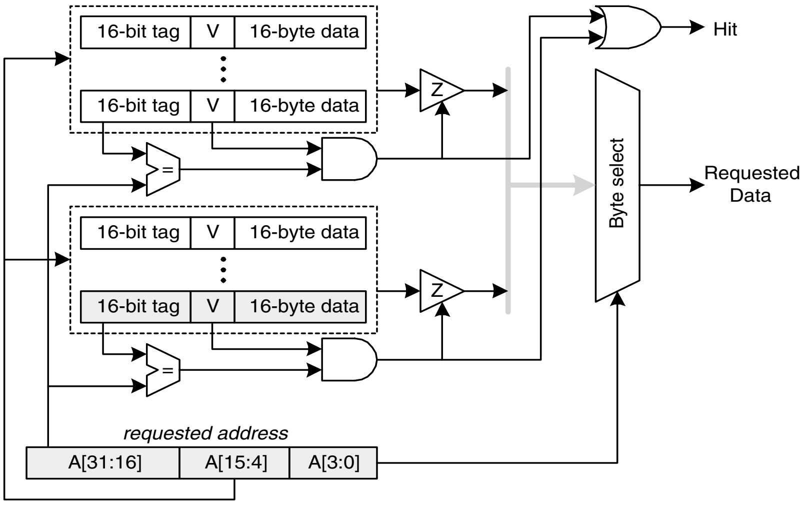

FIGURE 7.4 128-kB two-way set associative cache.

cache has expanded to 128 kB in size using two 64 kB elements. If cost constraints dictate keeping a 64 kB cache, it would be preferable to reduce the set size to 2,048 rather than halve the line size, which is already at a practical minimum of 16 bytes. Reducing the set size to 2^{11} would increase the line tag to 17 bits to maintain a 32-bit address space representation. In a cache of this type, the controller can choose which of two (or four, or *n*) cache line locations to flush when a miss is encountered.

Deciding which line to flush when a cache miss occurs can be done in a variety of ways, and different cache architectures dictate varying approaches to this problem. A fully associative cache can place any main memory block into any line, while a direct mapped cache has only one choice for any given memory block. Three basic flush, or replacement, algorithms are as follows:

- *First-in-first-out (FIFO).* Track cache line ages and replace the oldest line.
- *Least-recently-used (LRU).* Track cache line usage and replace the line that has not been accessed longest.
- *Random.* Replace a random line.

A fully associative cache has the most flexibility in selecting cache lines and therefore the most complexity in tracking line usage. To perform either a FIFO or LRU replacement algorithm on a fully associative cache, each line would need a tracking field that could be updated and checked in parallel with all other lines. N-way set associative caches are the most interesting problems from a practical perspective, because they are used most frequently. Replacement algorithms for these caches are simplified, because the number of replacement choices is restricted to N. A two-way set associative cache can implement either FIFO or LRU algorithms with a single bit per line entry. For a FIFO algorithm, the entry being loaded anew has its FIFO bit cleared, and the other entry has its FIFO bit set, indicating that the other entry was loaded first. For an LRU algorithm, the entry being accessed at any given time has its LRU bit cleared, and the other has its LRU bit set, indicating that the other entry was used least recently. These algorithms and associated hardware are only slightly more complex for a four-way set associative cache that would require two status bits per line entry.

7.3 CACHES IN PRACTICE

Basic cache structures can be applied and augmented in different ways to improve their efficacy. One common manner in which caches are implemented is in pairs: an *I-cache* to hold instructions and a *D-cache* to hold data. It is not uncommon to see high-performance RISC microprocessors with integrated I/D caches on chip. Depending on the intended application, these integrated caches can be relatively small, each perhaps 8 kB to 32 kB in size. More often than not, these are two-way or four-way set associative caches. There are two key benefits to integrating two separate caches. First, instruction and data access patterns can combine negatively to cause thrashing on a single normal cache. If a software routine operates on a set of data whose addresses happen to overlap with the I-cache's index bits, alternate instruction and data fetch operations could cause repeated thrashing on the same cache lines. Second, separate caches can effectively provide a Harvard memory architecture from the microprocessor's local perspective. While it is often not practical to provide dual instruction and data memory interfaces at the chip level, as a result of excessive pin count, such considerations are much less restrictive within a silicon die. Separate I/D caches can feed from a shared chip-level memory interface but provide independent interfaces to the microprocessor core itself. This dual-bus arrangement increases the microprocessor's load/store bandwidth by enabling it to simultaneously fetch instructions and operands without conflict.

Dual I/D caches cannot guarantee complete independence of instruction and data memory, because, ultimately, they are operating through a shared interface to a common pool of main memory. The performance boost that they provide will be dictated largely by the access patterns of the applications running on the microprocessor. Such application-dependent performance is fundamental to all types of caches, because caches rely on locality to provide their benefits. Programs that scatter instructions and data throughout a memory space and alternately access these disparate locations will show less performance improvement with the cache. However, most programs exhibit fairly beneficial locality characteristics. A system with dual I/D caches can show substantial throughput improvement when a software routine can fit its core processing instructions into the instruction cache with minimal thrashing and its data sets exhibit good locality properties. Under these circumstances, the data cache can have more time to pull in data via the common memory interface, enabling the microprocessor to simultaneously access instruction and data memory with a low miss rate.

Computer systems with caches require some assistance from the operating system and applications to maximize cache performance benefits and to prevent unexpected side effects of cached memory. It is helpful to cache certain areas of memory, but it is performance degrading or even harmful to cache other areas. Memory-mapped I/O devices are generally excluded from being cached, because I/O is a class of device that usually responds with some behavior when a control register is modified. Likewise, I/O devices frequently update their status registers to reflect conditions that they are monitoring. If a cache prevents the microprocessor from directly interacting with an I/O device, unexpected results may appear. For example, if the microprocessor wants to send data out a serial port, it might write a data value to a transmit register, expecting that the data will be sent immediately. However, a write-back cache would not actually perform the write until the associated cache line is flushed—a time delay that is unbounded. Similarly, the serial port controller could reflect handshaking status information in its status registers that the microprocessor wants to periodically read. An unknowing cache would fetch the status register memory location once and then continue to return the originally fetched value to the microprocessor until its cache line was flushed, thereby preventing the microprocessor from reading the true status of the serial port. I/O