Physical and Virtual Memory

This chapter gives an overview of the management of physical and virtual memory in the MIPS® R4*x*00®, R8000™, and R10000™ processors. Access to physical devices is included in this topic, because device registers and bus attachments are accessed using physical memory addresses.

When you are designing a kernel-level driver, this information helps you understand the operation of the kernel functions that you call on, and the constraints on their operations. This information is only of academic interest if you intend to control a device from a user-level process.(See Chapter 3, "Device Control Software," for the difference between these two types of drivers.)

The following main topics are covered in this chapter.

- "Physical Address Space" on page 4 describes the range and meaning of address numbers on the hardware bus.
- "Addresses of Memory and Devices" on page 5 summarizes the hardware architecture by which the CPU accesses memory.
- "The 32-Bit Address Space" on page 14 describes the divisions of the 32-bit virtual address space and their uses.
- "The 64-Bit Address Space" on page 18 describes the divisions of the 64-bit virtual address space and their uses.
- "Device Driver Use of Memory" on page 24 describes the techniques and rules for how kernel-level device drivers allocate and use memory.

Note: This chapter tells only enough about memory access and cache management to explain the rules of the driver/kernel interface. For complete details on the MIPS hardware processors, see the hardware manuals listed under "Additional Reading" on page xxxix.

Physical Address Space

Physical addresses are used to select RAM, ROM, device registers, and bus attachments. Physical addresses start at 0 and can (in some systems) go as high as 2^{40}. This range includes 1.1e12 unique numbers, or 1,024 *gigabytes* (GB), or 1 *terabyte* (TB).

Software never uses physical addresses directly. Kernel-level software can access physical memory and devices using indirect addressing discussed later.

Device Addresses

The MIPS processor architecture has no I/O instructions. Certain ranges of physical addresses are reserved as device addresses. That is, an attempt to access a reserved address is decoded by the hardware as an access to a particular device or bus attachment.

Each Silicon Graphics computer model has a particular set of device addresses. The choice of device addresses is part of the architecture of the whole computer system; it is not designed into the processor chip.

For example, the EISA bus attachment in the Indigo^{2}™ responds to physical memory addresses beginning at 0x000A 0000. (The EISA bus is discussed in detail in Part VII, “EISA Drivers.”) To see some actual device addresses, examine VECTOR statements in files in */var/sysgen/system*.

In some cases, the relationship between device addresses and physical addresses can be changed dynamically. For example, in the Silicon Graphics Challenge and Onyx™ systems, different 8 MB portions of the VME bus address space can be mapped dynamically to different blocks of physical addresses. (VME bus details are in Part IV, “VME Device Drivers.”)

Memory Addresses

Some physical addresses are decoded to select memory hardware. Physical memory includes ROM as well as RAM. Each block of physical memory has a range of physical addresses. The physical addresses where RAM or ROM can be found depend on the particular computer system.

Physical memory does not necessarily occupy sequential addresses. There can be (and often are) gaps, ranges of physical addresses that do not relate to either memory or devices, between ROM addresses and RAM addresses. In most systems, all RAM is given a single sequential span of physical addresses. However, this is not a requirement. Blocks of RAM addresses can also be separated by gaps that are not populated with memory. Since all software uses virtual addresses, software always sees a sequential range of addresses without gaps.

Addresses of Memory and Devices

Each Silicon Graphics computer system has one or more CPU modules. The CPU reads memory or a device by placing an address on a system bus, and receiving data back from the addressed memory or device. Access to memory can pass through multiple levels of cache.

CPU Modules

A CPU is a hardware module containing a MIPS processor chip such as the R8000, together with system interface chips and possibly a secondary cache. Silicon Graphics CPU modules have model designation of the form IP*nn*; for example, the IP22 module is used in the Indy™ workstation. The CPU modules supported by IRIX 6.2 are listed in Table 1-1.

Table 1-1 CPU Modules and System Names

Module	MIPS Processor	System Families
IP17	R4000	Crimson™
IP19	R4*x*00	Challenge (other than S model), Onyx
IP20	R4*x*00	Indigo®

Table 1-1 (continued) CPU Modules and System Names

Module	MIPS Processor	System Families
IP21	R8000	POWER Challenge™, POWER Onyx™
IP22	R4*x*00	Indigo^2, Indy, Challenge S
IP25	R10000	POWER Challenge R10000
IP26	R8000	POWER Indigo^2™

Modules with the same IP designation can be ordered in a variety of clock speeds, and they can differ in other ways. Also, the choice of graphics hardware is independent of the CPU model. However, all these CPUs are identical as seen from software.

Interrogating the CPU Type

At the interactive command line, you can determine which CPU module a system uses with the command

```
hinv -c processor
```

Within a shell script, it is more convenient to process the terse output of

```
uname -m
```

(See the uname(1) and hinv(1) reference pages.)

Within a program, you can get the CPU model using the **getinvent()** function. For an example, see “Testing the Inventory In Software” on page 30.

CPU Access to Memory

The CPU generates the address of data that it needs—the address of an instruction to fetch, or the address of an operand of an instruction. It requests the data through a mechanism that is depicted in simplified form in Figure 1-1.

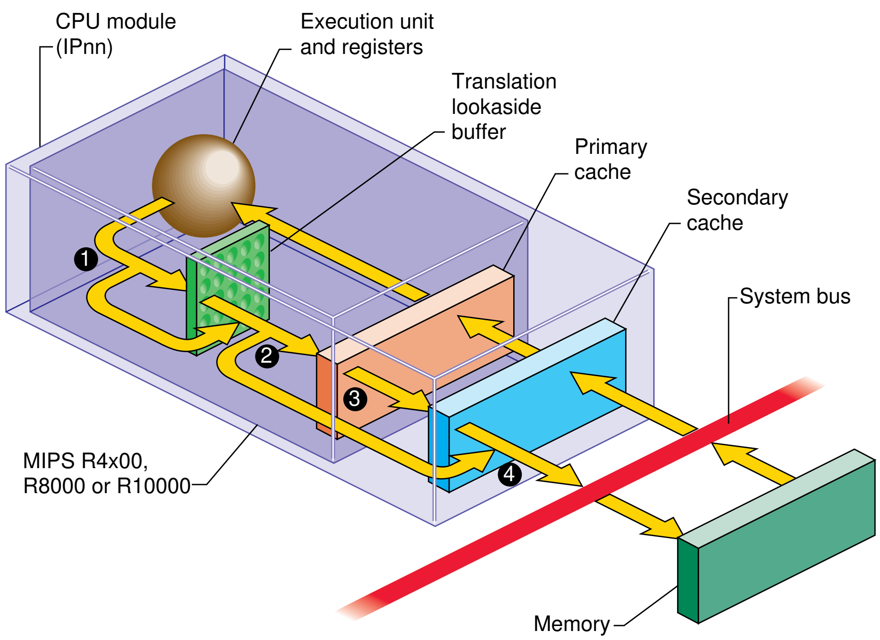

Figure 1-1 CPU Access to Memory

1. The address of the needed data is formed in the processor execution or instruction-fetch unit. Most addresses are then mapped from virtual to real through the Translation Lookaside Buffer (TLB). Certain ranges of addresses are not mapped, and bypass the TLB.

2. Most addresses are presented to the *primary cache*, the cache in the processor chip. If a copy of the data with that address is found, it is returned immediately. Certain address ranges are never cached; these addresses pass directly to the bus.

3. When the primary cache does not contain the data, the address is presented to the secondary cache. If it contains a copy of the data, the data is returned immediately. The size and the architecture of the secondary cache differ from one CPU model to another, and some CPUs do not have a secondary cache.

4. The address is placed on the system bus. The memory module that recognizes the address places the data on the bus.

Processor Operating Modes

The MIPS processor under IRIX operates in one of two modes: kernel and user. The processor enters the more privileged kernel mode when an interrupt, a system instruction, or an exception occurs. It returns to user mode only with a “Return from Exception” instruction.

Certain instructions cannot be executed in user mode. Certain segments of memory can be accessed only in kernel mode, and other segments only in user mode.

Virtual Address Mapping

The MIPS processor contains an array of Translation Lookaside Buffer (TLB) entries that map, or translate, virtual addresses to physical ones. Most memory accesses are first mapped by reference to the TLB. This permits the IRIX kernel to implement *virtual memory* for user processes, and permits it to relocate parts of the kernel itself. The translation scheme is summarized in the following sections and covered in detail in the hardware manuals listed under “Additional Reading” on page xxxix.

TLB Misses and TLB Sizes

Each TLB entry describes a segment of memory containing two adjacent *pages*. When the input address falls in a page described by a TLB entry, the TLB supplies the physical memory address for that page. The translated address, now physical instead of virtual, is passed on to the cache, as shown in Figure 1-1 on page 7.

When the input address is not covered by any active TLB entry, the MIPS processor takes a “TLB miss” interrupt to an IRIX kernel routine. The kernel routine inspects the address. When the address has a valid translation to some page in the address space, the kernel loads a TLB entry to describe that page, and restarts the instruction.

The size of the TLB is important for performance. The size of the TLB in different processors is shown in Table 1-2.

Table 1-2 Number of TLB Entries by Processor Type

Processor Type	Number of TBL Entries
R3000 (not supported by IRIX 6.2)	64
R4x00	96
R8000	384
R10000	128

Address Space Creation

There are not sufficient TLB entries to describe all the address space of every process. The IRIX kernel creates a page table for each process, containing one entry for each virtual memory page in the address space of that process. Whenever an executing program refers to an address for which there is no current TLB entry, the processor traps to the handler for the TLB miss exception. The exception handler loads one TLB entry from the appropriate page table entry of the current process, in order to describe the needed virtual address. Then it resumes execution with the failed instruction.

The kernel maintains a page table in kernel memory for each process, and a page table for the kernel virtual address space as well. In order to extend a virtual address space, the kernel takes the following two steps.

- It allocates unused page table entries to describe the needed pages. This defines the virtual addresses the pages will have.
- It allocates page frames in memory to contain the pages themselves, and puts their physical addresses in the page table entries.

Address Exceptions

When the CPU requests an invalid address—because the processor is in the wrong mode, or an address does not translate to a valid location in the address space, or an address refers to hardware that does not exist in the system—an addressing exception occurs. The processor traps to a particular address in the kernel.

An addressing exception can also be detected while handling a TLB miss—if there is no page table entry assigned for the desired address, that address is not part of the address space of the processs.

When a user-mode process caused the addressing exception, the kernel sends the process a SIGSEGV (see the signal(5) reference page), usually causing a segmentation fault. When kernel-level code such as a device driver caused the addressing exception, the kernel executes a “panic,” taking a crash dump and shutting down the system.

CPU Access to Device Registers

The CPU accesses a device register using the mechanism illustrated in Figure 1-2. Access to device registers is always uncached. It is not affected by considerations of cache coherency in any system (see “Cache Use and Cache Coherency” on page 13).

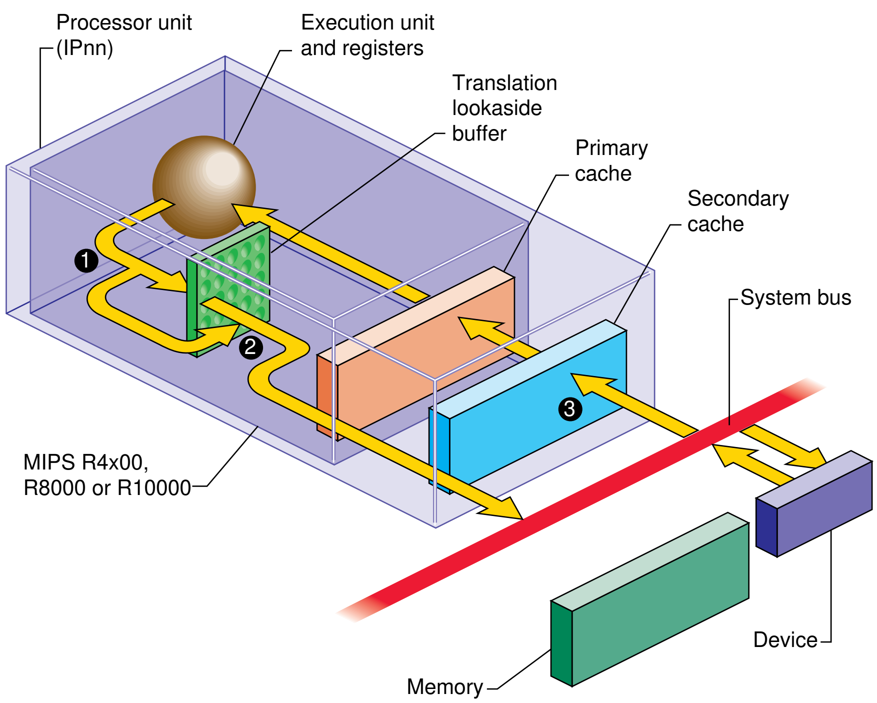

Figure 1-2 CPU Access to Device Registers

1. The address of the device is formed in the Execution unit. It may or may not be an address that is mapped by the TLB.
2. A device address, after mapping if necessary, always falls in one of the ranges that is not cached, so it passes directly to the system bus.
3. The device or bus attachment recognizes its physical address and responds with data.

Direct Memory Access

Some devices can perform *direct memory access (DMA)*. In order to read or write a sequence of memory addresses, the device has to be told of the proper physical address range to use. This is done by storing target address numbers into device registers from the CPU. When the device's DMA address registers are loaded, it can access memory through the system bus, as shown in Figure 1-3.

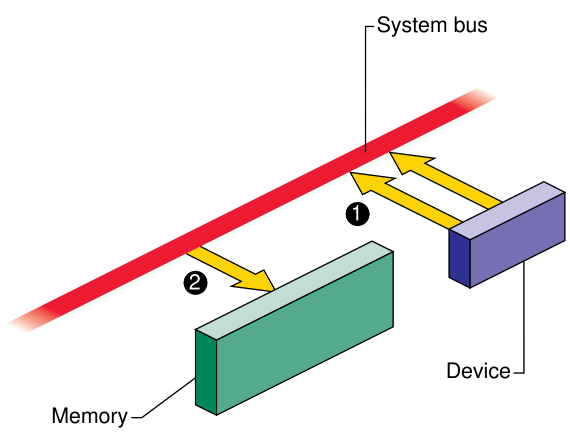

Figure 1-3 Device Access to Memory

1. The device places the next physical address, and data, on the system bus.
2. The memory module stores the data.

When a device is programmed with an invalid physical address, the result is a bus error interrupt.

Bus Virtual Addresses

Figure 1-3 is too simple for some devices that are attached through a bus adapter. A bus adapter connects a bus of a different type to the system bus, as shown in Figure 1-4.

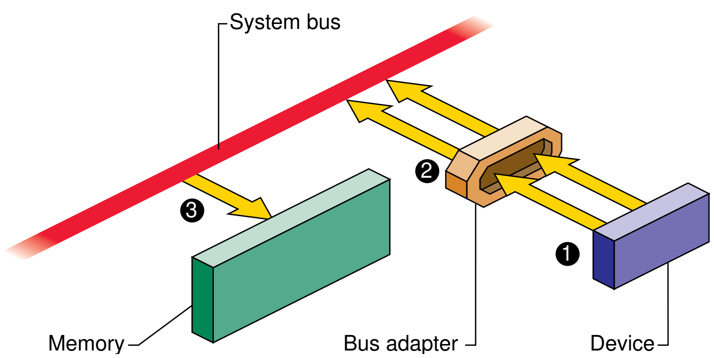

Figure 1-4 Device Access Through a Bus Adapter

For example, the VME adapter connects a VME bus to the system bus. Multiple VME devices can be plugged into the VME bus, and can use the VME bus to read and write. The VME bus adapter translates the VME bus protocol into the system bus protocol. (For details on the VME bus adapter, see Chapter 13, “VME Device Attachment.”)

Another example of a bus adapter is the EISA bus adapter that connects EISA devices to a GIO bus in an Indigo^2 workstation. (For details, see Chapter 17, “EISA Device Drivers.”)

One task of a bus adapter is to translate between the physical addresses used on the system bus, and the addressing scheme used within the proprietary bus. Translation is most visible with VME bus devices. The VME bus protocol defines several different address space conventions: A16, 16-bit addresses; A32, 32-bit addresses; and so on. A device on the VME bus uses addresses of this form. But VME addresses have no direct relationship to the physical addresses used on the system bus.

The bus adapter translates the addresses used on the proprietary bus to corresponding addresses on the system bus. Considering Figure 1-4, the operation of a DMA device is as follows:

1. The device places a bus virtual address and data on the proprietary (EISA or VME) bus.
2. The bus adapter translates the address to a meaningful physical address, and places that address and the data on the system bus.
3. The memory modules stores the data.

The translation of bus virtual to physical addresses is fixed in some systems. For example, the device in EISA slot 1 on an Indigo^2 is always translated to physical addresses 0x0001 0000 through 0x0001 FFFF.

In most systems, however, the bus translation is programmed by the IRIX kernel to suit the devices that are configured into the system. For example, the VME bus adapter in a Challenge or Onyx system can be programmed to place 15 different “windows” of VME address space at different locations in physical address space. There is no fixed relation between a given VME bus address and a physical address.

Cache Use and Cache Coherency

The primary and secondary caches shown in Figure 1-1 on page 7 are essential to CPU performance. There is an order of magnitude difference in the speed of access between cache memory and main memory. Execution speed remains high only as long as a very high proportion of memory accesses are satisfied from the primary or secondary cache.

The use of caches means that there are often multiple copies of data: a copy in main memory, a copy in the secondary cache (when one is used) and a copy in the primary cache. Moreover, a multiprocessor system has multiple CPU modules like the one shown, and there can be copies of the same data in the cache of *each* CPU.

The problem of *cache coherency* is to ensure that all cache copies of data are true reflections of the data in main memory. Different Silicon Graphics systems use different hardware designs to achieve cache coherency.

In most cases, cache coherence is achieved by the hardware, without any effect on software. In a few cases, specialized software, such as a kernel-level device driver, must take specific steps to maintain cache coherency.

Cache Coherency in Multiprocessors

Multiprocessor systems have more complex cache coherency protection because it is possible to have data in multiple caches. In a multiprocessor system, the hardware ensures that cache coherency is maintained under all conditions, including DMA input and output, without action by the software. However, in some systems the cache coherency hardware works correctly only when a DMA buffer is aligned on a cache-line-sized boundary. You ensure this by using the KM_CACHEALIGN flag when allocating buffer space with **kmem_alloc()** (see the kmem_alloc(D3) reference page).

Note: In one specific hardware configuration of Challenge and Onyx systems, a hardware problem can cause cache coherency errors that a device driver must deal with; see Appendix B, "Challenge DMA with Multiple IO4 Boards."

Cache Coherency in Uniprocessors

In some uniprocessor systems, it is possible for the CPU cache to have newer information than appears in memory. This is a problem only when a device is going to perform DMA. In these systems, a device driver calls a kernel function to ensure that all cached data has been written to memory prior to DMA output (see the dki_cache_wb(D3) reference page). The device driver calls a kernel function to ensure that the CPU receives the latest data following a DMA input (see the dki_cache_inval(D3) reference page). In a multiprocessor these functions do nothing, but it is always safe to call them.

The 32-Bit Address Space

The MIPS processors can operate in one of two address modes: 32-bit and 64-bit. The choice of address mode is independent of other features of the instruction set architecture such as the number of available registers and the precision of integer arithmetic (for example, programs compiled to the n32 binary interface use 32-bit addresses but 64-bit integers). The implications for user programs are documented in manuals listed under "Additional Reading" on page xxxix.

The addressing mode can be switched dynamically; for example, the IRIX kernel can operate with 64-bit addresses, but the kernel can switch to 32-bit address when it dispatches a user program that was compiled for that mode. The 32-bit address space is the range of all addresses that can be used when in 32-bit mode. This space is discussed first because it is simpler and more familiar than the 64-bit space.

Segments of the 32-bit Address Space

When operating in 32-bit mode, the MIPS architecture uses addresses that are 32-bit unsigned integers from 0x0000 0000 to 0xFFFF FFFF. However, this address space is not uniform. The MIPS hardware divides it into segments, and treats each segment differently. The ranges are shown graphically in Figure 1-5.

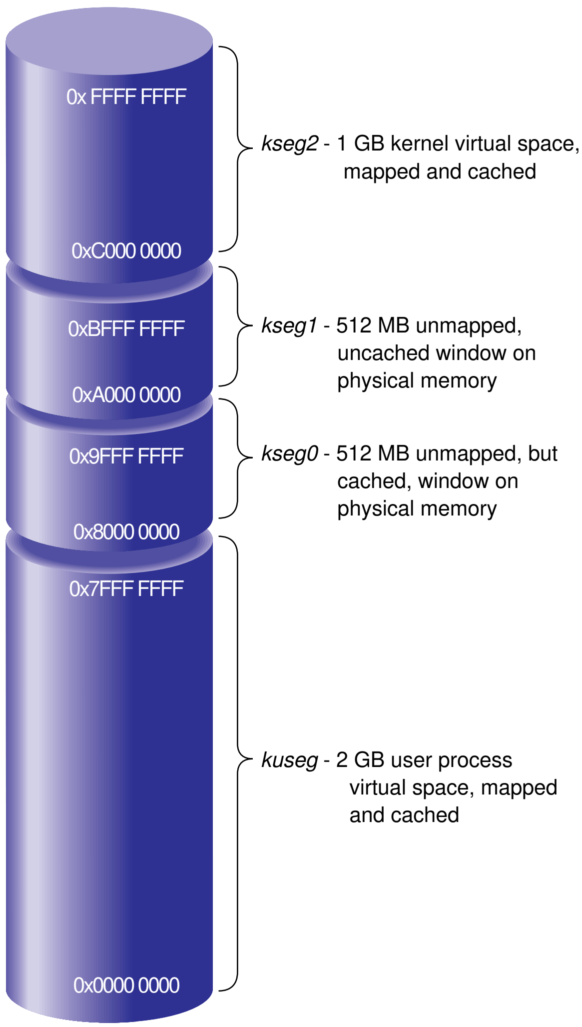

Figure 1-5 The 32-Bit Address Space

The address segments differ in three characteristics:

- whether access to an address is mapped; that is, passed through the translation lookaside buffer (TLB)
- whether an address can be accessed when the CPU is operating in user mode or in kernel mode
- whether access to an address is cached; that is, looked up in the primary and secondary caches before it is sent to main memory

Virtual Address Mapping

In the mapped segments, each 32-bit address value is treated as shown in Figure 1-6.

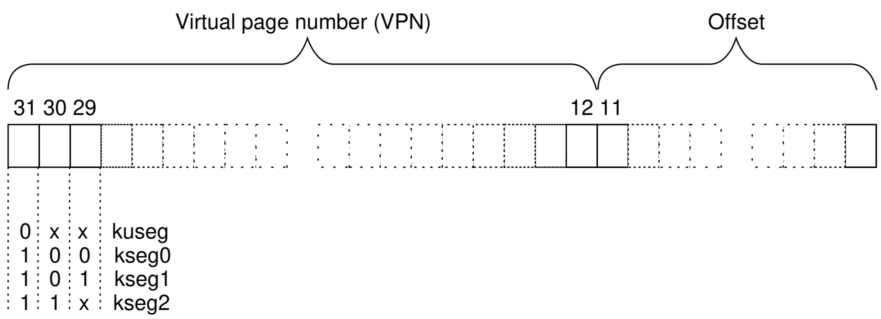

Figure 1-6 MIPS 32-Bit Virtual Address Format

The three most significant bits of the address choose the segment among those drawn in Figure 1-5. When bit 31 is 0, bits 30:12 select a *virtual page number* (VPN) from 2^{19} possible pages in the address space of the current user process. When bits 31:30 are 11, bits 29:12 select a VPN from 2^{18} possible pages in the kernel virtual address space.

User Process Space—kuseg

The total 32-bit address space is divided in half. Addresses with a most significant bit of 0 constitute the 2 GB user process space. When executing in user mode, only addresses in *kuseg* are valid; an attempt to use an address with bit 31=1 causes an addressing exception.

Access to *kuseg* is always mapped through the TLB. The kernel creates a unique address space for each user process. Of the 2^{19} possible pages in an address space, most are typically unassigned—few processes ever occupy more than a fraction of *kuseg*—and many are shared pages of program text from dynamic shared objects (DSOs) that are mapped into the address space of every process that needs them.

Kernel Virtual Space—kseg2

When bits 31:30 are 11, access is to kernel virtual memory. Only code that is part of the kernel can access this space. References to this space are translated through the TLB. The kernel uses the TLB to map kernel pages in memory as required, possibly in noncontiguous locations. Although pages in kernel space are mapped, they are always associated with real memory. Kernel memory is never paged to secondary storage.

This is the space in which the IRIX kernel allocates such objects as stacks, user page tables, and per-process data that must be accessible on context switches. This area contains automatic variables declared by loadable device drivers. It is the space in which kernel-level device drivers allocate memory. Since kernel space is mapped, addresses in *kseg2* that are apparently contiguous need not be contiguous in physical memory. However, a device driver can can allocate space that is both logically and physically contiguous, when that is required (see for example the kmem_alloc(D3) reference page).

Cached Physical Memory—kseg0

When address bits 31:29 contain 100, access is directed to physical memory through the cache. If the addressed location is not in the cache, bits 28:0 are placed on the system bus as a physical memory address, and the data presented by memory or a device is returned.

Since only 29 bits are available for mapping physical memory, only 512 MB of physical memory space can be accessed through this segment in 32-bit mode (some of which must be reserved for device addressing). It is possible to gain cached access to wider physical addresses by mapping through the TLB into *kseg2*, but systems that need access to more physical memory typically run in 64-bit mode (see "Cache-Controlled Physical Memory—xkphys" on page 22).

Kseg0 contains the exception address to which the MIPS processor branches it when it detects an exception such as an addressing exception or TLB miss.

Uncached Physical Memory—kseg1

When address bits 31:29 contain 101, access is directly to physical memory, bypassing the cache. Bits 28:0 are placed on the system bus for memory or device transfer.

The kernel refers to *kseg1* when interacting with I/O devices because loads or stores from device registers should not pass through cache memory. The kernel also uses *kseg1* when operating on certain data structures that might be *volatile*. Kernel-level device drivers sometimes need to write to uncached memory, and must take special precautions when doing so (see "Uncached Memory Access in the IP26 CPU" on page 27).

Portions of *kseg0* or *kseg1* can be mapped into *kuseg* by the **mmap()** function. This is covered at more length under "Memory Use in User-Level Drivers" on page 25.

The 64-Bit Address Space

The 64-bit mode is an upward extension of 32-bit mode. All MIPS processors from the R4000 on support 64-bit mode. However, this mode was not used in Silicon Graphics software until IRIX 6.0 was released.

Segments of the 64-Bit Address Space

When operating in 64-bit mode, the MIPS architecture uses addresses that are 64-bit unsigned integers from 0x0000 0000 0000 0000 to 0xFFFF FFFF FFFF FFFF. This is an immense span of numbers—if it were drawn to a scale of 1 millimeter per terabyte, the drawing would be 16.8 kilometers long (just over 10 miles).

The MIPS hardware divides the address space into segments based on the most significant bits, and treats each segment differently. The ranges are shown graphically in Figure 1-7. These major segments define only a fraction of the 64-bit space. Most of the possible addresses are undefined and cause an addressing exception (segmentation fault) if used.

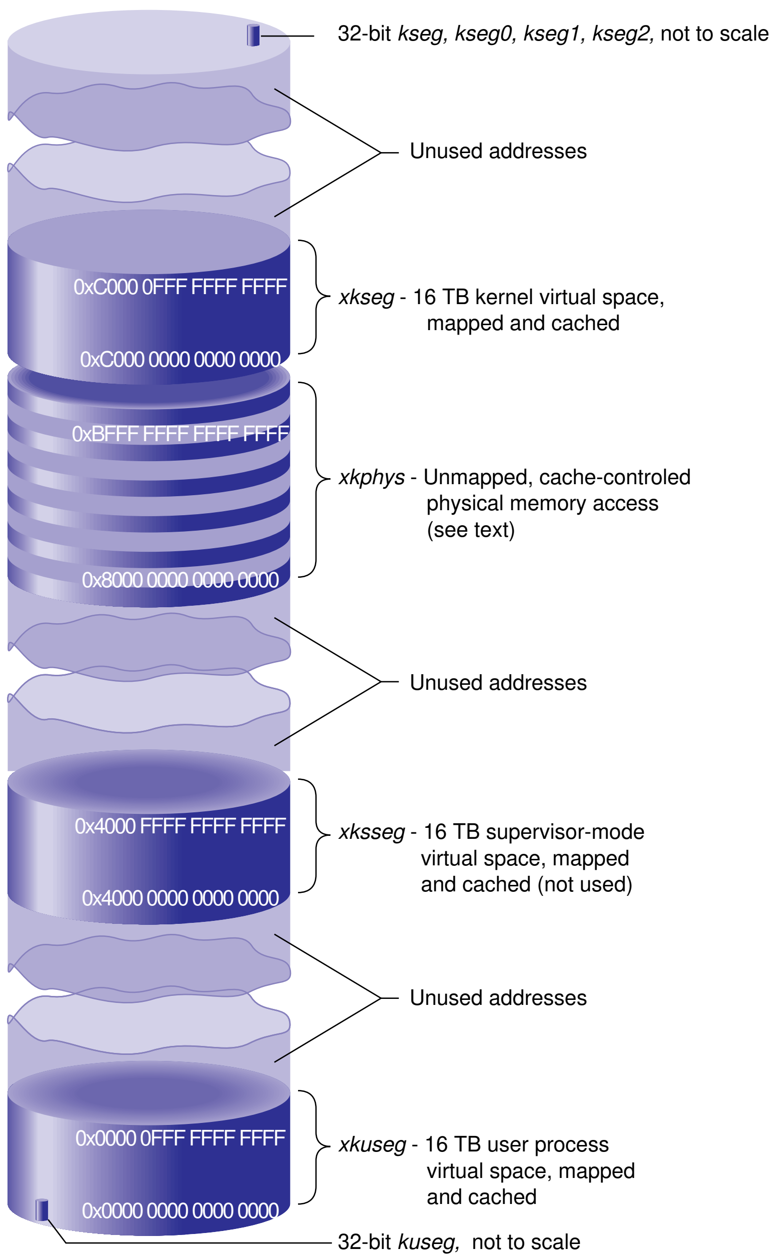

Figure 1-7 Main Parts of the 64-Bit Address Space

As in the 32-bit space, these major segments differ in three characteristics:

- whether access to an address is mapped; that is, passed through the translation lookaside buffer (TLB)
- whether an address can be accessed when the CPU is operating in user mode or in kernel mode.
- whether access to an address is cached; that is, looked up in the primary and secondary caches before it is sent to main memory

Compatibility of 32-Bit and 64-Bit Spaces

The MIPS-3 instruction set (which is in use when the processor is in 64-bit mode) is designed so that when a 32-bit instruction is used to generate or to load an address, the 32-bit operand is automatically sign-extended to fill the high-order 32 bits.

As a result, any 32-bit address that falls in the user segment *kuseg*, and which must have a sign bit of 0, is extended to a 64-bit integer with 32 high-order 0 bits. This automatically places the 32-bit *kuseg* in the bottom of the 64-bit *xkuseg*, as shown in Figure 1-7.

A 32-bit kernel address, which must have a sign bit of 1, is automatically extended to a 64-bit integer with 32 high-order 1 bits. This places all kernel segments shown in Figure 1-5 at the extreme top of the 64-bit address space. However, these 32-bit kernel spaces are not used by a kernel operating in 64-bit mode.

Virtual Address Mapping

In the mapped segments, each 64-bit address value is treated as shown in Figure 1-8.

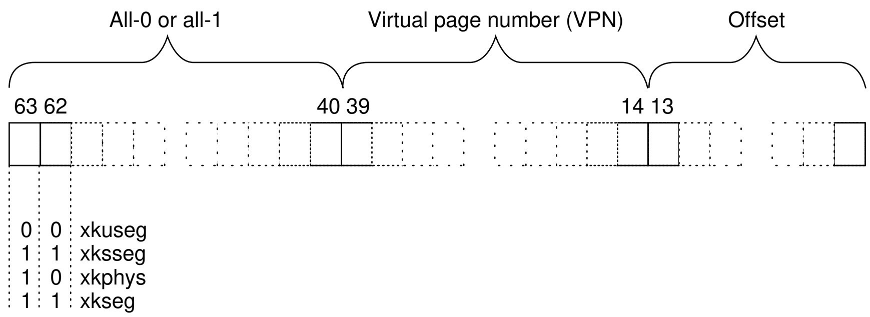

Figure 1-8 MIPS 64-Bit Virtual Address Format

The two most significant bits select the major segment (compare these to the address boundaries in Figure 1-7). Bits 61:40 must all be 0. (In principle, references to 32-bit kernel segments would have bits 61:40 all 1, but these segments are not used in 64-bit mode.)

The size of a page of virtual memory is a compile-time parameter when the kernel is created. In IRIX 6.2, the page size in a 32-bit kernel is 4 KB and in a 64-bit kernel is 16 KB. (Either size could change in later releases, so always determine it dynamically. In a user-level program, call the **getpagesize()** function (see the getpagesize(2) reference page). In a kernel-level driver, use the **ptob()** kernel function (see the ptob(D3) reference page) or the constant NBPP declared in *sys/immu.h*.)

When the page size is 16 KB, bits 13:0 of the address represent the offset within the page, and bits 39:14 select a VPN from the 2^{26}, or 64 M, pages in the virtual segment..

User Process Space—xkuseg

The first 16 TB of the address space are devoted to user process space. Access to *xkuseg* is always mapped through the TLB. The kernel creates a unique address space for each user process. Of the 2^{26} possible pages in a process's address space, most are typically unassigned, and many are shared pages of program text from dynamic shared objects (DSOs) that are mapped into the address space of every process that needs them.

Supervisor Mode Space—xksseg

The MIPS architecture permits three modes of operation: user, kernel, and supervisor. When operating in kernel or supervisor mode, the 2 TB space beginning at 0x4000 0000 0000 0000 is accessible. IRIX does not employ the supervisor mode, and does not use *xksseg*. If *xksseg* were used, it would be mapped and cached.

Kernel Virtual Space—xkseg

When bits 63:62 are 11, access is to kernel virtual memory. Only code that is part of the kernel can access this space, a 2 TB segment starting at 0xC000 0000 0000 0000. References to this space are translated through the TLB, and cached. The kernel uses the TLB to map kernel pages in memory as required, possibly in noncontiguous locations. Although pages in kernel space are mapped, they are always associated with real memory. Kernel pages are never paged to secondary storage.

This is the space in which the IRIX kernel allocates such objects as stacks, per-process data that must be accessible on context switches, and user page tables. This area contains automatic variables declared by loadable device drivers. It is the space in which kernel-level device drivers allocate memory. Since kernel space is mapped, addresses in *kseg2* that are apparently contiguous need not be contiguous in physical memory. However, a device driver can can allocate space that is both logically and physically contiguous, when that is required (see for example the kmem_alloc(D3) reference page).

Cache-Controlled Physical Memory—xkphys

One-quarter of the 64-bit address space—all addresses with bits 63:62 containing 10—are devoted to special access to the 1 TB physical address space. In 64-bit mode this space replaces the *kseg0* and *kseg1* spaces used in 32-bit mode. Addresses in this space are interpreted as shown in Figure 1-9.

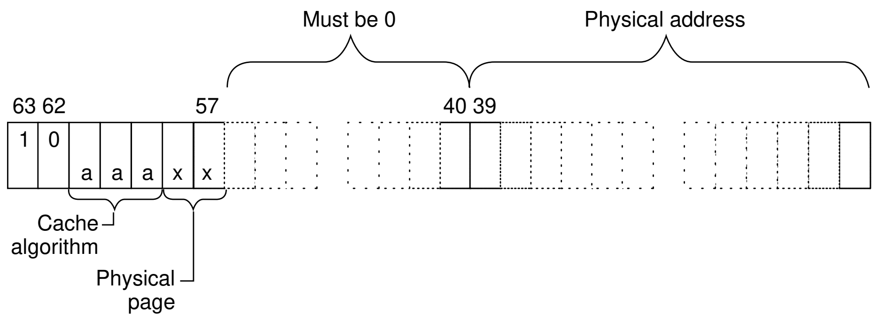

Figure 1-9 Address Decoding for Physical Memory Access

Bits 39:0 select a physical address in a 1 TB range. As a result, a system operating in 64-bit mode can access a much larger physical address space than the 512 MB space allowed by *kseg0*. This permits more physical memory to be installed, and it gives more freedom in assigning device and bus addresses.

Bits 57:40 must always contain 0. Bits 61:59 select the hardware cache algorithm to be used. The only values defined for these bits are summarized in Table 1-3.

Table 1-3 Cache Algorithm Selection

Address 61:59	Algorithm	Meaning
010	Uncached	This is the 64-bit equivalent of *kseg1* in 32-bit mode—uncached access to physical memory.
110	Cacheable coherent exclusive on write	This is the 64-bit equivalent of *kseg0* in 32-bit mode—cached access to physical memory, coherent access in a multiprocessor.
011	Cacheable non-coherent	Data is cached; on a cache miss the processor issues a non-coherent read (one without regard to other CPUs).
100	Cacheable coherent exclusive	Data is cached; on a read miss the processor issues a coherent read exclusive.

Table 1-3 (continued) Cache Algorithm Selection

Address 61:59	Algorithm	Meaning
101	Cacheable coherent update on write	Same as 110, but updates memory on a store hit in cache.
111	Uncached Accelerated	Same as 010, but the cache hardware is permitted to defer writes to memory until it has collected a larger block, improving write utilization.

Only the 010 (uncached) and 110 (cached) algorithms are implemented on all systems. The others may or may not be implemented on particular systems.

Bits 58:59 must be 00 unless the cache algorithm is 010 (uncached) or 111(uncached accelerated). Then bits 58:59 can in principle be used to select four other properties to qualify the operation. No present Silicon Graphics computer system supports these properties, so bits 58:59 always contain 00 at this time.

Portions of *xkphys* and *xkseg* can be mapped to user process space by the **mmap()** function. This is covered in more detail under "Memory Use in User-Level Drivers" on page 25.

Device Driver Use of Memory

Memory use by device drivers is simpler than the details in this chapter suggest. The primary complication for the designer is the use of 64-bit addresses, which may be unfamiliar.

Allowing for 64-Bit Mode

You must take account of a number of considerations when porting an existing C program to an environment where 64-bit mode is used, or might be used. For details see the *MIPSpro 64-Bit Porting and Transition Guide* listed on page xxxix.

The most common problems arise because the size of a pointer and of a long int changes between a program compiled with the -64 option and one compiled -32. When pointers, longs, or types derived from longs, are used in structures, the field offsets differ between the two modes.

When all programs in the system are compiled to the same mode, there is no problem. This is the case for a system in which the kernel is compiled to 32-bit mode: only 32-bit user programs are supported. However, a kernel compiled to 64-bit mode executes user programs in 32-bit or 64-bit mode. A structure prepared by a 32-bit program—a structure passed as an argument to **ioctl()**, for example—does not have fields at the offsets expected by a 64-bit kernel device driver. For more on this specific problem, see “Handling 32-Bit and 64-Bit Execution Models” on page 170.

The header files *sgidefs.h* and *sys/types.h* define type names that you can use to declare structures that always have the same size.

Memory Use in User-Level Drivers

When you control a device from a user process, your code executes entirely in user process space, and has no direct access to any of the other spaces described in this chapter.

Depending on the device and other considerations, you may use the **mmap()** function to map device registers into the address space of your process (see the mmap(2) reference page). When the kernel maps a device address into process space, it does it using the TLB mechanism. From **mmap()** you receive a valid address in process space. This address is mapped through a TLB entry to an address in segment that accesses uncached physical memory. When your program refers to this address, the reference is directed to the system bus and the device.

Portions of kernel virtual memory (*kseg0* or *xkseg*) can be accessed from a user process. Access is based on the use of device special files (see the mem(7) reference page). Access is done using two models, a device model and a memory map model.

Access Using a Device Model

The device special file */dev/mem* represents physical memory. A process that can open this device can use **lseek()** and **read()** to copy physical memory into process virtual memory. If the process can open the device for output, it can use **write()** to patch physical memory.

The device special file */dev/kmem* represents kernel virtual memory (*kseg0* or *xkseg*). It can be opened, read and written similarly to */dev/mem*. Clearly both of these devices should have file permissions that restrict their use even for input.

Access Using mmap()

The **mmap()** function allows a user process to map an open file into the process address space (see the mmap(2) reference page). When the file that is mapped is */dev/mem*, the process can map a specified segment of physical memory. The effect of **mmap()** is to set up a page table entry and TLB entry so that access to a range of virtual addresses in user space is redirected to the mapped physical addresses in cached physical memory (*kseg0* or the equivalent segment of *xkphys*).

The */dev/kmem* device, representing kernel virtual memory, cannot be used with **mmap()**. However, a third device special, */dev/mmem* (note the double "m"), represents access to only those addresses that are configured in the file */var/sysgen/master.d/mem*. As distributed, this file is configured to allow access to the free-running timer device and, in some systems, to graphics hardware.

For an example of mapped access to physical memory, see the example code in the syssgi(2) reference page related to the SGI_QUERY_CYCLECNTR option. In this operation, the address of the timer (a device register) is mapped into the process's address space using a TLB entry. When the user process accesses the mapped address, the TLB entry converts it to an address in *kseg1*/*xkphys*, which then bypasses the cache.

Mapped Access Provided by a Device Driver

A kernel-level device driver can provide mapped access to device registers or to memory allocated in kernel virtual space. An example of such a driver is shown in Part III, "Kernel-Level Drivers."

Memory Use in Kernel-Level Drivers

When you control a device from a kernel-level driver, your code executes in kernel virtual space. The allocation of memory for program text, local (stack) variables, and static global variables is handled automatically by the kernel. Besides designing data structures so they have a consistent size, you have to consider these special cases:

- dynamic memory allocation for data and for buffers
- getting addresses of device registers for the device you control
- transferring data between kernel space and user process space

The kernel supplies utility functions to help you deal with each of these issues, all of which are discussed in Chapter 9, "Device Driver/Kernel Interface."

Uncached Memory Access in the Challenge and Onyx Series

Access to uncached memory is not supported. The Challenge and Onyx systems have coherent caches; cache coherency is maintained by the hardware, even under access from CPUs and concurrent DMA. There is never a need (and no approved way) to access uncached memory in these systems.

Note: In certain specific hardware configurations, a device driver may have to deal with cache issues in a Challenge or Onyx system. See Appendix B, "Challenge DMA with Multiple IO4 Boards."

Uncached Memory Access in the IP26 CPU

The IP26 CPU module is used in the Silicon Graphics Power Indigo^2 workstation and the Power Challenge M workstation. Both are deskside workstations using the R8000 processor chip.

Late in the design of these systems, the parity-based memory that had been planned for them was replaced with ECC memory (error-correcting code memory, which can correct for single-bit errors on the fly). ECC memory is also used in large multiprocessor systems from Silicon Graphics, where it has no effect on performance.

Owing to the hardware design of the IP26, ECC memory could be added with no impact on the performance of cached memory access, but uncached memory access can be permitted only when the CPU is placed in a special, "slow" access mode.

In some cases a kernel-level device driver must be sure that stored data has been written into main memory, rather than being held in the cache. There are two ways to ensure this:

- Store the data into cached memory, then use the **dki_dcache_wb()** function to force a range of cached addresses to be written to memory. This method works in all systems including the IP26; however, the function call is an expensive one when the amount of data is small.
- Write directly to uncached memory using addresses in *kseg1*. This works in all systems, but in the IP26 (only) it will fail unless the CPU is first put into "slow" mode.

In order to put the CPU into “slow” mode, call the function **ip26_enable_ucmem()**. As soon as the uncached store is complete, return the system to “fast” mode by calling **ip26_return_ucmem()**. (See the ip26_ucmem(D3) reference page.) While the CPU is in “slow” mode, several clock cycles are added to every memory access, so do not keep it in “slow” mode any longer than necessary.

These functions can be called in any system. They do nothing unless the CPU is an IP26. Alternatively, you could save the current CPU type using a function like the one shown in Example 2-2 on page 31, and call the functions only when that function returns INV_IP26BOARD.