The **replacement algorithm** chooses, within the constraints of the mapping function, which block to replace when a new block is to be loaded into the cache and the cache already has all slots filled with other blocks. We would like to replace the block that is least likely to be needed again in the near future. Although it is impossible to identify such a block, a reasonably effective strategy is to replace the block that has been in the cache longest with no reference to it. This policy is referred to as the least-recently-used (LRU) algorithm. Hardware mechanisms are needed to identify the least-recently-used block.

If the contents of a block in the cache are altered, then it is necessary to write it back to main memory before replacing it. The **write policy** dictates when the memory write operation takes place. At one extreme, the writing can occur every time that the block is updated. At the other extreme, the writing occurs only when the block is replaced. The latter policy minimizes memory write operations but leaves main memory in an obsolete state. This can interfere with multiple-processor operation and with direct memory access by I/O hardware modules.

1.7 I/O COMMUNICATION TECHNIQUES

Three techniques are possible for I/O operations:

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

Programmed I/O

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. In the case of programmed I/O, the I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed. For this purpose, the processor periodically checks the status of the I/O module until it finds that the operation is complete.

With this technique, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. I/O software is written in such a way that the processor executes instructions that give it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. Thus, the instruction set includes I/O instructions in the following categories:

- **Control:** Used to activate an external device and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record.
- **Status:** Used to test various status conditions associated with an I/O module and its peripherals.
- **Transfer:** Used to read and/or write data between processor registers and external devices.

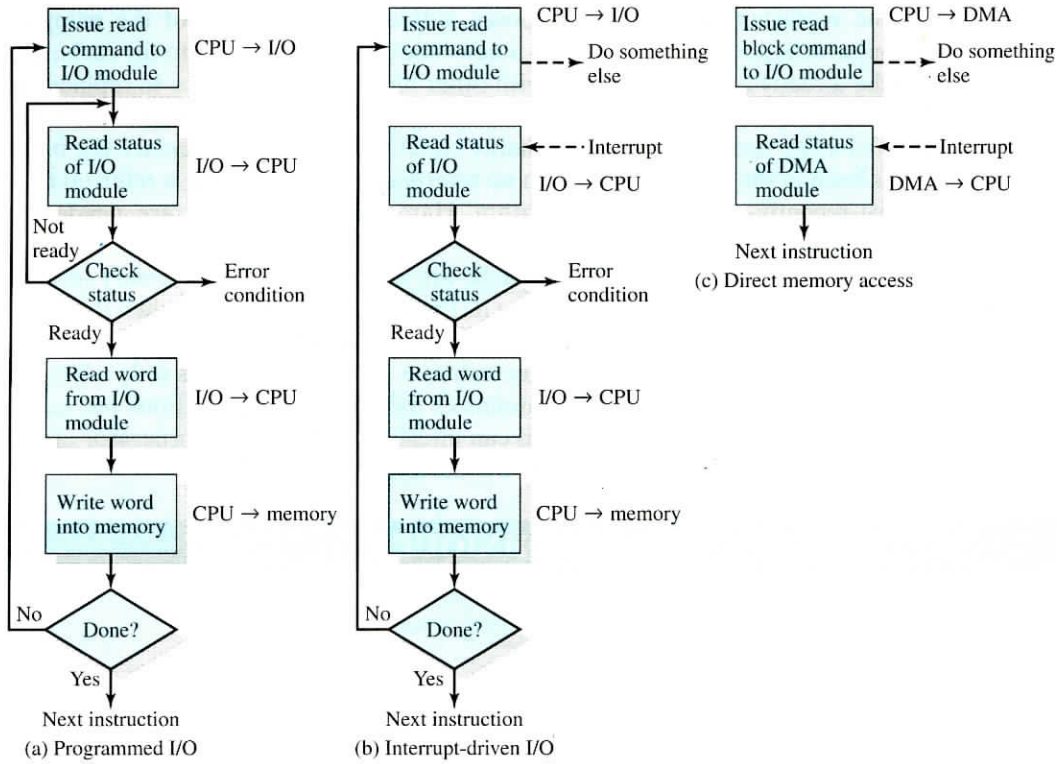

Figure 1.19 Three Techniques for Input of a Block of Data

Figure 1.19a gives an example of the use of programmed I/O to read in a block of data from an external device (e. g., a record from tape) into memory. Data are read in one word (e. g., 16 bits) at a time. For each word that is read in, the processor must remain in a status-checking loop until it determines that the word is available in the I/O module's data register. This flowchart highlights the main disadvantage of this technique: It is a time-consuming process that keeps the processor busy needlessly.

Interrupt-Driven I/O

With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the performance level of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data

are in the module's data register, the module signals an interrupt to the processor over a control line. The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then saves the context (e. g., program counter and processor registers) of the current program and goes off and does something else (e. g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts (Figure 1.7). When the interrupt from the I/O module occurs, the processor saves the context of the program it is currently executing and begins to execute an interrupt-handling program that processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program that had issued the I/O command (or some other program) and resumes execution.

Figure 1.19b shows the use of interrupt-driven I/O for reading in a block of data. Interrupt-driven I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt-driven I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

Almost invariably, there will be multiple I/O modules in a computer system, so mechanisms are needed to enable the processor to determine which device caused the interrupt and to decide, in the case of multiple interrupts, which one to handle first. In some systems, there are multiple interrupt lines, so that each I/O module signals on a different line. Each line will have a different priority. Alternatively, there can be a single interrupt line, but additional lines are used to hold a device address. Again, different devices are assigned different priorities.

Direct Memory Access

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus both of these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA). The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested
- The address of the I/O device involved

- The starting location in memory to read data from or write data to
- The number of words to be read or written

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus the processor is involved only at the beginning and end of the transfer (Figure 1.19c).

The DMA module needs to take control of the bus to transfer data to and from memory. Because of this competition for bus usage, there may be times when the processor needs the bus and must wait for the DMA module. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle (the time it takes to transfer one word across the bus). The overall effect is to cause the processor to execute more slowly during a DMA transfer when processor access to the bus is required. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

1.8 RECOMMENDED READING AND WEB SITES

[STAL06] covers the topics of this chapter in detail. In addition, there are many other texts on computer organization and architecture. Among the more worthwhile texts are the following. [PATT07] is a comprehensive survey; [HENN07], by the same authors, is a more advanced text that emphasizes quantitative aspects of design.

[DENN05] looks at the history of the development and application of the locality principle, making for fascinating reading.

DENN05 Denning, P. "The Locality Principle" *Communications of the ACM*, July 2005.

HENN07 Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 2007.

PATT07 Patterson, D., and Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, CA: Morgan Kaufmann, 2007.

STAL06 Stallings, W. *Computer Organization and Architecture, 7th ed.* Upper Saddle River, NJ: Prentice Hall, 2006.

Recommended Web sites:

- **WWW Computer Architecture Home Page:** A comprehensive index to information relevant to computer architecture researchers, including architecture groups and projects, technical organizations, literature, employment, and commercial information