Chapter 1

BASIC CONCEPTS

INTRODUCTION

This chapter will introduce the basic concepts and definitions relating to computer programming. The reader already familiar with these concepts may want to glance quickly at the contents of this chapter and then move on to Chapter 2. It is suggested, however, that even the experienced reader look at the contents of this introductory chapter. Many significant concepts are presented here including, for example, two's complement, BCD, and other representations. Some of these concepts may be new to the reader; others may improve the knowledge and skills of experienced programmers.

WHAT IS PROGRAMMING?

Given a problem, one must first devise a solution. This solution, expressed as a step-by-step procedure, is called an algorithm. An algorithm is a step-by-step specification of the solution to a given problem. It must terminate in a finite number of steps. This algorithm may be expressed in any language or symbolism. A simple example of an algorithm is:

1. insert key in the keyhole
2. turn key one full turn to the left
3. seize doorknob
4. turn doorknob left and push the door

At this point, if the algorithm is correct for the type of lock in volved, the door will open. This four-step procedure qualifies as an algorithm for door opening.

Once a solution to a problem has been expressed in the form of an algorithm, the algorithm must be executed by the computer. Unfortunately, it is now a well-established fact that computers cannot understand or execute ordinary spoken English (or any other human language). The reason lies in the syntactic ambiguity of all common human languages. Only a well-defined subset of natural language can be "understood" by the computer. This is called a programming language.

Converting an algorithm into a sequence of instructions in a pro gramming language is called programming. To be more specific, the actual translation phase of the algorithm into the program ming language is called coding. Programming really refers not just to the coding but also to the overall design of the programs and "data structures" which will implement the algorithm.

Effective programming requires not only understanding the possible implementation techniques for standard algorithms, but also the skillful use of all the computer hardware resources, such as internal registers, memory, and peripheral devices, plus a creative use of appropriate data structures. These techniques will be covered in the next chapters.

Programming also requires a strict documentation discipline, so that the programs are understandable to others, as well as to the author. Documentation must be both internal and external to the program.

Internal program documentation refers to the comments placed in the body of a program, which explain its operation.

External documentation refers to the design documents which are separate from the program: written explanations, manuals, and flowcharts.

FLOWCHARTING

One intermediate step is almost always used between the algorithm and the program. It is called a flowchart. A flowchart is simply a symbolic representation of the algorithm expressed as a sequence of rectangles and diamonds containing the steps of the algorithm. Rectangles are used for commands, or "executable statements." Diamonds are used for tests such as: If information X is true, then take action A, else B. Instead of presenting a formal definition of flowcharts at this point, we will introduce and discuss flowcharts later on in the book when we present programs.

Flowcharting is a highly recommended intermediate step be tween the algorithm specification and the actual coding of the solution. Remarkably, it has been observed that perhaps 10% of the programming population can write a program successfully with out having to flowchart. Unfortunately, it has also been observed that 90% of the population believes it belongs to this 10%! The result: 80% of these programs, on the average, will fail the first time they are run on a computer. (These percentages are naturally not meant to be accurate.) In short, most novice programmers seldom see the necessity of drawing a flowchart. This usually results in "unclean" or erroneous programs. They must then spend a long time testing and correcting their program (this is called the debugging phase). The discipline of flowcharting is therefore highly recommended in all cases. It will require a small amount of additional time prior to the coding, but will usually result in a clear program which executes correctly and quickly. Once flowcharting is well understood, a small percentage of programmers will be able to perform this step mentally without having to do it on paper. Unfortunately, in such cases the programs that they write will usually be hard to understand for anybody else without the documentation provided by flowcharts. As a result, it is universally recommended that flowcharting be used as a strict discipline for any significant program. Many examples will be provided throughout the book.

INFORMATION REPRESENTATION

All computers manipulate information in the form of numbers or in the form of characters. Let us examine here the external and internal representations of information in a computer.

INTERNAL REPRESENTATION OF INFORMATION

All information in a computer is stored as groups of bits. A *bit* stands for a *binary digit* ("0" or "1"). Because of the limitations of conventional electronics, the only practical representation of information uses two-state logic (the representation of the state "0" and "1"). The two states of the circuits used in digital electronics are generally "on" or "off", and these are represented logically by the symbols "0" or "1". Because these circuits are used to implement "logical" functions, they are called "binary logic." As a result, virtually all information-processing today is performed in binary format. In the case of microprocessors in general, and of the 6502 in particular, these bits are structured in groups of eight. A group of eight bits is called a *byte.* A group of four bits is called a *nibble.*

Let us now examine how information is represented internally in this binary format. Two entities must be represented inside the computer. The first one is the program, which is a sequence of instructions. The second one is the data on which the program will operate, which may include numbers or alphanumeric text. We will discuss below three representations: program, numbers, and alphanumerics.

Program Representation

All instructions are represented internally as single or multiple bytes. A so-called "short instruction" is represented by a single byte. A longer instruction will be represented by two or more bytes. Because the 6502 is an eight-bit microprocessor, it fetches bytes successively from its memory. Therefore, a single-byte instruction always has a potential for executing faster than a twoor three-byte instruction. It will be seen later that this is an important feature of the instruction set of any microprocessor and in particular the 6502, where a special effort has been made to provide as many single-byte instructions as possible in order to improve the efficiency of the program execution. However, the limitation to 8 bits in length has resulted in important restrictions which will be outlined. This is a classic example of "the compromise between speed and flexibility in programming. The binary code used to represent instructions is dictated by the manufacturer. The 6502, like any other microprocessor, comes equipped with a fixed instruction set. These instructions are defined by the manufacturer and are listed at the end of this book, with their code. Any program will be expressed as a sequence of these binary instructions. The 6502 instructions are presented in Chapter 4.

Representing Numeric Data

Representing numbers is not quite straightforward, and several cases must be distinguished. We must first represent integers, then signed numbers, i.e., positive and negative numbers, and finally we must be able to represent decimal numbers. Let us now address these requirements and possible solutions.

Representing integers may be performed by using a *direct binary* representation. The direct binary representation is simply the representation of the decimal value of a number in the binary system. In the binary system, the right-most bit represents 2 to the power 0. The next one to the left represents 2 to the power 1, the next represents 2 to the power 2, and the left-most bit represents 2 to the power 7 = 128.

$$\begin{gathered} b_7b_6b_5b_4b_3b_2b_1b_0 \\ \text{represents} \\ b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0 \end{gathered}$$

The powers of 2 are:

$$2^7 = 128, 2^6 = 64, 2^5 = 32, 2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$$

The binary representation is analogous to the decimal representation of numbers, where "123" represents:

$$\begin{array}{rrr} 1\times & 100 = & 100 \\ +2\times & 10 = & 20 \\ +3\times & 1 = & 3 \\ \hline & & = 123 \end{array}$$

Note that 100 = 102, 10 = 101, 1 = 10°.
In this "positional notation," each digit represents a power of 10.
In the binary system, each binary digit or "bit" represents a power of 2, instead of a power of 10 in the decimal system.

Example: "00001001" in binary represents:

$$\begin{array}{r}
1\times \quad 1 = 1\ (2_0) \\
0\times \quad 2 = 0\ (2_1) \\
0\times \quad 4 = 0\ (2_2) \\
1\times \quad 8 = 8\ (2_3) \\
0\times \ 16 = 0\ (2_4) \\
0\times \ 32 = 0\ (2_5) \\
0\times \ 64 = 0\ (2_6) \\
0\times 128 = 0\ (2_7) \\
\hline
\textit{in decimal}: \quad = 9
\end{array}$$

Let us examine some more examples:
"10000001" represents:

$$\begin{array}{r}
1\times \quad 1 = \quad 1 \\
0\times \quad 2 = \quad 0 \\
0\times \quad 4 = \quad 0 \\
0\times \quad 8 = \quad 0 \\
0\times \ 16 = \quad 0 \\
0\times \ 32 = \quad 0 \\
0\times \ 64 = \quad 0 \\
1\times 128 = 128 \\
\hline
\textit{in decimal}: \quad = 129
\end{array}$$

"10000001" represents, therefore, the decimal number 129.

By examining the binary representation of numbers, you will understand why bits are numbered from 0 to 7, going from right to left. Bit 0 is "b0" and corresponds to 2°. Bit 1 is "b/; and corresponds to 2 and so on.

Decimal	**Binary**	**Decimal**	**Binary**
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	01000000
12	00001100	129	01000001
13	00001101	•	
14	00001110	•	
15	00001111	•	
16	00010000	•	
17	00010001	•	
•		•	
•		•	
•		254	11111110
31	00011111	255	11111111

Table 1.1: Fig. 1-2: Decimal-Binary table

The binary equivalents of the numbers from 0 to 255 are shown in Fig. 1-2.

Exercise 1.1: *What is the decimal value of "11111100"?*

Decimal to Binary

Conversely, let us compute the binary equivalent of "11" decimal:

$$\begin{array}{rl}
11 \div 2 = 5 \textit{ remains } 1 \rightarrow & 1 \quad \text{(LSB)} \\
5 \div 2 = 2 \textit{ remains } 1 \rightarrow & 1 \\
2 \div 2 = 1 \textit{ remains } 0 \rightarrow & 0 \\
1 \div 2 = 0 \textit{ remains } 1 \rightarrow & 1 \quad \text{(MSB)}
\end{array}$$

The binary equivalent is 1011 (read right-most column from bottom to top). The binary equivalent of a decimal number may be obtained by dividing successively by 2 until a quotient of 0 is obtained.

Exercise 1.2: *What is the binary for 257?*

Exercise 1.3: *Convert 19 to binary, then back to decimal.*

Operating on Binary Data

The arithmetic rules for binary numbers are straightforward. The rules for addition are:

$$\begin{array}{lcc}
0+0= & & 0 \\
0+1= & & 1 \\
1+0= & & 1 \\
1+1= & (1) & 0
\end{array}$$

where (1) denotes a “carry” of 1 (note that “10” is the binary equivalent of “2” decimal). Binary subtraction will be performed by ”adding the complement” and will be explained once we learn how to represent negative numbers.
Example:

$$\begin{array}{rr}
(2) & 10 \\
+(1) & +01 \\
\hline
=(3) & 11
\end{array}$$

Addition is performed just like in decimal, by adding columns, from right to left:

Adding the right-most column:

```
         10
        +01
  ───────────
   (0 + 1 = 1  . No carry.)
```

Adding the next column:

```
   10
  +01
  ────
   11  (1+0 =1. No carry.)
```

Exercise 1.4: *Compute 5 + 10 in binary. Verify that the result is 15.*

Some additional examples of binary addition:

```
   0010  (2)      0011  (3)
  +0001  (1)     +0001  (1)
  ──────────     ──────────
  =0011  (3)     =0100  (4)
```

This last example illustrates the role of the carry.

Looking at the right-most bits: 1 + 1 = (1) 0
A carry of 1 is generated, which must be added to the next bits:

```
       001  —        column 0 has just been adjusted
      +000  —
        +1  (carry)
  ─────────────────
                     Where (1) indicates a new
  =   (1)0  —        carry into column 2.
```

The final result is: 0100

Another example:

$$\begin{array}{rr} 0111 & (7) \\ +0011 & +\ (3) \\ \hline 1010 & =(10) \end{array}$$

In this example, a carry is again generated, up to the left-most column.

Exercise 1.5: *Compute the result of:*

$$\begin{array}{r} 1111 \\ +0001 \\ \hline =? \end{array}$$

Does the result hold in four bits?

With eight bits, it is therefore possible to represent directly the numbers "00000000" to "11111111," i.e., "0" to "255". Two obstacles should be visible immediately. First, we are only representing positive numbers. Second, the magnitude of these numbers is limited to 255 if we use only eight bits. Let us address each of these problems in turn.

Signed Binary

In a signed binary representation, the left-most bit is used to in dicate the sign of the number. Traditionally, "0" is used to denote a positive number while "1" is used to denote a negative number. Now "11111111" will represent -127, while "01111111" will represent +127. We can now represent positive and negative numbers, but we have reduced the maximum magnitude of these numbers to 127.

Example: "0000 0001" represents +1 (the leading "0" is " + ", followed by "000 0001" = 1).
"1000 0001" is -1 (the leading "1" is "-").

Exercise 1.6: *What is the representation of "—5" in signed binary?*

Let us now address the magnitude problem: in order to represent larger numbers, it will be necessary to use a larger number of bits. For example, if we use sixteen bits (two bytes) to represent numbers, we will be able to represent numbers from — 32K to +32K in signed binary (IK in computer jargon represents 1,024). Bit 15 is used for the sign, and the remaining 15 bits (bit 14 to bit 0) are used for the magnitude: 215 = 32K. If this magnitude is still too small, we will use 3 bytes or more. If we wish to represent large integers, it will be necessary to use a larger number of bytes inter nally to represent them. This is why most simple BASICs, and other languages, provide only a limited precision for integers. This way, they can use a shorter internal format for the numbers which they manipulate. Better versions of BASIC, or of these other languages, provide a larger number of significant decimal digits at the expense of a large number of bytes for each number.

Now let us solve another problem, the one of speed efficiency. We are going to attempt performing an addition in the signed binary representation which we have introduced. Let us add " — 5" and"+7".

+7 is represented by	00000111
-5 is represented by	10000101
The binary sum is:	10001100, or -12

This is not the correct result. The correct result should be +2. In order to use this representation, special actions must be taken, de pending on the sign. This results in increased complexity and re duced performance. In other words, the binary addition of signed numbers does not "work correctly." This is annoying. Clearly, the computer must not only represent information, but also perform arithmetic on it.

The solution to this problem is called the *two's complement* representation, which will be used instead of the signed binary representation. In order to introduce two's complement let us first introduce an intermediate step: one's complement.

One's Complement

In the one's complement representation, all positive integers are represented in their correct binary format. For example "+3" is represented as usual by 00000011. However, its complement "—3" is obtained by complementing every bit in the original representa tion. Each 0 is transformed into a 1 and each 1 is transformed into a 0. In our example, the one's complement

representation of ”—3” will be 11111100.
Another example:

+2	is	00000010
-2	is	11111101

Note that, in this representation, positive numbers start with a ”0” on the left, and negative ones with a ”1” on the left.

Exercise 1.7: *The representation of ”+6” is ”00000110”. What is the representation of ”—6” in one's complement?*

As a test, let us add minus 4 and plus 6:

	-4	is	11111011	
	+6	is	00000110	
the sum is:	(1)		00000001	where (1) indicates a carry

The ”correct result” should be ”2”, or ”00000010”.

Let us try again:

	-3	is	11111100
	-2	is	11111101
the sum is:	(1)		00000001

or ”1,” plus a carry. The correct result should be ”-5.” The representation of ”-5” is 11111010. It did not work.

This representation does represent positive and negative numbers. However the result of an ordinary addition does not always come out ”correctly.” We will use still another representa tion. It is evolved from the one's complement and is called the two's complement representation.

Two's Complement Representation

In the two's complement representation, positive numbers are still represented, as usual, in signed binary, just like in one's com plement. The difference lies in the representation of *negative numbers.* A negative number represented in two's complement is obtained by first computing the one's complement, and then *adding one.* Let us examine this in an example:

+3 is represented in signed binary by 00000011. Its one's complement representation is 11111100. The two's complement is obtained by adding one. It is 11111101.

Let us try an addition:

$$\begin{array}{rr} (3) & 00000011 \\ +(5) & +00000101 \\ \hline =(8) & =00001000 \end{array}$$

The result is correct.

Let us try a subtraction:

$$\begin{array}{rr} (3) & 00000011 \\ (-5) & +11111011 \\ \hline & =11111110 \end{array}$$

Let us identify the result by computing the two's complement:

$$\begin{array}{lrl} \text{the one's complement of 11111110 is} & 00000001 & \\ \text{Adding 1} & + \quad\quad 1 & \\ \cline{2-2} \text{therefore the two's complement is} & 00000010 & \text{or } +2 \end{array}$$

Our result above, "11111110" represents "—2". It is correct.

We have now tried addition and subtraction, and the results were correct (ignoring the carry). It seems that two's complement works!

Exercise 1.8: *What is the two's complement representation of "+127"?*

Exercise 1.9: *What is the two's complement representation of "-128"?*

Let us now add +4 and —3 (the subtraction is performed by adding the two's complement):

	+4	is	00000100
	-3	is	11111101
The result is:	(1)		00000001

If we ignore the carry, the result is 00000001, i.e., "1" in decimal. This is the correct result. Without giving the complete mathematical proof, let us simply state that this representation does work. In two's complement, it is possible to add or subtract signed numbers regardless of the sign. Using the usual rules of binary addition, the result comes out correctly, including the sign. The carry is ignored. This is a very significant advantage. If it were not the case, one would have to correct the result for sign every time, causing a much slower addition or subtraction time.

For the sake of completeness, let us state that two's complement is simply the most convenient representation to use for simpler processors such as microprocessors. On complex processors, other representations may be used. For example, one's complement may be used, but it requires special circuitry to "correct the result."

From this point on, all signed integers will implicitly be represented internally in two's complement notation. See Fig. 1-3 for a table of two's complement numbers.

Exercise 1.10: *What are the smallest and the largest numbers which one may represent in two's complement notation, using only one byte?*

Exercise 1.11: *Compute the two's complement of 20. Then compute the two's complement ofyour result. Do you find 20 again?*

The following examples will serve to demonstrate the rules of two’s complement. In particular, C denotes a possible carry (or borrow) condition. (It is bit 8 of the result.)

V denotes a two’s complement overflow, i.e., when the sign of the result is changed ”accidentally” because the numbers are too large. It is an essentially internal carry from bit 6 into bit 7 (the sign bit). This will be clarified below.

Let us now demonstrate the role of the carry ”C” and the overflow ”V”.

The Carry C

Here is an example of a carry:

$$\begin{array}{rrr} (128) & & 10000000 \\ +(129) & & +10000001 \\ \hline (257) & =(1) & 00000001 \end{array}$$

where (1) indicates a carry.

The result requires a ninth bit (bit ”8”, since the right-most bit is ”0”). It is the carry bit.

If we assume that the carry is the ninth bit of the result, we recognize the result as being 100000001 = 257.

However, the carry must be recognized and handled with care. Inside the microprocessor, the registers used to hold information are generally only eight-bit wide.When storing the result, only bits 0 to 7 will be preserved.

A carry, therefore, always requires special action: it must be detected by special instructions, then processed. Processing the carry means either storing it somewhere (with a special instruction), or ignoring it, or deciding that it is an error (if the largest authorized result is ”11111111”).

+	2's complement code	-	2's complement code
+127	01111111	-128	10000000
+126	01111110	-127	10000001
+125	01111101	-126	10000010
...		-125	10000011
		...	
+65	01000001	-65	10111111
+64	01000000	-64	11000000
+63	00111111	-63	11000001
...		...	
+33	00100001	-33	11011111
+32	00100000	-32	11100000
+31	00011111	-31	11100001
...		...	
+17	00010001	-17	11101111
+16	00010000	-16	11110000
+15	00001111	-15	11110001
+14	00001110	-14	11110010
+13	00001101	-13	11110011
+12	00001100	-12	11110100
+11	00001011	-11	11110101
+10	00001010	-10	11110110
+9	00001001	-9	11110111
+8	00001000	-8	11111000
+7	00000111	-7	11111001
+6	00000110	-6	11111010
+5	00000101	-5	11111011
+4	00000100	-4	11111100
+3	00000011	-3	11111101
+2	00000010	-2	11111110
+1	00000001	-1	11111111
0	00000000		

Table 1.2: fig 1-3: 2s Complement Table

Overflow V

Here is an example of overflow:

```
bit 6
bit 7
           01000000       (64)
          +01000001      +(65)
         ----------------------
          =10000001   =(-127)
```

An internal carry has been generated from bit 6 into bit 7. This is called an overflow.

The result is now negative, ”by accident.” This situation must be detected, so that it can be corrected.

Let us examine another situation:

```
                 11111111     (-1)
                +11111111   +(-1)
      -----------------------------
       =(1)
        ▼        11111110   =(-2)
      carry
```

In this case, an internal carry has been generated from bit 6 into bit 7, and also from bit 7 into bit 8 (the formal “Carry” C we have examined in the preceding section). The rules of two’s complement arithmetic specify that this carry should be ignored. The result is then correct.

This is because the carry from bit 6 into bit 7 did not change the sign bit.

This is not an *overflow* condition. When operating on negative numbers, the overflow is not simply a carry from bit 6 into bit 7.
Let us examine one more example.

```
             11000000   (-64)
            +10111111   (-65)
  ---------------------------
  =(1)
    ▼        01111111   (+127)
  carry
```

This time, there has been no internal carry from bit 6 into bit 7, but there has been an external carry. The result is incorrect, as bit 7 has been changed. An overflow condition should be indicated.

Overflow will occur in four situations:

1. —adding large positive numbers
2. —adding large negative numbers
3. —subtracting a large positive number from a large negative number
4. —subtracting a large negative number from a large positive number.

Let us now improve our definition of the overflow:

Technically, the overflow indicator, a special bit reserved for this purpose, and called a "flag," will be set when there is a carry from bit 6 into bit 7 and no external carry, or else when there is no carry from bit 6 into bit 7 but there is an external carry. This indicates that bit 7, i.e., the sign of the result, has been accidentally changed. For the technically-minded reader, the overflow flag is set by Exclusive ORing the carry-in and carry-out of bit 7 (the sign bit). Practically every microprocessor is supplied with a special overflow flag to automatically detect this condition, which re quires corrective action.

Overflow indicates that the result of an addition or a subtraction requires more bits than are available in the standard eight-bit register used to contain the result.

The Carry and the Overflow

The carry and the overflow bits are called "flags." They are provided in every microprocessor, and in the next chapter we will learn to use them for effective programming. These two indicators are located in a special register called the flags or " status" register. This register also contains additional indicators whose function will be clarified in Chapter 4.

Examples

Let us now illustrate the operation of the carry and the overflow in actual examples. In each example, the symbol V denotes the overflow, and C the carry.

If there has been no overflow, V = 0. If there has been an overflow, V = 1 (same for the carry C). Remember that the rules of two's complement specify that the carry be ignored. (The mathematical proof is not supplied here.)

Positive-Positive

$$\begin{array}{ccc} & 00000110 & (+6) \\ + & 00001000 & (+8) \\ \hline = & 00001110 & (+14) \end{array} \quad \text{V:0} \quad \text{C:0}$$

(CORRECT)

Positive-Positive with Overflow

$$\begin{array}{ccr} & 01111111 & (+127) \\ + & 00000001 & (+1) \\ \hline = & 10000000 & (-128) \end{array} \quad \text{V:1} \quad \text{C:0}$$

The above is invalid because an overflow has occurred.
(ERROR)

Positive-Negative (result positive)

$$\begin{array}{ccr} & 00000100 & (+4) \\ + & 11111110 & (-2) \\ \hline =(1) & 00000010 & (+2) \end{array} \quad \text{V:0} \quad \text{C:1 (disregarded)}$$

(CORRECT)

Positive-Negative (result negative)

$$\begin{array}{ccr} & 00000010 & (+2) \\ + & 11111100 & (-4) \\ \hline =(1) & 11111110 & (-2) \end{array} \quad \text{V:0} \quad \text{C:0}$$

(CORRECT)

Negative-Negative

	11111110	(-2)			
+	11111100	(-4)			
=(1)	11111010	(-6)	V:0	C:1 (disregarded)	

(CORRECT)

Negative-Negative with Overflow

	10000001	(-127)		
+	11000010	(-62)		
=(1)	01000011	(67)	V:1	C:1

(ERROR)

This time an "underflow" has occurred, by adding two large negative numbers. The result would be -189, which is too large to reside in eight bits.

Exercise 1.12: *Complete the following additions. Indicate the result, the carry C, the overflow V, and whether the result is correct or not:*

	10111111	(___)			11111010	(___)	
+	11000001	(___)		+	11111001	(___)	
=	________	V:___	C:___	=	________	V:___	C:___
[ ] CORRECT		[ ] ERROR		[ ] CORRECT		[ ] ERROR	

	00010000	(___)			01111110	(___)	
+	01000000	(___)		+	00101010	(___)	
=	________	V:___	C:___	=	________	V:___	C:___
[ ] CORRECT		[ ] ERROR		[ ] CORRECT		[ ] ERROR	

Exercise 1.13: *Can you show an example of overflow when adding a positive and a negative number? Why?*

Fixed Format Representation

Now we know how to represent signed integers. However, we have not yet resolved the problem of magnitude. If we want to represent larger integers, we will need several bytes. In order to perform arithmetic operations efficiently, it is necessary to use a fixed number of bytes rather than a variable one. Therefore, once the number of bytes is chosen, the maximum magnitude of the number which can be represented is fixed.

Exercise 1.14: *What are the largest and the smallest numbers which may be represented in two bytes using two's complement?*

The Magnitude Problem

When adding numbers we have restricted ourselves to eight bits because the processor we will use operates internally on eight bits at a time. However, this restricts us to the numbers in the range -128 to +127. Clearly, this is not sufficient for many applications. Multiple precision will be used to increase the number of digits which can be represented. A two-, three-, or N-byte format may then be used. For example, let us examine a 16-bit, "double-pre cision" format:

```
00000000  00000000  is "0"
00000000  00000001  is "1"

01111111  11111111  is "32767"
11111111  11111111  is "-1"
11111111  11111110  is "-2"
```

Exercise 1.15: *What is the largest negative integer which can be represented in a two's complement triple-precision format?*

However, this method will result in disadvantages. When adding two numbers, for example, we will generally have to add them eight bits at a time. This will be explained in Chapter 4 (Basic Programming Techniques). It results in slower processing. Also, this representation uses 16 bits for any number, even if it could be represented with only eight bits. It is, therefore, common to use 16 or perhaps 32 bits, but seldom more.

Let us consider the following important point: whatever the number of bits N chosen for the two's complement representation, it is fixed. If any result or intermediate computation should generate a number requiring more than N bits, some bits will be lost. The program normally retains the N left-most bits (the most significant) and drops the low-order ones. This is called truncating the result.

Here is an example in the decimal system, using a six digit representation:

$$\begin{array}{rr} & 123456 \\ \times & 1.2 \\ \hline & 246912 \\ & 123456 \\ \hline = & 148147.2 \end{array}$$

The result requires 7 digits! The "2" after the decimal point will be dropped and the final result will be 148147. It has been truncated. Usually, as long as the position of the decimal point is not lost, this method is used to extend the range of the operations which may be performed, at the expense of precision.

The problem is the same in binary. The details of a binary multiplication will be shown in Chapter 4.

This fixed-format representation may cause a loss of precision, but it may be sufficient for usual computations or mathematical operations.

Unfortunately, in the case of accounting, no loss of precision is tolerable. For example, if a customer rings up a large total on a cash register, it would not be acceptable to have a five figure amount to pay, which would be approximated to the dollar. Another representation must be used wherever precision in the result is essential. The solution normally used is BCD, or binary-coded decimal.

BCD Representation

The principle used in representing numbers in BCD is to encode each decimal digit separately, and to use as many bits as necessary to represent the complete number exactly. In order to encode each of the digits from 0 through 9, four bits are necessary. Three bits would only supply eight combinations, and can therefore not en code the ten digits. Four bits allow sixteen combinations and are therefore sufficient to encode the digits "0" through "9". It can also be noted that six of the possible codes will not be used in the BCD representation (see Fig. 1-3). This will result later on in a po tential problem during additions and subtractions, which we will have to solve. Since only four bits are needed to encode a BCD

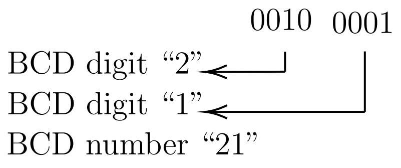
CODE	BCD SYMBOL	CODE	BCD SYMBOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	unused
0011	3	1011	unused
0100	4	1100	unused
0101	5	1101	unused
0110	6	1110	unused
0111	7	1111	unused

Table 1.3: Fig. 1-4: BCD Table

digit, two BCD digits may be encoded in every byte. This is called "*packed BCD*."

As an example, "00000000" will be "00" in BCD. "10011001" will be "99".

A BCD code is read as follows:

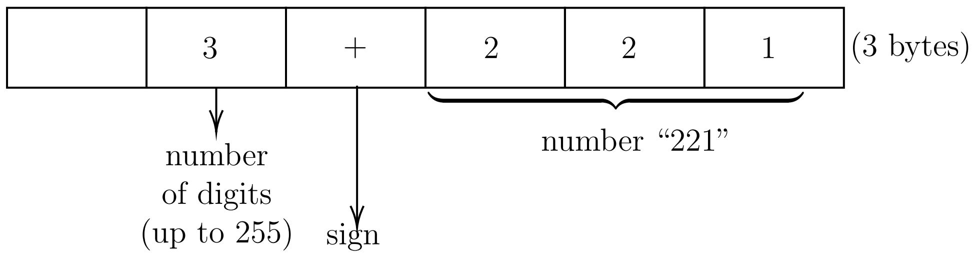


Exercise 1.16: *What is the BCD representation for "29"? "91"?*

Exercise 1.17: *Is "10100000" a valid BCD representation? Why?*

As many bytes as necessary will be used to represent all BCD digits. Typically, one or more nibbles will be used at the beginning of the representation to indicate the total number of nibbles, i.e., the total number of BCD digits used. Another nibble or byte will be used to denote the position of the decimal point. However, conventions may vary.

Here is an example of a representation for multibyte BCD integers:

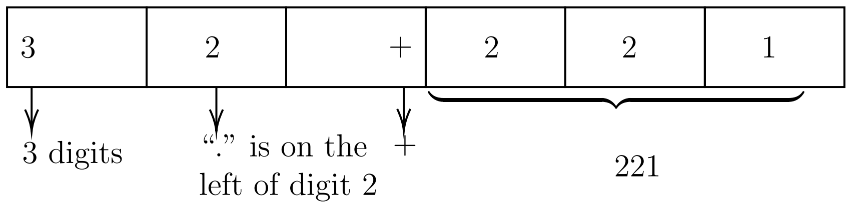


This represents +221
(The sign may be represented by 0000 for +, and 0001 for -, for example.)

Exercise 1.18: *Using the same convention, represent "-23123". Show it in BCDformat, as above, then in binary.*

Exercise 1.19: *Show the BCD for "222" and "111", then for the result of 222 × 111. (Compute the result by hand, then show it in the above representation.)*

The BCD representation can easily accommodate decimal numbers.
For example, +2.21 may be represented by:



The advantage of BCD is that it yields absolutely correct results. Its disadvantage is that it uses a large amount of memory and results in slow arithmetic operations. This is acceptable only in an accounting environment and is normally not used in other cases.
Exercise 1.20: *How many bits are required to encode '9999" in BCD? And in Two's complement?*

We have now solved the problems associated with the representation of integers, signed integers and even large integers. We have even already presented one possible method of representing decimal numbers, with BCD representation. Let us now examine the problem of representing decimal numbers in a fixed length for mat.

Floating-Point Representation

The basic principle is that decimal numbers must be represented with a fixed format. In order not to waste bits, the representation will *normalize* all the numbers.

For example, "0.000123" wastes three zeros on the left of the number, which have no meaning except to indicate the position of the decimal point. Normalizing this number results in .123 × 103. ".123" is called a *normalized mantissa*, "-3" is called the *exponent*. We have normalized this number by eliminating all the meaning less zeros on the left of it and adjusting the exponent.

Let us consider another example:

22.1 is normalized as $.221 \times 10^2$
or $M \times 10^E$ where M is the mantissa, and E is the exponent.

It can be readily seen that a normalized number is characterized by a mantissa less than 1 and greater or equal to .1 in all cases where the number is not zero. In other words, this can be repre sented mathematically by:

$$.1 \leqslant M < 1 \textit{ or } 10^{-1} \leqslant M < 10^0$$

Similarly, in the binary representation:

$$2^{-1} \leqslant M < 2^0 \ (\textit{or } .5 \leqslant M < 1)$$

Where M is the absolute value of the mantissa (disregarding the sign).

For example:

$$111.01 \textit{ is normalized as}: \ .11101 \times 2^3.$$

The mantissa is 11101.

The exponent is 3.

Now that we have defined the principle of the representation, let us examine the actual format. Atypical floating-point representation appears below.

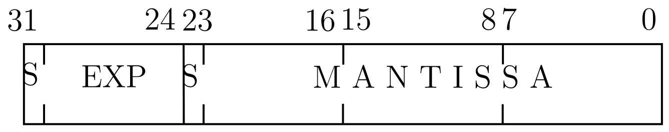


Fig. 1-5: Typical Floating-Point Representation In the representation used in this example, four bytes are used for a total of 32 bits. The first byte on the left of the illustration is used to represent the exponent. Both the exponent and the man tissa will be represented in two's complement. As a result, the maximum exponent will be -128. "S" in Fig. 1-5 denotes the sign bit. Three bytes are used to represent the mantissa. Since the first bit in the two's complement representation indicates the sign, this leaves 23 bits for the representation of the magnitude of the man tissa. 30BASIC CONCEPTS Exercise 1.21: How many decimal digits can the mantissa repre sent with the 23 bits? This is only one example of a floating point representation. It is possible to use only three bytes, or it is possible to use more. The four-byte representation proposed above is just a common one which represents a reasonable compromise in terms of accuracy, magnitude of numbers, storage utilization, and efficiency in arithmetic operation. We have now explored the problems associated with the rep resentation of numbers and we know how to represent them in in teger form, with a sign, or in decimal form. Let us now examine how to represent alphanumeric data internally. Representing Alphanumeric Data The representation of alphanumeric data, i.e. characters, is com pletely straightforward: all characters are encoded in an eight-bit code. Only two codes are in general use in the computer world, the ASCII Code, and the EBCDIC Code. ASCII stands for *'American Standard Code for Information Interchange," and is universally used in the world of microprocessors. EBCDIC is a variation of ASCII used by IBM, and therefore not used in the microcomputer world unless one interfaces to an IBM terminal. Let us briefly examine the ASCII encoding. We must encode 26 letters of the alphabet for both upper and lower case, plus 10 numeric symbols, plus perhaps 20 additional special symbols. This can be easily accomplished with 7 bits, which allow 128 possible codes. (See Fig. 1-6.) All characters are therefore encoded in 7 bits. The eighth bit, when it is used, is the parity bit Parity is a tech nique for verifying that the contents of a byte have not been ac cidentally changed. The number of l's in the byte is counted and the eighth bit is set to one if the count was odd, thus making the total even. This is called even parity. One can also use odd parity, i.e. writing the eighth bit (the left-most) so that the total number of l's in the byte is odd. Example:

let us compute the parity bit for "0010011" using even parity. The number of l's is 3. The parity bit must therefore be a 1 so that the total number of bits is 4, i.e. even. The result is 10010011, where the leading 1 is the parity bit and 0010011 iden tifies the character. 31PROGRAMMING THE 6502 The table of 7-bit ASCII codes is shown in Fig. 1-6. In practice, it is used "as is," i.e. without parity, by adding a 0 in the left-most position, or else with parity, by adding the appropriate extra bit on the left. Exercise 1.22: Compute the 8-bit representation of the digits "0" through "0", using even parity. (This code will be used in applica tion examples of Chapter 8.) Exercise 1.23: Same for the letters "A " through 44F". Exercise 1.24: Using a non-parity ASCII code (where the left-most bit is "0'7, indicate the binary contents of the 4 bytes below: BIT NUMBERS br b. b. J 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 b¿ 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 b, 0 0 1 1 0 0 1 1 0 0 1 J 0 0 1 1 b. 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 HEXOS. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 0 0 0 NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SO SI 0 0 1 1 DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EAA SUB ESC FS GS RS US 0 1 0 2 SP 1 030123456789 < − >
?1004mABCDEFGH1JKLMN01015PQRSTUVWXYZ[V]$A_1$106abcdef9hik1mn01117PqrstuVwX
ELFig.1−6 : ASCIIConversionTableInspecializedsituationssuchastelecommunications, othercodin
correctingcodes.Howevertheyarebeyondthescopeofthisbook.32BASICCONCEPTSWehaveexamine
binary, octalorhexdecimal, andsymbolic.1.BinaryIthasbeenseenthatinformationisstoredinternallyin
bitmicroprocessor, afrontpanelwilltypicallybeequippedwitheightLEDstodisplaythecontentsofanyinte
33PROGRAMMINGTHE6502binary000001010Oil100101110111octal01234567Fig.1−
7 : OctalSymbolsForexample, "00100YT04or"044"inoctal.Anotherexample :
11Y3or"377"inoctal.100"binaryisrepresentedby : Y4111Y.7111is : Y7Conversely, theoctal"211"repres
010001001or"10001001"binary.Octalhastraditionallybeenusedonoldercomputerswhichwereemploying
bitmicroprocessors, theeight−bitformathasbecomethestandard, andanothermorepracticalrepresentat
8).34BASICCONCEPTSDECIMAL0123456789101112131415BINARY000000010010001101000101
8 : HexadecimalCodes35PROGRAMMINGTHE6502Example : 10100001inbinaryisrepresentedbyA
Whatisthehexadecimalrepresentationof"10101010?"Exercise1.26 : Conversely, whatisthebinaryequ
Whatistheoctalof"01000001"?Hexadecimaloffersthe advantageofencodingeightbitsintoonlytwodigit
typescreenusedtodisplaytextorgraphics.)Unfortunately, insmallersystemssuchasone−
boardmicrocomputers, itisuneconomicaltoprovidesuchdisplays, andtheuserisrestrictedtohexadecimal
buggingatthehardwareorthesoftwarelevelisthebinaryrepresentationused.Binarydirectlydisplaysthe
Whatistheadvantageoftwo'scomplementoverotherrepresentationsusedtorepresentsignednumbers?E
Howwouldyourepresent"1024"indirectbinary?Signedbinary?Two'scomplement?Exercise1.30 :
WhatistheV−bit?Shouldtheprogrammertestitafteranadditionorsubtraction?Exercise1.31 :
Computethetwo'scomplementof" +16", " +17", " +18", " −16", " −17", " −

18”.$Exercise1.32 : Showthehexadecimalrepresentationofthefollowingtext, whichhasbeensto$
”$MESSAGE$”.