for WRITE_BACK in state #2
7			B:OK	B signs-off on READ_PRIV in state #0
8	Host	DATA0		Read return for READ_PRIV in state #0
9	Host	DATA1		Read return for READ_PRIV in state #0
10	Host	DATA2		Read return for READ_PRIV in state #0
11	Host	DATA3		Read return for READ_PRIV in state #0

Table 6: Runway Transaction Ordering Example

Since **B** issued the write-back before it issued a coherency response to **A**'s coherent read, the write-back is logically ordered ahead of the coherent read. The host (memory controller) returns the data from **B**'s WRITE_BACK for **A**'s READ_PRIV.

1.4.2 Simple Memory System Design

The simplest memory system design that preserves Runway logical ordering[1] uses a coherency FIFO to keep track of *coherent transactions with coherency checks* that have not completed the coherency response phase and a memory issue FIFO to keep track of transactions that are ready to be issued to memory. All *coherent transactions with coherency checks* would initially be placed in the coherency FIFO. Once the coherency response phase completes for a transaction in the coherency FIFO, the transaction is either moved into the memory issue FIFO or discarded depending on the coherency response. All *non-coherent transactions* and *coherent transactions without coherency checks* would be placed directly in the memory issue FIFO and thus pass all *coherent transactions with coherency checks* that have not completed their coherency response phase.

1.4.3 Memory System Performance Enhancements

The simplest memory system design does not allow Astro to take advantage of the following architectural freedoms to improve memory system performance:

- Coherent reads can be opportunistically issued to memory before the coherency response phase completes to reduce read latency. Extra logic is needed to throw away the read return if the coherency response is COPY_OUT or reissue the read if a non-coherent write to the same cache line is issued before the coherency response phase completes.

- Reads can be promoted ahead of writes to differing cache lines to reduce read latency.

- Read and write transactions to differing cache lines can be reordered to improve open page hit rates and thus improve memory bandwidth.

In Astro, all three performance enhancements are used. The coherency map will opportunistically issue coherent read transactions to memory before the coherency response phase completes. The coherency map will clean up cases where the read should not have been issued or cases when the read needs to be reissued. Both the coherency map and memory controller work together to promote reads ahead of writes to differing cache lines. The memory controller will reorder memory read and write transactions to differing cache lines to improve open page hit rates.

1.4.4 Conflict Detection and Resolution

Conflicts are cases where multiple accesses to the same cache line occur close enough in time that the older access is not completely processed by the memory system before the newer access is issued. In memory systems that reorder transactions to improve memory performance, conflicts

[1] At least it is the simplest memory controller design I could think of...

require some special case handling to adequately preserve the logical order of transactions. In Astro, the coherency map is responsible for detecting conflicts and both the coherency map and the memory controller are responsible for resolving conflicts.

The coherency map keeps track of all transactions that are pending in the memory system. A read is pending from the time the Runway interface sends the read to the coherency map until the read return is complete or the read return is discarded with a COPY_OUT coherency response. A write transaction is pending from the time the Runway interface sends the write to the coherency map until the memory controller issues the write to memory.

1.4.4.1 Write / Write Conflict

A write/write conflict occurs when two writes to the same cache line are pending in Astro. The writes must make it to memory in the same order they were issued on the Runway bus.

A FIFO called the write queue (WQ) prevents writes from passing writes in the coherency map. The coherency map will issue writes to the memory controller in the order they were received from the Runway bus.

The memory controller will never reorder two writes to the same address.

Under these conditions, a write can never pass another write to the same cache line.

1.4.4.2 Read / Read Conflict

A read/read conflict occurs when two reads to the same cache line are pending in Astro. Since the state of memory is the same for both reads, the reads can complete in any order.

1.4.4.3 Write / Read Conflict

A write/read conflict occurs when a write and read to the same cache line are pending in Astro. (The write was before the read on the Runway bus.) The write must make it to memory before the read. (The read can not pass the write in Astro.)

When a read is sent from the Runway interface to the coherency map, the read is entered into the RQ and the CRQ. (Note: In certain cases, the read may be issued immediately to the memory controller without it being entered the RQ, but it is always entered into the CRQ.) The coherency map checks the read address against all pending write transactions in the memory system. If there is a conflict, the conflict is recorded in the coherency map entries for both the read and the write.

As long as the coherency map has a pending write transaction with its read conflict bit set, the coherency map will assert the **marked_read_fence** signal. When the **marked_read_fence** is asserted, the memory controller will not issue any reads that are "marked" to memory. A marked read is a read that has a conflict with a write.

Once the read reaches the top of the RQ, the coherency map is checked to see if the read conflicts with a write. If so, writes are unloaded from the WQ until all conflicting writes are issued to the memory controller. Then the read at the top of the RQ is issued to the memory controller as a marked read. Once the memory controller has retired all writes that have a read conflict, the coherency map will deassert the **marked_read_fence** signal. The memory controller is now free to issue the marked read to memory.

Under these conditions a read can never pass a write to the same cache line.

1.4.4.4 Read / Write Conflict

A read/write conflict occurs when a read and write to the same cache line are pending in Astro. (The read was before the write on the Runway bus.) The write must make it to memory before the read. (The write must pass the read in Astro.) See Rule #3 in **Section 1.4.1 Runway Transaction Ordering Rules** for more information.

When a write is sent from the Runway interface to the coherency map, the write is entered into the WQ. The coherency map checks the write address against all pending read transactions in the memory system. If there is a conflict, it is recorded in the coherency map entries for both the read and the write.

As long as the coherency map has a pending write transaction with its read conflict bit set, the coherency map will assert the **marked_read_fence** signal to the memory controller to tell it not to issue any reads that are "marked" to memory. A marked read is a read that has a conflict with a write.

Once the coherency response for the read reaches the top of the CRQ, the coherency map entry for the read is checked to see if the read has been sent to the memory controller and whether a write conflicted with the read after it had been sent to the memory controller. If this is the case, and the coherency response is not COPY_OUT, then writes are unloaded from the WQ until all conflicting writes are issued to the memory controller. Then the read at the top of the CRQ is reissued to the memory controller as a marked read. Once the memory controller has retired all writes that have a read conflict, the coherency map will deassert the **marked_read_fence** signal. The memory controller is now free to issue the marked read to memory.

Under these conditions a write will always pass a read to the same cache line.

1.5 Block Diagram

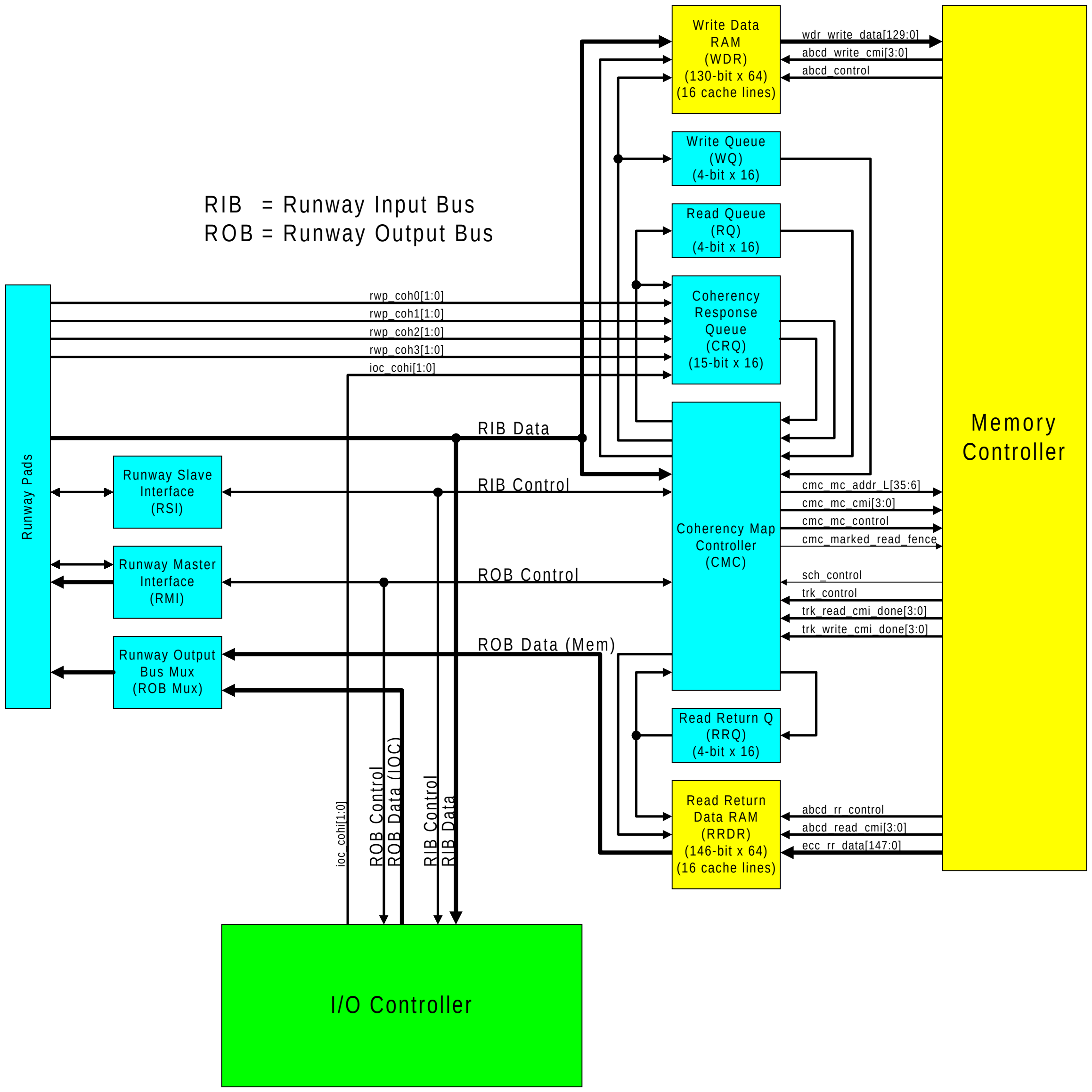

Figure 14: Runway Bus Interface Block Diagram

The complete RBIB block diagram is show in **Figure 14: Runway Bus Interface Block Diagram**. The RBIB breaks down into the following blocks:

Runway Interface Blocks

- Runway Slave Interface (RSI)
- Runway Master Interface (RMI)
- Runway Arbiter (RA)

Coherency Map Blocks

- Write Data RAM (WDR)
- Write Queue (WQ)
- Read Queue (RQ)
- Coherency Response Queue (CRQ)
- Coherency Map Controller (CMC)
- Read Return Queue (RRQ)
- Read Return Data RAM (RRDR)

1.6 Block Descriptions

1.6.1 Runway Interface Blocks

1.6.1.1 Runway Slave Interface

The RSI performs the following functions:

- Receives all Runway in-bound transactions.
- Decodes address range for Runway transactions to determine if they are to memory or I/O space.
- Routes in-bound transactions to the coherency map, the I/O controller, or both via the Runway Input Bus.
- Receives Runway coherency responses and forwards them to the coherency map.
- Detects and logs Runway parity errors on the Runway control, address, and data busses.
- Detects and logs address out-of-range errors for Runway transactions.

1.6.1.2 Runway Master Interface

The RMI performs the following functions:

- Masters all Runway out-bound transactions.
- Receives DMA transactions from the I/O controller via the Runway Output Bus and forwards them to the Runway bus.

- Receives read return transactions from the coherency map via the Runway Output Bus and forwards them to the Runway bus.
- Performs arbitration control for the Runway Output Bus.
- Uses the CLIENT_OP bus to implement system wide flow control for Astro and the processors on the Runway bus.
- Uses the CLIENT_OP bus to arbitrate for the Runway bus for memory read returns.
- Uses the IOA arbitration out signal to arbitrate for the Runway bus for the I/O controller.

1.6.2 Coherency Map Interface Blocks

1.6.2.1 Write Data RAM

The WDR performs the following function:

- Buffers Runway write data for the memory controller.

The WDR buffers 16 cache lines of write data from Runway to be issued to the memory controller. Data is written into the WDR by the Runway interface and read from the WDR by the memory controller.

1.6.2.2 Write Queue

The WQ performs the following functions:

- Buffers the CMI for all write transactions that have been received from the Runway interface but have not been issued to the memory controller.
- Keeps write transactions in the order they were issued on Runway.

The CMC will load a new CMI (see **Section 1.6.2.5 Coherency Map Controller** for definition of CMI) into the WQ when the Runway interface sends a write to the coherency map. The CMC will unload the WQ when it issues the write to the memory controller.

1.6.2.3 Read Queue

The RQ performs the following functions:

- Buffers the CMI for all read transactions that have been received from the Runway interface but have not been issued to the memory controller.
- Keeps read transactions in the order they were issued on Runway.

The CMC will load a new CMI (see **Section 1.6.2.5 Coherency Map Controller** for definition of CMI) into the RQ when the Runway interface sends a read to the coherency map. The CMC will unload the RQ when it issues the read to the memory controller. Coherent reads at the head of the RQ may be issued to the memory controller before the coherency response and read/write conflict checking phases are complete. In the case where a read uses the medium RQ bypass path to go to the memory controller, no CMI is loaded into the RQ.

1.6.2.4 Coherency Response Queue

The CRQ performs the following functions:

- Buffers the CMI for all coherent transactions that have been received from the Runway interface but have not completed the coherency response phase.
- Buffers the coherency responses from all processors on Runway.
- Buffers the coherency response from the I/O controller.
- Keeps coherency responses in the order that the coherent transactions were issued on Runway.

The CMC loads a CMI (see **Section 1.6.2.5 Coherency Map Controller** for definition of CMI) into the CRQ when the Runway interface sends a coherent transaction to the coherency map. The Runway interface loads coherency responses from all processors into the CRQ. The I/O controller also loads its coherency response into the CRQ. (Note: All data is loaded into the CRQ in the order that the coherent transaction was issued on Runway.) Once the coherent transaction at the head of the CRQ has a coherency response from all caching agents, the CMC will update the coherency response for the transaction in the read coherency map. If there is no read/write conflict, then the CRQ is unloaded. Else, if there is a read/write conflict, then the CRQ is unloaded once the read is reissued to the memory controller by the coherency map.

1.6.2.5 Coherency Map Controller

The CMC performs the following functions:

- Tracks all transactions that are pending in the memory system.
- Tracks the coherency response for all coherent transactions that are pending in the memory system.
- Loads the CMI for writes, reads, and coherent transactions into the WQ, RQ, and CRQ respectively.
- Detects write/read and read/write conflicts.
- Issues writes to the memory controller from the WQ.

- Resolves write/read conflicts, if any, before issuing reads to the memory controller from the RQ.
- Resolves read/write conflicts, if any, before reissuing reads from the CRQ, if necessary.
- Generates read return transactions to the Runway interface.
- Generates completion responses for sync and purge allocate transactions to the Runway interface.

The CMC has a 16 entry read coherency map and a 16 entry write coherency map that keep track of all memory transactions from the time they are received from Runway until they are complete. (A transaction is complete when a write is issued to memory or a read returns data to the Runway interface.) Every transaction in the read and write coherency maps is referenced by a 4-bit coherency map index (CMI). See **Section 1.7 Read/Write Coherency Map Definition** for more information.

1.6.2.6 Read Return Queue

The RRQ performs the following functions:

- Buffers the CMI for all read transactions that are ready to be returned (i.e. read return data is in the RRDR, the coherency response phase is complete, and the coherency response is not COPY_OUT) but have not been issued to either the Runway interface or the I/O controller.

The RRQ is loaded by the CMC when one of the following conditions are met:

- The memory controller issues read return data to the RRDR for a read that has completed its coherency response phase and the coherency response is not COPY_OUT.

- OR -

- The CRQ issues a coherency response that is not COPY_OUT for a read that has read return data in the RRDR.

The CMC will unload the RRQ when it issues the read return to the Runway Output Bus.

1.6.2.7 Read Return Data RAM

The RRDR performs the following function:

- Buffers Runway read return data for the memory controller.

The RRDR buffers up to 16 cache lines of read return data yet to be issued to the Runway interface. Data is written into the RRDR by the memory controller and read from the RRDR by the coherency map. Data can be unloaded from the RRDR in a different order than it was loaded.

1.7 Read/Write Coherency Map Definition

The CMC block contains a 16 entry write coherency map that tracks the status of write transactions and a 16 entry read coherency map that tracks the status of read transactions. Each entry in each coherency map is referenced by a 4-bit coherency map index (**cmi[3:0]**). Below is a definition of the fields in write and read coherency map entries.

Write Coherency Map Fields

Each write coherency map entry contains the following fields:

wcm_pending	wcm_addr[35:6]	wcm_in_wq	wcm_read_conf
0 - No write pending 1- Write Pending	Physical Address	0 - Write not in WQ 1 - Write in WQ	0 - No read conflict 1 - Read conflict

Table 7: Write Coherency Map Fields

The "wcm" at the beginning of each field stands for "write coherency map." The **wcm_pending** field keeps track of whether the write is pending in the MC or not. This bit is used as a valid bit for the entire write coherency map entry. The **wcm_addr** field keeps track of the 36-bit physical address. (Since the MC will only do 64-byte cache line transactions, the 6 LSBs of the physical address are always 6'b000000.) The **wcm_in_wq** field keeps track of whether the CMI for the write is in the WQ or not. The **mcm_read_conf** field keeps track of whether this write transaction conflicts with a read transaction that is pending in the read coherency map. The **cmc_marked_read_fence** signal is asserted whenever there is one or more lines in the write coherency map that have **wcm_pending** and **wcm_read_conf** both asserted. The **cmc_marked_read_fence** is used to tell the MC to not allow a read with **cmc_marked_read** asserted to be issued to memory.

Read Coherency Map Fields

Each read coherency map entry contains the following fields:

rcm_pending	rcm_addr[35:6]	rcm_master_id[2:0]	rcm_trans_id[5:0]
0 - No read pending 1- Read Pending	Physical Address	Runway master ID	Runway transaction ID

rcm_bs	rcm_0_length	rcm_type[1:0]	rcm_coh_resp[1:0]
0 – No byte swap 1 – Byte swap	0 – Not 0 length 1 – 0 length	00 - READ 01 - PURGE_ALLOC 10 - SYNC 11 - NOP	00 - OK 01 – SHARED 10 – COPY_OUT 11 - NO_RESPONSE

rcm_in_rq	rcm_rr_count[1:0]	rcm_write_conf	rcm_late_write_conf	rcm_error
0 - Read not in RQ 1 – Read in RQ	A count of the number of read returns pending in the MC.	0 - No write conflict 1 - Write conflict	0 - No late write conflict 1 – Late write conflict	0 – No error 1 – Error

Table 8: Read Coherency Map Fields

The “rcm” at the beginning of each field stands for “read coherency map.” The **rcm_pending** field is asserted when a transaction enters the read coherency map entry and is deasserted when the transaction is retired from the read coherency map entry. This bit is used as a valid bit for the entire read coherency map entry. The **rcm_addr** field keeps track of the 36-bit physical address. (Since the MC will only do 64-byte cache line transactions, the 6 LSBs of the physical address are always 6’b000000.) The **rcm_master_id** and **rcm_trans_id** fields keep track of the Runway master and transaction IDs respectively. The **rcm_bs** field is asserted for Runway read transactions that require byte swapping of the read return data. The **rcm_0_length** field is set for Runway read transactions that do not require read return data. (This feature is only possible for read transactions that started in the I/O controller.) The **rcm_type** field keeps track of the transaction type. The values of READ, PURGE_ALLOC, SYNC, and NOP represent coherent or non-coherent read, purge allocate, sync, and coherent transactions that require no operation to be performed in the coherency map respectively. (Flush and purge transactions are examples of transactions that would have an **rcm_type** of NOP.) The **rcm_coh_resp** field keeps track of the coherency response status for coherent reads. This field is initially set to OK for non-coherent reads and NO_RESPONSE for all other transactions. The **rcm_in_rq** field keeps track of whether the CMI for the read is in the RQ or not. The **rcm_rr_count** field keeps track of the number of read returns pending in the MC. This field is incremented each time the read is issued to the MC and decremented each time the MC returns data for the read in the RRDR. The **rcm_write_conf** field is asserted when a read transaction conflicts with a write transaction that is pending in the write coherency map. (The **cmc_marked_read** bit is asserted whenever a read is issued to the MC with the **rcm_write_conf** bit asserted.) The **rcm_late_write_conf** field is asserted when a later write transaction conflicts with a read transaction. When this condition occurs, the CMC will re-issue the read to the memory controller at the end of the coherency response phase. The **rcm_error** field is asserted when an error is detected for the transaction. If the **rcm_error** field is asserted, then the RBIB will not do a read return for read transactions, a SYNC_DONE for sync transactions, or a DPURGE_ALLOC_DONE for purge allocate transactions.

1.8 Runway Bus Interface Block Registers

The following registers are implemented in the Runway Bus Interface Block. All registers must be accessed with 64-bit I/O load/store instructions.

FROM A FIRMWARE AND SOFTWARE POINT-OF-VIEW: All “reserved” register fields must be written with “0”s and will read as “X”s. This allows future revisions of Astro to add control bits in reserved fields to control unsightly bug fixes without breaking firmware or software. This is different than the “unimplemented” register fields used elsewhere in Astro. Unimplemented fields may be written with “X”s and will read as “0”s.

1.8.1 Runway Slave Interface Registers

1.8.1.1 Memory Size Register

MSB	MEM_SIZE Register (address: 0xFF_FED0_8000)																														LSB
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
unimplemented																									size[38:32]						
Power On Initialization																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size[31:26]						unimplemented																									re
Power On Initialization																															
X	X	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Size: 64-bit only		
Field	**Access**	**Description**
size	R/W	Size of Astro memory. Size[38:37] is hardwired to 2’b00.
re	R/W	Range Enable. Set to “1” to enable Astro memory.

Register 1: MEM_SIZE Register

The MEM_SIZE register specifies how much memory is installed (and mapped) in Astro, including any memory that is remapped because of the memory hole at addresses 0x00_F000_0000 to 0x00_FFFF_FFFF.

Memory size is defined as follows:

Mem_Size = {**size[38:26]**, 26’b0}

Note that this register specifies the amount of memory, so the value is 1 larger than the highest (unrelocated) memory location.

The MEM_SIZE register and MEM_HOLE_RELOC Register are currently defined to support up to 64 GBytes of memory, which corresponds to 1 GBit SDRAMs in by-four packaging.

1.8.1.2 Memory Hole Relocation Register

MSB	MEM_HOLE_RELOC Register (address: 0xFF_FED0_8008)																														LSB
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
unimplemented																									distance[38:32]						
Power On Initialization																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
unimplemented																															re
Power On Initialization																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Size: 64-bit only		
Field	**Access**	**Description**
distance	R	Relocation distance for LMMIO memory hole. Distance[38:32] is hardwired to 7'b0010000.
re	R/W	Range Enable. Set to "1" to enable relocation range.

Register 2: MEM_HOLE_RELOC Register

The MEM_HOLE_RELOC register specifies where the memory normally at the address 0x00_F000_0000 to 0x00_FFFF_FFFF gets remapped.

Memory relocation distance is defined as follows:

Mem_Reloc = {**distance[38:32]**, 32'b0}

The location of the relocated memory is defined as follows:

```
if (Mem_Size > 0xFFFF_FFFF) {
   Hole_Base = Mem_Reloc + LMMIO_Base;
   Hole_Top = Mem_Reloc + 0xFFFF_FFFF;
} else if (Mem_Size + LMMIO_Size > 0xFFFF_FFFF) {
   Hole_Base = Mem_Reloc + LMMIO_Base;
   Hole_Top = Mem_Reloc + Mem_Size – 1;
} else {
   /* No memory relocation is needed */
}
```

1.8.2 Coherency Map Controller Registers

1.8.2.1 Runway Bus Interface Block Control Register

MSB	RBIB_CTRL Register (address: 0xFF_FED0_C000)	LSB

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved																															
Power On Initialization																															
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved			throttle_rr_en	reserved		med_read_bypass_en	reserved	wq_hp_hw[3:0]				wq_hp_lw[3:0]				wcm_fc_hw[3:0]				wcm_fc_lw[3:0]				rcm_fc_hw[3:0]				rcm_fc_lw[3:0]			
Power On Initialization																															
X	X	X	0	X	X	0	X	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0

Size: 64-bit only		
Field	**Access**	**Description**
throttle_rr_en	R/W	Enable read return throttling on Runway.
med_read_bypass_en	R/W	Enable medium read bypass logic.
wq_hp_hw	R/W	Write queue high priority high water mark.
wq_hp_lw	R/W	Write queue high priority low water mark.
wcm_fc_hw	R/W	Write coherency map flow control high water mark.
wcm_fc_lw	R/W	Write coherency map flow control low water mark.
rcm_fc_hw	R/W	Read coherency map flow control high water mark.
rcm_fc_lw	R/W	Read coherency map flow control low water mark.

Register 3: RBIB_CTRL Register

The RBIB_CTRL register controls performance tuning and flow control for the RBIB. The RBIB_CTRL register can only be written when there are no reads or writes to memory pending in Astro. PDC firmware should program the RBIB_CTRL register before it enables the memory controller.

Note: The **rsi_error_enable** field in bit position 31 was removed for Astro 2.0. The **rsi_error_enable** bit was a metal fix used in Astro 1.0 to disable any error detection in the RBIB in case a falsely detected error would prevent the system from operating. For Astro 2.0 and beyond, the error signaling bits in the **ERROR_ENABLE** register will take over this task.

When the **throttle_rr_en** field is 1, then read returns from memory will be throttled back when either a processor or the I/O controller is arbitrating for the Runway bus. This feature helps keep the memory pipeline from filling and draining as often.

When the **med_read_bypass_en** field is 1, then memory read requests are allowed to bypass the RQ and issue directly to memory in one state. This feature should be able to trim 1 state off of read latency.

The **wq_hp_hw** and **wq_hp_lw** fields control how far the WQ can get backed up before the CMC prioritizes writes ahead of reads. When the number of entries in the WQ is *greater than* **wq_hp_hw**, then writes are treated as high priority. Writes will continue to be high priority until the number of entries in the WQ is *less than or equal to* **wq_hp_lw**.

The **wcm_fc_hw** and **wcm_fc_lw** fields control how far the write coherency map can get backed up before the CMC flow controls the Runway bus to the NONE_ALLOWED state. (In the NONE_ALLOWED state, no transactions are allowed.) When the number of entries in the write coherency map is *greater than* **wcm_fc_hw**, then Runway will be flow controlled. Runway will continue to be flow controlled until the number of entries in the write coherency map is *less than or equal to* **wcm_fc_lw**.

The **rcm_fc_hw** and **rcm_fc_lw** fields control how far the read coherency map can get backed up before the CMC flow controls the Runway bus to the RET_ONLY state. (In the RET_ONLY state, only memory write backs, memory flush backs, cache-to-cache writes, sync done transactions, data purge allocate done transactions, error transactions, memory read returns and I/O read returns are allowed.) When the number of entries in the read coherency map is *greater than* **rcm_fc_hw**, then Runway will be flow controlled. Runway will continue to be flow controlled until the number of entries in the read coherency map is *less than or equal to* **rcm_fc_lw**.

1.8.2.2 Write Coherency Map Diagnostic Read Registers

MSB	WCM_DIAG_READ_0 to WCM_DIAG_READ_15 Register (address: 0xFF_FED0_C100 to 0xFF_FED0_C178)	LSB

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved																													wcm_pending	wcm_in_wq	wcm_read_conf
Power On Initialization																															
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	X	X

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	wcm_addr[35:6]																														
Power On Initialization																															
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Size: 64-bit only		
Field	**Access**	**Description**
wcm_pending	R	Entry in WCM is pending.
wcm_in_wq	R	Entry in WCM is in the WQ.
wcm_read_conf	R	Entry in WCM conflicts with a read in the RCM.
wcm_addr	R	Address of transaction.

Register 4: WCM_DIAG_READ_0 to 15 Register

The WCM Diagnostic Read Registers are 16 read-only registers that allow diagnostic code to read the state of any line in the write coherency map. These registers are used to help debug Astro. Firmware does not need to access these registers. See **Section 1.7 Read/Write Coherency Map Definition** for information on write coherency map field definitions.

1.8.2.3 Read Coherency Map Diagnostic Read Registers

MSB	RCM_DIAG_READ_0 to RCM_DIAG_READ_15 Register (address: 0xFF_FED0_C180 to 0xFF_FED0_C1F8)	LSB

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
reserved											rcm_pending	rcm_master_id[2:0]			rcm_trans_id[5:0]						rcm_bs	rcm_0_length	rcm_type[1:0]		rcm_coh_resp[1:0]		rcm_in_rq	rcm_rr_count[1:0]		rcm_write_conf	rcm_late_write_conf
Power On Initialization																															
X	X	X	X	X	X	X	X	X	X	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	rcm_addr[35:6]																														
Power On Initialization																															
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Size: 64-bit only		
Field	**Access**	**Description**
rcm_pending	R	Entry in RCM is pending.
rcm_master_id	R	Master ID of transaction.
rcm_trans_id	R	Transaction ID of transaction.
rcm_bs	R	Byte swap read return.
rcm_0_length	R	Transaction is a 0 length read request.
rcm_type	R	Type of transaction.
rcm_coh_resp	R	Coherency response.
rcm_in_rq	R	Entry in RCM is in the RQ.
rcm_rr_count	R	Number of reads outstanding in the memory controller for this entry in the RCM.
rcm_write_conf	R	Entry in RCM conflicts with a write in the WCM.
rcm_late_write_conf	R	Entry in RCM conflicts with a write in the WCM after the read was issued to memory.
rcm_addr	R	Address of transaction.

Register 5: RCM_DIAG_READ_0 to 15 Register

The RCM Diagnostic Read Registers is a collection of 16 read-only registers that allow diagnostic code to read the state of any line in the read coherency map. These registers are used to help debug Astro. Firmware does not need to access these registers. See **Section 1.7**

Read/Write Coherency Map Definition for information on read coherency map field definitions.

1.9 Internal Bus Definitions

Below are the definitions of the signals on the Runway Input Bus and the Runway Output Bus. The first field in a signal name identifies the block that drives the signal. The only exception to this rule is for signals on the Runway Output Bus that are driven by more than one block. These signals will begin with 'rob' to identify them as Runway Output Bus signals.

1.9.1 Runway Input Bus

Signal	Description
rwp_addr_data[127:0]	Multiplexed address and data bus. Addresses are mapped to **rwp_addr_data[31:24,39:32,47:40,55:48,63:56]**. Data use the entire **rwp_addr_data** bus.
rsi_byte_enable[15:0]	Byte enables for I/O data (not address, not memory data) on the **rwp_addr_data** bus. Byte enables are valid with the address for an I/O write or I/O read transaction. (i.e. Byte enables are valid when either **rsi_io_read** or **rsi_io_write** are asserted.)
rwp_master_id[2:0]	Master ID for address and data on the **rwp_addr_data** bus.
rwp_trans_id[5:0]	Transaction ID for address and data on the **rwp_addr_data** bus.
rsi_pdh_read64	PDH cache line read strobe. Asserted when a memory read to PDH memory address space is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_mem_read	Memory read strobe. Asserted when a memory read address is on the Runway Input Bus. Not asserted when a PDH cache line read or a 0 length read is on the Runway Input Bus. See **rsi_pdh_read64** and **rsi_0_length** for more information. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_mem_write	Memory write strobe. Asserted when a memory write address is on the Runway Input Bus. No memory in-range or parity checks are performed for this transaction. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.

Signal	Description
rsi_io_read	I/O read strobe. Asserted when an I/O read address is on the Runway Input Bus. Note: No I/O address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_io_write	I/O write strobe. Asserted when an I/O write address is on the Runway Input Bus. Note: No I/O address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_ioc_c2c_write	I/O controller cache-to-cache write strobe. Asserted when a cache-to-cache write address with the I/O controller master ID is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_ioc_rr_data_valid	I/O controller read return data valid. Asserted when read return data for the I/O controller is on the Runway Input Bus. Used for read returns from memory, cache-to-cache copies and I/O read returns. NOTE: The **rsi_ioc_rr_data_valid** signal will assert for Runway path errors (when both ADDR_VALID and DATA_VALID are asserted). The **rsi_byte_parity[15:0]** signal will be used to notify the I/O controller that a path error occurred.
rsi_flush	Flush strobe. Asserted when a memory flush address is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_purge	Purge strobe. Asserted when a memory purge address is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_purge_alloc	Purge allocate strobe. Asserted when a memory purge allocate address is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.

Signal	Description
rsi_sync	Sync strobe. Asserted when a sync transaction is on the Runway Input Bus. Note: No parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_coherent	Coherent transaction strobe. Asserted when a coherent transaction is on the Runway Input Bus. (i.e. Asserted when ADDR_VALID is asserted, DATA_VALID is deasserted, and TTYPE[2] is asserted.) Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_byte_swap	Byte swap transaction strobe. Asserted when a byte swapped read address is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_0_length	Zero length read strobe. Asserted when a 0 length read address is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_private	Private read strobe. Asserted when a private read address is on the Runway Input Bus. Note: No memory address in-range or parity checks are performed for this transaction. The **rsi_error** signal will assert on the next state if an error occurred.
rsi_ioc_mastered	I/O controller mastered transaction. Asserted when am address that was mastered by the I/O controller is on the Runway Input Bus. The I/O controller will only sample this signal when **rsi_coherent** is also asserted. This tells the I/O controller it mastered the coherent transaction.
rsi_error	Error strobe. Asserted when the previous Runway Input Bus transaction was either a broadcast error, or an out-of-range memory transaction, or an out of range I/O transaction, or a transaction with an address parity error, or a transaction with a control parity error. See **Section 1.9.1.1:Runway Input Bus Error Strategy** for more information.
rsi_memory_oor	Memory out-of-range strobe. Asserted when the previous Runway Input Bus transaction contained a memory address that was out-of-range. When asserted, **rsi_error** must also be asserted.

Signal	Description
rsi_parity_error[1:0]	Uncorrectable data parity error. A data parity error occurred on **rwp_addr_data[127:64]** when **rsi_parity_error[1]** is asserted and a data parity error occurred on **rwp_addr_data[63:0]** when **rsi_parity_error[0]** is asserted. All bits of **rsi_parity_error[1:0]** will assert for a Runway path error (when both ADDR_VALID and DATA_VALID are asserted). See **Section 1.9.1.1:Runway Input Bus Error Strategy** for more information.
rsi_byte_parity[15:0]	Byte parity for data (not address) on the **rwp_addr_data** bus. Parity is even. (There will always be an even number of 1's for data and parity combined.) Byte parity is poisoned for Runway parity errors at 4-byte granularity. All bits of **rsi_byte_parity[15:0]** will assert for a Runway path error (when both ADDR_VALID and DATA_VALID are asserted). See **Section 1.9.1.1:Runway Input Bus Error Strategy** for more information.
rwp_coh0[1:0] **rwp_coh1[1:0]** **rwp_coh2[1:0]** **rwp_coh3[1:0]**	Runway Coherency response from processors. Each processor is mapped to an aligned two-bit field. Coherency responses are: • 2'b00: OK • 2'b01: COPYOUT • 2'b10: SHARED • 2'b11: NO_RESPONSE
ioc_cohi[1:0]	I/O cache coherency response. Uses the same coherency response encoding as **rwp_coh0**.
cmc_rq_cmi	Next available read coherency map index for transactions entered into the RQ or CRQ.
cmc_wq_cmi	Next available write coherency map index for transactions entered into the WQ.

Table 9: Runway Input Bus Signal Definitions

Runway Transaction Type	rsi_pdh_read64	rsi_mem_read	rsi_mem_write	rsi_ioc_c2c_write	rsi_io_read	rsi_io_write	rsi_flush	rsi_purge	rsi_purge_alloc	rsi_sync	rsi_coherent	rsi_byte_swap	rsi_0_length	rsi_private
READ (PDH)	X													
READ (Memory)		X												
READ_BS (PDH)	X											X		
READ_BS (Memory)		X										X		
READ_SHAR_OR_PRIV		X									X			
READ_CURRENT		X									X			
READ_PRIV		X									X			X
READ_PRIV (0 length hint)											X		X	X
CLEAR_PRIV		X									X			X
WRITE			X											
WRITE_BACK			X											
FLUSH_BACK			X											
C2C_WRITE (not to IOC)			X											
C2C_WRITE (to IOC)			X	X										
READ_SHORT					X									
WRITE_SHORT						X								
DFLUSH							X				X			
DFLUSH_WR							X				X			
DFLUSH_RD							X				X			
DFLUSH_RD_WR							X				X			
IFLUSH							X				X			
DPURGE								X			X			
DPURGE_ALLOC									X		X			
CACHE_SYNC										X	X			
DMA_SYNC										X	X			
Other with TTYPE[2] = 0														
Other with TTYPE[2] = 1											X			

Table 10: Runway Input Bus Transaction Decoding

1.9.1.1 Runway Input Bus Error Strategy

Runway Input Bus transactions that have address parity errors, address control parity errors, and address out-of-range errors are signaled by asserting **rsi_error** one state after the transaction is issued on the Runway Input Bus. **Table 11: Runway Input Bus Error Responses** define how Runway Input Bus transactions are resolved if **rsi_error** is asserted.

RIB Transaction	Error Response
READ (PDH) **READ_BS (PDH)**	The PDH cache line read transaction is ignored by the I/O controller. The read will never be issued to Dillon.

RIB Transaction	Error Response
READ (Memory) READ_BS (Memory) READ_SHAR_OR_PRIV READ_CURRENT READ_PRIV READ_PRIV (0 length hint) CLEAR_PRIV	If the memory read transaction's address is out-of-range, then the read will be sent to the MBAT and the MBAT will ignore the transaction. There will be no read issued to memory, there will be no read return to the RRDR. Otherwise, the transaction will be sent to memory, the read return will go to the RRDR, and the CMC will never issue a read return.
WRITE WRITE_BACK C2C_WRITE (not to IOC)	The memory write transaction is ignored by the CMC. The write will never make it to memory.
C2C_WRITE (to IOC)	The memory write transaction is ignored by the CMC. The write will never make it to memory. The I/O controller cache-to-cache write transaction will complete as normal in the I/O controller. (The I/O controller does not care about the address phase of a cache-to-cache write, it only cares about the data associated with the master and transaction ID.)
READ_SHORT	The I/O read transaction is ignored by the I/O controller. There will be no I/O read issued to the Rope Bus.
WRITE_SHORT	The I/O write transaction is ignored by the I/O controller. There will be no I/O write issued to the Rope Bus.
DFLUSH DFLUSH_WR DFLUSH_RD DFLUSH_RD_WR IFLUSH	The CMC will track the coherency response to keep the CRQ up-to-date. The I/O controller will ignore the transaction, but give an OK coherency response.
DPURGE	The CMC will track the coherency responses to keep the CRQ up-to-date. The I/O controller will ignore the transaction, but give an OK coherency response.
DPURGE_ALLOC	The CMC will track the coherency responses to keep the CRQ up-to-date. The CMC will not issue a DPURGE_ALLOC_DONE to Runway. The I/O controller will ignore the transaction, but give an OK coherency response.
CACHE_SYNC DMA_SYNC	The CMC will track the coherency responses to keep the CRQ up-to-date. The CMC will not issue a SYNC_DONE to Runway. The I/O controller will ignore the transaction, but give an OK coherency response.
Other with TTYPE[2] = 0	The transaction will be ignored.

RIB Transaction	Error Response
Other with TTYPE[2] = 1	The CMC will track the coherency responses to keep the CRQ up-to-date. The I/O controller will ignore the transaction, but give an OK coherency response.

Table 11: Runway Input Bus Error Responses

Data parity errors, data control parity errors, and data path errors (when both ADDR_VALID and DATA_VALID are asserted) are signaled by poisoning **rsi_byte_parity[15:0]** and asserting **rsi_parity_error[1:0]**.

1.9.2 Runway Output Bus

Signal	Description
rob_addr_data[127:0]	Multiplexed address and data bus. Addresses are mapped to **rob_addr_data[31:24,39:32,47:40,55:48,63:56]**. Virtual indexes are mapped to **rob_addr_data[19:18,11:8,23:20]**. Data use the entire **rob_addr_data** bus. Note: Due to the addition of the ROB multiplexer, there is a memory controller and I/O controller version of this bus.
rob_byte_parity[15:0]	Byte parity for data (not address) on the **rob_addr_data** bus. Parity is even. (There will always be an even number of 1's for data and parity combined.) Byte parity is poisoned for I/O controller parity errors and memory uncorrectable ECC errors. Note: Due to the addition of the ROB multiplexer, there is a memory controller and I/O controller version of this bus.
rob_master_id[2:0]	Master ID for address and data on the **rob_addr_data** bus. Note: Due to the addition of the ROB multiplexer, there is a memory controller and I/O controller version of this bus.
rob_trans_id[5:0]	Transaction ID for address and data on the **rob_addr_data** bus. Note: Due to the addition of the ROB multiplexer, there is a memory controller and I/O controller version of this bus.
rob_ctl_parity	Parity for **rob_master_id** and **rob_trans_id**. Parity is even. (There will always be an even number of 1's for data and parity combined.) Note: Due to the addition of the ROB multiplexer, there is a memory controller and I/O controller version of this bus.
rmi_cmc_gnt	CMC Runway Output Bus Grant. When asserted, the CMC/RRDR drives the tri-state Runway Output Busses.
ioc_pdh_rr_req	I/O controller PDH memory read return request. Asserted when there is a PDH memory read return transaction ready for the Runway bus. Held asserted until **rmi_pdh_rr_gnt** is asserted and there are no more PDH memory read return transactions ready for the Runway bus.
rmi_pdh_rr_gnt	I/O controller PDH memory read return grant. A PDH memory read return begins on the first state when both **ioc_pdh_rr_req** and **rmi_pdh_rr_gnt** are asserted. The transaction requires four data states (one cache line) with no embedded idle cycles.
ioc_stop_pdh	I/O controller stop PDH memory reads. Asserted to flow control the Runway bus to prevent any memory read transactions. (CLIENT_OP should change to RET_ONLY.)
cmc_mem_rr_req	Memory read return request. Asserted when there is a memory read return transaction ready for the Runway bus. Held asserted until **rmi_mem_rr_gnt** is asserted and there are no more memory read return transactions ready for the Runway bus.

Signal	Description
rmi_mem_rr_gnt	Memory read return grant. A Runway memory read return begins on the first state when both **cmc_mem_rr_req** and **rmi_mem_rr_gnt** are asserted. The transaction requires four data states (one cache line) with no embedded idle cycles.
cmc_0_length_rr	I/O controller zero length read return strobe. Asserted with **cmc_0_length_trans_id** to indicate a 0 length read return to the I/O controller.
cmc_0_length_trans_id[3:0]	I/O controller zero length read return transaction ID.
cmc_shared	Shared read return flag. Asserted with **cmc_mem_rr_req** to indicate a shared return.
ioc_mem_read_req	I/O controller memory read request. Asserted when there is a memory read transaction ready for the Runway bus. Held asserted until **rmi_mem_read_gnt** is asserted and there are no more memory read transactions ready for the Runway bus.
rmi_mem_read_gnt	I/O controller memory read grant. An I/O controller memory read begins on the first state when both **ioc_mem_read_req** and **rmi_mem_read_gnt** are asserted. The transaction requires one address state.
ioc_0_length	I/O controller zero length read hint. Asserted with **ioc_mem_read_req** to hint that if the read is not COPYOUT, then the memory controller does not need to return data with the memory read return.
ioc_mem_write_req	I/O controller memory write request. Asserted when there is a memory write transaction ready for the Runway bus. Held asserted until **rmi_mem_write_gnt** is asserted and there are no more memory write transactions ready for the Runway bus.
rmi_mem_write_gnt	I/O controller memory write grant. An I/O controller memory write begins on the first state when both **ioc_mem_write_req** and **rmi_mem_write_gnt** are asserted. The transaction requires one address state and four data states (one cache line) with no embedded idle cycles.
ioc_io_read_req	I/O controller I/O read request. Asserted when there is an I/O read transaction ready for the Runway bus. Held asserted until **rmi_io_read_gnt** is asserted and there are no more I/O read transactions ready for the Runway bus.
rmi_io_read_gnt	I/O controller I/O read grant. An I/O controller I/O read begins on the first state when both **ioc_io_read_req** and **rmi_io_read_gnt** are asserted. The transaction requires one address state.
ioc_io_rr_req	I/O controller I/O read return request. Asserted when there is an I/O read return transaction ready for the Runway bus. Held asserted until **rmi_io_rr_gnt** is asserted and there are no more I/O read return transactions ready for the Runway bus.
rmi_io_rr_gnt	I/O controller I/O read return grant. An I/O controller I/O read return begins on the first state when both **ioc_io_rr_req** and **rmi_io_rr_gnt** are asserted. The transaction requires one data state (8-bytes).

Signal	Description
ioc_io_write_req	I/O controller I/O write request. Asserted when there is an I/O write transaction ready for the Runway bus. Held asserted until **rmi_io_write_gnt** is asserted and there are no more I/O write transactions ready for the Runway bus.
rmi_io_write_gnt	I/O controller I/O write grant. An I/O controller I/O write begins on the first state when both **ioc_io_write_req** and **rmi_io_write_gnt** are asserted. The transaction requires one address state and one data states (8-bytes) with no embedded idle cycles.
cmc_sync_done_req	Sync done request. Asserted when there is a sync done transaction ready for the Runway bus. Held asserted until **rmi_sync_done_gnt** is asserted and there are no more sync done transactions ready for the Runway bus.
rmi_sync_done_gnt	Sync done grant. A sync done begins on the first state when both **cmc_sync_done_req** and **rmi_sync_done_gnt** are asserted. The transaction requires one state.
cmc_purge_alloc_done_req	Purge allocate done request. Asserted when there is a purge allocate done transaction ready for the Runway bus. Held asserted until **rmi_purge_alloc_done_gnt** is asserted and there are no more purge allocate done transactions ready for the Runway bus.
rmi_purge_alloc_done_gnt	Purge allocate done grant. A purge allocate done begins on the first state when both **cmc_purge_alloc_done_req** and **rmi_purge_alloc_done_gnt** are asserted. The transaction requires one state.
rsi_broad_error_req	Broadcast error request. Asserted when there is a broadcast error to send to the Runway bus. Held asserted until **rmi_broad_error_gnt** is asserted and there are no more broadcast error transactions ready for the Runway bus.
rmi_broad_error_gnt	Broadcast error grant. A broadcast error begins on the first state when both **rsi_broad_error_req** and **rmi_broad_error_gnt** are asserted. The transaction requires one state.
ioc_stop_io	Stop I/O transactions. Asserted to prevent I/O reads and I/O writes from being issued on the Runway bus. Assuming that the RMI block obeys the Runway meta-protocol and only issues one I/O transaction every two Runway states, then there could be up to two I/O transactions received on the Runway Input Bus once **ioc_stop_io** is asserted on the Runway Output Bus.
ioc_cmd_reset	Command reset. Asserted for 1 state by the I/O controller after a broadcast reset command (WRITE_SHORT) is seen on the Runway Input Bus.

Table 12: Runway Output Bus Signal Definitions

Runway Transaction Type	ioc_pdh_rr_req	cmc_mem_rr_req	ioc_mem_read_req	ioc_mem_write_req	ioc_io_read_req	ioc_io_rr_req	ioc_io_write_req	cmc_sync_done_req	cmc_purge_alloc_done_req	rsi_broad_error_req	cmc_shared	cmc_0_length_rr	ioc_0_length
HOST_CONTROL Read Return	X												
HOST_CONTROL Read Return		X											
SHAR_RTN Read Return		X									X		
(0 length read return to IOC)												X	
READ_SHAR_OR_PRIV			X										
READ_CURRENT			X										
READ_PRIV			X										
READ_PRIV (0 length hint)			X										X
WRITE_BACK				X									
READ_SHORT					X								
(I/O read return from IOC)						X							
WRITE_SHORT							X						
SYNC_DONE								X					
DPURGE_ALLOC_DONE									X				
BROAD_ERROR										X			

Table 13: Runway Output Bus Transaction Encoding

1.10 Runway Transaction Summary

Table 14 summarizes the Runway transactions that Astro supports. Astro does partial TTYPE decoding in compliance with Appendix B of the Runway Specification (Rev 1.3). The **TTYPE[0:7]** field is the Runway transaction type. The **Master** field is marked with “I” or “M” if the I/O controller or the memory controller can master the transaction respectively. The **Receive** field is marked with “I”, “M”, or “B” if the I/O controller, memory controller, or both can receive the transaction respectively. The **Coherent** field is checked for coherent transactions. The **# Cycles** field identifies the minimum number of cycles the transaction requires including data cycles. Astro does not generate any imbedded idle cycles in Runway transactions. Astro is tolerant of Runway devices that do generate imbedded idle cycles in Runway transactions.

Runway Transaction	TTYPE[0:7]	Master	Receive	Coherent	# Cycles
DIR_ERROR	0x08				1
ITLB_PURGE_DONE	0x10				1
DTLB_PURGE_DONE	0x14				1
SYNC_DONE	0x18	M			1

Runway Transaction	TTYPE[0:7]	Master	Receive	Coherent	# Cycles
WRITE_SHORT_DONE	0x1A				1
DPURGE_ALLOC_DONE	0x1C	M			1
BROAD_ERROR	0x48	M	I		1
ITLB_PURGE	0x50				1
DTLB_PURGE	0x54				1
HOST_CMD	0x58				1
HOST_CMD	0x59				1
UPDATE_DUP_TAGS	0x5C				1
CACHE_SYNC	0x60		I	✓	1
DMA_SYNC	0x64		I	✓	1
IFLUSH	0x68		I	✓	1
IFLUSH_ENTRY	0x69			✓	1
CLEAR_WORD	0x70			✓	2
CLEAR_DBL	0x72			✓	2
DFLUSH	0x74		I	✓	1
DFLUSH_WR	0x75		I	✓	1
DFLUSH_RD	0x76		I	✓	1
DFLUSH_RD_WR	0x77		I	✓	1
DPURGE	0x7C		I	✓	1
DPURGE_ALLOC	0x7E		I	✓	1
WRITE_BURST	0x80				5
WRITE_SHORT (Memory)	0x81				2
WRITE_SHORT (I/O)	0x81	I	I		2
WRITE	0x90		M		5
WRITE16	0x92				3
C2C_WRITE	0x94		B		5
WRITE_BACK	0x98	I	M		5
FLUSH_BACK	0x9C		M		5
WRITE_P_UPD	0xB8			✓	5
WRITE16_P_UPD	0xBA			✓	3
WRITE_PURGE	0xBC			✓	5
WRITE16_PURGE	0xBE			✓	3
READ_BURST	0xC0			✓	1
READ_SHORT (Memory)	0xC1			✓	1
READ_SHORT (I/O)	0xC1	I	I	✓	2
FETCH_AND_DEC	0xC5			✓	1
FETCH_AND_INC	0xC7			✓	1
READ	0xD0		M		1
READ_BS	0xD4		M		1
READ (PDH Memory Space)	0xD0		I		1
READ_BS (PDH Memory Space)	0xD4		I		1
READ_CURRENT	0xF0	I	B	✓	1
READ_SHAR_OR_PRIV	0xF4	I	B	✓	1
READ_PRIV	0xF8	I	B	✓	1
CLEAR_PRIV	0xFC		B	✓	1

Runway Transaction	TTYPE[0:7]	Master	Receive	Coherent	# Cycles
HOST_CONTROL Read Return	N/A	M	I		4
SHAR_RTN Read Return	N/A	M	I		4
READ_SHORT (I/O) Read Return	N/A	I	I		1

Table 14: Runway Transaction Summary

1.11 Transaction Examples

This section walks through some transaction flow examples to develop some high level understanding for Astro's Runway Bus Interface.

1.11.1 Processor Initiated Write Transactions

Processor initiated writes are received from the Runway bus by the RSI. The RSI forwards write address and data to the CMC and WDR respectively on the Runway Input Bus. If the write is a cache-to-cache copy with the IOC's master ID, then the write is also sent to the I/O controller. The CMC allocates a CMI in the write coherency map to keep track of the write until it is issued to memory. The CMC places a CMI for the write in the WQ. When the CMI for the write reaches the top of the WQ, the CMC will pop the CMI off the WQ and issue the write address to the MC. The MC will pull the write data from the WDR and tell the CMC when the write is completed to memory. The CMC will free the write coherency map entry for the write.

1.11.2 Processor Initiated Non-Coherent Read Transactions

Processor initiated non-coherent reads are received from the Runway bus by the RSI. The RSI forwards the read address to the CMC on the Runway Input Bus. The CMC allocates a CMI in the read coherency map to keep track of the read until the read return phase is complete. If there is nothing in the RQ, then a medium RQ bypass path is enabled to issue the read address directly to the MC with 1 state of delay. Otherwise, the CMC places the CMI for the read in the RQ where it will take a minimum of 2 states to issue to the MC.

When the CMI for the read reaches the top of the RQ, the CMC will check the read coherency map for read/write conflicts. If there is a conflict, then the CMC will issue writes to the MC (if necessary) to clear the conflict. Once the write conflict is cleared, the CMC will pop the CMI for the read off the RQ and issue the read address to the MC.

The MC will read data from memory and place the read return data in the RRDR. The MC will give the CMI for the read return to the CMC early so the CMC can arbitrate for the Runway bus and not waste any states in returning the read data. The CMC will place the CMI for the read return in the RRQ.

Once the CMI for the read return reaches the top of the RRQ, the CMC will pop the CMI for the read return off the RRQ and issue the read return from the RRDR to the RMI block via the

Runway Output Bus. The RMI block will issue the read return on the Runway bus. The CMC will free the read coherency map entry for the read.

1.11.3 Processor Initiated Coherent Read Transactions

Processor initiated coherent reads are received from the Runway bus by the RSI. The RSI forwards the read address to the CMC on the Runway Input Bus. The RSI also forwards the read address to the I/O controller to be snooped in the I/O cache. The CMC allocates a CMI in the read coherency map to keep track of the read until the read return phase or coherency response phase is complete. The CMC places the CMI for the read in the CRQ. If there is nothing in the RQ, then a medium RQ bypass path is enabled to issue the read address directly to the MC with 1 state of delay. Otherwise, the CMC places the CMI for the read in the RQ where it will take a minimum of 2 states to issue to the MC.

When the CMI for the read reaches the top of the RQ, the CMC will check the read coherency map for read/write conflicts. If there is a conflict, then the CMC will issue writes to the MC (if necessary) to clear the conflict. Once the write conflict is cleared, the CMC will pop the CMI for the read off the RQ and issue the read address to the MC.

When the CMI for the read reaches the top of the CRQ and all coherency responses are completed, then the CMC will record the coherency response in the read coherency map and check for read/write conflicts. If there is a conflict, then the CMC will send writes from the WQ to the MC until the conflict is cleared, wait (if necessary) for the read to reach the top of the RQ, pop the CMI for the read off the CRQ, and issue the read address to the MC. Otherwise, the CMC will pop the CMI for the read off the CRQ without issuing the read to the MC. (If there is not a read/write conflict at the end of the coherency response phase, then there will never be one. We are certain that the RQ has sent, or will send, the read to the MC, so the CRQ does not have to do it.) If the read return data for the read is already in the RRDR and the coherency response is COPYOUT, then the CMC will free the read coherency map entry for the read. If the read return data for the read is already in the RRDR and the coherency response is OK or SHARED, then the CMC will place the CMI for the read in the RRQ to schedule a read return. Otherwise, the read return will be scheduled once the read return data is available.

Note: It is possible for the RQ to issue a read to the MC, later have a read/write conflict, detect the conflict at the end of the coherency response phase and have the CRQ reissue the read to the MC. In this case, the MC must never let the two reads pass each other. The only time the MC will allow two conflicting reads to pass each other is when the first read is marked, the second read is not marked, and the marked read fence is asserted. This can not happen since any read reissued from the CRQ will be marked.

The MC will read data from memory and place the read return data in the RRDR. The MC will give the CMI for the read return to the CMC early so the CMC can arbitrate for the Runway bus and not waste any states in returning the read data. If the coherency response phase is complete, the CMC will place the CMI for the read return in the RRQ to schedule a read return.

Once the CMI for the read return reaches the top of the RRQ, the CMC will pop the CMI for the read return off the RRQ and issue the read return from the RRDR to the RMI block via the Runway Output Bus. The RMI block will issue the read return on the Runway bus. The CMC will free the read coherency map entry for the read.

1.11.4 I/O Controller Initiated Write Transactions

I/O Controller initiated writes are sent to the Runway Output Bus on the **rob_addr_data** bus. The RMI masters the write on Runway to be reflected back into Astro by the RSI. From this point on, the IOC initiated write looks like a Runway initiated write.

1.11.5 I/O Controller Initiated Coherent Read Transactions

I/O Controller initiated coherent reads are sent to the RMI on the **rob_addr_data** bus. The RMI masters the read on Runway to be snooped by all processors and reflected back into Astro by the RSI. The middle part of an I/O controller initiated coherent read looks identical to a processor initiated coherent read described in **Section 1.11.3**. After the read return data hits the Runway bus, it is reflected onto the Runway Input Bus by the RSI block and sent to the I/O controller.