sity of SRAM as compared to DRAM. An SRAM-based main memory requires more devices, more circuit board area, and more connecting wires—all requirements that add cost and reduce the reliability of a system. Some supercomputers have been built with main memory composed entirely of SRAM, but keep in mind that these products have minimal cost constraints, if any.

If software running on microprocessors tended to access every main memory location with equal probability, not much could be done to improve memory bandwidth without substantial increases in size and cost. Under such circumstances, a choice would have to be made between a large quantity of slow memory or a small quantity of fast memory. Fortunately, software tends to access fairly constrained sets of instructions and data in a given period of time, thereby increasing the probability of accessing sequential memory locations and decreasing the probability of truly random accesses. This property is generally referred to as *locality*. Instructions tend to be executed sequentially in the order in which they are stored in memory. When branches occur, the majority are with small displacements for purposes of forming loops and local "if…then…else" logical decisions. Data also tend to be grouped into sequential elements. For example, if a string of characters forming a person's name in a database is being processed, the characters in the string will be located in sequential memory locations. Furthermore, the entire database entry for the person will likely be stored as a unit in nearby memory locations.

Caches largely overcome main memory latency problems. A cache, pronounced "cash," is a small quantity of fast memory that is used to temporarily store portions of main memory that the microprocessor accesses often or is predicted to access in the near future. Being that cache memory is relatively small, SRAM becomes practical to use in light of its substantial benefits of fast access time and simplicity—a memory controller is not needed to perform refresh or address multiplexing operations. As shown in Fig. 7.1, a cache sits between a microprocessor and main memory and is composed of two basic elements: cache memory and a cache controller.

The cache controller watches all memory transactions initiated by the microprocessor and selects whether read data is fetched from the cache or directly from main memory and whether writes go into the cache or into main memory. Transactions to main memory will be slower than those to the cache, so the cache controller seeks to minimize the number of transactions that are handled directly by main memory.

Locality enables a cache controller to increase the probability of a *cache hit*—that data requested by the microprocessor has already been loaded into the cache. A 100 percent hit rate is impossible, because the controller cannot predict the future with certainty, resulting in a *cache miss* every so often. *Temporal* and *spatial* locality properties of instructions and data help the controller improve its hit rate. Temporal locality says that, if a memory location is accessed once, it is likely to be accessed again in the near future. This can be readily observed by considering a software loop: instructions in the body of the loop are very likely to be fetched again in the near future during the next loop itera-

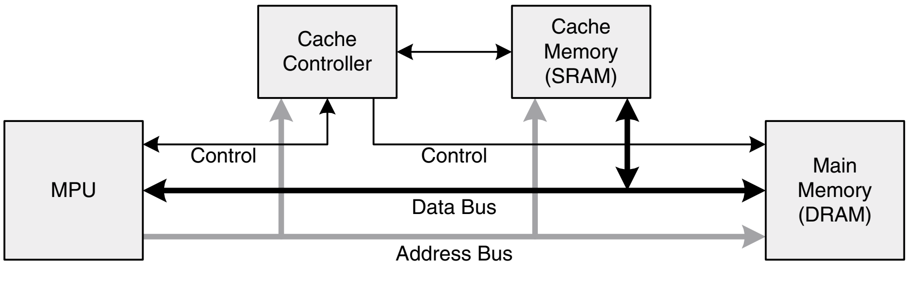

FIGURE 7.1 Computer with cache.

tion. Spatial locality says that, if a memory location is accessed, it is likely that nearby locations will be accessed in the near future. When a microprocessor fetches an instruction, there is a high probability that it will soon fetch the instructions immediately following that instruction. Practically speaking, temporal locality tells the cache controller to attempt to retain recently accessed memory locations in the expectation that they will be accessed again. Spatial locality tells the cache controller to preload additional sequential memory locations when a single location is fetched by the microprocessor, in the expectation that these locations will be soon accessed.

Given the locality properties, especially spatial locality, that need to be incorporated into the cache controller, a basic cache organization emerges in which blocks of data rather than individual bytes are managed by the controller and held in cache memory. These blocks are commonly called *lines,* and they vary in size, depending on the specific implementation. Typical cache line sizes are 16, 32, or 64 bytes. When the microprocessor reads a memory location that is not already located in the cache (a miss), the cache controller fetches an entire line from main memory and stores it as a unit. To maintain the simplicity of power-of-two logic, cache lines are typically mapped into main memory on boundaries defined by the line size. A 16-byte cache line will always hold memory locations at offsets represented by the four least-significant address bits. Main memory is therefore effectively divided into many small 16-byte lines with offsets from 0x0 to 0xF. If a microprocessor with a 32-bit address bus fetches location 0x1000800C and there is a cache miss, the controller will load locations 0x10008000 through 0x1000800F into a designated cache line. If the cache is full, and a miss occurs, the controller must *flush* a line that has a lower probability of use so as to make room for the new data. If the flushed line has been modified by writes that were not already reflected in main memory, the controller must store the line to prevent losing and corrupting the memory contents.

As more cache lines are implemented, more sections of main memory can be simultaneously held in the cache, increasing the hit rate. However, a cache's overall size must be bounded by a system's target size and cost constraints. The size of a cache line is a compromise between granularity, load/store time, and locality benefits. For a fixed overall size, larger lines reduce the granularity of unique blocks of main memory that can be simultaneously held in the cache. Larger cache lines increase the time required to load a new line and update main memory when flushing an old line. Larger cache lines also increase the probability that a subsequent access will result in a hit.

Cache behavior on reads is fairly consistent across different implementations. Writes, however, can be handled in one of three basic manners: *no-write*, *write-through*, and *write-back*. A no-write cache does not support the modification of its contents. When a write is performed to a block of memory held in a cache line, that line is flushed, and the write is performed directly into main memory. This scheme imposes two penalties on the system: writes are always slowed by the longer latency of main memory, and locality benefits are lost because the flush forces any subsequent accesses to that line to result in a miss and reload of the entire line that was already present in the cache.

Write-through caches support the modification of their contents but do not support incoherency between cache memory and main memory. Therefore, a write to a block of memory held in a cache line results in a parallel write to both the cache and main memory. This is an improvement over a no-write cache in that the cache line is not forcibly flushed, but the write is still slowed by a direct access to main memory.

A write-back cache minimizes both penalties by enabling writes to valid cache lines but not immediately causing a write to main memory. The microprocessor does not have to incur the latency penalty of main memory, because the write completes as fast as the cache can accept the new data. This scheme introduces complexity in the form of incoherency between cache and main memory: each memory structure has a different version of the same memory location. To solve the incoherency problem, a write-back cache must maintain a status bit for each line that indicates whether the