Rawhide Bus Spec

Revision 1.1

Change bars from previous Revision 1.0

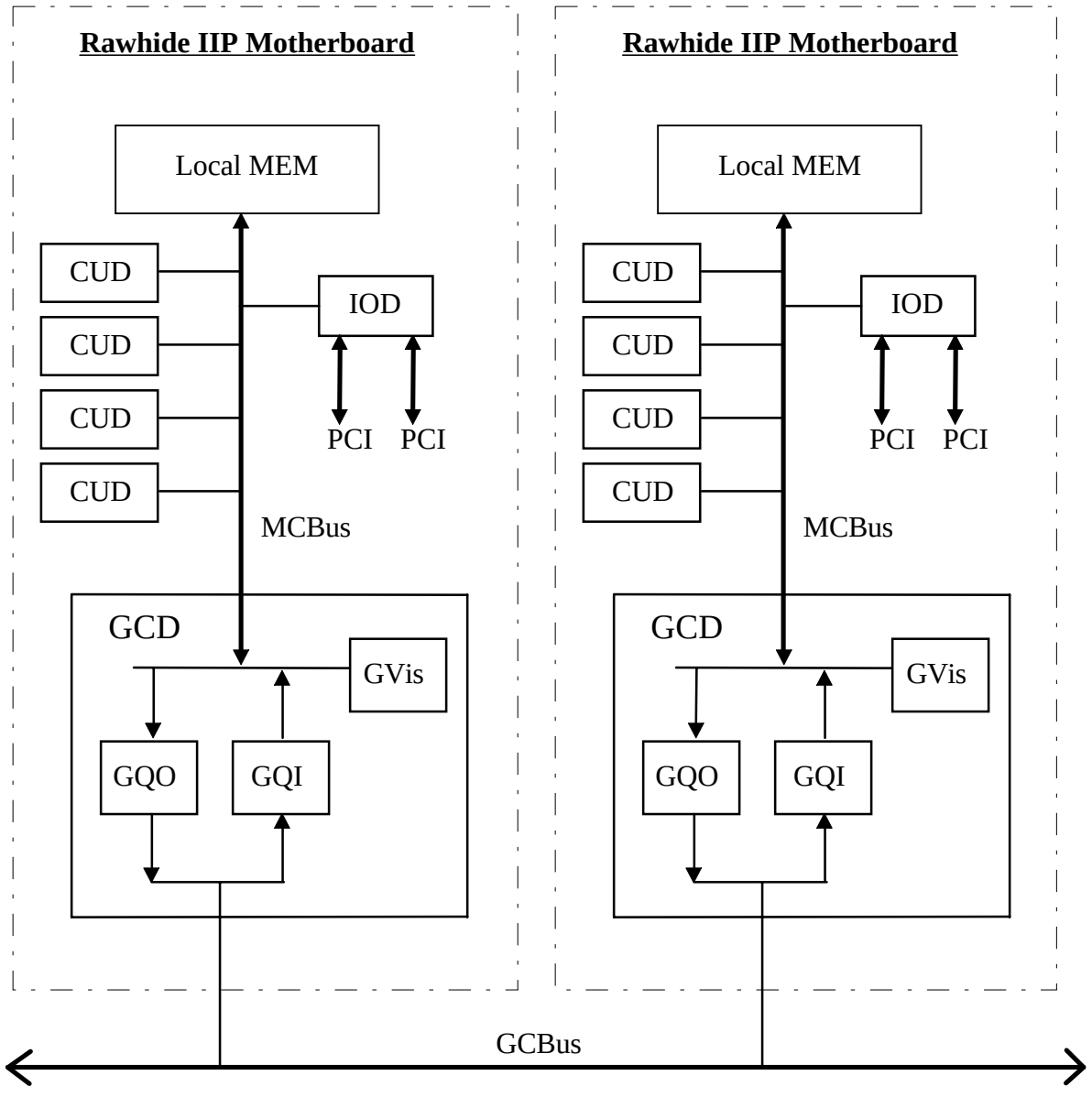
Glenn Herdeg

POBOXA::HERDEG

May 1, 1995

DIGITAL CONFIDENTIAL

Rawhide System Busses (two IIP's shown)



MCBus Overview

The MCBus (Module Connection Bus or Memory Connection Bus) is a synchronous interconnect to allow any of the following building blocks to be connected on a Motherboard: CUD's (No-Bcache CPU Daughtercard or B-Cache CPU Daughtercard), local memory modules (SDSIMMs or ADSIMMs), IOD's (IO Daughtercards), and a GCD (Global Connection Interface, see note about No-Bcache CUD restriction below). The architecture of the MCBus will support 0-4 CUD's, 0-4 IOD's (with a maximum of 6 CUD's and IOD's combined), 0-4 pairs of memory modules (0,2,4,6 or 8 SDSIMMs or ADSIMMs), and 0-1 GCD's.

The GCD provides an interface to the GCBus (Global Connection Bus). The architecture of the MCBus will support up to 8 nodes on the GCBus.

Note:
The No-Bcache CUD will not support the GCD in NUMA mode (support for RM mode is TBD).

- **MCBus node ID assignments**

  The MCBus node ID is assigned as follows for a maximum of 7 bus commanders (memory is assigned to an 8th node for arbitration purposes only) . Configuration rules are TBD.

MID<2:0>	IIP	PIO
0 0 0	GCD	GCD
0 0 1	Memory	Memory
0 1 0	CUD-0	CUD-0
0 1 1	CUD-1	CUD-1
1 0 0	IOD-0	IOD-0
1 0 1	IOD-1	IOD-1
1 1 0	CUD-2	IOD-2
1 1 1	CUD-3	IOD-3

- **Datapath overview**

  Data is transferred on the MCBus datapath in 4 consecutive TBD 15ns cycles during either a Write, Read, or Fill transaction. Write data is always driven in four preassigned data cycles relative to the start of the Write transaction. Read data is returned either in four preassigned data cycles relative to the start of the Read transaction (non-pended Read) or at a later time during a separate MCBus Fill transaction (pended Read). Fill data is always driven in four preassigned data cycles relative to the start of the Fill transaction. A dead cycle will usually be inserted on the MCBus datapath after each burst of four data cycles to allow for tri-state turnoff/turnon. The exception to this is if two non-pended Reads are returning data from the same SDRAM. In this case the dead cycle is not inserted and data can be issued at the maximum bandwidth of the MCBus of TBD 1GB/sec.

- **Basic Addressing scheme**

The basic addressing scheme of the MCBus is described below. For more details see the Rawhide System Programmer's Manual.

⇒ **MC_ADR<39>=0 Memory Space (cacheable)**

MC_ADR<38:4> = address of 64-byte cache block, with both Read and Write wrapping on the four MCBus data cycles as follows:

MC_ADR <39>	MC_CMD <5:0>	MC_ADR <5:4>	Data order
0	Any R/W-Mem	0 0	0, 1, 2 ,3
0	Any R/W-Mem	0 1	1, 0, 3, 2
0	Any R/W-Mem	1 0	2, 3, 0, 1
0	Any R/W-Mem	1 1	3, 2, 1, 0

⇒ **MC_ADR<39>=1 IO Space (non-cacheable)**

MC_ADR<38:36> = GID<2:0> (GCBus node ID -- defaults to 111 without a GCD)
MC_ADR<35:33> = MID<2:0> (MCBus node ID -- see MCBus node ID assignments above)
MC_ADR<32:4> = Node-specific IO addressing.

Note:
The No-Bcache CUD will decode any WriteThru-IO transaction it issues with MC_ADR<32:28> asserted (ADR<32:28>=11111) as an IO_INTR Acknowledge.

Also note that CUD nodes do not support returning data to IO Space Reads that target the CUD nodespace defined above (there are no CUD CSR's visible from the MCBus). IO Space Writes to CUD nodes are used for various types of Interrupt transactions.

The wrapping on the four MCBus data cycles for IO Space transactions is as follows:

MC_ADR <39>	MC_CMD <5:0>	MC_ADR <5>	MC_ADR <4>	Data order
1	WriteThru-IO	0	0	0, 1, --, --
		1	0	2, 3, --, --
		X	1	illegal
1	WriteMask-IO	0	0	0, 1, 2, 3
		0	1	illegal
		1	X	illegal
1	Read-IO	0	0	0, 1, --, --
		0	1	1, 0, --, --
		1	0	2, 3, --, --
		1	1	3, 2, --, --
1	ReadPeer-IO	0	0	0, 1, 2, 3
		0	1	illegal
		1	X	illegal

- **Arbitration scheme**

Arbitration for the MCBus is implemented by a central arbiter located on the motherboard. It operates with a mixture of Round-robin, fixed, bus hold, and bus parking features. Each of the 8 nodes indicated by MID<2:0> sends a request signal to the central arbiter on MC_REQ_L<7:0>. The arbiter sends a grant signal to each of the 8 nodes on the corresponding signal MC_GRANT_L<7:0>.

There are two arbitration priority schemes as follows (one for the IIP motherboard and one for the PIO motherboard).

IIP Arbitration (maximum 4 CUD's, 2 IOD's)

MID<2:0>	IIP	MC_REQ_L<n> MC_GRANT_L<n>
0 0 0	GCD	6
0 0 1	Memory	7
0 1 0	CUD-0	0
0 1 1	CUD-1	1
1 0 0	IOD-0	4
1 0 1	IOD-1	5
1 1 0	CUD-2	2
1 1 1	CUD-3	3

MID to MC_REQ_L<n> mapping (IIP)

IIP Arbitration Algorithm (listed in descending order of priority)

⇒ At absolute highest priority, all nodes currently granted the MCBus will continue to be granted the bus as long as they keep MC_REQ_L asserted. The exception to this is that MC_GRANT_L<6> will not be held when MC_REQ_L<6> is held if MC_REQ_L<7> is also asserted. Note that MC_REQ_L<7> is only used to issue MemRefresh transactions, so the GCD can still issue atomic read-modify-write sequences by keeping MC_REQ_L<6> asserted. It is the responsibility of each node to assert and deassert MC_REQ_L according to the protocol described in the MCBus Signal descriptions.

⇒ MC_REQ_L<7>. This is used only by memory control logic located on the motherboard to issue Memory Refresh transactions to the Memory modules.

⇒ MC_REQ_L<6>. This is used only by the GCD (if included).

⇒ MC_REQ_L<5:4>. These operate at round-robin amongst themselves, used by IOD<1:0>.

⇒ MC_REQ_L<3:0>. These operate at round-robin amongst themselves, used by CUD<3:0>.

PIO Arbitration (maximum 2 CUD's, 4 IOD's)

MID<2:0>	PIO	MC_REQ_L<n> MC_GRANT_L<n>
0 0 0	GCD	6
0 0 1	Memory	7
0 1 0	CUD-0	0
0 1 1	CUD-1	1
1 0 0	IOD-0	4
1 0 1	IOD-1	5
1 1 0	IOD-2	2
1 1 1	IOD-3	3

MID to MC_REQ_L<n> mapping (PIO)

PIO Arbitration Algorithm (listed in descending order of priority)

⇒ At absolute highest priority, all nodes currently granted the MCBus will continue to be granted the bus as long as they keep MC_REQ_L asserted. The exception to this is that MC_GRANT_L<6> will not be held when MC_REQ_L<6> is held if MC_REQ_L<7> is also asserted. Note that MC_REQ_L<7> is only used to issue MemRefresh transactions, so the GCD can still issue atomic read-modify-write sequences by keeping MC_REQ_L<6> asserted. It is the responsibility of each node to assert and deassert MC_REQ_L according to the protocol described in the MCBus Signal descriptions.

⇒ MC_REQ_L<7>. This is used only by memory control logic located on the motherboard to issue Memory Refresh transactions to the Memory modules.

⇒ MC_REQ_L<6>. This is used only by the GCD (if included).

⇒ MC_REQ_L<5:2>. These operate at round-robin amongst themselves, used by IOD<3:0>.

⇒ MC_REQ_L<1:0>. These operate at round-robin amongst themselves, used by CUD<1:0>.

Additional Arbitration features (for both IIP and PIO)

When none of the MC_REQ_L signals are asserted, the central arbiter will assert MC_GRANT_L to the last CUD that was granted the MCBus (bus parking). Note that even when the bus is granted to a node, a new transaction will not begin until MC_REQ_L is asserted.

The central arbiter on the motherboard asserts the signal MC_CA to indicate that a new transaction is beginning on the MCBus. On an idle MCBus, this signal may be asserted either one or three cycles after MC_REQ_L is asserted. If MC_GRANT_L<n> is already asserted on an idle bus to a CUD (due to the bus-parking feature described above) then when MC_REQ_L<n> is asserted the central arbiter will assert MC_CA one cycle later. If MC_GRANT_L<n> is not asserted on an idle bus then when MC_REQ_L<n> is asserted the central arbiter will assert MC_CA three cycles later.

Since CUD modules operate at lower priority than IOD and GCD modules, it is possible to keep the CUD module from being granted the bus under certain conditions. The MC_LOCKOUT_L signal should be used by a CUD that has issued MC_REQ_L but not received MC_GRANT_L within TBD MCBus cycles. See the description of the MCBus signal MC_LOCKOUT_L for more details.

- **Restrictions on Multiple MCBus transaction from a single node.**

A maximum of two (2) Pended Read-type transactions (Memory or IO) may be issued from the same commander on the MCBus. The FILL transaction for one of the two Reads must be issued on the MCBus before a third Read-type transaction may be issued (or one of the two Reads must receive MC_CNF_L=Ack or MC_CNF_L=Noack). Note the one exception to this below.

A maximum of one (1) Pended Write-type transaction (Memory or IO) may be issued from the same commander on the MCBus. The WritePendAck transaction for the Write must be issued on the MCBus before a second Write-type transaction may be issued (or the Write must receive MC_CNF_L=Ack). Note the one exception to this below.

Note: The IOD may issue the ReadMod-Mem transaction with the Cmdr_Merge bit asserted even if that same IOD has two Pended Read-type transactions outstanding (the GCD will ignore the ReadMod-Mem since a WriteMerge-Mem transaction will follow).

Note: The IOD may issue the WriteIntr-IO transaction even if a Write-type transaction from the same IOD has been issued and the corresponding WritePendAck transaction has not been received. See the description of the WriteIntr-IO transaction for more details.

Note: It is possible that more outstanding transactions than are indicated above may be allowed from an IOD or GCD. This depends on the available buffer space available in the IOD. This issue is TBD.

- **MCBus transactions with MC_ADR<39>=0**

  In the following descriptions, the term "X" implies that the value may be either a "1" or a "0", but good parity must be maintained over these signals (where applicable).

  All MCBus transactions with MC_ADR<39>=0 must honor the memory-bank arbitration rules (in which MC_ADR<6> must be different for any two MCBus transactions issued four cycles apart if both have MC_ADR<39>=0).

  ⇒ **Read0-Mem, Read1-Mem**

  The commander of the transaction is reading a 64-byte block in cacheable space. If the data is located in Local Memory then MC_CNF_L=Ack. If the data is not located in Local Memory then the GCD will return MC_CNF_L=Pend. All CUD's will snoop their caches and return dirty (modified) data. Any CUD wishing to keep a copy of the data will assert MC_SHARED_L.

  If a Read0-Mem transaction receives MC_CNF_L=Pend then the data will be returned by a subsequent FILL0 transaction to the commander from the GCD. If a Read1_Mem transaction receives MC_CNF_L=Pend then a FILL1 transaction will follow. This allows two simultaneous Read transactions to be issued from the same commander.

  Note that the MCBus architecture allows up to 16 uniquely-identifiable Read transactions to be issued from the same commander by using the RID<2:0> Read_Extension bits in the second cycle of the MC_ADR transfer. However, due to buffer size limitations in the GCD and/or IOD, a maximum of 2 outstanding Read-type transactions may be issued from the same commander on the MCBus.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X	1000, 1001 1110, 1111	0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:29	28:25	24	23:4
2	X	X	X	Cmdr GID	Cmdr MID	X	Cmdr RID	X	CA par	X

Read-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **ReadMod0-Mem, ReadMod1-Mem**

This transaction is similar to the Read0-Mem, Read1-Mem transaction except the commander wishes to modify (write to) the 64-byte block. All CUD's will invalidate their copy of the 64-byte block after snooping and will not assert MC_SHARED_L.

If this transaction is the first transaction in the atomic read-modify-write sequence, then the Cmdr_Merge bit should be asserted (MC_ADR<32>=1 in the second half of the address transfer). This is used by the GCD to ignore the ReadMod since a WriteMerge-Mem transaction will follow.

Note:

IOD's should always assert the Cmdr_Merge bit (since all ReadMod-Mem transactions are always the first command in a ReadMod/MemIdle/WriteMerge sequence).

CUD's should always deassert the Cmdr_Merge bit (in which all ReadMod transactions with MC_CNF_L=Pend will wait for the subsequent FILL transaction).

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X	1 0 1 0 1 0 1 1	0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:29	28:25	24	23:5
2	X	X	X	Cmdr GID	Cmdr MID	Cmdr Merge	Cmdr RID	X	CA par	X

ReadMod-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteThru-Mem**

This transaction is used to write an entire (unmasked) 64-byte shared block (write-thru) to cacheable space. The value on MC_BMSK<15:0> must be all 1's (FFFF) for each of the four data cycles. If the data is located in Local Memory then MC_CNF_L=Ack. If the data is not located in Local Memory then the GCD will return MC_CNF_L=Pend. All CUD's will snoop their caches and invalidate their copy of the 64-byte block.

If a WriteThru-Mem transaction receives MC_CNF_L=Ack then the write has successfully completed.

If a WriteThru-Mem transaction receives MC_CNF_L=Pend then the commander will wait for the WriteThru-Mem transaction to be terminated in one of two ways. If the WriteThru-Mem is successful (if it has reached the cache coherence point by being issued on the GCBus) then a subsequent WritePendAck transaction will be issued on the MCBus to the commander from the GCD. If the WriteThru-Mem transaction is unsuccessful (another commander issues a Write to the same 64-byte block on the GCBus) then a subsequent Invalidate transaction will be issued on the MCBus from the GCD causing the commander to issue a ReadMod-Mem in order to reattempt its write.

Note that the GCD may also return MC_CNF_L=Pend for flow control in order to stop subsequent write transactions from the commander of the current WriteThru-Mem transaction. A subsequent WritePendAck transaction will be issued on the MCBus to the commander from the GCD to complete the WriteThru-Mem and allow the commander to continue to issue write transactions.

Between the time Node A issues a WriteThru-Mem transaction with MC_CNF_L=Pend to a given 64-byte address X and the subsequent WritePendAck or Invalidate transaction is received, any ReadMod-Mem or WriteType-Mem from another node to the same 64-byte address X must not cause Node A to invalidate its copy of address X (this restriction only applies to Node A).

Note that each of the MC_BMSK<15:0> signals has a pullup resistor on the Motherboard so nodes that only issue full 64-byte writes are not required to drive the MC_BMSK<15:0> signals (they will default to all 1's when they are not driven by any node).

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X 0	0 1 1 0	0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

WriteThru-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteBack-Mem**

This transaction is used to write an entire (unmasked) 64-byte victimized block (write-back) to Local Memory. The value on MC_BMSK<15:0> must be all 1's (FFFF) for each of the four data cycles. Note that CUD's need not snoop to invalidate their caches on this transaction.

All WriteBack-Mem transactions will receive MC_CNF_L=Ack indicating that the write has successfully completed.

Note that each of the MC_BMSK<15:0> signals has a pullup resistor on the Motherboard so CUD nodes that only issue full 64-byte writes are not required to drive the MC_BMSK<15:0> signals (they will default to all 1's when they are not driven by any node).

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X 1	0 1 1 0	0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

WriteBack-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteFull-Mem**

This transaction is used to write a full (unmasked) 64-byte block to cacheable space. The value on MC_BMSK<15:0> must be all 1's (FFFF) for each of the four data cycles. All CUD's will snoop their caches and invalidate their copy of the 64-byte block.

Normally the WriteFull-Mem transaction will receive MC_CNF_L=Ack. However, MC_CNF_L=Pend may be returned for flow control in order to stop subsequent write transactions from the commander of the current WriteFull-Mem transaction. A subsequent WritePendAck transaction will be issued on the MCBus to the commander to complete the WriteFull-Mem and allow the commander to continue to issue write transactions.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	0 0	0 1 1 1	0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

WriteFull-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WritePart-Mem**

This transaction is used to write one or more unmasked 16-byte alligned blocks in a full 64-byte block to cacheable space (if all 64 bytes are being written then the WriteFull-Mem transaction may be used instead). The value on MC_BMSK<15:0> must be all 1's (FFFF) or all 0's (0000) for each of the four data cycles (any mixture of FFFF or 0000 cycles is allowed). Only those cycles with MC_BMSK<15:0> asserted (FFFF) will be written to memory. The cycles with MC_BMSK<15:0> deasserted (0000) will leave the corresponding (wrapped) 16-byte alligned memory block unchanged. Note that all four data cycles must have good ECC on MC_CHK<15:0>.

If the Cached-CUD contains a dirty copy of the alligned 64-byte block then it keeps that copy dirty (no invalidate is issued to its caches) and asserts MC_WRITE_DIRTY_L causing the IOD to issue a ReadMod/MemIdle/WriteMerge sequence instead. If the Cached-CUD does not contain a dirty copy then it must invalidate its copy of the 64-byte block.

Note:
The WritePart-Mem transaction may only be issued on an MCBus without a No-Bcache CUD node.

Normally the WritePart-Mem transaction will receive MC_CNF_L=Ack. However, MC_CNF_L=Pend may be returned for flow control in order to stop subsequent write transactions from the commander of the current WritePart-Mem transaction. A subsequent WritePendAck transaction will be issued on the MCBus to the commander to complete the WritePart-Mem and allow the commander to continue to issue write transactions.

Note:
This revision of the MCBus Spec does not allow a CUD to have a dirty copy of a block unless it is located in the local memory, therefore if MC_CNF_L=Pend then MC_WRITE_DIRTY_L will always be deasserted during the WritePart-Mem transaction.

	MC_CMD<5:4>	MC_CMD<3:0>		MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	1 0	0 1 1 1		0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>		39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X		X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

WritePart-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteMerge-Mem**

The WriteMerge-Mem transaction may only be issued as the third transaction in a ReadMod-Mem/ MemIdle/ WriteMerge-Mem sequence. Note that CUD's need not snoop to invalidate their caches on this transaction because the preceding ReadMod-Mem will cause an invalidate snoop.

If the preceding ReadMod-Mem transaction receives MC_CNF_L=Ack, then all 64 bytes of data must be valid during the WriteMerge-Mem transaction and MC_BMSK<15:0> must be all 1's (FFFF) for each of the four data cycles.

If the preceding ReadMod-Mem transaction receives MC_CNF_L=Pend, only those bytes of data indicated by MC_BMSK<15:0> must be valid. The remainder of the data is don't care but ECC must be correct for all four data cycles.

Normally the WriteMerge-Mem transaction will receive MC_CNF_L=Ack. However, MC_CNF_L=Pend may be returned for flow control in order to stop subsequent write transactions from the commander of the current WriteMerge-Mem transaction. A subsequent WritePendAck transaction will be issued on the MCBus to the commander to complete the WriteMerge-Mem and allow the commander to continue to issue write transactions.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X 1	0 1 1 1	0	Memory Adr<38:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

WriteMerge-Mem: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **Invalidate**

This transaction is issued by the GCD to invalidate the 64-byte cache block indicated by MC_ADR<38:6>. All CUD's will snoop their caches and invalidate their copy of the 64-byte block.

Note that MC_ADR<39>=0 for the Invalidate transaction and note also that it must honor the memory-bank arbitration rules (in which MC_ADR<6> must be different for any two MCBus transactions issued four cycles apart if both have MC_ADR<39>=0). A subtle side effect of this is that an Invalidate may not be issued to the same 64-byte location as the MCBus transaction issued four cycles before or after it. This removes some complexity on managing the boundary conditions on nodes issuing transactions to the same address as the snooped address.

The value on MC_CNF_L should be ignored on an Invalidate transaction.

There are no data cycles associated with the Invalidate transaction.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X 0	0 0 1 0	0	Memory Adr<38:6>	X

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

Invalidate: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **SetDirty**

This optional transaction will only be issued by the Cached-CUD in order to transition the 64-byte cache block indicated by MC_ADR<38:6> to the DIRTY state. Note that the other CUD's need not snoop to invalidate their caches on this transaction (it is only used by the Cached-CUD for cache coherency private to the Cached-CUD module).

Note that MC_ADR<39>=0 for the SetDirty transaction and note also that it must honor the memory-bank arbitration rules (in which MC_ADR<6> must be different for any two MCBus transactions issued four cycles apart if both have MC_ADR<39>=0). A subtle side effect of this is that a SetDirty may not be issued to the same 64-byte location as the MCBus transaction issued four cycles before or after it. This removes some complexity on managing the boundary conditions on nodes issuing transactions to the same address as the SetDirty address.

The value on MC_CNF_L should be ignored on a SetDirty transaction.

There are no data cycles associated with the SetDirty transaction.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X	0 1 0 1	0	Memory Adr<38:6>	X

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

SetDirty: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **ReadFILL0, ReadFILL1**

The ReadFill transaction is a combination of a local memory Read and a Fill to a previous Pended local memory Read.

The ReadFILL0 transaction is used to return data to a Pended Read0-Mem or a Pended ReadMod0-Mem transaction while the ReadFILL1 transaction is used to return data to a Pended Read1-Mem or a Pended ReadMod1-Mem transaction. During the first half of the MCBus address transfer the MID of the commander of the Pended Read is driven as shown. (Note that this is a copy of the "commander" information driven during the second half of the address transfer during the Pended Read transaction). The local memory address bits <32:4> are also driven as shown. Note that the upper memory address bits <38:33> are not included in this transaction.

If the Pended Read transaction that initiated the ReadFill was a ReadMod0-Mem or a ReadMod1-Mem, then MC_CMD<4>=1 during the first half of the command/address transfer. Otherwise MC_CMD<4>=0 during the first half of the command/address transfer.

MC_CMD<5>=0 during the first half of the command/address transfer.

The value on MC_CNF_L should be ignored on a ReadFill transaction.

Nodes should not snoop their caches on this transaction. The four data cycles of the ReadFill transaction occur in the same relative four cycles as a Read-type and/or Fill transaction. Note that MC_RHOLD_L<n>, MC_DIRTY_EN_L<n>, MC_NOT_OUR_DIRTY_L and MC_DIRTY_L must/will be deasserted and MC_DAT_IN_CKE_L will be asserted during the relevant cycles of the ReadFill transaction.

During the ReadFill transaction, the value on MC_SHARED_L will leave the block being returned shared if MC_SHARED_L is asserted and non-shared if MC_SHARED_L is deasserted. See the description of the signal MC_SHARED_L for more details.

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:6	5:4
1	0 , ReadMod	1 1 0 0 1 1 0 1	0	X	Read Cmdr MID	Memory Adr<32:6>	Wrap Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:29	28:25	24	23:4
2	X	X	X	RFILL Cmdr GID	RFILL Cmdr MID	X	X	X	CA par	X

ReadFill: MC_CMD & MC_ADR -- First Half, Second Half

- **MCBus transactions with MC_ADR<39>=1**

MCBus transactions with MC_ADR<39>=1 do not have to honor the memory-bank arbitration rules (any address may be issued on MC_ADR<38:4> if MC_ADR<39>=1).

⇒ **Read0-IO, Read1-IO**

The commander of the transaction is reading a 32-byte block in non-cacheable space. If this transaction receives MC_CNF_L=Pend, then the data will be returned in the first two cycles of the subsequent FILL0 or FILL1 transaction (the last two data cycles of the Fill are driven with don't care data with good ECC on all four data cycles). If this transaction receives MC_CNF_L=Noack, then no subsequent FILL transaction will be issued.

There are no data cycles associated with the Read0-IO or Read1-IO transactions.

The first two data cycles of the FILL contain the 32-bytes indicated by MC_ADR<39:5>. The data will be wrapped based on MC_ADR<4>.

If the target of the Read-IO is an IOD, additional sparse/dense address information may be indicated by the value on the MC_ADR signals in the first and second half of the address transfer. See the Rawhide System Programmer's Manual for more details on the decoding of MC_ADR<32:4> during the first half of the address transfer. During the second half of the MCBus address transfer the quadword mask information (if necessary) is placed on MC_ADR<11:8> (the Int4_Valid<3:0> bits).

If the Read-IO transaction is reading the WHAMI register in the IOD or the GCD, then the information found on MC_ADR<38:33, 20:16, 14, 7, 4> during the second half of the address transfer is returned as data during the subsequent FILL transaction on MC_DAT<13:0> as indicated below. The use of these bits is module-specific. See the Rawhide System Programmer's Manual and the Cached-CUD spec for more details.

	MC_CMD<5:4>	MC_CMD<3:0>		MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X	1 0 0 0 1 0 0 1		1	IO-space Adr<38:6>	Adr<5:4>

	MC_CMD<5:4>	MC_CMD<3:0>		39	38:36	35:33	32	31:29	28:25	24
2	X	X		X	Cmdr GID WHAMI<5:3>	Cmdr MID WHAMI<2:0>	X	Cmdr RID	X	CA par

	23:21	20:16	15	14	13:12	11:8	7	6:5	4
2	X	WHAMI D<13:9>	X	WHAMI D<8>	X	Int4_Valid <3:0>	WHAMI D<7>	X	WHAMI D<6>

Read-IO: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **ReadPeer0-IO, ReadPeer1-IO**

IOD 64-byte Peer-to-peer Reads.

MC_ADR<5:4>=00 for the ReadPeer-IO transaction (read wrapping is not supported).

MC_CNF_L=Pend for all ReadPeer-IO transactions.

There are no data cycles associated with the ReadPeer0-IO or ReadPeer1-IO transactions.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X	1 0 1 0 1 0 1 1	1	Peer-to-peer IO Adr<38:6>	0 0

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:29	28:25	24	23:4
2	X	X	X	Cmdr GID	Cmdr MID	X	Cmdr RID	X	CA par	X

ReadPeer-IO: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **FILL0, FILL1**

The FILL0 transaction is used to return data to a Pended Read0-type transaction (either Memory-space or IO-space) while the FILL1 transaction is used to return data to a Pended Read1-type transaction. During the first half of the MCBus address transfer the GID, MID and RID (the Read_Extension bits) of the commander of the Pended Read are driven as shown. (Note that this is a copy of the "commander" information driven during the second half of the address transfer during the Read-type transaction).

If the Read-type transaction that initiated the FILL was a ReadMod0-Mem or a ReadMod1-Mem, then MC_CMD<4>=1 during the first half of the command/address transfer. Otherwise MC_CMD<4>=0 during the first half of the command/address transfer.

If the Read-type transaction that initiated the FILL was a Read0-IO, Read1-IO, ReadPeer0-IO or ReadPeer1-IO, then MC_CMD<5>=1 during the first half of the command/address transfer. Otherwise MC_CMD<5>=0 during the first half of the command/address transfer.

If the data being returned is valid and no errors have occurred (Successful FILL), then MC_ADR<32>=0 during the first half of the address transfer. If the data being returned has an error or the address was not found (NXM) then MC_ADR<32>=1 to indicate a FILL Error.

The four data cycles of the FILL transaction occur in the same relative four cycles as a Read-type transaction. Note that MC_RHOLD_L<n>, MC_DIRTY_EN_L<n>, MC_NOT_OUR_DIRTY_L and MC_DIRTY_L must/will be deasserted and MC_DAT_IN_CKE_L will be asserted during the relevant cycles of the FILL transaction.

The value on MC_CNF_L should be ignored on a FILL transaction.

If the Read-type transaction that initiated the FILL was a Read0-Mem, Read1-Mem, ReadMod0-Mem or a ReadMod1-Mem, then the value on MC_SHARED_L will leave the block being returned shared if MC_SHARED_L is asserted and non-shared if MC_SHARED_L is deasserted. MC_SHARED_L must be deasserted if the Read-type transaction that initiated the FILL was a Read-IO or ReadPeer-IO transaction. See the description of the signal MC_SHARED_L for more details.

Note:
The IOD should issue all FILL transactions with MC_CMD<5:4>=10 and MC_SHARED_L deasserted.

Note that the data on the FILL transaction must be returned in the correct wrapped order indicated by the Read-type transaction. Note also that all four data cycles must have good ECC on MC_CHK<15:0> even if the data is not valid (only 32 bytes of data or no data due to a FILL error).

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:29	28:4
1	ReadIO, ReadMod	1 1 0 0 1 1 0 1	1	Read Cmdr GID	Read Cmdr MID	ERR	Read Cmdr RID	X

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:29	28:25	24	23:4
2	X	X	X	FILL Cmdr GID	FILL Cmdr MID	X	X	X	CA par	X

Fill: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteThru-IO**

The WriteThru-IO transaction is used by the CUD for all IO-space Writes including "Programmed IO Writes" to the IOD, "CSR Writes" to the IOD and GCD, "Interprocessor Interrupts" from one CUD to another, and "Interrupt Acknowledge" Writes from a CUD to itself or from a CUD to the IOD.

The IOD dues not use the WriteThru-IO transaction (see the WriteIntr-IO transaction for more details of IOD IO-space writes).

The first two data cycles contain the 32-bytes indicated by MC_ADR<39:5>. MC_ADR<4> must be deasserted for all WriteThru-IO transactions (data will not be wrapped based on MC_ADR<4>).

WriteThru-IO transactions from a CUD that target an IOD node may be used for "Programmed IO Writes" in which all or part of a 32-byte data block is written to non-cacheable space. The mask is determined by the sparse/dense protocol as described in the Rawhide System Programmer's Manual. The four Int4_Valid<3:0> bits corresponding to the first data cycle are placed on MC_ADR<11:8> during the second half of the MCBus address transfer and the four Int4_Valid<3:0> bits corresponding to the second data cycle are driven on the MC_IMSK<3:0> signals. Note that the last two data cycles are ignored and the value on MC_BMSK<15:0> during all four data cycles will be ignored.

WriteThru-IO transactions from a CUD that target an IOD or GCD may be used for "CSR Writes".

WriteThru-IO transactions from a CUD to itself, to another CUD, or to the IOD may be used for various types of interrupts and interrupt acknowledge operations. See the description of MCBus Interrupts for more details.

Normally the WriteThru-IO transaction will receive MC_CNF_L=Ack. However, MC_CNF_L=Pend may be returned for flow control in order to stop subsequent write transactions from the commander of the current WriteThru-IO transaction. A subsequent WritePendAck transaction will be issued on the MCBus to the commander to complete the WriteThru-IO and allow the commander to continue to issue write transactions.

Note that the four Int4_Valid<3:0> bits are ignored for all WriteThru-IO transactions except for "Programmed IO Writes". Note also that all four data cycles must have good ECC on MC_CHK<15:0> even if the data is not valid.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5>	MC_ADR<4>
1	X 0	0 1 1 0	1	IO-space Adr<38:6>	Adr<5>	0

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:12	11:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	Int4_Valid <3:0>	X

WriteThru-IO: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteIntr-IO**

The WriteIntr-IO transaction is used only by the IOD for issuing IO Interrupts and NodeHalt Interrupts. See the description of MCBus Interrupts for more details.

The value on MC_CNF_L should be ignored on a WriteIntr-IO transaction.

Each IOD may only issue a maximum of TBD two WriteIntr-IO transactions until a IO_ACK is received (see the description of MCBus Interrupts for more information). When the first IO_ACK is received the IOD may issue a third WriteIntr-IO and consequently may have no more than TBD two outstanding WriteIntr-IO transactions at a time.

Since the WriteIntr-IO transaction will never receive MC_CNF_L=Pend, the GCD must have enough buffer space to hold TBD two WriteIntr-IO transactions from each IOD node on the MCBus even if the GCD has asserted MC_CNF_L=Pend to a previous Write-type transaction from the IOD issuing the WriteIntr-IO transaction. Note that there is no data associated with this transaction and only a subset of the full address information (only MC_ADR<38:33, 31> needs to be stored in the GCD).

Note that the data during the WriteIntr-IO transaction is ignored and the ECC on MC_CHK<15:0> is ignored during all four data cycles (ECC is not checked during the data cycles of the WriteIntr-IO transaction).

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:28>	MC_ADR<27:4>
1	X 1	0 1 1 0	1	IO-space Adr<38:28>	X

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:12	11:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

WriteIntr-IO: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WriteMask-IO**

The WriteMask-IO transaction is used for "Peer-to-peer Writes" in which all or part of a 64-byte data block is written to non-cacheable space. The value on MC_BMSK<15:0> in each of the four data cycles on the MCBus indicates which byte of data is valid during that data cycle. MC_ADR<5:4>=00 for the WriteMask-IO transaction (write wrapping is not supported).

Normally the WriteMask-IO transaction will receive MC_CNF_L=Ack. However, MC_CNF_L=Pend may be returned for flow control in order to stop subsequent write transactions from the commander of the current WriteMask-IO transaction. A subsequent WritePendAck transaction will be issued on the MCBus to the commander to complete the WriteMask-IO and allow the commander to continue to issue write transactions.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X 0	0 1 1 1	1	Peer-to-peer IO Adr<38:6>	0 0

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	Quadword Mask	X

WriteMask-IO: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **MemIdle**

The MemIdle transaction is used to insert a NOP between two MCBus transactions that reference the same memory-bank (in which MC_ADR<39>=0 and MC_ADR<6> match). Note that MC_ADR<39>=1 for the MemIdle transaction. See the description of MCBus Memory-space Masked Writes below for more details.

The value on MC_CNF_L should be ignored on a MemIdle transaction.

There are no data cycles associated with the MemIdle transaction.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:6>	MC_ADR<5:4>
1	X	0 0 0 0	1	X	X

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32:25	24	23:16	15:8	7:4
2	X	X	X	Cmdr GID	Cmdr MID	X	CA par	X	X	X

MemIdle: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **WritePendAck**

The WritePendAck transaction is used to successfully terminate a Pended WriteThru-Mem transaction or to allow the commander of any Pended Write-type transaction to continue sending Writes (for flow control). See the description of the WriteThru-Mem transaction above for more details.

During the first half of the MCBus address transfer the GID and MID of the commander of the Pended Write are driven as shown.

The value on MC_CNF_L should be ignored on a WritePendAck transaction.

If the WritePendAck is in response to a WriteThru-Mem transaction, then the value on MC_CMD<4> (the SHARED bit) during the first half of the cmd/address transfer will leave the block being written shared if MC_CMD<4>=1 or non-shared if MC_CMD<4>=0.

If the WritePendAck is in response to a WriteThru-Mem transaction, then the value on MC_CMD<5> (the INVAL bit) during the first half of the cmd/address transfer will leave the block being written valid if MC_CMD<5>=0 or invalid if MC_CMD<1>=1.

Note:
The IOD should Issue all WritePendAck transactions with MC_CMD<5:4>=00.

There are no data cycles associated with the WritePendAck transaction.

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:4
1	INVAL, SHARED	0 0 1 0	1	Write Cmdr GID	Write Cmdr MID	X	X

	MC_CMD<5:4>	MC_CMD<3:0>	39	38:36	35:33	32	31:25	24	23:4
2	X	X	X	WPAck Cmdr GID	WPAck Cmdr MID	X	X	CA par	X

WritePendAck: MC_CMD & MC_ADR -- First Half, Second Half

⇒ **MemRefresh**

The MemRefresh is used by the Local Memory control logic on the motherboard to issue a Refresh command to all the Local Memory modules. The Local Memory control logic issues a Refresh sequence on the MCBus every TBD 13 μsec by asserting MC_REQ_L<7> (the highest priority MCBus request level) and issuing the following three atomic MCBus transactions:

MemRefresh	(NOP to local memory)
MemRefresh	(CBR Refresh to all DRAMS)
MemRefresh	(NOP to local memory)

The first MemRefresh (NOP to local memory) guarantees that all banks within the Synchronous DRAMs are precharged before the CBR Refresh command is issued to the SDRAMs and the third MemRefresh (NOP to local memory) guarantees that all banks within the SDRAMs are precharged before a new bank is activated by a MCBus transaction immediately following this un-interruptable sequence.

The value on MC_CNF_L will be ignored on a MemRefresh transaction.

There are no data cycles associated with the MemRefresh transaction.

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	MC_ADR<38:4>
1	X	0 0 1 1	1	X

	MC_CMD<5:4>	MC_CMD<3:0>	MC_ADR<39>	38:25	24	23:4
2	X	X	X	X	CA Par	X

MemRefresh: MC_CMD & MC_ADR -- First Half, Second Half

- **MCBus Memory-space Masked Writes**

  The Local Memory on the MCBus does not support writing less than a full unmasked 128-bit (alligned 16-byte) data cycle. Any node wishing to write less than a full 64-byte block must issue either the read-modify-write atomic sequence described below or a WritePart-Mem transaction.

  By keeping MC_REQ_L asserted a node can hold onto the MCBus (see the arbitration scheme description above). Therefore, by keeping MC_REQ_L asserted, the following sequence can be used to write a partial 64-byte block:

  ReadMod-Mem
  MemIdle
  WriteMerge-Mem

- **MCBus Interrupts**

  The various types of MCBus interrupts and interrupt acknowledge operations are implemented using the WriteIntr-IO transaction (from an IOD) or the WriteThru-IO transaction (from a CUD). The address decoding of the first half of the address transfer is as follows (the GID and MID uniquely identify the target of the Interrupt):

Type	Cmdr	Target	MC_CMD<5:0>	39	38:36	35:33	32 31 30 29 28	27:4
IO_INTR	IOD	CUD	WriteIntr-IO	1	Target GID	Target MID	0 1 1 1 1	X
IO_ACK	CUD	IOD	WriteThru-IO	1	Target GID	Target MID	1 1 1 1 1	X
NodeHalt_INTR	IOD	CUD	WriteIntr-IO	1	Target GID	Target MID	0 0 1 1 1	X
NodeHalt_ACK	CUD	itself	WriteThru-IO	1	Target GID	Target MID	1 0 1 1 1	X
IntervalTimer_ACK	CUD	itself	WriteThru-IO	1	Target GID	Target MID	1 0 0 0 0	X
IP_INTR	CUD	another CUD	WriteThru-IO	1	Target GID	Target MID	0 0 0 0 1	X
IP_ACK	CUD	itself	WriteThru-IO	1	Target GID	Target MID	1 0 0 0 1	X
MCHK_ACK	CUD	itself	WriteThru-IO	1	Target GID	Target MID	1 0 0 1 0	X
Not used (IRQ0_ACK)	CUD	itself	WriteThru-IO	1	Target GID	Target MID	1 0 0 1 1	X
DTAG_ENABLE	CUD	B-Cache CUD	WriteThru-IO	1	Target GID	Target MID	1 0 1 0 0	X
DTAG_DISABLE	CUD	B-Cache CUD	WriteThru-IO	1	Target GID	Target MID	1 0 1 0 1	X
CCC_REG (note1)	CUD	B-Cache CUD	WriteThru-IO	1	Target GID	Target MID	1 0 1 1 0	CCC
NOP_ACK (note2)	CUD	itself	WriteThru-IO	1	Target GID	Target MID	1 1 1 1 0	X

Interrupts and Interrupt Acknowledge: MC_CMD & MC_ADR -- First Half

Note1: CCC_REQ<27:4> = Cached-CUD Configuration Register (Write-only) -- see Cached-CUD Spec.

Note2: A CUD can write to itself at this address with no side effects (trigger or debugging address)

⇒ **I/O Generated Interrupt/Acknowledge**

When the IOD wishes to forward an IO_INTR from one of its busses to a CUD to service the interrupt (one and only one CUD may receive IO_INTR's from a unique IOD) it will write the vector of the interrupt to memory using a read-modify-write sequence (see the description of MCBus Memory-space Masked Writes) followed by an "IO_INTR" (the WriteIntr-IO transaction) in which the target CUD is indicated by the GID and MID in the first half of the MCBus address. See the Rawhide System Programmer's Manual for more details on IO Interrupts.

By keeping MC_REQ_L asserted a node can hold onto the MCBus (see the arbitration scheme description above). Therefore, by keeping MC_REQ_L asserted, the following sequence should be used to send an IO_INTR:

ReadMod-Mem
MemIdle
WriteMerge-Mem
WriteIntr-IO

When the CUD receives the WriteIntr-IO transaction it increments its IO_INTR counter up to a maximum of 32 IO Interrupts system-wide (with a GCD) or 8 IO Interrupts per Motherboard (without a GCD). Whenever the IO_INTR counter is non-zero, the IRQ signal (as specified in the Rawhide System Programmer's Manual) to the CPU will be asserted (level-sensitive). When the same CUD issues the WriteThru-IO transaction on the MCBus for the IO_ACK it decrements its IO_INTR counter.

⇒ **Node Halt Interrupt/Acknowledge**

When the IOD wishes to forward a NodeHalt Interrupt to one and only one target CPU, it will issue a WriteIntr-IO transaction with the first half of the address as described above. The target CUD will set its NodeHalt_INTR flop and clear it on the subsequent WriteThru-IO transaction on the MCBus (to itself) for the NodeHalt_ACK.

⇒ **Interval Timer Acknowledge**

The CUD will set its IntTim_INTR flop when it receives a signal from the motherboard. The CUD will clear it on the subsequent WriteThru-IO transaction on the MCBus (to itself) for the IntervalTimer_ACK.

⇒ **Inter-processor Interrupt (IP_INTR)**

When a CUD wishes to send an IP_INTR to one and only one target CUD, it will issue a WriteThru-IO transaction with the first half of the address as described above. The target CUD will set its IP_INTR flop and clear it on the subsequent WriteThru-IO transaction on the MCBus (to itself) for the IP_ACK.

MCBus Signals

A description of each MCBus signal is included here, with precise timing diagrams showing the waveforms of these signals in the MCBus Transaction section.

Included in the timing diagrams are the signals CC_<2:7> (Command Cycle 2-7). These are not actual MCBus signals, but are included as a suggested implementation of tracking the command sequencing of the MCBus. These signals provide a reference cycle number for the MCBus signal descriptions and the MCBus transaction descriptions. The algorithm used by the central arbiter on the motherboard and the CUD to produce these internal signals is as follows:

```
(syntax expressions)
        "!"             = NOT
        "&"             = AND
        "#"             = OR
        ":="            = asserted on the next rising edge of MCLK
(temporary definitions)
        MC_CA           =  !MC_CA_L;
        MC_STALL        =  !MC_STALL_L;
(logic equations)
        CC_2            :=  (  MC_CA) # ( MC_STALL & CC_2);
        CC_3            :=  ( !MC_STALL & CC_2) # ( MC_STALL & CC_3);
        CC_4            :=  ( !MC_STALL & CC_3);
        CC_5            :=  (  CC_4);
        CC_6            :=  (  CC_5) # ( MC_STALL & CC_6);
        CC_7            :=  ( !MC_STALL & CC_6) # ( MC_STALL & CC_7);
```

Note that the signals CC_4 and CC_5 will only be asserted for one cycle maximum, while CC_2, CC_3, CC_6, and CC_7 may be asserted for more than one cycle if MC_STALL_L is asserted. Also note that the central arbiter on the motherboard will assert MC_CA_L in such a manner that CC_2 and CC_6 may be asserted in the same cycle (two nested MCBus transactions) or CC_3 and CC_7 may be asserted in the same cycle.

The central arbiter logic is located on the motherboard. The signals called "Input signals" below are received by a MCBus node, "Output signals" are transmitted by a MCBus node and "Bi-directional signals" (either tri-state or wire-or) may be both received and/or transmitted by a MCBus node.

Output signals to central arbiter (single driver)

- **MC_REQ_L<n>**

  A node asserts MC_REQ_L<n> (see MCBus node ID assignments) whenever it wishes to issue a MCBus transaction and the following restrictions are met.

  Memory-bank arbitration rule: The value of MC_ADR<6> must be different for any two MCBus transactions issued four cycles apart if both have MC_ADR<39>=0. To accomplish this, any node must deassert MC_REQ_L<n> in cycles CC_2, CC_3, CC_4, and CC_5 of the current MCBus transaction if its pending Adr<39> and the current MC_ADR<39> both equal 0 and its pending Adr<6> matches the current MC_ADR<6>.

  Maintain request rule: The value (either asserted or deasserted) of MC_REQ_L<n> in cycle CC_2 of the current transaction must be held unchanged during all instances of cycles CC_2, CC_3, CC_4, and CC_5 of the current transaction.

  Note that if MC_CA_L is not asserted in cycle CC_5, then any node may assert MC_REQ_L<n> in cycle CC_6 or beyond. If MC_CA_L is asserted in cycle CC_5, then this cycle becomes cycle 1 of the next transaction and the above rules apply to cycle CC_2 of this transaction.

- **MC_RHOLD_L<n> (CUD only)**

  This output signal is asserted in cycle CC_5 of a Read-Mem or ReadMod-Mem transaction for one or more cycles by a node that can't respond with MC_SHARED_L or MC_DIRTY_EN_L<n> in cycle CC_6. This signal should not be asserted under normal operations since it increases the latency of the local read data and causes MC_STALL_L to be asserted.

- **MC_DIRTY_EN_L<n> (CUD only)**

  During a Read-Mem or ReadMod-Mem transaction, this output signal is asserted in cycle CC_6 for a minimum of five cycles if the data is found dirty (modified) in the cache on the CUD node. It is possible to delay the sampling of MC_DIRTY_EN_L<n> by the use of MC_RHOLD_L<n>.

  During a Read-Mem or ReadMod-Mem transaction, MC_DIRTY_EN_L<n> must remain asserted for a minimum of four CC_7 cycles (note that CC_6 may be asserted for more than one cycle) and must be deasserted in the same cycle that the first cycle of dirty data is being returned on the MCBus.

- **MC_WRITE_STALL_L<n> (CUD only)**

  During any Write-type transaction, this output signal may be asserted in cycle CC_2 for one or more cycles by a node that can't provide data on the MC_DAT and MC_CHK signals at the appropriate time.

  The input signal MC_WHOLD_L overrides this signal, so a node that is delaying driving data due to MC_WHOLD_L does not need to also assert MC_WRITE_STALL_L<n>. However, both MC_WHOLD_L and MC_WRITE_STALL_L<n> may be asserted at the same time.

  MC_WRITE_STALL_L<n> must be deasserted in the same cycle in which the first data cycle is driven onto the MCBus. Note that as with all data transfers, once the first data is valid on the MCBus the remaining three data cycle must immediately follow (MC_WRITE_STALL_L<n> must remain deasserted for the remainder of the transaction).

Input signals from central arbiter (single driver to individual nodes)

- **MC_GRANT_L<n>**

  This input signal is used as a qualifier along with the assertion of MC_CA_L to determine if node <n> is the current bus master. If it is not already asserted (due to bus-parking) it will be asserted in the cycle before MC_CA_L and will always be held unchanged until the third cycle after MC_CA_L (a minimum of four cycles total).

- **MC_ADR_OE_L<n>**

  This is an asynchronous input signal and should be connected directly to the output enable pin of the output drivers for MC_CMD<5:0> and MC_ADR<39:4>. It is an early asynchronous version of MC_GRANT_L<n>. Because it is asynchronous it must not be sampled on MCLK for any other control purposes.

- **MC_CA_L**

  This input signal indicates that a new transaction is beginning (cycle 1 of the current transaction).

- **MC_WHOLD_L**

  This input signal is asserted in cycle CC_4 of an MCBus Write. It is also asserted in cycles CC_4, CC_5, CC_6, CC_7, and CC_8 (the cycle after CC_7) of a non-dirty MCBus Read (and longer for a dirty MCBus Read). It is used as a qualifier along with the assertion of MC_CA_L and MC_GRANT_L<n> to keep a node <n> from driving Write data onto MC_DAT when a previous Read or Write is still using the data bus.

- **MC_STALL_L**

  This signal is an input to each node's MCBus state machine and freezes the cycle counter CC_n for a variety of cases. MC_STALL_L may be asserted by the central arbiter in the following cycles:

  ⇒ In cycle CC_3 of a MCBus Read or Fill transaction for one cycle if the immediately preceding transaction is a Fill or a Local memory Read to a different SDRAM (to allow a dead cycle on the SDRAM data outputs and/or MC_DAT when changing between drivers).

  ⇒ In cycle CC_3 of a MCBus Write transaction for one or more cycles when MC_WHOLD_L is asserted from the previous Read or Write transaction (to allow a dead cycle on MC_DAT when changing between drivers).

  ⇒ In cycle CC_3 of a MCBus Write transaction for one or more cycles when MC_WRITE_STALL_L is asserted by the node issuing the MCBus Write transaction.

  ⇒ In cycle CC_6 of a MCBus Read transaction for the same number of cycles in which any of the MC_RHOLD_L signals are asserted (to freeze the read data pipeline if a node delays the sampling of shared or dirty and then the response is not dirty).

  ⇒ In cycle CC_7 of a MCBus Read transaction for one more cycle than the number of cycles in which any of the MC_DIRTY_EN_L signals are asserted (to account for the extra read data cycles required to return the dirty data).

⇒ In cycle CC_7 of all MCBus Write transactions for exactly one cycle to satisfy the SDRAM's nine-cycle Write cycle time restriction.

⇒ In cycle CC_7 of all MCBus Read transactions for exactly one cycle to satisfy some SDRAM vendor's nine-cycle Read cycle time restriction (only if the static configuration signal SIM_SDRAM_R9_L is asserted by any of the installed Memory modules).

⇒ In cycle CC_7 of all MCBus FILL transactions for exactly one cycle (to allow a dead cycle on MC_DAT when changing between drivers).

- **MC_DAT_IN_CKE_L**

  This input signal is deasserted in cycle CC_6 for the same number of cycles in which any of the MC_RHOLD_L signals are asserted. It is used to delay the sampling of the first cycle of non-dirty MCBus Read data and freeze this first data cycle in the input flop of the MCBus data transceiver until MC_DAT_IN_CKE_L is asserted.

- **MC_DIRTY_L (non-CUD only)**

  During a Read-Mem or ReadMod-Mem transaction, this input signal is asserted in cycle CC_7 for a minimum of five cycles if the data will be returned from a CUD instead of Local memory. If MC_DIRTY_L is asserted in cycle CC_7 then the data being returned on the MC_DAT during the usual four data cycles must be ignored. Instead, the node receiving the read data must wait until MC_DIRTY_L deasserts to receive the four consecutive cycles of Read data. MC_DIRTY_L will be deasserted in the cycle after the cycle in which the first cycle of dirty data is being returned on the MCBus.

- **MC_NOT_OUR_DIRTY_L<n> (CUD only)**

  This input signal is asserted with the same timing as MC_DIRTY_L except it will not be asserted to the node which asserted the original MC_DIRTY_EN_L<n> signal.

Bi-directional signals (tri-state, multiple drivers)

- **MC_CMD<5:0>**
  The timing of the MCBus Command signals MC_CMD<5:0> is the same as the MCBus Address signals. See the description of MC_ADR<39:4> for more details.

First Half MC_CMD		First Half MC_ADR	MCBus Transaction	No-BCache CUD cmdr	B-Cache CUD cmdr	IOD cmdr	GCD cmdr
5 4	3 2 1 0	39					
x x	0 0 0 0	0	(illegal)				
x x	0 0 0 0	1	MemIdle		Y	Y	Y
x x	0 0 0 1	x	(illegal)				
x 0	0 0 1 0	0	Invalidate				Y
x 1	0 0 1 0	0	(illegal)				
0 0	0 0 1 0	1	WritePendAck (NoShared)			Y	Y
0 1	0 0 1 0	1	WritePendAck (Shared)				Y
1 x	0 0 1 0	1	WritePendAck (Invalidate)				Y
x x	0 0 1 1	0	(illegal)				
x x	0 0 1 1	1	MemRefresh				
x x	0 1 0 0	x	(illegal)				
x x	0 1 0 1	0	SetDirty		Y		
x x	0 1 0 1	1	(illegal)				
x 0	0 1 1 0	0	WriteThru-Mem	Y	Y		
x 0	0 1 1 0	1	WriteThru-IO	Y	Y		Y
x 1	0 1 1 0	0	WriteBack-Mem	Y	Y		
x 1	0 1 1 0	1	WriteIntr-IO			Y	Y
0 0	0 1 1 1	0	WriteFull-Mem			Y	Y
1 0	0 1 1 1	0	WritePart-Mem (B-Cache CUD only)			Y	Y
x 0	0 1 1 1	1	WriteMask-IO			Y	Y
x 1	0 1 1 1	0	WriteMerge-Mem			Y	Y
x 1	0 1 1 1	1	(illegal)				
x x	1 0 0 0	0	Read0-Mem	Y	Y	Y	Y
x x	1 0 0 0	1	Read0-IO	Y	Y		Y
x x	1 0 0 1	0	Read1-Mem	Y	Y	Y	Y
x x	1 0 0 1	1	Read1-IO	Y	Y		Y
x x	1 0 1 0	0	ReadMod0-Mem	Y	Y	Y	Y
x x	1 0 1 0	1	ReadPeer0-IO			Y	Y
x x	1 0 1 1	0	ReadMod1-Mem	Y	Y	Y	Y
x x	1 0 1 1	1	ReadPeer1-IO			Y	Y
0 -	1 1 0 0	0	ReadFill0				Y
0 0	1 1 0 0	1	FILL0 (due to Read0-Mem)				Y
0 1	1 1 0 0	1	FILL0 (due to ReadMod0-Mem)				Y
1 0	1 1 0 0	1	FILL0 (due to Read0/Peer0-IO)			Y	Y
1 1	1 1 0 0	1	(illegal)				
0 -	1 1 0 1	0	ReadFill1				Y
0 0	1 1 0 1	1	FILL1 (due to Read1-Mem)				Y
0 1	1 1 0 1	1	FILL1 (due to ReadMod1-Mem)				Y
1 0	1 1 0 1	1	FILL1 (due to Read1/Peer1-IO)			Y	Y
1 1	1 1 0 1	1	(illegal)				
x x	1 1 1 0	0	Read0-Mem	Y	Y		Y
x x	1 1 1 0	1	(illegal)				
x x	1 1 1 1	0	Read1-Mem	Y	Y		Y
x x	1 1 1 1	1	(illegal)				

- **MC_ADR<39:4>**

  Whenever the input signal MC_ADR_OE_L<n> is asserted, node<n> will drive MC_ADR<39:4> and MC_CMD<5:0>. However, the value driven is undefined except during the cycle in which MC_CA_L is asserted, the preceeding cycle, and the cycle following MC_CA_L. In these three cycles, the commander node will drive the MC_ADR and MC_CMD signals with two separate pieces of address and control information referred to as the two "halfs" of the command/address transfer.

  In the cycle in which MC_CA_L is asserted, the asserting edge of MC_CA_L will cause all nodes to close their input address latches sampling MC_ADR<39:4> and MC_CMD<5:0>. This latches the cmd/address information present during the first half of the MCBus address transfer. Then the same asserting edge of MC_CA_L may cause the commander node to drive MC_ADR<39:4> and MC_CMD<5:0> with the address and control information present during the second half of the MCBus command/address transfer *or* this information may be driven from the clock at the beginning of cycle CC_2 if feasible. The commander will continue to drive the second half information until two cycles after the cycle in which MC_CA_L is asserted.

  The first half command/address information can be found in the MCBus descriptions above.

  The second half command/address information common to all transactions is described below. Second half information unique to a given transaction can be found in the specific MCBus descriptions above.

  ⇒ The Cmdr_GID is the GCBus node ID<2:0> (defaults to 111 without a GCD).

  ⇒ The Cmdr_MID is the MCBus node ID<2:0>.

  ⇒ The Cmdr_RID is the Read_Extension ID<2:0> (optional, default to 000 if not used).

  ⇒ The CA_par bit is odd parity over MC_CMD<3:0> and MC_ADR<39:4> in the first half of the command/address transfer. Note this is over the first half only, and MC_CMD<5:4> is not included.

  All nodes must generate command/address parity as described above. The IOD and GCD will check parity on all transactions. The CUDs will not check command/address parity on any transaction.

- **MC_DAT<127:0>**

  These signals along with MC_CHK<15:0> make up the MCBus datapath. See the Datapath overview above.

  On Read-Mem or ReadMod-Mem transactions, the first data is valid in the input flop of the MCBus data transceiver (assuming MC_DAT_IN_CKE_L is used to freeze the data as indicated in the description of MC_DAT_IN_CKE_L above) under the following conditions:

  ⇒ If MC_CNF_L=Ack and MC_DIRTY_L (or MC_NOT_OUR_DIRTY_L) is deasserted in cycle CC_7 then the first data is valid in the input flop of the MCBus data transceiver during cycle CC_7. The rest of the data is valid in the next three consecutive cycles.

  ⇒ If MC_CNF_L=Pend then the first data is valid in the input flop of the MCBus data transceiver during cycle CC_7 of the subsequent FILL0 or FILL1 transaction. The rest of the data is valid during the next three consecutive cycles.

On Read-IO transactions the first data is valid in the input flop of the MCBus data transceiver during cycle CC_7 of the subsequent FILL0 or FILL1 transaction. The second data is valid in the next consecutive cycle (see ECC restriction below).

On Write-type transactions the first data is valid in the input flop of the MCBus data transceiver during cycle CC_3 if MC_STALL_L is deasserted. The rest of the data is valid during the next three consecutive cycles (or only one cycle for WriteThru-IO transactions).

The following transactions must have good ECC on MC_CHK<15:0> during all four data cycles:

WriteThru-Mem
WriteThru-IO
WriteBack-Mem (with MC_FALSE_WRITEBACK_L deasserted)
WriteFull-Mem
WritePart-Mem
WriteMask-IO
WriteMerge-Mem
Read0-Mem, Read1-Mem
ReadMod0-Mem, ReadMod1-Mem
FILL0, FILL1
ReadFill0, ReadFill1

The following transactions do not require good ECC on MC_CHK<15:0> during any of the four data cycles:

WriteBack-Mem (with MC_FALSE_WRITEBACK_L asserted)
WriteIntr-IO

The following transactions do not have any data cycles associated with them:

MemIdle
Invalidate
WritePendAck
MemRefresh
SetDirty
Read0-IO, Read1-IO
ReadPeer0-IO, ReadPeer1-IO

- **MC_CHK<15:0>**

These signals are the ECC codes used by the EV5. The timing of MC_CHK<15:0> is identical to MC_DAT<127:0>. See the description of MC_DAT<127:0> above for more information.

- **MC_BMSK<15:0> (non-CUD only)**

These signals are used only by the IOD and GCD. The timing of MC_BMSK<15:0> is identical to MC_DAT<127:0>. When MC_BMSK<n>=1 the corresponding byte on MC_DAT<127:0> is valid. When MC_BMSK<n>=0 the corresponding byte is invalid.

Note that each of the MC_BMSK<15:0> signals has a pullup resistor on the Motherboard so nodes that only issue unmasked writes are not required to drive the MC_BMSK<15:0> signals (they will default to all 1's when they are not driven by any node).

- **MC_IMSK<3:0>**

  These signals are used only for the WriteThru-IO transaction (for “Programmed IO Writes”). MC_IMSK<3:0> is valid during the same cycle as the second write data cycle on MC_DAT. MC_IMSK<3:0> must follow the same OE timing as MC_DAT (it must be tri-state during the same cycles as MC_DAT).

Bidirectional signals (tri-state wire-or, multiple drivers)

- **MC_CNF_L**

  The confirmation signal MC_CNF_L is sampled in cycle CC_4. It should be driven at least one cycle before cycle CC_4 (the settling time is longer than most other MCBus signals because it is a multi-drop wire-or signal). It may also be asserted in cycles CC_2 (and CC_3) if desired and will be ignored.

MC_CNF_L	First Half MC_ADR39	Confirmation	Applies to these MCBus transactions...
1	0	Ack {Mem Read}	Read0-Mem, Read1-Mem, ReadMod0-Mem, ReadMod1-Mem
1	0	Ack {Mem Write}	WriteThru-Mem, WriteBack-Mem, WriteFull-Mem, WritePart-Mem, WriteMerge-Mem
1	1	Noack {IO Read}	Read0-IO, Read1-IO, ReadPeer0-IO, ReadPeer1-IO
1	1	Ack {IO Write}	WriteThru-IO, WriteMask-IO
0	0	Pend {Mem Read}	Read0-Mem, Read1-Mem, ReadMod0-Mem, ReadMod1-Mem
0	0	Pend {Mem Write}	WriteThru-Mem, WriteBack-Mem, WriteFull-Mem, WritePart-Mem, WriteMerge-Mem
0	1	Pend {IO Read}	Read0-IO, Read1-IO, ReadPeer0-IO, ReadPeer1-IO
0	1	Pend {IO Write}	WriteThru-IO, WriteMask-IO

- **MC_FALSE_WRITEBACK_L**

  This signal is asserted by the No-Bcache CUD during the WriteBack-Mem transaction to disable ECC checking of all four data cycles. It is asserted whenever the EV5 has failed to return a WriteBack command to the pins of the EV5 following a snoop (the EV5 may have issued a WriteThru-Mem, WriteThru-IO, or Read-IO command instead and will re-issue the original WriteBack-Mem command later). As a result it is possible that the No-Bcache CUD will drive undefined data with bad ECC during the four cycles of the WriteBack-Mem transaction.

  The timing of MC_FALSE_WRITEBACK_L is the same as MC_CNF_L.

- **MC_WRITE_DIRTY_L**

  During a WritePart-Mem transaction, MC_WRITE_DIRTY_L is asserted (by a B-Cache CUD) if the 64-byte block is found dirty in its caches. See the description of the WritePart-Mem transaction for more information.

  MC_WRITE_DIRTY_L is sampled in the first occurance of cycle CC_7. It should be driven at least one cycle before cycle CC_7 (the settling time is longer than most other MCBus signals because it is a multi-drop wire-or signal). MC_WRITE_DIRTY_L may also be asserted in cycles CC_6 or CC_8 (the cycle after CC_7) if desired and will be ignored.

- **MC_SHARED_L (non-IOD only)**

  During a Read-Mem or ReadMod-Mem transaction, MC_SHARED_L is sampled in cycle CC_7 (note that CC_7 may be asserted for more than one cycle and MC_SHARED_L must be valid during all occurances of cycle CC_7). MC_SHARED_L should be driven at least one cycle before cycle CC_7 (the settling time is longer than most other MCBus signals because it is a multi-drop wire-or signal). MC_SHARED_L may also be asserted in cycle CC_6 or CC_8 (the cycle after CC_7) if desired and

will be ignored. It is possible to delay the sampling of MC_SHARED_L by the use of MC_RHOLD_L (during a Read-type transaction only).

During a FILL transaction MC_SHARED_L is also sampled in cycle CC_7 (note that CC_7 may be asserted for more than one cycle and MC_SHARED_L must be valid during all occurances of cycle CC_7), however MC_RHOLD_L may not be asserted. See the description of the FILL transaction for more information.

- **MC_LOCKOUT_L (asynchronous)**

  This signal is asserted by a MCBus node to disable new MCBus requests from all other MCBus nodes in order to get guaranteed access to the MCBus within a finite period of time.

  One use of this signal may be if the CUD has asserted MC_REQ_L<n> for TBD MCBus cycles without receiving MC_GRANT_L<n> due to the MCBus arbitration priority scheme. Since the IOD's are given higher priority than the CUD's it is possible to starve out the CUD's under severe IOD traffic.

  All nodes must receive MC_LOCKOUT_L and must not assert MC_REQ_L due to a new transaction unless they are also asserting MC_LOCKOUT_L. Nodes that have already asserted MC_REQ_L (and may have removed them temporarily due to a bank conflict) may continue to assert MC_REQ_L until their current transaction has completed successfully. Also, multi-sequence transaction (ReadMod/MemIdle/WriteMerge, etc) may be completed if MC_LOCKOUT_L is asserted during the sequence.

  MC_LOCKOUT_L may be asserted and deasserted at any time asynchronous to the MCBus clock. It must be synchronized by all receiving nodes.

MCBus Signal Timing Specifications

These timing values are very crude estimates and are included only to establish a basic understanding of the timing of the MCBus signals. More complete timing analysis of each MCBus signal is being conducted by the Rawhide Signal Integrity group.

These timing values do not include clock skew.

MCBus Signal class	Signal type	Tsetup (std load) [cycles]	Clk->Q (std load) [cycles]
MC_REQ_L [cud]	output from [cud]	4.0 [1]	
MC_REQ_L [non-cud]	output from [non-cud]	7.0 [1]	
MC_RHOLD_L [cud]	output from [cud]	4.0 [1]	
MC_DIRTY_EN_L [cud]	output from [cud]	4.0 [1]	
MC_WRITE_STALL_L [cud]	output from [cud]	9.0 [1]	
MC_GRANT_L	input to [all dcards]		9.0 [1]
MC_ADR_OE_L	input to [all dcards]		9.0 [1]
MC_CA_L	input to [all dcards]		5.0 [1]
MC_WHOLD_L	input to [all dcards]		9.0 [1]
MC_STALL_L	input to [all dcards]		5.0 [1]
MC_DAT_IN_CKE_L	input to [all dcards]		9.0 [1]
MC_DIRTY_L [non-cud]	input to [non-cud]		5.0 [1]
MC_NOT_OUR_DIRTY_L [cud]	input to [cud]		5.0 [1]
MC_CMD, MC_ADR	bi-direct	20.0 [2]	
MC_DAT, MC_CHK	bi-direct	6.0 [1]	
MC_BMSK	bi-direct	6.0 [1]	
MC_IMSK	bi-direct	15.0 [2]	
MC_CNF_L	bi-direct	15.0 [2]	
MC_FALSE_WRITEBACK_L	bi-direct [from No-cache CUD]	15.0 [2]	
MC_WRITE_DIRTY_L	bi-direct [from B-cache CUD]	15.0 [2]	
MC_SHARED_L	bi-direct	15.0 [2]	
MC_LOCKOUT_L	bi-direct	asynchronous	

MCBus Transactions

Diagrams 1 through TBD 19 have been generated from real structural hardware simulations of a version of the IIP containing a fully structural EV5NC CUD with a behavioral model of the EV5 running Alpha macrocode, a fully structural IIP motherboard (containing the central arbiter), a collection of structural SDSIMMs with a behavioral model of the SDRAMs, and a number of DECSIM behavioral MCBus commanders.

The first signal, MCLK, is the MCBus reference clock running at up to TBD 66 MHz (TBD 15ns).

The next vectored signal, CC_<2:7>, is the Command Cycle 2-7. These are not actual MCBus signals, but are included as a suggested implementation of tracking the command sequencing of the MCBus. These signals provide a reference cycle number for the MCBus signal descriptions and the MCBus transaction descriptions. The algorithm used by the central arbiter on the motherboard and the CUD to produce these internal signals is described in the MCBus signal descriptions above.

The next collection of signals, SIM_XXX, are the signals on the SDSIMMs (Local memory module) around which the MCBus has been optimized.

The next collection of signals, MC_XXX, are the MCBus signals that are driven and received by each CUD, IOD, GCD and central arbiter.

The last collection of signals, EV_XXX (at NODE0 in this case) are unique to the EV5NC CUD and are shown as a basis for understanding one CUD design. Other CUDs, the IOD, and the GCD will have unique signals not shown in these diagrams.

Note that some signal may temporarily switch in the middle of a cycle. Most signals are only sampled on the rising edge of MCLK (all except MC_ADR_OE_L) so these glitches should be ignored.

MCBus local Read transactions

Reads to the same local memory bank

Two consecutive MCBus local (non-pended) Read transactions to the same memory bank are shown in Diagram 1. When the MCBus is idle, a node <n> wishing to issue a Read asserts MC_REQ_L<n>. If MC_ADR_OE_L<n> is asserted by the central arbiter the node will also drive MC_CMD<5:0> and MC_ADR<39:4> with the desired command and address.

The first read begins when MC_CA_L is asserted by the central arbiter (for the purpose of discussion this cycle is referred to as cycle 1). One of the MC_GRANT_L<7:0> signals is also asserted by the central arbiter indicating which node has issued the transaction. The assertion of MC_CA_L also guarantees that MC_CMD and MC_ADR have been driven for at least two cycles. Also in cycle 1, irregardless of the address or command, the central arbiter asserts SIM_ROW_CS_L to all SDSIMMs along with SIM_CMD<2:0> = ACT (memory bank activate command) and SIM_ADR<11:0> containing the desired SDRAM row address (SIM_CMD and SIM_ADR are guaranteed to be valid for two cycles). Each SDSIMM will gate SIM_ROW_CS_L along with some upper-order address bits and its Memory slot number to assert the actual CS_L signal to the desired SDRAMs.

In cycles CC_2, CC_3, CC_4, and CC_5 all nodes must deassert MC_REQ_L<n> unless their pending MCBus transaction is to a different bank than the current MCBus transaction in cycle 1. Also in cycle CC_2, the central arbiter switches the SIM_CMD and SIM_ADR signals from “row” to “column”.

Beginning in cycle CC_2 and held during cycle CC_3 and CC_4, MC_CNF_L is driven by multiple responders (MC_CNF_L is not shown). The GCD asserts MC_CNF_L=Pend if the address was not to Local Memory in which case the read data will be returned later from the GCD with a FILL transaction or the IOD asserts MC_CNF_L=Pend if the address was to itself in which case the read data will be returned later from the IOD with a FILL transaction. If MC_CNF_L=Ack, then Read Data will be returned as indicated in the pre-assigned cycles for non-pended read data.

Also in cycle CC_3, the central arbiter asserts SIM_COL_CS_L to all SDSIMMs along with SIM_CMD<2:0> = READA (burst-4 read with auto-precharge) and SIM_ADR containing the desired column address (SIM_CMD and SIM_ADR are guaranteed to be valid for two cycles).

In cycle CC_4, the central arbiter asserts MC_WHOLD_L for five cycles in case the next MCBus transaction is a Write-type. This signal keeps write data from being driven onto the MCBus while read data is being returned from the previous Read transaction. Also in cycle CC_4, the central arbiter switches the SIM_CMD and SIM_ADR signals from “column” back to “row”.

In cycle CC_5, the data drivers on the SDSIMMs that will be returning the read data are turned on and drive MC_DAT<127:0> and MC_CHK<15:0> with undefined data.

In cycle CC_6, the first 16-bytes of data are driven onto MC_DAT<127:0> and MC_CHK<15:0>. Also, cycle CC_6 is the earliest that nodes may assert MC_REQ_L<n> for transactions that are to the same bank as the current transaction.

In the next consecutive three cycles, the second, third, and fourth 16-byte data cycles are driven onto MC_DAT<127:0> and MC_CHK<15:0>, and then the data drivers on the SDSIMM turn off in the following cycle.

The second Read begins in cycle 9, which now becomes the new cycle 1, and the above sequence repeats itself for the next nine cycles.