The ARM architecture provides support for up to 16 co-processors. Co-processors are able to perform load and store operations on their own registers. They can also move data between the co-processor registers and main ARM registers.

An example ARM co-processor is the floating-point unit. The unit occupies two co-processor units in the ARM architecture, numbered 1 and 2, but it appears as a single unit to the programmer. It provides eight 80-bit floating-point data registers, floating-point status registers, and an optional floating-point status register.

## 3.4 MEMORY SYSTEM MECHANISMS

Modern microprocessors do more than just read and write a monolithic memory. Architectural features improve both the speed and capacity of memory systems. Microprocessor clock rates are increasing at a faster rate than memory speeds, such that memories are falling further and further behind microprocessors every day. As a result, computer architects resort to ***caches*** to increase the average performance of the memory system. Although memory capacity is increasing steadily, program sizes are increasing as well, and designers may not be willing to pay for all the memory demanded by an application. ***Modern microprocessor units (MMUs)*** perform address translations that provide a larger virtual memory space in a small physical memory. In this section, we review both caches and MMUs.

### 3.4.1 Caches

Caches are widely used to speed up memory system performance. Many microprocessor architectures include caches as part of their definition. The cache speeds up average memory access time when properly used. It increases the variability of memory access times—accesses in the cache will be fast, while access to locations not cached will be slow. This variability in performance makes it especially important to understand how caches work so that we can better understand how to predict cache performance and factor variabilities into system design.

A cache is a small, fast memory that holds copies of some of the contents of main memory. Because the cache is fast, it provides higher-speed access for the CPU; but since it is small, not all requests can be satisfied by the cache, forcing the system to wait for the slower main memory. Caching makes sense when the CPU is using only a relatively small set of memory locations at any one time; the set of active locations is often called the ***working set***.

Figure 3.6 shows how the cache support reads in the memory system. A ***cache controller*** mediates between the CPU and the memory system comprised of the main memory. The cache controller sends a memory request to the cache and main memory. If the requested location is in the cache, the cache controller forwards the location's contents to the CPU and aborts the main memory request; this condition is known as a ***cache hit***. If the location is not in the cache, the controller waits for the value from main memory and forwards it to the CPU; this situation is known as a ***cache miss***.

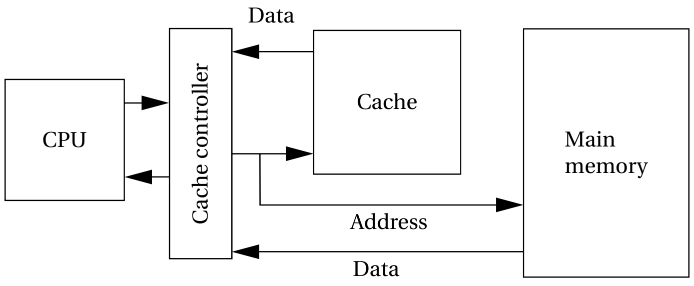

**FIGURE 3.6**

The cache in the memory system.

We can classify cache misses into several types depending on the situation that generated them:

- a ***compulsory miss*** (also known as a **cold miss**) occurs the first time a location is used,
- a ***capacity miss*** is caused by a too-large working set, and
- a ***conflict miss*** happens when two locations map to the same location in the cache.

Even before we consider ways to implement caches, we can write some basic formulas for memory system performance. Let $h$ be the ***hit rate***, the probability that a given memory location is in the cache. It follows that $1 - h$ is the ***miss rate***, or the probability that the location is not in the cache. Then we can compute the average memory access time as

$$t_{av} = ht_{cache} + (1 - h)t_{main}. \tag{3.1}$$

where $t_{cache}$ is the access time of the cache and $t_{main}$ is the main memory access time. The memory access times are basic parameters available from the memory manufacturer. The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators, as is described in more detail in Section 5.6. The best-case memory access time (ignoring cache controller overhead) is $t_{cache}$, while the worst-case access time is $t_{main}$. Given that $t_{main}$ is typically 50–60 ns for DRAM, while $t_{cache}$ is at most a few nanoseconds, the spread between worst-case and best-case memory delays is substantial.

Modern CPUs may use multiple levels of cache as shown in Figure 3.7. The ***first-level cache*** (commonly known as ***L1 cache***) is closest to the CPU, the ***second-level cache (L2 cache)*** feeds the first-level cache, and so on.

The second-level cache is much larger but is also slower. If $h_1$ is the first-level hit rate and $h_2$ is the rate at which access hit the second-level cache but not the first-level cache, then the average access time for a two-level cache system is

$$t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1 - h_1 - h_2)t_{main}. \tag{3.2}$$

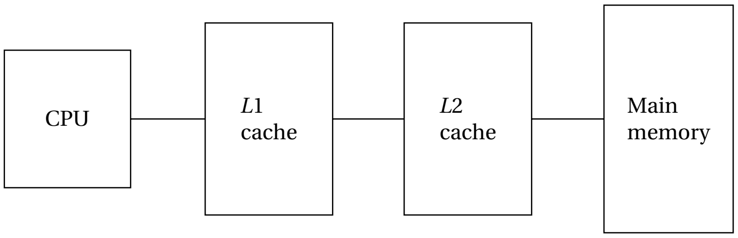

**FIGURE 3.7**

A two-level cache system.

As the program's working set changes, we expect locations to be removed from the cache to make way for new locations. When set-associative caches are used, we have to think about what happens when we throw out a value from the cache to make room for a new value. We do not have this problem in direct-mapped caches because every location maps onto a unique block, but in a set-associative cache we must decide which set will have its block thrown out to make way for the new block. One possible replacement policy is least recently used (LRU), that is, throw out the block that has been used farthest in the past. We can add relatively small amounts of hardware to the cache to keep track of the time since the last access for each block. Another policy is random replacement, which requires even less hardware to implement.

The simplest way to implement a cache is a ***direct-mapped cache***, as shown in Figure 3.8. The cache consists of cache ***blocks***, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is divided into three sections. The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location. If the length of the data field is longer than the minimum addressable unit, then the lowest bits of the address are used as an offset to select the required value from the data field. Given the structure of the cache, there is only one block that must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache.

Writes are slightly more complicated than reads because we have to update main memory as well as the cache. There are several methods by which we can do this. The simplest scheme is known as ***write-through***—every write changes both the cache and the corresponding main memory location (usually through a write buffer). This scheme ensures that the cache and main memory are consistent, but may generate some additional main memory traffic. We can reduce the number of times we write to main memory by using a ***write-back*** policy: If we write only when we remove a location from the cache, we eliminate the writes when a location is written several times before it is removed from the cache.

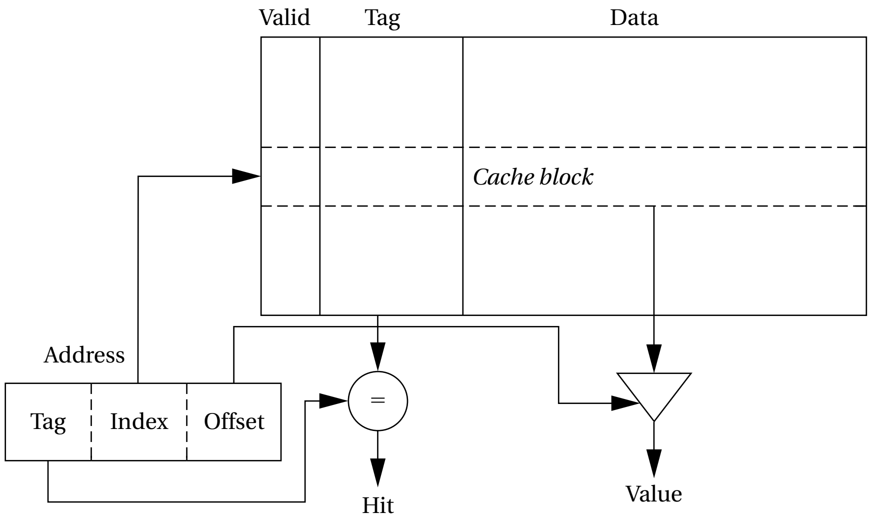

**FIGURE 3.8**

A direct-mapped cache.

The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power due to its simple scheme for mapping the cache onto main memory. Consider a direct-mapped cache with four blocks, in which locations 0, 1, 2, and 3 all map to different blocks. But locations 4, 8, 12, ... all map to the same block as location 0; locations 1, 5, 9, 13, ... all map to a single block; and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. As seen in Section 5.6, this can create program performance problems.

The limitations of the direct-mapped cache can be reduced by going to the ***set-associative*** cache structure shown in Figure 3.9. A set-associative cache is characterized by the number of ***banks*** or ***ways*** it uses, giving an *n*-way set-associative cache. A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit. Although memory locations map onto blocks using the same function, there are *n* separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that happen to map onto the same cache block. The set-associative cache structure incurs a little extra overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

The set-associative cache generally provides higher hit rates than the direct-mapped cache because conflicts between a small number of locations can be resolved within the cache. The set-associative cache is somewhat slower, so the CPU designer has to be careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program

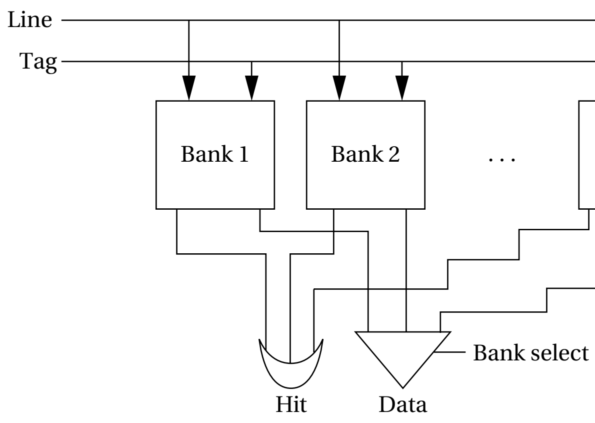

**FIGURE 3.9**

A set-associative cache.

design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs have good behavior in the cache. It is relatively easy to determine when two memory locations will conflict in a direct-mapped cache. Conflicts in a set-associative cache are more subtle, and so the behavior of a set-associative cache is more difficult to analyze for both humans and programs. Example 3.8 compares the behavior of direct-mapped and set-associative caches.

## Example 3.8

### *Direct-mapped vs. set-associative caches*

For simplicity, let's consider a very simple caching scheme. We use 2 bits of the address as the tag. We compare a direct-mapped cache with four blocks and a two-way set-associative cache with four sets, and we use LRU replacement to make it easy to compare the two caches.

A 3-bit address is used for simplicity. The contents of the memory follow:

| Address | Data | Address | Data |
|---|---|---|---|
| 000 | 0101 | 100 | 1000 |
| 001 | 1111 | 101 | 0001 |
| 010 | 0000 | 110 | 1010 |
| 011 | 0110 | 111 | 0100 |

We will give each cache the same pattern of addresses (in binary to simplify picking out the index): 001, 010, 011, 100, 101, and 111.

To understand how the direct-mapped cache works, let's see how its state evolves.

After 001 access:

| Block | Tag | Data |
|---|---|---|
| 00 | — | — |
| 01 | 0 | 1111 |
| 10 | — | — |
| 11 | — | — |

After 010 access:

| Block | Tag | Data |
|---|---|---|
| 00 | — | — |
| 01 | 0 | 1111 |
| 10 | 0 | 0000 |
| 11 | — | — |

After 011 access:

| Block | Tag | Data |
|---|---|---|
| 00 | — | — |
| 01 | 0 | 1111 |
| 10 | 0 | 0000 |
| 11 | 0 | 0110 |

After 100 access (notice that the tag bit for this entry is 1):

| Block | Tag | Data |
|---|---|---|
| 00 | 1 | 1000 |
| 01 | 0 | 1111 |
| 10 | 0 | 0000 |
| 11 | 0 | 0110 |

After 101 access (overwrites the 01 block entry):

| Block | Tag | Data |
|---|---|---|
| 00 | 1 | 1000 |
| 01 | 1 | 0001 |
| 10 | 0 | 0000 |
| 11 | 0 | 0110 |

After 111 access (overwrites the 11 block entry):

| Block | Tag | Data |
|---|---|---|
| 00 | 1 | 1000 |
| 01 | 1 | 0001 |
| 10 | 0 | 0000 |
| 11 | 1 | 0100 |

We can use a similar procedure to determine what ends up in the two-way set-associative cache. The only difference is that we have some freedom when we have to replace a block with new data. To make the results easy to understand, we use a least-recently-used replacement policy. For starters, let's make each way the size of the original direct-mapped cache. The final state of the two-way set-associative cache follows:

| Block | Way 0 tag | Way 0 data | Way 1 tag | Way 1 data |
|---|---|---|---|---|
| 00 | 1 | 1000 | — | — |
| 01 | 0 | 1111 | 1 | 0001 |
| 10 | 0 | 0000 | — | — |
| 11 | 0 | 0110 | 1 | 0100 |

Of course, this is not a fair comparison for performance because the two-way set-associative cache has twice as many entries as the direct-mapped cache. Let's use a two-way, set-associative cache with two sets, giving us four blocks, the same number as in the direct-mapped cache. In this case, the index size is reduced to 1 bit and the tag grows to 2 bits.

| Block | Way 0 tag | Way 0 data | Way 1 tag | Way 1 data |
|---|---|---|---|---|
| 0 | 01 | 0000 | 10 | 1000 |
| 1 | 00 | 0111 | 11 | 0100 |

In this case, the cache contents are significantly different than for either the direct-mapped cache or the four-block, two-way set-associative cache.

The CPU knows when it is fetching an instruction (the PC is used to calculate the address, either directly or indirectly) or data. We can therefore choose whether to cache instructions, data, or both. If cache space is limited, instructions are the highest priority for caching because they will usually provide the highest hit rates. A cache that holds both instructions and data is called a ***unified cache***.

Various ARM implementations use different cache sizes and organizations [Fur96]. The ARM600 includes a 4-KB, 64-way (wow!) unified instruction/data cache. The StrongARM uses a 16-KB, 32-way instruction cache with a 32-byte block and a 16-KB, 32-way data cache with a 32-byte block; the data cache uses a write-back strategy.

The C5510, one of the models of C55x, uses a 16-K byte instruction cache organized as a two-way set-associative cache with four 32-bit words per line. The instruction cache can be disabled by software if desired. It also includes two RAM sets that are designed to hold large contiguous blocks of code. Each RAM set can hold up to 4-K bytes of code organized as 256 lines of four 32-bit words per line. Each RAM has a tag that specifies what range of addresses are in the RAM; it also includes a tag valid field to show whether the RAM is in use and line valid bits for each line.

### 3.4.2 Memory Management Units and Address Translation

A MMU translates addresses between the CPU and physical memory. This translation process is often known as ***memory mapping*** since addresses are mapped from a logical space into a physical space. MMUs in embedded systems appear primarily in the host processor. It is helpful to understand the basics of MMUs for embedded systems complex enough to require them.

Many DSPs, including the C55x, do not use MMUs. Since DSPs are used for compute-intensive tasks, they often do not require the hardware assist for logical address spaces.

Early computers used MMUs to compensate for limited address space in their instruction sets. When memory became cheap enough that physical memory could be larger than the address space defined by the instructions, MMUs allowed software to manage multiple programs in a single physical memory, each with its own address space.

Because modern CPUs typically do not have this limitation, MMUs are used to provide ***virtual addressing***. As shown in Figure 3.10, the MMU accepts logical addresses from the CPU. Logical addresses refer to the program's abstract address space but do not correspond to actual RAM locations. The MMU translates them from tables to physical addresses that do correspond to RAM. By changing the MMU's tables, you can change the physical location at which the program resides without modifying the program's code or data. (We must, of course, move the program in main memory to correspond to the memory mapping change.)

Furthermore, if we add a secondary storage unit such as flash or a disk, we can eliminate parts of the program from main memory. In a virtual memory system, the MMU keeps track of which logical addresses are actually resident in main memory; those that do not reside in main memory are kept on the secondary storage device.

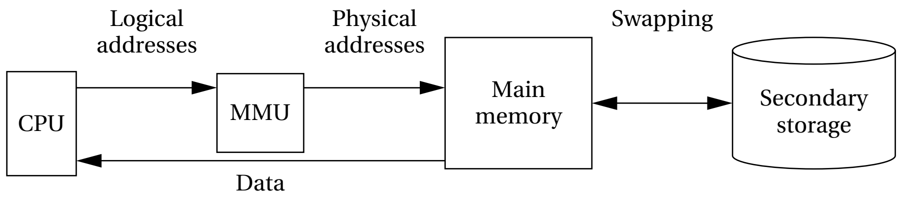

**FIGURE 3.10**

A virtually addressed memory system.

When the CPU requests an address that is not in main memory, the MMU generates an exception called a ***page fault***. The handler for this exception executes code that reads the requested location from the secondary storage device into main memory. The program that generated the page fault is restarted by the handler only after

- the required memory has been read back into main memory, and
- the MMU's tables have been updated to reflect the changes.

Of course, loading a location into main memory will usually require throwing something out of main memory. The displaced memory is copied into secondary storage before the requested location is read in. As with caches, LRU is a good replacement policy.

There are two styles of address translation: ***segmented*** and ***paged***. Each has advantages and the two can be combined to form a segmented, paged addressing scheme. As illustrated in Figure 3.11, segmenting is designed to support a large, arbitrarily sized region of memory, while pages describe small, equally sized regions. A segment is usually described by its start address and size, allowing different segments to be of different sizes. Pages are of uniform size, which simplifies the hardware required for address translation. A segmented, paged scheme is created by dividing each segment into pages and using two steps for address translation. Paging introduces the possibility of ***fragmentation*** as program pages are scattered around physical memory.

In a simple segmenting scheme, shown in Figure 3.12, the MMU would maintain a segment register that describes the currently active segment. This register would point to the base of the current segment. The address extracted from an instruction (or from any other source for addresses, such as a register) would be used as the offset for the address. The physical address is formed by adding the segment base to the offset. Most segmentation schemes also check the physical address against the upper limit of the segment by extending the segment register to include the segment size and comparing the offset to the allowed size.

The translation of paged addresses requires more MMU state but a simpler calculation. As shown in Figure 3.13, the logical address is divided into two sections, including a page number and an offset. The page number is used as an index into a page table, which stores the physical address for the start of each page. However,

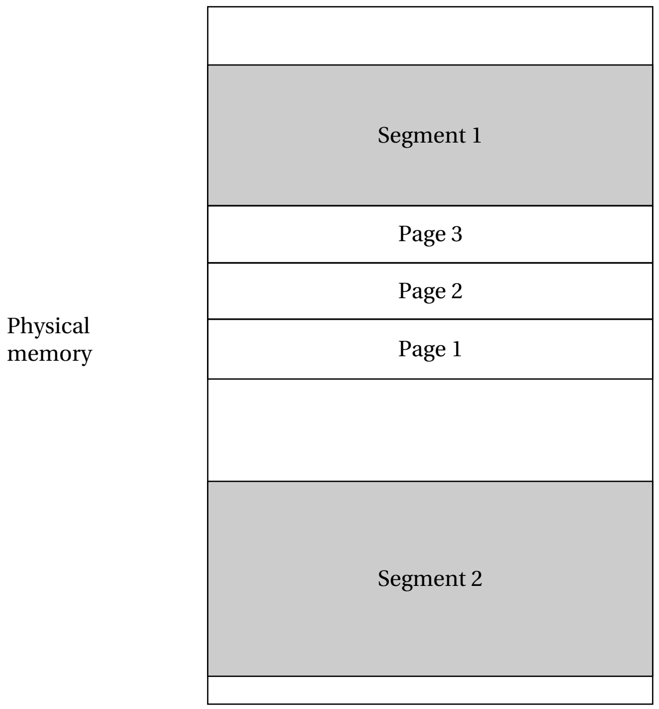

**FIGURE 3.11**

Segments and pages.

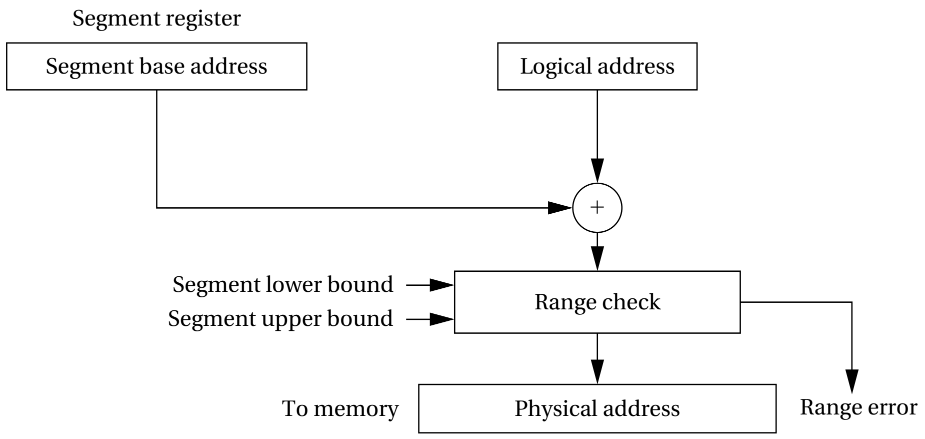

**FIGURE 3.12**

Address translation for a segment.

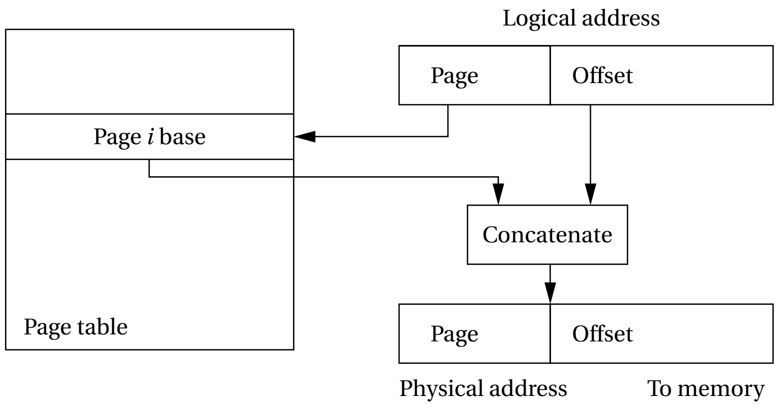

**FIGURE 3.13**

Address translation for a page.

since all pages have the same size and it is easy to ensure that page boundaries fall on the proper boundaries, the MMU simply needs to concatenate the top bits of the page starting address with the bottom bits from the page offset to form the physical address. Pages are small, typically between 512 bytes and 4 KB. As a result, the page table is large for an architecture with a large address space. The page table is normally kept in main memory, which means that an address translation requires memory access.

The page table may be organized in several ways, as shown in Figure 3.14. The simplest scheme is a flat table. The table is indexed by the page number and each entry holds the page descriptor. A more sophisticated method is a tree. The root entry of the tree holds pointers to pointer tables at the next level of the tree; each pointer table is indexed by a part of the page number. We eventually (after three levels, in this case) arrive at a descriptor table that includes the page descriptor we are interested in. A tree-structured page table incurs some overhead for the pointers, but it allows us to build a partially populated tree. If some part of the address space is not used, we do not need to build the part of the tree that covers it.

The efficiency of paged address translation may be increased by caching page translation information. A cache for address translation is known as a ***translation lookaside buffer (TLB)***. The MMU reads the TLB to check whether a page number is currently in the TLB cache and, if so, uses that value rather than reading from memory.

Virtual memory is typically implemented in a paging or segmented, paged scheme so that only page-sized regions of memory need to be transferred on a page fault. Some extensions to both segmenting and paging are useful for virtual memory:

- At minimum, a present bit is necessary to show whether the logical segment or page is currently in physical memory.

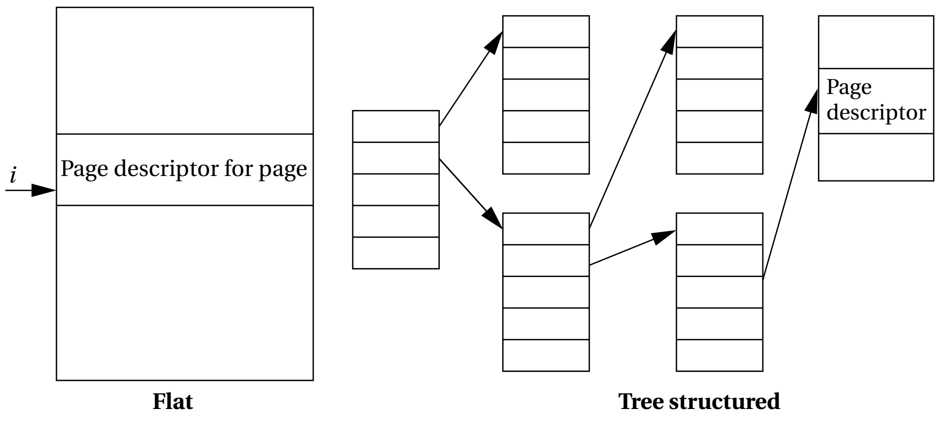

**FIGURE 3.14**

Alternative schemes for organizing page tables.

- A dirty bit shows whether the page/segment has been written to. This bit is maintained by the MMU, since it knows about every write performed by the CPU.
- Permission bits are often used. Some pages/segments may be readable but not writable. If the CPU supports modes, pages/segments may be accessible by the supervisor but not in user mode.

A data or instruction cache may operate either on logical or physical addresses, depending on where it is positioned relative to the MMU.

A MMU is an optional part of the ARM architecture. The ARM MMU supports both virtual address translation and memory protection; the architecture requires that the MMU be implemented when cache or write buffers are implemented. The ARM MMU supports the following types of memory regions for address translation:

- a ***section*** is a 1-MB block of memory,
- a ***large page*** is 64 KB, and
- a ***small page*** is 4 KB.

An address is marked as section mapped or page mapped. A two-level scheme is used to translate addresses. The first-level table, which is pointed to by the Translation Table Base register, holds descriptors for section translation and pointers to the second-level tables. The second-level tables describe the translation of both large and small pages. The basic two-level process for a large or small page is illustrated in Figure 3.15. The details differ between large and small pages, such as the size of the second-level table index. The first- and second-level pages also contain access control bits for virtual memory and protection.

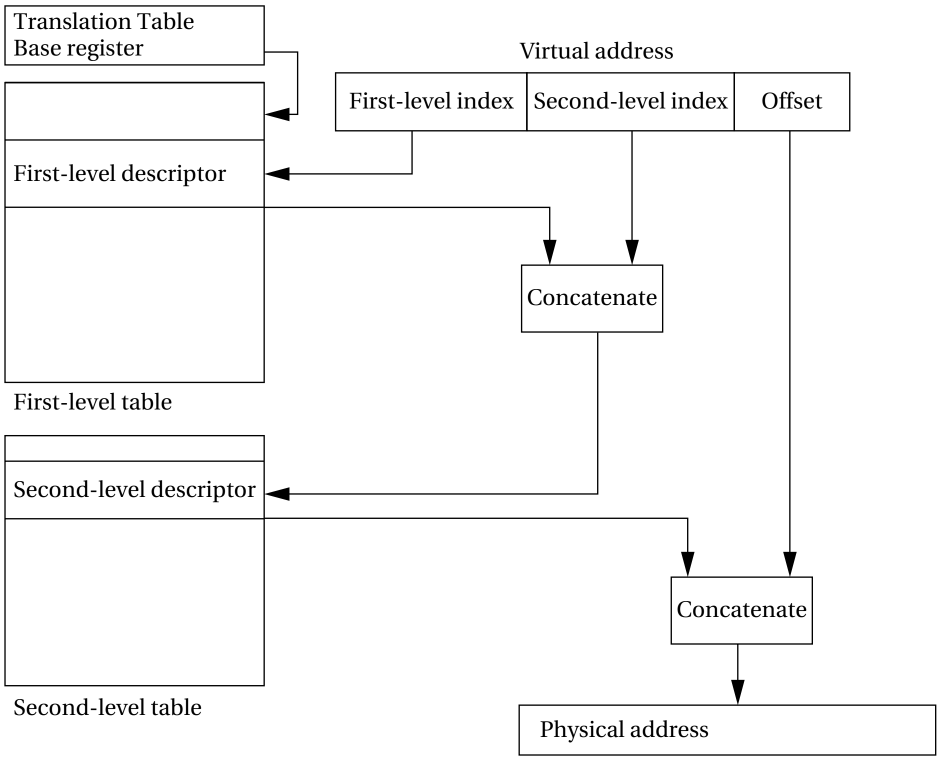

**FIGURE 3.15**

ARM two-stage address translation.

## 3.5 CPU PERFORMANCE

Now that we have an understanding of the various types of instructions that CPUs can execute, we can move on to a topic particularly important in embedded computing: How fast can the CPU execute instructions? In this section, we consider three factors that can substantially influence program performance: pipelining and caching.

### 3.5.1 Pipelining

Modern CPUs are designed as ***pipelined*** machines in which several instructions are executed in parallel. Pipelining greatly increases the efficiency of the CPU. But like any pipeline, a CPU pipeline works best when its contents flow smoothly. Some sequences of instructions can disrupt the flow of information in the pipeline and, temporarily at least, slow down the operation of the CPU.

CHAPTER

4

# Bus-Based Computer Systems

- CPU buses, I/O devices, and interfacing.
- The CPU system as a framework for understanding design methodology.
- System-level performance and power consumption.
- Development environments and debugging.
- An alarm clock design.

## INTRODUCTION

In this chapter, we concentrate on ***bus-based computer systems*** created using microprocessors, I/O devices, and memory components. The microprocessor is an important element of the embedded computing system, but it cannot do its job without memories and I/O devices. We need to understand how to interconnect microprocessors and devices using the CPU bus. Luckily, there are many similarities between the platforms required for different applications, so we can extract some generally useful principles by examining a few basic concepts.

In the next section, we study the CPU bus, which forms the backbone of the hardware system. Because memories are very important components of embedded platforms, Section 4.2 studies types of memory devices. Section 4.3 introduces a variety of types of I/O devices. Section 4.4 introduces basic techniques for interfacing memories and I/O devices to the CPU bus. Section 4.5 focuses on the structure of the complete platform, while Section 4.6 considers development and debugging. Section 4.7 looks at system-level performance analysis for bus-based systems. Section 4.8 wraps up with an alarm clock as a design example.

## 4.1 THE CPU BUS

A computer system encompasses much more than the CPU; it also includes memory and I/O devices. The ***bus*** is the mechanism by which the CPU communicates with memory and devices. A bus is, at a minimum, a collection of wires, but the bus also defines a protocol by which the CPU, memory, and devices communicate. One of the major roles of the bus is to provide an interface to memory. (Of course, I/O devices also connect to the bus.) Based on understanding of the bus, we study the characteristics of memory components in this section.

### 4.1.1 Bus Protocols

The basic building block of most bus protocols is the ***four-cycle handshake***, illustrated in Figure 4.1. The handshake ensures that when two devices want to communicate, one is ready to transmit and the other is ready to receive. The handshake uses a pair of wires dedicated to the handshake: ***enq*** (meaning enquiry) and ***ack*** (meaning acknowledge). Extra wires are used for the data transmitted during the handshake. The four cycles are described below.

1. *Device 1* raises its output to signal an enquiry, which tells *device 2* that it should get ready to listen for data.

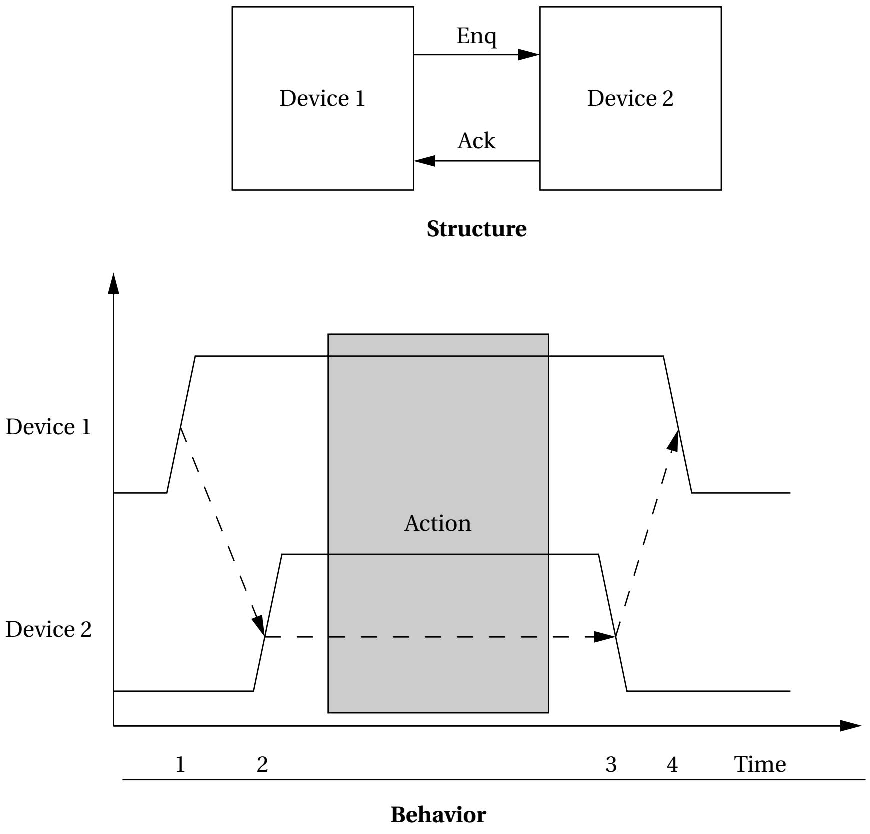

**FIGURE 4.1**

The four-cycle handshake.

2. When *device 2* is ready to receive, it raises its output to signal an acknowledgment. At this point, *devices 1* and *2* can transmit or receive.
3. Once the data transfer is complete, *device 2* lowers its output, signaling that it has received the data.
4. After seeing that *ack* has been released, *device 1* lowers its output.

At the end of the handshake, both handshaking signals are low, just as they were at the start of the handshake. The system has thus returned to its original state in readiness for another handshake-enabled data transfer.

Microprocessor buses build on the handshake for communication between the CPU and other system components. The term *bus* is used in two ways. The most basic use is as a set of related wires, such as address wires. However, the term may also mean a protocol for communicating between components. To avoid confusion, we will use the term ***bundle*** to refer to a set of related signals. The fundamental bus operations are reading and writing. Figure 4.2 shows the structure of a typical bus that supports reads and writes. The major components follow:

- *Clock* provides synchronization to the bus components,
- *R/W* is true when the bus is reading and false when the bus is writing,
- *Address* is an *a*-bit bundle of signals that transmits the address for an access,
- *Data* is an *n*-bit bundle of signals that can carry data to or from the CPU, and
- *Data ready* signals when the values on the data bundle are valid.

All transfers on this basic bus are controlled by the CPU—the CPU can read or write a device or memory, but devices or memory cannot initiate a transfer. This is reflected by the fact that *R/W* and address are unidirectional signals, since only the CPU can determine the address and direction of the transfer.

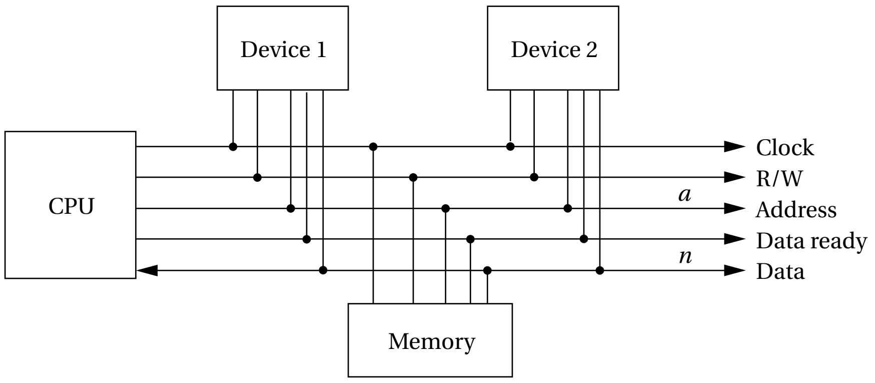

**FIGURE 4.2**

A typical microprocessor bus.

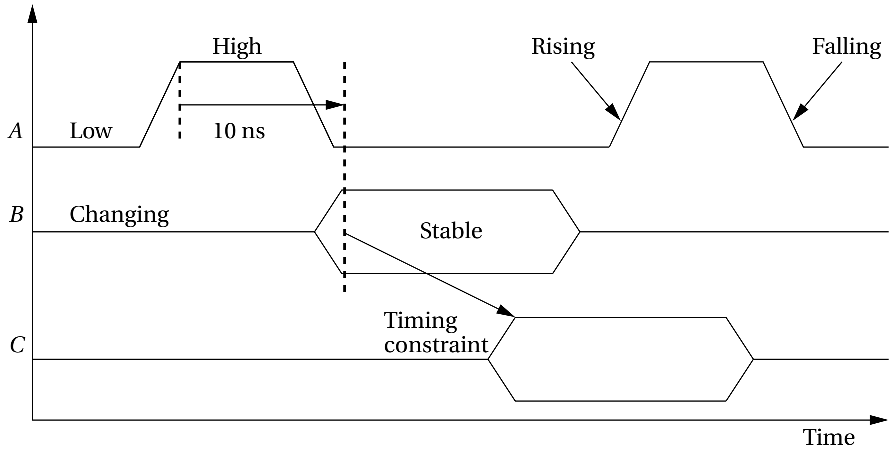

**FIGURE 4.3**

Timing diagram notation.

The behavior of a bus is most often specified as a ***timing diagram***. A timing diagram shows how the signals on a bus vary over time, but since values like the address and data can take on many values, some standard notation is used to describe signals, as shown in Figure 4.3. *A*'s value is known at all times, so it is shown as a standard waveform that changes between zero and one. *B* and *C* alternate between ***changing*** and ***stable*** states. A stable signal has, as the name implies, a stable value that could be measured by an oscilloscope, but the exact value of that signal does not matter for purposes of the timing diagram. For example, an address bus may be shown as stable when the address is present, but the bus's timing requirements are independent of the exact address on the bus. A signal can go between a known 0/1 state and a stable/changing state. A changing signal does not have a stable value. Changing signals should not be used for computation. To be sure that signals go to their proper values at the proper times, timing diagrams sometimes show ***timing constraints***. We draw timing constraints in two different ways, depending on whether we are concerned with the amount of time between events or only the order of events. The timing constraint from *A* to *B*, for example, shows that *A* must go high before *B* becomes stable. The constraint from *A* to *B* also has a time value of 10 ns, indicating that *A* goes high at least 10 ns before *B* goes stable.

Figure 4.4 shows a timing diagram for the example bus. The diagram shows a read and a write. Timing constraints are shown only for the read operation, but similar constraints apply to the write operation. The bus is normally in the read mode since that does not change the state of any of the devices or memories. The CPU can then ignore the bus data lines until it wants to use the results of a read. Notice also that the direction of data transfer on bidirectional lines is not specified in the timing diagram. During a read, the external device or memory is sending a value on the data lines, while during a write the CPU is controlling the data lines.

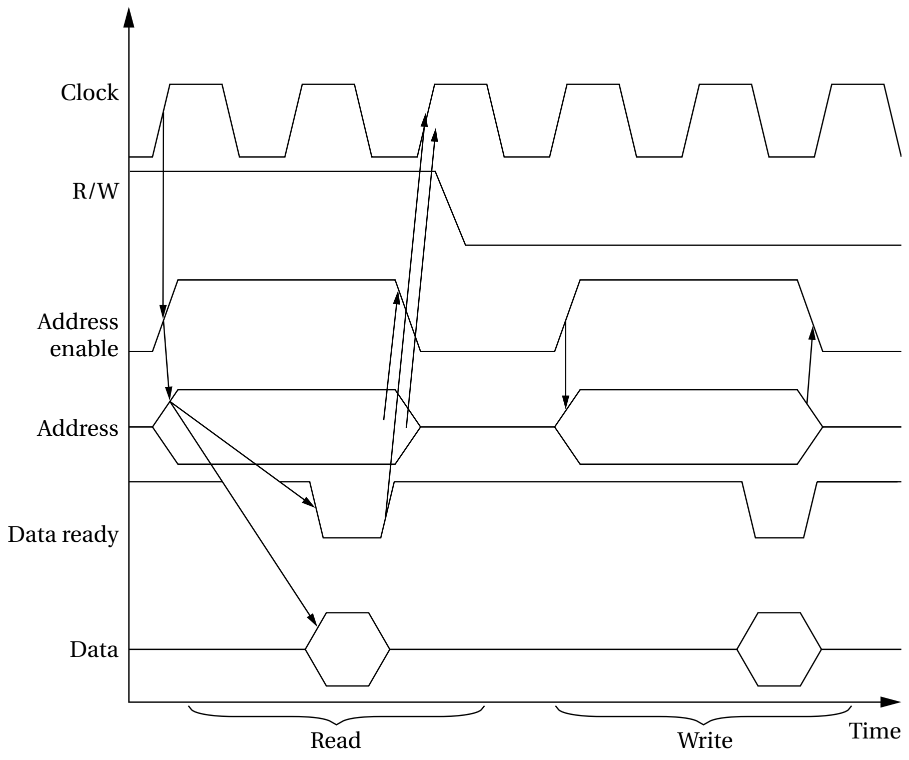

**FIGURE 4.4**

Timing diagram for the example bus.

With practice, we can see the sequence of operations for a read on the timing diagram as follows:

- A read or write is initiated by setting address enable high after the clock starts to rise. We set $R/W = 1$ to indicate a read, and the address lines are set to the desired address.
- One clock cycle later, the memory or device is expected to assert the data value at that address on the data lines. Simultaneously, the external device specifies that the data are valid by pulling down the *data ready* line. This line is ***active low***, meaning that a logically true value is indicated by a low voltage, in order to provide increased immunity to electrical noise.
- The CPU is free to remove the address at the end of the clock cycle and must do so before the beginning of the next cycle. The external device has a similar requirement for removing the data value from the data lines.

The write operation has a similar timing structure. The read/write sequence does illustrate that timing constraints are required on the transition of the *R/W* signal between read and write states. The signal must, of course, remain stable within a read or write. As a result there is a restricted time window in which the CPU can change between read and write modes.

The handshake that tells the CPU and devices when data are to be transferred is formed by data ready for the acknowledge side, but is implicit for the enquiry side. Since the bus is normally in read mode, enq does not need to be asserted, but the acknowledge must be provided by *data ready*.

The *data ready* signal allows the bus to be connected to devices that are slower than the bus. As shown in Figure 4.5, the external device need not immediately assert *data ready*. The cycles between the minimum time at which data can be

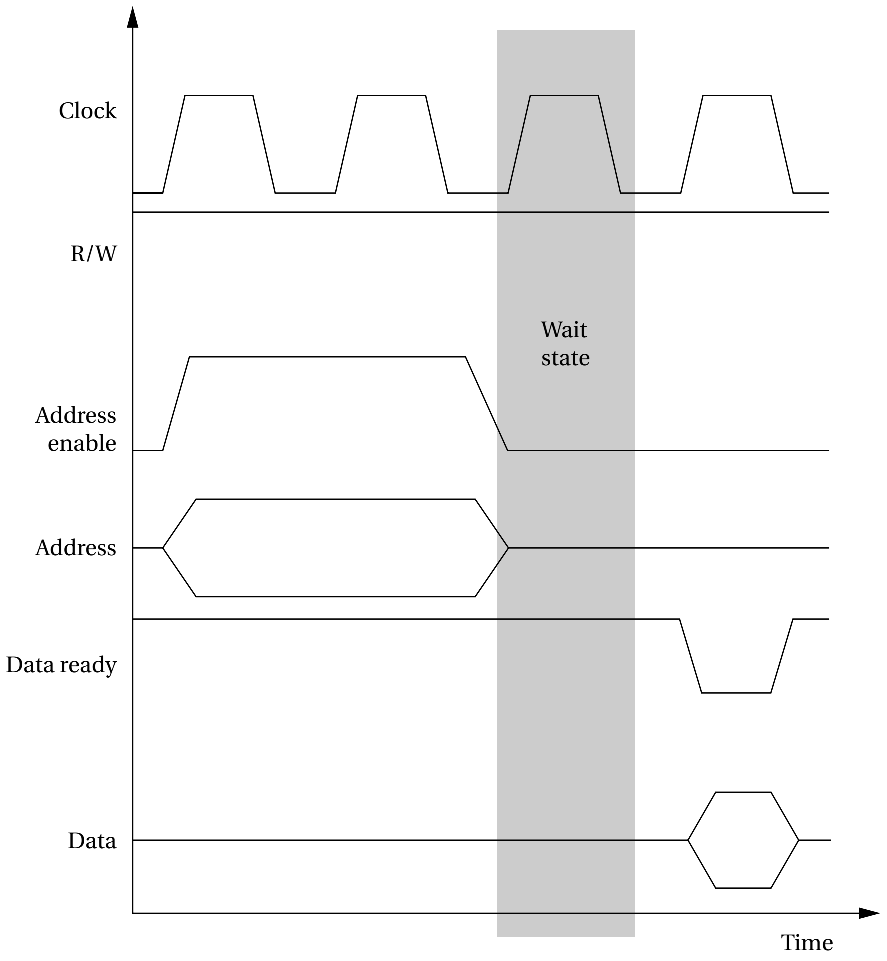

**FIGURE 4.5**

A wait state on a read operation.

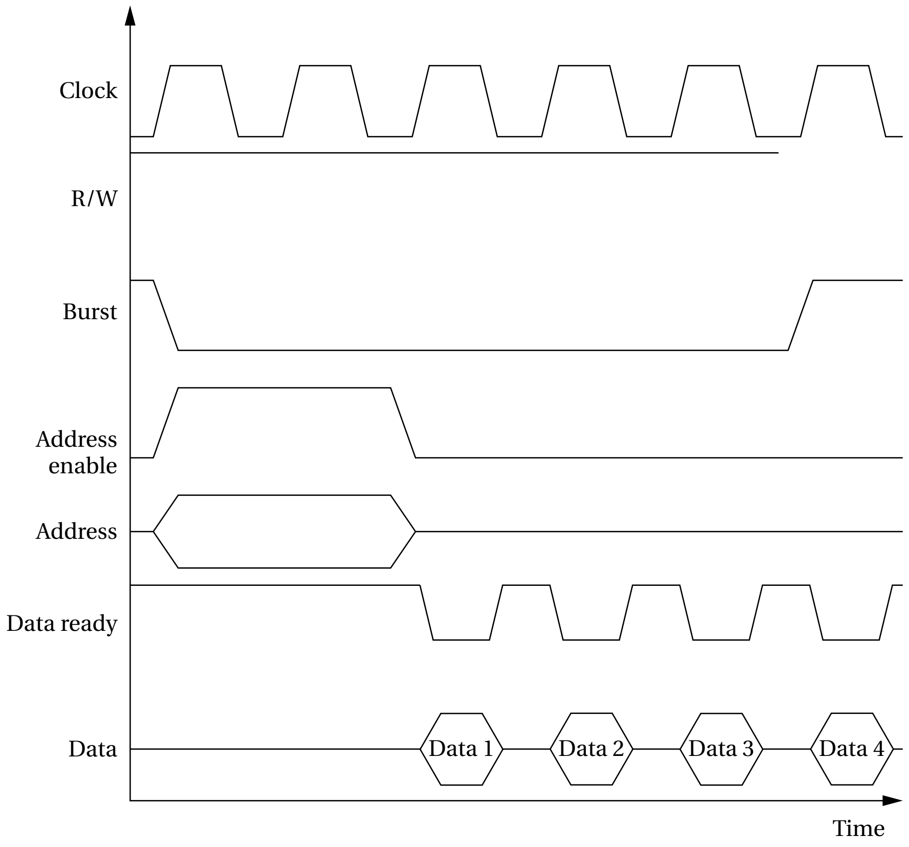

**FIGURE 4.6**

A burst read transaction.

asserted and when it is actually asserted are known as ***wait states***. Wait states are commonly used to connect slow, inexpensive memories to buses.

We can also use the bus handshaking signals to perform ***burst transfers***, as illustrated in Figure 4.6. In this burst read transaction, the CPU sends one address but receives a sequence of data values. We add an extra line to the bus, called *burst9* here, which signals when a transaction is actually a burst. Releasing the *burst9* signal tells the device that enough data has been transmitted. To stop receiving data after the end of *data 4*, the CPU releases the *burst9* signal at the end of *data 3* since the device requires some time to recognize the end of the burst. Those values come from successive memory locations starting at the given address.

Some buses provide ***disconnected transfers***. In these buses, the request and response are separate. A first operation requests the transfer. The bus can then be used for other operations. The transfer is completed later, when the data are ready.

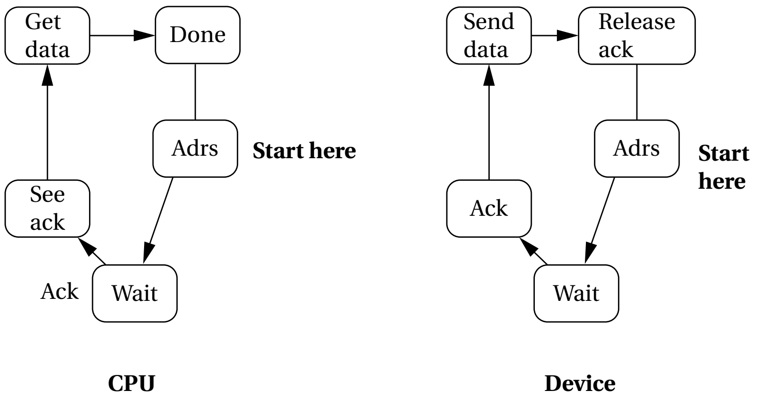

**FIGURE 4.7**

State diagrams for the bus read transaction.

The state machine view of the bus transaction is also helpful and a useful complement to the timing diagram. Figure 4.7 shows the CPU and device state machines for the read operation. As with a timing diagram, we do not show all the possible values of address and data lines but instead concentrate on the transitions of control signals. When the CPU decides to perform a read transaction, it moves to a new state, sending bus signals that cause the device to behave appropriately. The device's state transition graph captures its side of the protocol.

Some buses have data bundles that are smaller than the natural word size of the CPU. Using fewer data lines reduces the cost of the chip. Such buses are easiest to design when the CPU is natively addressable. A more complicated protocol hides the smaller data sizes from the instruction execution unit in the CPU. Byte addresses are sequentially sent over the bus, receiving one byte at a time; the bytes are assembled inside the CPU's bus logic before being presented to the CPU proper.

Some buses use multiplexed address and data. As shown in Figure 4.8, additional control lines are provided to tell whether the value on the address/data lines is an address or data. Typically, the address comes first on the combined address/data lines, followed by the data. The address can be held in a register until the data arrive so that both can be presented to the device (such as a RAM) at the same time.

### 4.1.2 DMA

Standard bus transactions require the CPU to be in the middle of every read and write transaction. However, there are certain types of data transfers in which the CPU does not need to be involved. For example, a high-speed I/O device may want to transfer a block of data into memory. While it is possible to write a program that alternately reads the device and writes to memory, it would be faster to eliminate the CPU's involvement and let the device and memory communicate directly. This

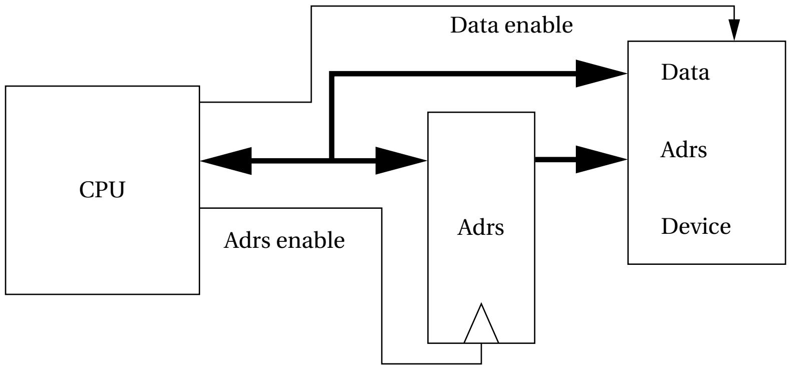

**FIGURE 4.8**

Bus signals for multiplexing address and data.

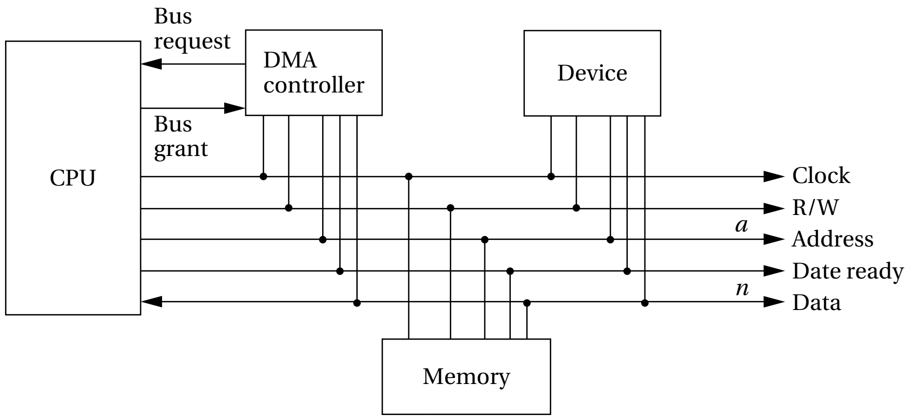

**FIGURE 4.9**

A bus with a DMA controller.

capability requires that some unit other than the CPU be able to control operations on the bus.

***Direct memory access (DMA)*** is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a ***DMA controller***, which requests control of the bus from the CPU. After gaining control, the DMA controller performs read and write operations directly between devices and memory.

Figure 4.9 shows the configuration of a bus with a DMA controller. The DMA requires the CPU to provide two additional bus signals:

- The ***bus request*** is an input to the CPU through which DMA controllers ask for ownership of the bus.
- The ***bus grant*** signals that the bus has been granted to the DMA controller.

A device that can initiate its own bus transfer is known as a ***bus master***. Devices that do not have the capability to be ***bus masters*** do not need to connect to a bus request and bus grant. The DMA controller uses these two signals to gain control of the bus using a classic four-cycle handshake. The bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready.

The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the other bus signals: R/W, address, and so on. Upon becoming bus master, the DMA controller has control of all bus signals (except, of course, for bus request and bus grant).

Once the DMA controller is bus master, it can perform reads and writes using the same bus protocol as with any CPU-driven bus transaction. Memory and devices do not know whether a read or write is performed by the CPU or by a DMA controller. After the transaction is finished, the DMA controller returns the bus to the CPU by deasserting the bus request, causing the CPU to deassert the bus grant.

The CPU controls the DMA operation through registers in the DMA controller. A typical DMA controller includes the following three registers:

- A starting address register specifies where the transfer is to begin.
- A length register specifies the number of words to be transferred.
- A status register allows the DMA controller to be operated by the CPU.

The CPU initiates a DMA transfer by setting the starting address and length registers appropriately and then writing the status register to set its start transfer bit. After the DMA operation is complete, the DMA controller interrupts the CPU to tell it that the transfer is done.

What is the CPU doing during a DMA transfer? It cannot use the bus. As illustrated in Figure 4.10, if the CPU has enough instructions and data in the cache and registers, it may be able to continue doing useful work for quite some time and may not notice the DMA transfer. But once the CPU needs the bus, it stalls until the DMA controller returns bus mastership to the CPU.

To prevent the CPU from idling for too long, most DMA controllers implement modes that occupy the bus for only a few cycles at a time. For example, the transfer may be made 4, 8, or 16 words at a time. As illustrated in Figure 4.11, after each block, the DMA controller returns control of the bus to the CPU and goes to sleep for a preset period, after which it requests the bus again for the next block transfer.

## 4.1.3 System Bus Configurations

A microprocessor system often has more than one bus. As shown in Figure 4.12, high-speed devices may be connected to a high-performance bus, while lower-speed

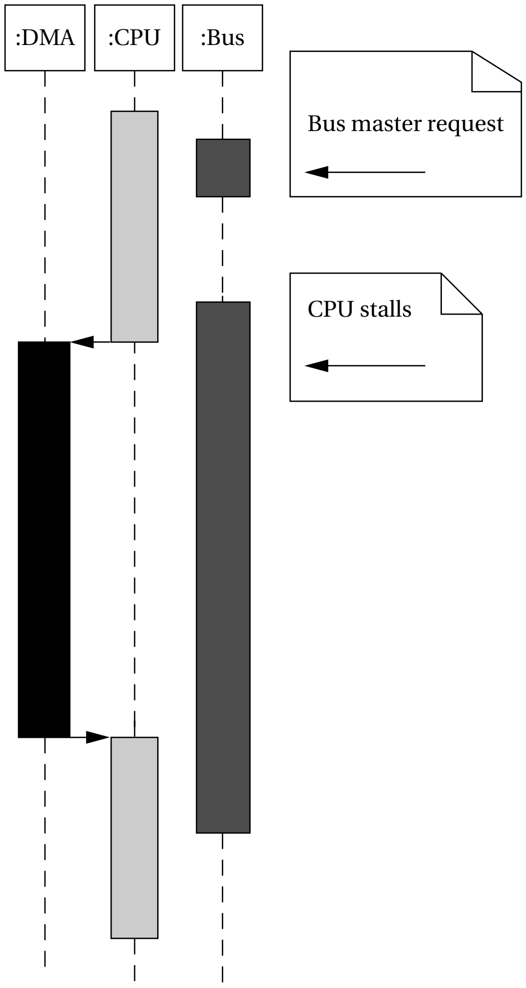

**FIGURE 4.10**

UML sequence diagram of system activity around a DMA transfer.

devices are connected to a different bus. A small block of logic known as a ***bridge*** allows the buses to connect to each other. There are several good reasons to use multiple buses and bridges:

- Higher-speed buses may provide wider data connections.
- A high-speed bus usually requires more expensive circuits and connectors. The cost of low-speed devices can be held down by using a lower-speed, lower-cost bus.

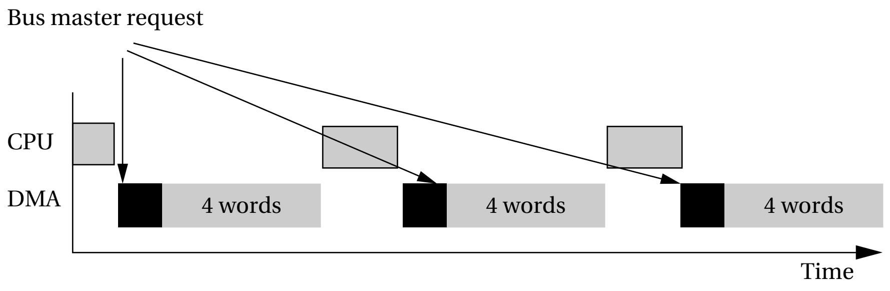

**FIGURE 4.11**

Cyclic scheduling of a DMA request.

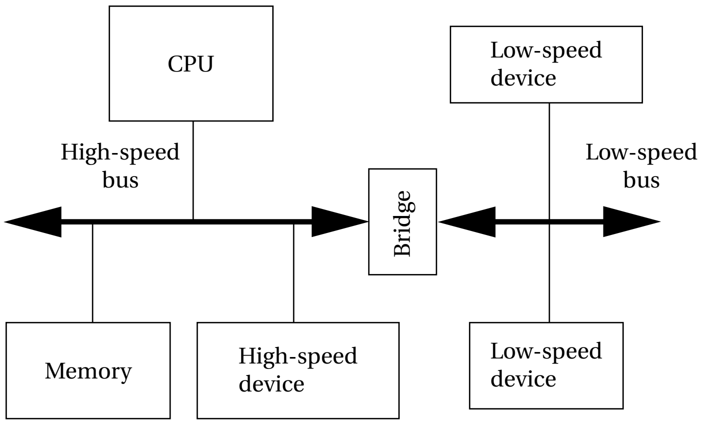

**FIGURE 4.12**

A multiple bus system.

- The bridge may allow the buses to operate independently, thereby providing some parallelism in I/O operations.

In Section 4.5.3, we see that PCs often use this methodology.

Let's consider the operation of a bus bridge between what we will call a fast bus and a slow bus as illustrated in Figure 4.13. The bridge is a slave on the fast bus and the master of the slow bus. The bridge takes commands from the fast bus on which it is a slave and issues those commands on the slow bus. It also returns the results from the slow bus to the fast bus—for example, it returns the results of a read on the slow bus to the fast bus.

The upper sequence of states handles a write from the fast bus to the slow bus. These states must read the data from the fast bus and set up the handshake for the slow bus. Operations on the fast and slow sides of the bus bridge should

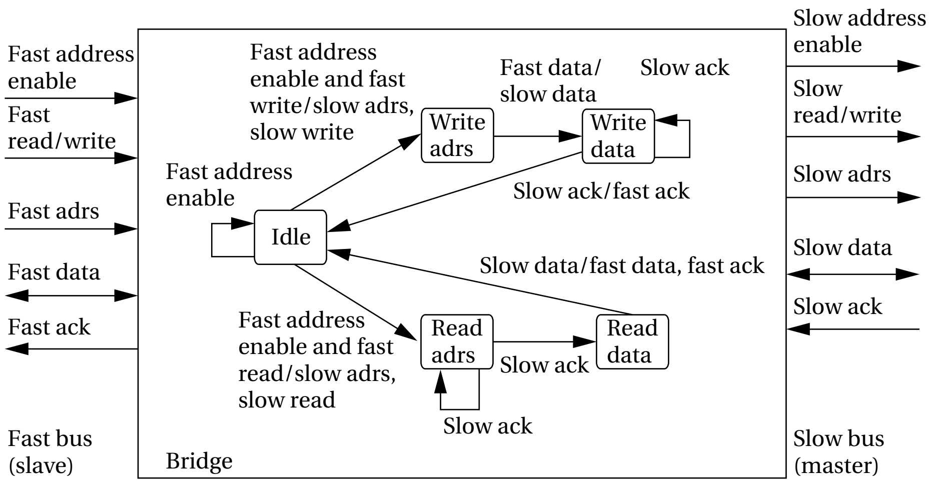

**FIGURE 4.13**

UML state diagram of bus bridge operation.

be overlapped as much as possible to reduce the latency of bus-to-bus transfers. Similarly, the bottom sequence of states reads from the slow bus and writes the data to the fast bus.

The bridge serves as a protocol translator between the two bridges as well. If the bridges are very close in protocol operation and speed, a simple state machine may be enough. If there are larger differences in the protocol and timing between the two buses, the bridge may need to use registers to hold some data values temporarily.

### 4.1.4 AMBA Bus

Since the ARM CPU is manufactured by many different vendors, the bus provided off-chip can vary from chip to chip. ARM has created a separate bus specification for single-chip systems. The AMBA bus [ARM99A] supports CPUs, memories, and peripherals integrated in a system-on-silicon. As shown in Figure 4.14, the AMBA specification includes two buses. The AMBA high-performance bus (AHB) is optimized for high-speed transfers and is directly connected to the CPU. It supports several high-performance features: pipelining, burst transfers, split transactions, and multiple bus masters.

A bridge can be used to connect the AHB to an AMBA peripherals bus (APB). This bus is designed to be simple and easy to implement; it also consumes relatively little power. The AHB assumes that all peripherals act as slaves, simplifying the logic required in both the peripherals and the bus controller. It also does not perform pipelined operations, which simplifies the bus logic.

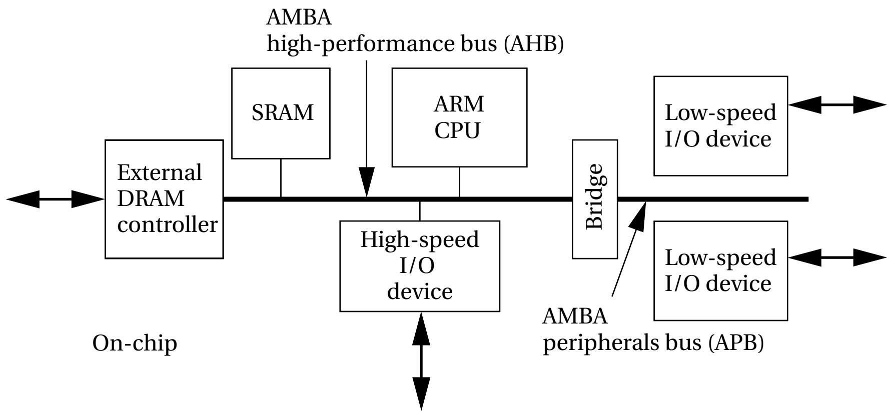

**FIGURE 4.14**

Elements of the ARM AMBA bus system.

## 4.2 MEMORY DEVICES

In this section, we introduce the basic types of memory components that are commonly used in embedded systems. Now that we understand the operation of the bus, we are able to understand the pinouts of these memories and how values are read and written. We also need to understand the varieties of memory cells that are used to build memories. There are several varieties of both read-only and read/write memories, each with its own advantages. After discussing some basic characteristics of memories, we describe RAMs and then ROMs.

### 4.2.1 Memory Device Organization

The most basic way to characterize a memory is by its capacity, such as 256 MB. However, manufacturers usually make several versions of a memory of a given size, each with a different data width. For example, a 256-MB memory may be available in two versions:

- As a 64 M $\times$ 4-bit array, a single memory access obtains an 8-bit data item, with a maximum of $2^{26}$ different addresses.
- As a 32 M $\times$ 8-bit array, a single memory access obtains a 1-bit data item, with a maximum of $2^{23}$ different addresses.

The height/width ratio of a memory is known as its ***aspect ratio***. The best aspect ratio depends on the amount of memory required.

Internally, the data are stored in a two-dimensional array of memory cells as shown in Figure 4.15. Because the array is stored in two dimensions, the $n$-bit address received by the chip is split into a row and a column address (with $n = r + c$).

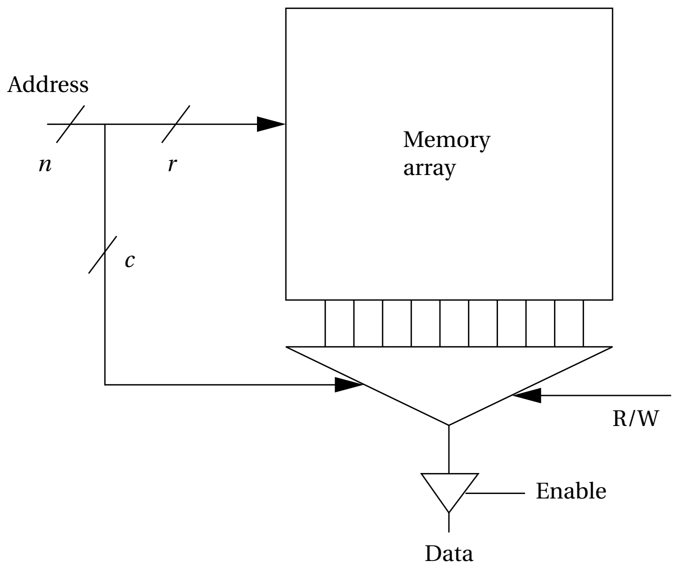

**FIGURE 4.15**

Internal organization of a memory device.

The row and column select a particular memory cell. If the memory's external width is 1 bit, the column address selects a single bit; for wider data widths, the column address can be used to select a subset of the columns. Most memories include an *enable* signal that controls the tri-stating of data onto the memory's pins. We will see in Section 4.4.1 how the enable pin can be used to easily build large memories from multiple banks of memory chips. A read/write signal (R/W in the figure) on read/write memories controls the direction of data transfer; memory chips do not typically have separate read and write data pins.

### 4.2.2 Random-Access Memories

Random-access memories can be both read and written. They are called random access because, unlike magnetic disks, addresses can be read in any order. Most bulk memory in modern systems is ***dynamic RAM (DRAM)***. DRAM is very dense; it does, however, require that its values be **refreshed** periodically since the values inside the memory cells decay over time.

The dominant form of dynamic RAM today is the ***synchronous DRAMs (SDRAMs)***, which uses clocks to improve DRAM performance. SDRAMs use Row Address Select (RAS) and Column Address Select (CAS) signals to break the address into two parts, which select the proper row and column in the RAM array. Signal transitions are relative to the SDRAM clock, which allows the internal SDRAM operations to be pipelined.

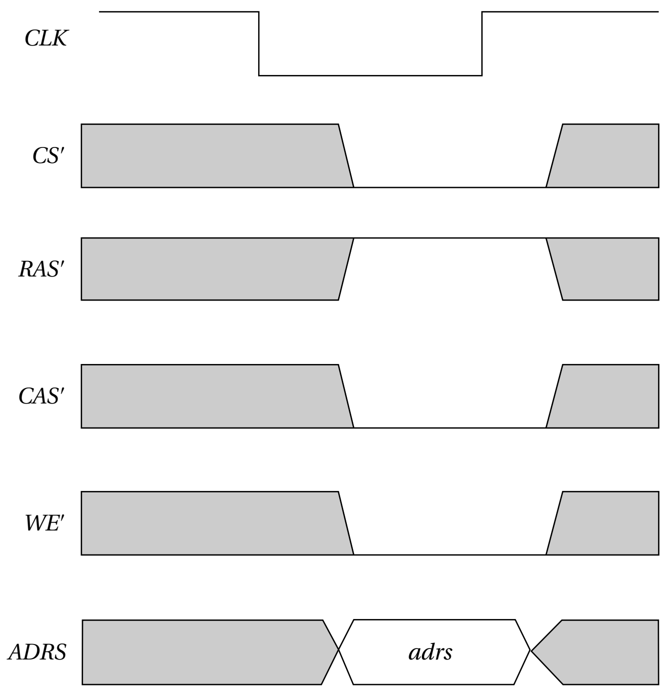

**FIGURE 4.16**

Timing diagram for a read on a synchronous DRAM.

As shown in Figure 4.16, transitions on the control signals are related to a clock [Mic00]. RAS′ and CAS′ can therefore become valid at the same time. The address lines are not shown in full detail here; some address lines may not be active depending on the mode in use. SDRAMs use a separate refresh signal to control refreshing. DRAM has to be refreshed roughly once per millisecond. Rather than refresh the entire memory at once, DRAMs refresh part of the memory at a time. When a section of memory is being refreshed, it cannot be accessed until the refresh is complete. The memory refresh occurs over fairly few seconds so that each section is refreshed every few microseconds.

SDRAMs include registers that control the mode in which the SDRAM operates. SDRAMs support burst modes that allow several sequential addresses to be accessed by sending only one address. SDRAMs generally also support an interleaved mode that exchanges pairs of bytes.

Even faster synchronous DRAMs, known as ***double-data rate (DDR)*** SDRAMs or **DDR2** and **DDR3** SDRAMs, are now in use. The details of DDR operation are beyond the scope of this book, but the basic capabilities of DDR memories are similar to those of single-rate SDRAMs; DDRs simply use sophisticated circuit techniques to perform more operations per clock cycle.

#### *SIMMs and DIMMs*

Memory for PCs is generally purchased as ***single in-line memory modules (SIMMs)*** or ***double in-line memory modules (DIMMs)***. A SIMM or DIMM is a small circuit board that fits into a standard memory socket. A DIMM has two sets of leads compared to the SIMM's one. Memory chips are soldered to the circuit board to supply the desired memory.

### 4.2.3 Read-Only Memories

***Read-only memories (ROMs)*** are preprogrammed with fixed data. They are very useful in embedded systems since a great deal of the code, and perhaps some data, does not change over time. Read-only memories are also less sensitive to radiation-induced errors.

There are several varieties of ROM available. The first-level distinction to be made is between ***factory-programmed ROM*** (sometimes called ***mask-programmed ROM***) and ***field-programmable ROM***. Factory-programmed ROMs are ordered from the factory with particular programming. ROMs can typically be ordered in lots of a few thousand, but clearly factory programming is useful only when the ROMs are to be installed in some quantity.

Field-programmable ROMs, on the other hand, can be programmed in the lab. ***Flash memory*** is the dominant form of field-programmable ROM and is electrically erasable. Flash memory uses standard system voltage for erasing and programming, allowing it to be reprogrammed inside a typical system. This allows applications such as automatic distribution of upgrades—the flash memory can be reprogrammed while downloading the new memory contents from a telephone line. Early flash memories had to be erased in their entirety; modern devices allow memory to be erased in blocks. Most flash memories today allow certain blocks to be protected. A common application is to keep the boot-up code in a protected block but allow updates to other memory blocks on the device. As a result, this form of flash is commonly known as ***boot-block flash***.

## 4.3 I/O DEVICES

In this section we survey some input and output devices commonly used in embedded computing systems. Some of these devices are often found as on-chip devices in micro-controllers; others are generally implemented separately but are still commonly used. Looking at a few important devices now will help us understand both the requirements of device interfacing in this chapter and the uses of devices in programming in this and later chapters.

### 4.3.1 Timers and Counters

***Timers*** and ***counters*** are distinguished from one another largely by their use, not their logic. Both are built from adder logic with registers to hold the current