Assembly Language Primer for IBM PC:

Featuring the 8088 Microprocessor

Learning and using assembly language
By Robert Lafore

INTRODUCTION

ASSEMBLY language has the reputation of being difficult to learn; especially from the pages of a magazine. We think this reputation is largely undeserved, and that assembly language can be taught as simply and easily as other languages such as BASIC and Pascal. In the following article, we present assembly language for the absolute beginner: the person who has never programmed in assembly language before.

This article is excerpted from the forthcoming *Assembly Language for the IBM PC,* a Waite Group book by Robert Lafore. (As you may recall, Mr. Lafore and Mr. Waite wrote *Soul of CP/M,* serialized in these pages in 1983.) This new book is a complete introduction to 8088 assembly language and forms part of an integrated series of **New American Library** language books on the IBM PC issued under the dual *Plume/Waite* imprint.

How is it possible to teach assembly language so simply? Mr. Lafore makes use of several innovative techniques. First, by basing his work on the IBM PC, he ensures that everyone will be working with exactly the same equipment. Books which attempt to teach assembly language for a particular *microprocessor chip* must be excessively vague about the actual commands used to perform a given operation, since the same chip can be used in many different machines.

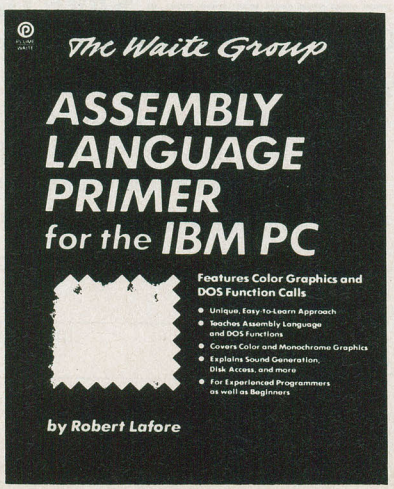
Assembly Language for the IBM PC also uses the "miniassembler" built into the DEBUG program. This feature provides a greatly simplified entry into assembly language, since the reader need not start off by learning the complexities of the full-scale IBM Macro-Assembler program. Finally, by making use of the powerful *DOS functions* built into the operating system, the book can start out with programs that are very short and simple, but that nevertheless accomplish significant tasks. As you will learn, programming through DOS functions is *the* professional approach; it means your programs will work on all computers that run a version of the popular MS-DOS operating system.

For those of you who have always wanted to learn assembly language, but have not known how to begin, this is your chance!

ASSEMBLY LANGUAGE is always the fastest and most powerful language for a given computer. It is essential in programs where pure speed of operation is important, such as graphics, sorting, and sustained number-crunching. It is also the only language that can make use of *all* of a particular machine's hardware features. With higher-level languages, such as BASIC or Pascal, the programmer is always insulated from the computer by the language itself—he can only do what the writers of the language decided he should be able to do, so inevitably he cannot tap the full power of the computer.

For these reasons, many types of programs—such as operating systems, compilers, word processors, and graphics programs—are almost always written in assembly language. If you want to do this sort of programming, then you need to know assembly language.

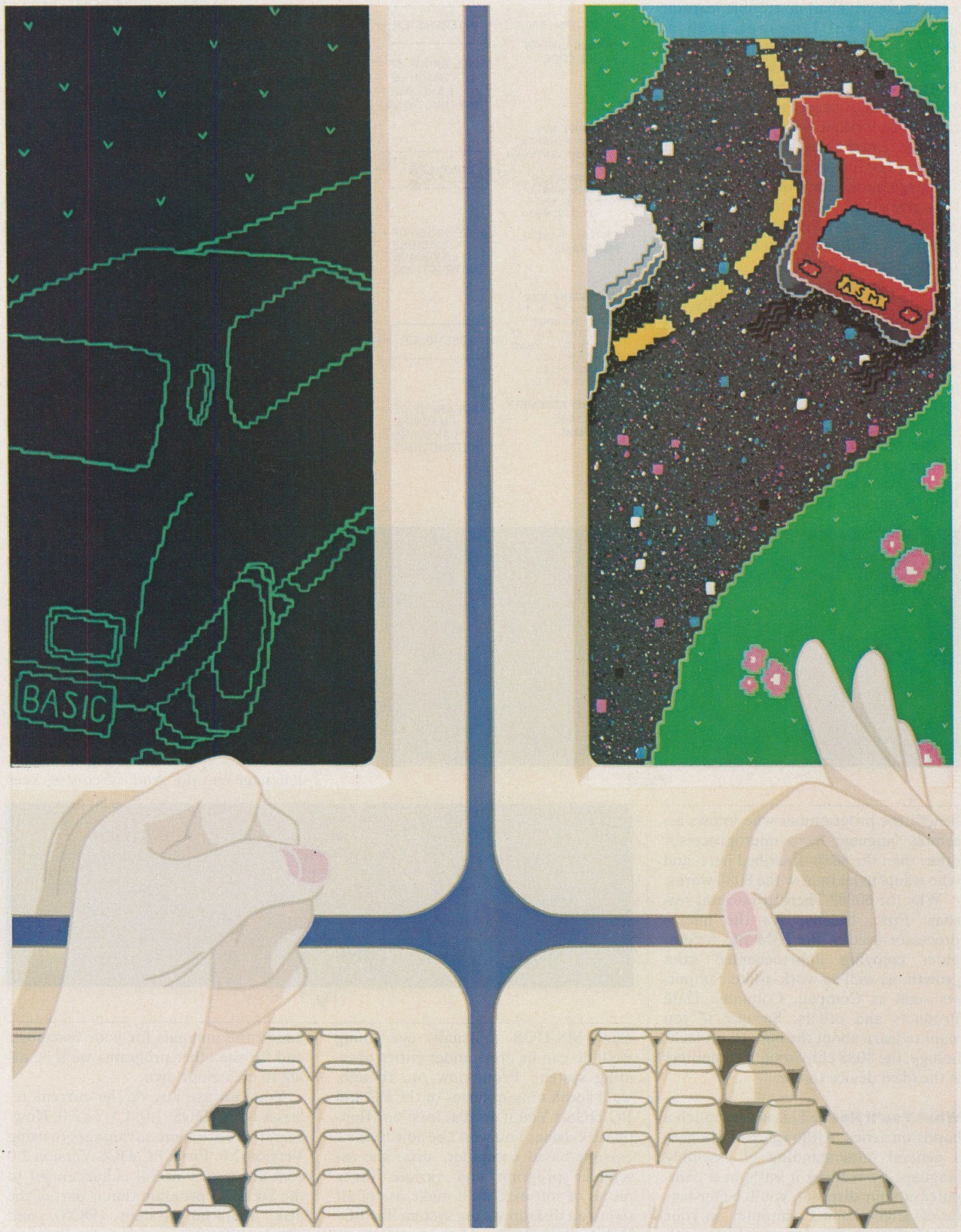
But assembly language is not only practical, it is also a fascinating and rewarding field of study. Because it is so close to the physical reality of the computer, everything you do in assembly language is the result of the way the computer operates, not the way designers of a higher-level language decided to do things for the sake of ease and convenience.



We can think of higher-level languages as being like stodgy luxury sedans: they're comfortable and easy to use, but the steering is imprecise, the suspension insulates you from the feel of the road, and if you try to push them too fast they slide into the ditch.

Assembly language, on the other hand, is the sports car of computer languages. In a sports car you're close to the road, the steering, brakes and gears are light and precise, and the car is built for speed and efficiency. It may not be as comfortable as a sedan, but it's fast and, more important, it's fun to drive.

Is Assembly Language Hard to Learn? Unfortunately, assembly language has developed the reputation of being difficult to learn. Many people—



Assembly language is the sports car of computer languages. . . . It may not be as comfortable as a sedan but it's fast and it's fun to drive

even those who had no trouble learning a higher-level language such as BASIC—think that assembly language is somehow beyond them. This belief is fostered by many assembly-language books which, strange as it may seem, appear to be written with the assumption that the reader already knows about the subject. For instance, many assembly-language books start off by listing and describing all of the scores of microprocessor instructions. As a result, most readers give up before finishing the text.

We believe that assembly language, in spite of its reputation, is actually not much harder to learn than any other computer language, provided it is presented so you do not feel overwhelmed at the beginning. It's this sort of easy, step-by-step presentation that will be used in this series.

The series is intended primarily for the person who has no previous acquaintance with assembly language: the rank beginner. However, it will also benefit the programmer who knows assembly language for a microprocessor other than the 8088 described here, and who wants to learn how the 8088 works.

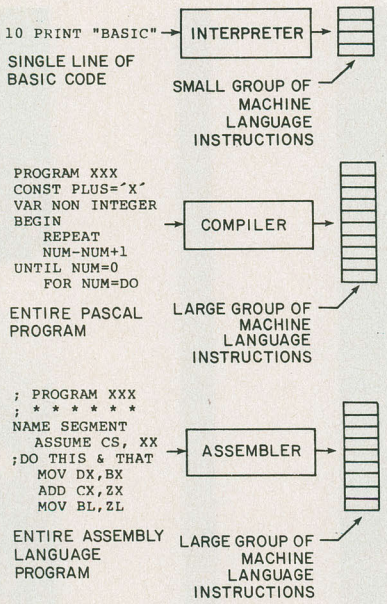


Fig. 1

```
A>debug
-d100
04B5:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-_
```

Fig. 2

What You'll Need. This is very much a hands-on series. Although you can gain a general understanding of assembly language by reading it without a computer at your disposal, you'll be far better off if you have a computer on your desk before you start to read. As with other languages—both human and computer—it's only through practice that real mastery is achieved.

We'll assume therefore that you have access to an IBM PC using PC-DOS or to another 8088-CPU based computer using MS-DOS. (Virtually everything we'll do can be done under either operating system. From now on, though, we'll speak only in terms of the PC and PC-DOS.) You'll need at least one floppy-disk drive. You won't be able to use a cassette-based computer, since the assembler program and various other pieces of software we'll make use of all require a disk operating system like PC-DOS or MS-DOS.

How much memory do you need to create assembly-language programs? That depends on which assembler you want to use. When you buy the standard IBM Macro Assembler for the PC, you actually get two assemblers in the same package: ASM and MASM. MASM stands for "Macro ASseMbler," and is a full-scale assembler with all the bells and whistles. If you use it, you'll need a minimum of 96K of RAM.

ASM, which is sometimes called the "Small Assembler," is a more modest version of MASM. It will run in a 64K system if you are using PC-DOS Version 1.0 or 1.1. However, if you are using DOS Version 2.0, then you will need a minimum of 96K, with 128K being preferable if you get into writing long programs.

You can use any video display you like, since the concepts we'll explain here don't require the use of color. However, all the examples we'll give are based on an 80-column display. If you are using only 40 columns, you will need to do a little mental reformatting to compare the printouts here with those on your screen, but that should not pose a major problem.

It's very nice, but not absolutely necessary, to have a printer when writing assembly-language programs. A printed listing is convenient for debugging and for following the overall operation of a program, but a printer is like a house in the country: If you have one, you'll love it, but if you don't, you'll get along just fine anyway.

You'll find the Technical Reference Manual for your computer extremely useful. It's packed full of details about the computer's operation, and many of those details will become important to us as we explore the things assembly language can do. And, of course, keep handy the manuals for your assembler and for the other programs we'll be using to create our own.

```
-d100
04B5:0100  03 EB 42 90 75 03 EB 41-90 20 30 2C 30 30 3C 0A  .kB.u.kA. 0,00<.
04B5:0110  73 34 52 8B D3 9F 03 DB-03 DA 03 DB D1 DE 9E D1  s4R.S..[.Z.[Q^.Q
04B5:0120  D6 8A D0 B6 00 9F 03 DA-D1 DE 9E D1 D6 5A E0 21  V.P6...ZQ^.QVZh!
04B5:0130  00 72 0A 2C 30 72 04 3C-0A 72 D3 41 4A 8A C7 0A  .r.,0r.<.rSAJ.G.
04B5:0140  C0 C3 9F 41 4A 9E F9 C3-E0 05 00 75 01 C3 EB F8  @C.AJ.yCh..u.Ckx
04B5:0150  0A C5 0A C1 75 01 C3 49-42 8B F2 AC 24 7F 0A C0  .E.Au.CIB.r,$..@
04B5:0160  F9 75 01 C3 3C 0A F9 75-01 C3 3C 0A F9 75 01 C3  yu.C<.yu.C<.yu.C
04B5:0170  3C 1A F9 75 01 C3 72 01-C3 F5 C3 A9 46 00 00 00  <.yu.Cr.CuC)F...
```

Fig. 3

We'll assume therefore that you have access to an IBM PC using PC-DOS or to another 8088-CPU based computer

Why the 8088? There are several reasons. First, because it's the microprocessor used in the IBM PC, a computer enjoying unprecedented sales growth, as well as work-a-like computers such as Compaq, Columbia Data Products and others. Second, if you want to learn about the new 16-bit technology, the 8088 chip in such computers is the ideal device to use.

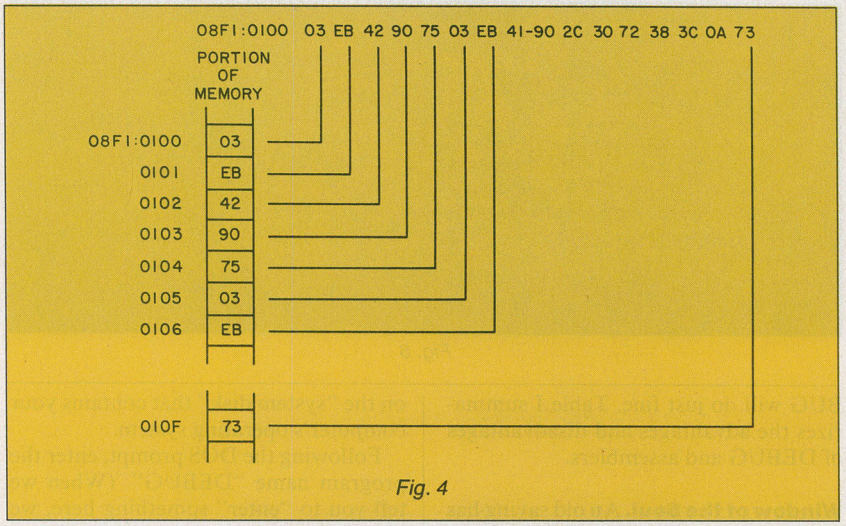


Fig. 4

BUG that do not contain the mini-assembler (we'll show you how), but it will be easier if you have it.

Three of the utility progams that come with PC-DOS are essential to following the lessons we're going to present. The first is DEBUG, which we've already mentioned. It's used to monitor, debug and edit assembly-language programs. Learning how to use it is vital to an understanding of assembly language.

The second utility is LINK, a program used to change an intermediate form of assembly language programs, call OBJ (for OBJect) files, into an executable program called an EXE file. (We'll explain these terms as we go along.)

The last is EXE2BIN, which converts EXE (EXEcutable) files to COM (COMmand) files. COM files are another, somewhat simpler, form of executable program.

Finally, once we start writing longer assembly-language programs, you'll need some sort of word processor to create what are known as the source files. These are text files, just like letters or other documents, but they contain the assembly-language instructions that will later be assembled for use by the microprocessor.

PC-DOS comes with a text editor called EDLIN (for EDit LINes). Though it's possible to use EDLIN to create assembly language source files, its limitations will become apparent as your programs become longer, and you'll probably want a good word processor anyway.

Assembly Language. Let's start by talking about assembly language in general—how it differs from higher-level languages such as BASIC—and about the operation of an assembler and how it differs from the interpreter or compiler used in higher-level languages.

If you're familiar with a higher-level language such as BASIC or Pascal, you know that there is a certain level of abstraction involved in program statements in these languages. A BASIC statement such as:

```
LET A=3
```

is operating on an abstract level. That is, we don't usually know, or *need* to know, where in the computer "A" is, or what changes are taking place in the computer when A is assigned the value of 3. This is because higher-level languages are oriented toward the handling of numbers with algebra-like formulas. Programmers using higher-level languages need that abstraction; they *want* to be insulated from what's really going on inside the computer so they can concentrate on the formulas.

In contrast, assembly language operates on a very concrete level. It deals with bits, with bytes, with *words* (two bytes side-by-side), with *registers*—which are physical places in the microprocessor where bytes and words are stored—and with memory locations, which have specific numerical addresses and specific physical locations in the memory chips inside the computer.

What Does an Assembler Do? If you've written programs in BASIC, you're familiar with the two-step process involved: First you write a group of BASIC program statements that make up a program; then later, when you execute the program, these statements are "interpreted" or changed into *machine-language* instructions that are executed by the microprocessor.

This process in BASIC is made to appear "invisible" to the user. The individual program lines are interpreted one at a time, and the resulting machine-language instructions for each line are executed by the microprocessor before the next line is interpreted. (Figure 1 shows how this works.)

In compiled languages such as Pascal, things are handled a little differently. The user first creates a *source file*, which is a text file of the entire program. This is then changed into machine-language instructions by a *compiler* program. (Actually, a utility called a *linker* is used too, but we'll ignore it for the moment.) In a compiled language the entire program is transformed into machine language all at once.

Assembly language resembles a compiled language more than it does an interpreted language such as BASIC. An assembly-language source file consisting of the text of the program is first created. This is then assembled into machine language by an *assembler* program. The assembler performs a process very similar to that of a compiler except that—as we'll see a little later—there is a far closer correspondence between an assembly-language instruction and a machine-language instruction than there is between a Pascal statement and the group of machine-language instructions that result from it.

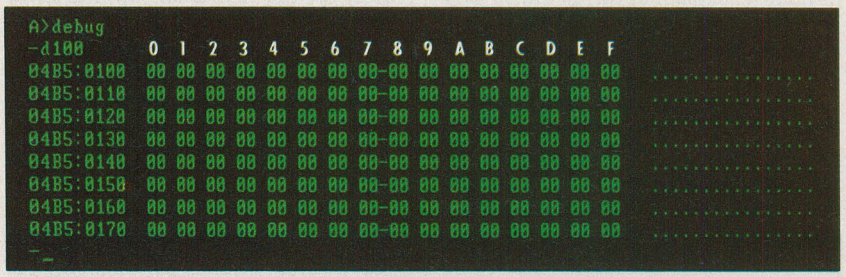


Fig. 5

What we've just described is the traditional way of transforming an assembly-language program into machine language instructions. To start with, however, we're going to use a different approach; we're going to use the mini-assembler in DEBUG. Using DEBUG, it's almost as easy to create and run short assembly-language programs as it is to create and run interpreted programs.

DEBUG vs. an Assembler. There are

several reasons why we've decided to start with DEBUG rather than with ASM or MASM. First, DEBUG is a much easier program to operate than the others. To type in and execute a program using DEBUG requires calling up only DEBUG itself, a simple process. Using an assembler, on the other hand, involves using a text editor, the assembler itself, a linker program called LINK, and often another program called EXE2BIN. Each of these programs requires a rather complex series of commands to make it work. We figured you'd have enough on your mind being introduced to a new computer language without having to learn how to operate all those other programs at the same time.

DEBUG's second advantage is that programs written with it require less "overhead" memory than those written with an assembler. This overhead comes in the form of program statements that must appear in the ASM source file, but are not necessary in DEBUG. (Don't worry if you don't understand what's meant by some of the terms we've used. Everything will be explained eventually.) The reason these additional statements are necessary in the assembler is difficult to explain at this point, so let's just say that by using DEBUG you avoid having to start your day with a lot of incomprehensible program lines.

Third, using DEBUG puts you in closer contact with what is *really* going on in your computer than using an assembler does. DEBUG has features that make it possible to get down to the most fundamental level of your computer's operation (short of opening up the cover and probing about with logic probes and oscilloscopes). Sooner or later, if you write programs in assembly language, you're going to have to understand this fundamental level and learn to use DEBUG; so now seems like a good time to start.

Of course, as we'll find out later, an assembler has all sorts of powerful features that make it indispensable for long programs, but for the moment, DEBUG will do just fine. Table I summarizes the advantages and disadvantages of DEBUG and assemblers.

```
A>debug
-f120 14f ff
-d100
04B5:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0120  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
04B5:0130  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
04B5:0140  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
04B5:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
-_
```

Fig. 6

Window of the Soul. An old saying has it that "the eyes are the windows of the soul."

We might say that DEBUG is the window of the 8088's soul. Besides being useful for assembling programs, DEBUG is also used to examine and modify the contents of memory locations; to load, store, and start programs; and to examine and modify registers. In other words, DEBUG is designed to put us in touch with various physical features of the PC-DOS or MS-DOS computer.

Before we write our landmark, first-ever 8088 assembly-language program, we're going to get to know our way around DEBUG; rev it up, so to speak, find out where the controls are, and taxi it out of the hangar and around the runway. Then we'll be ready for takeoff.

```
A>debug
-f100 117 61
-f178 17f 24
-d100
04B5:0100  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
04B5:0110  61 61 61 61 61 61 61 61-00 00 00 00 00 00 00 00  aaaaaaaa........
04B5:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
04B5:0170  00 00 00 00 00 00 00 00-24 24 24 24 24 24 24 24  ........$$$$$$$$
-_
```

Fig. 7

Getting DEBUG Rolling. All right, let's leap into the cockpit, get a firm grip on the keyboard, and get DEBUG rolling! We'll assume that you have a disk with DEBUG on it inserted into drive A, and that the "A>" prompt is waiting for your next move. (If you have a Winchester disk you'll have to make sure that DEBUG has been copied to it, and you'll also have to imagine a "C>" whenever you see "A>" here.) DEBUG is one of the programs provided on the "system disk" that contains your computer's operating system.

Following the DOS prompt, enter the program name "DEBUG". (When we tell you to "enter" something here, we mean to type the "something" and then press the ENTER key just to the left of the numeric keypad on the PC's keyboard.)

When DEBUG is loaded into the computer it will display its prompt, a single dash ("–"), which tells you that it's ready to listen to what you have to tell it.

The "D" Command. You can tell DEBUG what to do by typing in single-letter commands, usually followed immediately by one or more numbers. When we refer to these single-letter commands here we'll usually use upper-case letters, like "D," to make them stand out. However, when you type them in you can use either upper or lower case. For example, enter the letter "d" followed by the digits "1" "0" "0". You should see a display like the one shown in Fig. 2.

Wow! Look at all those numbers! What does it all mean? First of all, you may not see all zeros on your display as we show here. What the "D" command has done is to "dump" or display a portion of your computer's memory on the screen. Each pair of numbers represents one byte, or eight bits, of data stored in a particular memory location. If your

TABLE I—ADVANTAGES AND DISADVANTAGES

DEBUG
Easy to run
Low-overhead programs
Close to the machine
Not so versatile
Good on short programs

Assembler
Hard to run
More program overhead
Isolated from the machine
Very versatile
Good on long programs

computer's memory happened to have other data in it before you loaded DEBUG, it will appear here when you type "D", so you may well see all sorts of junky-looking numbers, like those in Fig. 3.

All the numbers there are in hexadecimal form. In fact, hexadecimal is the only numbering system DEBUG knows about, so if you aren't already acquainted with this way of representing numbers, now is the time to find out about it. A good discussion of numbering systems appeared in the December 1983 issue of COMPUTERS & ELECTRONICS.

Let's adopt this convention: Hexadecimal numbers—except for those in program listings or where the context makes clear what they are—will be followed by the small letter "h" to distinguish them from decimal numbers. Decimal numbers—again, unless the context makes it clear—will be followed by a small "d". Numbers from 0 to 9 are the same in both systems, so they don't really need a distinguishing letter, although they sometimes will have one for consistency. Of course, since DEBUG speaks *only* hexadecimal (hex for short), it doesn't use an "h" on its displays, and you should not put one after hexadecimal numbers you type in as DEBUG commands.

It requires two hexadecimal digits to represent an 8-bit byte of data. This two-digit hex number can range in value from 00h to FFh (0d to 255d). Thus, all the two-digit numbers in Fig. 3 fall into this range. There are 16 of these numbers on each line of the display. The dashes in the middle of the display are placed there for clarity, to separate the eight left-hand bytes on the line from the eight right-hand bytes.

Addresses. The numbers in the column at the left of Figs. 2 and 3 (like 08F1:0120) are the *memory addresses* of the bytes of data displayed. Each byte shown in the dumps occupies a specific address, as illustrated in Fig. 4.

The vertical column to the left represents an actual section of your computer's memory. Note how the byte in each memory location corresponds to a particular number in the DEBUG dump shown in Fig. 3.

Each address consists of two numbers separated by a colon. The number to the right of the colon (like 0100) is called the *offset address,* and for our purpose is the only portion of the address we'll be concerned with. The number to the left of the colon is called the *segment address,* and is used when addressing very high (above 64K) memory locations. Since our programs are not going to be anywhere near that large, we can safely ignore it.

Notice how each offset address (we'll just refer to them as addresses from now on, since we will not be using the segment portion) in the left-hand column of a DEBUG dump ends with a zero. If you're familiar with hexadecimal numbers you should understand why this is so. There are 16d, or 10h, bytes in each line, so when you've counted from 0h to to Fh, you're ready to increase the ten's (actually sixteen's) column by one, since 10 is the number that comes after F in hex. So we display 16d (10h) bytes, and then move down one line, increment the address by 10h, and display 10h more bytes. The display would be easier to read and understand if it had the 1's column values of the address printed across the top, as shown in Fig. 5, but it doesn't.

Anyway, it should be clear that the first byte on the top row is at memory

location 0100, the next is at 0101, the next at 0102, and so on. Similarly, the first byte on the second row is at 0110, the second at 0111, etc.

The "F" Command. Want to see the display change? An easy way to do this is to use DEBUG's "F" or FILL command. This command fills a part of memory with a particular hex number.

To use FILL, you enter "f" followed immediately by three numbers separated by spaces. The first of these numbers is the address where you want to *start* filling, the second is the address where you want to *stop* filling, and the third is the constant (from 00h to FFh) that you want the locations from the first address to the second to be filled with. Notice that while the data to be filled in consists of two-digit hex numbers (representing bytes), the addresses are four-digit hex numbers. Of course, you don't need to type leading zeros, so you can type fewer than four digits for the addresses when appropriate, as it is here. At the DEBUG prompt ("–") enter this:

```
f120 14f ff
```

The first "F" (remember, it can be upper or lower case) is the FILL command. It is followed by, respectively, the starting address of the fill, the ending address of the fill, and the value (FFh) that will be used. When you hit ENTER, nothing appears to happen. To see what's changed, you have to dump the same part of memory again. A dump of locations 100h through 170h is shown in Fig. 6.

Look at that! All the memory locations between 120h and 14Fh are now filled with FF, just as you specified with the "F" command. As for the rest, if you started out with numbers other than zero in memory, they'll still be there instead of the zeros shown in the dump.

ASCII Codes. You may be wondering about all the little dots and odd characters on the right-hand side of the dump display. These are the characters (like "A", "B", and so on) that the numbers to the left represent. The number that represents a particular character is called its *ASCII value.* (ASCII stands for "American Standard Code for Information Interchange.") As you probably know, the ASCII code is the normal way to represent characters in a computer's memory.

Since neither 00 nor FF represents a printable ASCII character, the positions in the ASCII display corresponding to those numbers are filled with dots, which indicate "no printable character." (If your computer's memory was filled with junk rather than all zeros to begin with, some of the numbers—20h through 7Eh—may have represented printable characters.) To see the character display change, along with the numbers, let's try filling in parts of memory with numbers that we know represent characters.

Enter the following three DEBUG commands (the "–" is, of course, DEBUG's prompt):

```
-f100 117 61
-f178 17f 24
-d100
```

You'll get the display shown in Fig. 7. The character corresponding to 61h is a small "a," and 24 hex is the code for the dollar sign, "$." We can see them both as numbers, and to the right as characters.

So far we've talked about how assembly language differs from higher-level languages, and also explained something about the operation of the DEBUG utility program. You may find it useful at this point to experiment a bit with DEBUG. Try filling in different constants to see how they look when you dump them. Examine different parts of memory. (You may be surprised at what you find!) You'll be using DEBUG a great deal as we go on, and you should feel comfortable with it.

Your First Program. Although you're now going to write your first assembly-language program, you don't know assembly language yet. There will be many aspects of the process that won't seem completely clear to you. Don't worry about this! Our approach is to show you first *what* something looks like, and afterwards explain *why* it looks that way and *how* it works.

By moving in this direction, from the concrete to the abstract, rather than the other way around, we hope to avoid the sort of academic theory-oriented descriptions that leave most readers confused, bored, and frustrated. Instead, you'll first get the *feeling* of the process (the roar of the motor and the rush of the wind in your hair, to return to our earlier flying analogy); later we'll explain what happened.

Different DOSs. There's a small problem we'd better deal with right away. This has to do with which version of DOS you're using. As noted previously, the DEBUG in DOS Version 2 (that is, Versions 2.0 and later) contains a built-in mini-assembler that will help in the creation of assembly-language programs. The DEBUG in DOS Version 1 (Versions 1.0 and 1.1) does not have this capability, so for those readers using this version we need to take a slightly different tack.

We'll handle the situation in the following way. We'll first explain how to type in a program if you're using DOS Version 1. Even if you have Version 2, you should read this part, try it out, and understand it. There are two good reasons for this. The first is that you will be introduced to a new DEBUG com-

```
-e100     <---you enter this
04B5:0100 61._
  :     \     /:\
  :      \     :_________the cursor will wait here for you to
segment   \               type in a two digit number
address    \
doesn't     offset address is the same as what
matter      you typed in
```

Fig. 8

```
-E100
04B5:0100 61.b2  61.1  61.b4  61.2  61.cd  61.21  61.cd  61.20
                :     :      :     :      :      :      :      :
-  <---DEBUG    \_____________________________________/       :
      prompt     press space bar following these numbers       press ENTER
                                                               following
                                                               last number
```

Fig. 9

```
-a100
08F1:0100 mov dl,1    \
08F1:0102 mov ah,2     \____ enter these lines
08F1:0104 int 21       /
08F1:0106 int 20      /
08F1:0108        <------simply press ENTER here
-_ <---------------now you're back in DEBUG
```

Fig. 10

```
-d100
04B5:0100  B2 01 B4 02 CD 21 CD 20-61 61 61 61 61 61 61 61   2.4.M!M aaaaaaaa
04B5:0110  61 61 61 61 61 61 61 61-00 00 00 00 00 00 00 00   aaaaaaaa........
04B5:0120  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
04B5:0130  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
04B5:0140  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
04B5:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
04B5:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
04B5:0170  00 00 00 00 00 00 00 00-24 24 24 24 24 24 24 24   ........$$$$$$$$
```

Fig. 11

mand: the "E" (for Enter) command. The second is that, after you've typed in the program using "E," you'll be better able to appreciate how lucky you are to have DOS Version 2, with its advanced version of DEBUG and its mini-assembler.

Using the "E" Command. Now we'll create an assembly-language program using DEBUG's "E" command. (The term "assembly language" is actually not quite right in this particular instance, as we'll see later on, but that needn't concern us now.) If you have DOS Version 1, this is the *only* way to use DEBUG to create a program. If you have Version 2, you should follow along anyway, typing in the commands.

The purpose of the "E" command is to enter a byte (or bytes) of data into memory. It's like the "F" command, except that you can enter a series of *different* bytes; they don't all have to have the same value, as they do with "F."

The series of bytes we're going to enter with "E" will constitute our program. To insert this program into memory, when you see the DEBUG prompt, you enter the command "E," followed immediately by the address where you want the program to go. In our case, we're going to put it at location 100h, so we enter "e" followed by "100." The program will respond by printing out the address, followed by its current contents, as shown in Fig. 8.

As shown there the content of location 100 happens already to be 61h, since that's what we put there with the "F" command earlier. However, it doesn't really matter what was there before: The important thing is that we're going to put it there now.

To "enter" a two-digit hex number into this location we type the number followed by—*not* the ENTER key—the *space bar.* The space bar has the effect of entering a number into one memory location and then advancing to the next one. The ENTER key, on the other hand, enters the number and then terminates the entire "E" command and returns us to the DEBUG prompt. After you have entered a number and pressed the space bar, the command will then print out the old contents of the next location and wait for you to type in the new contents.

The series of hexadecimal numbers we want to type in is the following:

```
B2
 1
B4
 2
CD
21
CD
20
```

These are the numbers that constitute our program. Type each number, press the space bar, type the next number, and so on. After you've typed in all eight numbers, the screen should look like Fig. 9 (minus our comments, of course).

Note that if you don't type any number at all before hitting the space bar, the byte in that location will remain unchanged, as you will discover if you make a typing mistake.

After you type the last number, press ENTER to tell DEBUG you're through. This should cause the DEBUG prompt ("–") to reappear.

If you make a mistake at any time, just press ENTER to get back to the DEBUG prompt and start over.

You've now placed your program in the computer's memory, from location 100 to location 107, using the "E" command. We'll explain how to execute, or "run," the program in a moment.

Using the "A" Command. Here's where you DOS Version I users become, briefly, just spectators. You should read through this section so you know what you're missing and, more important, because future program descriptions are going to be based on the "A" command approach outlined here. You'll need to know both approaches so you can use "E" even though we're talking about "A"; that is, translate our descriptions of the "A" approach into operations with "E." This won't be as difficult as it probably seems, so read on. (Or, better yet, hurry out and buy a copy of DOS Version 2.)

The "A" command accomplishes the same thing as the "E" command. Therefore, it is putting the bytes that constitute our program into memory—but it does so in a different way.

When we use the "A" command we don't insert hexadecimal bytes into memory directly. Instead, we type in a series of *mnemonic* symbols. ("Mnemonic" simply means "easy to remember." The idea is that these symbols are supposed to be easier to remember than the hexadecimal numbers they represent.) These codes are two- or three-character names that stand for certain assembly-language *instructions.* The instruction tells the microprocessor *what operation is to be done.* The instruction mnemonic is usually followed by a space and then by some letters and numbers that indicate what the operation is to be done *to.*

Expressed in mnemonic symbols, our program looks like this:

```
mov dl,1
mov ah,2
int 21
int 20
```

```
-u100,106 <---- you enter this

08F1:0100 B201          MOV     DL,01   \
08F1:0102 B402          MOV     AH,02    ;____ the program prints
08F1:0104 CD21          INT     21       ;     out all this
08F1:0106 CD20          INT     20      /
-_
```

Fig. 12

It looks short, but absolutely incomprehensible, doesn't it? That's all right, it won't be long before you can churn out this kind of thing in your sleep. We're going to type in this program, then dissect it a little and see if we can get a feel for how the "A" approach differs from the "E" approach, and for what assembly language is all about.

Enter the letter "A," followed by the address where you want the program to start. Here's a rule you should remember: *Programs written in DEBUG should always start at 100h.* The reasons for this will become clear later when we talk about the difference between COM files and EXE files.

When you enter "A" followed by an

address, DEBUG will automatically echo the address:

```
-a100
08f1:0100 -
```

DEBUG will then sit there waiting for you to type in the mnemonic codes for your program.

On the first line, enter "mov d1,1". That's "mov" as in the first three letters of "move," followed by a space, which is important, then "dl,1". with the *letter* "1", followed immediately by a comma and the *number* "1." Don't confuse letters and numbers. The screen should now look like this:

```
08F1:0100 mov dl,1
08F1:0102_
```

You've just typed your first line of assembly language!

The assembler is waiting for line two. Enter "mov ah,2". Then the third line, "int 21" and the fourth, "int 20".

After you've finished these four lines, you're done. So when the program says:

```
08f1:0108
```

you simply press ENTER to let it know you're through assembling this program and want to get back to DEBUG's prompt. Your screen should then look like Fig. 10.

So you now have two different ways to enter your program, depending on which version of DOS you have. In either case the program itself should be sitting in memory, waiting to be executed. There's a lot to say about the relationship between these two approaches to putting a program into memory, but if you're a real red-blooded programmer you can't wait to run the program. Let's do that first, and talk later.

Running the Program. What does the program do? Does it balance your checkbook? Calculate accounts receivable? We're afraid it's not as ambitious as that. Let's see what happens when we run it. To execute the program we use the "G" (for "Go") command. Simply enter the letter "g." It's not followed by any numbers (at this time). This will cause the program to be executed, just as entering "RUN" does in BASIC, and you'll see a little happy face on the screen smiling at you. Isn't that the cutest thing you ever saw? (No? Well, what did you expect from an 8-byte program?)

If you didn't get a happy face, you probably made a mistake typing in the program. It's easy to mistype something, what with all the numbers and unfamiliar symbols. Start with the "E" or "A" command and try again.

Unfortunately, mistakes in assembly-language programs can have more serious consequences than those in higher-level languages like BASIC. In higher-level languages the interpreter or compiler usually protects the operating system from the consequences of errors in programming by keeping your machine running and displaying something like, "Error in line 2034."

In assembly language, however, there is no such protection. Assembly language is the most fundamental level of the machine. There is nothing on a "supervisory level" overseeing the assembly-language program, as the interpreter or compiler does in higher-level languages. So if you make a mistake in assembly language it is woefully easy to "crash" your operating system—that is, alter parts of it in memory so that it no longer works and you need to reset the entire computer, either by hitting the ALT, CTRL, and DEL keys simultaneously or, in even worse cases, by turning the entire computer off and then on again. But all this is academic. You're never going to make a programming error! Are you?

Figure 11 shows the results of dumping locations 100 through 170 using the "D" command. And there, in the first eight locations, from 100 to 107, is our program. You can see this by comparing the numbers on the display with those typed in using "E." Our program has overlayed the 61s that were there before.

The symbols at the right—2.4.M!M—are meaningless. They just happen to be the ASCII equivalents of the numbers that make up the program.

What Assemblers Really Do. If you typed in the program using "E," you probably aren't too surprised to see these numbers reappear when you look in memory with "D." After all, you entered the numbers into memory, and there they are, just where you put them.

But how did they get there if you used the "A" command? You typed in mnemonic instructions and, lo and behold, there are the numbers sitting in memory. What's happened here is what this series is all about: The "A" command *assembled* the mnemomic instructions into hex numbers. This is the function of an assembler; both of DEBUG's mini-assembler that you invoke with the "A" command and of its large-scale relatives ASM and MASM. We'll have more to say about this later. First let's look at our program from another perspective.

The "U" Command. There's a more elegant and useful way to look at our program than by using "D" as we did above: the "U" command. We've assembled our program with "A" or typed in the pre-assembled hex numbers with "E." Now let's use the "U" command to *unassemble* it. "U" is the opposite of the "A" command. Where "A" takes us from symbolic mnemonic codes to the hex digits of machine language. "U" takes us from hex digits back to mnemonic codes. (Actually, the usual word for "unassemble" is "disassemble.")

To "unassemble" your program, enter "U," followed by the address where you want to start disassembling, then a comma, and then the address where you want to *stop* disassembling, as you see it in Fig. 12.

"U" shows the program in *both* hex codes and mnemonic instructions, all nicely arranged for you to admire! Thus the new number B201 is the machine-language equivalent of the assembly language statement "MOV DL,01," and so on for the other instructions. As before, the numbers on the left, such as "08F1:0100' " are the addresses of the locations occupied by the program. There's an address printed for each instruction, and since each of the instructions happens to occupy just two bytes, the addresses are all even numbered: 100, 102, 104, and 106.

Machine Language and Assembly Language. The hex numbers on the left in the "U" listing in Fig. 12 are what is called *machine language.* These numbers occupy specific memory locations, and the 8088 microprocessor looks in these locations, takes the numbers out of them, figures out what they mean, and executes them. These numbers are called "machine language" because it's the *machine*—the microprocessor—that understands them and operates on them. As far as we humans go, such numbers are hard to understand and virtually impossible to remember. For a human to decipher a program written entirely in hex numbers requires the most masochistic form of mental discipline, while the microprocessor chip, no larger than a fingernail, handles it easily. It is perhaps better not to dwell on the philosophical implications of this.

Take heart, however. The mnemonic instructions in the column on the right in the listing are *not* comprehensible to the microprocessor, clever though it may be. They form what is properly called "assembly language." And while you may not understand them now, you will when you finish reading this material.

If you want to pursue learning assembly language with the IBM PC, additional articles that build on this one can be read in upcoming issues of our sister publication, *PC* magazine. ◇