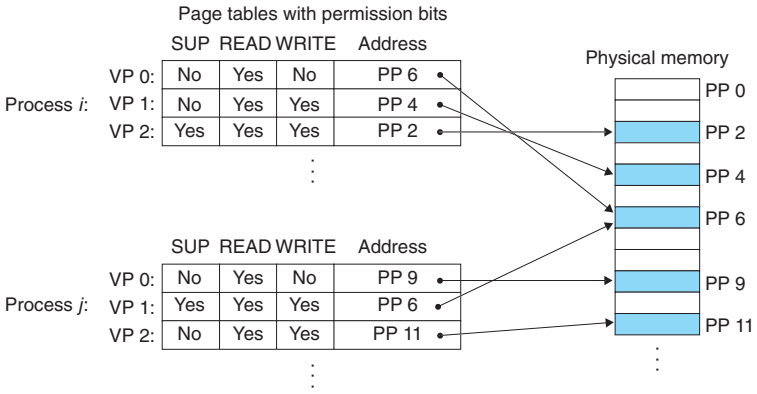


Figure 9.10
Using VM to provide page-level memory protection.

to modify its read-only code section. Nor should it be allowed to read or modify any of the code and data structures in the kernel. It should not be allowed to read or write the private memory of other processes, and it should not be allowed to modify any virtual pages that are shared with other processes, unless all parties explicitly allow it (via calls to explicit interprocess communication system calls).

As we have seen, providing separate virtual address spaces makes it easy to isolate the private memories of different processes. But the address translation mechanism can be extended in a natural way to provide even finer access control. Since the address translation hardware reads a PTE each time the CPU generates an address, it is straightforward to control access to the contents of a virtual page by adding some additional permission bits to the PTE. Figure 9.10 shows the general idea.

In this example, we have added three permission bits to each PTE. The SUP bit indicates whether processes must be running in kernel (supervisor) mode to access the page. Processes running in kernel mode can access any page, but processes running in user mode are only allowed to access pages for which SUP is 0. The READ and WRITE bits control read and write access to the page. For example, if process *i* is running in user mode, then it has permission to read VP 0 and to read or write VP 1. However, it is not allowed to access VP 2.

If an instruction violates these permissions, then the CPU triggers a general protection fault that transfers control to an exception handler in the kernel, which sends a SIGSEGV signal to the offending process. Linux shells typically report this exception as a "segmentation fault."

9.6 Address Translation

This section covers the basics of address translation. Our aim is to give you an appreciation of the hardware's role in supporting virtual memory, with enough detail so that you can work through some concrete examples by hand. However, keep in mind that we are omitting a number of details, especially related to timing,

Symbol	Description
Basic parameters	
$N = 2^n$	Number of addresses in virtual address space
$M = 2^m$	Number of addresses in physical address space
$P = 2^p$	Page size (bytes)
Components of a virtual address (VA)	
VPO	Virtual page offset (bytes)
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag
Components of a physical address (PA)	
PPO	Physical page offset (bytes)
PPN	Physical page number
CO	Byte offset within cache block
CI	Cache index
CT	Cache tag

Figure 9.11 Summary of address translation symbols.

that are important to hardware designers but are beyond our scope. For your reference, Figure 9.11 summarizes the symbols that we will be using throughout this section.

Formally, address translation is a mapping between the elements of an N-element virtual address space (VAS) and an M-element physical address space (PAS),

$$\text{MAP: VAS} \rightarrow \text{PAS} \cup \emptyset$$

where

$$\text{MAP}(A) = \begin{cases} A' & \text{if data at virtual addr. } A \text{ are present at physical addr. } A' \text{ in PAS} \\ \emptyset & \text{if data at virtual addr. } A \text{ are not present in physical memory} \end{cases}$$

Figure 9.12 shows how the MMU uses the page table to perform this mapping. A control register in the CPU, the *page table base register (PTBR)* points to the current page table. The n-bit virtual address has two components: a p-bit *virtual page offset (VPO)* and an $(n - p)$-bit *virtual page number (VPN)*. The MMU uses the VPN to select the appropriate PTE. For example, VPN 0 selects PTE 0, VPN 1 selects PTE 1, and so on. The corresponding physical address is the concatenation of the *physical page number (PPN)* from the page table entry and the VPO from the virtual address. Notice that since the physical and virtual pages are both P bytes, the *physical page offset (PPO)* is identical to the VPO.

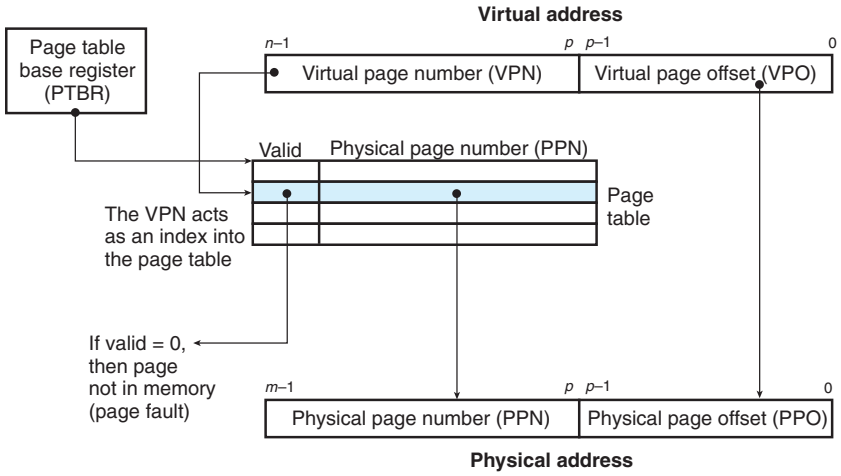


Figure 9.12 **Address translation with a page table.**

Figure 9.13(a) shows the steps that the CPU hardware performs when there is a page hit.

Step 1. The processor generates a virtual address and sends it to the MMU.

Step 2. The MMU generates the PTE address and requests it from the cache/main memory.

Step 3. The cache/main memory returns the PTE to the MMU.

Step 4. The MMU constructs the physical address and sends it to the cache/main memory.

Step 5. The cache/main memory returns the requested data word to the processor.

Unlike a page hit, which is handled entirely by hardware, handling a page fault requires cooperation between hardware and the operating system kernel (Figure 9.13(b)).

Steps 1 to 3. The same as steps 1 to 3 in Figure 9.13(a).

Step 4. The valid bit in the PTE is zero, so the MMU triggers an exception, which transfers control in the CPU to a page fault exception handler in the operating system kernel.

Step 5. The fault handler identifies a victim page in physical memory, and if that page has been modified, pages it out to disk.

Step 6. The fault handler pages in the new page and updates the PTE in memory.

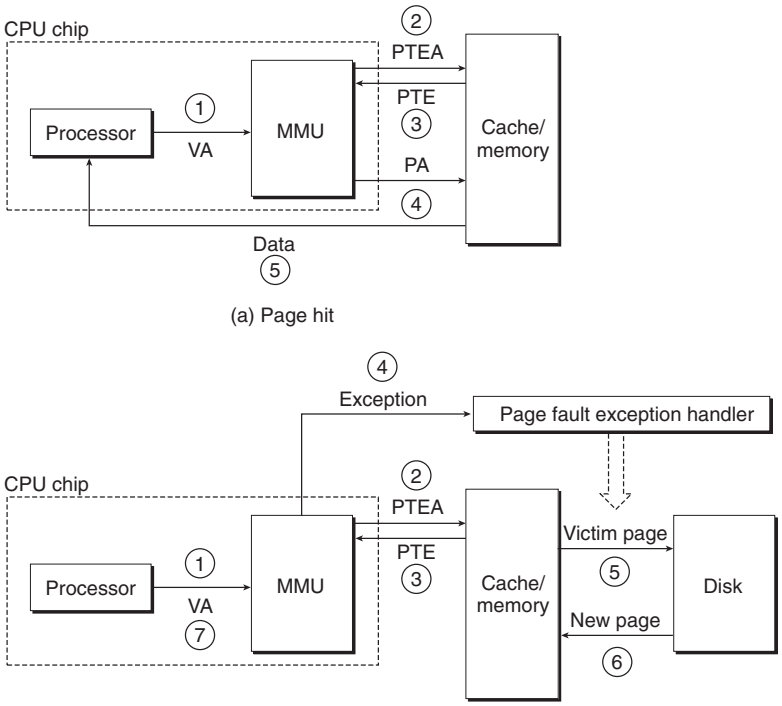


Figure 9.13 Operational view of page hits and page faults. VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

Step 7. The fault handler returns to the original process, causing the faulting instruction to be restarted. The CPU resends the offending virtual address to the MMU. Because the virtual page is now cached in physical memory, there is a hit, and after the MMU performs the steps in Figure 9.13(a), the main memory returns the requested word to the processor.

Practice Problem 9.3 (solution page 917)

Given a 64-bit virtual address space and a 32-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P:

	Number of			
P	VPN bits	VPO bits	PPN bits	PPO bits
1 KB				
2 KB				
4 KB				
16 KB				

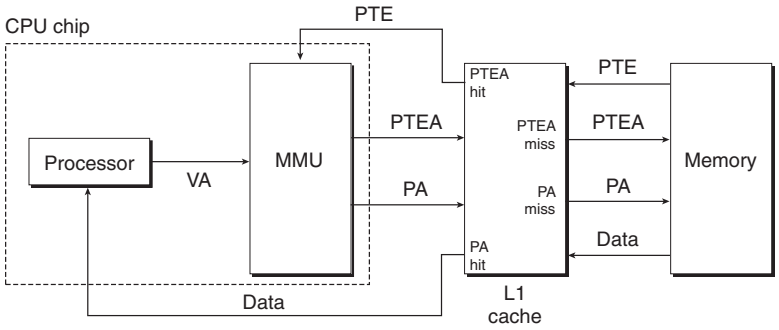


Figure 9.14 Integrating VM with a physically addressed cache. VA: virtual address. PTEA: page table entry address. PTE: page table entry. PA: physical address.

9.6.1 Integrating Caches and VM

In any system that uses both virtual memory and SRAM caches, there is the issue of whether to use virtual or physical addresses to access the SRAM cache. Although a detailed discussion of the trade-offs is beyond our scope here, most systems opt for physical addressing. With physical addressing, it is straightforward for multiple processes to have blocks in the cache at the same time and to share blocks from the same virtual pages. Further, the cache does not have to deal with protection issues, because access rights are checked as part of the address translation process.

Figure 9.14 shows how a physically addressed cache might be integrated with virtual memory. The main idea is that the address translation occurs before the cache lookup. Notice that page table entries can be cached, just like any other data words.

9.6.2 Speeding Up Address Translation with a TLB

As we have seen, every time the CPU generates a virtual address, the MMU must refer to a PTE in order to translate the virtual address into a physical address. In the worst case, this requires an additional fetch from memory, at a cost of tens to hundreds of cycles. If the PTE happens to be cached in L1, then the cost goes down to a handful of cycles. However, many systems try to eliminate even this cost by including a small cache of PTEs in the MMU called a *translation lookaside buffer (TLB)*.

A TLB is a small, virtually addressed cache where each line holds a block consisting of a single PTE. A TLB usually has a high degree of associativity. As shown in Figure 9.15, the index and tag fields that are used for set selection and line matching are extracted from the virtual page number in the virtual address. If the TLB has $T = 2^t$ sets, then the *TLB index (TLBI)* consists of the t least significant bits of the VPN, and the *TLB tag (TLBT)* consists of the remaining bits in the VPN.

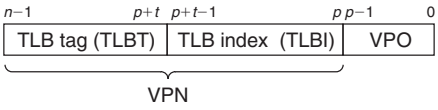


Figure 9.15
Components of a virtual address that are used to access the TLB.

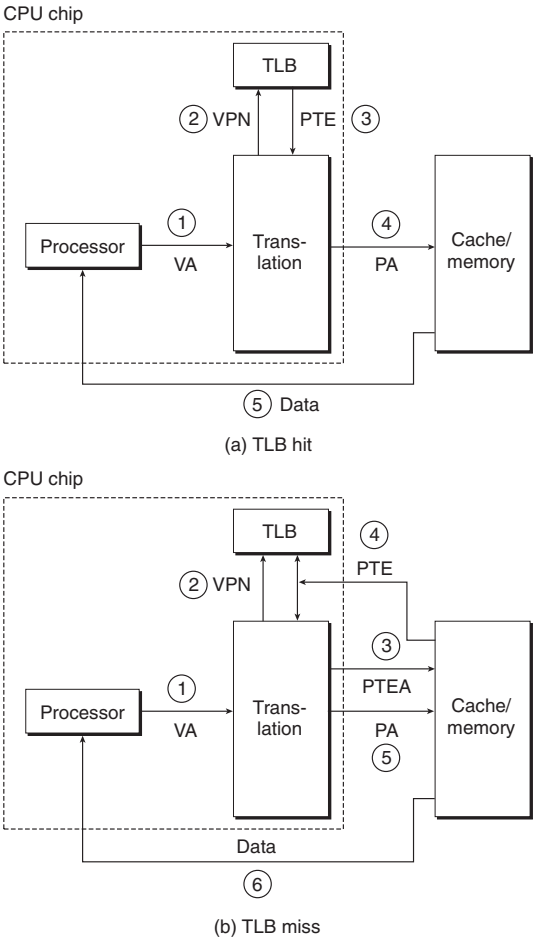


Figure 9.16
Operational view of a TLB hit and miss.

Figure 9.16(a) shows the steps involved when there is a TLB hit (the usual case). The key point here is that all of the address translation steps are performed inside the on-chip MMU and thus are fast.

Step 1. The CPU generates a virtual address.

Steps 2 and 3. The MMU fetches the appropriate PTE from the TLB.

Step 4. The MMU translates the virtual address to a physical address and sends it to the cache/main memory.

Step 5. The cache/main memory returns the requested data word to the CPU.

When there is a TLB miss, then the MMU must fetch the PTE from the L1 cache, as shown in Figure 9.16(b). The newly fetched PTE is stored in the TLB, possibly overwriting an existing entry.

9.6.3 Multi-Level Page Tables

Thus far, we have assumed that the system uses a single page table to do address translation. But if we had a 32-bit address space, 4 KB pages, and a 4-byte PTE, then we would need a 4 MB page table resident in memory at all times, even if the application referenced only a small chunk of the virtual address space. The problem is compounded for systems with 64-bit address spaces.

The common approach for compacting the page table is to use a hierarchy of page tables instead. The idea is easiest to understand with a concrete example. Consider a 32-bit virtual address space partitioned into 4 KB pages, with page table entries that are 4 bytes each. Suppose also that at this point in time the virtual address space has the following form: The first 2 K pages of memory are allocated for code and data, the next 6 K pages are unallocated, the next 1,023 pages are also unallocated, and the next page is allocated for the user stack. Figure 9.17 shows how we might construct a two-level page table hierarchy for this virtual address space.

Each PTE in the level 1 table is responsible for mapping a 4 MB chunk of the virtual address space, where each chunk consists of 1,024 contiguous pages. For example, PTE 0 maps the first chunk, PTE 1 the next chunk, and so on. Given that the address space is 4 GB, 1,024 PTEs are sufficient to cover the entire space.

If every page in chunk i is unallocated, then level 1 PTE i is null. For example, in Figure 9.17, chunks 2–7 are unallocated. However, if at least one page in chunk i is allocated, then level 1 PTE i points to the base of a level 2 page table. For example, in Figure 9.17, all or portions of chunks 0, 1, and 8 are allocated, so their level 1 PTEs point to level 2 page tables.

Each PTE in a level 2 page table is responsible for mapping a 4-KB page of virtual memory, just as before when we looked at single-level page tables. Notice that with 4-byte PTEs, each level 1 and level 2 page table is 4 kilobytes, which conveniently is the same size as a page.

This scheme reduces memory requirements in two ways. First, if a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist. This represents a significant potential savings, since most of the 4 GB virtual address space for a typical program is unallocated. Second, only the level 1 table needs to be in main memory at all times. The level 2 page tables can be created and paged in and out by the VM system as they are needed, which reduces pressure on main memory. Only the most heavily used level 2 page tables need to be cached in main memory.

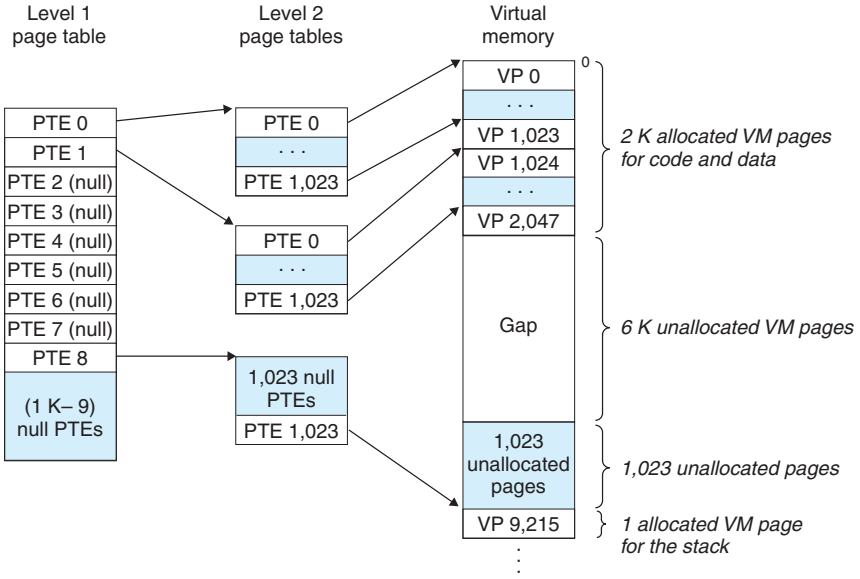


Figure 9.17 **A two-level page table hierarchy.** Notice that addresses increase from top to bottom.

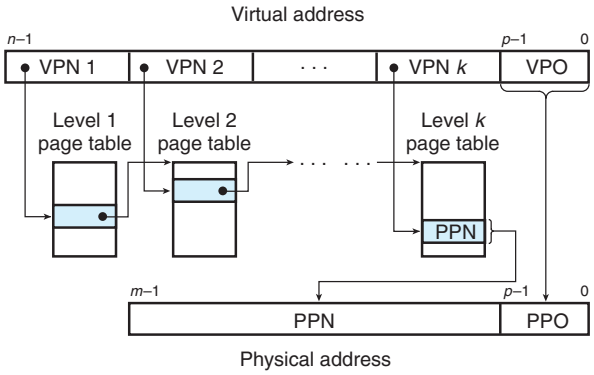


Figure 9.18 **Address translation with a k-level page table.**

Figure 9.18 summarizes address translation with a k-level page table hierarchy. The virtual address is partitioned into k VPNs and a VPO. Each VPN i, $1 \leq i \leq k$, is an index into a page table at level i. Each PTE in a level j table, $1 \leq j \leq k-1$, points to the base of some page table at level $j+1$. Each PTE in a level k table contains either the PPN of some physical page or the address of a disk block. To construct the physical address, the MMU must access k PTEs before it can

determine the PPN. As with a single-level hierarchy, the PPO is identical to the VPO.

Accessing k PTEs may seem expensive and impractical at first glance. However, the TLB comes to the rescue here by caching PTEs from the page tables at the different levels. In practice, address translation with multi-level page tables is not significantly slower than with single-level page tables.

9.6.4 Putting It Together: End-to-End Address Translation

In this section, we put it all together with a concrete example of end-to-end address translation on a small system with a TLB and L1 d-cache. To keep things manageable, we make the following assumptions:

- The memory is byte addressable.
- Memory accesses are to *1-byte words* (not 4-byte words).
- Virtual addresses are 14 bits wide ($n = 14$).
- Physical addresses are 12 bits wide ($m = 12$).
- The page size is 64 bytes ($P = 64$).
- The TLB is 4-way set associative with 16 total entries.
- The L1 d-cache is physically addressed and direct mapped, with a 4-byte line size and 16 total sets.

Figure 9.19 shows the formats of the virtual and physical addresses. Since each page is $2^6 = 64$ bytes, the low-order 6 bits of the virtual and physical addresses serve as the VPO and PPO, respectively. The high-order 8 bits of the virtual address serve as the VPN. The high-order 6 bits of the physical address serve as the PPN.

Figure 9.20 shows a snapshot of our little memory system, including the TLB (Figure 9.20(a)), a portion of the page table (Figure 9.20(b)), and the L1 cache (Figure 9.20(c)). Above the figures of the TLB and cache, we have also shown how the bits of the virtual and physical addresses are partitioned by the hardware as it accesses these devices.

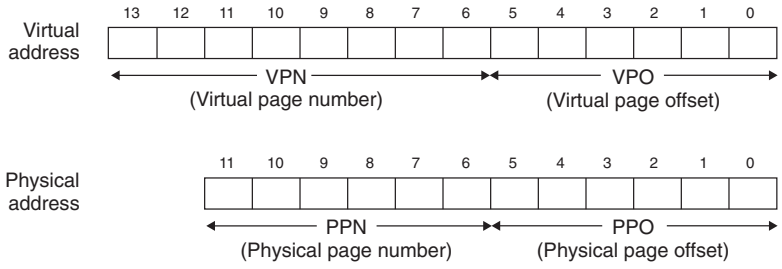
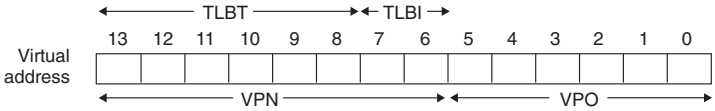


Figure 9.19 Addressing for small memory system. Assume 14-bit virtual addresses ($n = 14$), 12-bit physical addresses ($m = 12$), and 64-byte pages ($P = 64$).



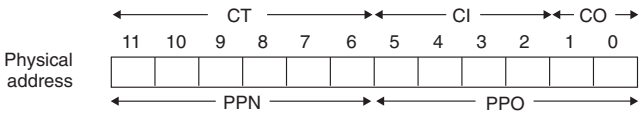
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

(a) TLB: 4 sets, 16 entries, 4-way set associative

VPN	PPN	Valid
00	28	1
01	—	0
02	33	1
03	02	1
04	—	0
05	16	1
06	—	0
07	—	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	—	0
0C	—	0
0D	2D	1
0E	11	1
0F	0D	1

(b) Page table: Only the first 16 PTEs are shown



Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	—	—	—	—
2	1B	1	00	02	04	08
3	36	0	—	—	—	—
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	—	—	—	—
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	—	—	—	—
A	2D	1	93	15	DA	3B
B	0B	0	—	—	—	—
C	12	0	—	—	—	—
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	—	—	—	—

(c) Cache: 16 sets, 4-byte blocks, direct mapped

Figure 9.20 TLB, page table, and cache for small memory system. All values in the TLB, page table, and cache are in hexadecimal notation.

TLB. The TLB is virtually addressed using the bits of the VPN. Since the TLB has four sets, the 2 low-order bits of the VPN serve as the set index (TLBI). The remaining 6 high-order bits serve as the tag (TLBT) that distinguishes the different VPNs that might map to the same TLB set.

Page table. The page table is a single-level design with a total of $2^8 = 256$ page table entries (PTEs). However, we are only interested in the first 16 of these. For convenience, we have labeled each PTE with the VPN that indexes it; but keep in mind that these VPNs are not part of the page table and not stored in memory. Also, notice that the PPN of each invalid PTE is denoted with a dash to reinforce the idea that whatever bit values might happen to be stored there are not meaningful.

Cache. The direct-mapped cache is addressed by the fields in the physical address. Since each block is 4 bytes, the low-order 2 bits of the physical address serve as the block offset (CO). Since there are 16 sets, the next 4 bits serve as the set index (CI). The remaining 6 bits serve as the tag (CT).

Given this initial setup, let's see what happens when the CPU executes a load instruction that reads the byte at address `0x03d4`. (Recall that our hypothetical CPU reads 1-byte words rather than 4-byte words.) To begin this kind of manual simulation, we find it helpful to write down the bits in the virtual address, identify the various fields we will need, and determine their hex values. The hardware performs a similar task when it decodes the address.

	TLBT						TLBI							
	0x03						0x03							
Bit position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VA = 0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0
	VPN								VPO					
	0x0f								0x14					

To begin, the MMU extracts the VPN (`0x0F`) from the virtual address and checks with the TLB to see if it has cached a copy of PTE `0x0F` from some previous memory reference. The TLB extracts the TLB index (`0x03`) and the TLB tag (`0x3`) from the VPN, hits on a valid match in the second entry of set `0x3`, and returns the cached PPN (`0x0D`) to the MMU.

If the TLB had missed, then the MMU would need to fetch the PTE from main memory. However, in this case, we got lucky and had a TLB hit. The MMU now has everything it needs to form the physical address. It does this by concatenating the PPN (`0x0D`) from the PTE with the VPO (`0x14`) from the virtual address, which forms the physical address (`0x354`).

Next, the MMU sends the physical address to the cache, which extracts the cache offset CO (`0x0`), the cache set index CI (`0x5`), and the cache tag CT (`0x0D`) from the physical address.

	CT						CI				CO	
	0x0d						0x05				0x0	
Bit position	11	10	9	8	7	6	5	4	3	2	1	0
PA = 0x354	0	0	1	1	0	1	0	1	0	1	0	0
	PPN						PPO					
	0x0d						0x14					

Since the tag in set 0x5 matches CT, the cache detects a hit, reads out the data byte (0x36) at offset CO, and returns it to the MMU, which then passes it back to the CPU.

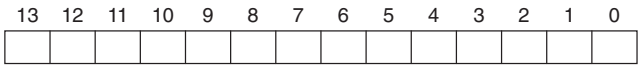
Other paths through the translation process are also possible. For example, if the TLB misses, then the MMU must fetch the PPN from a PTE in the page table. If the resulting PTE is invalid, then there is a page fault and the kernel must page in the appropriate page and rerun the load instruction. Another possibility is that the PTE is valid, but the necessary memory block misses in the cache.

Practice Problem 9.4 (solution page 917)

Show how the example memory system in Section 9.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, physical address, and cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter "—" for "Cache byte returned." If there is a page fault, enter "—" for "PPN" and leave parts C and D blank.

Virtual address: 0x03d7

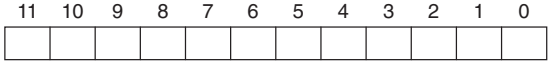
A. Virtual address format



B. Address translation

Parameter	Value
VPN	_____
TLB index	_____
TLB tag	_____
TLB hit? (Y/N)	_____
Page fault? (Y/N)	_____
PPN	_____

C. Physical address format



D. Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit? (Y/N)	
Cache byte returned	

9.7 Case Study: The Intel Core i7/Linux Memory System

We conclude our discussion of virtual memory mechanisms with a case study of a real system: an Intel Core i7 running Linux. Although the underlying Haswell microarchitecture allows for full 64-bit virtual and physical address spaces, the current Core i7 implementations (and those for the foreseeable future) support a 48-bit (256 TB) virtual address space and a 52-bit (4 PB) physical address space, along with a compatibility mode that supports 32-bit (4 GB) virtual and physical address spaces.

Figure 9.21 gives the highlights of the Core i7 memory system. The *processor package* (chip) includes four cores, a large L3 cache shared by all of the cores, and

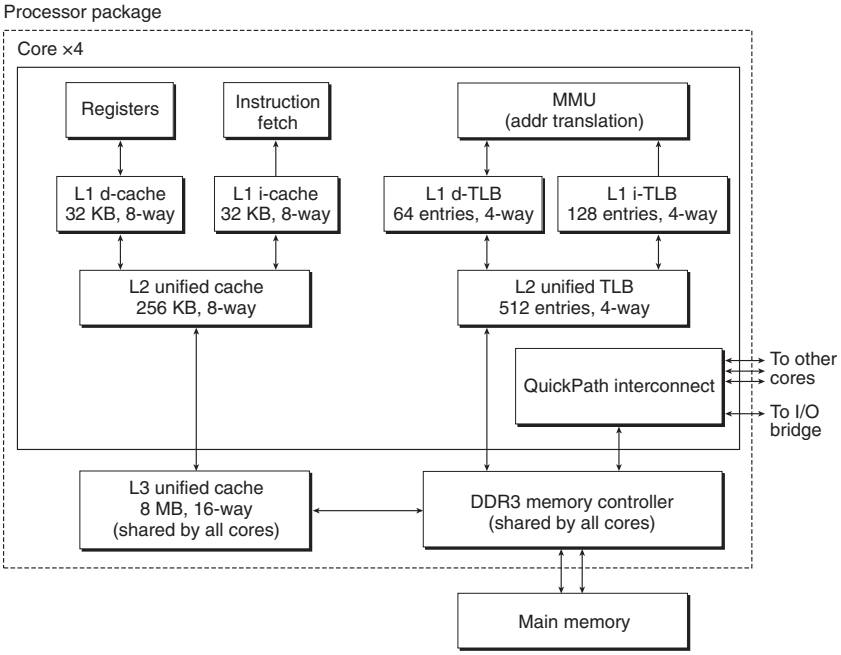


Figure 9.21 The Core i7 memory system.