Silicon Graphics
Computer Systems

Virtual DMA Specification

FastForward Project

Draft 1.5
February 13, 1992

Karim Abdalla
James Tornes

Silicon Graphics. Inc.

Contents

List of Figures

List of Tables

1 Introduction

This document describes the GIO DMA master within the *FastForward* memory controller (MC). The primary goal is to provide an efficient mechanism for transferring variable-sized blocks of data to and from user space in main memory to the graphics subsystem.

In past architectures, the underlying mechanism relied on purely physical addresses[1], even for the user-space memory region. This mechanism placed some serious requirements that had to be satisfied by the kernel software, much to the degradation system throughput.

The most important novel feature of new DMA engine is the support of virtual addressing of the user-space memory region to be transferred. The design is optimized so that the most frequent translations will be done on the fly by the DMA engine.

Unfortunately, due to the complexity of the virtual memory system, in particular, the current operating system structures involved in translating user virtual addresses, some help will still be required from the operating system executing on the CPU. However, this does not imply that the user must make a system call to set up a DMA.

The following section outlines the main obstacles that must be overcome in order to implement an efficient virtual DMA scheme. Various trade-offs can be made that affect the ease of implementation, system throughput and programming model.

Much of this document draws from the MC specifications by James Tonnes and the "Fast Forward Graphics DMA Specification" outlined by Robert Liston. The specification is significantly defined by the requirements of the graphics subsystem hardware and software architects. Great ideas and valuable information as always from James Tonnes and Wiltse Carpenter.

2 Key Issues

There are several issues that must be resolved in order to support efficient virtual DMA. The goal is to minimize the latency involved in the user setting up the DMA (via a system call or not), and the subsequent DMA transfer.

2.1 Cache Coherency

The first hard problem that must be overcome in order to implement any DMA scheme is that the data in memory to be DMA'd may not be up to date with respect to the cache. This was never an issue in the R3000 which had a write-through cache, as opposed to the write-back policy used in the R4000. The

[1]Except, of course, in the three-way transfer mechanism

converse problem entails the possibility that cached data may become stale if a DMA updates main memory.

There are two mechanisms for ensuring data coherency between cache and memory. The first method is to flush the cache, and the second method is for the DMA engine to snoop the cache. Cache snooping is only supported by the R4000MP which comes in the large package.

The R4000 does support a cache flushing instruction, but unfortunately, it is a privileged instruction. If the cache-flushing mechanism is to be used, it must be part of a system call. Given that we must take the latency penalty of making a system call through the general exception mechanism, it would be more efficient for the software (see section 2.2 below) and possibly simpler for the hardware (see section 2.3 below) to do the entire DMA setup and transfer process via a system call.

The cache-snooping approach would be more costly in hardware in terms of requiring an R4000MP as well as extra gates in MC, but it does lend itself to the possibility of a much faster implementation. This is also the only mechanism that would allow the user to set up and start the DMA process.

2.2 Synchronization

This is an issue that the software must deal with. The basic problem is that the user program that just kicked off a DMA process needs to know when the DMA is complete. For example, if the program initiated a DMA from graphics to memory, it has to know when it can read memory and get the new data. Another example is when the program can initiate another DMA process.

Polling a status bit in the DMA engine wastes CPU cycles, that could otherwise be doing some useful work. Waiting on an interrupt to a user routine is not really an option under IRIX.

If the DMA was set up and initiated by the operating system, the system could immediately swap in another process and only give control back to the DMA requesting process after the DMA transfer is complete. The processor can do useful work until it receives a DMA complete interrupt.

2.3 Virtual Address Translation

Since the DMA is a feature that should be available to a user application, and that application must be running in a virtual address space, all memory addresses that the user specifies must be translated before they can be used to address physical memory or hardware. For GIO (graphics) addresses, the user can specify the physical address since the device map is hard wired.

2.3.1 Why Virtual DMA in Hardware

Irrespective of whether the DMA is set up through a system call, the DMA hardware will translate the host memory addresses on a page-by-page basis. This requirement will greatly improve system performance since it removes a huge burden from the operating system, and also enable stride DMA and negative increments to work across page boundaries.

In past systems, when a user process made a system call for a DMA, the operating system had to translate every single page within the required virtual address range, set up a long descriptor list to the physical pages which could be scattered all over memory, and finally lock down all the pages in memory until the transfer was over.

By having the hardware translate the addresses, the DMA engine could use the virtual address to look into the user page table, and obtain the physical address if the page were resident, otherwise signal a page fault to the processor via an interrupt. This process can be repeated on a per-page basis.

2.3.2 Required Help from OS

The above scheme implies that the DMA hardware has physical pointers (PTEBases) to the user's page tables. Each PTEBase allows the hardware to translate (via one memory indirection, the actual page table look-up) a single two megabyte[2] aligned block in the user virtual address space. Since the user has a virtual address space of the order of gigabytes[3], a single page table may be insufficient to translate the desired block.

The PTEBases can be supplied by a number of mechanisms. If the DMA is set up by the system on behalf of the user, the system software can calculate the address of the page table specified by the starting virtual address and write it into a special PTEBase register inside the DMA engine.

If the DMA is set up directly by the user, or a system-initiated DMA runs over a 2 Byte boundary, the DMA engine must obtain a new PTEBase. There are two actions that the DMA hardware can take to rectify the missing PTEBase problem: either interrupt the processor for the new PTEBase, or look it up directly through the system structures in memory. This problem is deferred momentarily until the next section.

The operating system could also, on context switching, initialize a moderatesized PTEBase μTLB to point to a set of most likely to be used user page tables. By having a μTLB of more than one entry, multiple page tables could be accessed without PTEBase look-ups.

If we implement a μTLB of PTEBases, and devise some scheme for servicing PTEBase misses, then a single descriptor can describe the entire user space. The

[2]In the current virtual memory system, which is subject to change

[3]precise number depends on MIPS I, II or III architecture

single effective descriptor could just be written directly (using programmed I/O) to the DMA engine's appropriate register's.

2.4 Short Transfers (or the *v3f()* Problem)

There are very frequent situation when the user needs to transfer a small number of bytes from memory to a virtually mapped uncached device such as the graphics subsystem. This happens, for example, in the graphics library function *v3f()* which sends a 3D vertex (three consecutive floating point numbers i.e. twelve bytes) to the graphics pipeline.

In past R3000-based machines, these very short virtual DMA's were accomplished via three way transfers. The latter is a trick to do virtual DMA that crossed at most one page boundary. Three way transfers rely on the R3000 translating the virtual address and making it available to PIC by way of a store.

Unfortunately this approach does not lend itself to the R4000 due to it's write-back cache policy. It would be an impractical solution to place a software restriction that makes the programmer declare all areas of data likely to be written by way of a three way transfer as uncached, not to mention the lash of efficiency due to not cacheing the data.

The R4000 presents two alternatives for short transfers; programmed I/O or DMA.

3 Proposed Solution

The following subsections list the proposed choices for the above trade-offs and the arguments by which these decisions were reached.

3.1 System Call Only on R4000PC, R4000SC

Since the user must make a system call to ensure cache coherency in all but the R4000MP and the R3000, for all other machines, the entire DMA process until the end of the transfer should be managed by the operating system. This also simplifies the synchronization issues.

3.2 User and System Calls on R4000MP and R3000

For the R4000MP supporting cache-snooping, it is desirable to allow user DMAs without a making a system call, in addition to providing a system call. This mechanism is also possible for the R5000 due to it's write-through cache.

If the user page table(s) and user data pages are all resident, it is likely that a DMA could be set up and executed entirely in user mode without interrupting the processor at all. Any interrupt would effectively incur a system call overhead, nullifying the main benefit of user-initiated DMA.

The synchronization issue for user-initiated DMAs will be resolved by having the processor poll a special MC register. In actuality, MC would stall the read response until either a snoop occurs or the DMA completes. This has the benefit of not slowing down MC by continually responding to the poll.

3.3 Page Table Look-Up with PTEBase μTLB

In order to make relatively short (see section 3.7 below) user DMA transfers efficient, the hardware must have the PTLBase(s) of a users most frequently used page tables.

This will be implemented using a 4-entry μTLB of PTEBases that is written by the CPU during a context switch by the operating system. When the user later does a virtual DMA, the user virtual address high bits (VPNhi) will associatively lookup the PTEBase in the μTLB. The page table pointed to in memory by the PTEBase is indexed by the users low virtual address bits (VPNlo) to yield a page table entry (PTE).

If the PTE indicates a resident page, then the transfer can proceed using the page frame number (PFN) from the PTE. Otherwise the DMA engine signals a user page fault, by interrupting the processor. If the μTLB does not contain the correct PTEBase, or if the matching entry has been marked as invalid, which could theoritically happen if the operating system decides to swap out the user page table that the entry used to point to, then the DMA hardware signals a PTEBase fault.

Figure 1 shows user page table in main memory, as well as the PTEBase μTLB and various registers within the DMA engine that are involved in the look-up process. The figure also shows how the GIO physical address is used.

Every time the OS does a user context switch, it invalidates or updates all entries in the μTLB.

3.4 Protected Physical GIO Address

Since the user will always refer to a GIO device that is physically mapped at a known address, the user can provide that address directly by writing it as data into an MC register. The GIO_ADR descriptor register described in section 5.2 is dedicated for this purpose.

To provide some measure of security with regards to what physical GIO addresses a user can access, two protected registers called the GIO_MASK and GIO_SUBST are provided in the DMA engine are provided. The operating system can configure these registers for each user during context switch to control access to the physical map. Please refer to section 5.3.1 for more details.

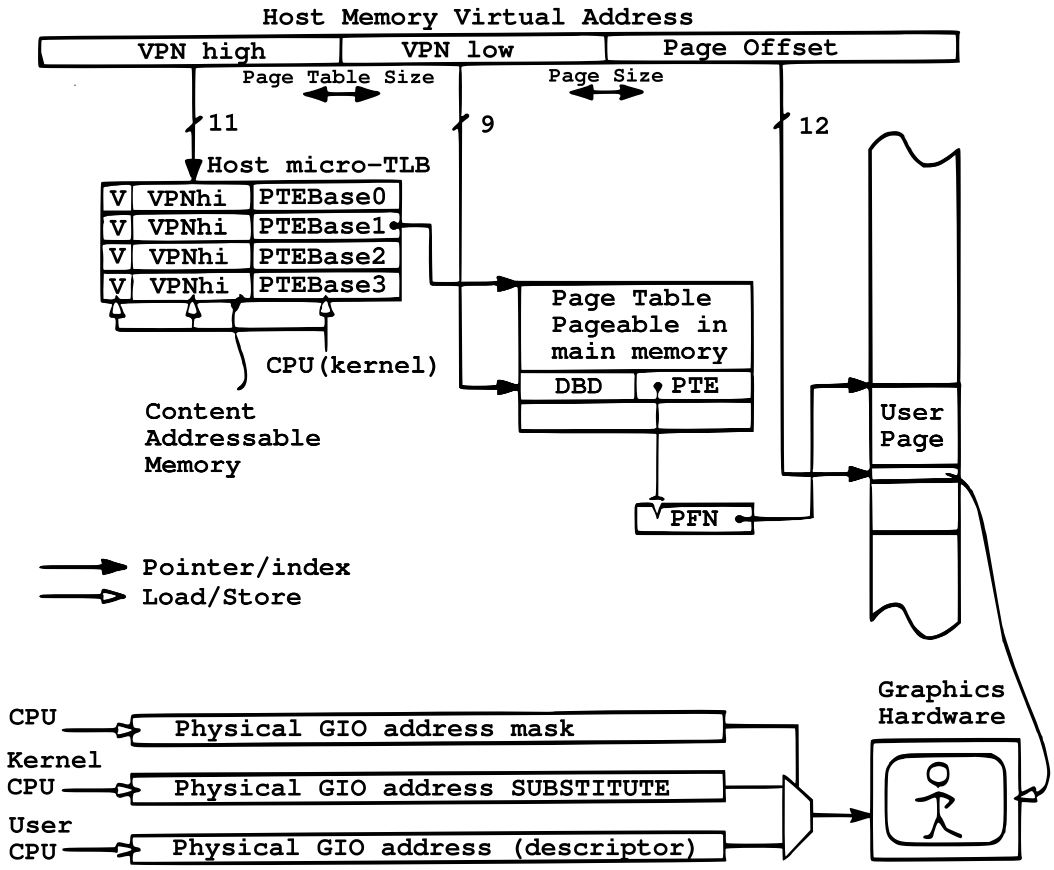

Figure 1: Address Translation and μTLB

3.5 Directly Written Single Descriptor

Since most DMA needs entail the movement of one contiguous block, or can otherwise be broken up into several consecutive DMA transfers (where a DMA transfer is defined as a contiguous unidirectional transfer), there will be a limit of a single descriptor per DMA transfer.

Since the addressable range of host memory by the DMA hardware is not a limiting factor, given that PTEBase μTLB misses can be serviced by the processor, a single descriptor can specify a DMA transfer of a single unlimited sized contiguous block.

The main motivation for limiting the number of descriptors to one is that the hardware responsible for descriptor fetching can be eliminated from the DMA engine. The one and only descriptor can be written directly to the DMA engine's appropriate registers using programmed I/O.

3.6 Programmed I/O for Very Short Transfers

Since three way transfers are impractical and system DMA calls (mandatory on all but the R4000MP and R3000) would incur too much CPU overhead for short transfers to mapped hardware addresses, the only remaining alternative for the R4000PC and R4000SC is to use programmed I/O for very short transfers.

Unfortunately, the R4000 incurs a seven- or eight-cycle penalty for uncached single writes. That is, the entire R4000 pipeline is frozen for seven or eight cycles after one write. An additional problem with programmed I/O common to any processor is that the data must be loaded into the cache from memory, if it is not there already, before it can be written with a store instruction. This not only takes time in reloading the cache, but also invalidates the previous data in those cache locations.

3.7 User Virtual DMA for Relatively Short Transfers

For the R4000MP with cache snooping and no system call overhead short DMAs might be more efficient than programmed I/O. The comparison can be made based on the performance estimates in section 4 below. One essential factor is the break-even point of the transfer size. This is about 8 words for the R4000MP. Transfers larger than that would be more efficiently done by the DMA mechanism.

For short DMAs to have any chance of being fast enough, the DMA engine has to be able to either look up the PTEBase by itself or have such a large uTLB that μTLB misses are very rare. The μTLB would have to be large enough to support many short DMAs scattered in virtual space, a situation quite likely if small structures to be DMA'd are frequently allocated and deallocated from the stack and heap. The current μTLB size chosen for implementation is 4 entries.

This mechanism can also be used with the R3000, since the memory is always consistent with the cache unless there has been a DMA into memory. The latter case should be handled by flushing the cache after a DMA write into memory. However, most user DMAs will DMA reads (writes to the graphics pipeline).

3.8 System Call for Long Transfers

As explained in section 3.2, very long transfers should be done through a system call so that the CPU does useful work instead of spinning. Choice of method will be up to the discretion of the programmer.

4 Performance Estimates

4.1 Programmed I/O *v3f()* Calls on R4000

In order to estimate the rate at which the *FastForward* CPU can execute *v3f()* calls, one must make certain assumptions such as the organization and state of the R4000 caches. In both of the following scenarios, a the R4000 line size is assumed to be 4 words for the primary cache, and 16 words for the secondary cache.

Furthermore, as a worst case, the vertices to be loaded from memory and stored to the graphics hardware are assumed to miss both caches on the load, and the secondary cache line is assumed to be dirty. The page accessed is assumed to be resident in physical memory, however the refill is assumed to start with a non-page-mode read due to the GIO synchronization operation in MC to switch from GIO bus (where the last set of vertices were being finally sent) to the processor bus.

The cycle counts can be broken down into the essential cycles needed by the CPU to execute the *v3f()* code plus the extra cycles for cache refills. The following is a cycle count of the essential cycles required by the CPU to execrate the instructions for a single *v3f()* call. All cycle counts are all normalized to Pclock cycles, the processor's internal 10ns clock.

#	instruction	each	total
1	load imm.	2	2
3	load word	1	3
3	store word	8.5	22.5
1	return	1	1
Execution Cycles			31.5

In the first scenario, it is assumed that the processor is continuously issuing *v3f()* calls as fast as possible in a zero-overhead loop. It is also assumed that all the vertices consecutive, thus there is one secondary cache miss per five vertices. The calculation is carried out to see how long it takes to issue five consecutive *v3f()* calls. Note that the second level refill concurrently refills the originally missed first level cache line.

#	operation	each	total
5	execution *v3f()* instructions	31.5	158
1	first level miss, second level miss	96	96
3	first level miss, second level hit	12	36
1	round-trip SysAD/GIO bus synch	16	16
Total Pclocks for five *v3f()* calls			306
Total Pclocks for one *v3f()* calls			61

As can be seen from the above calculation, given the assumptions in the first scenario, a *v3f()* call takes 61 Pclock cycles or 610ns on average. This implies a peak rate of 1.64 million *v3f()* calls per second.

In the second scenario, it is assumed that the vertices have little locality, and that one secondary level refill is required for each call. The first level refill is done in parallel.

#	operation	each	total
1	execution *v3f()* instructions	31.5	31.5
1	first level miss, second level miss	96	96
1	round-trip sysad/gio bus synch	16	16
Total Pclocks for one *v3f()* call			145

In this scenario of bad vertex locality, it takes 145 Pclock cyles or 1.45 us per *v3f()* call. This corresponds to a rate of 692 thousand *v3f()* calls per second.

4.2 User-Initiated DMA *v3f()* on R4000MP

In these calculations, the cache line sizes are assumed to be as in the above calculations for programmed I/O, i.e. 16 word secondary, 4 word primary. Furthermore, we assume that both μTLB entries are valid, i.e. we have the PTE-Base for the user's page table in memory, and we have the page frame number for the graphics hardware. We also assume that the user page is resident in memory.

The CPU instructions are not counted since the synchronization poll actually allows the processor to continue as soon as the actual DMA transfer starts and the write buffer in MC is empty. Thus the next transfer can be set up by the CPU while the last one is in progress, so for short set-up code, the entire CPU time is overlapped with the transfer.

CPU Cycles	Operation
15	page table look-up
4	synchronise FIFO to CPU clock
12/47	snoop clean/dirty data
4	synchronise FIFO to GIO clock
20	transfer data
55/90	Total Tclocks for one *v3f()* call

From the above calculations, it takes 55 or 90 (20 ns) Tclock cycles, depending on if the cached data is clean or dirty, to do a single *v3f()* via a user mode virtual DMA on the R4000MP. These numbers correspond to rates of 909/536 thousand *v3f()* calls per second.

4.3 GIO Bus Rates for Small User-Initiated DMA on R4000MP

For the above *v3f()* 3-word virtual DMA transfers, the GIO bus attains a rate of 10.9/6.67 MBytes/sec, depending on if the snoops are clean or dirty snoops.

For larger DMA blocks, the table look up time is amortized over the data transfer. For 8 word DMA, the GIO bus attains 29/20 MBytes/sec, and for 16 word DMA, the GIO bus reaches 51 MBytes/sec assuming clean snoops all the way.

4.4 Three-Way Transfer *v3f()* in Magnum

The three-way transfer mechanism in Magnum takes only 6 CPU cycles per *v3f()* call. This rate is equivalent to 5.6 million calls per second, at 33 Mhz. Thus in a zero-overhead loop, the R3000 could theoretically issue the calls faster than PIC could transfer the vertices on the GIO bus, resulting in Magnum's limit being set by PIC/GIO bus at 1.27 million vertices per second.

CPU Cycles	Operation
2	load immediate
1	trigger writes
1	start address writes
1	end address writes
1	subroutine return
6	Total SysOut cycles for 1 *v3f()* call

5 Software Interface

The programmer view of the DMA machinery consists primarily of a set of addressable registers in MC. These registers are mapped as uncached locations in the user's virtual address space by the operating system. The user writes various parameters describing the nature of the DMA into these registers, and then writes a special location to start the transfer.

In user-initiated DMA, the user normally polls a special register to determine if the transfer is complete. The operating system can take an interrupt generated by the DMA engine on completion of the transfer. The transfer complete interrupt is maskable by the OS and should always be disabled in user mode. The DMA engine can also interrupt the processor for various faults. The OS gets a DMA interrupt and can read a cause register in MC that specifies the cause of the interrupt.

5.1 Main Features

The main features supported by the DMA engine are:

- Virtual addressing of host memory and graphics
- Incrementing / Decrementing line scanning
- Up / Down line striding
- Y Zooming DMA
- Fill mode memory-write DMA
- Preemptable / Restartable DMA
- Save / Restore multiple DMA contexts

5.2 Descriptor Registers

This section describes the structure that specifies a DMA transfer. Recall that this is not the traditional linked list in memory, but a set of special registers in the DMA engine.

This set of registers, referred to as the descriptor registers, specify certain details about the transfer of a block in user virtual space on the host to/from the graphics subsystem. Some of these registers are the only copies of the addr sequence counters within the DMA engine.

Once a DMA transfer starts, the descriptor registers implicitly define the state of the transfer so far, so if the DMA gets stopped by the processor or is waiting on a fault, the entire state can be read out or restored for context switching.

- GIO_MEMADR
  This register holds the 32 bit address of the block to be transferred to/from memory. This address is normally treated as a virtual address, and the top bit is ignored. If the Xlate bit of the GIO_CTL register is clear (should always be set in user mode), then the full 32 bits indicate a physical address.

  The contents of this register are incremented according to the counting scheme designated by the GIO_SIZE, GIO_ZOOM and GIO_STRIDE registers described below. GIO_MEMADR always contains the next address

Name	Access	Field						
GIO_MEMADR	RW	Memory Address[31:0], Don't Set Defaults						
GIO_MEMADRD	RW	Memory Address[31:0], Set Defaults						
GIO_SIZE	RW	Line Count[15:0]			Line Width[15:0]			
GIO_STRIDE	RW	Unused	Line Zoom[9:0]		Stride[15:0]			
GIO_ADR	RW	GIO Address, Don't Start DMA						
GIO_ADRS	RW	GIO Address, Start DMA						
GIO_MODE	RW	Unused	Long	Snoop	Dir	Fill	Sync	Mode
GIO_COUNT	RW	Unused	Zoom Count[9:0]		Byte Count[15:0]			

Table 1: DMA Descriptor Register Summary

to be DMA'd to/from memory. This is so that when GIO_MEMADR is read and then, later written back as part of the state save/restore code of the context switcher, the DMA transfer will be resumed at the correct address.

- GIO_MEMADRD
  Writing to this address writes the given value into the GIO_MEMADR register as a write to GIO_MEMADR would with an extra side effect. The side effect entails automatically writing a set of default values into some of the descriptor fields. The descriptor fields that get loaded and the corresponding default values are shown in table 2.

- GIO_SIZE
  This register consists of two distinct fields that are each 16-bit wide unsigned integers. The Line Count field denotes the total remaining number of scan lines to be transferred. This is the normal way of specifying the length of a DMA transfer. It is also made visible for context switchability. This field is cleared when GIO_MEMADRD is written.

  The Line Width field denotes the number of bytes per scan line in memory. A scan line in memory is transferred as consecutive (given a scan direction) bytes until the line width is satisfied, at which point the address of the next scan line is computed using the stride. This field is cleared when GIO_MEMADRD is written.

- GIO_STRIDE
  This register contains two fields. The Line Zoom field is an 10-bit unsigned integer denoting the number of times that a given scan line is to be transferred. This is useful for vertical zooming of bitmaps. This number gets reloaded into the Zoom Count field each time the Zoom Count decrements to zero.

Field	Value
Line Count	0x0001
Stride	0x000
Line Zoom	0x01
Zoom Count	0x01
Line Width	0x00C
Byte Count	0x00C
Long	0
Snoop	1
Fill	1
Dir	0
Sync	0
Mode	00

Table 2: Default Descriptor

The Stride field is a 16-bit signed integer. This quantity is added to the address at the end of one line to compute the address of the next line in (virtual) memory. Zero stride means that the lines are contiguous in memory. Negative stride is useful for bitmap reflection in the x-axis. The Line Zoom and Stride fields get loaded with default values of one and zero respectively when GIO_MEMADRD is written.

- GIO_ADR
  This register contains the physical address of the GIO device to be read or written. The physical address is subject to certain constraints that can be programmed by the operating system into the GIO_MASK and GIO_SUBST protected registers in order to limit access by the user to the physical map. See discussion of these registers for more details on the protection scheme.

  Also, if the DMA transfer is a memory write (GIO read) and the Fill bit is set, then the address is used directly as the read data instead of indirecting through the GIO location.

- GIO_ADRS
  This address is a write-only alias for the GIO_ADR register described above. Writing to this address also has the side effect of starting a DMA transfer.

  With the given defaults and aliased registers, *v3f()* DMAs can be set up and triggered in two writes: GIO_MENADRD and GIO_ADRS. Simple contiguous DMA writes like VBFs but of a different size can be set up and started in only three writes: GIO_MEMADRD, GIO_SIZE, and GIO_ADRS.

- GIO_MODE
  This register is composed of six separate fields. The Long field is the bit (GIO_MODE[6]) that specifies whether the DMA transfer should be treated as a long burst or short burst transfer. If the Long bit is set, the transfer will occur during the DMA time slice of the GIO bus, otherwise the transfer will occur during the CPU time slice.

  The Snoop field is a single-bit flag (GIO_MODE[5]), that if set, enables cache snooping (R4000MP only) during the DMA transfer. In some cases it may be faster to flush the cache before the transfer and avoiding the snooping overhead during the transfer.

  The Host Dir is a single bit (GIO_MODE[4]) and defines the direction that a scan line is scanned. A value of one signifies address incrementing while a value of zero signifies address decrementing, useful for bitmap reflection in the y-axis.

  The Fill field is a single bit (GIO_MODE[3]) that, if set, attaches a special meaning to the GIO_ADR register when mode field is set to "write". When the Fill bit is set and the DMA mode is write (to memory), then instead of using the data at the given GIO address, GIO_ADR is used as data directly. The GIO / graphics device is not involved in the transaction.

  The Sync field is a single bit flag (GIO_MODE[2]) that delays the start of a DMA until the DMASYNC pin on MC is asserted by an external device. This is normally used to synchronize the DMA with the vertical retrace of the monitor to eliminate screen flicker during graphics DMA.

  The Mode field is a two-bit field (GIO_MODE[1:0]) that specifies if the DMA is a memory read, memory write, or accumulation mode transfer. The mode is encoded in the following way:

mode [1:0]	operation
00	DMA read (from memory to GIO)
10	DMA write (from GIO to memory)
X1	accumulation buffering (unimplemented)

- GIO_COUNT
  This register contains two down counter fields that contain state normally used only for context switching. Most users will not want to read or write this register.

  Zoom Count is an 10-bit unsigned integer that specifies the number of times to transfer the first (current) line before moving onto the next line.

This field is automatically loaded with the value being written into Line Zoom when GIO_STRIDE is written, and cleared when GIO_MEMADRD is written.

Byte Count is an 16-bit unsigned integer that specifies the number of bytes to transfer in the fast (current) line before decrementing the Zoom Count. This field is automatically loaded with the value being written into Line Width when GIO_SIZE is written, and cleared when GIO_MEMADRD is written.

Since this register gets initialized by writes to GIO_STRIDE, GIO_SIZE or GIO_MEMADRD, if any other value is intended by the user (most likely the kernel doing a context switch), GIO_COUNT should be written after those registers.

The following fragment of C code describes the DMA algorithm. Note that all variables are initialized using either the last value in the register (as when resuming a stopped DMA), or as a result of a direct register write (as when restoring the entire context). When initializing a DMA for the first time, zoomcount and bytecount are automatically initialized with linezoom and linewidth respectively, with the option of being modified.

```
while (linecount > 0) {
  linecount--;
  while (zoomcount > 0) {
    zoomcount--;
    while (bytecount > 0) {
      bytecount--;
      transfer(gio_addr, memory_addr, mode, fill);
      if (dir == UP)
        memory_vaddr++;
      else
        memory_vaddr--;
      bytecount = linewidth;
    }
    if (zoomcount > 0)
      if (dir == UP)
        memory_vaddr -= linewidth;
      else
        memory_vaddr += linewidth;
  }
  zoomcount = linezoom;
  memory_vaddr += stride;
}
```

Name	Access	Field							
GIO_MASK	RW	GIO Mask[31:0]							
GIO_SUBST	RW	GIO Substitute Value[31:0]							
GIO_CAUSE	RW	Unused				Complete	Clean	TLB miss	Page Fault
GIO_CTL	RW	T Limit	S Limit	DecSlv	Xlate	IntMask	Cache Size	Page Size	PTE Size

Table 3: Kernel DMA Control Register Summary

5.3 Control Registers

The GIO DMA master control registers are split into two sets that are addressed within different page boundaries. This is for security reasons. One set of registers (see table 4) can be mapped as accessible to the user (and kernel, of course) while the other set of registers (see table 3) can be accessed only by the kernel.

5.3.1 Protected Registers

- GIO_MASK
  This register contains a 32-bit field that the operating system can write to individually specifies how each corresponding bit of the user-specified GIO physical address gets transformed before accessing the GIO device. A one in a certain position indicates that the DMA engine should substitute the corresponding bit of the GIO_SUBST register instead of the user-specified bit. A zero in the specific bit position in the GIO_MASK indicates that the user address bit should be used as-is.

- GIO_SUBST
  This 32-bit register contains the values to be subsituted for the user specified value according to the GIO_MASK register to form the actual physical address used to access a GIO device.

- GIO_CAUSE
  All MC interrupt events (except for bus errors) are multiplexed onto a single MC interrupt pin. The CPU has to read this register in MC to determine the cause of the interrupt. The fields of this register are self-explanatory single bit flags that are set if the respective event caused the interrupt.

  Upon completion of a DMA transfer, the complete (GIO_CAUSE[3]) field is set, and, if the IntMask field of GIO_CTL had been set, an interrupt is asserted on the interrupt pin.

  There are also three kinds of DMA faults that can cause interrupts. These are the result of an attempted write to a page that has been marked as clean, indicated by Clean (GIO_CAUSE[2]), a missing or invalid TLB entry during address translation, indicated by TLB miss (GIO_CAUSE[1]), or a missing user data page, indicated by Page Fault (GIO_CAUSE[0]), at the end of a lookup.

The CPU can clear interrupts by writing zeros to this register.

- GIO_CTL

  This register contains six control fields that are used in configuring the virtual DMA system. Most of these fields will not change after boot-time initialization. The only likely exception to this rule is if the kernel wishes to do an DMA directly to/from physical memory space, it would need to clear the Xlate bit.

  The T Limit field (GIO_CTL[29:20]) is a 10-bit unsigned integer that specifies the maximum number of GIO bus cycles that the DMA engine can hold the GIO bus before relinquishing it back for rearbitration, giving the CPU a chance to make external accesses.

  The S Limit field (GIO-CTL[16:12]) is a 5-bit unsigned integer that specifies the maximum number of secondary cache lines (R4000SC only) that the DMA engine can snoop and transfer before relinquishing the GIO bus back for rearbitration, again giving the CPU a chance to make external acceses.

  Periodically allowing the CPU to make external accesses MC registers, memory or GIO devices is essential during long DMA transfers, since it permits the CPU to service any interrupts that occur during that transfer.

  GIO_CTL[9] contains the DecSlv field which, if set, indicates that that the (graphics) GIO slave supports the true GIO bus protocol for decrementing DMA. If the slave does not support the GIO bus protocol for decrementing DMA, clearing this bit allows memory address decrementing DMA transfers, with the restriction that the transfers must be word aligned in memory. This bit can be programmed on power-up, and should not need to be changed later.

  GIO_CTL[8] contains the Xlate field which, if clear, causes the DMA hardware to interpret GIO_MEMADR as a physical address. This field should only be cleared during system set up DMA, since the user should not have access to the physical address space. This feature is useful for debugging and when the operating system wishes to make a DMA on behalf of itself to/from a known physical memory block.

  The IntMask field (GIO_CTL[4]) is the mask for interrupts generated by the GIO DMA master. Currently the DMA engine can produce an interrupt due to four events in addition to MC's bus error interrupt. Of the four interrupt causes Clean, TLB Miss, Page Fault and DMA Complete, only the latter is maskable by IntMask. This mask should be accessible

Name	Access	Field		
GIO_STDMA	RW	0		Start
GIO_RUN	R	0	Run	Cause

Table 4: User DMA Control Register Summary

only to the kernel who should clear it before handing control back to the user. This prevents the DMA complete event from causing an interrupt during user-initiated DMA.

The Cache Size field (GIO_CTL[3:2]) specifies the cache line size in terms of x, where $line\_size = 16 \times 2^x$ bytes.

The Page Size field (GIO_CTL[1]) specifies the page size. A value of zero implies that pages are 4Kbytes, and a value of one implies 16Kbytes. Finally, the PT Size field (GIO_CTL[0]) specifies the number of bytes per page table entry, so that the page table may be indexed correctly. A value of zero implies 4 bytes per PTE and a value of one implies 8 bytes per PTE.

5.3.2 Unprotected Registers

- GIO_STDMA
  This register contains a single-bit field that specifies the run status of the DMA engine. The setting of Start (GIO_STDMA[0]) by the CPU will cause the DMA engine to begin transferring data according to the current state of the descriptor registers. This bit should also be written to restart a DMA transfer that had been waiting for a fault to be handled by the CPU.

  By writing a zero, the CPU can stop the current DMA. The response time from the software clearing the Start bit to the DMA engine stopping is not guaranteed since it depends on some event that disturbs the normal operation of the DMA transfer. This event might be dirty cache data in response to a snoop or any one of the three faults.

  Before being stopped by the processor, the DMA engine needs to proceed with the transfer up to a point where the entire state of the transfer can be captured in the descriptor registers, in case the stop is due to a context switch.

- GIO_RUN
  This register contains the run status of the DMA engine. A single bit field Run (GIO_RUN[4]) can be read (polled) by the CPU to determine

Name	Access	Field				
GIO_TLBHI	RW	VPNhi[31:21]	Unused (all zeros)			
GIO_TLBLO	RW	Unused	PTEBase[25:6]	Unused	V[1]	Unused

Table 5: μTLB Entry Format

if a DMA transfer is in progress. A one in this bit location means that the transfer is currently in progress with no unserviced faults. A zero means that the DMA has stopped either due to successful completion or due to a fault. The exact cause can be determined by examining the remaining status bits in this register. A read-only copy of the cause field of the GIO_CAUSE register (GIO_RUN[3 downto O]) is available for this purpose.

Note that while polling this register, it is possible to find that one or more Cause bits are a one at the same time that the Run bit is also a one. This is due to the fact that faults can modify the Cause field before the DMA engine has had time to fully wind down, as indicated by the Run bit still being a one.

5.4 μTLB

From the CPU point of view, the μTLB consists of a set of protected registers that can be read or written like any other register. The DMA engine relies totally on the CPU to ensure that entries marked as valid correspond to the current state of the virtual memory system. The DMA engine itself cannot modify the contents of these registers.

Each of the four μTLB entries consists of two registers; GIO_TLBHI[n] and GIO_TLBLO[n], where n is the entry number (0 through 3 inclusively). The format of these registers is shovn in table 5.

- GIO_TLBHI
  This register contains the VPNhi field which is the CAM tag that is associatively looked up by the VPNhi of the user's virtual address. The exact bits that define the user's VPNhi depend on the page size as defined in the GIO_CTL control register.

- GIO_TLBLO
  This register contains the PTEBase for the page table that maps the VPNhi address range. The number of significant bits and how they are combined with the user's VPNlo to yield a PTE address depend on the definition of the page size defined in the GIO_CTL control register. The remaining bits of this register are unused and return zero when read.

The V field is the hardware valid flag which must be cleared by the CPU if that entry becomes invalid for any reason. The CPU must set this bit when writing a valid μTLB entry. The remaining bits of this register are unused and return zero when read.

The format of the GIO_TLBLO is chosen for maximum compatibility with the R4000 (MIPS II & III) EntryLo format.