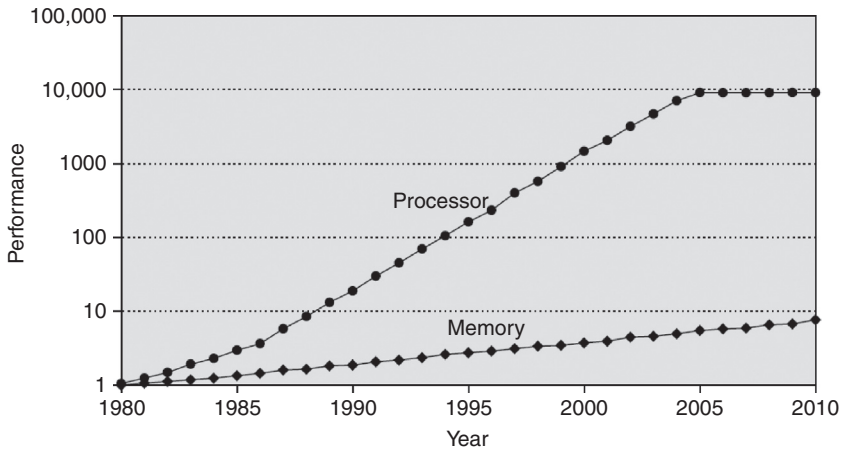


Figure 8.2 Diverging processor and memory performance Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011.

expensive than a single large fast memory. These principles extend to using an entire hierarchy of memories of increasing capacity and decreasing speed.

Computer memory is generally built from DRAM chips. In 2015, a typical PC had a *main memory* consisting of 8 to 16 GB of DRAM, and DRAM cost about $7 per gigabyte (GB). DRAM prices have declined at about 25% per year for the last three decades, and memory capacity has grown at the same rate, so the total cost of the memory in a PC has remained roughly constant. Unfortunately, DRAM speed has improved by only about 7% per year, whereas processor performance has improved at a rate of 25 to 50% per year, as shown in Figure 8.2. The plot shows memory (DRAM) and processor speeds with the 1980 speeds as a baseline. In about 1980, processor and memory speeds were the same. But performance has diverged since then, with memories badly lagging.[2]

DRAM could keep up with processors in the 1970s and early 1980's, but it is now woefully too slow. The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond).

To counteract this trend, computers store the most commonly used instructions and data in a faster but smaller memory, called a *cache*. The cache is usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed, because SRAM is inherently faster than DRAM, and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip. In 2015, on-chip SRAM costs were on the order of $5,000/GB, but the

[2] Although recent single processor performance has remained approximately constant, as shown in Figure 8.2 for the years 2005–2010, the increase in multi-core systems (not depicted on the graph) only worsens the gap between processor and memory performance.

cache is relatively small (kilobytes to several megabytes), so the overall cost is low. Caches can store both instructions and data, but we will refer to their contents generically as "data."

If the processor requests data that is available in the cache, it is returned quickly. This is called a cache *hit*. Otherwise, the processor retrieves the data from main memory (DRAM). This is called a cache *miss*. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the hard drive. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard drive to store data that does not fit in main memory. In 2015, a hard disk drive (HDD), built using magnetic storage, cost less than \$0.05/GB and had an access time of about 5 ms. Hard disk costs have decreased at 60%/year but access times scarcely improved. Solid state drives (SSDs), built using flash memory technology, are an increasingly common alternative to HDDs. SSDs have been used by niche markets for over two decades, and they were introduced into the mainstream market in 2007. SSDs overcome some of the mechanical failures of HDDs, but they cost about ten times as much at \$0.40/GB.

The hard drive provides an illusion of more capacity than actually exists in the main memory. It is thus called virtual memory. Like books in the basement, data in virtual memory takes a long time to access. Main memory, also called physical memory, holds a subset of the virtual memory. Hence, the main memory can be viewed as a cache for the most commonly used data from the hard drive.

Figure 8.3 summarizes the memory hierarchy of the computer system discussed in the rest of this chapter. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical costs, access times, and bandwidth in 2015 technology. As access time decreases, speed increases.

Section 8.2 introduces memory system performance analysis. Section 8.3 explores several cache organizations, and Section 8.4 delves into virtual memory systems.

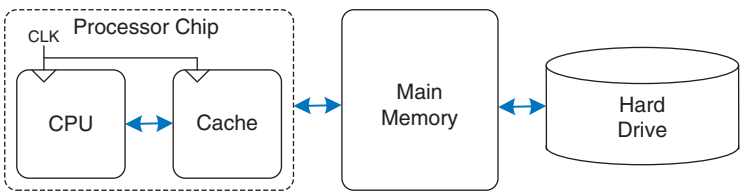
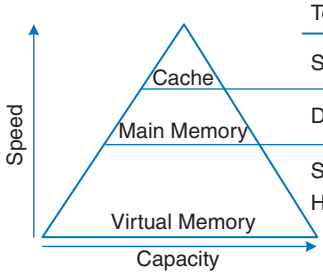


Figure 8.3 A typical memory hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$5,000	0.5	25+
DRAM	$7	10–50	10
SSD	$0.40	20,000	0.5
HDD	$0.05	5,000,000	0.75

Figure 8.4 **Memory hierarchy components, with typical characteristics in 2015**

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need quantitative ways to measure the performance of memory systems to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are *miss rate* or *hit rate* and *average memory access time*. Miss and hit rates are calculated as:

$$\begin{aligned} \textit{Miss Rate} &= \frac{\textit{Number of misses}}{\textit{Number of total memory accesses}} = 1 - \textit{Hit Rate} \\ \textit{Hit Rate} &= \frac{\textit{Number of hits}}{\textit{Number of total memory accesses}} = 1 - \textit{Miss Rate} \end{aligned} \tag{8.1}$$

Example 8.1 CALCULATING CACHE PERFORMANCE

Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

Solution: The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 − 0.375 = 62.5%.

Average memory access time (*AMAT*) is the average time a processor must wait for memory per load or store instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$AMAT = t_{\text{cache}} + MR_{\text{cache}}(t_{MM} + MR_{MM}t_{VM}) \tag{8.2}$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Example 8.2 CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates in Table 8.1?

Solution: The average memory access time is $1 + 0.1(100) = 11$ cycles.

Table 8.1 Access times and miss rates

Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Gene Amdahl, 1922–. Most famous for Amdahl's Law, an observation he made in 1965. While in graduate school, he began designing computers in his free time. This side work earned him his Ph.D. in theoretical physics in 1952. He joined IBM immediately after graduation, and later went on to found three companies, including one called Amdahl Corporation in 1970.

Example 8.3 IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

Solution: If the miss rate is m, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%.

As a word of caution, performance improvements might not always be as good as they sound. For example, making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's performance is due to loads and stores, a tenfold memory system improvement only means a 1.82-fold improvement in program performance. This general principle is called *Amdahl's Law*, which says that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance.

8.3 CACHES

A cache holds commonly used memory data. The number of data words that it can hold is called the *capacity*, C. Because the capacity

Cache: a hiding place especially for concealing and preserving provisions or implements.
– Merriam Webster Online Dictionary, 2015. www.merriam-webster.com

of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.

When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is held in the cache? (2) How is data found? and (3) What data is replaced to make room for new data when the cache is full?

When reading the next sections, keep in mind that the driving force in answering these questions is the inherent spatial and temporal locality of data accesses in most applications. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache.

As we explain in the following sections, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N).

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache. Data cache store operations are similar and are discussed further in Section 8.3.4.

8.3.1 What Data is Held in the Cache?

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate. Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations. Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a *cache block* or *cache line*. The number of words in the cache block, b, is called the *block size*. A cache of capacity C contains $B = C/b$ blocks.

The principles of temporal and spatial locality have been experimentally verified in real programs. If a variable is used in a program, the same

variable is likely to be used again, creating temporal locality. If an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality.

8.3.2 How is Data Found?

A cache is organized into S *sets*, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.

Caches are categorized based on the number of blocks in a set. In a *direct mapped* cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a unique block in the cache. In an *N-way set associative* cache, each set contains N blocks. The address still maps to a unique set, with $S = B/N$ sets. But the data from that address can go in any of the N blocks in that set. A *fully associative* cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

To illustrate these cache organizations, we will consider an ARM memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eight-word capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has one block in each set, so it is organized into $S = B$ sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B - 1$ of main memory maps to block $B - 1$ of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of eight words and a block size of one word. The cache has eight sets, each of which contains a one-word block. The bottom two bits of the address are always 00, because they are word aligned. The next $\log_2 8 = 3$ bits indicate the set onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, . . ., 0xFFFFFFE4 all

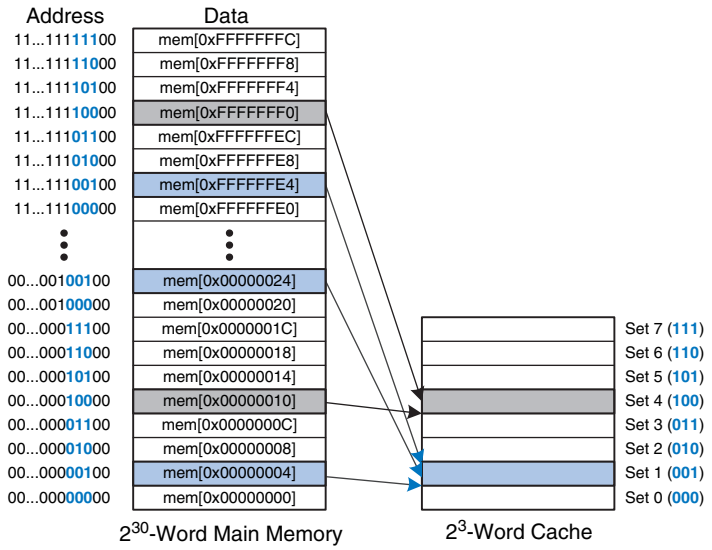


Figure 8.5 Mapping of main memory to a direct mapped cache

map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, . . . , 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

Example 8.4 CACHE FIELDS

To what cache set in Figure 8.5 does the word at address 0x00000014 map? Name another address that maps to the same set.

Solution: The two least significant bits of the address are 00, because the address is word aligned. The next three bits are 101, so the word maps to set 5. Words at addresses 0x34, 0x54, 0x74, . . . , 0xFFFFFFF4 all map to this same set.

Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the *tag* and indicate which of the many possible addresses is held in that set.

In our previous examples, the two least significant bits of the 32-bit address are called the *byte offset*, because they indicate the byte within the word. The next three bits are called the *set bits*, because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the data stored in a given cache set. Figure 8.6 shows the cache fields for address 0xFFFFFFE4. It maps to set 1 and its tag is all 1's.

Figure 8.6 Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

Example 8.5 CACHE FIELDS

Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

Solution: A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining $32 - 10 - 2 = 20$ bits form the tag.

Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a *valid bit* for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry SRAM. Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks the tag and valid bits. If the tag matches the most significant 27 bits of the

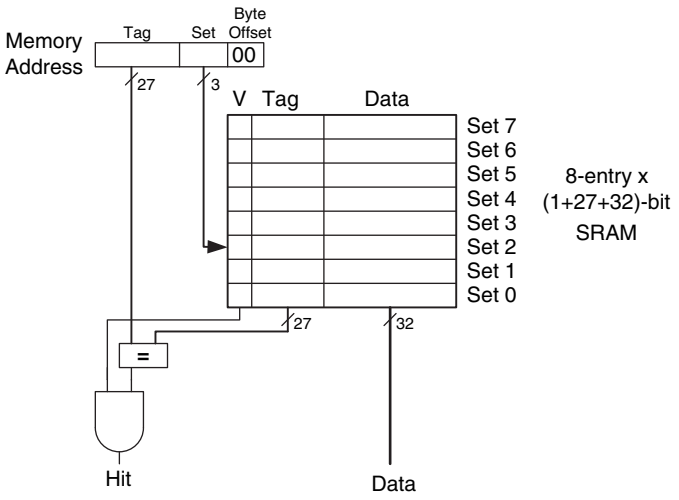


Figure 8.7 Direct mapped cache with 8 sets

address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in ARM assembly code. Assume that the cache is initially empty. What is the miss rate?

```
       MOV R0, #5
       MOV R1, #0
LOOP   CMP R0, #0
       BEQ DONE
       LDR R2, [R1, #4]
       LDR R3, [R1, #12]
       LDR R4, [R1, #8]
       SUB R0, R0, #1
       B   LOOP
DONE
```

Solution: The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is 3/15 = 20%.

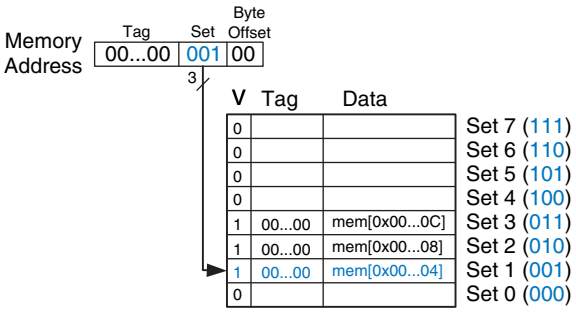


Figure 8.8 Direct mapped cache contents

When two recently accessed addresses map to the same cache block, a *conflict* occurs, and the most recently accessed address *evicts* the previous one from the block. Direct mapped caches have only one block in each

set, so two addresses that map to the same set always cause a conflict. Example 8.7 illustrates conflicts.

Example 8.7 CACHE BLOCK CONFLICT

What is the miss rate when the following loop is executed on the eight-word direct mapped cache from Figure 8.7? Assume that the cache is initially empty.

```
        MOV R0, #5
        MOV R1, #0
LOOP    CMP R0, #0
        BEQ DONE
        LDR R2, [R1, #0x4]
        LDR R3, [R1, #0x24]
        SUB R0, R0, #1
        B   LOOP
DONE
```

Solution: Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict, and the miss rate is 100%.

Multi-way Set Associative Cache

An *N-way set associative* cache reduces conflicts by providing *N* blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the *N* blocks

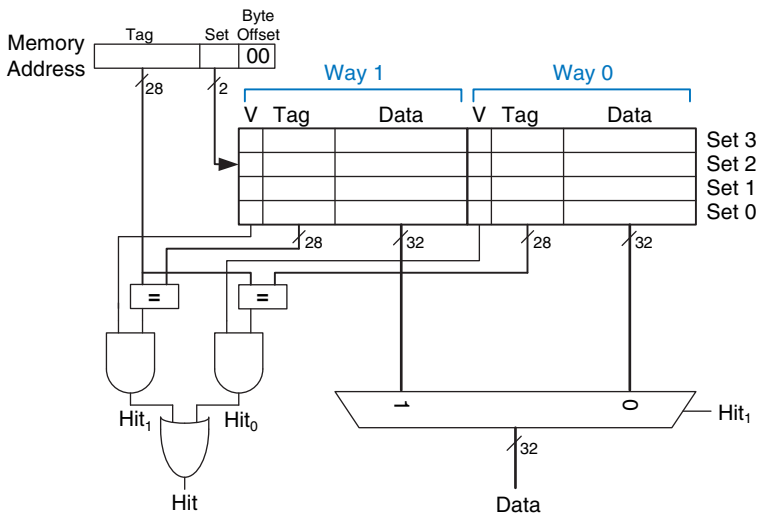


Figure 8.9 Two-way set associative cache

in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the *degree of associativity* of the cache.

Figure 8.9 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8. Thus, only $\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8 SET ASSOCIATIVE CACHE MISS RATE

Repeat Example 8.7 using the eight-word two-way set associative cache from Figure 8.9.

Solution: Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1, as shown in Figure 8.10. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%. Recall that the direct mapped cache of the same size from Example 8.7 had a miss rate of 100%.

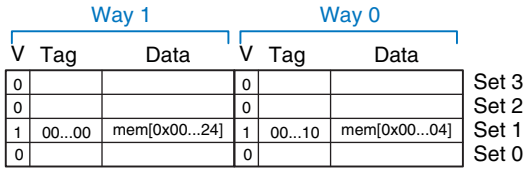


Figure 8.10 Two-way set associative cache contents

Fully Associative Cache

A *fully associative* cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag comparisons (not shown) must be made, because the data could be in any block. Similarly, an 8:1

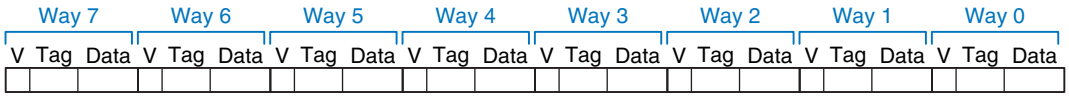


Figure 8.11 Eight-block fully associative cache

multiplexer chooses the proper data if a hit occurs. Fully associative caches tend to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage only of temporal locality, because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a $C = 8$-word direct mapped cache with a $b = 4$-word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache has one block in each set, so this cache

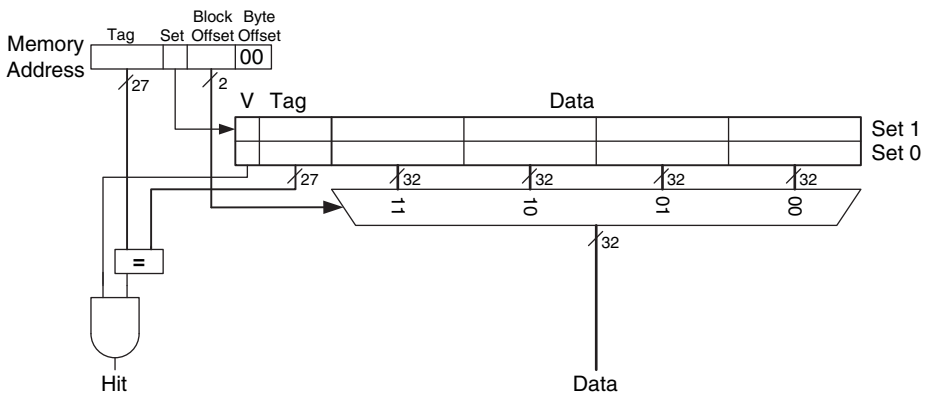


Figure 8.12 Direct mapped cache with two sets and a four-word block size

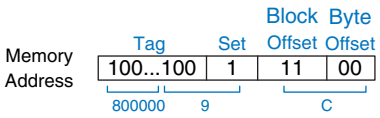


Figure 8.13 Cache fields for address 0x8000009C when mapping to the cache of Figure 8.12

is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block. And the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

Example 8.9 SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution: Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is 1/15 = 6.67%.

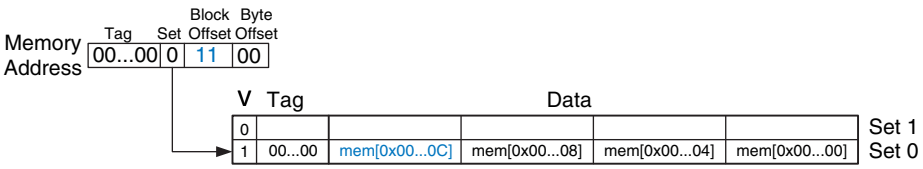


Figure 8.14 Cache contents with a block size *b* of four words

Putting it All Together

Caches are organized as two-dimensional arrays. The rows are called sets, and the columns are called ways. Each entry in the array

Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

consists of a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Table 8.2 summarizes the various cache organizations. Each address in memory maps to only one set but can be stored in any of the ways.

Cache capacity, associativity, set size, and block size are typically powers of 2. This makes the cache fields (tag, set, and block offset bits) subsets of the address bits.

Increasing the associativity N usually reduces the miss rate caused by conflicts. But higher associativity requires more tag comparators. Increasing the block size b takes advantage of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and therefore could lead to more conflicts. It also increases the miss penalty.

8.3.3 What Data is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. Hence, most associative caches have a *least recently used* (*LRU*) replacement policy.

In a two-way set associative cache, a *use bit*, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which *group* of ways was least recently used. Upon replacement, the new block replaces a random

block within the least recently used group. Such a policy is called *pseudo-LRU* and is good enough in practice.

Example 8.10 LRU REPLACEMENT

Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

```
MOV R0, #0
LDR R1, [R0, #4]
LDR R2, [R0, #0x24]
LDR R3, [R0, #0x54]
```

Solution: The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache, shown in Figure 8.15(a). $U = 0$ indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0, as shown in Figure 8.15(b). The use bit U is set to 1 to indicate that data in way 1 was the least recently used.

Figure 8.15 Two-way associative cache with LRU replacement

8.3.4 Advanced Cache Design*

Modern systems use multiple levels of caches to decrease memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

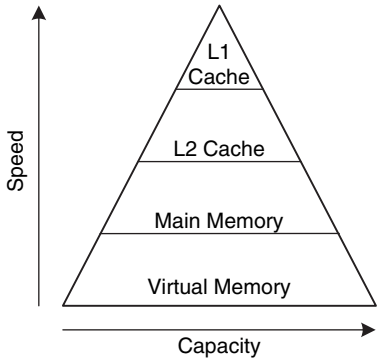


Figure 8.16 Memory hierarchy with two levels of cache

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and therefore have lower miss rates. However, large caches tend to be slower than small ones. Modern systems often use at least two levels of caches, as shown in Figure 8.16. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger, and therefore slower, than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Many modern systems add even more levels of cache to the memory hierarchy, because accessing main memory is so slow.

Example 8.11 SYSTEM WITH AN L2 CACHE

Use the system of Figure 8.16 with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (*AMAT*)?

Solution: Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from main memory. Using Equation 8.2, we calculate the average memory access time as follows: 1 cycle + 0.05[10 cycles + 0.2(100 cycles)] = 2.5 cycles

The L2 miss rate is high because it receives only the "hard" memory accesses, those that miss in the L1 cache. If all accesses went directly to the L2 cache, the L2 miss rate would be about 1%.

Reducing Miss Rate

Cache misses can be reduced by changing capacity, block size, and/or associativity. The first step to reducing the miss rate is to understand the causes of the misses. The misses can be classified as compulsory, capacity, and conflict. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more type of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are complicated enough that the best way to evaluate their performance is by running benchmarks while varying cache parameters. Figure 8.17 plots miss rate versus cache size and degree of associativity for the SPEC2000 benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve. Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

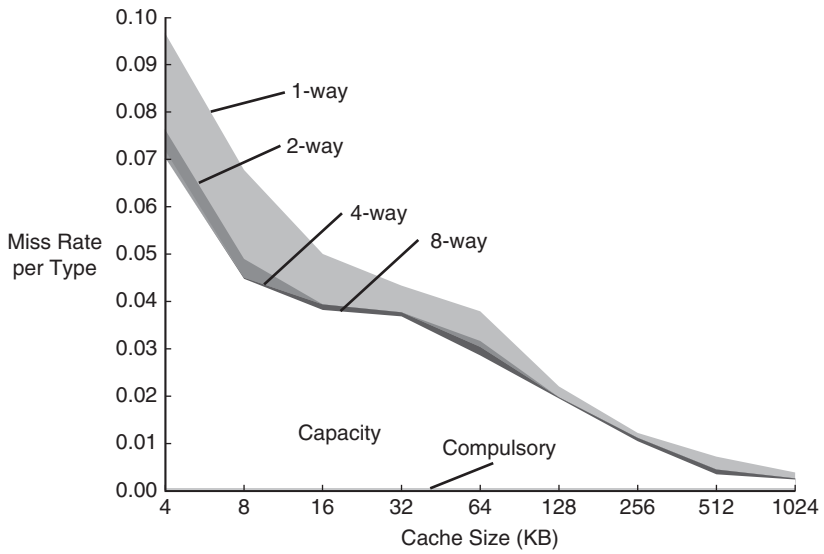


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark (Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.)

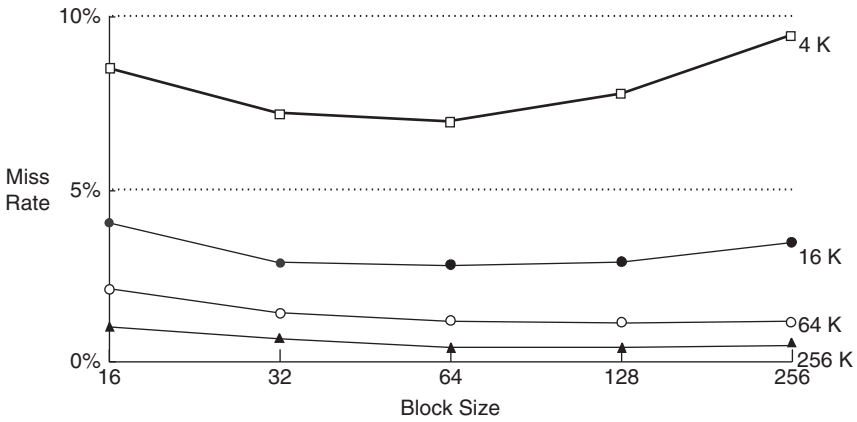


Figure 8.18 Miss rate versus block size and cache size on SPEC92 benchmark
(Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2012.)

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. But as block size increases, the number of sets in a fixed-size cache decreases, increasing the probability of conflicts. Figure 8.18 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4-KB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size beyond 64 bytes does not change the miss rate. However, large block sizes might still increase execution time because of the larger miss penalty, the time required to fetch the missing cache block from main memory.

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is fetched from main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is simply written to the cache block.

Caches are classified as either write-through or write-back. In a *write-through* cache, the data written to a cache block is simultaneously written to main memory. In a *write-back* cache, a *dirty bit* (D) is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through cache requires no dirty bit but usually requires more main memory writes than a write-back cache. Modern caches are usually write-back, because main memory access time is so large.

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
MOV R5, #0
STR R1, [R5]
STR R2, [R5, #12]
STR R3, [R5, #8]
STR R4, [R5, #4]
```

Solution: All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

8.3.5 The Evolution of ARM Caches*

Table 8.3 traces the evolution of cache organizations used by the ARM processor from 1985 to 2012. The major trends are the introduction of multiple levels of cache, larger cache capacity, and separation of instruction and data L1 caches. These trends are driven by the growing disparity between CPU frequency and main memory speed and the decreasing cost

Table 8.3 ARM cache evolution

Year	CPU	MHz	L1 Cache	L2 Cache
1985	ARM1	8	None	None
1992	ARM6	30	4 KB, unified	None
1994	ARM7	100	8 KB, unified	None
1999	ARM9E	300	0–128 KB, I/D	None
2002	ARM11	700	4–64 KB, I/D	0–128 KB, off-chip
2009	Cortex-A9	1000	16–64 KB, I/D	0–8 MB
2011	Cortex-A7	1500	32 KB, I/D	0–4 MB
2011	Cortex-A15	2000	32 KB, I/D	0–4 MB
2012	Cortex-M0+	60–250	None	None
2012	Cortex-A53	1500	8–64 KB, I/D	128 KB–2 MB
2012	Cortex-A57	2000	48 KB I / 32 KB D	512 KB–2 MB

of transistors. The increasing difference between CPU and memory speeds necessitates a lower miss rate to avoid the main memory bottleneck, and the decreasing cost of transistors allows larger cache sizes.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard drive* made of magnetic or solid state storage as the lowest level in the memory hierarchy (see Figure 8.4). Compared with the ideal large, fast, cheap memory, a hard drive is large and cheap but terribly slow. It provides a much larger capacity than is possible with a cost-effective main memory (DRAM). However, if a significant fraction of memory accesses involve the hard drive, performance is dismal. You may have encountered this on a PC when running too many programs at once.

Figure 8.19 shows a hard drive made of magnetic storage, also called a *hard disk*, with the lid of its case removed. As the name implies, the hard disk contains one or more rigid disks or *platters*, each of which has a *read/write head* on the end of a long triangular arm. The head

Figure 8.19 Hard disk

moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes several milliseconds to *seek* the correct location on the disk, which is fast from a human perspective but millions of times slower than the processor. Hard disk drives are increasingly being replaced by solid state drives because reading is orders of magnitude faster (see Figure 8.4) and they are not as susceptible to mechanical failures.

The objective of adding a hard drive to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 128 MB of DRAM, for example, could effectively provide 2 GB of memory using the hard drive. This larger 2-GB memory is called *virtual memory*, and the smaller 128-MB main memory is called *physical memory*. We will use the term physical memory to refer to main memory throughout this section.

A computer with 32-bit addresses can access a maximum of 2^{32} bytes = 4 GB of memory. This is one of the motivations for moving to 64-bit computers, which can access far more memory.

Programs can access data anywhere in virtual memory, so they must use *virtual addresses* that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.

Virtual memory systems use different terminologies for the same caching principles discussed in Section 8.3. Table 8.4 summarizes the analogous terms. Virtual memory is divided into *virtual pages*, typically 4 KB in size. Physical memory is likewise divided into *physical pages* of the same size. A virtual page may be located in physical memory (DRAM) or on the hard drive. For example, Figure 8.20 shows a virtual memory that is larger than physical memory. The rectangles indicate pages. Some virtual pages are present in physical memory, and some are located on the hard drive. The process of determining the physical address from the virtual address is called *address translation*. If the processor attempts to access a virtual address that is not in physical memory, a *page fault*

Table 8.4 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

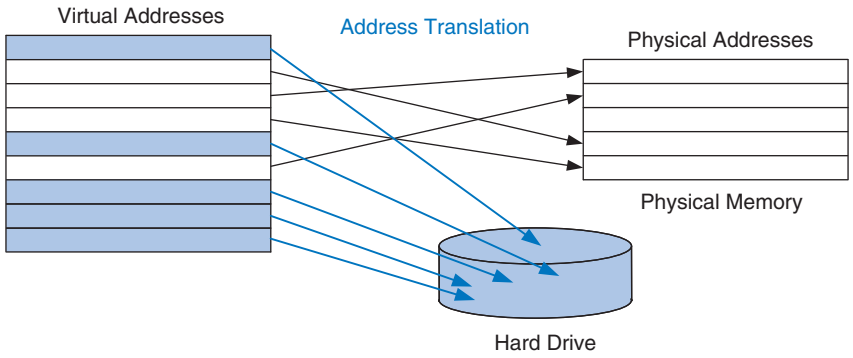


Figure 8.20 Virtual and physical pages

occurs, and the operating system loads the page from the hard drive into physical memory.

To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A realistic virtual memory system has so many physical pages that providing a comparator for each page would be excessively expensive. Instead, the virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the hard drive. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access, and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

The remainder of this section elaborates on address translation, page tables, and TLBs.

8.4.1 Address Translation

In a system with virtual memory, programs use virtual addresses so that they can access a large memory. The computer must translate these virtual

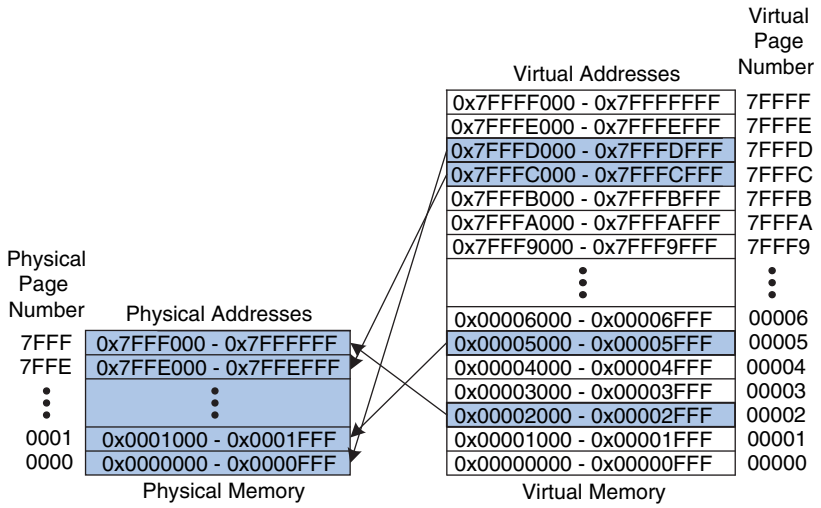


Figure 8.21 **Physical and virtual pages**

addresses to either find the address in physical memory or take a page fault and fetch the data from the hard drive.

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical *page number*. The least significant bits specify the word within the page and are called the *page offset*.

Figure 8.21 illustrates the page organization of a virtual memory system with 2 GB of virtual memory and 128 MB of physical memory divided into 4-KB pages. MIPS accommodates 32-bit addresses. With a 2-GB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128-MB = 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

Because the page size is 4 KB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on the hard drive.

Figure 8.21 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

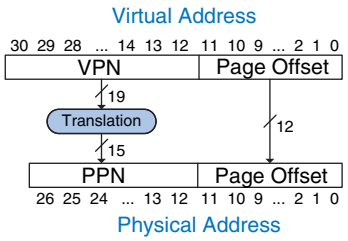


Figure 8.22 Translation from virtual address to physical address

Figure 8.22 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the *virtual page number* (*VPN*) and are translated to a 15-bit *physical page number* (*PPN*). The next two sections describe how page tables and TLBs are used to perform this address translation.

Example 8.13 VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the virtual memory system shown in Figure 8.21.

Solution: The 12-bit page offset (0x47C) requires no translation. The remaining 19 bits of the virtual address give the virtual page number, so virtual address 0x247C is found in virtual page 0x2. In Figure 8.21, virtual page 0x2 maps to physical page 0x7FFF. Thus, virtual address 0x247C maps to physical address 0x7FFF47C.

V	Physical Page Number	Virtual Page Number
0		7FFFF
0		7FFFE
1	0x0000	7FFFD
1	0x7FFE	7FFFC
0		7FFFB
0		7FFFA
	⋮	⋮
0		00007
0		00006
1	0x0001	00005
0		00004
0		00003
1	0x7FFF	00002
0		00001
0		00000

Page Table

Figure 8.23 The page table for Figure 8.21

8.4.2 The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. The page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on the hard drive.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.23. This page table contains the mapping of the memory system of Figure 8.21. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), so virtual page 6 is located on the hard drive.

Example 8.14 USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the page table shown in Figure 8.23.

Solution: Figure 8.24 shows the virtual address to physical address translation for virtual address 0x247C. The 12-bit page offset requires no translation. The remaining 19 bits of the virtual address are the virtual page number, 0x2, and give the index into the page table. The page table maps virtual page 0x2 to physical page 0x7FFF. So, virtual address 0x247C maps to physical address 0x7FFF47C. The least significant 12 bits are the same in both the physical and the virtual address.

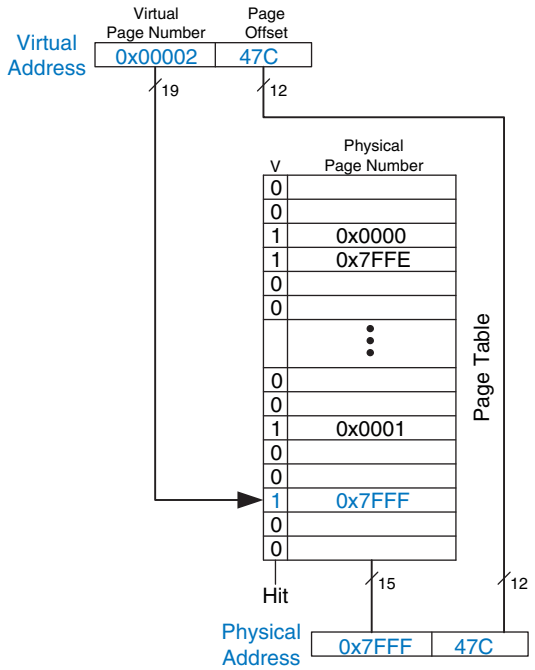


Figure 8.24 Address translation using the page table

The page table can be stored anywhere in physical memory, at the discretion of the OS. The processor typically uses a dedicated register, called the *page table register*, to store the base address of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.4.3 The Translation Lookaside Buffer

Virtual memory would have a severe performance impact if it required a page table read on every load or store, doubling the delay of loads and stores. Fortunately, page table accesses have great temporal locality. The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. In general, the processor can keep the last several page table entries in a small cache called a *translation lookaside buffer* (*TLB*). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. In real programs, the vast majority of accesses hit in the TLB, avoiding the time-consuming page table reads from physical memory.

A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle. Even so, TLBs typically have a hit rate of greater than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to one.

Example 8.15 USING THE TLB TO PERFORM ADDRESS TRANSLATION

Consider the virtual memory system of Figure 8.21. Use a two-entry TLB or explain why a page table access is necessary to translate virtual addresses 0x247C and 0x5FB0 to physical addresses. Suppose the TLB currently holds valid translations of virtual pages 0x2 and 0x7FFFD.

Solution: Figure 8.25 shows the two-entry TLB with the request for virtual address 0x247C. The TLB receives the virtual page number of the incoming address, 0x2, and compares it to the virtual page number of each entry. Entry 0 matches and is valid, so the request hits. The translated physical address is the physical page number of the matching entry, 0x7FFF, concatenated with the page offset of the virtual address. As always, the page offset requires no translation.

The request for virtual address 0x5FB0 misses in the TLB. So, the request is forwarded to the page table for translation.

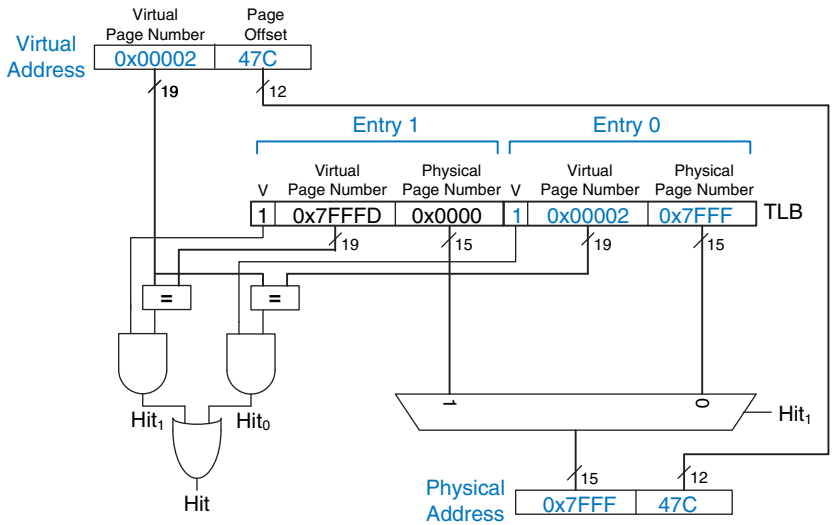


Figure 8.25 Address translation using a two-entry TLB

8.4.4 Memory Protection

So far, this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally important reason to use virtual memory is to provide protection between concurrently running programs.

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages, because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The operating system adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

8.4.5 Replacement Policies*

Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. A write-through policy, where each write to physical memory initiates a write to the hard drive, would be impractical. Store instructions would operate at the speed of the hard drive instead of the speed of the processor (milliseconds instead of nanoseconds). Under the writeback policy, the physical page is written back to the hard drive only when it is evicted from physical memory. Writing the physical page back to the hard drive and reloading it with a different virtual page is called *paging*, and the hard drive in a virtual memory system is sometimes called *swap space*. The processor pages out one of the least recently used physical pages when a page fault occurs, then replaces that page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit D and a use bit U.

The dirty bit is 1 if any store instructions have changed the physical page since it was read from the hard drive. When a physical page is paged out, it needs to be written back to the hard drive only if its dirty bit is 1; otherwise, the hard drive already holds an exact copy of the page.

The use bit is 1 if the physical page has been accessed recently. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS approximates LRU replacement by periodically resetting all the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS finds a page with $U = 0$ to page out of physical memory. Thus, it does not necessarily replace the least recently used page, just one of the least recently used pages.

8.4.6 Multilevel Page Tables*

Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GB virtual memory with 4 KB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^2$ bytes $= 2^{21}$ bytes $= 2$ MB.

To conserve physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is not actively used, the corresponding second-level page table can be paged out to the hard drive so it does not waste physical memory.

In a two-level page table, the virtual page number is split into two parts: the *page table number* and the *page table offset*, as shown in Figure 8.26. The page table number indexes the first-level page table, which must reside in physical memory. The first-level page table entry gives the base address of the second-level page table or indicates that it must be fetched from the

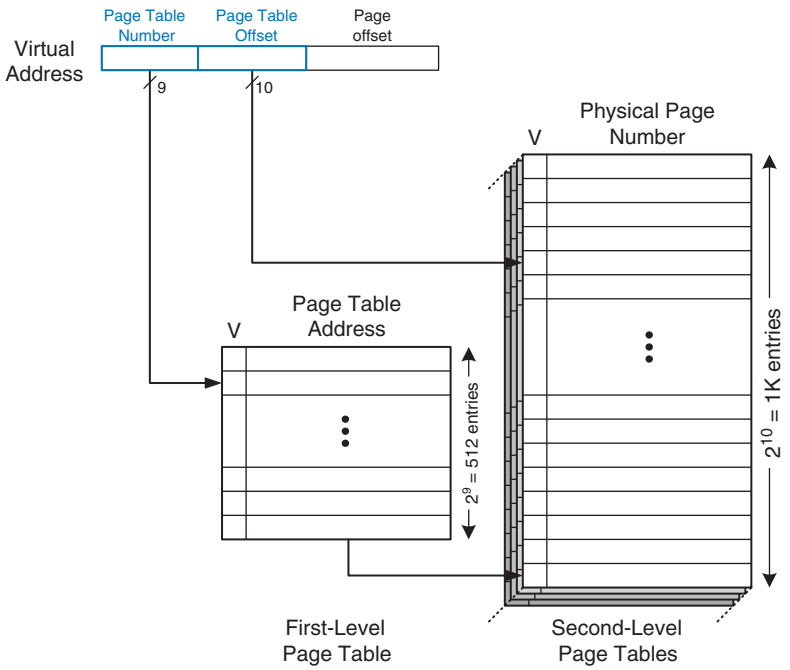


Figure 8.26 Hierarchical page tables

hard drive when V is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of $2^{12} = 4$ KB.

In Figure 8.26 the 19-bit virtual page number is broken into 9 and 10 bits, to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has $2^9 = 512$ entries. Each of these 512 second-level page tables has $2^{10} = 1$ K entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the hierarchical page table uses only $(512 \times 4 \text{ bytes}) + 2 \times (1 \text{ K} \times 4 \text{ bytes}) =$ 10 KB of physical memory. The two-level page table requires a fraction of the physical memory needed to store the entire page table (2 MB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

Example 8.16 USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION

Figure 8.27 shows the possible contents of the two-level page table from Figure 8.26. The contents of only one second-level page table are shown. Using this two-level page table, describe what happens on an access to virtual address 0x003FEFB0.

Solution: As always, only the virtual page number requires translation. The most significant nine bits of the virtual address, 0x0, give the page table number, the index into the first-level page table. The first-level page table at entry 0x0 indicates that the second-level page table is resident in memory ($V = 1$) and its physical address is 0x2375000.

The next ten bits of the virtual address, 0x3FE, are the page table offset, which gives the index into the second-level page table. Entry 0 is at the bottom of the second-level page table, and entry 0x3FF is at the top. Entry 0x3FE in the second-level page table indicates that the virtual page is resident in physical memory ($V = 1$) and that the physical page number is 0x23F1. The physical page number is concatenated with the page offset to form the physical address, 0x23F1FB0.

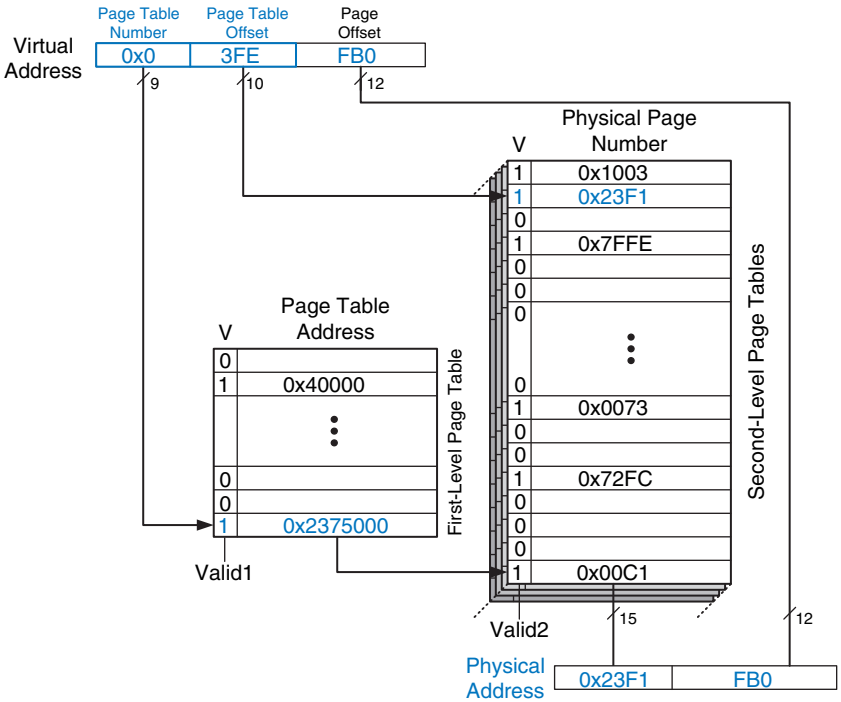


Figure 8.27 Address translation using a two-level page table

8.5 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies, such as DRAM, SRAM, and hard drives, offer trade-offs in capacity, speed, and cost. This chapter

introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard drive to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory.

EPILOGUE

This chapter brings us to the end of our journey together into the realm of digital systems. We hope this book has conveyed the beauty and thrill of the art as well as the engineering knowledge. You have learned to design combinational and sequential logic using schematics and hardware description languages. You are familiar with larger building blocks such as multiplexers, ALUs, and memories. Computers are one of the most fascinating applications of digital systems. You have learned how to program an ARM processor in its native assembly language and how to build the processor and memory system using digital building blocks. Throughout, you have seen the application of abstraction, discipline, hierarchy, modularity, and regularity. With these techniques, we have pieced together the puzzle of a microprocessor's inner workings. From cell phones to digital television to Mars rovers to medical imaging systems, our world is an increasingly digital place.

Imagine what Faustian bargain Charles Babbage would have made to take a similar journey a century and a half ago. He merely aspired to calculate mathematical tables with mechanical precision. Today's digital systems are yesterday's science fiction. Might Dick Tracy have listened to iTunes on his cell phone? Would Jules Verne have launched a constellation of global positioning satellites into space? Could Hippocrates have cured illness using high-resolution digital images of the brain? But at the same time, George Orwell's nightmare of ubiquitous government surveillance becomes closer to reality each day. Hackers and governments wage undeclared cyberwarfare, attacking industrial infrastructure and financial networks. And rogue states develop nuclear weapons using laptop computers more powerful than the room-sized supercomputers that simulated Cold War bombs. The microprocessor revolution continues to accelerate. The changes in the coming decades will surpass those of the past. You now have the tools to design and build these new systems that will shape our future. With your newfound power comes profound responsibility. We hope that you will use it, not just for fun and riches, but also for the benefit of humanity.

Exercises

Exercise 8.1 In less than one page, describe four everyday activities that exhibit temporal or spatial locality. List two activities for each type of locality, and be specific.

Exercise 8.2 In one paragraph, describe two short computer applications that exhibit temporal and/or spatial locality. Describe how. Be specific.

Exercise 8.3 Come up with a sequence of addresses for which a direct mapped cache with a size (capacity) of 16 words and block size of 4 words outperforms a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Exercise 8.4 Repeat Exercise 8.3 for the case when the fully associative cache outperforms the direct mapped cache.

Exercise 8.5 Describe the trade-offs of increasing each of the following cache parameters while keeping the others the same:

(a) block size

(b) associativity

(c) cache size

Exercise 8.6 Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size? Explain.

Exercise 8.7 Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

(a) A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.

(b) A 16-KB direct mapped cache always has a lower miss rate than an 8-KB direct mapped cache with the same block size.

(c) An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 8.8 A cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits.

(a) In terms of the parameters described, what is the cache capacity, C?

(b) In terms of the parameters described, what is the total number of bits required to store the tags?

(c) What are S and N for a fully associative cache of capacity C words with block size b?

(d) What is S for a direct mapped cache of size C words and block size b?

Exercise 8.9 A 16-word cache has the parameters given in Exercise 8.8. Consider the following repeating sequence of `LDR` addresses (given in hexadecimal):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignoring startup effects (i.e., compulsory misses).

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 1$ word

(c) two-way set associative cache, $b = 1$ word

(d) direct mapped cache, $b = 2$ words

Exercise 8.10 Repeat Exercise 8.9 for the following repeating sequence of `LDR` addresses (given in hexadecimal) and cache configurations. The cache capacity is still 16 words.

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

(a) direct mapped cache, $b = 1$ word

(b) fully associative cache, $b = 2$ words

(c) two-way set associative cache, $b = 2$ words

(d) direct mapped cache, $b = 4$ words

Exercise 8.11 Suppose you are running a program with the following data access pattern. The pattern is executed only once.

0x0 0x8 0x10 0x18 0x20 0x28

(a) If you use a direct mapped cache with a cache size of 1 KB and a block size of 8 bytes (2 words), how many sets are in the cache?

(b) With the same cache and block size as in part (a), what is the miss rate of the direct mapped cache for the given memory access pattern?

(c) For the given memory access pattern, which of the following would decrease the miss rate the most? (Cache capacity is kept constant.) Circle one.

(i) Increasing the degree of associativity to 2.

(ii) Increasing the block size to 16 bytes.

(iii) Either (i) or (ii).

(iv) Neither (i) nor (ii).

Exercise 8.12 You are building an instruction cache for an ARM processor. It has a total capacity of $4C = 2^{c+2}$ bytes. It is $N = 2^n$-way set associative ($N \geq 8$), with a block size of $b = 2^{b'}$ bytes ($b \geq 8$). Give your answers to the following questions in terms of these parameters.

(a) Which bits of the address are used to select a word within a block?

(b) Which bits of the address are used to select the set within the cache?

(c) How many bits are in each tag?

(d) How many tag bits are in the entire cache?

Exercise 8.13 Consider a cache with the following parameters: N (associativity) = 2, b (block size) = 2 words, W (word size) = 32 bits, C (cache size) = 32 K words, A (address size) = 32 bits. You need consider only word addresses.

(a) Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field.

(b) What is the size of *all* the cache tags in bits?

(c) Suppose each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?

(d) Design the cache using the building blocks in Figure 8.28 and a small number of two-input logic gates. The cache design must include tag storage, data

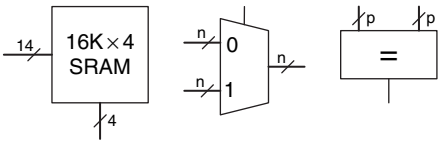


Figure 8.28 Building blocks

storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (n or p bits wide, respectively), but the SRAM blocks must be 16K × 4 bits. Be sure to include a neatly labeled block diagram. You need only design the cache for reads.

Exercise 8.14 You've joined a hot new Internet startup to build wrist watches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in Figure 8.29. The processor includes a small on-chip cache in addition to a large off-chip second-level cache. (Yes, the watch weighs 3 pounds, but you should see it surf!)

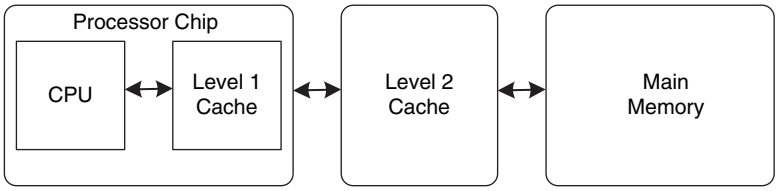


Figure 8.29 Computer system

Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.5. The DRAM has an access time of t_m and a size of 512 MB.

Table 8.5 Memory characteristics

Characteristic	On-chip Cache	Off-chip Cache
Organization	Four-way set associative	Direct mapped
Hit rate	A	B
Access time	t_a	t_b
Block size	16 bytes	16 bytes
Number of blocks	512	256K

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?

(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?

(c) Give an expression for the average memory read access time. The caches are accessed in sequence.

(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.

Exercise 8.15 This chapter described the least recently used (LRU) replacement policy for multiway associative caches. Other, less common, replacement policies include first-in-first-out (FIFO) and random policies. FIFO replacement evicts the block that has been there the longest, regardless of how recently it was accessed. Random replacement randomly picks a block to evict.

(a) Discuss the advantages and disadvantages of each of these replacement policies.

(b) Describe a data access pattern for which FIFO would perform better than LRU.

Exercise 8.16 You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the ARM multicycle processor from Figure 7.30 running at 1 GHz.

(a) Suppose the instruction cache is perfect (i.e., always hits) but the data cache has a 5% miss rate. On a cache miss, the processor stalls for 60 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.5 that has 25% loads, 10% stores, 13% branches, and 52% data-processing instructions. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 7% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.17 Repeat Exercise 8.16 with the following parameters.

(a) The instruction cache is perfect (i.e., always hits) but the data cache has a 15% miss rate. On a cache miss, the processor stalls for 200 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.5 that has 25% loads, 10% stores, 13% branches, and 52% data-processing instructions. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 10% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.18 If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 *terabyte*, 2^{50} bytes = 1 *petabyte*, and 2^{60} bytes = 1 *exabyte*.

Exercise 8.19 A supercomputer designer chooses to spend $1 million on DRAM and the same amount on hard disks for virtual memory. Using the prices from Figure 8.4, how much physical and virtual memory will the computer have? How many bits of physical and virtual addresses are necessary to access this memory?

Exercise 8.20 Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space, but are limited to only 8 MB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) Suppose that you come up with a direct mapped scheme that maps virtual pages to physical pages. The mapping uses the least significant bits of the virtual page number to determine the physical page number. How many virtual pages are mapped to each physical page? Why is this "direct mapping" a bad plan?

(f) Clearly, a more flexible and dynamic scheme for translating virtual addresses into physical addresses is required than the one described in part (e). Suppose you use a page table to store mappings (translations from virtual page number to physical page number). How many page table entries will the page table contain?

(g) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(h) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.21 Consider a virtual memory system that can address a total of 2^{50} bytes. You have unlimited hard drive space, but are limited to 2 GB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) How many page table entries will the page table contain?

(f) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (*V*) and a dirty bit (*D*). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(g) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.22 You decide to speed up the virtual memory system of Exercise 8.20 by using a translation lookaside buffer (TLB). Suppose your memory system has the characteristics shown in Table 8.6. The TLB and cache miss rates indicate how

Table 8.6 Memory characteristics

Memory Unit	Access Time (Cycles)	Miss Rate
TLB	1	0.05%
Cache	1	2%
Main memory	100	0.0003%
Hard drive	1,000,000	0%

often the requested entry is not found. The main memory miss rate indicates how often page faults occur.

(a) What is the average memory access time of the virtual memory system before and after adding the TLB? Assume that the page table is always resident in physical memory and is never held in the data cache.

(b) If the TLB has 64 entries, how big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(c) Sketch the TLB. Clearly label all fields and dimensions.

(d) What size SRAM would you need to build the TLB described in part (c)? Give your answer in terms of depth × width.

Exercise 8.23 You decide to speed up the virtual memory system of Exercise 8.21 by using a translation lookaside buffer (TLB) with 128 entries.

(a) How big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(b) Sketch the TLB. Clearly label all fields and dimensions.

(c) What size SRAM would you need to build the TLB described in part (b)? Give your answer in terms of depth × width.

Exercise 8.24 Suppose the ARM multicycle processor described in Section 7.4 uses a virtual memory system.

(a) Sketch the location of the TLB in the multicycle processor schematic.

(b) Describe how adding a TLB affects processor performance.

Exercise 8.25 The virtual memory system you are designing uses a single-level page table built from dedicated hardware (SRAM and associated logic). It supports 25-bit virtual addresses, 22-bit physical addresses, and 2^{16}-byte (64 KB) pages. Each page table entry contains a physical page number, a valid bit (V), and a dirty bit (D).

(a) What is the total size of the page table, in bits?

(b) The operating system team proposes reducing the page size from 64 to 16 KB, but the hardware engineers on your team object on the grounds of added hardware cost. Explain their objection.

(c) The page table is to be integrated on the processor chip, along with the on-chip cache. The on-chip cache deals only with physical (not virtual) addresses. Is it possible to access the appropriate set of the on-chip cache concurrently with the page table access for a given memory access? Explain briefly the relationship that is necessary for concurrent access to the cache set and page table entry.

(d) Is it possible to perform the tag comparison in the on-chip cache concurrently with the page table access for a given memory access? Explain briefly.

Exercise 8.26 Describe a scenario in which the virtual memory system might affect how an application is written. Be sure to include a discussion of how the page size and physical memory size affect the performance of the application.

Exercise 8.27 Suppose you own a personal computer (PC) that uses 32-bit virtual addresses.

(a) What is the maximum amount of virtual memory space each program can use?

(b) How does the size of your PC's hard drive affect performance?

(c) How does the size of your PC's physical memory affect performance?

Interview Questions

The following exercises present questions that have been asked on interviews.

Question 8.1 Explain the difference between direct mapped, set associative, and fully associative caches. For each cache type, describe an application for which that cache type will perform better than the other two.

Question 8.2 Explain how virtual memory systems work.

Question 8.3 Explain the advantages and disadvantages of using a virtual memory system.

Question 8.4 Explain how cache performance might be affected by the virtual page size of a memory system.

I/O Systems

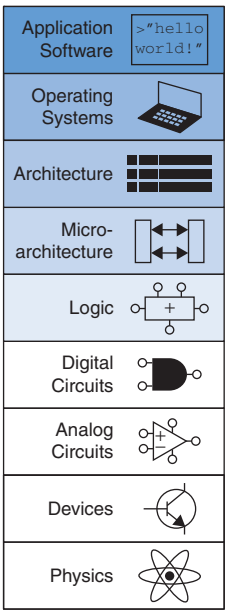
9

9.1 INTRODUCTION

Input/Output (I/O) systems are used to connect a computer with external devices called peripherals. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

This chapter provides concrete examples of I/O devices. Section 9.2 shows the basic principles of interfacing an I/O device to a processor and accessing it from a program. Section 9.3 examines I/O in the context of embedded systems, showing how to use an ARM-based Raspberry Pi single-board computer to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers. Section 9.4 gives examples of interfacing with other common devices such as character LCDs, VGA monitors, Bluetooth radios, and motors. Section 9.5 describes bus interfaces and illustrates the popular AHB-Lite bus. Section 9.6 surveys the major I/O systems used in PCs.

The rest of this chapter is available online as a downloadable PDF from the book's companion site: http://booksite.elsevier.com/9780128000564.



Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00009-1

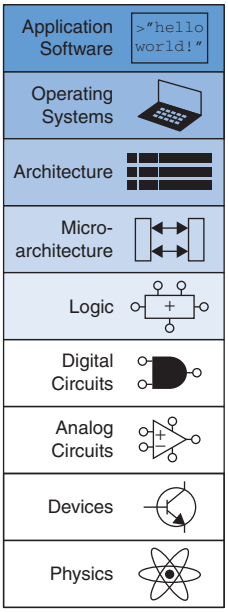
I/O Systems

e9

9.1 INTRODUCTION

Input/Output (I/O) systems are used to connect a computer with external devices called peripherals. In a personal computer, the devices typically include keyboards, monitors, printers, and wireless networks. In embedded systems, devices could include a toaster's heating element, a doll's speech synthesizer, an engine's fuel injector, a satellite's solar panel positioning motors, and so forth. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

This chapter provides concrete examples of I/O devices. Section 9.2 shows the basic principles of interfacing an I/O device to a processor and accessing it from a program. Section 9.3 examines I/O in the context of embedded systems, showing how to use an ARM-based Raspberry Pi single-board computer to access on-board peripherals including general-purpose, serial, and analog I/O as well as timers. Section 9.4 gives examples of interfacing with other common devices such as character LCDs, VGA monitors, Bluetooth radios, and motors. Section 9.5 describes bus interfaces and illustrates the popular AHB-Lite bus. Section 9.6 surveys the major I/O systems used in PCs.



9.2 MEMORY-MAPPED I/O

Recall from Section 6.5.1 that a portion of the address space is dedicated to I/O devices rather than memory. For example, suppose that physical addresses in the range 0x20000000 to 0x20FFFFFF are used for I/O. Each I/O device is assigned one or more memory addresses in this range. A store to the specified address sends data to the device. A load receives data from the device. This method of communicating with I/O devices is called *memory-mapped* I/O.

In a system with memory-mapped I/O, a load or store may access either memory or an I/O device. Figure e9.1 shows the hardware needed to support two memory-mapped I/O devices. An address decoder determines which device communicates with the processor. It uses the *Address* and *MemWrite* signals to generate control signals for the rest of the hardware. The *ReadData* multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00015-7

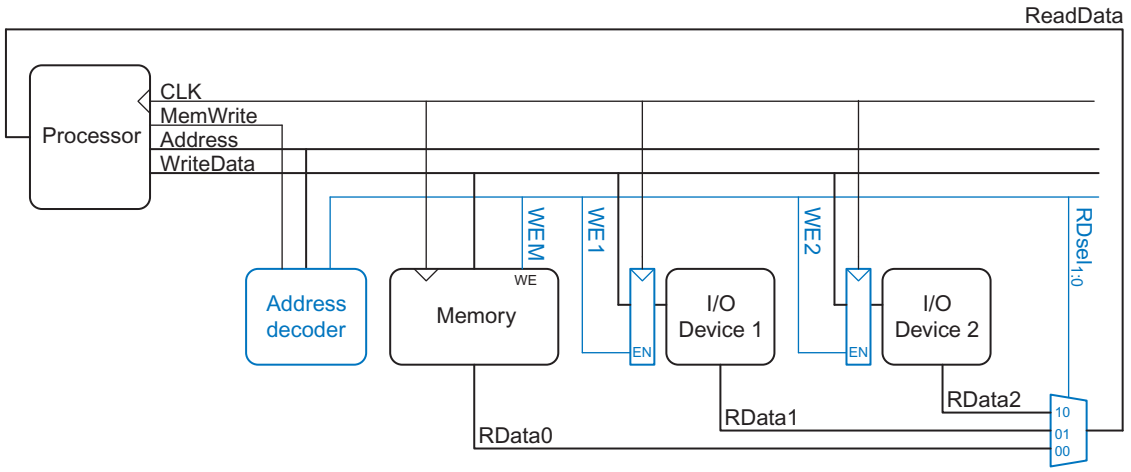


Figure e9.1 **Support hardware for memory-mapped I/O**

Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the ARM assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution: The following assembly code writes the value 7 to I/O Device 1.

```
      MOV R1, #7
      LDR R2, =ioadr
      STR R1, [R2]
ioadr DCD 0x20001000
```

The address decoder asserts *WE1* because the address is 0x20001000 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

To read from I/O Device 1, the processor executes the following assembly code.

```
  LDR R1, [R2]
```

The address decoder sets $RDsel_{1:0}$ to 01, because it detects the address 0x20001000 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into R1 in the processor.

The addresses associated with I/O devices are often called *I/O registers* because they may correspond with physical registers in the I/O device like those shown in Figure e9.1.

Software that communicates with an I/O device is called a *device driver*. You have probably downloaded or installed device drivers for your printer or other I/O device. Writing a device driver requires detailed knowledge about the I/O device hardware including the addresses and behavior of the memory-mapped I/O registers. Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.

Some architectures, notably x86, use specialized instructions instead of memory-mapped I/O to communicate with I/O devices. These instructions are of the following form, where `device1` and `device2` are the unique IDs of the peripheral device:

```
LDRIO R1, device1
STRIO R2, device2
```

This type of communication with I/O devices is called *programmed I/O*.

9.3 EMBEDDED I/O SYSTEMS

Embedded systems use a processor to control interactions with the physical environment. They are typically built around microcontroller units (MCUs) which combine a microprocessor with a set of easy-to-use peripherals such as general-purpose digital and analog I/O pins, serial ports, timers, etc. Microcontrollers are generally inexpensive and are designed to minimize system cost and size by integrating most of the necessary components onto a single chip. Most are smaller and lighter than a dime, consume milliwatts of power, and range in cost from a few dimes up to several dollars. Microcontrollers are classified by the size of data that they operate upon. 8-bit microcontrollers are the smallest and least expensive, while 32-bit microcontrollers provide more memory and higher performance.

Approximately $19B of microcontrollers were sold in 2014, and the market is forecast to reach $27B by 2020. The average price of a microcontroller is less than $1, and an 8-bit microcontroller can be integrated on a system-on-chip for less than a penny. Microcontrollers have become ubiquitous and nearly invisible, with an estimated 150 in each home and 50 in each automobile in 2010. The 8051 is a classic 8-bit microcontroller originally developed by Intel in 1980 and now sold by a host of manufacturers. Microchip's PIC16 and PIC18-series are 8-bit market leaders. The Atmel AVR series of microcontrollers has been popularized among hobbyists as the brain of the Arduino platform. Among 32-bit microcontrollers, Renesas leads the overall market. Freescale, Samsung, Texas Instruments, and Infineon are other major microcontroller manufacturers. ARM processors are found in nearly all smart phones and tablets today and are usually part of a system-on-chip containing the multi-core applications processor, a graphics processing unit, and extensive I/O.

For the sake of concreteness, this section will illustrate embedded system I/O in the context of a real system. Specifically, we will focus on the popular and inexpensive Raspberry Pi board, which contains a Broadcom BCM2835 system-on-chip (SoC) with a 700 MHz 32-bit ARM1176JZ-F processor implementing the ARMv6 instruction set. The principles in each subsection will be followed by specific examples that run on the Pi. All of the examples have been tested on a Pi running NOOBS Raspbian Linux in 2014.

Figure e9.2 shows a photograph of a Raspberry Pi Model B+ board, which is a complete Linux computer about the size of a credit card that sells for $35. The Pi draws up to 1 A from a 5 V USB power supply. It has 512 MB of onboard RAM and an SD card socket for a memory card that contains the operating system and user files. Connectors provide video and audio output, USB ports for a mouse and keyboard, and an Ethernet (Local Area Network) port, along with 40 general-purpose I/O (GPIO) pins that are the main subject of this chapter.

While the BCM2835 SoC has many capabilities beyond those in a typical inexpensive microcontroller, the general-purpose I/O is very similar. This chapter begins by describing the BCM2835 on the Raspberry Pi and describing a device driver for memory-mapped I/O. The remainder of this chapter will illustrate how embedded systems perform general-purpose digital, analog, and serial I/O. Timers are also commonly used to generate or measure precise time intervals.

The Raspberry Pi was developed in 2011-12 by the nonprofit Raspberry Pi Foundation in the UK to promote teaching computer science. Built around the brain of an inexpensive smartphone, the computer has become wildly popular, selling more than 3 million units by 2014. The name pays homage to early home computers including Apple, Apricot, and Tangerine. Pi is derived from Python, a programming language often used in education. Documentation and purchasing information can be found at

raspberrypi.org

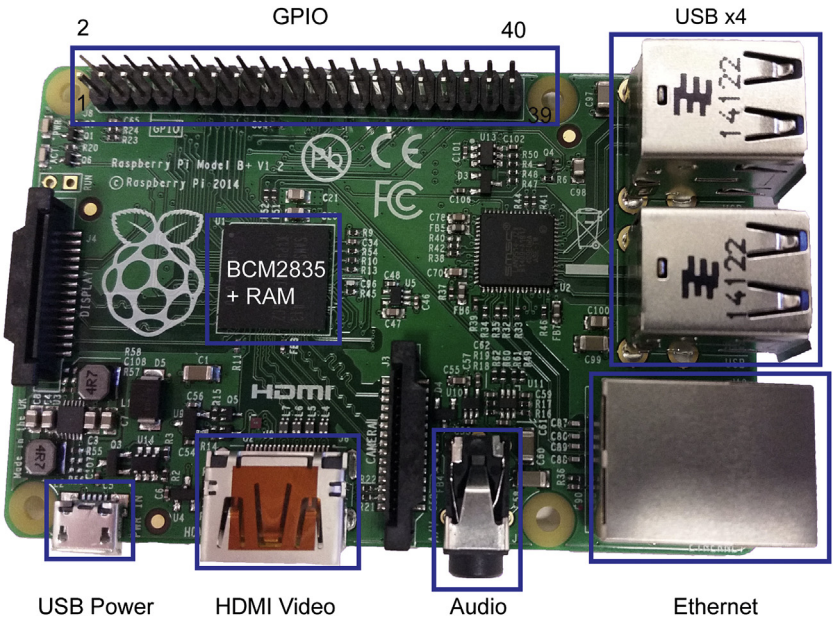


Figure e9.2 **Raspbery Pi Model B+**

Eben Upton (1978-) is the architect of the Raspberry Pi and a founder of the Raspberry Pi Foundation. He received his Bachelor's and Ph.D. from the University of Cambridge before joining Broadcom Corporation as a chip architect.

(Photograph © Eben Upton. Reproduced with permission.)

9.3.1 BCM2835 System-on-Chip

The BCM2835 SoC is a powerful yet inexpensive chip designed by Broadcom for mobile devices and other multimedia applications. The SoC includes an ARM microprocessor known as the *applications processor*, a VideoCore processor for graphics, video, and cameras, and many I/O peripherals. The BCM2835 is packaged in a plastic ball grid array with tiny solder balls underneath; it is best soldered by a robot that aligns the package to matching copper pads on a printed circuit board and applies heat. Broadcom does not publish a complete datasheet, but an abbreviated datasheet is available on the Raspberry Pi site describing how to access peripherals from the ARM processor. The datasheet describes many features and I/O registers that are omitted in this chapter for simplicity.

www.raspberrypi.org/documentation/hardware/

Figure e9.3 shows a simplified schematic of the Raspberry Pi model board. The board receives 5 V power from a USB power supply and regulators produce 3.3, 2.5, and 1.8 V levels for I/O, analog, and

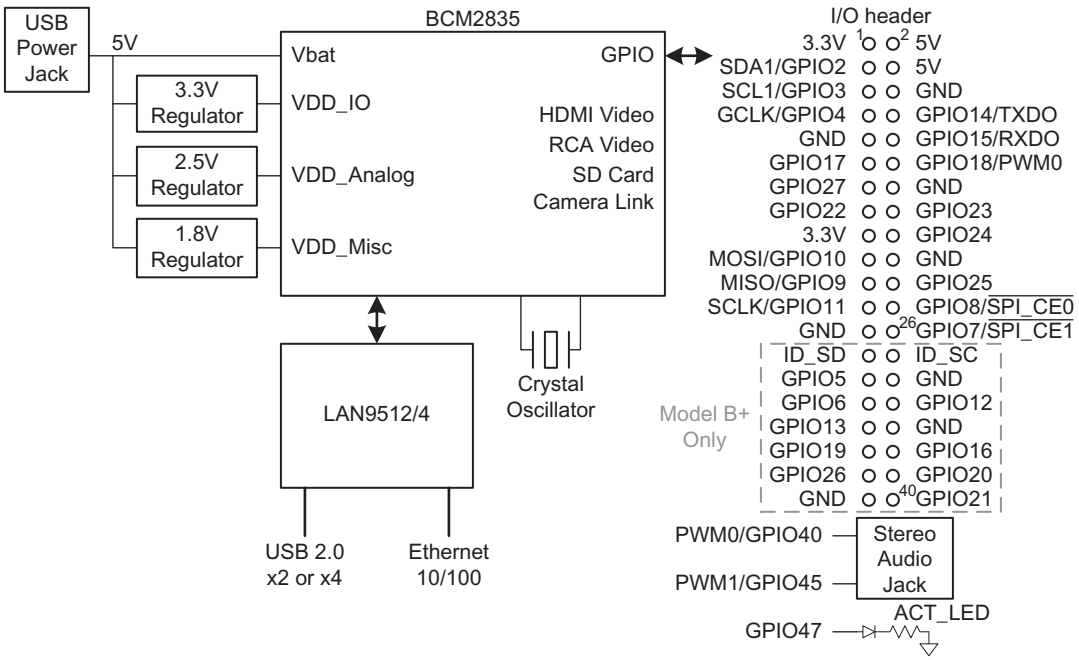


Figure e9.3 **Raspberry PI I/O schematic**

miscellaneous functions. The BCM2835 also has an internal switching regulator that produces a variable lower voltage for the power-efficient SoC. The BCM2835 connects to a USB/Ethernet controller and also directly outputs video. It also has 54 configurable I/O signals but, for space reasons, only a fraction of these are accessible to the user via header pins. The header also provides 3.3 and 5 V and ground to conveniently power small devices attached to the Pi, but the maximum total current is 50 mA from 3.3 V and ~300 mA from 5 V. The model B and B+ are similar, but B+ boosts the number of I/O header pins from 26 to 40 and the number of USB ports from 2 to 4. Various cables including the Adafruit Pi Cobbler are available to connect these header pins to a breadboard.

The Raspberry Pi uses an SD card as a Flash memory disk. The card is typically preloaded with Raspbian Linux, a small version of Linux that fits on an 8 GB SD card. You can work with the Pi either by attaching an HDMI monitor and USB mouse and keyboard to turn it into a full computer, or by connecting to it from another computer over an Ethernet cable.

The Raspberry Pi continues to advance and by the time you read this, a newer model might be available with a more advanced processor and a different set of embedded I/O. Nevertheless, the same principles will apply, and the principles also apply to other types of microcontrollers. You can expect to find the same types of I/O peripherals. You will need to consult the data sheet to look up the mapping between the peripheral, the pin on the chip, and the pin on the board, as well as the addresses of the memory-mapped I/O registers associated with each peripheral. You'll write configuration registers to initialize the peripheral and read and write data registers to access the peripheral.

As this book was going to press, the Raspberry Pi Foundation released the Raspberry Pi 2 Model B with a BCM2836 SoC containing a quad Cortex-A7 processor and 1 GB of RAM. The Pi 2 runs about 6 times faster than the B+ but has the same I/O as the B+ described in this chapter. The peripheral base address has moved from 0x20000000 to 0x3F000000. An updated EasyPIO supporting both models is posted to the textbook website.

Caution: the I/O connector pinout has changed between Raspberry Pi board revisions.

Caution: connecting 5 V to one of the 3.3 V I/Os will damage the I/O and possibly the entire Raspberry Pi. If you probe the I/O pins with a voltmeter, beware that you do not accidentally make contact between the 5 V pins and a nearby pin!

Pin 1 of the I/O header is labeled in Figure e9.3. When you make connections, be sure you have properly identified it and aren't rotated by 180 degrees. This is an easy mistake that could cause you to accidentally damage the Pi.

EasyPIO and the code examples in this chapter can be downloaded from the textbook website: http://booksite.elsevier.com/9780128000564. The WiringPi driver and documentation is at `wiringpi.com`.

9.3.2 Device Drivers

Programmers can manipulate I/O devices directly by reading or writing the memory-mapped I/O registers. However, it is better programming practice to call functions that access the memory-mapped I/O. These functions are called *device drivers*. Some of the benefits of using device drivers include:

- The code is easier to read when it involves a clearly named function call rather than a write to bit fields at an obscure memory address.
- Somebody who is familiar with the deep workings of the I/O devices can write the device driver and casual users can call it without having to understand the details.
- The code is easier to port to another processor with different memory mapping or I/O devices because only the device driver must change.
- If the device driver is part of the operating system, the OS can control access to physical devices shared among multiple programs running on the system and can manage security (e.g. so a malicious program can't read the keyboard while you are typing your password into a web browser).

This section will develop a simple device driver called EasyPIO to access BCM2835 devices so that you can understand what is happening under the hood in a device driver. Casual users are likely to prefer WiringPi, an open-source I/O library for the Pi, which has functions similar to but not exactly matching those in EasyPIO.

The memory-mapped I/O on the BCM2835 is found at physical addresses 0x20000000-0x20FFFFFF. The physical base addresses used by various peripherals are summarized in Table e9.1. Peripherals have multiple I/O registers starting at their base address. For example, reading address 0x20200034 will return the values of GPIO (general-purpose I/O) pins 31:0. The peripherals in bold will be discussed further in subsequent sections.

The Raspberry Pi typically runs a Linux operating system using virtual memory, which further complicates memory-mapped I/O. Loads and stores in a program refer to virtual addresses, not physical, so a program cannot immediately access memory-mapped I/O. Instead, it must begin by asking the operating system to map the physical addresses of interest to the program's virtual address space. The `pioInit` function from EasyPIO in Example e9.2 performs this task. The code involves some heavy duty pointer manipulation in C. The general principle is to open `/dev/mem`, which is a Linux method of accessing physical memory. Then the `mmap` function is used to set `gpio` as a pointer to physical address 0x20200000, the beginning of the GPIO registers. The pointer is declared

Table e9.1 Memory mapped I/O addresses

Physical Base Address	Peripheral
0x20003000	**System Timer**
0x2000B200	Interrupts
0x2000B400	ARM Timer
0x20200000	**GPIO**
0x20201000	**UART0**
0x20203000	PCM Audio
0x20204000	**SPI0**
0x20205000	I^2C Master #1
0x2020C000	**PWM**
0x20214000	I^2C Slave
0x20215000	miniUART1, SPI1, SPI2
0x20300000	SD Card Controller
0x20804000	I^2C Master #2
0x20805000	I^2C Master #3

`volatile`, telling the compiler that the memory-mapped I/O value might change on its own, so the program should always read the register directly instead of relying on an old value. GPLEV0 accesses the I/O register 13 words past GPIO, e.g. at 0x20200034, which contains the values of GPIO 31:0. For brevity, this example omits error checking that takes place in the actual EasyPIO library. Subsequent subsections define more registers and functions to access I/O devices.

Example e9.2 INITIALIZING MEMORY-MAPPED I/O

```
#include <sys/mman.h>
#define BCM2835_PERI_BASE  0x20000000
#define GPIO_BASE          (BCM2835_PERI_BASE + 0x200000)
volatile unsigned int      *gpio; //Pointer to base of gpio
#define GPLEV0             (*(volatile unsigned int *)(gpio + 13))
#define BLOCK_SIZE         (4*1024)
```

For security reasons, Linux only grants the *superuser* access to memory-mapped hardware. To run a program as the superuser, type `sudo` before the Linux command. The next section will give an example.

```
void pioInit(){
    int  mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in Linux
    mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
    reg_map = mmap(
      NULL,                  // Address at which to start local mapping (null = don't-care)
      BLOCK_SIZE,            // 4KB mapped memory block
      PROT_READ|PROT_WRITE,  // Enable both reading and writing to the mapped memory
      MAP_SHARED,            // Nonexclusive access to this memory
      mem_fd,                // Map to /dev/mem
      GPIO_BASE);            // Offset to GPIO peripheral

    gpio = (volatile unsigned *)reg_map;
    close(mem_fd);
}
```

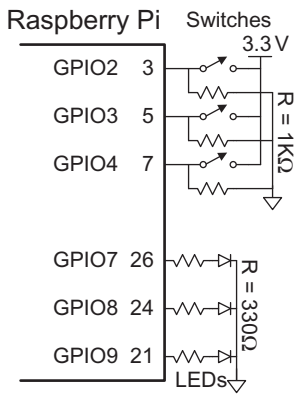


Figure e9.4 LEDs and switches connected to GPIO pins

9.3.3 General-Purpose Digital I/O

General-purpose I/O (GPIO) pins are used to read or write digital signals. For example, Figure e9.4 shows three light-emitting diodes (LEDs) and three switches connected to six GPIO pins. The LEDs are wired to glow when driven with a 1 and to turn off when driven with a 0. The current-limiting resistors are placed in series with the LEDs to set the brightness and avoid overloading the current capability of the GPIO. The switches are wired to produce a 1 when closed and a 0 when open. The schematic indicates the pin name as well as the corresponding header pin number.

At a minimum, any GPIO pin requires registers to read input pin values, write output pin values, and set the direction of the pin. In many embedded systems, the GPIO pins can be shared with one or more special-purpose peripherals, so additional configuration registers are necessary to determine whether the pin is general or special-purpose. Furthermore, the processor may generate interrupts when an event such as a rising or falling edge occurs on an input pin, and configuration registers may be used to specify the conditions for an interrupt.

Recall that the BCM2835 has 54 GPIOs. They are controlled by the GPFSEL, GPLEV, GPSET, and GPCLR registers. Figure e9.5 shows a memory map for these GPIO registers. GPFSEL5…0 determine whether each pin is a general-purpose input, output, or special-purpose I/O. Each of these function select registers uses 3 bits to specify each pin and thus

In the context of bit manipulation, "setting" means writing to 1 and "clearing" means writing to 0.

Table e9.2 GPFSEL register bit field to GPIO mapping

	GPFSEL0	GPFSEL1	GPFSEL2	GPFSEL3	GPFSEL4	GPFSEL5
[2:0]	GPIO0	GPIO10	GPIO20	GPIO30	GPIO40	GPIO50
[5:3]	GPIO1	GPIO11	GPIO21	GPIO31	GPIO41	GPIO51
[8:6]	GPIO2	GPIO12	GPIO22	GPIO32	GPIO42	GPIO52
[11:9]	GPIO3	GPIO13	GPIO23	GPIO33	GPIO43	GPIO53
[14:12]	GPIO4	GPIO14	GPIO24	GPIO34	GPIO44	
[17:15]	GPIO5	GPIO15	GPIO25	GPIO35	GPIO45	
[20:18]	GPIO6	GPIO16	GPIO26	GPIO36	GPIO46	
[23:21]	GPIO7	GPIO17	GPIO27	GPIO37	GPIO47	
[26:24]	GPIO8	GPIO18	GPIO28	GPIO38	GPIO48	
[29:27]	GPIO9	GPIO19	GPIO29	GPIO39	GPIO49	

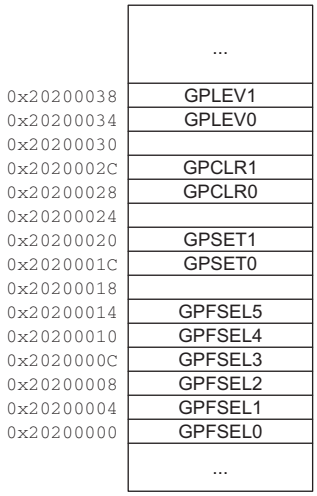


Figure e9.5 GPIO memory map

Table e9.3 GPFSEL configuration

GPFSEL	Pin Function
000	Input
001	Output
010	ALT5
011	ALT4
100	ALT0
101	ALT1
110	ALT2
111	ALT3

each 32-bit register controls 10 GPIOs as given in Table e9.2 and six GPFSEL registers are necessary to control all 54 GPIOs. For example, GPIO13 is configured by GPFSEL1[11:9]. The configurations are summarized in Table e9.3; many pins have multiple special-purpose functions that will be discussed in subsequent sections; ALT0 is most commonly used. Reading GPLEV1…0 returns the values of the pins. For example, GPIO14 is read as GPLEV0[14] and GPIO34 is read as GPLEV1[2]. The pins cannot be directly written; instead, bits are forced high or low by asserting the corresponding bit of GPSET1…0 or GPCLR1…0. For example, GPIO14 is forced to 1 by writing GPSET0[14] = 1 and forced to 0 by writing GPCLR0[14] = 1.

The BCM2835 datasheet does not specify the logic levels or output current capability of the GPIOs. However, users have determined empirically that one should not try to draw more than 16 mA from any single I/O or 50 mA total from all the I/Os. Thus, a GPIO pin is suitable for driving a small LED but not a motor. The I/Os are generally compatible with other 3.3 V chips but are not 5 V-tolerant.

The BCM2835 has unusually complex GPIO access. Some microcontrollers use a single register to configure whether each pin is input or output and another register to read and write the pins.

Example e9.3 GPIO FOR SWITCHES AND LEDS

Enhance EasyPIO with `pinMode`, `digitalRead`, and `digitalWrite` functions to configure a pin's direction and read or write it. Write a C program using these functions to read the three switches and turn on the corresponding LEDs using the hardware in Figure e9.4.

Solution: The additional EasyPIO code is given below. Because multiple registers are used to control the I/O, the functions must compute which register to access and what bit offset to use within the register. `pinMode` then clears the 0 bits and sets the 1 bits for the intended 3-bit function. `digitalWrite` handles writing either 1 or 0 by using GPSET or GPCLR. `digitalRead` pulls out the value of the desired pin and masks off the others.

```
#define GPFSEL  ((volatile unsigned int *)(gpio + 0))
#define GPSET   ((volatile unsigned int *)(gpio + 7))
#define GPCLR   ((volatile unsigned int *)(gpio + 10))
#define GPLEV   ((volatile unsigned int *)(gpio + 13))
#define INPUT   0
#define OUTPUT  1
...
void pinMode(int pin, int function) {
    int reg       =  pin/10;
    int offset    = (pin%10)*3;
    GPFSEL[reg] &= ~((0b111 & ~function) << offset);
    GPFSEL[reg] |=  ((0b111 &  function) << offset);
}

void digitalWrite(int pin, int val) {
    int reg     = pin / 32;
    int offset  = pin % 32;

    if  (val)  GPSET[reg] = 1 << offset;
    else       GPCLR[reg] = 1 << offset;
}

int digitalRead(int pin) {
    int reg     = pin / 32;
    int offset  = pin % 32;

    return (GPLEV[reg] >> offset) & 0x00000001;
}
```

The program to read switches and write LEDs is given below. It initializes GPIO access, then sets pins 2–4 as inputs for the switches and pins 7–9 as outputs for the LEDs. It then continuously reads the switches and writes their values to the corresponding LEDs.

```
#include "EasyPIO.h"

void main(void) {
    pioInit();
```

```
    // Set GPIO 4:2 as inputs
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    pinMode(4, INPUT);

    // Set GPIO 9:7 as an output
    pinMode(7, OUTPUT);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);

    while (1) { // Read each switch and write corresponding LED
        digitalWrite(7, digitalRead(2));
        digitalWrite(8, digitalRead(3));
        digitalWrite(9, digitalRead(4));
    }
}
```

Assuming the program is in a file named dip2led.c and that EasyPIO.h is in the same directory, you can compile and run the program using the following commands on the Raspberry Pi command line . `gcc` is the C compiler. Note that `sudo` is required so that the program can access the protected I/O memory. To stop a running program, press Ctrl-C.

```
gcc dip2led.c -o dip2led
sudo ./dip2led
```

9.3.4 Serial I/O

If a microcontroller needs to send more bits than the number of free GPIO pins, it must break the message into multiple smaller transmissions. In each step, it can send either one bit or several bits. The former is called serial I/O and the latter is called parallel I/O. Serial I/O is popular because it uses few wires and is fast enough for many applications. Indeed, it is so popular that many standards for serial I/O have been established and microcontrollers offer dedicated hardware to easily send data via these standards. This section describes the Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver/Transmitter (UART) standard serial interfaces.

Other common serial standards include Inter-Integrated Circuit (I^2C), Universal Serial Bus (USB), and Ethernet. I^2C (pronounced "I squared C") is a 2-wire interface with a clock and a bidirectional data pin; it is used in a fashion similar to SPI. USB and Ethernet are more complex, high-performance standards described in Sections 9.6.1 and 9.6.4, respectively. All five of these standards are supported on the Raspberry Pi.

SPI always sends data in both directions on each transfer. If the system only needs unidirectional communication, it can ignore the unwanted data. For example, if the master only needs to send data to the slave, the byte received from the slave can be ignored. If the master only needs to receive data from the slave, it must still trigger the SPI communication by sending an arbitrary byte that the slave will ignore. It can then read the data received from the slave. The SPI clock only toggles while the master is transmitting data.

9.3.4.1 Serial Peripheral Interface (SPI)

SPI (pronounced "S-P-I") is a simple synchronous serial protocol that is easy to use and relatively fast. The physical interface consists of three pins: Serial Clock (SCK), Master Out Slave In (MOSI, also known as SDO), and Master In Slave Out (MISO, also known as SDI). SPI connects a *master* device to a *slave* device, as shown in Figure e9.6(a). The master produces the clock. It initiates communication by sending a series of clock pulses on SCK. If it wants to send data to the slave, it puts the data on MOSI, starting with the most significant bit. The slave may simultaneously respond by putting data on MISO. Figure e9.6(b) shows the SPI waveforms for an 8-bit data transmission. Bits change on the falling edge of SCK and are stable to sample on the rising edge. The SPI interface may also send an active-low chip enable to alert the receiver that data is coming.

The BCM2835 has three SPI master ports and one slave port. This section describes SPI Master Port 0, which is readily accessible on the Raspberry Pi on GPIO pins 11:9. To use these pins for SPI rather than GPIO, their GPFSEL must be set to ALT0. The Pi must then configure the port. When the Pi writes to the SPI, the data is transmitted serially to the slave. Simultaneously, data received from the slave is collected and the Pi can read it when the transfer is complete.

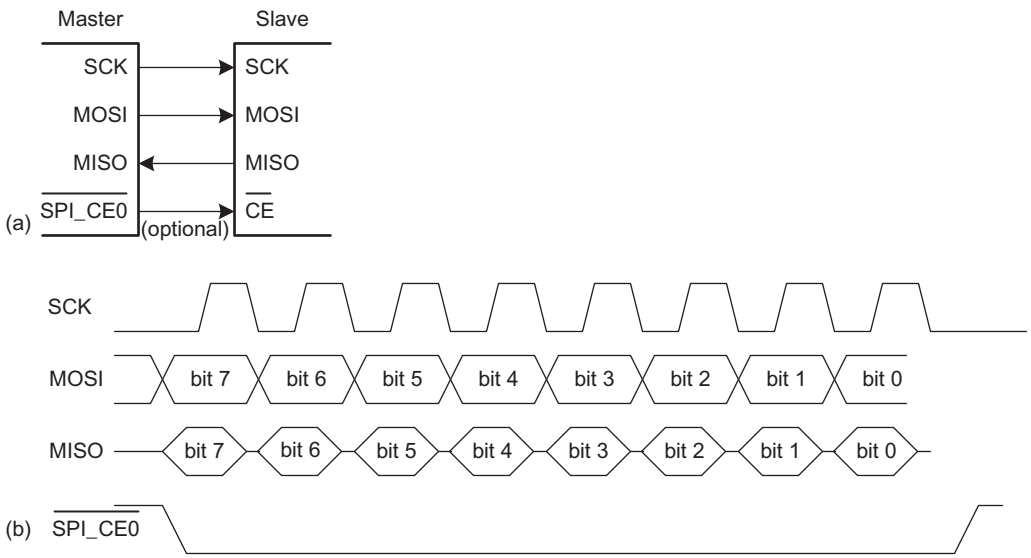


Figure e9.6 SPI connection and waveforms

Table e9.4 SPI0CS register fields

Bit	Name	Function	Meaning for 0	Meaning for 1
16	DONE	Transfer Done	Transfer in progress	Transfer complete
7	TA	Transfer Active	SPI disabled	SPI enabled
3	CPOL	Clock Polarity	Clock idles low	Clock idles high
2	CPHA	Clock Phase	First SCK transition at middle of data bit	First SCK transition at beginning of data bit

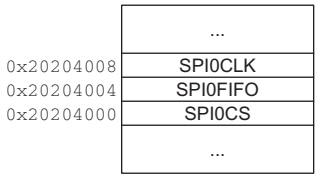


Figure e9.7 SPI Master Port 0 registers

SPI Master Port 0 is associated with three registers, given in the memory map in Figure e9.7. SPI0CS is the control register. It is used to turn the SPI on and set attributes such as the polarity of the clock. Table e9.4 lists the names and functions of some of the bits in SPI0CS that are relevant to this discussion. All have a default value of 0 on reset. Most of the functions, such as chip selects and interrupts, are not used in this section but can be found in the datasheet. SPI0FIFO is written to transmit a byte and read to get the byte received back. SPI0CLK configures the SPI clock frequency by dividing the 250 MHz peripheral clock by a power of two specified in the register. Thus, the SPI clock frequency is summarized in Table e9.5.

If the frequency is too high (>~1 MHz on a breadboard or tens of MHz on an unterminated printed circuit board), the SPI may become unreliable due to reflections, crosstalk, or other signal integrity issues.

Table e9.5 SPI0CLK frequencies

SPI0CLK	SPI Frequency (kHz)
2	125000
4	62500
8	31250
16	15625
32	7812
64	3906
128	1953
256	976
512	488
1024	244
2048	122

Example e9.4 SENDING AND RECEIVING BYTES OVER SPI

Design a system to communicate between a Raspberry Pi master and an FPGA slave over SPI. Sketch a schematic of the interface. Write the C code for the Pi to send the character 'A' and receive a character back. Write HDL code for an SPI slave on the FPGA. How could the slave be simplified if it only needs to receive data?

Solution: Figure e9.8 shows the connection between the devices using SPI Master Port 0. The pin numbers are obtained from the component datasheets (e.g., Figure e9.3). Notice that both the pin numbers and signal names are shown on the diagram to indicate both the physical and logical connectivity. When the SPI is enabled, these pins cannot be used for GPIO.

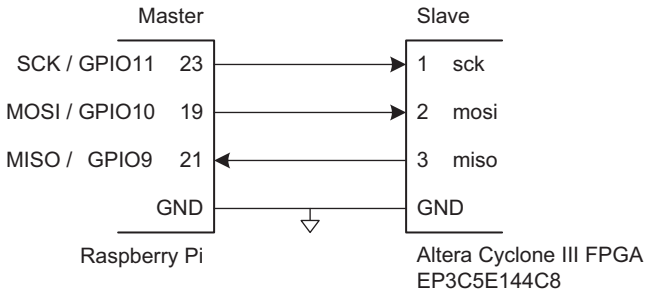


Figure e9.8 SPI connection between Pi and FPGA

The following code from EasyPIO.h is used to initialize the SPI and to send and receive a character. The code to set up the memory map and define the register addresses is similar to that for GPIO and is not reprinted here.

```
void spiInit(int freq, int settings) {
    pinMode(8, ALT0);           // CE0b
    pinMode(9, ALT0);           // MISO
    pinMode(10, ALT0);          // MOSI
    pinMode(11, ALT0);          // SCLK

    SPIOCLK = 250000000/freq; // Set SPI clock divider to desired
                                 freq
    SPIOCS = settings;
    SPIOCSbits.TA = 1;          // Turn SPI on
}

char spiSendReceive(char send){
    SPIOFIFO = send;            // Send data to slave
    while (!SPIOCSbits.DONE);   // Wait until SPI complete
    return SPIOFIFO;            // Return received data
}
```

The C code below initializes the SPI and then sends and receives a character. It sets the SPI clock to 244 kHz.

```
#include "EasyPIO.h"

void main(void) {
    char received;

    pioInit();
    spiInit(244000, 0);                 // Initialize the SPI:
                                        // 244 kHz clk, default settings
    received = spiSendReceive('A');  // Send letter A and receive byte
}
```

The HDL code for the FPGA is listed below. Figure e9.9 shows a block diagram and timing. The FPGA uses a shift register to hold the bits that have been received from the master and the bits that remain to be sent to the master.

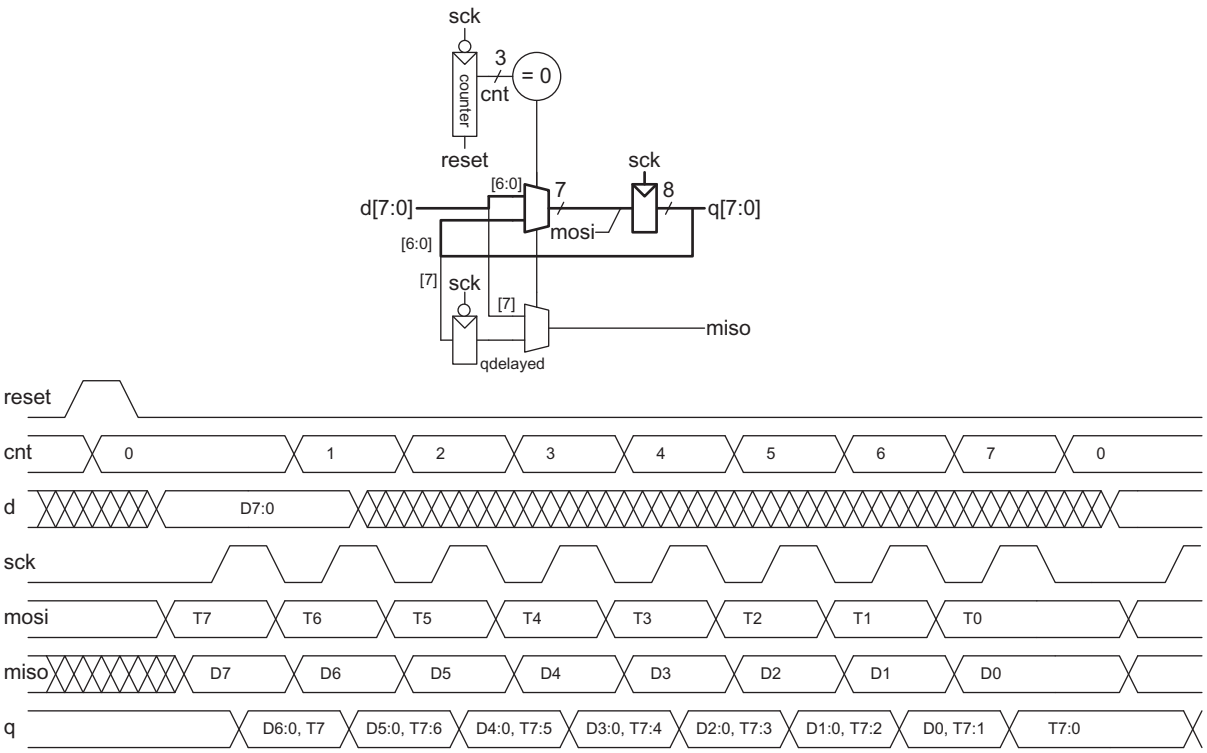


Figure e9.9 **SPI slave circuitry and timing**

On the first rising `sck` edge after reset and each 8 cycles thereafter, a new byte from `d` is loaded into the shift register. On each subsequent cycle, a bit is shifted in on `mosi` and a bit is shifted out of `miso`. `miso` is delayed until the falling edge of `sck` so that it can be sampled by the master on the next rising edge. After 8 cycles, the byte received can be found in `q`.

```
module spi_slave(input  logic       sck,     // From master
                 input  logic       mosi,    // From master
                 output logic       miso,    // To master
                 input  logic       reset,   // System reset
                 input  logic [7:0] d,       // Data to send
                 output logic [7:0] q);      // Data received

  logic [2:0] cnt;
  logic       qdelayed;

  // 3-bit counter tracks when full byte is transmitted
  always_ff @(negedge sck, posedge reset)
    if (reset) cnt = 0;
    else       cnt = cnt + 3'b1;
```

```
  // Loadable shift register
  // Loads d at the start, shifts mosi into bottom on each step
  always_ff @(posedge sck)
    q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

  // Align miso to falling edge of sck
  // Load d at the start
  always_ff @(negedge sck)
    qdelayed = q[7];
  assign miso = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

If the slave only needs to receive data from the master, it reduces to a simple shift register given in the following HDL code.

```
module spi_slave_receive_only(input  logic       sck, //From master
                              input  logic       mosi,//From master
                              output logic [7:0] q);  //Data received
  always_ff @(posedge sck)
    q <= {q[6:0], sdi}; // shift register
endmodule
```

SPI ports are highly configurable so that they can talk to a wide variety of serial devices. Unfortunately, this leads to the possibility of incorrectly configuring the port and garbling the data transmission. Sometimes it is necessary to change the configuration bits to communicate with a device that expects different timing. When CPOL = 1, SCK is inverted. When CPHA = 1, the clocks toggle half a cycle earlier relative to the data. These modes are shown in Figure e9.10. Be aware that different SPI products may use different names and polarities for these options; check the waveforms carefully for your device. It can also be helpful to examine

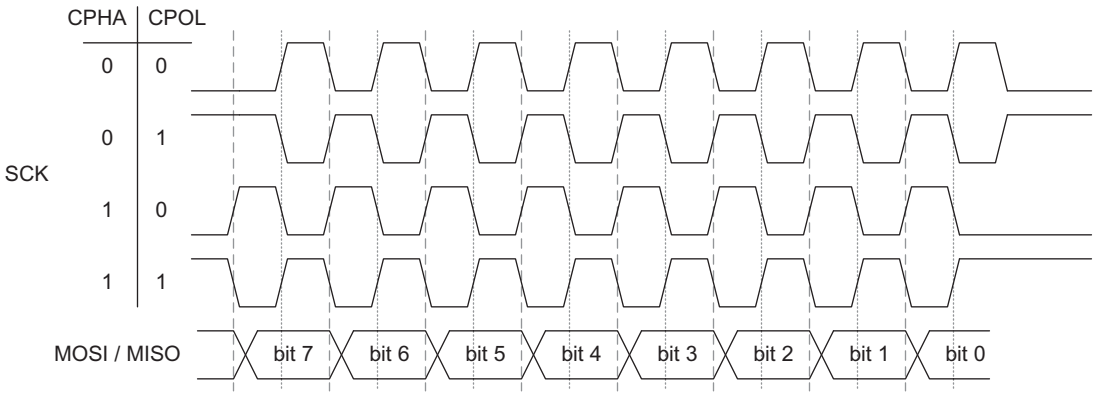


Figure e9.10 SPI clock and data timing configurations

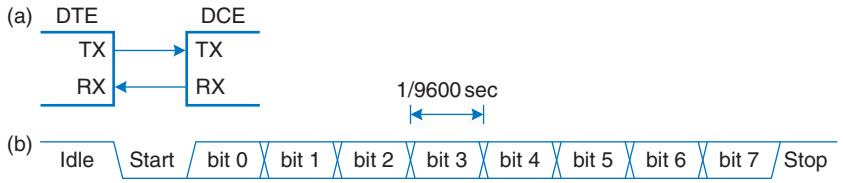


Figure e9.11 Asynchronous serial link

SCK, MOSI, and MISO on an oscilloscope if you are having communication difficulties.

9.3.4.2 Universal Asynchronous Receiver/Transmitter (UART)

A UART (pronounced "you-art") is a serial I/O peripheral that communicates between two systems without sending a clock. Instead, the systems must agree in advance about what data rate to use and must each locally generate its own clock. Hence, the transmission is asynchronous because the clocks are not synchronized. Although these system clocks may have a small frequency error and an unknown phase relationship, the UART manages reliable asynchronous communication. UARTs are used in protocols such as RS-232 and RS-485. For example, old computer serial ports use the RS-232C standard, introduced in 1969 by the Electronics Industries Associations. The standard originally envisioned connecting *Data Terminal Equipment* (DTE) such as a mainframe computer to *Data Communication Equipment* (DCE) such as a modem. Although a UART is relatively slow compared to SPI and prone to misconfiguration issues, the standards have been around for so long that they remain important today.

Figure e9.11(a) shows an asynchronous serial link. The DTE sends data to the DCE over the TX line and receives data back over the RX line. Figure e9.11(b) shows one of these lines sending a character at a data rate of 9600 baud. The lines idle at a logic '1' when not in use. Each character is sent as a start bit (0), 7 or 8 data bits, an optional parity bit, and one or more stop bits (1's). The UART detects the falling transition from idle to start to lock on to the transmission at the appropriate time. Although seven data bits is sufficient to send an ASCII character, eight bits are normally used because they can convey an arbitrary byte of data.

The optional parity bit allows the system to detect if a bit was corrupted during transmission. It can be configured as *even* or *odd*; even parity means that the parity bit is chosen such that the total collection of data and parity has an even number of 1's; in other words, the parity bit is the XOR of the data bits. The receiver can then check if an even number of 1's was received and signal an error if not. Odd parity is the reverse.

Baud rate gives the signaling rate, measured in symbols per second, whereas bit rate gives the data rate, measured in bits per second. The signaling we've discussed in this text is 2-level signaling, where each symbol represents a bit. However, multi-level signaling can send multiple bits per symbol; for example, 4-level signaling sends two bits per symbol. In that case, the bit rate is twice the baud rate. In a simple system like SPI where each symbol is a bit and each symbol represents data, the baud rate is equal to the bit rate. UARTs and some other signaling conventions require overhead bits in addition to the data. For example, a two-level signaling system that adds start and stop bits for each 8 bits of data and operates at a baud rate of 9600 has a bit rate of (9600 symbols/second)×(8 bits/10 symbols) = 7680 bits/second = 960 characters/second.

In the 1950s through 1970s, early hackers calling themselves phone phreaks learned to control the phone company switches by whistling appropriate tones. A 2600 Hz tone produced by a toy whistle from a Cap'n Crunch cereal box e9.12 could be exploited to place free long-distance and international calls.

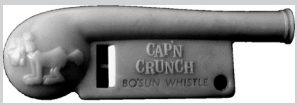


Figure e9.12 Cap'n Crunch Bosun Whistle
(Photograph by Evrim Sen, reprinted with permission.)

Handshaking refers to the negotiation between two systems; typically, one system signals that it is ready to send or receive data, and the other system acknowledges that request.

A common choice is 8 data bits, no parity, and 1 stop bit, making a total of 10 symbols to convey an 8-bit character of information. Hence, signaling rates are referred to in units of baud rather than bits/sec. For example, 9600 baud indicates 9600 symbols/sec, or 960 characters/sec,. Both systems must be configured for the appropriate baud rate and number of data, parity, and stop bits or the data will be garbled. This is a hassle, especially for nontechnical users, which is one of the reasons that the Universal Serial Bus (USB) has replaced UARTs in personal computer systems.

Typical baud rates include 300, 1200, 2400, 9600, 14400, 19200, 38400, 57600, and 115200. The lower rates were used in the 1970's and 1980's for modems that sent data over the phone lines as a series of tones. In contemporary systems, 9600 and 115200 are two of the most common baud rates; 9600 is encountered where speed doesn't matter, and 115200 is the fastest standard rate, though still slow compared to other modern serial I/O standards.

The RS-232 standard defines several additional signals. The Request to Send (RTS) and Clear to Send (CTS) signals can be used for *hardware handshaking*. They can be operated in either of two modes. In *flow control* mode, the DTE clears RTS to 0 when it is ready to accept data from the DCE. Likewise, the DCE clears CTS to 0 when it is ready to receive data from the DTE. Some datasheets use an overbar to indicate that they are active-low. In the older *simplex* mode, the DTE clears RTS to 0 when it is ready to transmit. The DCE replies by clearing CTS when it is ready to receive the transmission.

Some systems, especially those connected over a telephone line, also use Data Terminal Ready (DTR), Data Carrier Detect (DCD), Data Set Ready (DSR), and Ring Indicator (RI) to indicate when equipment is connected to the line.

The original standard recommended a massive 25-pin DB-25 connector, but PCs streamlined to a male 9-pin DE-9 connector with the pinout shown in Figure e9.13(a). The cable wires normally connect straight across as shown in Figure e9.13(b). However, when directly connecting two DTEs, a *null modem* cable shown in Figure e9.13(c) may be needed to swap RX and TX and complete the handshaking. As a final insult, some connectors are male and some are female. In summary, it can take a large box of cables and a certain amount of guess-work to connect two systems over RS-232, again explaining the shift to USB. Fortunately, embedded systems typically use a simplified 3- or 5-wire setup consisting of GND, TX, RX, and possibly RTS and CTS.

RS-232 represents a 0 electrically with 3 to 15 V and a 1 with −3 to −15 V; this is called *bipolar* signaling. A transceiver converts the digital logic levels of the UART to the positive and negative levels expected by RS-232, and also provides electrostatic discharge protection to protect the serial port from getting zapped when the user plugs in a cable. The

MAX3232E is a popular transceiver compatible with both 3.3 and 5 V digital logic. It contains a charge pump that, in conjunction with external capacitors, generates ±5 V outputs from a single low-voltage power supply. Some serial peripherals intended for embedded systems omit the transceiver and just use 0 V for a 0 and 3.3 or 5 V for a 1; check the datasheet!

The BCM2835 has two UARTs named UART0 and UART1. Either can be configured to communicate on pins 14 and 15, but UART0 is more fully featured and is described here. To use these pins for UART0 rather than GPIO, their GPFSEL must be set to ALT0. As with SPI, the Pi must first configure the port. Unlike SPI, reading and writing can occur independently because either system may transmit without receiving and vice versa. UART0's registers are shown in Figure e9.14.

To configure the UART, first set the baud rate. The UART has an internal 3 MHz clock that must be divided down to produce a clock that is 16x the desired baud rate. Hence, the appropriate divisor, BRD, is

$$\text{BRD} = 3000000/(16 \times \text{baud rate})$$

BRD is represented with a 16-bit integer portion in UART_IBRD and a 6-bit fractional portion in UART_FBRD: BRD = IBRD + FBRD/64. Table e9.6 shows these settings for popular baud rates.[1]

Table e9.6 BRD settings

Target Baud Rate	UART_IBRD	UART_FBRD	Actual Baud Rate	Error (%)
300	625	0	300	0
1200	156	16	1200	0
2400	78	8	2400	0
9600	19	34	9600	0
19200	9	49	19200	0
38400	4	56	38461	0.16
57600	3	16	57692	0.16
115200	1	40	115384	0.16

[1] The baud rates do not all evenly divide 3 MHz, so some divisors produce a frequency error. The UART, by its asynchronous nature, accommodates this error so long as it is small enough.

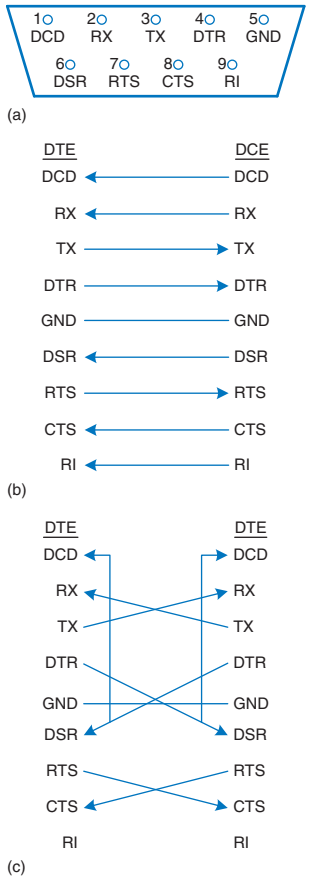


Figure e9.13 DE-9 male cable (a) pinout, (b) standard wiring, and (c) null modem wiring

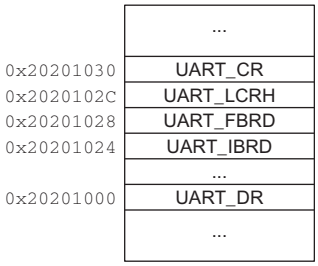


Figure e9.14 UART0 registers

Next, set the number of data, stop, and parity bits using the UART_LCRH line control register. By default, the UART has 1 stop bit and no parity, but strangely only transmits and receives 5-bit words. Hence, the WLEN field (bits 6:5) of UART_LCRH must be set to 3 to handle 8-bit words. Finally, enable the UART by turning on bit 0 (UARTEN) of the UART_CR control register.

Data is transmitted and received using the UART_DR data register and UART_FR framing register. To transmit data, wait until bit 7 (TXFE) of UART_FR is 1 to indicate that the transmitter is not busy, then write a byte to UART_DR. To receive data, wait until bit 4 (RXFE) of UART_FR is 0 to indicate that the receiver has data, then read the byte from UART_DR.

Example e9.5 SERIAL COMMUNICATION WITH A PC

Develop a circuit and a C program for a Raspberry Pi to communicate with a PC over a serial port at 115200 baud with 8 data bits, 1 stop bit, and no parity. The PC should be running a console program such as PuTTY[2] to read and write over the serial port. The program should ask the user to type a string. It should then tell the user what she typed.

Solution: Figure e9.15(a) shows a basic schematic of the serial link illustrating the issues of level conversion and cabling. Because few PCs still have physical serial ports, we use a Plugable USB to RS-232 DB9 Serial Adapter from `plugable.com` shown in Figure e9.16 to provide a serial connection to the PC. The adapter connects to a female DE-9 connector soldered to wires that feed a transceiver, which converts the voltages from the bipolar RS-232 levels to the Pi's 3.3 V level. The Pi and PC are both Data Terminal Equipment, so the TX and RX pins must be cross-connected in the circuit. The RTS/CTS handshaking from the Pi is not used, and the RTS and CTS on the DE9 connector are tied together so that the PC will shake its own hand. Figure e9.15(b) shows an easier approach with an Adafruit 954 USB to TTL serial cable. The cable is directly compatible with 3.3 V levels and has female header pins that plug directly into the Raspberry Pi male headers.

To configure PuTTY to work with the serial link, set *Connection type* to Serial and *Speed* to 115200. Set *Serial line* to the COM port assigned by the operating system to the Serial to USB Adapter. In Windows, this can be found

---

[2] PuTTY is available for free download at `www.putty.org`.

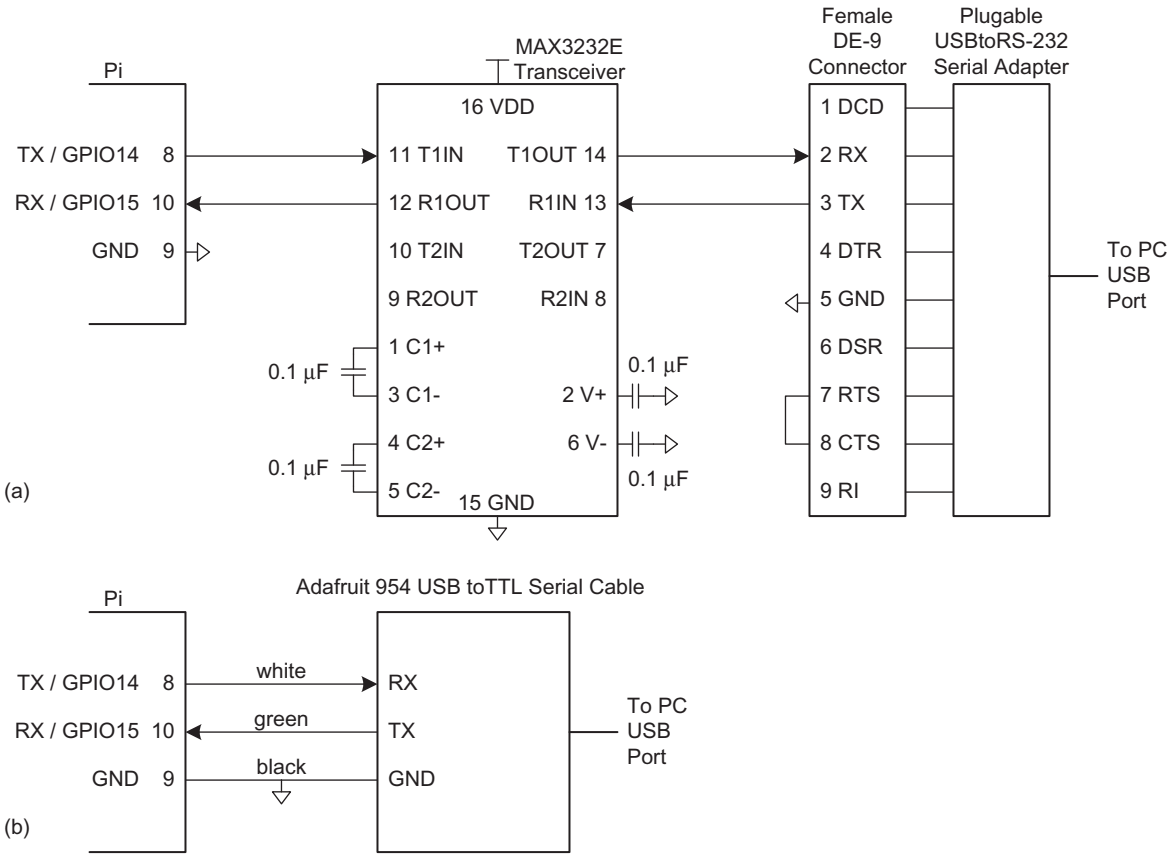


Figure e9.15 **Raspberry Pi to PC serial link (a) Plugable cable, (b) Adafruit cable**

in the Device Manager; for example, it might be COM3. Under the *Connection Serial* tab, set flow control to NONE or RTS/CTS. Under the Terminal tab, set Local Echo to Force On to have characters appear in the terminal as you type them.

The serial port device driver code in EasyPIO.h is listed below. The Enter key in the terminal program corresponds to a carriage return character represented as `'\r'` in C with an ASCII code of 0x0D. To advance to the beginning of the next line when printing, send both the `'\n'` and `'\r'` (new line and carriage return) characters.[3] The `uartInit` function configures the UART as described above. Similarly, `getCharSerial` and `putCharSerial`

Note that the operating system also prints a log-in prompt to the serial port. You may see some interesting interactions between the OS and your program when both use the port.

Figure e9.16 **Plugable USB to RS-232 DB9 serial adapter**
(© 2012 Plugable Technologies; reprinted with permission.)

wait until the UART is ready and then read or write a byte from the data register.

```
void uartInit(int baud) {
    uint fb = 12000000/baud;        // 3 MHz UART clock

    pinMode(14, ALT0);              // TX
    pinMode(15, ALT0);              // RX
    UART_IBRD = fb >> 6;            // 6 Fract, 16 Int bits of BRD
    UART_FBRD = fb & 63;
    UART_LCRHbits.WLEN = 3;         // 8 Data, 1 Stop, no Parity, no FIFO, no Flow
    UART_CRbits.UARTEN = 1;         // Enable uart
}

char getCharSerial(void) {
    while (UART_FRbits.RXFE);       // Wait until data is available
    return UART_DRbits.DATA;        // Return char from serial port
}

void putCharSerial(char c) {
   while (!UART_FRbits.TXFE);       // Wait until ready to transmit
   UART_DRbits.DATA = c;            // Send char to serial port
}
```

The `main` function demonstrates printing to the console and reading from the console using the `putStrSerial` and `getStrSerial` functions.

```
#include "EasyPIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
    int i = 0;
    do {                                                    // Read an entire string until
        str[i] = getCharSerial();                           // Carriage return
    } while ((str[i++] != '\r') && (i < MAX_STR_LEN));      // Look for carriage return
    str[i-1] = 0;                                           // Null-terminate the string
}

void putStrSerial(char *str) {
    int i = 0;
    while (str[i] != 0) {                                  // Iterate over string
        putCharSerial(str[i++]);                           // Send each character
    }
}

            int main(void) {
                char str[MAX_STR_LEN];
```

[3] PuTTY prints correctly even if the `\r` is omitted.

```
    pioInit();
    uartInit(115200);              // Initialize UART with baud rate

    while (1) {
        putStrSerial("Please type something: \r\n");
        getStrSerial(str);
        putStrSerial("You typed: ");
        putStrSerial(str);
        putStrSerial("\r\n");
    }
}
```

Communicating with the serial port from a C program on a PC is a bit of a hassle because serial port driver libraries are not standardized across operating systems. Other programming environments such as Python, Matlab, or LabVIEW make serial communication painless.

9.3.5 Timers

Embedded systems commonly need to measure time. For example, a microwave oven needs a timer to keep track of the time of day and another to measure how long to cook. It might use yet another to generate pulses to the motor spinning the platter, and a fourth to control the power setting by only activating the microwave's energy for a fraction of every second.

The BCM2835 has a system timer with a 64-bit free-running counter that increments every microsecond (i.e. at 1 MHz) and four 32-bit timer compare channels. Figure e9.17 shows the memory map for the system timer. SYS_TIMER_CLO and CHI contain the lower and upper 32 bits of the 64-bit counter value. SYS_TIMER_C0…C3 are 32-bit compare channels. When any of the compare channels match SYS_TIMER_CLO, the corresponding match bit (M0-M3) in the bottom four bits of SYS_TIMER_CS is set. A match bit is cleared by writing a 1 to that bit of SYS_TIMER_CS. This may seem counterintuitive, but it prevents inadvertently clearing other match bits. One can measure a particular number of microseconds by adding that time to CLO and putting it in C1, clearing SYS_TIMER_CS.M1, then waiting until SYS_TIMER_CS.M1 is set.

Unfortunately, Linux is a multitasking operating system that may switch between processes without warning. If your program is waiting for a timer match and then another process begins executing, your program may not resume until long after the match occurs and

The graphics processing unit and operating system may use channels 0, 2, and 3, so user code should check SYSTEM_TIMER_C1.

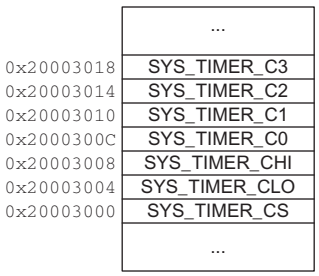


Figure e9.17 System timer registers

you may measure the wrong amount of time. To avoid this, your program can turn off interrupts during critical timing loops so that Linux will not switch processes. Be sure to turn the interrupts back on when you are done. EasyPIO defines `noInterrupts` and `interrupts` functions to disable and enable interrupts, respectively. While interrupts are disabled, the Pi will not switch between processes and cannot even respond to the user pressing Ctrl-C to kill a program. If your program hangs, you'll need to turn off power and reboot your Pi to recover.

Example e9.6 BLINKING LED

Write a program that blinks the status LED on the Raspberry Pi 5 times per second for 4 seconds.

Solution: The `delayMicros` function in EasyPIO creates a delay of a specified number of microseconds using the timer compare channel 1.

```
void delayMicros(int micros) {
    SYS_TIMER_C1 = SYS_TIMER_CLO + micros;  // Set the compare register
    SYS_TIMER_CSbits.M1 = 1;                // Reset match flag to 0
    while (SYS_TIMER_CSbits.M1 == 0);       // Wait until match flag is set
}

void delayMillis(int millis) {
    delayMicros(millis*1000);               // 1000 µs per ms
}
```

GPIO47 drives the activity LED on the Pi B+. The program sets this pin to be an output and disables interrupts. It then turns the LED OFF and ON through a series of digital writes with a 200 ms repetition rate (5 Hz). The program finally reenables interrupts.

```
#include "EasyPIO.h"

void main(void) {
    int i;

    pioInit();

    pinMode(47, OUTPUT);     // Status led as output
    noInterrupts();          // Disable interrupts

    for (i=0; i<20; i++) {
        delayMillis(150);
        digitalWrite(47, 0); // Turn led off
        delayMillis(50);
        digitalWrite(47, 1); // Turn led on
    }
    interrupts();            // Re-enable interrupts
}
```

9.3.6 Analog I/O

The real world is an analog place. Many embedded systems need analog inputs and outputs to interface with the world. They use analog-to-digital converters (ADCs) to quantize analog signals into digital values, and digital-to-analog-converters (DACs) to do the reverse. Figure e9.18 shows symbols for these components. Such converters are characterized by their resolution, dynamic range, sampling rate, and accuracy. For example, an ADC might have $N = 12$-bit resolution over a range V_{ref}^- to V_{ref}^+ of 0–5 V with a sampling rate of $f_s = 44$ kHz and an accuracy of ±3 least significant bits (lsbs). Sampling rates are also listed in samples per second (sps), where 1 sps = 1 Hz. The relationship between the analog input voltage $V_{in}(t)$ and the digital sample $X[n = t / f_s]$ is

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref-}}{V_{ref+} - V_{ref-}}$$

For example, an input voltage of 2.5 V (half of full scale) would correspond to an output of $100000000000_2 = 800_{16}$, with an uncertainty of up to 3 lsbs.

Similarly, a DAC might have $N = 16$-bit resolution over a full-scale output range of $V_{ref} = 2.56$ V. It produces an output of

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref}$$

Many microcontrollers have built-in ADCs of moderate performance. For higher performance (e.g., 16-bit resolution or sampling rates in excess of 1 MHz), it is often necessary to use a separate ADC connected to the microcontroller. Fewer microcontrollers have built-in DACs, so separate chips may also be used. However, microcontrollers often produce analog outputs using a technique called pulse-width modulation (PWM).

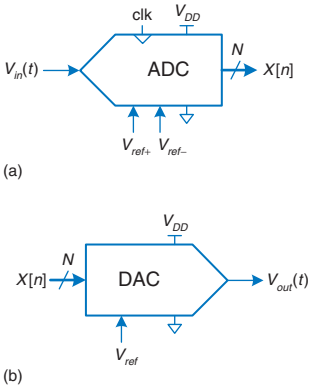


Figure e9.18 ADC and DAC symbols

9.3.6.1 D/A Conversion

The BCM2835 has a specialized DAC for composite video output, but no general-purpose converter. This section describes D/A conversion using external DACs and illustrates interfacing the Raspberry Pi to other chips over parallel and serial ports. The next section achieves the same result using pulse-width modulation.

Some DACs accept the N-bit digital input on a parallel interface with N wires, while others accept it over a serial interface such as SPI. Some DACs require both positive and negative power supply voltages, while others operate off of a single supply. Some support a flexible range of supply voltages, while others demand a specific voltage. The input logic levels should be compatible with the digital source. Some DACs produce a voltage output proportional to the digital input, while others produce a

current output; an operational amplifier may be needed to convert this current to a voltage in the desired range.

In this section, we use the Analog Devices AD558 8-bit parallel DAC and the Linear Technology LTC1257 12-bit serial DAC. Both produce voltage outputs, run off a single 5–15 V power supply, use $V_{IH} = 2.4$ V such that they are compatible with 3.3 V I/O, come in DIP packages that make them easy to breadboard, and are easy to use. The AD558 produces an output on a scale of 0-2.56 V, consumes 75 mW, comes in a 16-pin package, and has a 1 µs settling time permitting an output rate of 1 Msamples/sec. The datasheet is at analog.com. The LTC1257 produces an output on a scale of 0–2.048 V, consumes less than 2 mW, comes in an 8-pin package, and has a 6 µs settling time. Its SPI operates at a maximum of 1.4 MHz. The datasheet is at linear.com.

Example e9.7 ANALOG OUTPUT WITH EXTERNAL DACS

Sketch a circuit and write the software for a simple signal generator producing sine and triangle waves using a Raspberry Pi, an AD558, and an LTC1257.

Solution: The circuit is shown in Figure e9.19. The AD558 connects to the Pi via GPIO14, 15, 17, 18, 22, 23, 24, and 25. It connects *Vout Sense* and *Vout Select* to *Vout* to set the 2.56 V full-scale output range. The LTC1257 connects

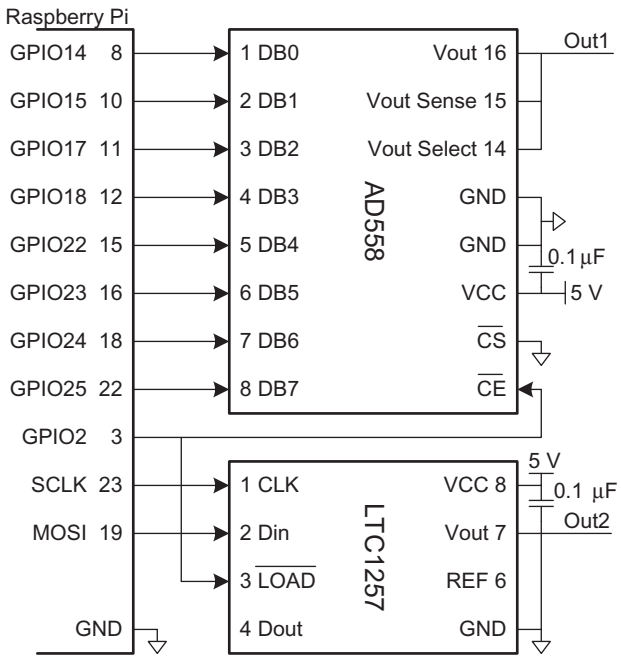


Figure e9.19 DAC parallel and serial interfaces to a Raspberry Pi

to the Pi via SPI0. Both ADCs use a 5 V power supply and have a 0.1 µF decoupling capacitor to reduce power supply noise. The active-low chip enable and load signals on the DACs indicate when to convert the next digital input. They are driven high while a new input is being loaded.

The program is listed below. `pinsMode` and `digitalWrites` are similar to `pinMode` and `digitalWrite` but operate on an array of pins. The program sets the 8 parallel port pins to be outputs and also configures GPIO2 as an output to drive the chip enable and load signals. It initializes the SPI to 1.4 MHz. `initWaveTables` precomputes an array of sample values for the sine and triangle waves. The sine wave is set to a 12-bit scale and the triangle to an 8-bit scale. There are 64 points per period of each wave; changing this value trades precision for frequency. `genWaves` cycles through the samples. It disables interrupts to avoid switching processes and garbling the waves. For each sample, it disables the CE and LOAD signals to the DACs, sends the new sample over the parallel and serial ports, reenables the DACs, and then waits until the timer indicates that it is time for the next sample. `spiSendReceive16` transmits two bytes, but the LTC1257 only cares about the last 12 bits sent. The maximum frequency of somewhat more than 1000 Hz (64 Ksamples/sec) is set by the time to send each point in the `genWaves` function, of which the SPI transmission is a major component.

```
#include "EasyPIO.h"
#include math.h> // required to use the sine function

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];
int parallelPins[8] = {14,15,17,18,22,23,24,25};

void initWaveTables(void) {
    int i;
    for (i=0; i<NUMPTS; i++) {
        sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1);  // 12-bit scale
        if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS;    //  8-bit scale
        else triangle[i] = 510-i*511/NUMPTS;
    }
}

void genWaves(int freq) {
    int i, j;
    int microPeriod = 1000000/(NUMPTS*freq);

    noInterrupts();     // disable interrupts to get accurate timing
    for (i=0; i<2000; i++){
        for (j=0; j<NUMPTS; j++) {
            SYS_TIMER_C1 = SYS_TIMER_CLO + microPeriod;  // Set time between samples
            SYS_TIMER_CSbits.M1 = 1;                     // Clear timer match
            digitalWrite(2,1);                           // No load while changing inputs
            spiSendReceive16(sine[j]);
            digitalWrites(parallelPins, 8, triangle[j]);
            digitalWrite(2,0);                           // Load new points into DACs
            while (!SYS_TIMER_CSbits.M1);                // Wait until timer matches
        }
```

```
    }
    interrupts();
}

void main(void) {
    pioInit();

    pinsMode(parallelPins, 8, OUTPUT);  // Set pins connected to the AD558 as outputs
    pinMode(2, OUTPUT);                 // Make pin 2 an output to control LOAD and CE
    spiInit(1400000, 0);                // 1.4MHz SPI clock, default settings
    initWaveTables();
    genWaves(1000);
}
```

9.3.6.2 Pulse-Width Modulation

Another way for a digital system to generate an analog output is with *pulse-width modulation* (PWM), in which a periodic output is pulsed high for part of the period and low for the remainder. The duty cycle is the fraction of the period for which the pulse is high, as shown in Figure e9.20. The average value of the output is proportional to the duty cycle. For example, if the output swings between 0 and 3.3 V and has a duty cycle of 25%, the average value will be $0.25 \times 3.3 = 0.825$ V. Low-pass filtering a PWM signal eliminates the oscillation and leaves a signal with the desired average value. Thus, PWM is an effective way to produce an analog output if the pulse rate is much higher than the analog output frequencies of interest.

The BCM2835 has a PWM controller capable of producing two simultaneous outputs. PWM0 is available at GPIO18 as pin function ALT5, while both PWM outputs are available on the stereo audio jack. Figure e9.21 shows the memory map for the PWM unit and for the clock manager that it depends upon.

The PWM_CTL register is used to turn on pulse width modulation. Bit 0 (PWEN1) must be set to enable the output. Bit 7 (MSEN1: mark-space enable) should also be set to produce pulse width modulation of the form of Figure e9.20 in which the output is HIGH for part of the period and LOW for the remainder.

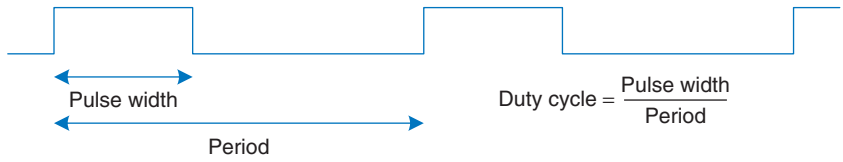


Figure e9.20 Pulse-width modulated (PWM) signal

The PWM signals are derived from a PWM clock generated by the BCM2835 clock manager. The PWM_RNG1 and PWM_DAT1 registers control the period and duty cycle, respectively, by specifying the number of PWM clock ticks for the overall waveform and for the HIGH portion. For example, if the clock manager produces a 25 MHz clock and PWM_RNG1 = 1000 and PWM_DAT1 = 300, the PWM output will operate at (25 MHz / 1000) = 25 kHz and the duty cycle will be 300 / 1000 = 30%.

The clock manager is configured using the CM_PWMCTL and the frequency is set using the CM_PWMDIV register. Table e9.7 summarizes the bit fields of the CM_PWMCTL register. The maximum frequency of the PWM clock is 25 MHz. It can be obtained from the 500 MHz PLLD clock on the Pi as follows:

- CM_PWMCTL: Write 0x5A to PASSWD and 1 to KILL to stop the clock generator
- CM_PWMCLT: Wait for BUSY to clear to indicate the clock is stopped
- CM_PWMCTL: Write 0x5A to PASSWD, 1 to MASH, and 6 to SRC to select PLLD with no audio noise shaping
- CM_PWMDIV: Write 0x5A to PASSWD and 20 to bits 23:12 to divide PLLD by 20 from 500 MHz down to 25 MHz
- CM_PWMCTL: Write 0x5A to PASSWD and 1 to ENAB to restart the clock generator
- CM_PWMCTL: Wait for BUSY to set to indicate the clock is running

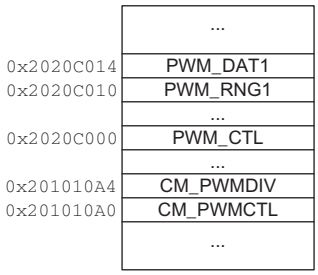


Figure e9.21 PWM and clock manager registers

The CM_PWM registers are not documented in the BCM2835 datasheet. You may find information on them by searching the Internet for "BCM2835 Audio & PWM Clocks" by G.J. van Loo.

Table e9.7 CM_PWMCTL register fields

Bit	Name	Description
31:24	PASSWD	Must be set to 5A when writing
10:9	MASH	Audio noise shaping
7	BUSY	Clock generator running
5	KILL	Write a 1 to stop the clock generator
4	ENAB	Write a 1 to start the clock generator
3:0	SRC	Clock source

Example e9.8 ANALOG OUTPUT WITH PWM

Write an `analogWrite(val)` function to generate an analog output voltage using PWM and an external RC filter. The function should accept an input between 0 (for 0 V output) and 255 (for full 3.3 V output).

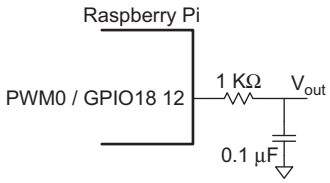


Figure e9.22 Analog output using PWM and low-pass filter

Solution: Use PWM0 to produce a 78.125 kHz signal on GPIO18. The low pass filter in Figure e9.22 has a corner frequency of

$$f_c = \frac{1}{2\pi RC} = 1.6 \text{ kHz}$$

to eliminate the high-speed oscillations and pass the average value.

The PWM functions in EasyPIO are given below. `pwmInit` initializes the PWM module on GPIO18 as described above. `setPWM` sets the frequency and duty cycle of the PWM output. Duty should be between 0 (always OFF) and 1 (always ON). The `analogWrite` function sets the duty cycle based on a full scale of 255.

```
    // Default PLLD value is 500 [MHz]
    #define PLL_FREQUENCY 500000000
    // Max pwm clk is 25 [MHz]
    #define CM_FREQUENCY 25000000
    #define PLL_CLOCK_DIVISOR (PLL_FREQUENCY / CM_FREQUENCY)

void pwmInit() {
    pinMode(18, ALT5);

    // Configure the clock manager to generate a 25 MHz PWM clock.
    // Documentation on the clock manager is missing in the datasheet
    // but found in "BCM2835 Audio and PWM Clocks" by G.J. van Loo 6 Feb 2013.
    // Maximum operating frequency of PWM clock is 25 MHz.
    // Writes to the clock manager registers require simultaneous writing
    // a "password" of 5A to the top bits to reduce the risk of accidental writes.

    CM_PWMCTL = 0; // Turn off PWM before changing
    CM_PWMCTL = PWM_CLK_PASSWORD|0x20; // Turn off clock generator
    while (CM_PWMCTLbits.BUSY); // Wait for generator to stop
    CM_PWMCTL = PWM_CLK_PASSWORD|0x206; // Src = unfiltered 500 MHz CLKD
    CM_PWMDIV = PWM_CLK_PASSWORD|(PLL_CLOCK_DIVISOR << 12); // 25 MHz
    CM_PWMCTL = CM_PWMCTL|PWM_CLK_PASSWORD|0x10;  // Enable PWM clock
    while (!CM_PWMCTLbits.BUSY);  // Wait for generator to start
    PWM_CTLbits.MSEN1 = 1;  // Channel 1 in mark/space mode
    PWM_CTLbits.PWEN1 = 1;  // Enable PWM
}

void setPWM(float freq, float duty) {
    PWM_RNG1 = (int)(CM_FREQUENCY / freq);
    PWM_DAT1 = (int)(duty * (CM_FREQUENCY / freq));
}
```

```
void analogWrite(int val) {
  setPWM(78125, val/255.0);
}
```

The main function tests the PWM by setting the output to half scale (1.65 V).

```
#include "EasyPIO.h"

void main(void) {
  pioInit();
  pwmInit();
  analogWrite(128);
}
```

9.3.6.3 A/D Conversion

The BCM2835 has no built-in ADC, so this section describes A/D conversion using an external converter similar to the external DAC.

Example e9.9 ANALOG INPUT WITH AN EXTERNAL ADC

Interface a 10-bit MCP3002 A/D converter to a Raspberry Pi using SPI and print the input value. Set a full scale voltage of 3.3 V. Search for the datasheet on the Web for full details of operation.

Solution: Figure e9.23 shows a schematic of the connection. The MCP3002 uses VDD as its full scale reference. It accepts a 3.3–5.5 V supply and we

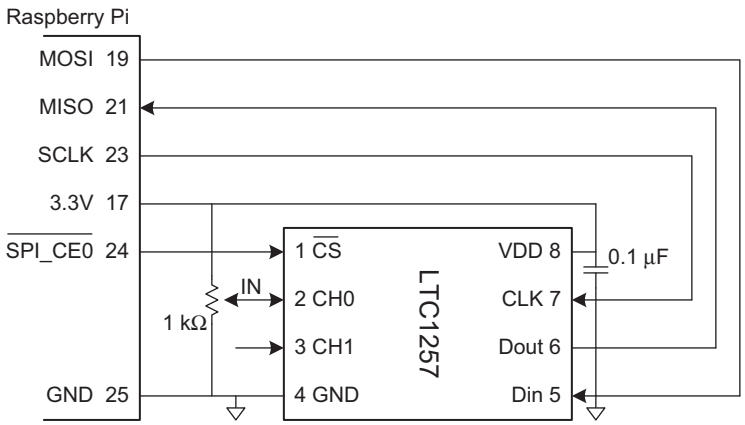


Figure e9.23 **Analog input using external ADC**

choose 3.3 V. The ADC has two input channels, and we connect channel 0 to a potentiometer that we can rotate to adjust the input voltage between 0 and 3.3 V.

The Pi code initializes the SPI and repeatedly reads and prints samples. According to the datasheet, the Raspberry Pi must send the 16-bit quantity 0x6000 over SPI to read CH0 and will receive the 10-bit result back in the bottom 10 bits of the 16-bit result. The converter also requires a chip select signal, conveniently provided by the SPI chip enable.

```
#include "EasyPIO.h"

void main(void) {
    int sample;

    pioInit();
    spiInit(200000, 0); // 200 kHz SPI clock, default settings

    while (1){
        sample = spiSendReceive16(0x6000);
        printf("Read %d\n", sample);
    }
}
```

9.3.7 Interrupts

So far, we have relied on *polling*, in which the program continually checks until an event occurs such as data arriving on a UART or a timer reaching its compare value. This can be a waste of the processor's power and makes it difficult to write programs that do interesting work while simultaneously waiting for events to occur.

Most microcontrollers support *interrupts*. When an event occurs, the microcontroller can stop regular program execution and jump to an interrupt handler that responds to the interrupt, then return seamlessly to where it left off.

The Raspberry Pi normally runs Linux, which intercepts interrupts before they get to the program. Therefore, it is presently not straightforward to write interrupt-based programs and this text does not provide examples on the Pi.

9.4 OTHER MICROCONTROLLER PERIPHERALS

Microcontrollers frequently interface with other external peripherals. This section describes a variety of common examples, including character-mode liquid crystal displays (LCDs), VGA monitors, Bluetooth wireless links, and motor control. Standard communication interfaces including USB and Ethernet are described in Sections 9.6.1 and 9.6.4.

9.4.1 Character LCDs

A character LCD is a small liquid crystal display capable of showing one or a few lines of text. They are commonly used in the front panels of appliances such as cash registers, laser printers, and fax machines that need to display a limited amount of information. They are easy to interface with a microcontroller over parallel, RS-232, or SPI interfaces. Crystalfontz America sells a wide variety of character LCDs ranging from 8 columns × 1 row to 40 columns × 4 rows with choices of color, backlight, 3.3 or 5 V operation, and daylight visibility. Their LCDs can cost $20 or more in small quantities, but prices come down to under $5 in high volume.

This section gives an example of interfacing a Raspberry Pi to a character LCD over an 8-bit parallel interface. The interface is compatible with the industry-standard HD44780 LCD controller originally developed by Hitachi. Figure e9.24 shows a Crystalfontz CFAH2002A-TMI-JT 20 × 2 parallel LCD.

Figure e9.25 shows the LCD connected to a Pi over an 8-bit parallel interface. The logic operates at 5 V but is compatible with 3.3 V inputs from the Pi. The LCD contrast is set by a second voltage produced with a potentiometer; it is usually most readable at a setting of 4.2–4.8 V. The LCD receives three control signals: RS (1 for characters, 0 for instructions), R/$\overline{W}$ (1 to read from the display, 0 to write), and E (pulsed high for at least 250 ns to enable the LCD when the next byte is ready). When the instruction is read, bit 7 returns the busy flag, indicating 1 when busy and 0 when the LCD is ready to accept another instruction.

To initialize the LCD, the Pi must write a sequence of instructions to the LCD as given in Table e9.8. The instructions are written by holding RS = 0 and R/$\overline{W}$ = 0, putting the value on the eight data lines, and pulsing E. After each instruction, it must wait for at least a specified amount of time (or sometimes until the busy flag is clear).

Figure e9.24 Crystalfontz CFAH2002A-TMI 20×2 character LCD (© 2012 Crystalfontz America; reprinted with permission.)

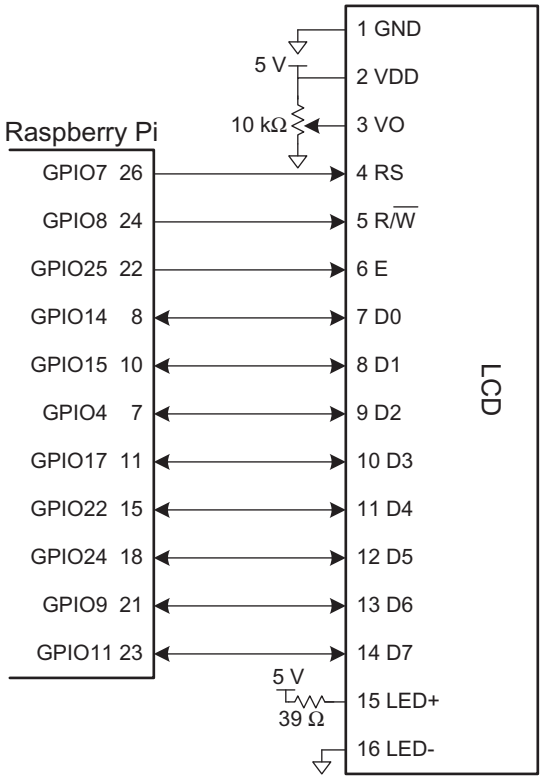


Figure e9.25 Parallel LCD interface

Table e9.8 LCD initialization sequence

Write	Purpose	Wait (μs)
(apply V_{DD})	Allow device to turn on	15000
0x30	Set 8-bit mode	4100
0x30	Set 8-bit mode again	100
0x30	Set 8-bit mode yet again	Until busy flag is clear
0x3C	Set 2 lines and 5 × 8 dot font	Until busy flag is clear
0x08	Turn display OFF	Until busy flag is clear
0x01	Clear display	1530
0x06	Set entry mode to increment cursor after each character	Until busy flag is clear
0x0C	Turn display ON with no cursor	

Then, to write text to the LCD, the Pi can send a sequence of ASCII characters. After each character, it must wait for the busy bit to clear. It may also send the instruction 0x01 to clear the display or 0x02 to return to the home position in the upper left.

Example e9.10 LCD CONTROL

Write a program to print "I love LCDs" to a character display.

Solution: The following program writes "I love LCDs" to the display by initializing the display and then sending the characters.

```
#include "EasyPIO.h"

int LCD_IO_Pins[] = {14, 15, 4, 17, 22, 24, 9, 11};

typedef enum {INSTR, DATA} mode;
#define RS 7
#define RW 8
#define E  25

char lcdRead(mode md) {
    char c;
    pinsMode(LCD_IO_Pins, 8, INPUT);
    digitalWrite(RS,(md == DATA));      // Set instr/data mode
    digitalWrite(RW, 1);                // Read mode
    digitalWrite(E, 1);                 // Pulse enable
    delayMicros(10);                    // Wait for LCD response
    c = digitalReads(LCD_IO_Pins, 8);   // Read a byte from parallel port
    digitalWrite(E, 0);                 // Turn off enable
    delayMicros(10);
    return c;
}

void lcdBusyWait(void) {
    char state;
    do {
       state = lcdRead(INSTR);
    } while (state & 0x80);
}

void lcdWrite(char val, mode md) {
    pinsMode(LCD_IO_Pins, 8, OUTPUT);
    digitalWrite(RS, (md == DATA));     // Set instr/data mode. OUTPUT=1, INPUT=0
    digitalWrite(RW, 0);                // Set RW pin to write  (aka: 0)
    digitalWrites(LCD_IO_Pins, 8, val); // Write the char to the parallel port
    digitalWrite(E, 1); delayMicros(10); // Pulse E
    digitalWrite(E, 0); delayMicros(10);
}

void lcdClear(void) {
    lcdWrite(0x01, INSTR); delayMicros(1530);
}
```

```
void lcdPrintString(char* str) {
    while (*str != 0) {
        lcdWrite(*str, DATA); lcdBusyWait();
        str++;
    }
}

void lcdInit(void) {
    pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
    // send initialization routine:
    delayMicros(15000);
    lcdWrite(0x30, INSTR); delayMicros(4100);
    lcdWrite(0x30, INSTR); delayMicros(100);
    lcdWrite(0x30, INSTR); lcdBusyWait();
    lcdWrite(0x3C, INSTR); lcdBusyWait();
    lcdWrite(0x08, INSTR); lcdBusyWait();
    lcdClear();
    lcdWrite(0x06, INSTR); lcdBusyWait();
    lcdWrite(0x0C, INSTR); lcdBusyWait();
}

void main(void) {
    pioInit();
    lcdInit();
    lcdPrintString("I love LCDs!");
}
```

9.4.2 VGA Monitor

A more flexible display option is to drive a computer monitor. The Raspberry Pi has built-in support for HDMI and Composite video output. This section explains the low-level details of driving a VGA monitor directly from an FPGA.

The *Video Graphics Array* (VGA) monitor standard was introduced in 1987 for the IBM PS/2 computers, with a 640×480 pixel resolution on a *cathode ray tube* (CRT) and a 15-pin connector conveying color information with analog voltages. Modern LCD monitors have higher resolution but remain backward compatible with the VGA standard.

In a cathode ray tube, an electron gun scans across the screen from left to right exciting fluorescent material to display an image. Color CRTs use three different phosphors for red, green, and blue, and three electron beams. The strength of each beam determines the intensity of each color in the pixel. At the end of each scanline, the gun must turn off for a *horizontal blanking interval* to return to the beginning of the next line. After all of the scanlines are complete, the gun must turn off again for a *vertical blanking interval* to return to the upper left corner. The process repeats about 60–75 times per second to refresh the fluorescence and give the visual illusion of a steady image. A liquid crystal display doesn't require the same electron scan gun, but uses the same VGA interface timing for compatibility.

In a 640×480 pixel VGA monitor refreshed at 59.94 Hz, the pixel clock operates at 25.175 MHz, so each pixel is 39.72 ns wide. The full screen can be viewed as 525 horizontal scanlines of 800 pixels each, but only 480 of the scanlines and 640 pixels per scan line actually convey the image, while the remainder are black. A scanline begins with a *back porch*, the blank section on the left edge of the screen. It then contains 640 pixels, followed by a blank *front porch* at the right edge of the screen and a horizontal sync (hsync) pulse to rapidly move the gun back to the left edge.

Figure e9.26(a) shows the timing of each of these portions of the scanline, beginning with the active pixels. The entire scan line is 31.778 μs long. In the vertical direction, the screen starts with a back porch at the top, followed by 480 active scan lines, followed by a front porch at the bottom and a vertical sync (vsync) pulse to return to the top to start the next frame. A new frame is drawn 60 times per second.

Figure e9.26(b) shows the vertical timing; note that the time units are now scan lines rather than pixel clocks. Higher resolutions use a faster pixel clock, up to 388 MHz at 2048×1536 at 85 Hz. For example, 1024×768 at 60 Hz can be achieved with a 65 MHz pixel clock.

The horizontal timing involves a front porch of 16 clocks, hsync pulse of 96 clocks, and back porch of 48 clocks. The vertical timing involves a front porch of 11 scan lines, vsync pulse of 2 lines, and back porch of 32 lines.

Figure e9.27 shows the pinout for a female connector coming from a video source. Pixel information is conveyed with three analog voltages for red, green, and blue. Each voltage ranges from 0–0.7 V, with more positive indicating brighter. The voltages should be 0 during the front and

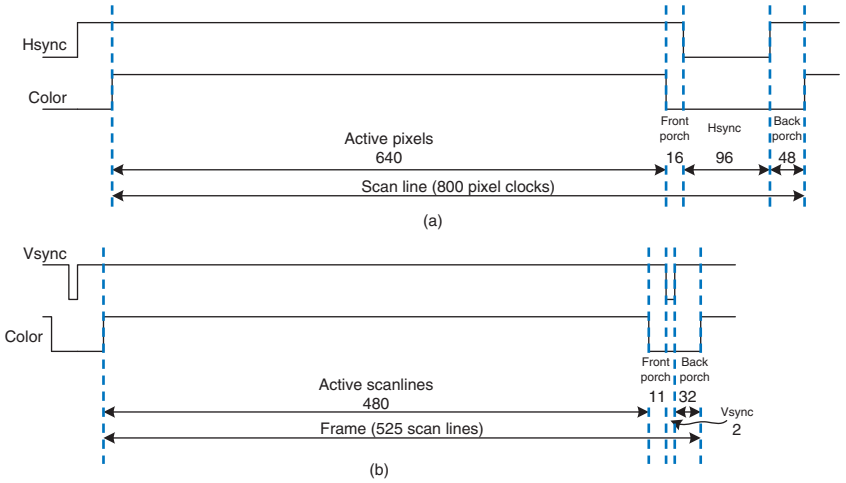


Figure e9.26 VGA timing: (a) horizontal, (b) vertical

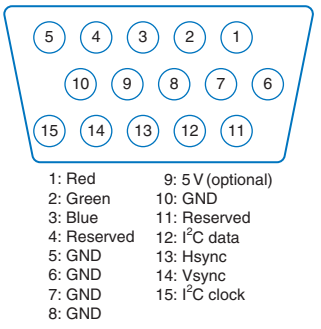


Figure e9.27 VGA connector pinout

back porches. The video signal must be generated in real time at high speed, which is difficult on a microcontroller but easy on an FPGA. A simple black and white display could be produced by driving all three color pins with either 0 or 0.7 V using a voltage divider connected to a digital output pin. A color monitor, on the other hand, uses a *video DAC* with three separate D/A converters to independently drive the three color pins. Figure e9.28 shows an FPGA driving a VGA monitor through an ADV7125 triple 8-bit video DAC. The DAC receives 8 bits of R, G, and B from the FPGA. It also receives a SYNC_b signal that is driven active low whenever HSYNC or VSYNC are asserted. The video DAC produces three output currents to drive the red, green, and blue analog lines, which are normally 75 Ω transmission lines parallel terminated at both the video DAC and the monitor. The R_{SET} resistor sets the scale of the output current to achieve the full range of color. The clock rate depends on the resolution and refresh rate; it may be as high as 330 MHz with a fast-grade ADV7125JSTZ330 model DAC.

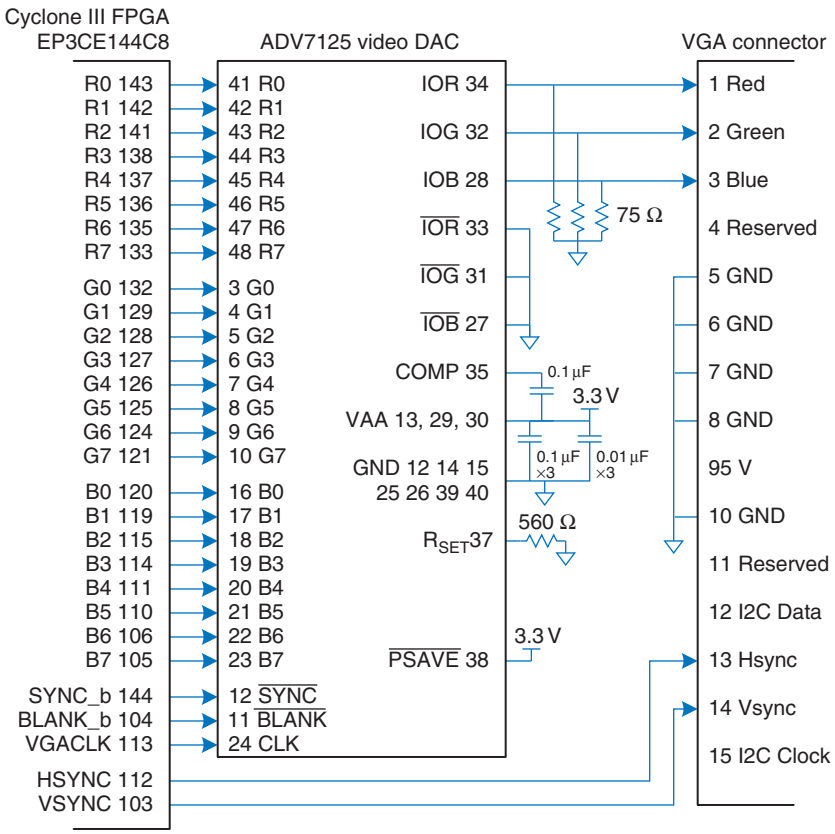


Figure e9.28 **FPGA driving VGA cable through video DAC**

Example e9.11 VGA MONITOR DISPLAY

Write HDL code to display text and a green box on a VGA monitor using the circuitry from Figure e9.28.

Solution: The code assumes a system clock frequency of 40 MHz and uses a phase-locked loop (PLL) on the FPGA to generate the 25.175 MHz VGA clock. PLL configuration varies among FPGAs; for the Cyclone III, the frequencies are specified with Altera's megafunction wizard. Alternatively, the VGA clock could be provided directly from a signal generator.

The VGA controller counts through the columns and rows of the screen, generating the hsync and vsync signals at the appropriate times. It also produces a blank_b signal that is asserted low to draw black when the coordinates are outside the 640×480 active region.

The video generator produces red, green, and blue color values based on the current (x, y) pixel location. (0, 0) represents the upper left corner. The generator draws a set of characters on the screen, along with a green rectangle. The character generator draws an 8×8-pixel character, giving a screen size of 80×60 characters. It looks up the character from a ROM, where it is encoded in binary as 6 columns by 8 rows. The other two columns are blank. The bit order is reversed by the SystemVerilog code because the leftmost column in the ROM file is the most significant bit, while it should be drawn in the least significant x-position.

Figure e9.29 shows a photograph of the VGA monitor while running this program. The rows of letters alternate red and blue. A green box overlays part of the image.

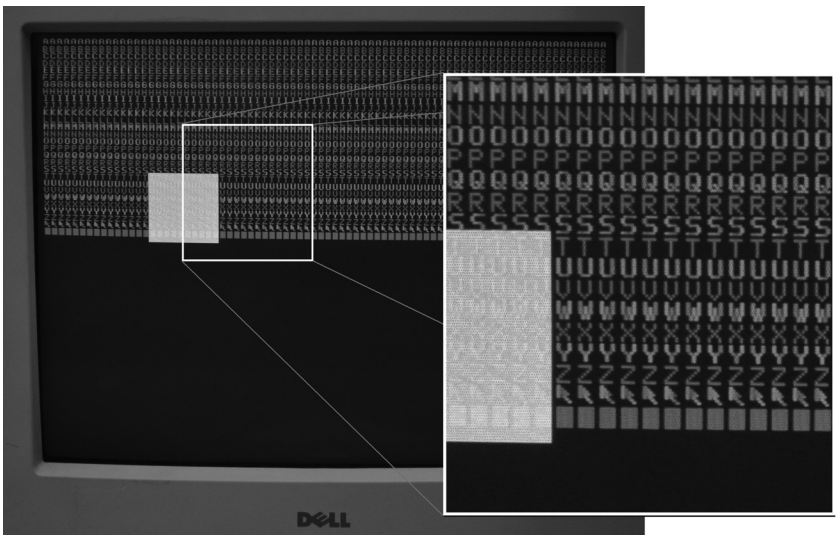


Figure e9.29 VGA output

vga.sv

```
module vga(input  logic       clk,
           output logic       vgaclk,          // 25.175 MHz VGA clock
           output logic       hsync, vsync,
           output logic       sync_b, blank_b, // To monitor & DAC
           output logic [7:0] r, g, b);        // To video DAC

  logic [9:0] x, y;

  // Use a PLL to create the 25.175 MHz VGA pixel clock
  // 25.175 MHz clk period = 39.772 ns
  // Screen is 800 clocks wide by 525 tall, but only 640 x 480 used
  // HSync = 1/(39.772 ns *800) = 31.470 kHz
  // Vsync = 31.474 kHz / 525 = 59.94 Hz (~60 Hz refresh rate)
  pll vgapll(.inclk0(clk), .c0(vgaclk));

  // Generate monitor timing signals
  vgaController vgaCont(vgaclk, hsync, vsync, sync_b, blank_b, x, y);

  // User-defined module to determine pixel color
  videoGen videoGen(x, y, r, g, b);
endmodule

module vgaController #(parameter HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE + HFP + HSYN + HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
                                 VFP     = 10'd11,
                                 VSYN    = 10'd2,
                                 VMAX    = VACTIVE + VFP + VSYN + VBP)
                       (input  logic       vgaclk,
                        output logic       hsync, vsync, sync_b, blank_b,
                        output logic [9:0] x, y);
  // counters for horizontal and vertical positions
  always @(posedge vgaclk) begin
    x++;
    if (x == HMAX) begin
      x = 0;
      y++;
      if (y == VMAX) y = 0;
    end
  end

  // Compute sync signals (active low)
  assign hsync = ~(hcnt >= HACTIVE + HFP & hcnt < HACTIVE + HFP + HSYN);
  assign vsync = ~(vcnt >= VACTIVE + VFP & vcnt < VACTIVE + VFP + VSYN);
  assign sync_b = hsync & vsync;

  // Force outputs to black when outside the legal display area
  assign blank_b = (hcnt < HACTIVE) & (vcnt < VACTIVE);
endmodule

module videoGen(inputlogic [9:0] x, y, output logic [7:0] r, g, b);
```

```
  logic    pixel, inrect;

  // Given y position, choose a character to display
  // then look up the pixel value from the character ROM
  // and display it in red or blue.Also draw a green rectangle.
  chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
  rectgen  rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
  assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00,{8{pixel}}};
  assign g =      inrect ? 8'hFF : 8'h00;
endmodule

module chargenrom(input  logic [7:0] ch,
                  input  logic [2:0] xoff, yoff,
                  output logic       pixel);

  logic [5:0] charrom[2047:0]; // character generator ROM
  logic [7:0] line;            // a line read from the ROM

  // Initialize ROM with characters from text file
  initial
    $readmemb("charrom.txt", charrom);

  // Index into ROM to find line of character
  assign line = charrom[yoff+{ch-65, 3'b000}]; // Subtract 65 because A
                                               // is entry 0

  // Reverse order of bits
  assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input  logic [9:0] x, y, left, top, right, bot,
               output logic       inrect);

  assign inrect = (x >= left & x < right & y >= top & y < bot);
endmodule
```

charrom.txt

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
```

```
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

Bluetooth is named for King Harald Bluetooth of Denmark, a 10th century monarch who unified the warring Danish tribes. This wireless standard is only partially successful at unifying a host of competing wireless protocols!

Table e9.9 Bluetooth classes

Class	Transmitter Power (mW)	Range (m)
1	100	100
2	2.5	10
3	1	5

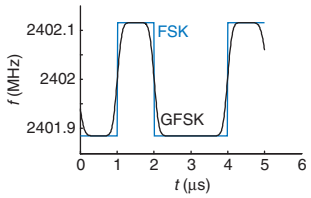


Figure e9.30 FSK and GFSK waveforms

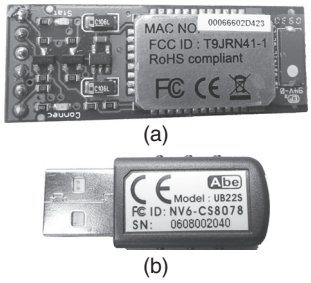


Figure e9.31 BlueSMiRF module and USB dongle

9.4.3 Bluetooth Wireless Communication

There are many standards now available for wireless communication, including Wi-Fi, ZigBee, and Bluetooth. The standards are elaborate and require sophisticated integrated circuits, but a growing assortment of modules abstract away the complexity and give the user a simple interface for wireless communication. One of these modules is the BlueSMiRF, which is an easy-to-use Bluetooth wireless interface that can be used instead of a serial cable.

Bluetooth is a wireless standard initially developed by Ericsson in 1994 for low-power, moderate speed communication over distances of 5–100 meters, depending on the transmitter power level. It is commonly used to connect an earpiece to a cellphone or a keyboard to a computer. Unlike infrared communication links, it does not require a direct line of sight between devices.

Bluetooth operates in the 2.4 GHz unlicensed industrial-scientific-medical (ISM) band. It defines 79 radio channels spaced at 1 MHz intervals starting at 2402 MHz. It hops between these channels in a pseudo-random pattern to avoid consistent interference with other devices, such as wireless routers operating in the same band. As given in Table e9.9, Bluetooth transmitters are classified at one of three power levels, which dictate the range and power consumption. In the basic rate mode, it operates at 1 Mbit/sec using Gaussian frequency shift keying (FSK). In ordinary FSK, each bit is conveyed by transmitting a frequency of $f_c \pm f_d$, where f_c is the center frequency of the channel and f_d is an offset of at least 115 kHz. The abrupt transition in frequencies between bits consumes extra bandwidth. In Gaussian FSK, the change in frequency is smoothed to make better use of the spectrum. Figure e9.30 shows the frequencies being transmitted for a sequence of 0's and 1's on a 2402 MHz channel using FSK and GFSK.

A BlueSMiRF Silver module, shown in Figure e9.31(a), contains a Class 2 Bluetooth radio, modem, and interface circuitry on a small card with a serial interface. It communicates with another Bluetooth device

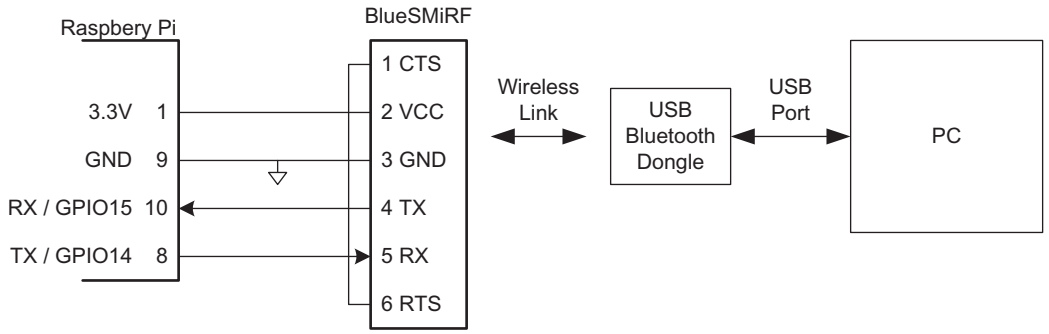


Figure e9.32 **BlueSMiRF Raspberry Pi to PC link**

such as a Bluetooth USB dongle connected to a PC. Thus, it can provide a wireless serial link between a Pi and a PC similar to the link from Figure e9.15 but without the cable. The wireless link is compatible with the same software as is the wired link.

Figure e9.32 shows a schematic for such a link. The TX pin of the BlueSMiRF connects to the RX pin of the Pi, and vice versa. The RTS and CTS pins are connected so that the BlueSMiRF shakes its own hand.

The BlueSMiRF defaults to 115.2 k baud with 8 data bits, 1 stop bit, and no parity or flow control. It operates at 3.3 V digital logic levels, so no RS-232 transceiver is necessary to connect with another 3.3 V device.

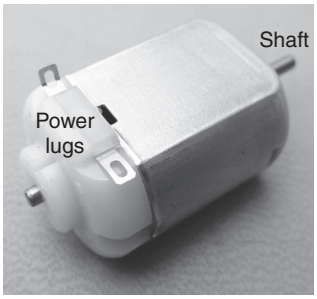
To use the interface, plug a USB Bluetooth dongle into a PC. Power up the Pi and BlueSMiRF. The red STAT light will flash on the BlueSMiRF indicating that it is waiting to make a connection. Open the Bluetooth icon in the PC system tray and use the Add Bluetooth Device Wizard to pair the dongle with the BlueSMiRF. The default passkey for the BlueSMiRF is 1234. Take note of which COM port is assigned to the dongle. Then communication can proceed just as it would over a serial cable. Note that the dongle typically operates at 9600 baud and that PuTTY must be configured accordingly.

9.4.4 Motor Control

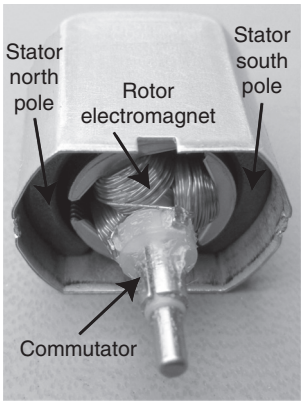
Another major application of microcontrollers is to drive actuators such as motors. This section describes three types of motors: DC motors, servo motors, and stepper motors. *DC motors* require a high drive current, so a powerful driver such as an *H-bridge* must be connected between the microcontroller and the motor. They also require a *shaft encoder* if the

user wants to know the current position of the motor. *Servo motors* accept a pulse-width modulated signal to specify their position over a limited range of angles. They are very easy to interface, but are not as powerful and are not suited to continuous rotation. *Stepper motors* accept a sequence of pulses, each of which rotates the motor by a fixed angle called a step. They are more expensive and still need an H-bridge to drive the high current, but the position can be precisely controlled.

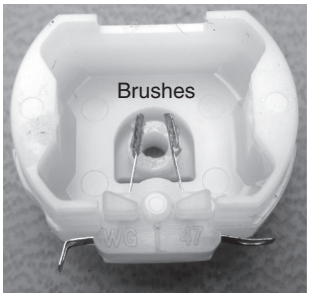
Motors can draw a substantial amount of current and may introduce glitches on the power supply that disturb digital logic. One way to reduce this problem is to use a different power supply or battery for the motor than for the digital logic.



(a)



(b)



(c)

Figure e9.33 **DC motor**

9.4.4.1 DC Motors

Figure e9.33 shows the structure of a brushed DC motor. The motor is a two terminal device. It contains permanent stationary magnets called the *stator* and a rotating electromagnet called the *rotor* or *armature* connected to the shaft. The front end of the rotor connects to a split metal ring called a *commutator*. Metal brushes attached to the power lugs (input terminals) rub against the commutator, providing current to the rotor's electromagnet. This induces a magnetic field in the rotor that causes the rotor to spin to become aligned with the stator field. Once the rotor has spun part way around and approaches alignment with the stator, the brushes touch the opposite sides of the commutator, reversing the current flow and magnetic field and causing it to continue spinning indefinitely.

DC motors tend to spin at thousands of rotations per minute (RPM) at very low torque. Most systems add a gear train to reduce the speed to a more reasonable level and increase the torque. Look for a gear train designed to mate with your motor. Pittman manufactures a wide range of high quality DC motors and accessories, while inexpensive toy motors are popular among hobbyists.

A DC motor requires substantial current and voltage to deliver significant power to a load. The current should be reversible so the motor can spin in both directions. Most microcontrollers cannot produce enough current to drive a DC motor directly. Instead, they use an H-bridge, which conceptually contains four electrically controlled switches, as shown in Figure e9.34(a). If switches A and D are closed, current flows from left to right through the motor and it spins in one direction. If B and C are closed, current flows from right to left through the motor and it spins in the other direction. If A and C or B and D are closed, the voltage across the motor is forced to 0, causing the motor to actively brake. If none of the switches are closed, the motor will coast to a stop. The switches in an H-bridge are power transistors. The H-bridge also contains some digital logic to conveniently control the switches.

When the motor current changes abruptly, the inductance of the motor's electromagnet will induce a large voltage spike that could damage the power transistors. Therefore, many H-bridges also have protection diodes in parallel with the switches, as shown in Figure e9.34(b). If the inductive kick drives either terminal of the motor above V_{motor} or below ground, the diodes will turn ON and clamp the voltage at a safe level. H-bridges can dissipate large amounts of power so a heat sink may be necessary to keep them cool.

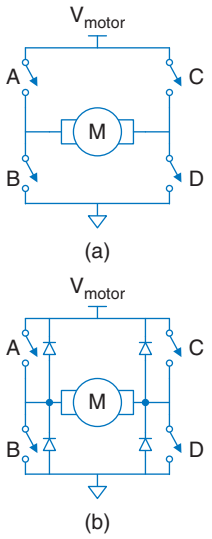


Figure e9.34 H-bridge

Example e9.12 AUTONOMOUS VEHICLE

Design a system in which a Raspberry Pi controls two drive motors for a robot car. Write a library of functions to initialize the motor driver and to make the car drive forward and back, turn left or right, and stop. Use PWM to control the speed of the motors.

Solution: Figure e9.35 shows a pair of DC motors controlled by a Pi via a Texas Instruments SN754410 dual H-bridge. The H-bridge requires a 5 V logic supply V_{CC1} and a 4.5–36 V motor supply V_{CC2}; it has $V_{IH} = 2$ V and is hence compatible with the 3.3 V I/O from the Pi. It can deliver up to 1 A of current to each of two motors. V_{motor} should come from a separate battery pack; the 5 V output of the Pi cannot supply enough current to drive most motors and the Pi could be damaged.

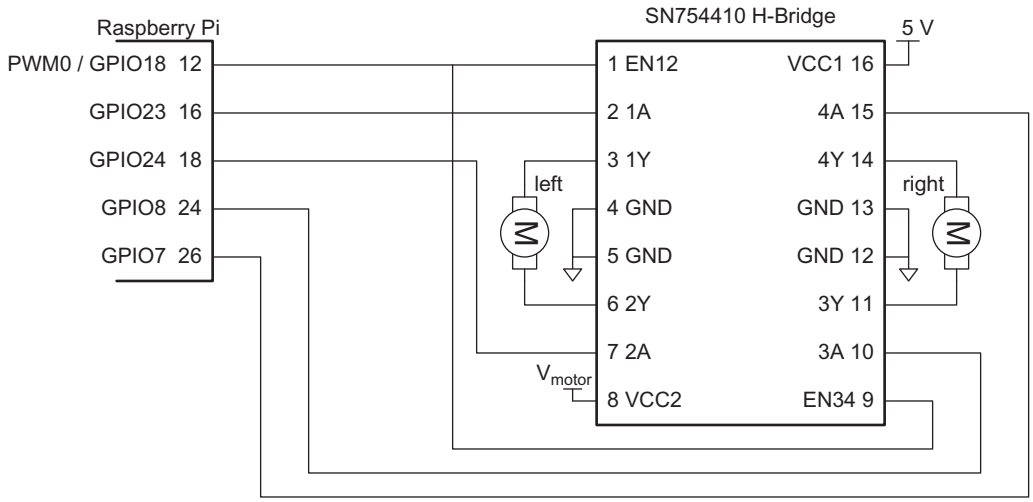


Figure e9.35 Motor control with dual H-bridge

Table e9.10 H-Bridge control

EN12	1A	2A	Motor
0	X	X	Coast
1	0	0	Brake
1	0	1	Reverse
1	1	0	Forward
1	1	1	Brake

Table e9.10 describes how the inputs to each H-bridge control a motor. The microcontroller drives the enable signals with a PWM signal to control the speed of the motors. It drives the four other pins to control the direction of each motor.

The PWM is configured to work at about 5 kHz with a duty cycle ranging from 0 to 100%. Any PWM frequency far higher than the motor's bandwidth will give the effect of smooth movement. Note that the relationship between duty cycle and motor speed is nonlinear and that below some duty cycle, the motor will not move at all.

```
#include "EasyPIO.h"

// Motor Constants
#define MOTOR_1A 23
#define MOTOR_2A 24
#define MOTOR_3A 8
#define MOTOR_4A 7

void setSpeed(float dutycycle) {        // pwmInit() must be called first.
    setPWM(5000, dutycycle);
}

void setMotorLeft(int dir) {            // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_1A, dir);
    digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir) {           // dir of 1 = forward, 0 = backward
    digitalWrite(MOTOR_3A, dir);
    digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
    setMotorLeft(1); setMotorRight(1); // Both motors drive forward
}

void backward(void) {
    setMotorLeft(0); setMotorRight(0); // Both motors drive backward
}

void left(void) {
    setMotorLeft(0); setMotorRight(1); // Left back, right forward

}
```

```
void right(void) {
    setMotorLeft(1); setMotorRight(0); // Right back, left forward
}

void halt(void) {                      // Turn both motors off
    digitalWrite(MOTOR_1A, 0);
    digitalWrite(MOTOR_2A, 0);
    digitalWrite(MOTOR_3A, 0);
    digitalWrite(MOTOR_4A, 0);
}

void initMotors(void) {
    pinMode(MOTOR_1A, OUTPUT);
    pinMode(MOTOR_2A, OUTPUT);
    pinMode(MOTOR_3A, OUTPUT);
    pinMode(MOTOR_4A, OUTPUT);
    halt();                            // Ensure motors are not spinning
    pwmInit();                         // Turn on PWM
    setSpeed(0.75);                    // Default to partial power
}

main(void) {
    pioInit();
    initMotors();
    forward(); delayMillis(5000);
    backward(); delayMillis(5000);
    left(); delayMillis(5000);
    right(); delayMillis(5000);
    halt();
}
```

In the previous example, there is no way to measure the position of each motor. Two motors are unlikely to be exactly matched, so one is likely to turn slightly faster than the other, causing the robot to veer off course. To solve this problem, some systems add shaft encoders. Figure e9.36(a) shows a simple shaft encoder consisting of a disk with slots attached to the motor shaft. An LED is placed on one side and a light sensor is placed on the other side. The shaft encoder produces a pulse every time the gap rotates past the LED. A microcontroller can count these pulses to measure the total angle that the shaft has turned. By using two LED/sensor pairs spaced half a slot width apart, an improved shaft encoder can produce quadrature outputs shown in Figure e9.36(b) that indicate the direction the shaft is turning as well as the angle by which it has turned. Sometimes shaft encoders add another hole to indicate when the shaft is at an index position.

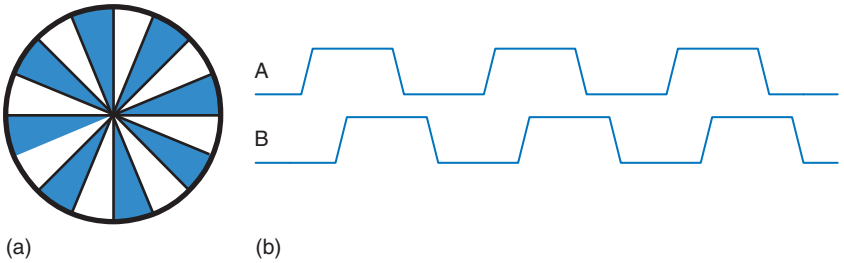


Figure e9.36 Shaft encoder (a) disk, (b) quadrature outputs

9.4.4.2 Servo Motor

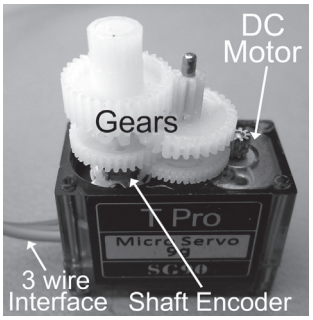


Figure e9.37 SG90 servo motor

A servo motor is a DC motor integrated with a gear train, a shaft encoder, and some control logic so that it is easier to use. They have a limited rotation, typically 180°. Figure e9.37 shows a servo with the lid removed to reveal the gears. A servo motor has a 3-pin interface with power (typically 5 V), ground, and a control input. The control input is typically a 50 Hz pulse-width modulated signal. The servo's control logic drives the shaft to a position determined by the duty cycle of the control input. The servo's shaft encoder is typically a rotary potentiometer that produces a voltage dependent on the shaft position.

In a typical servo motor with 180 degrees of rotation, a pulse width of 0.5 ms drives the shaft to 0°, 1.5 ms to 90°, and 2.5 ms to 180°. For example, Figure e9.38 shows a control signal with a 1.5 ms pulse width. Driving the servo outside its range may cause it to hit mechanical stops and be damaged. The servo's power comes from the power pin rather than the control pin, so the control can connect directly to a microcontroller without an H-bridge. Servo motors are commonly used in remote-control model airplanes and small robots because they are small, light, and convenient. Finding a motor with an adequate datasheet can be difficult. The center pin with a red wire is normally power, and the black or brown wire is normally ground.

Example e9.13 SERVO MOTOR

Design a system in which a Raspberry Pi drives a servo motor to a desired angle.

Solution: Figure e9.39 shows a diagram of the connection to an SG90 servo motor, including the colors of the wires on the servo cable. The servo operates off of a 4.0–7.2 V power supply. It can draw as much as 0.5 A if it must deliver a large amount of force, but may run directly off the Raspberry Pi power supply if the load is light. A single wire carries the PWM signal, which can be provided at 5 or 3.3 V logic levels. The code configures the PWM generation and computes

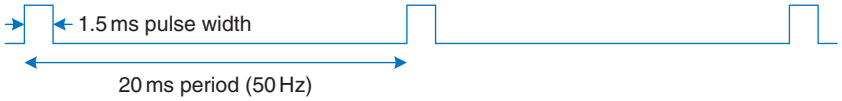


Figure e9.38 Servo control waveform

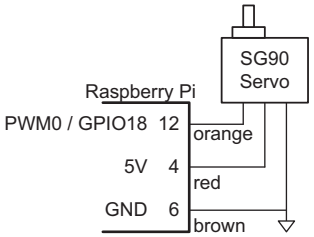


Figure e9.39 Servo motor control

the appropriate duty cycle for the desired angle. It cycles through positioning the servo at 0, 90, and 180 degrees.

```
#include "EasyPIO.h"
void setServo(float angle) {
  setPWM(50.0, 0.025 + (0.1 * (angle / 180)));
}

void main(void) {
  pioInit();
  pwmInit();
  while (1) {
    setServo(0.0);     // Left
    delayMillis(1000);
    setServo(90.0);    // Center
    delayMillis(1000);
    setServo(180.0);   // Right
    delayMillis(1000);
  }
}
```

It is also possible to convert an ordinary servo into a continuous rotation servo by carefully disassembling it, removing the mechanical stop, and replacing the potentiometer with a fixed voltage divider. Many websites show detailed directions for particular servos. The PWM will then control the velocity rather than position, with 1.5 ms indicating stop, 2.5 ms indicating full speed forward, and 0.5 ms indicating full speed backward. A continuous rotation servo may be more convenient and less expensive than a simple DC motor combined with an H-bridge and gear train.

9.4.4.3 Stepper Motor

A stepper motor advances in discrete steps as pulses are applied to alternate inputs. The step size is usually a few degrees, allowing precise positioning and continuous rotation. Small stepper motors generally come with two sets of coils called *phases* wired in *bipolar* or *unipolar* fashion. Bipolar motors are more powerful and less expensive for a given size but require an H-bridge driver, while unipolar motors can be driven with transistors acting as switches. This section focuses on the more efficient bipolar stepper motor.

Figure e9.40(a) shows a simplified two-phase bipolar motor with a 90° step size. The rotor is a permanent magnet with one north and one south pole. The stator is an electromagnet with two pairs of coils comprising

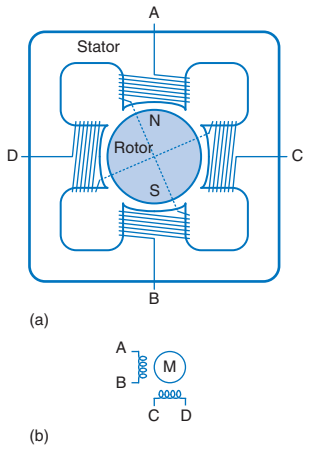


Figure e9.40 **Two-phase bipolar motor: (a) simplified diagram, (b) symbol**

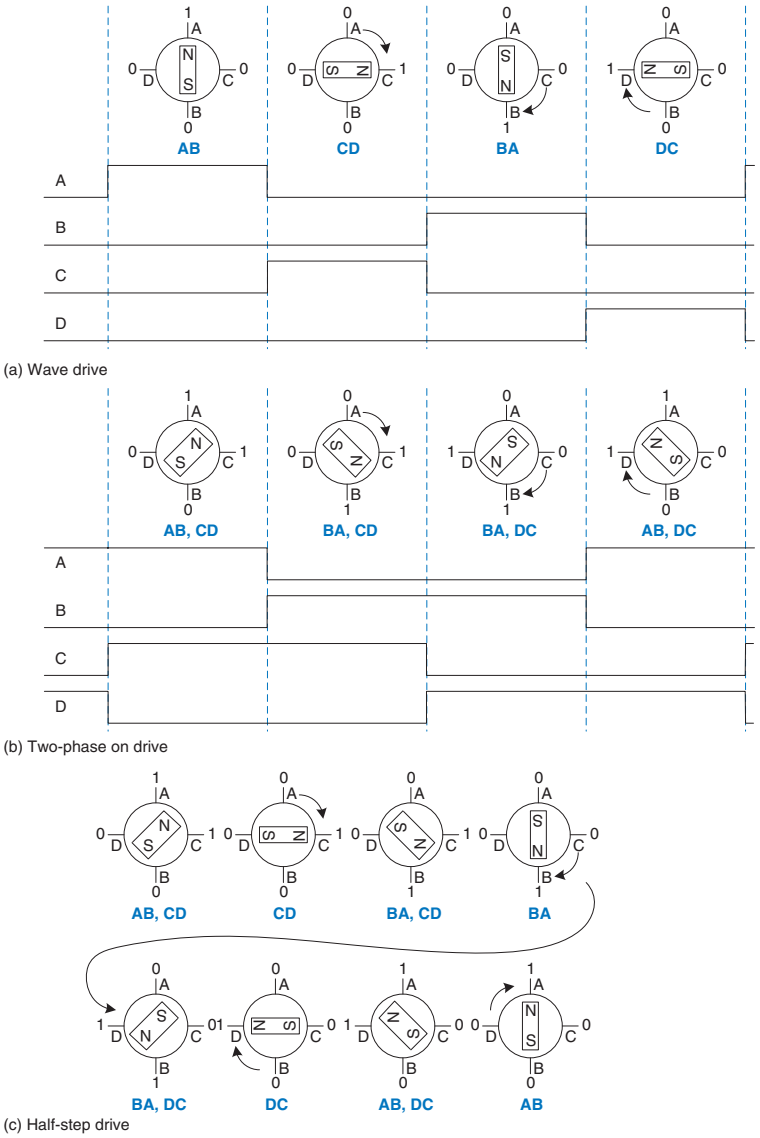


Figure e9.41 **Bipolar motor drive**

the two phases. Two-phase bipolar motors thus have four terminals. Figure e9.40(b) shows a symbol for the stepper motor modeling the two coils as inductors. Practical motors add gearing to reduce the output step size and increase torque.

Figure e9.41 shows three common drive sequences for a two phase bipolar motor. Figure e9.41(a) illustrates *wave drive*, in which the coils

are energized in the sequence AB – CD – BA – DC. Note that BA means that the winding AB is energized with current flowing in the opposite direction; this is the origin of the name bipolar. The rotor turns by 90 degrees at each step. Figure e9.41(b) illustrates *two-phase-on drive*, following the pattern (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC). (AB, CD) indicates that both coils AB and CD are energized simultaneously. The rotor again turns by 90 degrees at each step, but aligns itself halfway between the two pole positions. This gives the highest torque operation because both coils are delivering power at once. Figure e9.41(c) illustrates *half-step drive*, following the pattern (AB, CD) – CD – (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. The rotor turns by 45 degrees at each half-step. The rate at which the pattern advances determines the speed of the motor. To reverse the motor direction, the same drive sequences are applied in the opposite order.

In a real motor, the rotor has many poles to make the angle between steps much smaller. For example, Figure e9.42 shows an AIRPAX LB82773-M1 bipolar stepper motor with a 7.5 degree step size. The motor operates off 5 V and draws 0.8 A through each coil.

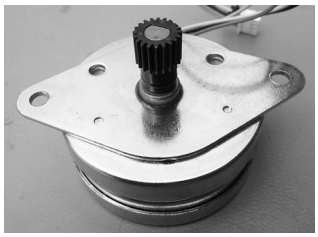


Figure e9.42 AIRPAX LB82773-M1 bipolar stepper motor

The torque in the motor is proportional to the coil current. This current is determined by the voltage applied and by the inductance L and resistance R of the coil. The simplest mode of operation is called *direct voltage drive* or *L/R drive*, in which the voltage V is directly applied to the coil. The current ramps up to $I = V/R$ with a time constant set by L/R, as shown in Figure e9.43(a). This works well for slow speed operation. However, at higher speed, the current doesn't have enough time to ramp up to the full level, as shown in Figure e9.43(b), and the torque drops off.

A more efficient way to drive a stepper motor is by pulse-width modulating a higher voltage. The high voltage causes the current to ramp up to full current more rapidly, then it is turned off (PWM) to avoid overloading the motor. The voltage is then modulated or *chopped* to maintain the current near the desired level. This is called *chopper constant current drive* and is shown in Figure e9.43(c). The controller uses a small resistor in series with the motor to sense the current being applied by measuring the voltage drop, and applies an enable signal to the H-bridge to turn off the drive when the current reaches the desired level. In principle, a microcontroller could generate the right waveforms, but it is easier to use a stepper motor controller. The L297 controller from ST Microelectronics is a convenient choice, especially when coupled with the L298 dual H-bridge with current sensing pins and a 2 A peak power capability. Unfortunately, the L298 is not available in a DIP package so it is harder to breadboard. ST's application notes AN460 and AN470 are valuable references for stepper motor designers.

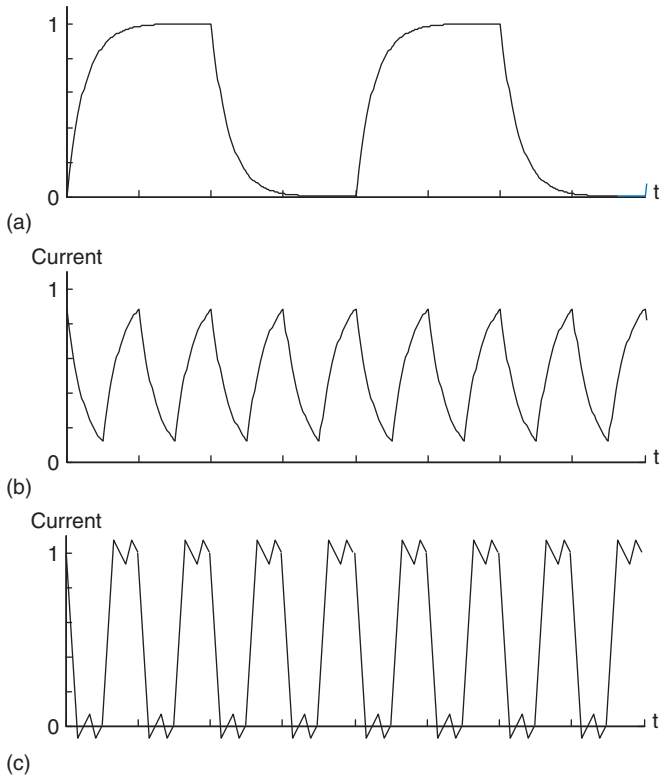


Figure e9.43 **Bipolar stepper motor direct drive current: (a) slow rotation, (b) fast rotation, (c) fast rotation with chopper drive**

Example e9.14 BIPOLAR STEPPER MOTOR DIRECT WAVE DRIVE

Design a system to drive an AIRPAX bipolar stepper motor at a specified speed and direction using direct wave drive.

Solution: Figure e9.44 shows the bipolar stepper motor driven directly by an H-bridge with the same interface as the DC motor. Note that VCC2 must supply enough voltage and current to meet the motor's demands or else the motor may skip steps as the rotation rate increases.

```
#include "EasyPIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MICROS_PER_SEC 1000000
#define DEG_PER_REV 360

int stepperPins[] = {18, 8, 7, 23, 24};
```

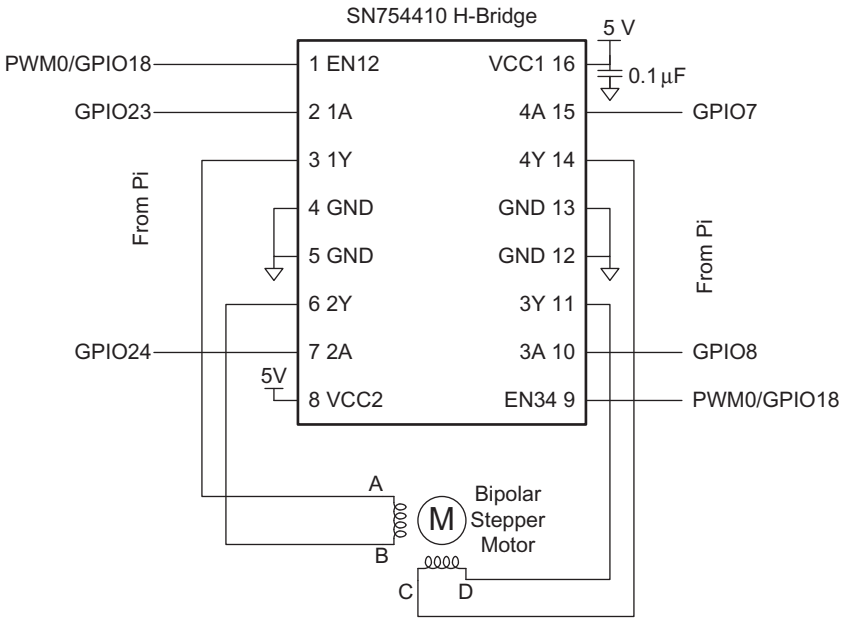


Figure e9.44 **Bipolar stepper motor direct drive with H-bridge**

```
int curStepState;  // Keep track of the current position of stepper motor

void stepperInit(void) {
    pinsMode(stepperPins, 5, OUTPUT);
    curStepState = 0;
}

void stepperSpin(int dir, int steps, float rpm) {
    int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001};    //{2A, 1A, 4A, 3A, EN}
    int step = 0;
    unsigned int microsPerStep = (SECS_PER_MIN * MICROS_PER_SEC * STEPSIZE) /
                                 (rpm * DEG_PER_REV);
    for (step = 0; step < steps; step++) {
        digitalWrites(stepperPins, 5, sequence[curStepState]);
        if (dir == 0) curStepState = (curStepState + 1) % 4;
        else          curStepState = (curStepState + 3) % 4;
        delayMicros(microsPerStep);
    }
}

void main(void) {
    pioInit();
    stepperInit();
    stepperSpin(1, 12000, 120);  // Spin 60 revolutions at 120 rpm
}
```

9.5 BUS INTERFACES

A *bus interface* connects processors to memory and/or peripherals. In general, a bus interface supports one or more *bus masters* that can initiate read or write requests to the bus and one or more *slaves* that respond to the requests; processors are normally masters and memory and peripherals are slaves.

The *Advanced Microcontroller Bus Architecture* (AMBA) is an open standard bus interface for connecting components on a chip. Introduced by ARM in 1996, it has developed through multiple revisions to boost performance and features and has become a *de facto* standard for embedded microcontrollers. The *Advanced High-performance Bus* (AHB) is one of the AMBA standards. AHB-Lite is a simplified version of AHB that supports a single bus master. This section describes AHB-Lite to illustrate the characteristics of a typical bus interface and to show how to design memory and peripherals that interface to a standard bus.

AHB is an example of a point-to-point read bus, in contrast with older bus architectures that use a single shared data bus where each slave accesses the bus via a tristate driver. Using point-to-point links between each slave and the read multiplexer allows the bus to run faster and avoids wasting power when one slave turns on its driver before another has turned off.

9.5.1 AHB-Lite

Figure e9.45 shows a simple AHB-Lite bus connecting a processor (bus master) to RAM, ROM, and two peripherals (slaves). Observe that the bus is very similar to the one from Figure e9.1 except that the names have changed. The master provides a synchronous clock (HCLK) to all of the slaves and can reset the slaves by asserting HRESETn low. The master sends an address. The address decoder uses the most significant bits to generate the HSEL signal selecting which slave to access, and the slaves use the least significant bits to define the memory location or register. The master sends HWDATA for writes. Each slave reads onto its own HRDATA, and a multiplexer chooses the data from the selected slave.

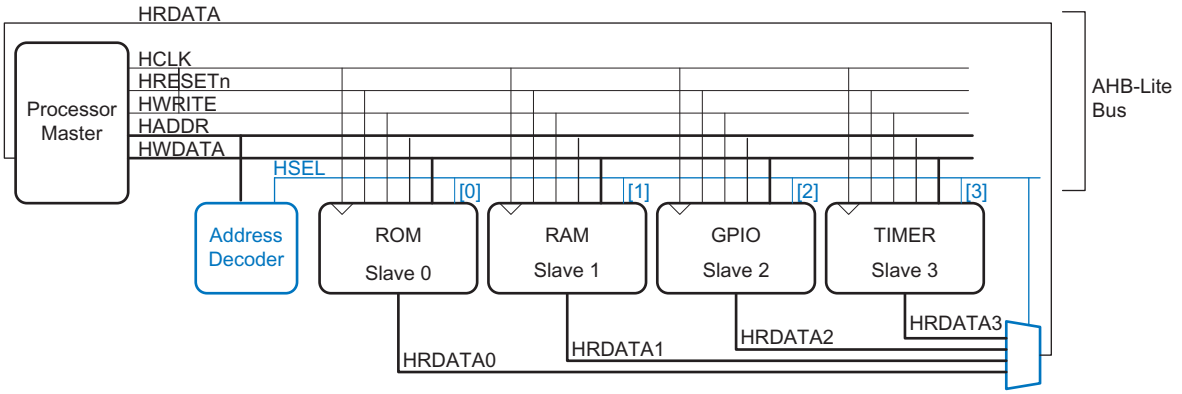


Figure e9.45 AHB-Lite bus

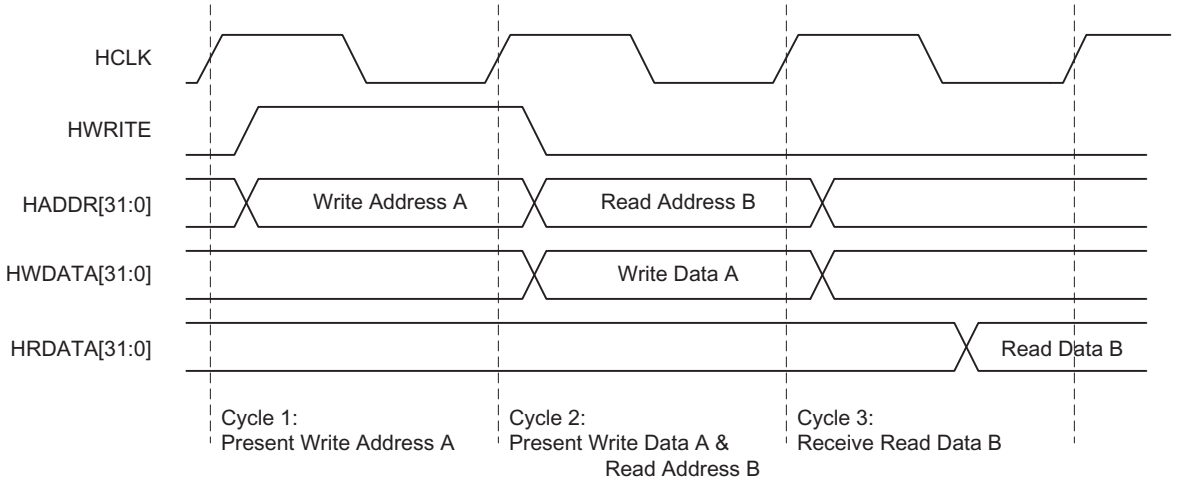


Figure e9.46 **AHB-Lite transfer timing**

The master sends a 32-bit address on one cycle and writes or reads data on the subsequent cycle. The write or read is called a *transfer*. For writes, the master raises HWRITE and sends the 32-bit HWDATA to write. For reads, the master lowers HWRITE and the slave responds with 32-bit HRDATA. Transfers can overlap so that the master can send the address of the next transfer while reading or writing data for the current transfer. Figure e9.46 illustrates the timing of the bus for a write followed immediately by a read. Observe how the data lags one cycle behind the address and how the two transfers partially overlap.

In this example, we assume the bus transfers a single 32-bit word at a time and that the slave responds in one clock cycle. AHB-Lite defines additional signals to specify the size of the transfer (8 – 1024 bits) and to transfer bursts of 4 to 16 elements. The master can also specify types of transfers, protection, and bus locking. Slaves can deassert HREADY to indicate that they need multiple clock cycles to respond, or can assert HRESP to indicate an error. Interested readers should consult the *AMBA 3 AHB-Lite Protocol Specification*, available online.

9.5.2 Memory and Peripheral Interface Example

This section illustrates connecting RAM, ROM, GPIO, and a timer to a processor over an AHB-Lite bus. Figure e9.47 shows a memory map for the system from Figure e9.45 with 128 KB of RAM and 64 KB of ROM. The GPIO controls 32 I/O pins. The 32-bit GPIO_DIR register controls whether each pin is an output (1) or an input (0). The 32-bit

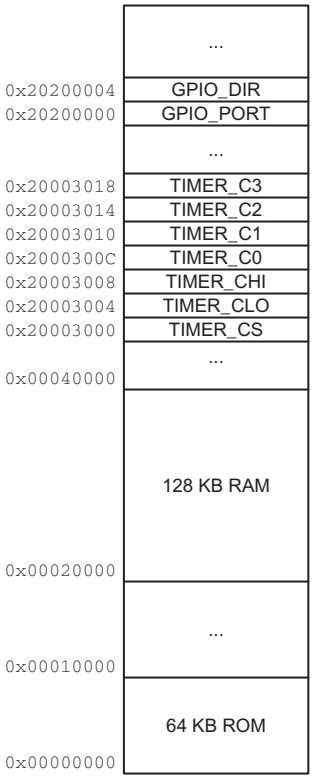


Figure e9.47 **System memory map**

GPIO_PORT register can be written to specify the value of outputs and read to return the values on the pins. The Timer module resembles the BCM2835 counter described in Section 9.3.5, containing a 64-bit counter running at the HCLK frequency (TIMER_CHI:TIMER_CLO), four 32-bit compare channels (TIMER_C3:0), and a match register (TIMER_CS).

HDL Example e9.1 lists SystemVerilog code for the system. The decoder is based on the memory map. The memories and peripherals interface to the bus. Unnecessary signals are omitted; for example, the ROM ignores writes. The GPIO module also connects to 32 I/O pins that can behave as inputs or outputs.

HDL Example e9.1

```
module ahb_lite(input logic          HCLK,
                input logic          HRESETn,
                input logic  [31:0]  HADDR,
                input logic          HWRITE,
                input logic  [31:0]  HWDATA,
                output logic [31:0]  HRDATA,
                inout tri    [31:0]  pins);

  logic [3:0] HSEL;
  logic [31:0] HRDATA0, HRDATA1, HRDATA2, HRDATA3;
  logic [31:0] pins_dir, pins_out, pins_in;
  logic [31:0] HADDRDEL;
  logic HWRITEDEL;

  // Delay address and write signals to align in time with data
  flop #(32) adrreg(HCLK, HADDR, HADDRDEL);
  flop #(1)       writereg(HCLK, HWRITE, HWRITEDEL);

  // Memory map decoding
  ahb_decoder dec(HADDRDEL, HSEL);
  ahb_mux  mux(HSEL, HRDATA0, HRDATA1, HRDATA2, HRDATA3,
               HRDATA);

  // Memory and peripherals
  ahb_rom     rom (HCLK, HSEL[0], HADDRDEL[15:2], HRDATA0);
  ahb_ram     ram (HCLK, HSEL[1], HADDRDEL[16:2], HWRITEDEL,
              HWDATA, HRDATA1);
  ahb_gpio    gpio (HCLK, HRESETn, HSEL[2], HADDRDEL[2],
                    HWRITEDEL, HWDATA, HRDATA2, pins);
  ahb_timer   timer(HCLK, HRESETn, HSEL[3], HADDRDEL[4:2],
                    HWRITEDEL, HWDATA, HRDATA3);
endmodule

module ahb_decoder(input logic [31:0] HADDR,
                   output logic [3:0] HSEL);

  // Decode based on most significant bits of the address
  assign HSEL[0]=(HADDR[31:16]
                  ==16'h0000); // 64KB ROM at 0x00000000 -
                               0x0000FFFF
  assign HSEL[1]=(HADDR[31:17]
                  ==15'h0001); // 128KB RAM at 0x00020000 -
                               0x003FFFFF
  assign HSEL[2]=(HADDR[31:4]
                  ==28'h2020000); // GPIO at 0x20200000 -
                  0x20200007
assign HSEL[3]=(HADDR[31:8]
                ==24'h200030); // Timer at 0x20003000 -
                               0x2000301B
endmodule

module ahb_mux(input logic [3:0] HSEL,
               input logic [31:0] HRDATA0, HRDATA1, HRDATA2,
               HRDATA3,
               output logic [31:0] HRDATA);

  always_comb
    casez(HSEL)
      4'b???1: HRDATA <= HRDATA0;
      4'b??10: HRDATA <= HRDATA1;
      4'b?100: HRDATA <= HRDATA2;
      4'b1000: HRDATA <= HRDATA3;
    endcase
endmodule

module ahb_ram(input logic        HCLK,
               input logic        HSEL,
               input logic [16:2] HADDR,
               input logic        HWRITE,
               input logic [31:0] HWDATA,
               output logic [31:0] HRDATA);

  logic [31:0] ram[32767:0]; // 128KB RAM organized as 32K
                             x 32 bits

  assign HRDATA = ram[HADDR]; // *** check addressing is
                              correct
  always_ff @(posedge HCLK)
    if (HWRITE & HSEL) ram[HADDR] <= HWDATA;
endmodule

module ahb_rom(input logic        HCLK,
               input logic        HSEL,
               input logic [16:2] HADDR,
               output logic [31:0] HRDATA);
  logic [31:0] rom[16383:0]; // 64KB ROM organized as 16K x
                             32 bits

  // *** load ROM from disk file

  assign HRDATA = rom[HADDR]; // *** check addressing is
                              correct

endmodule
```

```
module ahb_gpio(input  logic        HCLK,
                input  logic        HRESETn,
                input  logic        HSEL,
                input  logic [2]    HADDR,
                input  logic        HWRITE,
                input  logic [31:0] HWDATA,
                output logic [31:0] HRDATA,
                output logic [31:0] pin_dir,
                output logic [31:0] pin_out,
                input  logic [31:0] pin_in);

  logic [31:0] gpio[1:0];  // GPIO registers

  // write selected register
  always_ff @(posedge HCLK or negedge HRESETn)
    if (~HRESETn) begin
      gpio[0] <= 32'b0;  // GPIO_PORT
      gpio[1] <= 32'b0;  // GPIO_DIR
    end else if (HWRITE & HSEL)
      gpio[HADDR] <= HWDATA;

  // read selected register
  assign HRDATA = HADDR ? gpio[1] : pin_in;

  // send value and direction to I/O drivers
  assign pin_out = gpio[0];
  assign pin_dir = gpio[1];
endmodule

module ahb_timer(input  logic        HCLK,
                 input  logic        HRESETn,
                 input  logic        HSEL,
                 input  logic [4:2]  HADDR,
                 input  logic        HWRITE,
                 input  logic [31:0] HWDATA,
                 output logic [31:0] HRDATA);

  logic [31:0] timers[6:0]; // timer registers
  logic [31:0] chi, clo;    // next counter value
  logic [3:0]  match, clr;  // determine if counter matches
                               compare reg

  // write selected register and update tiers and match
    always_ff @(posedge HCLK or negedge HRESETn)
  if (~HRESETn) begin
    timers[0] <= 32'b0;  // TIMER_CS
    timers[1] <= 32'b0;  // TIMER_CLO
    timers[2] <= 32'b0;  // TIMER_CHI
    timers[3] <= 32'b0;  // TIMER_CO
    timers[4] <= 32'b0;  // TIMER_C1
    timers[5] <= 32'b0;  // TIMER_C2
    timers[6] <= 32'b0;  // TIMER_C3
  end else begin
    timers[0] <= {28'b0, match};
    timers[1] <= (HWRITE & HSEL & HADDR == 3'b000) ?
                 HWDATA : clo
    timers[2] <= (HWRITE & HSEL & HADDR == 3'b000) ?
                 HWDATA : chi;
    if (HWRITE & HSEL & HADDR == 3'b011) timers[3] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b100) timers[4] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b101) timers[5] <= HWDATA;
    if (HWRITE & HSEL & HADDR == 3'b110) timers[6] <= HWDATA;
  end

  // read selected register
  assign HRDATA = timers[HADDR];

  // increment 64-bit counter as pair of TIMER_CHI, TIMER_CLO
  assign {chi, clo} = {timers[2], timers[1]} + 1;

  // generate matches: set match bit when counter matches
     compare register
  //  clear bit when a 1 is written to that position of the match
      register
  assign clr = (HWRITE & HSEL & HADDR == 3'b000 & HWDATA[3:0]);
  assign match[0] = ~clr[0] & (timers[0][0] |
                    (timers[1] == timers[3]));
  assign match[1] = ~clr[1] & (timers[0][1] |
                    (timers[1] == timers[4]));
  assign match[2] = ~clr[2] & (timers[0][2] |
                    (timers[1] == timers[5]));
  assign match[3] = ~clr[3] & (timers[0][3] |
                    (timers[1] == timers[6]));
endmodule

module gpio_pins(input  logic [31:0] pin_dir, // 1 = output,
                                                 0 = input
                 input  logic [31:0] pin_out, // value to drive
                                                 on outputs
                 output logic [31:0] pin_in,  // value read
                                                 from pins
                 inout  tri   [31:0] pin);  // tristate pins

  // Individual tristate control of each pin

  // No graceful way to control tristates on a per-bit basis in
     SystemVerilog
  genvar i;
  generate
  for (i=0; i<32; i=i+1) begin: pinloop
    assign pin[i] = pin_dir[i] ? pin_out[i] : 1'bz;
  end
  endgenerate

  assign pin_in = pin;
endmodule
```

9.6 PC I/O SYSTEMS

Personal computers (PCs) use a wide variety of I/O protocols for purposes including memory, disks, networking, internal expansion cards, and external devices. These I/O standards have evolved to offer very high performance and to make it easy for users to add devices. These attributes

come at the expense of complexity in the I/O protocols. This section explores the major I/O standards used in PCs and examines some options for connecting a PC to custom digital logic or other external hardware.

Figure e9.48 shows a PC motherboard for a Core i5 or i7 processor. The processor is packaged in a *land grid array* with 1156 gold-plated pads to supply power and ground to the processor and connect the processor to memory and I/O devices. The motherboard contains the DRAM memory module slots, a wide variety of I/O device connectors, and the power supply connector, voltage regulators, and capacitors. A pair of DRAM modules are connected over a DDR3 interface. External peripherals such as keyboards or webcams are attached over USB. High-performance expansion cards such as graphics cards connect over the PCI Express x16 slot, while lower-performance cards can use PCI Express x1 or the older PCI slots. The PC connects to the network using the Ethernet jack. The hard disk connects to a SATA port. The remainder of this section gives an overview of the operation of each of these I/O standards.

One of the major advances in PC I/O standards has been the development of high-speed serial links. Until recently, most I/O was built around parallel links consisting of a wide data bus and a clock signal. As data rates increased, the difference in delay among the wires in the bus set a limit to how fast the bus could run. Moreover, busses connected to multiple devices suffer from transmission line problems such as reflections and different flight times to different loads. Noise can also corrupt the data. Point-to-point serial

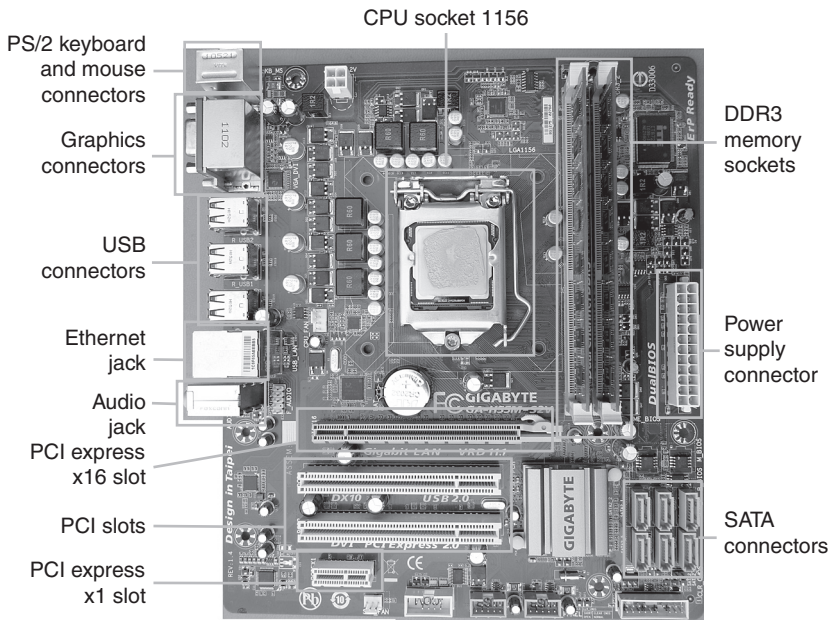


Figure e9.48 **Gigabyte GA-H55M-S2V motherboard**

links eliminate many of these problems. The data is usually transmitted on a differential pair of wires. External noise that affects both wires in the pair equally is unimportant. The transmission lines are easy to properly terminate, so reflections are small (see Section A.8 on transmission lines). No explicit clock is sent; instead, the clock is recovered at the receiver by watching the timing of the data transitions. High-speed serial link design is a specialized subject, but good links can run faster than 10 Gb/s over copper wires and even faster along optical fibers.

9.6.1 USB

Until the mid-1990s, adding a peripheral to a PC took some technical savvy. Adding expansion cards required opening the case, setting jumpers to the correct position, and manually installing a device driver. Adding an RS-232 device required choosing the right cable and properly configuring the baud rate, and data, parity, and stop bits. The *Universal Serial Bus* (USB), developed by Intel, IBM, Microsoft, and others, greatly simplified adding peripherals by standardizing the cables and software configuration process. Billions of USB peripherals are now sold each year.

USB 1.0 was released in 1996. It uses a simple cable with four wires: 5 V, GND, and a differential pair of wires to carry data. The cable is impossible to plug in backward or upside down. It operates at up to 12 Mb/s. A device can pull up to 500 mA from the USB port, so keyboards, mice, and other peripherals can get their power from the port rather than from batteries or a separate power cable.

USB 2.0, released in 2000, upgraded the speed to 480 Mb/s by running the differential wires much faster. With the faster link, USB became practical for attaching webcams and external hard disks. Flash memory sticks with a USB interface also replaced floppy disks as a means of transferring files between computers.

USB 3.0, released in 2008, further boosted the speed to 5 Gb/s. It uses the same shape connector, but the cable has more wires that operate at very high speed. It is well suited to connecting high-performance hard disks. At about the same time, USB added a Battery Charging Specification that boosts the power supplied over the port to speed up charging mobile devices.

The simplicity for the user comes at the expense of a much more complex hardware and software implementation. Building a USB interface from the ground up is a major undertaking. Even writing a simple device driver is moderately complex.

9.6.2 PCI and PCI Express

The *Peripheral Component Interconnect* (PCI) bus is an expansion bus standard developed by Intel that became widespread around 1994. It was

used to add expansion cards such as extra serial or USB ports, network interfaces, sound cards, modems, disk controllers, or video cards. The 32-bit parallel bus operates at 33 MHz, giving a bandwidth of 133 MB/s.

The demand for PCI expansion cards has steadily declined. More standard ports such as Ethernet and SATA are now integrated into the motherboard. Many devices that once required an expansion card can now be connected over a fast USB 2.0 or 3.0 link. And video cards now require far more bandwidth than PCI can supply.

Contemporary motherboards often still have a small number of PCI slots, but fast devices like video cards are now connected via *PCI Express* (PCIe). PCI Express slots provide one or more lanes of high-speed serial links. In PCIe 3.0, each lane operates at up to 8 Gb/s. Most motherboards provide an x16 slot with 16 lanes giving a total of 16 GB/s of bandwidth to data-hungry devices such as video cards.

9.6.3 DDR3 Memory

DRAM connects to the microprocessor over a parallel bus. In 2015, the present standard is DDR3, a third generation of double-data rate memory bus operating at 1.5 V. Typical motherboards now come with two DDR3 channels so they can simultaneously access two banks of memory modules. DDR4 is emerging, operating at 1.2V and higher speed.

Figure e9.49 shows a 4 GB DDR3 dual inline memory module (DIMM). The module has 120 contacts on each side, for a total of 240 connections, including a 64-bit data bus, a 16-bit time-multiplexed address bus, control signals, and numerous power and ground pins. In 2015, DIMMs typically carry 1–16 GB of DRAM. Memory capacity has been doubling approximately every 2–3 years.

DRAM presently operates at a clock rate of 100–266 MHz. DDR3 operates the memory bus at four times the DRAM clock rate. Moreover, it transfers data on both the rising and falling edges of the clock. Hence, it sends 8 words of data for each memory clock. At 64 bits/word, this corresponds to 6.4–17 GB/s of bandwidth. For example, DDR3-1600 uses a 200 MHz memory clock and an 800 MHz I/O clock to send 1600 million words/sec, or 12800 MB/s. Hence, the modules are

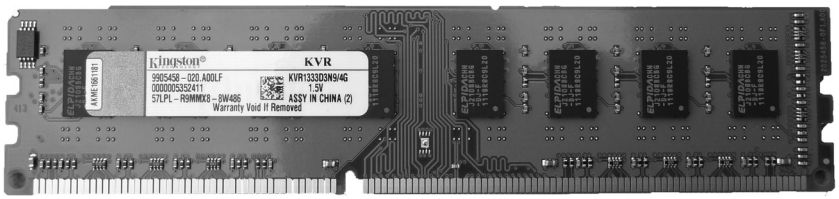


Figure e9.49 DDR3 memory module

also called PC3-12800. Unfortunately, DRAM latency remains high, with a roughly 50 ns lag from a read request until the arrival of the first word of data.

9.6.4 Networking

Computers connect to the Internet over a network interface running the *Transmission Control Protocol and Internet Protocol* (TCP/IP). The physical connection may be an Ethernet cable or a wireless Wi-Fi link.

Ethernet is defined by the IEEE 802.3 standard. It was developed at Xerox Palo Alto Research Center (PARC) in 1974. It originally operated at 10 Mb/s (called 10 Mbit Ethernet), but now is commonly found at 100 Mbit (Mb/s) and 1 Gbit (Gb/s) running on Category 5 cables containing four twisted pairs of wires. 10 Gbit Ethernet running on fiber optic cables is increasingly popular for servers and other high-performance computing, and 100 Gbit Ethernet is emerging.

Wi-Fi is the popular name for the IEEE 802.11 wireless network standard. It operates in the 2.4 and 5 GHz unlicensed wireless bands, meaning that the user doesn't need a radio operator's license to transmit in these bands at low power. Table e9.11 summarizes the capabilities of three generations of Wi-Fi; the emerging 802.11ac standard promises to push wireless data rates beyond 1 Gb/s. The increasing performance comes from advancing modulation and signal processing, multiple antennas, and wider signal bandwidths.

9.6.5 SATA

Internal hard disks require a fast interface to a PC. In 1986, Western Digital introduced the *Integrated Drive Electronics* (IDE) interface, which evolved into the *AT Attachment* (ATA) standard. The standard uses a bulky 40 or 80-wire ribbon cable with a maximum length of 18″ to send data at 16–133 MB/s.

Table e9.11 802.11 Wi-Fi protocols

Protocol	Release	Frequency Band (GHz)	Data Rate (Mb/s)	Range (m)
802.11b	1999	2.4	5.5–11	35
802.11g	2003	2.4	6–54	38
802.11n	2009	2.4/5	7.2–150	70
802.11ac	2013	5	433+	variable

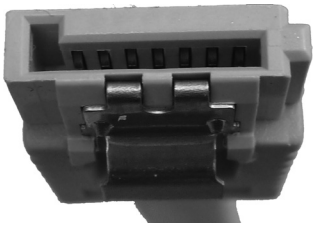


Figure e9.50 SATA cable

ATA has been supplanted by Serial ATA (SATA), which uses high-speed serial links to run at 1.5, 3, or 6 Gb/s over a more convenient 7-conductor cable shown in Figure e9.50. The fastest solid-state drives in 2015 exceed 500 MB/s of bandwidth, taking full advantage of SATA.

A related standard is Serial Attached SCSI (SAS), an evolution of the parallel SCSI (Small Computer System Interface). SAS offers performance comparable to SATA and supports longer cables; it is common in server computers.

9.6.6 Interfacing to a PC

All of the PC I/O standards described so far are optimized for high performance and ease of attachment but are difficult to implement in hardware. Engineers and scientists often need a way to connect a PC to external circuitry, such as sensors, actuators, microcontrollers, or FPGAs. The serial connection described in Section 9.3.4.2 is sufficient for a low-speed connection to a microcontroller with a UART. This section describes two more means: data acquisition systems, and USB links.

9.6.6.1 Data Acquisition Systems

Data Acquisition Systems (DAQs) connect a computer to the real world using multiple channels of analog and/or digital I/O. DAQs are now commonly available as USB devices, making them easy to install. National Instruments (NI) is a leading DAQ manufacturer.

High-performance DAQ prices tend to run into the thousands of dollars, mostly because the market is small and has limited competition. Fortunately, NI sells their handy myDAQ system at a student discount price of $200 including their LabVIEW software. Figure e9.51 shows a myDAQ. It has two analog channels capable of input and output

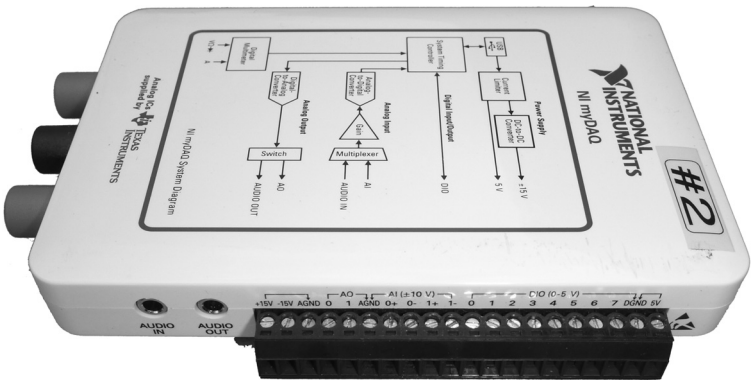


Figure e9.51 NI myDAQ

at 200 ksamples/sec with a 16-bit resolution and ±10 V dynamic range. These channels can be configured to operate as an oscilloscope and signal generator. It also has eight digital input and output lines compatible with 3.3 and 5 V systems. Moreover, it generates 5, 15, and −15 V power supply outputs and includes a digital multimeter capable of measuring voltage, current, and resistance. Thus, the myDAQ can replace an entire bench of test and measurement equipment while simultaneously offering automated data logging.

Most NI DAQs are controlled with LabVIEW, NI's graphical language for designing measurement and control systems. Some DAQs can also be controlled from C programs using the LabWindows environment, from Microsoft .NET applications using the Measurement Studio environment, or from Matlab using the Data Acquisition Toolbox.

9.6.6.2 USB Links

An increasing variety of products now provide simple, inexpensive digital links between PCs and external hardware over USB. These products contain predeveloped drivers and libraries, allowing the user to easily write a program on the PC that blasts data to and from an FPGA or microcontroller.

FTDI is a leading vendor for such systems. For example, the FTDI C232HM-DDHSL USB to Multi-Protocol Synchronous Serial Engine (MPSSE) cable shown in Figure e9.52 provides a USB jack at one end and, at the other end, an SPI interface operating at up to 30 Mb/s, along with 3.3 V power and four general purpose I/O pins. Figure e9.53 shows an example of connecting a PC to an FPGA using the cable. The cable can optionally supply 3.3 V power to the FPGA. The three SPI pins connect to an FPGA slave device like the one from Example e9.4. The figure also shows one of the GPIO pins used to drive an LED.

The PC requires the D2XX dynamically linked library driver to be installed. You can then write a C program using the library to send data over the cable.

Figure e9.52 FTDI USB to MPSSE cable
(© 2012 by FTDI; reprinted with permission.)

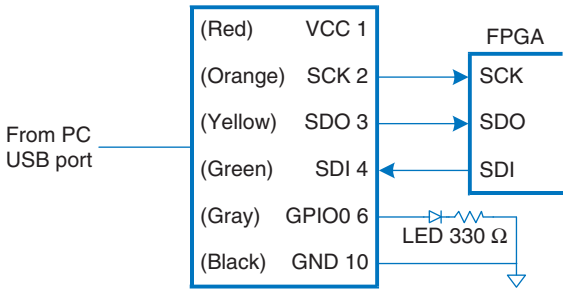


Figure e9.53 C232HM-DDHSL USB to MPSESE interface from PC to FPGA

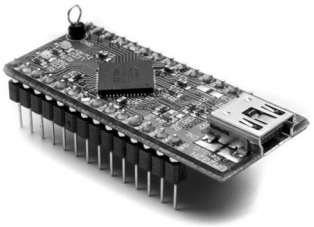


Figure e9.54 FTDI UM232H module
(© 2012 by FTDI; reprinted with permission.)

If an even faster connection is required, the FTDI UM232H module shown in Figure e9.54 links a PC's USB port to an 8-bit synchronous parallel interface operating up to 40 MB/s.

9.7 SUMMARY

Most processors use memory-mapped I/O to communicate with the real world. Microcontrollers offer a range of basic peripherals including general-purpose, serial, and analog I/O and timers. PCs and advanced microcontrollers support more complex I/O standards including USB, Ethernet, and SATA.

This chapter has provided many specific examples of I/O using the Raspberry Pi. Embedded system designers continually encounter new processors and peripherals. The general principal for simple embedded I/O is to consult the datasheet to identify the peripherals that are available and which pins and memory-mapped I/O registers are involved. Then it is usually straightforward to write a simple device driver that initializes the peripheral and then transmits or receives data.

For more complex standards such as USB, writing a device driver is a highly specialized undertaking best done by an expert with detailed knowledge of the device and the USB protocol stack. Casual designers should select a processor that comes with proven device drivers and example code for the devices of interest.

Digital System Implementation

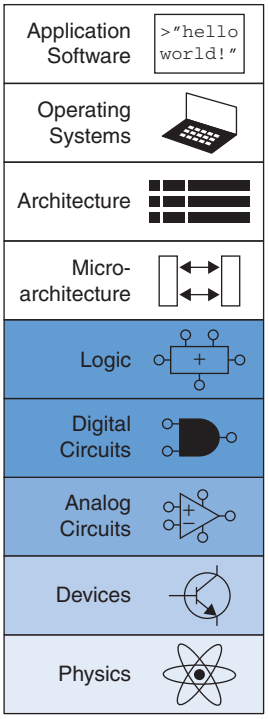
A

A.1 INTRODUCTION

This appendix introduces practical issues in the design of digital systems. The material is not necessary for understanding the rest of the book, however, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

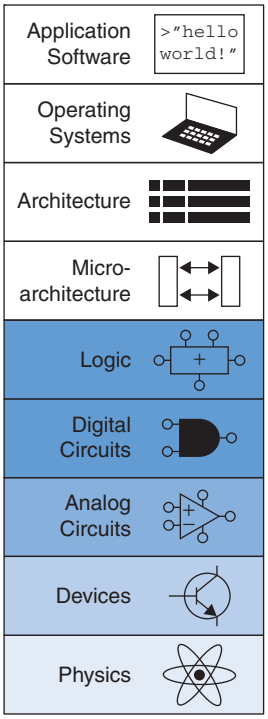
Digital systems are usually built using one or more chips. One strategy is to connect together chips containing individual logic gates or larger elements such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.

The rest of this chapter is available online as a downloadable PDF from the book's companion site: http://booksite.elsevier.com/9780128000564.



Digital System Implementation

eA



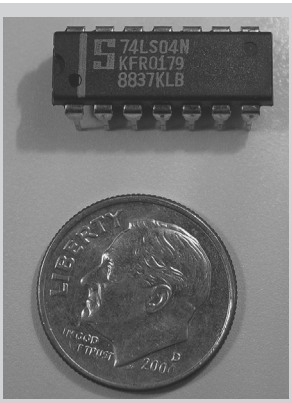
A.1 INTRODUCTION

This appendix introduces practical issues in the design of digital systems. The material is not necessary for understanding the rest of the book, however, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

Digital systems are usually built using one or more chips. One strategy is to connect together chips containing individual logic gates or larger elements such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.

A.2 74xx LOGIC

In the 1970s and 1980s, many digital systems were built from simple chips, each containing a handful of logic gates. For example, the 7404 chip contains six NOT gates, the 7408 contains four AND gates, and the 7474 contains two flip-flops. These chips are collectively referred to as *74xx-series* logic. They were sold by many manufacturers, typically for 10 to 25 cents per chip. These chips are now largely obsolete, but they are still handy for simple digital systems or class projects, because they are so inexpensive and easy to use. 74xx-series chips are commonly sold in 14-pin *dual inline packages* (*DIPs*).



74LS04 inverter chip in a 14-pin dual inline package. The part number is on the first line. LS indicates the logic family (see Section A.6). The N suffix indicates a DIP package. The large S is the logo of the manufacturer, Signetics. The bottom two lines of gibberish are codes indicating the batch in which the chip was manufactured.

A.2.1 Logic Gates

Figure eA.1 shows the pinout diagrams for a variety of popular 74xx-series chips containing basic logic gates. These are sometimes called *small-scale integration* (*SSI*) chips, because they are built from a few transistors. The 14-pin packages typically have a notch at the top or a dot on the top left to indicate orientation. Pins are numbered starting with 1 in the upper left and going counterclockwise around the package. The chips need to receive power ($V_{DD} = 5$ V) and ground (GND = 0 V) at pins 14 and 7, respectively. The number of logic gates on the chip is determined by the number of pins. Note that pins 3 and 11 of the 7421 chip are not connected (NC) to anything. The 7474 flip-flop has the usual D, CLK, and Q terminals. It also has a complementary output, $\overline{Q}$. Moreover, it receives asynchronous set (also called preset, or PRE) and reset (also called clear, or CLR) signals. These are active low; in other words, the flop sets when $\overline{PRE} = 0$, resets when $\overline{CLR} = 0$, and operates normally when $\overline{PRE} = \overline{CLR} = 1$.

A.2.2 Other Functions

The 74xx series also includes somewhat more complex logic functions, including those shown in Figures eA.2 and eA.3. These are called *medium-scale integration* (*MSI*) chips. Most use larger packages to accommodate more inputs and outputs. Power and ground are still provided at the upper right and lower left, respectively, of each chip. A general functional description is provided for each chip. See the manufacturer's data sheets for complete descriptions.

A.3 PROGRAMMABLE LOGIC

Programmable logic consists of arrays of circuitry that can be configured to perform specific logic functions. We have already introduced three forms of programmable logic: programmable read only memories (PROMs), programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs). This section shows chip implementations for each of these. Configuration of these chips may be performed by blowing on-chip fuses to connect or disconnect circuit elements. This is called *one-time programmable* (*OTP*) logic because, once a fuse is blown, it cannot be restored. Alternatively, the configuration may be stored in a memory that can be reprogrammed at will. Reprogrammable logic is convenient in the laboratory because the same chip can be reused during development.

A.3.1 PROMs

As discussed in Section 5.5.7, PROMs can be used as lookup tables. A 2^N-word $\times$ M-bit PROM can be programmed to perform any combinational function of N inputs and M outputs. Design changes simply involve

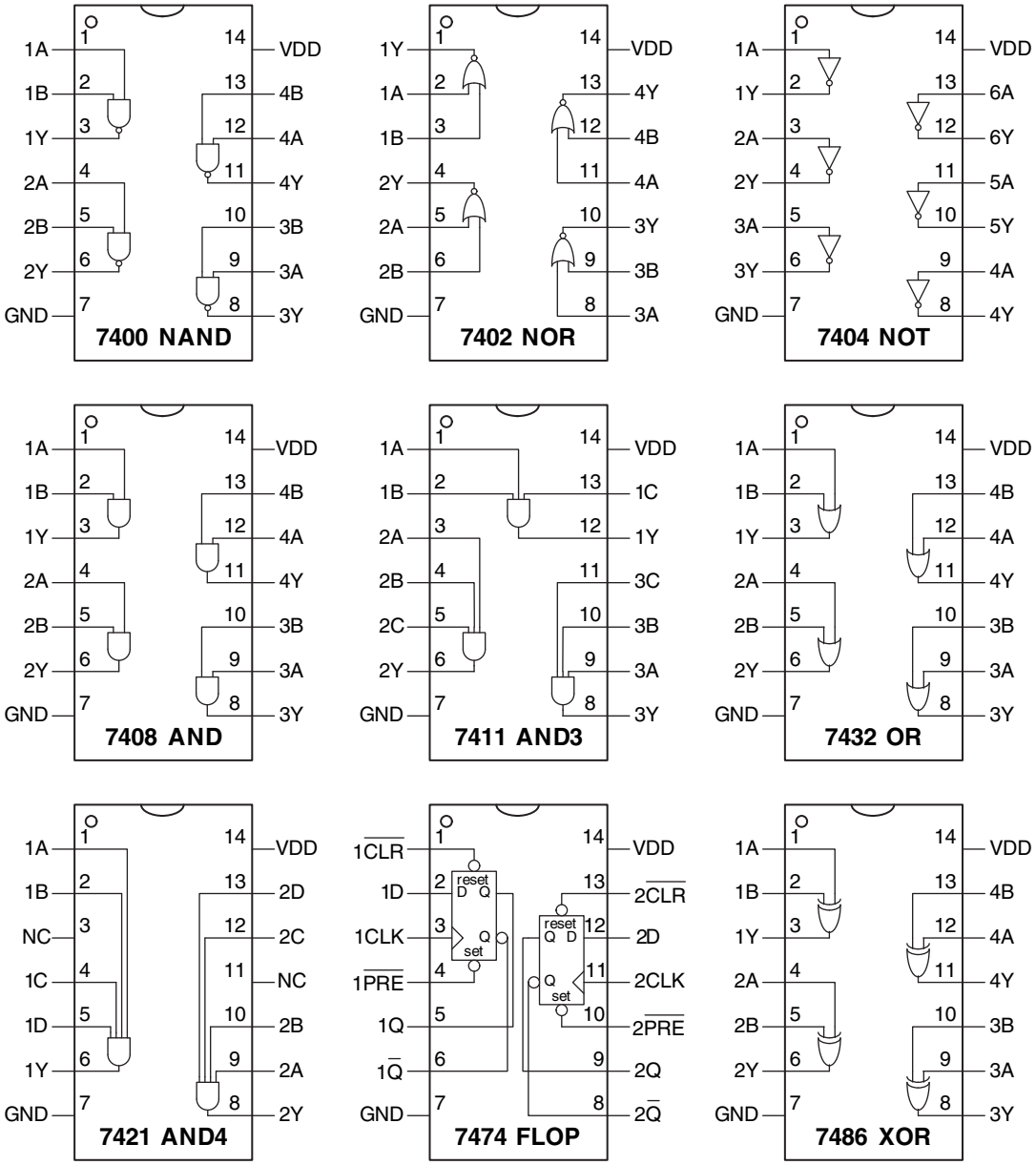
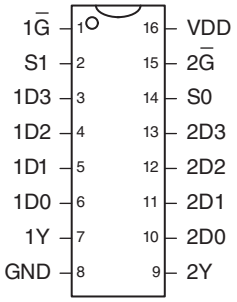


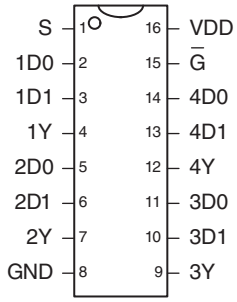
Figure eA.1 Common 74xx-series logic gates



74153 4 :1 Mux

Two 4:1 Multiplexers
$D_{3:0}$: data
$S_{1:0}$: select
Y: output
Gb: enable

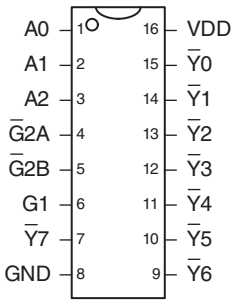
```
always_comb
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always_comb
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```



74157 2 :1 Mux

Four 2:1 Multiplexers
$D_{1:0}$: data
S: select
Y: output
Gb: enable

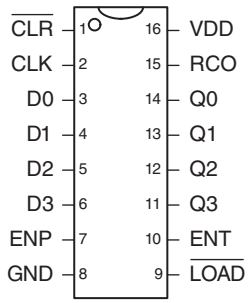
```
always_comb
  if  (Gb) 1Y = 0;
  else     1Y = S ? 1D[1] : 1D[0];
  if  (Gb) 2Y = 0;
  else     2Y = S ? 2D[1] : 2D[0];
  if  (Gb) 3Y = 0;
  else     3Y = S ? 3D[1] : 3D[0];
  if  (Gb) 4Y = 0;
  else     4Y = S ? 4D[1] : 4D[0];
```



74138 3 :8 Decoder

3:8 Decoder
$A_{2:0}$: address
$Yb_{7:0}$: output
G1: active high enable
G2: active low enables

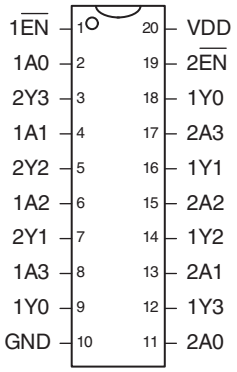
G1	G2A̅	G2B̅	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111



74161/163 Counter

4-bit Counter
CLK: clock
$Q_{3:0}$: counter output
$D_{3:0}$: parallel input
CLRb: async reset (161)
sync reset (163)
LOADb: load Q from D
ENP, ENT: enables
RCO: ripple carry out

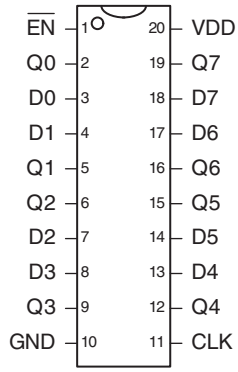
```
always_ff @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```



74244 Tristate Buffer

8-bit Tristate Buffer
$A_{3:0}$: input
$Y_{3:0}$: output
ENb: enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENb ? 4'bzzzz : 2A;
```



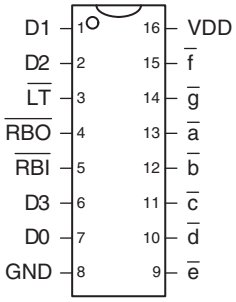
74377 Register

8-bit Enableable Register
CLK: clock
$D_{7:0}$: data
$Q_{7:0}$: output
ENb: enable

```
always_ff @(posedge CLK)
  if (~ENb) Q <= D;
```

Note: SystemVerilog variable names cannot start with numbers, but the names in the example code in Figure A.2 are chosen to match the manufacturer's data sheet.

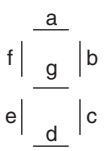
Figure eA.2 Medium-scale integration chips



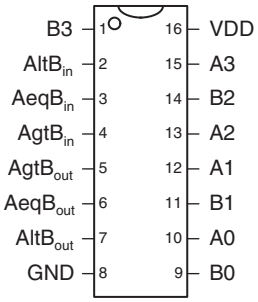
7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$: data
a...f: segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out



RBO	LT	RBI	D3:0	a	b	c	d	e	f	g
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0

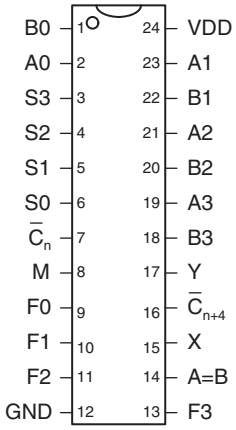


7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always_comb
   if (A > B | (A == B & AgtBin)) begin
     AgtBout = 1; AeqBout = 0; AltBout = 0;
   end
   else if (A < B | (A == B & AltBin) begin
     AgtBout = 0; AeqBout = 0; AltBout = 1;
   end else begin
     AgtBout = 0; AeqBout = 1; AltBout = 0;
   end
```



74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M: mode select
Cb_n: carry in
Cb_{nplus4}: carry out
AeqB: equality (in some modes)
X,Y: carry lookahead adder outputs

```
always_comb
  case (F)
    0000: Y = M ? ~A       :  A                    + ~Cbn;
    0001: Y = M ? ~(A | B) :  A       +  B         + ~Cbn;
    0010: Y = M ? (~A) & B :  A       + ~B         + ~Cbn;
    0011: Y = M ? 4'b0000  :  4'b1111              + ~Cbn;
    0100: Y = M ? ~(A & B) :  A       + (A & ~B)   + ~Cbn;
    0101: Y = M ? ~B       : (A | B)  + (A & ~B)   + ~Cbn;
    0110: Y = M ? A ^ B    :  A       - B          -  Cbn;
    0111: Y = M ? A & ~B   : (A & ~B)              -  Cbn;
    1000: Y = M ? ~A + B   :  A       + (A & B)    + ~Cbn;
    1001: Y = M ? ~(A ^ B) :  A       +  B         + ~Cbn;
    1010: Y = M ? B        : (A | ~B) + (A & B)    + ~Cbn;
    1011: Y = M ? A & B    : (A & B)               + ~Cbn;
    1100: Y = M ? 1        :  A       +  A         + ~Cbn;
    1101: Y = M ? A | ~B   : (A | B)  +  A         + ~Cbn;
    1110: Y = M ? A | B    : (A | ~B) +  A         + ~Cbn;
    1111: Y = M ? A        :  A                    -  Cbn;
  endcase
```

Figure eA.3 More medium-scale integration (MSI) chips

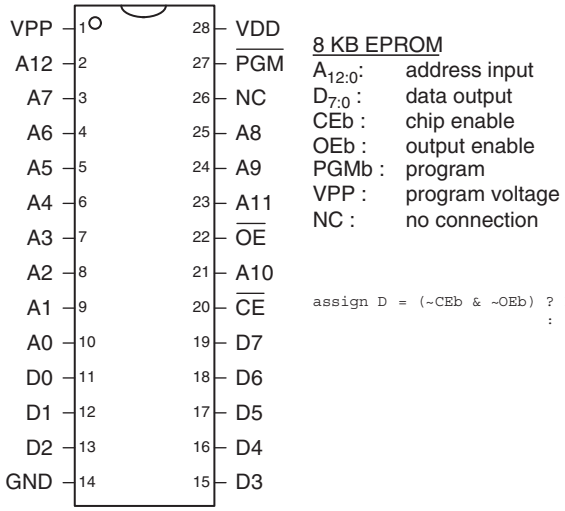


Figure eA.4 2764 8KB EPROM

replacing the contents of the PROM rather than rewiring connections between chips. Lookup tables are useful for small functions but become prohibitively expensive as the number of inputs grows.

For example, the classic 2764 8-KB (64-Kb) erasable PROM (EPROM) is shown in Figure eA.4. The EPROM has 13 address lines to specify one of the 8K words and 8 data lines to read the byte of data at that word. The chip enable and output enable must both be asserted for data to be read. The maximum propagation delay is 200 ps. In normal operation, $\overline{PGM} = 1$ and *VPP* is not used. The EPROM is usually programmed on a special programmer that sets $\overline{PGM} = 0$, applies 13 V to *VPP*, and uses a special sequence of inputs to configure the memory.

Modern PROMs are similar in concept but have much larger capacities and more pins. Flash memory is the cheapest type of PROM, selling for about $0.30 per gigabyte in 2015. Prices have historically declined by 30 to 40% per year.

A.3.2 PLAs

As discussed in Section 5.6.1, PLAs contain AND and OR planes to compute any combinational function written in sum-of-products form. The AND and OR planes can be programmed using the same techniques for PROMs. A PLA has two columns for each input and one column for each output. It has one row for each minterm. This organization is more efficient than a PROM for many functions, but the array still grows excessively large for functions with numerous I/Os and minterms.

Many different manufacturers have extended the basic PLA concept to build *programmable logic devices* (*PLDs*) that include registers. The 22V10

is one of the most popular classic PLDs. It has 12 dedicated input pins and 10 outputs. The outputs can come directly from the PLA or from clocked registers on the chip. The outputs can also be fed back into the PLA. Thus, the 22V10 can directly implement FSMs with up to 12 inputs, 10 outputs, and 10 bits of state. The 22V10 costs about $2 in quantities of 100. PLDs have been rendered mostly obsolete by the rapid improvements in capacity and cost of FPGAs.

A.3.3 FPGAs

As discussed in Section 5.6.2, FPGAs consist of arrays of configurable *logic elements* (*LEs*), also called *configurable logic blocks* (*CLBs*), connected together with programmable wires. The LEs contain small lookup tables and flip-flops. FPGAs scale gracefully to extremely large capacities, with thousands of lookup tables. Xilinx and Altera are two of the leading FPGA manufacturers.

Lookup tables and programmable wires are flexible enough to implement any logic function. However, they are an order of magnitude less efficient in speed and cost (chip area) than hard-wired versions of the same functions. Thus, FPGAs often include specialized blocks, such as memories, multipliers, and even entire microprocessors.

Figure eA.5 shows the design process for a digital system on an FPGA. The design is usually specified with a hardware description language (HDL), although some FPGA tools also support schematics. The design is then simulated. Inputs are applied and compared against expected outputs to *verify* that the logic is correct. Usually some debugging is required. Next, logic *synthesis* converts the HDL into Boolean functions. Good synthesis tools produce a schematic of the functions, and the prudent designer examines these schematics, as well as any warnings produced during synthesis, to ensure that the desired logic was produced. Sometimes sloppy coding leads to circuits that are much larger than intended or to circuits with asynchronous logic. When the synthesis results are good, the FPGA tool *maps* the functions onto the LEs of a specific chip. The *place and route* tool determines which functions go in which lookup tables and how they are wired together. Wire delay increases with length, so critical circuits should be placed close together. If the design is too big to fit on the chip, it must be reengineered. *Timing analysis* compares the timing constraints (e.g., an intended clock speed of 100 MHz) against the actual circuit delays and reports any errors. If the logic is too slow, it may have to be redesigned or pipelined differently. When the design is correct, a file is generated specifying the contents of all the LEs and the programming of all the wires on the FPGA. Many FPGAs store this *configuration* information in static RAM that must be reloaded each time the FPGA is

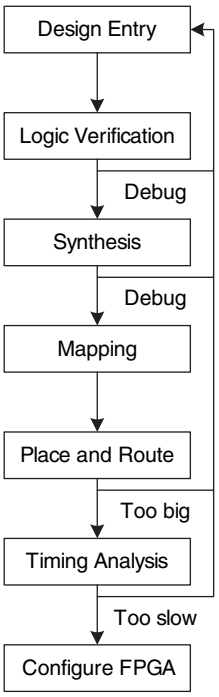


Figure eA.5 FPGA design flow

turned on. The FPGA can download this information from a computer in the laboratory, or can read it from a nonvolatile ROM when power is first applied.

Example eA.1 FPGA TIMING ANALYSIS

Alyssa P. Hacker is using an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. Her design is implemented as an FSM, and she is using a Cyclone IV FPGA. According to the data sheet, the FPGA has the timing characteristics shown in Table eA.1.

Alyssa would like her FSM to run at 100 MHz. What is the maximum number of LEs on the critical path? What is the fastest speed at which her FSM could possibly run?

Solution: At 100 MHz, the cycle time, T_c, is 10 ns. Alyssa uses Equation 3.14 to figure out the minimum combinational propagation delay, t_{pd}, at this cycle time:

$$t_{pd} \leq 10\,\text{ns} - (0.199\,\text{ns} + 0.076\,\text{ns}) = 9.725\,\text{ns} \tag{A.1}$$

With a combined LE and wire delay of 381 ps + 246 ps = 627 ps, Alyssa's FSM can use at most 15 consecutive LEs (9.725/0.627) to implement the next-state logic.

The fastest speed at which an FSM will run on this Cyclone IV FPGA is when it is using a single LE for the next state logic. The minimum cycle time is

$$T_c \geq 381\,\text{ps} + 199\,\text{ps} + 76\,\text{ps} = 656\,\text{ps} \tag{A.2}$$

Therefore, the maximum frequency is 1.5 GHz.

Table eA.1 Cyclone IV timing

Name	Value (ps)
t_{pcq}	199
t_{setup}	76
t_{hold}	0
t_{pd} (per LE)	381
t_{wire} (between LEs)	246
t_{skew}	0

Altera advertises the Cyclone IV FPGA with 14,400 LEs for $25 in 2015. In large quantities, medium-sized FPGAs typically cost several dollars. The largest FPGAs cost hundreds or even thousands of dollars.

The cost has declined at approximately 30% per year, so FPGAs are becoming extremely popular.

A.4 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Application-specific integrated circuits (*ASICs*) are chips designed for a particular purpose. Graphics accelerators, network interface chips, and cell phone chips are common examples of ASICs. The ASIC designer places transistors to form logic gates and wires the gates together. Because the ASIC is hardwired for a specific function, it is typically several times faster than an FPGA and occupies an order of magnitude less chip area (and hence cost) than an FPGA with the same function. However, the *masks* specifying where transistors and wires are located on the chip cost hundreds of thousands of dollars to produce. The fabrication process usually requires 6 to 12 weeks to manufacture, package, and test the ASICs. If errors are discovered after the ASIC is manufactured, the designer must correct the problem, generate new masks, and wait for another batch of chips to be fabricated. Hence, ASICs are suitable only for products that will be produced in large quantities and whose function is well defined in advance.

Figure eA.6 shows the ASIC design process, which is similar to the FPGA design process of Figure eA.5. Logic verification is especially important because correction of errors after the masks are produced is expensive. Synthesis produces a *netlist* consisting of logic gates and connections between the gates; the gates in this netlist are placed, and the wires are routed between gates. When the design is satisfactory, masks are generated and used to fabricate the ASIC. A single speck of dust can ruin an ASIC, so the chips must be tested after fabrication. The fraction of manufactured chips that work is called the *yield*; it is typically 50 to 90%, depending on the size of the chip and the maturity of the manufacturing process. Finally, the working chips are placed in packages, as will be discussed in Section A.7.

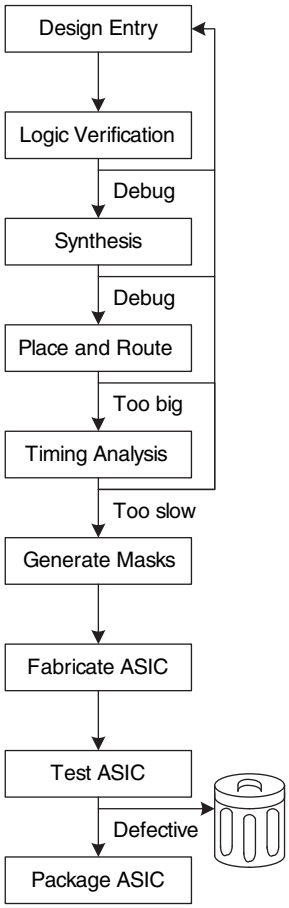


Figure eA.6 ASIC design flow

A.5 DATA SHEETS

Integrated circuit manufacturers publish *data sheets* that describe the functions and performance of their chips. It is essential to read and understand the data sheets. One of the leading sources of errors in digital systems comes from misunderstanding the operation of a chip.

Data sheets are usually available from the manufacturer's Web site. If you cannot locate the data sheet for a part and do not have clear documentation from another source, don't use the part. Some of the entries in the data sheet may be cryptic. Often the manufacturer publishes data books containing data sheets for many related parts. The beginning of

the data book has additional explanatory information. This information can usually be found on the Web with a careful search.

This section dissects the Texas Instruments (TI) data sheet for a 74HC04 inverter chip. The data sheet is relatively simple but illustrates many of the major elements. TI still manufacturers a wide variety of 74xx-series chips. In the past, many other companies built these chips too, but the market is consolidating as the sales decline.

Figure eA.7 shows the first page of the data sheet. Some of the key sections are highlighted in blue. The title is SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS means that the chip contains six inverters. SN indicates that TI is the manufacturer. Other manufacture codes include MC for Motorola and DM for National Semiconductor. You can generally ignore these codes, because all of the manufacturers build compatible 74xx-series logic. HC is the logic family (high speed CMOS). The logic family determines the speed and power consumption of the chip, but not the function. For example, the 7404, 74HC04, and 74LS04 chips all contain six inverters, but they differ in performance and cost. Other logic families are discussed in Section A.6. The 74xx chips operate across the commercial or industrial temperature range (0 to 70 °C or −40 to 85 °C, respectively), whereas the 54xx chips operate across the military temperature range (−55 to 125 °C) and sell for a higher price but are otherwise compatible.

The 7404 is available in many different packages, and it is important to order the one you intended when you make a purchase. The packages are distinguished by a suffix on the part number. N indicates a *plastic dual inline package* (*PDIP*), which fits in a breadboard or can be soldered in through-holes in a printed circuit board. Other packages are discussed in Section A.7.

The function table shows that each gate inverts its input. If A is HIGH (H), Y is LOW (L) and vice versa. The table is trivial in this case but is more interesting for more complex chips.

Figure eA.8 shows the second page of the data sheet. The logic diagram indicates that the chip contains inverters. The *absolute maximum* section indicates conditions beyond which the chip could be destroyed. In particular, the power supply voltage (V_{CC}, also called V_{DD} in this book) should not exceed 7 V. The continuous output current should not exceed 25 mA. The *thermal resistance* or impedance, θ_{JA}, is used to calculate the temperature rise caused by the chip's dissipating power. If the *ambient* temperature in the vicinity of the chip is T_A and the chip dissipates P_{chip}, then the temperature on the chip itself at its *junction* with the package is

$$T_J = T_A + P_{\text{chip}}\,\theta_{JA} \tag{A.3}$$

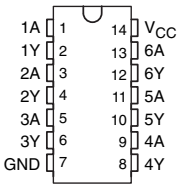
For example, if a 7404 chip in a plastic DIP package is operating in a hot box at 50 °C and consumes 20 mW, the junction temperature will climb

SN54HC04, SN74HC04
HEX INVERTERS

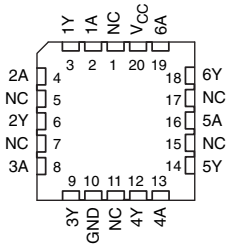
SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- **Wide Operating Voltage Range of 2 V to 6 V**
- **Outputs Can Drive Up To 10 LSTTL Loads**
- **Low Power Consumption, 20-μA Max I_{cc}**
- **Typical tpd = 8 ns**
- **±4-mA Output Drive at 5 V**
- **Low Input Current of 1 μA Max**

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)



SN54HC04 . . . FK PACKAGE
(TOPVIEW)



NC – No internal connection

description/ordering information

The 'HC04 devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers there to appears at the end of this data sheet.

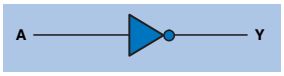
TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Figure eA.7 7404 data sheet page 1

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)



absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC} . . . . . –0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1) . . . . . ±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1) . . . . . ±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC}) . . . . . ±25 mA
Continuous current through V_{CC} or GND . . . . . ±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package . . . . . 86° C/W
N package . . . . . 80° C/W
NS package . . . . . 76° C/W
PW package . . . . . 131° C/W
Storage temperature range, T_{stg} . . . . . –65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

Figure eA.8 7404 data sheet page 2

to 50 °C + 0.02 W × 80 °C/W = 51.6 °C. Internal power dissipation is seldom important for 74xx-series chips, but it becomes important for modern chips that dissipate tens of watts or more.

The *recommended operating conditions* define the environment in which the chip should be used. Within these conditions, the chip should meet specifications. These conditions are more stringent than the absolute maximums. For example, the power supply voltage should be between 2 and 6 V. The input logic levels for the HC logic family depend on V_{DD}. Use the 4.5 V entries when $V_{DD} = 5$ V, to allow for a 10% droop in the power supply caused by noise in the system.

Figure eA.9 shows the third page of the data sheet. The *electrical characteristics* describe how the device performs when used within the recommended operating conditions if the inputs are held constant. For example, if $V_{CC} = 5$ V (and droops to 4.5 V) and the output current I_{OH}/I_{OL} does not exceed 20 μA, $V_{OH} = 4.4$ V and $V_{OL} = 0.1$ V in the worst case. If the output current increases, the output voltages become less ideal, because the transistors on the chip struggle to provide the current. The HC logic family uses CMOS transistors that draw very little current. The current into each input is guaranteed to be less than 1000 nA and is typically only 0.1 nA at room temperature. The *quiescent* power supply current (I_{DD}) drawn while the chip is idle is less than 20 μA. Each input has less than 10 pF of capacitance.

The *switching characteristics* define how the device performs when used within the recommended operating conditions if the inputs change. The *propagation delay*, t_{pd}, is measured from when the input passes through 0.5 V_{CC} to when the output passes through 0.5 V_{CC}. If V_{CC} is nominally 5 V and the chip drives a capacitance of less than 50 pF, the propagation delay will not exceed 24 ns (and typically will be much faster). Recall that each input may present 10 pF, so the chip cannot drive more than five identical chips at full speed. Indeed, stray capacitance from the wires connecting chips cuts further into the useful load. The *transition time*, also called the rise/fall time, is measured as the output transitions between 0.1 V_{CC} and 0.9 V_{CC}.

Recall from Section 1.8 that chips consume both *static* and *dynamic power*. Static power is low for HC circuits. At 85 °C, the maximum quiescent supply current is 20 μA. At 5 V, this gives a static power consumption of 0.1 mW. The dynamic power depends on the capacitance being driven and the switching frequency. The 7404 has an internal power dissipation capacitance of 20 pF per inverter. If all six inverters on the 7404 switch at 10 MHz and drive external loads of 25 pF, then the dynamic power given by Equation 1.4 is $\frac{1}{2}(6)(20\text{ pF} + 25\text{ pF})(5^2)(10\text{ MHz}) = 33.75$ mW and the maximum total power is 33.85 mW.

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A = 25 °C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	$V_I = V_{IH}$ or V_{IL}	I_{OH} = –20 µA	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		I_{OH} = –4 mA	4.5 V	3.98	4.3		3.7		3.84		
		I_{OH} = –5.2 mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	$V_I = V_{IH}$ or V_{IL}	I_{OL} = 20 µA	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		I_{OL} = 4 mA	4.5 V		0.17	0.26		0.4		0.33	
		I_{OL} = 5.2 mA	6 V		0.15	0.26		0.4		0.33	
I_I	$V_I = V_{CC}$ or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	$V_I = V_{CC}$ or 0, I_O = 0		6 V			2		40		20	µA
C_i			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, C_L = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25 °C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, T_A = 25 °C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

Figure eA.9 7404 data sheet page 3

A.6 LOGIC FAMILIES

The 74xx-series logic chips have been manufactured using many different technologies, called *logic families*, that offer different speed, power, and logic level trade-offs. Other chips are usually designed to be compatible with some of these logic families. The original chips, such as the 7404, were built using bipolar transistors in a technology called *Transistor-Transistor Logic* (*TTL*). Newer technologies add one or more letters after the 74 to indicate the logic family, such as 74LS04, 74HC04, or 74AHCT04. Table eA.2 summarizes the most common 5-V logic families.

Advances in bipolar circuits and process technology led to the *Schottky* (*S*) and *Low-Power Schottky* (*LS*) families. Both are faster than TTL. Schottky draws more power, whereas Low-Power Schottky draws less. *Advanced Schottky* (*AS*) and *Advanced Low-Power Schottky* (*ALS*) have improved speed and power compared to S and LS. *Fast* (*F*) logic is faster and draws less power than AS. All of these families provide more current for LOW outputs than for HIGH outputs and hence have

Table eA.2 Typical specifications for 5-V logic families

	Bipolar / TTL						CMOS		CMOS / TTL Compatible	
Characteristic	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (ns)	22	9	12	7.5	10	6	21	7.5	30	7.7
V_{IH} (V)	2	2	2	2	2	2	3.15	3.15	2	2
V_{IL} (V)	0.8	0.8	0.8	0.8	0.8	0.8	1.35	1.35	0.8	0.8
V_{OH} (V)	2.4	2.7	2.7	2.5	2.5	2.5	3.84	3.8	3.84	3.8
V_{OL} (V)	0.4	0.5	0.5	0.5	0.5	0.5	0.33	0.44	0.33	0.44
I_{OH} (mA)	0.4	1	0.4	2	0.4	1	4	8	4	8
I_{OL} (mA)	16	20	8	20	8	20	4	8	4	8
I_{IL} (mA)	1.6	2	0.4	0.5	0.1	0.6	0.001	0.001	0.001	0.001
I_{IH} (mA)	0.04	0.05	0.02	0.02	0.02	0.02	0.001	0.001	0.001	0.001
I_{DD} (mA)	33	54	6.6	26	4.2	15	0.02	0.02	0.02	0.02
C_{Pd} (pF)	n/a						20	12	20	14
cost* (US $)	obsolete	0.63	0.25	0.53	0.32	0.22	0.12	0.12	0.12	0.12

*Per unit in quantities of 1000 for the 7408 from Texas Instruments in 2012.

asymmetric logic levels. They conform to the "TTL" logic levels: $V_{IH} = 2$ V, $V_{IL} = 0.8$ V, $V_{OH} > 2.4$ V, and $V_{OL} < 0.5$ V.

As CMOS circuits matured in the 1980s and 1990s, they became popular because they draw very little power supply or input current. The *High Speed CMOS* (*HC*) and *Advanced High Speed CMOS* (*AHC*) families draw almost no static power. They also deliver the same current for HIGH and LOW outputs. They conform to the "CMOS" logic levels: $V_{IH} = 3.15$ V, $V_{IL} = 1.35$ V, $V_{OH} > 3.8$ V, and $V_{OL} < 0.44$ V. Unfortunately, these levels are incompatible with TTL circuits, because a TTL HIGH output of 2.4 V may not be recognized as a legal CMOS HIGH input. This motivates the use of *High Speed TTL-compatible CMOS* (*HCT*) and *Advanced High Speed TTL-compatible CMOS* (*AHCT*), which accept TTL input logic levels and generate valid CMOS output logic levels. These families are slightly slower than their pure CMOS counterparts. All CMOS chips are sensitive to *electrostatic discharge* (*ESD*) caused by static electricity. Ground yourself by touching a large metal object before handling CMOS chips, lest you zap them.

The 74xx-series logic is inexpensive. The newer logic families are often cheaper than the obsolete ones. The LS family is widely available and robust and is a popular choice for laboratory or hobby projects that have no special performance requirements.

The 5-V standard collapsed in the mid-1990s, when transistors became too small to withstand the voltage. Moreover, lower voltage offers lower power consumption. Now 3.3, 2.5, 1.8, 1.2, and even lower voltages are commonly used. The plethora of voltages raises challenges in communicating between chips with different power supplies. Table eA.3 lists some of the low-voltage logic families. Not all 74xx parts are available in all of these logic families.

All of the low-voltage logic families use CMOS transistors, the workhorse of modern integrated circuits. They operate over a wide range of V_{DD}, but the speed degrades at lower voltage. *Low-Voltage CMOS* (*LVC*) logic and *Advanced Low-Voltage CMOS* (*ALVC*) logic are commonly used at 3.3, 2.5, or 1.8 V. LVC withstands inputs up to 5.5 V, so it can receive inputs from 5-V CMOS or TTL circuits. *Advanced Ultra-Low-Voltage CMOS* (*AUC*) is commonly used at 2.5, 1.8, or 1.2 V and is exceptionally fast. Both ALVC and AUC withstand inputs up to 3.6 V, so they can receive inputs from 3.3-V circuits.

FPGAs often offer separate voltage supplies for the internal logic, called the *core*, and for the input/output (I/O) pins. As FPGAs have advanced, the core voltage has dropped from 5 to 3.3, 2.5, 1.8, and 1.2 V to save power and avoid damaging the very small transistors. FPGAs have configurable I/Os that can operate at many different voltages, so as to be compatible with the rest of the system.

Table eA.3 Typical specifications for low-voltage logic families

	LVC			ALVC			AUC		
V_{dd} (V)	3.3	2.5	1.8	3.3	2.5	1.8	2.5	1.8	1.2
t_{pd} (ns)	4.1	6.9	9.8	2.8	3	?*	1.8	2.3	3.4
V_{IH} (V)	2	1.7	1.17	2	1.7	1.17	1.7	1.17	0.78
V_{IL} (V)	0.8	0.7	0.63	0.8	0.7	0.63	0.7	0.63	0.42
V_{OH} (V)	2.2	1.7	1.2	2	1.7	1.2	1.8	1.2	0.8
V_{OL} (V)	0.55	0.7	0.45	0.55	0.7	0.45	0.6	0.45	0.3
I_O (mA)	24	8	4	24	12	12	9	8	3
I_I (mA)	0.02			0.005			0.005		
I_{DD} (mA)	0.01			0.01			0.01		
C_{pd} (pF)	10	9.8	7	27.5	23	?*	17	14	14
cost (US $)	0.17			0.20			not available		

*Delay and capacitance not available at the time of writing.

A.7 PACKAGING AND ASSEMBLY

Integrated circuits are typically placed in *packages* made of plastic or ceramic. The packages serve a number of functions, including connecting the tiny metal I/O pads of the chip to larger pins in the package for ease of connection, protecting the chip from physical damage, and spreading the heat generated by the chip over a larger area to help with cooling. The packages are placed on a breadboard or printed circuit board and wired together to assemble the system.

Packages

Figure eA.10 shows a variety of integrated circuit packages. Packages can be generally categorized as *through-hole* or *surface mount* (*SMT*). Through-hole packages, as their name implies, have pins that can be inserted through holes in a printed circuit board or into a socket. *Dual inline packages* (*DIPs*) have two rows of pins with 0.1-inch spacing between pins. *Pin grid arrays* (*PGAs*) support more pins in a smaller package by placing the pins under the package. SMT packages are soldered directly to the surface of a printed circuit board without using holes. Pins on SMT parts are called *leads*. The *thin small outline package* (*TSOP*) has two rows of closely spaced leads (typically 0.02-inch spacing). *Plastic leaded chip carriers* (*PLCCs*) have J-shaped leads on all four

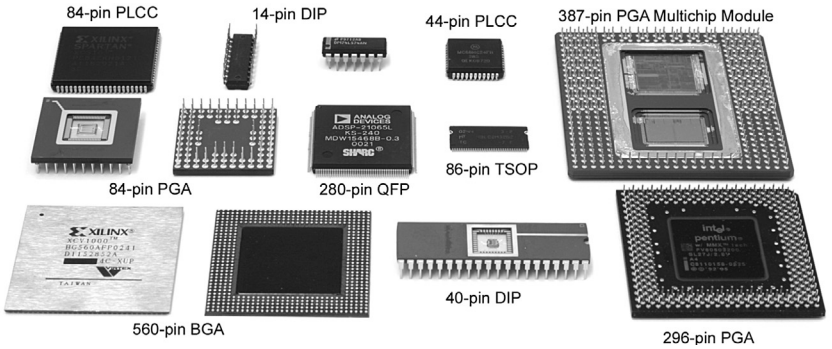


Figure eA.10 Integrated circuit packages

sides, with 0.05-inch spacing. They can be soldered directly to a board or placed in special sockets. *Quad flat packs* (*QFPs*) accommodate a large number of pins using closely spaced legs on all four sides. *Ball grid arrays* (*BGAs*) eliminate the legs altogether. Instead, they have hundreds of tiny solder balls on the underside of the package. They are carefully placed over matching pads on a printed circuit board, then heated so that the solder melts and joins the package to the underlying board.

Breadboards

DIPs are easy to use for prototyping, because they can be placed in a *breadboard*. A breadboard is a plastic board containing rows of sockets, as shown in Figure eA.11. All five holes in a row are connected together. Each pin of the package is placed in a hole in a separate row. Wires can be placed in adjacent holes in the same row to make connections to the pin. Breadboards often provide separate columns of connected holes running the height of the board to distribute power and ground.

Figure eA.11 shows a breadboard containing a majority gate built with a 74LS08 AND chip and a 74LS32 OR chip. The schematic of the circuit is shown in Figure eA.12. Each gate in the schematic is labeled with the chip (08 or 32) and the pin numbers of the inputs and outputs (see Figure eA.1). Observe that the same connections are made on the breadboard. The inputs are connected to pins 1, 2, and 5 of the 08 chip, and the output is measured at pin 6 of the 32 chip. Power and ground are connected to pins 14 and 7, respectively, of each chip, from the vertical power and ground columns that are attached to the banana plug receptacles, Vb and Va. Labeling the schematic in this way and checking off connections as they are made is a good way to reduce the number of mistakes made during breadboarding.

Unfortunately, it is easy to accidentally plug a wire in the wrong hole or have a wire fall out, so breadboarding requires a great deal of care (and usually some debugging in the laboratory). Breadboards are suited only to prototyping, not production.

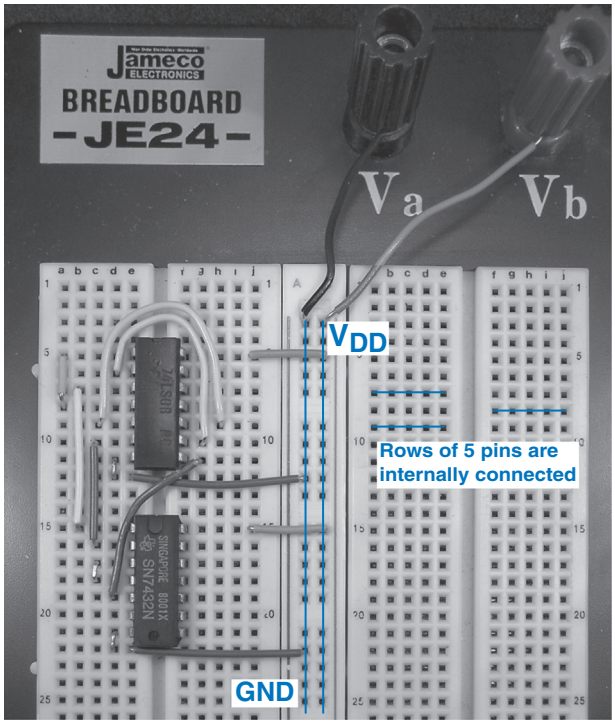


Figure eA.11 Majority circuit on breadboard

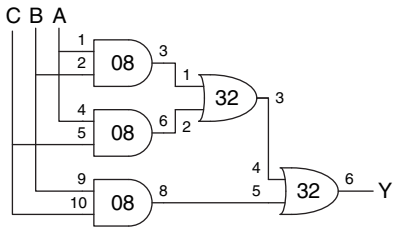


Figure eA.12 Majority gate schematic with chips and pins identified

Printed Circuit Boards

Instead of breadboarding, chip packages may be soldered to a *printed circuit board* (*PCB*). The PCB is formed of alternating layers of conducting copper and insulating epoxy. The copper is etched to form wires called *traces*. Holes called *vias* are drilled through the board and plated with metal to connect between layers. PCBs are usually designed with *computer-aided design* (*CAD*) tools. You can etch and drill your own simple boards in the laboratory, or you can send the board design to a specialized factory for inexpensive mass production. Factories have turnaround times of days (or weeks, for cheap mass production runs) and typically charge a few hundred dollars in setup fees and a few dollars per board for moderately complex boards built in large quantities.

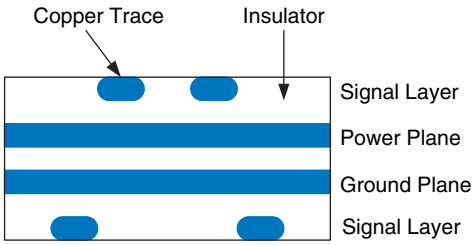


Figure eA.13 Printed circuit board cross-section

PCB traces are normally made of copper because of its low resistance. The traces are embedded in an insulating material, usually a green, fire-resistant plastic called FR4. A PCB also typically has copper power and ground layers, called *planes*, between signal layers. Figure eA.13 shows a cross-section of a PCB. The signal layers are on the top and bottom, and the power and ground planes are embedded in the center of the board. The power and ground planes have low resistance, so they distribute stable power to components on the board. They also make the capacitance and inductance of the traces uniform and predictable.

Figure eA.14 shows a PCB for a 1970s vintage Apple II+ computer. At the top is a 6502 microprocessor. Beneath are six 16-Kb ROM chips forming 12 KB of ROM containing the operating system. Three rows of eight 16-Kb DRAM chips provide 48 KB of RAM. On the right are several rows of 74xx-series logic for memory address decoding and other functions. The lines between chips are traces that wire the chips together. The dots at the ends of some of the traces are vias filled with metal.

Putting It All Together

Most modern chips with large numbers of inputs and outputs use SMT packages, especially QFPs and BGAs. These packages require a printed circuit board rather than a breadboard. Working with BGAs is especially challenging because they require specialized assembly equipment. Moreover, the balls cannot be probed with a voltmeter or oscilloscope during debugging in the laboratory, because they are hidden under the package.

In summary, the designer needs to consider packaging early on to determine whether a breadboard can be used during prototyping and whether BGA parts will be required. Professional engineers rarely use breadboards when they are confident of connecting chips together correctly without experimentation.

A.8 TRANSMISSION LINES

We have assumed so far that wires are *equipotential* connections that have a single voltage along their entire length. Signals actually propagate along wires at the speed of light in the form of electromagnetic waves. If the wires are short enough or the signals change slowly, the equipotential

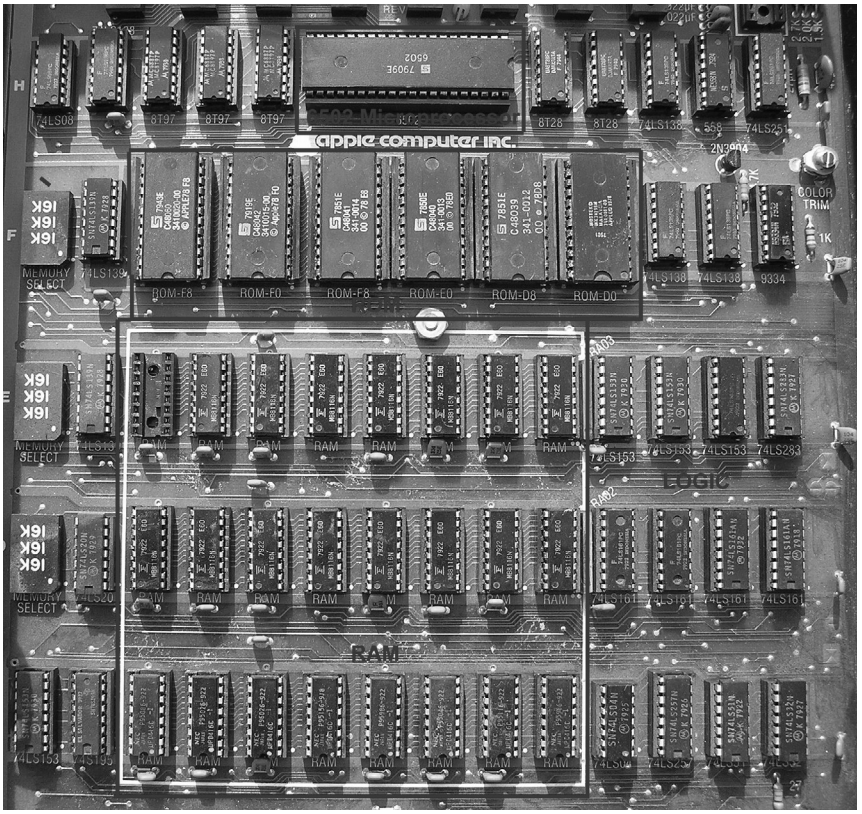


Figure eA.14 Apple II+ circuit board

assumption is good enough. When the wire is long or the signal is very fast, the *transmission time* along the wire becomes important to accurately determine the circuit delay. We must model such wires as *transmission lines*, in which a wave of voltage and current propagates at the speed of light. When the wave reaches the end of the line, it may reflect back along the line. The reflection may cause noise and odd behaviors unless steps are taken to limit it. Hence, the digital designer must consider transmission line behavior to accurately account for the delay and noise effects in long wires.

Electromagnetic waves travel at the speed of light in a given medium, which is fast but not instantaneous. The speed of light, v, depends on the *permittivity*, ε, and *permeability*, μ, of the medium[1]: $v = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}$.

[1] The capacitance, C, and inductance, L, of a wire are related to the permittivity and permeability of the physical medium in which the wire is located.

The speed of light in free space is $v = c = 3 \times 10^8$ m/s. Signals in a PCB travel at about half this speed, because the FR4 insulator has four times the permittivity of air. Thus, PCB signals travel at about 1.5×10^8 m/s, or 15 cm/ns. The time delay for a signal to travel along a transmission line of length l is

$$t_d = \frac{l}{v} \tag{A.4}$$

The *characteristic impedance* of a transmission line, Z_0 (pronounced "Z-naught"), is the ratio of voltage to current in a wave traveling along the line: $Z_0 = V/I$. It is *not* the resistance of the wire (a good transmission line in a digital system typically has negligible resistance). Z_0 depends on the inductance and capacitance of the line (see the derivation in Section A.8.7) and typically has a value of 50 to 75 Ω.

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.5}$$

Figure eA.15 shows the symbol for a transmission line. The symbol resembles a *coaxial cable* with an inner signal conductor and an outer grounded conductor like that used in television cable wiring.

The key to understanding the behavior of transmission lines is to visualize the wave of voltage propagating along the line at the speed of light. When the wave reaches the end of the line, it may be absorbed or reflected, depending on the termination or load at the end. Reflections travel back along the line, adding to the voltage already on the line. Terminations are classified as matched, open, short, or mismatched. The following sections explore how a wave propagates along the line and what happens to the wave when it reaches the termination.

A.8.1 Matched Termination

Figure eA.16 shows a transmission line of length l with a *matched termination*, which means that the load impedance, Z_L, is equal to the characteristic impedance, Z_0. The transmission line has a characteristic impedance of 50 Ω. One end of the line is connected to a voltage source

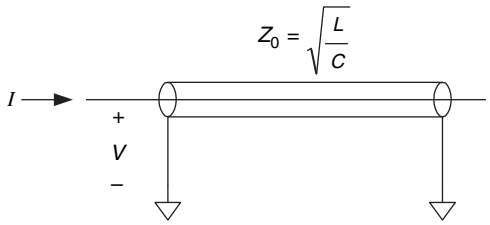


Figure eA.15 Transmission line symbol

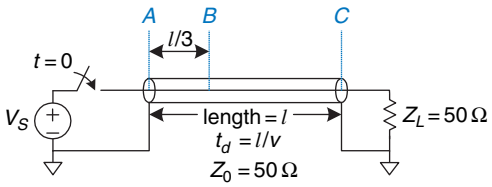


Figure eA.16 Transmission line with matched termination

through a switch that closes at time $t = 0$. The other end is connected to the 50 Ω matched load. This section analyzes the voltages and currents at points A, B, and C—at the beginning of the line, one-third of the length along the line, and at the end of the line, respectively.

Figure eA.17 shows the voltages at points A, B, and C over time. Initially, there is no voltage or current flowing in the transmission line, because the switch is open. At time $t = 0$, the switch closes, and the voltage source launches a wave with voltage $V = V_S$ along the line. This is called the *incident wave*. Because the characteristic impedance is Z_0, the wave has current $I = V_S/Z_0$. The voltage reaches the beginning of the line (point A) immediately, as shown in Figure eA.17(a). The wave propagates along the line at the speed of light. At time $t_d/3$, the wave reaches point B. The voltage at this point abruptly rises from 0 to V_S, as shown in Figure eA.17(b). At time t_d, the incident wave reaches point C at the end of the line, and the voltage rises there too. All of the current, I, flows into the resistor, Z_L, producing a voltage across the resistor of $Z_L I = Z_L (V_S/Z_0) = V_S$ because $Z_L = Z_0$. This voltage is consistent with the wave flowing along the transmission line. Thus, the wave is *absorbed* by the load impedance, and the transmission line reaches its *steady state*.

In steady state, the transmission line behaves like an ideal equipotential wire because it is, after all, just a wire. The voltage at all points along the line must be identical. Figure eA.18 shows the steady-state equivalent model of the circuit in Figure eA.16. The voltage is V_S everywhere along the wire.

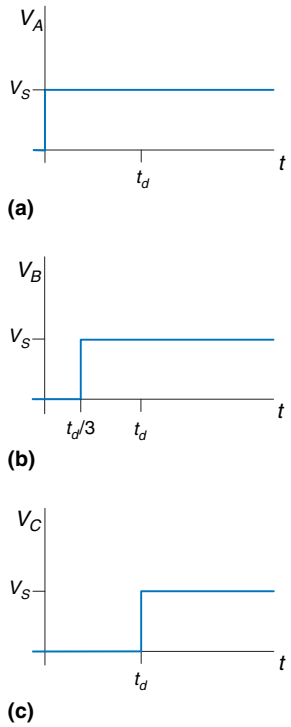


Figure eA.17 Voltage waveforms for Figure eA.16 at points *A*, *B*, and *C*

Example eA.2 TRANSMISSION LINE WITH MATCHED SOURCE AND LOAD TERMINATIONS

Figure eA.19 shows a transmission line with matched source and load impedances Z_S and Z_L. Plot the voltage at nodes A, B, and C versus time. When does the system reach steady-state, and what is the equivalent circuit at steady-state?

Solution: When the voltage source has a source impedance Z_S in series with the transmission line, part of the voltage drops across Z_S, and the remainder propagates down the transmission line. At first, the transmission line behaves as an impedance

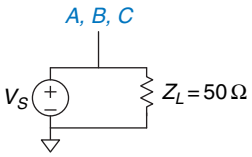


Figure eA.18 Equivalent circuit of Figure eA.16 at steady state

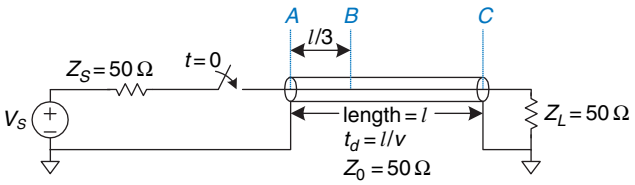


Figure eA.19 **Transmission line with matched source and load impedances**

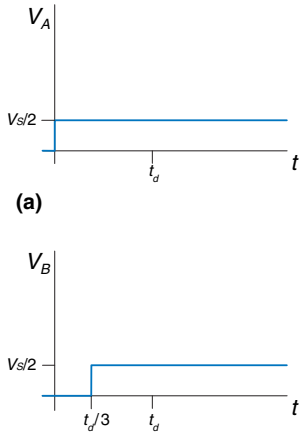


Figure eA.20 **Voltage waveforms for Figure eA.19 at points *A*, *B*, and *C***

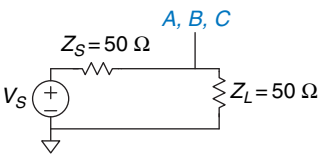


Figure eA.21 **Equivalent circuit of Figure eA.19 at steady state**

Z_0, because the load at the end of the line cannot possibly influence the behavior of the line until a speed of light delay has elapsed. Hence, by the *voltage divider equation*, the incident voltage flowing down the line is

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \tag{A.6}$$

Thus, at $t = 0$, a wave of voltage, $V = \frac{V_S}{2}$, is sent down the line from point A. Again, the signal reaches point B at time $t_d/3$ and point C at t_d, as shown in Figure eA.20. All of the current is absorbed by the load impedance Z_L, so the circuit enters steady-state at $t = t_d$. In steady-state, the entire line is at $V_S/2$, just as the steady-state equivalent circuit in Figure eA.21 would predict.

A.8.2 Open Termination

When the load impedance is not equal to Z_0, the termination cannot absorb all of the current, and some of the wave must be reflected. Figure eA.22 shows a transmission line with an open load termination. No current can flow through an open termination, so the current at point C must always be 0.

The voltage on the line is initially zero. At $t = 0$, the switch closes and a wave of voltage, $V = V_S\frac{Z_0}{Z_0 + Z_S} = \frac{V_S}{2}$, begins propagating down the line. Notice that this initial wave is the same as that of Example eA.2 and is independent of the termination, because the load at the end of the line cannot influence the behavior at the beginning until at least $2t_d$ has elapsed. This wave reaches point B at $t_d/3$ and point C at t_d as shown in Figure eA.23.

When the incident wave reaches point C, it cannot continue forward because the wire is open. It must instead reflect back toward the source. The reflected wave also has voltage $V = \frac{V_S}{2}$, because the open termination reflects the entire wave.

The voltage at any point is the sum of the incident and reflected waves. At time $t = t_d$, the voltage at point C is $V = \frac{V_S}{2} + \frac{V_S}{2} = V_S$. The reflected wave reaches point B at $5t_d/3$ and point A at $2t_d$. When it reaches point A,

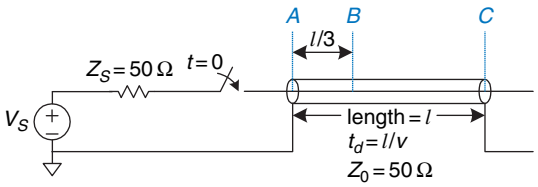


Figure eA.22 Transmission line with open load termination

the wave is absorbed by the source termination impedance that matches the characteristic impedance of the line. Thus, the system reaches steady state at time $t = 2t_d$, and the transmission line becomes equivalent to an equipotential wire with voltage V_S and current $I = 0$.

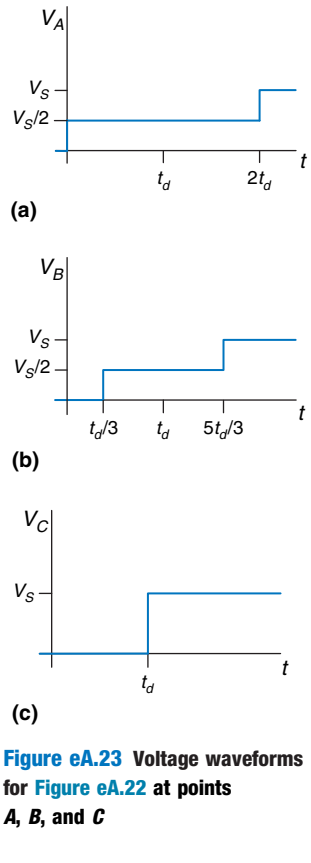


Figure eA.23 Voltage waveforms for Figure eA.22 at points *A*, *B*, and *C*

A.8.3 Short Termination

Figure eA.24 shows a transmission line terminated with a short circuit to ground. Thus, the voltage at point C must always be 0.

As in the previous examples, the voltages on the line are initially 0. When the switch closes, a wave of voltage, $V = \frac{V_S}{2}$, begins propagating down the line (Figure eA.25). When it reaches the end of the line, it must reflect with opposite polarity. The reflected wave, with voltage $V = \frac{-V_S}{2}$, adds to the incident wave, ensuring that the voltage at point C remains 0. The reflected wave reaches the source at time $t = 2t_d$ and is absorbed by the source impedance. At this point, the system reaches steady state, and the transmission line is equivalent to an equipotential wire with voltage $V = 0$.

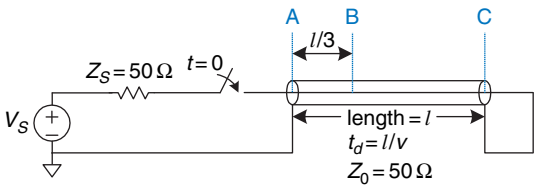


Figure eA.24 Transmission line with short termination

A.8.4 Mismatched Termination

The termination impedance is said to be *mismatched* when it does not equal the characteristic impedance of the line. In general, when an incident wave reaches a mismatched termination, part of the wave is absorbed and part is reflected. The reflection coefficient k_r indicates the fraction of the incident wave V_i that is reflected: $V_r = k_r V_i$.

Section A.8.8 derives the reflection coefficient using conservation of current arguments. It shows that, when an incident wave flowing along

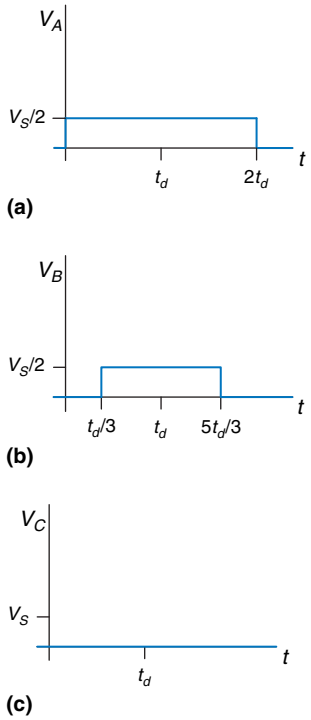


Figure eA.25 Voltage waveforms for Figure eA.24 at points *A*, *B*, and *C*

a transmission line of characteristic impedance Z_0 reaches a termination impedance Z_T at the end of the line, the reflection coefficient is

$$k_r = \frac{Z_T - Z_0}{Z_T + Z_0} \tag{A.7}$$

Note a few special cases. If the termination is an open circuit ($Z_T = \infty$), $k_r = 1$, because the incident wave is entirely reflected (so the current out the end of the line remains zero). If the termination is a short circuit ($Z_T = 0$), $k_r = -1$, because the incident wave is reflected with negative polarity (so the voltage at the end of the line remains zero). If the termination is a matched load ($Z_T = Z_0$), $k_r = 0$, because the incident wave is completely absorbed.

Figure eA.26 illustrates reflections in a transmission line with a *mismatched load termination* of 75 Ω. $Z_T = Z_L = 75$ Ω, and $Z_0 = 50$ Ω, so $k_r = 1/5$. As in previous examples, the voltage on the line is initially 0. When the switch closes, a wave of voltage $V = \frac{V_S}{2}$ propagates down the line, reaching the end at $t = t_d$. When the incident wave reaches the termination at the end of the line, one fifth of the wave is reflected, and the remaining four fifths flows into the load impedance. Thus, the reflected wave has a voltage $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$. The total voltage at point C is the sum of the incoming and reflected voltages, $V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$. At $t = 2t_d$, the reflected wave reaches point *A*, where it is absorbed by the matched 50 Ω termination, Z_S. Figure eA.27 plots the voltages and currents along the line. Again, note that, in steady state (in this case at time $t > 2t_d$), the transmission line is equivalent to an equipotential wire, as shown in Figure eA.28. At steady state, the system acts like a voltage divider, so

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = V_S\left(\frac{75\Omega}{75\Omega + 50\Omega}\right) = \frac{3V_S}{5}$$

Reflections can occur at both ends of the transmission line. Figure eA.29 shows a transmission line with a source impedance, Z_S, of 450 Ω and an open termination at the load. The reflection coefficients at the load and source, k_{rL} and k_{rS}, are 1 and 4/5, respectively. In this case, waves reflect off both ends of the transmission line until a steady state is reached.

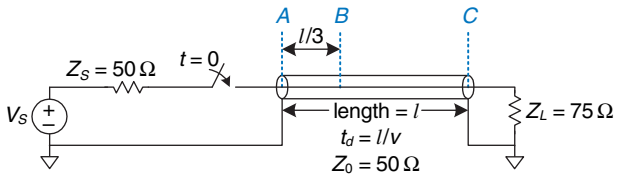


Figure eA.26 Transmission line with mismatched termination

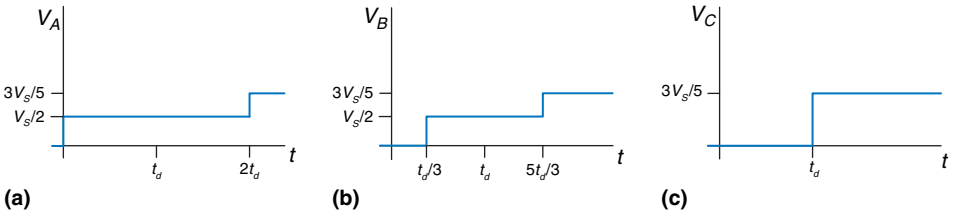


Figure eA.27 Voltage waveforms for Figure eA.26 at points *A*, *B*, and *C*

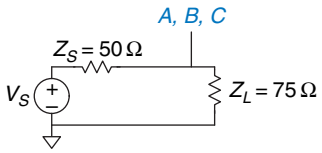


Figure eA.28 Equivalent circuit of Figure eA.26 at steady state

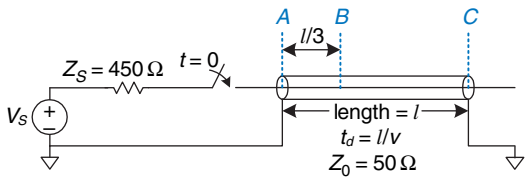


Figure eA.29 Transmission line with mismatched source and load terminations

The *bounce diagram* shown in Figure eA.30 helps visualize reflections off both ends of the transmission line. The horizontal axis represents distance along the transmission line, and the vertical axis represents time, increasing downward. The two sides of the bounce diagram represent the source and load ends of the transmission line, points *A* and *C*. The incoming and reflected signal waves are drawn as diagonal lines between points *A* and *C*. At time $t = 0$, the source impedance and transmission line behave as a voltage divider, launching a voltage wave of $\frac{V_S}{10}$ from point *A* toward point *C*. At time $t = t_d$, the signal reaches point *C* and is completely reflected ($k_{rL} = 1$). At time $t = 2t_d$, the reflected wave of $\frac{V_S}{10}$ reaches point *A* and is reflected with a reflection coefficient, $k_{rS} = 4/5$, to produce a wave of $\frac{2V_S}{25}$ traveling toward point *C*, and so forth.

The voltage at a given time at any point on the transmission line is the sum of all the incident and reflected waves. Thus, at time $t = 1.1t_d$, the voltage at point *C* is $\frac{V_S}{10} + \frac{V_S}{10} = \frac{V_S}{5}$. At time $t = 3.1t_d$, the voltage at point *C* is $\frac{V_S}{10} + \frac{V_S}{10} + \frac{2V_S}{25} + \frac{2V_S}{25} = \frac{9V_S}{25}$, and so forth. Figure eA.31 plots the voltages

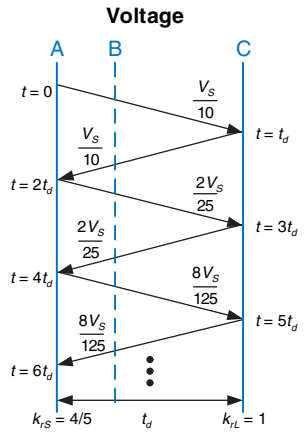


Figure eA.30 Bounce diagram for Figure eA.29

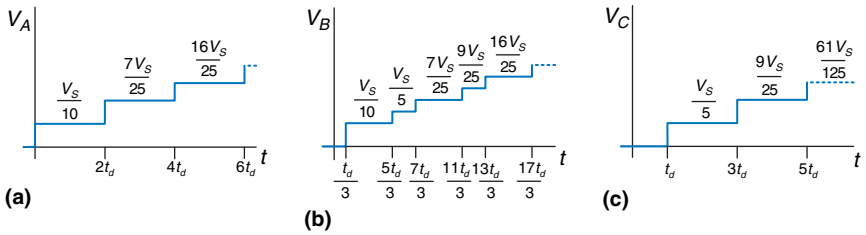


Figure eA.31 **Voltage and current waveforms for Figure eA.29**

against time. As t approaches infinity, the voltages approach steady state with $V_A = V_B = V_C = V_S$.

A.8.5 When to Use Transmission Line Models

Transmission line models for wires are needed whenever the wire delay, t_d, is longer than a fraction (e.g., 20%) of the edge rates (rise or fall times) of a signal. If the wire delay is shorter, it has an insignificant effect on the propagation delay of the signal, and the reflections dissipate while the signal is transitioning. If the wire delay is longer, it must be considered in order to accurately predict the propagation delay and waveform of the signal. In particular, reflections may distort the digital characteristic of a waveform, resulting in incorrect logic operations.

Recall that signals travel on a PCB at about 15 cm/ns. For TTL logic, with edge rates of 10 ns, wires must be modeled as transmission lines only if they are longer than 30 cm (10 ns × 15 cm/ns × 20%). PCB traces are usually less than 30 cm, so most traces can be modeled as ideal equipotential wires. In contrast, many modern chips have edge rates of 2 ns or less, so traces longer than about 6 cm (about 2.5 inches) must be modeled as transmission lines. Clearly, use of edge rates that are crisper than necessary just causes difficulties for the designer.

Breadboards lack a ground plane, so the electromagnetic fields of each signal are nonuniform and difficult to model. Moreover, the fields interact with other signals. This can cause strange reflections and crosstalk between signals. Thus, breadboards are unreliable above a few megahertz.

In contrast, PCBs have good transmission lines with consistent characteristic impedance and velocity along the entire line. As long as they are terminated with a source or load impedance that is matched to the impedance of the line, PCB traces do not suffer from reflections.

A.8.6 Proper Transmission Line Terminations

There are two common ways to properly terminate a transmission line, shown in Figure eA.32. In *parallel termination*, the driver has a low impedance ($Z_S \approx 0$). A load resistor Z_L with impedance Z_0 is placed in parallel

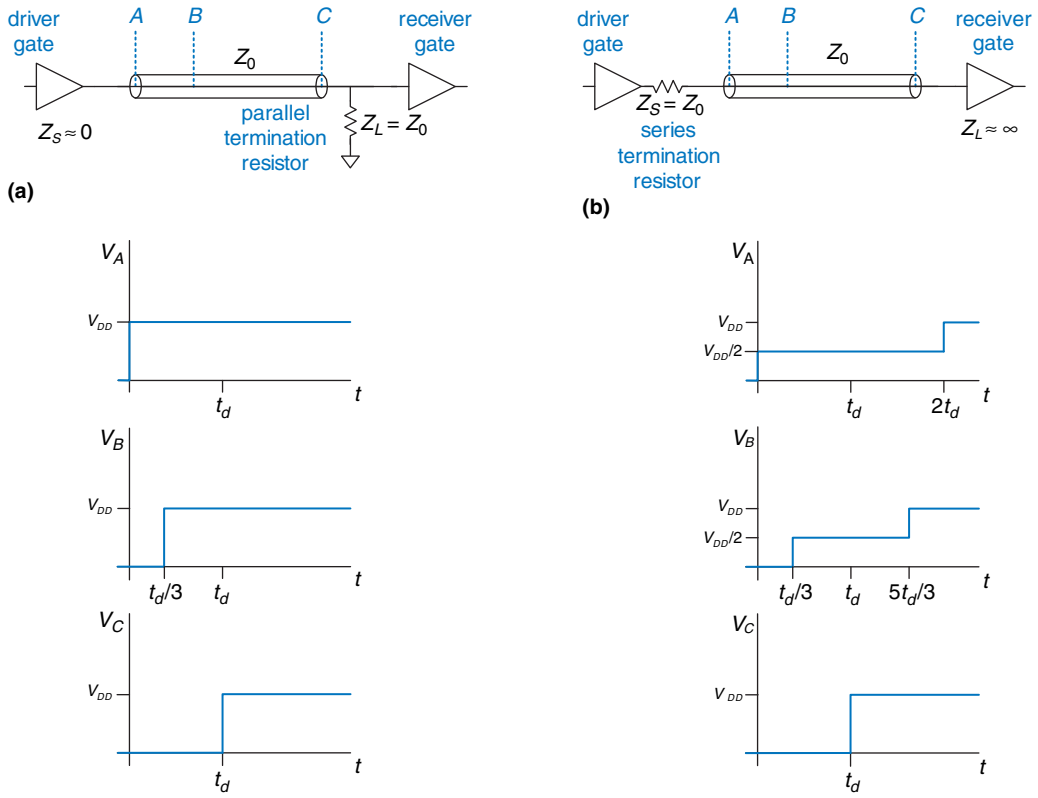


Figure eA.32 Termination schemes: (a) parallel, (b) series

with the load (between the input of the receiver gate and ground). When the driver switches from 0 to V_{DD}, it sends a wave with voltage V_{DD} down the line. The wave is absorbed by the matched load termination, and no reflections take place. In *series termination*, a source resistor Z_S is placed in series with the driver to raise the source impedance to Z_0. The load has a high impedance ($Z_L \approx \infty$). When the driver switches, it sends a wave with voltage $V_{DD}/2$ down the line. The wave reflects at the open circuit load and returns, bringing the voltage on the line up to V_{DD}. The wave is absorbed at the source termination. Both schemes are similar in that the voltage at the receiver transitions from 0 to V_{DD} at $t = t_d$, just as one would desire. They differ in power consumption and in the waveforms that appear elsewhere along the line. Parallel termination dissipates power continuously through the load resistor when the line is at a high voltage. Series termination dissipates no DC power, because the load is an open circuit. However, in series terminated lines, points near the middle of the transmission line initially see a voltage of $V_{DD}/2$,

until the reflection returns. If other gates are attached to the middle of the line, they will momentarily see an illegal logic level. Therefore, series termination works best for *point-to-point* communication with a single driver and a single receiver. Parallel termination is better for a *bus* with multiple receivers, because receivers at the middle of the line never see an illegal logic level.

A.8.7 Derivation of Z_0*

Z_0 is the ratio of voltage to current in a wave propagating along a transmission line. This section derives Z_0; it assumes some previous knowledge of resistor-inductor-capacitor (RLC) circuit analysis.

Imagine applying a step voltage to the input of a semi-infinite transmission line (so that there are no reflections). Figure eA.33 shows the semi-infinite line and a model of a segment of the line of length dx. R, L, and C, are the values of resistance, inductance, and capacitance per unit length. Figure eA.33(b) shows the transmission line model with a resistive component, R. This is called a *lossy* transmission line model, because energy is dissipated, or lost, in the resistance of the wire. However, this loss is often negligible, and we can simplify analysis by ignoring the resistive component and treating the transmission line as an *ideal* transmission line, as shown in Figure eA.33(c).

Voltage and current are functions of time and space throughout the transmission line, as given by Equations eA.8 and eA.9.

$$\frac{\partial}{\partial x} V(x, t) = L\frac{\partial}{\partial t} I(x, t) \tag{A.8}$$

$$\frac{\partial}{\partial x} I(x, t) = C\frac{\partial}{\partial t} V(x, t) \tag{A.9}$$

Taking the space derivative of Equation eA.8 and the time derivative of Equation eA.9 and substituting gives Equation eA.10, the *wave equation*.

$$\frac{\partial^2}{\partial x^2} V(x, t) = LC\frac{\partial^2}{\partial t^2} V(x, t) \tag{A.10}$$

Z_0 is the ratio of voltage to current in the transmission line, as illustrated in Figure eA.34(a). Z_0 must be independent of the length of the line, because the behavior of the wave cannot depend on things at a distance. Because it is independent of length, the impedance must still equal Z_0 after the addition of a small amount of transmission line, dx, as shown in Figure eA.34(b).

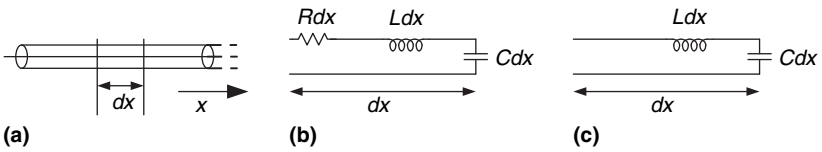


Figure eA.33 Transmission line models: (a) semi-infinite cable, (b) lossy, (c) ideal

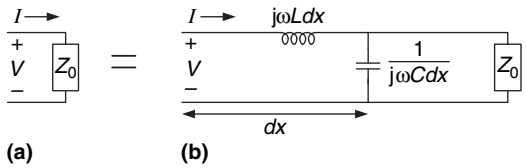


Figure eA.34 Transmission line model: (a) for entire line and (b) with additional length, *dx*

Using the impedances of an inductor and a capacitor, we rewrite the relationship of Figure eA.34 in equation form:

$$Z_0 = j\omega L dx + [Z_0 || (1/(j\omega C dx))] \tag{A.11}$$

Rearranging, we get

$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 L C dx = 0 \tag{A.12}$$

Taking the limit as dx approaches 0, the last term vanishes and we find that

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.13}$$

A.8.8 Derivation of the Reflection Coefficient*

The reflection coefficient k_r is derived using conservation of current. Figure eA.35 shows a transmission line with characteristic impedance Z_0 and load impedance Z_L. Imagine an incident wave of voltage V_i and current I_i. When the wave reaches the termination, some current I_L flows through the load impedance, causing a voltage drop V_L. The remainder of the current reflects back down the line in a wave of voltage V_r and current I_r. Z_0 is the ratio of voltage to current in waves propagating along the line, so $\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$.

The voltage on the line is the sum of the voltages of the incident and reflected waves. The current flowing in the positive direction on the line is the difference between the currents of the incident and reflected waves.

$$V_L = V_i + V_r \tag{A.14}$$

$$I_L = I_i - I_r \tag{A.15}$$

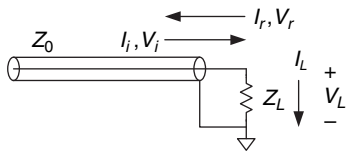


Figure eA.35 Transmission line showing incoming, reflected, and load voltages and currents

Using Ohm's law and substituting for I_L, I_i, and I_r in Equation eA.15, we get

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{A.16}$$

Rearranging, we solve for the reflection coefficient, k_r:

$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \tag{A.17}$$

A.8.9 Putting It All Together

Transmission lines model the fact that signals take time to propagate down long wires because the speed of light is finite. An ideal transmission line has uniform inductance L and capacitance C per unit length and zero resistance. The transmission line is characterized by its characteristic impedance Z_0 and delay t_d which can be derived from the inductance, capacitance, and wire length. The transmission line has significant delay and noise effects on signals whose rise/fall times are less than about $5t_d$. This means that, for systems with 2 ns rise/fall times, PCB traces longer than about 6 cm must be analyzed as transmission lines to accurately understand their behavior.

A digital system consisting of a gate driving a long wire attached to the input of a second gate can be modeled with a transmission line as shown in Figure eA.36. The voltage source, source impedance Z_S, and switch model the first gate switching from 0 to 1 at time 0. The driver gate cannot supply infinite current; this is modeled by Z_S. Z_S is usually small for a logic gate, but a designer may choose to add a resistor in series with the gate to raise Z_S and match the impedance of the line. The input to the second gate is modeled as Z_L. CMOS circuits usually have little input current, so Z_L may be close to infinity. The designer may also choose to add a resistor in parallel with the second gate, between the gate input and ground, so that Z_L matches the impedance of the line.

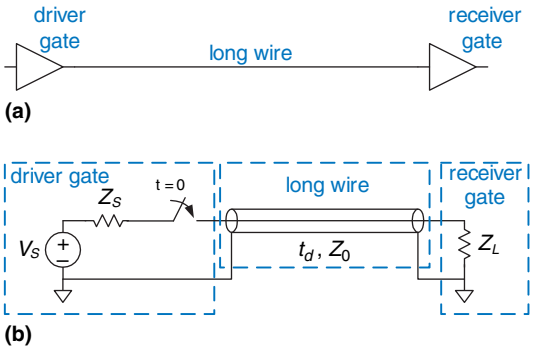


Figure eA.36 Digital system modeled with transmission line

When the first gate switches, a wave of voltage is driven onto the transmission line. The source impedance and transmission line form a voltage divider, so the voltage of the incident wave is

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{A.18}$$

At time t_d, the wave reaches the end of the line. Part is absorbed by the load impedance, and part is reflected. The reflection coefficient k_r indicates the portion that is reflected: $k_r = V_r/V_i$, where V_r is the voltage of the reflected wave and V_i is the voltage of the incident wave.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{A.19}$$

The reflected wave adds to the voltage already on the line. It reaches the source at time $2t_d$, where part is absorbed and part is again reflected. The reflections continue back and forth, and the voltage on the line eventually approaches the value that would be expected if the line were a simple equipotential wire.

A.9 ECONOMICS

Although digital design is so much fun that some of us would do it for free, most designers and companies intend to make money. Therefore, economic considerations are a major factor in design decisions.

The cost of a digital system can be divided into *nonrecurring engineering costs* (*NRE*), and *recurring costs*. NRE accounts for the cost of designing the system. It includes the salaries of the design team, computer and software costs, and the costs of producing the first working unit. The fully loaded cost of a designer in the United States in 2015 (including salary, health insurance, retirement plan, and a computer with design tools) was roughly $200,000 per year, so design costs can be significant. Recurring costs are the cost of each additional unit; this includes components, manufacturing, marketing, technical support, and shipping.

The sales price must cover not only the cost of the system but also other costs such as office rental, taxes, and salaries of staff who do not directly contribute to the design (such as the janitor and the CEO). After all of these expenses, the company should still make a profit.

Example eA.3 BEN TRIES TO MAKE SOME MONEY

Ben Bitdiddle has designed a crafty circuit for counting raindrops. He decides to sell the device and try to make some money, but he needs help deciding what implementation to use. He decides to use either an FPGA or an ASIC. The

development kit to design and test the FPGA costs $1500. Each FPGA costs $17. The ASIC costs $600,000 for a mask set and $4 per chip.

Regardless of what chip implementation he chooses, Ben needs to mount the packaged chip on a printed circuit board (PCB), which will cost him $1.50 per board. He thinks he can sell 1000 devices per month. Ben has coerced a team of bright undergraduates into designing the chip for their senior project, so it doesn't cost him anything to design.

If the sales price has to be twice the cost (100% profit margin), and the product life is 2 years, which implementation is the better choice?

Solution: Ben figures out the total cost for each implementation over 2 years, as shown in Table eA.4. Over 2 years, Ben plans on selling 24,000 devices, and the total cost is given in Table eA.4 for each option. If the product life is only two years, the FPGA option is clearly superior. The per-unit cost is $445,500/24,000 = $18.56, and the sales price is $37.13 per unit to give a 100% profit margin. The ASIC option would have cost $732,000/24,000 = $30.50 and would have sold for $61 per unit.

Table eA.4 ASIC vs FPGA costs

Cost	ASIC	FPGA
NRE	$600,000	$1500
chip	$4	$17
PCB	$1.50	$1.50
TOTAL	$600,000 + (24,000 × $5.50) = **$732,000**	$1500 + (24,000 × $18.50) = **$445,500**
per unit	$30.50	$18.56

Example eA.4 BEN GETS GREEDY

After seeing the marketing ads for his product, Ben thinks he can sell even more chips per month than originally expected. If he were to choose the ASIC option, how many devices per month would he have to sell to make the ASIC option more profitable than the FPGA option?

Solution: Ben solves for the minimum number of units, N, that he would need to sell in 2 years:

$$\$600,000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

Solving the equation gives $N = 46{,}039$ units, or 1919 units per month. He would need to almost double his monthly sales to benefit from the ASIC solution.

Example eA.5 BEN GETS LESS GREEDY

Ben realizes that his eyes have gotten too big for his stomach, and he doesn't think he can sell more than 1000 devices per month. But he does think the product life can be longer than 2 years. At a sales volume of 1000 devices per month, how long would the product life have to be to make the ASIC option worthwhile?

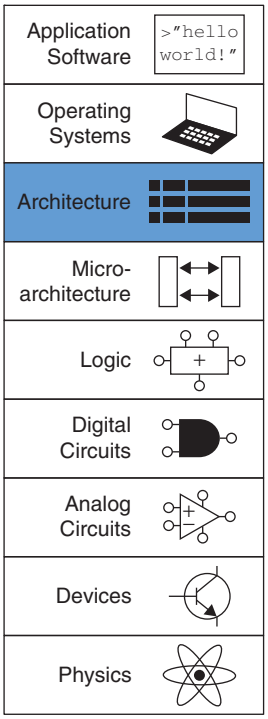
Solution: If Ben sells more than 46,039 units in total, the ASIC option is the best choice. So, Ben would need to sell at a volume of 1000 per month for at least 47 months (rounding up), which is almost 4 years. By then, his product is likely to be obsolete.

Chips are usually purchased from a distributor rather than directly from the manufacturer (unless you are ordering tens of thousands of units). Digikey (`www.digikey.com`) is a leading distributor that sells a wide variety of electronics. Jameco (`www.jameco.com`) and All Electronics (`www.allelectronics.com`) have eclectic catalogs that are competitively priced and well suited to hobbyists.

ARM Instructions

B

This appendix summarizes ARMv4 instructions used in this book. Condition encodings are given in Table 6.3.



B.1 DATA-PROCESSING INSTRUCTIONS

Standard data-processing instructions use the encoding in Figure B.1. The 4-bit *cmd* field specifies the type of instruction as given in Table B.1. When the *S*-bit is 1, the status register is updated with the condition flags produced by the instruction. The *I*-bit and bits 4 and 7 specify one of three encodings for the second source operand, *Src2*, as described in Section 6.4.2. The *cond* field specifies which condition codes to check, as given in Section 6.3.2.

Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-800056-4.00017-0

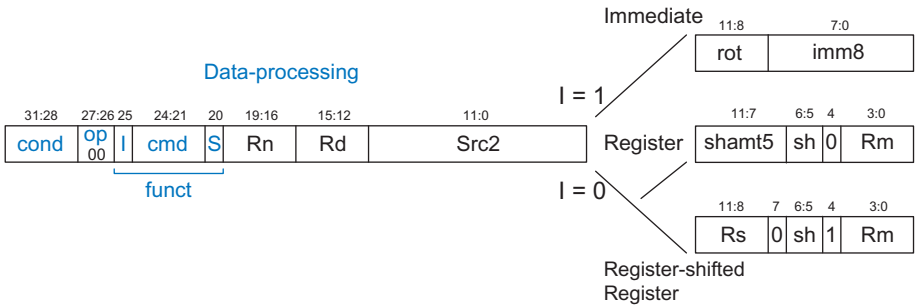


Figure B.1 Data-processing instruction encodings

Table B.1 Data-processing instructions

cmd	Name	Description	Operation
0000	`AND Rd, Rn, Src2`	Bitwise AND	`Rd ← Rn & Src2`
0001	`EOR Rd, Rn, Src2`	Bitwise XOR	`Rd ← Rn ^ Src2`
0010	`SUB Rd, Rn, Src2`	Subtract	`Rd ← Rn - Src2`
0011	`RSB Rd, Rn, Src2`	Reverse Subtract	`Rd ← Src2 - Rn`
0100	`ADD Rd, Rn, Src2`	Add	`Rd ← Rn+Src2`
0101	`ADC Rd, Rn, Src2`	Add with Carry	`Rd ← Rn+Src2+C`
0110	`SBC Rd, Rn, Src2`	Subtract with Carry	`Rd ← Rn - Src2 -` $\overline{\texttt{C}}$
0111	`RSC Rd, Rn, Src2`	Reverse Sub w/ Carry	`Rd ← Src2 - Rn -` $\overline{\texttt{C}}$
1000 ($S = 1$)	`TST Rd, Rn, Src2`	Test	`Set flags based on Rn & Src2`
1001 ($S = 1$)	`TEQ Rd, Rn, Src2`	Test Equivalence	`Set flags based on Rn ^ Src2`
1010 ($S = 1$)	`CMP Rn, Src2`	Compare	`Set flags based on Rn - Src2`
1011 ($S = 1$)	`CMN Rn, Src2`	Compare Negative	`Set flags based on Rn+Src2`
1100	`ORR Rd, Rn, Src2`	Bitwise OR	`Rd ← Rn | Src2`
1101	Shifts:		
$I = 1$ OR ($\text{instr}_{11:4} = 0$)	`MOV Rd, Src2`	Move	`Rd ← Src2`
$I = 0$ AND ($sh = 00$; $\text{instr}_{11:4} \neq 0$)	`LSL Rd, Rm, Rs/shamt5`	Logical Shift Left	`Rd ← Rm << Src2`
$I = 0$ AND ($sh = 01$)	`LSR Rd, Rm, Rs/shamt5`	Logical Shift Right	`Rd ← Rm >> Src2`

(*continued*)

Table B.1 Data-processing instructions—Cont'd

cmd	Name	Description	Operation
$I = 0$ AND ($sh = 10$)	`ASR Rd, Rm, Rs/shamt5`	Arithmetic Shift Right	`Rd ← Rm>>>Src2`
$I = 0$ AND ($sh = 11$; $\text{instr}_{11:7,\ 4} = 0$)	`RRX Rd, Rm, Rs/shamt5`	Rotate Right Extend	`{Rd, C} ← {C, Rd}`
$I = 0$ AND ($sh = 11$; $\text{instr}_{11:7} \neq 0$)	`ROR Rd, Rm, Rs/shamt5`	Rotate Right	`Rd ← Rn ror Src2`
1110	`BIC Rd, Rn, Src2`	Bitwise Clear	`Rd ← Rn & ~Src2`
1111	`MVN Rd, Rn, Src2`	Bitwise NOT	`Rd ← ~Rn`

`NOP` (no operation) is typically encoded as 0xE1A000, which is equivalent to `MOV R0, R0`.

B.1.1 Multiply Instructions

Multiply instructions use the encoding in Figure B.2 The 3-bit *cmd* field specifies the type of multiply, as given in Table B.2.

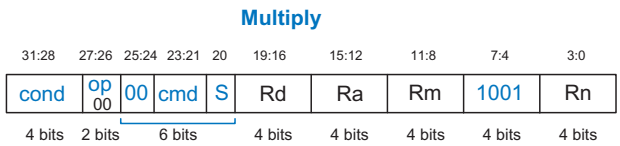


Figure B.2 Multiply instruction encoding

Table B.2 Multiply instructions

cmd	Name	Description	Operation
000	`MUL Rd, Rn, Rm`	Multiply	`Rd ← Rn × Rm (low 32 bits)`
001	`MLA Rd, Rn, Rm, Ra`	Multiply Accumulate	`Rd ← (Rn × Rm)+Ra (low 32 bits)`
100	`UMULL Rd, Rn, Rm, Ra`	Unsigned Multiply Long	`{Rd, Ra} ← Rn × Rm (all 64 bits, Rm/Rn unsigned)`
101	`UMLAL Rd, Rn, Rm, Ra`	Unsigned Multiply Accumulate Long	`{Rd, Ra} ← (Rn × Rm)+{Rd, Ra} (all 64 bits, Rm/Rn unsigned)`
110	`SMULL Rd, Rn, Rm, Ra`	Signed Multiply Long	`{Rd, Ra} ← Rn × Rm (all 64 bits, Rm/Rn signed)`
111	`SMLAL Rd, Rn, Rm, Ra`	Signed Multiply Accumulate Long	`{Rd, Ra} ← (Rn × Rm)+{Rd, Ra} (all 64 bits, Rm/Rn signed)`

B.2 MEMORY INSTRUCTIONS

The most common memory instructions (LDR, STR, LDRB, and STRB) operate on words or bytes and are encoded with *op* = 01. Extra memory instructions operating on halfwords or signed bytes are encoded with *op* = 00 and have less flexibility generating *Src2*. The immediate offset is only 8 bits and the register offset cannot be shifted. LDRB and LDRH zero-extend the bits to fill a word, while LDRSB and LDRSH sign-extend the bits. Also see memory indexing modes in Section 6.3.6.

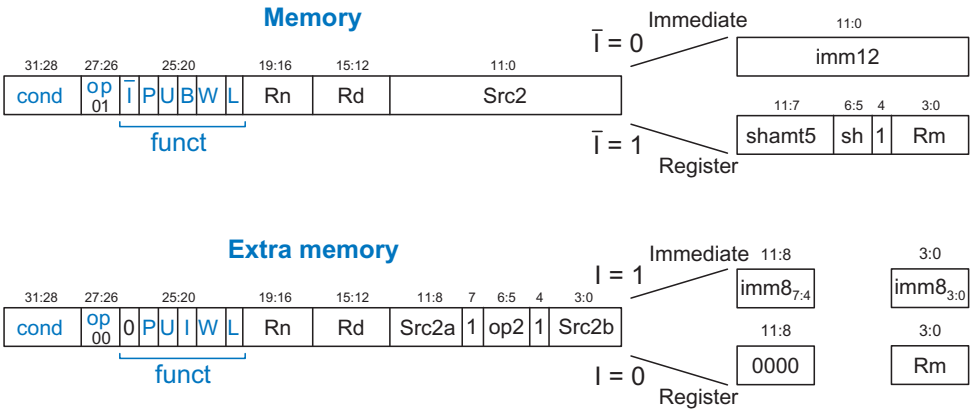


Figure B.3 Memory instruction encodings

Table B.3 Memory instructions

op	B	op2	L	Name	Description	Operation
01	0	N/A	0	STR Rd, [Rn, ±Src2]	Store Register	Mem[Adr] ← Rd
01	0	N/A	1	LDR Rd, [Rn, ±Src2]	Load Register	Rd ← Mem[Adr]
01	1	N/A	0	STRB Rd, [Rn, ±Src2]	Store Byte	Mem[Adr] ← $Rd_{7:0}$
01	1	N/A	1	LDRB Rd, [Rn, ±Src2]	Load Byte	Rd ← $Mem[Adr]_{7:0}$
00	N/A	01	0	STRH Rd, [Rn, ±Src2]	Store Halfword	Mem[Adr] ← $Rd_{15:0}$
00	N/A	01	1	LDRH Rd, [Rn, ±Src2]	Load Halfword	Rd ← $Mem[Adr]_{15:0}$
00	N/A	10	1	LDRSB Rd, [Rn, ±Src2]	Load Signed Byte	Rd ← $Mem[Adr]_{7:0}$
00	N/A	11	1	LDRSH Rd, [Rn, ±Src2]	Load Signed Half	Rd ← $Mem[Adr]_{15:0}$

B.3 BRANCH INSTRUCTIONS

Figure B.4 shows the encoding for branch instructions (B and BL) and Table B.4 describes their operation.

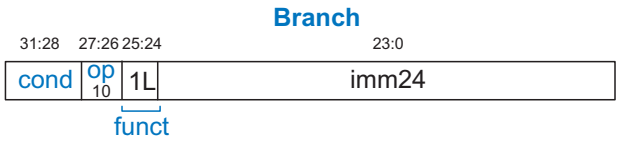


Figure B.4 Branch instruction encoding

Table B.4 Branch instructions

L	Name	Description	Operation
0	B label	Branch	PC ← (PC+8)+imm24 << 2
1	BL label	Branch with Link	LR ← (PC+8) - 4; PC ← (PC+8)+imm24 << 2

B.4 MISCELLANEOUS INSTRUCTIONS

The ARMv4 instruction set includes the following miscellaneous instructions. Consult the ARM Architecture Reference Manual for details.

Instructions	Description	Purpose
LDM, STM	Load/store multiple	Save and recall registers in subroutine calls
SWP / SWPB	Swap (byte)	Atomic load and store for process synchronization
LDRT, LDRBT, STRT, STRBT	Load/store word/byte with translation	Allow operating system to access memory in user virtual memory space
SWI[1]	Software Interrupt	Create an exception, often used to call the operating system
CDP, LDC, MCR, MRC, STC	Coprocessor access	Communicate with optional coprocessor
MRS, MSR	Move from/to status register	Save status register during exceptions

[1] SWI was renamed SVC (supervisor call) in ARMv7.

B.5 CONDITION FLAGS

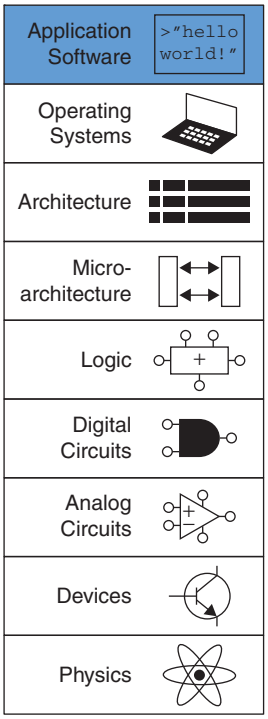
Condition flags are changed by data-processing instructions with $S = 1$ in the machine code. All instructions except `CMP`, `CMN`, `TEQ`, and `TST` must have an "`S`" appended to the instruction mnemonic to make $S = 1$. Table B.5 shows which condition flags are affected by each instruction.

Table B.5 Instructions that affect condition flags

Type	Instructions	Condition Flags
Add	`ADDS, ADCS`	*N*, *Z*, *C*, *V*
Subtract	`SUBS, SBCS, RSBS, RSCS`	*N*, *Z*, *C*, *V*
Compare	`CMP, CMN`	*N*, *Z*, *C*, *V*
Shifts	`ASRS, LSLS, LSRS, RORS, RRXS`	*N*, *Z*, *C*
Logical	`ANDS, ORRS, EORS, BICS`	*N*, *Z*, *C*
Test	`TEQ, TST`	*N*, *Z*, *C*
Move	`MOVS, MVNS`	*N*, *Z*, *C*
Multiply	`MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS`	*N*, *Z*

C Programming

C



C.1 INTRODUCTION

The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This Appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the Appendix so that readers may easily cover or skip it depending on previous experience.

The rest of this chapter is available online as a downloadable PDF from the book's companion site: http://booksite.elsevier.com/9780128000564.

C Programming

eC

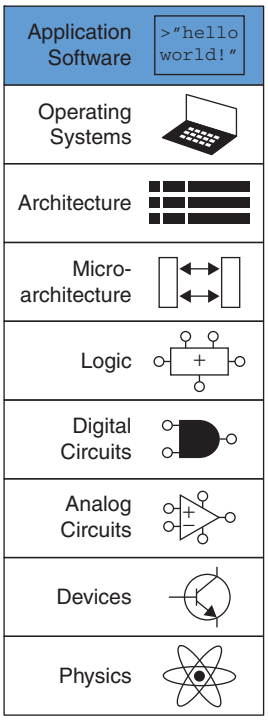
C.1 INTRODUCTION

The overall goal of this book is to give a picture of how computers work on many levels, from the transistors by which they are constructed all the way up to the software they run. The first five chapters of this book work up through the lower levels of abstraction, from transistors to gates to logic design. Chapters 6 through 8 jump up to architecture and work back down to microarchitecture to connect the hardware with the software. This Appendix on C programming fits logically between Chapters 5 and 6, covering C programming as the highest level of abstraction in the text. It motivates the architecture material and links this book to programming experience that may already be familiar to the reader. This material is placed in the Appendix so that readers may easily cover or skip it depending on previous experience.

Programmers use many different languages to tell a computer what to do. Fundamentally, computers process instructions in *machine language* consisting of 1's and 0's, as is explored in Chapter 6. But programming in machine language is tedious and slow, leading programmers to use more abstract languages to get their meaning across more efficiently. Table eC.1 lists some examples of languages at various levels of abstraction.

One of the most popular programming languages ever developed is called C. It was created by a group including Dennis Ritchie and Brian Kernighan at Bell Laboratories between 1969 and 1973 to rewrite the UNIX operating system from its original assembly language. By many measures, C (including a family of closely related languages such as C++, C#, and Objective C) is the most widely used language in existence. Its popularity stems from a number of factors including its:

- Availability on a tremendous variety of platforms, from supercomputers down to embedded microcontrollers
- Relative ease of use, with a huge user base



Digital Design and Computer Architecture, Second Edition. DOI: 10.1016/B978-0-12-394424-5.00017-3

Dennis Ritchie, 1941–2011

Brian Kernighan, 1942–

C was formally introduced in 1978 by Brian Kernighan and Dennis Ritchie's classic book, *The C Programming Language*. In 1989, the American National Standards Institute (ANSI) expanded and standardized the language, which became known as ANSI C, Standard C, or C89. Shortly thereafter, in 1990, this standard was adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO/IEC updated the standard in 1999 to what is called C99, which we will be discussing in this text.

Table eC.1 Languages at roughly decreasing levels of abstraction

Language	Description
Matlab	Designed to facilitate heavy use of math functions
Perl	Designed for scripting
Python	Designed to emphasize code readability
Java	Designed to run securely on any computer
C	Designed for flexibility and overall system access, including device drivers
Assembly Language	Human-readable machine language
Machine Language	Binary representation of a program

- Moderate level of abstraction providing higher productivity than assembly language, yet giving the programmer a good understanding of how the code will be executed
- Suitability for generating high performance programs
- Ability to interact directly with the hardware

This chapter is devoted to C programming for a variety of reasons. Most importantly, C allows the programmer to directly access addresses in memory, illustrating the connection between hardware and software emphasized in this book. C is a practical language that all engineers and computer scientists should know. Its uses in many aspects of implementation and design – e.g., software development, embedded systems programming, and simulation – make proficiency in C a vital and marketable skill.

The following sections describe the overall syntax of a C program, discussing each part of the program including the header, function and variable declarations, data types, and commonly used functions provided in libraries. Chapter 9 (available as a web supplement, see Preface) describes a hands-on application by using C to program an ARM-based Raspberry Pi computer.

SUMMARY

- **High-level programming:** High-level programming is useful at many levels of design, from writing analysis or simulation software to programming microcontrollers that interact with hardware.
- **Low-level access:** C code is powerful because, in addition to high-level constructs, it provides access to low-level hardware and memory.

C.2 WELCOME TO C

A C program is a text file that describes operations for the computer to perform. The text file is *compiled*, converted into a machine-readable format, and run or *executed* on a computer. C Code Example eC.1 is a simple C program that prints the phrase "Hello world!" to the *console*, the computer screen. C programs are generally contained in one or more text files that end in ".c". Good programming style requires a file name that indicates the contents of the program – for example, this file could be called `hello.c`.

C is the language used to program such ubiquitous systems as Linux, Windows, and iOS. C is a powerful language because of its direct access to hardware. As compared with other high level languages, for example Perl and Matlab, C does not have as much built-in support for specialized operations such as file manipulation, pattern matching, matrix manipulation, and graphical user interfaces. It also lacks features to protect the programmer from common mistakes, such as writing data past the end of an array. Its power combined with its lack of protection has assisted hackers who exploit poorly written software to break into computer systems.

C Code Example eC.1 SIMPLE C PROGRAM

```
// Write "Hello world!" to the console
#include <stdio.h>

int main(void){
  printf("Hello world!\n");
}
```

Console Output

```
Hello world!
```

C.2.1 C Program Dissection

In general, a C program is organized into one or more functions. Every program must include the `main` function, which is where the program starts executing. Most programs use other functions defined elsewhere in the C code and/or in a library. The overall sections of the `hello.c` program are the header, the `main` function, and the body.

Header: `#include <stdio.h>`
The header includes the *library functions* needed by the program. In this case, the program uses the `printf` function, which is part of the standard I/O library, `stdio.h`. See Section C.9 for further details on C's built-in libraries.

Main function: `int main(void)`
All C programs must include exactly one `main` function. Execution of the program occurs by running the code inside `main`, called the *body* of `main`. Function syntax is described in Section C.6. The body of a function contains a sequence of *statements*. Each statement ends with a semicolon. `int` denotes that the `main` function outputs, or *returns*, an integer result that indicates whether the program ran successfully.

While this chapter provides a fundamental understanding of C programming, entire texts are written that describe C in depth. One of our favorites is the classic text *The C Programming Language* by Brian Kernighan and Dennis Ritchie, the developers of C. This text gives a concise description of the nuts and bolts of C. Another good text is *A Book on C* by Al Kelley and Ira Pohl.

Body: `printf("Hello world!\n");`
The body of this `main` function contains one statement, a call to the `printf` function, which prints the phrase "Hello world!" followed by a newline character indicated by the special sequence `"\n"`. Further details about I/O functions are described in Section C.9.1.

All programs follow the general format of the simple `hello.c` program. Of course, very complex programs may contain millions of lines of code and span hundreds of files.

C.2.2 Running a C Program

C programs can be run on many different machines. This *portability* is another advantage of C. The program is first compiled on the desired machine using the C *compiler*. Slightly different versions of the C compiler exist, including *cc* (C compiler), or *gcc* (GNU C compiler). Here we show how to compile and run a C program using gcc, which is freely available for download. It runs directly on Linux machines and is accessible under the Cygwin environment on Windows machines. It is also available for many embedded systems such as the ARM-based Raspberry Pi. The general process described below of C file creation, compilation, and execution is the same for any C program.

1. Create the text file, for example `hello.c`.
2. In a terminal window, change to the directory that contains the file `hello.c` and type `gcc hello.c` at the command prompt.
3. The compiler creates an executable file. By default, the executable is called `a.out` (or `a.exe` on Windows machines).
4. At the command prompt, type `./a.out` (or `./a.exe` on Windows) and press return.
5. "Hello world!" will appear on the screen.

SUMMARY

- `filename.c`: C program files are typically named with a .c extension.
- `main`: Each C program must have exactly one `main` function.
- `#include`: Most C programs use functions provided by built-in libraries. These functions are used by writing `#include <library.h>` at the top of the C file.
- `gcc filename.c`: C files are converted into an executable using a compiler such as the GNU compiler (gcc) or the C compiler (cc).
- **Execution:** After compilation, C programs are executed by typing `./a.out` (or `./a.exe`) at the command line prompt.

C.3 COMPILATION

A compiler is a piece of software that reads a program in a high-level language and converts it into a file of machine code called an executable. Entire textbooks are written on compilers, but we describe them here briefly. The overall operation of the compiler is to (1) preprocess the file by including referenced libraries and expanding macro definitions, (2) ignore all unnecessary information such as comments, (3) translate the high-level code into simple instructions native to the processor that are represented in binary, called machine language, and (4) compile all the instructions into a single binary executable that can be read and executed by the computer. Each machine language is specific to a given processor, so a program must be compiled specifically for the system on which it will run. For example, the ARM machine language is covered in Chapter 6 in detail.

C.3.1 Comments

Programmers use comments to describe code at a high-level and clarify code function. Anyone who has read uncommented code can attest to their importance. C programs use two types of comments: Single-line comments begin with `//` and terminate at the end of the line; multiple-line comments begin with `/*` and end with `*/`. While comments are critical to the organization and clarity of a program, they are ignored by the compiler.

```
// This is an example of a one-line comment.

/* This is an example
   of a multi-line comment. */
```

A comment at the top of each C file is useful to describe the file's author, creation and modification dates, and purpose. The comment below could be included at the top of the `hello.c` file.

```
// hello.c
// 1 Jan 2015 Sarah_Harris@hmc.edu, David_Harris@hmc.edu
//
// This program prints "Hello world!" to the screen
```

C.3.2 #define

Constants are named using the `#define` directive and then used by name throughout the program. These globally defined constants are also called *macros*. For example, suppose you write a program that allows at most 5 user guesses, you can use `#define` to identify that number.

```
#define MAXGUESSES 5
```

Number constants in C default to decimal but can also be hexadecimal (prefix "0x") or octal (prefix "0"). Binary constants are not defined in C99 but are supported by some compilers (prefix "0b"). For example, the following assignments are equivalent:

```
char x = 37;
char x = 0x25;
char x = 045;
```

The # indicates that this line in the program will be handled by the *preprocessor*. Before compilation, the preprocessor replaces each occurrence of the identifier `MAXGUESSES` in the program with `5`. By convention, `#define` lines are located at the top of the file and identifiers are written in all capital letters. By defining constants in one location and then using the identifier in the program, the program remains consistent, and the value is easily modified – it need only be changed at the `#define` line instead of at each line in the code where the value is needed.

C Code Example eC.2 shows how to use the `#define` directive to convert inches to centimeters. The variables `inch` and `cm` are declared to be `float` to indicate they represent single-precision floating point numbers. If the conversion factor (`INCH2CM`) were used throughout a large program, having it declared using `#define` obviates errors due to typos (for example, typing 2.53 instead of 2.54) and makes it easy to find and change (for example, if more significant digits were required).

Globally defined constants eradicate *magic numbers* from a program. A magic number is a constant that shows up in a program without a name. The presence of magic numbers in a program often introduces tricky bugs – for example, when the number is changed in one location but not another.

C Code Example eC.2 USING `#define` TO DECLARE CONSTANTS

```
// Convert inches to centimeters
#include <stdio.h>

#define INCH2CM 2.54

int main(void) {
  float inch = 5.5;     // 5.5 inches
  float cm;

  cm = inch * INCH2CM;
  printf("%f inches = %f cm\n", inch, cm);
}
```

Console Output

```
5.500000 inches = 13.970000 cm
```

C.3.3 `#include`

Modularity encourages us to split programs across separate files and functions. Commonly used functions can be grouped together for easy reuse. Variable declarations, defined values, and function definitions located in a *header file* can be used by another file by adding the `#include` preprocesser directive. *Standard libraries* that provide commonly used functions are accessed in this way. For example, the following line is required to use the functions defined in the standard input/output (I/O) library, such as `printf`.

```
#include <stdio.h>
```

The ".h" postfix of the include file indicates it is a header file. While `#include` directives can be placed anywhere in the file before the included

functions, variables, or identifiers are needed, they are conventionally located at the top of a C file.

Programmer-created header files can also be included by using quotation marks (" ") around the file name instead of brackets (< >). For example, a user-created header file called `myfunctions.h` would be included using the following line.

```
#include "myfunctions.h"
```

At compile time, files specified in brackets are searched for in system directories. Files specified in quotes are searched for in the same local directory where the C file is found. If the user-created header file is located in a different directory, the path of the file relative to the current directory must be included.

SUMMARY

- **Comments:** C provides single-line comments (`//`) and multi-line comments (`/* */`).
- `#define NAME val`: the `#define` directive allows an identifier (`NAME`) to be used throughout the program. Before compilation, all instances of `NAME` are replaced with `val`.
- `#include`: `#include` allows common functions to be used in a program. For built-in libraries, include the following line at the top of the code: `#include <library.h>` To include a user-defined header file, the name must be in quotes, listing the path relative to the current directory as needed: i.e., `#include "other/myFuncs.h"`.

C.4 VARIABLES

Variables in C programs have a type, name, value, and memory location. A variable declaration states the type and name of the variable. For example, the following declaration states that the variable is of type `char` (which is a 1-byte type), and that the variable name is `x`. The compiler decides where to place this 1-byte variable in memory.

```
char x;
```

C views memory as a group of consecutive bytes, where each byte of memory is assigned a unique number indicating its location or *address*, as shown in Figure eC.1. A variable occupies one or more bytes of memory, and the address of multiple-byte variables is indicated by the lowest numbered byte. The type of a variable indicates whether to interpret the byte(s) as an integer, floating point number, or other type. The rest of this section describes C's primitive data types, the declaration of global and local variables, and the initialization of variables.

Variable names are case sensitive and can be of your choosing. However, the name may not be any of C's reserved words (i.e., `int`, `while`, etc.), may not start with a number (i.e., `int 1x;` is not a valid declaration), and may not include special characters such as `\`, `*`, `?`, or `-`. Underscores (_) are allowed.

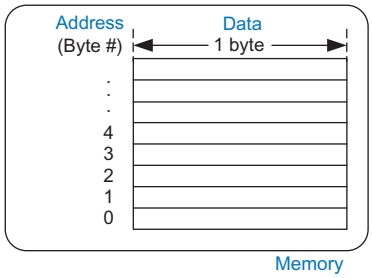


Figure eC.1 C's view of memory

C.4.1 Primitive Data Types

C has a number of primitive, or built-in, data types available. They can be broadly characterized as integers, floating-point variables, and characters. An integer represents a two's complement or unsigned number within a finite range. A floating-point variable uses IEEE floating point representation to describe real numbers with a finite range and precision. A character can be viewed as either an ASCII value or an 8-bit integer.[1] Table eC.2 lists the size and range of each primitive type. Integers may be 16, 32, or 64 bits. They use two's complement unless qualified as `unsigned`.

Table eC.2 Primitive data types and sizes

Type	Size (bits)	Minimum	Maximum
`char`	8	$-2^{-7} = -128$	$2^7 - 1 = 127$
`unsigned char`	8	0	$2^8 - 1 = 255$
`short`	16	$-2^{15} = -32{,}768$	$2^{15} - 1 = 32{,}767$
`unsigned short`	16	0	$2^{16} - 1 = 65{,}535$
`long`	32	$-2^{31} = -2{,}147{,}483{,}648$	$2^{31} - 1 = 2{,}147{,}483{,}647$
`unsigned long`	32	0	$2^{32} - 1 = 4{,}294{,}967{,}295$
`long long`	64	-2^{63}	$2^{63} - 1$
`unsigned long`	64	0	$2^{64} - 1$
`int`	machine-dependent		
`unsigned int`	machine-dependent		
`float`	32	$\pm 2^{-126}$	$\pm 2^{127}$
`double`	64	$\pm 2^{-1023}$	$\pm 2^{1022}$

[1] Technically, the C99 standard defines a character as "a bit representation that fits in a byte," without requiring a byte to be 8 bits. However, current systems define a byte as 8 bits.

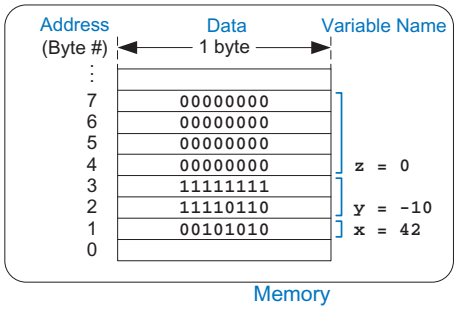


Figure eC.2 **Variable storage in memory for C Code Example eC.3**

The size of the `int` type is machine dependent and is generally the native word size of the machine. For example, on a 32-bit ARM processor, the size of an `int` or `unsigned int` is 32 bits. Floating point numbers may be 32- or 64-bit single or double precision. Characters are 8 bits.

C Code Example eC.3 shows the declaration of variables of different types. As shown in Figure eC.2, `x` requires one byte of data, `y` requires two, and `z` requires four. The program decides where these bytes are stored in memory, but each type always requires the same amount of data. For illustration, the addresses of `x`, `y`, and `z` in this example are 1, 2, and 4. Variable names are case-sensitive, so, for example, the variable `x` and the variable `X` are two different variables. (But it would be very confusing to use both in the same program!)

The machine-dependent nature of the `int` data type is a blessing and a curse. On the bright side, it matches the native word size of the processor so it can be fetched and manipulated efficiently. On the down side, programs using `int`s may behave differently on different computers. For example, a banking program might store the number of cents in your bank account as an `int`. When compiled on a 64-bit PC, it will have plenty of range for even the wealthiest entrepreneur. But if it is ported to a 16-bit microcontroller, it will overflow for accounts exceeding $327.67, resulting in unhappy and poverty-stricken customers.

C Code Example eC.3 EXAMPLE DATA TYPES

```
// Examples of several data types and their binary representations
unsigned char x = 42;          // x = 00101010
short y = -10;                 // y = 11111111 11110110
unsigned long z = 0;           // z = 00000000 00000000 00000000 00000000
```

C.4.2 Global and Local Variables

Global and local variables differ in where they are declared and where they are visible. A global variable is declared outside of all functions, typically at the top of a program, and can be accessed by all functions. Global variables should be used sparingly because they violate the principle of modularity, making large programs more difficult to read. However, a variable accessed by many functions can be made global.

A local variable is declared inside a function and can only be used by that function. Therefore, two functions can have local variables with the same names without interfering with each other. Local variables are declared at the beginning of a function. They cease to exist when the function ends and are recreated when the function is called again. They do not retain their value from one invocation of a function to the next.

The *scope* of a variable is the context in which it can be used. For example, for a local variable, its scope is the function in which it is declared. It is out of scope everywhere else.

C Code Examples eC.4 and eC.5 compare programs using global versus local variables. In C Code Example eC.4, the global variable `max` can be accessed by any function. Using a local variable, as shown in C Code Example eC.5, is the preferred style because it preserves the well-defined interface of modularity.

C Code Example eC.4 GLOBAL VARIABLES

```
// Use a global variable to find and print the maximum of 3 numbers

int max;                // global variable holding the maximum value

void findMax(int a, int b, int c) {
   max = a;
   if (b > max) {
     if (c > b) max = c;
     else       max = b;
   } else if (c > max) max = c;
}

void printMax(void) {
   printf("The maximum number is: %d\n", max);
}

int main(void) {
   findMax(4, 3, 7);
   printMax();
}
```

C Code Example eC.5 LOCAL VARIABLES

```
// Use local variables to find and print the maximum of 3 numbers

int getMax(int a, int b, int c) {
   int result = a;  // local variable holding the maximum value

   if (b > result) {
     if (c > b) result = c;
     else       result = b;
   } else if (c > result) result = c;

   return result;
}

void printMax(int m) {
   printf("The maximum number is: %d\n", m);
}

int main(void) {
   int max;

   max = getMax(4, 3, 7);
   printMax(max);
}
```

C.4.3 Initializing Variables

A variable needs to be *initialized* – assigned a value – before it is read. When a variable is declared, the correct number of bytes is reserved for that variable in memory. However, the memory at those locations retains whatever value it had last time it was used, essentially a random value. Global and local variables can be initialized either when they are declared or within the body of the program. C Code Example eC.3 shows variables initialized at the same time they are declared. C Code Example eC.4 shows how variables are initialized before their use, but after declaration; the global variable `max` is initialized by the `getMax` function before it is read by the `printMax` function. Reading from uninitialized variables is a common programming error, and can be tricky to debug.

SUMMARY

- **Variables:** Each variable is defined by its data type, name, and memory location. A variable is declared as `datatype name`.
- **Data types:** A data type describes the size (number of bytes) and representation (interpretation of the bytes) of a variable. Table eC.2 lists C's built-in data types.
- **Memory:** C views memory as a list of bytes. Memory stores variables and associates each variable with an address (byte number).
- **Global variables:** Global variables are declared outside of all functions and can be accessed anywhere in the program.
- **Local variables:** Local variables are declared within a function and can be accessed only within that function.
- **Variable initialization:** Each variable must be initialized before it is read. Initialization can happen either at declaration or afterward.

C.5 OPERATORS

The most common type of statement in a C program is an *expression*, such as

```
y = a + 3;
```

An expression involves operators (such as + or *) acting on one or more operands, such as variables or constants. C supports the operators shown in Table eC.3, listed by category and in order of decreasing precedence. For example, multiplicative operators take precedence over additive

Table eC.3 Operators listed by decreasing precedence

Category	Operator	Description	Example
Unary	++	post-increment	`a++; // a = a+1`
	--	post-decrement	`x--; // x = x-1`
	&	memory address of a variable	`x = &y; // x = the memory // address of y`
	~	bitwise NOT	`z = ~a;`
	!	Boolean NOT	`!x`
	-	negation	`y = -a;`
	++	pre-increment	`++a; // a = a+1`
	--	pre-decrement	`--x; // x = x-1`
	(type)	casts a variable to (type)	`x = (int)c; // cast c to an // int and assign it to x`
	sizeof()	size of a variable or type in bytes	`long int y; x = sizeof(y); // x = 4`
Multiplicative	*	multiplication	`y = x * 12;`
	/	division	`z = 9 / 3; // z = 3`
	%	modulo	`z = 5 % 2; // z = 1`
Additive	+	addition	`y = a + 2;`
	-	subtraction	`y = a - 2;`
Bitwise Shift	<<	bitshift left	`z = 5 << 2; // z = 0b00010100`
	>>	bitshift right	`x = 9 >> 3; // x = 0b00000001`
Relational	==	equals	`y == 2`
	!=	not equals	`x != 7`
	<	less than	`y < 12`
	>	greater than	`val > max`
	<=	less than or equal	`z <= 2`
	>=	greater than or equal	`y >= 10`

(*continued*)

Table eC.3 Operators listed by decreasing precedence—Cont'd

Category	Operator	Description	Example
Bitwise	`&`	bitwise AND	`y = a & 15;`
	`^`	bitwise XOR	`y = 2 ^ 3;`
	`|`	bitwise OR	`y = a | b;`
Logical	`&&`	Boolean AND	`x && y`
	`||`	Boolean OR	`x || y`
Ternary	`? :`	ternary operator	`y = x ? a : b; // if x is TRUE, // y=a, else y=b`
Assignment	`=`	assignment	`x = 22;`
	`+=`	addition and assignment	`y += 3;` `// y = y + 3`
	`-=`	subtraction and assignment	`z -= 10;` `// z = z - 10`
	`*=`	multiplication and assignment	`x *= 4;` `// x = x * 4`
	`/=`	division and assignment	`y /= 10;` `// y = y / 10`
	`%=`	modulo and assignment	`x %= 4;` `// x = x % 4`
	`>>=`	bitwise right-shift and assignment	`x >>= 5;` `// x = x>>5`
	`<<=`	bitwise left-shift and assignment	`x <<= 2;` `// x = x<<2`
	`&=`	bitwise AND and assignment	`y &= 15;` `// y = y & 15`
	`|=`	bitwise OR and assignment	`x |= y;` `// x = x | y`
	`^=`	bitwise XOR and assignment	`x ^= y;` `// x = x ^ y`

operators. Within the same category, operators are evaluated in the order that they appear in the program.

Unary operators, also called monadic operators, have a single operand. Ternary operators have three operands, and all others have two. The ternary operator (from the Latin *ternarius* meaning consisting of three) chooses the second or third operand depending on whether the first value is TRUE (nonzero) or FALSE (zero), respectively. C Code Example eC.6 shows how to compute `y = max(a,b)` using the ternary operator, along with an equivalent but more verbose `if/else` statement.

The Truth, the Whole Truth, and Nothing But the Truth
C considers a variable to be TRUE if it is nonzero and FALSE if it is zero. Logical and ternary operators, as well as control-flow statements such as `if` and `while`, depend on the truth of a variable. Relational and logical operators produce a result that is 1 when TRUE or 0 when FALSE.

C Code Example eC.6 (a) TERNARY OPERATOR, AND (b) EQUIVALENT `if/else` STATEMENT

```
(a) y = (a > b) ? a : b; // parentheses not necessary, but makes it clearer
(b) if (a > b) y = a;
    else       y = b;
```

Simple assignment uses the = operator. C code also allows for compound assignment, that is, assignment after a simple operation such as addition (+=) or multiplication (*=). In compound assignments, the variable on the left side is both operated on and assigned the result. C Code Example eC.7 shows these and other C operations. Binary values in the comments are indicated with the prefix "`0b`".

C Code Example eC.7 OPERATOR EXAMPLES

Expression	Result	Notes
`44 / 14`	`3`	Integer division truncates
`44 % 14`	`2`	44 mod 14
`0x2C && 0xE  //0b101100 && 0b1110`	`1`	Logical AND
`0x2C || 0xE  //0b101100 || 0b1110`	`1`	Logical OR
`0x2C & 0xE  //0b101100 & 0b1110`	`0xC (0b001100)`	Bitwise AND
`0x2C | 0xE  //0b101100 | 0b1110`	`0x2E (0b101110)`	Bitwise OR
`0x2C ^ 0xE  //0b101100 ^ 0b1110`	`0x22 (0b100010)`	Bitwise XOR
`0xE << 2  //0b1110 << 2`	`0x38 (0b111000)`	Left shift by 2
`0x2C >> 3  //0b101100 >> 3`	`0x5 (0b101)`	Right shift by 3
`x = 14;` `x += 2;`	`x=16`	

`y = 0x2C;  // y = 0b101100`
`y &= 0xF;  // y &= 0b1111`
	`y=0xC (0b001100)`	

`x = 14; y = 44;` `y = y + x++;`	`x=15, y=58`	Increment `x` after using it
`x = 14; y = 44;` `y = y + ++x;`	`x=15, y=59`	Increment `x` before using it

C.6 FUNCTION CALLS

Modularity is key to good programming. A large program is divided into smaller parts called functions that, similar to hardware modules, have well-defined inputs, outputs, and behavior. C Code Example eC.8 shows the `sum3` function. The function declaration begins with the return type, `int`, followed by the name, `sum3`, and the inputs enclosed within parentheses `(int a, int b, int c)`. Curly braces `{}` enclose the body of the function, which may contain zero or more statements. The `return` statement indicates the value that the function should return to its caller; this can be viewed as the output of the function. A function can only return a single value.

C Code Example eC.8 `sum3` FUNCTION

```
// Return the sum of the three input variables
int sum3(int a, int b, int c) {
  int result = a + b + c;
  return result;
}
```

After the following call to `sum3`, `y` holds the value 42.

```
int y = sum3(10, 15, 17);
```

Although a function may have inputs and outputs, neither is required. C Code Example eC.9 shows a function with no inputs or outputs. The keyword `void` before the function name indicates that nothing is returned. `void` between the parentheses indicates that the function has no input arguments.

Nothing between the parentheses also indicates no input arguments. So, in this case we could have written:

```
void printPrompt()
```

C Code Example eC.9 FUNCTION `printPrompt` WITH NO INPUTS OR OUTPUTS

```
// Print a prompt to the console
void printPrompt(void)
{
  printf("Please enter a number from 1-3:\n");
}
```

A function must be declared in the code before it is called. This may be done by placing the called function earlier in the file. For this reason, `main` is often placed at the end of the C file after all the functions it calls. Alternatively, a function *prototype* can be placed in the program before the function is defined. The function prototype is the first line of

With careful ordering of functions, prototypes may be unnecessary. However, they are unavoidable in certain cases, such as when function `f1` calls `f2` and `f2` calls `f1`. It is good style to place prototypes for all of a program's functions near the beginning of the C file or in a header file.

the function, declaring the return type, function name, and function inputs. For example, the function prototypes for the functions in C Code Examples eC.8 and eC.9 are:

```
int sum3(int a, int b, int c);
void printPrompt(void);
```

C Code Example eC.10 shows how function prototypes are used. Even though the functions themselves are after `main`, the function prototypes at the top of the file allow them to be used in `main`.

C Code Example eC.10 FUNCTION PROTOTYPES

```
#include <stdio.h>

// function prototypes
int sum3(int a, int b, int c);
void printPrompt(void);

int main(void)
{
  int y = sum3(10, 15, 20);

  printf("sum3 result: %d\n", y);
  printPrompt();
}

int sum3(int a, int b, int c) {
  int result = a+b+c;
  return result;
}

void printPrompt(void) {
  printf("Please enter a number from 1-3:\n");
}
```

Console Output

```
sum3 result: 45
Please enter a number from 1-3:
```

As with variable names, function names are case sensitive, cannot be any of C's reserved words, may not contain special characters (except underscore _), and cannot start with a number. Typically function names include a verb to indicate what they do.

Be consistent in how you capitalize your function and variable names so you don't have to constantly look up the correct capitalization. Two common styles are to camelCase, in which the initial letter of each word after the first is capitalized like the humps of a camel (e.g., `printPrompt`), or to use underscores between words (e.g., `print_prompt`). We have unscientifically observed that reaching for the underscore key exacerbates carpal tunnel syndrome (my pinky finger twinges just thinking about the underscore!) and hence prefer camelCase. But the most important thing is to be consistent in style within your organization.

The `main` function is always declared to return an `int`, which conveys to the operating system the reason for program termination. A zero indicates normal completion, while a nonzero value signals an error condition. If `main` reaches the end without encountering a `return` statement, it will automatically return 0. Most operating systems do not automatically inform the user of the value returned by the program.

C.7 CONTROL-FLOW STATEMENTS

C provides control-flow statements for conditionals and loops. Conditionals execute a statement only if a condition is met. A loop repeatedly executes a statement as long as a condition is met.

C.7.1 Conditional Statements

`if`, `if/else`, and `switch/case` statements are conditional statements commonly used in high-level languages including C.

if Statements

An `if` statement executes the statement immediately following it when the expression in parentheses is TRUE (i.e., nonzero). The general format is:

```
if (expression)
  statement
```

C Code Example eC.11 shows how to use an `if` statement in C. When the variable `aintBroke` is equal to 1, the variable `dontFix` is set to 1. A block of multiple statements can be executed by placing curly braces `{}` around the statements, as shown in C Code Example eC.12.

Curly braces, `{}`, are used to group one or more statements into a *compound statement* or *block*.

C Code Example eC.11 `if` STATEMENT

```
int dontFix = 0;

if (aintBroke == 1)
  dontFix = 1;
```

C Code Example eC.12 `if` STATEMENT WITH A BLOCK OF CODE

```
// If amt >= $2, prompt user and dispense candy
if (amt >= 2) {
  printf("Select candy.\n");
  dispenseCandy = 1;
}
```

if/else Statements

`if/else` statements execute one of two statements depending on a condition, as shown below. When the `expression` in the `if` statement is TRUE, `statement1` is executed. Otherwise, `statement2` is executed.

```
if (expression)
  statement1
else
  statement2
```

C Code Example eC.6(b) gives an example `if/else` statement in C. The code sets `y` equal to `a` if `a` is greater than `b`; otherwise `y = b`.

switch/case Statements

`switch/case` statements execute one of several statements depending on the conditions, as shown in the general format below.

```
switch (variable) {
   case (expression1): statement1 break;
   case (expression2): statement2 break;
   case (expression3): statement3 break;
   default:            statement4
}
```

For example, if `variable` is equal to `expression2`, execution continues at `statement2` until the keyword `break` is reached, at which point it exits the `switch/case` statement. If no conditions are met, the `default` executes.

If the keyword `break` is omitted, execution begins at the point where the condition is TRUE and then falls through to execute the remaining cases below it. This is usually not what you want and is a common error among beginning C programmers.

C Code Example eC.13 shows a `switch/case` statement that, depending on the variable `option`, determines the amount of money `amt` to be disbursed. A `switch/case` statement is equivalent to a series of nested `if/else` statements, as shown by the equivalent code in C Code Example eC.14.

C Code Example eC.13 `switch/case` STATEMENT

```
// Assign amt depending on the value of option
switch (option) {
   case 1:  amt = 100; break;
   case 2:  amt = 50;  break;
   case 3:  amt = 20;  break;
   case 4:  amt = 10;  break;
   default: printf("Error: unknown option.\n");
}
```

C Code Example eC.14 NESTED `if/else` STATEMENT

```
// Assign amt depending on the value of option
if      (option == 1)  amt = 100;
else if (option == 2)  amt = 50;
else if (option == 3)  amt = 20;
else if (option == 4)  amt = 10;
else printf("Error: unknown option.\n");
```

C.7.2 Loops

`while`, `do/while`, and `for` loops are common loop constructs used in many high-level languages including C. These loops repeatedly execute a statement as long as a condition is satisfied.

while Loops

`while` loops repeatedly execute a statement until a condition is not met, as shown in the general format below.

```
while (condition)
  statement
```

The `while` loop in C Code Example eC.15 computes the factorial of 9 = $9 \times 8 \times 7 \times \cdots \times 1$. Note that the condition is checked before executing the `statement`. In this example, the statement is a compound statement or block, so curly braces are required.

C Code Example eC.15 `while` LOOP

```
// Compute 9! (the factorial of 9)
int i = 1, fact = 1;

// multiply the numbers from 1 to 9
while (i < 10) { // while loops check the condition first
  fact *= i;
  i++;
}
```

do/while Loops

`do/while` loops are like `while` loops but the `condition` is checked only after the `statement` is executed once. The general format is shown below. The `condition` is followed by a semi-colon.

```
do
  statement
while (condition);
```

The `do/while` loop in C Code Example eC.16 queries a user to guess a number. The program checks the condition (if the user's number is equal to the correct number) only after the body of the `do/while` loop executes once. This construct is useful when, as in this case, something must be done (for example, the guess retrieved from the user) before the condition is checked.

C Code Example eC.16 `do/while` LOOP

```
// Query user to guess a number and check it against the correct number.
#define MAXGUESSES 3
#define CORRECTNUM 7
int guess, numGuesses = 0;
```

```
do {
  printf("Guess a number between 0 and 9. You have %d more guesses.\n",
         (MAXGUESSES-numGuesses));
  scanf("%d", &guess);     // read user input
  numGuesses++;
} while ( (numGuesses < MAXGUESSES) & (guess != CORRECTNUM) );
// do loop checks the condition after the first iteration

if (guess == CORRECTNUM)
   printf("You guessed the correct number!\n");
```

for Loops

`for` loops, like `while` and `do/while` loops, repeatedly execute a `statement` until a condition is not satisfied. However, `for` loops add support for a *loop variable*, which typically keeps track of the number of loop executions. The general format of the `for` loop is

```
for (initialization; condition; loop operation)
   statement
```

The `initialization` code executes only once, before the `for` loop begins. The `condition` is tested at the beginning of each iteration of the loop. If the `condition` is not TRUE, the loop exits. The `loop operation` executes at the end of each iteration. C Code Example eC.17 shows the factorial of 9 computed using a `for` loop.

C Code Example eC.17 `for` LOOP

```
// Compute 9!
int i;    // loop variable
int fact = 1;

for (i=1; i<10; i++)
   fact *= i;
```

Whereas the `while` and `do/while` loops in C Code Examples eC.15 and eC.16 include code for incrementing and checking the loop variable `i` and `numGuesses`, respectively, the `for` loop incorporates those statements into its format. A `for` loop could be expressed equivalently, but less conveniently, as

```
initialization;
while (condition) {
   statement
   loop operation;
}
```

SUMMARY

- **Control-flow statements:** C provides control-flow statements for conditional statements and loops.
- **Conditional statements:** Conditional statements execute a statement when a condition is TRUE. C includes the following conditional statements: `if`, `if/else`, and `switch/case`.
- **Loops:** Loops repeatedly execute a statement until a condition is FALSE. C provides `while`, `do/while`, and `for` loops.

C.8 MORE DATA TYPES

Beyond various sizes of integers and floating-point numbers, C includes other special data types including pointers, arrays, strings, and structures. These data types are introduced in this section along with dynamic memory allocation.

C.8.1 Pointers

A pointer is the address of a variable. C Code Example eC.18 shows how to use pointers. `salary1` and `salary2` are variables that can contain integers, and `ptr` is a variable that can hold the address of an integer. The compiler will assign arbitrary locations in RAM for these variables depending on the runtime environment. For the sake of concreteness, suppose this program is compiled on a 32-bit system with `salary1` at addresses 0x70-73, `salary2` at addresses 0x74-77, and `ptr` at 0x78-7B. Figure eC.3 shows memory and its contents after the program is executed.

In a variable declaration, a star (*) before a variable name indicates that the variable is a pointer to the declared type. In using a pointer variable, the * operator *dereferences* a pointer, returning the value stored at the

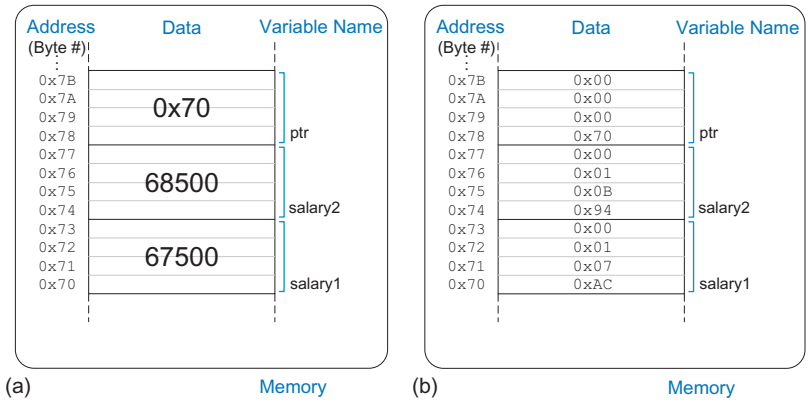


Figure eC.3 Contents of memory after C Code Example eC.18 executes shown (a) by value and (b) by byte using little-endian memory

Dereferencing a pointer to a non-existent memory location or an address outside of the range accessible by the program will usually cause a program to crash. The crash is often called a *segmentation fault*.

indicated memory address contained in the pointer. The & operator is pronounced "address of," and it produces the memory address of the variable being referenced.

Pointers are particularly useful when a function needs to modify a variable, instead of just returning a value. Because functions can't modify their inputs directly, a function can make the input a pointer to the variable. This is called passing an input variable *by reference* instead of *by value*, as shown in prior examples. C Code Example eC.19 gives an example of passing `x` by reference so that `quadruple` can modify the variable directly.

C Code Example eC.18 POINTERS

```
// Example pointer manipulations
int salary1, salary2;  // 32-bit numbers
int *ptr;              // a pointer specifying the address of an int variable

salary1 = 67500;       // salary1 = $67,500 = 0x000107AC
ptr = &salary1;        // ptr = 0x0070, the address of salary1
salary2 = *ptr + 1000; /* dereference ptr to give the contents of address 70 = $67,500,
                          then add $1,000 and set salary2 to $68,500 */
```

C Code Example eC.19 PASSING AN INPUT VARIABLE BY REFERENCE

```
// Quadruple the value pointed to by a
#include <stdio.h>

void quadruple(int *a)
{
  *a = *a * 4;
}

int main(void)
{
  int x = 5;

  printf("x before: %d\n", x);
  quadruple(&x);
  printf("x after: %d\n", x);
  return 0;
}
```

Console Output

```
x before: 5
x after: 20
```

A pointer to address 0 is called a *null pointer* and indicates that the pointer is not actually pointing to meaningful data. It is written as `NULL` in a program.

C.8.2 Arrays

An array is a group of similar variables stored in consecutive addresses in memory. The elements are numbered from 0 to $N-1$, where N is the length of the array. C Code Example eC.20 declares an array variable called `scores` that holds the final exam scores for three students. Memory space is reserved for three `long`s, that is, $3 \times 4 = 12$ bytes. Suppose the `scores` array starts at address 0x40. The address of the 1st element (i.e., `scores[0]`) is 0x40, the 2nd element is 0x44, and the 3rd element is 0x48, as shown in Figure eC.4. In C, the array variable, in this case `scores`, is a pointer to the 1st element. It is the programmer's responsibility not to access elements beyond the end of the array. C has no internal bounds checking, so a program that writes beyond the end of an array will compile fine but may stomp on other parts of memory when it runs.

C Code Example eC.20 ARRAY DECLARATION

```
long scores[3];    // array of three 4-byte numbers
```

The elements of an array can be initialized either at declaration using curly braces `{}`, as shown in C Code Example eC.21, or individually in the body of the code, as shown in C Code Example eC.22. Each element of an array is accessed using brackets `[]`. The contents of memory containing the array are shown in Figure eC.4. Array initialization using curly braces `{}` can only be performed at declaration, and not afterward. `for` loops are commonly used to assign and read array data, as shown in C Code Example eC.23.

C Code Example eC.21 ARRAY INITIALIZATION AT DECLARATION USING {}

```
long scores[3]={93, 81, 97}; // scores[0]=93; scores[1]=81; scores[2]=97;
```

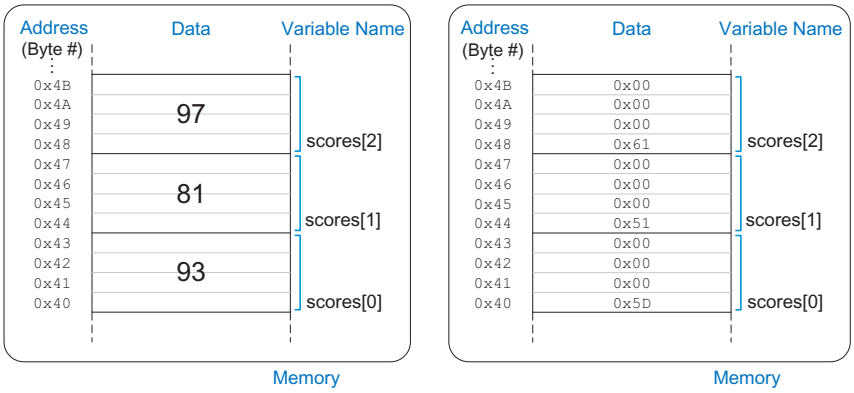


Figure eC.4 `scores` **array stored in memory**

C Code Example eC.22 ARRAY INITIALIZATION USING ASSIGNMENT

```
long scores[3];

scores[0] = 93;
scores[1] = 81;
scores[2] = 97;
```

C Code Example eC.23 ARRAY INITIALIZATION USING A `for` LOOP

```
// User enters 3 student scores into an array
long scores[3];
int i, entered;

printf("Please enter the student's 3 scores.\n");
for (i=0; i<3; i++) {
  printf("Enter a score and press enter.\n");
  scanf("%d", &entered);
  scores[i] = entered;
}
printf("Scores: %d %d %d\n", scores[0], scores[1], scores[2]);
```

When an array is declared, the length must be constant so that the compiler can allocate the proper amount of memory. However, when the array is passed to a function as an input argument, the length need not be defined because the function only needs to know the address of the beginning of the array. C Code Example eC.24 shows how an array is passed to a function. The input argument `arr` is simply the address of the 1st element of an array. Often the number of elements in an array is also passed as an input argument. In a function, an input argument of type `int[]` indicates that it is an array of integers. Arrays of any type may be passed to a function.

C Code Example eC.24 PASSING AN ARRAY AS AN INPUT ARGUMENT

```
// Initialize a 5-element array, compute the mean, and print the result.
#include <stdio.h>

// Returns the mean value of an array (arr) of length len
float getMean(int arr[], int len) {
  int i;
  float mean, total = 0;

  for (i=0; i < len; i++)
    total += arr[i];

  mean = total / len;
  return mean;
}
```

```
int main(void) {
  int data[4] = {78, 14, 99, 27};
  float avg;

  avg = getMean(data, 4);

  printf("The average value is: %f.\n", avg);
}
```

Console Output

```
The average value is: 54.500000.
```

An array argument is equivalent to a pointer to the beginning of the array. Thus, `getMean` could also have been declared as

```
float getMean(int *arr, int len);
```

Although functionally equivalent, `datatype[]` is the preferred method for passing arrays as input arguments because it more clearly indicates that the argument is an array.

A function is limited to a single output, i.e., return variable. However, by receiving an array as an input argument, a function can essentially output more than a single value by changing the array itself. C Code Example eC.25 sorts an array from lowest to highest and leaves the result in the same array. The three function prototypes below are equivalent. The length of an array in a function declaration (i.e., `int vals[100]`) is ignored.

```
void sort(int *vals, int len);
void sort(int vals[], int len);
void sort(int vals[100], int len);
```

C Code Example eC.25 PASSING AN ARRAY AND ITS LENGTH AS INPUTS

```
// Sort the elements of the array vals of length len from lowest to highest
void sort(int vals[], int len)
{
  int i, j, temp;

  for (i=0; i<len; i++) {
    for (j=i+1; j<len; j++) {
      if (vals[i] > vals[j]) {
        temp = vals[i];
        vals[i] = vals[j];
        vals[j] = temp;
      }
    }
  }
}
```

Arrays may have multiple dimensions. C Code Example eC.26 uses a two-dimensional array to store the grades across eight problem sets for ten students. Recall that initialization of array values using `{}` is only allowed at declaration.

C Code Example eC.26 TWO-DIMENSIONAL ARRAY INITIALIZATION

```
// Initialize 2-D array at declaration
 int grades[10][8] = { {100, 107, 99,  101, 100, 104, 109, 117},
                       {103, 101, 94,  101, 102, 106, 105, 110},
                       {101, 102, 92,  101, 100, 107, 109, 110},
                       {114, 106, 95,  101, 100, 102, 102, 100},
                       {98,  105, 97,  101, 103, 104, 109, 109},
                       {105, 103, 99,  101, 105, 104, 101, 105},
                       {103, 101, 100, 101, 108, 105, 109, 100},
                       {100, 102, 102, 101, 102, 101, 105, 102},
                       {102, 106, 110, 101, 100, 102, 120, 103},
                       {99,  107, 98,  101, 109, 104, 110, 108} };
```

C Code Example eC.27 shows some functions that operate on the 2-D `grades` array from C Code Example eC.26. Multi-dimensional arrays used as input arguments to a function must define all but the first dimension. Thus, the following two function prototypes are acceptable:

```
void print2dArray(int arr[10][8]);
void print2dArray(int arr[][8]);
```

C Code Example eC.27 OPERATING ON MULTI-DIMENSIONAL ARRAYS

```
#include <stdio.h>

// Print the contents of a 10×8 array
void print2dArray(int arr[10][8])
{
  int i, j;

  for (i=0; i<10; i++) {          // for each of the 10 students
    printf("Row %d\n", i);
    for (j=0; j<8; j++) {
      printf("%d ", arr[i][j]); // print scores for all 8 problem sets
    }
    printf("\n");
  }
}

// Calculate the mean score of a 10×8 array
float getMean(int arr[10][8])
{
  int i, j;
  float mean, total = 0;

  // get the mean value across a 2D array
  for (i=0; i<10; i++) {
```

```
    for (j=0; j<8; j++) {
      total += arr[i][j];       // sum array values
    }
  }
  mean = total/(10*8);
  printf("Mean is: %f\n", mean);
  return mean;
}
```

Note that because an array is represented by a pointer to the initial element, C cannot copy or compare arrays using the = or == operators. Instead, you must use a loop to copy or compare each element one at a time.

C.8.3 Characters

A character (`char`) is an 8-bit variable. It can be viewed either as a two's complement number between −128 and 127 or as an ASCII code for a letter, digit, or symbol. ASCII characters can be specified as a numeric value (in decimal, hexadecimal, etc.) or as a printable character enclosed in single quotes. For example, the letter A has the ASCII code 0x41, B=0x42, etc. Thus `'A' + 3` is 0x44, or `'D'`. Table 6.5 lists the ASCII character encodings, and Table eC.4 lists characters used to indicate formatting or special characters. Formatting codes include carriage return (`\r`), newline (`\n`), horizontal tab (`\t`), and the end of a string (`\0`). `\r` is shown for completeness but is rarely used in C programs. `\r` returns the carriage (location of typing) to the beginning (left) of the line, but any text that was there is overwritten. `\n`, instead, moves the location of typing to the beginning of a new line.[2] The *NULL* character (`'\0'`) indicates the end of a text string and is discussed next in Section C.8.4.

Table eC.4 Special characters

Special Character	Hexadecimal Encoding	Description
`\r`	0x0D	carriage return
`\n`	0x0A	new line
`\t`	0x09	tab
`\0`	0x00	terminates a string
`\\`	0x5C	backslash
`\"`	0x22	double quote
`\'`	0x27	single quote
`\a`	0x07	bell

[2] Windows text files use `\r\n` to represent end-of-line while UNIX-based systems use `\n`, which can cause nasty bugs when moving text files between systems.

The term "carriage return" originates from typewriters that required the carriage, the contraption that holds the paper, to move to the right in order to allow typing to begin at the left side of the page. A carriage return lever, shown on the left in the figure below, is pressed so that the carriage would both move to the right and advance the paper by one line, called a line feed.

A Remington electric typewriter used by Winston Churchill. (*http://cwr.iwm.org.uk/server/show/conMediaFile.71979*)

C strings are called *null terminated* or *zero terminated* because the length is determined by looking for a zero at the end. In contrast, languages such as Pascal use the first byte to specify the string length, up to a maximum of 255 characters. This byte is called the *prefix byte* and such strings are called *P-strings*. An advantage of null-terminated strings is that the length can be arbitrarily great. An advantage of P-strings is that the length can be determined immediately without having to inspect all of the characters of the string.

C.8.4 Strings

A string is an array of characters used to store a piece of text of bounded but variable length. Each character is a byte representing the ASCII code for that letter, number, or symbol. The size of the array determines the maximum length of the string, but the actual length of the string could be shorter. In C, the length of the string is determined by looking for the null terminator (ASCII value 0x00) at the end of the string.

C Code Example eC.28 shows the declaration of a 10-element character array called `greeting` that holds the string `"Hello!"`. For concreteness, suppose `greeting` starts at memory address 0x50. Figure eC.5 shows the contents of memory from 0x50 to 0x59 holding the string `"Hello!"` Note that the string only uses the first seven elements of the array, even though ten elements are allocated in memory.

C Code Example eC.28 STRING DECLARATION

```
char greeting[10] = "Hello!";
```

C Code Example eC.29 shows an alternate declaration of the string `greeting`. The pointer `greeting` holds the address of the 1st element of a 7-element array comprised of each of the characters in "Hello!" followed by the null terminator. The code also demonstrates how to print strings by using the `%s` format code.

C Code Example eC.29 ALTERNATE STRING DECLARATION

```
char *greeting = "Hello!";
printf("greeting: %s", greeting);
```

Console Output

```
greeting: Hello!
```

Unlike primitive variables, a string cannot be set equal to another string using the equals operator, =. Each element of the character array must be individually copied from the source string to the target string. This is true for any array. C Code Example eC.30 copies one string, `src`, to another, `dst`. The sizes of the arrays are not needed, because the end of the `src` string is indicated by the null terminator. However, `dst` must be large enough so that you don't stomp on other data. `strcpy` and other string manipulation functions are available in C's built-in libraries (see Section C.9.4).

C Code Example eC.30 COPYING STRINGS

```
// Copy the source string, src, to the destination string, dst
void strcpy(char *dst, char *src)
{
```

```
  int i = 0;

  do {
    dst[i] = src[i];        // copy characters one byte at a time
  } while (src[i++]);       // until the null terminator is found
}
```

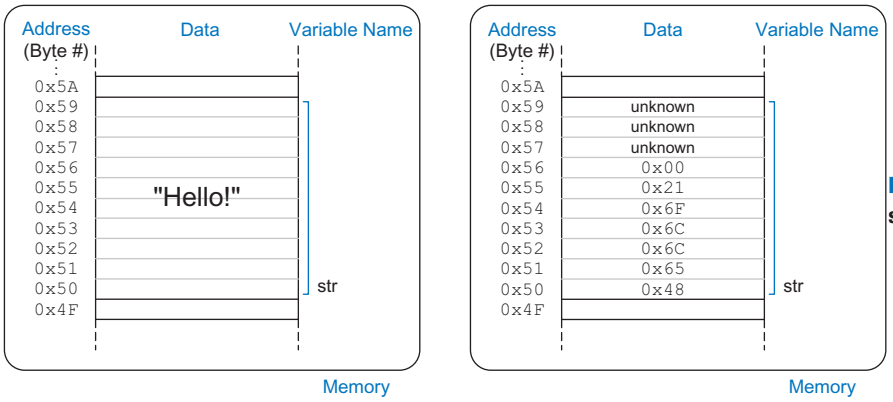


Figure eC.5 **The string "Hello!" stored in memory**

C.8.5 Structures

In C, structures are used to store a collection of data of various types. The general format of a structure declaration is

```
struct name {
  type1 element1;
  type2 element2;
  ...
};
```

where `struct` is a keyword indicating that it is a structure, `name` is the structure tag name, and `element1` and `element2` are members of the structure. A structure may have any number of members. C Code Example eC.31 shows how to use a structure to store contact information. The program then declares a variable `c1` of type `struct contact`.

C Code Example eC.31 STRUCTURE DECLARATION

```
struct contact {
  char name[30];
  int phone;
  float height; // in meters
};

struct contact c1;
```

```
strcpy(c1.name, "Ben Bitdiddle");
c1.phone = 7226993;
c1.height = 1.82;
```

Just like built-in C types, you can create arrays of structures and pointers to structures. C Code Example eC.32 creates an array of contacts.

C Code Example eC.32 ARRAY OF STRUCTURES

```
struct contact classlist[200];
classlist[0].phone = 9642025;
```

It is common to use pointers to structures. C provides the *member access operator* -> to dereference a pointer to a structure and access a member of the structure. C Code Example eC.33 shows an example of declaring a pointer to a `struct contact`, assigning it to point to the 42nd element of `classlist` from C Code Example eC.32, and using the member access operator to set a value in that element.

C Code Example eC.33 ACCESSING STRUCTURE MEMBERS USING POINTERS AND ->

```
struct contact *cptr;
cptr = &classlist[42];
cptr->height = 1.9;  // equivalent to: (*cptr).height = 1.9;
```

Structures can be passed as function inputs or outputs by value or by reference. Passing by value requires the compiler to copy the entire structure into memory for the function to access. This can require a large amount of memory and time for a big structure. Passing by reference involves passing a pointer to the structure, which is more efficient. The function can also modify the structure being pointed to rather than having to return another structure. C Code Example eC.34 shows two versions of the `stretch` function that makes a contact 2 cm taller. `stretchByReference` avoids copying the large structure twice.

C Code Example eC.34 PASSING STRUCTURES BY VALUE OR BY NAME

```
struct contact stretchByValue(struct contact c)
{
  c.height += 0.02;
  return c;
}
```

```
void stretchByReference(struct contact *cptr)
{
  cptr->height += 0.02;
}

int main(void)
{
  struct contact George;

  George.height = 1.4; // poor fellow has been stooped over
  George = stretchByValue(George); // stretch for the stars
  stretchByReference(&George);     // and stretch some more
}
```

C.8.6 typedef

C also allows you to define your own names for data types using the `typedef` statement. For example, writing `struct contact` becomes tedious when it is often used, so we can define a new type named `contact` and use it as shown in C Code Example eC.35.

C Code Example eC.35 CREATING A CUSTOM TYPE USING `typedef`

```
typedef struct contact {
  char name[30];
  int phone;
  float height; // in meters
} contact;        // defines contact as shorthand for "struct contact"

contact c1;       // now we can declare the variable as type contact
```

`typedef` can be used to create a new type occupying the same amount of memory as a primitive type. C Code Example eC.36 defines `byte` and `bool` as 8-bit types. The `byte` type may make it clearer that the purpose of `pos` is to be an 8-bit number rather than an ASCII character. The `bool` type indicates that the 8-bit number is representing TRUE or FALSE. These types make a program easier to read than if one simply used `char` everywhere.

C Code Example eC.36 `typedef byte` AND `bool`

```
typedef unsigned char byte;
typedef char bool;
#define TRUE 1
#define FALSE 0

byte pos = 0x45;
bool loveC = TRUE;
```

C Code Example eC.37 illustrates defining a 3-element `vector` and a 3×3 `matrix` type using arrays.

C Code Example eC.37 `typedef vector` AND `matrix`

```
typedef double vector[3];
typedef double matrix[3][3];

vector a = {4.5, 2.3, 7.0};
matrix b = {{3.3, 4.7, 9.2}, {2.5, 4, 9}, {3.1, 99.2, 88}};
```

C.8.7 Dynamic Memory Allocation*

In all the examples thus far, variables have been declared *statically*; that is, their size is known at compile time. This can be problematic for arrays and strings of variable size because the array must be declared large enough to accommodate the largest size the program will ever see. An alternative is to *dynamically* allocate memory at run time when the actual size is known.

The `malloc` function from `stdlib.h` allocates a block of memory of a specified size and returns a pointer to it. If not enough memory is available, it returns a NULL pointer instead. For example, the following code allocates 10 shorts ($10 \times 2 = 20$ bytes). The `sizeof` operator returns the size of a type or variable in bytes.

```
// dynamically allocate 20 bytes of memory
short *data = malloc(10*sizeof(short));
```

C Code Example eC.38 illustrates dynamic allocation and de-allocation. The program accepts a variable number of inputs, stores them in a dynamically allocated array, and computes their average. The amount of memory necessary depends on the number of elements in the array and the size of each element. For example, if an `int` is a 4-byte variable and 10 elements are needed, 40 bytes are dynamically allocated. The `free` function de-allocates the memory so that it could later be used for other purposes. Failing to de-allocate dynamically allocated data is called a *memory leak* and should be avoided.

C Code Example eC.38 DYNAMIC MEMORY ALLOCATION AND DE-ALLOCATION

```
// Dynamically allocate and de-allocate an array using malloc and free
#include <stdlib.h>

// Insert getMean function from C Code Example eC.24.

int main(void) {
  int len, i;
```

```
  int *nums;

  printf("How many numbers would you like to enter?    ");
  scanf("%d", &len);
  nums = malloc(len*sizeof(int));
  if (nums == NULL) printf("ERROR: out of memory.\n");
  else {
    for (i=0; i<len; i++) {
      printf("Enter number:    ");
      scanf("%d", &nums[i]);
    }
    printf("The average is %f\n", getMean(nums, len));
  }
  free(nums);
}
```

C.8.8 Linked Lists*

A *linked list* is a common data structure used to store a variable number of elements. Each element in the list is a structure containing one or more data fields and a link to the next element. The first element in the list is called the *head*. Linked lists illustrate many of the concepts of structures, pointers, and dynamic memory allocation.

C Code Example eC.39 describes a linked list for storing computer user accounts to accommodate a variable number of users. Each user has a user name, a password, a unique user identification number (UID), and a field indicating whether they have administrator privileges. Each element of the list is of type `userL`, containing all of this user information along with a link to the next element in the list. A pointer to the head of the list is stored in a global variable called `users`, and is initially set to NULL to indicate that there are no users.

The program defines functions to insert, delete, and find a user and to count the number of users. The `insertUser` function allocates space for a new list element and adds it to the head of the list. The `deleteUser` function scans through the list until the specified UID is found and then removes that element, adjusting the link from the previous element to skip the deleted element and freeing the memory occupied by the deleted element. The `findUser` function scans through the list until the specified UID is found and returns a pointer to that element, or NULL if the UID is not found. The `numUsers` function counts the number of elements in the list.

C Code Example eC.39 LINKED LIST

```
#include <stdlib.h>
#include <string.h>

typedef struct userL {
```

```
  char uname[80];     // user name
  char passwd[80];    // password
  int uid;            // user identification number
  int admin;          // 1 indicates administrator privileges
  struct userL *next;
} userL;

userL *users = NULL;

void insertUser(char *uname, char *passwd, int uid, int admin) {
  userL *newUser;

  newUser = malloc(sizeof(userL));  // create space for new user
  strcpy(newUser->uname, uname);    // copy values into user fields
  strcpy(newUser->passwd, passwd);
  newUser->uid = uid;
  newUser->admin = admin;
  newUser->next = users;              // insert at start of linked list
  users = newUser;
}

void deleteUser(int uid) {   // delete first user with given uid
  userL *cur = users;
  userL *prev = NULL;

  while (cur != NULL) {
    if (cur->uid == uid) { // found the user to delete
      if (prev == NULL) users = cur->next;
      else prev->next = cur->next;
      free(cur);
      return; // done
    }
    prev = cur;        // otherwise, keep scanning through list
    cur = cur->next;
  }
}

userL *findUser(int uid) {
  userL *cur = users;

  while (cur != NULL) {
    if (cur->uid == uid) return cur;
    else cur = cur->next;
  }
  return NULL;
}

int numUsers(void) {
  userL *cur = users;
  int count = 0;

  while (cur != NULL) {
    count++;
    cur = cur->next;
  }
  return count;
}
```

SUMMARY

- **Pointers:** A pointer holds the address of a variable.
- **Arrays:** An array is a list of similar elements declared using square brackets `[]`.
- **Characters:** `char` types can hold small integers or special codes for representing text or symbols.
- **Strings:** A string is an array of characters ending with the null terminator 0x00.
- **Structures:** A structure stores a collection of related variables.
- **Dynamic memory allocation:** `malloc` is a built-in functions for allocating memory as the program runs. `free` de-allocates the memory after use.
- **Linked Lists:** A linked list is a common data structure for storing a variable number of elements.

C.9 STANDARD LIBRARIES

Programmers commonly use a variety of standard functions, such as printing and trigonometric operations. To save each programmer from having to write these functions from scratch, C provides *libraries* of frequently used functions. Each library has a header file and an associated object file, which is a partially compiled C file. The header file holds variable declarations, defined types, and function prototypes. The object file contains the functions themselves and is linked at compile-time to create the executable. Because the library function calls are already compiled into an object file, compile time is reduced. Table eC.5 lists some of the most frequently used C libraries, and each is described briefly below.

C.9.1 stdio

The standard input/output library `stdio.h` contains commands for printing to a console, reading keyboard input, and reading and writing files. To use these functions, the library must be included at the top of the C file:

```
#include <stdio.h>
```

printf

The *print formatted* statement `printf` displays text to the console. Its required input argument is a string enclosed in quotes `" "`. The string contains text and optional commands to print variables. Variables to be printed are listed after the string and are printed using format codes shown in Table eC.6. C Code Example eC.40 gives a simple example of `printf`.

Table eC.5 Frequently used C libraries

C Library Header File	Description
`stdio.h`	**Standard input/output library.** Includes functions for printing or reading to/from the screen or a file (`printf`, `fprintf` and `scanf`, `fscanf`) and to open and close files (`fopen` and `fclose`).
`stdlib.h`	**Standard library.** Includes functions for random number generation (`rand` and `srand`), for dynamically allocating or freeing memory (`malloc` and `free`), terminating the program early (`exit`), and for conversion between strings and numbers (`atoi`, `atol`, and `atof`).
`math.h`	**Math library.** Includes standard math functions such as `sin`, `cos`, `asin`, `acos`, `sqrt`, `log`, `log10`, `exp`, `floor`, and `ceil`.
`string.h`	**String library.** Includes functions to compare, copy, concatenate, and determine the length of strings.

Table eC.6 `printf` format codes for printing variables

Code	Format
`%d`	Decimal
`%u`	Unsigned decimal
`%x`	Hexadecimal
`%o`	Octal
`%f`	Floating point number (`float` or `double`)
`%e`	Floating point number (`float` or `double`) in scientific notation (e.g., 1.56e7)
`%c`	Character (`char`)
`%s`	String (null-terminated array of characters)

C Code Example eC.40 PRINTING TO THE CONSOLE USING `printf`

```
// Simple print function
#include <stdio.h>

int num = 42;
```

```
int main(void) {
  printf("The answer is %d.\n", num);
}
```

Console Output:

```
The answer is 42.
```

Floating point formats (`floats` and `doubles`) default to printing six digits after the decimal point. To change the precision, replace `%f` with `%w.df`, where `w` is the minimum width of the number, and `d` is the number of decimal places to print. Note that the decimal point is included in the width count. In C Code Example eC.41, `pi` is printed with a total of four characters, two of which are after the decimal point: 3.14. `e` is printed with a total of eight characters, three of which are after the decimal point. Because it only has one digit before the decimal point, it is padded with three leading spaces to reach the requested width. `c` should be printed with five characters, three of which are after the decimal point. But it is too wide to fit, so the requested width is overridden while retaining the three digits after the decimal point.

C Code Example eC.41 FLOATING POINT NUMBER FORMATS FOR PRINTING

```
// Print floating point numbers with different formats
float pi = 3.14159, e = 2.7182, c = 2.998e8;
printf("pi = %4.2f\ne  = %8.3f\nc  = %5.3f\n", pi, e, c);
```

Console Output:

```
pi = 3.14
e  =    2.718
c  = 299800000.000
```

Because `%` and `\` are used in print formatting, to print these characters themselves, you must use the special character sequences shown in C Code Example eC.42.

C Code Example eC.42 PRINTING `%` AND `\` USING `printf`

```
// How to print % and \ to the console
printf("Here are some special characters: %% \\ \n");
```

Console Output:

```
Here are some special characters: % \
```

scanf

The scanf function reads text typed on the keyboard. It uses format codes in the same way as printf. C Code Example eC.43 shows how to use scanf. When the scanf function is encountered, the program waits until the user types a value before continuing execution. The arguments to scanf are a string indicating one or more format codes and pointers to the variables where the results should be stored.

C Code Example eC.43 READING USER INPUT FROM THE KEYBOARD WITH scanf

```
// Read variables from the command line
#include <stdio.h>

int main(void)
{
  int a;
  char str[80];
  float f;

  printf("Enter an integer.\n");
  scanf("%d", &a);
  printf("Enter a floating point number.\n");
  scanf("%f", &f);
  printf("Enter a string.\n");
  scanf("%s", str);   // note no & needed: str is a pointer
}
```

File Manipulation

Many programs need to read and write files, either to manipulate data already stored in a file or to log large amounts of information. In C, the file must first be opened with the fopen function. It can then be read or written with fscanf or fprintf in a way analogous to reading and writing to the console. Finally, it should be closed with the fclose command.

The fopen function takes as arguments the file name and a *print mode*. It returns a *file pointer* of type FILE*. If fopen is unable to open the file, it returns NULL. This might happen when one tries to read a nonexistent file or write a file that is already opened by another program. The modes are:

"w": Write to a file. If the file exists, it is overwritten.

"r": Read from a file.

"a": Append to the end of an existing file. If the file doesn't exist, it is created.

C Code Example eC.44 shows how to open, print to, and close a file. It is good practice to always check if the file was opened successfully and to provide an error message if it was not. The `exit` function will be discussed in Section C.9.2. The `fprintf` function is like `printf` but it also takes the file pointer as an input argument to know which file to write. `fclose` closes the file, ensuring that all of the information is actually written to disk and freeing up file system resources.

C Code Example eC.44 PRINTING TO A FILE USING `fprintf`

```
// Write "Testing file write." to result.txt
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  FILE *fptr;

  if ((fptr = fopen("result.txt", "w")) == NULL) {
    printf("Unable to open result.txt for writing.\n");
    exit(1);   // exit the program indicating unsuccessful execution
  }
  fprintf(fptr, "Testing file write.\n");
  fclose(fptr);
}
```

It is idiomatic to open a file and check if the file pointer is NULL in a single line of code, as shown in C Code Example eC.44. However, you could just as easily separate the functionality into two lines:

```
fptr = fopen("result.txt", "w");
if (fptr == NULL)
  ...
```

C Code Example eC.45 illustrates reading numbers from a file named data.txt using `fscanf`. The file must first be opened for reading. The program then uses the `feof` function to check if it has reached the end of the file. As long as the program is not at the end, it reads the next number and prints it to the screen. Again, the program closes the file at the end to free up resources.

C Code Example eC.45 READING INPUT FROM A FILE USING `fscanf`

```
#include <stdio.h>

int main(void)
{
  FILE *fptr;
  int data;

  // read in data from input file
  if ((fptr = fopen("data.txt", "r")) == NULL) {
    printf("Unable to read data.txt\n");
    exit(1);
  }
```

```
  while (!feof(fptr)) {   // check that the end of the file hasn't been reached
    fscanf(fptr, "%d", &data);
    printf("Read data: %d\n", data);
  }

  fclose(fptr);
}
```

data.txt

```
25 32 14 89
```

Console Output:

```
Read data: 25
Read data: 32
Read data: 14
Read data: 89
```

Other Handy stdio Functions

The `sprintf` function prints characters into a string, and `sscanf` reads variables from a string. The `fgetc` function reads a single character from a file, while `fgets` reads a complete line into a string.

`fscanf` is rather limited in its ability to read and parse complex files, so it is often easier to `fgets` one line at a time and then digest that line using `sscanf` or with a loop that inspects characters one at a time using `fgetc`.

C.9.2 stdlib

The standard library `stdlib.h` provides general purpose functions including random number generation (`rand` and `srand`), dynamic memory allocation (`malloc` and `free`, already discussed in Section C.8.7), exiting the program early (`exit`), and number format conversions. To use these functions, add the following line at the top of the C file.

```
#include <stdlib.h>
```

rand and srand

`rand` returns a pseudo-random integer. Pseudo-random numbers have the statistics of random numbers but follow a deterministic pattern starting with an initial value called the *seed*. To convert the number to a particular range, use the modulo operator (`%`) as shown in C Code Example eC.46 for a range of 0 to 9. The values `x` and `y` will be random but they will be the same each time this program runs. Sample console output is given below the code.

C Code Example eC.46 RANDOM NUMBER GENERATION USING `rand`

```
#include <stdlib.h>
int x, y;

x = rand();        // x = a random integer
y = rand() % 10;   // y = a random number from 0 to 9
printf("x = %d, y = %d\n", x, y);
```

Console Output:

```
x = 1481765933, y = 3
```

A programmer creates a different sequence of random numbers each time a program runs by changing the seed. This is done by calling the `srand` function, which takes the seed as its input argument. As shown in C Code Example eC.47, the seed itself must be random, so a typical C program assigns it by calling the `time` function, that returns the current time in seconds.

C Code Example eC.47 SEEDING THE RANDOM NUMBER GENERATOR USING `srand`

```
// Produce a different random number each run
#include <stdlib.h>
#include <time.h>    // needed to call time()

int main(void)
{
  int x;

  srand(time(NULL));    // seed the random number generator
  x = rand() % 10;      // random number from 0 to 9
  printf("x = %d\n", x);
}
```

For historical reasons, the `time` function usually returns the current time in seconds relative to January 1, 1970 00:00 UTC. UTC stands for Coordinated Universal Time, which is the same as Greenwich Mean Time (GMT). This date is just after the UNIX operating system was created by a group at Bell Labs, including Dennis Ritchie and Brian Kernighan, in 1969. Similar to New Year's Eve parties, some UNIX enthusiasts hold parties to celebrate significant values returned by `time`. For example, on February 1, 2009 at 23:31:30 UTC, `time` returned 1,234,567,890. In the year 2038, 32-bit UNIX clocks will overflow into the year 1901.

`exit`

The `exit` function terminates a program early. It takes a single argument that is returned to the operating system to indicate the reason for termination. 0 indicates normal completion, while nonzero conveys an error condition.

Format Conversion: `atoi`, `atol`, `atof`

The standard library provides functions for converting ASCII strings to integers, long integers, or doubles using `atoi`, `atol`, and `atof`, respectively, as shown in C Code Example eC.48. This is particularly useful

when reading in mixed data (a mix of strings and numbers) from a file or when processing numeric command line arguments, as described in Section C.10.3.

C Code Example eC.48 FORMAT CONVERSION

```
// Convert ASCII strings to ints, longs, and floats
#include <stdlib.h>

int main(void)
{
  int x;
  long int y;
  double z;

  x = atoi("42");
  y = atol("833");
  z = atof("3.822");

  printf("x = %d\ty = %d\tz = %f\n", x, y, z);
}
```

Console Output:

```
x = 42   y = 833   z = 3.822000
```

C.9.3 math

The math library `math.h` provides commonly used math functions such as trigonometry functions, square root, and logs. C Code Example eC.49 shows how to use some of these functions. To use math functions, place the following line in the C file:

```
#include <math.h>
```

C Code Example eC.49 MATH FUNCTIONS

```
// Example math functions
#include <stdio.h>
#include <math.h>

int main(void) {
  float a, b, c, d, e, f, g, h;

  a = cos(0);          // 1, note: the input argument is in radians
  b = 2 * acos(0);     // pi (acos means arc cosine)
  c = sqrt(144);       // 12
  d = exp(2);          // e^2 = 7.389056,
  e = log(7.389056);   // 2 (natural logarithm, base e)
```

```
  f = log10(1000);     // 3 (log base 10)
  g = floor(178.567); // 178, rounds to next lowest whole number
  h = pow(2, 10);      // computes 2 raised to the 10th power

  printf("a = %.0f, b = %f, c = %.0f, d = %.0f, e = %.2f, f = %.0f, g = %.2f, h = %.2f\n",
         a, b, c, d, e, f, g, h);
}
```

Console Output:

```
a = 1, b = 3.141593, c = 12, d = 7, e = 2.00, f = 3, g = 178.00, h = 1024.00
```

C.9.4 string

The string library `string.h` provides commonly used string manipulation functions. Key functions include:

```
// copy src into dst and return dst
char *strcpy(char *dst, char *src);

// concatenate (append) src to the end of dst and return dst
char *strcat(char *dst, char *src);

// compare two strings. Return 0 if equal, nonzero otherwise
int strcmp(char *s1, char *s2);

// return the length of str, not including the null termination
int strlen(char *str);
```

C.10 COMPILER AND COMMAND LINE OPTIONS

Although we have introduced relatively simple C programs, real-world programs can consist of tens or even thousands of C files to enable modularity, readability, and multiple programmers. This section describes how to compile a program spread across multiple C files and shows how to use compiler options and command line arguments.

C.10.1 Compiling Multiple C Source Files

Multiple C files are compiled into a single executable by listing all file names on the compile line as shown below. Remember that the group of C files still must contain only one `main` function, conventionally placed in a file named `main.c`.

```
gcc main.c file2.c file3.c
```

C.10.2 Compiler Options

Compiler options allow the programmer to specify such things as output file names and formats, optimizations, etc. Compiler options are not

Table eC.7 Compiler options

Compiler Option	Description	Example
`-o` outfile	specifies output file name	`gcc -o hello hello.c`
`-S`	create assembly language output file (not executable)	`gcc -S hello.c` this produces `hello.s`
`-v`	verbose mode – prints the compiler results and processes as compilation completes	`gcc -v hello.c`
`-Olevel`	specify the optimization level (level is typically 0 through 3), producing faster and/or smaller code at the expense of longer compile time	`gcc -O3 hello.c`
`--version`	list the version of the compiler	`gcc -version`
`--help`	list all command line options	`gcc --help`
`-Wall`	print all warnings	`gcc -Wall hello.c`

standardized, but Table eC.7 lists ones that are commonly used. Each option is typically preceded by a dash (`-`) on the command line, as shown. For example, the `"-o"` option allows the programmer to specify an output file name other than the `a.out` default. A plethora of options exist; they can be viewed by typing `gcc --help` at the command line.

C.10.3 Command Line Arguments

Like other functions, `main` can also take input variables. However, unlike other functions, these arguments are specified at the command line. As shown in C Code Example eC.50, `argc` stands for *argument count*, and it denotes the number of arguments on the command line. `argv` stands for *argument vector*, and it is an array of the strings found on the command line. For example, suppose the program in C Code Example eC.50 is compiled into an executable called `testargs`. When the lines below are typed at the command line, `argc` has the value 4, and the array `argv` has the values `{"./testargs", "arg1", "25", "lastarg!"}`. Note that the executable name is counted as the 1st argument. The console output after typing this command is shown below C Code Example eC.50.

```
gcc -o testargs testargs.c
./testargs arg1 25 lastarg!
```

Programs that need numeric arguments may convert the string arguments to numbers using the functions in `stdlib.h`.

C Code Example eC.50 COMMAND LINE ARGUMENTS

```
// Print command line arguments

#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  for (i=0; i<argc; i++)
    printf("argv[%d] = %s\n", i, argv[i]);
}
```

Console Output:

```
argv[0] = ./testargs
argv[1] = arg1
argv[2] = 25
argv[3] = lastarg!
```

C.11 COMMON MISTAKES

As with any programming language, you are almost certain to make errors while you write nontrivial C programs. Below are descriptions of some common mistakes made when programming in C. Some of these errors are particularly troubling because they compile but do not function as the programmer intended.

C Code Mistake eC.1 MISSING & IN `scanf`

Erroneous Code

```
int a;
printf("Enter an integer:\t");
scanf("%d", a); // missing & before a
```

Corrected Code:

```
int a;
printf("Enter an integer:\t");
scanf("%d", &a);
```

C Code Mistake eC.2 USING = INSTEAD OF == FOR COMPARISON

Erroneous Code

```
if (x = 1)    // always evaluates as TRUE
  printf("Found!\n");
```

Corrected Code

```
if (x == 1)
  printf("Found!\n");
```

Debugging skills are acquired with practice, but here are a few hints.

- Fix bugs starting with the first error indicated by the compiler. Later errors may be downstream effects of this error. After fixing that bug, recompile and repeat until all bugs (at least those caught by the compiler!) are fixed.
- When the compiler says a valid line of code is in error, check the code above it (i.e., for missing semicolons or braces).
- When needed, split up complicated statements into multiple lines.
- Use `printf` to output intermediate results.
- When a result doesn't match expectations, start debugging the code at the *first* place it deviates from expectations.
- Look at all compiler warnings. While some warnings can be ignored, others may alert you to more subtle code errors that will compile but not run as intended.

C Code Mistake eC.3 INDEXING PAST LAST ELEMENT OF ARRAY

Erroneous Code

```
int array[10];
array[10] = 42;    // index is 0-9
```

Corrected Code

```
int array[10];
array[9] = 42;
```

C Code Mistake eC.4 USING = IN #define STATEMENT

Erroneous Code

```
// replaces NUM with "= 4" in code
#define NUM = 4
```

Corrected Code

```
#define NUM 4
```

C Code Mistake eC.5 USING AN UNINITIALIZED VARIABLE

Erroneous Code

```
int i;
if (i == 10) // i is uninitialized
  ...
```

Corrected Code

```
int i = 10;
if (i == 10)
  ...
```

C Code Mistake eC.6 NOT INCLUDING PATH OF USER-CREATED HEADER FILES

Erroneous Code

```
#include    "myfile.h"
```

Corrected Code

```
#include    "othercode\myfile.h"
```

C Code Mistake eC.7 USING LOGICAL OPERATORS (!, ||, &&) INSTEAD OF BITWISE (~, |, &)

Erroneous Code

```
char x=!5;      // logical NOT: x = 0
char y=5||2;    // logical OR:  y = 1
char z=5&&2;    // logical AND: z = 1
```

Corrected Code

```
char x=~5; // bitwise NOT: x = 0b11111010
char y=5|2;// bitwise OR:  y = 0b00000111
char z=5&2;// logical AND: z = 0b00000000
```

C Code Mistake eC.8 FORGETTING `break` IN A `switch/case` STATEMENT

Erroneous Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1;
  case 'd': direction = 2;
  case 'l': direction = 3;
  case 'r': direction = 4;
  default: direction = 0;
}
// direction = 0
```

Corrected Code

```
char x = 'd';
...
switch (x) {
  case 'u': direction = 1; break;
  case 'd': direction = 2; break;
  case 'l': direction = 3; break;
  case 'r': direction = 4; break;
  default: direction = 0;
}
// direction = 2
```

C Code Mistake eC.9 MISSING CURLY BRACES { }

Erroneous Code

```
if (ptr == NULL)  // missing curly braces
  printf("Unable to open file.\n");
  exit(1);        // always executes
```

Corrected Code

```
if (ptr == NULL) {
  printf("Unable to open file.\n");
  exit(1);
}
```

C Code Mistake eC.10 USING A FUNCTION BEFORE IT IS DECLARED

Erroneous Code

```
int main(void)
{
  test();
}

void test(void)
{...
}
```

Corrected Code

```
void test(void)
{...
}

int main(void)
{
  test();
}
```

C Code Mistake eC.11 DECLARING A LOCAL AND GLOBAL VARIABLE WITH THE SAME NAME

Erroneous Code

```
int x = 5;      // global declaration of x
int test(void)
{
  int x = 3;   // local declaration of x
  ...
}
```

Corrected Code

```
int x = 5;      // global declaration of x
int test(void)
{
  int y = 3;   // local variable is y
  ...
}
```

C Code Mistake eC.12 TRYING TO INITIALIZE AN ARRAY WITH {} AFTER DECLARATION

Erroneous Code

```
int scores[3];
scores = {93, 81, 97}; // won't compile
```

Corrected Code

```
int scores[3] = {93, 81, 97};
```

C Code Mistake eC.13 ASSIGNING ONE ARRAY TO ANOTHER USING =

Erroneous Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

scores2 = scores;
```

Corrected Code

```
int scores[3] = {88, 79, 93};
int scores2[3];

for (i=0; i<3; i++)
  scores2[i] = scores[i];
```

C Code Mistake eC.14 FORGETTING THE SEMI-COLON AFTER A do/while LOOP

Erroneous Code

```
int num;
do {
  num = getNum();
} while (num < 100)    // missing ;
```

Corrected Code

```
int num;
do {
  num = getNum();
} while (num < 100);
```

C Code Mistake eC.15 USING COMMAS INSTEAD OF SEMICOLONS IN for LOOP

Erroneous Code

```
for (i=0, i < 200, i++)
  ...
```

Corrected Code

```
for (i=0; i < 200; i++)
  ...
```

C Code Mistake eC.16 INTEGER DIVISION INSTEAD OF FLOATING POINT DIVISION

Erroneous Code

```
// integer (truncated) division occurs when
// both arguments of division are integers
float x = 9 / 4;  // x = 2.0
```

Corrected Code

```
// at least one of the arguments of
// division must be a float to
// perform floating point division
float x = 9.0 / 4;  // x = 2.25
```

C Code Mistake eC.17 WRITING TO AN UNINITIALIZED POINTER

Erroneous Code

```
int *y = 77;
```

Corrected Code

```
int x, *y = &x;
*y = 77;
```

C Code Mistake eC.18 GREAT EXPECTATIONS (OR LACK THEREOF)

A common beginner error is to write an entire program (usually with little modularity) and expect it to work perfectly the first time. For non-trivial programs, writing modular code and testing the individual functions along the way are essential. Debugging becomes exponentially harder and more time-consuming with complexity.

Another common error is lacking expectations. When this happens, the programmer can only verify that the code produces a result, not that the result is correct. Testing a program with known inputs and expected results is critical in verifying functionality.

This appendix has focused on using C on a system such as a personal computer. Chapter 9 (available as a web supplement) describes how C is used to program an ARM-based Raspberry Pi computer that can be used in embedded systems. Microcontrollers are usually programmed in C because the language provides nearly as much low-level control of the hardware as assembly language, yet is much more succinct and faster to write.

Index

Note: Page numbers in *italics* indicate figures, tables and text boxes; page numbers preceded by "e" refer to online material.

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z