There are some practical limits to the duration of a page-mode transaction. First, there is an absolute maximum time during which RAS* can remain asserted. The durations of RAS* and CAS* are closely specified to guarantee proper operation of the DRAM. Operating the DRAM with a minimum CAS* cycle time and a maximum RAS* assertion time will yield a practical limitation on the data burst that can be read or written without reloading a new row address. In reality, a common asynchronous DRAM can support over 1,000 back-to-back accesses for a given row-address. DRAM provides its best performance when operated in this manner. The longer the burst, the less overhead is experienced for each byte transferred, because the row-address setup time is amortized across each word in the burst. Cache subsystems on computers help manage the bursty nature of DRAM by swallowing a set of consecutive memory locations into a small SRAM cache where the microprocessor will then have easy access to them later without having to wait for a lengthy DRAM transaction to execute.

The second practical limitation on page-mode transactions, and all DRAM transactions in general, is refresh overhead. The DRAM controller must be smart enough to execute periodic refresh operations at the required frequency. Even if the microprocessor is requesting more data, refresh must take priority to maintain memory integrity. At any given instant in time, a scheduled refresh operation may be delayed slightly to accommodate a CPU request, but not to the point where the controller falls behind and fails to execute the required number of refresh operations. There are a variety ways to initiate a refresh operation, but most involve a so-called *CAS-before-RAS* signaling where the normal sequence of the address strobes is reversed to signal a refresh. Asserting CAS* before RAS* signals the DRAM's internal control logic to perform a row-refresh at the specific row indicated by its internal counter. Following this operation, the refresh counter is incremented in preparation for the next refresh event.

DRAM has numerous advantages over SRAM, but at the price of increased controller complexity and decreased performance in certain applications. DRAMs use multiplexed address buses, which saves pins and enables smaller, less expensive packaging and circuit board wiring. Most DRAMs are manufactured with data bus widths smaller than what is actually used in a computer to save pins. For example, when most computers used 8- or 16-bit data buses, most DRAMs were 1 bit wide. When microprocessors grew to 32 and 64 bit data buses, mainstream DRAMs grew to 4- and then 8-bit widths. This is in contrast to SRAMs, which have generally been offered with wide buses, starting out at 4 bits and then increasing to 72 bits in more modern devices. This width disparity is why most DRAM implementations in computers involve groups of four, eight, or more DRAMs on a single module. In the 1980s, eight 64k $\times$ 1 DRAMs created a 64 kB memory array. Today, eight 32M $\times$ 8 DRAMs create a 256 MB memory array that is 64 bits wide to suit the high-bandwidth 32- or 64-bit microprocessor in your desktop PC.

A key architectural attribute of DRAM is its inherent preference for sequential transactions and, accordingly, its weakness in handling random single transactions. Because of their dense silicon structures and multiplexed address architecture, DRAMs have evolved to provide low-cost bulk memory best suited to burst transactions. The overhead of starting a burst transaction can be negligible when spread across many individual memory words in a burst. However, applications that are not well suited to long bursts may not do very well with DRAM because of the constant startup penalty involved in fetching 1 word versus 1,000 words. Such applications may work better with SRAM. Planning memory architecture involves making these trade-offs between density/cost and performance.

4.7 MULTIPORT MEMORY

Most memory devices, whether volatile or nonvolatile, contain a single interface through which their contents are accessed. In the context of a basic computer system with a single microprocessor, this

single-port architecture is well suited. There are some architectures in which multiple microprocessors or logic blocks require access to the same shared pool of memory. A shared pool of memory can be constructed in a couple of ways. First, conventional DRAM or SRAM can be combined with external logic that takes requests from separate entities (e.g., microprocessors) and arbitrates access to one requestor at a time. When the shared memory pool is large, and when simultaneous access by multiple requesters is not required, arbitration can be an efficient mechanism. However, the complexity of arbitration logic may be excessive for small shared-memory pools, and arbitration does not enable simultaneous access. A means of sharing memory without arbitration logic and with simultaneous access capability is to construct a true multiport memory element.

A multiport memory provides simultaneous access to multiple external entities. Each port may be read/write capable, read-only, or write-only depending on the implementation and application. Multiport memories are generally kept relatively small, because their complexity, and hence their cost, increases significantly as additional ports are added, each with its own decode and control logic. Most multiport memories are *dual-port* elements as shown in Fig. 4.16.

A true dual-port memory places no restrictions on either port's transactions at any given time. It is the responsibility of the engineer to ensure that one requester does not conflict with the other. Conflicts arise when one requester writes a memory location while the other is either reading or writing that same location. If a simultaneous read/write occurs, what data does the reader see? Is it the data before or after the write? Likewise, if two writes proceed at the same time, which one wins? While these riddles could be worked out for specific applications with custom logic, it is safer not to worry about such corner cases. Instead, the system design should avoid such conflicts unless there is a strong reason to the contrary.

One common application of a dual-port memory is sharing information between two microprocessors as shown in Fig. 4.17. A dual-port memory sits between the microprocessors and can be partitioned into a separate message bin, or memory area, for each side. Bin A contains messages written by CPU A and read by CPU B. Bin B contains messages written by CPU B and read by CPU A.

Notification of a waiting message is accomplished via a CPU interrupt, thereby releasing the CPUs from having to constantly poll the memory as they wait for messages to arrive. The entire process might work as follows:

1. CPU A writes a message for CPU B into Bin A.
2. CPU A asserts an interrupt to CPU B indicating the a message is waiting in Bin A.
3. CPU B reads the message in Bin A.
4. CPU B acknowledges the interrupt from CPU A.
5. CPU A releases the interrupt to CPU B.

An implementation like this prevents dual-port memory conflicts because one CPU will not read a message before it has been fully written by the other CPU and neither CPU writes to both bins.

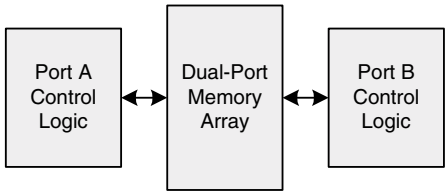

FIGURE 4.16 Dual-port memory.