Chapter 13

VME Device Attachment

Several Silicon Graphics computer systems support the VME bus. This chapter gives a high-level overview of the VME bus, and describes how the VME bus is attached to, and operated by, each system.

This chapter contains useful background information if you plan to control a VME device from a user-level program. It contains important details on VME addressing if you are writing a kernel-level VME device driver.

- "Overview of the VME Bus" on page 298 summarizes the history and features of the VME bus architecture.
- "VME Bus in Silicon Graphics Systems" on page 300 gives an overview of how the VME bus is integrated into Silicon Graphics computer systems.
- "VME Bus Addresses and System Addresses" on page 304 discusses the relationship between addresses on the VME bus and addresses in the physical address space of the system.
- "Configuring VME Devices" on page 310 tells how to configure a device so that IRIX can recognize it and initialize its device driver.
- "VME Hardware in Challenge and Onyx Systems" on page 313 documents the hardware details of the VME implementation on those systems.

More information about VME device control appears in these chapters:

- Chapter 4, "User-Level Access to VME and EISA," covers PIO and DMA access from the user process.
- Chapter 14, "Services for VME Drivers," discusses the kernel services used by a kernel-level VME device driver, and contains an example.

Overview of the VME Bus

The VME bus dates to the early 1980s. It was designed as a flexible interconnection between multiple master and slave devices using a variety of address and data precisions, and has become a popular standard bus used in a variety of products. (For ordering information on the standards documents, see "Standards Documents" on page xxxviii.)

In Silicon Graphics systems, the VME bus is treated as an I/O device, not as the main system bus.

VME History

The VME bus descends from the VERSAbus, a bus design published by Motorola, Inc., in 1980 to support the needs of the MC68000 line of microprocessors. The bus timing relationships and some signal names still reflect this heritage, although the VME bus is used by devices from many manufacturers today.

The original VERSAbus design specified a large form factor for pluggable cards. Because of this, it was not popular with European designers. A bus with a smaller form factor but similar functions and electrical specifications was designed for European use, and promoted by Motorola, Phillips, Thompson, and other companies. This was the VersaModule European, or VME, bus. Beginning with rev B of 1982, the bus quickly became an accepted standard.

VME Features

A VME bus is a set of parallel conductors that interconnect multiple processing devices. The devices can exchange data in units of 8, 16, 32 or 64 bits during a bus cycle.

VME Address Spaces

Each VME device identifies itself with a range of bus addresses. A bus address has either 16, 24, or 32 bits of precision. Each address width forms a separate address space. That is, the same numeric value can refer to one device in the 24-bit address space but a different device in the 32-bit address space. Typically, a device operates in only one address space, but some devices can be configured to respond to addresses in multiple spaces.

Each VME bus cycle contains the bits of an address. The address is qualified by sets of address-modifier bits that specify the following:

- the address space (A16, A24, or A32)
- whether the operation is single or a block transfer
- whether the access is to what, in the MC68000 architecture, would be data or code, in a supervisor or user area (Silicon Graphics systems support only supervisor-data and user-data requests)

Master and Slave Devices

Each VME device acts as either a bus master or a bus slave. Typically a bus master is a device with some level of programmability, usually a microprocessor. A disk controller is an example of a master device. A slave device is typically a nonprogrammable device like a memory board.

Each data transfer is initiated by a master device. The master

- asserts ownership of the bus
- specifies the address modifier bits for the transfer, including the address space, single/block mode, and supervisor/normal mode
- specifies the address for the transfer
- specifies the data unit size for the transfer (8, 16, 32 or 64 bits)
- specifies the direction of the transfer with respect to the master

The VME bus design permits multiple master devices to use the bus, and provides a hardware-based arbitration system so that they can use the bus in alternation.

A slave device responds to a master when the master specifies the slave’s address. The addressed slave accepts data, or provides data, as directed.

VME Transactions

The VME design allows for four types of data transfer bus cycles:

- A read cycle returns data from the slave to the master.
- A write cycle sends data from the master to the slave.
- A read-modify-write cycle takes data from the slave, and on the following bus cycle sends it back to the same address, possibly altered.
- A block-transfer transaction sends multiple data units to adjacent addresses in a burst of consecutive bus cycles.

The VME design also allows for interrupts. A device can raise an interrupt on any of seven *interrupt level*s. The interrupt is acknowledged by a bus master. The bus master interrogates the interrupting device in an interrupt-acknowledge bus cycle, and the device returns an interrupt vector number.

In Silicon Graphics systems, it is always the Silicon Graphics VME controller that acknowledges interrupts. It passes the interrupt to one of the CPUs in the system.

VME Bus in Silicon Graphics Systems

The VME bus was designed as the system backplane for a workstation, supporting one or more CPU modules along with the memory and I/O modules they used. However, no Silicon Graphics computer uses the VME bus as the system backplane. In all Silicon Graphics computers, the main system bus that connects CPUs to memory is a proprietary bus design, with higher speed and sometimes wider data units than the VME bus provides. The VME bus is attached to the system as an I/O device.

This section provides an overview of the design of the VME bus in any Silicon Graphics system. It is sufficient background for most users of VME devices. For a more detailed look at the Challenge or Onyx implementation of VME, see “VME Hardware in Challenge and Onyx Systems” on page 313.

The VME Bus Controller

A VME bus controller is attached to the system bus to act as a bridge between the system bus and the VME bus. This arrangement is shown in Figure 13-1.

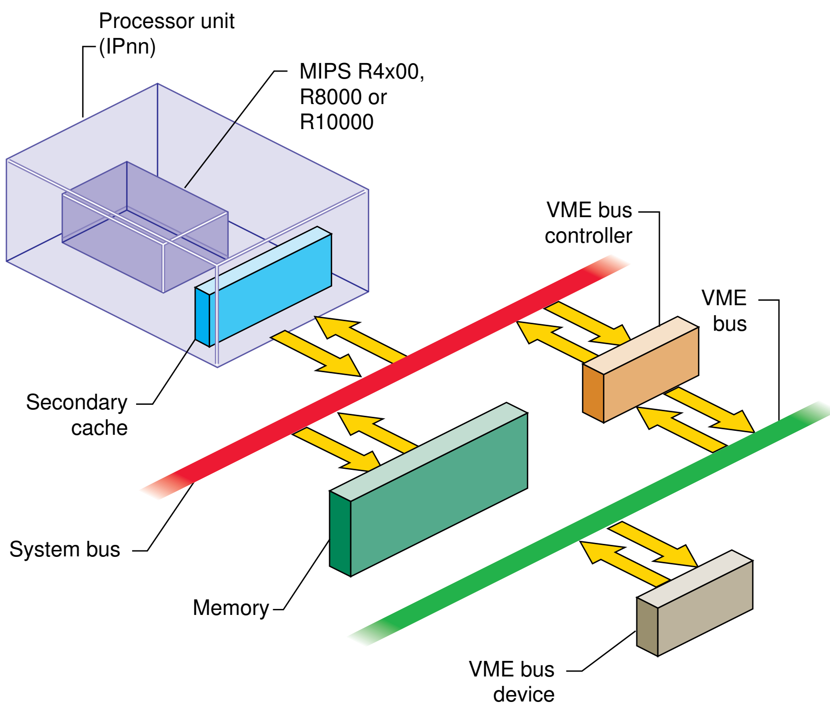

Figure 13-1 Relationship of VME Bus to System Bus

On the Silicon Graphics system bus, the VME bus controller acts as an I/O device. On the VME bus, the bus controller acts as a VME bus master.

The VME controller has several tasks. Its most important task is mapping; that is, translating some range of physical addresses in the Silicon Graphics system address space to a range of VME bus addresses. The VME controller performs a variety of other duties for different kinds of VME access.

VME PIO Operations

During programmed I/O (PIO) to the VME bus, software in the CPU loads or stores the contents of CPU registers to a device on the VME bus. The operation of a CPU load from a VME device register is as follows:

1. The CPU executes a load from a system physical address.
2. The system recognizes the physical address as one of its own.
3. The system translates the physical address into a VME bus address.
4. Acting as a VME bus master, the system starts a read cycle on the VME bus.
5. A slave device on the VME bus responds to the VME address and returns data.
6. The VME controller initiates a system bus cycle to return the data packet to the CPU, thus completing the load operation.

A store to a VME device is similar except that it performs a VME bus write, and no data is returned.

PIO input requires two system bus cycles—one to request the data and one to return it—separated by the cycle time of the VME bus. PIO output takes only one system bus cycle, and the VME bus write cycle run concurrently with the next system bus cycle. As a result, PIO input always takes at least twice as much time as PIO output. (For details, see “VME PIO Bandwidth” on page 66.)

VME DMA Operations

A VME device that can act as a bus master can perform DMA into memory. The general sequence of operations in this case is as follows:

1. Software in the Silicon Graphics CPU uses PIO to program the device registers of the VME device, instructing it to perform DMA to a certain VME bus address for a specified length of data.
2. The VME bus master initiates the first read, write, block-read, or block-write cycle on the VME bus.
3. The VME controller, responding as a slave device on the VME bus, recognizes the VME bus address as one that corresponds to a physical memory address in the system.

4. If the bus master is writing, the VME controller accepts the data and initiates a system bus cycle to write the data to system memory.

   If the bus master is reading, the VME controller uses a system bus cycle to read data from system memory, and returns the data to the bus master.

5. The bus master device continues to use the VME controller as a slave device until it has completed the DMA transfer.

During a DMA transaction, the VME bus controller operates independently of any CPU. CPUs in the system execute software concurrently with the data transfer. Since the system bus is faster than the VME bus, the data transfer takes place at the maximum data rate that the VME bus master can sustain.

Operation of the DMA Engine

In Silicon Graphics Crimson systems, the VME controller supports DMA by VME bus master devices. However, in the Challenge and Onyx lines, the VME controller contains an additional "DMA Engine" that can be programmed to perform DMA-type transfers between memory and a VME device that is a slave, not a bus master. The general course of operations in a DMA engine transfer is as follows:

1. The VME bus controller is programmed to perform a DMA transfer to a certain physical address for a specified amount of data from a specified device address in VME address space.
2. The VME bus controller, acting as the VME bus master, initiates a block read or block write to the specified device.
3. As the slave device responds to successive VME bus cycles, the VME bus controller transfers data to or from memory using the system bus.

The DMA engine transfers data independently of any CPU, and at the maximum rate the VME bus slave can sustain. In addition, the VME controller collects smaller data units into blocks of the full system bus width, minimizing the number of system bus cycles needed to transfer data. For both these reasons, DMA engine transfers are faster than PIO transfers for all but very short transfer lengths. (For details, see "DMA Engine Bandwidth" on page 73.)

VME Bus Addresses and System Addresses

Devices on the VME bus exist in one of the following address spaces:

- The 16-bit space (A16) contains numbers from 0x0000 to 0xffff.
- The 24-bit space (A24) contains numbers from 0x00 0000 to 0xff ffff.
- The 32-bit space (A32) uses numbers from 0x0000 0000 to 0xffff ffff.
- The 64-bit space (A64), defined in the revision D specification, uses 64-bit addresses.

The Silicon Graphics system bus also uses 32-bit or 64-bit numbers to address memory and other I/O devices on the system bus. In order to avoid conflicts between the meanings of address numbers, certain portions of the physical address space are reserved for VME use. The VME address spaces are mapped, that is, translated, into these ranges of physical addresses.

The translation is performed by the VME bus controller: It recognizes certain physical addresses on the system bus and translates them into VME bus addresses; and it recognizes certain VME bus addresses and translates them into physical addresses on the system bus.

Even with mapping, the entire A32 or A64 address space cannot be mapped into the physical address space. As a result, no Silicon Graphics system provides access to all of the VME address spaces. Only parts of the VME address spaces are available at any time. The mapping methods of the Silicon Graphics Crimson series differ from the methods of the later Challenge and Onyx series of machines.

User-Level and Kernel-Level Addressing

In a user-level program you can perform PIO and certain types of DMA operations (see Chapter 4, “User-Level Access to VME and EISA”), but you do not program these in terms of the physical addresses mapped to the VME bus. Instead, you call on the services of a kernel-level device driver to map a portion of VME address space into the address space of your process. The requested segment of VME space is mapped dynamically to a segment of your user-level address space—a segment that can differ from one run of the program to the next.

In a kernel-level device driver, you program PIO and DMA operations in terms of specific addresses in kernel space—memory addresses that are mapped to specified addresses in the VME bus address space. The mapping is either permanent, established by the system hardware, or dynamic, set up temporarily by a kernel function.

Note: The remainder of this chapter has direct meaning only for kernel-level drivers, which must deal with physical mappings of VME space.

PIO Addressing and DMA Addressing

The addressing needs of PIO access and DMA access are different.

PIO deals in small amounts of data, typically single bytes or words. PIO is directed to device registers that are identified with specific VME bus addresses. The association between a device register and its bus address is fixed, typically by setting jumpers or switches on the VME card.

DMA deals with extended segments of kilobytes or megabytes. The addresses used in DMA are not fixed in the device, but are programmed into it just before the data transfer begins. For example, a disk controller device can be programmed to read a certain sector and to write the sector data to a range of 512 consecutive bytes in the VME bus address space. The programming of the disk controller is done by storing numbers into its registers using PIO. While the registers respond only to fixed addresses that are configured into the board, the address to which the disk controller writes its sector data is just a number that is programmed into it each time a transfer is to start.

The key differences between addresses used by PIO and addresses used for DMA are these:

- PIO addresses are relatively few in number and cover small spans of data, while DMA addresses can span large ranges of data.
- PIO addresses are closely related to the hardware architecture of the device and are configured by hardware or firmware, while DMA addresses are simply parameters programmed into the device before each operation.

In Crimson systems, portions of the physical address space are permanently mapped to portions of the VME address space. PIO is performed to these designated, fixed physical addresses. Some DMA addresses are also assigned fixed mappings, but additional segments are assigned dynamically as needed.

In Challenge and Onyx systems, all VME mappings are dynamic, assigned as needed. Kernel functions are provided to create and release mappings between designated VME addresses and kernel addresses.

It is important to realize that the kernel functions provided for dynamic mapping in the Challenge or Onyx systems also work in the Crimson line. A device driver that uses these functions is portable to all systems supported by IRIX 6.2.

PIO Addressing in Challenge and Onyx Systems

The Challenge and Onyx systems and their Power versions support from one to five VME buses. It is impossible to fit adequate segments of five separate A16, A24, and A32 address spaces into fixed mappings in the 40-bit physical address space available in these systems.

The VME controller in Challenge and Onyx systems uses programmable mappings. The IRIX kernel can program the mapping of twelve separate 8 MB “windows” on VME address space on each bus (a total of 96 MB of mapped space per bus). The kernel sets up VME mappings by setting the base addresses of these windows as required. A kernel-level VME device driver asks for and uses PIO mappings through the functions documented in “Mapping PIO Addresses” on page 325.

Note: The same kernel functions can be called to set up mappings for PIO addresses in the Crimson series. It is not necessary to write code that depends on the fixed addresses documented in the following section.

PIO Addressing in the Crimson Series

Systems in the Silicon Graphics Crimson series can have one or two independent VME buses. The first bus is bus 0 and the second is bus 1. For PIO purposes, all or part of the VME A16, A24, and A32 address spaces from each bus are assigned to fixed mappings in system physical address space.

Note: When you read a device driver written specifically for the Crimson series you may see use of the addresses documented in this section. When writing a new driver, you are urged to avoid the use of these fixed addresses, and instead use the functions for dynamic assignment of PIO maps discussed under “PIO Addressing in Challenge and Onyx Systems” on page 306. The end result will be the same, but the driver will be portable to all systems.

Crimson Mapping of A16 Space

The A16 address space for each bus is mapped in full as shown in Table 13-1.

Table 13-1 VME A16 Addressing in Crimson Systems

VME A16 Address Range	Address Modifier	Kernel Address VME Bus 0	Kernel Address VME Bus 1
0x0000-0xFFFF	0x29 (normal data)	0xB7C0 0000–0xB7C0 FFFF	0xB780 0000–0xB780 FFFF
0x0000-0xFFFF	0x2D (supervisor data)	0xB7C1 0000–0xB7C1 FFFF	0xB781 0000–0xB781 FFFF

Crimson Mapping of A24 Space

The upper 8 MB of the A24 address space for supervisory and program data access is mapped as shown in Table 13-2.

Table 13-2 VME A24 Addressing in Crimson Systems

VME A24 Address Range	Address Modifier	Kernel Address VME Bus 0	Kernel Address VME Bus 1
0x80 0000–0xFF FFFF	0x39 (normal data)	0xB280 0000–0xB2FF FFFF	0xF080 0000–0xF0FF FFFF
0x80 0000–0xFF FFFF	0x3D (supervisor data)	0xB380 0000–0xB3FF FFFF	0xF180 0000–0xF1FF FFFF

Crimson Mapping of A32 Space

The Crimson architecture allows up to 256 MB of physical address space for mapping a segment of VME A32 addresses. When the system contains only one VME bus, a single 256 MB segment of the A32 space is mapped as shown in Table 13-3.

Table 13-3 VME A32 Addressing in Crimson Systems (One Bus)

VME A32 Address Range	Address Modifier	Kernel Address
0x1000 0000–0x1FFF FFFF	0x09 (extended normal data)	0xD000 0000–0xDFFF FFFF
0x1000 0000–0x1FFF FFFF	0x0D (extended supervisor data)	0xE000 0000–0xEFFF FFFF

When the system contains two buses, the available space is divided so that a segment of 128 MB from each bus is mapped, as shown in Table 13-4.

Table 13-4 VME A32 Addressing in Crimson Systems (Two Buses)

VME A32 Address Range	Address Modifier	Kernel Address VME Bus 0	Kernel Address VME Bus 1
0x1800 0000–0x1FFF FFFF	0x09 (extended normal data)	0xD800 0000–0xDFFF FFFF	0xD000 0000–0xD7FF FFFF
0x1800 0000–0x1FFF FFFF	0x0D (extended supervisor data)	0xE800 0000–0xEFFF FFFF	0xE000 0000–0xE800 0000

DMA Addressing

DMA is supported only for the A24 and A32 address spaces. DMA addresses are always assigned dynamically in the Challenge and Onyx systems.

The Crimson series supports two modes of DMA mapping: direct and mapped. In direct mode, the VME bus address is the same as the target physical address in memory. In mapped mode, a segment of VME bus addresses is mapped dynamically to a segment of kernel virtual addresses.

DMA Addressing in the Challenge and Onyx Systems

In order to establish a mapping for DMA use, a kernel-level driver should call the functions documented in “Mapping DMA Addresses” on page 330. These functions work in all systems supported by IRIX 6.2.

Direct DMA Addressing in the Crimson Series

When you examine a device driver written for the Crimson series you may find direct DMA addressing used. Under direct addressing, the target physical address of the buffer in memory is programmed into the VME bus master device as its target address in the VME address space. For example, a device driver could get the physical address of a page buffer in memory, and program that into a VME disk controller as the target address for a sector read or write.

Direct mapping is not recommended. It is hardware-dependent and nonportable. Although direct mapping is simple, it has the disadvantage that the target buffer must be in contiguous physical memory locations. When a buffer in kernel virtual memory spans two or more pages, the parts of the buffer are likely to be in noncontiguous page frames. In order to handle noncontiguous pages using direct mapping, the device driver must program a separate transfer for each page.

Mapped DMA Addressing in the Crimson Series

The Crimson series also supports DMA mapping. Using DMA mapping, the device driver calls a kernel function passing it a kernel virtual address and a length (which can exceed a page in size). The kernel function programs the VME bus controller with mapping information.

When a mapping is established, the device driver can use a kernel function to get the VME bus address that has been mapped to the first byte of the buffer. This address is programmed into the VME bus master as its target address.

A mapped target address in the Crimson always has a most significant bit of 1. When the VME bus controller, acting as a slave device, is told to read or write to an address with a most significant bit of 1, it recognizes a mapped address, and uses the mapping tables set up by the kernel to translate the VME bus address into a physical memory address.

Mapped DMA addressing has two important advantages. It takes care of the problem of discontiguous page frames—the VME bus master uses a contiguous range of VME addresses, and never knows that the VME bus controller is scattering or gathering data from different pages. And it is portable to all systems—the kernel functions work identically in Crimson and in Challenge and Onyx systems.

Configuring VME Devices

To install a VME device in a Silicon Graphics system, you need to configure the device itself to respond to PIO addresses in a supported range of VME bus addresses, and you need to inform IRIX of the device addresses.

Configuring Device Addresses

Normally a VME card can be programmed to use different VME addresses for PIO, based on jumper or switch settings on the card. The devices on a single VME bus must be configured to use unique addresses. Errors that are hard to diagnose can arise when multiple cards respond to the same bus address. Devices on different VME buses can use the same addresses.

Not all parts of each address space are accessible. The accessible parts are summarized in Table 13-5.

Table 13-5 Accessible VME Addresses in Each System

Address Space	Crimson Series	Challenge and Onyx Systems
A16	All	All
A24	0x80 0000–0xFF FFFF	0x80 0000–0xFF FFFF
A32	0x1000 0000–0x1FFF FFFF (one bus) 0x1800 0000–0x1FFF FFFF (2 buses)	0x0000 0000–0x7FFF FFFF (maximum of 96 MB in 8 MB units)

Within the accessible ranges, certain VME bus addresses are used by Silicon Graphics VME devices. You can find these addresses documented in the */var/sysgen/system/irix.sm* file. You must configure OEM devices to avoid the addresses used by Silicon Graphics devices that are installed on the same system.

Finally, on the Challenge and Onyx systems, take care to cluster PIO addresses in the A32 space so that they occupy at most a 96 MB span of addresses. The reasons are explained under “Fixed PIO Maps” on page 328.

Configuring the System Files

Inform IRIX and the device driver of the existence of a VME device by adding a VECTOR statement to a file in the directory */var/sysgen/system* (see "System Configuration Files" on page 39). The syntax of a VECTOR statement is documented in two places:

- The */var/sysgen/system/irix.sm* file itself contains descriptive comments on the syntax and meaning of the statement, as well as numerous examples.
- The system(4) reference page gives a more formal definition of the syntax.

In addition to the VECTOR statement, you may need to code an IPL statement.

Coding the VECTOR Statement

The important elements in a VECTOR line are as follows:

bustype	Specified as *VME* for VME devices. The VECTOR statement can be used for other types of buses as well.
module	The base name of the device driver for this device, as used in the */var/sysgen/master.d* database (see "Master Configuration Database" on page 38 and "How Names Are Used in Configuration" on page 232).
adapter	The number of the VME bus where the device is attached—0 or 1 in a Crimson system; the bus number in a Challenge or Onyx machine.
ipl	The interrupt level at which the device causes interrupts, from 0 to 7.
vector	An 8-bit value between 1 and 254 that the device returns during an interrupt acknowledge cycle.
ctlr	The "controller" number is simply an integer parameter that is passed to the device driver at boot time. It can be used for example to specify a logical unit number.
iospace, *iospace2*, *iospace3*	Each *iospace* group specifies the VME address space, the starting bus address, and the size of a segment of VME address space used by this device.
probe or *exprobe*	Specify a hardware test that can be applied at boot time to find out if the device exists.

Use the *probe* or *exprobe* parameter to program a test for the existence of the device at boot time. If the device does not respond (because it is offline or because it has been removed from the system), the *lboot* command will not invoke the device driver for this device.

The device driver specified by the *module* parameter is invoked at its *pfx***edtinit()** entry point, where it receives most of the other information specified in the VECTOR statement (see “Entry Point edtinit()” on page 144).

Omit the *vector* parameter in either of two cases: when the device does not cause interrupts, or when it supports a programmable interrupt vector (see “Allocating an Interrupt Vector Dynamically” on page 332).

Use the *iospace* parameters to pass in the exact VME bus addresses that correspond to this device, as configured in the device. Up to three address space ranges can be passed to the driver. This does not restrict the device—it can use other ranges of addresses, but the device driver has to deduce their addresses from other information. The device driver typically uses this data to set up PIO maps (see “Mapping PIO Addresses” on page 325).

Using the IPL Statement

In a Challenge or Onyx system only, you can direct VME interrupts to specific CPUs. This is done with the IPL statement, also written into a file in */var/sysgen/system*. The IPL statement, which like the VECTOR statement is documented in both the system(4) reference page and the */var/sysgen/system/irix.sm* file itself, has only two parameters:

level
: The VME interrupt level to be directed, 0 to 7 (the same value that is coded as *ipl*= in the VECTOR statement.

cpu
: The number of the CPU that should handle all VME interrupts at this level.

The purpose of the IPL statement is to send interrupts from specific devices to a specific CPU. There are two contradictory reasons for doing this:

- That CPU is dedicated to handling those interrupts with minimum latency
- Those interrupts would disrupt high-priority work being done on other CPUs if they were allowed to reach the other CPUs.

The IPL statement cannot direct interrupts from a specific device; it directs all interrupts that occur at the specified level.

VME Hardware in Challenge and Onyx Systems

The overview topic, “VME Bus in Silicon Graphics Systems” on page 300, provides sufficient orientation for most users of VME devices. However, if you are designing hardware or a high-performance device driver specifically for the Challenge and Onyx systems, the details in this topic are important.

Note: For information on physical cabinets, panels, slot numbering, cables and jumpers, and data about dimensions and airflow, refer to the Owner’s Guide manual for your machine. For example, see the *POWER CHALLENGE AND CHALLENGE XL Rackmount Owner’s Guide* (007-1735) for the physical layout and cabling of VME busses in the large Challenge systems.

VME Hardware Architecture

The VME bus interface circuitry for Challenge and Onyx systems resides on a mezzanine board called the VMEbus Channel Adapter Module (VCAM) board. One VCAM board is standard in every system and mounts directly on top of the IO4 board in the system card cage.

The IO4 board is the heart of the I/O subsystem. The IO4 board supplies the system with a basic set of I/O controllers and system boot and configuration devices such as serial and parallel ports, and Ethernet.

In addition, the IO4 board provides these interfaces:

- two Flat Cable Interconnects (FCIs) for connection to Card Cage 3 (CC3)
- two SCSI-2 cable connections
- two Ibus connections

A Challenge or Onyx system can contain multiple IO4 boards, which can operate in parallel. (Early versions of the IO4 have a possible hardware problem that is described in Appendix B, “Challenge DMA with Multiple IO4 Boards.”)

Main System Bus

The main set of buses in the Challenge and Onyx system architecture is the Everest address and data buses, Ebus for short. The Ebus provides a 256-bit data bus and a 40-bit address bus that can sustain a bandwidth of 1.2 GB per second.

The 256-bit data bus provides the data transfer capability to support a large number of high-performance RISC CPUs. The 40-bit address bus is also wide enough to support 16 GB of contiguous memory in addition to an 8 GB I/O address space.

Ibus

The 64-bit Ibus (also known as the HIO bus) is the main internal bus of the I/O subsystem and interfaces to the high-power Ebus through a group of bus adapters.The Ibus has a bandwidth of 320 MB per second that can sufficiently support a graphics subsystem, a VME64 bus, and as many as eight SCSI channels operating simultaneously.

Bus Interfacing

Communication with the VME and SCSI buses, the installed set or sets of graphics boards, and Ethernet takes place through the 64-bit Ibus. The Ibus interfaces to the main system bus, the 256-bit Ebus, through a set of interface control devices, an I address (IA) and four I data (ID). The ID ASICs latch the data, and the IA ASIC clocks the data from each ID to the Flat Cable Interface (FCI) through the F controller (or F chip).

Two FCI controllers (or F controllers) help handle the data transfers to and from an internal graphics board set (if present) and any VMEbus boards in optional CC3 applications. The SCSI-2 (S1) controller serves as an interface to the various SCSI-2 buses. The Everest peripheral controller (EPC) device manages the data movement to and from the Ethernet, a parallel port, and various types of on-board PROMs and RAM.

Maximum Latency

The worst-case delay for the start of a VME access, if all of the devices on the IO4 simulataneously request the IO channel for a 128 byte write and the VME adapter receives the grant last, the VME access start could be delayed for a total of about 2 microseconds. Only a VME read would suffer this delay; a VME write would not.

There is a another potential delay from an independent cause, which depends on the number of bus master (IO4 and CPU) boards on the system bus. If all the E-bus masters in a fairly large configuration hit the bus at once, a VME read or write could be held up by as much as 1 microsecond in a large system.

VME Bus Numbering

The Challenge and Onyx systems support up to five independent VME buses in a single system. The numbering of these buses is not sequential.

There is always a VME bus number 0. This is the bus connected to the system midplane. It is always connected by the primary IO4 board (the IO4 board attached to the highest slot on the system bus).

Bus numbers for other VME buses depend on the Ebus slot number where their IO4 is attached, and on the I/O adapter number of the VCAM card on the IO4. Each IO4 board supports adapter numbers from 1 to 7, but a VME bus can only be attached to adapter number 2, 3, 5, or 6. These four adapters are given VME index numbers of 0, 1, 2, and 3 respectively.

The bus number of a VME bus is given by E*4+A, where E is the Ebus slot of the IO4 card, and A is the index of the adapter on the IO4. A VME bus attached through adapter 3 on the IO4 in Ebus slot 13 is bus number 53, from the sum of adapter index 1 and 4 times the slot number.

VMEbus Channel Adapter Module (VCAM) Board

The VCAM board provides the interface between the Ebus and the VMEbus and manages the signal level conversion between the two buses. The VCAM also provides a pass-through connection that ties the graphics subsystem to the Ebus.

The VCAM can operate as either a master or a slave. It supports DMA-to-memory transactions on the Ebus and programmed I/O (PIO) operations from the system bus to addresses on the VMEbus. In addition, the VCAM provides virtual address translation capability and a DMA engine that increases the performance of non-DMA VME boards.

VMECC

The VMECC (VME Cache Controller) gate array is the major active device on the VCAM. The VMECC interfaces and translates host CPU operations to VMEbus operations (see Figure 13-2). The VMECC also decodes VMEbus operations to translate them to the host side.

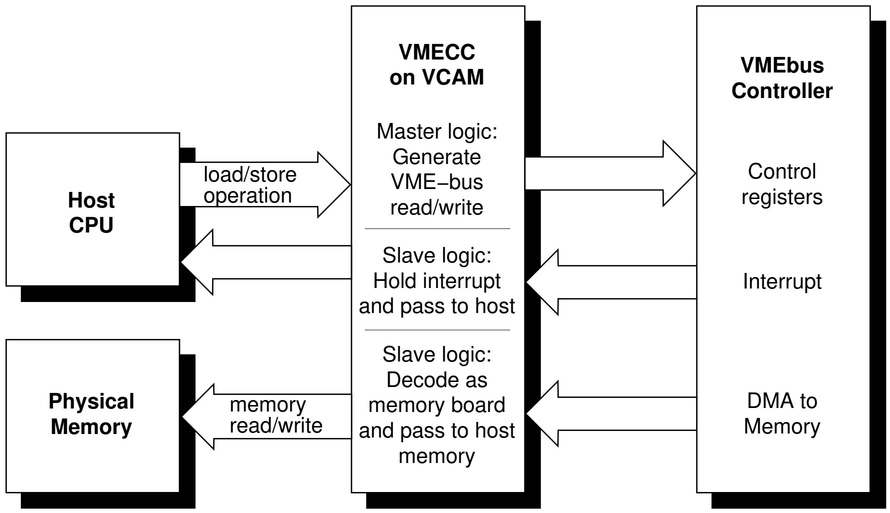

Figure 13-2 VMECC, the VMEbus Adapter

The VMECC provides the following features:

- an internal DMA engine to speed copies between physical memory and VME space (see "Operation of the DMA Engine" on page 303)
- a 16-entry deep PIO FIFO to smooth writing to the VME bus from the host CPUs
- a built-in VME interrupt handler and built-in VME bus arbiter
- an explicit internal delay register to aid in spacing PIOs for VME controller boards that cannot accept back-to-back operations
- support for issuing A16, A24, A32, and A64 addressing modes as a bus master during PIO
- support for single-item transfers (D8, D16, D32, and D64) as a bus master during PIO
- support for response as a slave to A24, A32, and A64 addressing modes to provide DMA access to the Ebus
- support for single-item transfers (D8, D16, and D32) as a slave during DMA access to the Ebus
- support for block item transfers (D8, D16, D32, and D64) as a slave during DMA access to the Ebus

The VMECC also provides four levels of VMEbus request grants, 0-3 (3 has the highest priority), for DMA arbitration. Do not confuse these bus request levels with the interrupt priority levels 1-7. Bus requests prioritize the use of the physical lines representing the bus and are normally set by means of jumpers on the interface board.

F Controller ASIC

Data transfers between VME controller boards and the host CPU(s) takes place through the VMECC on the VCAM board, then through a flat cable interface (FCI), and onto the F controller ASIC.

The F controller acts as an interface between the Ibus and the Flat Cable Interfaces (FCIs). This device is primarily composed of FIFO registers and synchronizers that provide protocol conversion and buffer transactions in both directions and translate 34-bit I/O addresses into 40-bit system addresses.

Two configurations of the F controller are used on the IO4 board; the difference between them is the instruction set they contain. One version is programmed with a set of instructions designed to communicate with the GFXCC (for graphics); the other version has instructions designed for the VMECC. All communication with the GFXCC or VMECC ICs is done over the FCI, where the F controller is always the slave.

Both versions of the F controller ASICs have I/O error-detection and handling capabilities. Data errors that occur on either the Ibus or the FCI are recorded by the F controller and sent to the VMECC or GFXCC.

ICs must report the error to the appropriate CPU and log any specific information about the operation in progress. FCI errors are recorded in the error status register. This register provides the status of the first error that occurred, as well as the cause of the most recent FCI reset.

VMEbus Interrupt Generation

The VME bus supports seven levels of prioritized interrupts, 1 through 7 (where 7 has the highest priority). The VMECC has a register associated with each level. When the system responds to the VMEbus interrupt, it services all devices identified in the interrupt vector register in order of their VMEbus priority (highest number first). The following list outlines how a VMEbus interrupt is generated:

1. A VME controller board asserts a VME interrupt on one of the IRQ levels.
2. The built-in interrupt handler in the VMECC chip checks if the interrupt level is presently enabled by an internal interrupt mask.
3. The interrupt handler in the VMECC issues a bussed IACK (interrupt acknowledge) response and acquires the vector from the device. The 3-bit response identifies one of the seven VME levels.
4. If multiple VME boards are present, the bussed IACK signal is sent to the first VME controller as an IACKIN. When the first controller is not the requesting master, it passes the IACKIN signal to the next board (in the daisy-chain) as IACKOUT.
5. The requesting board responds to IACKIN by issuing a DTACK* (data acknowledge signal), blocking the IACKOUT signal to the next board, and placing an 8-bit interrupt vector number on the data bus.
6. The VMECC latches the interrupt vector, and an interrupt signal is sent over the FCI interface to the F-chip and is queued awaiting completion of other F-chip tasks.
7. The F controller ASIC requests the I-bus and sends the interrupt to the IA chip.
8. The IA chip requests the Ebus and sends the interrupt over the Ebus to the CC chip on the IP19/IP21 board.
9. The CC chip interrupts the R4400/R8000, provided that the interrupt level is not masked.

The time for this to complete is normally less than 3 microseconds, but will be queued to await completion of other VME activities.

VME Interface Features and Restrictions

The Challenge and Onyx VME interface supports all protocols defined in Revision C of the VME specification plus the A64 and D64 modes defined in Revision D. The D64 mode allows DMA bandwidths of up to 60 MB. This bus also supports the following features:

- seven levels of prioritized processor interrupts
- 16-bit, 24-bit, and 32-bit data addresses and 64-bit memory addresses
- 16-bit and 32-bit accesses (and 64-bit accesses in MIPS III mode)
- 8-bit, 16-bit, 32-bit, and 64-bit data transfer
- DMA to and from main memory

DMA Multiple Address Mapping

In the Challenge and Onyx series, a DMA address from a VME controller goes through a two-level translation to generate an acceptable physical address. This requires two levels of mapping. The first level of mapping is done through the map RAM on the IO4 board. The second level is done through the map tables in system memory. This mapping is shown in Figure 13-3.

Note: The second level mapping requires system memory to be reserved for the mapping tables. The current limit on the number of pages that is allocated for map tables is 16 pages and the maximum memory allotted for the map tables is 64 KB. The R4400 provides a 4 KB page size for 16 pages (4 KB * 16 pages= 64 KB). The R8000 provides a 16 KB page size for 4pages (16 KB * 4 pages = 64 KB).

Each second-leel map table entry corresponds to 4 KB of physical memory. In addition, each second-level map table entry is 4 bytes. With 64 KB of mapping table, there are a total of 16 K entries translating a maximum of 64 MB of DMA mapping, setting a limit of 64 MB that can be mapped at any time for each VME bus. This does not set any limit on the amount of DMA that can be done by a board during its operation.

Referring to the top of Figure 13-3, bits 32 and 33 from the IBus address come from the VMECC. These two bits determine a unique VMEbus number for systems with multiple VME buses. Of the remaining 32 bits (31 to 0), 12 are reserved for an offset in physical memory, and the other 20 bits are used to select up to 2^{20} or 1 million pages into the main memory table. However, as stated earlier only 64 KB is allocated for map tables.

As shown in Figure 13-3, thirteen bits go to the static RAM table. Recall that two of the thirteen bits are from the VMECC to identify the VMEbus number. The static RAM table then generates a 29-bit identifier into the main memory table. These 29 bits select a table in the main memory table. An additional nine bits select an entry or element within the table. A 00 (two zeros) are appended to form a 40-bit address into the main memory table.

The main memory table then generates 28-bit address which is then appended to the 12-bit offset of the IBus to form the final 40-bit physical address.

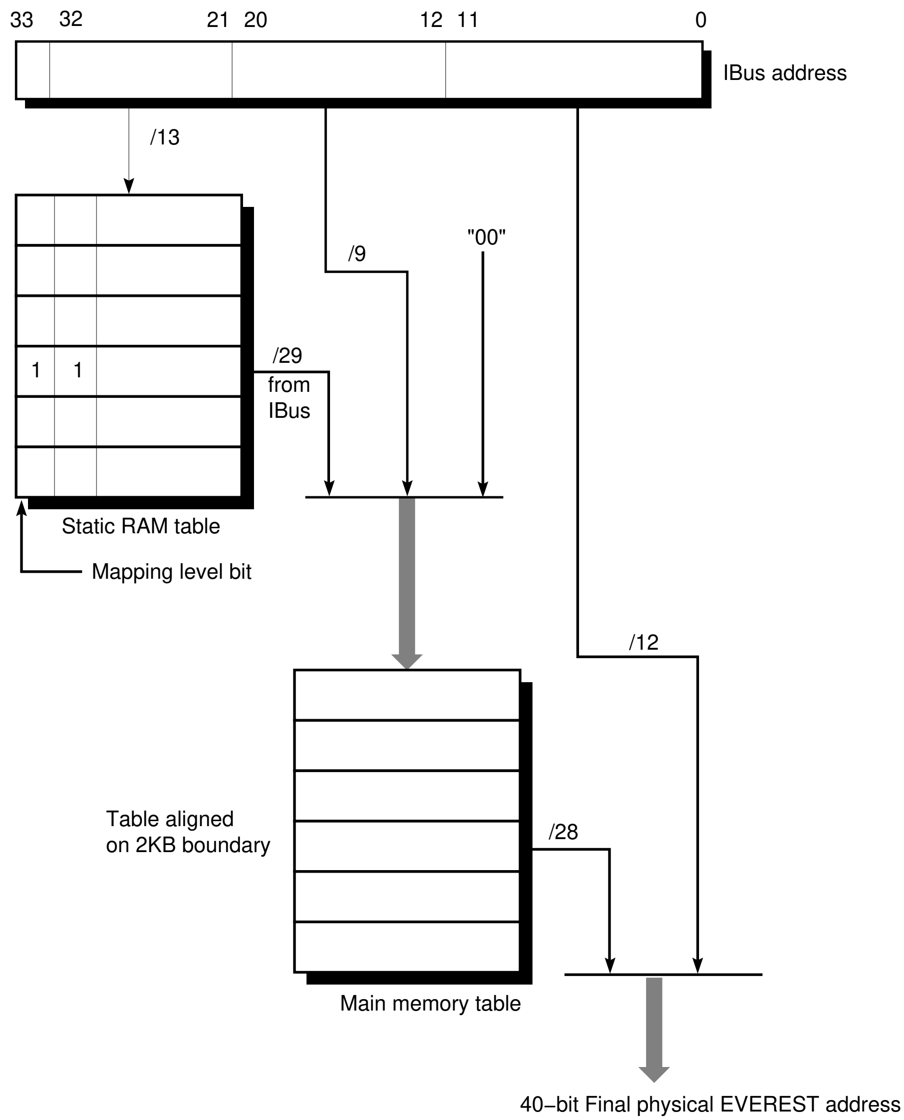

Figure 13-3 I/O Address to System Address Mapping

VME Interrupt Priority

Interrupts within the Challenge/Onyx architecture are managed by a set of interrupt vectors. An interrupt generated by an I/O device like a VME controller in turn generates an interrupt to the CPU on one of the previously assigned levels.

Each IRQ on each VME bus is assigned an internal interrupt level, and by default all these levels are at the same priority. If multiple interrupts arrive simultaneously within a single bus, for example IRQ 3 and IRQ 4 at once, priority is given to the higher-numbered IRQ.

All VME interrupts are made to go to CPU 0 (unless configured differently with IPL statements). This prevents one interrupt level from preempting the driver handling a different VME interrupt level.

VME Hardware Features and Restrictions

When designing an OEM hardware board to interface to the Challenge or Onyx VME bus, observe the following restrictions:

- Devices should require 8-bit interrupt vectors only. This is the only interrupt vector size that is supported by the VMECC or recognized by the IRIX kernel.
- Devices must not require UAT (unaligned transfer or tri-byte) access.
- Devices in slave mode must not require address modifiers other than Supervisory/Nonprivileged data access.
- While in master mode, a device must use only nonprivileged data access or nonprivileged block transfers.
- The Challenge or Onyx VME bus does not support VSBbus boards. In addition, there are no pins on the back of the VME backplane. This area is inaccessible for cables or boards.
- Metal face plates or front panels on VME boards may prevent the I/O door from properly closing and can possibly damage I/O bulkhead. (In some VME enclosures, a face plate supplies required EMI shielding. However, the Challenge chassis already provides sufficient shielding, so these plates are not necessary.)

Designing a VME Bus Master for Challenge and Onyx Systems

The following notes are related to the design of a VME bus master device to work with a machine in the Challenge/Onyx series. A VME bus master, when programmed using the functions described in "Mapping DMA Addresses" on page 330, can transfer data between itself and system memory.

The setup time at the start of a DMA transfer is as follows:

- First word of a read is delivered to the master in 3 to 8 microseconds.
- First word of a write is retrieved from the master in 1 to 3 microseconds.

The F controller does the mapping from A32 mode into system memory and automatically handles the crossing of page boundaries. The VME Bus Master is not required to make AS go high and then low on 4 KB boundaries. However, when using A64 addressing, the device may have to change the address on the 4 KB boundaries and cause a transition on AS low to high, and then back to low. This incurs new setup delays.

The important parts of the VME handshake cycle are diagrammed in Figure 13-4.

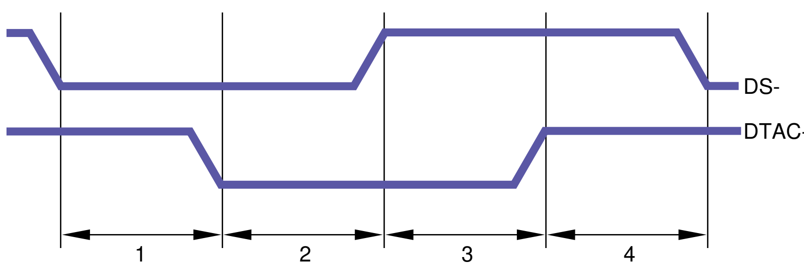

Figure 13-4 VMECC Contribution to VME Handshake Cycle Time

Intervals 1 and 3 represent the response latency in the bus slave (the VMECC). Intervals 2 and 4 represent the latency in the VME Bus Master. In the Challenge and Onyx systems,

- part 1 is approximately 40 nanoseconds
- part 3 is approximately 25 nanoseconds

The total contribution by the VMECC is approximately 65 nanoseconds. If the total of the four intervals can be held to 125 nanoseconds, the absolute peak transfer rate (in D64 mode) is 64 MB per second.

Note: Startup and latency numbers are averages and may occasionally be longer. The system design does not have any guaranteed latency.

The VME specification provides for a burst length of 265 bytes in D8, D16, and D32 modes, or 2 KB in D64. The burst length is counted in bytes, not transfer cycles.

Operating at this burst length, the duration of a single burst of 2 KB would be 256 transfers at 125 nanoseconds each, plus a startup of roughly 5 microseconds, giving a total of 37 microseconds per burst. Continuous, back-to-back bursts could achieve at most 55 MB per second.

However, the Challenge and Onyx VMECC uses a 20-bit burst counter allowing up to 2 MB in a burst of any data size. Suppose the bus master transfers 64 KB per burst, transferring 4-byte data words. The duration of a single burst would be 8,192 times 125 nanoseconds, plus 5 microseconds startup, or 1,029 microseconds per burst. Continuous bursts of this size achieve a data rate of 63.7 MB per second.

The use of long bursts violates the VME standard, and a bus master that depends on long bursts is likely not to work in other computers. If you decide to exceed the VME bus specifications, you should condition this feature with a field in a control register on the VME board, so that it can be disabled for use on other VME systems.