2.2.2 DMA Structure

In a simple terminal-input driver, when a line is to be read from the terminal, the first character typed is sent to the computer. When that character is received, the asynchronous-communication (or serial-port) device to which the terminal line is connected interrupts the CPU. When the interrupt request from the terminal arrives, the CPU is about to execute some instruction. The address of this interrupted instruction is saved, and control is transferred to the interrupt service routine for the appropriate device.

The interrupt service routine saves the contents of any CPU registers that it will need to use. It checks for any error conditions that might have resulted from the most recent input operation. It then takes the character from the device, and stores that character in a buffer. The interrupt routine must also adjust pointer and counter variables, to be sure that the next input character will be stored at the next location in the buffer. The interrupt routine next sets a flag in memory indicating to the other parts of the operating system that new input has been received. The other parts are responsible for processing the data in the buffer, and for transferring the characters to the program that is requesting input. Then, the interrupt service routine restores the contents of any saved registers, and transfers control back to the interrupted instruction.

A well-written interrupt service routine to input characters into a buffer may require 2 microseconds per character, leaving 998 microseconds out of every 1000 for CPU computation (and for servicing of other interrupts).

Given this disparity, asynchronous I/O is usually assigned a low interrupt priority, allowing other, more important interrupts to be processed first, or even to preempt the current interrupt for another. A high-speed device, however such as a tape, disk, or communications network-may be able to transmit information at close to memory speeds; if the CPU needs two microseconds to respond to each interrupt and interrupts arrive every four microseconds, for example, that does not leave much time for process execution. To solve this problem, **direct memory access (DMA)** is used for high-speed I/O devices. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, rather than the one interrupt per byte (or word) generated for low-speed devices. The basic operation of the CPU is the same. A user program, or the operating system itself, may request data transfer. The operating system finds a buffer (an empty buffer for input, or a full buffer for output) from a pool of buffers for the transfer. Next, a portion of the operating system called a **device driver** sets the DMA controller registers to use appropriate source and destination addresses, and transfer length. The DMA controller is then instructed to start the I/O operation. While the DMA controller is performing the data transfer, the CPU is free to perform other tasks. Since the memory generally can transfer only one word at a time, the DMA controller "steals" memory cycles from the CPU. This cycle stealing can slow down the CPU execution while a DMA transfer is in progress. The DMA controller interrupts the CPU when the transfer has been completed.

2.3 Storage Structure

Computer programs must be in main memory to be executed. Main memory is the only large storage area that the processor can access directly. It is implemented in a semiconductor technology called **dynamic random-access memory** (DRAM), which forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. A side from explicit loads and stores, the CPU automatically loads instructions from main memory for execution. Ideally, we want the programs and data to reside in main memory permanently. This arrangement is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory is a volatile storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk,** which provides storage of both programs and data. In a larger sense, however, the storage structure that we have described consisting of registers, main memory, and magnetic disks-is only one of many possible storage systems. There are also cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum, and of holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility.

2.4 Storage Hierarchy

The wide variety of storage systems in a computer system can be organized in a hierarchy (Figure 2.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This tradeoff is reasonable; if a given storage system were both faster and less expensive than another-other properties being the same-then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top three levels of memory in Figure 2.4 may be constructed using semiconductor memory.

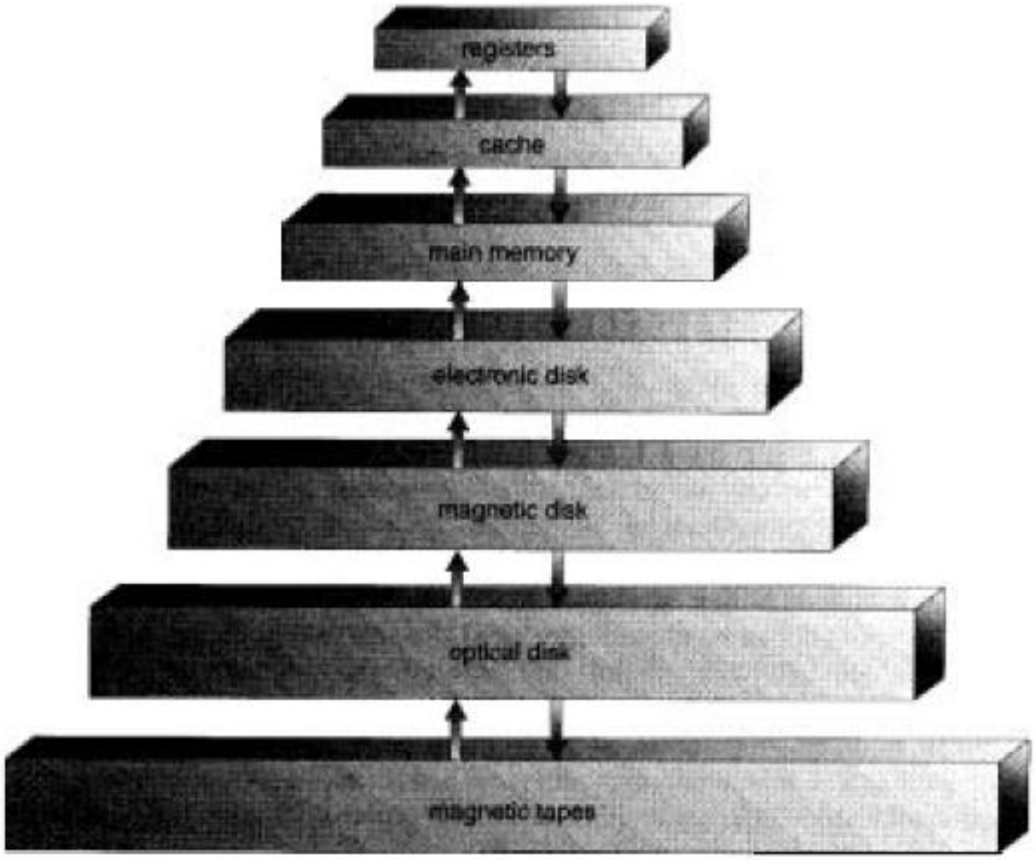

Fig. 2.4 Storage-device hierarchy.

In addition to having differing speed and cost, the various storage systems are either volatile or nonvolatile. **Volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in Figure 2.4, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile. An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic disk controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into the RAM. The design of a complete memory system must balance all these factors:
It uses only as much expensive memory as necessary, while providing as much inexpensive, nonvolatile memory as possible. Caches can be installed to improve performance where a large access-time or transfer-rate disparity exists between two components.

2.4.1 Caching

Caching is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system-the cache-on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache; if it is not, we use the information from the main storage system, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. There are also caches that are implemented totally in hardware. For instance, most systems have an instruction cache to hold the next instructions expected to be executed. Without this cache, the CPU would have to wait several cycles while an instruction is fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy.

Because caches have limited size, cache management is an important design problem. Careful selection of the cache size and of a replacement policy can result in 80 to 99 percent of all accesses being in the cache, greatly increasing performance.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. On the other hand, transfer of data from disk to memory is usually controlled by the operating system.

2.4.2 Coherency and Consistency

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A is located in file B that is to be incremented by 1, and file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (see Figure 2.5). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to the integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A. The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency,** and it is usually a hardware problem (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In such an environment, several copies (or replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, we must ensure that, when a replica is updated in one place, all other replicas are brought up-to-date as soon as possible. There are various ways to achieve this guarantee.

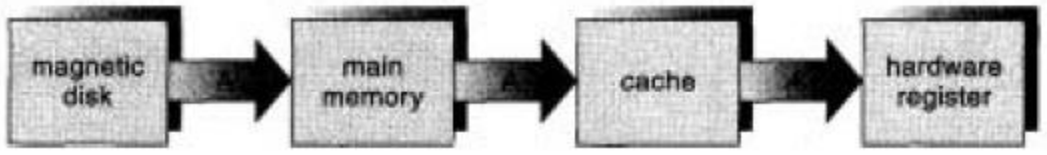

Fig. 2.5 Migration of integer A from disk to register.