Assembly Language

Contents

1.1 Microprocessors and the 8086 family
1.2 Assembly language
1.3 Number systems used in assembly language
1.4 Some simple 8086-family assembly language instructions
1.5 Eight 'new' registers
1.6 The flags register
   Summary
   Self-check quiz

At the heart of every microcomputer there is a microprocessor which can be programmed using assembly language. This book describes the assembly language of a family of microprocessors, the 8086 family, one of which – the 8088 microprocessor – is found inside the IBM-PC, and the necessary internal computer organization when a microcomputer is built around one of the microprocessors in this family.

This chapter explains what assembly language is, its relationship to higher level languages such as Pascal, and discusses when and when not to use it. We also summarize the background knowledge concerning binary and hexadecimal arithmetic which is assumed in this book, and describe the various ways of representing numeric and text data in binary form. In the final sections we begin our examination of 8086-family assembly language instructions, including a description of the four general-purpose registers AX, BX, CX and DX, the eight general-purpose 8-bit registers which pairwise make up AX, BX, CX and DX, and the flags register.
By the end of this chapter the reader should be able to write some very simple programs in 8086-family assembly language and know how to use the technique of tracing to follow the effect of a small program. Details of how to have these programs executed by the computer will follow in Chapter 4.

### 1.1 Microprocessors and the 8086 family

In any microcomputer, the component which actually processes data is entirely contained on a single silicon chip called a **microprocessor**. There are many different kinds of microprocessor just as there are different makes of computer.

Table 1.1 gives a short list of some microcomputers popular at the time of writing and the type of microprocessor each uses. Incidentally, microprocessors tend to have mysterious sounding names like QB994 but just as the name of a car tells you little about the car itself, so it often is with microprocessors.

We call the family of microprocessors on which the IBM-PC range is based, the 8086 family. Intel Corporation, the manufacturer of these microprocessors, first produced the 8086 microprocessor as an updated version of one which was used in many of the microcomputers on sale in the late 1970s. The other members of the family are the 8088 microprocessor (which, as we shall see, is less powerful than the 8086 microprocessor but was introduced to enable computer designs with lower manufacturing costs) and then the 80186, 80188, 80286, 80386, 80386SX and 80486 microprocessors, the last of which is one of the most powerful on the market at the time of writing. Figure 1.1 shows the relationship between them.

<table>
<thead>
<tr>
<th>Microcomputer</th>
<th>Microprocessor it uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Personal Computer</td>
<td>8088</td>
</tr>
<tr>
<td>Apricot</td>
<td>8086</td>
</tr>
<tr>
<td>Research Machines Nimbus</td>
<td>80186</td>
</tr>
<tr>
<td>IBM PC AT</td>
<td>80286</td>
</tr>
<tr>
<td>Tandy 5000</td>
<td>80386</td>
</tr>
<tr>
<td>IBM Model 70-A21</td>
<td>80486</td>
</tr>
<tr>
<td>Acorn Archimedes</td>
<td>ARM</td>
</tr>
<tr>
<td>Apple Macintosh</td>
<td>68000</td>
</tr>
<tr>
<td>Olivetti M-20</td>
<td>Z8000</td>
</tr>
<tr>
<td>Atari ST</td>
<td>68000</td>
</tr>
<tr>
<td>Commodore Amiga</td>
<td>68000</td>
</tr>
<tr>
<td>Increasing computer power</td>
<td>8086 (8088 is functionally the same but less powerful)</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>80186 (80188 is functionally the same but less powerful)</td>
</tr>
<tr>
<td></td>
<td>80286</td>
</tr>
<tr>
<td></td>
<td>80386 (80386SX is functionally the same but less powerful)</td>
</tr>
<tr>
<td></td>
<td>80486</td>
</tr>
</tbody>
</table>

We use the term ‘family’ because there is a great deal of compatibility between the microprocessors: software written to be executed by the 8088 microprocessor will work just as well if executed by the 80386 microprocessor. This is because they all share the same set of instructions, but later additions to the family have an extended and more sophisticated repertoire. Chapter 2 explains some of these differences and Chapter 23 will consider instructions unique to the more sophisticated processors such as the 80386 microprocessor.

In the meantime, we shall cover some of the necessary background, starting with the preferred language for programming microprocessors, assembly language.

### 1.2 Assembly language

Most computer programs these days are written in high-level languages such as BASIC, COBOL and Pascal. Writing programs in any of these is quicker and easier (and therefore cheaper if you are paying somebody to do it) than in the machine’s own language – **machine code**. There are some occasions, however, when a high-level language just cannot be used. Pascal may be able to do the job from a logical point of view, but the machine code generated from it may be too slow for the application in question. For example, when a computer is being used to control a nuclear reactor in a power station, if the controlling program cannot respond very quickly to changes in the reactor’s state the reactor may go critical and there will be danger of an explosion. Also, if it happens to be your lot to write programs to control peripherals such as disk drives, then a high-level language such as Pascal will neither work fast enough to give users the rapid response they expect, nor will it give you close enough control over the hardware to do the job efficiently. Indeed, using a high-level language in this sort of context is often like trying to eat a bar of chocolate with a knife and fork while wearing a thick pair of mittens – just about possible but not much fun if you have to do it every day.

Regrettably, then, there are occasions when the programmer must program at the machine’s own level though very few programmers actually use machine code, the language of 0s and 1s which the computer’s circuits understand directly. Machine code programs are tedious to write and highly
error-prone. Imagine the errors likely to arise from an interdepartmental telephone conversation about the following machine code program:

0010101111000011
1000101111100100
101101100000000000000000

In situations where a high-level language is inappropriate we avoid working in machine code most of the time by making the computer do more of the work. We humans write our programs in a more readable form – assembly language – and then get the computer to turn this assembly language program into machine code. The above machine code program was produced by a computer from the following assembly language version:

SUB AX, BX
MOV CX, AX
MOV DX, 0

Here, as you may guess, the mnemonic (memory aid) SUB is short for SUBtract and MOV represents a MOVE instruction. It is much easier to remember SUB AX, BX than 0010101111000011 – the actual instruction code the machine uses.

Assembly languages are an intermediate step between high-level languages and machine code. For example, the actual execution of a Pascal program is sometimes achieved by automatically converting the program into assembly language form and then finally converting that into machine code which is executed. This process is illustrated in Figure 1.2. Conversion between these languages is performed at each stage by computer programs: from high-level language to assembly language by a compiler and from assembly language to machine code by an assembler.

Pascal programmers work in a cushioned environment as do most high-level language programmers. In Pascal, variables can be of integer, real, char, array or record types – or even one of the programmer’s own defined types – but machine code has just binary numbers. Modern versions of assembly language do now handle some of the data types high-level languages provide.

---

**Figure 1.2**
The relationship between high-level languages and machine code.

Pascal program A

\[\text{Compiler}\]

Assembly language version of A

\[\text{Assembler}\]

Machine code version of A

---

This version is the one actually executed
but the accompanying programming facilities to handle objects of these types are often very limited. Indeed the main advantage of high-level languages is that the effect of each programmable operation is at a higher functional level than in assembly language programming. Thus, printing the result of the addition of two numbers is a one line instruction in Pascal whereas the equivalent in assembly language involves several instructions. In APL, a given array can be assigned to the product of two other arrays by a single instruction. Generally, a much greater level of detail is involved in using assembly language to make even simple things happen.

All this makes writing big programs in assembly language harder and more error-prone than high-level language programming. Unfortunately assembly language sometimes has to be used – for example, because of the loss of speed experienced when using a compiled language such as Pascal. Thus, in a commercial environment, assembly language is used only when no other higher level programming language is deemed able to give the necessary performance from the hardware. But then there is an even greater penalty for using assembly language than the extra costs involved. Assembly language programs are very much more dependent upon the precise design of the computer they are to run on than their high-level language equivalents. In consequence, it is usually much easier to get a Pascal program, written originally for machine A, up and running on machine B than it is to transport an assembly language program from machine A to machine B. Indeed, if the architecture of machine B differs in any considerable way from that of machine A, virtually a complete rewrite may be necessary.

More often than not, operating systems have to be written in assembly language as do programs to control external devices such as printers and plotters. But even this use of assembly language may sometimes be avoided by using high-level languages such as C, in which the vast majority of the UNIX operating system, including even the disk drive controllers, was originally written for the (now obsolescent) DEC PDP/11 series of computers. Within a Pascal type of programming environment, C allows programmers the efficient access to actual memory locations – and hence hardware devices – denied them by Pascal. More recent implementations of UNIX for DEC VAX and IBM AS/400 series minicomputers and several microcomputers have used the same idea.

If C is not available to the programmer for some reason, it may be that linking assembly language modules with a high-level language main program is the best way to avoid doing the whole job in assembly language. Different versions of the same high-level language vary enormously in both the provision they make for doing this and the complexity involved in making it all work, but overall this will almost certainly be quicker than developing a large program entirely in assembly language.

Hopefully the message is clear: for real-life problems use assembly language only as a last resort. Sometimes you don’t have any choice, as is the case if you want to modify an operating system already written in assembly language. From now on we assume that circumstances force you to read this book (!) and it is now time to commence our detailed study of assembly language.
1.3 Number systems used in assembly language

As was remarked earlier, binary numbers are the only data type available in machine code. Assembly language frees us from this constraint in some respects, not least by allowing us to work with one of three number systems — binary, decimal or hexadecimal — the assembler making any necessary conversions into binary. Thanks to our 10 fingers and 10 toes the decimal number system is the one with which we are all most familiar. Unfortunately, assembly language cannot allow us just to stick to decimal numbers: computer designers have seen to that! Also, letters of the alphabet and punctuation marks have to be coded up into numeric form so that text can be processed by assembly language programs. In this section a brief overview of these three number systems and the ASCII text coding system which has become the de facto standard on microcomputers is presented. It is intended only as a summary and a reader familiar with this material can move directly on to Section 1.4. Readers needing more information on these topics should consult Introduction to Computer Science by Neil Graham, West Publishing Co., St. Paul, Minnesota, 1985.

1.3.1 Binary numbers

As we learned at school, the decimal number 294 means

\[ \begin{align*}
2 \text{ hundreds} & + 9 \text{ tens} + 4 \text{ units} \\
    &= 2 \times 100 + 9 \times 10 + 4 \times 1 \\
    &= 2 \times 10^2 + 9 \times 10^1 + 4 \times 10^0 
\end{align*} \]

All that is different about the binary number system is that only the digits 0 and 1 are used and each place to the left represents a higher power of 2 (instead of 10). Thus the binary number 11011 is

\[ \begin{align*}
1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
    &= 16 + 8 + 0 + 2 + 1 \\
    &= 27 
\end{align*} \]

in decimal.

To avoid any possible confusion between the decimal number 10 and the binary number 10 (which is just 2 in decimal) we shall in future write the letter B after a binary number and D after a decimal number where there is any possible ambiguity. Thus 10B = 2D and 11011B = 27D as we saw above.

The number 11011B contains five binary digits (or bits). Since computers are largely concerned with performing arithmetic on binary numbers and with moving such numbers from one place in memory to another, it simplifies matters greatly if a fixed number of bits is moved each time. Members of the 8086 family of microprocessors move data around in multiples of eight bits, a
collection of eight bits being called a byte. Often two bytes (known as a word) are moved at the same time. For example:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a bit</td>
</tr>
<tr>
<td>1011</td>
<td>4 bits (sometimes called a nibble)</td>
</tr>
<tr>
<td>11100010</td>
<td>a byte</td>
</tr>
<tr>
<td>1010111001110010</td>
<td>a word</td>
</tr>
</tbody>
</table>

In a byte or word (for example, 1010110001110010) the rightmost digit is referred to as the **least significant digit** (in the example it is a 0) and the leftmost digit is referred to as the **most significant digit** (1 in the example).

Amongst the programming instructions for the 8086 family are several concerned with performing arithmetic, including those with mnemonics **ADD**, **SUB**, **MUL** and **DIV**. Arithmetic can be performed between either two bytes or two words, whichever the programmer chooses. Moreover, just as a programmer must decide whether to represent a numerical quantity by an integer or real variable in Pascal, in assembly language programmers choose whether to represent a value in a byte or word and whether as a signed or unsigned binary number. Unsigned binary numbers are the ones we have met so far and, while they do not admit the possibility of representing negative decimal numbers such as $-8147$ and $-1$ in binary, they are adequate for many purposes. Signed numbers allow both positive and negative numbers to be represented, but every signed number must be given a sign. This is done by taking the most significant digit in a byte or word as corresponding not to $2^7$ or $2^{15}$ respectively, but to $-2^7$ or $-2^{15}$. Thus, as a signed number the byte 10110000 is, in decimal,

$$1 \times (-2^7) + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= -128 + 32 + 16$$

$$= -80$$

and as a signed number the byte 00110000 is, in decimal,

$$0 \times (-2^7) + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

$$= 32 + 16$$

$$= 48$$

It follows that the most significant digit of a byte or word representing a signed number will be 0 if the number is positive and 1 if it is negative.

The signed representation of a negative number can be found in the following way. If $x$ is positive, $-x$ is represented by $2^b - x$ (word) or $2^w - x$ (byte) ignoring the most significant bit of the result of the subtraction. In binary the necessary arithmetic is surprisingly easy. For example, to represent the decimal $-42$ in signed 16-bit form we subtract 42 from $2^{16}$ as follows:

$$2^{16} = 1 \ 0000000000000000$$

$$42 = 0 \ 0000000000101010$$

$$\text{Difference} = (0) \ 11111111010110$$

and hence the 16-bit signed representation of $-42$ is $111111111010110$. This is
Table 1.2  Examples of two’s complement representations.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>One byte signed binary representation</th>
<th>Two’s complement of binary no. in column 2</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>00000001</td>
<td>11111111</td>
<td>−1</td>
</tr>
<tr>
<td>+2</td>
<td>00000010</td>
<td>11111110</td>
<td>−2</td>
</tr>
<tr>
<td>+3</td>
<td>00000011</td>
<td>11111101</td>
<td>−3</td>
</tr>
<tr>
<td>+4</td>
<td>00000100</td>
<td>11111000</td>
<td>−4</td>
</tr>
</tbody>
</table>

called the **two’s complement** representation of −42.

Another way of working out the two’s complement of a number is to carry out the following procedure on its positive equivalent: change all the 0s to 1s and all the 1s to 0s; then add 1 to the result forgetting any carry digit which goes beyond the number of bits in the representation you are seeking. Thus +37 in signed 8-bit form is 00100101 so −37 in the same form is 11011010 + 1 = 11011011 and −2 is 11111110 so +2 is 00000001 + 1 = 00000010. If you are not familiar with two’s complement representation, check the entries in Table 1.2 using both methods.

The range of signed numbers which can be held in a byte goes from −128 to +127 as follows:

−128 10000000
−127 10000001

... 
−37 11011011
... 
−2 11111110
−1 11111111
0 00000000
+1 00000001
+2 00000010

... 
+37 00100101
... 
+126 01111110
+127 01111111
Table 1.3 Comparison of signed and unsigned numbers.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Value in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsigned</td>
</tr>
<tr>
<td>00000011</td>
<td>3</td>
</tr>
<tr>
<td>11111101</td>
<td>253</td>
</tr>
<tr>
<td>01000110</td>
<td>70</td>
</tr>
<tr>
<td>11000110</td>
<td>198</td>
</tr>
<tr>
<td>0010110110001100</td>
<td>11 660</td>
</tr>
<tr>
<td>1010110110001100</td>
<td>44 428</td>
</tr>
</tbody>
</table>

For a word the corresponding range is as follows:

-32 768  10000000  00000000
-32 767  10000000  00000001
...

-2       11111111  11111110
-1       11111111  11111111
0        00000000  00000000
+1       00000000  00000001
+2       00000000  00000010
...

+32 766  01111111  11111110
+32 767  01111111  11111111

For unsigned numbers the ranges are 0 to 255 (byte) and 0 to 65535 (word). It is interesting to compare signed and unsigned representations, as shown in Table 1.3.

In general a given byte or word can represent one of two decimal numbers depending on whether it is to be regarded as signed or unsigned. Consequently, for each numeric data item in a program, the programmer must take a decision to use either a signed or unsigned representation and must stick to that choice: the two forms must not be mixed. This is rather like deciding in Pascal whether to store a date of birth as a number or as a string: both may be of use for certain applications but the two cannot be used interchangeably!

**Numbering the bits in a byte or word**

In subsequent chapters we shall often have occasion to refer to the individual bits in a given byte or word. Tradition has it that these are numbered from the
least significant end of the byte or word, and from zero upwards. Thus, given
the byte 01101100 individual bits are numbered as follows:

<table>
<thead>
<tr>
<th>bit number</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0 1 1 0 1 1 0 0</td>
</tr>
</tbody>
</table>

and given the word 0110001110000111 individual bits are numbered as:

<table>
<thead>
<tr>
<th>bit number</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>0 1 1 0 0 0 1 1 0 0 0 1 1 0 1 1 1</td>
</tr>
</tbody>
</table>

1.3.2 Hexadecimal numbers

Given that 0000000000000111 is one of the possible forms in which the
computer handles the 'harmless' decimal number 7, there is a strong
motivation for a shorthand for all those 0s and 1s. An essential feature of any
shorthand is that it must be easy to go from the full form to the shorthand and
vice versa. This is why a third number system, hexadecimal, is introduced.

In the hexadecimal number system we use the ordinary decimal digits
0,1,2,3, . . .,9 together with the first six letters of the alphabet – A, B, C, D, E
and F. A stands for 10 in decimal, B for 11 and so on up to F which stands for
15 in decimal. Otherwise everything works in the same way as for decimal and
binary except that, in this case, each place to the left represents a higher power
of 16. Thus the hexadecimal number 3FA04 is equivalent to the decimal

\[
3 \times 16^4 + F \times 16^3 + A \times 16^2 + 0 \times 16^1 + 4 \times 16^0
\]

\[
= 3 \times 16^4 + 15 \times 16^3 + 10 \times 16^2 + 0 \times 16^1 + 4 \times 16^0
\]

\[
= 196608 + 61440 + 2560 + 0 + 4
\]

\[
= 260612
\]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

Table 1.4 Hexadecimal conversion table.
At first sight the idea of a number with letters in it may appear rather strange, but it should always be borne in mind that the letters stand for numbers which in decimal require two digits (Table 1.4).

Conversion between hexadecimal and binary is very easy. To convert from hexadecimal to binary involves replacing each hexadecimal digit by its binary equivalent written as a 4-bit number. Thus, to represent hexadecimal 3FA04 in binary:

<table>
<thead>
<tr>
<th>3</th>
<th>F</th>
<th>A</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>1111</td>
<td>1010</td>
<td>0000</td>
<td>0100</td>
</tr>
</tbody>
</table>

so the binary equivalent of 3FA04 is 0011111101000000100. Going the other way is just as easy:

<table>
<thead>
<tr>
<th>3</th>
<th>F</th>
<th>A</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>0110</td>
<td>0011</td>
<td>1001</td>
<td>1110</td>
</tr>
</tbody>
</table>

so that the hexadecimal equivalent of binary 101101100011001110 is B639E. Hopefully this justifies the use of the hexadecimal number system as a shorthand for binary.

**Pronunciation**

It you’re not sure how to pronounce hexadecimal numbers such as 1A, 1B..., you could always follow Bilbo Baggins and refer to 1A as onety-A and so on. More often than not, however, hexadecimal numbers like B639E are read as ‘bee-six-three-nine-ee’.

### 1.3.3 Converting decimal to hexadecimal

To convert a decimal number to hexadecimal we repeatedly divide the decimal number by 16 until a zero quotient is obtained. The remainders from the divisions then give you the equivalent hexadecimal number, the last remainder obtained being the highest order digit of the hexadecimal number. For example, the conversion of decimal 1103 to hexadecimal looks like:

\[
\begin{align*}
1103 \div 16 &= 68 \text{ remainder } F \\
68 \div 16 &= 4 \text{ remainder } 4 \\
4 \div 16 &= 0 \text{ remainder } 4 \\
\end{align*}
\]

so that the hexadecimal equivalent of 1103 is 44F.

Converting decimal numbers to binary is carried out in a similar fashion except that you divide by 2 each time. Thus, to find the binary equivalent of decimal 46:

\[
\begin{align*}
46 \div 2 &= 23 \text{ remainder } 0 \\
23 \div 2 &= 11 \text{ remainder } 1 \\
11 \div 2 &= 5 \text{ remainder } 1 \\
5 \div 2 &= 2 \text{ remainder } 1 \\
\end{align*}
\]
Table 1.5  Decimal and hexadecimal equivalence table.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>1000</td>
<td>3E8</td>
</tr>
<tr>
<td>10000</td>
<td>2710</td>
</tr>
<tr>
<td>100000</td>
<td>186A0</td>
</tr>
<tr>
<td>1000000</td>
<td>F4240</td>
</tr>
<tr>
<td>10000000</td>
<td>989680</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
2 \div 2 &= 1 \text{ remainder } 0 \\
1 \div 2 &= 0 \text{ remainder } 1
\end{align*}
\]

so the binary equivalent of decimal 46 is 101110.

For future use, some equivalences between decimal and hexadecimal numbers are shown in Table 1.5. As an exercise in converting decimal numbers to hexadecimal form the reader is invited to verify as many of the entries as time allows.

Further practice in conversion from hexadecimal to binary can be obtained by converting the right-hand entries to binary and then checking that the decimal value of the binary numbers obtained is that given in the left-hand column. For instance, 3E8 in hexadecimal is 001111101000 in binary which is 512 + 256 + 128 + 64 + 32 + 8 = 1000 in decimal so that row four of Table 1.5 is verified.

### 1.3.4 Hexadecimal addition and subtraction

We need to be able to do hexadecimal addition and subtraction in order to verify the results of running assembly language programs during the debugging phase. Because we are so used to thinking in decimal, hexadecimal multiplication and division are best done by converting the numbers to decimal, multiplying or dividing, and then converting the result back to hexadecimal.

In order to distinguish between hexadecimal and decimal numbers, we write the letter H after a hexadecimal number. Thus, at the end of a number, a B denotes a binary number, a D denotes a decimal number and an H denotes a hexadecimal number.

Addition and subtraction are relatively easy. When adding or subtracting two hexadecimal numbers, the secret is to think of the decimal equivalent of each digit, to add or subtract the decimal equivalents, and then to reconvert the results. Thus, to perform the addition:

\[
\begin{align*}
+ & \quad 3A6CH \\
B2E1H & \\
\hline
\end{align*}
\]
we think of the decimal equivalents of the first pair of digits (12 and 1 respectively), add those (giving 13), and convert this back (giving the hexadecimal digit D):

\[
\begin{array}{c}
3A6C \\
\downarrow \\
+ B2E1H \\
\hline
D \\
\end{array}
\]

For the next pair of hexadecimal digits to be added, we think of their decimal equivalents (6 and 14 respectively), add these (giving 20), and convert this back to hexadecimal (giving 14H) so we write down the 4 and carry the 1:

\[
\begin{array}{c}
3A6C \\
\downarrow \\
+ B2E1H \\
\hline
4D \\
\hline
1
\end{array}
\]

Now we think of the decimal equivalents of the next pair of digits (10 and 2 respectively), add these and the carry (giving 13), and convert this back (giving the hexadecimal digit D):

\[
\begin{array}{c}
3A6C \\
\downarrow \\
+ B2E1H \\
\hline
D4D \\
\hline
1
\end{array}
\]

Finally we think of the decimal equivalents of the remaining pair of digits (3 and 11 respectively), add these (giving 14), and convert this back (giving the hexadecimal digit E):

\[
\begin{array}{c}
3A6C \\
\downarrow \\
+ B2E1H \\
\hline
ED4D \\
\hline
1
\end{array}
\]

Subtraction is performed in a similar way.

### 1.3.5 Representing text

Computers are able to handle non-numeric data as well as numbers. In fact, most ‘real world’ computer use at the moment involves non-numeric computing, such as searching a file of accounts to find all outstanding invoices
in the name of J.S. Gruntfuttock. The ability to handle text derives from coding the alphabet and punctuation marks into numbers. For example, we could use the number 41H as a code for the letter A, 42H for the letter B, 43H for the letter C, 2CH for the symbol ‘,’ (comma), 3BH for the symbol ‘;’ (semicolon), and so on. Given a numerical code, the processing of text is reduced to processing numbers. Thus searching a file for an invoice under the name BACCA actually involves searching for an invoice beginning with 42H, 41H, 43H, 43H and 41H.

In microcomputers such as the IBM-PC the code most often used is the ASCII (pronounced ass-key) code – the American Standard Code for Information Interchange – from which the above examples were taken. The complete ASCII code is given in Appendix 1 for ease of reference. By the time you have finished reading this book that appendix will be well-thumbed. Take a first look at it now and verify that the examples given above conform to the ASCII code.

Besides the alphabetic characters A, B, C, ..., Z, a, b, ..., z and punctuation marks, the ASCII code also includes codes for certain characters which are never printed but rather are used to control devices external to the main computer, such as a printer. Other codes are used to control communications between the computer and a remote device, rather like the ‘over and out’ communications protocol used by aeroplane pilots. Thus, in Appendix 1 there is an ASCII code to get a printer to move onto a new line – LF (Line Feed); to send the printing head back to the beginning of the line – CR (Carriage Return); to signal an ENQuiry – ENQ; and to signal the End Of Transmission – EOT.

Originally ASCII codes contained just seven bits and could thus be used to represent 128 ($2^7$) characters. When placing data in memory, multiples of eight bits are used and so it is natural to use a single byte for each character. The additional eighth bit (added at the most significant end of the code) is put to use in two different ways.

First, in order to extend the range of characters which the PC can display on its screen, IBM has extended the ASCII code by using this eighth bit to represent an additional 128 characters. This permits such weird and wonderful display characters as Greek letters, playing-card suits, square root signs and smiling faces to be represented.

Second, when data is transmitted over relatively long distances, for example over telephone lines, there is the possibility of corruption due to electrical noise. The eighth bit can be used to afford protection against such data corruption by setting it to 0 or 1 as necessary to ensure that the code for every character has an even number of bits set to 1 (even parity) or to ensure that every character’s code has an odd number of bits set to 1 (odd parity).

For example, the 7-bit ASCII code for the letter A is 1000001. If even parity is used this becomes 01000001 while odd parity would give 11000001. Hardware can be designed to check for parity discrepancies. Thus, if we were using odd parity and 11000011 were received by the computer over a telephone line, the hardware would detect that this could not be the code of any real data
(since it contains an even number of bits set to 1) and would signal an error.

To summarize, inside the PC itself the eighth bit is used to give extra characters. When communicating with external devices – another computer or a printer perhaps – the eighth bit can be used to guard against transmission errors if we do not want to transmit or receive these extra characters.

EXERCISES

1.1 Complete the following table:

<table>
<thead>
<tr>
<th>16-bit binary (unsigned)</th>
<th>4-digit hexadecimal</th>
<th>Decimal</th>
<th>16-bit binary (signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000000000000000</td>
<td>001F</td>
<td>31</td>
<td>0101010101010101010101</td>
</tr>
<tr>
<td>000111000111000000000000</td>
<td>AE2B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000111000111000000000000</td>
<td>1E1E</td>
<td>2345</td>
<td></td>
</tr>
<tr>
<td>000111000111000000000000</td>
<td>ABCD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.2 Complete the following table:

<table>
<thead>
<tr>
<th>16-bit (signed)</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000000000000000011111</td>
<td></td>
</tr>
<tr>
<td>100011100011100100000000</td>
<td></td>
</tr>
<tr>
<td>000010010010000000000000</td>
<td></td>
</tr>
<tr>
<td>010000000000000000000000</td>
<td></td>
</tr>
<tr>
<td>110000000000000000000000</td>
<td></td>
</tr>
</tbody>
</table>

1.3 Find the two's complement of the 16-bit signed number 0110110110110111. Regard the result as a binary number \( x \) and find the two's complement of \( x \).

1.4 Decipher the following ASCII-coded secret message:

02 4D 45 65 54 20 6D 45 20 41 74 20 43 41 52 4E 45
67 49 65 20 48 41 6C 4C 20 31 34 32 3E 33 30 20 03

1.5 Perform the following hexadecimal additions and subtractions:

(a) 23ABH + 0AC34H  (d) 296BH + 7A4FH
(b) 9458H + 9977H   (e) 6901H + 996FH
(c) 0F0BH + 8FFFH   (f) 1111H + 0FFFFH
(g) 734BH – 9F3BH
(h) 9938H – 34FFH
(i) 290DH – 1FA8H
(j) 0FE43H – 0E229H
(k) 5FEFH – 1212H
(l) 89CBH – 123CH

1.4 Some simple 8086-family assembly language instructions

Since we have now covered the number systems used in assembly language we can start to explore the 8086-family's instructions which permit arithmetic to be performed. The majority of these instructions involve registers in some way or other and it is to a description of four of the registers possessed by all members of the 8086 family that we turn first.

1.4.1 Registers

My pocket calculator is one of the simplest and cheapest available. To get it to add two numbers, say 3 and 2, you have to follow the three steps of Figure 1.3.

A calculator display is a good model for a register in a microprocessor. Registers are places where data can be processed particularly quickly. Members of the 8086 family are very well endowed in this respect. One of their registers is called the AX register and this can be used to add 3 and 2 in much the same way as the simple calculator just discussed. Here is the program:

```
MOV AX,3 ;put 3 into register AX
ADD AX,2 ;add 2 to contents of register AX
```

(Here, anything appearing after a semicolon (;) is a comment to help humans understand the program. Only the instructions to the left of the ; are obeyed.

![Figure 1.3](image)

A simple calculator.
by the microprocessor.) As was the case with the calculator example above, the
answer '5' would be left in the register AX.

Actually the members of the 8086 family have four general-purpose
registers: AX, BX, CX and DX, each of which holds one 16-bit number. Our
simple addition program could have used any one of these four by replacing
AX with the name of one of the other registers throughout. Moving data
between registers is accomplished by the MOV instruction. Appending the
following instructions to the addition program would result in the value 5 being
put into all the registers AX, BX, CX and DX.

    MOV BX,AX ; copy what's in register AX into register BX
    MOV CX,AX ; copy what's in register AX into register CX
    MOV DX,CX ; copy what's in register CX into register DX

So often in programming do we want to add 1 to the contents of a
register that the 8086-family have an instruction specially for that purpose:
the INC instruction. Thus,

    INC DX

would add 1 to the current contents of register DX.

But we can do more than just add! Subtraction and three sorts of multiplic-
tion and division are possible as well. Consider the following little program:

    MOV AX,5 ; put 5 into the AX register
    MOV BX,4 ; put 4 into the BX register
    SUB AX,2 ; subtract 2 from the AX register
    MUL BX ; multiply the unsigned 16-bit number in the BX
             ; register by the unsigned 16-bit number in the
             ; AX register. Leave the 32-bit answer
             ; in the DX and AX registers. DX
             ; containing the binary digits
             ; corresponding to $2^5 = 65536$ and
             ; higher powers of 2

It will calculate $(5 - 2) \times 4$ and leave the answer (12 in decimal, C in
hexadecimal, 00000000 00000000 00000000 00001100 in unsigned 32-bit form)
in 32-bit form in registers DX and AX: thus DX will contain 0000000000000000
and AX will contain 000000000001100. The stages of the calculation are
illustrated in Figure 1.4 in which we have adopted the convention which will
often be followed in this book, namely, the contents of registers are given in
hexadecimal notation. Note that in the MUL instruction, AX is one of the

<table>
<thead>
<tr>
<th>AX</th>
<th>BX</th>
<th>CX</th>
<th>DX</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AX,5</td>
<td>5</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>MOV BX,4</td>
<td>5</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>SUB AX,2</td>
<td>3</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>MUL BX</td>
<td>C</td>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>

Figure 1.4
Stages of the calculation of $(5 - 2) \times 4$.

? denotes a value
which we do not know in advance.
(Before our
program starts the
registers will contain
whatever the
previous
program left in
them.)
operands even though the assembly language mnemonic does not mention it: 
**MUL** **BX** causes 8086-family microprocessors to work out **BX** multiplied by **AX**. It 
would be a reasonable criticism of 8086-family assembly language instruction 
mnemonics to say that each and every instruction should have explicit 
operands. Some versions of 8086-family assembly language do require this, but 
we shall adopt the most popular version, MASM, which does not.

**Removing ambiguity**

Calculators work in decimal of course, but microprocessors use binary 
arithmetic. Thanks to assembly language we can write instructions like

```
MOV AX,7
```

and leave it to the assembler to convert the number 7 into its appropriate 
binary equivalent. However, there is now the possibility of ambiguity. For 
example, does

```
MOV AX,26
```

refer to the decimal number 26, or hexadecimal 26 (2 \* 16^1+6 \* 16^0 = 38 in 
decimal)?

To remove any ambiguity as far as the assembly language conversion is 
concerned we write 26H for hexadecimal 26 and 26D for decimal 26. Likewise 
10B is binary, 10D is decimal and 10H is hexadecimal. Actually, assembly 
language instructions are usually assumed to specify numbers in decimal unless 
the programmer decides otherwise by adding a code letter after the number (or 
in some other way). For the time being it is better always to add a letter 
explicitly indicating which sort of number it is that you are talking about. In 
this way you will become used to thinking about the problem of ambiguity of 
meaning.

### 1.4.2 Size of registers

Given the benefits of assembly language, we still cannot ignore completely the 
fundamental reliance of microprocessors upon the binary number system. A 
pocket calculator has a limit to the number of decimal digits you can enter — 
mine accepts 98 765 432 (eight digits) but not 198 765 432 (nine digits), for 
example. Likewise the registers **AX**, **BX**, **CX** and **DX** are limited to 16 binary 
digits (bits) so that in an instruction

```
MOV AX,n
```

the number **n** must not need more than 16 bits for its binary representation. In 
other words, in decimal **n** must be non-negative and less than or equal to 
65 535 for an unsigned number, or between −32 768 and 32 767 for a signed 
number. This also means that all four registers are limited to four hexadecimal 
digits.
<table>
<thead>
<tr>
<th>Program</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AX</td>
</tr>
<tr>
<td>MOV AX, 0</td>
<td>?</td>
</tr>
<tr>
<td>ADD AX, 6</td>
<td>0</td>
</tr>
<tr>
<td>MOV BX, 4</td>
<td>6</td>
</tr>
<tr>
<td>MOV CX, 5</td>
<td>6</td>
</tr>
<tr>
<td>ADD AX, BX</td>
<td>0A</td>
</tr>
<tr>
<td>SUB AX, 3</td>
<td>7</td>
</tr>
<tr>
<td>MUL BX</td>
<td>1C</td>
</tr>
</tbody>
</table>

Unlike most pocket calculators, the number entered into AX, BX, CX or DX must consist of exactly 16 binary digits. Thus, the assembly language

```assembly
MOV AX, 7H
```

would actually be converted to the machine code equivalent of

```assembly
MOV AX, 00010000 00000000 00000001 0110 0000
```

since the binary for 7H is 111. Fortunately this is done for you automatically.

### 1.4.3 Making traces

Following through the execution of any assembly language program consisting of more than a couple of lines can be a difficult undertaking. It helps to keep a trace of register contents as we did in Figure 1.4. As another demonstration of this technique we now give an assembly language program involving ADD, SUB and MUL together with a trace of its execution. In the trace, an entry ‘?’ will denote a value which we do not know. Thus, the contents of all four registers are indicated as ‘?’ at the beginning of the trace when a particular program makes no assumption about the contents of registers before it is executed.

To make the trace we simply write down the contents of each of the registers before the program starts and then again after each instruction has been executed (Figure 1.5). Notice that the program uses the instruction:

```assembly
ADD CX, BX ; add the number in register CX to the number in register BX. Put the answer in register CX and leave BX unchanged
```

There are similar forms of the ADD instruction involving each of the 12 possible combinations of two different registers chosen from AX, BX, CX and DX, with a similar effect in each case.
EXERCISES

1.6 What will be the contents of the AX, BX, CX and DX registers after executing each of the following program fragments?

(a) MOV CX,3
   ADD CX,5
   MOV BX,CX
   INC BX
   MOV AX,BX
   (b) MOV DX,8
   MOV AX,9
   SUB DX,4
   MUL DX
   MOV CX,DX
   INC CX
   SUB CX,1
   MOV BX,CX

1.7 Write a program to leave the result of $5 \times (7+1)-6$ in register CX.

1.8 By making a careful trace determine the post-execution contents of all the registers involved in each of the following program fragments.

(a) MOV AX,0F1H
    MOV BX,0ABCH
    ADD AX,BX
    MOV CX,3BH
    SUB AX,OFH
    MUL BX
    (b) MOV AX,2
    MUL AX
    MUL AX
    MOV BX,3
    ADD AX,BX
    ADD AX,BX
    MUL BX
    (c) MOV AX,0H
    MOV BX,34H
    MOV AX
    MOV BX,35H
    MOV DX,0FH
    ADD BX,DX
    ADD AX,BX
    INC CX
    MUL CX

1.5 Eight ‘new’ registers

This section could have been entitled ‘New Lamps for Old’ because the promised eight new registers actually come from our four familiar ones, AX, BX, CX and DX. We already know that each of these is a 16-bit register, but

![Figure 1.6](image)

Each 16-bit register AX, BX, CX, and DX can be used as two 8-bit registers.
the 8086 family allows each of them to be used as two 8-bit registers as well. The leftmost eight bits of AX form an 8-bit register, AH, and the rightmost eight bits of AX form an 8-bit register, AL. Similarly, we get BH and BL from BX, CH and CL from CX, and DH and DL from DX (see Figure 1.6).

One can use instructions like MOV AL,BH; MOV DL,CL; MOV BH,3; ADD BL,DH; and ADD DL,DH, which have effects similar to those for their 16-bit register equivalents. If MUL is used with an 8-bit register as one of its operands, then the other operand is AL and the result is left in AX. Thus, MUL BH will multiply the unsigned number in BH by the unsigned number in AL and leave a 16-bit unsigned product in AX.

Any instruction must either operate entirely with 8-bit registers or entirely with 16-bit registers. Thus, MOV AX,DL is not allowed. (By the way, the H in AH stands for High and the L in AL stands for Low since if AX contains a 16-bit number, AH will contain the highest order bits and AL the lowest.)

It is important to remember that changing the value of one of the 8-bit registers will affect the corresponding 16-bit register and vice versa. Thus, after execution of:

```
MOV DH,98H
MOV DX,23ABH
ADD DH,1
```

DH will contain 24H (not 99H) and DX will contain 24ABH.

To see why, first note that 98H = 1001 1000B. MOV DH,98H therefore sets register DH to 10011000 with register DL containing whatever value it had previously:

```
          DX
10011000????????
          DH  DL
```

But MOV DX,23ABH affects the whole of DX and so affects both DH and DL. Since

```
23ABH = 0010 0011 1010 1011B
```

the effect of MOV DX,23ABH is to leave DX containing

```
          DX
0010001110101011
          DH  DL
```

Now ADD DH,1 will add 1 to the eight most significant bits of DX, that is to say 00100011, giving 00100100, and put this result in place of the original eight bits. As a result, DX will then contain:

```
          DX
0010010010101011
          DH  DL
```

and 0010 0100 1010 1011B = 24ABH.
**Danger — ambiguity**

There is now the risk of confusion as to whether

```
MOV DL, AH
```

means 'copy the contents of register AH into register DL' or 'put the hexadecimal number A into register DL'. To eliminate such ambiguity there is a rule in assembly language programming that all hexadecimal numbers beginning with a letter (for example A123H and E1FCH) must be written with a zero preceding them (so that our examples become 0A123H and 0E1FCH). Hence 'copy the contents of register AH into register DL' is now unambiguously

```
MOV DL, AH
```

and 'put the hexadecimal number A into register DL' becomes

```
MOV DL, 0AH
```

---

**EXERCISES**

1.9 Complete the following:

```
<table>
<thead>
<tr>
<th>AX</th>
<th>BX</th>
<th>CX</th>
<th>DX</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td>BH</td>
<td>BL</td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>DH</td>
<td>DL</td>
</tr>
</tbody>
</table>
```

00000000 00000010 01010101 00000000 11111111 11111111 10011001 11111111

to make a trace of the execution of the following program fragment:

```
MOV DL, 3
MOV CX, 5
MOV BH, DL
SUB CX, 7
ADD AH, 2
ADD CH, 5
MOV DX, AX
```

1.10 By making a careful trace determine the post-execution contents of all the registers involved in each of the following program fragments.

(a) MOV DL, 0F1H

```
MOV BL, 0BCH
ADD DL, BL
MOV CH, 3BH
SUB DL, OFH
```

(b) MOV AH, 2

```
MOV BH, 3
ADD AH, BH
ADD AH, BH
MOV AL, 2
MOV BL, 1
MUL BX
```

(c) MOV AX, 0H

```
MOV BL, 34H
MOV CH, 35H
MOV DX, 0FH
MOV CL, DL
MOV BH, CH
ADD BX, DX
SUB CX, 01CH
ADD BL, CH
INC CX
MUL CX
```
1.6 The flags register

The 8086-family microprocessors have a special 16-bit register called the **flags register** because the individual bits are used as flags to indicate the result of executing certain instructions. Thus, just as the British flag flies above London’s Buckingham Palace only when the Queen is in residence (as tourists in England hoping to catch a glimpse of Her Majesty soon learn), so the Z-flag in the flags register can be set to 1 by any of a certain group of instructions to indicate that the result of executing that instruction was zero.

Altogether, five of the bits in the flags register (Figure 1.7) are used to indicate the results of arithmetic and related operations and are referred to as the **arithmetic flags**. These are the O-flag (overflow), the S-flag (sign), the Z-flag (zero), the A-flag (auxiliary carry) and the C-flag (carry). The P-flag indicates the parity of a result and is primarily used in data communications while three other bits are used to control processor actions like the direction in which large blocks of data are to move. The rest of the bits in the flags register are unused except that the 80286 microprocessor uses bits 12, 13 and 14 to organize **multitasking** (see Section 2.2).

To see how the flags work in relation to arithmetic instructions let us consider the carry flag. This is often used to hold the carry bit (either 0 or 1) resulting from an arithmetic operation. Thus, the sequence of instructions:

\[
\text{MOV } \text{AL}, 3H \\
\text{MOV } \text{BL}, 0FFH \\
\text{ADD AL, BL}
\]

will try to set AL to the result of \(3H + 0FFH = 102H\). But \(102H = 100000010\)H which is nine bits in total and therefore too much to be held in AL. Because of the particular values in AL and BL when it is executed, \text{ADD AL, BL} will therefore set the carry flag to 1 to indicate that the total is too big to fit in AL. Flags will be dealt with in more detail in Chapter 6, which examines their role in arithmetic operations, and in Chapter 7, which looks at how we can take different actions depending on the current value of a flag.
SUMMARY

In this chapter we have seen that assembly language is a last resort for most programming tasks. High-level languages offer environments in which the programming of a task can proceed much more efficiently. Assembly language offers two basic data types – binary signed or unsigned numbers – and all data items for a given problem have to be represented in numeric form before they can be used in a machine code program; in particular, text is represented using the ASCII code.

We met the idea of a microprocessor register which enables the programmer to store binary numbers and perform various operations on them, including arithmetic. These operations are determined by 8086-family instructions and we encountered the MOV, ADD, SUB, INC and MUL instructions for the first time.

To understand what happens during the execution of a sequence of 8086-family instructions the technique of tracing was presented as being particularly useful to the beginner. As we shall see in due course, much more sophisticated facilities exist to help the programmer, but even very experienced assembly language programmers use tracing as the last-resort method for locating errors.

We saw how AX, BX, CX and DX can also be thought of as eight 8-bit registers. While a large number of registers can sometimes seem unnecessarily complicated to the beginner, just a little more assembly language programming experience will soon lead you to believe the professional programmer’s maxim that ‘you can never have too many registers’ (though, as with most maxims, there are plenty of exceptional cases).

Finally, we briefly introduced the flags register which is very important because individual flags within it indicate the outcome of arithmetic operations, and because which instruction is executed next can be made to depend on certain flag settings – as we shall see in Chapters 6 and 7.

SELF-CHECK QUIZ

1. For each of the following programming tasks, say whether you think it would be more appropriate to write the corresponding program in Pascal or assembly language:

   (a) A college student record system to ensure that all students pay their fees and to record each student’s academic progress at the college.

   (b) A spreadsheet.

   (c) A system to record rainfall directly from an electronic rain detector placed on the roof of the main campus administration block and connected directly to the computer. Monthly printed reports on the levels of daily rainfall are required.
2. The following extract shows part of a program written for a computer based on a member of the 8086 family. Is it in machine code? If you think it is, explain why. If you think it isn’t, explain what changes would have to be made to the program extract for it to be in machine code.

```
1 E 3 6
2 A 1 2
3 D0 3
4 D1 3
5 C 0
6 J 2
```

3. Convert the following numbers as required:

(a) from decimal to binary
   (i) 18 (ii) 78 (iii) 364 (iv) 14,241

(b) from hexadecimal to binary
   (i) 1AB8H (ii) 2EFAH (iii) 1364H
   (iv) 14AAH

(c) to hexadecimal
   (i) 1011001101000111B
   (ii) 0011010110111111B
   (iii) 849D (iv) 7134D

(d) from hexadecimal to decimal
   (i) 19A26H (ii) 0AFFFH (iii) 1364H
   (iv) 2CH

4. Given that
   (a) \( P = 11001001B \) \( Q = 01110001B \)
      \( P \) and \( Q \) are signed numbers
   (b) \( P = 01110011B \) \( Q = 11100001B \)
      \( P \) and \( Q \) are unsigned numbers
   (c) \( P = 3ABCH \) \( Q = 0CD2FH \)

   evaluate the following arithmetic expressions in each case:
   (i) \( P + Q \) (ii) \( P - Q \) (iii) \( P + P - Q \)
   (iv) \( Q + Q - P \)

   (Use two’s complement arithmetic between signed numbers.)

5. Write an explanation of the difference between signed numbers and unsigned numbers. Your explanation must not contain any kind of numeric example: use words only!

6. (a) Write the following messages as sequences of ASCII codes:
   (i) PETER GRIMES
   (ii) The Burning Fiery Furnace
   (iii) Opus 64B No 3
   (iv) −273

   (b) Decode the following ASCII sequences into the corresponding text equivalent:
   (i) 6C 75 78 75 72 79 20 63 61 72
   (ii) 41 4E 44 4C 59 53 49 53
   (iii) 31 39 38 39 31 39 39 30 31 39 39 31

7. Write sequences of 8086-family assembly language instructions which use registers AX, BX, CX and DX to evaluate the following expressions:
   (a) \( 36 - (4 + 8) \)
   (b) \( (41 - 26) + (83 - 69) \)
   (c) \( (8 \times 2) - 7 \)
   (d) \( (9 \times 8) - (4 \times 7) \)

8. Use the 8-bit registers AH, AL, .., DH, and DL to evaluate the following expressions:
   (a) \( (16 - 5) - (8 + 11) - (2 \times (9 - 4)) \)
   (b) \( (11 \times 4) + (2 \times 5) + ((3 \times 4) \times (2 + 3)) \)

   Try to repeat this exercise using only registers AX, BX, CX and DX and hence explain an advantage of having eight registers rather than four. Can you describe any disadvantages of having many registers?

9. What will be the contents of AX, BX, CX, DX, AH, AL, BH, BL, CH, CL, DH and DL after execution of the following program fragment?

   ```
   MOV AX, 4C0DH
   MOV BX, 2AAFH
   MOV CX, 2
   MUL CX
   SUB BX, AX
   ADD AL, BL
   ADD DL, CL
   ```
10. Write assembly language instruction sequences to evaluate each of the expressions:

\[ 11001001_B + 01110001_B \]
\[ 01011101_B + 00110101_B \]

\[ 01111001_B + 11100001_B \]

using 8-bit registers. What will be the contents of the carry flag after executing each such sequence?