Excerpt from: Stephen H. Unger, *The Essence of Logic Circuits*, Second Ed., Wiley, 1997

APPENDIX

A.1 Number systems and codes

Since ten-fingered humans are addicted to the decimal system, and since computers operate most efficiently in binary, it is necessary to understand how both integers and fractions can be translated from one system to the other. Certainly those involved in the design of computers must understand how to do arithmetic in the binary system. But computers operate on text as well as on numbers. This motivated the development of American Standard Code for Information Exchange (ASCII) and other standard coding schemes that can specify alphabetic characters and other non-numerical symbols in terms of 1s and 0s.

In addition to binary and decimal, we are sometimes interested in other number bases. For example, octal and hexadecimal are often used to represent strings of 0s and 1s compactly. We begin with a discussion of number systems in general.

A.1.1 Number Systems

When an integer is expressed in positional notation with base b, each numeral is weighted by a power of b, depending on its position. The rightmost, or *least significant*, digit is given weight $b_0 = 1$, and, in general, the ith digit from the right has weight b^{i-1}. Thus, in base b the number x would be expressed as $a_{n-1}a_{n-2}\ldots a_0$ (where the a_i-digits range from 0 through b-1) and we have the equation:

$$x = a_{n-1}b^{n-1}+a_{n-2}b^{n-2}+\ldots+b_0 \qquad (1)$$

Consider, for example, the number 7542 expressed in our usual decimal, or base-10, notation. In the notation just presented, we would express it as: $7x10^3+5x10^2+4x10^1+2x10^0$. The value of the numeral 4 is $4\ x10^1 = 40$, the value of the 7 is $7x10^3 = 7000$, etc.

For engineering reasons related to reliability and other factors, it has been found most effective to use two-valued signals within computers. Hence, it is best to represent numbers in base-2 notation; we refer to these as binary numbers.

Because so many problems in logic design, computer programming, and other aspects of the computer field involve powers of 2, it is very useful for people working in the area to be able to compute mentally the value of any power of 2. Fortunately this is not a difficult skill to acquire. One must first memorize the first 10 powers of 2 (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024). Note then that $2^{10} = 1024$ is within 2.5% of $1000 = 10^3$. Thus, to estimate the value of a fairly large power of 2, we can replace each multiple of 10 in the exponent of 2 by 10^3 in the result, and then multiply by the value of 2 raised to the power of the exponent modulo-10. This is easier done than said. For example, 2^{25} can be found quickly by noting, in effect, that it is equal to $2^{10} \times 2^{10} \times 2^5 =$ (approximately) $10^3 \times 10^3 \times 32 = 32$ million. In the computer field the abbreviation K is commonly used to mean 2^{10}, so that a 64K-memory chip contains $64x\ 2^{10}$ or 65,536 bits.

A.1.2 Arithmetic in Binary

Arithmetic operations in the binary number system are carried out the same way as in the more familiar decimal system. But the individual steps are easier to execute since there is only one non-zero numeral. Hence, only one operation each, for addition and multiplication need be memorized. Each of these operations (1x1 = 1 and 1 + 1 = 10)

involve two non-zero operands. One example of each of the four basic arithmetic operations is shown below.

1. . Addition

```
  110100110
+ 101111111
 1100100101
```

2. Subtraction

```
  1100011001
- 1011111011
       11110
```

3. Multiplication

```
    1011
  x 1101
    1011
   1011
  1011
 10001111
```

4. Division

```
      101
110 | 100010
    - 110
      1010
    - 110
      110 (the remainder)
```

Since computers usually operate in binary and people operate in decimal, it is necessary to be able to convert numbers between decimal and binary forms. Before discussing these specific conversions, it will be useful to consider the more general case of conversions between decimal and base-b numbers for arbitrary values of b.

A.1.3 Negative Numbers

Since digital computers do a great deal of arithmetic computation, negative numbers will often be encountered and we must have a convenient way of dealing with them. That is, we need a simple way to represent numbers as being either positive or negative.

The most obvious approach is to reserve one bit of a word representing a number as the sign bit. (Although real computers represent numbers with anywhere from 16 to 64 bits, the principles involved are easier to illustrate with a much smaller word. Thus, 4-bit words will be used here.) For example, if our computer had 4-bit words to represent numbers, then we might specify that all positive numbers have a 0 in the leftmost bit and that this bit be 1 for all negative numbers. The rightmost three bits would specify the magnitude of the number in binary, as discussed earlier. Thus, the number 5 would be specified as 0101 while -4 would be written as 1100. There are several drawbacks to this apparently simple scheme, commonly referred to as the sign-magnitude method. One is that if two arbitrary numbers are to be added, it is necessary to examine the signs of each

of them to determine whether to generate the sum or difference of the magnitudes, and a further logical operation is required to decide what sign to attach to the result. A second difficulty is that there are two representations of 0, since +0 = -0. In our four-bit example, both 0000 and 1000 represent 0. This complicates comparison operations, since if two numbers are being tested for equality, special provision must be made for the possibility that one is +0 and the other -0. For these reasons, most computers use different approaches, and by far the most common technique is the *two's complement* method presented below.

In order to understand this method, it is essential to have a firm grasp of a few simple points about binary number representations.

1. The most significant (leftmost) bit of an n-bit binary number has weight 2^{n-1}. Thus, the binary number 1000 represents the number $2^{4-1} = 2^3=8$.
2. With n bits, we can represent magnitudes ranging from 0 to 2^n-1. Thus, with four bits we can specify numbers from 0 (0000) through 15 (1111).
3. If the result of an arithmetic operation (say addition) exceeds the number of bits in the register provided for, it then is easy to arrange for the lower bits to go into the register with the excess upper bits simply being discarded. Thus, what appears in the register is the original number modulo-2^n. For example, if we try to stuff the binary number representing 37 (whose binary representation is 100101) into our four-bit register, what actually goes in is the number 0101 (5 in decimal notation), which is 37 modulo-16.

Finally, a word about notation. In our discussion, symbols such as k or k_i, will represent absolute values, i.e., positive numbers. Negative numbers will be written as (-k).

Now we can introduce two's complement numbers. For an n-bit system, the two's complement of a number k is defined as 2^n-k. Thus, for n = 4, the two's complement of 5 is $2^4-5 = 16-5 = 11$, which would be written as 1011. The basic idea is to represent negative numbers as the two's complements of their absolute values. Positive numbers are represented as the simple binary equivalents, so 5 would simply be 0101.

But then how do we distinguish between positive and negative numbers? In the preceding example, how do we know that 1011 stands for -5 as opposed to 11? This is accomplished by recognizing that, given n bits, a total of 2^n different numbers (positive and negative) can be specified. We must partition this set between the non-negative numbers (positive or 0) and the negative numbers. This is done by reserving all of the codes with 0 for the most significant bit for the non-negative numbers, and all of the codes with 1 in the first position for the negative numbers, the latter being represented by the two's complements of their magnitudes. Thus, as in sign-magnitude notation, the leftmost bit serves as a sign bit. This works out very nicely because the binary representations of all of the non-negative numbers from 0 through $2^{n-1}-1$ have 0 for the most significant bit, and the two's complements of the numbers from -1 through -2^{n-1} all have 1 for the most significant bits. This is illustrated in the table of Figure A.1 for the case of n = 4.

Decimal Number	Two's Complement Representation
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Figure A.1 Two's complement system for four-bit numbers.

Readers should examine this table carefully to verify the various statements made earlier. Note that all 16 codes have been allocated. There is only one representation of 0. However, since 0 takes up one code, this leaves an odd number of codes to be allocated between the positive and negative numbers. In this system, the extra code goes to the negative numbers in that (-2^{n-1}) is represented, but 2^{n-1} is not represented. In our four-bit example, -8 appears, but 8 does not appear. This asymmetry is an inevitable price of insisting on a single representation of 0. More about this later.

Let us now see how the system works out with respect to arithmetic operations. First we note that a number, whether positive or negative can be negated (i.e., its sign changed) by taking its two's complement. This is obviously true by definition for positive numbers. Suppose now we have the two's complement representation of (-k), which would be 2^n-k. If we take the two's complement of this, we obtain $2^n-(2^n-k) = k$, which is as it should be. The case of 0 merits special consideration. What is the two's complement of 0? Our procedure specifies this as $2^n-0 = 2^n$. At first, this looks like trouble. But then we note that 2^n exceeds the maximum number that can be represented by n bits, and so, as noted at the outset of this discussion (item 3), the actual number that will be stored is the calculated number modulo-2^n, which, in this case is 2^n modulo-2^n, or simply 0. Thus, 0 is its own two's complement, and all is well.

Now consider what happens when two's complement numbers are added. Adding two non-negative numbers produces another non-negative number, provided that the sum does not exceed 2^{n-1}-1 (which we specified as the largest allowable non-negative number). For example, adding 0100 to 0010 yields 0110. (The situation in which the sum is too large, known as an overflow, is treated at a later point.)

Suppose two negative numbers $(-k_1)$, and $(-k_2)$ expressed in two's complement are added. This is equivalent to finding $2^n-k_1+2^n-k_2 = 2^n+2^n-(k_1+k_2)$. Since what actually appears in an n-bit register is the result modulo-2^n, this is equivalent to $2^n-(k_1+k_2)$, which is the two's

complement of $(-k_1)+(-k_2)$, assuming k_1+k_2 does not exceed 2^n (which would be another instance of overflow). As an example, consider, for our four-bit word size, adding -2 and -3. In two's complement form this would be equivalent to (16-2)+(16-3) = 16+16-(+2+3), which, modulo-2^4, is equal to 16-(2+3) = 16-(5) = 11, the two's complement version of -5.

If a non-negative number k_1 is added to a negative number $(-k_2)$, where $k_1<k_2$, the result would be $k_1+(2^n-k_2) = 2^n-(k_2-k_1)$ which is the two's complement representation of the correct result, a negative number. For example, 1+(-6) would correspond to 1+(24-6) = 24-(6-1) = 24-5, the two's complement form of -5. If $k_1 \geq k_2$, then the result can be expressed as $2^n+(k_1-k_2)$, which is the correct answer, a positive number, modulo-2^n. An instance of this situation is the sum 5+(-3), which in two's complement form, corresponds to $5+(2^4-3) = 2^4+(5-3) = 2^4+2 = 2$ modulo-2^4. No overflow is possible when numbers of opposite signs are added.

Thus, in all cases, correct results are obtained. Let us now consider how to compute the two's complement of an n-bit binary number k. If we have an arithmetic unit that can perform subtraction, the simplest approach is to generate 0-k. This is clearly equal, modulo-2^n, to 2^n-k. For example, in binary, with n = 4, 0000-0101 = 1011. The carry out from the leftmost position is simply ignored.

An approach that does not require subtraction is based on the fact that the binary representation of 2^{n-1} consists of a string of n 1s. For example, $2^4-1 = 15$ is written in binary as 1111. If we subtract k from 2^n-1, the effect is simply to complement each bit of the binary form of k. In other words, all that is necessary to convert the binary representation of k to the representation of 2^{n-1}-k is to complement each bit. Thus, 1111-0101 = 1010. But this number is just 1 less than the two's complement. Hence, we can obtain the two's complement of a binary number by complementing each bit of the number and then incrementing the result by 1. Thus, to obtain the two's complement of 6 in our four-bit system, we go from 0110 to 1001 to 1010. We will sometimes use the notation TC(k) to represent the two's complement of k. Thus, for the preceding example, we might write TC(0110) = 1010. Of course, this works both ways, i.e., TC(1010) = 0110.

Another method is based on the rule for incrementing by 1 (see the discussion of counting on p. 201). Starting at the least significant end, all bits up to and including the first 0 encountered are complemented, and all of the more significant bits are left unchanged. Suppose we apply this to the complement-increment technique. Because of the complementing step, the incrementing operation restores to the original values all of the bits starting with the least significant (rightmost) 1 of the original string. It inverts all of the bits to the left of this bit. So the rule is simply to complement all bits to the left of the rightmost 1 in the number. For example, applying this idea to 0110, means that we should complement the two leftmost bits, yielding 1010. This technique allows us to generate two's complement s very easily by inspection. It also lends itself to implementation by relatively simple iterative circuits (see Sections 5.2 and 5.3 and Problem A.9).

If a string of 1s and 0s represents an unsigned binary number, the magnitude of this number can be calculated, as pointed out in Section A.1.1, by adding the weights corresponding to each position containing a 1, where these weights are 2^{i-1} for the ith bit

from the right. That is, the weight of the rightmost position is 1, the next position has weight 2, the next 4, etc. A careful examination will show that if the same string of bits is considered to represent a signed two's complement number, the only change that need be made is to change sign of the weight of the leftmost bit (the sign bit) to a minus sign. This does not change the value assigned to positive numbers, since the sign bit for a positive number is 0. But the value of a negative number is changed, negatively, by twice the weight of the leftmost bit (which is 2^{n-1}). Twice this number is 2^n, so in effect we are converting 2^n-k to (2^n-k)-2^n = -k, the desired result. Thus, for the four-bit system, the number represented in the two's complement realm by the sequence 1010 can be calculated as $-2^3+2^1 = -6$.

As indicated earlier, the magnitude of the largest positive integer in an n-bit system is 2^{n-1}-1, and the largest magnitude of a negative number is 2^{n-1}. What happens if, as a result of an addition of two valid numbers, the result is out of the permissible range? In our four-bit system, this would occur as a result of the operation 5+4. In virtually all computers, such an event, called an overflow, is detected by the hardware, so that programmers have the opportunity to specify what, if any, remedial action should be taken. In the two's complement system, there is a relatively simple solution to the overflow detection problem. It is based on carries in and out of the sign bit (the leftmost bit). Let us begin with the addition of positive integers.

Since the sign bit of a positive numbers is 0, when we add two such numbers, there can never be a carry out of this position. If, however, the result of the addition exceeds the limit of 2^{n-1}-1, there will be a carry out of the n-1th bit into the sign-bit position. For example, elaborating on the 5+4 example, we have 0101+0100 = 1001. (The value of the sign bit was changed to 1 by the carry.) Thus, for the addition of two positive numbers, there is overflow if and only if there is a carry *into* the sign-bit position, but no carry *out* of it. Consider next the adding of two negative numbers.

Since the sign bit of a negative number is 1, adding two such numbers always results in a carry out of the sign-bit position. If the result of the addition is a valid number, then the sign bit of the result will also be 1. This can occur only if there was a carry into this position. Thus, for the valid addition (-3)+(-5) = (-8), we have 1101+1011 = 1000. An example of such addition with overflow is: (-4)+(-5), which corresponds to 1100+1011 = 10111. In the overflow case, there is no overflow into the sign-bit position. So for the case of addition of negative numbers, there is overflow if, and only if there is a carry out of the sign-bit position, but no carry into it. Now consider adding a positive and a negative number. Since one of the sign bits is a 1 and the other a 0, there will be a carry out of the sign position if and only if there is a carry into it. For the example (-3)+(5), corresponding to 1101+0101 = 0010, there is a carry in and a carry out of the sign position. On the other hand, for the case of (3)+(-5), corresponding to 0011+1011 = 1110, there is no carry in and no carry out of the sign position. For the sum of a positive and negative number, there can never be any overflow.

Summarizing, then, we see that all cases are covered by the following rule: For the addition of any two valid numbers expressed in the two's complement system, there is

overflow if and only if the carry out is not equal to the carry in to the sign-bit position. This condition can be detected by an XOR-gate with inputs corresponding to the carries in and out of the leftmost bit.

One more situation must be considered. In our initial discussion of two's complements, it was pointed out that there is a valid negative number without a matching valid positive number with the same absolute value. In general, this is (-2^{n-1}). For our n = 4 example, this number is -8 (see Figure A.1). This is a perfectly valid number, behaving the same way as all other negative numbers in addition operations. What happens if we attempt to negate this number, that is execute the operation $-(-2^{n-1})$? The result would be a positive number out of range, since 2^{n-1} is not expressible in an n-bit two's complement system. If the subtraction method is used, then it is easy to see that there will be a carry out of the sign position, but no carry in to it. If the two's complementing operation, which is how negation is accomplished, is carried out using the complement-increment method, the result of the incrementing part is a carry in to the sign-bit position, but no carry out. If the iterative circuit method is used, it is not hard to see that a similar simple detection method exists. In all cases, the behavior of 2^{n-1} is unique with respect to the carries involving the sign bit, so that detection is easily achieved.
When a computer has 32-bit words, it is extremely unlikely that the number 2^{n-1} will be encountered in any particular operation. (What are the odds?) But this number *will* occasionally appear and be negated. Unless this rare event is provided for, it may result in unexpected and perhaps catastrophic consequences. This is precisely the kind of situation that causes embarrassing, costly, and sometimes tragic computer failures.

A.1.4 Integer Conversions to Base 10

Consider first conversions from base b to base 10. Working in base-10 arithmetic, this is a straightforward process. We simply apply Equation (1). For example, to convert the base-3 (ternary) number 2102 (the base is often indicated by a subscript, e.g., 2102_3) to decimal, we obtain $2x3^3 + 1x3^2 + 2x3^0 = 65$. For an n-digit number, this process requires a maximum of n(n-1)/2 multiplications. A more efficient procedure, requiring no more than n-1 multiplications, is derived by repeated factoring of the right side of equation (1). For the case of 2102, this yields

$((2x3 + 1)x3 + 0)x3 + 2 = 65.$

In the general case we have:

$$x = (...(((a_{n-1}b + a_{n-2})b + a_{n-3})b + a_{n-4})b + ... + a_1)b + a_0 \qquad (2)$$

This procedure is easily executed starting from within the innermost parentheses and working out, i.e., multiply the leftmost digit of x by b, then add the next digit of x, then multiply the result by b, and continue alternately adding a digit of x and multiplying by b. The process terminates with the addition of a_0.

A.1.5 Conversions From Base 10

Consider now the inverse process, the conversion of a number from base 10 to base b. If we divide both sides of Equation (1) by b, an examination of the right side indicates that the remainder is a0, the least significant digit of x written in base-b notation. The quotient is in the same general form as the original equation, but now a1 is the rightmost term of the left side. Dividing the quotient by b therefore yields a1 as the remainder. Repeating this process generates all of the digits of the base-b representation of x, from

right to left. As an example, consider finding the base-8 (octal) representation of 97110. We have the following sequence of operations:.

971/8 = 121+3/8; 121/8= 15+1/8; 15/8 = 1+7/8; 1/8 = 0+1/8.

The octal representation of 971 is therefore 1713. (Verify this by using the method of Equation (2) to convert back to decimal.)

Note that all operations in this process are carried out in the decimal system, which corresponds to the base of the *original* number. The previous algorithm, for converting from base b to base 10 is also executed using decimal arithmetic-- in this case, the system of the *final* number. Thus, we now have methods generally applicable to any number base conversion, and we can choose to operate in the arithmetic of either of the two systems involved. Suppose it is essential (or very convenient) always to use one particular base, b*, for all such arithmetic, even where b* is not the base of either of the systems involved (e.g., b* might be 10 if this is to be a hand computation, or it might be 2 if it is to be executed by a computer). Then we might convert the original number into a base-b* number, using the division method, and then use the multiply and add method to convert from b* to the base of the final number. In both cases, we would be operating with base-b* arithmetic.

A.1.6 Fractions

In a base-b system, we express "radix" fraction (generalization of decimal fraction) x as: 0.a1 a2... an, where the ai are numbers between 0 and b-1, and where

$$x = a_1 b^{-1} + a_2 b^{-2} + \ ... + a_n b^{-n} \ . \qquad (3)$$

In the decimal system, multiplication by 10 is equivalent to moving the decimal point one position to the right, and division by 10 is equivalent to moving the decimal point one position to the left. Analogously, it is evident from (1) and (3) that for a number written in base b, multiplication or division by b has the effect of moving the radix point (the generalization of the decimal point) one step to the right or left respectively.

To convert the decimal fraction x to base b, we must identify the ai-coefficients in (3). The key point is to note that if the right side of (3) is multiplied by b, the result is a_1, an integer, added to a sum of fractions totaling less than 1. Hence, the *integer part* of the product of x and b is a_1. If we multiply the fractional part of this initial product by b, the result will be a_2, and if we continue this process, we generate digits of the base-b equivalent of x in decreasing order of significance (i.e., from left to right). The procedure just described is illustrated below by the generation of the binary equivalent of 0.32410 to three significant figures:

0.324x2 = 0.648; 0.648x2 = 1.296; 0.296x2 = 0.592; 0.592x2 = 1.184

Hence, we have 0.01012 as the desired result. (Strictly speaking, we should generate one extra digit and use it for rounding purposes. In this case, it is clear that the value of the next digit is 0, so our result is unchanged.)

Conversion of radix fraction y from base b to base 10 can be accomplished efficiently by factoring the right side of Equation (3) to obtain Equation (4) below (which is analogous to Equation (2)):

$$y = (...((\ a_k/b + a_{k-1})/b + a_{k-2})/b + ... + a_1)/b \qquad (4)$$

Note that, in contrast to the analogous procedure for integers based on (2), we start with the *rightmost* digit and terminate with *division*. In the example below we convert the ternary fraction, 0.1202 to decimal form.

$y = (((2/3 + 0)/3 + 2)/3 + 1)/3$

Evaluate the expression by starting with the innermost term and working out. We divide the rightmost digit by 3, add the next digit, divide the result by 3, add the next digit, etc., terminating with a final division by 3. The result in this case is 0.580. (Precision for this procedure depends on the precision of our arithmetic.)

A.1.7 Conversions Between Base b and Base b^k

Consider now the case of converting a number from base b to base b^k. Using the division method, we would be operating in base-b arithmetic, repeatedly dividing by the base of the "destination" system, namely b^k. Each such division is equivalent to k divisions by b. But each of these amounts to moving the radix point one place to the left. Hence, each division by the destination base consists of moving the radix point k places to the left. Since the number being divided (the dividend) each time is an integer, the remainder, which corresponds to a digit of the desired number, is the part of the quotient to the right of the radix point, i.e., the rightmost k digits of the dividend.

It follows then that to convert a base-b number x to base b^k, all we need do is partition the digits of x into subsequences of length k, starting from the right, and convert each of these subsequences into the corresponding base-b^k digit. For example, to convert x = 10010101112 to octal ($8 = 2^3$ -- so k = 3) we group the digits of x as follows: {001}{001}{010}{111}. Then we convert each group to an octal digit yielding 1127_8. Conversions in the reverse direction are correspondingly simple.

To go from base b^k to base b, we need only convert each digit of the original number to a k-digit (*don't forget to add leading 0s as necessary to fill out the required length)* base-b number. Thus, to convert 6207_9 to base-3, we replace 6 by 20, 2 by 02, 0 by 00, and 7 by 21 to obtain 20020021_3.

A.1.8 Binary, Octal, and Hexadecimal

In return for the simplicity of the binary number system (very simple arithmetic operations and only two symbols), we must pay the price of writing relatively long sequences of digits to represent a given number-- more than triple the number of digits required for a decimal representation. Here is where the techniques just described come in handy. Since $8 = 2^3$, it is very easy to convert back and forth between binary and octal numbers. Thus, we can use octal numbers to express binary numbers in compact form. For example,

1000111010110110_2 can be abbreviated by 107266_8. (Note that the same technique can be used to abbreviate *any* sequence of 0s and 1s, even if it does *not* represent a number.)

Although converting from decimal to binary is a straightforward matter using the division method, it is a lengthy computation to perform by hand when more than a few digits are involved. The reader might wish to verify this assertion by converting 97,603 to binary form (the answer is 10111110101000011). It is much easier to convert first to octal (obtaining 276,503) and *then* to binary-- try it! A similar advantage exists when octal is used as an intermediate stage for conversions from binary to decimal.

In most computers, information is usually in multiples of four-bits (16, 32, and 64 are common word sizes, and the 8-bit *byte* is almost universally used). It is therefore often convenient to abbreviate bit strings in groups of four, using the base 16, or *hexadecimal* (generally abbreviated as *hex*) number system. Then each byte is represented by two hex numerals. Since there are 16 numerals in hex, as opposed to only 10 in our everyday decimal system, 6 additional numerals are needed. Rather than invent new symbols, the common practice is to use the letters A through F to represent the numerals with values from 10 through 15, respectively. Thus, 506_{10} corresponds to $1FA_{16}$ ($1x16^2 + 15x16 + 10$).

A.1.9 Binary-Coded Decimal

It is important that computers be able to represent numbers internally in decimal form. In some cases, where internal arithmetic operations are relatively simple, as is often the case for commercial data processing, conversions back and forth between decimal and binary would be more time-consuming than the arithmetic intrinsic to the problem. Hence, it might be advantageous to have the machine operate internally in decimal. Even where the internal operations are to be in binary, it is necessary to represent the decimal numbers that constitute the input to the machine, and it is necessary to generate decimal numbers as the output. How can this be done, given the assertion made earlier in this appendix that two-valued signals are much preferred? The answer is to encode each decimal digit independently as a sequence of binary digits (or bits). Several different encodings have been used for this purpose, but we confine ourselves here to the most common, and most straightforward, scheme, which is simply to convert each decimal digit to its binary equivalent. This scheme, called *binar-coded decimal* (*BCD)*, requires four bits per digit. Leading 0s are inserted as necessary to fill out the four-bit slots for digits less than eight in value. Thus, 2089_{10} is represented in BCD as the concatenation of the four sequences 0010, 0000, 1000, 1001, or 0010000010001001. Note that the *binary* representation of 2089 is 100000101001, a 12-bit as opposed to a 16-bit number. This less efficient use of bits reflects the fact that only 10 of the 16 possible sequences of the four-bits utilized to represent a decimal digit.

A .1.10 ASCII Code

Although digital computers were originally developed to perform complex numerical computations, they are currently used for a wide range of other purposes, many of which involve the processing of symbols other than numbers, particularly alphanumeric characters. (Indeed the very words you are now reading were typed into a computer controlled by a word processing program.) It is therefore necessary to have a means for representing, within a computer, the sort of characters found on a standard typewriter. Since such data is often circulated among different computers, and may be transmitted over communications networks, it is useful to standardize the codes used.

The most popular such standard is American Standard Code for Information Interchange (ASCII, pronounced as' kee). Since 7 bits are used per character, a total of 2^7 or 128 different characters can be represented. The first three bits of an ASCII code word specify a class of characters, and the last four bits indicate the precise character within that class. For example, the numerals are all prefixed by 011, the four-bit suffix indicating the particular numeral in BCD. Other characters with prefix 011 include the colon, semi-colon and question mark. Uppercase letters are prefixed by 100 or 101, and lowercase letters are prefixed by 110 or 111. For example, 5 is encoded as 0110101, C

as 1000011, D as 1000100, c as 1100011. Various other symbols are prefixed by 010, for example, 0100000 is a space and 0101110 is a period. A number of ASCII codes are used for control signals such as "end of transmission", or "carriage return". A major competitor of ASCII is Extended BCD Interchange Code (EBCDIC), the 8-bit code originated by IBM.