(A5)

VESA®

VUMA Proposal

(Draft)

Video Electronics Standards Association

2150 North First Street, Suite 440
San Jose, CA 95131-2029

Phone: (408) 435-0333
FAX: (408) 435-8225

VESA Unified Memory Architecture Hardware Specifications Proposal

Version: 1.0p

Document Revision: 0.4p

October 31, 1995

Important Notice: This is a **draft** document from the Video Electronics Standards Association (VESA) Unified Memory Architecture Committee (VUMA). It is only for discussion purposes within the committee and with any other persons or organizations that the committee has determined should be invited to review or otherwise contribute to it. It has not been presented or ratified by the VESA general membership.

Purpose

To enable core logic chipset and VUMA device designers to design VUMA devices supporting the Unified Memory Architecture.

Summary

This document contains a specification for VUMA devices' hardware interface. It includes logical and electrical interface specifications. The BIOS protocol is described in VESA document VUMA VESA BIOS Extensions (VUMA-SBE) rev. 1.0.

Samsung Exhibit 1025

Scope

Because this is a draft document, it cannot be considered complete or accurate in all respects although every effort has been made to minimize errors.

Intellectual Property

© Copyright 1995 – Video Electronics Standards Association. Duplication of this document within VESA member companies for review purposes is permitted. All other rights are reserved.

Trademarks

All trademarks used in this document are the property of their respective owners. VESA and VUMA are trademarks owned by the Video Electronics Standards Association.

Patents

The proposals and standards developed and adopted by VESA are intended to promote uniformity and economies of scale in the video electronics industry. VESA strives for standards that will benefit both the industry and end users of video electronics products. VESA cannot ensure that the adoption of a standard; the use of a method described as a standard; or the making, using, or selling of a product in compliance with the standard does not infringe upon the intellectual property rights (including patents, trademarks, and copyrights) of others. VESA, therefore, makes no warranties, expressed or implied, that products conforming to a VESA standard do not infringe on the intellectual property rights of others, and accepts no liability direct, indirect or consequential, for any such infringement.

Support For This Specification

If you have a product that incorporates VUMA™, you should ask the company that manufactured your product for assistance. If you are a manufacturer of the product, VESA can assist you with any clarification that you may require. All questions must be sent in writing to VESA via:

(The following list is the preferred order for contacting VESA.)

VESA World Wide Web Page:	*www.vesa.org*
Fax:	(408) 435-8225
Mail:	VESA 2150 North First Street Suite 440 San Jose, California 95131-2029

Acknowledgments

This document would not have been possible without the efforts of the members of the 1995 VESA Unified Memory Architecture Committee and the professional support of the VESA staff.

Work Group Members

Any industry standard requires information from many sources. The following list recognizes members of the VUMA Committee, which was responsible for combining all of the industry input into this proposal.

Chairperson

Rajesh Shakkarwar	OPTi

Members

Jonathan Claman	S3
Jim Jirgal	VLSI Technology Inc.
Don Pannell	Sierra Semiconductor
Wallace Kou	Western Digital
Derek Johnson	Cypress
Andy Daniel	Alliance Semiconductor
Long Nguyen	Oak Technology
Robert Tsay	Pacific Micro Computing Inc.
Sunil Bhatia	Mentor Arc
Peter Cheng	Samsung
Alan Mormann	Micron
Solomon Alemayehu	Hitachi America Ltd.
Larry Alchesky	Mitsubishi
Dean Hays	Weitek

Revision History

Initial Revision 0.1p — Sept. 21 '95

Revision 0.2p — Oct 5 '95

- Added sync DRAM support
- Electrical Section
- Boot Protocol
- Reformatted document

Revision 0.3p — Oct 19 '95

- Graphics controller replaced with VUMA device
- MD[n:0] changed to t/s
- Modified Aux Memory description
- Added third solution to Memory Expansion Problem
- Synch DRAM burst length changed to 2/4
- Modified all the bus hand off diagrams
- Added DRAM Driver Characteristics section

Revision 0.4p — Oct 19 '95

- Sync DRAM Burst Length changed to 1/2/4
- DRAM controller pin multiplexing added
- Changed AC timing parameters

TABLE OF CONTENTS

1.0 INTRODUCTION......6

2.0 SIGNAL DEFINITION......6

2.1 SIGNAL TYPE DEFINITION......7
2.2 ARBITRATION SIGNALS......7
2.3 FAST PAGE MODE, EDO AND BEDO DRAMS......7
2.4 SYNCHRONOUS DRAM......8

3.0 PHYSICAL INTERFACE......8

3.1 PHYSICAL SYSTEM MEMORY SHARING......9
3.2 MEMORY REGIONS......10
3.3 PHYSICAL CONNECTION......11

4.0 ARBITRATION......12

4.1 ARBITRATION PROTOCOL......12
4.2 ARBITER......12
4.3 ARBITRATION EXAMPLES......15
4.4 LATENCIES......18

5.0 MEMORY INTERFACE......19

5.1 MEMORY DECODE......19
5.2 MAIN VUMA MEMORY MAPPING......20
5.3 FAST PAGE EDO AND BEDO......23
5.4 SYNCHRONOUS DRAM......27
5.5 MEMORY PARITY SUPPORT......32
5.6 MEMORY CONTROLLER PIN MULTIPLEXING......32

6.0 BOOT PROTOCOL......32

6.1 MAIN VUMA MEMORY ACCESS AT BOOT......33
6.2 RESET STATE......34

7.0 ELECTRICAL SPECIFICATION......35

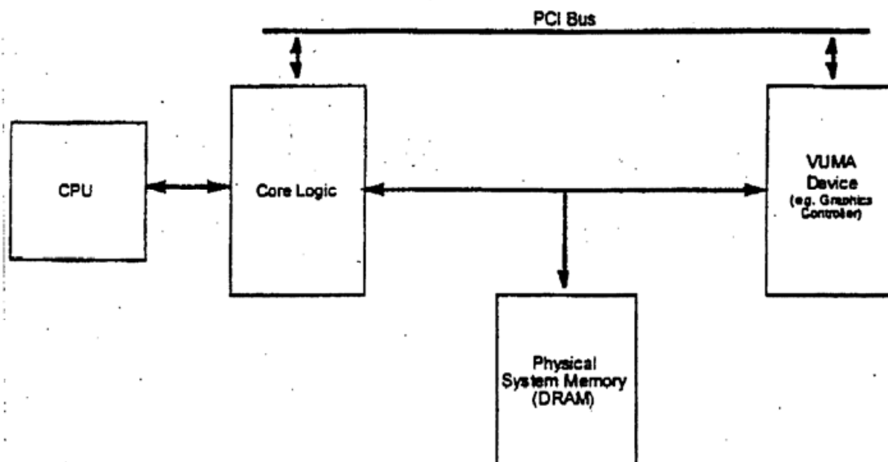
7.1 SIGNAL LEVELS......35
7.2 AC TIMING......35
7.3 PULLUPS......37
7.4 STRAPS......37
7.5 DRAM DRIVER CHARACTERISTICS......37

1.0 Introduction

The concept of VESA Unified Memory Architecture (VUMA) is to share physical system memory (DRAM) between system and an external device, a VUMA device; as shown in Figure 1-1. A VUMA device could be any type of controller which needs to share physical system memory (DRAM) with system and directly access it. One example of a VUMA device is graphics controller. In a VUMA system, graphics controller will incorporate graphics frame buffer in physical system memory (DRAM) or in other words VUMA device will use a part of physical system memory as its frame buffer, thus, sharing it with system and directly accessing it. This will eliminate the need for separate graphics memory, resulting in cost savings. Memory sharing is achieved by physically connecting core logic chipset (hereafter referred to as core logic) and VUMA device to the same physical system memory DRAM pins. Though the current version covers sharing of physical system memory only between core logic and a motherboard VUMA device, the next version will cover an expansion connector, connected to physical system memory DRAM pins. An OEM will be able to connect any type of device to the physical system memory DRAM pins through the expansion connector.

Though a VUMA device could be any type of controller, the discussion in the specifications emphasizes a graphics controller as it will be the first VUMA system implementation.

Figure 1-1 VUMA System Block Diagram



2.0 Signal Definition

2.1 Signal Type Definition

in Input is a standard input-only signal.

out Totem Pole Output is a standard active driver

t/s Tri-State is a bi-directional, tri-state input/output pin.

s/t/s Sustained Tri-state is an active low or active high tri-state signal owned and driven by one and only one agent at a time. The agent that drives an s/t/s pin active must drive it high for at least one clock before letting it float. A pullup is required to sustain the high state until another agent drives it. Either internal or external pullup must be provided by core logic. A VUMA device can also optionally provide an internal or external pullup.

2.2 Arbitration Signals

MREQ#	in out	MREQ# is out for VUMA device and in for core logic. This signal is used by VUMA device to inform core logic that it needs to access shared physical system memory bus.
MGNT#	in out	MGNT# is out for core logic and in for VUMA device. This signal is used by core logic to inform VUMA device that it can access shared physical system memory bus.
CPUCLK	in	CPUCLK is driven by a clock driver. CPUCLK is in for core logic, VUMA device and synchronous DRAM.

2.3 Fast Page Mode, EDO and BEDO DRAMs

RAS#	s/t/s	Active low row address strobe for memory banks. Core logic will have multiple RAS#s to support multiple banks. VUMA device could have a single RAS# or multiple RAS#s. These signals are shared by core logic and VUMA device. They are driven by current bus master.
CAS[n:0]#	s/t/s	Active low column address strobes, one for each byte lane. In case of pentium-class systems n is 7. These signals are shared by core logic and VUMA device. They are driven by current bus master.
WE#	s/t/s	Active low write enable. This signal is shared by core logic and VUMA device. It is driven by current bus master.
OE#	s/t/s	Active low output enable. This signal exists only on EDO and BEDO. This signal is shared by core logic and VUMA device.

		It is driven by current bus master.
MA[11:0]	s/t/s	Multiplexed memory address. These signals are shared by core logic and VUMA device. They are driven by current bus master.
MD[n:0]	t/s	Bi-directional memory data bus. In case of pentium-class systems n is 63. These signals are shared by core logic and VUMA device. They are driven by current bus master.

2.4 Synchronous DRAM

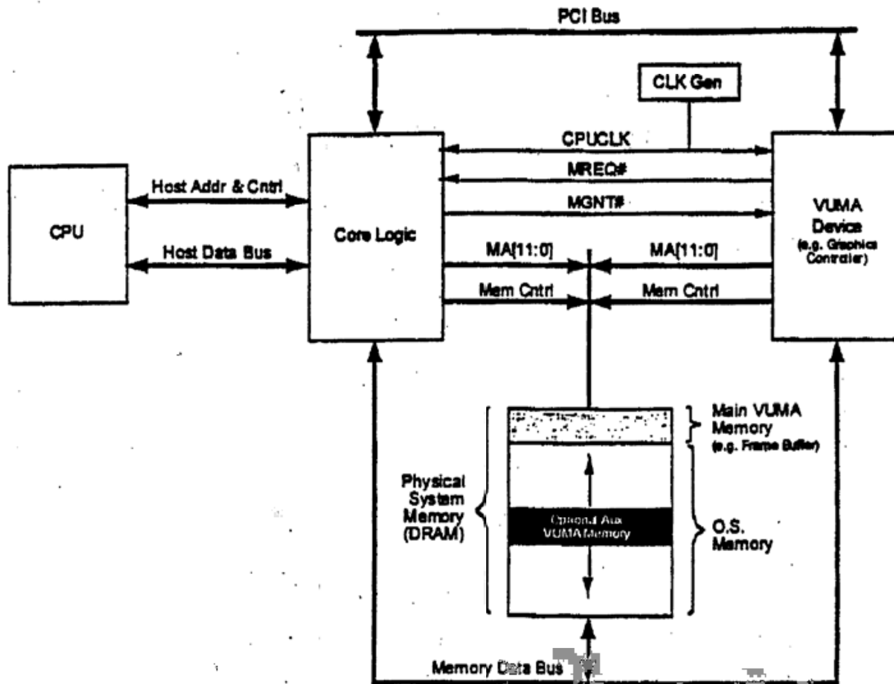
CPUCLK	in	CPUCLK is the master clock input (referred to as CLK in synchronous DRAM data books). All DRAM input/ output signals are referenced to the CPUCLK rising edge.
CKE	s/t/s	CKE determines validity of the next CPUCLK. If CKE is high, the next CPUCLK rising edge is valid; otherwise it is invalid. This signal also plays role in entering power down mode and refresh modes. This signal is shared by core logic and VUMA device. It is driven by current bus master.
CS#	s/t/s	CS# low starts the command input cycle. CS# is used to select a bank of Synchronous DRAM. Core logic will have multiple CS#s to support multiple banks. VUMA device could have a single CS# or multiple CS#s. These signals are shared by core logic and VUMA device. They are driven by current bus master.
RAS#	s/t/s	Active low row address strobe. This signal is shared by core logic and VUMA device. It is driven by current bus master.
CAS#	s/t/s	Active low column address strobe. This signal is shared by core logic and VUMA device. It is driven by current bus master.
WE#	s/t/s	Active low write enable. This signal is shared by core logic and VUMA device. It is driven by current bus master.
MA[11:0]	s/t/s	Multiplexed memory address. These signals are shared by core logic and VUMA device. They are driven by current bus master.
DQM[n:0]	s/t/s	I/O buffer control signals, one for each byte lane. In case of pentium-class systems n is 7. In read mode they control the output buffers like a conventional OE# pin. In write mode, they control the word mask. These signals are shared by core logic and VUMA device. They are driven by current bus master.
MD[n:0]	t/s	Bi-directional memory data bus. In case of pentium-class systems n is 63. These signals are shared by core logic and VUMA device. They are driven by current bus master.

3.0 Physical Interface

3.1 Physical System Memory Sharing

Figure 3-1 depicts the VUMA Block Diagram. Core logic and VUMA device are physically connected to the same DRAM pins. Since they share a common resource, they need to arbitrate for it. PCI/VL/ISA external masters also need to access the same DRAM resource. Core logic incorporates the arbiter and takes care of arbitration amongst various contenders.

Figure 3-1 VUMA Block Diagram



As shown in Figure 3-1, VUMA device arbitrates with core logic for access to the shared physical system memory through a three signal arbitration scheme viz. MREQ#, MGNT# and CPUCLK. MREQ# is a signal driven by VUMA device to core logic and MGNT# is a signal driven by the core logic to VUMA device. MREQ# and MGNT# are active low signals driven and sampled synchronous to CPUCLK common to both core logic and VUMA device.

Core logic is always the default owner and ownership will be transferred to VUMA device upon demand. VUMA device could return ownership to core logic upon completion of its activities or park on the bus. Core logic can always preempt VUMA device from the bus.

VUMA device needs to access the physical system memory for different reasons and the level of urgency of the needed accesses varies. If VUMA device is given the access to the physical system memory right away, every time it needs, the CPU performance will suffer and as it may not be needed right away by the VUMA device, there would not be any improvement in VUMA device performance. Hence two levels of priority are defined viz. low priority and high priority. Both priorities are conveyed to core logic through a single signal, MREQ#.

3.2 Memory Regions

As shown in Figure 3-1, physical system memory can contain two separate physical memory blocks, Main VUMA Memory and Auxiliary (Aux) VUMA Memory. As cache coherency for Main VUMA Memory and Auxiliary VUMA Memory is handled by this standard, a VUMA device can access these two physical memory blocks without any separate cache coherency considerations. If a VUMA device needs to access other regions of physical system memory, designers need to take care of cache coherency.

Main VUMA Memory is programmed as non-cacheable region to avoid cache coherency overhead. How Main VUMA Memory is used depends on the type of VUMA device; e.g., when VUMA device is a graphics controller, main VUMA memory will be used for Frame buffer.

Auxiliary VUMA Memory is optional for both core logic and VUMA device. If supported, it can be programmed as non-cacheable region or write-through region. Auxiliary VUMA Memory can be used to pass data between core logic and VUMA device without copying it to Main VUMA Memory or passing through a slower PCI bus. This capability would have significant advantages for more advanced devices. How Auxiliary VUMA Memory is used depends on the type of VUMA device e.g. when VUMA device is a 3D graphics controller, Auxiliary VUMA memory will be used for texture mapping.

When core logic programs Auxiliary VUMA Memory area as non-cacheable, VUMA device can read from or write to it. When core logic programs Auxiliary VUMA Memory area as write through, VUMA device can read from it but can not write to it.

Both core logic and VUMA device have an option of either supporting or not supporting the Auxiliary VUMA Memory feature. Whether Auxiliary VUMA memory is supported or not should be transparent to an application. The following algorithm explains how it is made transparent. The algorithm is only included to explain the feature. Refer to the latest VUMA VESA BIOS Extensions for the most updated BIOS calls:

1. When an application needs this feature, it needs to make a BIOS call, <Report VUMA

- core logic capabilities (refer to VUMA VESA BIOS Extensions)>, to find out if core logic supports the feature.
2. If core logic does not support the feature, the application needs to use some alternate method.
3. If core logic supports the feature, the application can probably use it and should do the following:

a. Request the operating system for a physically contiguous block of memory of required size.
b. If not successful in getting physically contiguous block of memory of required size, use some alternate method.
c. If successful, get the start address of the block of memory.
d. Read <VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)>, to find out if VUMA device can access the bank in which Auxiliary VUMA Memory has been assigned.
e. If VUMA device can not access that bank, the application needs to either retry the procedure from "step a" to "step c" till it can get Auxiliary VUMA Memory in a VUMA device accessible bank or use some alternate method.
f. If VUMA device can access that bank, make a BIOS call function <Set (Request) VUMA Auxiliary memory (refer to VUMA VESA BIOS Extensions)>, to ask core logic to flush Auxiliary VUMA Memory block of the needed size from the start address from "step c" and change it to either non-cacheable or write through. How a core logic flushes cache for the block of memory and programs it as non-cacheable/ write through is implementation specific.
g. Use VUMA Device Driver, to give VUMA device the Auxiliary VUMA Memory parameters viz. size, start address from "step c" and whether the block should be non-cacheable or write through.

3.3 Physical Connection

A VUMA device can be connected in two ways:

1. VUMA device can only access one bank of physical system memory - VUMA device is connected to a single bank of physical system memory. In case of Fast Page Mode, EDO and BEDO VUMA device has a single RAS#. In case of Synchronous DRAM VUMA device has a single CS#. Main VUMA memory resides in this memory bank. If supported, Auxiliary VUMA Memory can only be used if it is assigned to this bank.

2. VUMA device can access all of the physical system memory - VUMA device has as many RAS# (for Fast Page Mode, EDO and BEDO)/CS# (for Synchronous DRAM) lines as core logic and is connected to all banks of the physical system memory. Both Main VUMA memory and Auxiliary VUMA Memory (if supported) can be assigned to any memory bank.

4.0 Arbitration

4.1 Arbitration Protocol

There are three signals establishing the arbitration protocol between core logic and VUMA device. MREQ# signal is driven by VUMA device to core logic to indicate it needs to access the physical system memory bus. It also conveys the level of priority of the request. MGNT# is driven by core logic to VUMA device to indicate that it can access the physical system memory bus. Both MREQ# and MGNT# are driven synchronous to CPUCLK.

As shown in Figure 4-1, low level priority is conveyed by driving MREQ# low. A high level priority can only be generated by first generating a low priority request. As shown in Figure 4-2 and Figure 4-3, a low level priority is converted to a high level priority by driving MREQ# high for one CPUCLK clock and then driving it low.

Figure 4-1 Low Level Priority

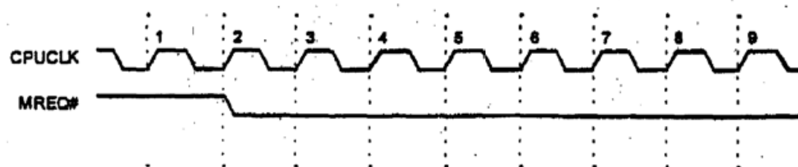


Figure 4-2 High Level Priority

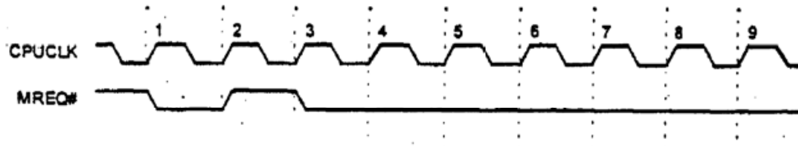
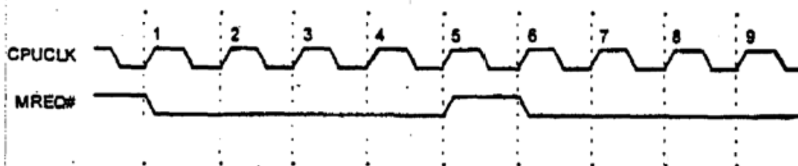


Figure 4-3 A Pending Low Level Priority converted to a High Level Priority



4.2 Arbiter

The arbiter, housed in core logic, needs to understand the arbitration protocol. State

Machine for the arbiter is depicted in Figure 4.4. As shown in Figure 4.4, the arbiter State Machine is resetted with PCI_Reset. Explanation of the arbiter is as follows:

1. HOST State - The physical system memory bus is with core logic and no bus request from VUMA device is pending.

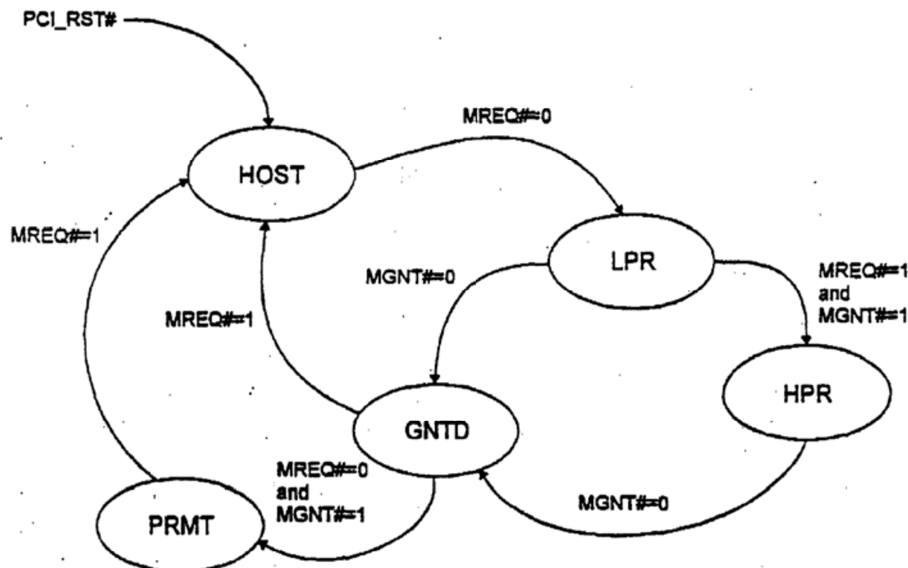
2. Low Priority Request (LPR) State - The physical system memory bus is with core logic and a low priority bus request from the VUMA device is pending.

3. High Priority Request (HPR) State - The physical system memory bus is with core logic and a pending low priority bus request has turned into a pending high priority bus request.

4. Granted (GNTD) State - Core logic has relinquished the physical system memory bus to VUMA device.

5. Preempt (PRMT) State - The physical system memory bus is owned by VUMA device, however, core logic has requested VUMA device to relinquish the bus and that request is pending.

Figure 4.4 Arbiter State Machine



Note:

1. Only the conditions which will cause a transition from one state to another have been

noted. Any other condition will keep the state machine in the current state.

4.2.1 Arbitration Rules

1. VUMA device asserts MREQ# to generate a low priority request and keeps it asserted until the VUMA device obtains ownership of the physical system memory bus through the assertion of MGNT#, unless the VUMA device wants to either raise a high priority request or raise the priority of an already pending low priority request. In the later case,

   a. If MGNT# is sampled asserted the VUMA device will not deassert MREQ#. Instead, the VUMA device will gain physical system memory bus ownership and maintain MREQ# asserted until it wants to relinquish the physical system memory bus.

   b. If MGNT# is sampled deasserted, the VUMA device will deassert MREQ# for one clock and assert it again irrespective of status of MGNT#. After reassertion, the VUMA device will keep MREQ# asserted until physical system memory bus ownership is transferred to the VUMA device through assertion of MGNT# signal.

2. VUMA device may assert MREQ# only for the purpose of accessing the unified memory area. Once asserted, MREQ# should not be deasserted before MGNT# assertion for any reason other than raising the priority of the request (i.e., low to high). No speculative request and request abortion is permitted. If MREQ# is deasserted to raise the priority, it should be reasserted in the next clock and kept asserted until MGNT# is sampled asserted.

3. Once MGNT# is sampled asserted by VUMA device, it gains and retains physical system memory bus ownership until MREQ# is deasserted.

4. The condition, VUMA device completing its required transactions before core logic needing the physical system memory bus back, can be handled in two ways:

   a. VUMA device can deassert MREQ#. In response, MGNT# will be deasserted in the next clock edge to change physical system memory bus ownership back to core logic.
   b. VUMA device can park on the physical system memory bus. If core logic needs the physical system memory bus, it should preempt VUMA device.

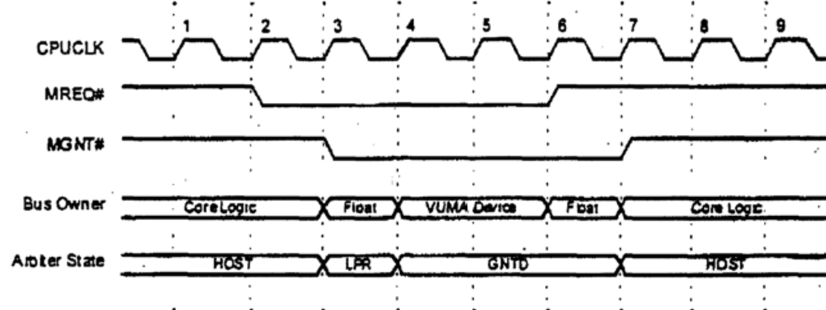
5. In case core logic needs the physical system memory bus before VUMA device releases it on its own, arbiter can preempt VUMA device from the bus. Preemption is signaled to VUMA device by deasserting MGNT#. VUMA device can retain ownership of the bus for a maximum of 60 CPUCLK clocks after it has been signaled

to preempt. VUMA device signals release of the physical system memory bus by deasserting MREQ#.

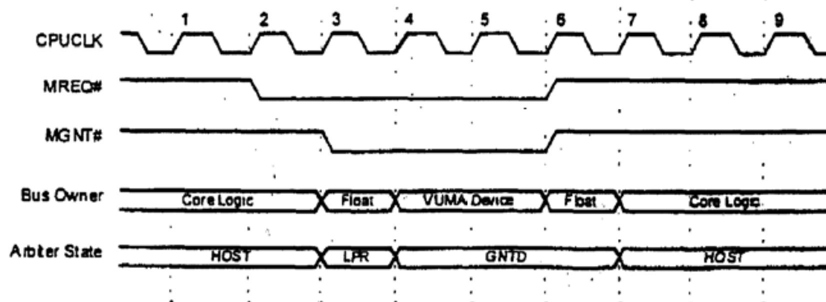
6. When VUMA device deasserts MREQ# to transfer bus ownership back to core logic, either on its own or because of a preemption request, it should keep MREQ# deasserted for at least two clocks of recovery time before asserting it again to raise a request.

4.3 Arbitration Examples

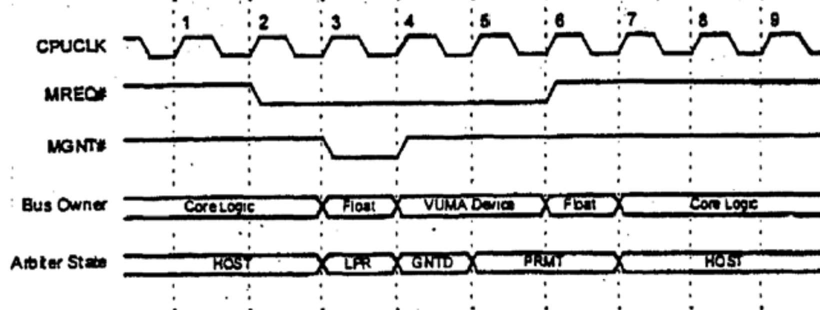
1. Low priority request and immediate bus release to VUMA device



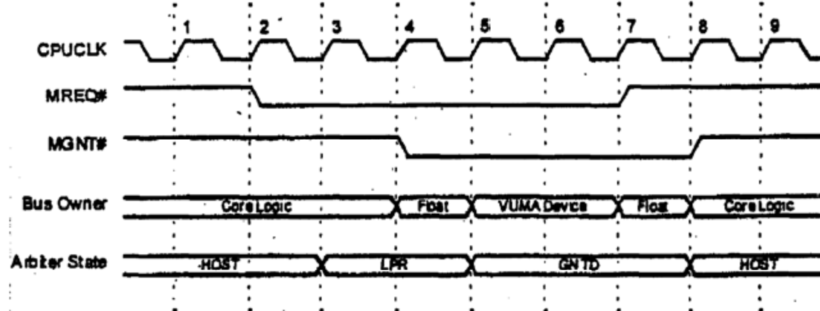
2. Low priority request and immediate bus release to VUMA device with preemption where removal of MGNT# and removal of MREQ# coincide



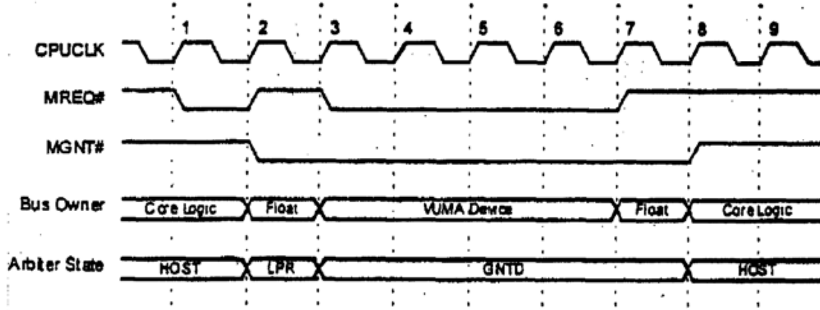
3. Low priority request and immediate bus release to VUMA device with preemption where MREQ# is removed after the current transaction because of preemption



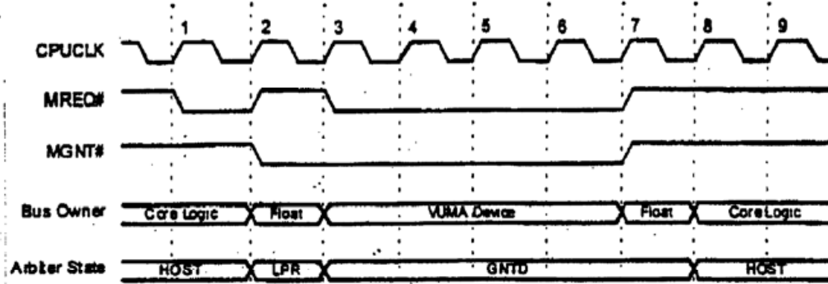
4. Low priority request and delayed bus release to VUMA device



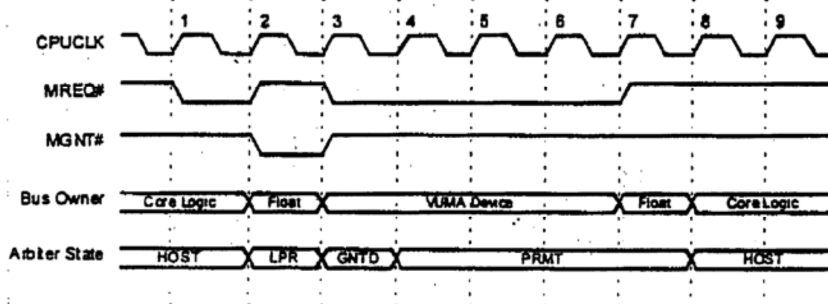
5. High priority request and immediate bus release to VUMA device



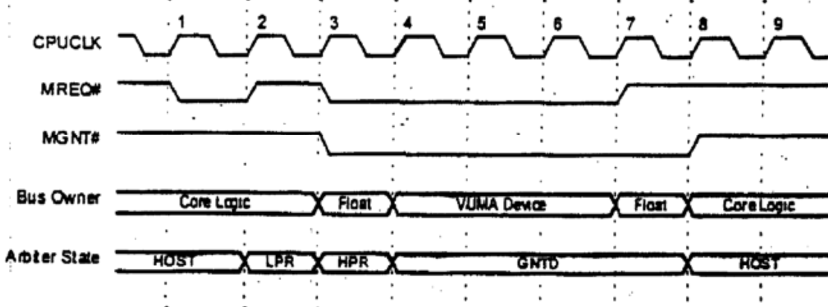
6. High priority request and immediate bus release to VUMA device with preemption where MGNT# and MREQ# removal coincides.



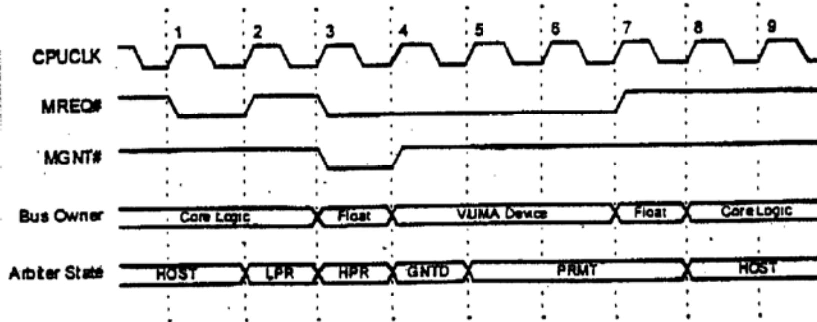
7. **High priority request and immediate bus release to VUMA device with preemption where MREQ# is removed after the current transaction because of preemption.**



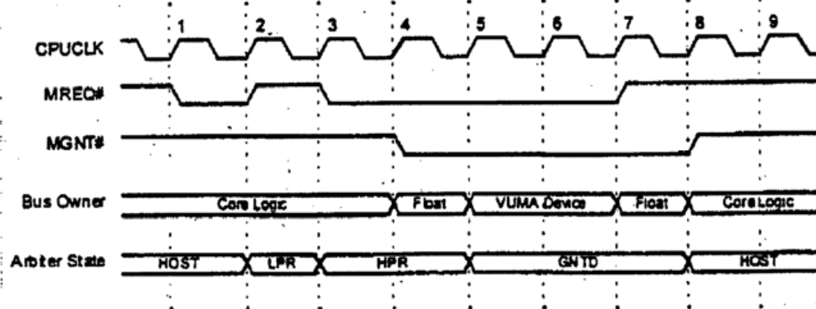
8. **High priority request and one clock delayed bus release to VUMA device**



9. **High priority request and one clock delayed bus release to VUMA device with preemption where MREQ# and MGNT# removal do not coincide**



10. High priority request and delayed bus release to VUMA device



4.4 Latencies

1. High Priority Request - Worst case latency for VUMA device to receive a grant after generating a high priority request is 35 CPUCLK clocks, i.e. after arbiter receives a high priority request from VUMA device, core logic does not need to relinquish the physical system memory bus right away and can keep the bus for up to 35 CPUCLK clocks.

2. Low Priority Request - No worst case latency number has been defined by this specification for low priority request. VUMA devices should incorporate some mechanism to avoid a low priority request being starved for an unreasonable time. The mechanism is implementation specific and not covered by the standard. One simple reference solution is as follows:

   VUMA device incorporates a programmable timer. The timer value is set at the boot time. The timer gets loaded when a low priority request is generated. When the timer times out, the low priority request is converted to a high priority request.

3. Preemption Request to VUMA device - Worst case latency for VUMA device to relinquish the physical system memory bus after receiving a preemption request is 60

CPUCLK clocks, i.e. after core logic requests VUMA device to relinquish the physical system memory bus, VUMA device does not need to relinquish the bus right away and can keep the bus for up to 60 CPUCLK clocks.

Design engineers should take in to consideration the above latencies for deciding FIFO sizes.

5.0 Memory Interface

The standard supports Fast Page Mode, EDO, BEDO and Synchronous DRAM technologies.

DRAM refresh for the physical system memory including Main VUMA Memory and Auxiliary VUMA Memory is provided by core logic during normal as well as suspend state of operation.

If VUMA device uses only a portion of its address space as Main VUMA Memory or Auxiliary VUMA Memory, it should drive unused upper MA address lines high.

5.1 Memory Decode

The way CPU address is translated in to DRAM Row and Column address decides the physical location in DRAM where a particular data will be stored. In the conventional architecture this could be implementation specific as there is a single DRAM controller. In unified memory architecture, multiple DRAM controllers (core logic resident and VUMA device resident DRAM controller) need to access the same data. Hence, all DRAM controllers should follow the same translation of CPU address into DRAM Row and Column address. The translation is as shown in Table 5-1.

Table 5-1 Translation of CPU address to DRAM Row and Column addresses

Symmetrical x9, x10, x11, x12

	MA11	MA10	MA9	MA8	MA7	MA6	MA5	MA4	MA3	MA2	MA1	MA0
clmn	A26	A24	A22	A11	A10	A9	A8	A7	A6	A5	A4	A3
row	A25	A23	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12

Asymmetrical x8

	MA11	MA10	MA9	MA8	MA7	MA6	MA5	MA4	MA3	MA2	MA1	MA0
clmn					A10	A9	A8	A7	A6	A5	A4	A3
row	A22	A21	A11	A20	A19	A18	A17	A16	A15	A14	A13	A12

Asymmetrical x9

	MA11	MA10	MA9	MA8	MA7	MA6	MA5	MA4	MA3	MA2	MA1	MA0
clmn				A11	A10	A9	A8	A7	A6	A5	A4	A3
row	A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12

Asymmetrical x10

	MA11	MA10	MA9	MA8	MA7	MA6	MA5	MA4	MA3	MA2	MA1	MA0
clmn			A22	A11	A10	A9	A8	A7	A6	A5	A4	A3
row	A24	A23	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12

Asymmetrical x11

	MA11	MA10	MA9	MA8	MA7	MA6	MA5	MA4	MA3	MA2	MA1	MA0
clmn		A24	A22	A11	A10	A9	A8	A7	A6	A5	A4	A3
row	A25	A23	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12

Synchronous 16Mb

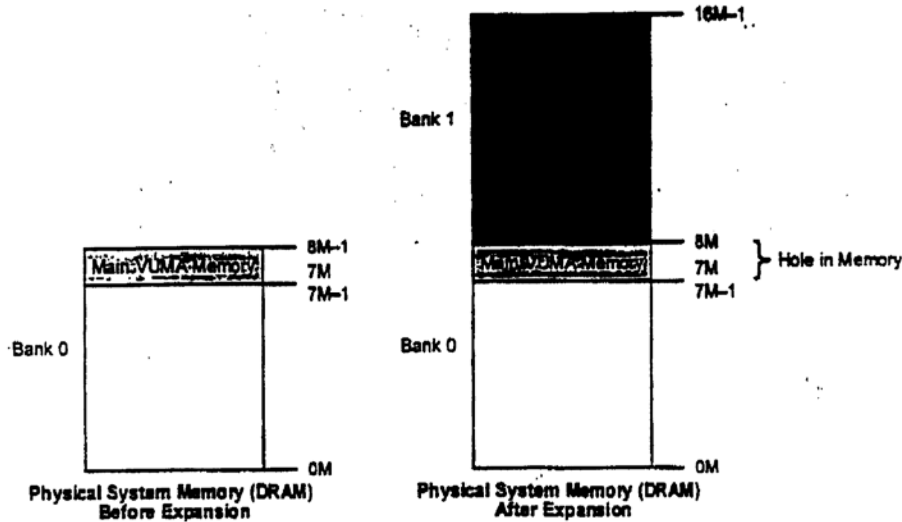
	MA11	MA10	MA9	MA8	MA7	MA6	MA5	MA4	MA3	MA2	MA1	MA0
clmn	A11		A24	A23	A10	A9	A8	A7	A6	A5	A4	A3
row	A11	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12

5.2 Main VUMA Memory Mapping

When physical system memory (DRAM) is expanded, unified memory architecture poses a unique problem not existing on the conventional architecture. The problem and three different solutions are described below:

Problem: Main VUMA Memory needs to be mapped at the top of existing memory for any given machine. When physical system memory (DRAM) is expanded, this would cause a hole in the physical system memory as shown in Figure 5-1. The example assumes an initial system with single bank 8MB memory (1MB allocated to Main VUMA Memory) expanded to 16MB memory (1MB allocated to Main VUMA Memory) by adding a bank of 8MB memory. All the numbers mentioned in this discussion are just examples and do not imply to be a part of the standard.

Figure 5-1 Memory Expansion Problem

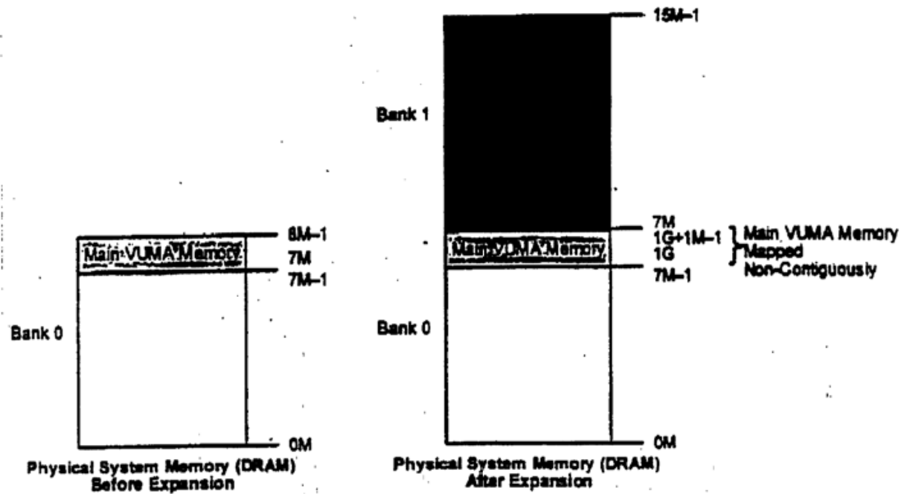


Three solutions are suggested for this problem. BIOS calls defined in VUMA VESA BIOS Extensions support all the three solutions. The BIOS calls are designed in such a way that a VUMA device can find out which of the three solutions is implemented by core logic and can configure the VUMA device appropriately.

Solution 1:

As depicted in Figure 5-2, core logic maps Main VUMA Memory to an address beyond core logic's possible physical system memory range. Main VUMA Memory is mapped non-contiguous to the O.S. memory. As shown in Figure 5-2, Main VUMA Memory is mapped from 1G to (1G+1M-1) and hence even if physical system memory is expanded to the maximum possible size, there will be no hole in the memory. As shown in Figure 5-2. Bank 0 is split with two separate blocks of memory with different starting addresses. If the VUMA device is a graphics controller, and if it wants to look at Main VUMA Memory also as a PCI address space, it can allocate a different address than what has been assigned by core logic (1G in this example).

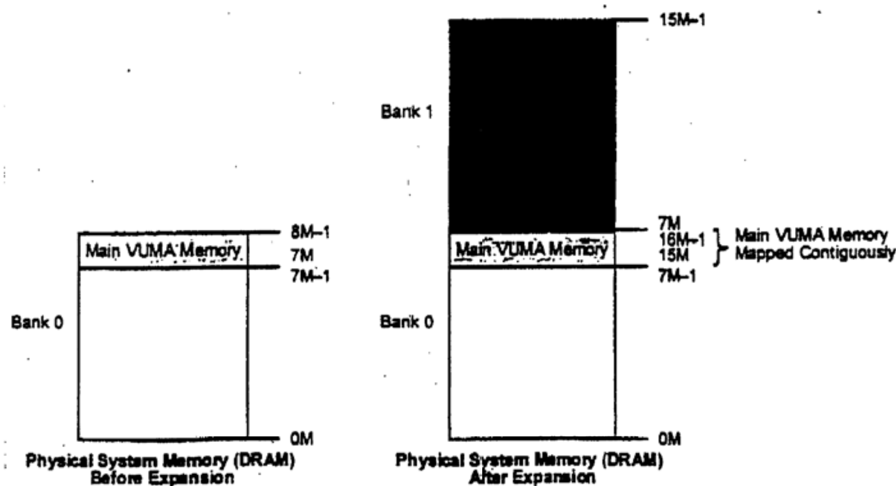
Figure 5-2 Main VUMA Memory mapped non-contiguously



Solution 2:

As depicted in Figure 5-3, core logic maps Main VUMA Memory to the top of memory. Main VUMA Memory is mapped contiguous to the O.S. memory. As shown in Figure 5-3, Main VUMA Memory is mapped from 15 M to (16M-1). As shown in Figure 5-3, Bank 0 is split with two separate blocks of memory with different starting addresses.

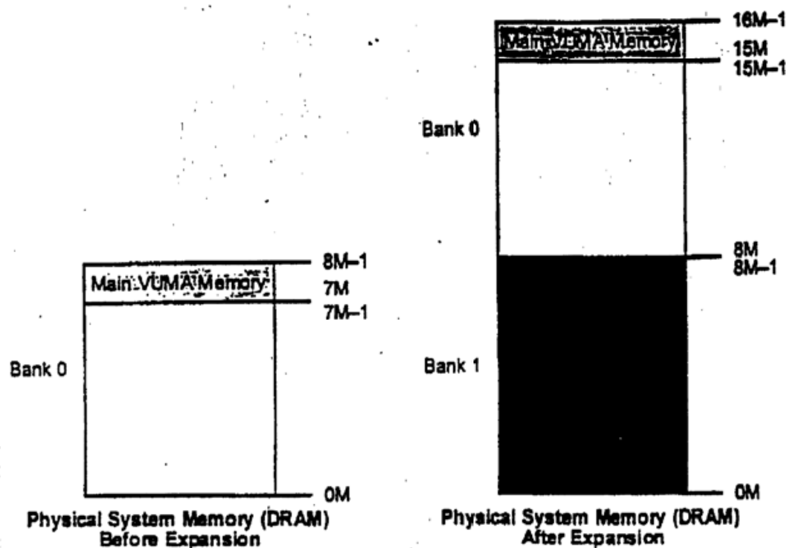
Figure 5-3 Main VUMA Memory mapped contiguously



Solution 3:

As depicted in Figure 5-4, core logic swaps the bank containing main VUMA Memory to the top of memory. As shown in Figure 5-4, Bank 0 is not split with two separate blocks of memory with different starting addresses like in solution 1 and solution 2.

Figure 5-4 Main VUMA Memory bank swapped



5.3 Fast Page EDO and BEDO

The logical interfaces for Fast Page, EDO and BEDO DRAMs are very similar and hence are grouped together. If no specific exception to a particular technology is mentioned, the description in this section applies to all the three types of DRAMs.

BEDO support is optional for both core logic and VUMA device. Various BEDO support scenarios are as follows:

1. Core logic does not support BEDO - Since core logic does not support BEDO, there will not be any BEDO as the physical system memory and hence whether VUMA device supports BEDO or not is irrelevant.

2. Core logic supports BEDO - When core logic supports BEDO, VUMA device may or may not support it. Whether core logic and VUMA device support BEDO or not should be transparent to the operating system and application programs. To achieve the transparency, system BIOS needs to find out if both core logic and VUMA device support this feature and set the system appropriately at boot. The following algorithm

explains how it can be achieved. The algorithm is only included to explain the feature. Refer to the latest VUMA VESA BIOS Extensions for the most updated BIOS calls:

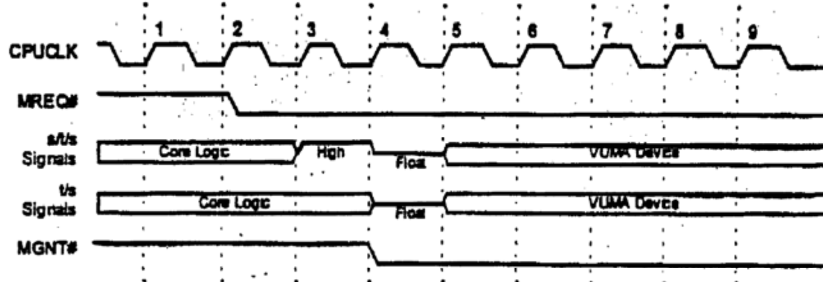
a. Read <VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)>. Check if VUMA device supports BEDO.
b. If VUMA device does not support BEDO, do not assign BEDO banks for Main VUMA Memory. Assign Main VUMA Memory to Fast Page Mode or EDO bank. Also, if Auxiliary VUMA Memory is assigned by operating system to BEDO banks, do not use it. Either repeat the request for Auxiliary VUMA Memory till it is assigned to Fast Page Mode or EDO bank or use some alternate method.
c. If VUMA device supports BEDO, read <VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)> to find out if VUMA device supports multiple banks access.
d. If only single bank access supported on VUMA device, exit, as the Main VUMA Memory and Auxiliary VUMA Memory bank is fixed.
e. If multiple banks access is supported and if the RAS for BEDO bank is supported on VUMA device, assign the Main VUMA Memory to obtain the best possible system performance and exit.

5.3.1 Protocol Description and Timing

All the DRAM signals are shared by core logic and VUMA device. They are driven by current bus master. When core logic and VUMA device hand over the bus to each other, they must drive all the shared s/t/s signals high for one CPUCLK clock and then tri-state them. Also, they should tri-state all the shared t/s signals.

The shared DRAM signals are driven by core logic when it is the owner of the physical system memory bus. VUMA device requests the physical system memory bus by asserting MREQ#. Bus Arbiter grants the bus by asserting MGNT#. Also, as mentioned above, before VUMA device starts driving the bus, core logic should drive the s/t/s signals high for one CPUCLK clock and tri-state them. Core logic should also tri-state all the shared t/s signals. The float condition on the bus should be for one CPUCLK clock, before VUMA device starts driving the bus. These activities are overlapped to improve performance as shown in Figure 5-5.

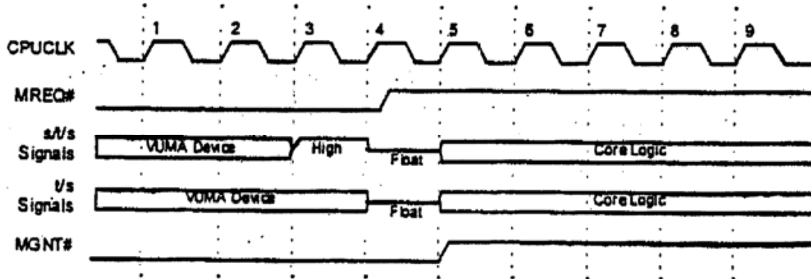
Figure 5-5 Bus hand off from core logic to VUMA device



MREQ# is driven low from clock edge 2. Core logic samples it active on clock edge 3. Arbiter can give the bus right away, so core logic drives all s/t/s signals high from the same clock edge. Core logic tri-states all the shared signals (s/t/s and t/s) and drives MGNT# active from clock edge 4 . VUMA device samples MGNT# active at clock edge 5 and starts driving the bus from the same edge.

The shared DRAM signals are driven by VUMA device when it is the owner of the physical system memory bus. VUMA device relinquishes the physical system memory bus by de-asserting MREQ#. Bus Arbiter gives the bus back to core logic by de-asserting MGNT#. Also, as mentioned above, before core logic starts driving the bus, VUMA device should drive the s/t/s signals high for one CPUCLK clock and tri-state them. VUMA device should also tri-state all the shared t/s signals. The float condition on the bus should be for one CPUCLK clock, before core logic starts driving the bus. These activities are overlapped to improve performance as shown in Figure 5-6.

Figure 5-6 Bus hand off from VUMA device to core logic



VUMA device drives all s/t/s signals high from clock edge 3. VUMA device tri-states all shared signals (s/t/s and t/s) and de-asserts MREQ# from clock edge 4. Core logic samples MREQ# inactive on clock edge 5. Core logic drives all shared signals and deasserts MGNT# from clock edge 5.

5.3.2 DRAM Precharge

When the physical system memory bus is handed off from core logic to VUMA device or vice a versa, the DRAM needs to be precharged before the new master starts driving it. Part of this precharge can be hidden by overlapping with the arbitration protocol. As shown in Figure 5-5 and 5-6, all the DRAM control signals (including RAS# lines) are driven high for and tri-stated for one CPUCLK clock each. When RAS# lines are tri-stated, pull ups on those lines pull them to a logical high. Thus when a new master gets the control, the RAS# lines are already been precharged for two CPUCLK clocks. The rest of the precharge needs to be taken care by the new master.

1. VUMA device gets the bus from core logic - As shown in Figure 5-5, when VUMA device gets the physical system memory bus at clock edge 5, the DRAM has been precharged for two CPUCLK clocks. VUMA device needs to take care of the rest of the DRAM precharge. This precharge can be overlapped by VUMA device over some of its activity e.g. VUMA device may be running at a different clock than the CPUCLK clock and the precharge can be overlapped with the synchronization of MGNT# signal. VUMA device can calculate the number of clocks it needs to precharge the DRAM with the following formula:

No. of VUMA device clocks for DRAM precharge = {RAS# Precharge Time (tRP) - (2 * CPUCLK Clock Time Period)}/ VUMA device Clock Time Period

Example: CPU running at 66.66 MHz, VUMA device running at 50 MHz. 70ns Fast Page DRAM used.

No. of VUMA device clocks for DRAM precharge = {50ns - (2* 15ns)}/ 20ns
= {20ns}/ 20ns
= 1 clock

2. Core logic gets the bus from VUMA device - As shown in Figure 5-6, when core logic gets the physical system memory bus at clock edge 5, the DRAM has been precharged for two CPUCLK clocks. core logic needs to take care of the rest of the DRAM precharge. This precharge can be overlapped by core logic over some of its activity e.g. driving of new row address. Core logic can calculate the number of clocks it needs to precharge the DRAM with the following formula:

No. of CPU clocks for DRAM precharge = {RAS# Precharge Time (tRP) - (2 * CPUCLK Clock Time Period)}/ CPUCLK Clock Time Period

Example: CPU running at 66.66 MHz, VUMA device running at 50 MHz. 70ns Fast Page DRAM used.

No. of CPU clocks for DRAM precharge = {50ns - (2* 15ns)}/ 15ns
= {20ns}/ 15ns
= 2 clock

5.4 Synchronous DRAM

Synchronous DRAM support is optional for both core logic and VUMA device. Various Synchronous DRAM support scenarios are as follows:

1. Core logic does not support Synchronous DRAM - Since core logic does not support Synchronous DRAM, there would not be any Synchronous DRAM as the physical system memory and hence whether VUMA device supports Synchronous DRAM or not is irrelevant.

2. Core logic supports Synchronous DRAM - When core logic supports Synchronous DRAM, VUMA device may or may not be supporting it. Whether core logic and VUMA device support Synchronous DRAM or not should be transparent to the operating system and application programs. To achieve the transparency, system BIOS needs to find out if both core logic and VUMA device support this feature and set the system appropriately at boot. The following algorithm explains how it can be achieved. The algorithm is only included to explain the feature. Refer to the latest VUMA VESA BIOS Extensions for the most updated BIOS calls:

   a. Read <VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)>. Check if VUMA device supports Synchronous DRAM.
   b. If VUMA device does not support Synchronous DRAM, do not assign Synchronous DRAM banks for Main VUMA Memory. Assign Main VUMA Memory to Fast Page Mode or EDO bank. Also, if Auxiliary VUMA Memory is assigned by operating system to Synchronous DRAM banks, do not use it. Either repeat the request for Auxiliary VUMA Memory till it is assigned to Fast Page Mode or EDO bank or use some alternate method.
   c. If VUMA device supports Synchronous DRAM, read < VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)> to find out if VUMA device supports multiple banks access.
   d. If only single bank access supported on VUMA device, exit, as the Main VUMA Memory and Auxiliary VUMA Memory bank is fixed.
   e. If multiple banks access is supported and if the CS# for Synchronous DRAM bank is supported on VUMA device, assign the Main VUMA Memory to obtain the best possible system performance and exit.

5.4.1 Programmable Parameters

Synchronous DRAMs have various programmable parameters. Core logic programs Synchronous DRAM parameters to obtain the best possible results. The most efficient way for VUMA device to program its DRAM controller is to make a BIOS call to find

out the parameters core logic has decided and program its DRAM controller with the same parameters. Alternately, VUMA device could program its DRAM controller with one or all different parameters. If VUMA device programs its DRAM controller with any different parameters, it is VUMA device's responsibility to reprogram Synchronous DRAM back with the original parameters, before the physical system memory bus is handed off to core logic. In other words, VUMA device is free to change any or all of the parameters, but the change should be transparent to core logic.

How core logic programs various parameters and how VUMA device could inquire them is as follows:

1. Burst Length - Burst Length can be programmed as 1, 2 or 4. VUMA device needs to make a BIOS call <Return Memory Speed Type (refer to VUMA VESA BIOS Extensions)> to find out the Burst Length.

2. CAS Latency - As CAS latency depends on the speed of Synchronous DRAM used and the clock speed, this standard does not want to fix this parameter. Core logic programs this parameter to an appropriate value. VUMA device needs to make a BIOS call <Return Memory Speed Type (refer to VUMA VESA BIOS Extensions)> to find out the CAS latency.

3. Burst Ordering - Most efficient Burst Ordering depends upon the type of CPU used. VUMA device needs to make a BIOS call <Return Memory Speed Type (refer to VUMA VESA BIOS Extensions)> to find out the Burst Order.

5.4.2 Protocol Description and Timing

All the DRAM signals are shared by core logic and VUMA device. They are driven by current bus master. When core logic and VUMA device hand over the bus to each other, they must drive all the shared s/t/s signals high for one CPUCLK clock and then tri-state them. Also, they should tri-state all the shared t/s signals.

Synchronous DRAMs are precharged by precharge command. When the physical system memory bus is handed off from core logic to VUMA device or vice a versa, the DRAM precharge has two options:

1. Precharge both the internal banks before hand-off - This is a simple case where both the internal banks of the active synchronous DRAM bank are precharged and then the bus is handed off.
2. Requesting Master snoops the physical system memory bus and synchronous DRAM internal banks need not be precharged - In this case the requesting master snoops the DRAM address and control signals to track the open pages in the internal banks of the active synchronous DRAM bank. The internal banks of the active synchronous DRAM

are not precharged when the physical system memory bus is handed-off to the requesting master. If needed, the requesting master takes care of precharge after getting the physical system memory bus.

Both core logic and VUMA device have an option of either implementing or not implementing DRAM snoop feature. Whether core logic and VUMA device support DRAM snoop or not should be transparent to the operating system and application programs. To achieve the transparency, system BIOS and VUMA BIOS need to find out if both core logic and VUMA device support this feature and set the system appropriately at boot. The following algorithm explains how it can be achieved. The algorithm is only included to explain the feature. Refer to the latest VUMA VESA BIOS Extensions for the most updated BIOS calls:

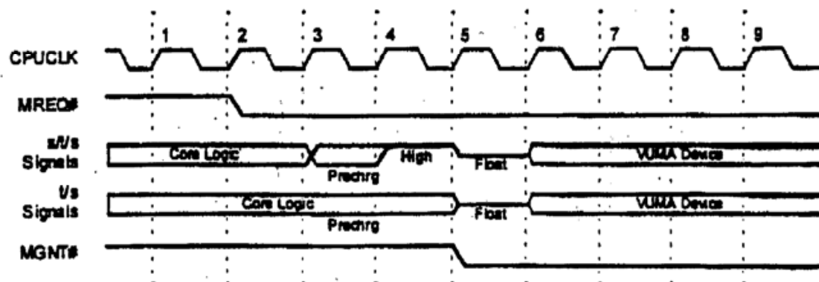
1. System BIOS reads <VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)>, to find out if VUMA device can snoop the physical system memory bus.
2. If no, System BIOS programs core logic to precharge synchronous DRAM before bus hand-off.
3. If yes, System BIOS programs core logic not to precharge synchronous DRAM before bus hand-off.
4. VUMA BIOS makes a call, <Report VUMA - core logic capabilities (refer to VUMA VESA BIOS Extensions)>, to find out if core logic can snoop the physical system memory bus.
5. If no, VUMA BIOS programs VUMA device to precharge synchronous DRAM before bus hand-off.
6. If yes, VUMA BIOS programs VUMA device not to precharge synchronous DRAM before bus hand-off.

None, only one, or both of core logic and VUMA device can support this feature. When only one of them supports this feature memory precharge will be asymmetrical i.e. there will be precharge before hand-off one way and no precharge the other way.

5.4.2.1 Non-Snoop Cases

The shared DRAM signals are driven by core logic when it is the owner of the physical system memory bus. VUMA device requests the physical system memory bus by asserting MREQ#. Bus Arbiter grants the bus by asserting MGNT#. Also, before VUMA device starts driving the bus, core logic should drive all the shared s/t/s signals high for one CPUCLK clock and tri-state them. Core logic should also tri-state all the shared t/s signals. The tri-state condition on the bus should be for one CPUCLK clock, before VUMA device starts driving the bus. These activities are overlapped to improve performance as shown in Figure 5-7. Since VUMA device does not support DRAM snoop feature, DRAM is precharged before handing off the physical system memory bus as shown in Figure 5-7.

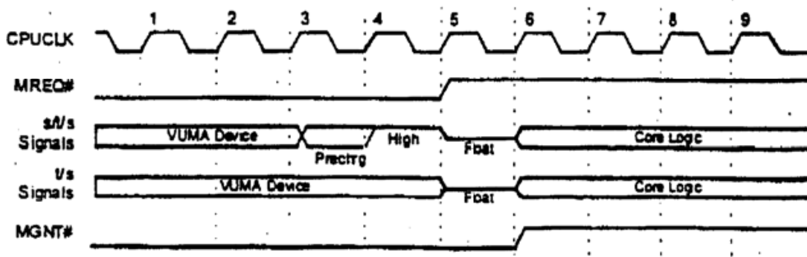
Figure 5-7 Bus hand off from Core Logic to VUMA device



MREQ# is driven low from clock edge 2. Core logic samples it active on clock edge 3. Arbiter can give the bus right away, so core logic gives precharge command to DRAM from the same clock edge. Core logic drives all the shared s/t/s signals high from clock edge 4. Core logic tri-states all the shared signals (s/t/s and t/s) and drives MGNT# active from clock edge 5. VUMA device samples MGNT# active at clock edge 6 and starts driving the bus from the same edge.

The shared DRAM signals are driven by VUMA device when it is the owner of the physical system memory bus. VUMA device relinquishes the physical system memory bus by de-asserting MREQ#. Bus Arbiter gives the bus back to core logic by de-asserting MGNT#. Also, as mentioned above, before core logic starts driving the bus, VUMA device should drive all the shared s/t/s signals high for one CPUCLK clock and tri-state them. VUMA device should also tri-state all the shared t/s signals. The float condition on the bus should be for one CPUCLK clock, before core logic starts driving the bus. These activities are overlapped to improve performance as shown in Figure 5-8. Since core logic does not support DRAM snoop feature, DRAM is precharged before handing off the physical system memory bus as shown in Figure 5-8.

Figure 5-8 Bus hand off from VUMA device to Core Logic



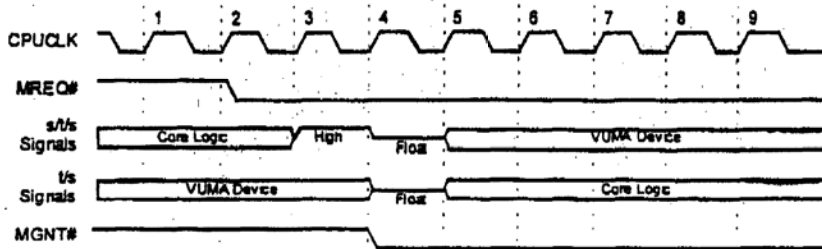
VUMA device gives precharge command from clock edge 3. It drives all shared s/t/s signals high from clock edge 4. It tri-states all shared signals (s/t/s and t/s) and de-asserts

MREQ# from clock edge 5. Core logic samples MREQ# inactive on clock edge 6. Core logic drives all shared signals and deasserts MGNT# from clock edge 6.

5.4.2.2 Snoop Cases

The shared DRAM signals are driven by core logic when it is the owner of the physical system memory bus. VUMA device requests the physical system memory bus by asserting MREQ#. Bus Arbiter grants the bus by asserting MGNT#. Also, before VUMA device starts driving the bus, core logic should drive all the shared s/t/s signals high for one CPUCLK clock and tri-state them. Core logic should also tri-state all the shared t/s signals. The tri-state condition on the bus should be for one CPUCLK clock, before VUMA device starts driving the bus. These activities are overlapped to improve performance as shown in Figure 5-9. Since VUMA device supports DRAM snoop feature, core logic does not precharge DRAM before handing off the physical system memory bus as shown in Figure 5-9.

Figure 5-9 Bus hand off from core logic to VUMA device

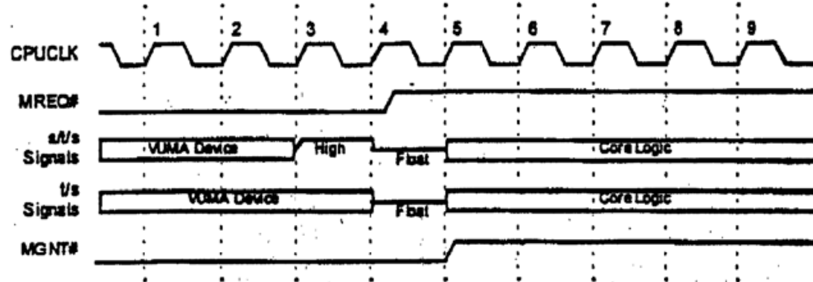


MREQ# is driven low from clock edge 2. Core logic samples it active on clock edge 3. Arbiter can give the bus right away and since VUMA device supports DRAM snoop feature, core logic drives all the shared s/t/s signals high from the same clock edge. Core logic tri-states all the shared signals (s/t/s and t/s) and drives MGNT# active from clock edge 4. VUMA device samples MGNT# active at clock edge 5 and starts driving the bus from the same edge.

The shared DRAM signals are driven by VUMA device when it is the owner of the physical system memory bus. VUMA device relinquishes the physical system memory bus by de-asserting MREQ#. Bus Arbiter gives the bus back to core logic by de-asserting MGNT#. Also, as mentioned above, before core logic starts driving the bus, VUMA device should drive all the shared s/t/s signals high for one CPUCLK clock and tri-state them. VUMA device should also tri-state all the shared t/s signals. The float condition on the bus should be for one CPUCLK clock, before core logic starts driving the bus. These activities are overlapped to improve performance as shown in Figure 5-10. Since core

logic supports DRAM snoop feature, VUMA device does not precharge DRAM before handing off the physical system memory bus as shown in Figure 5-10.

Figure 5-10 Bus hand off from VUMA device to core logic



VUMA device drives all shared s/t/s signals high from clock edge 3. It tri-states all shared signals (s/t/s and t/s) and de-asserts MREQ# from clock edge 4. Core logic samples MREQ# inactive on clock edge 5. Core logic drives all shared signals and deasserts MGNT# from clock edge 5.

5.5 Memory Parity support

Memory Parity support is optional on both core logic and VUMA device. If core logic supports parity it should be able to disable parity check for Main VUMA Memory and Auxiliary VUMA Memory areas while parity check on the rest of the physical system memory is enabled.

5.6 Memory Controller Pin Multiplexing

The logical interfaces for Fast Page, EDO and BEDO DRAMs are very similar but are significantly different than that of Synchronous DRAM. If mother board designers want to mix different DRAM technologies on the same mother board, core logic will have to multiplex DRAM control signals. The meaning of a multiplexed signal will depend on the type of DRAM core logic is accessing at a given time. If a VUMA device supports multiple banks access and mix of different DRAM technologies, it will also have to multiplex DRAM control signals. Both core logic and VUMA devices will have to have same multiplexing scheme. The appropriate JEDEC standard should be followed for multiplexing scheme.

6.0 Boot Protocol

6.1 Main VUMA Memory Access at Boot

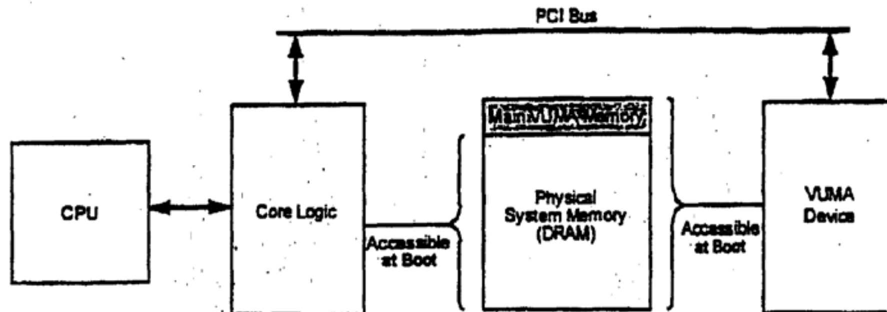
In unified memory architecture, part of the physical system memory is assigned to Main VUMA Memory. The existing operating systems are not aware of unified memory architecture. Also, some of the existing operating systems size memory themselves. This poses a problem as the operating systems after sizing the total physical system memory, will assume that they could use all of the memory and might overwrite Main VUMA Memory. The solution to this problem is explained below:

As shown in Figure 6-1, the solution to this problem is to disable core logic access to Main VUMA Memory area at boot time. In that case even if operating system, sizes the memory, it will find only (total physical system memory - Main VUMA Memory) and will not be aware of the Main VUMA Memory existence. This will avoid operating system ever writing to the Main VUMA Memory area. If VUMA device supports multiple banks access, it can access total physical system memory all the time. If VUMA device supports single bank access, it can access the bank of Main VUMA Memory all the time.

If VUMA device is a graphics controller, it needs a special consideration. Video screen is required during boot and.since core logic can not access the Main VUMA Memory, it can not write to it. The problem is solved by programming the graphics controller into a pseudo legacy mode. In this mode graphics controller treats Main VUMA Memory exactly the same way as in non unified memory architecture situations i.e. as if it has its own separate frame buffer. So now, the total system looks just like a non unified memory architecture system and this mode is called as pseudo legacy mode. Core logic performs accesses to video through legacy video memory address space of A000:0 and B000:0. These accesses go on the PCI bus. Graphics controller claims these cycles. Graphics controller still needs to arbitrate for the physical system memory bus. After getting the bus, graphics controller performs reads/writes to Main VUMA Memory (frame buffer). After the system boots, it is still in the pseudo legacy mode. When operating system calls display driver, the driver programs core logic to allow access to Main VUMA Memory and switches the system from pseudo legacy mode to unified memory architecture.

In the case of other type of VUMA devices, device driver needs to program core logic to allow access to Main VUMA Memory.

Figure 6-1 Pseudo Legacy Mode



The following algorithm sums up the boot process in the case of VUMA device being a graphics controller:

1. System BIOS sizes the physical system memory.
2. System BIOS reads the size of Main VUMA Memory at previous boot (where this value is stored is System BIOS dependent, but needs to be in some sort of non volatile memory).
3. System BIOS programs its internal registers to reflect that total memory available is [total physical system memory(from step 1) - Main VUMA Memory at previous boot (from step 2)].
4. System boots and operating system calls display driver.
5. Display driver makes a System BIOS call, <Enable/Disable Main VUMA Memory (refer to VUMA VESA BIOS Extensions)>, to program core logic internal registers to reflect that it can access total physical system memory.
6. Display driver switches VUMA device to unified memory architecture mode.

Even though core logic can not access Main VUMA Memory till the time display driver enables it, core logic is responsible for Main VUMA Memory refresh.

VUMA device should claim PCI Master accesses to Main VUMA Memory till display driver enables core logic access to that area. Core logic should claim PCI Master accesses to Main VUMA Memory after display driver enables core logic access to that area.

6.2 Reset State

On power on reset, both core logic and VUMA device have their unified memory architecture capabilities disabled. MREQ# is de-asserted by VUMA device and MGNT# is de-asserted by core logic. System BIOS can detect if VUMA device supports unified memory architecture capabilities by reading <VUMA BIOS signature string (refer to VUMA VESA BIOS Extensions)>.

7.0 Electrical Specification

7.1 Signal Levels

This section describes the electrical signal levels for the arbitration signals only. DRAM signal levels depend on the type of DRAM used and hence can not be specified by the standard.

MREQ#	**output**	5v TTL or 3.3v LVTTL
	input	5v TTL for 5v buffer, 5v tolerant LVTTL for 3.3v buffer
MGNT#	**output**	5v TTL or 3.3v LVTTL
	input	5v TTL for 5v buffer, 5v tolerant LVTTL for 3.3v buffer
CPUCLK	**output**	5v TTL or 3.3v LVTTL
	input	5v TTL for 5v buffer, 5v tolerant LVTTL for 3.3v buffer

7.2 AC Timing

This section describes the AC timing parameters for the arbitration signals only. DRAM AC timing parameters depend on the type of DRAM used and hence can not be specified by the standard. Both MREQ# are MGNT# timing parameters are with respect to CPUCLK rising edge.

MREQ#	**output**	tClk to Out (max)	- 10 ns
		tClk to Out (min)	- 2 ns
	input	Set up time tSU (min)	- 3 ns
		Hold time tH (min)	- 0 ns
MGNT#	**output**	tClk to Out (max)	- 10 ns
		tClk to Out (min)	- 2 ns
	input	Set up time tSU (min)	- 3 ns
		Hold time tH (min)	- 0 ns
CPUCLK	**output**	clock frequency (max)	- 66.66 MHz

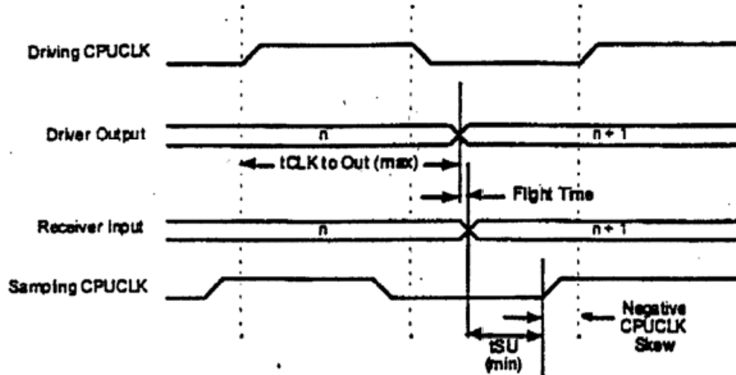
7.2.1 Timing Budget

A margin for signal flight time and clock skew is added to the timing parameters. ± 2ns is allowed for the total of CPUCLK skew and signal flight time. Worst case timing budget calculations for setup and hold time are as follows:

7.2.1.1 Worst case for Setup time

Figure 7-1 shows the worst case for setup time. tClk to Out, flight time and clock skew have converged to reduce available setup time.

Figure 7-1 Worst case for setup time

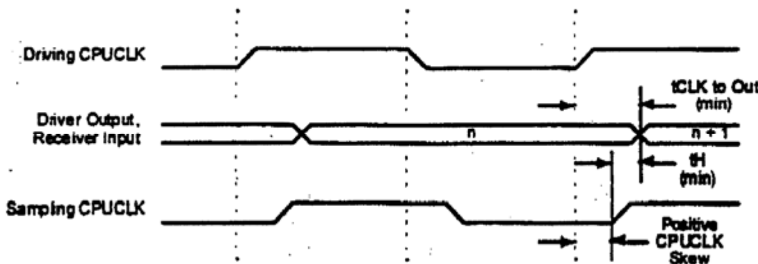


[tCLK to Out (max) + flight time + tSU (min) + negative CPUCLK skew] ≤ CPUCLK period i.e. 15ns @ 66.66 MHz.
[10ns + flight time + 3ns + negative CPUCLK skew] ≤ 15ns
[flight time + negative CPUCLK skew] ≤ 2ns

7.2.1.2 Worst case for Hold time

Figure 7-2 shows the worst case for hold time. tClk to Out and clock skew have converged to reduce available hold time. Positive flight time number helps in this case and hence it is assumed to be zero.

Figure 7-2 Worst case for hold time



[positive CPUCLK skew + tH (min)] ≤ tCLK to Out (min)
[positive CPUCLK skew + 0ns] ≤ 2ns
positive CPUCLK skew ≤ 2ns

7.3 Pullups

All s/t/s signals need pullups to sustain the inactive state until another agent drives them. Core logic has to provide pullups for all the s/t/s signals. VUMA device has as option of providing pullups on some of the s/t/s signals. All t/s signals need pulldowns. Core logic has to provide pulldowns for all the t/s signals. VUMA device has as option of providing pulldowns on the t/s signals. Pullups and pulldowns could either be internal to the chips or external on board.

DRAM Address - Core logic is responsible for pullups on DRAM Address lines.
DRAM control signals - Core logic is responsible for pullups on DRAM control signals. VUMA device has as option of providing pullups on them.
DRAM Data Bus - Core logic is responsible for pulldowns on DRAM data bus. VUMA device has as option of providing pulldowns on them.

Pullups and pulldowns are used to sustain the inactive state until another agent drives the signals and hence need to be weak. Recommended value for pullups and pulldowns is between 50 kohm and 80 kohm.

7.4 Straps

As some VUMA devices and core logic chips use DRAM data bus for straps, DRAM data bus needs to be assigned for straps for different controllers. The assignment of DRAM Data Bus for straps is as follows:

MD [0:19] VUMA device on Motherboard
MD [20:55] Reserved
MD [56:63] Core Logic

All the straps need to be pullups of 10 kohm.

7.5 DRAM Driver characteristics

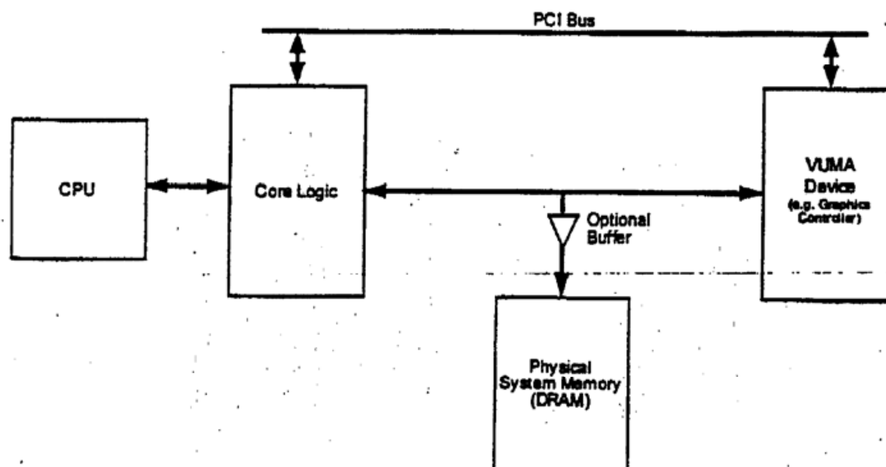
Loading plays a critical role in DRAM access timing. In case of PC motherboards end users can expand the existing memory of a system by adding extra SIMMs. Hence, typically the total DRAM signal loading is not constant and could vary significantly. Both Core Logic and VUMA device must be able to drive the maximum load that the

system motherboard is designed to accommodate. In typical motherboard designs DRAM signal loading can be excessive (on the order of 1000pF for some signals) and hence care must be taken for DRAM driver selection. Some general guide lines for DRAM driver design are as follows:

Slew-rate controlled drivers are recommended. Drivers with selectable current drive (such as 8/16 mA drivers) may be used. This can reduce overshoot and undershoot associated with over-driving lightly loaded signals and can prevent excessive rise and fall time delay due to not providing enough current drive on heavily loaded signals.

As shown in Figure 7-3, buffers may be placed on the system motherboard to reduce the per signal loading and/or provide larger drive strength capabilities. DRAM Write Enable and DRAM Address signals are typically the most heavily loaded signals. Column Address Strobe signals may also become overloaded when more than two DRAM banks are designed into a system. TTL or CMOS buffers (typically 244 type) may be used to isolate and duplicate heavily loaded signals on a per bank basis. 244 type buffers typically have very good drive characteristics as well and can be used to drive all of the heavily loaded DRAM control signals if the Core Logic and/or VUMA device has relatively weak drive characteristics. If external buffers are used, the buffer delays should be taken in to timing considerations.

Figure 7-3 Optional Buffers for DRAM Signals



Wider DRAM devices offer reduced system loading on some of the control signals. x4 DRAMs require four times the physical connections on RAS, MA (Address), and write enables as x16 DRAMs. The reduction in loading can be significant. If the designer has control over the DRAMs which will be used in the system, the DRAM width should be chosen to provide the least loading.