GIO BUS

SPECIFICATION

version 1.1 (draft)
7April 1992

Silicon Graphics
Computer Systems

Table of Contents

Chapter One: Overview

Chapter Two: GIO32 Specification

Chapter Three: GIO32-bis Specification

Chapter Four: GIO64 Specification

List of Figures

Chapter 1

GIO Bus Overview

1.1 Introduction

The GIO Bus is a family of synchronous, multiplexed address-data busses for connecting high-speed devices to main memory and CPU for entry-level SGI systems. The GIO Bus has three varieties: GIO32, GIO32-bis, and GIO64. Each variety is described in a separate chapter: GIO32 is in Chapter 2, GIO32-bis is in Chapter 3, and GIO64 is in Chapter 4.

- The GIO32 is a 32-bit, synchronous, multiplexed address-data bus that runs at speeds from 25 to 33 MHz.
- The GIO32-bis is a 32-bit version of the non-pipelined GIO64 bus.
- The GIO64 bus is a 64-bit, synchronous, multiplexed address-data bus that can run at speeds up to 40 MHz. It supports both 32- and 64-bit GIO64 devices. GIO64 has two slightly different varieties: non-pipelined for internal system memory and GIO32-bis slot devices, and pipelined for graphics and pipelined GIO64 slot devices.

The members of the GIO Bus Family are all very similar, however the GIO32 and GIO64 are not compatible. A GIO32 device does not work in a GIO64 slot. However, a GIO32-bis device can function in either a GIO32 or GIO64 option slot, as long as the appropriate connector/socket is available.

	GIO32 Slot	GIO64 Slot	
	32-pin connector	*32-pin connector*	*64-pin connector*
Bus Protocol supported:	GIO32 GIO32-bis	GIO64 GIO32-bis	GIO64

1.2 Conventions Used in this Document: Signal Names

Signal names that are overscored ($\overline{\text{EXAMPLE}}$) denote signals that are active low signals. All other signals are active high.

Signals that are one-per-device are denoted by the letter ‘n’ in parentheses, following the signal name: SIGNAL(n).

This page has been left blank intentionally.

Chapter 2

GIO32 Specification

2.1 Introduction

The GIO32 bus is a 32-bit, synchronous, multiplexed address-data bus that runs at speeds from 25 to 33 MHz. The bus connects high speed devices to memory and to the CPU.

The GIO32 bus supports two types of devices: 1.the CPU and other long-burst devices that do long-burst transfers between themselves and system memory, and 2. real-time I/O devices that require guaranteed maximum bus latency.

A bus arbiter, implemented by the Processor Interface Controller (PIC) in the Indigo system, arbitrates between competing bus masters in the system. The PIC also acts as master in transactions between the Indigo CPU and other GIO32 devices.

Maximum performance of the 33 MHz GIO32 bus is as follows:

long-burst read/write	132 MBytes/second
back-to-back 32-bit word writes	44 MBytes/second
back-to-back 32-bit word reads from memory	16 MBytes/second
back-to-back 32-bit word reads (theoretical)	26 MBytes/second[1]

2.2 Conventions Used in this Chapter

The GIO32 bus uses a 32-bit byte address. The byte numbering scheme is big endian; the bit numbering scheme is little endian. (See Figure 2-1.) Thus, byte 0 is bits <31:24>, byte 1 is bits <23:16>, byte 2 is bits <15:8>, and byte 3 is bits <7:0>.

The following figure shows the byte and bit numbering schemes.

31 24	23 16	15 8	7 0
0	1	2	3

Figure 2-1. Traditional SGI: big endian byte numbering / little endian bit numbering

[Due to the Indigo's ability to run either as a traditionally big-endian system or as a little-endian system, the GIO32 bus supports both little- and big-endian byte numbering schemes. The endian selection is made at power-up time based on the endianness of the software found on the hard disk. GIO32 option cards need to be capable of running in big- and little-endian mode. How they detect the endianness of the system at power-up time is TBD.]

1. Due to the specific implementation on the Indigo, back-to-back 32-bit word reads from the Indigo CPU are about 8.8 MBytes/second.

31 … 24	23 … 16	15 … 8	7 … 0
3	2	1	0

Figure 2-2. Little endian byte numbering / little endian bit numbering

2.3 GIO32 Transfer Size and Preemption

GIO32 bus transfers can be one byte to 4K bytes long. A byte count field specifies the length of a transaction. The transfer cannot cross a 4K processor virtual page boundary; i.e., the data must originate from or be destined for a single page of main memory. A bus request by a real-time I/O device preempts an ongoing long-burst transaction. The interrupted transaction can be resumed as a new transaction when the bus is free.

2.4 GIO32 Bus Signals

The GIO32 bus has 32 address/data signals; control lines that include address strobe, read/write select, master delay, and slave delay; arbitration signals that include bus request, bus grant, and bus preempt; and CPU interrupt and reset signals. The table below lists the GIO32 signals.

TABLE 1 GIO Bus Signals.

SIGNAL	Description
AD<31:0>	32 address/data signals
$\overline{\text{AS}}$	address strobe
READ	read/write select
MASDLY	master delay
SLVDLY	slave delay
$\overline{\text{BREQ}}$(n)	bus request, 1 per master
$\overline{\text{BGNT}}$(n)	bus grant, 1 per master
$\overline{\text{BPRE}}$	bus preempt
$\overline{\text{INT}}$<2:0>	interrupts
CLK	clock
$\overline{\text{RESET}}$	reset

AD<31:0>: The address/data signals are the multiplexed address and data lines. During a slave address cycle, they contain the slave address for the bus transaction. The next cycle in a transaction, the byte count cycle, uses AD<12:0> to indicate the number of bytes to transfer and AD<31:30> to indicate the master starting byte address. The master starting byte address defines the byte where the data starts (byte 0, 1, 2, or 3) when placed on or taken off the bus. Master devices use this field to define the bus alignment that slave devices must follow (i.e., obey). There is one major exception: option slot devices exchanging data with CPU or memory. Master option slot devices cannot use the master starting byte field to modify the alignment of data on the bus when interacting with CPU or memory, because CPU and memory require all bus data to be aligned to the workstation’s main memory. Below is a summary for option slot device usage of the master starting byte address:

- as slave, the master’s starting byte address (AD<31:30>) must be obeyed since all data must be aligned with main memory or CPU, and
- as master, anything can be written to AD<31:30>, however it will not be interpreted by the CPU or memory. All data on the bus will be and must be aligned to main memory or CPU.

$\overline{\text{AS}}$: Address strobe signals the start of a new bus transaction. The device that is currently the bus master asserts address strobe and places a slave address on the bus. All other devices on the bus should latch the address to determine if they are the device being addressed. Address strobe is only asserted for one cycle. The byte count cycle follows the address cycle.

READ: The READ signal serves two purposes. During the slave address cycle the master asserts READ for a read (master receives data) or deasserts READ for a write (master supplies data) transaction. After the slave address cycle, the bus master drives READ low to indicate that a bus transaction is taking place. The master holds READ low until the bus transaction ends naturally or through preemption.

MASDLY: The master asserts the master delay signal to throttle the data transfer rate. The signal has different meanings for read and write transactions. For a read transaction, the master asserts MASDLY when it is not ready to receive data in the current bus cycle. For a write, MASDLY indicates that the data currently on the bus is invalid. MASDLY is ignored during address and byte count cycles.

SLVDLY: The slave device asserts SLVDLY to throttle the data transfer rate. Like MASDLY, this signal has different meanings for read and write transactions. When the slave asserts SLVDLY during a bus read, the data on the bus is invalid. During a bus write, SLVDLY indicates that the slave cannot accept data in the current bus cycle. SLVDLY is pulled high on the CPU board so that it is asserted during the byte count cycle. The slave must drive SLVDLY high at the end of a bus transaction before tristating it because the pullup on the CPU board is not strong enough to pull SLVDLY high in one cycle.

$\overline{\text{BREQ}}$(n): Every bus master in the system has its own bus request signal. To request the bus, a device asserts its bus request signal. The device must hold its $\overline{\text{BREQ}}$(n) signal asserted until the bus arbiter grants it the bus, even if the device decides it does not need the bus. The bus master holds $\overline{\text{BREQ}}$(n) asserted until the end of the bus transaction. When the bus arbiter preempts the bus, the master device must deassert its bus request signal to indicate that it is off the bus. A preempted device cannot reassert its bus request signal until the bus arbiter deasserts the preemption signal. In order to avoid bus time-outs, a bus master must still accept requests from other bus masters even if it is requesting the bus.

$\overline{\text{BGNT}}$(n): Every bus master in the system has its own bus grant signal, which indicates to a master device that it owns the bus. The bus arbiter asserts a device’s matching bus grant signal for as long as the bus master asserts its bus request signal. When the bus arbiter preempts a master device, the arbiter does not deassert the bus grant signal until the bus master deasserts its $\overline{\text{BREQ}}$(n) signal.

$\overline{\text{BPRE}}$: The GIO arbiter asserts the bus preempt signal to preempt the current bus transaction. The bus master must respond to $\overline{\text{BPRE}}$ within {missing information: number of clock cycles is somewhere between four and nine} GIO32 clock cycles by deasserting its $\overline{\text{BREQ}}$(n) signal and asserting READ. Real-time devices are not preemptable and can ignore $\overline{\text{BPRE}}$.

$\overline{\text{INT}}$<2:0>: The GIO32 bus has three interrupt/status signals, $\overline{\text{INT}}$<2:0>, which are shared by all devices: the two GIO32 bus option slots and the graphics board(s). $\overline{\text{INT}}$<0> and $\overline{\text{INT}}$<1> are low priority interrupts. $\overline{\text{INT}}$<2> is a high priority interrupt. These signals can generate CPU interrupts and be read by the CPU.

CLK: The GIO32 clock runs at 25, 30, or 33 MHz. Data is valid at the rising edge of CLK.

$\overline{\text{RESET}}$: An asynchronous signal for resetting/restarting all devices on the bus.

2.5 GIO32 Bus Transfers

The GIO32 bus supports three kinds of cycles during a bus transaction. The first cycle of every transaction is a slave address cycle. The second cycle is the byte count cycle. Subsequent cycles are one or more data cycles. After the final data cycle of a transaction, AD<31:0>, READ, $\overline{\text{AS}}$, MASDLY, and SLVDLY must be tristated.

In the slave address cycle, the bus master sends the slave address out on AD<31:0> and asserts $\overline{\text{AS}}$. For a read transaction, the master asserts READ; for a write the master deasserts READ.

In the byte count cycle that follows, the master sends the transfer byte count on AD<12:0>. In this second cycle, the meaning of the READ signal changes to mean that a bus transaction is taking place on the bus. To this end, the master deasserts READ and holds it deasserted until the end of the bus transaction. The address strobe signal is not active in this cycle. The master also writes the address (offset) for the master starting address onto AD<31:30> during this cycle. All slave devices must obey these bits to place data onto or take data off of the bus so that the data on the bus is always aligned for the master device. Devices talking to the CPU or memory (regardless of whether they are slave or master) must always use the alignment described in Figure 2-1 or Figure 2-2 of this specification so that the data is properly aligned for main memory. Note that option slot master devices cannot use the “master starting byte address” feature with CPU and memory since the CPU and memory only support their own alignment.

> IMPLEMENTATION RESTRICTION:
> Due to implementation details of the bus arbiter, the maximum byte count for a master is 1023 for any single transaction.

The byte count and 32-bit slave address determine alignment (in slave memory) of the slave start and end addresses. The master starting byte and the byte count determine the alignment in master memory. The master starting byte field consists of the two least significant bits of the master’s byte address. For transfers where the slave and master addresses are aligned, the least significant bits (AD<31:30>) match on the first and second cycle: the slave address sent in the first cycle of the transaction and the master starting byte of the second. If the transfer is unaligned, the bits do not match. A slave GIO device must deal with the data (either place data on the bus during a read or rearrange data taken from the bus on a write) in the master’s alignment (according to these bits). Note that CPU and memory, as slave devices, do not support this feature; they do not rearrange data. CPU and memory always place data on the bus as it is aligned in memory and they assume that data taken from the bus is already aligned to main memory.

> CAUTION:
> Master option slot devices must not expect CPU or memory to obey the settings of AD<31:30> during the byte count cycle (i.e., the master starting byte address). CPU and memory only deal with data that is aligned to main memory.

For example, for a read, if the slave address is 0x2, the byte count is 0x2, and the master starting byte address is 0x1, the slave would read its own memory from byte addresses 0x2-0x3 and drive that data on the bus with the byte from slave address 0x2 placed at address 0x1 (i.e., onto AD<23:16>), packing the other byte in the same word. Figure 2-3 illustrates this case.

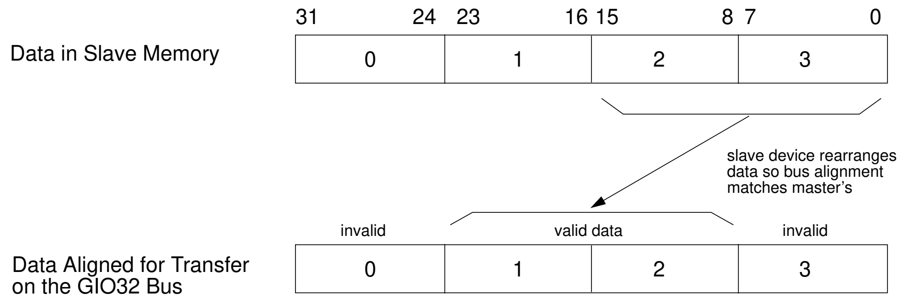

Figure 2-3. A Read Cycle: Slave's Placement of Data on Bus for Data Aligned in Slave Memory differently than in Master Memory.

Following the slave address and byte count cycles are a variable number of data cycles. The number depends on the byte count, bus preemption, and the flow control signals MASDLY and SLVDLY. The master drives MASDLY and the slave drives SLVDLY. These signals throttle the data transfer rate.

2.5.1 GIO32 Write Transactions

GIO32 bus write transactions take a minimum of three bus clock cycles. The transaction begins with the slave address and byte count cycles. In the slave address cycle, the master does the following:

- asserts $\overline{\text{AS}}$
- drives a 32-bit slave address on AD<31:0>
- deasserts READ to indicate that this is a write cycle.

In the byte count cycle, the master does the following:

- deasserts READ and holds it low until the end of the transaction
- drives a byte count on AD<12:0>
- drives the master starting byte address on AD<31:30>.

After the byte count cycle, the master can begin data transfer. During a write, the bus master drives data out onto the AD signals and deasserts MASDLY to indicate that valid data is on the bus. The master looks at SLVDLY at the end of the clock cycle to determine whether the slave is ready to accept the data. If the slave deasserts SLVDLY by the end of the cycle, the master knows that the slave will pick up the data at the rising clock edge, and that the master can put a new data word on the bus in the next cycle. Otherwise, if the slave does not deassert SLVDLY by the end of the cycle, the master must continue to drive the current word of data for additional bus clock cycles until it sees SLVDLY go low. The bus master can transfer a word of new data during every bus cycle that the slave is deasserting SLVDLY. The master can throttle the transfer rate by asserting MASDLY during a cycle when it is not ready to send a new word of data on the bus.

The bus master continues to transfer data until the byte count is satisfied or until the bus arbiter asserts the $\overline{\text{BPRE}}$ signal. The bus slave also keeps track of the number of bytes that have been transferred so that it can handle the last data word correctly if it is a partial word transfer.

At the end of the transaction, the master asserts MASDLY. The slave asserts SLVDLY within one cycle after the last data word and then tristates the signal. The bus master, if it does not wish to start another transaction, asserts READ, then tristates $\overline{\text{AS}}$, AD<31:0>, READ, and MASDLY. If the master has another transaction to do, it immediately begins the transaction by executing a slave address cycle.

If the bus arbiter preempts a bus transaction, the byte count will not be zero when the bus master asserts READ in response to the preemption. The slave needs to monitor READ so that it can detect a preemption and assert SLVDLY within one cycle. The bus master must keep all of the information -- remaining byte count and updated slave address -- necessary to restart the transaction where it was interrupted.

Figure 2-4, Figure 2-5, and Figure 2-6 illustrate sample GIO32 write transactions:

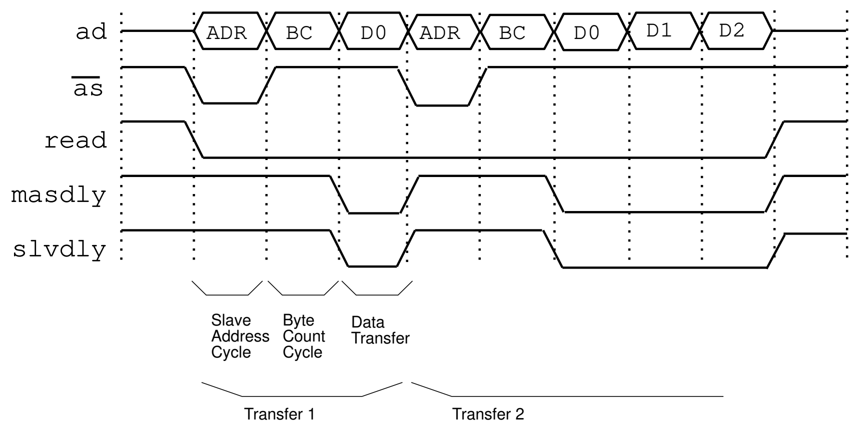

Figure 2-4. Back-to-Back Simple GIO32 Writes

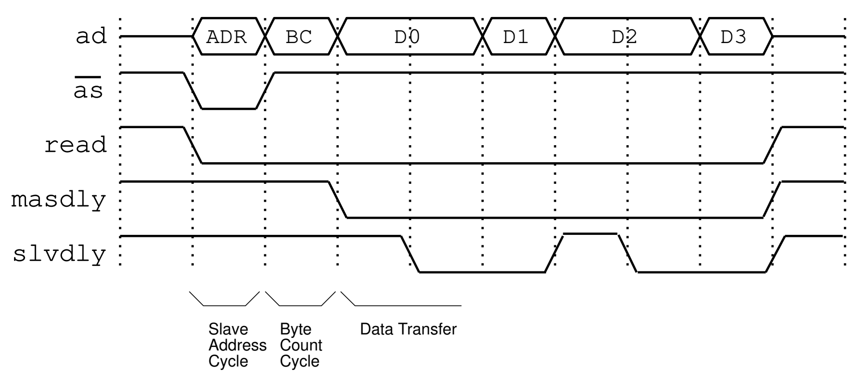

Figure 2-5. GIO32 Write, Slave Stalls

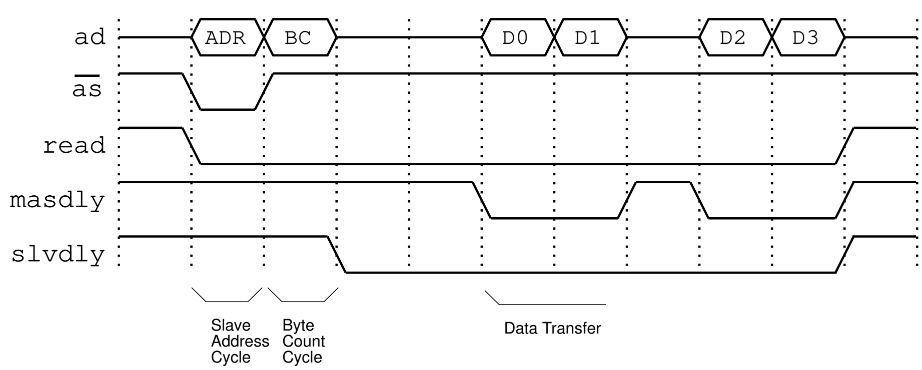

Figure 2-6. GIO32 Write, Master Stalls

2.5.2 GIO32 Read Transactions

GIO32 bus read transactions are very similar to write transactions except that the bus slave, not the bus master, drives the data on the bus.

The GIO32 bus requires one dead cycle on the bus when the device driving the bus changes. Thus a read transaction has a dead cycle between the master driving the byte count cycle and the slave driving the first data cycle, as well as between the final data cycle of a transaction and a subsequent address strobe that starts a new transaction. This requirement makes the minimum length of back-to-back read transactions four bus cycles.

The READ signal, asserted by the bus master during the slave address cycle to indicate a read transaction, is held deasserted from the byte count cycle onward to indicate that a bus transaction is taking place. During data cycles, the slave deasserts SLVDLY when it is driving valid data on the AD signals. The master deasserts MASDLY during cycles when is ready to take data off the bus.

Examples of GIO32 bus read transactions are shown in Figure 2-7, Figure 2-8, and Figure 2-9.

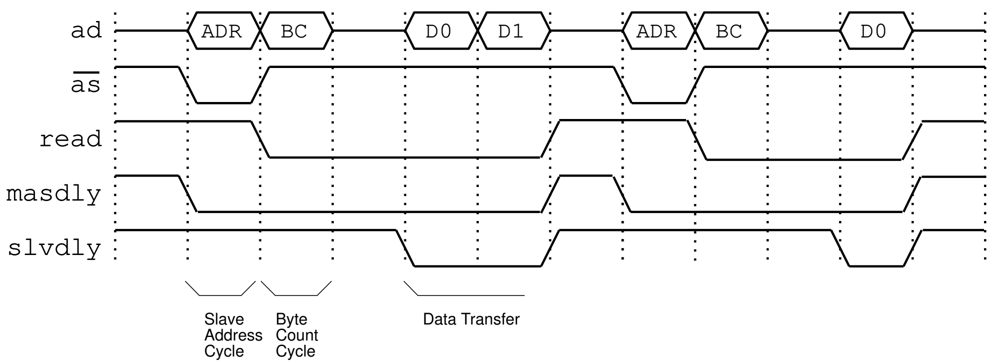

Figure 2-7. Simple GIO32 Reads

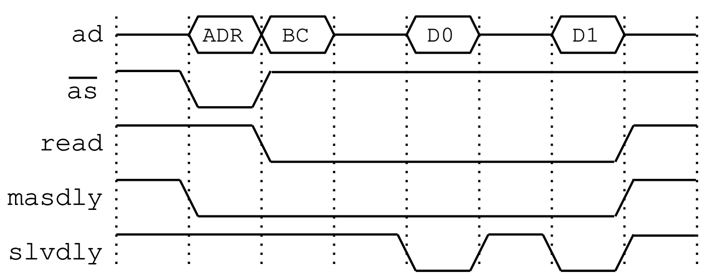

Figure 2-8. GIO32 Read, Slave Delay

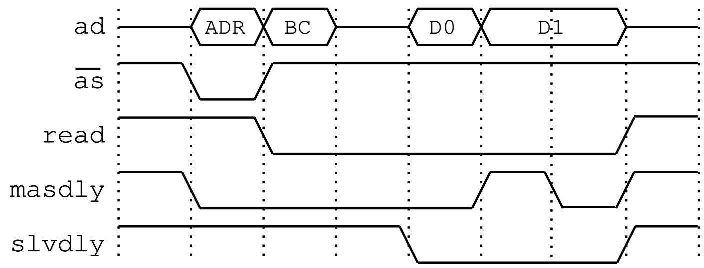

Figure 2-9. GIO32 Read, Master Delay

2.6 GIO32 Bus Arbitration

The GIO32 bus arbitration scheme supports three requirements:

- The CPU must run at a minimum, guaranteed rate in the most highly loaded system to allow for acceptable interrupt response times.
- Long-burst DMAs for devices such as the graphics subsystem must be allowed to use the bus for long stretches of time to move large amounts of data.
- Real-time I/O devices such as network connections (e.g., Ethernet, FDDI) and audio must be guaranteed access to the bus within a predefined maximum delay.

These three requirements -- to support the CPU, long burst, and real-time I/O devices -- result in the following rules of bus behavior.

- Long-burst devices have low priority as well as limits on the time they can be master of the bus. A long-burst device can perform multiple transactions once it becomes bus master. The bus arbiter preempts a long-burst device master to service a real-time I/O device or because the long-burst device has used up its time slot on the bus.
- The CPU is a long-burst device, but differs in two ways from all other long-burst devices. First, the bus arbiter will only preempt CPU bus mastership between bus transactions. Second, the CPU becomes bus master after every grant to some other long-burst bus master.
- Real-time I/O devices have high priority. They can use the bus for no more than a fixed unit of time -- 5 microseconds -- per acquisition and can only request the bus at a predefined frequency.

The GIO32 bus arbiter has two time slot registers--DELAY and BURST--that limit the time a bus master can hold the bus. The DELAY register limits the CPU's time on the bus. The BURST register limits the time on the bus for all other long-burst devices. The operating system chooses the time values to load into these registers when a device becomes bus master. For example, the CPU time limit might be set to 2 microseconds, while the time limit for all other long-burst devices might be 38 microseconds. If a bus master is still asserting its $\overline{\text{BREQ}}$(n) signal when its time limit expires, the bus arbiter preempts the bus. If the CPU receives and services an interrupt during a bus transaction, the CPU resets the time value in the BURST register to zero and the DELAY register to its maximum value so that the CPU can immediately become bus master.

Real-time I/O devices preempt long-burst devices immediately. The Indigo Peripheral Controller chip (HPC), which manages Ethernet and SCSI access, is the highest priority real-time I/O device in the system, followed by option slot 0 and then option slot 1. A preempted long-burst device regains the GIO Bus after the preempting transaction completes. When no other device requests the bus, the CPU becomes bus master.

> IMPLEMENTATION RESTRICTION:
> Due to implementation details of the bus arbiter, non-CPU, long burst masters must not simply drive $\overline{\text{BREQ}}$(n) inactive when they wish to relinquish the bus. They must wait for $\overline{\text{BPRE}}$ to be asserted and then deassert $\overline{\text{BREQ}}$(n).

2.6.1 Arbitration Handshake

Connecting:
Each bus master on the GIO32 bus connects to the bus arbiter with a pair of $\overline{\text{BREQ}}$(n) and $\overline{\text{BGNT}}$(n) signals. A device requests the bus by asserting its $\overline{\text{BREQ}}$(n) signal. The arbiter grants the bus to the requesting device by asserting the matching $\overline{\text{BGNT}}$(n) signal. Once a device asserts its $\overline{\text{BREQ}}$(n) signal, it must hold the signal asserted until the arbiter has granted the bus.

> IMPLEMENTATION RESTRICTION:
> Due to implementation details of the bus arbiter, it is possible that the arbiter may assert $\overline{\text{BGNT}}$(n) for one cycle and then deassert it, without actually granting the bus. Therefore, bus masters must not conclude that they have been granted the bus until $\overline{\text{BGNT}}$(n) has been asserted for two consecutive cycles. Once asserted, $\overline{\text{BREQ}}$(n) must not be deasserted until the bus has been granted.

Doing the transaction:
A bus master must keep its $\overline{\text{BREQ}}$(n) signal asserted until the bus transaction is complete.

Terminating:
When the bus transaction is complete, the bus master deasserts its $\overline{\text{BREQ}}$(n) signal and the arbiter then deasserts the $\overline{\text{BGNT}}$(n).[2]

2.6.2 GIO32 Bus Preemption

The GIO32 bus arbiter preempts a long-burst bus master when a real-time I/O device needs the bus or when the bus master uses up its time slot on the bus. The arbiter asserts $\overline{\text{BPRE}}$. In response, the current bus master asserts READ within {missing information: four to nine} bus cycles to indicate end of transaction. If a write transaction was underway, the master stops driving AD and MASDLY in the same cycle as the deassertion of READ. In the case of a preempted read transaction, the slave may continue driving data as the master is deasserting READ; however, the data driven will not be accepted by the bus master. The bus master must keep track of the slave address and remaining byte count in order to resume the transfer later.

A bus master can preempt its own bus transaction before the byte count has been satisfied and never resume the transaction. This type of preemption is useful for devices that may not know the byte count at the start of the transaction. The master can drive a maximum byte count during the byte count cycle and then preempt the transaction by asserting the

2. Note the implementation restriction (on page 2-9) associated with deasserting $\overline{\text{BREQ}}$(n).

READ signal when it has received the desired bytes. Note that this technique requires the last transfer to be aligned to the bus because the byte count cannot indicate how many bytes to transfer on the last cycle.

> IMPLEMENTATION RESTRICTION:
> Due to implementation details of the bus arbiter, the self-preemption with unfinished transaction feature will not work and must not be used. Bus masters ***must*** provide an accurate and exact byte count.

Preemption examples are presented in Figure 2-10 to Figure 2-13.

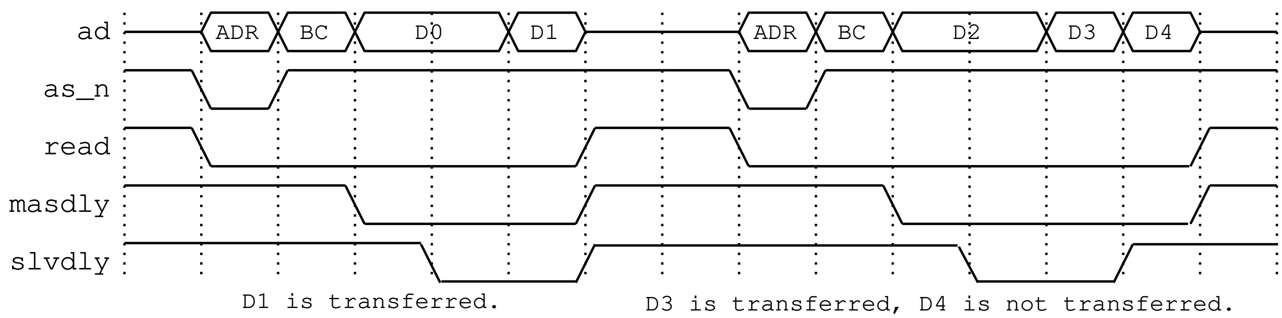

Figure 2-10. Preempted GIO32 Write, Slave Stall

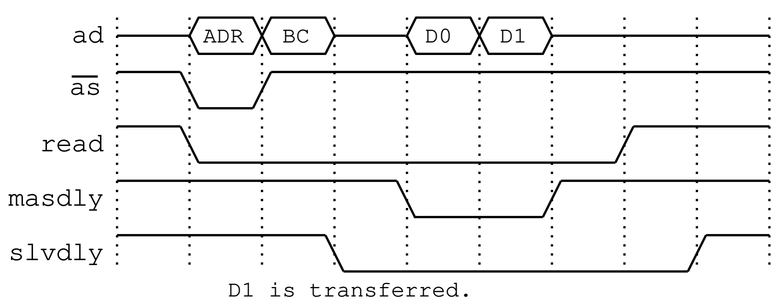

Figure 2-11. Preempted GIO32 Write, Master Stall

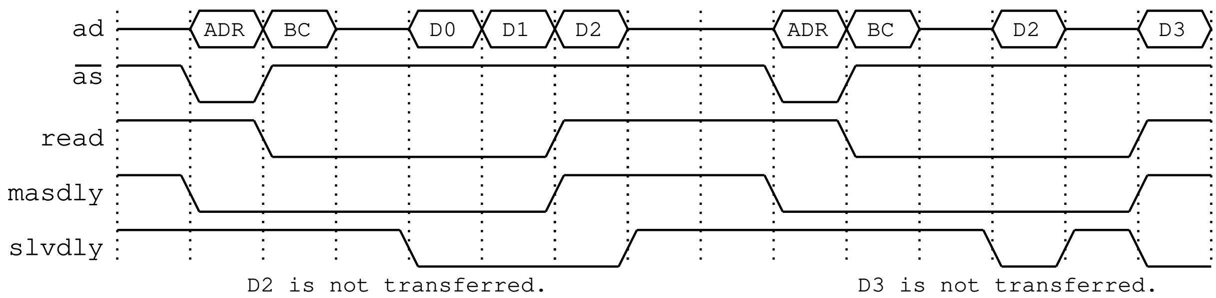

Figure 2-12. Preempted GIO32 Read, Slave Stall

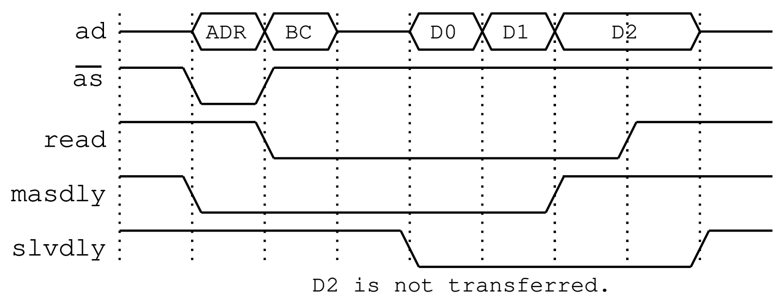

Figure 2-13. Preempted GIO32 Read, Master Stall

2.7 GIO32 Bus Time-outs

A GIO32 bus transaction times out if a slave does not deassert SLVDLY in reply to an address strobe within 25 microseconds after the byte count cycle. This time-out prevents accesses to non-existent locations on the GIO32 bus from hanging the bus. When a time-out occurs, the bus arbiter deasserts SLVDLY until the transaction completes. Then the arbiter generates a bus error interrupt to the CPU. The device that was transferring data does not know that a time-out occurred. Software can use this time-out mechanism to determine whether a device exists in a GIO32 slot. The guaranteed bus acquisition time of five microseconds for real time devices may be violated when the GIO32 bus times out.

2.8 Issues for Slave-only GIO32 Devices

GIO32 devices that will not perform as master devices, can simplify their design in the following manner:

- preemption will never occur and so is not an issue,
- all transfers will include only one data cycle,
- AD<12:0> (the byte count) in the byte count cycle can be ignored, and
- three signals are not used: $\overline{\text{BREQ}}$, $\overline{\text{BGNT}}$, and $\overline{\text{BPRE}}$.

2.9 Interrupts

Each interrupt from a GIO device will cause every device driver associated with the interrupt line to be called. For example, if two device drivers are associated with $\overline{\text{INT}}$0, whenever an interrupt occurs on this line each driver is called in turn, thus giving each the opportunity to process any interrupt generated by it's device (at that level). Since interrupt levels may be shared, each device (or the protocol between the device and the host) must provide a reliable mechanism for allowing a device driver to determine whether a given interrupt call was caused by the associated device or not. All device drivers must, at all times, be able to gracefully handle spurious interrupts.[3]

> CAUTION:
>
> GIO Interrupts 2 and 0 ($\overline{\text{INT}}$2 and $\overline{\text{INT}}$0) are reserved for use by graphics boards only. Option slot devices must use $\overline{\text{INT}}$1.

Note: Each device must provide a register

2.10 Identifying GIO Devices

2.10.1 Product Identification Word

During the configuration procedure, the CPU does a slave (host) word read of the Product Identification Word associated with each potential GIO device. The read must be done as a word read. (This ensures endian independence.) These reads establish the presence and identity of all the GIO devices that are present. (GIO devices do not use the "address probe" scheme used in VME-based devices.) For slot devices, the Product Identification Word is read by accessing the first (base) address within the slot's address range. During the slave cycle following any read of the slot's base address, the Product Identification Word information is driven onto AD<31:0>.

The Product Identification Word contains five pieces of information as listed below and illustrated in Figure 2-14.

-	Bits <7:0>:	Product ID Code	unique ID assigned by SGI (see section "Product ID Code" for more detail)
-	Bits <15:8>:	Product Revision	product revision value assigned by the manufacturer
-	Bit <16>:	GIO Interface size	0=32bits, 1=64bits For GIO32 and GIO32-bis, must be zero. For GIO64, may be 0 or 1.
-	Bit <17>:	ROM Present	0=no ROM, 1=ROM present When set to one, indicates that the next three words of the device's address space are dedicated to special registers, as explained in the section "Special Registers". When set to zero, the device's address space contains no reserved areas except the base address.
-	Bits <31:18>:	Manufacturer's Code	value and purpose assigned by the manufacturer.

3. How to handle the linking of a device driver into one or more interrupt chains is TBD. One technique is a table of routine addresses with an entry per slot per level. All entries would be initialized to a stub routine address before calling the boot-time `autoconfig` which would overwrite any used entry with the actual driver entry point(s). In this way, the interrupt dispatch code would not need any conditional tests.

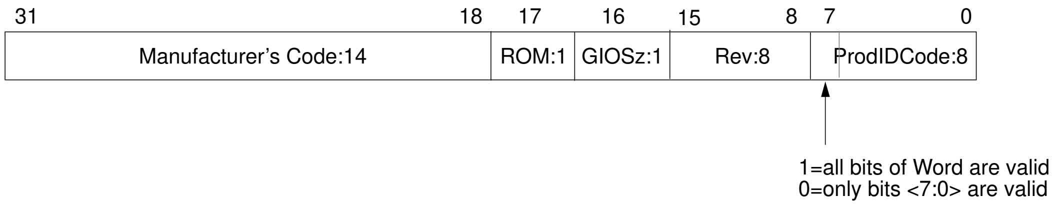

Figure 2-14. Product Identification Word

2.10.2 Product ID Code

The lowest byte of the Product Identification Word contains a unique, 8-bit Product ID Code that is hardwired into each GIO device. During the slave cycle following any read of the slot's base address, this 8-bit Product ID Code is driven onto the lower bits (bits 7:0) of AD (at the same time as all the other Product Identification Word).

Product ID Codes are unique across all GIO devices and must be registered with Silicon Graphics.[4]

When the assigned Product ID Code has bit 7 equal to zero, the device does not provide meaningful (neither correct nor repeatable) data on AD<31:8> during the slave cycle following a read of a slot's base address. Such boards must be recognized based on AD<7:0> and the other fields must be treated by software as if they contained the following values:

- Bits <7:0>: Product ID Code — unique ID assigned by SGI where bit 7=0
- Bits <15:8>: Product Revision — 0
- Bit <16>: GIO Interface size — 0 (i.e., =32bits)
- Bit <17>: ROM present — 0 (i.e., =no ROM)
- Bits <31:18>: Board Manufacturer — not defined (i.e., manufacturer must be deduced from Product/ Board Identification code).

2.11 GIO32 Option Slot Issues

2.11.1 Address Ranges

Each option slot has a 2 MByte address range:

- slot 0 = 0x1f400000 to 0x1f5fffff,
- slot 1 = 0x1f600000 to 0x1f7fffff.

Note: addresses 0x1f800000 to 0x1f9fffff are reserved for future definition by Silicon Graphics.

2.11.2 Configuration Registers

GIO32 option cards configure themselves by writing into their Configuration Register. Each slot has its own register in the format shown in Figure 2-15. Slot 0's is located at 0x1FA20000. Slot 1's is at 0x1FA20004.

4. The procedure and contact point for doing this are TBD. The numbers will be recorded in an appropriate header file in the kernel source tree.

The setting of bit 1 communicates the device type:1 indicates real-time I/O and 0 indicates long-burst. Bits <31:2> and bit 0 are not used.

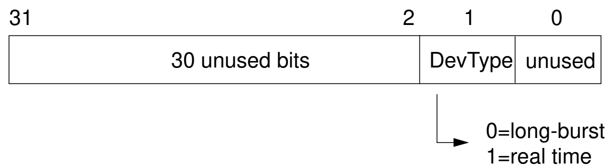

Figure 2-15. Slot Configuration Register Format

2.11.3 Special Registers within the Device Address Range

When an option slot device has ROM, it indicates the ROM's presence to the CPU by setting to one the ROM Present bit in the Product Identification Word. Whenever this bit is set, the three words following the base address of the device address range are reserved as three special 32-bit registers, as explained below.

1. The word at base address + 0x4 is the optional Board Serial Number register. This serial number is assigned by the manufacturer.
2. The word at base address + 0x8 is the ROM Index register. This register is written to zero by the CPU to initiate reading of the ROM. Subsequent reads of the ROM Read register cause this register to automatically increment by 4, indicating the currently available word in the ROM Read register.
3. The word at base address + 0xc is the ROM Read register. This register contains the contents of one word from the ROM. It always contains the word from the address indicated in the ROM Index register.

2.11.4 Device Interface and Configuration

GIO32 option slot device drivers configure themselves by calling `setgiovector`, `setgioconfig`, and the apporpriate `splgio`#. These calls and their parameters are listed below:

- `setgiovector(INT_LEVEL, GIO_SLOT, GIO_FUNC, GIO_ARG);`
- `setgioconfig(GIO_SLOT, GIO_ARB);`
- `splgio0();`
- `splgio1();`
- `splgio2();`

`setgiovector(INT_LEVEL, GIO_SLOT, GIO_FUNC, GIO_ARG)`

A device driver registers its interrupt service function with the kernel's interrupt dispatcher by calling `setgiovecotor`. The call requires four parameters explained below.

1. The *level* parameter is an integer specifying which one of the three possible GIO bus interrupts is used by the device. This parameter must be one of the following:
   - 0 =GIO_INTERRUPT_0 ,
   - 1 =GIO_INTERRUPT_1 , or
   - 2 =GIO_INTERRUPT_2 .

CAUTION:

Due to limitations in the graphics subsystem, GIO Interrupts 2 and 0 ($\overline{\text{INT}}$2 and $\overline{\text{INT}}$0) are reserved for use by graphics boards only. Option slot devices must use $\overline{\text{INT}}$1.

2. The *slot* parameter is an integer and specifies the physical slot occupied by the GIO bus board. This parameter must be one of the following:
   - 0 =GIO_SLOT_0 ,
   - 1 =GIO_SLOT_1 , or
   - 2 =GIO_SLOT_GFX .
3. The *function* parameter is a pointer to the interrupt service routine that will be called when the associated interrupt occurs.

   Note that because interrupts are shared among devices, `func` [i.e., `void (*func)(int);`] may be called when there is no pending interrupt from the particular slot specified, in which case `func` should simply return. The interrupt handler therefore needs to be able to determine when its device is actually interrupting, and when it is not, in a non-destructive manner.
4. The *argument* paramenter is an integer that is passed to the interrupt service routine whenever the interrupt routine is called; *arg* may contain any value. The interrupt service routine will be called with the processor interrupt mask set to disable further interrupts from the device.

`setgioconfig(GIO_SLOT, GIO_ARB)`

Device drivers call `setgioconfig` to set up the GIO bus arbitration mode for the GIO slot specified by the *slot* parameter. The call requires two parameters explained below.

1. The *slot* parameter is an integer and specifies the physical slot occupied by the GIO bus board. This parameter must be one of the following:
   - 0 =GIO_SLOT_0 ,
   - 1 =GIO_SLOT_1 , or
   - 2 =GIO_SLOT_GFX .
2. The *arbitration* mode is specified as a bit-wise OR of the following two flags:
   - GIO_CONFIG_LONG
where 0=real-time device and 1=long-burst device,
   - GIO_CONFIG_SLAVE
where 0=device is both master and slave, and 1=device is slave only.

`splgio0()`, `splgio1()`, or `splgio2()`

The `splgio0()`, `splgio1()`, and `splgio2()` functions set the processor interrupt mask to block GIO bus interrupts. `splgio0` sets the mask for $\overline{\text{INT}}$0, `splgio1` sets the mask for $\overline{\text{INT}}$1, etc..

2.12 INDIGO-specific Information

2.12.1 Indigo GIO32 Bus Arbiter

The Processor Interface Controller chip (PIC) is the GIO32 arbiter in the Indigo system. The PIC acts as master device for transactions between the CPU and GIO32 devices. These transactions are one to four bytes in length. PIC timing and synchronization with the CPU, which runs at a different speed than the GIO32 bus, requires about fifteen bus clock cycles to complete a single-word GIO32 bus read transaction.

2.12.2 Indigo Memory Timing

When a GIO32 bus device initiates a long-burst read transfer from main memory, the Indigo memory subsystem requires seven cycles for RAS and CAS before it drives the first data word on the GIO32 bus. Subsequent data words are driven at the full bus speed of one 32-bit word per bus cycle unless throttled by the bus master device or by the memory controller.

2.13 GIO32 Option Card Requirements

2.13.1 Mechanical Details

The Indigo CPU board has two 96-pin, high density, GIO32 bus connectors on its top side. The connector is a Fujitsu 230-series straight header with post. An option card plugs into a GIO32 bus connector, pops onto stabilizing card standoffs on the CPU board, and presents its I/O connections (if any) out an I/O panel on the back of the Indigo system box. Each option card has its own I/O connector area, which is 2.84 inches long and 0.80 inches wide.

Option cards are 6.44 inches long by 3.375 inches wide. Components may be placed on both sides of the card, with a top side component height limit of 0.65 inches and a bottom side limit of 0.10 inches.

Figure 2-16 shows two option cards mounted on the Indigo CPU board:

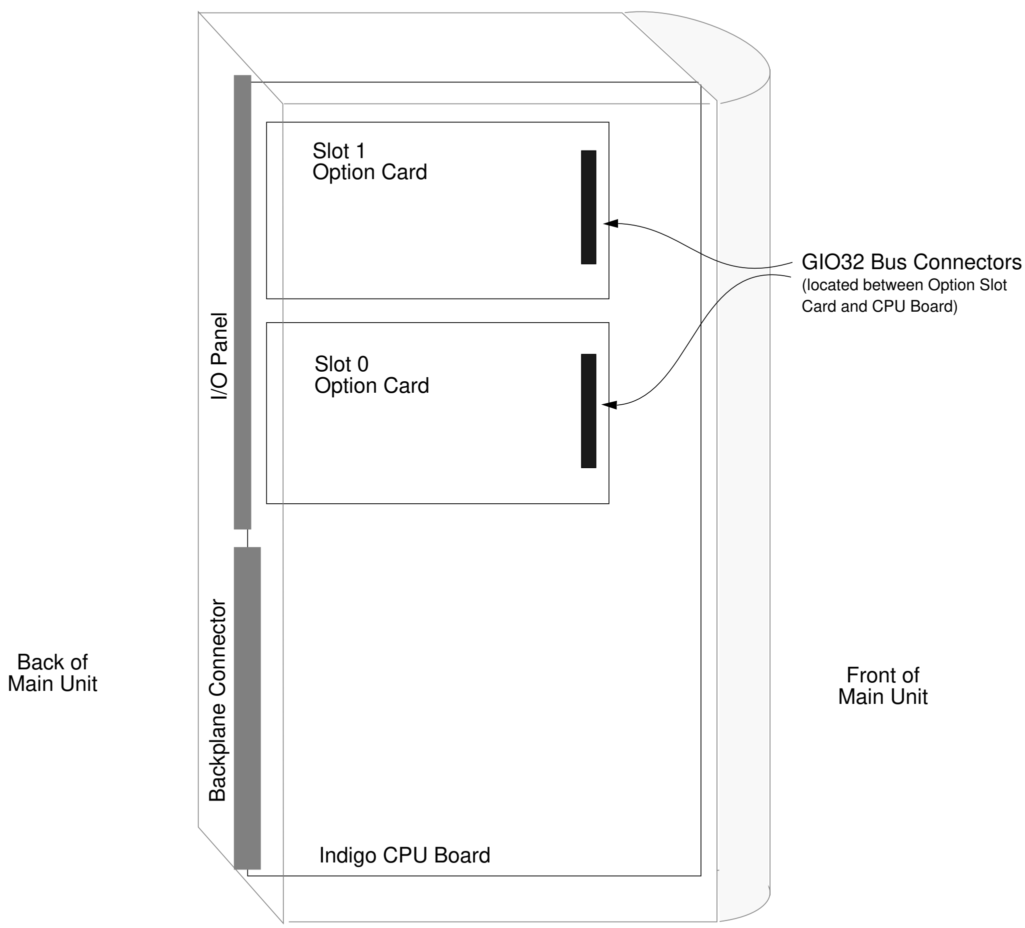

Figure 2-16. Option Slots for Indigo.

2.13.2 GIO32 Electrical Details

A GIO32 option card must meet the following electrical specifications:

- 13.6 watts maximum power consumption
- +5 volts @ 2 amps
- +12 volts @ 0.15 amps
- -12 volts @ 0.15 amps
- TTL-compatible signals with CMOS compatible input currents
- 50 pF AC loading for clock and reset signals
- 15 pF AC loading for all other bus signals
- 74FCT652AT or similar bus transceivers for AD<31:0>
- FCC Class B with 6 dB margin

2.14 GIO32 Slot Pinout

TABLE 2 GIO32 Pinout.

SIGNAL	PIN#	SLAVE	MASTER
RESERVED N/C	01	-	-
RESERVED N/C	02	-	-
RESERVED N/C	03	-	-
RESERVED N/C	04	-	-
GND	05	-	-
RESERVED N/C	06	-	-
RESERVED N/C	07	-	-
RESERVED N/C	08	-	-
RESERVED N/C	09	-	-
GND	10	-	-
RESERVED N/C	11	-	-
GIO.$\overline{RESET}$	12	I	I
+12V	13	-	-
GIO.$\overline{INT0}$	14	O	O
GIO.$\overline{BREQ}$(n)	15	-	O
GND	16	-	-
GIO.READ	17	I	O
GIO.$\overline{AS}$	18	I	O
VCC	19	-	-
GIO.MASDLY	20	I	O
GND	21	-	-
GIO.AD00	22	I/O	I/O
GIO.AD02	23	I/O	I/O
GND	24	-	-
GIO.AD04	25	I/O	I/O
GIO.AD06	26	I/O	I/O
GND	27	-	-
GIO.AD08	28	I/O	I/O
GIO.AD10	29	I/O	I/O
GND	30	-	-
GIO.AD12	31	I/O	I/O

SIGNAL	PIN#	SLAVE	MASTER
GIO.AD14	32	I/O	I/O
SLOTID	33	-	-
GND	34	-	-
GIO.AD16	35	I/O	I/O
VCC	36	-	-
GIO.AD18	37	I/O	I/O
GND	38	-	-
GIO.AD20	39	I/O	I/O
GIO.AD22	40	I/O	I/O
GND	41	-	-
GIO.AD24	42	I/O	I/O
GND	43	-	-
GIO.AD26	44	I/O	I/O
GND	45	-	-
GIO.AD28	46	I/O	I/O
GIO.AD30	47	I/O	I/O
-12V	48	-	-
RESERVED N/C	49	-	-
GND	50	-	-
RESERVED N/C	51	-	-
RESERVED N/C	52	-	-
RESERVED N/C	53	-	-
GND	54	-	-
RESERVED N/C	55	-	-
RESERVED N/C	56	-	-
RESERVED N/C	57	-	-
RESERVED N/C	58	-	-
GND[a]	59	-	-
GIO.$\overline{\text{BGNT}}$(n)	60	-	I
GIO.$\overline{\text{BPRE}}$	61	-	I
GND	62	-	-
GIO.$\overline{\text{INT01}}$	63	O	O
GIO.$\overline{\text{INT02}}$	64	O	O
GND	65	-	-
GIO.SLVDLY	66	O	I
GND	67	-	-

SIGNAL	PIN#	SLAVE	MASTER
GIO.CLK	68	I	I
GND	69	-	-
GIO.AD01	70	I/O	I/O
VCC	71	-	-
GIO.AD03	72	I/O	I/O
GND	73	-	-
GIO.AD05	74	I/O	I/O
GIO.AD07	75	I/O	I/O
GND	76	-	-
GIO.AD09	77	I/O	I/O
GND	78	-	-
GIO.AD11	79	I/O	I/O
VCC	80	-	-
GIO.AD13	81	I/O	I/O
GND	82	-	-
GIO.AD15	83	I/O	I/O
GND	84	-	-
GIO.AD17	85	I/O	I/O
GIO.AD19	86	I/O	I/O
GND	87	-	-
GIO.AD21	88	I/O	I/O
GND	89	-	-
GIO.AD23	90	I/O	I/O
GIO.AD25	91	I/O	I/O
VCC	92	-	-
GIO.AD27	93	I/O	I/O
GND	94	-	-
GIO.AD29	95	I/O	I/O
GIO.AD31	96	I/O	I/O

a. This pin is used for differentiating btween GIO32 and GIO32-bis boards.

Chapter 3

GIO32-bis Specification

3.1 Introduction

The GIO32-bis bus is a 32-bit version of the GIO64 Bus. It obeys the timing specifications of the GIO64 with the GIO32 pinout. In general, GIO32-bis is correctly described by Chapter 2 except for those items specified below. Two items differentiate GIO32 from GIO32-bis:

- one of the GIO32 pins is redefined for differentiating GIO32 and GIO32-bis devices, and
- the timing protocols for master and slave delay signals.

3.2 Pin for Differentiating GIO32 from GIO32-bis

The grounded pin labeled GND #59 in the GIO32 Specification, is tied high on GIO32-bis slots. This makes it possible for GIO devices to tell whether they are plugged into a GIO32 or a GIO32-bis slot.

3.3 GIO32-bis Bus Transfers

This timing described in this section is identical to that in the GIO64 section “Nonpipelined GIO64 Bus Transfers.” The use of MASDLY and SLVDLY is slightly different from (and incompatible with) the section “GIO32 Bus Transfers” in the GIO32 Specification. The device *driving* data onto the bus asserts its delay signal (MASDLY for memory writes or SLVDLY for memory reads) synchronously with the data, as in GIO32. However, in GIO32-bis, the device *receiving* data from the bus must assert its delay line (MASDLY for memory reads, SLVDLY for memory writes) one clock earlier than in GIO32.

3.3.1 GIO32-bis Bus Writes

GIO32-bis bus writes start with a slave address and byte count cycle. The READ signal will be deasserted in the slave address cycle to indicate that this is a write cycle. After the slave address cycle, the READ signal is used to indicate that a bus cycle is in progress and will remain low for the rest of the transfer.

After the third cycle, data can be transferred. During a write, the bus master will drive data out onto the bus and drive MASDLY low to indicate that valid data is on the bus. The master looks at the SLVDLY signal that was sent the cycle before to determine if the slave can accept the data. If the SLVDLY signal was not low in the previous cycle, the master must continue driving the current data until SLVDLY is low in the previous cycle. This is a change from the GIO32 bus in that the SLVDLY signal is flopped and used in the next cycles instead of being used in the cycle it is on the bus. This change is necessary so that the SLVDLY signal can be directly registered before any gating takes place. This provides one whole cycle for the signal propagation between chips and one whole cycle for on-chip gating. The GIO32 bus scheme works at 33 MHz, but at higher speeds it becomes very difficult to get the timing to work. The bus master can continue to transfer new data every cycle that SLVDLY from the previous cycle is low. If SLVDLY was not low then the current data must be driven until SLVDLY is deasserted in the previous cycle. The master can throttle the transfer by driving MASDLY high during a cycle that it does not have new data to transfer. Note that this remains the same as GIO32 and is not sent one cycle early like SLVDLY.

Since SLVDLY is being sent for the next cycle, it will take an extra cycle for all complete transfers, (not one cycle per word). A one word write will take at least four cycles. This is one more cycle then it took with the GIO32 bus.

The bus master continues to transfer data until the byte count is satisfied. The bus slave also keeps track of the number of bytes that have been transferred so that the last write will be handled correctly if it is a partial word transfer. At the end of the transfer the master drives READ high. Two cycles after the slave receives the last piece of data it drives SLVDLY high and in the following cycle tristates the SLVDLY signal. The bus master will tristate the AD, READ, $\overline{\text{AS}}$, and MASDLY signals two cycles after the last piece of data is transferred if it does not have another transfer to execute. The master does not have to drive the MASDLY and READ signals high before tristating them. If it does have another transfer it can drive the address in the cycle immediately following the last piece of data.

If a transfer is preempted, the byte count will not be zero when the READ signal is driven high by the bus master. The slave needs to monitor the READ signal and not just the remaining byte count, so that it can tell if a transfer has been preempted. The bus master must keep all of the information that is necessary to restart the transfer where it left off. This includes the slave data address. Below are some examples of GIO32-bis writes.

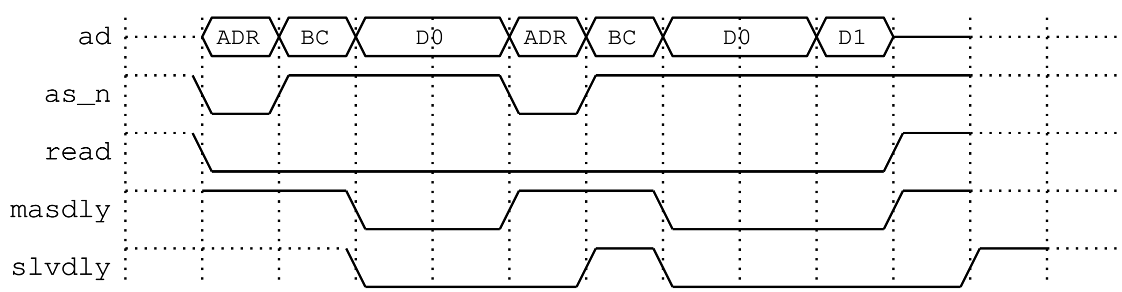

Figure 3-1. Back-to-Back Simple GIO32-bis Writes

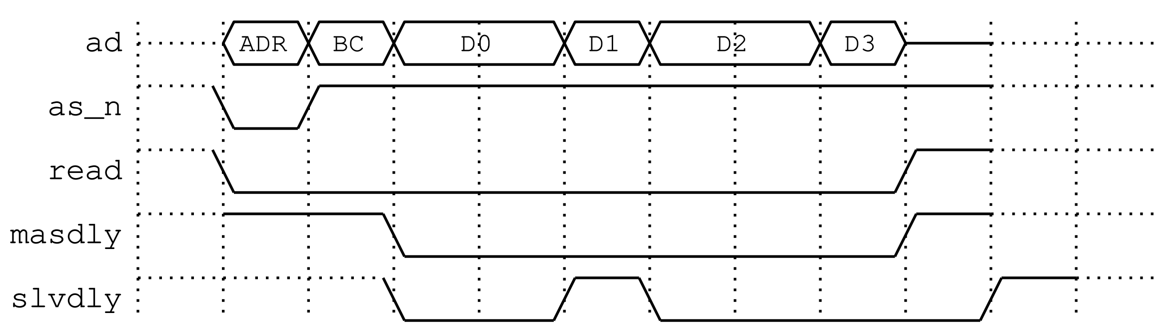

Figure 3-2. GIO32-bis Write, Slave Stall

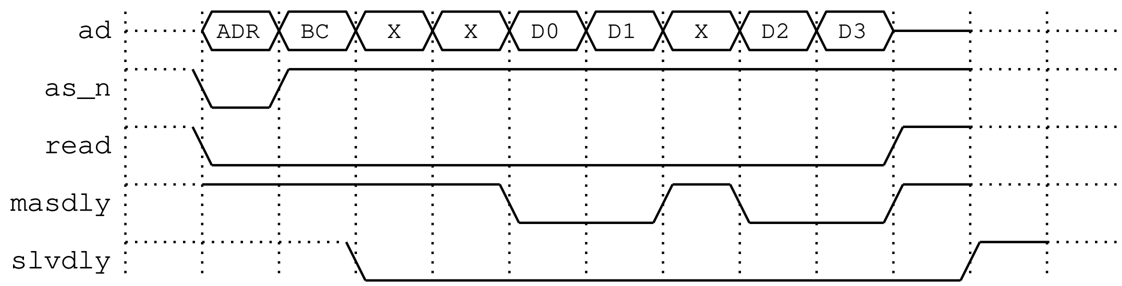

Figure 3-3. GIO32-bis Write, Master Stalls

3.3.2 GIO32-bis Bus Reads

GIO32-bis bus reads are a lot like GIO32-bis bus writes except that the slave is sending the data over the bus instead of the master. Notice that the READ signal is used to indicate that a bus cycle is in progress after the slave address cycle, by being deasserted for the rest of the transfer. Since the slave is sending the data, the SLVDLY signal is used to indicate that there is valid data on the bus and the MASDLY signal is used to indicate to the slave that the master can accept data in the next cycle. Note that this is different from the GIO32 bus in that MASDLY is sent one cycle earlier. The slave must tristate the AD bus signals in the cycle after the last piece of data is transferred. The slave must drive the SLVDLY signal high in the cycle after the last piece of data is transferred and then tristate it in the following cycle. The master will tristate the $\overline{\text{AS}}$, READ, and MASDLY signals three cycles after the last piece of data has been transferred if it does not have another transfer to execute. If it does have another transfer to do it can drive the address two cycles after the last piece of data has been transferred. Some examples of GIO64 reads are shown below.

It is important that the slave does not wait for MASDLY to be deasserted before it drives the read data and deasserts SLVDLY, or that once the slave has deasserted SLVDLY and driven the read data, that it does not stop driving it, even if it asserted SLVDLY until the cycle after the master deasserts MASDLY. Figure 3-6 shows this condition.

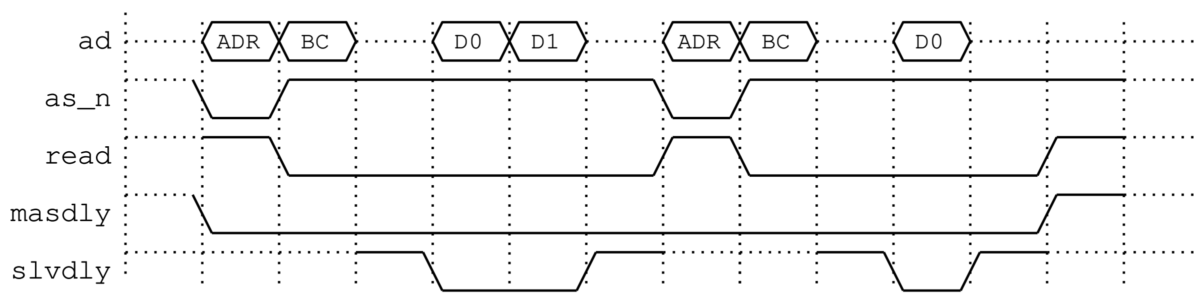

Figure 3-4. Simple GIO32-bis Read

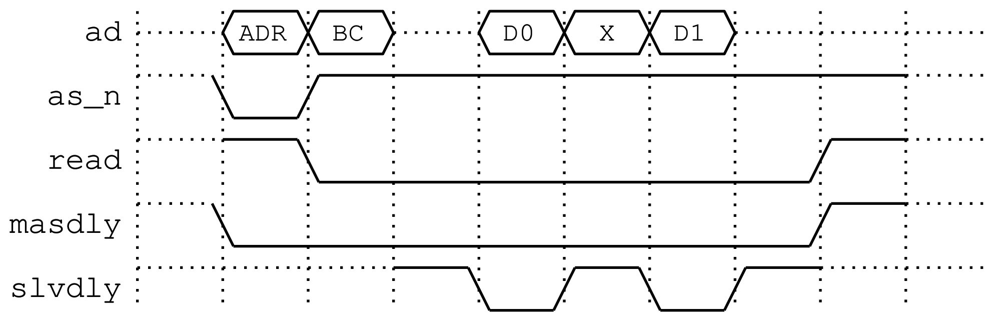

Figure 3-5. GIO32-bis Read, Slave Delay

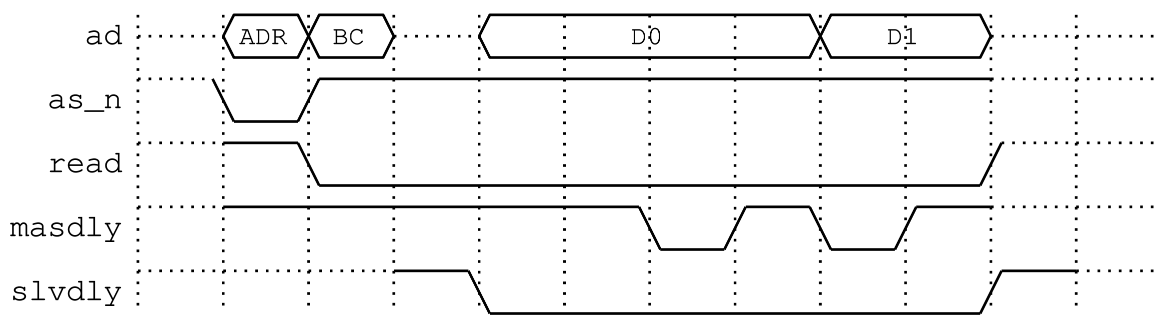

Figure 3-6. GIO32-bis Read, Master Delay

3.3.3 GIO32-bis Preemption

When a GIO32-bis device gets preempted, the master drives the READ signal high to indicate an end of the transfer. For GIO32-bis writes, the bus master also stops driving data and MASDLY in the same cycle as it stops driving READ. GIO32-bis reads are preempted in basically the same way, except that the slave may continue driving data in the cycle that the master drives the READ and MASDLY signals high. The data that the slave sends in the cycle that READ is driven high is not accepted. The bus master must tristate all of the signals it is driving in the cycle after READ is driven high. The slave must drive SLVDLY high in the cycle after READ is driven high and tristate it in the following cycle. Different preemption cases are shown below.

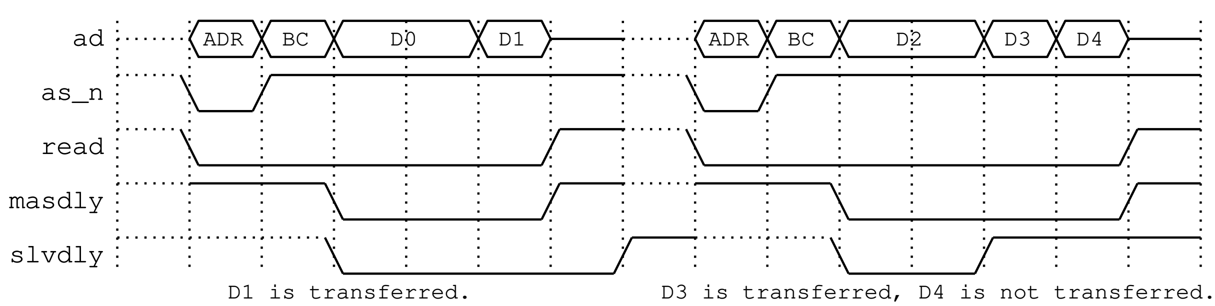

Figure 3-7. Preempted GIO32-bis Write, Slave Stall

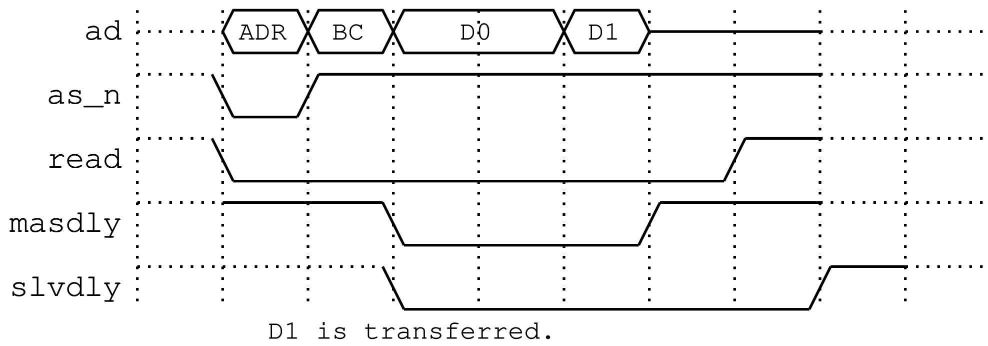

Figure 3-8. Preempted GIO32-bis Write, Master Stall

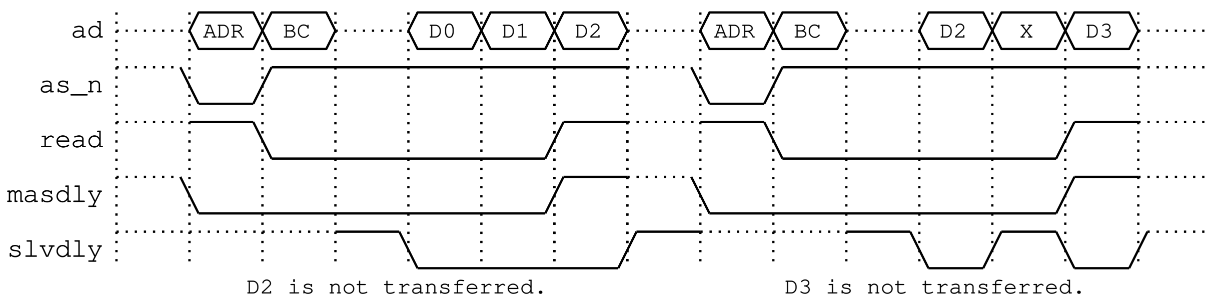

Figure 3-9. Preempted GIO32-bis Reads, Slave Stall

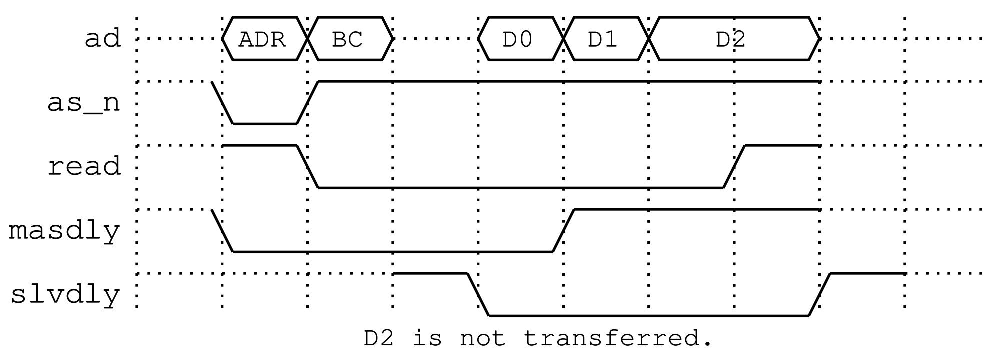

Figure 3-10. Preempted GIO32-bis Reads, Mater Stall

This page has been left blank intentionally.

Chapter 4

GIO Bus 64

4.1 Introduction

The GIO64 bus is a 64-bit, synchronous, multiplexed address-data bus that can run at speeds up to 40 MHz. The bus supports both 32- and 64-bit GIO64 devices. This bus is used to connect high speed devices to main memory. It is also possible for the CPU to issue reads and writes to GIO64 devices.

The maximum bandwidth of this bus is 320 MBytes/second for burst transfers running at 40 MHz.

There are two different forms of the GIO64 bus. The first form is the bus that is local to system memory. This form will just be referred to as the GIO64 bus or the nonpipelined GIO64 bus. The second form of the bus is a pipelined version of the GIO64 bus. This form of the bus is used by devices that plug into GIO64 slots, like graphics. There is a bidirectional pipeline register between the GIO64 bus that connects to main memory and the devices in the GIO64 slots. This second form of GIO64 will always be referred to as pipelined GIO64 bus. The two forms of the bus are basically the same except that the handshaking signals are different on the two different forms of the bus.

4.2 Conventions Used in this Chapter

4.2.1 Byte Addressing

The GIO64 bus uses a 32 bit byte address. This can be a big or little endian address. One of the control bits indicates the endianess of the bus. The bus itself does not do anything different for big or little endian transfers, but the devices that are on the bus need to know if the data is big or little endian data so that they can interpret the address and byte count correctly.

For devices that cannot switch between big and little endian mode the software will be responsible for fixing the data if it is necessary. Transfers that are not word aligned between 32 bit devices will not work if the two devices are running in a different endian modes. The same is true for 64 bit devices that are transferring data that is not double word aligned. In both cases the wrong data is written since the address of the bytes are different.

For a big endian 32 bit transfer, byte 0 is bits (31:24), byte 1 is bits (23:16), byte 2 is bits (15:8), and byte 3 is bits (7:0). For a big endian 64 bit transfer, byte 0 is bits (63:56), byte 1 is bits (55:48), byte 2 is bits (47:40), byte 3 is bits (39:32), byte 4 is bits (31:24), byte 5 is bits (23:16), byte 6 is bits (15:8), and byte 7 is bits (7:0). Little-endian is just the opposite so for a 32 bit transfer, byte 0 is bits (7:0), byte 1 is bits (15:8), byte 2 is bits (23:16), and byte 3 is bits (31:24). For a 64 bit little endian transfer, byte 0 is bits (7:0), byte 1 is bits (15:8), byte 2 is bits (23:16), byte 3 is bits (31:24), byte 4 is bits (39:32), byte 5 is bits (47:40), byte 6 is bits(55:48), and byte 7 is bits (63:56).

The bit numbering scheme is always little-endian, so that bit zero is always the least significant bit and bit 63 is the most significant bit.

The following tables show byte addressing for the big and little endian modes.

TABLE 3 Big Endian Words: addresses of bytes.

BITS / Word Address	31..........24	23..........16	15..........8	7..........0
8	8	9	10	11
4	4	5	6	7
0	0	1	2	3

TABLE 4 Big Endian Double Words: addresses of bytes.

BITS / Word Address	63...56	55...48	47...40	39...32	31...24	23...16	15...8	7...0
16	16	17	18	19	20	21	22	23
8	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7

TABLE 5 Little Endian Words: addresses of bytes.

BITS / Word Address	31..........24	23..........16	15..........8	7..........0
8	11	10	9	8
4	7	6	5	4
0	3	2	1	0

TABLE 6 Little Endian Double Words: addresses of bytes.

BITS / Word Address	63...56	55...48	47...40	39...32	31...24	23...16	15...8	7...0
18	23	22	21	20	19	18	17	16
8	15	14	13	12	11	10	9	8
0	7	6	5	4	3	2	1	0

4.2.2 Waveform Conventions

There are many waveform diagrams in this document so it is important to interpret them correctly. The dotted lines represent a signal that has been tristated. A box with “ADR” in it is an address, “BC” is byte count, “D0”, “D1” ..., is a data element, and “X” represents invalid data, but not tristate.

4.3 GIO64 Bus Signals

4.3.1 Non-pipelined Bus Signals

The GIO64 bus has 64 address/data signals AD<63:0>, 8 byte parity signals ADP<7:0>, a valid parity signal $\overline{\text{VLD\_PARITY}}$, and four basic control lines. These control lines include address strobe $\overline{\text{AS}}$, read READ, master delay MASDLY, and slave delay SLVDLY. In addition to the above, there are bus request $\overline{\text{BREQ}}$(n) and bus grant $\overline{\text{BGNT}}$(n) signals for each bus master. For long burst devices there is a preempt signal called $\overline{\text{BPRE}}$. Sixty-four bit bus slaves also need to get a signal called GSIZE64, which indicates the size of the current bus master.

AD<63:0>: These signals are the multiplexed slave address and data lines. During slave address cycles AD(31:0) will contain the slave address for the bus transaction. The next cycle in a transaction, the byte count cycle, uses AD(31:0) to indicate the number of bytes to transfer, the endian mode of the transfer, the device id of the device being accessed, DMA count direction (for decrementing DMA), subblock ordering (CPU cache misses), and the starting byte address. During slave address and byte count cycles parity should be checked across AD(31:0) if $\overline{\text{VLD\_PARITY}}$ is asserted. The top 32 bits of the address/data bus are never used during slave address and byte count cycles so parity should not be checked across these bits even if $\overline{\text{VLD\_PARITY}}$ is asserted. During data cycles parity should be checked across AD(31:0) for 32 bit transfers and AD(63:0) for 64 bit transfers if $\overline{\text{VLD\_PARITY}}$ is asserted. For a 32 bit device all 4 bytes of the bus should be driven with correct parity if $\overline{\text{VLD\_PARITY}}$ is asserted even for single byte transfers. Likewise, for 64 bit transfers, all 8 bytes should be driven with correct parity if $\overline{\text{VLD\_PARITY}}$ is asserted.

ADP(7:0): Even byte parity is used across the address/data bus. Parity should be checked if $\overline{\text{VLD\_PARITY}}$ is asserted with the data on the bus. ADP(0) is the parity bit for AD(7:0), ADP(1) covers AD(15:8), etc. These signals can be driven by the bus master during slave address, byte count, and write data cycles. They can be driven by the addressed bus slave during read cycles. During slave address cycles, byte count cycles and 32 bit transfers, ADP(7:4) are unused. If $\overline{\text{VLD\_PARITY}}$ is not asserted then ADP(7:0) are undefined.

$\overline{\text{VLD\_PARITY}}$: This signal indicates that valid parity is being driven on the bus. This signal will be asserted in the same cycle as there is data on the bus. Devices that are receiving addresses or data should check parity if $\overline{\text{VLD\_PARITY}}$ is asserted. This signal is pulled up on the board so if a device does not generate parity it does not have to drive the $\overline{\text{VLD\_PARITY}}$ signal. This signal is driven by the bus master during slave address, byte count, and write data cycles. It is driven by the addressed bus slave during read cycles. It is strongly encouraged that all GIO64 devices should generate and check parity. Parity should not be checked on data from bus cycles where no data was transferred, as determined by the state of MASDLY or SLVDLY. The slave or master that is driving this signal must drive it high before they tristate it in the cycle after they are finished driving the AD signals.

$\overline{\text{AS}}$: The address strobe is asserted whenever there is a slave address cycle on the bus. All devices on the bus should flop the address that is being driven on the bus to determine if they are the device that is being addressed. The byte count cycle is always the next cycle after the slave address cycle. The address strobe is only asserted during the slave address cycle. This signal should only be driven by a bus master that owns the GIO64 bus.

READ: The READ signal serves two purposes. During the slave address cycle the READ signal is used as its name implies to indicate that this is a read transaction, (the bus master will be receiving data). After the slave address cycle, the READ signal is driven low to indicate that a active bus cycle is taking place. When a transaction is preempted the READ signal is driven high to indicate to the slave that the transaction has been preempted. This signal should only be driven by a bus master that owns the GIO64 bus.

MASDLY: The master delay signal, MASDLY, is used by the bus master to throttle the data transfer rate. The state of the MASDLY signal does not affect the slave address or byte count cycles. This signal has two different meanings for read and write transactions. For a read transaction, when MASDLY is asserted it indicates that the bus master cannot receive data in the next cycle. During write transactions, when the MASDLY signal is asserted it indicates that the data currently on the bus is invalid. Only a bus master that owns the GIO64 bus should drive the MASDLY signal.

SLVDLY: The slave delay signal, SLVDLY, is used by the addressed bus slave to throttle the data transfer rate. This signal has no effect on the address or byte count cycles. Like MASDLY, this signal has different meanings for reads and writes. When SLVDLY is asserted during a bus reads, it indicates that the data on the bus is invalid. For bus writes, this signal is asserted when the slave cannot accept data in the next cycle. This signal is pulled high on the board so that during the byte count cycle it will be asserted, but it must be driven high by the slave when a transfer is finished, before the slave can tristates this signal. Only the addressed slave device should drive this signal.

$\overline{\text{BREQ}}$(n): There is a unique bus request for every bus master in the system. To request the bus the bus master asserts its bus request signal. The bus master must keeps its bus request signal asserted until it is finished using the bus. Once a bus master has asserted its bus request signal it is not allowed to take it away until it has been granted the bus, even it the device decides it does not need the bus. When a devices that owns the bus has been preempted, it must deassert its bus request signal to indicate that it is off the bus. A preempted device cannot assert its bus request signal until the preemption signal has been deasserted. A bus master must still accept requests from other bus masters even if it is requesting the bus. If it does not do this, transfers to the device that is requesting the bus will time-out and will not be completed.

$\overline{\text{BGNT}}$(n): There is a unique bus grant signal for each bus master on the GIO64 bus. The bus grant signal is used to indicate to a GIO64 master that it owns the bus. The grant signal will be asserted as long as the device asserts its bus request signal, $\overline{\text{BREQ}}$(n), indicating that it owns the GIO64 bus. When a device is preempted the bus grant signal will not be deasserted until the device deasserts its bus request signal. This signal is driven by the GIO64 arbiter.

$\overline{\text{BPRE}}$: The bus preempt signal is asserted by the GIO64 arbiter to preempt the current bus transaction. Once the preempt signal is asserted, a device must get off the bus within 4 GIO64 clock cycles, and deassert its $\overline{\text{BREQ}}$(n) signal. This signal is driven by the GIO64 arbiter.

GSIZE64: The GSIZE64 signal is driven by the GIO64 arbiter to indicate to 64 bit GIO64 slaves the size of the bus master. A 64 bit GIO64 slave must be able to accept transactions from 32 or 64 bit bus masters. It is the responsibility of 64 bit GIO64 bus masters to know if the slave they are addressing is a 32 or 64 bit slave. If it is a 32 bit slave then obviously a 32 bit transfer will take place. This should not be a problem since most bus masters will only have to know if the memory interface is 32 or 64 bits.

4.3.2 Pipelined GIO64 Bus Signals

Pipelined GIO64 devices, like graphics devices, have the same basic signals as the nonpipelined GIO64 bus, but are advanced or delayed by one cycle. There is a 64 bit address/data bus P_AD(63:0), 8 byte parity signals P_ADP(7:0), a valid parity signal $\overline{\text{P\_VLD\_PARITY}}$, and 4 basic control signals: address strobe $\overline{\text{P\_AS}}$, READ or write P_READ, a slot specific GIO64 slot device to memory/CPU handshake signal GRXDLY(n), and a memory/CPU to GIO64 slot handshake signal MEMDLY, which is flopped MASDLY, although the meaning of the signal is different. Unlike the nonpipelined GIO64 bus the MEMDLY signal is always used by the CPU or memory to throttle data to and from the pipelined GIO64 device. The GRXDLY(n) signal is always used by the pipelined GIO64 device to throttle transfers when it is a bus master or bus slave. In addition to the above basic control signals there are bus request $\overline{\text{BREQ}}$(n) and bus grant $\overline{\text{BGNT}}$(n) signals for each GIO64 slot. For long burst devices there is a preempt signal called $\overline{\text{P\_BPRE}}$. Sixty-four bit bus slaves also need to get a signal called P_GSIZE64, which indicates the size of the master.

P_AD(63:0): These signals are the multiplexed address and data lines. Their function is the same as AD(63:0) signals. The only difference between AD(63:0) and P_AD(63:0) is that there is a bidirectional registered transceiver between these signals. When a pipelined GIO64 device is driving the bus the AD(63:0) signals will be delayed by one clock from the P_AD(63:0). When a nonpipelined GIO64 device is driving the bus, P_AD(63:0) will be delayed by one clock cycle.

P_ADP(7:0): Even byte parity is used for the P_AD(63:0) signals. The $\overline{\text{P\_VLD\_PARITY}}$ signal indicates if valid parity is being driven. These signal function just like P_ADP(7:0) on the nonpipelined form of the bus.

$\overline{\text{P\_VLD\_PARITY}}$: This signal indicates that valid parity is being driven on the bus. It has the same function as the $\overline{\text{VLD\_PARITY}}$ signal on the nonpipelined GIO64 bus.

$\overline{\text{P\_AS}}$: The pipelined GIO64 address strobe is asserted whenever there is an slave address cycle on the bus. All devices on the bus should flop the address that is being driven on the bus to determine if they are the device that is being addressed. The byte count cycle is always the next cycle after the slave address cycle. The address strobe is only asserted during the slave address cycle. This signal should only be driven by a bus master that owns the GIO64 bus.

P_READ: The P_READ signal serves the same function as the READ signal on the nonpipelined GIO64 bus.

GRXDLY(n): The graphics delay signal is a unique signal for each GIO64 slot that is used by pipelined GIO64 devices to throttle the data transfer. Unlike the nonpipelined GIO64 bus, the GRXDLY(n) is always driven by the pipelined GIO64 device for reads and writes, and when the device is the bus master or slave. Since this is a slot specific pin the device can always drive it. To make transfers faster to a GIO64 device the device can deassert GRXDLY(n) all of the time except when it is not capable of accepting another transfer. This makes it possible for the memory or the CPU to issue the following sequence of GIO64 bus cycles: slave address, byte count, data, slave address, byte count, data, etc. If the GRXDLY(n) signal is only deasserted after a device has decoded the address then the bus sequence would be: slave address, byte count, stall, stall, data, slave address, byte count, stall, stall, data, etc. There is some very important differences between the meaning of nonpipelined GIO64 flow control signals (MASDLY and SLVDLY) and their pipelined GIO64 counterparts. When a pipelined GIO64 device is driving the bus, the GRXDLY(n) signal is used to indicate that valid data is on the bus. When a nonpipelined device issues a read to a pipelined GIO64 device, there is one dead cycle after the byte count, on the pipelined side, when data will not be returned and GRXDLY(n) will not be asserted. After that, either GRXDLY(n) will be asserted or valid data will be driven on the bus. For writes to a pipelined GIO64 device, the GRXDLY(n) signal will be asserted when the device cannot accept data in two clocks. Therefore the device must accept the data that is on the bus in the current cycle and in the next cycle.

MEMDLY: This is a flopped version of the MASDLY signal except that its meaning is different when a transaction with a pipelined GIO64 device is taking place. This signal is used by the CPU or memory to throttle a pipelined GIO64 device, in master or slave mode. When data is being sent to a pipelined GIO64 device this signal is deasserted in the same cycle as the data to indicate that the current data is valid. When data is being sent by the pipelined GIO64 device this signal indicates that data cannot be accepted in the next cycle.

$\overline{\text{BREQ}}$(n): There is unique a bus request for every bus master in the system. The bus requests function the same on the nonpipelined or pipelined side of the GIO64 bus.

$\overline{\text{BGNT}}$(n): There is unique a bus grant signal for each bus master on the GIO64 bus. These function the same on the pipelined or nonpipelined side of the GIO64 bus.

$\overline{\text{P\_BPRE}}$: The $\overline{\text{P\_BPRE}}$ signal is a the $\overline{\text{BPRE}}$ signal flopped once.

P_GSIZE64: The P_GSIZE64 signal is a flopped version of GSIZE64 signal.

$\overline{\text{RESET}}$: This is a synchronous RESET signal for GIO64 devices.

$\overline{\text{DMASYNC}}$(n): This signal is used by the DMA master to synchronize a DMA transfer to a device. This is useful for graphics devices where a DMA may have to be synchronized with vertical retrace.

GIO64_CLK(n): The positive GIO64 bus clock. All bus signals are clocked on the rising edge of this clock signal. This is an ECL signal.

$\overline{\text{GIO64\_CLK}}$(n): The negative GIO64 bus clock. All bus signals are clocked on the falling edge of this clock signal. This is an ECL signal.

GIO64_CLK2(n): The positive, ECL, double frequency GIO64 bus clock.

$\overline{\text{GIO64\_CLK2}}$(n): The positive, ECL, double frequency GIO64 bus clock.

$\overline{\text{INTERRUPT}}$(n)(2:0): There are three interrupt lines per slot that can be used to interrupt the CPU or the CPU can read the status of these lines.

STATUS(n): There is one status line per slot that the CPU can read.

GIO64_SPEED(1:0): The speed of the GIO64_CLK(n) is indicated/determined by these bits. These are bidirectional signals that are driven by each GIO64 device. The signals are pulled high with a resister on the system board. Each GIO64 device should pull the lines to indicate the fastest speed the card can run. The GIO64 devices should not pull the lines high. The GIO64 devices can use these signals as inputs to determine at what speed the bus is running. These bits are encoded as follows:

00	25 MHz GIO64_CLK, 50 MHz GIO64_CLK2
01	33 MHz GIO64_CLK, 66 MHz GIO64_CLK2
10	Reserved

11 40 MHz GIO64_CLK, 80 MHz GIO64_CLK2

SLOT_NUMBER(3:0): The slot number is used by a GIO64 board to determine its board address. A GIO64 board address is constructed as follows:

board_address(31:26) = "000111"

board_address(25:22) = SLOT_NUMBER

board_address(21:0) = offset from base address

Each slot is four megabytes and a GIO64 card can occupy more than one slot if the slots it uses are not used on that machine. A machine does not have to implement all 16 slots and may scatter the slots out in the address space so that each physical slot can use more than four Mbytes of address space.

GIO64 cards that do not use the slot number to determine the base address of the card must be installed in the same slot as its base address(25:22). For example if a card's address space is 0x1f000000 to 0x1f7ffffff this card must be plugged into slot 0xc and also uses the address space of slot 0xd. Nonpipelined GIO64 device may also use some of the slots.

4.4 GIO64 Bus Transfers

There are three different kinds of cycles on the GIO64 bus. The first cycle of every transaction is a slave address cycle. The second cycle is the byte count cycle. The byte count is followed by one or more data cycles.

In the slave address cycle, the bus master sends the slave address out on AD(31:0) or P_AD(31:0), and the address strobe $\overline{\text{AS}}$ or $\overline{\text{P\_AS}}$ is asserted. If this is a read operation then the READ or P_READ signal is asserted and, for a write it is deasserted. If the bus master is driving parity it is sent on ADP(3:0) or P_ADP(3:0) and $\overline{\text{VLD\_PARITY}}$ or $\overline{\text{P\_VLD\_PARITY}}$ is asserted.

In the following cycle, the byte count, the DMA direction, (up or down), device identification, subblock ordering, endian mode, and the starting byte address are sent over AD(31:0). The format for this second cycle is as follows:

TABLE 7 Format for AD Signal during Second Cycle

Bit	Field
15:0	Byte Count
19:15	Reserved
25:20	Device Identification
26	DMA Count Direction Down
27	CPU Subblock Ordering
28	Little Endian Transfer
29	Starting Byte(2)
31:30	Starting Byte(1:0)
63:32	Unused

The byte count is the number of bytes to transfer. The byte count and slave address determine alignment of the beginning and end of the transfer. This allows any byte alignment for the beginning and end of the transfer. A transaction must not cross a processor virtual page boundary or a DRAM page boundary. A page is 4K bytes for the R3000 and 4K bytes to

16M bytes for the R4000. The size of a DRAM page depends on the memory system of the machine and the smallest size DRAMs that it supports.

The DMA count direction down bit is asserted if the DMA slave should use decrementing addresses. This means the address that was sent to the slave during the slave address cycle should be decremented instead of incremented at the end of each data cycle. The direction bit is also used to determine which bytes get read or written in an unaligned transfer. If the DMA direction is down then bytes less than and equal to the slave address are being transferred, otherwise bytes greater than and equal to the slave address are being transferred.

The starting byte address is the three least significant bits of the master byte address. This address is in the same endian mode as the slave address, which is indicated by the endian bit in the byte count cycle. For transfers that are aligned starting byte address bits match the slave address bits (2:0). If the transfer is unaligned then it is up to the device transferring data to or from memory to align the data correctly using these bits. Thirty-two bit devices only use starting byte bits zero and one. The starting byte address bits only have to be driven by the main memory system DMA master, but they must be received by all DMA slaves that support unaligned transfers. All DMA masters, except for the main memory DMA master, do not have to drive the starting byte address since the data on the bus must be aligned to main memory.

For example, an unaligned little endian DMA read from a 32 bit devices to main memory, when the slave address is 0x2, the byte count is 0x3, the DMA direction is up, and the main memory address, master address, is 0x1 so the starting byte bits are 0x1, then the slave would have to read from byte addresses 0x2-0x4 and drive that data on the bus with the byte from slave address 0x2 on AD(15:8) or P_AD(15:8) and the rest of the bytes are packed in the same word. This example is shown below:

Not all devices have to support unaligned transfers. There is a software restriction on devices that do not support unaligned transfers however since data that is to be read from memory or written into memory must be aligned to memory on the GIO64 bus.

The little endian bit in the byte count is used to indicate if the slave address and starting byte address should be interpreted as a big or little endian address. This signal is asserted, high, for a little endian transfer.

The subblock ordering bit will be set if the transfer is for servicing a CPU cache miss. If a device does not support this mode then data from that device cannot be directly cached. This is really for caching the boot ROM. The address that is sent for a subblock order request will be the address of the first piece of data that is to be returned not the starting address of the block of data. The byte count will indicate the size of the block of data. The address sequence can be generated by XORing a count of the bytes transferred with the starting slave address. For example a read from a 32 bit device with the slave address 0x14 and a byte count of 64, the data that would be transferred is as follows:

Cycle	Start Address	Count	Address of Word Trans-ferred
1	0x14	0x0	0x14
2	0x14	0x4	0x10
3	0x14	0x8	0x1c
4	0x14	0xc	0x18
5	0x14	0x10	0x04
6	0x14	0x14	0x00
7	0x14	0x18	0x0c
8	0x14	0x1c	0x08
9	0x14	0x20	0x34
10	0x14	0x24	0x30
11	0x14	0x28	0x3c

Cycle	Start Address	Count	Address of Word Transferred
12	0x14	0x2c	0x38
13	0x14	0x30	0x24
14	0x14	0x34	0x20
15	0x14	0x38	0x2c
16	0x14	0x3c	0x28

For a read from a 64 bit device with the slave address of 0x14 and a byte count of 64, the data that would be transferred is as follows:

Cycle	Start Address	Count	Address of Double Word Transferred
1	0x14	0x0	0x10
2	0x14	0x8	0x18
3	0x14	0x10	0x0
4	0x14	0x18	0x8
5	0x14	0x20	0x30
6	0x14	0x28	0x38
7	0x14	0x30	0x20
8	0x14	0x38	0x28

The last field in the byte count cycle is the device identification field. This can be used by multiprocessor systems to maintain cache coherency. Each device on the bus will have a unique device identification so the cache coherency hardware will know the destination or source of the data on the GIO64 bus. Each new GIO64 device will need a new device id. The current list of device identifiers is as follows:

ID	Device
0	CPU R/W
15-1	I/O Devices (HPC)
16	Graphics Head 0
17	Graphics Head 1
18	Token Ring Card
19	FDDI Card
20	Professional Audio
31-21	Reserved
47-32	EISA Bus 0

ID	Device
63-48	EISA Bus 1

Even though the GIO64 bus is a 64 bit bus, the slave address and byte count cycles will always take two cycles instead of one 64 bit cycle. This part of the transfer is the same for 32 and 64 bit devices. The address strobe signal is not active in the byte count cycle. The meaning of the READ and P_READ changes to indicate an active bus cycle after the slave address cycle. During the slave address cycle the READ or P_READ signal indicates the direction of the data transfer. At all other times when the READ and P_READ signals are deasserted, low, they indicate that the bus is busy. This signal is used to indicate a bus preemption or end of the data transfer when it is driven high.

During the slave address and byte count cycles and 32 bit data transfer cycles, the high bits of the address/data bus, AD(63:32) and P_AD(63:32) are undefined, and so parity should not be checked on these bits. Parity on AD(31:0) and P_AD(31:0) should be generated and checked during the slave address and byte count cycles. The parity for AD(7:0) is ADP(0), AD(15:8) is ADP(1) etc., likewise for the pipelined address/data bus and parity bits.

The slave address and byte count cycles are followed by some number of data cycles. The number of data cycles depends on the byte count, size of the transfer (32 or 64 bits), preemption, and flow control signals. The slave address and byte count cycles are the same for the nonpipelined and pipelined GIO64 bus. Data cycles are different for the nonpipelined and pipelined GIO64 bus. To avoid confusion the following sections will describe transfers between devices on the nonpipelined GIO64 bus, and then transfers between the CPU or main memory and a pipelined GIO64 device. Note that the only transfers that are supported for pipelined GIO64 devices is between the device and main memory or the CPU. Pipelined GIO64 devices cannot communicate via the GIO64 bus due to the nature of the pipelined GIO64 flow control signals.

4.4.1 Nonpipelined GIO64 Bus Transfers

There are two signals, MASDLY and SLVDLY that are used to throttle the data transfer rate on the nonpipelined GIO64 bus. The bus master drives MASDLY and the slave drives SLVDLY. These signals are used to indicate that valid data is on the bus by the device that is driving data. These signals are also used to throttle the data by the receiving device. Pipelined GIO64 devices, like graphics, use a different set of lines called GRXDLY(n) and MEMDLY.

4.4.1.1 Non-pipelined GIO64 Bus Writes

GIO64 bus writes start with a slave address and byte count cycle. The READ signal will be deasserted in the slave address cycle to indicate that this is a write cycle. After the slave address cycle, the READ signal is used to indicate that a bus cycle is in progress and will remain low for the rest of the transfer.

After the third cycle, data can be transferred. During a write, the bus master will drive data out onto the bus and drive MASDLY low to indicate that valid data is on the bus. The master looks at the SLVDLY signal that was sent the cycle before to determine if the slave can accept the data. If the SLVDLY signal was not low in the previous cycle, the master must redrive the current data until SLVDLY is low in the previous cycle. This is a change from the GIO32 bus in that the SLVDLY signal is flopped and used in the next cycles instead of being used in the cycle it is on the bus. This change is necessary so that the SLVDLY signal can be directly registered before any gating takes place. This provides one whole cycle for the signal propagation between chips and one whole cycle for on-chip gating. The GIO32 bus scheme works at 33 MHz, but at higher speeds it becomes very difficult to get the timing to work. The bus master can continue to transfer new data every cycle that SLVDLY from the previous cycle is low. If SLVDLY was not low then the current data must be driven until SLVDLY is deasserted in the previous cycle. The master can throttle the transfer by driving MASDLY high during a cycle that it does not have new data to transfer. Note that this remains the same as GIO32 and is not sent one cycle early like SLVDLY.

Since SLVDLY is being sent for the next cycle, it will take an extra cycle for all complete transfers, (not one cycle per word). A one word write will take at least four cycles. This is one more cycle then it took with the GIO32 bus.

The position of bytes on the bus varies with the size of the transfer and if the machine is running in big or little endian mode. Figure 4-1 shows the position of bytes for different size transfers (32 or 64 bits wide) and for big and little endian mode transfers.

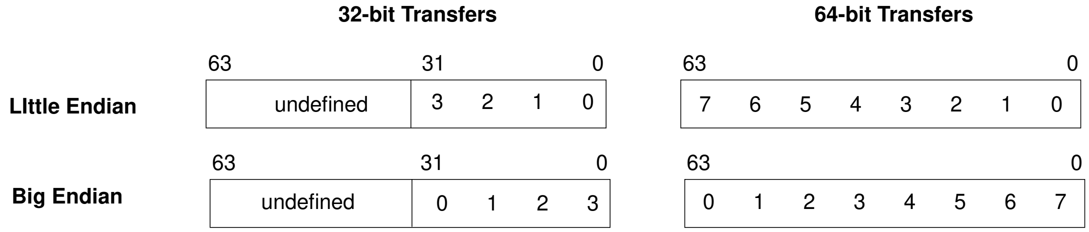

Figure 4-1. Byte Order for Big and Little Endian Transfers

The bus master continues to transfer data until the byte count is satisfied. The bus slave also keeps track of the number of bytes that have been transferred so that the last write will be handled correctly if it is a partial word or partial double word transfer. At the end of the transfer the master drives READ high. Two cycles after the slave receives the last piece of data it drives SLVDLY high and in the following cycle tristates the SLVDLY signal. The bus master will tristate the AD, READ, $\overline{\text{AS}}$, and MASDLY signals two cycles after the last piece of data is transferred if it does not have another transfer to execute. The master does not have to drive the MASDLY and READ signals high before tristating them. If it does have another transfer it can drive the address in the cycle immediately following the last piece of data.

If a transfer is preempted, the byte count will not be zero when the READ signal is driven high by the bus master. The slave needs to monitor the READ signal and not just the remaining byte count, so that it can tell if a transfer has been preempted. The bus master must keep all of the information that is necessary to restart the transfer where it left off. This includes the slave data address. Below are some examples of nonpipelined GIO64 writes.

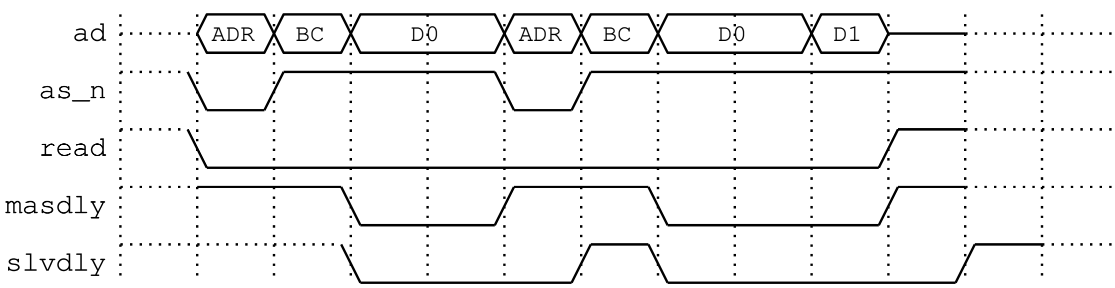

Figure 4-2. Back-to-Back Simple GIO64 Writes

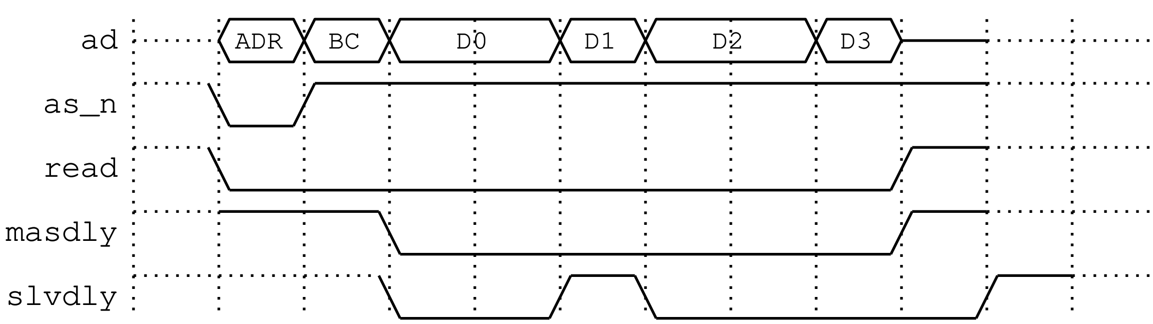

Figure 4-3. GIO64 Write, Slave Stall

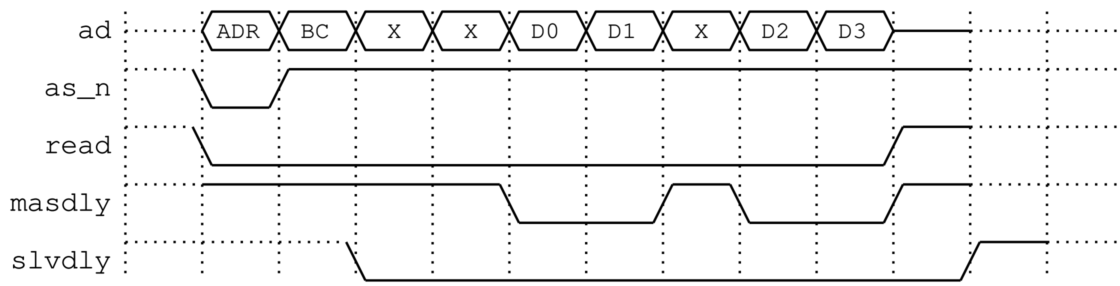

Figure 4-4. GIO64 Write, Master Stalls

4.4.1.2 Non-pipelined GIO64 Bus Reads

GIO64 bus reads are a lot like GIO64 bus writes except that the slave is sending the data over the bus instead of the master. Notice that the READ signal is used to indicate that a bus cycle is in progress after the slave address cycle, by being deasserted for the rest of the transfer. Since the slave is sending the data, the SLVDLY signal is used to indicate that there is valid data on the bus and the MASDLY signal is used to indicate to the slave that the master can accept data in the next cycle. Note that this is different from the GIO32 bus in that MASDLY is sent one cycle earlier. The slave must tristate the AD bus signals in the cycle after the last piece of data is transferred. The slave must drive the SLVDLY signal high in the cycle after the last piece of data is transferred and then tristate in the following cycle. The master will tristate the $\overline{AS}$, READ, and MASDLY signals three cycles after the last piece of data has been transferred if it does not have another transfer to execute. If it does have another transfer to do it can drive the address two cycles after the last piece of data has been transferred. Some examples of GIO64 reads are shown below.

It is important that the slave does not wait for MASDLY to be deasserted before it drives the read data and deasserts SLVDLY, or that once the slave has deasserted SLVDLY and driven the read data, that it does not stop driving it, even if it asserted SLVDLY until the cycle after the master deasserts MASDLY. Figure 4-7 shows this condition.

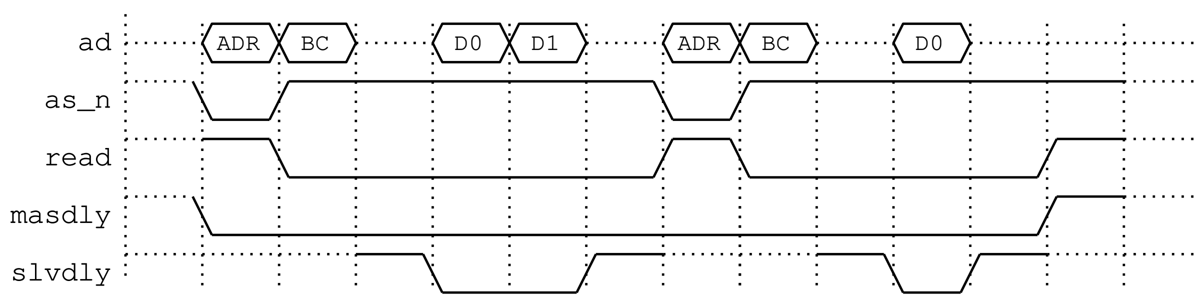

Figure 4-5. Simple GIO64 Read

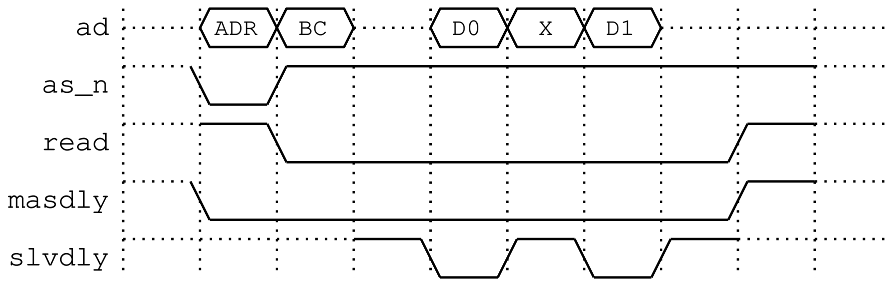

Figure 4-6. GIO64 Read, Slave Delay

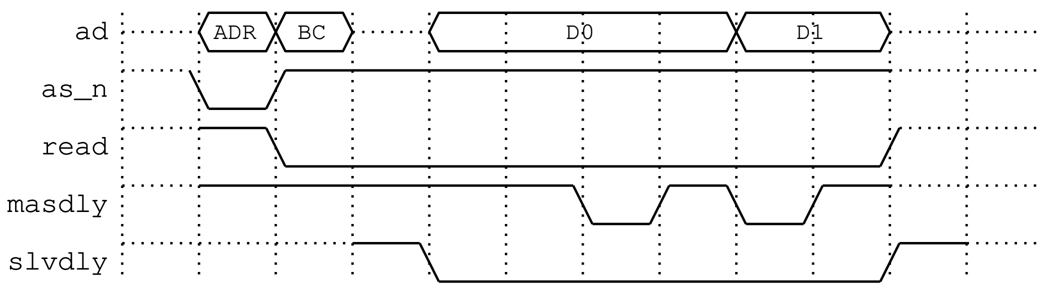

Figure 4-7. GIO64 Read, Master Delay

4.4.1.3 Non-pipelined GIO64 Preemption

When a GIO64 device gets preempted, the master drives the READ signal high to indicate an end of the transfer. For GIO64 writes, the bus master also stops driving data and MASDLY in the same cycle as it stops driving READ. GIO64 reads are preempted in basically the same way, except that the slave may continue driving data in the cycle that the master drives the READ and MASDLY signals high. The data that the slave sends in the cycle that READ is driven high is not accepted. The bus master must tristate all of the signals it is driving in the cycle after READ is driven high. The slave must drive SLVDLY high in the cycle after READ is driven high and tristate it in the following cycle. Different preemption cases are shown below.

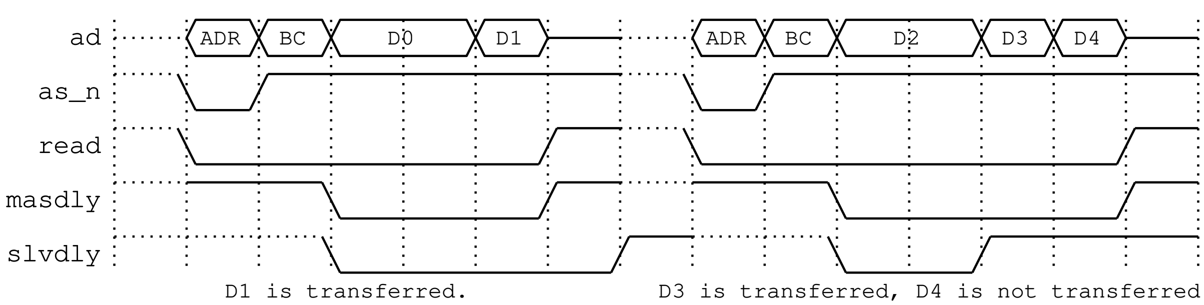

Figure 4-8. Preempted GIO64 Write, Slave Stall

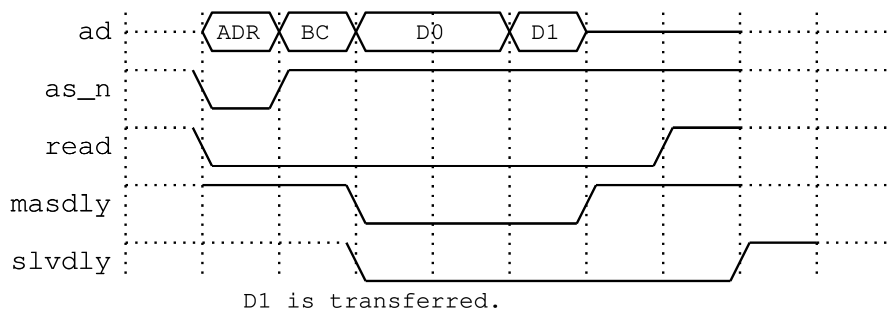

Figure 4-9. Preempted GIO64 Write, Master Stall

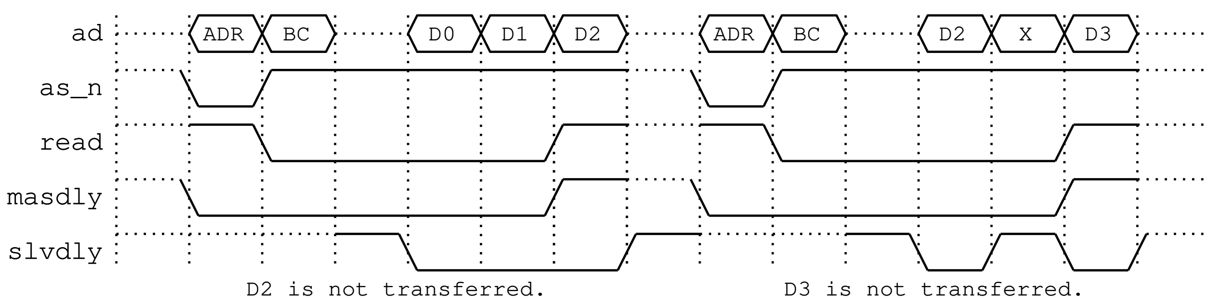

Figure 4-10. Preempted GIO64 Reads, Slave Stall

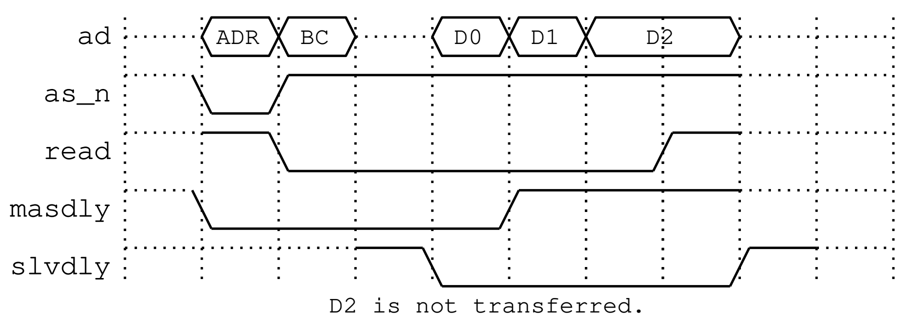

Figure 4-11. Preempted GIO64 Reads, Mater Stall

4.4.2 Pipelined GIO64 Transfers

Pipelined transfers are structured just like the nonpipelined transfers in that there is 3 different kinds of cycles: slave address, byte count, and data. The slave address and byte count cycles are identical to the nonpipelined slave address and byte count cycles. The data cycles use different flow control signals, but otherwise are the same as the nonpipelined data cycles. There are two flow control signals on the pipelined bus, MEMDLY and a slot specific GRXDLY(n) signal. Unlike the nonpipelined bus, GRXDLY(n) is always driven by the pipelined GIO64 device and MEMDLY is always driven by the memory system or the CPU.

The flow control signals will change meaning for reads and writes depending on whether memory or the pipelined GIO64 device is the bus master. It may appear that there are 4 different combinations and 4 meanings to the control signals, but really there are only 2 different meanings to the control signals depending on what device is driving data, the memory or pipelined GIO64 device.

The MASDLY signal gets flopped and becomes MEMDLY on the pipelined side of the bus. The memory master always uses MASDLY as its flow control signal even when it is a slave to a pipelined device.

4.4.2.1 Pipelined GIO64 Writes

Pipelined GIO64 writes begin just like the nonpipelined GIO64 writes with a slave address cycle and a byte count cycle. The P_READ signal is deasserted during the slave address cycle to indicate a write and then remains deasserted for the remainder of the transfer to indicate a bus transfer is in progress.

When the CPU or memory is the bus master, the MEMDLY signal is used to indicate that there is valid data on the bus. It will be deasserted in the same cycle that the data is valid. The GRXDLY(n) signal is used to indicate that the GIO64 device cannot accept data two cycles from now. The GRXDLY(n) signals becomes F_GRXDLY(n) on the nonpipelined side of the GIO64 bus. On the nonpipelined side of the GIO64 bus, the F_GRXDLY(n) signal indicates that the data currently on the bus cannot be accepted in this cycle and should be redriven in the next cycle.

When the pipelined GIO64 device is the bus master, the GRXDLY(n) signal is deasserted in cycles that the device puts valid data on the bus. The MEMDLY signal is used to indicate that the memory system cannot accept data in the next cycle. The memory system must accept data for three cycles from the time it asserts MASDLY. The memory system may deassert MEMDLY when a pipelined GIO64 device has the bus so that the pipelined device does not have to wait for the memory system to decode the address before deasserting MEMDLY.

Unlike the nonpipelined side of the GIO64 bus the pipelined GIO64 device always drives the GRXDLY(n) signal and never tristates it. The master will tristate $\overline{\text{AS}}$ or $\overline{\text{P\_AS}}$, READ or P_READ, and AD or P_AD signals in the cycle after the last piece of data is transferred.

Some examples of pipelined GIO64 writes are show below.

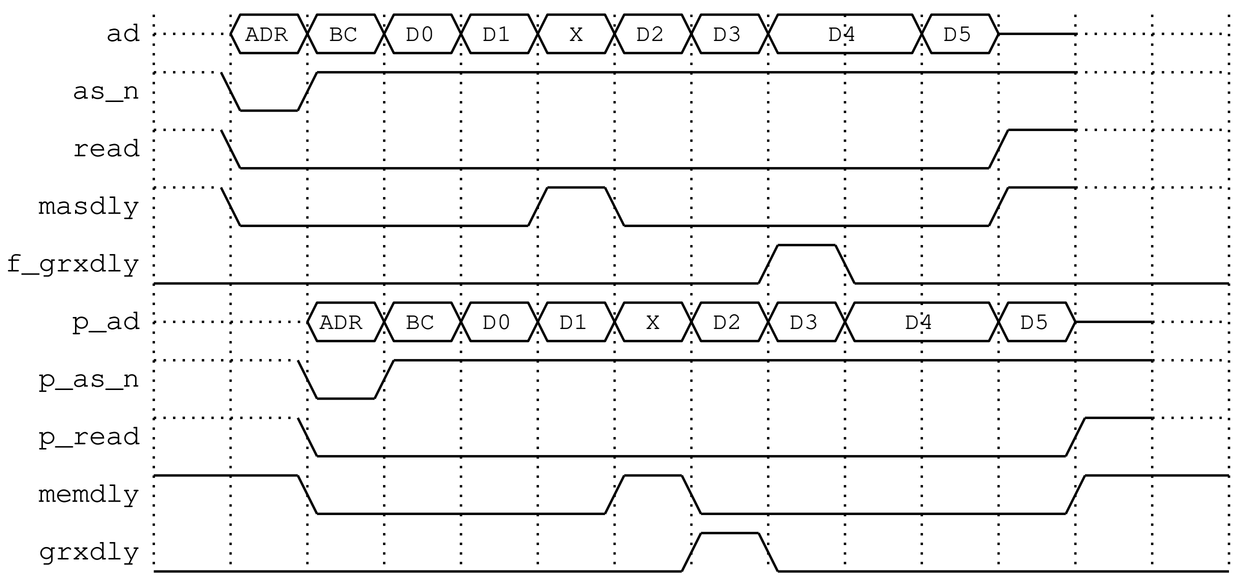

Figure 4-12. Pipelined GIO64 Writes, Memory Master

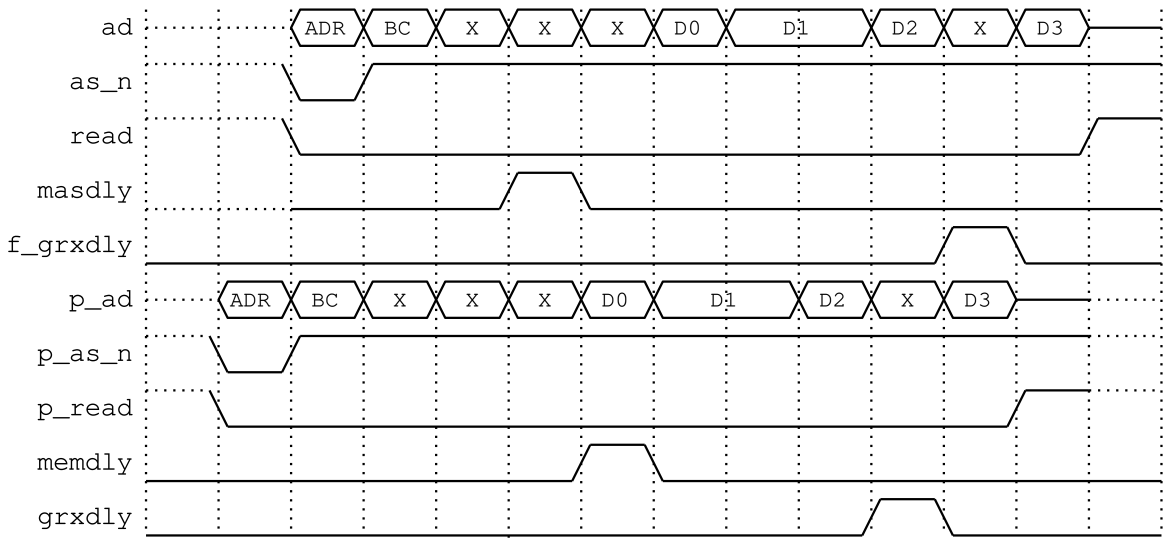

Figure 4-13. Pipelined GIO64 Writes, Pipelined Device Master

4.4.2.2 Pipelined GIO64 Reads

Pipelined GIO64 reads are a lot like the nonpipelined GIO64 reads, as they both start with a slave address cycle followed by a byte count cycle and then a number of data cycles. There are two cases of reads that need to be examined. The first is when the CPU or memory is the bus master. The GRXDLY(n) signal is used to indicate that there is valid data on the bus. There is one dead cycle after the byte count on the pipelined side of the bus, (three dead cycles on the nonpipelined side of the bus), when data will not be transferred. The state of GRXDLY(n) during these cycles does not matter. Therefore, if the pipelined device can return data in the fourth cycle after the byte count GRXDLY(n) does not have to be asserted before the data is returned. This is an artifact of pipelined GIO64 write, since the pipelined GIO64 device can hold GRXDLY(n) low when there is not a transfer in progress. This is necessary to support three cycles word/double word writes to a pipelined GIO64 device from the CPU or memory. The MEMDLY signal is used to indicate that the master cannot accept data from the GIO64 pipelined device in the next cycle. The GRXDLY(n) and MEMDLY signals function the same, except for the single dead cycle after the byte count, in this case as in a write when a GIO64 devices is the bus master.

When a pipelined GIO64 device is the bus master, the GRXDLY(n) signal indicates that the device cannot accept data in two cycles. The MEMDLY signal is used to indicate that the data on the bus is invalid during this cycle. There are three dead cycles, on the pipelined side of the bus, after the byte count cycle when no data will be transferred even though MEMDLY may be deasserted during that time. The GRXDLY(n) and MEMDLY signals function the same, except for the three dead cycles after the byte count, in this case as in a write when the CPU or memory is the bus master. The master must tristate the AD or P_AD signals after the byte count cycle. The slave must tristate the AD or P_AD signals in the cycle after the last piece of data has been transferred. Some examples of pipelined GIO64 reads are show below.

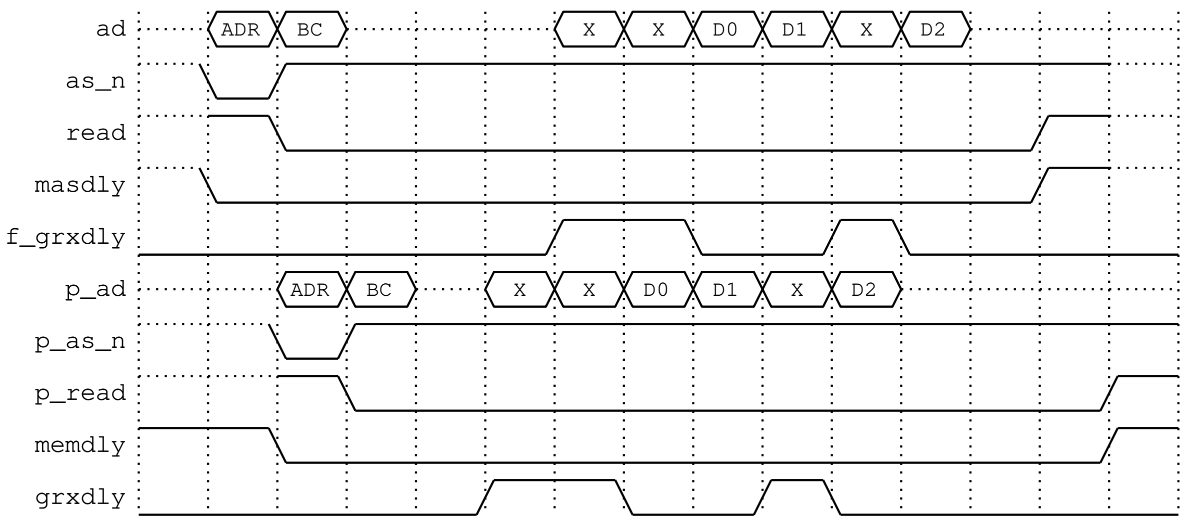

Figure 4-14. Pipelined GIO64 Reads, Memory Master

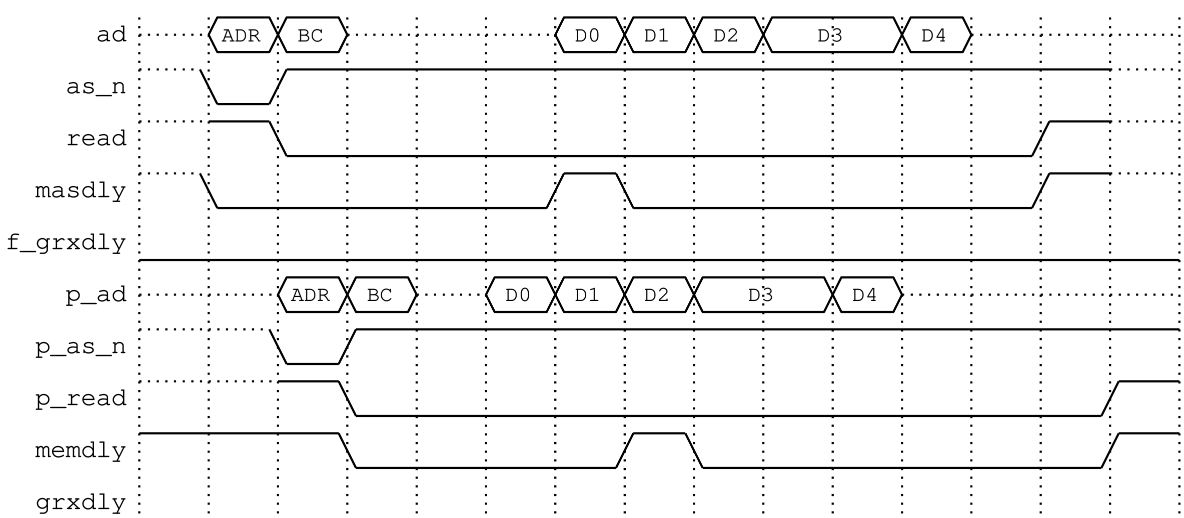

Figure 4-15. Pipelined GIO64 Reads, Memory Master

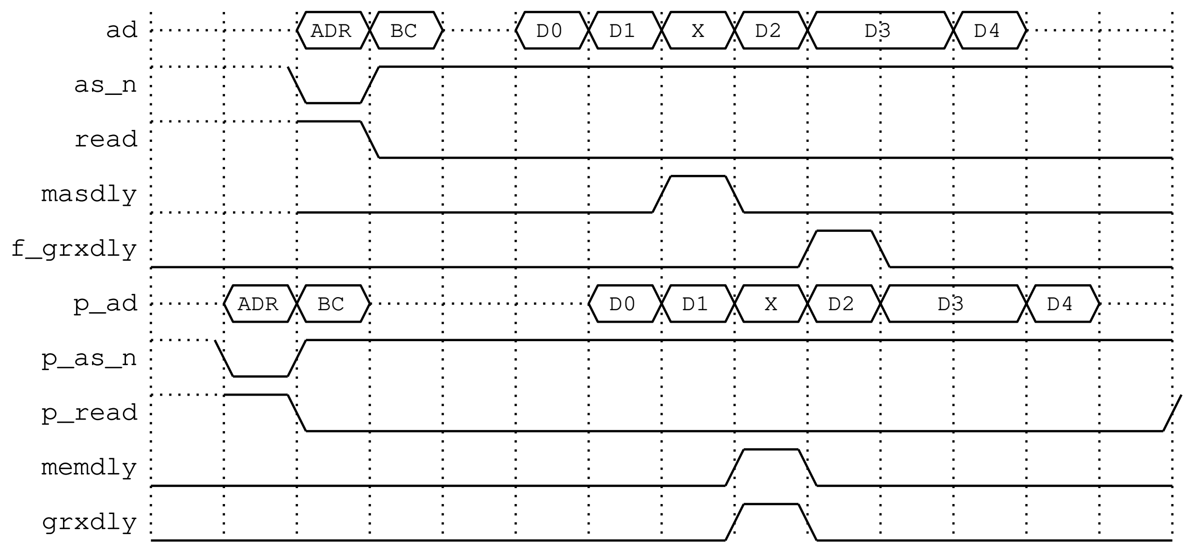

Figure 4-16. Pipelined GIO64 Reads, Pipelined Device Master

4.4.2.3 Pipelined GIO64 Preemption

Pipelined GIO64 devices can be preempted just like nonpipelined GIO64 devices. The preemption is indicated in the same way by the bus master asserting the READ or P_READ signal. The MEMDLY and GRXDLY(n) signals may or may not be asserted during a preemption. The major difference between pipelined and nonpipelined preemption is the bus cycle in which transfers complete and which cycles have to be retransferred when the device gets the bus again.

There are four different preemption cases that must be handled: reads to memory from a GIO64 pipelined device, writes to memory from a GIO64 pipelined device, reads to a pipelined device from memory or the CPU, and writes to a pipelined device from memory or the CPU.

When a write from memory to a pipelined device is preempted, the P_READ and MEMDLY signals will be asserted in the same cycle and no new data is sent to the pipelined device in that cycle. An example of a preempted pipelined GIO64 write, with memory as the bus master is shown below.

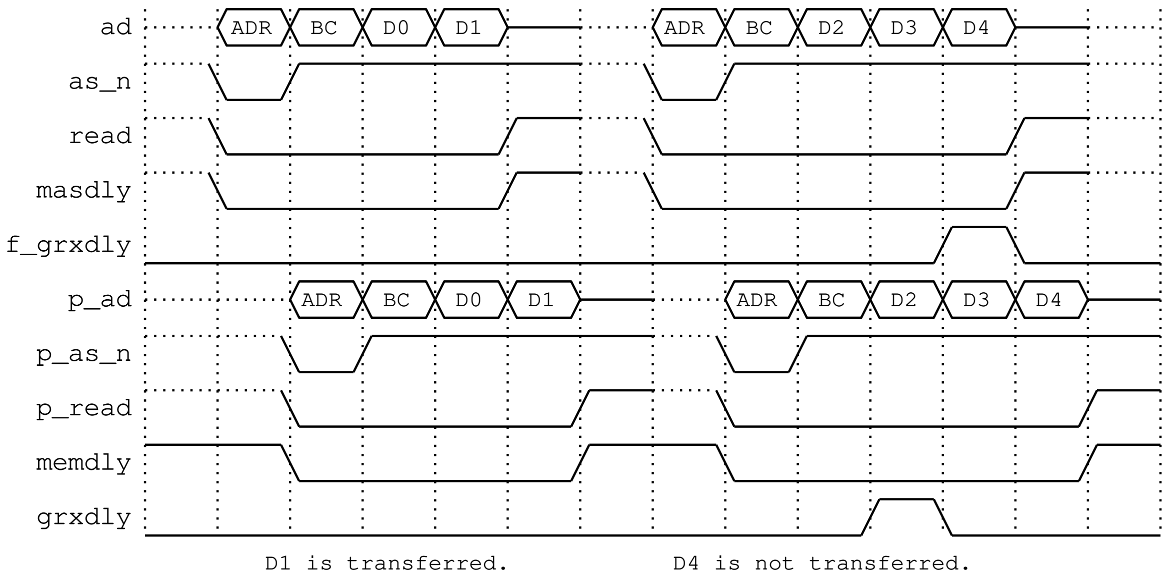

Figure 4-17. Preempted, Pipelined GIO64 Write, Memory Master

When a read from a pipelined GIO64 device to memory or CPU is preempted the P_READ and MEMDLY signals are asserted in the same cycle, and data is transferred in that cycle. An example is shown below.

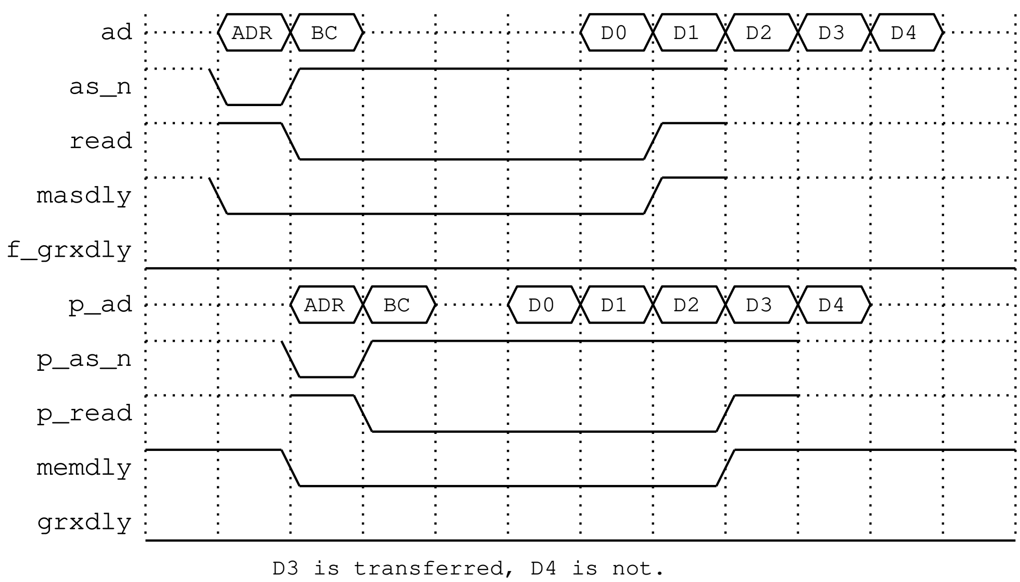

Figure 4-18. Preempted, Pipelined GIO64 Read, Memory Master

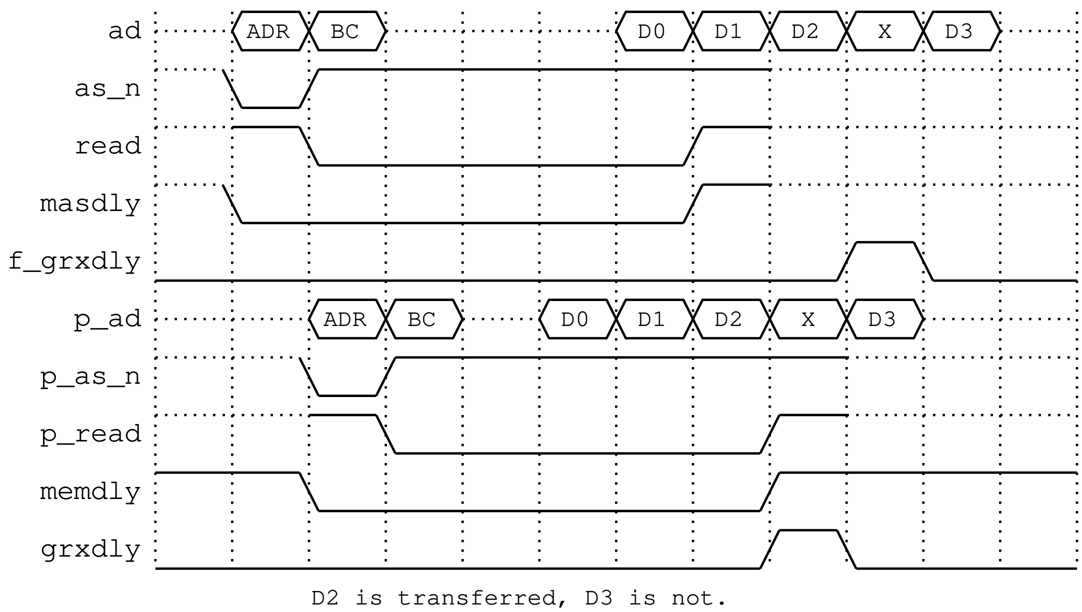

Figure 4-19. Preempted, Pipelined GIO64 Read, Memory Master

When a pipelined device is the bus master and a write to memory gets preempted P_READ will be asserted and no new data will be transferred. An example of a preempted write is shown below.

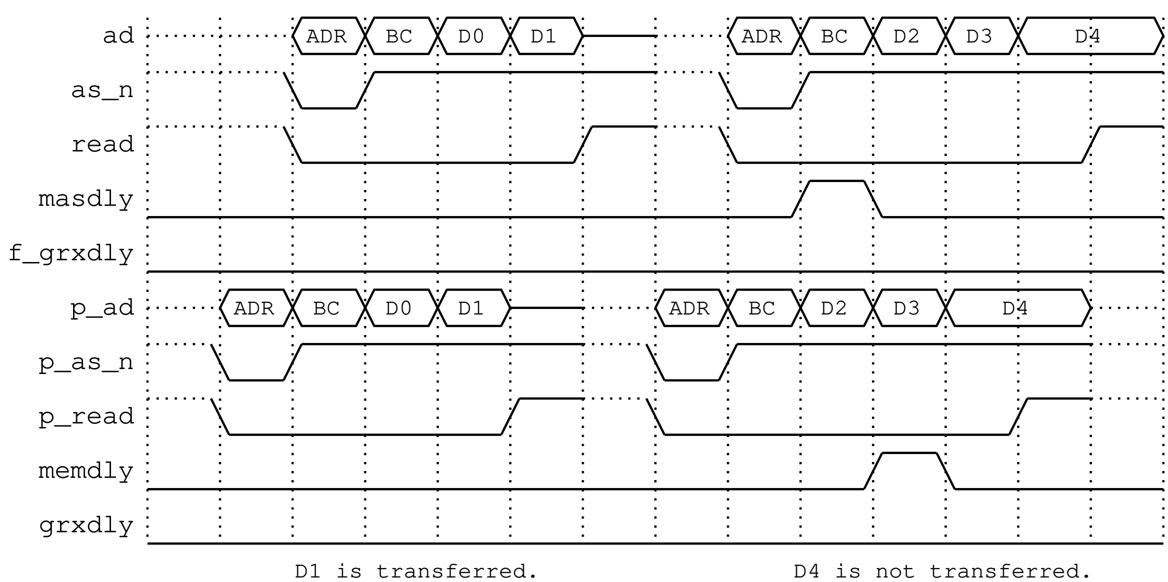

Figure 4-20. Preempted GIO64 Write, Pipelined Device Master

The last preemption case is when a pipelined GIO64 device is the bus master doing a read from memory and gets preempted. The three cycles after the P_READ signal is deasserted will not be preempted, after the third cycle data will not be transferred. An example is given below.

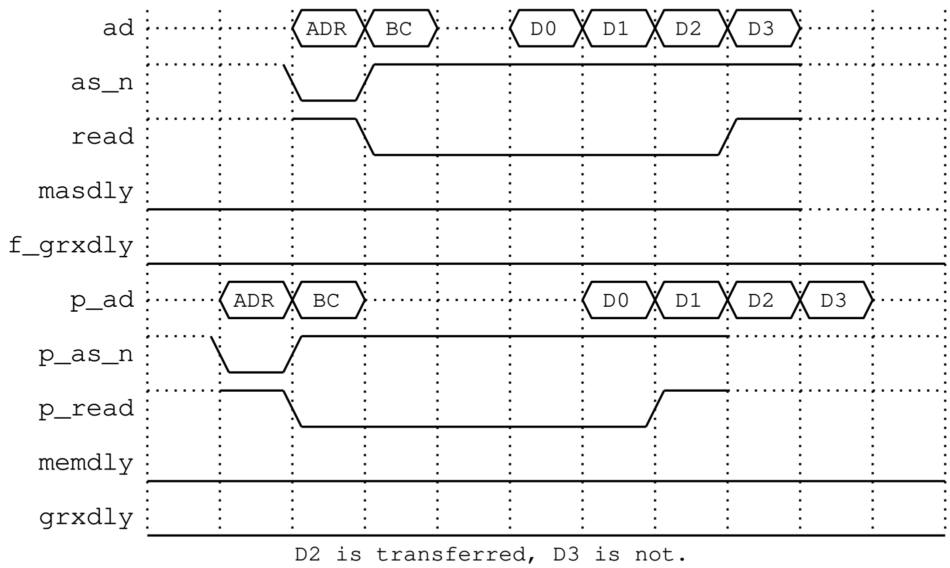

Figure 4-21. Preempted GIO64 Read, Pipelined Device Master

4.4.3 GIO64 Transfer Size

It is the responsibility of GIO64 bus masters to determine if the slave they are communicating with is a 32 bit or 64 bit device. This information is not provided on the bus. Thirty-two bit slaves require that the master transfer data in 32 bit mode. Sixty-four bit slaves should transfer 64 bits of data with 64 bit masters and 32 bits of data with 32 bit masters. The GIO64 arbiter drives the signals GSIZE64 and P_GSIZE64 that indicates the data width of the current bus master. Sixty-four bit bus masters and slaves must receive this signal to determine the width of the data being transferred.

Thirty-two bit bus masters and slaves do not need to receive the GSIZE64 or P_GSIZE64 signal since they can only send or receive 32 bits of data. Sixty-four bit bus masters do not need the GSIZE64 or P_GSIZE64 signal when they are the bus master, but these devices are also bus slaves when the CPU reads or writes to them so they need this signal if they need to work with different size memory/CPU masters. The 32 bit slaves can only receive 32 bit data so this signal does not provide any information for them either. It is the responsibility of the 64 bit master to know the size of slaves it is communicating with.

4.4.4 GIO64 Bus Time-outs

GIO64 bus cycles will time-out if SLVDLY, MEMDLY, or GRXDLY(n) are not asserted in reply to an address strobe within 25 microseconds after the byte count cycle. This time-out will prevent accesses to non-existent locations on the GIO64 bus from hanging the bus. When a time-out occurs, the GIO64 arbiter will respond with SLVDLY, MEMDLY, or GRXDLY(n) until the transfer is complete and generate a bus error interrupt to the CPU. The device that was transferring data will not be notified that a time-out has occurred. The only time software should use this time-out mechanism is to check to see if a device exists in one of the GIO64 slots. The guaranteed bus acquisition time for real time devices may be violated when the GIO64 bus times out.

4.4.5 GIO64 Bus Tristate Turnover Cycles

There needs to be one dead cycle on the bus when the device driving the AD and P_AD bus signals is changed. Therefore during a read cycle there needs to be a dead cycle between the byte count cycle and the first data cycle. This is necessary to prevent tristate overlap of the two bus drivers.

When a transfer to a nonpipelined bus slave is complete or preempted the bus slave must drive SLVDLY high before tristating SLVDLY. Devices that drive valid parity must drive $\overline{\text{VLD\_PARITY}}$ or $\overline{\text{P\_VLD\_PARITY}}$ high the cycle after the transfer is complete or preempted before tristating this signal.

4.4.6 GIO64 Bus Request And Preemption

An example of a nonpipelined GIO64 long burst device requesting the bus and then getting preempted is shown below. The dashed line represents a tristated signal.

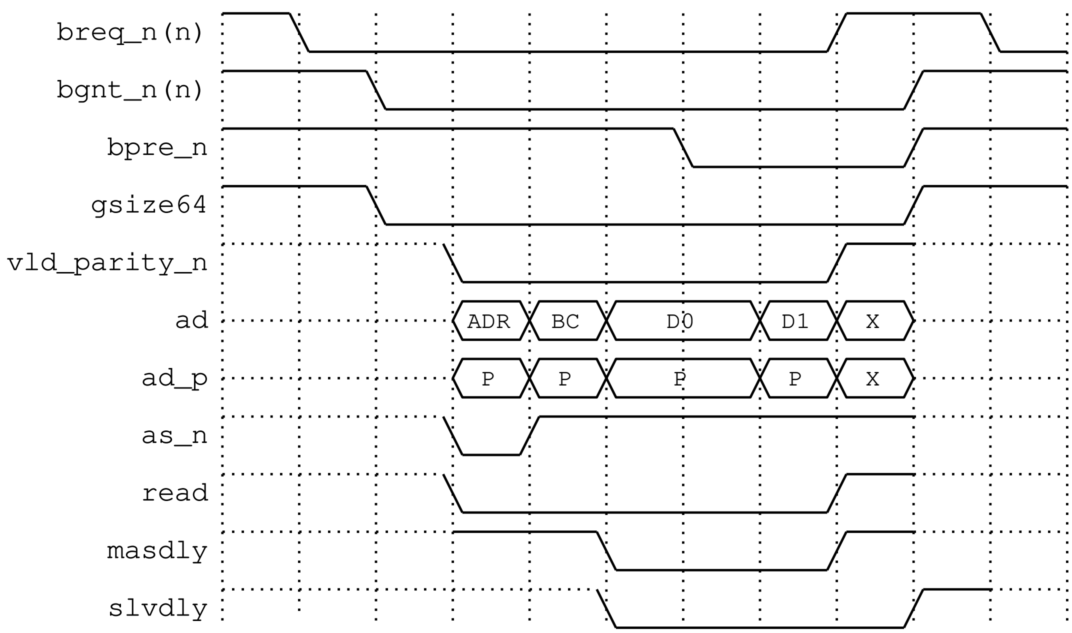

Figure 4-22. Preempted GIO64 32-bit Write

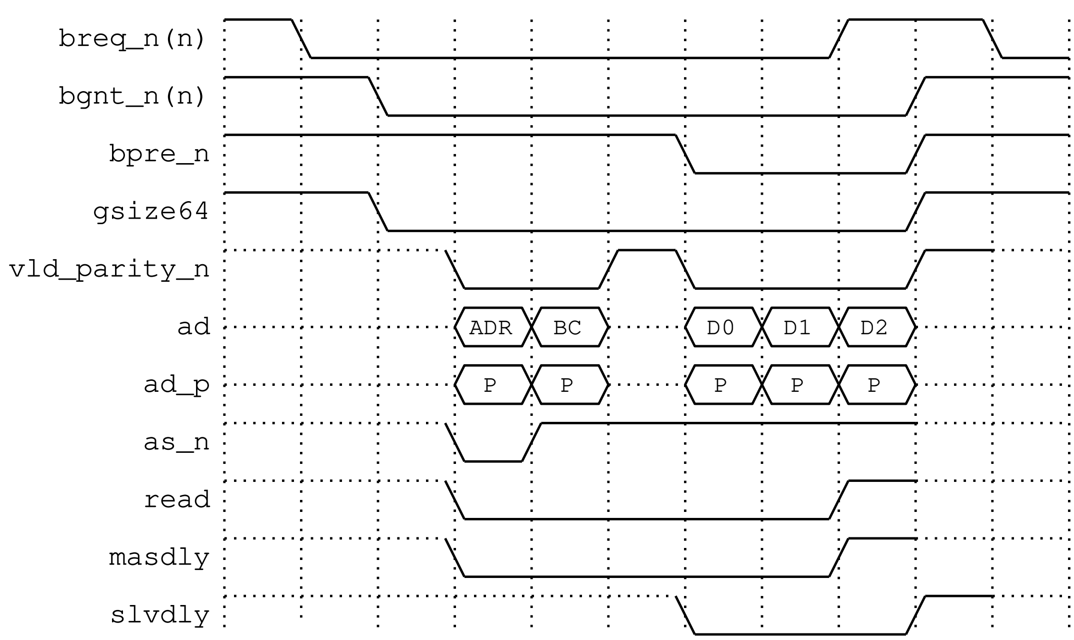

Figure 4-23. Preempted GIO64 32-bit Read

4.5 GIO64 Bus Arbitration

The GIO64 bus arbitration scheme has a number of requirements to meet. The requirements are:

- The CPU must run at a minimum guaranteed rate in the most highly loaded system to allow for an acceptable interrupt response time.
- Burst DMAs such as graphics DMA must be allowed to use the bus for long stretches of time for better bus utilization.
- Real-time devices such as audio must be guaranteed access to the bus within a predefined maximum delay.
- The EISA bus needs frequent access to memory, every 4 microseconds.

4.5.1 Three Kinds of Bus Requests

With the above requirement in mind there are three basic types of devices: real-time devices, short burst devices, and long burst devices. Audio is a good example of a real-time device. The EISA bus is a short burst device. The graphics system is a long burst devices. Long burst devices are preemptable whereas the real-time and short burst devices are not. Real-time devices, however, must not use the bus for more then 5 microseconds per acquisition and not request the bus more often then every 20 microseconds. A short burst device has the highest priority, but can only keep the bus for 1 microsecond and will only get the bus every 4 microseconds if other devices want the bus. If a real-time device requests the bus when it is being used by a long burst device, the long burst device will be preempted immediately. The bus will be given back to the long burst device that was preempted once the real-time device is finished. A long burst device can also be preempted by a short burst device if it has been 4 microseconds since the last time the short burst device had the bus. If no other devices want the bus when a short burst device requests the bus then it will be given the bus immediately. A long burst device will keep the bus until it is finished with its transfer or its time period is up and gets preempted. If a long burst device gets preempted then the time period counter will stop counting while the real time device owns the bus and will restart once the long burst device has been given back the bus. When no other devices want the bus, control of the bus is given to the CPU. The exact implementation of the arbiter depends on the machine and the number of real time and long burst device that are being supported.

4.5.2 Arbitration Handshake

Each bus master on the GIO64 bus has bus request, $\overline{\text{BREQ}}$(n), and bus grant, $\overline{\text{BGNT}}$(n), signals that go to the GIO64 arbiter. When a GIO64 device wants to request the bus, it asserts its bus request line. When the arbiter grants the bus to the requester, it asserts the bus grant signal to that device. Once the transfer is complete the bus master deasserts the bus request and the arbiter will deassert the bus grant. It is important that the bus master does not deassert the bus request before the transfer is complete since the arbiter does not monitor any of the bus control signals except the bus requests and grants. If the transfer is not complete and a bus master deasserts $\overline{\text{BREQ}}$(n), a bus collision could occur. Once a device has requested the bus it must hold its request signal active until it has been granted the bus even if the device no longer needs the bus after it has requested it.

4.5.3 GIO64 Preemption

A long burst GIO64 device can be preempted by the $\overline{\text{BPRE}}$ or $\overline{\text{P\_BPRE}}$ signals from the GIO64 arbiter. The bus master must terminate the transfer within 4 clocks of when the preempt signal is asserted. The bus master is responsible for keeping track of the address and remaining byte count so that the transfer can be resumed later. This includes the slave address.

A bus master is allowed to preempt a transfer before the byte count has been satisfied and never restart the transfer. This is useful for devices that may not know the byte count when they start the transfer. The bus master can drive a maximum byte count during the byte count cycle and then preempt the transfer when they have transferred as much as the bus master wants to as long as it is less than or equal to the byte count. Note that this requires the last transfer to be aligned to the bus since the byte count cannot indicate how many bytes to transfer on the last cycle.

4.6 GIO Compatibility Issues

The goal of the GIO64 bus is to be as close to the original GIO bus compatible as possible, but still allow for a higher bandwidth bus in the future. The one change that was made that is not GIO compatible is the MASDLY and SLVDLY signals for nonpipelined transfers are different. The device that is receiving data indicates that it can receive data in the next cycle with its flow control signal instead of indicating that it can receive data in the current cycle, as described in the section detailing nonpipelined transfers.

Parity has been added to the bus, but it has been added in such a way that devices that do not generate or check parity will still work, as parity does not necessarily have to be generated.

4.7 Clocking

The bus is designed to run anywhere from 25 to 40 MHz and all GIO64 devices should be capable of running at any speed between 25 and 40 MHz. There are two clocks provided to each GIO64 device, one clock is the bus clock which will run at 25 to 40 MHz. The second clock will be twice the frequency of the bus clock. Both of these clocks are differential ECL signals.

The worst case clock skew from any flop on the GIO64 but to any other flop on the GIO64 bus is 3.3ns.

4.8 GIO64 Interrupts

There are three interrupt/status lines, $\overline{\text{INTERRUPT}}$(n)(2:0), for each GIO64 slot. These signals can generate CPU interrupts or their state can be read by the CPU. There is also one status signal, STATUS(n). The CPU can read the state of the STATUS signal, but it cannot generate an interrupt.

4.9 Pipelined GIO64 Slot Pinout

The following list of signals are available to a pipelined GIO64 device. All devices on the bus must be CMOS, although TTL thresholds are used. The bus will not support bipolar input buffer loading. The exact pinout of the connector and type of connector has not been determined at this time. The power limits and voltages also have not been determined.

TABLE 8 GIO64 Signals.

Signal	Device	Direction	Pin Number
P_AD(63:0)		i/o	
P_ADP(7:0)		i/o	
$\overline{\text{P\_VLD\_PARITY}}$		i/o	
$\overline{\text{P\_AS}}$		i/o	
P_READ		i/o	
GRXDLY(n)		o	
MEMDLY		i	

Signal	Device	Direction	Pin Number
$\overline{\text{BREQ}}$(n)		o	
$\overline{\text{BGNT}}$(n)		i	
$\overline{\text{P\_BPRE}}$(n)		i	
$\overline{\text{INTERRUPT}}$(2:0)		o	
STATUS(n)		o	
GIO64_CLK		i	
$\overline{\text{GIO64\_CLK}}$		i	
GIO64_CLK2		i	
$\overline{\text{GIO64\_CLK2}}$		i	
P_GSIZE64		i	
$\overline{\text{DMASYNC}}$(n)		i	
gio64_40mhz		i	
$\overline{\text{RESET}}$		i	
GIO64_SPEED(1:0)		i/o	

4.10 GIO64 Timing: Nonpipelined and Pipelined

In calculating the worst case path on the GIO64 bus, there is a basic assumption that the clock is distributed in such a way that the clock arrives at every GIO64 device flip-flop at the same point in time. Therefore, the clock to a gate array that is connected to the GIO64 bus will get an early clock as seen at the gate array pin compared to a discrete flip-flop on the GIO64 bus. Since it is not possible to get the clock to every flip-flop on the GIO64 bus at the exact same time there is a clock skew budget of 3.3 nanoseconds to get the clock distributed to every device on the GIO64 bus.

There are four basic parts to the GIO64 cycle timing: clock to output time of the output flip-flop, the bus wire delay, setup time to the input flip-flops, and the clock skew. There are also some hold time requirements.

The 3.3 nanoseconds of clock skew can be divided into three components: clock driver skew, net length skew, and clock distribution skew on the GIO64 device. The GIO64 device could be a gate array or a board. The clock distribution scheme is as follows:

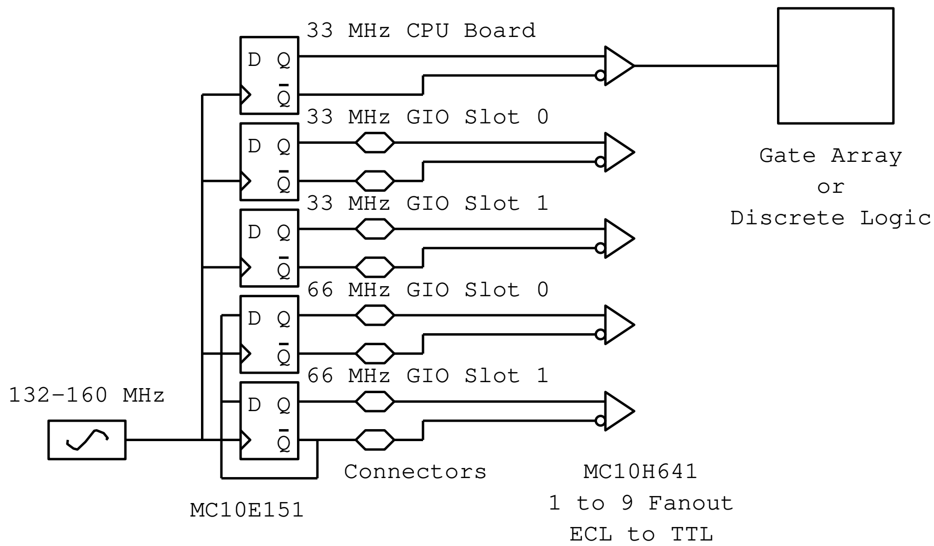

Figure 4-24. GIO64 Clock Distribution.

All of the clock divider flops, MC10E151, are in the same package so that the output to output skew will be minimized. Since the temperature, voltage and process of that part is the same, the worst case maximum clock to output time, 800 ps, minus the best case minimum clock to output time, 450 ps, can be derated. To be conservative, this will be derated to a 300 ps output to output skew. The outputs of the MC10E151 then go to a clock fanout buffer, MC10H641. This is an ECL to TTL, 1 to 9 output fanout buffer. The output to output skew is 500 ps for the same package and 1 ns for different packages. This configuration should minimize clock skew associated with the clock drivers.

The next contributor to clock skew is the net delays. All net lengths will be matched to within 1 inch on each board so that the worst case difference between clocks due to different length etch will be 0.4 ns, (0.2 ns/inch). This includes matching the delay for the clock to a discrete part to the delay of the clock fanout tree in a gate array by changing the net length from the clock buffer to the different parts. Another factor is impedance mismatches between boards, through connectors, and different loading of the net, due to a different number of MC10H641 parts. For these differences, 0.6 ns will be allotted. In order to make this work the clock at the GIO64 connector will be early by some amount. The exact amount has not been determined, but it will be on the order of 8.0 ns.

The last contributor to skew is from the device that is being clocked. There is a delay from when the clock changes at the pin of a gate array to when it changes at the flip-flop inside the gate array. There are two numbers to examine: the first is the delay from the pin to the flip-flops. This can be very large and therefore the skew over process, temperature, and voltage can be very large. The second number that is important is the skew from any flip-flop to any flip-flop in an array. It may require a PLL on the gate array to reduce the skew caused by these factors. The total skew budget for a gate array is 1.0 ns. It is important to realize that the delay can be more than 1.0 ns, but it is the skew which is important. The delay is compensated by different length etch to each of the chips that have a different clock fanout delay.

The total skew budget for the CPU board to a GIO64 device is:

10E151 Clock Divider Flip-Flop	0.3 ns
Different Length Etch	0.4 ns
Different Impedance of Nets	0.6 ns
MC10H641 Fanout Buffer	1.0 ns
Gate Array Fanout Skew	1.0 ns
Total:	3.3 ns

The resultant skew from different parts on the CPU board is somewhat less. Since none of the clocks are double frequency, the skew associated with the 10E151 can be eliminated. Also the skew between 10H641 is just the output to output skew for outputs in the same part instead of different parts so this is only 0.5 ns. The rest of the skew factors are the same. On the CPU board the GIO64 clock skew is:

Different Length Etch	0.4 ns
Different Impedance of Nets	0.6 ns
MC10H641 Fanout Buffer	0.5 ns
Gate Array Fanout Skew	1.0 ns
Total:	2.5 ns

Another factor in determining the longest GIO64 path is the worst case setup time for a device, which will be to a gate array. Keep in mind the clock edge is defined to take place at all of the flip-flops on the GIO64 bus at the exact same time so the delays associated with getting a clock from the gate array pin to the flop are not incorporated in the setup time calculation. The setup time to a FCT part, 74FCT652A, is 2.0 ns. These parts are used to implement the bidirectional registered transceiver between the nonpipelined and pipelined GIO64 bus. The worst case setup time for a gate array, LSI LCA100K can be calculated as follows:

TTL threshold input buffer, TLCHT	1.66 ns
Mux to hold the data, MUX21L	1.04 ns
Setup to scan flop, FD1S	1.88 ns
Total:	4.58 ns

These numbers are for a moderate load to account for net delays. To allow some extra margin the worst case setup time for any GIO64 input should be 5.0 ns or less.

The next factor in calculating the longest GIO64 path is the clock to output path. Since the delay for the gate array output buffers are given for different loading conditions, the output buffer delay will be incorporated by the net delay value. The worst case clock to output delay of a LSI LCA100K flip-flop, FD1S, is 3.0 ns with a moderate load to take care of net delays. The delay of a 2:1 mux will also be included for boundary scan. The delay of a MUX21L is 1.0 ns. Since a gate array driving the bus will be slower than discrete parts because the gate array has limited drive, this case will be examined. Twelve mA output buffers with slew rate control will be used for the GIO64 gate array outputs. The bus loading on the nonpipelined side of the GIO64 bus should be less than 130 pF. The worst case delay for driving 130 pF is 12.3 ns. The worst case bus loading is calculated as follows:

MC chip, 304 MQUAD	7.0 pF
MUX chip, 208 pqfp	7.0 pF
HPC3, 304 MQUAD	7.0 pF
Pipelined GIO64 Transceivers, 2 sets	24.0 pF
EISA chips, 4 chips, 208 pqfp	28.0 pF
Wire, 4 pF/inch, 12 inches	48.0 pF
Total:	121.0 pF

On the pipelined side of the bus the load should be about the same:

Pipelined GIO64 Transceivers, 1 set	12.0 pF
GIO64 Slot Connectors, 2 at 8 pF	16.0 pF

Wire, 4 pF/inch, 18 inches	72.0 pF
Gate array load/slot, 2 slots	30.0 pF
Total:	130.0 pF

Adding up the total bus delay:

Flop clock to output and mux	4.0 ns
Output buffer, including wire	12.3 ns
Setup time	5.0 ns
Clock skew	3.3 ns
Total:	24.6 ns

This leaves 0.4 ns of margin when the bus is running at 40 MHz. It may be possible to build a system with reduced loading that would be able to run at 40 MHz, although it will be hard. To achieve the numbers given above requires attention to the placement of the flops in gate arrays to achieve 5 ns of setup and 14.3 ns clock to output delay. It is also very important to be careful with the length of the bus and the number of loads on the bus. On the pipelined side of the bus, it is important to be careful with the stub length of the GIO64 signals. Also the capacitive loading needs to be less than 23 pF per slot counting the connector.

The hold time requirements can be calculated by adding the hold time of the device with the worst case hold time, which will be a discrete part. The hold time for a 74FCT652A is 1.5 ns. This time needs to be added to the clock skew of 3.3 ns to get the bus hold time requirement which is 4.8 ns. The minimum clock to output time for a discrete part is 2.0 ns and for a gate array is 1.5 ns. Since it will be very hard to achieve the 3.3 ns of hold time required from the capacitive loading of the bus the clock to the discrete flops can be made 0.5 ns early to trade off setup time for hold time. The setup time to the discrete flop is only 2.0 ns so adding 0.5 ns to it is still much better then the gate array setup time of 5.0 ns. The minimum clock to output time from the real GIO64 clock will be 1.5 ns for both the flop and the gate array. The maximum hold time from the real GIO64 clock is 1.0 ns for the discrete part and most likely negative for the gate array. Therefore the bus hold time for the pipelined side of the bus is:

Clock Skew	3.3 ns
Worst Case Hold Time	1.0 ns
Minimum Clock to Output Delay	-1.5 ns
Total Bus Hold Time	2.8 ns

The bus hold time on the nonpipelined side of the bus is less since the clock skew is only 2.5 ns.

Clock Skew	2.5 ns
Worst Case Hold Time	1.0 ns
Minimum Clock to Output Delay	-1.5 ns
Total Nonpipelined Hold Time	2.0 ns

The capacitive loading and etch delays will prevent the bus from changing within this 2.5 ns window.

Timing Work TBD:

- gio64 slot stub lengths on signals
- bus termination
- connectors
- spice results of fully loaded system to check signals and hold time

4.11 Pipelined GIO64 Mechanicals

The GIO64 slot connector and card mechanical are TBD.

4.12 Device Identification, Serial Number and ROM Registers

Every GIO64 device should have a set of special registers that can be used by the software to identify the devices that are installed. If a device has ROM, then it should, in addition, have a set of ROM registers that allow the software to easily read the ROM. These special registers should be located at the beginning of the devices address space. The base address of a device is determined by the GIO64 SLOT_NUMBER signals.

These registers need to be accessed in a special way since reading them is the only way to determine information about the size of a device (32 or 64 bits). The first requirement is that 64-bit devices must always return this register read data on AD(31:0). These special registers are aligned on double word addresses and the device must also respond to either of the two word addresses with the same register data. This allows the CPU to run in big or little endian mode and easily interpret the data the device returns.

There are four special registers:

1. The first is the Product Identification Word register. This register is located at the base address and the base address + 0x4. A read from either the base address or the base address + 0x4 should return the value of this register on AD(31:0). A unique, 8-bit Product ID Code is contained in the lower byte of the Product Identification Word. Refer to Chapter 2 for complete details about this Product Identification Word.

   Note: The ROM Present bit of this register must be a 1 if the option card has ROM that can be read with the ROM Index/ROM Read registers (described below).

2. The second special register, the Board Serial Number, is located at the base address + 0x8 and the base address + 0xc. This register contains a 32-bit board serial number. This register is optional.
3. The third register is the 32-bit ROM Index register located at the base address + 0x10 and the base address + 0x14. This register is present when the ROM Present bit in the Product Identification Word register is set to one. The CPU writes 0 to this register to initiate reading the ROM. Each subsequent read of the ROM Read register causes this Index to increment by 4.
4. The fourth register is the 32-bit ROM Read register located at the base address + 0x18 and the base address + 0x1c. This register is present when the ROM Present bit in the Product Identification Word register is set to one. This register always contains the contents of the ROM word identified by the address in the ROM Index register. Each read of this register causes the ROM Index to increment by 4 and a new word to be retrieved from ROM.

4.13 Miscellaneous Timing Diagrams

The following pages illustrate the GIO64 timing protocol for a variety of situations.

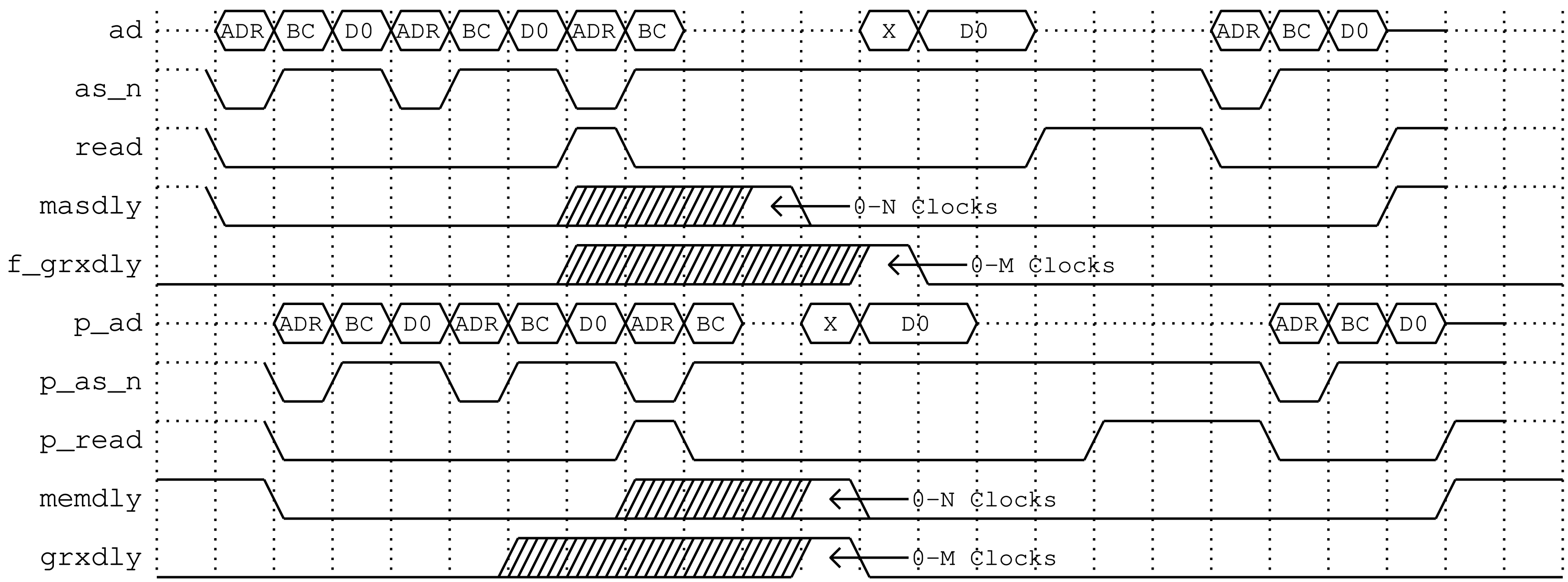

Figure 4-25. Two Writes, a single Read, followed by a Write for Pipelined GIO64

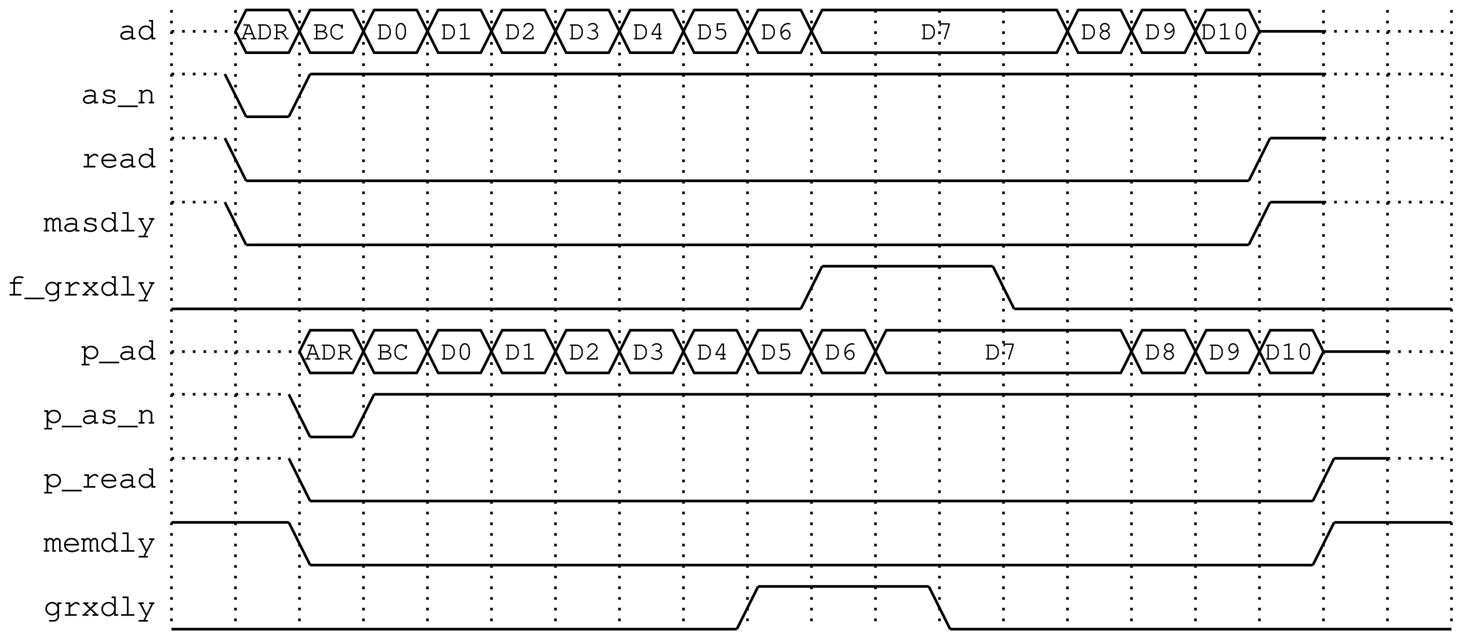

Figure 4-26. GRXDLY Asserted During a long Write to a Pipelined GIO64 Device

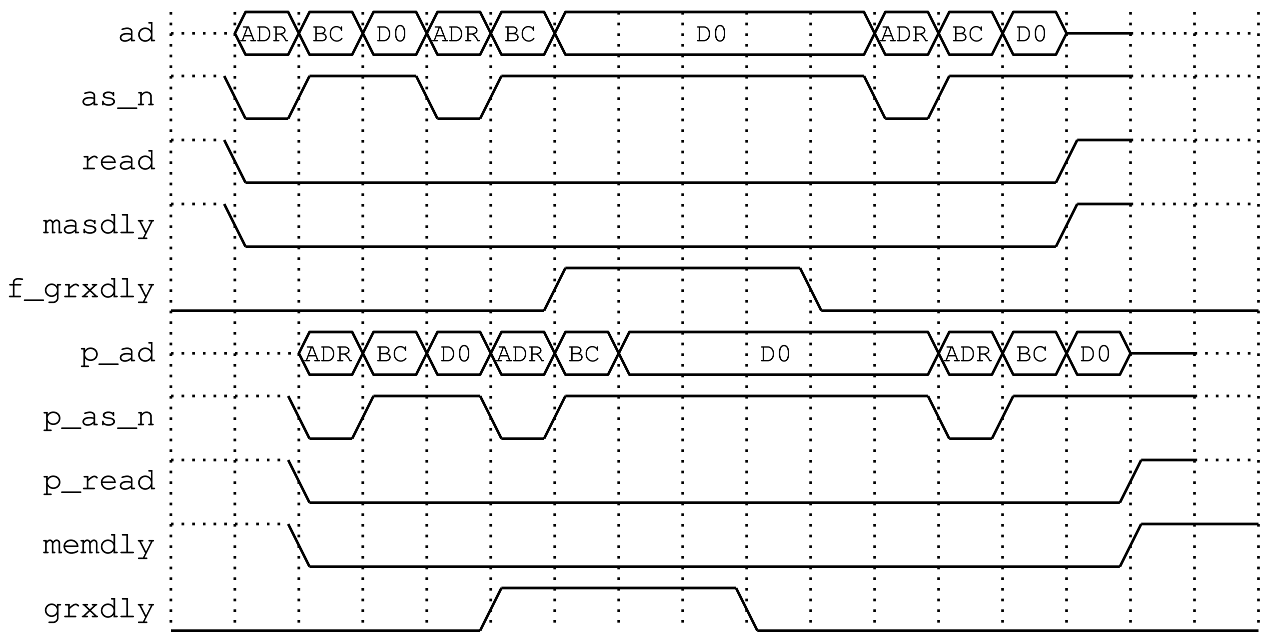

Figure 4-27. Single Write to a Pipelined GIO64 Device and GRXDLY

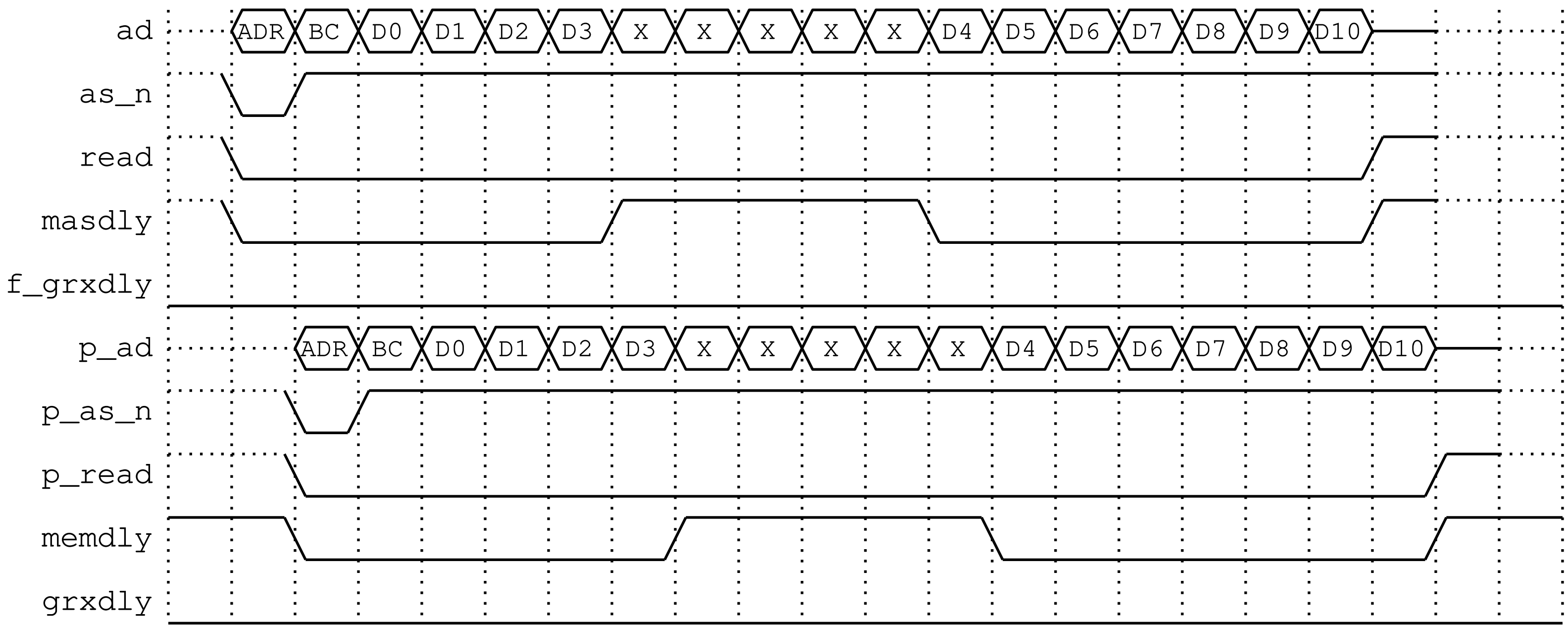

Figure 4-28. A Pipelined GIO64 Write and MEMDLY

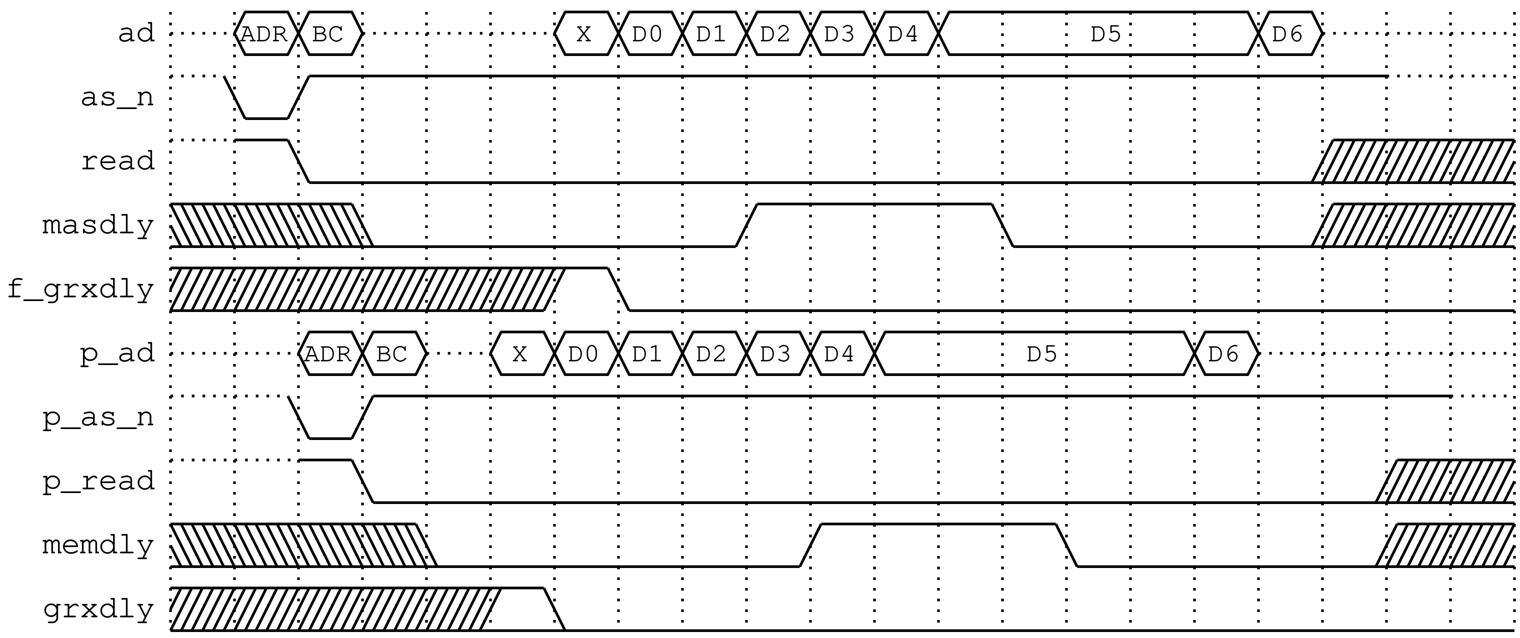

Figure 4-29. Read from a Pipelined GIO64 Device and MEMDLY

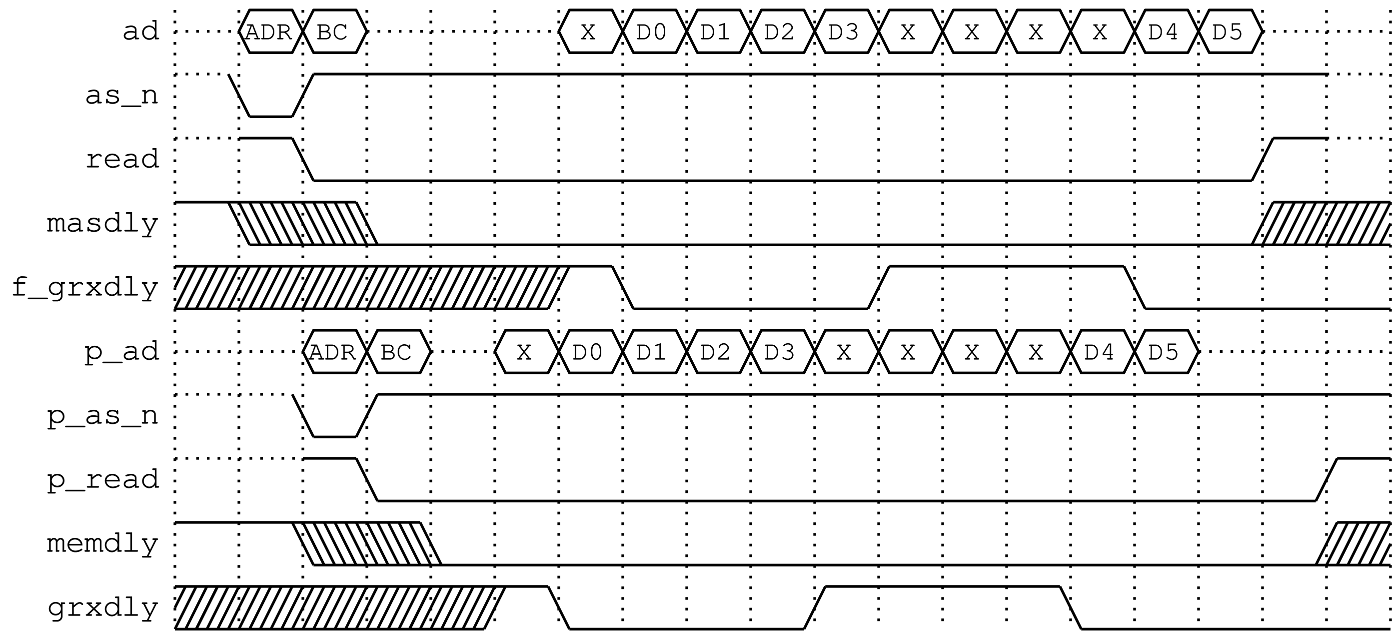

Figure 4-30. Read from a Pipelined GIO64 Device and GRXDLY

This page has been left blank intentionally.