Hi!

Welcome to "The PC Assembler Helper" and "The PC Assembler Tutor". Both the program and the tutorial are designed to help those who are just starting to learn assembler language as well as those who know some assembler instructions but want to have a firmer grasp of the complete instruction set for the 8086.

There are two significant problems to learning assembler language. First, it is difficult to do either input or output at the assembler level. Imagine trying to learn BASIC if you were not allowed the following two instructions:

```
PRINT   RESULT
INPUT   NEW.DATA
```

Without PRINT and INPUT, you might be able to write a program but you would not be able to see the results. You would never be sure that the results were what you wanted. Also, you would not be able to vary the data. It would have to be coded into the program.

"The PC Assembler Helper" has taken care of this problem. It provides input and output of all standard 8086/8087 integral data types. These include 1 byte, 2 byte, 4 byte and 8 byte signed and unsigned numbers along with 1 byte and 2 byte hex, ASCII and binary data. Lastly, there is i/o for 10 byte BCD numbers. The interface has been designed so that beginners can use it with a minimum of trouble.

The second major problem is that most assembler books regard the 8086 as a black box. There is no way of seeing the workings of the chip itself. What exactly happens when you add two numbers? What about multiplication?

Once again, "The PC Assembler Helper" has come up with the solution. It allows you to view all the registers and flags at will. These registers can be independently formatted. If one register holds ASCII data while another has binary information and yet a third has a signed number, then each register can be set to display the appropriate type of data. This, you will find, is invaluable.

A third, though less important problem, is assembler overhead. There is a certain structure that must be followed to get the program to assemble and run correctly. This has been provided in template files. All you need to do is copy the appropriate template file and put the code in a predefined location to get your program to run correctly. For simple programs this can cut your work in half. It also minimizes the number of typos.

"The PC Assembler Tutor" is built around the Helper. It systematically goes through the 8086 instruction set, having you write small programs to illustrate how each instruction works. At

the end you should have a feeling for all the instructions except a few which involve the 8087 or peripheral hardware. These will be mentioned, but not used.

You will be a better programmer if you know how all the instructions work. There are times when one specific instruction is just what you want. If you don't know that it exists or how it works, you won't use it. This way, if you run across a situation where you think that a certain instruction might be useful to you, you can go back to the Tutor to refresh your memory and be able to put the instruction to use almost immediately.

If you are a beginner, I feel confident that you will learn faster and more thoroughly than with any other method. If you know some assembler but would like to know more, I'm sure that there is lots that would interest you.

```
AAD
SBB
XLAT
REPNE
SCAS
```

Do you know what these are? What about segment overrides? Do you know when to use them and when to avoid them? Do you know ALL the allowable addressing modes? What actually is an ASSUME statement?

In order to let you see if you find the material interesting, you may go through chapters 0 - 4 chapters without any obligation. You may also make an archival copy of the disks (you are urged to do so). If you continue after the fourth chapter then please register by sending $9.95 (or $10.60 for Californians) to Nelsoft. The registration form is at the end of this introduction.

If I followed the pricing structure of other people I would be charging several times as much. My goal is different. I want everyone who can benefit from the program and tutorial to use them, and I want everyone who uses them to do so legally. Therefore, I have priced them so that everyone can pay for them without any inconvenience. If you use them, you can certainly afford my minimal price.

The material is sequential. Chapter 0 should be read before starting on the other chapters and the chapters should be read in order. Appendix 1 contains all the subroutine calls in "The Assembler Helper" and how to access them. Appendix 2 is an alphabetical list of all the 8086 instructions, telling what they do and showing all allowable syntaxes. Appendix 3 gives the speed of all instructions along with a list of which flags are affected (you will learn what this means in the Tutor).

It is to your benefit to start at the beginning and work your way through. Chapter 0 contains material that you need to know, so you must read it. The text has been broken up into sections so that no printout is longer than 10 pages or so. All text files have a file extension .DOC. If a chapter is much longer than

that, it will be divided into parts, indicated by -1, -2, -3 after the chapter number. These files should be printable with the DOS 'print' command. The only imbedded printer command code is form feed for the next page. The text runs about 2500 characters a page, so you can estimate the size of the printout from the size of the text file.

Curly brackets in the text denote a footnote.{1} Some of the footnotes are technical and will be understood by only a quarter of the people. If you are one of that quarter, fine. If not, the important thing is not that you understand the outline of the proof, but that you believe that what is being proved is true.

The assembler level is for those who have some degree of intellegence. You have an unparalleled opportunity to screw things up at this level. If you got Cs and Ds in high school algebra because you didn't quite understand what was going on, then you probably shouldn't do assembler programming.{2}

In addition, I assume that you have done a lot of programming, preferably in either Pascal or C. BASIC is a nice language, but it is missing a certain type of structure which is vital for creating robust code in assembler language. If BASIC is all you know, I would recommend that you learn C first and then come back to assembler. You will be a better programmer for it.{3}

Finally, "The Assembler Helper" assumes that it has control of the screen. If you are hooked up to a debugger, there may be a conflict. There is a subroutine in the Helper called "set_timer" which may help minimize this conflict. You need to be in chapter 5 or so before you will be able to use it. See \APPENDIX\APP1.DOC for details.

If you are ready to go, please look at the following two pages and then read INTRO2.DOC. It will explain a little about what an assembler is. I hope you enjoy using the Helper and the Tutor as much as I enjoyed writing them.

Chuck Nelson

---

1. Like this one.

2. If you got Cs and Ds because you were too busy reading "Tales from the Crypt" and Isaac Asimov, that's something entirely different.

3. On re-reading this I decided that it is true, but pretensious. If you like BASIC and program well in BASIC, then you should learn assembler and continue using BASIC. There are certain inherent difficulties with BASIC, so before you start you should read BAS1.DOC. This is on DISK2.

TRADEMARK ACKNOWLEDGEMENT

The PC Assembler Helper was designed as a learning tool. It is meant to be used in conjunction with simple assembler programs to display the results of individual assembler instructions. It should not be used with high-level languages nor with programs that modify the screen.

HELPMEM.COM, the memory resident version, uses the same interrupts as a debugger. Therefore, if there is a debugger attatched to any program that is being used, HELPMEM.COM should not be loaded into memory.

WARRANTY

****

REGISTRATION

Hey, Chuck, I'm no chump!

I'm using your programs/manual, and I want to pay my fair share. Please make me a registered user of "The PC Assembler Tutor" and "The PC Assembler Helper". Enclosed is a check for $9.95 (plus 6.5% tax or $10.60 for California residents). Say, that's cheaper than a large pizza!

Name_____
Last First Initial

Address_____
Street Address

_____
City, State, and Zip Code

I got my copy from _____

Make checks payable to NELSOFT and send your registration to:

NELSOFT
P.O. Box 21389
Oakland, CA 94620

++++++++++

REGISTRATION BENEFITS

As a registered user of "The PC Assembler Helper" and "The PC Assembler Tutor" you are entitled to:

1) Use asmhelp.obj and helpmem.com for personal use.
2) Make 1 (one) printer copy of "The PC Assembler Tutor".
3) Use all programs in "The PC Assembler Tutor" for personal use.
4) Make an archival copy of the disks.
5) Distribute UNALTERED disks to friends for their perusal.
6) Use any updates to either "The PC Assembler Helper" or "The PC Assembler Tutor" under the same registration conditions.

Though copies of the disk may be given away if there is no charge, it is illegal to charge for redistribution of the disk or its contents without permission of the author. Under no circumstances may you distribute printed copies of "The PC Assembler Tutor". If you intend to charge for distributing the disk or its information, please read and sign the following distribution agreement.

****

DISTRIBUTION LICENSING AGREEMENT FOR THE PC ASSEMBLER HELPER AND THE PC ASSEMBLER TUTOR

Anyone wishing to charge people a fee for giving them a copy of The PC Assembler Helper and/or The PC Assembler Tutor must have the written authorization of the author, without which the distributor is guilty of copyright violation. To receive such authorization, send this completed application, along with a copy of your software library's order form to:

NELSOFT
P.O. Box 21389
Oakland, CA 94620

If you want a distribution disk with the latest copy of these programs, please include $7.00 to cover the cost of the disks, mailing and handling. (This offer is for bona fide user groups and shareware distributors only).

NAME OF ORGANIZATION _____

YOUR NAME _____

ADDRESS _____

CITY, STATE _____

TERMS OF DISTRIBUTION

1. The fee charged for each disk may not exceed $7.00. On high-density disks, the fee may not be over $10.00.

2. Your library's catalog or listing must state that this material is not free, but is copyrighted material that is provided to allow the user to evaluate it before paying.

3. The offering and sale of disks containing The PC Assembler Helper and The PC Assembler Tutor will be stopped at any time the author so requests.

4. The Tutor and the Helper must be distributed together. The compressed files and the information document must remain in the subdirectory \PCTUTOR. There may be no additional files in this subdirectory. Both the name and the contents of \PCREADME.DOC must remain unaltered.

5. Problems or complaints will be reported to the author.

In return for the right to charge a fee for the distribution of The PC Assembler Helper and The PC Assembler Tutor, I agree to comply with the above terms of distribution.

Signed,

_____ _____
Your Signature Date

WHAT'S AN ASSEMBLER?

What is the difference between a compiler and an assembler?

A compiler is a program that takes the source code you have written and turns it into machine language instructions that are usable by the computer. A machine language instruction is a binary number that tells the computer to do one specific thing. This is something very specific like: add 1 to a specific variable. The compiler does this in two steps. First, the compiler takes each line of source code and turns the expression on the line into a number of simple tasks to accomplish what is desired, generating a number of assembler TEXT instructions and data definitions. When the compiler is through with the source code, it then takes these TEXT instructions and assembles them to form an object module. An object module is a file that can be linked with other files to form a larger program.

Why two steps instead of one? There are several reasons. This allows a compiler writer to write a first part for any language like Pascal, C, BASIC, etc., and then use the same assembler part. This saves development time in a company that has more than one type of compiler. You can insert a new assembler part without worrying about the text generation part, or you can insert a new text generation part without worrying about the assembler part. Secondly, though an assembler is not a simple program, the compiler's text generator is even more complicated. Putting the two together is like trying to juggle eight balls instead of four balls.

This leads to the most vital reason. Not only would a unified text-generator/assembler be more error prone, it would be almost impossible to debug. If you are getting an error due to one type of Pascal instruction, is this because it is being misunderstood by the compiler or because the compiler is giving it the wrong machine codes? In the two part system, the compiler writer can look at the intermediate text code and isolate the problem into one of the two halves.

An assembler is a program that takes a TEXT file where each line corresponds to a specific machine instruction or type of data, calculates the address in memory where each piece of data or machine instruction will be, translates each instruction and piece of data into machine readable form, and inserts the addresses of data and labels into machine instructions where appropriate.{1}

---

1. A label is just a name which marks a certain spot in the assembler code.

The text name for a machine instruction is called a mnemonic.{2} It indicates what is being done by the instruction. Which would you rather use for multiplication, 'MUL' or '11110111xx100xxx'? These 'x's indicate a digit whose value depends on where something is in memory. For each mnemonic there is a single machine instruction which performs the operation.

This means that the assembler's task is relatively simple. It only needs to allocate space for all the variables and instructions, to translate each mnemonic and data value into its corresponding machine code, and finally put it into a machine usable file.

Here you need to know what different forms of file there are.

1) An executable (.EXE) file contains certain information for the operating system when the program is started. This allows the program to be as large as is wanted.

2) A .COM file contains no information for the operating system. When the operating system starts a .COM file it simply puts it in memory and starts it. Files with a .COM extension are limited to a length of 64k bytes.

3) Binary files are files which must be loaded into a .COM or .EXE program before being run. They cannot be used by themselves. They are archaic. They are a crutch for those compilers which don't support .OBJ files, and are disappearing.

4) An object (.OBJ) file is a section of a program. It contains code and variables, but also contains information that can be used to combine it with other object files into a larger program. A linker can convert one or more object files into an executable file.

Things have moved along in the past few years. TurboPascal 3.0 generated .COM files. This wasn't because .COM files were superior but because it was too difficult to generate the extra information needed to produce an .OBJ file. Interpreted BASIC required binary files because it did not have the ability to use .OBJ files. The situation now is:

If you want to link with the current Turbo Pascal, you should use an .OBJ file. If you want to link with QuickC, you need an .OBJ file. If you want to combine an external subprogram with QuickBASIC, you need an .OBJ file. Get the picture?

No assembler makes .EXE files. If you have a single file that you want made onto a stand alone .EXE file, you first make an .OBJ file and then use that single file with LINK.

---

2. You don't pronounce that first 'm'.

When making a .COM file, the normal route is to make .OBJ files, link them together into an EXE file, and then convert them to a .COM file with a program called EXE2BIN. This allows you to divide the problem into a number of subproblems and put them all together at the end.

As you can see from the above, the job for the assembler is to take a text file and convert it into an .OBJ file. The three assemblers that you are most likely to have are MASM, A86, and TurboAssembler. They all produce object files. Since assemblers simply supply the machine code for each instruction, they will all produce the exact same code.{3}

This is one of the differences between assemblers and compilers. The text generation phase of the compiler requires creativity. It is the compiler writer's idea of how to solve a certain problem in a specific language. This generated text is copyrighted, and you need a license to distribute a program that includes this generated text. An assembler. on the other hand, is just a drudge. If you had a book with the machine codes and had enough time, you could produce the same file byte for byte that the assembler would produce for a .COM file. There is no creativity involved in the generated code, and there is no license involved in its distribution.

There is some difference in the speed of those three assemblers, but I'll have a comment about that after I give you the numbers. These numbers are from the time of hitting the ENTER key to the return of the DOS prompt ('>'). These are on a low speed machine so your numbers should be better, but won't be any worse.

	A86	TurboASM	MASM
one page of code	3.2	8.8	10.3
20 pages of code	7.3	12.7	20.4
60 pages of code	12.5	22.9	43.3

All these numbers are in seconds. A86 is the fastest. It loads faster because it itself is a .COM file, and it works faster. But even the slowest (MASM), is fast enough. How long does it take to write 60 pages of code? Probably a week or two. Writing assembler code is normally slower than writing code for a high-level language. Even the slowest finishes the assembly in well under a minute. Think of the time it would take to compile 60 pages of Pascal code.

In fact, the normal length of a file will be from 10 to 20 pages, so these are the numbers you need to think of. These will be the smaller .OBJ files which are linked together by the linker.

If you have one of these assemblers you don't need anything different. If you need to get one, you can use this information

---

3. Or functionally the same code. Sometimes there are two different instructions that do the same thing, just as 6+1, 5+2 and 4+3 all produce 7.

to help you in your selection. The following prices are as of mid-1990.

A86 is available through shareware. It costs $50.00 for the assembler alone, $80.00 with the debugger. Add another $10.00 if you want a printed manual.

Both MASM and Turbo Assembler come with assembler, debugger, a number of utility programs and several manuals. They both retail for $150.00, but even a quick glance at BYTE magazine will find you a place that is selling them for $105.00 - $110.00. They come bundled, so you cannot buy the assembler without the debugger (The Microsoft debugger is Codeview and the Borland debugger is Turbo Debugger).

Speaking of debuggers, you may be thinking, "Well, I have DEBUG, so why do I want another debugger?" DEBUG has been outdated for several years now. It has been supplanted by symbolic debuggers which associate code with specific lines in the original text file. You give the symbolic debugger the .EXE file along with the text files that produced it, and it shows you your source code as you go along. Here's a section of code we will meet later in the Tutor:

```
; - - - - - - - - - - - - - - - - - - - -
start: push  ds               ; set up for return
     sub   ax,ax
     push  ax
     mov   ax, DATASTUFF    ; load ds
     mov   ds,ax

outer_loop:
     lea  ax, multiplicand    ; load multiplicand
     call get_unsigned_8byte
     call print_unsigned_8byte
     call get_unsigned        ; unsigned word to multiplier
     mov  multiplier, ax

     lea  si, multiplicand    ; load pointers
     lea  bx, result
; - - - - - - - - - - - - - - - - - - - -
```

Don't worry about what these instructions do. You'll learn that later. Here is DEBUG's idea of what is going on:

```
******************** DEBUG SCREEN SHOT ************************
-r
AX=0000 BX=0000 CX=2749 DX=0000 SP=0A00 BP=0000 SI=0000 DI=0000
DS=0D7E ES=0D7E SS=0D8E CS=0E7F  NV UP DI PL NZ NA PO NC
0E7F:0000 1E            PUSH    DS
-u0E7F:0000 2A
0E7F:0000 1E            PUSH    DS
0E7F:0001 2BC0          SUB     AX,AX
0E7F:0003 50            PUSH    AX
0E7F:0004 B82E0E        MOV     AX,0E2E
0E7F:0007 8ED8          MOV     DS,AX
0E7F:0009 8D060800      LEA     AX,[0008]
```

```
0E7F:000D E80C14        CALL    141C
0E7F:0010 E84B11        CALL    115E
0E7F:0013 E8F505        CALL    060B
0E7F:0016 A31000        MOV     [0010],AX
0E7F:0019 8D360800      LEA     SI,[0008]
0E7F:001D 8D1E1200      LEA     BX,[0012]
********************** END DEBUG ******************************
```

Part of this is understandable, but a lot of it is confusing, and you have lost the concept of what you are trying to do. To see what a symbolic debugger does with the same code, here is the Turbo Debugger's idea of what is happening:

```
************************ TURBO SCREEN SHOT {4} ****************************
  File   View   Run   Breakpoints   Data   Window   Options             READY
.Module: debugtst  File: debugtst.asm 74.....................................1.
.                                                                             .
.  start: push  ds               ; set up for return      .Registers......3. .
.         sub   ax,ax                                     .  ax 5C94   .c=0. .
.         push  ax                                        .  bx 0000   .z=1. .
.                                                         .  cx 0000   .s=0. .
.         mov   ax, DATASTUFF    ; load ds                .  dx 0000   .o=0. .
.         mov   ds,ax                                     .  si 0000   .p=1. .
.                                                         .  di 0000   .a=0. .
.  outer_loop:                                            .  bp 0000   .i=1. .
.          lea     ax, multiplicand        ; load multipli.  sp 09FA   .d=0. .
.          call    get_unsigned_8byte                     .  ds 4AD6   .   . .
.          call    print_unsigned_8byte                   .  es 4A26   .   . .
.          call    get_unsigned            ; unsigned word.  ss 4A36   .   . .
.          mov     multiplier, ax                         .  cs 4B27   .   . .
.                                                         .  ip 0019   .   . .
.                                                         ..................  .
.          lea     si, multiplicand        ; load pointers                    .
..............................................................................
.Watches.....................................................................2.
.multiplier,d                 23700                                          .
.multiplicand                 qword 00000042E843515D                         .
..............................................................................
 F1-Help F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run
******************************************************************************
```

DEBUG really doesn't meet our needs. Of course, those of you who still use EDLIN to write 20 page documents should feel free to use DEBUG.

Modern language compilers have their own debuggers in their environments, so we only need a debugger for the assembler. Turbo Debugger and Code View do the job very well. D86 is better than DEBUG but it has some problems.

---

4. Turbo Debugger is (C) Copyright 1988-1989 Borland International.

If you actually want a debugger that will symbolically debug everything that supports symbolic debugging, you might want to take a look at the Turbo Debugger. It has more power than you are ever likely to need.

"The Assembler Helper", the program which comes with the Tutor, is NOT a debugger. Debuggers are designed to show you what is happening with your code; the Helper is designed to show you what is happening with the 8086. It's a fundamental difference in outlook.

If you want to use a debugger while you are doing this tutorial it is possible but the results are not guaranteed. Please see DEBUGGER.DOC in \COMMENTS for some information about the different debuggers and how to use ASMHELP with a debugger. You should not try to do this before you are in chapter 5 or 6.

You may have noticed that CHASM, an assembler distributed through shareware, was not listed. There is a reason for this. It can't produce .OBJ files, so it cannot produce the standard files for use with current compilers, including QuickBASIC. CHASM is also unusable with this tutorial because it cannot produce files to link with ASMHELP.OBJ, the i/o interface program. If you are getting a shareware assembler, get A86. It's a quality assembler.

This tutorial was originally written for those using MASM. In order to allow those using the Turbo Assembler and A86 to follow along, there is a document for each one in \COMMENTS which explains any differences between what is in the chapters (which use MASM as an example) and what the respective assemblers do. There aren't that many differences. The pathnames are \COMMENTS\TURBO.DOC and \COMMENTS\A86.DOC.

TABLE OF CONTENTS

++

ANCILLARY MATERIAL

INDEX

CHAPTER 0.1 - NUMBERS AND ARITHMETIC

You don't habitually use the base two system to balance your checkbook, so it would be counterproductive to teach you machine arithmetic on a base two system. What number systems have you had a lot of experience with? The base 10 system springs to mind. I'm going to show you what happens on a base 10 system so you will understand the structure of what happens with computer arithmetic.

BASE 10 MACHINE

Each place inside the microprocessor that can hold a number is called a REGISTER. Normally there are a dozen or so of these. Our base 10 machine has 4 digit registers. They can represent any number from 0000 to 9999. They are exactly like an industrial counters or the counters on your tape machines.{1} If you add 27 to a register, the microprocessor counts forward 27; if you subtract 153 from a register, the microprocessor counts backwards 153. Every time you add 1 to a register, it increments by 1 - that is 0245, 0246, 0247, 0248. Every time you subtract 1 from a register, it decrements by 1 - that is 3480, 3479, 3478, 3477.

Let's do some more incrementing. 9997, 9998, 9999, 0000, 0001, 0002. Whoops! That's a problem. When the register reaches 9999 and we add 1, it changes to 0000, not 10,000. How can we tell the difference between 0000 and 10,000? We can't without a little help from the CPU.{2} Immediately after an arithmetical operation, the CPU knows whether you have gone through 10,000 (9999->0000). The CPU has something called a carry flag. It is internal to the CPU and can have the value 0 or 1. After each arithmetical operation, the CPU sets the CARRY FLAG to 1 if you went through the 9999/0000 boundary, and sets the carry flag to 0 if you didn't.{3}

Here are some examples, showing addition, the result, and the carry flag. The carry flag is normally abbreviated by CF.

number 1	number 2	result	CF
0289	4782	5071	0
4398	2964	7382	0
8177	5826	4003	1

---

1. Exactly like industrial counters that have several hundred thousand parts, that is.

2. The CPU (central processing unit) is the chip(s) that does all the arithmetic. In the case of the PC, it is the 8086.

3. When you set a flag to 0, it is called CLEARING the flag.

6744	4208	0952	1

Note that you must check the carry flag immediately after the arithmetical operation. If you wait, the CPU will reset it after the next arithmetical operation.

Now let's do some decrementing. 0003, 0002, 0001, 0000, 9999, 9998. Golly gosh! Another problem. When we got to 0000, rather than getting -1, -2, we got 9999, 9998. Apparently 9999 stands for -1, 9998 stands for -2. Yes, that's the system on this, on the 8086, and on all computers. (Back to that in a moment.) How do we tell that the number went through 0 ; i.e. 0000->9999? The carry flag comes to the rescue again. If the number goes through the 9999/0000 boundary in either direction, the CPU sets the CF to 1; if it doesn't, the CPU sets the CF to 0. Here's some subtraction, with the result and the carry flag.

number 1	number 2	result	CF
8473	2752	5721	0
2836	4583	1747	1
0654	9281	8627	1
9281	0654	8627	0

Look at examples 3 and 4. The numbers are reversed. The results are the same but they have different signs. But that is as it should be. When you reverse the order in a subtraction, you get the same absolute value, only a different sign (15 - 7 = 8 but 7 - 15 = -8). Remember, the CF is reliable only immediately after the operation.

NEGATIVE NUMBERS

The negative numbers go 9999=-1, 9998=-2, 9997=-3, 9996=-4, 9995=-5 etc. A more negative number is denoted by a smaller number in the register; -5 = 10,000 -5 = 9995; -498 = 10,000 -498 = 9502, and in general, -x = 10,000 -x. Here are some negative numbers and their representations on our machine.

number	machine no	number	machine no
-27	9973	-4652	5348
-8916	1084	-6155	3845

As you will notice, these numbers look exactly the same as the unsigned numbers. They ARE exactly the same as the unsigned numbers. The machine has no way of knowing whether a number in a register is signed or unsigned. Unlike BASIC or PASCAL which will complain whenever you try to use a number in an incorrect way, the machine will let you do it. This is the power and the curse of machine language. You are in complete control. It is your responsibility to keep track of whether a number is signed or unsigned.

Which signed numbers should be positive and which negative? This has already been decided for you by the computer, but let's think

out what a reasonable solution might be. We could have from 0000 to 8000 positive and from 9999 to 8001 negative, but that would give us 8001 positive numbers and 1999 negative numbers. That seems unbalanced. More importantly, if we take -(3279) the machine will give us 6721, which is a POSITIVE number. We don't want that. For reasons of symmetry, the positive numbers are 0000-4999 and the negative numbers are 9999-5000.{4} Our most negative number is -5000 = 10,000 -5000 = 5000.

10'S COMPLEMENT

It's time for a digression. If we are going to be using negative numbers like -(473), changing from an external number to an internal number is going to be a bother: i.e. -473 -> 9527. Going the other way is going to be a pain too: i.e. 9527 -> -473. Well, it would be a problem except that we have some help.

```
    0000 =    10,000    =     9999     +1
                          - 473
result                     9526     +1   = 9527
```

Let's work this through carefully. On our machine, 0000 and 10000 (9999+1) are the same thing, so 0 - 473 is the same as 9999+1-473 which is the same as 9999-473+1. But when we have all 9s, this is a cinch. We never have to borrow - all we have to do is subtract each digit from 9 and then add 1 to the total. We may have to carry at the end, but that is a lot better than all those borrows. We'll do a few examples:

```
(-4276)
    0000 =    10,000    =     9999     +1
                          -4276
result                     5723     +1   = 5724


(-3982)
    0000 =    10,000    =     9999     +1
                          -3982
result                     6017     +1   = 6018


(-2400)
    0000 =    10,000    =     9999     +1
                          -2400
result                     7599     +1   = 7600


(-1989)
    0000 =    10,000    =     9999     +1
```

4. That way, if we tell the machine that we are working with signed numbers, all it has to do is look at the left digit. If the digit is 5-9, we have a negative number, if it is 0-4, we have a positive number. Note that 0000 is considered to be positive. This is true on all computers.

```
                              -1989
result                         8010     +1   = 8011
```

This is called 10s complement. Subtract each digit from 9, then add 1 to the total. One thing we should check is whether we get the same number back if we negate the negative result; i.e. does -(-1989)) = 1989? From the last example, we see that -1989 = 8011, so:

```
(-8011)
    0000 =    10,000    =     9999     +1
                           -8011
result                      1988     +1   = 1989
```

It seems to work. In fact, it always works. See the footnote for the proof.{5} You are going to use this from time to time, so you might as well practice some. Here are 10 numbers to put into 10s complement form. The answers are in the footnote. (1) -628, (2) -4194, (3) -9983, (4) -1288, (5) -4058, (6) -6952, (7) -162, (8) -9, (9) -2744, (10) -5000.{6}

The computer keeps track of whether a number is positive or negative. After an arithmetical operation, it sets a flag to tell whether the result is positive or negative. This flag has no meaning if you are using unsigned numbers. The computer is saying, "If the last arithmetical operation was with signed numbers, then this is the sign of the result." The flag is called the sign flag (SF). It is 0 if the number is positive and 1 if the number is negative. Let's decrement again and look at both the sign flag and carry flag.

NUMBER	SIGN	CARRY
3	0	0
2	0	0
1	0	0
0	0	0
9999	1	1

---

5. Let x be any number. Then:

```
-x     = ( 10,000 - x)     = ( 9999 + 1 - x ) ;

-(-x)  = ( 10,000 - (-x) ) = ( 9999 + 1 - (-x) )
                           = ( 9999 + 1 - ( 9999 + 1 - x ) )
                           = ( 9999 + 1 - 9999 - 1 + x )
                           = x
```

6. (1) -628 = 9372 , (2) -4194 = 5806 , (3) -9983 = 0017, (4) -1288 = 8712 , (5) -4058 = 5942 , (6) -6952 = 3048 (7) -162 = 9838 , (8) -9 = 9991 , (9) -2744 = 7256, (10) -5000 = 5000. This last one is a little strange. It changes 5000 into itself. In our system, 5000 is a negative number and it winds up as a negative number. This happens on all computers. If you take the maximum negative number and take its negative, you get the same number back.

9998	1	0
9997	1	0
9996	1	0

That worked pretty well. The sign flag changed from 0 to 1 when we went from 0 to 9999 and the carry flag was set to 1 for that one operation so we could see that we had gone through the 9999/0000 boundary.

Let's do some more decrementing.

NUMBER	SIGN	CARRY
5003	1	0
5002	1	0
5001	1	0
5000	1	0
4999	0	0
4998	0	0
4997	0	0
4996	0	0

This one didn't work too well. 5000 is our most negative number (-5000) and 4999 is our most positive number; when we crossed the 4999/5000 boundary, the sign changed but there was nothing to tell us that the sign had changed. We need to make another flag. This one is called the overflow flag. We check the carry flag (CF) for the 0000/9999 boundary and we check the overflow flag for the 5000/4999 boundary. The last decrementing example with the overflow flag:

NUMBER	SIGN	CARRY	OVERFLOW
5003	1	0	0
5002	1	0	0
5001	1	0	0
5000	1	0	0
4999	0	0	1
4998	0	0	0
4997	0	0	0
4996	0	0	0

This time we can find out that we have gone through the boundary. We'll come back to how the computer sets the overflow flag later, but let's do some addition and subtraction now.

UNSIGNED ADDITION AND SUBTRACTION

Unsigned addition is done the same way as normally. The computer adds the two numbers. If the result is over 9999, it sets the carry flag and drops the left digit (i.e. 14625 -> 4625, CF = 1, 19137 -> 9137 CF = 1, 10000 -> 0000 CF = 1). The largest possible addition is 9999 + 9999 = 19998. This still has a 1 in the left digit. If the carry flag is set after an addition, the result must be between 10000 and 19998.

Since this is unsigned addition, we won't worry about the sign flag or the overflow flag for the moment. Here are some examples of unsigned addition.

NUMBER 1	NUMBER 2	RESULT	CF
5147	2834	7981	0
6421	8888	5309	1
2910	6544	9454	0
6200	6321	2521	1

Directly after the addition, the computer has complete information about the number. If the carry flag is set, that means that there is an extra 10,000, so the result of the second example is 15309 and the result of the fourth example is 12521. There is no way to store all that information in 4 digits in memory so that extra information will be lost if it is not used immediately.

Subtraction is similar. The machine subtracts, and if the answer is below 0000, it sets the carry flag, borrows 10000 and adds it to the result. -3158 -> -3135 + 10000 -> 6842 CF = 1 ; -8197 -> -8197 + 10000 -> 1803 CF = 1. After a subtraction, if the carry flag is set, you know the number is 10000 too big. Once again, the carry flag information must be used immediately or it will be lost. Here are some examples:

NUMBER 1	NUMBER 2	RESULT	CF
3872	2655	1217	0
9826	5967	3859	0
4561	7143	7418	1
2341	4907	7434	1

If the carry flag is set, the computer borrowed 10000, so example 3 is 7418 - 10000 = -2582 and example 4 is 7434 - 10000 = -2566.

MODULAR ARITHMETIC

What the computer is doing is modular arithmetic. Modular arithmetic is like a clock. If it is 11 o'clock and you go forward 1 hour it's now 12 o'clock; if it's 11 and you go backwards 1 hour it's now 10. If it's 11 and you go forward 4 hours it's not 15, it's 3. If it's 11 and you go backward 15 hours it's not -4, it's 8.

The clock is doing mod 12 arithmetic.{7}

(A+B) mod 12
(A-B) mod 12

From the clock's viewpoint, 11 o'clock today, 11 o'clock yesterday and 11 o'clock, June 8, 1754 are all the same thing. If

7. To be a perfect analogy 12 o'clock should be 0 o'clock.

you go forward 200 hours (that's 12X16 + 8) you will have the same result as going forward 8 hours. If you go backwards 200 hours (that's -(12X16 + 8) = -(12X16) -8) you get the same result as going backwards 8 hours. If you go forward 4 hours from 11 (11+4) mod 12 = 3 you get the same result as going backwards 8 hours (11-8) mod 12 = 3. In fact, these come in pairs. If A + B = 12, then going forward A hours gives the same result as going backwards B hours. Forwards 9 = backwards 3; forwards 7 = backwards 5; forwards 11 = backwards 1.

In the mod 12 system, the following things are equivalent:

```
(+72 + 4)      (+72 - 8)
(+60 + 4)      (+60 - 8)
(+48 + 4)      (+48 - 8)
(+36 + 4)      (+36 - 8)
(+24 + 4)      (+24 - 8)
(+12 + 4)      (+12 - 8)
(  0 + 4)      (  0 - 8)
(-12 + 4)      (-12 - 8)
(-24 + 4)      (-24 - 8)
(-36 + 4)      (-36 - 8)
(-48 + 4)      (-48 - 8)
(-60 + 4)      (-60 - 8)
```

They form what is known as an equivalence class mod 12. If you use any one of them for addition or subtraction, you will get the same result (mod 12) as with any other one. Here's some addition:{8}

```
(+48 + 4) + 7 = (48 + 11) mod 12 = 11
(-48 - 8) + 7 = (48 - 1 ) mod 12 = 11
(  0 - 8) + 7 = ( 0 - 1 ) mod 12 = 11
(-60 + 4) + 7 = (-60 +11) mod 12 = 11
```

And some subtraction:

```
(+48 + 4) - 2 = (48 + 2 ) mod 12 = 2
(-48 - 8) - 2 = (48 - 10) mod 12 = 2
(  0 - 8) - 2 = ( 0 - 10) mod 12 = 2
(-60 + 4) - 2 = (-60 + 2) mod 12 = 2
```

Our pretend computer doesn't cycle every 12 numbers, it cycles every 10,000 numbers - it is a mod 10,000 machine. On our machine, the number 6453 has the following equivalence class:

```
(+30000 + 6453)          (+30000 - 3547)
(+20000 + 6453)          (+20000 - 3547)
(+10000 + 6453)          (+10000 - 3547)
(     0 + 6453)          (     0 - 3547)
(-10000 + 6453)          (-10000 - 3547)
(-20000 + 6453)          (-20000 - 3547)
(-30000 + 6453)          (-30000 - 3547)
```

---

8. (-10) mod 12 = 2 ; (-11) mod 12 = 1

Any one of these will act the same as any other one. Notice that 10000 - 3547 is the subtraction that we did to get the representation of -3547 on the machine.

```
-3547    = 9999 + 1
           3547
           6452 + 1 = 6453
```

6453 and -3547 act EXACTLY the same on this machine. What this means is that there is no difference in adding signed or unsigned numbers on the machine. The result will be correct if interpreted as an unsigned number; it will also be correct if interpreted as a signed number.

```
    6821 + 3179 = 10000  so  -3179 = 6821   and  3179 = -6821
    5429 + 4571 = 10000  so  -4571 = 5429   and  4571 = -5429
```

Since -3179 and 6821 act the same on our machine and since -4571 and 5429 act the same, let's do some addition. Take your time so you understand why the signed and unsigned numbers are giving the same results mod 10000:

```
     ------------------------------------------------------------
     6821 + 497 = 7318
    -3179 + 497 = (10000 - 3179) + 497 = 10000 -2682  = -2682

     7318 + 2682 = 10000      so    -2682 = 7318

     ------------------------------------------------------------

     5429 + 876 = 6305
    -4571 + 876 = (10000 - 4571) + 876 = 10000 - 3695 = -3695

     6305 + 3695 = 10000      so    -3695 = 6305

     ------------------------------------------------------------
```

Here's some subtraction:

```
     ------------------------------------------------------------

     6821 - 507 = 6314
    -3179 - 507 = (10000 - 3179) - 507 = 10000 - 3686 = -3686

     6314 + 3686 = 10000     so     -3686 = 6314

     ------------------------------------------------------------

     5429 - 178 = 5251
    -4571 - 178 = (10000 - 4571) - 178 = 10000 - 4749 = -4749

     5251 + 4749 = 10000    so      -4749 = 5251

     ------------------------------------------------------------
```

It is the same addition or subtraction. Interpreted one way it is

signed addition or subtraction; interpreted another way it is unsigned addition or subtraction.

The machine could have one operation for signed addition and another operation for unsigned addition, but this would be a waste of computer resources. These operations are exactly the same. This machine, like all computers, has only one integer addition operation and one integer subtraction operation. For each operation, it sets the flags of importance for both signed and unsigned arithmetic.

For unsigned addition and subtraction, CF, the carry flag tells whether the 0000/9999 boundary has been crossed.

For signed addition and subtraction, SF, the sign flag tells the sign of the result and OF, the overflow flag tells whether the result was too negative or too positive.

SIGN EXTENSION

Although our base 10 machine is set up for 4 digit numbers, it is possible to use it for numbers of any size by writing the appropriate software. We'll use 12 digit numbers as an example, though they could be of any length. The first problem is converting 4 digit numbers into 12 digit numbers. If the number is an unsigned number, this is no problem (we'll write the number in groups of 4 digits to keep it readable):

```
4816      ->   0000 0000 4816
9842      ->   0000 0000 9842
 127      ->   0000 0000 0127
```

what if it is a signed number? The first thing we need to know about signed numbers is, what is positive and what is negative? Once again, for reasons of symmetry, we choose positive to be 0000 0000 0000 to 4999 9999 9999 and negative to be 5000 0000 0000 to 9999 9999 9999.{9} This longer number system cycles from

9999 9999 9999 to 0000 0000 0000. Therefore, for longer numbers,
0000 0000 0000 = 1 0000 0000 0000. They are equivalent.
0000 0000 0000 = 9999 9999 9999 + 1.

If it is a positive signed number, it is still no problem (recall that in our 4 digit system, a positive number is between 0000 and 4999, a negative signed number is between 5000 and 9999). Here are some positive signed numbers and their conversions:

```
1974      ->   0000 0000 1974
   1      ->   0000 0000 0001
3909      ->   0000 0000 3909
```

---

9. Once again, the sign will be decided by the left hand digit. If it is 0-4 it is a positive number; if it is 5-9 it is a negative number.

If it is a negative number, where did its representation come from in our 4 digit system? -x -> 9999 + 1 -x = 9999 - x + 1. This time it won't be 9999 + 1 but 9999 9999 9999 + 1. Let's have some examples.

```
    4 DIGIT SYSTEM       12 DIGIT SYSTEM

-1964
     9999     + 1        9999 9999 9999 + 1
    -1964                         -1964
     8035   -> 8036      9999 9999 8035 + 1 -> 9999 9999 8036

-2867
     9999     + 1        9999 9999 9999 + 1
    -2867                         -2867
     7132   -> 7133      9999 9999 7132 + 1 -> 9999 9999 7133

-182
     9999     + 1        9999 9999 9999 + 1
     -182                          -182
     9817   -> 9818      9999 9999 9817 + 1 -> 9999 9999 9818
```

As you can see, all you need to do to sign extend a negative number is to put 9s to the left.

Can't those 9s on the left become 0s when we add that 1 at the end? No. In order for that to happen, the right four digits must be 9999. But that can only happen if the number to be negated is 0000:

```
     9999 9999 9999 + 1
              -0000
     9999 9999 9999 + 1 -> 0000 0000 0000
```

In all other cases, adding 1 does not carry anything out of the right four digits.

It is impossible to truncate one of these 12 digit numbers to a 4 digit number without making the results unreliable. Here are two examples:

```
(number)      0000 0168 7451 ->   7451  (now a negative number)
(actual value)     +168 7451     -2549

(number)      9999 9643 2170 ->   2170  (now a positive number)
(actual value)     -356 7830     +2170
```

We now have 12 digit numbers. Is it possible to add them and subtract them? Yes but only 4 digits at a time. When you add with pencil and paper you carry left from each digit. The computer can carry left from each group of 4 digits. We'll do the following addition:

```
              0138 6715 6037
            + 2514 2759 7784
```

Do this with pencil and paper and write down all the carries. The computer is going to do this in 3 parts:

1) 6037 + 7784
2) 6715 + 2759 + carry (if any)
3) 0138 + 2514 + carry (if any)

The first addition is our regular addition. It will set the carry flag if the 0000/9999 boundary was crossed (i.e. the result was larger than 9999). In our case CF = 1 since the result is 13821. The register holds 3821. We store 3821. Next, we need to add three things: 6715 + 2759 + CF (=1). There is an instruction like this on all computers. It adds two numbers plus the value of the carry flag. Our first addition was ADD (add two numbers). This time the machine instruction is ADC (add two numbers and the carry). The result of our second addition is 9475. The register holds 9475 and CF = 0. We store 9475. Finally, we need to add three more things: 0138 + 2514 + CF (=0). Once again we use ADC. The result is 2652, CF = 0. We store the 2652. That is the whole result:

2652 9475 3821

If CF = 1 at this point, the number has crossed the 9999,9999,9999/0000,0000,0000 boundary. This will work for signed numbers also. The only difference is that at the very end we don't check CF, we check OF to see if the 4999,9999,9999/5000,0000,0000 boundary has been crossed.

Just to give you one more example we'll do a subtraction using the same numbers:

0138 6715 6037
2514 2759 7784

Notice that in order for you to do this with pencil and paper you'll have to put the larger number on top before you subtract. With the machine this is unnecessary. Go ahead and do the subtraction with pencil and paper.

The machine can do this 4 digits at a time, so this is a three step process:

1) 6037 - 7784
2) 6715 - 2759 - borrow (if any)
3) 0138 - 2514 - borrow (if any)

The first one is a regular subtraction and since the bottom number is larger, the result is 8253, CF = 1. (Perhaps you are puzzled because that's not the result that you got. Don't worry, it all comes out in the wash). Step two subtracts but also subtracts any borrow (We had a borrow because CF = 1). There is a special instruction called SBB (subtract with borrow) that does just that. 6715 - 2759 - 1 = 3955, CF = 0. We store the 3955 and go on to the third part. This also is SBB, but since we had no

borrow, we have 0138 - 2514 - 0 = 7624, CF = 1. We store 7624. This is the end result, and since CF = 1, we have crossed the 9999,9999,9999/0000,0000,0000 boundary. This is going to be the representation of a negative number mod 1,0000,0000,0000. With pencil and paper, your result was:

```
-2375 6044 1747
```

The machine result was:

```
 7624 3955 8253
```

But CF was 1 at the end, so this represents a negative number. What number does it represent? Let's take its negative to get a positive number with the same absolute value:

```
9999 9999 9999  + 1
7624 3955 8253
2375 6044 1746  + 1  = 2375 6044 1747
```

This is the same thing you got with pencil and paper. The reason it looked wierd is that a negative number is always stored as its modular equivalent. If you want to read a negative number, you need to take its negative to get a positive number with the same absolute value.

If we had been working with signed numbers, we wouldn't have checked CF at the very end, we would have checked OF to see if the 4999,9999,9999/5000,0000,0000 boundary had been crossed. If OF = 1 at the end, then the result was either too negative or too positive.

OVERFLOW

How does the machine decide that overflow has occured? First, what exactly is overflow and when is it possible for overflow to occur?

Overflow is when the result of a signed addition or subtraction is either larger than the largest positive number or more negative than the most negative number. In the case of the 4 digit machine, larger than +4999 or more negative than -5000.

If one number is negative and the other is positive, it is not possible for overflow to occur. Take +32 and -4791 as examples. If we start with the positive number (+32) and add the negative number (-4791), the result can't possibly be too positive. Similarly, if we start with the negative number (-4791) and add the positive number (+32), the result can't be too negative. Therefore, the result can be neither too positive nor too negative. Make sure you understand this before going on.

What if both are positive? Then overflow is possible. Here are some examples:

```
    (+3500) + (+4500) = 8000 = -2000
    (+2872) + (+2872) = 5744 = -4256
    (+1799) + (+4157) = 5956 = -4044
```

In each case, two positive numbers give a negative result. How about two negative numbers?

```
                    (7154) + (6000) = 3154 = +3154
(actual value)      -2946    -4000

                    (5387) + (5826) = 1213 = +1213
(actual value)      -4613    -4174

                    (8053) + (6191) = 4244 = +4244
(actual value)      -1947    -3809
```

The numbers underneath are the negative numbers that the numbers above them represent. In these cases, adding two negative numbers gives a positive result.

This is what the machine checks for. Before the addition, it checks the signs of the numbers. If the signs are the same, then the result must also be the same sign or overflow has occurred.{10} Thus + and + must have a + result; - and - must have a - result. If not, OF (the overflow flag) is set (OF = 1). Otherwise OF is cleared (OF = 0).

MULTIPLICATION

Unsigned multiplication is easy. The machine simply multiplies the two numbers. Since the result can be up to 8 digits (the maximum result is 9999 X 9999 = 9998 0001) the machine uses two registers to hold the result. We'll call them R1 and R2.

```
    5436 X 174      R1   0094
                    R2   5864

    2641 X 2003     R1   0528
                    R2   9923
```

You need to know which register holds which half of the result, but besides that, everything is straightforward. On this machine R1 holds the left four digits and R2 holds the right four digits.

Notice that our machine has changed the modular base from N to N*N (from 1 0000 to 1 0000 0000). What this means is that two things which are modularly equivalent under addition and subtraction are not necessarily equivalent under multiplication and division. 6281 and -3719 will not work the same.

---

10. The machine checks something considerably more obscure because it is easier to implement in semiconductor logic, but what it is actually doing is checking to see if the two numbers being added have the same sign. If they do, the result must be the same sign or overflow has occurred.

The machine can't do signed multiplication. What it actually does is convert the numbers to positive numbers (if necessary), perform unsigned multiplication, and then do sign adjustment of the results (if necessary). It uses 2 registers for the result.

	SIGNED MULTIPLICATION	REGS	RESULT
(number)	(5372) X (3195)	R1 8521	= -1478 6460
(actual value)	-4628 X +3195	R2 3540	
(number)	(9164) X (8746)	R1 0104	= +104 8344
(actual value)	-836 X -1254	R2 8344	
(number)	(9927) X (0013)	R1 9999	= -949
(actual value)	-73 X +13	R2 9051	

Looking at the last example, if we performed unsigned multiplication on those two numbers, we would have 9927 X 0013 = 0012 9051, a completely different answer from the one we got. Therefore, whenever you do multiplication, you have to tell the machine whether you want unsigned or signed multiplication.

DIVISION

Unsigned division is easy too. The machine divides one number by the other, puts the quotient in one register and the remainder in another. Once again, the only problem is remembering which register has the quotient and which register has the remainder. For us, the quotient is R1 and the remainder is R2.

6190 / 372	R1 0016	16	remainder 238
	R2 0238		
9845 / 11	R1 0895	895	remainder 0
	R2 0000		

As with multiplication, signed division is handled by the machine changing all numbers to positive numbers, performing unsigned division, then putting back the appropriate signs.

	SIGNED DIVISION	REGS	RESULT
(number)	(7192) / (9164)	R1 0003	+3 rem. -300
(actual value)	-2808 / -836	R2 9700	
(number)	(3753) / (9115)	R1 9996	-4 rem. +213
(actual value)	+3753 / -885	R2 0213	

Looking at the last example, 3753 / 9115, if that were unsigned multiplication the answer would be 0 remainder 3753, a completely different answer from the signed division. Every time you do a division, you have to state whether you want unsigned or signed division.

CHAPTER 0.2 - BASES 2 AND 16

I'm making the assumption that if you are along for the ride you already know something about binary and hex numbers. This is a review only.

BASE 2 AND BASE 16

Base 2 (binary) allows only 0s and 1s. Base 16 (hexadecimal) allows 0 - 9, and then makes up the next six numbers by using the letters A - F. A = 10, B=11, C=12, D=13, E=14 and F=15. You can directly translate a hex number to a binary number and a binary number to a hex number. A group of four digits in binary is the same as a single digit in hex. We'll get to that in a moment.

The binary digits (BITS) are the powers of 2. The values of the digits (in increasing order) are 1, 2, 4, 8, 16, 32, 64, 128, 256 and so on. 1 + 2 + 4 + 8 = 15, so the first four digits can represent a hex number. This repeats itself every four binary digits. Here are some numbers in binary, hex, and decimal

BINARY	HEX	DECIMAL
0100	4	4
1111	F	15
1010	A	10
0011	3	3

Let's go from binary to hex. Here's a binary number.

```
0110011010101101
```

To go from binary to hex, first divide the binary number up into groups of four starting from the right.

```
0110 0110 1010 1101
```

Now simply change each group into a hex number.

```
0110 ->   4 + 2     ->   6
0110 ->   4 + 2     ->   6
1010 ->   8 + 2     ->   A
1101 ->   8 + 4 + 1 ->   D
```

and we have 66AD as the result. Similarly, to go from hex to binary:

```
D39F
```

change each hex digit into a set of four binary digits:

```
D = 13    ->   8 + 4 + 1 ->   1101
```

```
3         ->   2 + 1     ->   0011
9         ->   8 + 1     ->   1001
F = 15    ->   8+4+2+1   ->   1111
```

and then put them all together:

```
1101001110011111
```

Of course, having 16 digits strung out like that makes it totally unreadable, so in this book, if we are talking about a binary number, it will always be separated every 4 digits for clarity.{1}

All computers operate on binary data, so why do we use hex numbers? Take a test. Copy these two binary numbers:

```
1011 1000 0110 1010 1001 0101 0111 1010
0111 1100 0100 1100 0101 0110 1111 0011
```

Now copy these two hex numbers:

```
B86A957A
7C4C56F3
```

As you can see, you recognize hex numbers faster and you make fewer mistakes in transcription with hex numbers.

ADDITION AND SUBTRACTION

The rules for binary addition are easy:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0  (carry 1 to the next digit left)
```

similarly for binary subtraction:

```
0 - 0 = 0
0 - 1 = 1  (borrow 1 from the next digit left)
1 - 0 = 1
1 - 1 = 0
```

On the 8086, you can have a 16 bit (binary digit) number represent a number from 0 - 65535. 65535 + 1 = 0 (65536). For binary numbers, the boundary is 65535/0. You count up or down through that boundary. The 8086 is a mod 65536 machine. That means the things that are equivalent to 35631 mod 65536 are:{2}

---

1. This will not be true of the actual assembler code, since the assembler demands an unseparated number.

2. 35631 + 29905 = 65536. -29905 = 35631 (mod 65536)

```
    (3*65536 + 35631)        (3*65536 - 29905)
    (2*65536 + 35631)        (2*65536 - 29905)
    (1*65536 + 35631)        (1*65536 - 29905)
    (      0 + 35631)        (      0 - 29905)
    (-1*65536 + 35631)       (-1*65536 - 29905)
    (-2*65536 + 35631)       (-2*65536 - 29905)
    (-3*65536 + 35631)       (-3*65536 - 29905)
```

The unsigned number 35631 and the signed number -29905 look the same. They ARE the same. In all addition, they will operate in the same fashion. The unsigned number will use CF (the carry flag) and the signed number will use OF (the overflow flag).

On all 16 bit computers, 0-32767 is positive and 32768 - 65535 is negative. Here's 32767 and 32768.

```
    32767     0111 1111 1111 1111
    32768     1000 0000 0000 0000
```

32768 and all numbers above it have the left bit 1. 32767 and all numbers below it have the left bit 0. This is how to tell the sign of a signed number. If the left bit is 0 it's positive and if the left bit is 1 it's negative.

TWO'S COMPLEMENT

In base 10 we had 10s complement to help us with negative numbers. In base 2, we have 2s complememt.

```
    0 = 65536 = 65535 + 1
```

so we have:

```
    1 0000 0000 0000 0000 =  1111 1111 1111 1111 + 1
```

To get the negative of a number, we subtract:

```
    -49 = 0 - 49 = 65536 - 49 = 65535 - 49 + 1

(65536)  1111 1111 1111 1111 + 1
   (49)  0000 0000 0011 0001
result   1111 1111 1100 1110 + 1 -> 1111 1111 1100 1111  (-49)
; - - - - -

-21874
(65536)  1111 1111 1111 1111 + 1
(21874)  0101 0101 0101 0111
result   1010 1010 1010 1000 + 1 -> 1010 1010 1010 1001 (-21847)
; - - - - -

-11628
(65536)  1111 1111 1111 1111 + 1
(11628)  0010 1101 0110 1100
result   1101 0010 1001 0011 + 1 -> 1101 0010 1001 0100 (-11628)
; - - - - -
```

```
-1764
(65536)  1111 1111 1111 1111 + 1
 (1764)  0000 0110 1110 0100
result   1111 1001 0001 1011 + 1 -> 1111 1001 0001 1100 (-1764)
; - - - - -
```

Notice that since:

```
    1 - 0 = 1
    1 - 1 = 0
```

when you subtract from 1, you are simply switching the value of the subtrahend (that's the number that you subtract).

```
    1    ->   0
    0    ->   1
```

1 becomes 0 and 0 becomes 1. You don't even have to think about it. Just switch the 1s to 0s and switch the 0s to 1s, and then add 1 at the end. Well do one more:

```
-348
(65536) 1111 1111 1111 1111 + 1
 (348)  0000 0001 0101 1100
result  1111 1110 1010 0011 + 1 ->  1111 1110 1010 0100 (-348)
```

Now two more, this time without the crutch of having the top number visible. Remember, even though you are subtracting, all you really need to do is switch 1s to 0s and switch 0s to 1s, and then add 1 at the end.

```
-658

 (658)  0000 0010 1001 0010
result  1111 1101 0110 1101 + 1 -> 1111 1101 0110 1110 (-658)
; - - - - -

-31403

(34103) 0111 1010 0100 0111
result  1000 0101 1011 1000 + 1 -> 1000 0101 1011 1001 (-31403)
```

SIGN EXTENSION

If you want to use larger numbers, it is possible to use multiple words to represent them.{3} The arithmetic will be done 16 bits at a time, but by using the method described in Chapter 0.1, it is possible to add and subtract numbers of any length. One normal length is 32 bits. How do you convert a 16 bit to a 32 bit number? If it is unsigned, simply put 0s to the left:

```
  0100 1100 1010 0111 ->  0000 0000 0000 0000 0100 1100 1010 0111
```

3. On the 8086, a word is 16 bits.

What if it is a signed number? The first thing we need to know about signed numbers is what is positive and what is negative. Once again, for reasons of symmetry, we choose positive to be
    from 0000 0000 0000 0000 0000 0000 0000 0000
    to   0111 1111 1111 1111 1111 1111 1111 1111
    (hex 00000000 to 7FFFFFFF)
and we choose negative to be
    from 1000 0000 0000 0000 0000 0000 0000 0000
     to   1111 1111 1111 1111 1111 1111 1111 1111
    (hex 10000000 to FFFFFFFF).{4}
This longer number system cycles
    from 1111 1111 1111 1111 1111 1111 1111 1111
    to   0000 0000 0000 0000 0000 0000 0000 0000
    (hex FFFFFFFF to 00000000).
Notice that by using binary numbers we are innundating ourselves with 1s and 0s.

If it is a positive signed number, it is still no problem (recall that in our 16 bit system, a positive number is between 0000 0000 0000 0000 and 0111 1111 1111 1111, a negative signed number is between 1000 0000 0000 0000 and 1111 1111 1111 1111). Just put 0s to the left. Here are some positive signed numbers and their conversions:

   (1974)
   0000 0111 1011 0110 -> 0000 0000 0000 0000 0000 0111 1011 0110
   (1)
   0000 0000 0000 0001 -> 0000 0000 0000 0000 0000 0000 0000 0001
   (3909)
   0000 1111 0100 0101 -> 0000 0000 0000 0000 0000 1111 0100 0101

If it is a negative number, where did its representation come from in our 16 bit system? -x -> 1111 1111 1111 1111 + 1 -x = 1111 1111 1111 1111 - x + 1. This time it won't be FFFFh + 1 but FFFFFFFFh + 1. Let's have some examples. (Here we have 8 bits to the group because there is not enough space on the line to accomodate 4 bits to the group).

16 BIT SYSTEM	32 BIT SYSTEM
-1964	
11111111 11111111 + 1	11111111 11111111 11111111 11111111 + 1
00000111 10101100	00000000 00000000 00000111 10101100
11111000 01010011 + 1	11111111 11111111 11111000 01010011 + 1
11111000 01010100	11111111 11111111 11111000 01010100

4. Once again, the sign will be decided by the left hand digit. If it is 0 it is a positive number; if it is 1 it is a negative number.

```
-2867
11111111 11111111 + 1     11111111 11111111 11111111 11111111 + 1
00001011 00110011         00000000 00000000 00001011 00110011

11110100 11001100 + 1     11111111 11111111 11110100 11001100 + 1

11110100 11001101         11111111 11111111 11110100 11001101



-182
11111111 11111111 + 1     11111111 11111111 11111111 11111111 + 1
00000000 10110110         00000000 00000000 00000000 10110110

11111111 01001001 + 1     11111111 11111111 11111111 01001001 + 1

11111111 01001010         11111111 11111111 11111111 01001010
```

As you can see, all you need to do to sign extend a negative number is to put 1s to the left.

Can't those 1s on the left become 0s when we add that 1 at the end? No. In order for that to happen, the right 16 bits must be 1111 1111 1111 1111. But that can only happen if the number to be negated is 0:

```
     1111 1111 1111 1111 1111 1111 1111 1111 + 1
                        -0000 0000 0000 0000
     1111 1111 1111 1111 1111 1111 1111 1111 + 1 ->

                        0000 0000 0000 0000 0000 0000 0000 0000
```

In all other cases, adding 1 does not carry anything out of the right 16 bits.

It is impossible to truncate one of these 32 bit numbers to a 16 bit number without making the results unreliable. Here are two examples:

```
+1,687,451
00000000 00011001 10111111 10011011 -> 10111111 10011011 (-16485)

-3,524,830
11111111 11001010 00110111 00100010 -> 00110111 00100010 (+14114)
```

Truncating has changed both the sign and the absolute value of the number.

CHAPTER 0.3 - LOGIC

Programs use numbers a lot. But they also ask questions that require a yes/no answer. Is there an 8087 chip in the computer? Is there a color monitor; how about a monochrome monitor? Is there keyboard input waiting to be processed? Are you going to get lucky on your date on Friday? Or, since you are a computer programmer, are you going to have a date this month? Did the file open correctly? Have we reached end of file?

In order to combine these logical questions to our heart's content, we need a few operations: AND, OR, XOR (exclusive or), and NOT.

AND

If we have two sentences "A" and "B", then ("A" AND "B") is true if (and only if) both "A" and "B" are true. "It is raining and I am wet" is true only if both "It is raining" and "I am wet" are true. If one or both are false, then 'A and B' is false. A shortcut for writing this is to use a truth table. A truth table tells under what conditions an expression is true or false. All we need to know is whether each component expression is true or false. T stands for true, F for false.

"A"	"B"	"A" AND "B"
T	T	T
T	F	F
F	T	F
F	F	F

Notice that the truth table does NOT say anything about whether the expression makes sense. The sentence:

"It's hot and I am sweating."

is a reasonable expression which may or may not be true. It will be true if both "It is hot" and "I am sweating" are true. But the sentence:

"The trees are green and Quito is the Capital of Ecuador."

is pure garbage. It does not satisfy our idea of what a sensible expression should be, and should NEVER be evaluated by means of a truth table. The warranty on a truth table is, if the expression makes sense, then the truth table will tell you under what conditions it is true or false. If the expression does not make sense, then the truth table tells you nothing.

Fortunately, this problem really belongs to philisophical logic. When you use logical operators in your program, there will be a

well defined reason for each use. If you start doing screwy things, your program probably won't run.

OR

There are two different types of OR alive and kicking in the English language - the exclusive OR (A or B but not both) and the inclusive OR (A or B or both).

A mother tells her child "You can have a piece of cake or a piece of candy." Does this mean that he can have both if he wants? Of course not. He can have one or the other, but not both. This is XOR, the exclusive or. The truth table for this is:

"A"	"B"	"A" XOR "B"
T	T	F
T	F	T
F	T	T
F	F	F

'A XOR B' is true if exactly one of them is true. If they both are true 'A XOR B' is false. If neither is true, 'A XOR B' is false. Examples of XOR are:

1) We will either go to Italy or to Japan on our vacation.
2) I'll either have a tuna salad or a chef's salad.
3) He'll either buy a Lamborghini or a BMW.

Consider this sentence: "To go to Harvard, you need to have connections or to be very smart." Do we want this to mean that if you have connections but are very smart, you are automatically excluded from going to Harvard? No. We want this to mean one or the other or perhaps both. Sometimes you write this as 'and/or' if you want to be absolutely clear. This is the inclusive OR. The truth table for OR is:

"A"	"B"	"A" OR "B"
T	T	T
T	F	T
F	T	T
F	F	F

'A OR B' is true if one or both of them are true. If both are false, then it is false. Examples of OR are:

1) They'll either go to Italy or to Austria on their vacation.
2) I'll have either steak or shrimp at The Sizzler.
3) He'll buy either a paisley tie or a rep tie.

The three sentences for XOR and OR mimic each other on purpose. In the English language, you know which type of OR is being used by what is being talked about. You know intuitively which one

applies. If someone buys two different ties you are not suprised. If someone buys two expensive cars at the same time you are quite surprised.{1} With very few exceptions, if you confuse the two you are doing it on purpose. If your father says "You can have the car on Friday night or on Saturday night." and you don't understand which OR applies, it's not his fault.

NOT

The final logical operation is NOT. The sentence: "It is not raining." is false if it is raining, and true otherwise. The truth table is:

"A"	NOT "A"
T	F
F	T

This is self-explanatory.

Amazingly enough, this is all you need to know about logic to be a quality programmer. Trying to make very complex combinations of these logical operations may be fun for philosophy, but it is death to a program. KISS is the operative word in programming.{2}

1. Especially if his job is working the cash register at Sears.

2. Which, if you don't know, is Keep It Simple, Stupid.

CHAPTER 0.4 - MEMORY

The basic unit of memory on 8086 machines is the byte.{1} This means that in every memory cell we can store a number from 0 to 255. Each memory cell has an address. The first one in memory has address 0, then address 1, then 2, then 3, etc.

The registers on the 8086 are one word (two bytes) long. This means that any register can store and operate on a number from 0 to 65,535. (It also has some registers which can operate on bytes and can store and operate on numbers from 0 to 255.)

Memory is physically external to the 8086. Registers are physically internal to the 8086; they are actually on the chip.

One of the ways that we can access memory on the 8086 is by putting the address of a memory cell in a register and then telling the 8086 that we want to use the data at that memory address.

Since each byte has its own address, and since we can't have a number larger that 65,535 in any one register, it is impossible to address more than 65,535 bytes with a single register. Back in the dark ages, that might have been enough memory, but it sure isn't enough nowdays.

Intel solved the problem by creating SEGMENTS. Each segment is 65,536 bytes long, going from address 0 to address 65,535.{2} You tell the 8086 where you want to go in memory by telling it which segment you are in and what the address is within that segment. Segments are numbered from 0 to 65,535. That is, there are 65,536 of them.

As a design decision, Intel decided that a segment should start every 16 bytes. This decision was made in the late 70's and is cast in stone. On the 8086, a segment starts every 16 bytes. Here is a list of some segments, with the segment number and the segment starting address in both decimal and hex.

SEGMENT NUMBER		STARTING ADDRESS	
0d	0h	0d	00h
1d	1h	16d	10h
2d	2h	32d	20h
3d	3h	48d	30h
4d	4h	64d	40h

---

1. As it is on all computers.

2. The last segments are actually less that 65,535, as will be explained later.

200d	C8h	3,200d	C80h
21694d	54BEh	347,104d	54BE0h
51377d	C8B1h	822,032d	C8B10h

One thing you should note is that in hex, the segment number is the same as the starting address, except that the starting address has an extra 0 digit on the right.

These segments overlap. A new segment starts every 16 bytes, and they are all 65,535 bytes long. This means that with the exception of the first 16 bytes, any address in memory is in more that one segment. Here are some addresses. The word "offset" means the number of bytes from the beginning of the segment. (It is possible for a memory cell to have an offset 0). The offset is shown in both decimal (d) and hex (h):

memory address 55 (37h)

Seg #	Offset	Offset
0	55d	37h
1	39d	27h
2	23d	17h
3	7d	7h

Thus the address 55 is in 4 different segments, and can be addressed relative to any one of them.

memory address 17,946 (461Ah)

Seg #	Offset	Offset
0	17,946d	461Ah
1	17,930d	460Ah
2	17,914d	45FAh
...	...	
1120	26d	1Ah
1121	10d	0Ah

The address 17,946 is in 1122 different segments, and can be addressed relative to any of them. Notice that as the segment number goes up one segment, the offset address goes down 16 bytes (10h).

Above the address 65,519, every memory cell can be addressed by 4,096 different segments.

Because of the way the addresses are generated on the 8086, the maximum memory address possible is 1,048,575 (FFFFF hex.) This means that the final segments are less than 65,536 bytes long. In this table "Address" is the starting address of the segment. All the following numbers are decimal:

Segment #	Address	Max Offset
65,535d	1,048,560d	15d
65,534d	1,048,544d	31d
65,533d	1,048,528d	47d
...	...	...

64,000d	1,024,000d	24,575d

Let's look at these same numbers in hex:

Segment #	Address	Max Offset
FFFFh	FFFF0h	Fh
FFFEh	FFFE0h	1Fh
FFFDh	FFFD0h	2Fh
...	...	...
FA00h	FA000h	5FFFh

The maximum addressable number is FFFFFh, which is why these segments are cut short. There are 4,095 segments at the top which are less than 65,536 bytes long. Don't worry, though, because this top section of memory belongs to the operating system, and your programs will never be put there. It will never affect you.

Back in the late 70s, a million bytes of memory seemed like a lot. In the 60s, large mainframe computers had only a half-million bytes of memory. In the 70s memory was still exhorbitantly expensive. Nowdays, however, you practically need a megabyte just to blow your nose. But this segmentation system is cast in stone, so there is no way to get more memory on the 8086.{3}

In the beginning, when we make a program, we will use one segment for the machine code, one segment for permanant data, and one segment for temporary data. If we need it, this gives us 196,608 bytes of usable memory right off the bat. As you will see by the time we are finished, ALL memory is addressable - we just need to do more work to get to it all.

All this talk about segments and offsets may have you concerned. If you have to keep track of all these offsets, programming is going to be very difficult.{4} Not to worry. It is the assembler's job to keep track of the offsets of every variable

---

3. This megabyte rule is unalterable. EMS (extended memory) is actually a memory swapping program. On the 28086 and 80386 you can have more than one megabyte of memory but the program can only access one megabyte. The program reserves a section of its one megabyte for a transfer area. It then calls EMS which transfers the data from this extended memory to the transfer area. It is in effect a RAM disk. It is like using a hard disk but is much faster. If Intel had bitten the bullet with the 80286 and said that a segment would start every 256 bytes instead of every 16 bytes, we would have 16 megabytes of directly accessible memory instead of 1 megabyte. Hindsight is such a wonderful thing.

4. Remember, an offset is just how many bytes a memory cell is from the beginning of the segment.

and every label in your program.{5}

Which segments your program uses is decided by the operating system when it loads your program into memory. It puts some of this information into the 8086. At the start of the program, you put the rest of the information into the 8086, and then you can forget about segments.

NUMBERS IN MEMORY

The largest number you can store in a single byte is 255. If you are calculating the distance from the sun to Alpha Centauri in inches, obviously one byte is not enough. Unfortunately, the 8086 can't really handle large numbers like that.{6} It can handle numbers which are 16 bits (2 bytes) long. However, with subprograms supplied with all compilers, we can handle large numbers, though rather slowly if we don't use an 8087. All these different programs need a standard way to write numbers in memory, and this standard is supplied by Intel. The standard is :

(1) integers can be 1, 2, or 4 bytes long. This corresponds to -128 to +127 , -32,768 to +32,767, and -2,147,483,648 to +2,147,483,647.

(2) scientific floating point numbers which have an exponent and can be very large. They come as 4 byte and 8 byte numbers. We will not deal with them at all, but we need to know how they are stored.

(3) Commercial or BCD numbers which occupy 1/2 byte per digit. Since some of the 8086 instructions are concerned with these we will cover them, but if you are not curious about them, you can skip that section. The standard is a 10 byte number.

Let's look at a number. For the rest of this section, all numbers will be hex, and if a number is longer than one byte, we will display it with a blank space between each byte. If it is a one byte number - e.g. 3C, we know exactly where we are going to put it. But what if a number is 4 bytes long - e.g. 2D F5 33 0A - and we want to put it in memory starting at offset 264. We have two choices:

2D F5 33 0A

Address	Choice 1	Choice 2
267	2D	0A
266	F5	33
265	33	F5

5. As in other languages, a label is a name that marks a place in the code. In BASIC, labels are actually numbers (such as 500 in the instruction GOTO 500). Labels are frowned on in Pascal and C, but are the lifeblood of assembler language.

6. But fortunately, the 8087 can.

```
         264              0A          2D
```

Neither choice is better than the other. Choice 1 puts the right-most byte in low memory, choice 2 puts the right-most byte in high memory.{7} The right-most byte is called the LEAST SIGNIFICANT BYTE because it has the least effect on a number, while the left-most byte is called the MOST SIGNIFICANT BYTE because it has the most effect on a number. In fact, Intel picked choice #1 for the 8086 (which has the least significant byte in low memory), and Motorola picked choice #2 for the 68000 (which has the most significant byte in low memory).

This is consistent for both the 8086 and the 8087: THE LEAST SIGNIFICANT BYTE IS ALWAYS IN LOW MEMORY: EACH NUMBER IN MEMORY STARTS WITH THE LEAST SIGNIFICANT BYTE. Remember that, and you'll save yourself some trouble.

One problem you will run up against is that when we draw pictures of memory, we often draw from left to right, that is:

```
  ADDRESS        264  265  266  267
```

When we do that, things start looking wierd. For 2D F5 33 0A we have:

```
  ADDRESS        264  265  266  267
  VALUE          0A   33   F5   2D
```

This is exactly backwards. Remember, memory doesn't go from left to right, it goes UP from 0, and THE LEAST SIGNIFICANT BYTE IS ALWAYS IN LOW MEMORY. You will certainly make some mistakes till you get used to being consistent about this. The right hand digit of a number is always in low memory. If you think of memory as being VERTICAL:

```
1E A3 07 B5
               Value       Address
                 1E         4782
                 A3         4781
                 07         4780
                 B5         4779
```

rather than being LEFT TO RIGHT:

```
  Address   4779 4780 4781 4782
  Value      B5   07   A3   1E
```

you will be much better off.

---

7. Low memory always means the low addresses and high memory always means the high addresses.

CHAPTER 0.5 - STYLE

Finally, it is time to say a word about style. Assembler code is by its nature difficult to read. It divides any concept into a number of sequential steps. If you have the BASIC statement:

```
MINUTES = DAYS * 1440
```

You get the idea because you can scan the line to see what is wanted. The assembler code for the above line is: {1}

```
mov  ax, days
mov  bx, 1440
mul  bx
mov  minutes, ax
```

In BASIC, the concept was imbedded in the expression. In assembler it was lost. This means two things. First, you must be religious about documenting every step. If you come back to something two or three months later and you haven't documented what you are doing, it may take you longer to figure out what you did than it would to completely rewrite what you did.

Secondly, if you are a person who likes code like this:

```
x = (y + k) / (z - 4)
```

you are headed for big trouble. At the assembler level it is CRITICAL that you give every variable a name that signifies exactly what it is. If you are counting the number of correct answers, then the variable should be:

```
correct_answers
```

not 'K'. If you are looking at the remainder from a division, then the variable should be:

```
remainder
```

not 'R'. If you try to use short names like 'x', 'k' and 'y', I will guarantee that for every minute you save by not having to type in long variable names, you will lose 10 minutes by not being able to figure out what is going on when you reread the code. In this tutorial we will use multiple words connected by underscores:

```
first_positive_prime
median_age
oldest_student
```

---

1. Don't worry about what this code does. You will learn soon enough.

The Microsoft assembler allows 31 significant characters in a name. Even though there are several other characters allowed, we will use only letters, the underscore, and numbers (where appropriate):

```
approximation1
approximation2
approximation3
```

This should make your code similar to well written C or Pascal code, and greatly increase the readability of the code.

CHAPTER 1 - SOME SIMPLE PROGRAMS

It is now time to start writing assembler code. One of the problems with writing in assembler is that there is no way to get input into the program or output from the program until you are very far along with learning assembler language. This is a Catch-22 situation. You can't learn assembler easily without access to input and output, and you can't write i/o routines till you know assembler.{1} Help is at hand. Included on this disk is a file called asmhelp.obj. It is actually a series of programs that will allow you to get input from the keyboard and print output to the screen. It has some other features which will be explained later.

The second problem at the start is that every assembler program has a lot of overhead. These are standard instructions and formats that you need to get the program to work AT ALL. This disk contains templates that contain all the overhead, so to write a program you just make a copy of the template and enter the code and data at the appropriate place. By the end of this sequence of lessons, you will know how to make templates yourself and know the meaning of each word in the template. For now, you have to have faith that what is written is necessary, and that you will learn the meaning of everything later.

Let's start. At the end of this chapter is the template we will use for now - temp1.asm. These templates are in the subdirectory \template.

Let's call the first program prog1.asm (very original). All programs in assembler must have the file extension .asm so make a copy by giving the command:

```
>copy \template\temp1.asm  prog1.asm
```

You are now ready to enter code. Open up prog1.asm with your editor, and take a look at it. It should look the same as temp1.asm.

Where it says "put name of program here" - that is for your personal use so you can see the program name while in the editor. The assembler ignores everything after a semicolon. All the lines that start with a semicolon are there for visual separation or for comments. The lines with asterisks separate segments. Yes, the assembler is going to make this program into three segments. You should put all code between the line labeled "START CODE BELOW THIS LINE" and the line labeled "END CODE ABOVE THIS LINE".

---

1 Just to give you an idea of how contradictory the situation is, asmhelp.obj was written in assembler language and consists of about 3500 lines (that's about 50 pages), yet you need to be using it from day one.

Later you will get more flexibility.

Also notice the lines starting with the word EXTRN. Those lines tell the assembler that the subroutines (such as print_num) are in a different file and must be found when this program is linked. The assembler enters each EXTRN name in a list and records each place that an EXTRN subroutine is requested. It is possible that one of these subroutines is never requested. That's fine. However, every one of these subroutines must be present at link time or you will get a link error. This is true even if the subroutine is on the list but never requested.

Since the overhead is so long, and I am so lazy, the template will never be included in the description of the program. What you will get is the name of the template, then any data you need, then the code. It will look like this:

```
TEMP1.ASM
; - - - - - START DATA BELOW THIS LINE
the data is written here
; - - - - - END DATA ABOVE THIS LINE

; - - - - - START CODE BELOW THIS LINE
the code is written here
; - - - - - END CODE ABOVE THIS LINE
```

If there is no data, no data section will be included. If there is data, it should be written in the segment named DATASTUFF between the lines "START DATA..." and "END DATA ...". The code should be written between the lines that say "START CODE ..." and "END CODE ...".

For our first program, the description will look like this:

```
TEMP1.ASM
; - - - - - START CODE BELOW THIS LINE

first_label:
    call get_num
    call print_num
    jmp  first_label

; - - - - - END CODE ABOVE THIS LINE
```

That's all we need. If we needed to write all the overhead (starting with the line "main proc far") we would have:

```
main     proc far
start:   push ds
         sub  ax, ax
         push ax

         mov  ax, DATASTUFF
         mov  ds, ax

; + + + + + + + + + + +
```

```
first_label:
    call get_num
    call print_num
    jmp  first_label

; + + + + + + + + + + +

    ret

mainendp
etc.
```

You can see that a simple four line program has blossomed into a monster. I'm assuming some intelligence on your part. Until further notice, the code goes between the "START CODE' and the "END CODE" lines and the data goes in the DATASTUFF segment between the "START DATA" and the "END DATA" lines.

It's time to type in the program listed above. Be careful when you type. When you are done I'll explain it.

In assembler we need a way to label different spots in the code. We use labels. A LABEL is a name (at the beginning of a line) which is immediately followed by a colon. A label doesn't generate any code. The assembler merely keeps track of where the label is for future use. The label we are using is named first_label.

The CALL instruction tells the assembler to call the subroutine listed after the call.{2} We are calling two subroutines; first get_num which gets a number, then print_num, which prints a number in a variety of styles.

Finally, JMP tells the assembler that you want to jump to the label listed after it. It is the same as GOTO in BASIC.

If you look at the program, you will notice that we have an infinite loop. It was designed that way. It takes a fair amount of code to exit gracefully, so we will always exit ungracefully. When you are tired of the program, simply press Control-C. That should get you out. That way you can try out something an indefinite number of times, and when you have finished you can press CTRL-C to quit the program.

One warning about machine language before we start. There is no safety net, so before you start a machine language program, make sure all files are closed (i.e. that you have no other programs in memory). We will NEVER open a file in one of our programs.

---

2 I am using the words subroutine, routine, program and procedure (the technical word) interchangebly throughout the book. A program is actually a group of one or more procedures, but I'm not going to be too strict about it. Context should tell whether we are talking about a single procedure or a whole program.

Your programs are almost certain to wind up in zombie space from time to time. If that happens, your choices are (1) hit CTRL-C. If that doesn't work, then (2) hit CTRL-ALT-DEL. As a last resort, you can (3) hit a reset button or shut the machine down.

For that reason, memorize this mantra: BACKUP YOUR PROGRAMS AND BACKUP YOUR BACKUPS.

Double check that you have typed in the assembler code correctly. Now it's time to assemble it. I am assuming that you have the Microsoft assembler.{3} Type:

```
>masm  prog1 ;
```

The first thing you will see is the copyright notice:

```
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.
```

The file extension .asm is unnecessary; it's understood.
The semicolon is to speed things up. If you don't use it, the assembler will ask you if you want to change any of the default choices. If you type just masm:

```
>masm
```

then you need to give the name of the assembler text file on the first line:

```
Source filename [.ASM]: prog1
Object filename [prog1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:
```

but press ENTER for the other options. If you type masm prog1:

```
>masm prog1
```

You don't want to change any of the default settings:

```
Object filename [prog1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:
```

When the assembler asks you about options, hit the ENTER key.

If you have made any errors, the assembler will tell you which line they are on and give you a description of the problem. Make a hard copy of them on the printer, then use your editor and find the line. Unfortunately, at this stage of the game, it will be

---

3 If you are using A86, then consult A86.APP. If you are using Turbo Assembler, then consult TASM.APP. They are both located in the \APPENDIX subdirectory on disk 3.

very difficult for you to figure out what the problem is. You will have to struggle through the first 4 or 5 programs before things start getting easier. All the programs on these disks have been compiled on a Microsoft v5.1 assembler. They have assembled. They have been run, and they work. Don't tamper with the template and copy the code exactly and everything should work.

If you haven't made any errors, the assembler will say:

```
      0 Warning Errors
      0 Severe  Errors
```

LINKING

The assembler has given you back another program named prog1.obj - the same name with the extension .obj. I am assuming that you have all used the linker with compiled programs. If you haven't, you may be getting in over your head by using machine language. All the extra subroutines are in a program called asmhelp.obj. Its pathname is \asmhelp\asmhelp.obj. You want to put it in the root directory of your current drive. In the whole book, we will assume that its pathname is:

```
    \asmhelp.obj
```

If you put it somewhere else, you will have to modify the pathname whenever it appears. Link the two modules by writing:

```
    >link prog1+\asmhelp ;
```

The copyright notice will appear:

```
   Microsoft (R) Overlay Linker  Version 3.61
   Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.
```

This time the file extensions are understood to be .obj. The semicolon is to avoid having to make default choices. If you type:

```
    >link
```

then you need to put the module names after the first prompt:

```
    Object Modules [.OBJ]: prog1+\asmhelp
    Run File [PROG1.EXE]:
    List File [NUL.MAP]:
    Libraries [.LIB]:
```

but press ENTER for the other choices. If you type:

```
    link prog1+\asmhelp
```

You need to do nothing extra:

```
    Run File [PROG1.EXE]:
```

```
    List File [NUL.MAP]:
    Libraries [.LIB]:
```

When the linker asks for choices, simply press the ENTER key.

The linker gives the executable file the name of the first object file on the line, so you should always put your program first and asmhelp.obj second.

If there are no errors, you are ready to go. If there are errors, once again, they will be very difficult to trace. Go back and check everything from the beginning.

You are now ready to run the program. Type:

```
    >prog1
```

The program will start. The first thing you will see is a copyright notice.

```
    The PC Assembler Helper   Version 1.0
    Copyright (C) 1989  Chuck Nelson   All rights reserved.
```

It appears the first time you call a subprogram in the module asmhelp.obj.

The program will request a number. Give it any legal signed or unsigned number. It should be no longer than 5 digits. Press ENTER, and it will display the possible ways that that number can be thought of by the computer.

```
    Enter any decimal number  4410
      HEX    SIGNED   UNSIGNED   CHAR          BINARY
     113AH   +04410    04410     11  : **  0001000100111010

    Enter any decimal number  30486
      HEX    SIGNED   UNSIGNED   CHAR          BINARY
     7716H   +30486    30486      w 16 **  0111011100010110
```

If the signed or unsigned number doesn't look the same as what you entered, then the number you entered is too big for a 16 bit computer. For signed numbers, the limits are +32767 to -32768 and for unsigned numbers, the limits are 0 to 65535.

```
    Enter any decimal number  -64661
      HEX    SIGNED   UNSIGNED   CHAR          BINARY
     036BH   +00875    00875     03  k **  0000001101101011

    Enter any decimal number  94547
      HEX    SIGNED   UNSIGNED   CHAR          BINARY
     7153H   +29011    29011      q  S *   0111000101010011
```

Lets look at the numbers. Each type of output is labeled. After a hex number, there is an 'H' and after the characters, there is a '*'. This is always true. Every time you print a hex number, there will be an 'H', and every time you print a character, there

will be a '*'. This is so you will always know what is being printed. Also notice that a signed integer ALWAYS has a sign and an unsigned integer NEVER has a sign.

Not all characters are visible. Ascii 0 - 32 are invisible (32 is a blank). On the PC, ascii 33-255 are visible, but ascii 127 and ascii 255 are problematic. Therefore, if the ascii code is 0-32, 127 or 255, that character will be printed as a hex number, not a character, and print_num will signal the event by printing a double asterisk '**' instead of a single one. This has happened in the first two examples. (11 : **) and (w 16 **). The first one is the hex number 11 followed by the character ':' and the second one is the character 'w' followed by the hex number 16. Both are signalled by the double asterisk '**' instead of the single asterisk '*'.

Do a few examples. When you are done looking at the numbers, press CTRL-C and you will exit the program.

```
    Enter any decimal number  ^C
```

PROGRAM 2

The second program is almost the same as the first one. The program takes input from the keyboard and displays it in a variety of styles. This time, however, it is going to ask for different inputs: ascii, hex, binary and decimal. If you make an error in the input, the subroutine will prompt you again for the input. Here's the program:

TEMP1.ASM

```
;+ + + + + + + + + + START CODE BELOW THIS LINE

first_label:
    call get_num            ; 1 to 5 digit signed or unsigned
    call print_num

    call get_ascii          ; 1 or 2 characters
    call print_num

    call get_binary         ; a 1 to 16 bit binary number
    call print_num

    call get_hex            ; a 1 to 4 digit hex number
    call print_num

    jmp  first_label

;+ + + + + + + + + + END CODE ABOVE THIS LINE
```

The things to the right of the semicolons are comments. You do not need to type them in if you don't want to. Once again, assemble the program. (There should be no warning or severe errors. If something is wrong, it is most likely a typing error.) Then link it with asmhelp.obj. Remember - your program should be

the first one listed.{4}

If all is well, run the program. It will ask you for a number (that is a signed or unsigned number), ascii characters, a binary number, and a 4 digit hex number (0-9,A-F).

```
Enter any decimal number  27959
  HEX    SIGNED   UNSIGNED   CHAR          BINARY
 6D37H   +27959    27959      m  7 *   0110110100110111

Enter one or two ascii characters  $%
  HEX    SIGNED   UNSIGNED   CHAR          BINARY
 2425H   +09253    09253      $  % *   0010010000100101

Enter a two byte binary number  0101111001100010
  HEX    SIGNED   UNSIGNED   CHAR          BINARY
 5E62H   +24162    24162      ^  b *   0101111001100010

Enter a two byte hex number  784d
  HEX    SIGNED   UNSIGNED   CHAR          BINARY
 784DH   +30797    30797      x  M *   0111100001001101
```

Once again, this is an infinite loop, so in order to quit, you need to hit CTRL-C.

The purpose of these first two programs is to remind you that the computer doesn't care whether you think you are storing binary numbers, characters, hex numbers, signed numbers or unsigned numbers. They all wind up in the computer as a series of 1s and 0s, and you can use these 1s and 0s any way you like. It's up to you to keep track of them.

If you feel comfortable with the way we are writing, assembling and linking programs, you are ready to start looking at the 8086 itself.

---

4 For your convenience, there is a batch file on the disk called asmlink.bat. Its pathname is \asmhelp\asmlink.bat. It is one line long and looks like this:

```
link %1+\asmhelp ;
```

If you use this batch file, you will never have an order problem. If your file is named myfile.asm, then type:

```
>asmlink myfile
```

```
; TEMP1.ASM        The first assembler template

; put name of program here

; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
STACKSEG    SEGMENT   STACK  'STACK'

             dw     100 dup (?)

STACKSEG    ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
DATASTUFF    SEGMENT   PUBLIC  'DATA'

; + + + + + + + + + + + + + + + START DATA BELOW THIS LINE

; + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

DATASTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
CODESTUFF    SEGMENT   PUBLIC  'CODE'

       EXTRN  print_num:NEAR , get_num:NEAR
       EXTRN  get_ascii:NEAR , get_hex:NEAR , get_binary:NEAR

       ASSUME cs:CODESTUFF, ds:DATASTUFF

main   proc far
start: push  ds               ; set up for return
       sub   ax,ax
       push  ax

       mov   ax, DATASTUFF
       mov   ds,ax

; + + + + + + + + + + + + + + + START CODE BELOW THIS LINE

; + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE

       ret

main   endp

CODESTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

       END     start
```

SUMMARY

CALL
CALL calls a subroutine.

```
call get_num
```

JMP
JMP jumps to the indicated label.

```
JMP  label47
```

LABEL
A label is a name at the beginning of a line which is followed by a colon. It is used to mark a spot in the program.

```
label47:
```

There are three different things which will be mentioned from time to time, so it's best to define them now.

ASSEMBLER INSTRUCTIONS (CODE) is the text that you type in and give to the assembler.

MACHINE CODE is the code that the assembler generates. After some adjustment by the linker, it is readable by the 8086. It is the actual code that controls the program.

MICROCODE is the code that is imbedded in the 8086 itself. Each instruction has its own set of mini instructions within the 8086. This is the MICROCODE.

CHAPTER 2 - DATA

Before you start using data, you need to know what data looks like. It is not necessary for the data to have a name. For instance, the following definition is perfectly legal:

```
    db   "Mary had a little lamb."
```

Unfortunately, the assembler has no way to find it. The normal thing is to start the line with a name, and then give the definition of the data. The assembler processes the data line by line, so a definition on one line does not carry over to another line. We can have:

```
    poem      db   "Mary had a little lamb,"
```

Notice that names for data don't have colons after them. What if we wanted to continue the poem? It isn't going to fit all on one line. No problem. All we need to do is define the following lines without a name.

```
    poem      db   "Mary had a little lamb,"
              db   "It's fleas were white as snow,"
              db   "And everywhere that Mary went,"
              db   "She scratched and scratched and scratched."
```

The assembler still can't find lines 2-4, but starting at the first byte of "poem", it can go all the way through the poem one byte after the other. By the way, there are no carriage returns in the poem right now. They will come later.

So we have the name part, the db part, and the data part. What is that db anyway. It stands for Define Byte. Whenever you give the name "poem" to the assembler, it knows that you want to deal with the data one byte at a time. If you try working a word at a time, you will get an assembler error. The legal definitions are:

```
    DB   define byte         [ 1 byte ]
    DW   define word         [ 2 bytes ]
    DD   define doubleword   [ 2X2 bytes = 4 bytes ]
    DQ   define quadword     [ 4X2 bytes = 8 bytes ]
    DT   define ten-byte     [ 10 bytes ]
    DF   define farword      [ 6 bytes - used for 80386 only ]
```

Every time you use one of these directives, the assembler allocates the number of bytes in brackets for EACH variable. For instance in:

```
    db        "Mary had a little lamb,"
```

each character inside the quotes is a variable. That's 23 variables X 1 byte = 23 bytes. In:

```
    dq        0, 1, 2, 3, 4
```

each number is a variable. 5 variables X 8 bytes = 40 bytes. Notice from these examples that you can have more than one variable on a line but they all share the same defining type. What do you do if you have an uninitialized variable, i.e. you don't know its starting value? Easy as pie. Here's a four byte variable:

```
    some_data      dd   ?
```

The question mark lets the assembler know that you didn't forget the number but rather you didn't know the number.

The commas are separators. When you write a comma, the assembler expects another piece of data on the line. If it doesn't get the number, it is an error. That means there can be no commas inside a number.

```
    dw   32,421
```

is two variables: 32 and 421.

What if you want to make an array? The assembler has a directive for that too:

```
    dw   150  dup ( 400 )
```

The 'dup' is for duplicate. This makes 150 two byte copies and puts the number 400 in each one.

```
    db   273  dup ( 'c' )
```

This makes 273 one byte copies and puts the letter 'c' in each one.

```
    dd   459  dup ( 1, 2, 3, 4, 5 )
```

This makes 459 copies of what is inside the parentheses. That means (5 variables X 4 bytes) X 459 for a total of 9180 bytes. Starting from the beginning of the array, we will have the sequence: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3,...

```
    dq   20000 dup ( 455 )
```

This makes 20000 eight byte copies and causes an assembler error because there is a limit of 65,536 bytes for the data and you have used 160,000 bytes (20,000 X 8).

```
    db   7  dup ( 'Mary had a little lamb,')
```

This makes 7 copies of 'Mary had ..' which is 23 bytes, for a total of 161 bytes.

```
    dw   39 dup ( 28 dup ( 0 ) )
```

The assembler even supports nesting, so you can make a multi-dimensional array. This is a 39 X 28 array initialized to zero. 39 copies of 28 two byte numbers is 2184 bytes.

The standard form for arrays is (1) first define the data type, (2) then say how long the array is followed by the keyword "dup" and (3) put the initial value inside the parentheses. What if you don't know the initial value? Simple:

```
dw   347  dup ( ? )
```

The question mark lets the assembler know that you don't know.

DEFINING NUMBERS

What kinds of data can you have?

1. A single character inside single or double quotes:
```
'a' , "&" , '|'
```

2. A string inside single or double quotes:

```
"Mary had a little lamb,"
'Mary had a little lamb,'
```

Each character is stored as a byte, and the bytes are stored consecutively. If the array starts at address 2743, 2743 = 'M', 2744 = 'a', 2745 = 'r', 2746 = 'y', 2747 = ' ', etc. As usual in these instances, if you want a double quote inside a double quoted string or a single quote inside a single quoted string, you need to use a pair:

```
"Mary asked her fleas ""Why don't you join the circus?"""
'Mary asked her fleas "Why don''t you join the circus?"'
```

3. A decimal number. Decimal is the default:

```
27, 44, 641, 89
```

4. A hex number. A hex number must start with a number, so if the highest digit is A - F, there must be a 0 in front.{1} b77h is illegal, 0b77h is legal. All hex numbers must be followed by an 'h':

```
0a162H , 0329H , 0DDDh , 7h
```

5. An octal (base 8) number. An octal is followed either by the

---

1 When the assembler looks at something it needs to know whether it is a name or a number. Is 'A7' a name or a hex number? Is '3D' a name or a number? To solve this problem, all assemblers and all compilers insist that -> if the first character is a number, it's a number; if the first character is not a number, it is not a number. That is why you can't start a variable name with a number.

letter q or the letter o:

```
    641q , 2345o , 1472o
```

6. A binary number. A binary number is followed by a b:

```
    0100100b , 1b , 01001000111010b
```

Any of these types can be mixed on a line. For instance:

```
    db   "Mary had a little lamb," , 13 , 10
```

13 followed by 10 is CRLF, the PC signal for a carriage return. A string in the C language ends with the number 0. If we wanted a C string with CRLF, we would have:

```
    db   "Mary had a little lamb," , 13 , 10 , 0
```

Another mixed example:

```
    dw   7 , 010010b , 0FFFFh , 037q
```

is dopey but legal.

You can also have an equation, as long it resolves to a number. This calculation is done by the assembler, so the values of variables are not allowed:

```
         dw   ( ( 19 * 7 * 25 ) + 6 ) / ( 9 + 7 )
```

is legal, but:

```
data1    dw   25
data2    dw   7
         dw   ( ( 19 * 7 * data1 ) + 6 ) / ( 9 + data2 )
```

is illegal. Everything must be a constant. Remember that when the assembler starts calculating it might truncate the partial answers, so don't get too fancy.

SUMMARY

The assembler works one line at a time. Each line with data must start with a data type declaration (after an optional name.)

DATA TYPES

DB	define byte	(1 byte)
DW	define word	(2 bytes)
DD	define doubleword	(2X2 bytes = 4 bytes)
DQ	define quadword	(4X2 bytes = 8 bytes)
DT	define ten-byte	(10 bytes)
DF	define farword	(6 bytes - used for 80386 only)

COMMON INTEGER TYPES

TYPE	MAX SIGNED	MAX UNSIGNED
byte	-128/+127	255
word	-32768/+32767	65535
doubleword	-2147483648/+2147483647	4294967295

Note that the max. negative integer is 1 larger than the max. positive integer.

POSSIBLE BASES FOR CONSTANTS

b	binary data
o,q	octal data
d	decimal data (default)
h	hex data (must start with a number 0 - 9)

ARRAY DEFINITIONS

```
d*        num1      dup ( data1 )
```

Using the d* data type (db, dw, dd, dq, etc.) make num1 copies of data1 (data1 may be either a single piece of data or a group of data.)

MULTIPLE DATA ON ONE LINE

Different data elements on the same line are separated by commas. All elements on the same line have the same data type.

CHAPTER 3 - ASMHELP

We are now going to introduce both the 8086 registers and a program for looking at them. You are going to get information flying at you at a rapid pace, so read both this and the next chapter carefully and slowly.

REGISTERS

The 8086 has a number of registers. Remember, registers are places for storing data that are internal to the 8086 chip. They are much faster, but there are very few of them.

There are 6 registers that you can use for addition and subtraction of word (2 byte) sized numbers, as well as logical operations on word (2 byte) numbers or data. These registers are AX, BX, CX, DX, SI, DI. In addition, there is a register which works the same way, but has a special function in all high-level languages (Basic, Pascal, C, etc.). This is BP, the base pointer.

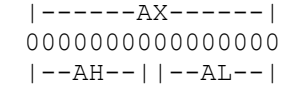
There is one more register that performs the same operations as the above seven, but it is RESERVED for special use and should never be used for anything. It is called SP (the stack pointer).

There are 4 registers that tell the 8086 which memory segments you are in. They just sit there and help the 8086 find things in memory. You will learn how they work later. They are CS, DS, SS, ES. (That is Code Segment, Data Segment, Stack Segment and Extra Segment respectively).

There is the flags register which contains all the information the 8086 needs to evaluate its state. We will learn about this later. The flags register has no name, and there are machine instructions for manipulating individual flags in the register.

Finally, there is IP, the instruction pointer, which points to the machine instructions. You have no direct access to this, which is good because you would screw it up for certain. The 8086 handles the IP automatically and correctly.

One word (two bytes or 16 bits) is the largest piece of data that the 8086 can handle naturally. It is possible to handle larger things, but we do it through software (which is slower), not hardware (which is faster). Sometimes we want to handle things one byte at a time as when we work with characters. The 8086 gives us this possibility by letting us divide the AX, BX, CX, and DX registers into an upper half and a lower half. For any or all of these registers, we can replace one 2 byte register by two 1 byte registers. The data in the full register stays the same, but we can look at each half. The two parts of AX are called AH (for A high) and AL (for A low).



This is the 16 bit binary number 0 in the AX register. Using AX allows us to manipulate all 16 bits. Using AH allows us to manipulate the upper 8 bits (without affecting the lower 8 bits), and using AL allows us to manipulate the lower 8 bits without affecting the upper 8 bits. Similarly, for BX we have BH, BL, for CX we have CH, CL, and for DX we have DH, DL.

SHOW_REGS

We have named all the registers, now let's take a look at them. Included in the module asmhelp.obj is a program called show_regs. It shows all the above registers on the top 10 lines of your screen and allows you to enter data normally underneath.

When you call show_regs, it puts the current value of all the registers on the screen. Those values stay on the screen until the next call - i.e. the program does not change what is on the screen even though the registers may be changing value. You need to call show_regs every time that you want to see the current values of the registers.

The first time you call show_regs, it clears the screen so you should call it right at the beginning of the program in order to initialize the screen.

This time we want temp2.asm for a template; we will call this program prog3.asm, so make a copy of temp2.asm and give it the new name. Let's take a look at it. The only differences are (1) there are a lot more programs in the EXTRN statements and (2) in the data segment DATASTUFF there are these definitons:

```
ax_byte  db   2
bx_byte  db   2
cx_byte  db   2
etc.
```

These will be used for show_regs later, but you need to learn a few assembler instructions first.

Here's our next program:

TEMP2.ASM

```
+ + + + + + + + START CODE BELOW THIS LINE
    call show_regs

label_one:
    call get_hex
    call show_regs

    call get_num
    call show_regs

    jmp  label_one

+ + + + + + + + END CODE ABOVE THIS LINE
```

That's all the program does. It asks for a number, then calls show_regs to show us what is in the registers. Note that one of the numbers is hex while the other number is decimal.

Compile this, and link it with

```
>link prog3+\asmhelp ;
```

and we're ready to go.

The program reads information in the computer to find out what kind of monitor you have and where the screen output goes. It then puts the register information on the top lines. If it doesn't appear there, we have a screwup somewhere. The text should appear in black and white, but if you have a color monitor you can make it a blue background with white letters.{1}

```
********************** SCREEN SHOT ***************************

    AX  19825                           SI  00000
    BX  00000                           DI  00256
    CX  00255                           BP  17113
    DX  02596                           SP  00508

    CS  0AA4H   DS  0A54H   ES  0A24H   SS  0A34H   IP  0018H

    OF   DF  IEF   TF   SF   ZF   AF   PF   CF
          +    x         +    x         E            COUNT  00003

-----------------------------------------------------------------

The PC Assembler Helper   Version 1.0
Copyright (C) 1989  Chuck Nelson   All rights reserved.
Enter a two byte hex number  4df9
Enter any decimal number  +19825
Enter a two byte hex number


*****************************************************************
```

This is how the screen looks after entering first a hex number, then a decimal number. The numbers in the registers will probably be different for you. Note that AX contains the last number that was entered. On the left side you will see the AX, BX, CX and DX

---

1 To make a blue background and white letters, insert the code "call set_blue" before the FIRST "call show_regs". i.e.:

```
        call  set_blue
        call  show_regs

    label_one:
        etc.
```

This only works if it is allowed by the video board.

registers. For the time being, these registers will display unsigned numbers. On the right are the SI, DI, BP and SP registers. They are also unsigned numbers for the moment. Below that are the segment registers CS, DS, ES, and SS and the instruction pointer (IP). These are hex numbers and will always be hex numbers. The bottom line has OF, DF, IEF, etc. These are the flags, and the marking underneath them (either a blank or some character) tells how they are set. Finally we have COUNT. This has nothing to do with the 8086. It is a counter that is incremented each time you call show_regs.

Keep entering numbers and watch the registers. You will notice that three things are changing - AX, IP and COUNT. AX has the last number you entered and IP keeps changing. Write down the value of IP each time it changes. It goes back and forth between two numbers. That is because you call show_regs in two different places in the loop, {2} and those are two different places in memory where the 8086 is reading the machine code.

Why is AX changing? You may have wondered in prog1.asm and prog2.asm how that information was going back and forth between your program and asmhelp.obj. The answer is that in all the programs in asmhelp.obj, if you need to pass information, it is passed via register AX. This is not the normal way to pass information. The normal way is more elegant but more complicated. We will cover that much later. The counter, of course, increases by 1 each time you call show_regs.

Try entering a few more numbers and then it's time to go on to the next program.

MOVING DATA

Obviously, we want to move data from memory to the 8086, from the 8086 to memory, and between registers. We have the following possibilities:

(1) move from a register to memory
(2) move from memory to a register
(3) move a constant to memory
(4) move a constant to a register
(5) move from one register to another register

That's it. There is no 8086 instruction that moves a single word or a byte from one place in memory to another.

The move mnemonic for the 8086 is "mov".{3} We need some sample data:

---

2 IP actually has three different values, since you call show_regs once before you enter the loop.

3 A mnemonic is a name for a machine instruction, which sounds like what the instruction is supposed to do - MOV for move, SUB for subtract, IMUL for integer multiplication, etc.

EXAMPLE DATA

```
this_year     dw   1989
total         dw   ?
average       dw   ?
time          dw   7

age           db   ?         ; I hope you aren't older than 255
poem          db   "In 1492 Columbus played a mean kazoo."
secret_code   db   3Bh
character     db   ?
```

Here is some sample code. To move from register to memory, we have:

```
        mov  total, ax
        mov  time, si
        mov  age, cl
        mov  character, bh
```

The first thing that strikes the eye is that the destination is on the left and the source is on the right.{4} This is standard for the 8086 instruction set, and it's going to take some getting used to. DESTINATION ON LEFT, SOURCE ON RIGHT. You are going to blow this one from time to time, so always double check to make sure that they are in the right order.

Also note that the 1 byte registers are matched up with 1 byte variables, and 2 byte registers are matched up with 2 byte data. If the sizes don't match, the assembler will complain.{5} Thus:

```
        mov  age, ax
```

is an illegal instruction.

For examples of memory to register moves, we have:

```
        mov  ch, secret_code
        mov  di, this_year
        mov  dl, poem       ; moves 'I' to dl
        mov  bx, time
```

---

4 The computer community likes the words "destination" and "source". "Source" means FROM, and "destination" means TO. The 8086 instruction set is designed:

```
    MOV  TO, FROM
```

which is exactly opposite to the way you would say it in an English sentence. For the 8086, it is always destination on the left, source on the right.

5 Half register and full register operations have different machine codes, and the assembler needs to know which code to use, so the two things must be the same number of bytes.

Once again: (1) DESTINATION ON LEFT, SOURCE ON RIGHT, and (2) the sizes of the two objects must match.

For constant to memory we have:

```
        mov  total, 2986
        mov  age, 36h
        mov  secret_code, 0110100b
        mov  average, (27/5) + 3
```

The arithmetic is done by the assembler and anything that is made up totally of constants is legal. Thus:

```
        mov  average, (((64+27)*51 )/(196-82))
```

is legal but:

```
        mov  average, this_year/time
```

is illegal. The assembler makes either a one byte or a two byte constant to match the size of the destination. The constants for "total" and "average" are two byte constants while those for "age" and "secret_code" are one byte constants. The constants must be within range, that is -129 is too negative for a byte, 256 is too positive for a byte, -32769 is too negative for a word, 65536 is too positive for a word. The assembler will give an error if the constant is out of range.

You can also move a constant to a register:

```
        mov  al, 'c'
        mov  ax, 'c'
        mov  di, 46280
        mov  bl, 99
```

The same rules apply. The constant must be within range and the assembler will make a constant the same size as the destination register (one or two bytes). These constants are actually imbedded in the machine code by the assembler, and are unchangable.

Lastly, you can move data from one register to another:

```
        mov  ax, cx         ; from cx to ax
        mov  ah, bl         ; from bl to ah
        mov  dl, dh         ; from dh to dl
        mov  di, dx         ; from dx to di
```

All this is summarized at the end of the chapter.

It's time for program #4. All this program is going to do is get input and the move it to different registers. We are still using temp2.asm. Here's the program:

TEMP2.ASM

```
;+ + + + + + + + + + START DATA BELOW THIS LINE

byte_data  db ?
word_data  dw ?

;+ + + + + + + + + + END DATA ABOVE THIS LINE

;+ + + + + + + + + + START CODE BELOW THIS LINE

    call show_regs

This_is_a_very_long_label_name:
    call get_hex                        ; (1)
    call show_regs_and_wait

    mov  dx, ax                         ; (2)
    call show_regs_and_wait

    mov  byte_data, al                  ; (3)
    mov  ch, byte_data
    call show_regs_and_wait

    mov  word_data, ax                  ; (4)
    mov  di, word_data
    call show_regs

    jmp  This_is_a_very_long_label_name

;+ + + + + + + + + + END CODE ABOVE THIS LINE
```

There is a data section in this one, so copy those variables into your data section. Here is what the program does. (1) it gets a hex number from the keyboard, (2) it moves the number in ax to dx, (3) it moves one byte from al to ch via the variable byte_data, and (4) it moves two bytes from ax to di via word_data.

There are two different subprograms - show_regs and show_regs_and_wait. They do the same thing except that show_regs_and_wait waits for you to hit the ENTER key before continuing. The computer works so fast that we wouldn't be able to see the changes in the screen if we didn't have a way of pausing. You can use COUNT on the screen to keep track of exactly where you are in the loop.

Assemble program 4, link it to asmhelp.obj, and watch it work. There are two things to notice here. First, we are entering a hex number, but AX is displaying an unsigned number. It is not self-evident that the unsigned number in AX is the same as the hex number that you are entering. Secondly, though CH is changing, there doesn't seem to be any relationship between the number in AX and the number in CX. We will solve both problems in the next chapter.

SUMMARY

MOVING DATA

You can:

(1) move from a register to memory
(2) move from memory to a register
(3) move a constant to memory
(4) move a constant to a register
(5) move from one register to another register

The constants are actual constants which are imbedded in the machine code.

REGISTERS

The normal registers are AX, BX, CX, DX, SI, DI AND BP. AX, BX, CX, and DX can be divided into AH-AL, BH-BL, CH-CL AND DH-DL. The 'H' is for high and the 'L' is for low.
SP is committed and may not be used.
The segment registers are CS, DS, ES, SS.
The instruction pointer (IP) is not available to you.
The register that holds the flags is manipulated by special machine instructions.

ASMHELP.OBJ

Call show_regs to see what is in the 8086 registers. Count increments by one every time you call show_regs.

show_regs_and_wait is the same as show_regs except that it waits for you to hit the ENTER key to allow you time to look at the screen.

Call set_blue at the outset if you have a color card and a color monitor and you want to have a blue background.

get_num gets a signed or unsigned number from the keyboard. (1-5 digits). It does no range checking to see whether the number is too big. All other input routines check to see if a number is too large for its data size.

get_hex gets a hex number from the keyboard. (1-4 digits)

get_ascii gets characters from the keyboard. (1 or 2 characters)

get_binary gets a binary number from the keyboard (1 - 16 digits)

print_num prints a number as hex, signed, unsigned, char, and binary.

All data transfer to or from ASMHELP.OBJ is via the AX register.

CHAPTER 4 - SHOW_REGS

We got started using the program show_regs in the last chapter, but we have already run into problems. The hex number doesn't look the same once we put it in the register - that's because what we are seeing in the arithmetic registers is an unsigned number. Also, when we moved a byte from AL to CH, it was clear that something had moved, but it wasn't clear what the number was. There are two problems here:

(1) We want to use data in hex, binary, ascii, unsigned and signed format depending on what we are doing in the program.

(2) Some of the registers can be used as half registers, so we want a whole register when we need it and a half register when we need it.

Noooooooo problem. There are eight registers whose formats we want to be able to change: AX, BX, CX, DX, SI, DI, BP and SP. We need to give each one a code to tell it what to display. The code is the following:

signed number	1
unsigned number	2
binary number	3
hex number	4
ascii	5

Also, we need to know whether AX, BX, CX and DX are half or full registers. The code for that is:

half register	128	(80 hex)
full register	0	

We will need to do two things - set up the codes, then tell show_regs about the code. We'll begin by setting up the codes.

First let's start with SI, DI, BP and SP. They must be full registers, so the half register information is irrelevant.{1} In the data section is a set of data starting ax_byte, bx_byte ... sp_byte. That is where you need to put the code. Don't change the order of these variables. Just put the correct formatting code in the appropriate byte.

```
mov  si_byte, 3
```

will display SI as a binary number.

---

1 show_regs is very forgiving. It only recognizes half registers where appropriate, and if you screw up on the format code, it just makes it an unsigned number.

```
    mov  bp_byte, 4
```

will display BP as a hex number.

```
    mov  di_byte, 1
```

will display DI as a signed number.

That's pretty easy. If you are using AX, BX, CX, or DX as a full register, they are exactly the same.

```
    mov  dx_byte, 5     ; full register, character format
    mov  ax_byte, 2     ; full register, unsigned format
    mov  bx_byte, 3     ; full register, binary
```

If we use half registers we need to pass more information. show_regs needs to know that it is half registers, not full registers; it also needs to know what the left format is and what the right format is. This is easier than it sounds but will take a little getting used to. The right nibble (half byte) gets the right format and the left nibble (half byte) gets the left format. Then you add 128 to the total. This works easily in hex. 13h means that the left half register is (1 = signed) and the right register is (3 = binary). Then we add (128 = 80h) for a total of 93h. Here are some more examples. Remember, 8 + 1 = 9, 8 + 2 = A, 8 + 3 = B, 8 + 4 = C, 8 + 5 = D

```
    C4h       ; 80h + 44h  left and right are hex
    A5h       ; 80h + 25h  left is unsigned, right is ascii
    B1h       ; 80h + 31h  left is binary, right is signed
    D2h       ; 80h + 52h  left is ascii, right is unsigned
    94h       ; 80h + 14h  left is signed, right is hex
```

There is a summary at the end of the chapter giving all the commands and codes for show_regs. It is important to take some time here and learn to make the registers look the way we want, because later on we have machine instructions for signed numbers, for ascii characters, for binary numbers, and you need to see what the registers look like in the appropriate formats. Spending a little time right now will save you a lot of time later on.

If you don't like using hex numbers you can use decimal numbers:

```
    code = type_of_register + left code + right code
```

where full register = 0, half register = 128d and:

NUMBER FORMAT	LEFT CODE	RIGHT CODE
signed	16d	1d
unsigned	32d	2d
binary	48d	3d
hex	64d	4d
ascii	80d	5d

Therefore, left binary, right hex = 128 + 48 + 4 = 180d. Some

more examples:

```
left ascii, right signed = 128 + 80 + 1 = 209d
left signed, right binary = 128 + 16 + 3 = 147d
left hex, right unsigned = 128 + 64 + 2 = 192d
```

We can put in the code the same way as before.

```
mov  ax_byte, 192d
mov  dx_byte, 147d
mov  cx_byte, 0D2h
mov  bx_byte, 94h
```

We now have the codes in out program, but show_regs doesn't know about them. In order to give the information to show_regs, we call set_reg_style. set_reg_style makes a copy of your information for use by show_regs. The next time that you call show_regs, it will use the new register formats. There is a small problem, however. The information we have is eight bytes long, and ax is only two bytes long. How do we pass the information? The answer is: we don't pass the information. Instead, we pass the address of the information. If you look in the data segment of temp2.asm, you will see that ax_byte is the first byte of this data. We pass the address of ax_byte (the first byte) and set_reg_style knows that that address plus the following 7 addresses have the information that it needs. There is a special machine instruction for putting an address in a register - it is LEA or load effective address. It looks like this:

```
lea  ax, ax_byte
```

This instruction says: put the address of ax_byte in the AX register. Combined with the call, we have:

```
lea  ax, ax_byte
call set_reg_style
```

Before we start writing programs with set_reg_style, we will run a pre-existing program called SETREGS.EXE. Its pathname is \ASMHELP\SETREGS.EXE. It puts the same (pseudo) random number in all arithmetic registers except SP, then requests a formatting code for each register. After cycling through all the registers, it asks you to press ENTER. It then puts a new random number in the registers and starts the cycle again. The hex codes are displayed on the screen before each request. As usual, use Control-C to exit the program.

Here is what the screen might look like after the first cycle. The prseudo random number 2571 will be the same, but your formatting might be different:

```
********************** SCREEN SHOT ***************************
    AX +02571                              SI +02571
    BH  00001010      BL  0B **            DI  02571
    CX  0A 0B **                           BP  0000101000001011
    DH  0A **         DL  0B **            SP  00C4H

    CS  0AA4H   DS  0A42H   ES  0A25H   SS  0A35H   IP  0115H

    OF   DF  IEF   TF   SF   ZF   AF   PF   CF
     x    +    x         +              O    x      COUNT  00009
-----------------------------------------------------------------
hex = C0h or 4h;   ascii = D0h or 5h
Enter a code for sp.
Enter a one byte hex number  4
Press ENTER to continue

*****************************************************************
```

The formats I have used are:

AX	full register	(signed)
BX	half registers	(binary, ascii)
CX	full register	(ascii)
DX	half registers	(ascii, ascii)
SI	full register	(signed)
DI	full register	(unsigned)
BP	full register	(binary)
SP	full register	(hex)

Cycle through the registers a couple of times. If you make them binary, they get longer, if you make them hex or ascii they get shorter; a sign appears if they are signed, and you can change from full to half registers for AX, BX, CX and DX.

You will always be able to tell what kind of number show_regs is printing because (1) a signed number always has a + or - in front of it, (2) a hex number always has an h after it, (3) a binary number is 8 digits long for a half register or 16 digits long for a full register, and (4) an ascii has an asterisk after it. Just as with print_num, if the ascii character has one of the values 0-32, 127 or 255, it will print a hex number and show a double asterisk '**' to signal the event. (5) If none of the above is true, then it is an unsigned decimal number.

If you have a feel for what's happening, it is time to take a mini-test. This is an untimed test, so just make sure that it is correct. I'll give you a particular style, and you figure out the code for that style. The answers are at the bottom of the page. You don't have to memorize the codes. You should be using the summary at the end of the chapter for this quiz.

1. full register, binary
2. half register, left ascii, right hex
3. half register, left signed, right unsigned
4. full register, ascii
5. half register, left binary, right ascii
6. half register, left hex, right signed
7. half register, left unsigned, right binary

8. full register, hex
9. full register, signed
10. half register, left signed, right binary.

If you feel comfortable with what's going on and are able to do set the registers with the help of the summary, we are ready to move on.

Here are the answers.{2}

2 Here are the answers, both in hex and decimal.

PROBLEM	HEX	DECIMAL
1.	3h	3d
2.	D4h	212d
3.	92h	146d
4.	5h	5d
5.	B5h	181d
6.	C1h	193d
7.	A3h	163d
8.	4h	4d
9.	1h	1d
10.	93h	147d

These things are slow to calculate. It took me about a minute per problem to do both the hex and binary.

SUMMARY

The registers may be displayed in signed, unsigned, binary, hex, and ascii formats. The basic codes for this are:

signed	1
unsigned	2
binary	3
hex	4
ascii	5

In addition you need to add the register type. They are:

full register	0
half register	128d or 80h

For the left half register, we have:

FORMAT	LEFT HEX	LEFT DECIMAL
signed	10h	16d
unsigned	20h	32d
binary	30h	48d
hex	40h	64d
ascii	50h	80d

Since the left code is of interest only when the half register type is being used, we simply add 80h and come up with:

FORMAT	LEFT CODE	RIGHT CODE
signed	90h	1h
unsigned	A0h	2h
binary	B0h	3h
hex	C0h	4h
ascii	D0h	5h

Or we add 128d and have:

FORMAT	LEFT CODE	RIGHT CODE
signed	144d	1d
unsigned	160d	2d
binary	176d	3d
hex	192d	4d
ascii	208d	5d

SETTING THE FORMATS

Formats are set by calling set_reg_style. The address of ax_byte must be in AX. The standard assembler instructions for this are:

```
    lea  ax, ax_byte
    call set_reg_style
```

set_reg_style makes a copy of your format data. It changes nothing on the screen. The next time that you call show_regs, it will use the new formatting data.

The correct order for the data in the data segment is:

ax_byte, bx_byte, cx_byte, dx_byte, si_byte, di_byte, bp_byte, sp_byte. They are, of course, all byte sized data.

****

REGISTRATION

Hey, Chuck, I'm no chump!

I'm using your programs/manual, and I want to pay my fair share. Please make me a registered user of "The PC Assembler Tutor" and "The PC Assembler Helper". Enclosed is a check for $9.95 (plus 6.5% tax or $10.60 for California residents). Say, that's cheaper than a large pizza!

Name_____
Last First Initial

Address_____
Street Address

_____
City, State, and Zip Code

I got my copy from _____

Make checks payable to NELSOFT and send your registration to:

NELSOFT
P.O. Box 21389
Oakland, CA 94620

++++++++++

REGISTRATION BENEFITS

As a registered user of "The PC Assembler Helper" and "The PC Assembler Tutor" you are entitled to:

1) Use asmhelp.obj and helpmem.com for personal use.
2) Make 1 (one) printer copy of "The PC Assembler Tutor".
3) Use all programs in "The PC Assembler Tutor" for personal use.
4) Make an archival copy of the disks.
5) Distribute UNALTERED disks to friends for their perusal.
6) Use any updates to either "The PC Assembler Helper" or "The PC Assembler Tutor" under the same registration conditions.

Though copies of the disk may be given away if there is no charge, it is illegal to charge for redistribution of the disk or its contents without permission of the author. Under no circumstances may you distribute printed copies of "The PC Assembler Tutor". If you intend to charge for distributing the disk or its information, please read and sign the distribution agreement which is in INTRO1.DOC.

****

CHAPTER 5 - ADDITION AND SUBTRACTION

The first arithmetic operations we will look at are addition and subtraction, but before we do that, we need to look at one instruction that controls program flow.

LOOP

We already have JMP which sends you to a label:

```
    jmp  label3
```

sends the program to label3, wherever that is in the code. Sometimes we want to repeat a section of code a specific number of times and then go on. For this, we have LOOP. LOOP decrements the CX register by 1. If CX is not zero after being decremented, LOOP jumps to the label indicated. If CX is zero after being decremented, LOOP falls through.

The 8086 does not have general purpose registers. A general purpose register is a register that can be used for ALL instructions. There are a number of instructions on the 8086 which must be done with specific registers, and LOOP is the first one we meet. LOOP always looks at the CX register.

This first program lets you enter a number and then loops that many times so you can watch the CX register. As usual, you exit the program by hitting Control-C. We use temp2.asm.

temp2.asm

```
; - - - - START CODE BELOW THIS LINE
    call show_regs      ; initialize

outer_loop:
    call get_unsigned
    mov  cx, ax         ; number to cx

inner_loop:
    call show_regs_and_wait
    loop inner_loop

    jmp  outer_loop

; - - - - END CODE ABOVE THIS LINE
```

A very simple program. As always, link it with asmhelp.obj. Get_unsigned gets a two byte number (less than 65536) and puts it in AX. We put that number in CX, and then watch the program loop. Make sure you use show_regs_and_wait, or everything will happen too fast for you to see. Try entering 0. On the first pass, loop will decrement CX from 0 to 65535. If CX is 0 when you enter, you

have to repeat the loop 65536 times before you exit the loop. Hit Control-C now to exit.

Throughout the book, I will use label names that end in '_loop' to indicate that they are the destination of a jump or loop instruction. The single word "loop" is a reserved word and may not be used as a label - it can only be used as an instruction.

The addition program will have 4 sections and LOOP will give us the ability to do each section a limited number of times before going on to the next section.

ADDITION

If you read the introductory section on numbers carefully, you know that it is the same instruction for both signed and unsigned addition. The 8086 sets the flags correctly for both signed and unsigned addition. For signed addition, the following flags are set:

OF the overflow flag is set (1) if the result is too negative or too positive, that is, if the result in the register does not show the correct result of the addition. It is cleared (0) otherwise.

ZF the zero flag is 1 if the result is zero, and is cleared (0) if the result is non-zero.

SF the sign flag is set (1) if the result is NEGATIVE and is cleared (0) if the result is POSITIVE. Zero is considered a positive number.

For unsigned addition, the following flags are set:

CF the carry flag is set (1) if the result is too large (over 255 for byte and over 65535 for word operations). It is cleared (0) otherwise.

ZF the zero flag is the same as above.

In addition, there are two more flags (PF the parity flag and AF the auxillary flag) which will be set or cleared; we will learn about them later.

Show_regs shows all the flags. The setting for each flag is underneath its name. For the flags OF, ZF and CF, there is an 'X' if the flag is set and a blank if the flag is cleared. SF, the sign flag, is '-' if the flag is set and '+' if the flag is cleared.

The addition program is fairly long because there are four things to look at - unsigned word addition, unsigned byte addition, signed word addition and signed byte addition. For that reason, it has already been typed in for you. It is called ADD1.ASM and its pathname is \XTRAFILE\ADD1.ASM. Print out a copy of it.

There are four blocks of code which are almost identical except the calls are a little different and two blocks refer to whole registers while the other two refer to half registers. At the head of each block is code to set the appropriate register styles for show_regs. SI, DI, and BP are not used and are set to 0 to make the screen easier to read. Here is the first block of code, which is typical.

```
; - - - CODE
       ; UNSIGNED WORD ADDITION
       mov   ax_byte, 2          ; ax, bx, dx unsigned
       mov   bx_byte, 2
       mov   dx_byte, 2
       lea   ax, ax_byte         ; call set_reg_style
       call  set_reg_style

       mov   cx, 3               ; 3 iterations
unsigned_loop:
       mov   ax, 0      ; clear the registers for visibility
       mov   bx, 0
       mov   dx, 0
       call  show_regs
       call  get_unsigned        ; first number to  ax
       call  show_regs
       push  ax                  ; temporarily save ax
       call  get_unsigned        ; second number to bx
       mov   bx, ax
       pop   ax                  ; get ax back
       mov   dx, ax              ; copy of ax to dx
       add   dx, bx              ; dx (=ax) + bx
       call  show_regs_and_wait
       loop  unsigned_loop

; - - - CODE
```

First, we set AX, BX, and DX for the appropriate register style. Here it is unsigned full register. We then put 3 in CX so we can have 3 iterations with loop. Upon entering the loop, AX, BX, and DX are cleared for reasons of visibility. We don't want the screen cluttered up with numbers. Get_unsigned gets a two byte unsigned number and returns it in AX. We want the first number to be visually on the top (which is AX), but there is a problem here. In order to get the second number we need to call get_unsigned again, and it is going to put another number in AX. We need to temporarily store the first number while we bring in the second number and transfer it to bx.

There is a special 8086 instruction to do this, it is called PUSH. Push temporarily stores a word. The word can be either a full register or a word (two bytes) in memory. You can have either:

```
    variable1 dw   10000

         push ax
         push variable1
```

These are stored in a special place called the stack which we will talk about much later. When you want it back, you use the instruction POP. POP gets back the LAST thing that you pushed onto the stack. Things come off the stack in REVERSE order of how they were put on.

```
push variable1
push variable2
push variable3
push variable4
pop  variable4
pop  variable3
pop  variable2
pop  variable1
```

is the correct order. This is used for temporary storage only, and the only thing which is accessable is the last thing which you PUSHed on the stack.

We push AX to store it temporarily, call get_unsigned again and transfer the number to BX. We then pop AX to get the number back. The situation now is: the first number is in AX, the second number is in BX. For the actual addition, we transfer AX to DX and then add DX and BX. AX and BX contain the two numbers, and DX contains the result. Then you must press ENTER to continue. LOOP will jump to 'unsigned_loop' two times. The third time it will fall through to the next section of code.

This program illustrates a hallmark of assembler code. It normally takes scads of code just to do something simple.

Assemble add1.asm and link it with asmhelp.obj. Run it:

```
******************** SCREEN SHOT ******************************

    AX  17428                           SI  00000
    BX  19755                           DI  00000
    CX  00003                           BP  00000
    DX  37183                           SP  00508

    CS  0AA5H   DS  0A55H   ES  0A25H   SS  0A35H   IP  004DH

    OF   DF  IEF   TF   SF   ZF   AF   PF   CF
     x    +    x         -              E           COUNT  00004
-----------------------------------------------------------------

The PC Assembler Helper   Version 1.0
Copyright (C) 1989  Chuck Nelson   All rights reserved.
Enter a number from 0 to 65535  17428
Enter a number from 0 to 65535  19755
Press ENTER to continue

*****************************************************************
```

This is the screen after the first addition. I have added 17428 (AX) and 19755 (BX). The result 37183 is in DX. CX is still 3 because it hasn't LOOPed yet.

Notice that even though it is the same assembler instruction:

```
add
```

in all four blocks of code, it is doing both signed and unsigned addition correctly. When you are doing signed addition, you want to look at OF, the overflow flag, SF, the sign flag, and ZF, the zero flag after each addition to see how they are set. When you do unsigned addition, you want to look at CF, the carry flag, and ZF, the zero flag to see how they are set. Play around with this for a while, and then it is time for the next program.

As in all 8086 instructions, the order is:

```
add  destination, source
```

We add both numbers, and put the result in the destination, the thing on the left.

There are five different types of addition you can do, (just as there are five different types of moves). They are:

1. add two registers
2. add a register to a variable (memory)
3. add a variable (memory) to a register
4. add a constant to a variable (memory)
5. add a constant to a register

Here's a program that does all 5 things. Use template.asm to make this program. template.asm is almost the same as the other ones we have used. It has a few changes. First, it now lists ALL the subroutines you can call in asmhelp.obj.{1} Appendix 1 (\APPENDIX\APP1.DOC) contains a description of all the subroutines, what they do, and how they are called. Second, the size of STACKSEG is larger. We don't need this large of a stack now; it is for later. Finally, there is a section:

```
; + + + + + + + + + + PUT SUBROUTINES BELOW THIS LINE

; + + + + + + + + + + PUT SUBROUTINES ABOVE THIS LINE
```

for subroutines. Ignore this. This is for later.

From now on, we will always use template.asm unless it is explicitly stated that something else is being used. Here's the program:

template.asm

```
; + + + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
```

---

1 This does not change the size of the .EXE file by even one byte, but it adds a lot of information to the .OBJ file, so they are much larger.

```
variable1    dw     ?
variable2    dw     ?
; + + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       call  show_regs
outer_loop:
       call  get_unsigned        ; first number to ax
       push  ax                  ; store ax
       call  get_unsigned        ; second number to bx
       mov   bx, ax
       pop   ax                  ; restore ax
       mov   variable1, ax       ; first number to variable1
       mov   variable2, bx       ; second number to variable2
       ; add 2 registers
       mov   cx, ax              ; cx + bx
       add   cx, bx
       ; add register to memory
       add   variable1, bx
       mov   dx, variable1       ; put in dx for display
       ; add memory to register
       mov   si, ax
       add   si, variable2
       ; add a constant to memory
       add   variable2, 25
       mov   di, variable2       ; put in di for display
       ; add a constant to a register
       mov   bp, bx
       add   bp, 25
       call show_regs
       jmp   outer_loop

; + + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

The program puts the first number in AX and the second number in BX. It then proceeds to do the same addition (first number plus second number) three times. These are:

1. CX = add two registers (CX + BX)
2. DX = add a register to memory (variable1 + BX)
3. SI = add memory to a register (SI + variable2)

Finally it adds a constant (second number + 25). These are:

4. DI = add a constant to memory (variable2 + 25)
5. BP = add a constant to a register (BP + 25)

On the 8086, it is not possible to add two things in memory. That is:

```
add  variable1, variable2
```

is an illegal instruction. Instead, you need to write:

```
mov  ax, variable2
```

```
    add  variable1, ax
```

SUBTRACTION

It is now time to do some subtraction. The instruction is SUB:

```
    sub  destination, source
```

subtracts source from destination and stores it in destination, the thing on the left.

```
    sub  ax, cx         ; (ax - cx) -> ax
```

In order to do subtraction we are going to modify add1.asm, so make a copy and call it sub1.asm:

```
    >copy  add1.asm  sub1.asm
```

How many instructions do we need to change to modify the program? Four.

```
    add  dx, bx    ->        sub  dx, bx
    add  dl, bl    ->        sub  dl, bl
```

Each of these is changed twice, and we are ready to roll. Assemble it, link it, and run it. Once again we want to look at the flags at the end of each subtraction. For unsigned subtraction, look at ZF, the zero flag, and CF, the carry flag. This time, CF will be set if the result is below zero. For signed subtraction, look at OF, the overflow flag, SF, the sign flag, and ZF, the zero flag. As with addition, subtraction changes PF, the parity flag and AF the auxillary flag. They don't concern us.

As with addition, there are five possibilities for subtraction. They are:

1. subtract one register from another
2. subtract a register from a variable (memory)
3. subtract a variable (memory) from a register
4. subtract a constant from a variable (memory)
5. subtract a constant from a register

the code for these is:

```
    sub  cx, bx         ; (cx - bx)         ->   cx
    sub  variable1, bx  ; (variable1 - bx)  ->   variable1
    sub  si, variable2  ; (si - variable2)  ->   si
    sub  variable2, 25  ; (variable2 - 25)  ->   variable2
    sub  bp, 25         ; (bp - 25)         ->   bp
```

You can copy add2.asm to sub2.asm if you want and change the five ADD instructions to SUB instructions. It will then do those five types of subtraction.

SIGNED AND UNSIGNED NUMBERS

What should you do if you are doing unsigned addition or subtraction and the carry flag gets set? It depends. Sometimes it makes a difference, sometimes it doesn't. If you have an error handling routine, then you can call it with the following code:

```
        add  ax, bx
        jnc  go_on
        call error_handler
    go_on:
```

JC and JNC are conditional jump instructions. JC (jump on carry) jumps if the carry flag is set (1) and JNC (jump on not carry) jumps if the carry flag is not set (0). Using reverse logic here, we skip the error handler if everything is ok.

For signed numbers, it is certainly an error if there is overflow. You are making mathematical calculations and you now have invalid data. One possibility is to do the same as above but with the overflow flag.

```
        add  ax, bx
        jno  go_on
        call error_handler
    go_on:
```

JO and JNO are two more conditional jump instructions. JO (jump on overflow) jumps if the overflow flag is set (1) and JNO (jump on not overflow) jumps if the overflow flag is not set (0). We use the same logic here.

However, there is one special instruction for signed numbers, and that is INTO (interrupt on overflow). It is possible to have an error handler external to your program. It sits permanantly in memory. When you make a signed arithmetic error, INTO interrupts your program and goes to the external error handler. The code looks like this:

```
    add  ax, bx
    into
```

You probably don't have an error handler in your computer right now. In that case, INTO simply goes looking for it and returns when it can't find it.

Let's find out if you have an error handler installed. Once again, use template.asm

```
; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       mov   ax_byte, 1          ; signed register style
       mov   bx_byte, 1
       mov   cx_byte, 1
       lea   ax, ax_byte
       call  set_reg_style
       call  show_regs
```

```
outer_loop:
       call  get_signed
       push  ax
       call  get_signed
       mov   bx, ax
       pop   ax
       mov   cx, ax
       add   cx, bx
       into
       call  show_regs
       jmp   outer_loop
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

This is basically the same thing as before, but using AX, BX, and CX. They are set for signed style, and then we get two signed numbers and add them. The result is in CX. Right after the addition instruction is INTO. If the result is too positive or too negative, OF will be set and INTO will look for the error handler.

Assemble this program and link it with asmhelp.obj. Try both numbers that do not set the overflow flag and numbers that do set the overflow flag. Did anything different happen when the overflow flag was set? If nothing different happened, you don't have an error handler for INTO.

Included on the disks is an error handler. It is called INTO.COM, and it's pathname is \XTRAFILE\INTO.COM. When you run it:

```
>into
```

it will install itself and then return to the command prompt:

```
>
```

INTO.COM will stay in memory until you reboot or shut off the machine. INTO.COM provides the type of sophisticated error handling that you might want to use in a real program. Install (run) INTO.COM, and then try the previous program again, both with numbers that cause an overflow and numbers that don't cause an overflow.

SUMMARY

ADD performs both signed and unsigned addition. It can:

1. add two registers
2. add a register to a variable (memory)
3. add a variable (memory) to a register
4. add a constant to a variable (memory)
5. add a constant to a register

SUB performs both signed and unsigned subtraction. It can:

1. subtract one register from another
2. subtract a register from a variable (memory)
3. subtract a variable (memory) from a register
4. subtract a constant from a variable (memory)
5. subtract a constant from a register

The flags affected by both ADD and SUB are:

CF the carry flag (for unsigned). Set if the 0/65535 (0/255) border was crossed.

ZF the zero flag (for signed and unsigned). Set if the result is 0.

SF the sign flag (for signed). Set if the result is negative.

OF the overflow flag (for signed). Set if the result was too negative or too positive.

PF the parity flag and AF, the auxillary flag

The following jump instructions are conditional on the setting of the flags:

JC jump on carry, JNC, jump on not carry
JO jump on overflow. JNO, jump on not overflow

LOOP

LOOP decrements cx by 1. If cx is then not zero, it jumps to the named label. If cx is zero, it falls through to the next instruction.

INTO

If the overflow flag is set, INTO (interrupt on overflow) interrupts the program and goes to an external error handler if one exists. It returns immediately if one doesn't exist.

PUSH and POP

PUSH stores either a register or a word (in memory) in a temporary storage area. POP retrieves the last word PUSHed.

#

CHAPTER 6 - MULTIPLICATION AND DIVISION

Unlike addition and subtraction, where the result can be in either memory or in one of the registers, the multiplication and division instructions have a rigid format.

MULTIPLICATION

You can multiply a one byte number by a one byte number and get a two byte result, or you can multiply a one word number by a one word number and get a two word result. The first number MUST be in AL for the byte operation or in AX for the word operation. The second number may be a register or a memory location (but not a constant). The result is in AH:AL for the byte operation and DX:AX for the word operation. Our possibilities are:

```
AL  X  (one byte register or memory)  ->  AH:AL
AX  X  (one word register or memory)  ->  DX:AX
```

Is there a difference between signed and unsigned numbers? Yes, a very big difference. For the byte operation FFh = -1 signed but FFh = 255 unsigned. -1 X -1 = 1 = 0001h. 255 X 255 = 65025 = FE01h. These are two completely different answers. You need to tell the 8086 whether you want signed multiplication or unsigned multiplication. The 8086 does the rest. Let's look at both signed and unsigned multiplication. We'll do byte multiplication for unsigned numbers and word multiplication for signed numbers. The instruction for unsigned multiplication is MUL. The instruction for signed multiplication is IMUL. AX or AL is understood to be the register, so it is not in the code. The instructions are:

```
variable1  db ?
variable2  dw ?

     mul  bx             ; unsigned word from a register
     mul  variable1      ; unsigned byte from memory
     imul ch             ; signed byte from a register
     imul variable2      ; signed word from memory
```

No AX or AL. It's understood. Here's our program:

```
template.asm
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
answer1      dw     ?
answer2      dw     ?
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; - - - - - START CODE BELOW THIS LINE

       mov     cx, 0              ; clear cx for visual effect
```

```
outer_loop:
       ; unsigned byte multiplication
       mov   ax_byte, 0A2h          ; half regs, unsigned
       mov   bx_byte, 0A2h          ; half regs, unsigned
       lea   ax, ax_byte
       call  set_reg_style

       mov   ax, 0               ; clear regs
       mov   bx, 0
       mov   dx, 0
       call  show_regs

       call  get_unsigned_byte    ; get two unsigned bytes
       call  show_regs
       push  ax                   ; save the first number
       call  get_unsigned_byte
       mov   bl, al
       pop   ax                   ; restore the first number
       call  show_regs_and_wait
       mul   bl                   ; unsigned multiplication
       call  print_unsigned       ; display the result (ax)
       call  show_regs_and_wait


       ; signed word multiplication
       mov   ax_byte, 01h          ; full reg, signed
       mov   bx_byte, 01h
       mov   dx_byte, 01h
       lea   ax, ax_byte
       call  set_reg_style

       mov   ax, 0               ; clear regs
       mov   bx, 0
       call  show_regs

       call  get_signed           ; get two numbers
       call  show_regs
       push  ax                   ; save the first number
       call  get_signed
       mov   bx, ax
       pop   ax                   ; restore the first number
       call  show_regs_and_wait
       imul  bx                   ; signed multiplication
       push  ax                   ; save result
       mov   answer1, ax          ; display 4 byte result
       mov   answer2, dx
       lea   ax, answer1
       call  print_signed_4byte
       pop   ax                   ; restore result
       call  show_regs_and_wait

      jmp    outer_loop
; - - - - - END CODE ABOVE THIS LINE
```

If the answer for the unsigned byte multiplication is greater than 255, it will be difficult to read the answer from the half

registers, so we print out the whole AX register.

If the answer for the signed word multiplication is greater than +32767 or is less than -32768, the answer will be unreadable in the DX:AX registers. We move the answer to memory, and then call print_signed_4byte. As with set_reg_style, the data is too long to be put in AX, so we pass the address of the first byte of data with:

```
    lea  ax, answer1
```

and then call print_signed_4byte. Everything from PUSH AX to POP AX is designed to do that.

Do MUL and IMUL set any flags? Yes. For byte multiplication, if AL contains the total answer, the 8086 clears the OF and CF flags. If part of the answer is in AH, then the 8086 sets both the OF and CF flags. For word multiplication, if AX contains the total answer, the 8086 clears the OF and CF flags. If part of the answer is in DX, then the 8086 sets both the OF and CF flags.

What do we mean by the total answer? This is simple for unsigned multiplication. If AH (or DX for word) is 0, then the total answer is in AL (or AX for word). It is more complicated for signed multiplication. Consider word multiplication. +30000 X +2 = +60000. But that's less than 65536, so it is completely contained in AX, right? WRONG. The leftmost bit of AX contains the sign. If the signed result is out of the range -32768 to +32767, information about the absolute value of the number is corrupting information about the sign of the number. AX will have the wrong number and the wrong sign. Only by combining AX with DX will you get the correct answer. Similarly for byte multiplication with AL, if the result is not in the range -128 to +127, The leftmost (sign) bit will be corrupted, and only by looking at AH:AL will you be able to get the correct result.

If CF and OF are set, you need to look at both registers to evaluate the number. You might want to do error handling, so once again, you can have:

```
        mul  bx
        jnc  go_on
        call error_handler
    go_on:
```

using the same reverse logic as before (if nothing is wrong, skip the error handler). We can also use:

```
        mul  bx
        into
```

if there is an INTO error handler.

DIVISION

Division operates in the same way as multiplication. Word

division operates on the DX:AX pair and byte division operates on the AH:AL pair. There are two instructions, DIV for unsigned division and IDIV for signed division. After the division:

```
byte      AL = quotient, AH = remainder
word      AX = quotient, DX = remainder
```

Both DIV and IDIV operate on BOTH registers. For bytes, they consider AH:AL a single number. This means that AH must be set correctly before the division or you will get an incorrect answer. For words, they consider DX:AX a single number. This means that DX must be set correctly before the division, or the result will be incorrect. Why did Intel include AH and DX in the division? Wouldn't it have been easier to use just AH (or AX for word division) and put the quotient and remainder in the same place? These instructions are actually designed for dividing a long number (4 or 8 bytes). How it works is pretty slick; you'll find out about it later in the book.

How do you set AH and DX correctly? For unsigned numbers, that's easy. Make them 0:

```
mov  al, variable
mov  ah, 0
div  cl        ; unsigned byte division
```

For signed division, set AH or DX to 0 (0000h) if it is a positive number and set them to -1 (FFFFh) if the number is negative. This is just standard sign extension that was covered in the chapter on numbers. Fortunately for us, Intel has provided instructions which do the sign extension for us. CBW (convert byte to word) correctly extends the signed number in AL through AH:AL. CWD (convert word to double) correctly extends the signed number in AX through DX:AX. The code is

```
mov  ax, variable5
cwd
idiv bx             ; signed word division
```

Of course with these two instructions you can convert a byte to a double word.

```
mov  al, variable6
cbw
cwd
idiv bx             ; signed word division
```

first converting to a word, then to a double word.

For the division program, we are going to use the multiplication program and make some small changes. Make a copy of your multiplication program:

```
>copy mult.asm  div.asm
```

and then make the following changes:

MULTIPLICATION	DIVISION
`; unsigned byte`	`; unsigned byte`
	`pop  ax`
`pop  ax`	`mov  ah, 0`
`call show_regs_and_wait`	`call show_regs_and_wait`
`mul  bl`	`div  bl`
`; signed word`	`; signed word`
	`pop  ax`
`pop  ax`	`cwd`
`call show_regs_and_wait`	`call show_regs_and_wait`
`imul bx`	`idiv bx`

The calls to print_unsigned and print_signed_4byte are irrelevant, so you may either delete them or ignore the output. All we did was change the multiplication instruction to division and prepare the upper register correctly (AH for byte, DX for word). That's all.

Assemble, link, and run it. Try out both positive and negative numbers and see what the remainder looks like. Also notice the sign extension just before the division. Remember, for division, the results are in the following places:

```
byte      AL = quotient, AH = remainder
word      AX = quotient, DX = remainder
```

Now divide by 0. Ka-pow! You should have exited the program and gotten an error message. Unlike the other arithmetical errors where you have the option of ignoring them or making an error handler for them, the 8086 considers division by 0 a major no-no. When the 8086 detects division by zero,{1} it interrupts the program and goes to the zero-divide handler (which is external to the program). Normally, this just exits the program since the data is now worthless.

---

1 What it actually detects is that the quotient is too large to fit in the lower register (AL for byte or AX for word). As long as the upper register is correctly sign extended, the only time this can happen is when you divide by 0. If the upper register is NOT sign extended correctly, you can have zero divide errors all over the place, even though you aren't dividing by 0. As an example, if AH:AL contain 3275 and bl contains 10, then:

```
div  bl
```

will give a quotient of 327 (> 255) and will generate a zero divide error.

SUMMARY

MUL and IMUL

MUL performs unsigned multiplication and IMUL performs signed multiplication. For bytes, the multiplicand is in AL and the result is in the AH:AL pair. For words, the multiplicand is in AX and the result is in the DX:AX pair. If the total result is contained in the lower register, CF and OF are cleared (0). If part of the result is in the upper register, CF and OF are set (1). The multiplier may be either a register or a variable in memory.

```
variable1 db ?
variable2 dw ?

     mul  variable1      ; unsigned byte
     mul  cx             ; unsigned word
     imul bl             ; signed byte
     imul variable2      ; signed word
```

DIV and IDIV

DIV performs unsigned division. IDIV performs signed division. For bytes, the dividend is the AH:AL pair. For words, the dividend is the DX:AX pair. In byte division, AH must be correctly prepared before the division. For word division, DX must be correctly prepared before the division. The divisor may be either a register or a variable in memory.

```
variable1 db ?
variable2 dw ?

     div  variable1      ; unsigned byte
     div  cx             ; unsigned word
     idiv bl             ; signed byte
     idiv variable2      ; signed word
```

The quotient and remainder are as follows:

```
byte      AL = quotient, AH = remainder
word      AX = quotient, DX = remainder
```

No flags are affected. If the quotient is too large for the lower register, or if you divide by zero, a zero divide program interrupt occurs.

CORRECT SIGN EXTENSION

To prepare for division, you must correctly sign extend the lower register into the upper register. For unsigned division, zero the upper register (AH = 0 or DX = 0). For signed division, use CBW and CWD. CBW (convert byte to word) extends a signed number in AL through AH:AL. CWD (convert word to double) extends a signed number in AX through DX:AX

CHAPTER 7 - LOGIC

There are a number of operations which work on individual bits of a byte or word. Before we start working on them, it is necessary for you to learn the Intel method of numbering bits. Intel starts with the low order bit, which is #0, and numbers to the left. If you look at a byte:

```
7 6 5 4 3 2 1 0
```

that will be the ordering. If you look at a word:

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

that is the ordering. The overwhelming advantage of this is that if you extend a number, the numbering system stays the same. That means that if you take the number 45 :

```
7 6 5 4 3 2 1 0
0 0 1 0 1 1 0 1  (45d)
```

and sign extend it:

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0  0  0  0  0  0  0  0  0  0  1  0  1  1  0  1
```

each of the bits keeps its previous numbering. The same is true for negative numbers. Here's -73:

```
7 6 5 4 3 2 1 0
1 0 1 1 0 1 1 1 (-73d)

15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 1  1  1  1  1  1  1  1  1  0  1  1  0  1  1  1  (-73d)
```

In addition, the bit-position number denotes the power of 2 that it represents. Bit 7 = 2 ** 7 = 128, bit 5 = 2 ** 5 = 32, bit 0 = 2 ** 0 = 1. {1}.

Whenever a bit is mentioned by number, e.g. bit 5, this is what is being talked about.

AND

We will use AND as the prototype. There are five different ways you can AND two numbers:

---

1 I'm using the Fortran convention for showing exponents. That is, 2 ** 7 is 2 to the 7th, 3 ** 19 is 3 to the 19th.

---

1. AND two register
2. AND a register with a variable
3 AND a variable with a register
4. AND a register with a constant
5. AND a variable with a constant

That is:

```
variable1 db   ?
variable2 dw   ?

and  cl, dh
and  al, variable1
and  variable2, si
and  dl, 0C2h
and  variable1, 01001011b
```

You will notice that this time the constants are expressed in hex and binary. These are the only two reasonable alternatives. These instructions work bit by bit, and hex and binary are the only two ways of displaying a number bitwise (bit by bit). Of course, with hex you must still convert a hex digit into four binary digits.

The table of bitwise actions for AND is:

```
1    1    ->   1
1    0    ->   0
0    1    ->   0
0    0    ->   0
```

That is, a bit in the result will be set if and only if that bit is set in both the source and the destination. What is this used for? Several things. First, if you AND a register with itself, you can check for zero.

```
and  cx, cx
```

If any bit is set, then there will be a bit set in the result and the zero flag will be cleared. If no bit is set, there will be no bit set in the result, and the zero flag will be set. No bit will be altered, and CX will be unchanged. This is the standard way of checking for zero. You can't AND a variable that way:

```
and  variable1, variable1
```

is an illegal instruction. But you can AND it with a constant with all the bits set:

```
and  variable1, 11111111b
```

If the bit is set in variable1, then it will be set in the result. If it is not set in variable1, then it won't be set in the result. This also sets the zero flag without changing the variable.

AND is also used in masks, which will be covered at the end of the chapter.

Finally, there is a variant of AND called TEST. TEST does exactly the same thing as AND but throws away the results when it is done. It does not change the destination. This means that it can check for specific things without altering the data. It has the same possibilities as AND:

```
variable1 db   ?
variable2 dw   ?

test cl, dh
test al, variable1
test variable2, si
test dl, 0C2h
test variable1, 01001011b
```

will set the flags exactly the same as the similar AND instructions but will not change the destination. We need a concrete example, and for that we'll turn to your video card. In text mode, your screen is 80 X 25. That is 2000 cells. Each cell has a character byte and an attribute byte. The character byte has the actual ascii number of the character. The attribute byte says what color the character is, what color the background is, whether the character is high or low intensity and whether it blinks. An attribute byte looks like this:

```
7 6 5 4 3 2 1 0
X R G B I R G B
```

Bits 0,1 and 2 are the foreground (character) color. 0 is blue, 1 is green, and 2 is red. Bits 4, 5, and 6 are the background color. 4 is blue, 5 is green, and 6 is red. Bit 3 is high intensity, and bit 7 is blinking. If the bit is set (1) that particular component is activated, if the bit is cleared (0), that component is deactivated.

The first thing to notice is how much memory we have saved by putting all this information together. It would have been possible to use a byte for each one of these characteristics, but that would have required 8 X 2000 bytes = 16000 bytes. If you add the 2000 bytes for the characters themselves, that would be 18000 bytes. As it is, we get away with 4000 bytes, a savings of over 75%. Since there are four different screens (pages) on a color card, that is 18000 X 4 = 72000 bytes compared to 4000 X 4 = 16000. That is a huge savings.

We don't have the tools to access these bytes yet, but let's pretend that we have moved an attribute byte into dl. We can find out if any particular bit is set. TEST dl with a specific bit pattern. If the zero flag is cleared, the result is not zero so the bit was on. If the zero flag is set, the result is zero so that bit was off

```
test dl, 10000000b       ; is it blinking?
test dl, 00010000b       ; is there blue in the background?
test dl, 00000100b       ; is there red in the foreground?
```

If we look at the zero flag, this will tell us if that component is on. It won't tell us if the background is blue, because maybe the green or the red is on too. Remember, test alters neither the source nor the destination. Its purpose is to set the flags, and the results go into the Great Bit Bucket in the Sky.

OR

The table for OR is:

```
1    1    ->   1
1    0    ->   1
0    1    ->   1
0    0    ->   0
```

If either the source or the destination bit is set, then the result bit is set. If both are zero then the result is zero. OR is used to turn on a specific bit.

```
or   dl, 10000000b  ; turn on blinking
or   dl, 00000001b  ; turn on blue foreground
```

After this operation, those bits will be on whether or not they were on before. It changes none of the bits where there is a 0. They stay the same as before.

XOR

The table for XOR is:

```
1    1    ->   0
1    0    ->   1
0    1    ->   1
0    0    ->   0
```

That is, if both are on or if both are off, then the result is zero. If only one bit is on, then the result is 1. This is used to toggle a bit off and on.

```
xor  dl, 10000000b  ; toggle blinking
xor  dl, 00000001b  ; toggle blue foreground
```

Where there is a 1, it will reverse the setting. Where there is a 0, the setting will stay the same. This leads to one of the favorite pieces of code for programmers.

```
xor  ax, ax
```

zeros the ax register. There are three ways to zero the ax register:

```
mov  ax, 0
sub  ax, ax
xor  ax, ax
```

The first one is very clear, but slightly slower. For the second one, if you subtract a number from itself, you always get zero. This is slightly faster and fairly clear.{2} For the third one, any bit that is 1 will become 0, and and bit that is 0 will stay 0. It zeros the register as a side effect of the XOR instruction. You'll never guess which one many programmers prefer. That's right, XOR. Many programmers prefer the third because it helps make the code more obsure and unreadable. That gives a certain aura of technical complexity to the code.

NEG and NOT

NOT is a logical operation and NEG is an arithmetical operation. We'll do both here so you can see the difference. NOT toggles the value of each individual bit:

```
1    ->   0
0    ->   1
```

NEG negates the value of the register or variable (a signed operation). NEG performs (0 - number) so:

```
neg  ax
neg  variable1
```

are equivalent to (0 - AX) and (0 - variable1) respectively. NEG sets the flags in the same way as (0 - number).

```
; negvsnot.asm
; compares the operations NEG and NOT
; + + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       mov   ax_byte, 1          ; signed
       mov   bx_byte, 1          ; signed
       mov   cx_byte, 1          ; signed
       mov   si_byte, 3          ; binary
       mov   di_byte, 3          ; binary
       mov   bp_byte, 3          ; binary
       mov   dx, 0               ; not used, so clear
       lea   ax, ax_byte
       call  set_reg_style
       call  show_regs

outer_loop:
       call  get_signed          ; get number
       mov   bx, ax              ; move it to all registers
       mov   cx, ax
       mov   si, ax
       mov   di, ax
       mov   bp, ax

       not   bx                  ; NOT the second row down
```

2 This is one of the first instructions in the template files.

```
        not   di
        neg   cx                    ; NEG the third row down
        neg   bp

        call  show_regs
        jmp   outer_loop

; + + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

This is set up so the left registers are signed and the right registers are binary. The top registers retain the original value, the second registers down will do NOT and the third registers down will do NEG.

Put a number in. On the right side, you will see that NOT (in DI) reverses the bit pattern, while on the left, NEG (in CX) negates the number. Do a few more. This is always true. But they seem to be related. In fact, BX will always be 1 too negative. Why? Remember that in the introduction on numbers, when we changed signs, we had:

```
    negative of number = one's complement + 1
```

but the one's complement is exactly NEG. It switches the value of each bit. To get the values in the third row (CX and BP), simply add 1 to the values in the second row (BX and DI). Remember, NOT is a logical operation, NEG is an arithmetic operation.

MASKS

To explain masks, we'll need some data, and we'll use the attribute byte for the monitor. Here it is again:

```
    7 6 5 4 3 2 1 0
    X R G B I R G B
```

Bits 0,1 and 2 are the foreground (character) color. 0 is blue, 1 is green, and 2 is red. Bits 4, 5, and 6 are the background color. 4 is blue, 5 is green, and 6 is red. Bit 3 is high intensity, and bit 7 is blinking.

What we want to do is turn certain bits on and off without affecting other bits. What if we want to make the background black without changing anything else? We use and AND mask.

```
    and  video_byte, 10001111b
```

Bits 0, 1, 2, 3 and 7 will remain unchanged, while bits 4, 5 and 6 will be zeroed. This will make the background black. What if we wanted to make the background blue? This is a two step process. First we make the background black, then set the blue background bit. This involves first the AND mask, then an OR mask.

```
    and  video_byte, 10001111b
    or   video_byte, 00010000b
```

The first instruction shuts off certain bits without changing others. The second turns on certain bits without effecting others. The binary constant that we are using is called a mask. You may write this constant as a binary or a hex number. You should never write it as a signed or unsigned number (unless you are one of those people who just adores making code unreadable).

If you want to turn off certain bits in a piece of data, use an AND mask. The bits that you want left alone should be set to 1, the bits that you want zeroed should be set to 0. Then AND the mask with the data.

If you want to turn on certain bits in a piece of data, use an OR mask. The bits that you want left alone should be set to 0. The bits that you want turned on should be set to 1. Then OR the mask with the data.

Go back to AND and OR to make sure you believe that this is what will happen.

SUMMARY

For AND, TEST, OR, and XOR you can have the following combinations:

1. two register
2. a register with a variable
3 a variable with a register
4. a register with a constant
5. a variable with a constant

AND is a bitwise logical operation.

```
1    1    ->   1
1    0    ->   0
0    1    ->   0
0    0    ->   0
```

TEST does the same thing as AND except that the result is discarded. It is used for setting the flags without altering the data.

OR is a bitwise logical operation.

```
1    1    ->   1
1    0    ->   1
0    1    ->   1
0    0    ->   0
```

XOR is a bitwise logical operation.

```
1    1    ->   0
1    0    ->   1
0    1    ->   1
0    0    ->   0
```

You can use NOT and NEG with either a register or a variable in memory.

NOT is a bitwise logical operation

```
0    ->   1
1    ->   0
```

NEG is an arithmetic operation. NUMBER -> - NUMBER. It gives the negative of a signed number.

MASKS

If you want to turn off certain bits of a piece of data, use an AND mask. The bits that you want left alone should be set to 1, the bits that you want zeroed should be set to 0. Then

AND the mask with the data.

If you want to turn on certain bits of a piece of data, use an OR mask. The bits that you want left alone should be set to 0. The bits that you want turned on should be set to 1. Then OR the mask with the data.

CHAPTER 8 - SHIFT AND ROTATE

There are seven instructions that move the individual bits of a byte or word either left or right. Each instruction works slightly differently. We'll make a standard program and then substitute each instruction into that program.

SAL - SHL

The instructions SHL (shift logical left) and SAL (shift arithmetic left) are exactly the same. They have the same machine code. They shift each bit to the left. How far? That depends. There are two (and only two) forms of this instruction. All other shift and rotate instructions have these two (and only these two) forms as well. The first form is:

```
    shl  al, 1
```

Which shifts each bit to the left one bit. The number MUST be 1. No other number is possible. The other form is:

```
    shl  al, cl
```

shifts the bits in AL to the left by the number in CL. If CL = 3, it shifts left by 3. If CL = 7, it shifts left by 7. The count register MUST be CL (not CX). The bits on the left are shifted out of the register into the bit bucket, and zeros are inserted on the right. The easy way to understand this is to fire up the standard program. Remember, from now on we always use template.asm.

```
;sal.asm
; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       mov   ax_byte, 0A3h       ; half reg, low reg binary
       mov   bx_byte, 0A4h       ; half reg, low reg hex
       mov   cx_byte, 0A1h       ; half reg, low reg signed
       mov   dx_byte, 0A2h       ; half reg, low reg unsigned
       lea   ax, ax_byte
       call  set_reg_style

       mov   ax, 0               ; clear registers
       mov   bx, 0
       mov   cx, 0
       mov   dx, 0
       mov   di, 0
       mov   bp, 0
       call  show_regs

outer_loop:
       call  get_hex_byte   ; get number and put in registers
       mov   bl, al
       mov   cl, al
```

```
       mov   dl, al
       mov   si, 8          ; 8 iterations of the loop
       and   al, al         ; set the flags
       call  show_regs_and_wait
shift_loop:
       sal   al, 1
       sal   bl, 1
       sal   cl, 1
       sal   dl, 1
       call  show_regs_and_wait
       dec   si
       jnz   shift_loop
       jmp   outer_loop

; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

This standard program is with bytes, not words. This is because if we had used words we would have performed 16 individual shifts and that would have been time consuming and boring. First we set the style to half registers. Notice that one is binary, one is hex, one is signed and one is unsigned. That covers all bases. All the registers are then cleared. It would be nice to use the loop instruction, but CX is committed, so we make our own loop instruction. We move 8 into SI. The loop instructions are:

```
    dec  si
    jnz  shift_loop
```

DEC decrements a register or a variable by 1. Its counterpart INC increments a register or variable by 1. JNZ (jump if not zero) jumps to 'shift_loop' if SI is not zero.

We get a hex byte in AL and put the same byte in BL, CL, and DL. This way we will be able to see what is happening in binary, hex, signed and unsigned. Before starting, we have:

```
    and  al, al
```

This is there to set the flags correctly before starting. All four are shifted left one bit each time, and then we look at the result.

Assemble, link and run it. Enter the number 7. In binary, that is (0000 0111). Take a look at the flags before starting. It is a positive number so SF shows '+'. ZF is not set. PF shows 'O'. O stands for odd. Every time you perform an arithmetic or logical operation, the 8086 checks parity. Parity is whether the number contains an even or odd number of 1 bits. This contains 3 1 bits, so the parity is odd. The possible settings are 'E' for even and 'O' for odd.{1} SAL checks for parity (though some of the other instructions don't). Now press ENTER. It will shift left 1 and you will have (0000 1110). What does the unsigned number say now? 14. Press ENTER again. (0001 1100) What does the unsigned number say? 28. Again (0011 1000) 56. Again (0111 0000) 112. Notice that

---

1 This is for use by communications programs.

the signed number reads +112. Look at the CF and OF. They are both cleared. Things are going to change now. Press ENTER again. (1110 0000). SF is now '-'. OF, the overflow flag is set because you changed the number from positive to negative (from +112 to -32). What is the unsigned number now? 224. CF is cleared. PF is '0'. Shift again. (1100 0000) OF is cleared because you didn't change signs. (Remember, the leftmost bit is the sign bit for a signed number). PF is now 'E' because you have two 1 bits, and two is even. CF is set because you shifted a 1 bit off the left end. Keep pressing ENTER and watch SF, OF, CF, and PF.

Let's look at the unsigned numbers we had until we started shifting 1 bits off the left end. We started with 7, then had 14, 28, 56, 112, 224. This instruction is multiplying by 2. That's right, and it is MUCH faster than multiplication (about 50 times faster). Far and away the fastest way to multiply a register by 2, 4 or 8 is to use sal.

```
; by 2              ;by 4               ; by 8
sal  di,1           sal  di, 1          sal  di, 1
                    sal  di, 1          sal  di, 1
                                        sal  di, 1
```

For a register, it is faster to use a series of 1 shifts than to load cl. For a variable in memory, anything over 1 shift is faster if you load cl.

Do a few more numbers to see what is happening both with the number and the flags. CF always signals when a 1 bit has been shifted off the end.

SAR and SHR

Unlike the left shift instruction, there are two completely different right shift instructions. SHR (shift logical right) shifts the bits to the right, setting CF if a 1 bit is pushed off the right end. It puts 0s in the leftmost bit. Make a copy of SAL.ASM and replace the four instructions:

```
sal   al, 1
sal   bl, 1
sal   cl, 1
sal   dl, 1
```

with SHR. We'll call the new program SHR.ASM. Run this one too. Instead of 7, use E0h (1110 0000) which is 224d. The first time you shift (0111 0000) the OF flag will be set because the sign changed. Keep shifting, noting the flags and the unsigned number. This time we have 224, 112, 56, 28, 14, 7, 3, 1. It is dividing by two and is once again MUCH faster than division. For a single shift, the remainder is in CF. For a shift of more than one bit, you lose the remainder, but there is a way around this which we will discuss in a moment. Do some more numbers till you are comfortable with the flags and the operation.

If you want to divide by 16, you will shift right four times, so

you'll lose those 4 bits. But those bits are exactly the value of the remainder. All we need to do is:

```
    mov  dx, ax    ; copy of number to dx
    and  dx, 0000000000001111b ; remainder in dx
    mov  cl, 4     ; shift right 4 bits
    shr  ax, cl    ; quotient in ax
```

Using a mask, we keep only the right four bits, which is the remainder.

SAR

SAR (shift arithmetic right) is different. It shifts right like SHR, but the leftmost bit always stays the same. This will make more sense when you run the program. Make another copy, call it SAR.ASM, and change the four instructions to SAR. The flags operate the same as for SHR and SHL. The overflow flag will never change since the left bit will always stay the same.

First enter 74h (+116). We will be looking at the signed numbers only. Copy down the signed numbers as you go along. They should be: 116, 58, 29, 14, 7, 3, 1, 0, 0. Now try 8Ch (-116). The numbers you should get are: -116, -58, -29, -15, -8, -4, -2, -1, -1. They started out the same, then they got off by one. The negative numbers are one too negative. Try 39h (+57). The numbers here are: 57, 28, 14, 7, 3, 1, 0, 0, 0. Just as it should be for division by 2. Now try C7 (-57). Here the numbers are: -57, -29, -15, -8, -4, -2, -1, -1, -1. This time it went screwy right off the bat. Once again, the negative numbers are one too negative.

SAR is an instruction for doing signed division by 2 (sort of). It is, however, an incomplete instruction. The rule for SAR is: SAR gives the correct answer if the number is positive. It gives the correct answer if the number is negative and the remainder is zero. If the number is negative but there is a remainder, then the answer is one too negative. The reason for this is a little complex, but we need to add some code if we want to do signed division.{2} For SHR, the remainder part was optional. Here it is not. We need to know whether the remainder is zero or not. For this example we will do a word shift left by 6. That's dividing by 64.

```
    remainder_mask dw   002Fh     ; 63

    call get_signed          ; number in ax
    mov  bx, ax              ; copy in bx
    and  bx, remainder_mask  ; the remainder
    mov  cl,6                ; shift right 6 bits
    sar  ax, cl
    jns  continue            ; is it positive?
```

2 Both the code and the reasons will be explained (but not proved) in the summary.

```
    and  bx, bx             ; is the remainder zero?
    jz   continue
    inc  ax
continue:
```

We get the remainder, then shift right 6 bits. Upon finishing SAR, the sign flag will be set correctly. Here is yet another jump. This one is JNS (jump on not sign) jumps if the sign flag is NOT set, that is if the number is positive. If it is positive, then everything is ok so we skip ahead. If the number is negative, then we check to see if there was a remainder. If there wasn't, everything is ok, so we go ahead. If there was a remainder, then we INC (add 1) ax.

Is the remainder correct? If the number was positive, the remainder is correct, but if the number was negative, then we need to do one more thing. After INC, but before 'continue' we have a SUB instruction:

```
        inc  ax
        sub  bx, 64    ; correct the remainder
    continue:
```

Why that is the correct number will be explained in the summary. What a lot of work when we could simply write:

```
    mov  cx, 64
    call get_signed
    cwd                ; sign extend
    idiv cx            ; signed division
```

Is there any advantage to this instruction? Not really. Remember that the more you shift, the longer it takes. If you shift 2, then it's about 1/3 faster than division. If you shift 14, then it is only 15% faster than division. Considering that even a slow PC can do 25000 divisions a second, you must be in serious need of speed to use this. In any case, you will never or almost never use SAR for signed division, while you will find lots of opportunity to use SHR and SHL for unsigned multiplication and division.

ROR and ROL

ROR (rotate right) and ROL (rotate left) rotate the bits around the register. We will just do one program since they operate the same way, only in opposite directions. Make another copy of SAL.ASM and put in ROR in the appropriate spots.

Enter a number. This time you will notice that the bits, rather than dissapearing off the end, reappear on the other side. They rotate around the register. The only flags that are defined are OF and CF. OF is set if the high bit changes, and CF is set if a 1 bit moves off the end of the register to the other side. Do a few more, and we'll go on to the last two instructions.

RCR and RCL

RCR (rotate through carry right) and RCL (rotate through carry left) rotate the same as the above instructions except that the carry flag is involved. Rotating right, the low bit moves to CF, the carry flag and CF moves to the high bit. Rotating left, the high bit moves to CF and CF moves to the low bit. There are 9 bits (or 17 bits for a word) involved in the rotation. Make yet another copy of the program, and change those 4 instructions to RCR. Also, since we have 9 bits instead of 8, change the loop count to 9 from 8:

```
    mov  si, 9
```

Enter a number and watch it move. Before you start moving, look at CF and see if there is anything in it. There are only two flags defined, OF and CF. Obviously, CF is set if there is something in it. OF is wierd. In RCL (the opposite instruction to the one we are using), OF operates normally, signalling a change in the top (sign) bit. In RCR, OF signals a change in CF. Why? I don't have the slightest idea. You really have no need for the OF flag anyways, so this is unimportant.

Well, those are the seven instructions, but what can you do with them besides multiply and divide?

First, you can work with multiple bit data. The 8087 has a word length register called the status register. Looking at the upper byte:

```
    15 14 13 12 11 10  9  8
           X  X  X
```

bits 11, 12 and 13 contain a number from 0 to 7. The data in this register is not directly accessable. You need to move the register into memory, then into an 8086 register. If you want to find what this number is, what do you do?

```
    mov  bx, status_register_data
    mov  cl, 3
    ror  bx, cl
    and  bh, 00000111b
```

we rotate right 3 and then mask off everything else. The number is now in BH. We could have used SHR if we wanted. Another 8087 register is the control register. In the upper byte it has:

```
    15 14 13 12 11 10  9  8
                 X  X
```

a number from 0 to 3 in bits 10 and 11. If we want the information, we do the same thing:

```
    mov  bx, control_register_data
    mov  cl, 2
    ror  bx, cl
```

```
    and  bh, 00000011b
```

and the number is in BH.

You are now going to write a program that inputs an unsigned number and prints out its hex representation. Here it is:

```
; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       mov   ax_byte, 0A5h        ; half regs, right ascii
       mov   bx_byte, 4           ; hex
       mov   dx_byte, 4           ; hex
       lea   ax, ax_byte
       call  set_reg_style
       call  show_regs

outer_loop:
       call  get_unsigned
       mov   bx, ax
       mov   dx, ax
       mov   cx, 4
inner_loop:
       push  cx                   ; save cx
       mov   cl, 4
       rol   bx, cl               ; rotate left 1/2 byte
       mov   al, bl               ; copy to al
       and   al, 0Fh              ; mask off upper 1/2 byte
       cmp   al, 10               ; < 10, 0 - 9 ;  > 9  A - F
       jae   use_letters
       add   al, '0'              ; change to ascii
       jmp   print_it
use_letters:
       add   al, 'A' - 10         ;  10 = 'A'
print_it:
       call  print_ascii_byte
       call  show_regs_and_wait
       pop   cx
       loop  inner_loop
       jmp   outer_loop
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

AL will be shown in ascii while BX and DX will be in hex. We save the original number in DX. Since the first thing we want to print is the left hex character, we rotate left, not right. We move the low byte to AL, mask off everything but the low hex number and then convert to an ascii character. If it is 0 - 9, we add '0' (the character, not the number). If it is > 9, we add "'A' - 10" and get a letter (if the number is 10, we get 'A'). JAE means jump if above or equal, and is an unsigned comparison.{3}

---

3 You are getting innundated with conditional jump instructions. Don't worry. As long as you understand each one when you run across it, you don't have to remember it. All jump instructions will be covered soon.

Finally, we print the ascii character that is in AL.{4}

Another thing to notice is that just inside the loop we push CX. That is because we use CL for the ROL instruction. It is then POPped just before the loop instruction. This is typical. CX is the only register that can be used for counting in indexed instructions. It is common for indexing instructions to be nested, so you temporarily store the old value of CX while you are using CX for something different.

```
    push cx        ; typical code for a shift
    mov  cl, 7
    shr  si, cl
    pop  cx
```

Finally, let's multiply large numbers by 2. Here's the code:

```
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
byte1  db    ?
byte2  db    ?
byte3  db    ?
byte4  db    ?
error_message  db  "Result is too large.", 0
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
outer_loop:
       lea   ax, byte1           ; get 4 byte number
       call  get_unsigned_4byte

       shl   byte1, 1
       rcl   byte2, 1
       rcl   byte3, 1
       rcl   byte4, 1
       jnc   go_on
       lea   ax, error_message
       call  print_string
go_on:
       lea   ax, byte1
       call  print_unsigned_4byte
       jmp   outer_loop
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

This will require some explaination. Get_unsigned_4byte gets a number from 1 to four billion. We put it in memory. Normally, the following instructions would be done word by word. We are doing them byte by byte so you can see the mechanics of the situation. The low byte is shifted left 1 bit. This doubles it, but may shift a 1 bit from the high bit into CF. If it does, then it will be present when we rotate byte2. That moves CF into the low bit and moves the high bit into CF. We do it again. And again. If there is an unsigned overflow, it will be signalled by CF being

---

4 Any subroutine in ASMHELP.OBJ that involves a one byte input or output has the data in AL.

set after:

```
    rcl  byte4, 1
```

JNC (jump on not carry) will skip the error message if everything is ok. Print_string prints a zero terminated string, that is a C string which is terminated by the number (not the character) 0. Finally, we print the number.

A word about large numbers in ASMHELP.OBJ. It is assumed that you would like to use commas if you could. Any data type over 1 word long allows commas. The following are considered the same by ASMHELP.OBJ in its input routines:

```
    23546787
    2,3,5,4,6,7,8,7
    23,,5,46,,78,7
    23,546787
    23,546,787
```

It always prints commas correctly in the print routines.

SUMMARY

All shift and rotate instructions operate on either a register or on memory. They can be either 1 bit shifts:

```
sal  cx, 1
ror  variable1, 1
shr  bl, 1
```

or shifts indexed by CL (it must be CL):

```
rcl  variable2, cl
sar  si, cl
rol  ah, cl
```

SHL and SAL

SHL (shift logical left) and SAL (shift arithmetic left) are exactly the same instruction. They move bits left. 0s are placed in the low bit. Bits are shoved off the register (or memory data) on the left side, and CF indicates whether the last bit shoved was a 1 or a 0. It is used for multiplying an unsigned number by powers of 2.

SHR

SHR (shift logical right) does the same thing as SHL but in the opposite direction. Bits are shifted right. 0s are placed in the high bit. Bits are shoved off the register (or memory data) on the right side and CF indicates whether the last bit shoved off was a 0 or a 1. It is used for dividing an unsigned number by powers of 2.

SAR

SAR (shift arithmetic right) shifts bits right. The high (sign) bit stays the same throughout the operation. Bits are shoved off the register (or memory data) on the right side. CF indicates whether the last bit shoved off was a 1 or a 0. It is used (with difficulty) for dividing a signed number by powers of 2.

ROR and ROL

ROR (rotate right) and ROL (rotate left) rotate the bits of a register (or memory data) right and left respectively. The bit which is shoved off one end is moved to the other end. CF indicates whether the last bit moved from one end to the other was a 1 or a 0.

RCR and RCL

RCR (rotate through carry right) and RCL (rotate through carry left) rotate the bits of a register (or of memory data) right and left respectively. The bit which is shoved off the register (or data) is placed in CF and the old CF is placed on the other side of the register (or data).

INC

INC increments a register or a variable by 1.

```
inc  ax
inc variable1
```

DEC

DEC decrements a register or a variable by 1.

```
dec  ax
dec  variable1
```

The following is fairly technical. It is only for those willing to wade their way through a turgid explaination. If you don't understand it, forget it.

CODE FOR SHL

If you are shifting an UNSIGNED number right by 'X' bits, it is the same as dividing by (2 ** X) 1 bit = (2**1 = 2), 2 bits = (2**2 = 4), 7 bits = (2**7 = 128). This is the same as dividing by a number which is all 0s except the Xth bit which is 1 (for 0 we have 0000 0001, for 1 we have 0000 0010, for 3 we have 0000 1000, for 7 we have 1000 0000). The remainder mask will be this number minus 1 (for 0 we have 0000 0000, for 1 we have 0000 0001, for 3 we have 0000 0111, for 7 we have 0111 1111).

CODE FOR SAR

The order of numbers is important for SAR. If you start with 0 and add 1 each time, the actual sequence of signed numbers that you get (from the bottom up) is:

```
-1
-2
 .
 .
-32767
-32768
+32767
+32766
 .
 .
 3
 2
 1
 0
```

The positive numbers are increasing in absolute value while the negative numbers are decreasing in absolute value. If you divide by shifting and there is no remainder, then the quotient is exact. If there is a remainder, the quotient will truncate towards 0 IN THE ABOVE DIAGRAM. This means that positive numbers will truncate down, while the negative numbers will truncate towards -32768, and will be one too negative.

If the number was positive, the remainder will be positive and will be exactly the same as for SHR. If the number was negative, then things are more complicated. We'll take division by 32 as an example. If we divide by 32 (0010 0000) the remainder mask will be 31 (0001 1111). If the number is negative, then what we get when we AND the mask:

```
    and  ax, 00011111b
```

is not the remainder but (remainder + 32). In order to get the actual negative remainder, we need to subtract 32. This gives us (remainder + 32 - 32).

```
    remainder mask = divisor - 1
    negative remainder correction = NEG divisor.
```

CHAPTER 9 - JUMPS

So far we have done almost exclusively sequential programming - the order of execution has been from one instruction to the next until the very end, where the jump instruction has brought us back to the top. This is not the normal way programs work. In order to have things like DO loops, FOR loops, WHILE loops, CASE constructions and IF-THEN-ELSE constructs, we need to make decisions and to be able to go to different blocks of code depending on our decisions.

Intel has provided a wealth of conditional jumps to answer all our needs. All of them are listed in a summary at the end of this chapter.

The thing we will do most often is compare the size of two numbers. In BASIC code:

```
IF   A < B  THEN
```

we need to see if A < B . If that is true, we do one thing, if it is false, we do something else. One thing that we need to watch out for is whether A and B are signed or unsigned numbers. Let A = F523h (signed -2781; unsigned 62755) and B = 59E0h (signed +23008; unsigned 23008). If A and B are signed numbers, then A < B. However, if they are unsigned numbers, then A > B. In C and PASCAL, the compiler takes care of whether you want signed or unsigned numbers (BASIC assumes that it is always signed numbers). You are now on the machine level, and must take care of this yourself.

To compare two numbers, you subtract one from the other, and then evaluate the result (less than 0, 0, or more than 0). To compare A and B, you do A minus B. To compare AX and BX, you can:

```
sub  ax, bx
```

and then evaluate it. Unfortunately, if you do that, you will destroy the information in AX. It will have been changed in the process. Intel has solved this problem with the CMP (compare) instruction.

```
cmp  ax, bx
```

subtracts BX from AX, sets the flags, and throws the answer away. CMP is exactly the same as SUB except that AX remains unchanged. We probably don't want to save the result anyway. If we do, we can always use:

```
sub  ax, bx
```

We have subtracted BX from AX. There are now three possibilities. (1) AX > BX so the answer is positive, (2) AX = BX so the answer is zero, or (3) AX < BX so the answer is negative. But are these

signed or unsigned numbers? The 8086 uses the same subtract (or CMP) instruction for both signed or unsigned numbers. You have to tell the 8086 in the following instruction whether you were talking about signed or unsigned numbers.

```
    cmp  ax, bx
    ja   some_label
```

asks the machine to compare AX and BX. It then says that AX and BX were UNSIGNED numbers and the machine should jump to "some_label" if AX was above BX.

```
    cmp  ax, bx
    jg   some_label
```

asks the machine to compare AX and BX. It then says that AX and BX were SIGNED numbers and the machine should jump to "some_label" if AX was greater than BX.

The 8086 makes the decision by looking at the flags. They give it complete information about the result. (The decision making rules are in the summary).

In our example on the previous page, we had A = -2781 (unsigned 62755) and B = +23008 (unsigned 23008). If we put A in AX and B in BX, then the instruction JA will execute the jump, while the instruction JG will not execute the jump. What happens if the 8086 doesn't execute the jump? It goes on to the next instruction.

As I said before, the 8086 has LOTS of conditional jumps. I am going to give you a list of them now, but be forewarned that the mnemonics for these suckers are confusing. Rather than doing something sensible like JUG for 'jump unsigned greater' and JSG for 'jump signed greater' - things that would be easy to distinguish, they have JA for 'jump above' (unsigned) and JG for 'jump greater' (signed).{1} Therefore, use the summary at the end of the chapter. With enough use, you will remember which is which.

The arithmetic jump instructions have two forms, a positive one and a negative one. For instance:

```
    ja        ; jump above
    jnbe      ; jump not (below or equal)
```

are actually the same machine instruction. You use whichever one makes the program clearer. We will have examples later to illustrate both positive and negative mnemonics. Here are the signed and unsigned instructions and their meaning. The

---

1 This wierd use of the words above and greater, below and less, is so confusing that my copy of the Microsoft assembler manual v5.1 has it reversed on one page. It calls the signed jumps unsigned and the unsigned jumps signed. And that's the one place where it SHOULD be right.

equivalent mnemonics will appear in pairs.

THESE ARE THE SIGNED JUMP INSTRUCTIONS

```
    jg        ; jump if greater
    jnle      ; jump if not (less or equal){2}

    jl        ; jump if less
    jnge      ; jump if not (greater or equal)

    je        ; jump if equal
    jz        ; jump if zero

    jge       ; jump if greater or equal
    jnl       ; jump if not less

    jle       ; jump if less or equal
    jng       ; jump if not greater

    jne       ; jump if not equal
    jnz       ; jump if not zero
```

These are self-explainatory, keeping in mind that these apply only to signed numbers.

THESE ARE THE UNSIGNED JUMP INSTRUCTIONS

```
    ja        ; jump if above
    jnbe      ; jump if not (below or equal)

    jb        ; jump if below
    jnae      ; jump if not (above or equal)

    je        ; jump if equal
    jz        ; jump if zero

    jae       ; jump if above or equal
    jnb       ; jump if not below

    jbe       ; jump if below or equal
    jna       ; jump if not above

    jne       ; jump if not equal
    jnz       ; jump if not zero
```

These apply to unsigned numbers, and should be clear.

JZ, JNZ, JE and JNE work for both signed and unsigned numbers. After all, zero is zero.

---

2 I was trying to decide whether or not to put in the parentheses. If there are two things after the "not" the "not" applies to both of them. By the rules of logic, not (less or equal) == not less AND not equal. If you don't understand this, try to find someone who can explain it to you.

Before we get going, there is one more thing you need to know. The unconditional jump:

```
jmp  some_label
```

can jump anywhere in the current code segment. XXXX is the current number in CS, then jmp can go from XXXX offset 0 to XXXX offset 65535.

Conditional jumps are something else entirely. ALL conditional jumps (including the loop instruction) are limited range jumps. They are from -128 to +127. That is, they can go backward up to 128 bytes and they can go forward up to 127 bytes.{3} You will find that you will get assembler errors because the conditional jumps are too far away, but don't worry because we can ALWAYS fix them. You will find out later how to deal with them.

As in the other arithmetic instructions, there are five forms of the CMP instruction.

1. compare two registers
2. compare a register with a variable
3. compare a variable with a register
4. compare a register with a constant
5. compare a variable with a constant

These look like:

```
cmp  ah, dl
cmp  si, memory_data
cmp  memory_data, ax
cmp  ch, 127
cmp  memory_data, 14938
```

Here are some decisions we might have to make in programs.

(1) We are writing a program to print hex numbers on the screen. We have one routine for 0 - 9 and a different one for A - F. Sound familiar?

```
cmp  al, 10
jb   decimal_routine
; here we are past the jump, so we can start the  A - F
; routine.
```

(2) We want to fire everyone over the age of 55 (this is an unsigned number since there are no negative ages):

---

3 But they don't jump from the beginning of the machine instruction, they jump from the END of the machine instruction (which is two bytes long). This means that they have an effective range of from -126 bytes to +129 bytes from the BEGINNING of the instruction.

```
    cmp  employee_age, 55
    ja   find_a_reason_for_termination
    ; start of routine for younger employees here.
```

(3) You want to know if you need to go to a loanshark:

```
    mov  di, bank_balance
    cmp  unpaid_bills, di
    jg   gotta_go_see_Vinnie
    ; continue normal routine here
```

(4) Notice that the last one could have also been written:

```
    mov  di, bank_balance
    cmp  di, unpaid_bills
    jl   gotta_go_see_Vinnie
    ; continue normal routine
```

though to my eye the first one seems clearer.

(5) You have the results from two calculations, the first one in DI and the second one in CX. You need to go to different routines depending on which is larger. If they are the same, you exit:

```
    cmp  di, cx
    jg   routine_one         ; di is greater
    jl   routine_two         ; cx is greater
    jmp  exit_label          ; they are equal
```

We had two conditional jumps in a row and both of them were able to look at the results of the CMP instruction because the first jump (JG) did not change any of the flags. This is true of all jumps - conditional jumps, the unconditional jump, and LOOP. They never change any of the flags, so you can have two jumps in a row and be certain that the flags will not be altered.

(6) Your dad has promised to buy you a Corvette if you don't get suspended from school more than three times this semester. Here's his decision code:

```
    cmp  number_of_suspensions, 3
    jng  buy_him_the_corvette
    ; better luck next time
```

JUMP OUT OF RANGE

If the code you are jumping to is more than -128 or +127 bytes from the end of a conditional jump, you will get an assembler error that the jump is out of range. This can always be fixed. Take this code:

```
         cmp  ax, bx
         jg   destination_label
    further_code:
         ; continue the program
```

If 'destination_label' is out of range, what you need to do is jump to 'further_code' with a conditional jump (you are within range of 'further_code') and use JMP (which can go anywhere in the segment) to go to 'destination_label'. To switch the jump, you simply negate the jump condition:

```
    jg   ->   jng
    je   ->   jne
    jne  ->   je
    jbe  ->   jnbe
```

We use reverse logic. Originally, if the condition was met we jumped. If the condition was not met we continued. Now, if the condition is NOT met, we jump, and if the condition is NOT not met (yes, there are two NOTs) which means it was met, we continue, and this sends us to the JMP instruction. Make sure you believe this works correctly before going on. The code then looks like this:

```
         cmp  ax, bx
         jng  further_code
         jmp  destination_label
    further_code:
         ; continue the program
```

This is the standard way of handling the situation.

SUMMARY

CMP

CMP performs the same operation as subtraction, but it does not change the registers or variables, it only sets the flags. It is the cousin of TEST. As usual, there are five possibilities:

1. compare two registers
2. compare a register with a variable
3. compare a variable with a register
4. compare a register with a constant
5. compare a variable with a constant

THESE ARE THE SIGNED JUMP INSTRUCTIONS

```
jg        ; jump if greater
jnle      ; jump if not (less or equal)

jl        ; jump if less
jnge      ; jump if not (greater or equal)

je        ; jump if equal
jz        ; jump if zero

jge       ; jump if greater or equal
jnl       ; jump if not less

jle       ; jump if less or equal
jng       ; jump if not greater

jne       ; jump if not equal
jnz       ; jump if not zero
```

THESE ARE THE UNSIGNED JUMP INSTRUCTIONS

```
ja        ; jump if above
jnbe      ; jump if not (below or equal)

jb        ; jump if below
jnae      ; jump if not (above or equal)

je        ; jump if equal
jz        ; jump if zero

jae       ; jump if above or equal
jnb       ; jump if not below

jbe       ; jump if below or equal
jna       ; jump if not above

jne       ; jump if not equal
jnz       ; jump if not zero
```

THESE JUMPS CHECK A SINGLE FLAG

These come in opposite pairs

```
    jc        ; jump if the carry flag is set
    jnc       ; jump if the carry flag is not set

    jo        ; jump if the overflow flag is set
    jno       ; jump if the overflow flag is not set

    jp or jpe ; jump if parity flag is set (parity is even)
    jnp or jpo  ;jump if parity flag is not set (parity is odd)


    js        ; jump if the sign flag is set (negative )
    jns       ; jump if the sign flag is not set (positive or 0)
```

THIS CHECKS THE CX REGISTER

```
    jcxz      ; jump if cx is zero
```

Why do we have this instruction? Remember, the loop instruction decrements CX and then checks for 0. If you enter a loop with CX set to zero, the loop will repeat 65536 times. Therefore, if you don't know what the value of CX will be when you enter the loop, you use this instruction just before the loop to skip the loop if CX is zero:

```
         jcxz after_the_loop

    loop_start:
              .
              .
              .
              .
         loop loop_start
    after_the_loop:
```

INFORMATION ABOUT JUMPS

The unconditional jump (JMP) can go anywhere in the code segment. All other jumps, which are conditional, can only go from -128 to +127 bytes from the END of the jump instruction (that's from -126 to +129 bytes from the beginning of the instruction).

Jumps have no effect on the 8086 flags.

How does the 8086 decide whether something is more, less, or the same? The rules for unsigned numbers are easy. If you subtract a larger number from a smaller, the subtraction will pass through zero and will set the carry flag (CF). If they are the same, the result will be zero and it will set the zero flag (ZF). If the first number is larger, the machine will clear the carry flag and the zero flag will be cleared (ZF = 0). Therefore, for unsigned numbers, we have:

First number is:

above	CF = 0	ZF = 0
equal	CF = 0	ZF = 1
not equal	CF = ?	ZF = 0
below	CF = 1	ZF = 0

All other unsigned compares are combinations of those three things:

```
jae  = ja OR je     (CF = 0 and ZF = 0) or ZF = 1
jbe  = jb OR je     CF = 1 or ZF = 1
```

When you have a negative condition, it is much easier to look at its equivalent positive condition to figure out what is going on:

```
jnae   is the same as    jb   CF = 1
jnbe   is the same as    ja   CF = 0    ZF = 0
```

SIGNED NUMBERS

This section is not for the fainthearted. It is not necessary, so if you find yourself getting confused, just remember that if you see documentation talking about a jump where the overflow flag equals the sign flag or the overflow flag doesn't equal the sign flag, it is talking about SIGNED numbers.

Zero is zero, so we won't concern ourselves with it here. It is exactly the same.

If A and B are two signed word sized numbers and we have:

```
cmp  A, B
```

then we can have four different cases:

1) If A is just a little greater than B [(A - B) <= +32767], then the result will be a small positive number, and there will be no overflow. SF = 0, OF = 0.

2) If A is much greater than B [+32767 < (A - B)], then the result will be too positive and it will overflow from positive to negative. This will set both the sign flag (it is now negative) and the overflow flag. SF = 1, OF = 1.

3) If A is a little less than B [-32768 <= (A - B)], that is if it is only a little negative, then the result is a small negative number, and there is no overflow. SF = 1, OF = 0.

4) If A is much less than B [(A - B) < -32768], then the result is a large negative number. It is too negative and overflows into a positive number. SF = 0, OF = 1.

Recapping, for a positive result:

1) SF = 0, OF = 0

2) SF = 1, OF = 1

and for a negative result:

3) SF = 1, OF = 0
4) SF = 0, OF = 1

For positive results (and zero), SF = OF. For negative results, SF is not equal to OF. This, in fact, is how the 8086 decides a signed jump. If SF = OF, it's positive, if SF is not equal to OF, it's negative. If ZF = 1, then obviously they are equal. Here is the list:

greater	SF = OF	ZF = 0
equal	SF = OF	ZF = 1
not equal		ZF = 0
less	SF is not equal to OF	

As with the unsigned numbers, if you have a negative condition, it is easier to change it into its equivalent positive condition and then figure out the requirements. For instance:

jnge	same as	jl	SF is not equal to OF
jnl	same as	jge	(SF = OF and ZF = 0) or (ZF = 1)

If you think about it, this OF = SF stuff does make sense. We are subtracting two numbers. If the first one is greater, then the answer will be positive. It can either be a little positive as in (cmp 0, -1) = 1 or it can be very positive, as in (cmp 32767, -32768) = 65,535 (same as -1). If it is just a little positive, there is no overflow and it has a positive sign (SF = 0, OF = 0). If the difference gets large, then the number overflows from + to -. At that point OF = 1, but it now has a negative sign, so OF = SF. The flags MUST match.

In the opposite case where the second number is greater, The answer is negative. It can either be a little negative as in (cmp 12, 13)= -1, or it can be very negative (cmp -32768, 32767) = -65535 =1. If it is a small difference, the sign is negative, but there is no overflow (SF = 1, OF = 0). As the difference gets larger, the number overflows from negative to positive so the sign flag is now positive, but the overflow flag is set (SF = 0, OF = 1). Those flags CAN'T match.

CHAPTER 10 - TEMPLATES

Do you remember when you were younger and you needed to look up a word in the dictionary? It would define the word in terms of a second word which you didn't know so you would look that up too. Most likely that second word was either defined in terms of a third word you didn't know or it referred you back to the first word.

This chapter is something like that. The items in the template file are interdependent. If you're lucky, everything will be clear by the time you have finished the chapter. If not, you'll have to reread it.

There are four different things which operate on the assembler instructions which you write - the ASSEMBLER, the LINKER, the LOADER and the 8086.

1) The ASSEMBLER takes your text and turns it into the machine code that is used by the 8086. It is complete except that the addresses of data and subroutines might change during linking and loading. The assembler generates information called HEADER files which give the LINKER and LOADER the information they need to update these addresses in the machine code. This means that you can move the code anywhere in memory.

2) If your program is made up of more than one file, the LINKER links them together. It then makes it ready for running. If there is only one file, the linker makes it ready for running. It does this by updating the addresses of anything it has moved. It still leaves the HEADER files which contain the segment addresses.

3) At run time, the LOADER, which is part of the operating system, decides where to put your program in memory. It loads the program, and adjusts any segment addresses in the program to reflect where the program actually is in memory. It then gives control to the program.

4) The code is fixed at the time the 8086 takes over. Any addresses are constants and are unchangable.

Keep this in mind as we work through the template file.

THE .LST FILE

The first thing we need to look at is segments. Let's look at a slightly modified version of the template file called segs.asm. Here it is.

```
;**********************************
; segs.asm
```

```
; - - - - - - - - - - - - - -
STACKSEG SEGMENT STACK  'STACK'

variable4    dw     4444h
             dw     100h dup (?)

STACKSEG  ENDS
; - - - - - - - - - - - - - -
MORESTUFF SEGMENT PUBLIC  'HOHUM'

variable2    dw   2222h

MORESTUFF    ENDS
; - - - - - - - - - - - - - -
DATASTUFF  SEGMENT PUBLIC  'DATA'

variable1    dw     1111h

DATASTUFF    ENDS
; - - - - - - - - - - - - - -
CODESTUFF SEGMENT PUBLIC  'CODE'

 EXTRN  print_num:NEAR , get_num:NEAR

 ASSUME cs:CODESTUFF,ds:DATASTUFF
 ASSUME es:MORESTUFF,ss:STACKSEG

variable3    dw     3333h

main   proc far
start: push  ds
       sub   ax,ax
       push  ax

       mov   ax, DATASTUFF
       mov   ds,ax
       mov   ax, MORESTUFF
       mov   es,ax

       mov   cx, variable1
       mov   variable1, cx

       ret

main   endp


CODESTUFF    ENDS
; - - - - - - - - - - - - -

    END     start
;**************************
```

There is an extra segment put in that has the definition

```
    MORESTUFF SEGMENT PUBLIC  'HOHUM'
```

There is a variable defined in each segment including the stack segment. These variables all have numbers in them, and the numbers are in hex so they will be easy to read. There are only two external subroutines (neither of which is called). It is time to take a look at an assembler listing.

----- THIS IS FROM THE SCREEN -----

```
C>masm segs
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

Object filename [segs.OBJ]:
Source listing  [NUL.LST]: segs
Cross-reference [NUL.CRF]:
```

If you don't put a semicolon after the filename with masm, you get some prompts. The first asks you if you want the object file name to be different from the asm file name. You may change either the name or the name and the extension. If you don't want to change either, just press ENTER. The second asks if you want a listing. Normally you don't, so you just press ENTER. This time we do, so we give it the same name as the assembler file. The assembler will generate a file SEGS.LST. Finally, it asks if you want the information needed to create a cross-reference file. We won't cover that. Once again, press ENTER. The assembler generates an object file and a listing. Here's the complete listing.

```
**********************
Microsoft (R) Macro Assembler Version 5.10
9/2/89 09:50:54

                                             Page     1-1


                    ; segs.asm

                    ; - - - - - - - - - - - - - -
 0000                    STACKSEG SEGMENT STACK  'STACK'

 0000  4444              variable4    dw     4444h
 0002  0100[                          dw     100h dup (?)
       ????
               ]


 0202                    STACKSEG  ENDS
                    ; - - - - - - - - - - - - - -
 0000                    MORESTUFF SEGMENT PUBLIC  'HOHUM'

 0000  2222              variable2    dw   2222h
```

```
0002                    MORESTUFF    ENDS
                  ; - - - - - - - - - - - - - - -
0000                    DATASTUFF  SEGMENT PUBLIC  'DATA'

0000  1111              variable1    dw     1111h

0002                    DATASTUFF    ENDS
                  ; - - - - - - - - - - - - - - -
0000                    CODESTUFF SEGMENT PUBLIC  'CODE'

                     EXTRN  print_num:NEAR , get_num:NEAR

                     ASSUME cs:CODESTUFF,ds:DATASTUFF
                     ASSUME es:MORESTUFF,ss:STACKSEG

0000  3333              variable3    dw     3333h

0002                    main   proc far
0002  1E             start: push  ds
0003  2B C0                    sub   ax,ax
0005  50                   push  ax

0006  B8 ---- R              mov   ax, DATASTUFF
0009  8E D8                  mov   ds,ax
000B  B8 ---- R              mov   ax, MORESTUFF
000E  8E C0                  mov   es,ax

0010  8B 0E 0000 R           mov   cx, variable1
0014  89 0E 0000 R           mov   variable1, cx

0018  CB                  ret

0019                    main   endp


0019                    CODESTUFF    ENDS

Microsoft (R) Macro Assembler Version 5.10
9/2/89 09:50:54

                                          Page     1-2


                  ; - - - - - - - - - - - -

                      END     start


Microsoft (R) Macro Assembler Version 5.10
9/2/89 09:50:54


Symbols-1


Segments and Groups:
```

```
          N a m e                 Length Align Combine  Class

CODESTUFF  . . . . . . . . . .  0019 PARA PUBLIC    'CODE'
DATASTUFF  . . . . . . . . . .  0002 PARA PUBLIC    'DATA'
MORESTUFF  . . . . . . . . . .  0002 PARA PUBLIC    'HOHUM'
STACKSEG . . . . . . . . . . .  0202 PARA STACK     'STACK'

Symbols:

        N a m e         Type     Value  Attr

GET_NUM  . . . . .      L NEAR    0000 CODESTUFF External

MAIN . . .   . . . . .  F PROC    0002 CODESTUFF Length = 0017

PRINT_NUM   . . . .     L NEAR    0000 CODESTUFF External

START  . . . . . . . .  L NEAR    0002 CODESTUFF

VARIABLE1  . . . . . .  L WORD    0000 DATASTUFF
VARIABLE2  . . . . . .  L WORD    0000 MORESTUFF
VARIABLE3  . . . . . .  L WORD    0000 CODESTUFF
VARIABLE4  . . . . . .  L WORD    0000 STACKSEG

@CPU . . . . . . . . . . . . .  TEXT  0101h
@FILENAME  . . . . . . . . . .  TEXT  segs
@VERSION . . . . . . . . . . .  TEXT  510


     54 Source  Lines
     54 Total   Lines
     21 Symbols

  48006 + 428261 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

**********************
```

As you can see, the listing, even for a short program, is very long. Let's take it apart section by section. The first large section is a copy of the text file except that there is information on the left. The number on the far left tells the offset address (in hex) from the beginning of the segment for each label, variable or instruction. In this section:

```
 0000  3333             variable3    dw     3333h

 0002                   main   proc far
 0002  1E          start: push  ds
 0003  2B C0                  sub   ax,ax
 0005  50                   push  ax

 0006  B8 ---- R              mov   ax, DATASTUFF
 0009  8E D8                  mov   ds,ax
```

```
 000B  B8 ---- R               mov   ax, MORESTUFF
 000E  8E C0                   mov   es,ax

 0010  8B 0E 0000 R            mov   cx, variable1
 0014  89 0E 0000 R            mov   variable1, cx

 0018  CB                 ret

 0019                   main   endp
```

"start" is at 0002h ,"mov cx, variable1" is at 0010h and "ret" is at 18h.

The second set of numbers is the actual machine instructions in hex. These are the what the 8086 operates on. "push ds" is 1E, "mov ds, ax" is 8E D8, and "ret" is CB. The instructions can be from 1 - 6 bytes long. Notice the "R" after some of the instructions. The "R" stands for relocatable. This means that it is an address that might be changed by either the linker or the loader. We'll talk about that later. In any case, the object file keeps track of these so they can be changed if necessary. Also, go back to the complete listing and look at the four variables; you will see that the values have been put in the object code; that is, 1111h, 2222h, 3333h and 4444h.

If we had had an error, the assembler would have placed an error message at the spot of the error in this part of the file.

The next part of the .LST file is the segment listing. It tells how the segments are defined.

```
        N a m e                 Length Align Combine  Class

    CODESTUFF  . . . . . .        0019 PARA PUBLIC    'CODE'
    DATASTUFF  . . . . . .        0002 PARA PUBLIC    'DATA'
    MORESTUFF  . . . . . .        0002 PARA PUBLIC    'HOHUM'
    STACKSEG . . . . . . .        0202 PARA STACK     'STACK'
```

We have the segment name, length, and some other information we'll talk about later. Notice that 'HOHUM' which is an artificial class, is dutifully listed with no complaints.

Then comes the listing of all labels, variables, and procedure names.

```
    Symbols:

        N a m e         Type     Value  Attr

    GET_NUM  . . . .    L NEAR    0000 CODESTUFF External
    MAIN   . . . . .    F PROC    0002 CODESTUFF Length = 0017
    PRINT_NUM    . .    L NEAR    0000 CODESTUFF External
    START  . . . . .    L NEAR    0002 CODESTUFF
    VARIABLE1  . . .    L WORD    0000 DATASTUFF
```

```
VARIABLE2  . . .    L WORD    0000 MORESTUFF
VARIABLE3  . . .    L WORD    0000 CODESTUFF
VARIABLE4  . . .    L WORD    0000 STACKSEG
```

It shows the segment and offset, whether they are bytes, words, processes etc. The "L" stands for label. The variables and procedures which are in an external file are so marked. Neither print_num nor get_num was called, but the assembler maintains a listing for them.

Finally, some internal info for the assembler.

```
@CPU . . . . . .  TEXT  0101h
@FILENAME  . . . . . .  TEXT  segs
@VERSION . . . . . .  TEXT  510
```

We will come back to parts of the .LST file, so make yourself comfortable with it.

SEGMENTS

It is now time for the nitty-gritty. We need to know what all those statements in the template file mean. Remember that there are four players in the game - (1) MASM, the Microsoft assembler, (2) LINK, the Microsoft linker, (3) the program loader and (4) the 8086 chip itself. Who does what to whom is the subject of this chapter.

You will notice that there are three segments in all the template files, one for data, one for code, and one for the stack. How many segments can a program have? An unlimited number for code, an unlimited number for data, and one for the stack.{1} Although you can have an unlimited number of segments, you can use only four at any one time - two for regular data (referenced by the DS and ES registers), one for code (referenced by the CS register), and one for temporary data (referenced by the SS register).

You don't have direct control over CS. You should NEVER change the value in SS. This means that you can only change which segments that ES and DS refer to. How do you do that? The 8086 does not allow you to move a constant into a segment register. Therefore it is a two step process. Put the constant into an arithmetic register (AX, BX, CX, DX, SI, DI or BP) and from there to the segment register. Suppose we have 327 different data segments in our file (named SEG1, SEG2, SEG3 ... SEG327) and we wanted to reference data in SEG27. The code would be:

```
mov  ax, SEG27
mov  ds, ax
```

1 Although if you REALLY need more space for a stack it is possible, if a little arcane.

This is the standard way to do it, and this is the same as the fourth and fifth instructions in the code segment of the template files where we are putting the address of DATASTUFF in ds.

What is that SEG27 in the instruction (mov ax, SEG27)? It is a constant. When the assembler assembles the program, it makes note of the fact that you want to have the starting address of SEG27 in that instruction (you saw the "R" in the listing for the instruction 'mov ax, DATASTUFF'). Later the linker makes sure there is a SEG27 segment in the complete program, gives it a temporary segment address, and puts this temporary address in every place that references that segment address. This address is guaranteed to be adjusted. You will see why when we look at the linker .MAP file.

Finally, the loader (which is the program that puts your program into memory) puts the segment where it wants and updates all references to the segment address to reflect where it now is. Thus, the program is complete only when this information is put in at run time. Each time you run the program SEG27 might be in a different place, but the loader will always update the references correctly.

We named the segments SEG1, SEG2, etc. Does SEG have to be part of the segment name? Not on your life. Here are three perfectly acceptable segment definitions:

```
CURLY     SEGMENT
LARRY     SEGMENT
MOE       SEGMENT
```

It is good practice to have 'SEG' as part of the segment name to remind you that these are segments, not variables, but this is a practice only, it is not a law. Any name you could use for a variable or a label you could use as a segment name. The reserved word SEGMENT after the name tells the assembler that this is the beginning of a segment with that name. You tell the assembler that you are starting a segment with 'SEGMENT'

```
CURLY     SEGMENT
```

and you tell the assembler that you are finished with that segment with the reserved word ENDS (END [of] Segment):

```
CURLY     ENDS
```

You need to put the name of the segment before the ENDS directive.

In the template file, the data segment definition reads:

```
DATASTUFF  SEGMENT  PUBLIC    'DATA'
```

DATASTUFF is the segment name, but what are PUBLIC and 'DATA' there for? To understand this, we need to look at the linker. First, let's assemble temp1.asm (our first template file) just the way it is.

```
---------- FROM THE SCREEN ----------
C>masm temp1.asm
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.


Object filename [temp1.OBJ]:
Source listing  [NUL.LST]: temp1
Cross-reference [NUL.CRF]:

----------
```

We have made the listing file so let's look at the segment information.

```
N a m e                         Length Align Combine Class

CODESTUFF  . . . . . . . . . .  000A PARA PUBLIC    'CODE'
DATASTUFF  . . . . . . . . . .  0000 PARA PUBLIC    'DATA'
STACKSEG . . . . . . . . . . .  00C8 PARA STACK     'STACK'
```

You will see that CODESTUFF is Ah (10d) bytes long, DATASTUFF has no data and is 0 bytes long, and STACKSEG is C8h (200d) bytes long.

Now let's link temp1.obj and asmhelp.obj.

```
---------- FROM THE SCREEN -----

C>link temp1+asmhelp

Microsoft (R) Overlay Linker  Version 3.61
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.


Run File [TEMP1.EXE]:
List File [NUL.MAP]: temp
Libraries [.LIB]:

----------
```

This time we have made a listing file for the link process. It is called TEMP.MAP. Let's look at it.

```
     Start  Stop   Length Name         Class
     00000H 000C7H 000C8H STACKSEG     STACK
     000D0H 00540H 00471H DATASTUFF    DATA
     00550H 01944H 013F5H CODESTUFF    CODE

    Program entry point at 0055:0000
```

This is what the map file looks like. There are still only three segments in the final executable file, STACKSEG, DATASTUFF and CODESTUFF. You will notice that the class name is still there, but the PUBLIC is missing. It's job is finished. "Start" says where the segment starts in the executable file, "Stop" says

where the segment ends in the executable file, and "Length" says the length in bytes of the segment. These numbers are 5 digit hex numbers instead of 4. That means that they are showing the total address. The segment number is the left 4 digits of 'Start'.

STACKSEG is C8h (200d) bytes long like before. Although DATASTUFF had no data, it is now 471h (1137d) bytes long, and CODESTUFF was Ah (10d) bytes long before but now it is a whopping 13F5h (5109d) bytes long. What happened? The linker did its work.

One of the things the linker does is combine things that we want to be in the same segment. It took the DATASTUFF segment from temp1.obj and appended the DATASTUFF segment from asmhelp.obj, combining them into one larger segment.{2} It took the CODESTUFF segment from temp1.obj and appended the CODESTUFF segment from asmhelp.obj, making them one large segment. Why did it do that? Because we put the word "PUBLIC" in the segment definition. When the assembler sees "PUBLIC" in the segment definition, it passes that information along to the linker in a header file.{3} When the linker has a segment which is "PUBLIC", it will append any other segment which (1) is "PUBLIC", (2) has the same name (i.e. CODESTUFF or DATASTUFF or CURLY etc.), and (3) has the same class name{4}. All three things must be true for the linker to combine them. We will actually check this out a little later to make sure you believe it.

One other thing to notice is that the linker is allocating only as much space as is needed. It could allocate 65536 bytes for each segment defined, but it uses only as much as the program needs and then starts the next segment at the next segment starting address. This is efficient management of memory.

What is the advantage of combining the smaller segments into one larger segment? For code, there is no big advantage. But for data, remember that every time we want to access data, we need to have the starting address of that particular segment in register ds. We do this by using:

```
    mov  ax, DATASTUFF
    mov  ds, ax
```

If we have a number of data segments, every time we access data

---

2 The linker always works from left to right. For each different type of segment, it starts with the first one it finds and then appends each succeeding one it finds.

3 A header is information for the linker or loader which is put in front of the machine code in an object file or an executable file. There are typically a number of headers in front of the machine code.

4 Remember that class names are somewhat arbitrary. I use 'CODE', 'DATA' and 'STACK' for clarity and because they are the standard Microsoft class names, but if you are not linking with anyone else's programs, you can use any class name you want.

we need to (1) make sure that ds contains the address of the correct data segment, and (2) if not, we need to write the code to change ds. This entails using a lot of code, can be confusing and is certainly error prone. With one data segment, you simply load ds with the correct address at the beginning of the program and then forget about it. This should be a rule for you. Unless you have truly humongous amounts of data (over 65535 bytes), ALWAYS put all your data in the same segment.

Do you remember those dashes '----' in the assembler listing? That was because the assembler didn't have a segment address to put there.

```
0004  B8 ---- R                mov   ax, DATASTUFF
0007  8E D8                    mov   ds,ax

0009  8B 0E 0000 R             mov   cx, variable1
000D  89 0E 0000 R             mov   variable1, cx
```

The linker now has a temporary address for the start of DATASTUFF (000D0h) so it will put the segment address (the left four hex bytes) in this spot. This is temporary, but will be updated by the loader. If variable1 has been moved, it will update that too.

Why am I sure that these temporary segments will be moved? The segment address of STACKSEG is 0000h. The segment address of DATASTUFF is 000Dh (13h) and the segment address of CODESTUFF is 0055h (85d). But the operating sysyem owns the first several THOUSAND segments. The loader will load your program in much higher memory. They must move.

So the linker combines all the segments we want to combine, and then it looks at the machine code and modifies every reference to the segments and to the variables which have been moved. That is a lot of work. For instance, when the linker appends asmhelp.obj, there are a hundred or so variables which it moves and a thousand or so references to those variables which it modifies. The linker does that every time you link a file with ASMHELP.OBJ. That's not too shabby.

Do all three conditions need to be met for the linker to combine segments into one segment?

1) They have the same name
2) They have the same class name
3) They are both defined PUBLIC

Joe Bob says check it out. Here are two .ASM files which contain a number of segments. Here's the first file:

```
;file1.asm
;- - - - - - - - - - - - - - - - - - - -
STACKSEG    SEGMENT   STACK  'STACK'
             dw     100 dup (?)
STACKSEG    ENDS
;- - - - - - - - - - - - - - - - - - - -
MORESTUFFA    SEGMENT   PUBLIC
variable21    dw     ?
MORESTUFFA    ENDS
;- - - - - - - - - - - - - - - - - - - -
DATASTUFF    SEGMENT   PUBLIC  'DATA'
variable1    dw     ?
DATASTUFF    ENDS
;- - - - - - - - - - - - - - - - - - - -
MORESTUFF    SEGMENT   PUBLIC  'DATA'
variable2    dw     ?
MORESTUFF    ENDS
;- - - - - - - - - - - - - - - - - - - -
EVENMORESTUFF    SEGMENT   PUBLIC  'DATA'
variable3    dw     ?
EVENMORESTUFF    ENDS
;- - - - - - - - - - - - - - - - - - - -
CODESTUFF    SEGMENT   PUBLIC  'CODE'
       ASSUME cs:CODESTUFF, ds:DATASTUFF
       ASSUME ds:MORESTUFF, es:MORESTUFFA, ds:EVENMORESTUFF
main   proc far
start: push  ds
       sub   ax,ax
       push  ax
       ret
main   endp
CODESTUFF    ENDS
;- - - - - - - - - - - - - - - - - - - -
       END     start
```

Here's the other file:

```
;file2.asm
; - - - - - - - - - - - - - - - - - - - -
STACKSEG    SEGMENT   STACK  'STACK'
             dw     100 dup (?)
STACKSEG    ENDS
; - - - - - - - - - - - - - - - - - - - -
NOTDATASTUFF    SEGMENT   PUBLIC  'DATA'
variable4    dw     ?
NOTDATASTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - -
```

```
DATASTUFF    SEGMENT   PUBLIC  'DATA'
variable5    dw     ?
DATASTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - -
MORESTUFFA    SEGMENT   PUBLIC
variable61    dw     ?
MORESTUFFA    ENDS
; - - - - - - - - - - - - - - - - - - - -
MORESTUFF    SEGMENT   PUBLIC  'CLASSOF68'
variable6    dw     ?
MORESTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - -
EVENMORESTUFF    SEGMENT   'DATA'
variable7    dw     ?
EVENMORESTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - -
CODESTUFF    SEGMENT   PUBLIC  'CODE'
 ASSUME cs:CODESTUFF, ds:DATASTUFF, ds:NOTDATASTUFF
 ASSUME ds:MORESTUFF,ds:MORESTUFFA, ds:EVENMORESTUFF
subroutine   proc far
       ret
subroutine    endp
CODESTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - -
       END
```

You will notice that the two CODESTUFFs, the two DATASTUFFs, the two MORESTUFFAs and the two STACKSEGs each have the same definitions, but that (1) NOTDATASTUFF has a different name than DATASTUFF, (2) one MORESTUFF has a different class name from the other, (3) one EVENMORESTUFF is PUBLIC and the other is not, and (4) the two MORESTUFFAs have NO class name.

Here's the segment information from file1.lst

```
N a m e                      Length Align  Combine  Class

CODESTUFF  . . . . . .  0005 PARA    PUBLIC  'CODE'
DATASTUFF  . . . . . .  0002 PARA    PUBLIC  'DATA'
EVENMORESTUFF  . . . . . .  0002 PARA    PUBLIC  'DATA'
MORESTUFF  . . . . . .  0002 PARA    PUBLIC  'DATA'
MORESTUFFA . . . . . .  0002 PARA    PUBLIC
STACKSEG . . . . . .  00C8 PARA    STACK   'STACK'
```

and from file2.lst

```
N a m e                    Length Align  Combine  Class

CODESTUFF  . . . . . .  0001 PARA    PUBLIC  'CODE'
```

```
DATASTUFF  . . . . . .   0002 PARA    PUBLIC  'DATA'
EVENMORESTUFF  . . . . .   0002 PARA    NONE    'DATA'
MORESTUFF  . . . . . .   0002 PARA    PUBLIC  'CLASSOF68'
MORESTUFFA . . . . . .   0002 PARA    PUBLIC
NOTDATASTUFF . . . . . .   0002 PARA    PUBLIC  'DATA'
STACKSEG . . . . . .   00C8 PARA    STACK   'STACK'
```

These are in alphabetical order. Before we link them together, let's think about what should happen if all three conditions must be met. Both CODESTUFF segments are PUBLIC with the same class name, so they should merge. Both DATASTUFF segments are PUBLIC with the same class name so they should merge. EVENMORESTUFF is PUBLIC in one file but not public in the other, so they should not merge. MORESTUFF is PUBLIC in both files, but they have different class names, so they should not merge. What about STACKSEG? The STACK combine type is similar to PUBLIC{1}, and they have the same class name, so they should merge. Finally, there are the MORESTUFFAs. They have the same name and are PUBLIC, but they have no class name. Will they combine?

Let's see what happens. Here is the .MAP file from the command

```
C> link file1+file2
```

```
 Start  Stop   Length Name                   Class
 00000H 0018FH 00190H STACKSEG              STACK
 00190H 001A1H 00012H MORESTUFFA
 001B0H 001C1H 00012H DATASTUFF             DATA
 001D0H 001D1H 00002H MORESTUFF             DATA
 001E0H 001E1H 00002H EVENMORESTUFF         DATA
 001F0H 001F1H 00002H NOTDATASTUFF          DATA
 00200H 00201H 00002H EVENMORESTUFF         DATA
 00210H 00220H 00011H CODESTUFF             CODE
 00230H 00231H 00002H MORESTUFF             CLASSOF68

Program entry point at 0021:0000
```

STACKSEG, DATASTUFF and CODESTUFF combined. MORESTUFFA combined. The others are separate. Doesn't this confuse the linker if it has more than one segment with the same name? No. The linker knows which variables are in which segments, and the names of the segments are not relevant.

If you look at the class information from the linker listing, you will notice that all things in the same class are grouped together. The linker works from left to right on the command line, so for the above, it read file1.obj first and then read

---

1 STACK tells the linker to combine any other segments which have STACK and the class type 'STACK' and it tells the loader to set the SS register to that segment, and set the SP register to point to the end of that segment.

file2.obj. It orders things (1) first by class (in the order encountered, and then (2) by segment (in the order encountered). For the linker ordering, a segment is like a subclass.

Look through the assembler files to check that if you link in the order file1+file2, the order of encountering classes is 'STACK', empty, 'DATA', 'CODE', and 'CLASSOF68'. check the segment ordering also. What if we link the opposite way?

```
> link file2+file1
```

Here's the listing:

```
 Start  Stop   Length Name                   Class
 00000H 0018FH 00190H STACKSEG              STACK
 00190H 00191H 00002H NOTDATASTUFF          DATA
 001A0H 001B1H 00012H DATASTUFF             DATA
 001C0H 001C1H 00002H EVENMORESTUFF         DATA
 001D0H 001D1H 00002H MORESTUFF             DATA
 001E0H 001E1H 00002H EVENMORESTUFF         DATA
 001F0H 00201H 00012H MORESTUFFA
 00210H 00211H 00002H MORESTUFF             CLASSOF68
 00220H 00234H 00015H CODESTUFF             CODE

Program entry point at 0022:0010
```

Assure yourself that this is the order the classes are encountered for file2+file1.

Before we go on, let's summarize what we have so far.

1) In an .asm file, each segment starts with a name followed by the word SEGMENT.

2) Each segment ends with the name followed by the word ENDS (end of segment).

This is the minimal segment definition:

```
; - - - - -
SEG_A  SEGMENT

SEG_A  ENDS
; - - - - -
```

In addition, if you want to combine a segment with segments from other files in order to make one large segment, then all the segments to be combined must:

1) have the same name.
2) have the same class name (type)
3) be declared PUBLIC

ASSUME

The next thing from the template file is the word ASSUME. Who is assuming what?

```
    ASSUME cs:CODESTUFF, ds:DATASTUFF
```

This is for the assembler. It says that whenever you are working in the CODESTUFF segment, CS will be set to the segment address of the CODESTUFF segment. Whenever you are working in the DATASTUFF segment, DS will be set to the segment address of the DATASTUFF segment. The CS register takes care of itself, but it is your responsibility to make sure that DS actually points to the proper segment.

If you just move a word from memory to a register:

```
    mov  cx, variable1
```

the 8086 automatically thinks that it is in the DS segment. But it doesn't have to be that way. The 8086 has something called segment overrides. Here is the list:

SEGMENT	HEX VALUE
CS	2E
DS	3E
ES	26
SS	36

An override is a 1 byte machine instruction that tells the 8086 that for the next instruction, the memory location will not reference the natural segment register; what it will reference is the segment register named in the override - CS if it is 2Eh, DS if it is 3Eh, ES if it is 26h, and SS if it is 36h.

We could plug these in ourselves, but that is a lot of work. Fortunately, the assembler takes care of this for us. Let's look at the code from the very beginning of the chapter.

```
;**********************************
; segs.asm

; - - - - - - - - - - - - - -
STACKSEG SEGMENT STACK  'STACK'

variable4dw   4444h
          dw     100h dup (?)

STACKSEG  ENDS
; - - - - - - - - - - - - - -
MORESTUFF SEGMENT PUBLIC  'HOKUM'

variable2    dw   2222h

MORESTUFF    ENDS
; - - - - - - - - - - - - - -
```

```
DATASTUFF  SEGMENT PUBLIC  'DATA'

variable1    dw     1111h

DATASTUFF    ENDS
; - - - - - - - - - - - - - -
CODESTUFF SEGMENT PUBLIC  'CODE'

 EXTRN  print_num:NEAR , get_num:NEAR

 ASSUME cs:CODESTUFF,ds:DATASTUFF
 ASSUME es:MORESTUFF,ss:STACKSEG

variable3    dw     3333h

main   proc far
start: push  ds
       sub   ax,ax
       push  ax

       mov   ax, DATASTUFF
       mov   ds,ax
       mov   ax, MORESTUFF
       mov   es, ax

       mov   cx, variable1
       mov   variable1, cx

       ret

main   endp


CODESTUFF    ENDS
; - - - - - - - - - - - - -

    END     start
;*************************
```

For the ASSUME statement we have:

```
 ASSUME cs:CODESTUFF,ds:DATASTUFF
 ASSUME es:MORESTUFF,ss:STACKSEG
```

What we want to look at is this section of code:

```
    mov  cx, variable1
    mov  variable1, cx
```

Here is the listing of the offset address and machine code:

```
   000E  8E C0                mov   es,ax

   0010  8B 0E 0000 R         mov   cx, variable1
   0014  89 0E 0000 R         mov   variable1, cx
```

```
0018  CB                    ret
```

Variable1 is in DATASTUFF (ASSUME ds:DATASTUFF), and DS is the natural segment for variables. Now let's change the code to:

```
    mov  cx, variable2
    mov  variable2, cx
```

This is the ONLY change in the file. Variable2 is in MORESTUFF and we have - ASSUME es:MORESTUFF. Here's the listing when we assemble the modified file.

```
000E  8E C0                  mov   es,ax

0010  26: 8B 0E 0000 R       mov   cx, variable2
0015  26: 89 0E 0000 R       mov   variable2, cx

001A  CB                     ret
```

The assembler has put 26h as a segment override. When the 8086 looks at the machine code, it knows that those two instructions reference the es, not the ds, segment register. Also note that the code is now two bytes longer - one byte for each segment override. The "ret" instruction is at 1Ah (26d) instead of 18h (24d).

Let's try it with:

```
    mov  cx, variable3
    mov  variable3, cx
```

Variable3 is in CODESTUFF and we have - ASSUME cs:CODESTUFF. Here's the listing:

```
000E  8E C0                  mov   es,ax

0010  2E: 8B 0E 0000 R       mov   cx, variable3
0015  2E: 89 0E 0000 R       mov   variable3, cx

001A  CB                     ret
```

The assembler put in the CS segment override. Now the 8086 knows that variable3 is in the CS segment. Finally:

```
    mov  cx, varaible4
    mov  variable4, cx
```

Variable4 is in STACKSEG and we have - ASSUME ss:STACKSEG. Here's the listing:

```
000E  8E C0                  mov   es,ax

0010  36: 8B 0E 0000 R       mov   cx, variable4
0015  36: 89 0E 0000 R       mov   variable4, cx

001A  CB                     ret
```

Once again, the assembler put in a segment override. This time it was the SS override.

That's nifty. We simply tell the assembler which segment register we will use for each segment and it does all the work. We will do more with segment overrides in the chapter on addressing modes.

Remember, though, that it is your responsibility to see that at the time this code is used, the segment register actually contains the appropriate segment address.

Is this ASSUME definition unique? That is, must there be a one to one correspondence between segments and registers, with each segment having its own register? No, not at all. Here are a two ASSUME statements, both of which are legal:

```
ASSUME cs:COMSEG, ds:COMSEG, es:COMSEG, SS:COMSEG
```

All four registers contain the address of the same segment. In fact, we will meet this statement when we talk about COM files. This is the only appropriate statement for a .COM file

```
ASSUME  ds:SEG_A, ds:SEG_B, es:SEG_C, es:SEG_D, es:SEG_A
```

Four different segments, two of which are referenced by DS and three of which are referenced by ES. Remember, ASSUME tells the assembler that whenever you access something in that segment, the named register will be set to the starting segment address. What exactly does this mean to the assembler? Let's rearrange this a little:

```
SEG_A     ds, es
SEG_B     ds
SEG_C     es
SEG_D     es
```

This is the list from the assembler's viewpoint. Suppose it has a variable that is in SEG_C. Does it need an override? Yes, it needs an ES override. Suppose it has a variable in SEG_A. Does it need an override? No, because DS is set to that segment.

SUBROUTINES

In assembler parlance, subroutines are called procedures. Why? You got me. In any case, whenever I say subroutine, process, subprogram, or anything like that, I mean a procedure. A procedure can have any name you want. You start a procedure by giving the name, using the reserved word 'proc' and then defining it as either near or far.

```
my_procedure  proc  near
```

is a near procedure with the name my_procedure. You end a procedure by giving the name and following it with the reserved word 'endp' (for end of procedure).

```
my_procedure  endp
```

What is a near procedure? It is one which is ALWAYS in the same segment as the calling program. When you call a near procedure, the value in CS stays the same, but IP (the instruction pointer) changes to the offset of the first byte of the procedure. The next instruction executed will be the first byte of the procedure.

If a procedure is called even once from a different segment, then it MUST be a far procedure.

```
my_procedure  proc  far

my_procedure  endp
```

When you call a far procedure, the CS register is changed to the segment of the called procedure and IP (the instruction pointer) is set to the first byte of the procedure. This will be covered in the chapter on subroutines.

How does the loader know where to start the program? The assembler tells the linker which tells the loader. How does the assembler know? You tell it. The last line of the file is the single word 'END'. That tells the assembler that you are done with the assembler code. If there is a word after the word 'END' (on the same line), then the assembler assumes that this word is the name of the label where the program starts. The first instruction executed will be whatever immediately follows that label. In the template files we have:

```
END start
```

so the label 'start:' indicates where the first instruction is. For an .EXE file, this can be anywhere at all, but we have it at the beginning. The label 'start:' is used for clarity, but we could just as easily have had:

```
END zzyx4
```

The assembler would then look for the label 'zzyx4:' as the place to start the program. If you look at the link .MAP file from our file1+file2 example you will see:

```
Program entry point at 0021:0000
```

That says that the starting address is CS = 0021h, IP = 0000h. Note that both CS and IP are different for the file2+file1 example:

```
Program entry point at 0022:0010
```

where CS = 0022h and IP = 0010h. The initial offset was given to the linker by the assembler. The linker did any adjustment to the offset if it moved the code, and then it calculated the segment address itself.

RET

When the loader loads the program, it puts the segment of the starting address in CS and the offset of the starting address in IP. This gives control to your program. When your program is done, how does it get back to the operating system? Good question.

When the loader loads the program, it creates something called the PSP (program segment prefix). This is a 100h (256d) byte block of information and code. The first byte (offset 0000) of this block is an 8086 instruction for an orderly exit from a program. What we need to do is set CS to the PSP segment and set IP to 0000. Then the next instruction executed will be the orderly exit code.

In talking about procedures, I said that when you call a far procedure, the 8086 puts the procedure's segment in CS and the procedure's offset in IP. But before that, it does two things:

```
    push CS         ; these are the old CS and IP
    push IP         ; this is not a real 8086 instruction {2}
```

When you have a RET (return) instruction in a far procedure, the 8086 does the following:

```
    pop  IP         ; this is not a real 8086 instruction
    pop  CS         ; put back the old CS and IP
```

so RET resets CS and IP to go back where it came from. That is its job.

What has been pushed on the stack before starting your program? NOTHING. That's right. That means that if you execute

```
    ret
```

at the end of your program, the 8086 will pop two pieces of garbage into IP and CS.

Fortunately, when setting up a program, the loader ALWAYS puts the segment address of the PSP in DS., the data segment. All we need to do is PUSH DS (the PSP) and then PUSH 0 (offset 0000) and we have the address of our orderly exit code. If we then execute RET, it will POP these two items into IP and CS, sending us to our orderly exit code. That is what is at the beginning of the code section of the template file. We cannot PUSH a constant, so we manufacture a 0 with 'sub ax, ax'. The code is:

```
    push ds         ; PSP segment
    sub  ax, ax     ; manufacture a 0
```

---

2 This is not actual 8086 code. You have no direct access to IP. This is, however, what the 8086 effectively does.

```
    push ax        ; offset = 0000
```

and the program is set up for the return.

That's a lot of things together, so let's review. To exit a procedure we use RET, but for the starting procedure we need to return to the operating system. The PSP has the code for an orderly return at offset 0000. At load time, the loader puts the segment address of the PSP in DS. We push the PSP segment address and offset 0000 for later use by the RET instruction. We do this with:

```
    push ds        ; PSP segment
    sub  ax, ax    ; manufacture a 0
    push ax        ; offset = 0000
```

These should be the first instructions in the program.

Now that you have stored the PSP, DS is free for other use. You can now use DS to hold the segment address of your data. DS is used because that is the segment register that the 8086 expects unless told otherwise. You can't move a constant to a segment register, so this is a two step process:

```
    mov  ax, DATASTUFF
    mov  ds, ax
```

EXTRN

Finally, an EXTRN statement tells the assembler that the procedure or data is in another file and you did not forget it. For a procedure, you need to say whether it is NEAR (push old IP and put in new IP) or far (push old CS and IP; put in new CS and IP). Here is the assembler listing for five calls:

```
       E8 09CA R           call  near_routine
       9A 15EE ---- R      call  far_routine
       E8 0000 E           call  near_external_routine
       9A 0000 ---- E      call  far_external_routine
       E8 0000 E           call  get_unsigned
```

The first two are in the same file, the next two are in an external file, and we have our friend 'get_unsigned'. 'R' means that the data may be changed, 'E' means that it is external, and will be done by the linker. The first four are labelled whether they are near or far. 'get_unsigned' is a near procedure. Notice that E8 is the near call while 9A is the far call. Also notice that the assembler reserves one word for the new IP in the near calls. If the call is in the same file, the assembler fills in this number, but if it is external the assembler sets it to 0. In the far calls the assembler reserves two words instead of one. The first word is again the new IP, which is either filled in or set to zero. The second word is for the segment address, and will be set by the linker.

Whew!!! It sure took a long time to go through all that and you still probably are unsure about some of this. Read the summary, and if you don't feel good about it, leave it for a day or two and reread it then.

At the end of the book I will show you how you can simplify a lot of these things by using standardized segment names and some other standardized instructions. For now, you need to get used to what the structure of programs is, and we will continue using the same type of templates.{3}

3 Just think of me as the computer equivalent of a woodshop teacher who forces you to use hand tools to make a coffee table rather than allowing you to use what you really want to be using - a chainsaw.

SUMMARY

SEGMENTS

Segments are defined by giving a name followed by the word SEGMENT. The end of a segment is signalled by the segment name, followed by the word ENDS (end of segment).

```
; - - - - -
SOME_NAME  SEGMENT

SOME_NAME   ENDS
; - - - - -
```

(As always, anything after a comma is a comment and is ignored by the assembler). In addition, if you want to combine a segment with other segments, then all the segments to be combined must:

1) have the same name.
2) have the same type (class)
3) be declared PUBLIC

THE STACK SEGMENT

The stack segment may have any name you want, but should be declared " SEGMENT STACK 'STACK' ". This forces the loader to do certain initialization for you. If you don't declare it this way, you have to do the initialization yourself.

```
ANY_NAME  SEGMENT  STACK  'STACK'
```

EXTRN

For procedures, an EXTRN statement tells the assembler that the procedure that you want to call is in a different file, that you didn't forget it. Procedures which are EXTRN must be declared either NEAR or FAR. The grammar is name colon NEAR or name colon FAR.

```
EXTRN     procedure1:NEAR, procedure2:FAR
```

You may declare as many things on one line as will fit, but you need to separate them with commas. There can be no comma at the end.

ASSUME

An ASSUME statement tells the assembler that when a statement references that particular segment, the corresponding segment register will be set to that segment address.

```
ASSUME    es:MORESTUFF
```

tells the assembler that no matter what you do in other parts of the program, every time a variable in MORESTUFF is referenced, es will have the segment address of MORESTUFF. This is for the purpose of correct coding of segment overrides.

SEGMENT OVERRIDES

Normally, when the 8086 accesses a variable in memory, it does so via the DS segment register. This can be changed with a segment override. The assembler puts the correct segment override code in front of the instruction and the 8086 will use that segment register to access the data in memory. The override codes are:

SEGMENT	HEX VALUE
CS	2E
DS	3E
ES	26
SS	36

CS

CS is the code segment. When the 8086 processes machine code, it ALWAYS uses CS. There is no override.

IP

IP, the instruction pointer, gives the offset in CS of the next instruction to be processed. When the 8086 processes an instruction, it looks at IP, gets the next instruction and updates IP. This is totally automatic and internal to the 8086. You have no direct access to IP.

PROCEDURES

A procedure is declared by giving a name followed by the word 'proc' followed by either NEAR or FAR. A procedure is ended by giving the name, followed by 'endp' (end of procedure).

```
; - - - - -
square_root    proc far

square_root    endp
; - - - - -
```

The words NEAR and FAR are for the assembler and the linker so they know whether to change just IP or both IP and CS in RET, the return statement as well as in CALL, the subroutine call.

RET

The assembler codes a near or a far return depending on whether you have declared a near or a far procedure. A NEAR return POPs

IP off of the stack while a FAR return POPs IP then POPs CS. Thus, a NEAR return stays in the same segment but a FAR return gets a new segment address in CS.{4}

END

The word END signals to the assembler that you are done with code. The assembler will ignore all following lines, whether they are blank or contain code.

If the line with END has a name after the word END, then the assembler assumes that this is the name of a label where execution will begin at run time. That means that the instruction at 'label:' will be the first instruction executed in the program.

SETUP

In order to setup the program in the beginning you need to (1) PUSH the segment address of the PSP (which is in DS), then push 0 (the offset of the orderly return code). Following this you need to put the segment address of the data segment in DS. The code for all of this is:

```
    push ds             ; PSP seg address is in ds
    sub, ax, ax         ; 0
    push ax             ; push 0000 offset

    mov  ax, DATA_SEG   ; data segment address to ds
    mov  ds, ax
```

4 Of course, it is possible for CS to keep the same value if the calling procedure is is the same segment.

CHAPTER 11 - ADDRESSING MODES AND POINTERS

In this chapter we are going to cover all possible ways of getting data to and from memory with the different addressing modes. Read this carefully, since it is likely this is the only time you will ever see ALL addressing possibilities covered.

The easiest way to move data is if the data has a name and the data is one or two bytes long. Take the following data:

```
; -----
variable1 dw  2000
variable2 db  -26
variable3 dw  -589
; -----
```

We can write:

```
    mov  variable1, ax
    mov  cl, variable2
    mov  si, variable3
```

and the assembler will write the appropriate machine code for moving the data. What can we do if the data is more than two bytes long? Here is some more data:

```
; -----
variable4 db  "This is a string of ascii data."
variable5 dd  -291578
variable6 dw  600 dup (-11000)
; -----
```

Variable4 is the address of the first byte of a string of ascii data. Variable5 is a single piece of data, but it won't fit into an 8086 register since it is 4 bytes long. Variable6 is a 600 element long array, with each element having the value -11000. In order to deal with these, we need pointers.

Some of you will be flummoxed at this point, while those who are used to the C language will feel right at home. A pointer is simply the address of a variable. We use one of the 8086 registers to hold the address of a variable, and then tell the 8086 that the register contains the address of the variable, not the variable itself. It "points" to a place in memory to send the data to or retrieve the data from. If this seems a little confusing, don't worry; you'll get the hang of it quickly.

As I have said before, the 8086 does not have general purpose registers. Many instructions (such as LOOP, MUL, IDIV, ROL) work only with specific registers. The same is true of pointers. You may use only BX, SI, DI, and BP as pointers. The assembler will give you an error if you try using a different register as a pointer.

There are two ways to put an address in a pointer. For variable4, we could write either:

```
    lea  si, variable4
```

or:

```
    mov  si, offset variable4
```

Both instructions will put the offset address of variable4 in SI.{1} SI now 'points' to the first byte (the letter 'T') of variable4. If we wanted to move the third byte of that array (the letter 'i') to CL, how would we do it? First, we need to have SI point to the third byte, not the first. That's easy:

```
    add  si, 2
```

But if we now write:

```
    mov  cl, si
```

we will generate an assembler error because the assembler will think that we want to move the data in SI (a two byte number) to CL (one byte). How do we tell the assembler that we are using SI as a pointer? By enclosing SI in square brackets:

```
    mov  cl, [si]
```

since CL is one byte, the assembler assumes you want to move one byte. If you write:

```
    mov  cx, [si]
```

then the assembler assumes that you want to move a word (two bytes). The whole thing now is:

```
    lea  si, variable4
    add  si, 2
    mov  cl, [si]
```

This puts the third byte of the string in CL. Remember, if a register is in square brackets, then it is holding the ADDRESS of a variable, and the 8086 will use the register to calculate where the data is in memory.

What if we want to put 0s in all the elements of variable6?

---

1 LEA stands for load effective address. Note that with LEA, we use only the name of the variable, while with:

```
    mov  si, offset variable4
```

we need to use the word 'offset'. The exact difference between the two will be explained later.

Here's the code:

```
        mov  bx, offset variable6
        mov  ax, 0
        mov  cx, 600
    zero_loop:
        mov  [bx], ax
        add  bx, 2
        loop zero_loop
```

We add 2 to BX each time since each element of variable6 is a word (two bytes) long. There is another way of writing this:

```
        mov  bx, offset variable6
        mov  cx, 600
    zero_loop:
        mov  [bx], 0
        add  bx, 2
        loop zero_loop
```

Unfortunately, this will generate an assembler error. Why? If the assembler sees:

```
        mov  [bx], ax
```

it knows that you want to move what is in AX to the address in BX, and AX is one word (two bytes) long so it generates the machine code for a word move. If the assembler sees:

```
        mov  [bx], al
```

it knows that you want to move what is in AL to the address in BX, and AL is one byte long, so it generates the machine code for a byte move. If the assembler sees:

```
        mov  [bx], 0
```

it doesn't know whether you want a byte move or a word move. The 8086 assembler has implicit sizing. It is the assembler's job to look at each instruction and decide whether you want to operate on a byte or a word. Other microprocessors do things differently. On the Motorola 68000, the assembler uses explicit sizing. Each instruction must explicitly state whether it is a byte or a word.{2} On the 68000 you have:

```
        move.b    #213, (A1)
        move.w    #213, (A1)
```

The first instruction says to move a byte (the number 213) to the address in register A1 while the second instruction says to move

---

2 Any of you who use the 68000 assembler know that this is fudging the facts a little bit.

a word (the number 213) to the address in register A1.{3}

Back to the 8086. If the 8086 assembler looks at an instruction and it can't tell whether you want to move a byte or a word, it generates an error. When you use pointers with constants, you should explicitly state whether you want a byte or a word. The proper way to do this is to use the reserved words BYTE PTR or WORD PTR.

```
        mov  [bx], BYTE PTR 213
        mov  [bx], WORD PTR 213
```

These stand for byte pointer and word pointer respectively. I find this terminology exceptionally clumsy, but that's life. Whenever you are moving a constant with a pointer, you should specify either BYTE PTR or WORD PTR.

The Microsoft assembler makes some assumptions about the size of a constant. If the number is 256 or below (either positive or negative), you MUST explicitly state whether it is a byte or a word operation. If the number is 257 or above (either positive or negative), the assembler assumes that you want a word operation.

Here's the previous code rewritten correctly:

```
        mov  bx, offset variable6
        mov  cx, 600
    zero_loop:
        mov  [bx], WORD PTR 0
        add  bx, 2
        loop zero_loop
```

Let's add 435 to every element in the variable6 array:

```
        mov  bx, offset variable6
        mov  cx, 600
    add_loop:
        add  [bx], WORD PTR 435
        add  bx, 2
        loop add_loop
```

How about multiplying every element in the array by 12?

```
        mov  di, offset variable6
        mov  cx, 600
        mov  si, 12
    mult_loop:
        mov  ax, [di]
        imul si
        mov  [di], ax
        add  di, 2
        loop mult_loop
```

3 A1 is a 68000 register.

None of these examples did any error checking, so if the result was too large, the overflow was ignored. This time we used DI for a change of pace. Remember, we may use BX, SI, DI or BP, but no others. You will notice that in all these examples, we started at the beginning of the array and went step by step through the array. That's fine, and that's what we normally would do, but what if we wanted to look at individual elements? Here's a sample program:

```
; + + + + + START DATA BELOW THIS LINE
;
poem_array  db "She walks in Beauty, like the night"
            db "Of cloudless climes and starry skies;"
            db "And all that's best of dark and bright"
            db "Meet in the aspect ratio of 1 to 3.14159"
character_count  db  149
; + + + + + END DATA ABOVE THIS LINE

; + + + + + START CODE BELOW THIS LINE

    mov  bx, offset poem_array
    mov  dl, character_count

character_loop:
    sub  ax, ax              ; clear ax
    call get_unsigned_byte
    dec  al                  ; character #1 = array[0]
    cmp  al, dl              ; out of range?
    ja   character_loop      ; then try again
    mov  si, ax              ; move char # to pointer register
    mov  al, [bx+si]         ; character to al
    call print_ascii_byte
    jmp  character_loop

; + + + + + END CODE ABOVE THIS LINE
```

You enter a number and the program prints the corresponding character. Before starting, we put the array address in BX and the maximum character count in DL. After getting the number from get_unsigned_byte, we decrement AL since the first character is actually poem_array[0]. The character count has been reduced by 1 to reflect this fact. It also makes 0 an illegal entry. Notice that the program checks to make sure you don't go past the end of the poem. This time we use BX to mark the beginning of the array and SI to count the number of the character.

Once again, there are only specific combinations of pointers that can be used. They are:

    BX with either SI or DI (but not both)
    BP with either SI or DI (but not both)

My version of the Microsoft assembler (v5.1) recognizes the forms [bx+si], [si+bx], [bx][si], [si][bx], [si]+[bx] and [bx]+[si] as the same thing and produces the same machine code for all six.

We can get even more complicated, but to show that, we need structures. In databases they are called records. In C they are called structures; in any case they are the same thing - a group of different types of data in some standard order. After the group is defined, we usually make an array with the identical structure for each element of the array.{4} Let's make a structure for an address book.

```
last_name  db  15 dup (?)
first_name db  15 dup (?)
age        db  ?
tel_no     db  10 dup (?)
```

In this case, all the data is bytes, but that is not necessary. It can be anything. Each separate piece of data is called a FIELD. We have the last_name field, the first_name field, the age field, and the tel_no field. Four fields in all. The structure is 41 bytes long. What if we want to have a list of 100 names in our telephone book? We can allocate memory space with the following definition:

```
address_book   db  100 dup ( 41 dup (' ')) {5}
```

Well, that allocates room in memory, but how do we get to anything? First, we need the array itself:

```
mov  bx, offset address_book
```

Then we need one specific entry. Let's take entry 29 (which is address_book[28]). Each entry is 41 bytes long, so:

```
mov  ax, 28    ; entry (less 1)
mov  cx, 41    ; entry length
mul  cx
mov  di, ax    ; move to pointer
```

That gives us the entry, but if we want to get the age, that's not the first byte of the structure, it's the 31st byte (actually address_book[28] + 30 since the first byte is at +0). We get it by writing:

```
mov  dl, [bx+di+30]
```

This is the most complex thing we have - two pointers plus a constant. The total code is then:

```
mov  bx, offset address_book
mov  ax, 28    ; entry (less 1)
mov  cx, 41    ; entry length
```

---

4 If you don't know about structures or records, now would be a good time to stop and go to a reference book about them. They are not actually covered here.

5 Nesting of dup statements is allowed. Rather than having uninitialized data, this has blanks in all the spaces.

```
    mul  cx        ; entry offset from array[0]
    mov  di, ax    ; move entry offset to pointer
    mov  dl, [bx+di+30]  ; total address
```

Though the machine code has only one constant in the code, the assembler will allow you to put a number of constants in the assembler instruction. It will add them together for you and resolve them into one number.{6}

Once again, there are a limited number of registers - they are the same registers as before:

    BX with either SI or DI (but not both) plus constant
    BP with either SI or DI (but not both) plus constant

We can work with structures on the machine level, but it looks like it's going to be hard to keep track of where each field is. Actually, it isn't so bad because of:

OUR FRIEND, THE EQU STATEMENT

The assembler allows you to do substitution. If you write:

```
    somestuff EQU  37 * 44
```

then every place that the assembler finds the word "somestuff", it will substitute what is on the right side of the EQU. Is that a number or text? Sometimes it's a number, sometimes it's text. Here are four statements which are defined totally in terms of numbers. This is from the assembler listing. (The assembler lists how it has evaluated the EQU statement on the left after the equal sign.)

```
 = 0023              statement1 EQU  5 * 7
 = 0025              statement2 EQU  statement1 + 6 - 4
 = 000F              statement3 EQU  statement2 - 22
 = 001F              statement4 EQU  statement3 + 16
```

and the assembler thinks of these as numbers (these numbers are in hex). Now in the next set, with only a minor change:

```
 = [bp + 3]                    statement1 EQU  [bp + 3]
 = [bp + 3] + 6 - 4            statement2 EQU  statement1 + 6 - 4
 = [bp + 3] + 6 - 4 - 22       statement3 EQU  statement2 - 22
```

6 And it does it quite well. The assembler correctly evaluated the following:

```
       add   ax, (-3*81)+44/8+[si+27]+6+[bx]-7+(43*96)-2
```

Not bad, huh?

```
 = [bp + 3] + 6 - 4 - 22 + 16  statement4 EQU  statement3 + 16
```

the assembler thinks of it as text. Obviously, the fact that it can be either may cause you some problems along the way. Consult the assembler manual for ways to avoid the problem.

Now we have a tool to deal with structures. Let's look at that structure again.

```
    last_name  db  15 dup (?)
    first_name db  15 dup (?)
    age        db  ?
    tel_no     db  10 dup (?)
```

We don't actually need a data definition to make the structure, we need equates:

```
    LAST_NAME      EQU  0
    FIRST_NAME     EQU  15
    AGE            EQU  30
    TEL_NO         EQU  31
```

this gives us the offset from the beginning of each record. If we again define:

```
    address_book   db  100 dup ( 41 dup (' '))
```

then to get the age field of entry 87, we write:

```
    mov  bx, offset address_book
    mov  ax, 86    ; entry (less 1)
    mov  cx, 41    ; entry length
    mul  cx        ; entry offset from array[0]
    mov  di, ax    ; move entry offset to pointer
    mov  dl, [bx+di+AGE]  ; total address
```

This is a lot of work for the 8086, but that is normal with complex structures. The only thing that takes a lot of time is the multiplication, but if you need it, you need it.{7}

How about a two dimensional array of integers, 60 X 40

```
    int_array  dw  40 dup  ( 60 dup ( 0 ))
```

These are initialized to 0. For our purposes, we'll assume that the first number is the row number and the second number is the column number; i.e. array [6,13] is row 6, column 13. We will have 40 rows of 60 columns. For ease of calculation, the first array element is int_array [0,0]. (If it is your array, you can

---

7 You will see more of the EQU statement.

set it up any way you want {8}). Each row is 60 words (120 bytes) long. To get to int_array [23, 45] we have:

```
    mov  ax, 120   ; length of one row in bytes
    mov  cx, 23    ; row number
    mul  cx
    mov  bx, ax    ; row offset to bx
    mov  si, 45    ; column offset
    sal  si, 1     ; multiply column offset by 2 (for word size)
    mov  dx, [bx+si]    ; integer to dx
```

Using SAL instead of MUL is about 50 times faster. Since most arrays you will be working with are either byte, word, or double word (4 bytes) arrays, you can save a lot of time. Let ELEMENT_NUMBER be the array number (starting at 0) of the desired element in a one-dimensional array. For byte arrays, no multiplication is needed. For a word:

```
    mov  di, ELEMENT_NUMBER
    sal  di,1      ; multiply by 2
```

and for a double word (4 bytes):

```
    mov  di, ELEMENT_NUMBER
    sal  di, 1
    sal  di, 1     ; multiply by 4
```

This means that a one-dimensional array can be accessed very quickly as long as the element length is a power of 2 - either 2, 4 or 8. Since the standard 8086 data types are all 1, 2, 4, or 8 bytes long, one dimensional arrays are fast. Others are not so fast.

As a quick review before going on, these are the legal ways to address a variable on the 8086:

(1) by name.

```
          mov  dx, variable1
```

It is also possible to have name + constant.

```
          mov  dx, variable1 + 27
```

The assembler will resolve this into a single offset number and will give the appropriate information to the linker.

(2) with the single pointers BX, SI, DI and BP (which are enclosed in square brackets).

```
          mov  cx, [si]
```

---

8 Bearing in mind that all compiled languages have fixed formats for arrays. If you want your array to interact with C, Fortran, Pascal or Basic, you'd better be sure you have the right format.

```
xor  al, [bx]
add  [di], cx
sub  [bp], dh
```

(3) with the single pointers BX, SI, DI and BP (which are enclosed in square brackets) plus a constant.

```
mov  cx, [si+421]
xor  al, 18+[bx]
add  93+[di]-7, cx
sub  (54/7)+81-3+[bp]-19, dh
```

(4) with the double pointers [bx+si], [bx+di], [bp+si], [bp+di] (which are enclosed in square brackets).

```
mov  cx, [bx][si]
xor  al, [di][bx]
add  [bp]+[di], cx
sub  [di+bp], dh
```

(5) with the double pointers [bx+si], [bx+di], [bp+si], [bp+di] (which are enclosed in square brackets) plus a constant.

```
mov  cx, [bx][si+57]
xor  al, 45+[di+23][bx+15]-94
add  [bp]+[di]-444, cx
sub  [6+di+bp]-5, dh
```

These are ALL the addressing modes allowed on the 8086. As for the constants, it is the ASSEMBLER'S job to resolve all numbers in the expression into a single constant. If your expression won't resolve into a constant, it is between you and the assembler. It has nothing to do with the 8086 chip.

We can consolidate all this information into the following list:

All the following addressing modes can be used with or without a constant:

```
variable_name  (+constant)
[bx]     (+constant)
[si]     (+constant)
[di]     (+constant)
[bp]     (+constant)
[bx+si]  (+constant)
[bx+di]  (+constant)
[bp+si]  (+constant)
[bp+di]  (+constant)
```

This is a complete list.

Thus, you can access a variable by name or with one of the eight pointer combinations. There are no other possibilities.

One thing that may confuse you about an addressing statement is all the plusses and minuses. As an example:

```
mov  cx, -45+27[bx+22]+[-195+di]+23-44
```

the total address is:

```
-45+27[bx+22]+[-195+di]+23-44
```

When the 8086 performs this instruction, it will ADD (1) BX (2) DI and (3) a single constant. That single constant can be a positive or a negative number; the 8086 will ADD all three elements. The '+' in front of 'di' is for convenience of the assembler only; [-195-di] is illegal and the assembler will generate an error. If you actually want the negative of what is in one of the registers, you must negate it before calling the addressing instruction:

```
neg  di
mov  cx, -45+27[bx+22]+[-195+di]+23-44
```

once again, the only allowable forms are +[di], [di] or [+di]. Either -[di] or [-di] will generate an assembler error.

If you ever see a technical description of the addressing modes, you will find a list of 24 different machine codes. The reason for this is that:

```
[bx]
[bx] + byte constant
[bx] + word constant
```

are three different machine codes. Here is a listing of the same machine instruction with the three different styles:

```
    MACHINE CODE             ASSEMBLER INSTRUCTION

     03 04                     add   ax, [si]
     03 44 1B                  add   ax, [si+27]
     03 44 E5                  add   ax, [si-27]
     03 84 5BA7                add   ax, [si+23463]
     03 84 A459                add   ax, [si-23463]
```

(27d = 1Bh , 23463d = 5BA7h). The first byte of code (03) is the add (word) instruction. The second byte is the addressing code, and the third and fourth bytes (if any) are the constant (in hex). Addressing code 04 is: (ax, [si]). Addressing code 44 is: (ax, [si] + byte constant). Addressing code 84 is: (ax, [si] + word constant). The fact that there are three different machine codes is of concern to the assembler, not to you. It is the assembler's job to make the machine code as efficient as possible. It is your job to write quality, robust code.

SEGMENT OVERRIDES

So far, we haven't talked about segment registers. You will remember from the last chapter that the 8086 assumes that a named variable is in the DS segment:

```
    mov  ax, variable1
```

If it isn't, the Microsoft assembler puts the correct segment override in the machine code. The segment overrides are:

```
    SEGMENT OVERRIDE         MACHINE CODE (hex)
         CS                       2E
         DS                       3E
         ES                       26
         SS                       36
```

As an example:

```
    MACHINE CODE        ASSEMBLER  INSTRUCTIONS

    2E: 03 06 0000 R      add   ax, variable3
    26: 2B 1E 0000 R      sub   bx, variable2
    31 36 0000 R          xor   variable1, si ; no override
    36: 21 3E 00C8 R      and   variable4, di
```

when the different variables were in segments with different ASSUME statements. If you don't remember this, you should reread the section on overrides in the last chapter. Remember, the colon is in the listing only to tell you that we have a segment override. The colon is not in the machine code.

What about pointers? The natural segment for anything with [bp] is SS, the stack segment.{1} Everything else has DS as its natural segment. The natural segments are:

(1) DS

```
variable + (constant)
[bx] + (constant)
[si] + (constant)
[di] + (constant)
[bx+si] + (constant)
[bx+di] + (constant)
```

(2) SS

```
[bp] + (constant)
[bp+si] + (constant)
[bp+di] + (constant)
```

where the constant is always optional. Can you use segment overrides? Yes, in all cases.{2} Here is some assembler code along with the machine code which was generated.

MACHINE CODE	ASSEMBLER INSTRUCTIONS
26: 03 07	add ax, es:[bx]
2E: 01 05	add cs:[di], ax
36: 2B 44 11	sub ax, ss:[si+17]
2E: 29 46 00	sub cs:[bp], ax
3E: 33 03	xor ax, ds:[bp+di]
26: 31 02	xor es:[bp+si], ax
26: 89 43 16	mov es:[bp+di+22], ax
03 04	add ax, [si]
03 44 1B	add ax, [si+27]
03 84 A459	add ax, [si-23463]
26: 03 04	add ax, es:[si]
26: 03 44 1B	add ax, es:[si+27]
26: 03 84 A459	add ax, es:[si-23463]

(17d = 11h, 22d = 16h, 27d = 1Bh, -23463d = 0A459h). The first number (which is followed by a colon) is the segment override that the assembler has inserted in the machine code. Remember, the colon is in the listing to inform you that an override is

---

1 We will see why when we look at subroutines. BP is called the base pointer [bp] and is used in a special way.

2 There are some special instructions for two independent pointers which we will cover at the end of the book. These allow segment overrides but force the override to refer to the first pointer.

involved; it is not in the machine code itself.

Unfortunately, when you use pointers you must put the override into the assembler instructions yourself. The assembler has no way of knowing that you want an override. This can cause some truly gigantic errors (if you reference a pointer seven times and forget the override once, the 8086 will access the wrong segment that one time), and those errors are extremely difficult to detect.

As you can see from above, you put the override in the instructions by writing the appropriate segment (CS, DS, ES or SS) followed by a colon. As always, it is your responsibility to make sure that the segment register holds the address of the appropriate segment before using an override.

We have talked about two different types of constants in the chapter, a constant which is part of the address:

```
mov  ax, [bx+17]
add  [si+2190], dx
and  [di-8179], cx
```

and a constant which is a number to used for an arithmetical or logical operation:

```
add  ax, 17
sub  dl, 45
add  dx, 22187
```

They are both part of the machine instruction, and are unchangeable (true constants). This machine code is going to be difficult to read, so just look for (1) the constant DATA and (2) the constant in the ADDRESS. All constants in the assembler instructions are in hex so that they look the same as in the listing of the machine code. Here's a listing of different combinations.

1. Pointer + constant as an address:

```
MACHINE CODE            ASSEMBLER INSTRUCTIONS
 01 44 1B                 add   [si+1Bh], ax
 29 85 0A04               sub   [di+0A04h], ax
 30 5C 1F                 xor   [si+1Fh], bl
 20 9E 1FAB               and   [bp+1FABh], bl
```

2. Arithmetic instruction with a constant:

```
MACHINE CODE            ASSEMBLER INSTRUCTIONS
 05 1065                  add   ax, 1065h
 2D 6771                  sub   ax, 6771h
 80 F3 37                 xor   bl, 37h
 80 E3 82                 and   bl, 82h
```

3. Pointer + constant as an address; arithmetic with a constant

```
MACHINE CODE            ASSEMBLER INSTRUCTIONS
 81 44 1B 1065            add   [si+1Bh], 1065h
 81 AD 0A04 6771          sub   [di+0A04h], 6771h
 80 74 1F 37              xor   [si+1Fh], BYTE PTR 37h
 80 A6 1FAB 82            and   [bp+1FABh], BYTE PTR 82h
```

You will notice that the ADD instruction (as well as the other instructions) changes machine code depending on the complete format of the instruction (byte or word? to a register or from a register? what addressing mode? is AX one of the registers?). That's part of the 8086 machine language encoding, and it makes the 8086 machine code extremely difficult to decipher without a table listing all the options.

OFFSET AND SEG

There are two special instructions that the assembler has - offset and seg. For any variable or label, offset gives the offset from the beginning of the segment, and seg gives the segment address. If you write:

```
mov  ax, offset variable1
```

the assembler will calculate the offset of variable1 and put it in the machine code. It also signals the linker and loader; if the linker should change the offset during linking, it will also adjust this number. If you write:

```
mov  dx, seg variable1
```

The assembler will signal to the linker and the loader that you want the address of the segment that variable1 is in. The linker and loader will put it in the machine code at that spot. You don't need to know the name of the segment. The linker takes care of that. We will use the seg operator later.

LEA

LEA (load effective address) is a completely different animal. It allows you to use any addressing mode to put an address in a register. One of the addressing modes covered before was for the following code:

```
xor  dx, 45+[di+23][bx+15]-94
```

The 8086 added DI, BX and the constant to calculate the address. It then XOR'ed the variable at that address with DX. If you write:

```
lea  dx, 45+[di+23][bx+15]-94
```

the 8086 will add DI, BX and the constant to calculate the address. It will then put the ADDRESS in DX. LEA can use any

addressing mode to calculate an address. The machine code looks almost the same:

```
    MACHINE CODE                  ASSEMBLER INSTRUCTIONS

     33 51 F5                  xor dx, 45+[di+23][bx+15]-94
     8D 51 F5                  lea dx, 45+[di+23][bx+15]-94
```

The first byte of the machine code is the instruction and the second and third byte are the addressing mode.

You almost never need LEA. It is slower than:

```
    mov  dx, offset variable1
```

However, when the addressing gets complicated (perhaps 1% of the time), it's nice to have. Remember, it will calculate ANY 8086 addressing mode.

Let's run a program so we can see what actually happens with LEA

```
;lea.asm
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
variable1 dw ?
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + START CODE BELOW THIS LINE
       ; reg style
       mov   si_byte, 1   ; signed
       lea   ax, ax_byte
       call  set_reg_style

       mov   bp, 0  ; clear unused registers
       mov   di, 0

      ;lea and mov show the two ways to address variable1
       lea   ax, variable1  ; effective address
       mov   bx, offset variable1  ; offset
       call  show_regs_and_wait

lea_loop:
       mov   si, 0  ; clear registers
       mov   dx, 0
       mov   cx, 0
       mov   bx, 0
       mov   ax, 0
       call  show_regs

       call  get_unsigned ; unsigned for bx
       mov   bx, ax
       mov   ax, 0   ; blank ax
       call  show_regs

       call  get_signed ; signed for si
       mov   si, ax

       mov   ax, 0   ; blank ax
```

```
        lea   cx, [bx+si]+100      ; addresses to cx and dx
        lea   dx, [si+bx-100]
        call  show_regs_and_wait

        jmp   lea_loop
 ; + + + + END CODE ABOVE THIS LINE
```

The first part of the program shows that LEA and MOV give the same offset address. Then we enter the loop. It gets an unsigned number, puts it in BX, gets a signed number, puts it in SI, then uses LEA to calculate [bx+si+100] and [bx+si-100]. The plus and minus 100 is simply to show you a difference of 200 in the two results. BX and SI could also have contained (1) both signed numbers or (2) both unsigned numbers. It doesn't make any difference. This program has a signed and an unsigned number for variety. Of special interest to you shold be when [bx+si] is within 100 of 65536 (or 0). One of the results will be > 0 while the other result will be < 65536 The address value wraps around from 65535 -> 0. Note that with minor alteration, this program can be used to look at ANY addressing mode that uses pointers.

You should make two executable files for this. First:

```
link  lea+asmhelp
```

and the second:

```
link asmhelp+lea
```

Give them different names and run them. Note the offset values for:

```
lea  ax, variable1
mov  bx, offset variable1
```

With lea+asmhelp you should have an offset of 8 for variable1 since there are 8 bytes in the array (ax_byte, bx_byte, etc.). This array appears before variable1 in the data segment. When you link it the other way (asmhelp+lea), all the data for asmhelp.obj is in front of your data and the offset should be something completely different for variable1.

SUMMARY

These are the natural (default) segments of all addressing modes:

(1) DS

```
variable + (constant)
[bx] + (constant)
[si] + (constant)
[di] + (constant)
[bx+si] + (constant)
[bx+di] + (constant)
```

(2) SS

```
[bp] + (constant)
[bp+si] + (constant)
[bp+di] + (constant)
```

Where the constant is optional. Segment overrides may be used. The segment overrides are:

SEGMENT OVERRIDE	MACHINE CODE (hex)
CS:	2E
DS:	3E
ES:	26
SS:	36

OFFSET

The reserved word 'offset' tells the assembler to calculate the offset of the variable from the beginning of the segment.

```
mov  ax, offset variable2
```

SEG

The reserved word 'seg' tells the assembler, linker and loader to get the segment address of the segment that the variable is in.

```
mov  ax, seg variable2
```

LEA

LEA calculates an address using any of the 8086 addressing modes, then puts the address in a register.

```
lea  cx, [bp+di+27]
```

CHAPTER 12 - MULTIPLE WORD ARITHMETIC I

Let's review the LOOP instruction. We often want to repeat an action a specific number of times. In a FOR loop, we write:

```
    FOR I = 1, 10
```

That means we want to repeat the code that follows ten times. The 8086 has an instruction for this, called the loop instruction. It looks like this:

```
    mov  cx, 10
label17:
    ...
    (a bunch of code)
    ...
    loop label17
```

The count MUST be in the CX register. This is the only register you can use for this. When the machine sees the loop instruction, it decrements the CX register by one, LEAVING ALL FLAGS ALONE, and if the result is not zero, it loops back to the label. If the result is zero, it falls through the loop. One problem we might have with this instruction is if you enter it with CX = 0, it is going to loop 65,536 times. Intel provided a second instruction to avoid this - JCXZ (jump if CX is zero). You put it right before the loop for insurance.

```
    jcxz label19
label17:
    ...
    (a bunch of code)
    ...
    loop label17
label19:
```

Obviously, in our first example this instruction is not necessary because we set CX to 10 just before entering the loop.

If you have seen the list of 8086 instructions, you will have noticed lots of strange looking add and subtract instructions. Why are they there? In this chapter we will look at ADC and SBB. The others will come in later chapters.

How do engineers decide what a reasonable size for a number is? When they started making 8 bit machines (the maximum unsigned number is 255) did they go out on the streets and take a poll to find out if 255 was the maximum number that people used? No, they didn't even think about what people needed. It was a question of what the technology would allow at that time.

Similarly, at the time the 8086 was engineered, 16 bits was pushing the limits of the technology. But 65,535 doesn't really cut the mustard. If those are 65,535 pennies, that isn't even your yearly food bill, let alone the cost of housing.

The 8086 instructions give us the option of making integers of whatever size we want. Because it is done in software, it is slower, but if we are doing thousands or tens of thousands of additions instead of hundreds of thousands or millions, it won't be a great inconvenience.{1}

ASMHELP.OBJ is set up to input 2 word (4 byte) and 4 word (8 byte) numbers so those are what we are going to use. 4 byte numbers are up to +/- 2 billion and 8 byte numbers are up to 9X10exp18. Those should be large enough.

The first instruction is ADC, add with carry. When you add by hand, you add everything in the right column, then carry to the next column left, repeating this over and over. We don't need to do it column by column, but it is necessary to do it word by word. In the data section we need:

```
variable1  dq  ?
variable2  dq  ?
```

and the code for a 4 word (8 byte) addition is the following:

```
    lea  si, variable1
    lea  di, variable2
    mov  ax, [di]                ; first addition
    add  [si], ax
    pushf                    ; save the flags
    mov  cx, 3
    add  si, 2               ; go to next higher word
    add  di, 2

long_add_loop:               ; next three additions
    mov  ax, [di]
    popf                     ; restore the flags
    adc  [si], ax
    pushf                    ; save the new flags
    add  di, 2
    add  si, 2
    loop long_add_loop

    popf                     ; pop the flags off the stack
```

ADC is the same as ADD, but it looks at the carry flag - if the carry flag is 1, it adds 1 to the result; if the carry flag is zero, it does nothing to the result. This works for both signed and unsigned numbers. If you don't believe it, you should go back

---

1 As a benchmark, it took a moderately slow PC 6.5 seconds to do 100,000 eight byte additions. The same PC can do over a million two byte additions in 6 seconds.

to the introductory chapter with the base 10 machine and look at long additions.

Notice PUSHF and POPF. These are special instructions called push flags and pop flags. Rather than pushing an arithmetic register on the stack, pushf pushes the register containing the flags. Popf pops them back into the flags register. This is necessary because ADC looks at the carry flag and the ADD instructions in the loop:

```
    add  di,2
    add  si,2
```

might change the carry flag. The last POPF after the loop is to get it off the stack (anything we put on the stack we need to take off the stack).

The first addition is a normal addition, the last three take the carry into account. We are moving the pointers a word at a time. Because the 8086 doesn't allow both operands to be in memory, we need to move one into a register. After the addition is performed, the result is in memory. We can discard what is in the register.

Notice that the first half of the code looks almost the same as the code inside the loop. If we could only use ADC instead of ADD, we could put the first addition inside the loop. It is possible to do this. There is another instruction, CLC, which clears the carry flag. Recall that if the carry flag is 0, ADC does nothing different from ADD. Therefore, we can have:

```
    lea  si, variable1
    lea  di, variable2
    mov  cx, 4              ; 4 additions in loop
    clc                     ; set cf to zero
    pushf                   ; save the flags

long_add_loop:
    mov  ax, [di]           ; word to a register
    popf                    ; restore the flags
    adc  [si], ax           ; register + memory
    pushf                   ; save the flags
    add  di, 2
    add  si, 2
    loop long_add_loop

    popf                    ; pop the flags off the stack
```

It's the same code. The number of loops was increased from 3 to 4, and the carry flag was cleared to insure that the first addition would have nothing extra added. Here is the basic program.

```
; - - - - - PUT DATA BELOW THIS LINE
variable1  dq  ?
variable2  dq  ?
; - - - - - PUT DATA ABOVE THIS LINE
```

```
; - - - - - PUT CODE BELOW THIS LINE

    call  show_regs

outer_loop:
    lea  ax, variable1            ; get the variables
    call get_signed_8byte
    call print_signed_8byte
    lea  ax, variable2
    call get_signed_8byte
    call print_signed_8byte

    lea  si, variable1            ; set the pointers
    lea  di, variable2
    mov  cx, 4                    ; loop 4 times (4 words)
    clc                           ; clear the cf
    pushf                         ; save the flags

long_add_loop:
    mov  ax, [di]             ; word to a register
    popf                      ; restore the flags
    adc  [si], ax             ; register + memory
    pushf                     ; save the flags
    add  di, 2
    add  si, 2
    loop long_add_loop

    popf                      ; restore the flags

    lea  ax, variable1
    call print_signed_8byte

    jmp  outer_loop

; - - - - - PUT CODE ABOVE THIS LINE
```

The calls to get_signed_8byte are followed by print_signed_8byte. This is so you can see what you have actually typed in. It will be neat and with commas, so it will be much easier to read. Everything else is the same as before.

As an aside, let's talk about commas. Though we can get along without commas if we have a 4 digit number, There is no reason to do without them when using larger numbers. I find printer output that has 15 digit floating point numbers without commas not only hard to read but also silly. It's not that the computer is incapable of putting in commas, it's that the prople who wrote the programs couldn't be bothered with doing 2 hours of work to save the users lots and lots of aggrivation. Therefore, for all large number input (that is, larger than 1 word) you can use commas. They don't even have to be in the right place, the computer ignores them. All the following are the same number:

```
723469746
723,4,69746
72,3,46974,6
```

```
7,2,3,4,6,9,7,4,6
723,469,746
```

The program strips all commas out of the line and then looks at the input. All large output has the commas in the right spot. In order to enlist you in the crusade to stamp out unreadable input and output, the summary at the end of this chapter has the C code necessary for stripping commas. This is my contribution to world culture.

If you have played with the signed addition program, all we need to do to make it unsigned is to change all the get_signed_8byte calls to get_unsigned_8byte calls. Change the print calls to unsigned also. Run the program.

Now, let's try subtraction. How much code do we have to alter to make it a subtraction program? The answer is - one word. Just change the ADC (add with carry) to SBB (subtract with borrow). SBB learns from the CF flag whether the last subtraction had to borrow, and tells the next subtraction via the CF flag whether it has had to borrow. Change it, and try it out. (Yes, really do it, don't just pretend that you are going to do it).

Why does this program work with exactly 8 bytes? Because we tell the loop via CX that it is 4 words long. If we put 2 in CX, the loop will think that the number is 2 words (4 bytes) long, and if we put 17 in CX, the loop will think that the number is 17 words (34 bytes) long. In fact, this little snippet of code can do either signed or unsigned addition or subtraction of any number of words simply by altering one number and one word of code.

The code as it stands has only one shortfall. Remember from our earlier subtraction that we might want to have an INTO (interrupt on overflow) instruction after signed addition and subtraction. It needs to be after the last addition (or subtraction). All we need to do is put it after the POPF just below the loop. At this point the flags show the condition right after the last addition or subtraction:

```
    loop long_add_loop

    popf                ; pop the flags off the stack
    into                ; interrupt if overflow is set
```

SUMMARY

ADC (add with carry) is used for multiple word arithmetic. It adds two numbers along with plus the value in CF, the carry flag (1 if the flag is set and 0 if the flag is cleared). CLC (clear the carry flag) should be used to clear CF before the first addition. After the addition, the flags register should be pushed in order to store CF unless it is certain that CF will not be effected by the rest of the loop.

```
popf
adc  [di], ax
pushf
```

SBB (subtract with borrow) is used for multiple word arithmetic. It subtracts one number from the other and subtracts the value in CF to account for any borrows from the right. CLC (clear the carry flag) should be used to clear CF before the first subtraction. After the subtraction, the flags register should be pushed in order to store CF unless it is certain that CF will not be effected by the rest of the loop.

```
popf
sbb  [di], ax
pushf
```

These operations have the typical 5 possibilities:

1) register, register
2) register, memory
3) memory, register
4) memory, constant
5) register, constant

You may manually control the value in CF with STC and CLC. STC (set the carry flag) sets the value to 1, while CLC (clear the carry flag) sets the value to 0. These are used for initial settings of multiple word operations.

In order to store the values in the flags register you use PUSHF (push the flags) until you need them again. At that time you can get them back with POPF (pop the flags).

STRIPPING COMMAS IN C

In order to get rid of commas, you need some discipline in what kind of data entry you have. Specifically, you can't enter large numbers on the same line as text strings because text strings are likely to have commas that should be kept. This is hardly a major restriction. You can have as many numbers as you want on the same line since in C you must have whitespace between pieces of data.

We will take all commas out of the line. You can retrofit most old programs with almost no change.

The only time that you need this capability is if you are getting something from the keyboard, and what you probably have in the program is:

```
    scanf ( "format string", variables );
```

The method is (1) import a text string as a single string, (2) strip the commas, and (3) use sscanf instead of scanf.

```
    char buffer[80] ;

    fgets (buffer, 80, stdin) ;
    strip_commas (buffer);
    sscanf (buffer, "format string", variables) ;
```

Both "format string" and the variables remain unchanged when you switch from scanf to sscanf. You might want to check for EOF with fgets.

Heres the program:

```
strip_commas (buffer)
char   *buffer ;
{
    char *ptr1, *ptr2 ;

    ptr1 = ptr2 = buffer ;
    while (1)
    {
         if ( *ptr2 == ',')  /* skip commas */
         {
              ptr2++ ;
              continue ;
         }
         /*move, increment, and check for 0 */
         if (!(*ptr1++ = *ptr2++))   /* this is '=', not '==' */
              break ;
    }
    return ;
}
```

CHAPTER 13 - MULTIPLE WORD ARITHMETIC II

We have just done multiple word addition and subtraction, which are easy. Now we have multiplication and division. We are going to multiply and divide long numbers by a one word (2 byte) number. Multiplying multiple-word numbers by multiple-word numbers is complex and time consuming but can be done. Dividing by a multiple-word number is an entirely different ballgame.{1}

We'll do unsigned numbers first, then in a later chapter add the code we need for signed numbers. The core routine is the same.

UNSIGNED MULTIPLICATION

If you multiply an n digit number by an m digit number, there is a possibility of n+m digits in the result. 863 is 3 digits, 4975 is 4 digits, 863 X 4975 = 4,293,425 is 7 digits = 4 + 3. We will be multiplying an 8 byte number by a 2 byte number, so we'll need 10 bytes for the possible maximum result. Here's the code:

```
; - - - - - - - - ENTER DATA BELOW THIS LINE
multiplicand      dq    ?
multiplier        dw    ?
result            db    10 dup (?)

; - - - - - - - - ENTER DATA ABOVE THIS LINE

; - - - - - - - - ENTER CODE BELOW THIS LINE

outer_loop:
    lea  ax, multiplicand    ; load multiplicand
    call get_unsigned_8byte
    call print_unsigned_8byte
    call get_unsigned        ; unsigned word to multiplier
    mov  multiplier, ax


    lea  si, multiplicand    ; load pointers
    lea  bx, result

    mov  cx, 4           ; number of words
    sub  di,di           ; clear di

mult_loop:
```

---

1 For those of you with a hankering for large multiplication and division, I have included subroutines which can multiply and divide numbers of any length in a file called MISHMASH.DOC. It is in \EXTRAFILE. You will need to finish all the chapters before looking at it, since it uses things that you don't know about yet.

```
    mov  ax, [si]       ; multiplicand to ax
    mul  multiplier     ; {2}
    add  ax, di         ; high word from last multiplication
    jnc  store_result
    inc  dx             ; {3}
store_result:
    mov  [bx], ax       ; store 1 word of result.
    mov  di, dx         ; save high word for next multiplication
    add  si, 2          ; increment pointers
    add  bx, 2
    loop mult_loop

    mov  [bx], di       ; move last word of result

    mov  ax, [bx]
    call print_hex
    lea  ax, result
    call print_unsigned_8byte
    jmp  outer_loop

; - - - - - - - - ENTER CODE ABOVE THIS LINE
```

There are two different input calls, an 8 byte one and a 2 byte one. Inside the loop we store the high word from the multiplication in DI and then add it to the next result. This is the same as when you multiply single digits in base 10 (9 X 7 = 63 carry the 6). Note that when you add DI, there can be a carry from AX to DX, but there can be no carry out of DX. After we drop out of the loop, we need to put the last word in result. We take it from DI, but we could take it from DX if we wanted. Finally, the printing. Print_unsigned_8byte can't print the whole result, so we are printing the high word in hex form. If those top two bytes are non zero, what 'print_unsigned_8byte' prints will be incorrect because it is missing the top 2 bytes. Note once again that the only thing constraining this program to an 8 byte number is the 4 that we put in CX - change that number and you can do any size number that you want.

Run a bunch of numbers through this, including a couple that have more than a 20 digit result.

UNSIGNED DIVISION

Division is done the same way in the software as it is done with pencil and paper, starting at the left and working right. On the computer, this means starting with the high order word and working down.

---

2 It would be about 3% faster to have this in a register, but unfortunately we are out of registers.

3 Do we need to check DX for a carry here? No. The maximum multiplication is FFFFh X FFFFh. The result is FFFE 0001h. That means that DX is a maximum FFFEh. If you add one, that's FFFFh, and no carry occurs.

```
; - - - - - - - - ENTER DATA BELOW THIS LINE
dividend    dq   ?
divisor     dw   ?
quotient    dq   ?
remainder   dw   ?
; - - - - - - - - ENTER DATA ABOVE THIS LINE

; - - - - - - - - ENTER CODE BELOW THIS LINE

outer_loop:

    lea  ax, dividend        ; get dividend
    call get_unsigned_8byte
    call print_unsigned_8byte
    call get_unsigned        ; get divisor
    mov  divisor, ax

    lea  si, dividend + 6    ; start at the top
    lea  bx, quotient + 6
    mov  di, divisor
    mov  cx, 4               ; number of words
    sub  dx, dx              ; clear dx for first division

division_loop:
    mov  ax, [si]            ; dividend word to ax
    div  di                  ; {4}
    mov  [bx], ax            ; word of result to quotient
    sub  si, 2               ; decrement the pointers
    sub  bx, 2
    loop division_loop

    mov  remainder, dx
    mov  ax, remainder
    call print_unsigned
    lea  ax, quotient
    call print_unsigned_8byte

    jmp  outer_loop

; - - - - - - - - ENTER CODE ABOVE THIS LINE
```

That's it? Yup. The division instruction is designed to work effeciently and simply. We start with the most significant digits, divide, put the quotient in the variable "quotient",

---

4 After this division, the quotient is in AX and the remainder is in DX. Say, aren't we going to do anything with the remainder? There's nothing in the code about DX until we get out of the loop. In fact, we ARE doing something with the remainder. Just like division with pencil and paper, when you have a remainder, you bring it down to the left of the next digits you are going to divide. These get divided the next time around. But we don't need to move the remainder because it's already in the right place. Pretty snappy, huh? You don't need to move anything; it all takes care of itself.

DECREMENT the pointers, and get the next word for division. After the final division, we have the remainder left in DX, so we move it to the variable "remainder". The final instructions print the remainder and the quotient.

Notice that we don't need to touch the remainder during the entire operation. The 8086 leaves it exactly where it needs to be for the next division. Using the DX register when you have single word division seems screwy, but using DX for multiple word division is both natural and elegant. The Intel people made one instruction do the work of both.

Remember from the earlier chapter on division that you can get a zero divide error if the quotient is larger than 65535. Is it possible to get a quotient larger than 65535 in this routine? NO. It is impossible to get a zero divide on anything other than a zero.{5}

Run the program and do several examples. You can even do a 0 divide if you feel like interrupting the program.

SIGNED NUMBERS

For byte or word signed multiplication and division, the 8086 changes the signed numbers into unsigned numbers, does unsigned multiplication/division, then adjusts for sign. For long numbers, we have to do these operations ourselves, so we need three sections of code. (1) change the numbers into unsigned numbers, (2) do unsigned multiplication/division and (3) adjust the signs. The routines that we have here are part two of this scheme. It will be easier to implement this once you know about subroutines, so signed division and multiplication will have to wait till later.

---

5 This is technical, so if you start getting lost, don't worry about it. How do we know that it's impossible? What we are putting in DX is the remainder (R). R is always less than the divisor (D). Let Q be the number in AX the next time around. What we are dividing is:

((R*65536) + Q) / D <= ((R*65536) + 65535) /D

since Q is less than to or equal to 65535. This is the maximum.

(((R*(65535 + 1)) + 65535) /D = (((R+1) * 65535) + R)/D (huh?)

= ((R+1) * 65535)/D + R/D

Let's do a few examples: if D = 1, R < D so R = 0 max.

= (1*65535)/1 + 0/1 = 65535 rem 0

where rem = remainder. If D = 2, R < D so R = 1 max.

= ((1+1)*65535)/2 + 1/2 = 65535 rem 1

If D = 3, R < D so R = 2 max.

= (2+1)*65535)/3 + 2/3 = 65535 rem 2

See a pattern here? R/D < 1, so the quotient can never be 65536. The maximum will always be 65535 with the remainder 1 less than the divisor. If you aren't a techie, ignore all this.

SUMMARY

For both signed and unsigned numbers, multiple word division and multiplication are based on an unsigned number routine. Signed numbers are changed into unsigned numbers, the operation is performed, and the signs of the results are adjusted.

Multiplication is done the same as for single words except that the high word from one result is saved and added to the low word of the next result, thus adding the two partial results. If this addition gives a carry, DX must be incremented.

Division operates from left to right. For the first division, DX is zeroed. After that it always contains the remainder from the last division. The quotients in AX are moved to memory one by one. At the end, the final remainder will be in DX.

CHAPTER 14 - ZOOM

There are only a couple of reasons for working at the assembler level. Perhaps you're curious about how the PC functions at the machine level. Maybe you want to optimize a time consuming section of code. Maybe you want to work easily with the DOS function calls and the BIOS calls (which will be introduced in a later chapter). Or maybe you want raw speed.

Every time that I enter:

```
>dir /w
```

and watch DOS meander its way down the screen at a leisurely pace, I think "Can't the computer go any faster?" Let's find out. This is going to be a short chapter. Make a copy of template.asm, and we'll call it zoom.asm. This is going to overwrite the entire screen 200 times.{1}

Before we write to the screen, we need to know where the screen is. When IBM designed the structure of the PC, they decided to put the memory for the monochrome card in one place and the memory for the color card in another place - apparently they thought you might want to have both a color monitor and a monochrome monitor running at the same time. The color monitor is in segment 0B800 at offset 0, and the monochrome monitor is in segment 0B000 at offset 0. We need to put the correct number into the program, so you need to know whether you have a color card or a monochrome card. If one segment number doesn't work, you can try the other.

TEMPLATE.ASM

```
; - - - - - - - - - -  START CODE BELOW THIS LINE
       call  show_regs_and_wait  ; {2}
                                 ; we are about to start
                                 ; this marks the time

       mov   ax, 0B800h       ; color seg. 0B000h is mono seg
       mov   es, ax           ; es is at video card segment
       mov   cx, 2            ; do this whole thing twice
outer_loop:
```

---

1 It is now time for you to go out and get a book about the internal structure and i/o interface of the PC. One good book is "The Peter Norton Programmer's Guide To The IBM PC", by guess who? It is clearly written and a good introduction. Another quality book is "DOS Programmer's Reference" by Terry Dettmann. It is very systematically laid out and is more techie oriented.

2 We need to initialize the video card to make sure it is in the right place. This is the easiest way. The registers themselves mean nothing to us.

```
       push  cx              ; save for outer loop instruction
       mov   cx, 100         ; 100 repeats ; zoom_loop count
       mov   al, '0'         ; 100 characters starting at '0'
       mov   ah, 07h         ; black background, white letters

zoom_loop:    ; draw the screen 100 times
       push  cx
       mov   cx, 2000        ; 80 X 25 screen is 2000 words long
       mov   si, 0           ; start at offset 0000.

inner_loop:    ; inner loop - fill the screen - 2000 words
       mov   es:[si], ax
       add   si, 2
       loop  inner_loop

       inc   al              ; next higher ASCII character
       pop   cx              ; zoom_loop count
       loop  zoom_loop

       pop   cx              ; outer_loop count
       loop  outer_loop

       call  get_continue    ; finished - this is for timing
; - - - - - - - - - - END CODE ABOVE THIS LINE
```

Show_regs_and_wait resets the video card, so we use it to make sure the video memory is set at offset 0000. It then waits for ENTER. At the end, get_continue waits for ENTER.{3} This way you can mark the beginning and the end in order to time it. We set the ES segment to the video segment. This is different depending on whether you have a monochrome card or a color card. The monochrome segment is at 0B000h and the color card is at 0B800h. You need to know which kind of card you have so you can put the right number into ES via AX.

Since the normal segment for SI is the DS segment, we need to put in a segment override to use SI with the ES segment.

We start with the ASCII character '0' and then do the next 99 characters in increasing ASCII sequence. Technically, ASCII characters end at 127, but the PC extended characters go up to 255, so we will start with ASCII 48d and end with ASCII 147d. The zoom_loop changes the character 100 different times, and the inner_loop fills the screen. That 07h in AH means that we will have white characters on a black background. The outer loop has a count of 2. This should be adjusted. If you have a medium speed machine make it 4 (400 screen fills) and if you have a high speed machine make it 8 (800 screen fills).

---

3 That is its mission in life. It is there so that you can time blocks of code. If you want to find out how long some code takes, repeat it 10,000 (or whatever is appropriate) times and put get_continue in front of it and behind it. That way you can control the start and mark the finish.

When this is done, the screen will be filled with characters so you will need to use the DOS command

```
> cls
```

to clear the screen for anything else.

When it is all assembled and linked, get a cup of coffee and a wristwatch with a second hand and we'll time it. Divide by 200 (or 400 or 800) to find out how long it takes to fill one screen. If the characters are not on your screen, you probably have the wrong segment address for your video card, so try the other one. Are you ready to time it? Then ready, set, go.

Hmmm. On my machine 200 repeats takes about 4 seconds, while the dir command takes about 1 second for one screen. That means that zoom.asm is about 50 times faster. That's the difference between someone running a 4 minute mile and someone running a 3hr. 20min. mile. This is one of the reasons people like to work at the assembler level.

CHAPTER 15 - SUBROUTINES

It is now time to talk about subroutines. If you have only used BASIC this may be difficult for you. It is assumed that you are familiar with subroutines and use them constantly in your programming.

You have been using subroutines since the very first program in this manual. When you wrote:

```
    call get_num
```

you called a subroutine in ASMHELP.OBJ. Now you are going to write subroutines yourself and have them call each other. There are different template files for programs with subroutines. They are SUBTEMP1.ASM and SUBTEMP2.ASM. We will start with SUBTEMP1. It has the entry subroutine and a space for additional subroutines. The entry subroutine is the subroutine where the operating system starts the program; it does the necessary initialization and has special code for that.

You will see some additions to the normal template file. At the top is the line:

```
    INCLUDE      \pushregs.mac
```

What this is will be explained later, but you must put the file PUSHREGS.MAC in the root directory of your current drive. You will find it in the \TEMPLATE subdirectory.

At the end of the SUBTEMP1.ASM is:

```
; + + + + + + + + + + + + START SUBROUTINES BELOW THIS LINE

; + + + + + + + + + + + + END SUBROUTINES ABOVE THIS LINE
```

This is where you will write all the subroutines except the entry subroutine which is still the same as before. All data for all subroutines still goes in the DATASTUFF segment.

Our first program will just call subroutines which will print out messages. Using SUBTEMP1.ASM, it looks like this:

```
;prog1.asm

; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
main_message  db   "This is the entry routine.", 0
sub1_message  db   "This is subroutine1.", 0
sub2_message  db   "This is subroutine2.", 0
sub3_message  db   "This is subroutine 3.", 0
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE
```

```
; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       mov   ax, offset main_message
       call  print_string
       call  sub1
       mov   ax, offset main_message
       call  print_string
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE

; + + + + + + + + + + + + START SUBROUTINES BELOW THIS LINE
;------------
sub1  proc near

       push  ax
       mov   ax, offset sub1_message
       call  print_string
       call  sub2
       mov   ax, offset sub1_message
       call  print_string
       pop   ax

       ret

sub1   endp
;------------
sub2  proc near

       push  ax
       mov   ax, offset sub2_message
       call  print_string
       call  sub3
       mov   ax, offset sub2_message
       call  print_string
       pop   ax

       ret

sub2   endp
;------------
sub3  proc near

       push  ax
       mov   ax, offset sub3_message
       call  print_string
       pop   ax

       ret

sub3   endp
; ----------
; + + + + + + + + + + + + END SUBROUTINES ABOVE THIS LINE
```

The data consists of messages to be printed by print_string. Print_string prints a zero terminated string (the number zero, not the character '0'), so there must be a zero after each message in the data segment. The entry subroutine prints a message and then calls sub1, the first subroutine, which prints a message and calls sub2 which prints a message and calls sub3.

Sub3 prints a message and then returns to sub2 which prints a message and returns to sub1 which prints a message and returns to the entry routine which prints a message and then exits. This program should print 7 messages in all. You will notice that the first thing that each subroutine does is save the value in AX, since it uses the AX register. This is the cardinal rule of robustness at the assembler level.

> IF YOU USE A REGISTER, YOU MUST SAVE ITS VALUE BY PUSHING IT ON THE STACK; YOU MUST THEN RESTORE THE VALUE JUST BEFORE EXITING.

It is impossible to overstress this. The routines which call your routine might rely on the registers remaining unaltered. If you disobey this rule and alter the registers, you'll be sorry.

Why doesn't the entry routine push and pop the registers it uses? Well, the operating system assumes the registers will contain trash upon return from the program, so it uses nothing in the data registers.

All the subroutines except the entry routine are near routines. We will only use near routines. Assemble this program, link it and run it. If it works ok, it is then time for program 2, which is the same as program1, but is in two files.

Often, we want parts of a program in different files. Perhaps parts are standard subprograms which you have already written and assembled, perhaps the total program is too large to be handled comfortably in one file, perhaps different people are writing different parts of the program. Not only must we write the programs, but we must be able to connect them. We will put the entry routine, sub2 and the associated data in subtemp1.asm. We will put sub1, sub3, and the associated data in subtemp2.asm.

Take a look at SUBTEMP2.ASM. It is slightly different. First, it does not have the variables that you need for set_reg_style (ax_byte, bx_byte, etc.) but it does have EXTRN statements for them. This means that you can change the register style from this file. SUBTEMP1.ASM has these variables declared PUBLIC so the linker can join them correctly.{1} We will talk about the correct way to declare external data later.

SUBTEMP2.ASM has no stack segment, though there could be one. There is no entry subroutine. Therefore at the very end, you have the line:

```
    END
```

with nothing after it. In SUBTEMP1.ASM, you have

---

1. The reason for having only one set of variables for the style is so that every time you change one of the style variables, the array is updated. If you had two different arrays you could have two different sets of information for set_reg_style.

```
    END  start
```

so the assembler and linker know that the program begins at the label "start".

Let's do the two programs. Here are the data, the entry code and the subroutine code from the first file.

```
;prog1.asm
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
main_message  db   "This is the entry routine.", 0
sub2_message  db   "This is subroutine2.", 0
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
PUBLIC sub2
EXTRN  sub1:NEAR, sub3:NEAR

       mov   ax, offset main_message
       call  print_string
       call  sub1
       mov   ax, offset main_message
       call  print_string
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE

; + + + + + + + + + + + + START SUBROUTINES BELOW THIS LINE
sub2  proc near

       push  ax
       mov   ax, offset sub2_message
       call  print_string
       call  sub3
       mov   ax, offset sub2_message
       call  print_string
       pop   ax

       ret

sub2   endp
; + + + + + + + + + + + + END SUBROUTINES ABOVE THIS LINE
```

Notice that sub1 and sub3 have been declared EXTRN before they were referenced, and the EXTRN statement tells the assembler that they are both near procedures. sub2 has been declared PUBLIC so the assembler will give the address of sub2 to the linker.

Here's the data and code for the other file.

```
;prog2.asm

; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
sub1_message  db   "This is subroutine1.", 0
sub3_message  db   "This is subroutine 3.", 0
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + START SUBROUTINES BELOW THIS LINE
```

```
PUBLIC sub1, sub3
EXTRN  sub2:NEAR
;------------
sub1  proc near

       push  ax
       mov   ax, offset sub1_message
       call  print_string
       call  sub2
       mov   ax, offset sub1_message
       call  print_string
       pop   ax

       ret

sub1   endp
;------------
sub3  proc near

       push  ax
       mov   ax, offset sub3_message
       call  print_string
       pop   ax

       ret

sub3   endp
; ----------
; + + + + + + + + + + + + + END SUBROUTINES ABOVE THIS LINE
```

Here sub1 and sub3 have been declared PUBLIC and sub2 has been declared EXTRN.

Assemble both programs and then link all three.

```
    link prog1+prog2+\asmhelp.obj
```

assuming that asmhelp is in the root directory. Run it. You should have the same results as before.

We are going to do one more thing with the same two files. Without changing any of the code, we are going to put the data for prog1 in prog2 and the data for prog2 in prog1 like this.

```
;prog1
; + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
sub1_message  db   "This is subroutine1.", 0
sub3_message  db   "This is subroutine 3.", 0
; + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

;prog2
; + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
main_message  db   "This is the entry routine.", 0
sub2_message  db   "This is subroutine2.", 0
; + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE
```

So far, so good. Obviously we are going to need some more PUBLIC

statements and some EXTRN statements so the linker can link the four messages, but where do they go and what do they look like? The PUBLIC statements are the easiest. Put them in the segment where the message data appears, either before or after the data declaration.

The EXTRN statement is a little more complicated. First, all data is declared EXTRN by giving the variable name followed by a colon followed by its data type. The data types are BYTE, WORD, DWORD (4bytes), QWORD (quadword or 8 bytes), and TBYTE (10 bytes). These are the standard 8086/7 data sizes. Therefore we have:

```
    EXTRN sub1_message:BYTE, sub3_message:BYTE
```

in prog2.asm and:

```
    EXTRN main_message:BYTE, sub2_message:BYTE
```

in prog1.asm. Where do they go? In order to know that, we need to talk about segment overrides again.

You will remember from our discussion of the ASSUME statement that every time the assembler writes code with a variable, it checks the ASSUME statements to see which segment register(s) have the address of the segment that that variable is in. If we have:

```
    ASSUME cs:SEG1, ds:SEG2, es:SEG3, ss:SEG4
```

then if variable1 is in SEG2, the assembler will write no override in the code since DS is the 8086 default segment.

```
    MACHINE CODE            ASSEMBLER INSTRUCTION
       A1 0000                mov   ax, variable1
```

If variable1 is in SEG1 or SEG3 or SEG4, the assembler will write the appropriate segment override in the code.

```
       MACHINE CODE             ASSEMBLER INSTRUCTION

       2E: A1 0000                mov   ax, variable1
       26: A1 0000                mov   ax, variable1
       36: A1 0000                mov   ax, variable1
```

(By the way, those zeros just mean that the variable is at 0000 offset from the beginning of the segment).

The same thing happens when you have an EXTRN statement. The assembler associates the externally declared variable with the segment it is declared in. When the variable is used, it then goes through the same actions as if the variable were actually in that segment. Let's declare variable5 external with:

```
    EXTRN variable5:WORD
```

If we have:

```
    ASSUME cs:SEG1, ds:SEG2, es:SEG3, ss:SEG4
```

then if variable5 is declared external in SEG2, the assembler will write no override in the code since DS is the 8086 default segment.

```
    MACHINE CODE            ASSEMBLER INSTRUCTION
       A1 0000 E              mov   ax, variable5
```

If variable5 is declared external in SEG1 or SEG3 or SEG4, the assembler will write the appropriate segment override in the code.

```
       MACHINE CODE              ASSEMBLER INSTRUCTION

       2E: A1 0000 E               mov   ax, variable5
       26: A1 0000 E               mov   ax, variable5
       36: A1 0000 E               mov   ax, variable5
```

The "E" after the machine code means that the assembler knows that the variable is external and it will tell the linker so the linker can put the correct offset address at that point in the machine code.

Remember, as always, that it is your responsibility to have the correct segment address in the segment register before using a variable.

Now we know where it goes. When you declare a variable external, you must put the EXTRN statement in a segment which uses the same segment register as the EXTRN variable is going to use. If the EXTRN variable will use DS, then the segment where the EXTRN statement is must use DS. If the variable uses ES, then the segment the EXTRN statement is in must use ES. In other words, the ASSUME statement for the segment the variable is in must match EXACTLY the ASSUME statement you would write if the variable were internal, not external.{2} Normally, this is DS, but in special circumstances you might want something else. Also, if there is no segment that exactly matches what you want, then you need to create a dummy segment:

```
    DUMMY_SEG  SEGMENT
         EXTRN  variable7:QWORD
    DUMMY_SEG  ENDS
```

and make the assume statement that you want:

---

2. This means that if the segment with the EXTRN statement has more than one segment register in the assume statement:

```
    ASSUME  ds:MORESTUFF, es:MORESTUFF
```

then both those registers must be set to the segment of the external variable when using it or your results may be unreliable.

```
    ASSUME    es:DUMMY_SEG
```

What segment has DS in an ASSUME statement? DATASTUFF in both files, so that is where the EXTRN declaration goes - in the DATASTUFF segment.

```
;prog1
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
PUBLIC sub1_message, sub3_message
EXTRN  main_message:BYTE, sub2_message:BYTE
sub1_message  db   "This is subroutine1.", 0
sub3_message  db   "This is subroutine 3.", 0
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

;prog2
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
PUBLIC main_message, sub2_message
EXTRN  sub1_message:BYTE, sub3_message:BYTE
main_message  db   "This is the entry routine.", 0
sub2_message  db   "This is subroutine2.", 0
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE
```

Change the data in the two files, assemble them again and link them again:

```
    link prog1+prog2+\asmhelp.obj
```

You should get the same results as before. We are now through with these programs. Make sure you understand how to define PUBLIC and EXTRN procedures and PUBLIC and EXTRN data before going on, since we are not going to cover it again. Everything else in this chapter will be done with a single file in order to make life easier.

PASSING DATA

When you pass data to the routines in ASMHELP.OBJ, you always pass it through the AX register. The reason for this is that you needed to use these routines before you knew much about 8086 assembler language. It is solely for the convenience of beginners and is totally non-standard. In the real world, when you call a subroutine you ALWAYS pass the data on the stack, no matter which language you are using.

If you have the C statement:

```
    my_procedure (variable1, variable2, variable3) ;
```

then the C compiler will generate the following code:

```
    push variable3
    push variable2
    push variable1
```

```
call my_procedure{3}
```

The C language pushes these variables in right to left order. Before the call instruction is executed variable1 is on the top of the stack, variable2 is the next down, and variable3 is third on the stack. Is variable1 still on the stack top after the call instruction is executed? No. The call instruction pushes either one or two words on the stack. Before you go any farther with subroutines you need to know how the call and return instruction operate.

Every time you have used show_regs, both CS the code segment address and IP the instruction pointer have been displayed. What does IP do? When the 8086 is ready to execute an instruction, it takes IP, adds it to CS to calculate the total address, and gets the instruction at that address. It then immediately figures out how long the instruction is going to be and adds that amount to IP.{4} What this means is that at any time, IP points to the NEXT instruction, not the current instruction. When you execute a call, the 8086 changes IP to point to the first byte of the called subroutine, so the next instruction executed is the first byte of the called subroutine.

There are two different types of procedures, near procedures and far procedures. In a near procedure, you keep CS, the code segment register, the same. In a far procedure you change CS. So, when you call a near procedure you change one thing (IP) and in a far procedure you change two things (IP and CS).

When you want to get back from the subroutine, you need to have CS with the segment of the calling routine and IP with the address of the instruction after the call. What are the mechanics of all this? Let's take a near procedure first.

In a near call, the 8086 first changes the instruction pointer to point to the next instruction. It then pushes IP on the stack, and puts the address of the called subroutine (which is in bytes 2 and 3 of the call instruction) in IP. IP now points to the called subroutine. There is one more word (2 bytes) on the stack. At the end of the called subroutine, a NEAR return (ret) pops the top word off the stack into IP. IP then points to the instruction after the call instruction.

In a far call, the 8086 first changes the instruction pointer (IP) to point to the next instruction. It then pushes CS on the stack, followed by IP. It then loads the offset address of the called subroutine in IP and the segment address of the called subroutine in CS. This new IP is in bytes 2 and 3 of the call instruction and the new CS is in bytes 4 and 5 of the call

---

3. You C fanatics will notice that there are some initial underscores missing. Let's not confuse the issue.

4. Instructions can vary from one byte long to six bytes long, and the 8086 can tell from the first (or first and second) byte(s) how long the total instruction will be.

instruction. IP and CS now have the address of the called subroutine. The stack has two words (4 bytes) more on the stack. The old IP is the stack top and the old CS is next on the stack. At the end of the subroutine, a FAR return (ret) pops the stack top into IP, then pops the next stack item into CS. Now IP and CS point to the instruction after the call instruction.

These are two different types of call and they have two different machine codes. These are two different types of returns and they have two different machine codes.

```
    MACHINE CODE                ASSEMBLER INSTRUCTIONS

                                ; a far routine
                                ;-----
                                far_routine proc far
      CB                          ret
                                far_routine endp
                                ;-----

                                ; a near routine
                                ;-----
                                near_routine proc near
      C3                          ret
                                near_routine endp
                                ;-----


                                  ; a near and far call
       E8 0A43 R                  call near_routine
       9A 015C ---- R             call far_routine
```

The machine code for a near return is C3; for a far return it's CB. The machine code for a near call is E8; for a far call it's 9A. The near call has the address of the called routine (0A43h) in the following two bytes. The far call has the address of the the called routine (015Ch) in the next two bytes followed by the segment of the called routine. The segment address isn't there yet. It will be put there by the linker and loader, but the assembler has saved the space for the address. That's why the dashes are there. Remember, the R is there because those addresses might be relocated by the linker or the loader.

You tell the assembler whether to code a near return or far return by telling it whether it is a near or a far procedure.

```
    routine1  proc near
    routine2  proc far
```

How does the assembler know whether to code a near or far call? If it has already seen the procedure, it knows what type it is.

If it hasn't seen it yet, it uses the default type.{5} If it is an external subroutine, the assembler knows because you have written an EXTRN statement.

```
    EXTRN  routine3:NEAR, routine4:FAR
```

This EXTRN statement should appear before the call.

What if the routine appears after the call in the source file but it isn't the default type? You can override the default type.

```
    call NEAR PTR routine5
    call FAR PTR routine6
```

This is the same cumbersome syntax that we had with pointers to data, but it's the only game in town. Normally, if the subroutine appears after the call, you don't need to do anything if it is a near call but you need to put a FAR PTR override if it is a far call.

5. The default is near for what we are doing. However, Microsoft has something called "simplified" directives and the default changes in these cases.

THE STACK

Up to this time we have used the stack for temporary storage. If you want to temporarily save either a register or a value in memory, you push it:

```
push ax
push variable1
```

and if you want to get them back you pop them:

```
pop  variable1
pop  ax
```

This is always a word (2 bytes) at a time. When you pop the stack, the 8086 gives you back the words in reverse order. Thus if you push the following:

```
push variable1
push variable2
push variable3
push variable4
push variable5
```

then in order to get the data back in the same place, you need to pop in this order:

```
pop variable5
pop variable4
pop variable3
pop variable2
pop variable1
```

It pops the last thing that was pushed that hasn't been popped yet.

Nothing has been said about where the stack is or how it operates. It's time to change that. When the operating system starts a program, it looks for a stack segment. If the stack segment has been properly defined, the operating system puts the stack segment's segment address in SS (the stack segment register) and sets SP (the stack pointer) to point to the first byte AFTER the end of the stack segment. Exactly where this is depends on how large you have defined your stack segment. SS and SP are set, and there is nothing on the stack.

When you push something:

```
push dx
```

the 8086 subtracts 2 from SP (making one word of space) and puts that thing at the new address in SP. SP contains the address of

the last thing pushed.

This means that SP is decreasing, and the stack segment is filling up from back to front. In the topsy-turvy world of stacks, when you put things on the stack, the stack grows downward. What makes things especially confusing is that many book writers will picture a stack:

```
variable1
variable2
ax
dx
```

and not bother to tell you whether the stack is growing upwards or downwards or where the stack top is. In this book, the stack TOP will always be visually on the BOTTOM. High addresses will be visually up and low addresses will be visually down. You need to get used to SP decreasing as the stack gets larger, and this is the easiest way to do it. So, if you have the instructions:

```
push ax
push variable1
push si
push di
```

after these instructions, the stack will look like this:

```
             VALUE              ADDRESS

             ax                 sp + 6
             variable1          sp + 4
             si                 sp + 2
sp ->        di                 sp + 0
```

When you pop a value, the 8086 moves the word (2 bytes) at SP to the appropriate location and INCREMENTS SP by 2.

```
pop  di
```

You would now have:

```
             VALUE              ADDRESS

             ax                 sp + 4
             variable1          sp + 2
sp ->        si                 sp + 0
```

As long as you are just using PUSH and POP, this is entirely self regulating. SS is set, and SP is modified by the 8086 without you doing anything. It is now time to get more sophisticated.

In our C example:

```
my_procedure (variable1, variable2, variable3) ;
```

we generated the code:

```
    push variable3
    push variable2
    push variable1
    call my_procedure
```

What will the stack look like upon entry to my_procedure? That depends on whether my_procedure is a near procedure or a far procedure. If it is a near procedure, you will have:

```
              VALUE               ADDRESS

              variable3           sp + 6
              variable2           sp + 4
              variable1           sp + 2
    sp ->     old IP              sp + 0
```

If it is a far procedure, you will have:

```
              VALUE               ADDRESS

              variable3           sp + 8
              variable2           sp + 6
              variable1           sp + 4
              old CS              sp + 2
    sp ->     old IP              sp + 0
```

Therefore, the variables are in different places relative to SP depending on whether it is a near or a far procedure. All examples will be with near procedures, but they are all valid for far procedures if you adjust for having the old CS on the stack.

How do we access these variables? By using a pointer. We could use BX, SI or DI, but they have DS, not SS as their natural segment register. The only pointer with SS as the natural segment register is BP, the base pointer. Since we are going to use BP, we need to push its current value in order to save it:

```
    push bp
```

The stack now looks like this:

```
              VALUE               ADDRESS

              variable3           sp + 8
              variable2           sp + 6
              variable1           sp + 4
              old IP              sp + 2
    sp ->     old bp              sp + 0
```

This is the standard way to do it and this is what the stack always looks like if you follow the standard method. The standard code for setting up the stack for access is:

```
    push bp
    mov  bp, sp
```

We give BP the same value as SP, so BP also points to the top of the stack and we use BP instead of SP. We now have:

```
              VALUE               ADDRESS

              variable3           bp + 8
              variable2           bp + 6
              variable1           bp + 4
              old IP              bp + 2
    bp ->     old bp              bp + 0
```

Now, if you want to push and pop things, you can do it to your heart's content. BP will always point to the set of data that you want to work with. Let's take the average of the three variables, and print it.

```
    mov  ax, [bp+4]     ; add the three numbers
    add  ax, [bp+6]
    add  ax, [bp+8]
    mov  dx, 0          ; prepare dx for division
    mov  bx, 3          ; unsigned divide by 3
    div  bx
    call print_unsigned      ; result is in ax
```

We are using AX, BX, and DX, so we need to push them before doing this:

```
    push ax
    push bx
    push dx
```

After we are done we need to (1) pop the registers and (2) restore BP. This is also a pop.

```
    pop  dx
    pop  bx
    pop  ax
    pop  bp
    ret
```

The whole subprogram now looks like this

```
;-----
my_procedure proc near

    push bp             ; set up base pointer
    mov  bp, sp
    push ax             ; push registers
    push bx
    push dx
    mov  ax, [bp+4]     ; add the three numbers
    add  ax, [bp+6]
    add  ax, [bp+8]
    mov  dx, 0          ; prepare dx for division
    mov  bx, 3          ; unsigned divide by 3
    div bx
    call print_unsigned      ; result is in ax
```

```
    pop  dx             ; pop registers
    pop  bx
    pop  ax
    pop  bp             ; restore old base pointer
    ret

my_procedure endp
;------
```

There is only one more improvement to make. If you look at the code, it is not clear what [bp+4] refers to. We know where it is, but what is it? Therefore, we will always use EQU statements to give names to our stack variables. It will be clearer, and if you need to change the code, it is much easier to change the EQU definition than to change the stack references in the code. As usual, we follow the C convention and put EQU names in capital letters.

```
;-----
my_procedure proc near

    VAR1 EQU [bp+4]
    VAR2 EQU [bp+6]
    VAR3 EQU [bp+8]

    push bp             ; set up base pointer
    mov  bp, sp
    push ax             ; push registers
    push bx
    push dx
    mov  ax, VAR1       ; add the three numbers
    add  ax, VAR2
    add  ax, VAR3
    mov  dx, 0          ; prepare dx for division
    mov  bx, 3          ; divide by 3
    div bx
    call print_unsigned      ; result is in ax
    pop  dx             ; pop registers
    pop  bx
    pop  ax
    pop  bp             ; restore old base pointer
    ret

my_procedure endp
;------
```

This is a simple example, so it doesn't look that important to use the EQU statements. Just wait till you have more complex subroutines. By the way, this program does no error checking. (If the sum is > 65535 it will give the wrong answer).

There is still one thing to do. When we called the subroutine, we pushed the variables on the stack:

```
    push variable3
    push variable2
    push variable1
```

```
    call my_procedure
```

We now want to take them off. Do we need to pop them? No, this is trash so they go into the Great Bit Bucket. There are two ways of doing this, and this is language dependent.{1} In C, it is the STANDARD that the calling routine takes them off, and it is done this way:

```
    push variable3
    push variable2
    push variable1
    call my_procedure
    add  sp, 6          ; 3 variables = 6 bytes
```

we simply INCREASE sp by the number of bytes that we pushed on the stack. Whoof, they're gone.

If you use PASCAL or FORTRAN, then the CALLED routine must take the variables off the stack on return. How does it do that? There is yet another type of return statement:

```
    ret (6)             ; 3 variables = 6 bytes {2}
```

causes the 8086 to increase sp by 6 as the last thing it does before returning from the subroutine. Which method you use is decided by which high-level language you are using.

```
    MACHINE CODE        ASSEMBLER INSTRUCTIONS

                  ;-----
                       far_routine proc far
       CA 001A              ret (26)     ; hex 1A
       CB                   ret
                       far_routine endp
                  ;-----

                  ;-----
                       near_routine proc near
       C2 002C              ret (44)     ; hex 2C
       C3                   ret
                       near_routine endp
                  ;-----
```

Here are the four different types of returns along with the machine code. Notice that the returns which increment the stack have the increment count coded in the machine code.

You may have noticed that even in this first subroutine, pushing and popping the registers takes a lot of space. It is fairly

---

1 And a major reason that is a real pain in keester to have multi-language programs.

2 The parentheses are not necessary.

normal to use 6 registers in a subroutine. This means that you will need to write:

```
push ax
push bx
push cx
push dx
push si
push di
```

at the beginning of the subroutine and:

```
pop  di
pop  si
pop  dx
pop  cx
pop  bx
pop  ax
```

before returning. This is a lot of space and it gets boring. Also, you have to remember to pop in the exact reverse order or you will screw things up. Fortunately we have two macros to help us. The file PUSHREGS.MAC has two macros, one called PUSHREGS and the other called POPREGS.

A macro is a set of directions for generating additional assembler code before the file is assembled. That's why it is called the Microsoft Macro Assembler. You include \pushregs.mac at the beginning of the file, and then everytime the assembler sees the word PUSHREGS followed by register names it generates push instructions. Every time the assembler sees POPREGS followed by register names, it generates pop instructions. It generates actual text which is assembled later.

The form for generating those push instructions above is:

```
PUSHREGS ax, bx, cx, dx, si, di
```

the word PUSHREGS followed by the registers separated by commas. (Make sure there is no comma after the last register). This must all be on one line. PUSHREGS pushes the registers in left to right order.

The form for generating those pop instructions above is

```
POPREGS ax, bx, cx, dx, si, di
```

The registers will be popped in the REVERSE order to the way they are listed on the line, that is, in RIGHT TO LEFT order.

Notice that the order of registers is the same for both PUSHREGS and POPREGS. This is so that you may write the push part:

```
PUSHREGS ax, bx, cx, dx, si, di
```

and then use your word processor to copy the line to the end of the subroutine, changing PUSHREGS to POPREGS. This insures that

the pushes and pops will be in exact reverse order. It saves space and time, and it generates exactly the same code as if you had written all those pushes and pops in the code. Whenever we have subroutines in the future, we will always use it.

MOVING A STRING

As a final example, we will create a subroutine that moves a Pascal string from one place to another. We'll assume that both strings are in the current DS, so no segments need to be changed.

```
    move_string ( from_string, to_string ) ;
```

where from_string and to_string are the ADDRESSES of the strings. The code generated by the Pascal compiler will be:

```
    mov  ax, offset from_string
    push ax
    mov  ax, offset to_string
    push ax
    call move_string
    ; this is Pascal, so the CALLED subroutine must
    ; get rid of the variables on the stack.
```

Notice that Pascal pushes this data in left to right order, exactly the opposite of how C would handle it. After setting up BP, we have:

```
              from_string offset       bp + 6
              to_string offset         bp + 4
              old IP                   bp + 2
    bp ->     old bp                   bp + 0
```

Before coding this, you need to know the structure of a Pascal text string. The first byte (string[0]) is not text, but the text count. The second byte is the first piece of text. This means two things. First, the maximum string size in Pascal is 255, the largest count that will fit in one byte. Second, you need to move 'count + 1' bytes. 'count' is how many text bytes there are, but then you need to move the count itself. If the string is empty (count = 0) you need to move 1 byte, the count byte. Here's the code

```
; - - - - -
move_string   proc near

    FROM_ADDRESS  EQU  [bp + 6]
    TO_ADDRESS    EQU  [bp + 4]

    push bp             ; set up bp
    mov  bp, sp
    PUSHREGS  ax, cx, si, di

    mov  si, FROM_ADDRESS    ; source
    mov  di, TO_ADDRESS      ; destination
    sub  cx, cx              ; zero cx
```

```
    mov  cl, [si]           ; text byte count of source
    inc  cl                 ; add 1 for byte count itself

move_loop:
    mov  al, [si]           ; source to al
    mov  [di], al           ; al to destination
    inc  si                 ; move pointers to next byte
    inc  di
    loop move_loop

    POPREGS  ax, cx, si, di
    pop  bp
    ret (4)                 ; Pascal, so pop offsets.

move_string  endp
; - - - - - - - -
```

We still have some more to do, and we'll do it in part three of the chapter.

What if both strings are in memory but we don't know which segments they are in? In that case, the calling subroutine needs to pass both the segment and the offset for both the from_string and the to_string. Let's do this in C.

```
move_string ( from_string, to_string ) ;
```

In C, this will pass the addresses of the arrays, not the arrays themselves. C, once again, pushes these variables in right to left order. If the compiler is set up to move both the segment and the offset, then the generated C code will be:

```
mov  ax, seg to_string
push ax
mov  ax, offset to_string
push ax
mov  ax, seg from_string
push ax
mov  ax, offset to_string
push ax
call move_string
add  sp, 8                    ; 4 pushes = 8 bytes
```

On the 8086, the low two bytes are ALWAYS the offset and the high two bytes are ALWAYS the segment. Remember, SEG gets the segment starting address of the named variable.

We will do the subroutine as a near routine. After setting up BP, we will have:

```
             to_string segment        bp + 10
             to_string offset         bp + 8
             from_string segment      bp + 6
             from_string offset       bp + 4
             old IP                   bp + 2
bp   ->      old bp                   bp + 0
```

In the subroutine we will have to move the segment and the offset for each pointer. Luckily for us, there are two 8086 instructions for doing this:

LDS (load DS) loads the first two bytes into the named register and the next two bytes into DS.

LES (load ES) loads the first two bytes into the named register and the next two bytes into ES.

If we write:

```
LES  di, [bp + 8]
```

then the 8086 will load the first two bytes (bp+8 and bp+9) into DI and the next two bytes (bp+10 and bp+11) into ES. This loads

the offset and the segment in the same instruction. If we write:

```
        LDS  si, [bp + 4]
```

the 8086 will load the first two bytes (bp+4 and bp+5) into SI and the next two bytes (bp+6 and bp+7) into ES, loading both the offset and segment in one instruction.

LDS and LES allow you to load the offset into any full arithmetic register (AX, BX, CX, DX, SI, DI, BP or SP) but you can't use AX, CX or DX as addressing registers, so it only makes sense to load BX, SI, DI and BP for use as pointers. The two strings will now be addressed by DS:SI and ES:DI. DS is SI's normal segment, so we don't need to do anything, but we need a segment override for ES:DI. Here is the code for a C subroutine:

A C string ends with a 0 byte, that is with a byte having the numeric value 0. It can be any length, but we need to test each byte to find out if it is 0.

Notice that we are using (and changing) both DS and ES this time, so we have to PUSH and POP them, just like other registers.

```
; - - - - -
move_string   proc near

    FROM_POINTER  EQU  [bp+4]
    TO_POINTER    EQU  [bp+8]

    push bp
    mov  bp, sp
    PUSHREGS  ds, es, ax, si, di

    lds  si, FROM_POINTER
    les  di, TO_POINTER

move_loop
    mov  al, [si]            ; source to al
    mov  es:[di], al         ; al to destination
    inc  si                  ; pointers to next byte
    inc  di
    and  al, al              ; is al 0?
    jnz  move_loop

    POPREGS  ds, es, ax, si, di
    pop  bp
    ret                      ; calling routine pops variables

move_string  endp
; - - - - -
```

Basically, the only difference between this and the Pascal move is that (1) here we check for 0 and there we had an actual count, and (2) in Pascal we used "ret (4)" and here the calling routine does the adjustment.

DRAWING THE STACK

Each time that we have used the stack we have drawn a picture of where everything is on the stack. In case you think that this is some trivial little learning technique, I'm telling you now that at the assembler level, if you are passing variables on the stack and you don't make a diagram on a piece of paper of where everything is, you are guaranteed to consistently reference things by the wrong address. ALWAYS make a paper diagram that includes BP, ALWAYS use EQU statements, and you'll avoid a lot of mistakes.

It is now time to get more complex. Everything that follows is more advanced, so it requires some programming experience. Everything that follows is about C modules and recursion. If this gets too complicated or obscure, just skip to the summary at the end of the chapter.

I am going to give you a sample subroutine in C and then show you where all the variables go. The following is a complete C file:

```
/* a complete C file  - - - - - - - - - - - - */

int            A, B ;
static int     C, D ;
extern int     E, F ;

sample_routine ( G, H )
int G, *H ;
{
    int            I, J ;
    static int     K, L ;

    A = I ;   /* transfer the words around */
    B = G ;
    J = E ;
    F = C ;
    D = K ;
    *H = I ;

    return ;
}

/* end of C file   - - - - - - - - - - - - - - */
```

The only thing this routine does is tranfer the words around. This is so you can see where things are stored and how they are accessed. If you don't know C, the only thing you really need to know here is that '*H' means that 'H' is the ADDRESS of an integer, not the integer itself, while '*H' is the actual integer that is being addressed.

For those of you who DO know C, you need to know exactly what extern, static and static mean. extern means that it is in an

external file. The first 'static' (which is outside of any subroutine) means that the data is INTERNAL to the file; it won't be shared with other files. Variables A and B, which don't have the word 'static' are GLOBAL and will be shared with other files. The other 'static' (inside the subroutine) means that its address is fixed in memory while I and J, which are not 'static' have their addresses generated every time you call the program. Say what? Yes, that's correct, every time you call the program they can be in a different place. That's what allows recursion, and you'll see how that is implemented in a second.

Here is the equivalent program in assembler:

```
; - - - - - START DATA BELOW THIS LINE
    PUBLIC A, B
    EXTRN  E:WORD, F:WORD

A   dw   ?
B   dw   ?
C   dw   ?
D   dw   ?
K   dw   ?
L   dw   ?
; - - - - - END DATA ABOVE THIS LINE

; - - - - - START SUBROUTINE BELOW THIS LINE
sample_routine proc near

    ADDRESS_OF_H EQU [bp + 6]
    G EQU  [bp + 4]

    I EQU  [bp - 2]
    J EQU  [bp - 4]

    push bp
    mov  bp, sp          ; set up bp
    sub  sp, 4           ; save space for I and J
    PUSHREGS ax, si

    mov  ax, I           ; A = I
    mov  A, ax

    mov  ax, G           ; B = G
    mov  B, ax

    mov  ax, E           ; J = E
    mov  J, ax

    mov  ax, C           ; F = C
    mov  F, ax

    mov  ax, K           ; D = K
    mov  D, ax

    mov  ax, I           ; *H = I
    mov  si, ADDRESS_OF_H
    mov  [si], ax
```

```
    POPREGS  ax, si
    mov  sp, bp         ; readjust sp
    pop  bp
    ret            ; a C routine so calling routine pops

sample_routine endp
; - - - - - END SUBROUTINE ABOVE THIS LINE
```

This time the setup is a little longer. It is:

```
    push bp
    mov  bp, sp
    sub  sp, 4
    PUSHREGS ax, si
```

I and J are not in the data segment, so we need to make space for them somewhere, and we do it on the stack. We subtract 4 from sp to provide ourselves with 4 bytes, two for I and two for J. There are two EQU statements that say exactly where I and J will be. We also push AX and SI because we will use them. After this setup, we have:

```
              address of H        bp + 6
              G                   bp + 4
              old IP              bp + 2
    bp ->     old bp              bp + 0
              I                   bp - 2
              J                   bp - 4
              old ax              bp - 6
    sp ->     old si              bp - 8
```

We have created space for some variables below bp. This is temporary and will disappear when we leave the subroutine. We do our dummy calculations,{1} and then do the end adjustment. There are two ways to readjust sp before returning:

```
    add  sp, 4
```

just adds the amount that we subtracted, so it winds up in the same place. But the place it winds up is ALWAYS the address of the old BP, and bp is now pointing to that, so

```
    mov  sp, bp
```

does exactly the same thing.

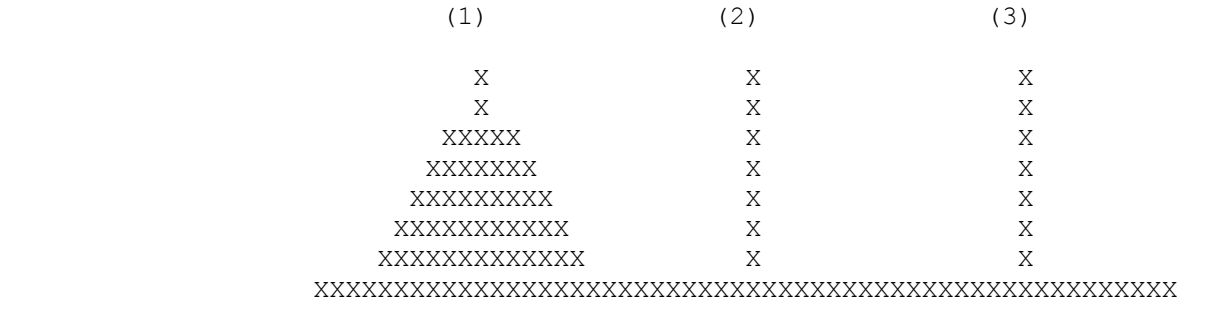
ARE YOU REALLY DEMENTED?

Yes, for those of you who are truly masochistic, we have - ta-dah! - the Towers of Hanoi in assembler language.

---

1. And that's 'dummy' in more ways than one.

The Towers of Hanoi is a game with three posts and a number of disks which have incrementally smaller diameters. At the beginning of the game, all the disks are on post one, ordered by size with the smallest on top and the largest on the bottom. For five disks it looks like this:



The object is to wind up with all the disks on post three in the same order. The game has only one rule: You may never put a larger disk on top of a smaller disk.

The general solution to this problem is that if you have posts A, B and C with N disks on post A and you want to move them to post C, first move (N-1) disks to post B, move the bottom disk from post A to post C, then move (N-1) disks from post B to post C. This works out to be the optimal recursive solution. Try it out with N = 1, N = 2 and N = 3. N = 4 is already complicated.

I won't discuss the game further or how to get the solution. If you don't know about it, you can look it up in either Doug Cooper's "Oh! Pascal" or Robert Kruse's "Data Structures and Program Design".

The fact that I need to resort to such an unususl example to illustrate recursion underlines the fundamental rule of recursion, which is:

> YOU SELDOM NEED TO USE RECURSION, BUT WHEN YOU NEED IT YOU REALLY REALLY NEED IT.

First, here is the solution in C:

```
/* the C solution - - - - - - - - - - - - - - - */

#define MAXIMUM 9

main ()
{
    int  count ;

    while (1)
    {
         printf ("Enter a number less than 10.\n" ) ;
         scanf ( "%d", &count ) ;
         if ( count > MAXIMUM )
              continue ;
```

```
          towers_of_hanoi ( count, 1, 3, 2 ) ;
     }
}
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
towers_of_hanoi ( count, from, to, via )
int  count, from, to, via ;
{
     if (count <= 0)
          return ;

     count-- ;
     towers_of_hanoi ( count, from, via, to ) ;
     printf ("Move a disk from %1d to %1d.\n" , from, to ) ;
     towers_of_hanoi ( count, via, to, from ) ;
     return ;
}
```

Notice that by letting a routine call itself, we have reduced it to just a few lines. And this is a problem that looks very complex. Here comes the assembler equivalent of the C code:

```
; + + + + + + + + + + + + + + + START DATA BELOW THIS LINE

     MAX_COUNT  EQU   9

enter_message        db  "Enter a number less than 10", 0
make_a_move_message  db  "Move a disk from "
from_byte            db  "X to "
to_byte              db  "X.", 0

; + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
outer_loop:
       lea   ax, enter_message        ; count message
       call  print_string
       call  get_unsigned_byte        ; count in al
       sub   ah, ah                   ; zero ah for 'push ax'
       cmp   al, MAX_COUNT            ; too big?
       ja    outer_loop

       ; move from post 1 to post 3 via post 2
       mov   bx, 2                    ; post 2 = 'via'
       push  bx
       mov   bx, 3                    ; post 3 = 'to'
       push  bx
       mov   bx, 1                    ; post 1 = 'from'
       push  bx
       push  ax                       ; al = count, ah = 0
       call  towers_of_hanoi
       add   sp, 8                    ; adjust the stack
       jmp   outer_loop
; + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE

; + + + + + + + + + + + + + START SUBROUTINES BELOW THIS LINE
```

```
towers_of_hanoi proc near

       VIA   EQU  [bp + 10]
       TO    EQU  [bp + 8]
       FROM  EQU  [bp + 6]
       COUNT EQU  [bp + 4]

       push  bp                  ; set up bp
       mov   bp, sp
       push  ax
       cmp   BYTE PTR COUNT, 0   ; if no disks, we are done
       jbe   exit

       dec   BYTE PTR COUNT      ; 1 less disk to move

       ; first half
       push  TO
       push  VIA
       push  FROM
       push  COUNT
       call  towers_of_hanoi
       add   sp, 8               ; adjust the stack

       ; print the message
       mov   al, FROM            ; get 'from' number
       add   al, '0'             ; convert to ascii
       mov   from_byte, al       ; put into message
       mov   al, TO              ; get 'to' number
       add   al, '0'             ; convert to ascii
       mov   to_byte, al         ; put into message
       lea   ax, make_a_move_message
       call  print_string

       ; second half
       push  FROM
       push  TO
       push  VIA
       push  COUNT
       call  towers_of_hanoi
       add   sp, 8               ; adjust the stack

exit:
       pop   ax
       pop   bp
       ret

towers_of_hanoi endp
; + + + + + + + + + + + + + END SUBROUTINES ABOVE THIS LINE
```

The main routine checks that the number you enter is not too big and then calls 'towers_of_hanoi'. After setting up BP, the stack looks like this:

```
              VIA post       bp + 10
              TO post        bp + 8
```

```
              FROM post      bp + 6
              count          bp + 4
              old IP         bp + 2
    bp    ->  old bp         bp + 0
```

We make some EQU statements to define where each variable is and set up BP. We are using only one register, so we have a single PUSH instead of using PUSHREGS. We then check to see if the count is 0. If it is we are done. If not, we decrement the count by 1 which gives us 'count - 1' and divide the problem into three parts. Part 1 calls 'towers_of_hanoi' and moves 'count - 1' disks from 'from' to 'via'. Part 2 prints a message of where to move the bottom disk. It converts the post numbers into ascii and inserts them in the string where the 'X's are.{2} Part 3 calls towers_of_hanoi again, this time moving the 'count - 1' disks from 'via' to 'to'. Assemble and link it with ASMHELP. When you run it, start with just 1 disk, then 2 then 3 etc. If you use the maximum number (9), it will print 511 lines. For N disks, you need (2 ** N) - 1 moves, so this gets very big very fast.

If you still feel no need to draw a picture of the stack or to use EQU statements, why don't you try writing this subroutine using the actual pointer values, i.e:

```
    push [bp + 6]
```

and so on. See how long it takes and see how easy it is to read once it's done. Can you get it to work correctly?

By the way, how large does the stack get? Well, if you raised the limit to 30 disks and entered 30, It would take your computer about a year, running 24 hours a day, to complete the solution (about 1 billion moves). The maximum stack size would be ((disks+1) * stack_use_per_disk). The extra 1 is for the calling routine. That is 31 * 14 bytes (including pushing IP, BP, and AX), or a mere 434 bytes.

---

2 It uses the fact that for a data declaration, a variable name has the address of the first piece of data on that line.

SUMMARY

To call a procedure you use CALL with the procedure name:

```
call subroutine1
```

Procedures may be either FAR or NEAR. If there is a mismatch between which type the assembler thinks it is and which type it really is, there will be an error, either at the assembler level for internal subroutines or at the linker level for external subroutines. You may override the default type for procedures with PTR:

```
call NEAR PTR subroutine5
call FAR PTR subroutine6
```

This is normally needed if the procedure comes after the call in the file and is not the default type.

To allow other files to use a procedure you declare it PUBLIC within its own segment.

```
PUBLIC  subroutine1
```

To use a PUBLIC procedure from another file you declare it EXTRN, stating which type it is:

```
EXTRN  subroutine1:NEAR, subroutine2:FAR
```

You should make this declaration in the code segment and you should make it before the procedure is referenced.

A procedure is defined by giving a name followed by the word 'proc' (procedure) followed by either FAR or NEAR

```
subroutine1 proc near
subroutine2 proc far
```

and a procedure is ended by giving the procedure name followed by 'endp' (end of procedure):

```
subroutine2 endp
```

One procedure must be ended before another is begun.

Data is normally passed from one procedure to another on the stack:

```
push variable1
push variable2
push variable3
call subroutine1
```

If this is done, then the called procedure references this data by using BP, the base pointer. The standard setup code is:

```
    push bp             ; save old bp
    mov  bp, sp         ; set bp to current top of stack
```

What the stack looks like at this point depends on whether it is a near or a far procedure. For a near procedure, we have:

```
                variable1           bp + 8
                variable2           bp + 6
                variable3           bp + 4
                old IP              bp + 2
    bp    ->    old BP              bp + 0
```

For a far procedure we have:

```
                variable1           bp + 10
                variable2           bp + 8
                variable3           bp + 6
                old CS              bp + 4
                old IP              bp + 2
    bp    ->    old BP              bp + 0
```

Although it is theoretically possible to access these variables by their pointer definition:

```
        mov  ax, [bp + 10]
```

It is much less error prone and much clearer to use EQU statements:

```
        VAR1 EQU [bp + 10]

        mov  ax, VAR1
```

If you are writing a recursive procedure and you need temporary variables, you can allot space on the stack for these variables:

```
        sub  sp, 6     ; room for 6 bytes of temp. variables
```

This should be done before any other pushes are done:

```
        push bp
        mov  bp, sp
        sub sp, 6
        PUSHREGS ax, bx, cx, dx
```

These variables should also be named with EQU statements, and as always, you should draw a picture of what is on the stack:

```
                variable1           bp + 10
                variable2           bp + 8
                variable3           bp + 6
                old CS              bp + 4
                old IP              bp + 2
    bp    ->    old BP              bp + 0
                VAR4                bp - 2
```

```
             VAR5                bp - 4
             VAR6                bp - 6


        VAR4 EQU  [bp - 2]
        VAR5 EQU  [bp - 4]
        VAR6 EQU  [bp - 6]
```

Data which is passed to the procedure is at a positive offset to BP while data that is created in the procedure is at a negative offset to BP.

If you have created a data area for yourself on the stack, then you must eliminate it before leaving the procedure. There are two ways of doing this. One way is to add back what you have subtracted:

```
        POPREGS ax, bx, cx, dx
        add  sp, 6
        pop  bp
        ret
```

The other way is to give SP the value in BP because this is the place where SP will wind up anyway:

```
        POPREGS ax, bx, cx, dx
        mov  sp, bp
        pop  bp
        ret
```

Use whichever one is clearer to you.

When you return from a procedure that has had data passed to it, the data must be taken off the stack. There are two ways of doing this. The C standard is that it is the calling program's responsibility to do this:

```
    push variable1
    push variable2
    push variable3
    call subroutine1
    add  sp, 6          ; 3 pushes = 6 bytes
```

The Pascal standard is that you take them off the stack on the return. There is a special return instruction for that:

```
    ret (6)             ; 3 pushes = 6 bytes
```

DATA

Data can be made available to other files and data can be accessed from other files. To make data available, declare it PUBLIC:

```
    PUBLIC variable1, variable2, variable3
```

To access PUBLIC data from other files, use an EXTRN statement which includes the data type:

```
EXTRN  variable7:BYTE, variable8:WORD, variable9:DWORD
EXTRN  variable10:QWORD, variable11:TBYTE
```

This EXTRN statement must be in a segment which has the same ASSUME segment register as will be used when accessing the data. Normally this is DS, but it can be something else. For instance, if the above EXTRN statements were in MORESTUFF and you have:

```
ASSUME es:MORESTUFF
```

then every time you access variable8:

```
mov  dx, variable8
```

the assembler will code an ES segment override.

PUSHREGS and POPREGS

When writing a subroutine, you should always save any registers that you use by pushing them.

```
push ax
push bx
push cx
```

They are then popped before returning

```
pop  cx
pop  bx
pop  ax
```

In order to save a lot of lines of code, there are two macros, PUSHREGS and POPREGS. They are designed so you may use a word processor to copy them. PUSHREGS pushes in left to right order and POPREGS pops in right to left order:

```
PUSHREGS ax, bx, cx, dx
POPREGS ax, bx, cx, dx
```

is a matched pair.

LES and LDS

LDS (load DS) loads the first two bytes into the named register and the next two bytes into DS. LES (load ES) loads the first two bytes into the named register and the next two bytes into ES.

```
        les  si, [bp+6]
        lds  di, [bp+10]
```

CHAPTER 16 - LONG SIGNED MULTIPLICATION AND DIVISION

Now that you have some subroutines under your belt, it is time to get back to multiple word arithmetic. This was put on the back burner because we needed to negate long numbers and it is more efficient to do that as a subroutine. First, let's negate a long number and then put the parts together.

To negate a number you complement it, then add 1. It looks like this:

```
NUMBER_LENGTH  EQU 4

variable1 dq   ?

     mov  si, offset variable1
     mov  cx, NUMBER_LENGTH
not_loop:
     not  WORD PTR [si]
     add  si, 2
     loop not_loop

     mov  si, offset variable1
     mov  cx, NUMBER_LENGTH
     stc                      ; set carry flag
add_loop:
     adc  WORD PTR [si], 0
     inc  si
     inc  si
     loop add_loop
```

This is straightforward. First negate, then add 1. The first add will add 1 because the carry flag is set. If there is a carry out, it will be taken care of in the next word with ADC (we add nothing but the carry). We can make this more compact and efficient with:

```
     mov  si, offset variable1
     mov  cx, NUMBER_LENGTH
     stc                      ; set carry flag
negate_loop:
     not  WORD PTR [si]
     adc  WORD PTR [si], 0
     inc  si
     inc  si
     loop negate_loop
```

Neither NOT nor INC effect CF, the carry flag, so the correct CF value will be propagated through the whole long number.

When we do negation during our multiplication, the multiplicand will be a 4 word negation while the result will be a 5 word

negation, so we will pass the length as a parameter. The call in C would look like this:

```
    negate_it ( &number, length ) ; {1}
```

On entry, the stack will look like this:

```
              length    bp + 6
              address   bp + 4
              old IP    bp + 2
    bp   ->   old BP    bp + 0
```

Here is the entire subroutine:

```
; - - - - - START SUBROUTINES BELOW THIS LINE
negate_it proc near

    NUMBER_LENGTH  EQU [bp+6]
    NUMBER_ADDRESS EQU [bp+4]

    push bp
    mov  bp, sp
    PUSHREGS cx, si


    mov  si, NUMBER_ADDRESS
    mov  cx, NUMBER_LENGTH
    stc                       ; set carry flag
negate_loop:
    not  WORD PTR [si]
    adc  WORD PTR [si], 0
    inc  si
    inc  si
    loop negate_loop

    POPREGS cx, si
    pop  bp
    ret                 ; calling routine adjusts stack

negate_it endp

; - - - - - END SUBROUTINES ABOVE THIS LINE
```

Well, so far we have the negation routine and the unsigned multiplication and division routines. What else is necessary? Only the main program, and here it is for multiplication:

```
; - - - - - START DATA BELOW THIS LINE
multiplicand       dq   ?
multiplier         dw   ?
result             dt   ?
result_sign_flag   db   ?
; - - - - - END DATA ABOVE THIS LINE
```

---

1. For you non-C people, the '&' stands for the address.

```
; - - - - - START CODE BELOW THIS LINE
outer_loop:
    mov  result_sign_flag, 0      ; assume positive
    mov  ax, offset multiplicand
    call get_signed_8byte

    test WORD PTR multiplicand + 6, 8000h   ; is it negative ?
    jz   get_next_number

    mov  ax,4                     ; negate 4 word number
    push ax
    mov  ax, offset multiplicand
    push ax
    call negate_it
    add  sp, 4                    ; clear 2 pushes off stack
    not  result_sign_flag         ; reverse sign of result

get_next_number:
    call get_signed               ; get signed multiplier
    mov  multiplier, ax
    test ax, 8000h                ; is it negative
    jz   do_the_multiplication

    neg  multiplier               ; negate
    not  result_sign_flag         ; reverse sign of result

do_the_multiplication:
    mov  ax, offset result
    push ax
    mov  ax, multiplier      ; the number, not the address
    push ax
    mov  ax, offset multiplicand
    push ax
    call multiply_it
    add  sp, 6                    ; clear 3 pushes off stack

    ; is the result negative?
    test result_sign_flag, 0FFh   ; 1111 1111 mask
    jz   print_it

    mov  ax, 5                    ; 5 word result
    push ax
    mov  ax, offset result
    push ax
    call negate_it
    add  sp, 4                    ; clear 2 pushes off stack

print_it:
    mov  ax, WORD PTR result + 8  ; top two bytes
    call print_hex
    mov  ax, offset result        ; the rest of result
    call print_signed_8byte

    jmp  outer_loop
; - - - - - END CODE ABOVE THIS LINE
```

The driver routine gets an 8 byte signed number. If the number is

negative it negates the number (to make it positive) and switches the sign of the result_sign_flag. The sign flag will either be 00h for positive or FFh for negative. It then gets a two byte signed number. If it is negative, the routine negates it and switches the sign flag. At this point both numbers are positive, so it calls the unsigned multiplication routine. At the end, it checks the result_sign_flag to see if the result should be positive or negative. If it should be negative, the routine calls negate_it one more time. Finally, the routine prints the number. The hex portion will be 0000 for positive or FFFF for negative unless the value is larger than an 8 byte signed number can hold, at which point the value of the 8 byte signed number will be incorrect.

Here's the unsigned multiplication routine which has been turned into a subroutine:

```
; - - - - -
multiply_it proc near

    RESULT_ADDRESS           EQU  [bp+8]
    MULTIPLIER_VALUE         EQU  [bp+6]
    MULTIPLICAND_ADDRESS     EQU  [bp+4]

    push bp
    mov  bp, sp
    PUSHREGS  ax, bx, cx, dx, si, di

    mov  si, MULTIPLICAND_ADDRESS ; load pointers
    mov  bx, RESULT_ADDRESS

    mov  cx, 4          ; number of words
    sub  di,di          ; clear di

mult_loop:
    mov  ax, [si]       ; multiplicand to ax
    mul  WORD PTR  MULTIPLIER_VALUE
    add  ax, di         ; add high word from last multiplication
    jnc  store_result
    inc  dx
store_result:
    mov  [bx], ax       ; store 1 word of result.
    mov  di, dx         ; save high word for next multiplication
    add  si, 2          ; increment pointers
    add  bx, 2
    loop mult_loop

    mov  [bx], di       ; move last word of result

    POPREGS   ax, bx, cx, dx, si, di
    pop  bp
    ret                 ; calling routine adjusts stack

multiply_it endp
; - - - - - - - - - - -
```

Draw a picture of the stack to verify that the EQU values are

correct. The multiplication and the negation subroutines go in the subroutine section of SUBTEMP1.ASM. The driver routine is the main routine. If you don't remember how this multiplication routine works, go back to the chapter on unsigned multiple word multiplication since the code is the same.

DIVISION

Division is the same situation. We need a driver routine, but the division itself will be the unsigned division. In division, the remainder is the same sign as the dividend, and the sign of the quotient is (dividend_sign XOR divisor_sign). If both signs are the same, the quotient is positive; if the signs are different the quotient is negative. Here's the driver routine:

```
; - - - - - START DATA BELOW THIS LINE
dividend                dq  ?
divisor                 dw  ?
quotient                dq  ?
remainder               dw  ?
quotient_sign_flag      db  ?
remainder_sign_flag     db  ?
; - - - - - END DATA ABOVE THIS LINE

; - - - - - START CODE BELOW THIS LINE
outer_loop:
       mov   quotient_sign_flag, 0            ; assume positive
       mov   remainder_sign_flag, 0

       mov   ax, offset dividend
       call  get_signed_8byte
       test  WORD PTR (dividend + 6), 8000h   ; is it negative?
       jz    get_next_number

       mov   ax,4                       ; negate 4 word number
       push  ax
       mov   ax, offset dividend
       push  ax
       call  negate_it
       add   sp, 4                      ; adjust stack
       not   quotient_sign_flag         ; switch sign of quotient
       mov   remainder_sign_flag, 0FFh ; remainder is negative

get_next_number:
       call  get_signed
       mov   divisor, ax
       test  ax, 8000h                  ; is it negative
       jz    do_the_division

       neg   divisor
       not   quotient_sign_flag       ; switch sign of quotient

do_the_division:
       mov   ax, offset remainder
       push  ax
```

```
        mov   ax, offset quotient
        push  ax
        mov   ax, divisor        ; the number, not the address
        push  ax
        mov   ax, offset dividend
        push  ax
        call  divide_it
        add   sp, 8                    ; clear 4 pushes off stack

        ; are the remainder and quotient negative?
        test  remainder_sign_flag, 0FFh
        jz    test_the_quotient
        neg   remainder

test_the_quotient:
        test  quotient_sign_flag, 0FFh      ; 1111 1111 mask
        jz    print_it

        mov   ax, 4                         ; 4 word result
        push  ax
        mov   ax, offset quotient
        push  ax
        call  negate_it
        add   sp, 4                   ; clear 2 pushes off stack

print_it:
        mov   ax, offset quotient
        call  print_signed_8byte
        mov   ax, remainder
        call  print_signed

        jmp  outer_loop
; - - - - - END CODE ABOVE THIS LINE
```

We get the dividend and check the sign. If it is negative, we (1) negate the number, (2) switch the sign of the quotient, and (3) set the remainder sign flag to negative. We get the divisor, check for negative; if it is negative we negate it and switch the sign of the quotient. We now have two unsigned numbers and do unsigned division. After division, both the quotient and remainder are adjusted for sign.

The division routine is the same as the unsigned routine before except it is now a subroutine:

```
; - - - - - - - - ENTER SUBROUTINE BELOW THIS LINE
divide_it proc near

        REMAINDER_ADDRESS         EQU  [bp+10]
        QUOTIENT_ADDRESS          EQU  [bp+8]
        DIVISOR_VALUE             EQU  [bp+6]
        DIVIDEND_ADDRESS          EQU  [bp+4]

        push  bp
        mov   bp, sp
        PUSHREGS  ax, bx, cx, dx, si, di
```

```
        mov   si, DIVIDEND_ADDRESS
        mov   bx, QUOTIENT_ADDRESS
        add   si, 6                 ; start at the top word
        add   bx, 6
        mov   di, WORD PTR  DIVISOR_VALUE
        mov   cx, 4                     ; number of words
        sub   dx, dx          ; clear dx for first division

division_loop:
        mov   ax, [si]        ; dividend word to ax
        div   di
        mov   [bx], ax        ; word of result to quotient
        sub   si, 2           ; decrement the pointers
        sub   bx, 2
        loop  division_loop

        mov   bx, REMAINDER_ADDRESS    ; store remainder
        mov   [bx], dx

        POPREGS  ax, bx, cx, dx, si, di
        pop   bp
        ret               ; calling routine adjusts the stack

divide_it  endp

; - - - - - - - - - ENTER SUBROUTINE ABOVE THIS LINE
```

Draw a picture of the stack to verify that the EQU statements are correct for a NEAR routine. The division and negation subroutines go in the SUBROUTINES section of SUBTEMP1.ASM. The driver is the main program. If you don't remember how this division works, go back to the division chapter and look it over. Try out a few numbers to make sure that it is working the way it should.

DATA INTEGRITY

One thing that may have been annoying some of you is that when the programs sent us numbers for multiplication and division we sometimes negated them, effectively changing the data in memory, but never changed them back when we were done. In an operational subroutine, you would have to do it differently. The logic would be:

```
    NUMBER NEGATIVE?    no   everything's o.k.

         yes

         make copy
         negate
         reset pointer
```

If the number is positive we won't change it. If the number is negative, we make a copy, negate the copy and use the copy for the operation.

CHAPTER 17 - INTERRUPTS

Your word processor will work on a Compaq, an IBM, an AST or any other type of PC compatible computer. Every time it wants to read a file from disk or write to a printer, it calls a DOS or BIOS function that takes care of it.{1} Yet every computer has its own versions of these subroutines, and they not only are different, they are in different places in memory. How does the word processor know where to find them? It uses interrupts.{2}

An interrupt is a glorified subprogram call. The first 1024 bytes of your computer's memory (that's from 0000:0000 to 0000:03FF) contain the addresses of all the DOS and BIOS interrupts. Which address contains the address of which subprogram was decided by the triumverate of Intel/IBM/Microsoft. We have two different sets of addresses here, so let's keep them straight. Starting at memory address 0000 there are 4 byte addresses called interrupt vectors. There is a different vector at 0000d, at 0004d, at 0008d, at 0012d at 0016d etc.; there are 256 of them in all. Each of these 256 places can hold the address of a subprogram somewhere in memory (although not all of them are used).

When you call an interrupt, the 8086 goes to the appropriate place in low memory (from 0000 to 3FFh) and finds the address of the subprogram that you want, loads it into CS and IP, and goes to that subprogram. When that subprogram is done, it goes back to the next instruction in your program after the interrupt.

How did those addresses get into the first 1024 bytes? The computer put them there when you started it up. It is one of the first things that the computer did when you turned it on. These subprograms do EVERYTHING, and you can't run the computer without them.

If you want to scroll the screen, you put the appropriate information in the 8086 registers and use:

```
    int 10h             ; decimal 16 {3}
```

The 8086 goes to the interrupt 16 address (4*16 = 64), gets the address of the program that contains the video subprograms, and

---

1. BIOS stands for basic input/output services.

2. If you haven't gotten it yet, it's time to get either "DOS Programmer's Reference" or "The Peter Norton Programmer's Guide to the IBM PC." I'm serious. They contain the information you need to make use of this chapter.

3. It is normal to use hex numbers for the interrupts, so if you read about an interrupt make sure you know if the numbers are hex or decimal.

---

goes to it. If you want to write to the printer, you put the appropriate information in the 8086 registers and call:

```
    int 21h             ; decimal 33
```

The 8086 goes to address 132 (4*33 = 132), gets the address of the DOS program, puts it in CS and IP, and starts it. If you want to get input from the keyboard, you put the appropriate information in the 8086 registers and call:

```
    int  21h            ; decimal 33
```

That's right, it is the same program as the one that does the printer. The 8086 goes to 132 (4*33), gets the program address, and goes to it.

The lowest interrupt is int 0 (address = 4*0 = 0000). The highest interrupt is int 255 (address = 4*255 = 1020).

This is an intelligent way to handle the situation. As long as everyone agrees which interrupt contains the address of which subprogram, our programs will work on any PC compatible. This is one of the things that is meant by PC compatible.

On my computer, here is a section of these addresses starting with int 1Eh (30d). High memory is at the top, low memory is at the bottom.

```
    INT #           DATA       LOCATION

                  0724h cs       146
      36          04A8h ip       144
                cs 0724h         142
      35        ip 01BDh         140
                  0724h cs       138
      34          01B0h ip       136
                cs 019Fh         134
      33        ip 05EBh         132
                  019Fh cs       130
      32          05E7h ip       128
                cs 0000h         126
      31        ip 0000h         124
                  0070h cs       122
      30          0EB8h ip       120
```

If we call int 30d, the 8086 goes to 120 (4*30), puts 0EB8h into the IP, and puts 0070h (from the next higher location) into CS. The next instruction it does will be 0070:0EB8. If we call int 35d, the 8086 goes to 140 (4*35), puts 01BDh in IP, puts 0724h (from the next higher location) in CS. The next instruction the 8086 does will be 0724:01BD. If we call int 33d, the 8086 goes to 132 (4*33), and puts 05EBh in IP, 019Fh (from the next higher location) in CS. The next instruction executed will be 019F:05EBh. The 0000:0000 for int 31d indicates that there is no int 31d (address 0000:0000 contains data, [the vectors for int 0], not a program). Make sure that you understand how this is working before you go on.

On the PC, information for the interrupts is always passed through registers, not on the stack. Each interrupt type has a specific register for each piece of information it needs. We will do a couple to see how they work.

DISPLAYING A CHARACTER

We can print a character at a time on the monitor, so we'll input a string, and then print out each character of the string individually. We will stop the printout when we see the 0 at the end of the string.

```
; - - - - - - - - - - START DATA BELOW THIS LINE
buffer  db  80 dup (?)
; - - - - - - - - - - START DATA BELOW THIS LINE
; - - - - - - - - - - START CODE BELOW THIS LINE
    call show_regs
outer_loop:
    mov  ax, offset buffer
    call get_string

    mov  si, offset buffer
inner_loop:
    mov  al, [si]
    cmp  al, 0
    je   next_string
    mov  ah, 14              ; ah contains function number
    mov  bh, 0               ; where in memory
    int  10h                 ; 33d
    inc  si
    jmp  inner_loop

next_string:
    call get_continue
    jmp  outer_loop

; - - - - - - - - - - START CODE BELOW THIS LINE
```

The program is simple. It gets a string, then checks each character for 0 (end of string), before shipping it off to the screen. I'll explain AH in a second. There are several places this character can be displayed{4}, so use show_regs to force the video screen to a certain place in memory, then put BH = 0 to tell the interrupt that the screen memory is in that place. Finally, with all the information in place, we do the interrupt. You will not print a carriage return, only the data, so we use get_continue to give us a carriage return. It's a little sloppy, but much easier.

We have lots and lots of subprograms for the disks, printer, screen, etc. There are only 255 interrupts, so it was decided at the beginning to make most of the interrupts groups of programs

---

4. Cf. one of those two books.

instead of a single program. Int 10h (16d) contains about two dozed different video subprograms. Each program is distinguished by a specific number in AH. For int 10h (16d):

```
    ah = 2h        Get cursor position
    ah = 6h        Scroll window up
    ah = Eh        (14d) Write character to screen
```

For int 21h (33d):

```
    ah = 1h        Keyboard input
    ah = 5h        Printer output
    ah = 17h       Rename a file
    ah = 2Ch       Get the time
```

As you can see, int 21h is a potpourri of subprograms. We'll do the same program as above, but with printer output. Everything is the same except that the inner loop should be changed to look like this:

```
; - - - - -
    mov  si, offset buffer
inner_loop:
    mov  dl, [si]
    cmp  dl, 0
    je   next_string
    mov  ah, 5               ; ah contains function number
    int  21h                 ; 33d
    inc  si
    jmp  inner_loop

; - - - - -
```

There is practically no change. We use DL instead of AL, have "int 21h" instead of "int 10h", and change the function number in AH to 5. Also, BH is not needed.

Leave your printer off at the beginning to see what happens. Turn your printer on and enter a string of 10 or 20 letters. Probably nothing happened. Enter another 20 or 30 letters. Nothing again. Try 50 letters this time. This time it should work. Lots of printers won't print anything unless (1) they get a carriage return (which you haven't sent) or (2) they have a backlog of more than 80 characters. If you ever use the printer interrupt, you'll have to remember that.

These interrupts which are in your program are called software interrupts. They are your interface with the peripheral devices on your machine, doing everything from disk i/o to handling memory allocation. They do all your housekeeping for you.{5} If you are going to work at this level then you should buy one of those two books. They contain all the software interrupts (and there are about a hundred of them), tell how they work and which registers to set for each interrupt. If you don't have access to

5. But they don't do Windows.

these interrupts it's like having your arms cut off - you're unable to do any i/o at all.

HARDWARE INTERRUPTS

The interrupts that we write in our programs aren't the only interrupts there are. The hardware uses interrupts to take temporary control of the computer. On a Macintosh, if you insert a disk, the program stops and the operating system reads in the disk directory. A modem that is in use will request time for doing i/o. Also, if the 8086 detects a zero divide, it will trigger a special interrupt. You have met int 4 already. It is the INTO instruction, which will trigger an interrupt if the overflow flag is set.

Your interrupts are in specific places in your code, but these machine interrupts can happen at any time. There are two lines (wires) into the 8086. One is for serious problems that need to be taken care of NOW, and it has non-maskable interrupts. The other line is for interrupts that need to be taken care of in a timely fashion, and they are maskable interrupts.

A non-maskable interrupt (NMI) is when the hardware detects that it is in deep doo doo. It sends a signal on the NMI line that says "Hey! I need an interrupt." The 8086 finishes the instruction it is processing and then IMMEDIATELY gives over control. The NMI uses the same 1024 bytes in low memory for interrupt vectors, but has its own interrupt numbers. Normally this is for very serious errors, so the interrupt program may decide to abort your program and return to the operating system; if it makes sense to, it will return to your program where your program left off.

A maskable interrupt is when a piece of hardware has some work to do. It sends a signal to the 8086 (on the INTR line), and the 8086 takes care of it when it is ready.

When the 8086 is ready depends on you. In the flags register is the IEF, the interrupt enable flag. It should always be set to 1 unless you are doing something critical. Basically, the only things that are critical are interrupts themselves and context switches. Context switches are done by the operating system in multitasking environments, so they don't concern you, and you are not writing interrupts, so they don't concern you. Therefore, always keep the IEF set. Just for your information, you set the IEF with:

```
    sti  ; set interrupt flag (interrupts enabled)
```

and clear it with:

```
    cli  ; clear interrupt flag (interrupts disabled)
```

Why do interrupt programs clear the interrupt flag? Because you could have interrupts interrupting other interrupts and wind up with scads of half finished interrupts lying around. This way,

one interrupt finishes before another can take over.

Suppose you are in the middle of the following two instructions when an interrupt hits:

```
cmp  al, 7
jne  some_label
```

If the hardware interrupt takes over after the CMP instruction but before the JNE instruction, the correctness of the result will depend on the zero flag staying the same. Can you trust it? The answer is yes. The first three things an interrupt request does (in the microcode) are:

```
push flags     ; push the flags
push old CS    ; push the code segment
push old IP    ; push the instruction pointer
```

On return, it pops the flags back in place, so they are exactly the same as just before the interrupt. You will notice that there are 3 things on the stack instead of the usual 2 in a far call. Therefore, there is a special RET instruction called IRET (return from interrupt) which pops not only IP and CS, but the flags as well. You can only use this instruction in an interrupt because it assumes that the flags register is right after CS on the stack.

DEBUGGING

Finally, if you have a debugger installed, the debugger uses two interrupts, int 1 and int 3. You code int 3 into the program by placing:

```
int
```

in the program with no number.{6} The interrupt will call the debugger. The reason for this int with no number is that afterwards, you can replace it with NOP, since they are both 1 byte instructions. Int 3 is a 2 byte instruction (they are coded differently, even though they wind up at the same place).

Int 1 is the trap instruction. In the flags register is the trap flag (TF). If TF is set, the 8086 will interrupt after the next instruction. That way, the debugger can single step through sections of code. You put Int 3 to set a breakpoint and then set TF to single step from there.

The only way to set TF from your own program is:

---

6. Though the Microsoft assembler considers this an error. You must code it as:

```
int  3
```

```
    pushf
    pop  ax
    or   ax, 0100h
    push ax
    popf
```

but you should let the debugger do it.

Just so you can get a feel for how the debugger system works, there is a pseudo-debugger called PSEUDDBG.COM in \XTRAFILE. It is not a debugger. It does nothing but tell you whether you have generated a breakpoint interrupt (int 3) or a single_step interrupt (int 1). It is memory resident. That means that once you have loaded it, it will stay in memory until you shut the machine off or reset the machine. You load it by executing:

```
    >pseuddbg
```

It will tell you that it has been loaded and then sit there waiting for interrupts. When you generate a breakpoint interrupt, PSEUDDBG will allow you to set the trap flag or just continue. When you generate a single step interrupt, PSEUDDBG will allow you to clear TF or to continue single stepping. All you need to try it out is a simple program:

```
; - - - - - ENTER CODE BELOW THIS LINE
outer_loop:
    call get_continue

    mov  cx, 4
    int  3
inner_loop:
    mov  ax, 5
    add  ax, 3
    add  ax, 7
    add  ax, 10
    loop inner_loop

    loop outer_loop
; - - - - - FINISH CODE ABOVE THIS LINE
```

Each time you start the outer loop, it will generate a breakpoint interrupt. At this point you can set TF to single step or continue. If you single step, watch IP. It will be changing, cycling through the loop till it gets to get_continue. If you start trapping through get_continue, things will get strange because get_continue stores the flags. This means that after you quit trapping, get_continue will POP some flags that have TF set and it will start trapping again. If this happens, just continue hitting 0 until you get out of the single step mode.

If you like this method of single stepping through code, there is a memory resident version of ASMHELP called HELPMEM.COM which contains show_regs and allows single stepping. It can be used in conjunction with ASMHELP.OBJ. For details consult APP1.DOC in the appendix.

SUMMARY

Software interrupts send the 8086 to the first 1024 bytes of memory where it finds the address of the DOS or BIOS subprogram that you want to go to. The correspondence between the interrupt number and the location of the address is:

```
address location = 4 * interrupt number
```

The address is stored in the next 4 bytes; first IP, then CS.

The first 5 interrupts are:

```
int 0     divide by zero
int 1     trap flag induced single stepping for the debugger
int 2     non_maskable_interrupt (NMI)
int 3     1 byte interrupt for the debugger
int 4     interrupt on overflow
```

There are two types of hardware interrupts. NMI (non maskable interrupt) interrupts are for serious problems that need to be taken care of IMMEDIATELY. They have priority and take over right after the current instruction is finished.

Maskable interrupts are hardware interrupts that should be taken care of in a timely fashion. If IEF (the interrupt enable flag) is set the maskable interrupts will go through. If IEF is cleared, the maskable interrupts must wait until it is set again. Set the IEF with:

```
sti  ; set interrupt flag
```

and clear it with:

```
cli  ; clear interrupt flag
```

The IEF should always stay set unless there is a specific reason for not allowing interrupts to happen.

If you are writing an interrupt routine, then instead of RET you use IRET (return from interrupt) which not only pops off IP and CS, but pops the flags register as well.

CHAPTER 18 - PORTS

In order to communicate with the outside world, the outside world being your printer, your disk drives, your modem etc., the 8086 uses ports. A port is an address distinct from a memory address where the i/o device can be reached.{1} When IBM set up the first PC, they decided what these port addresses would be and what the function of each bit at each address would be. Any VGA, EGA, CGA or monochrome card has to have it's ports at the same addresses as those set by IBM, and these ports have to do the same thing.

Normally, an i/o device has more than one port address. The device not only has to transfer the data, it has to tell the computer whether it is ready, find out if the computer wants to send information, confirm that the transfer was successful etc. COM1 is port addresses 3F8 - 3FF. COM2 is 2F8 - 2FF. The CGA adapter is ports 3D0 - 3DF.

In the CGA, port 3D9 sets the different colors. The 8086 writes to it, but can't read it. Port 3DA, it's neighbor, tells certain status information and is read only.

All control information for the video cards is passed back and forth through the ports. However, the video card comes into memory 50 times a second to see what it should write to the screen. The characters on the screen don't pass through the ports. The mountain comes to Mohammed.

The data does pass through a port on the way to your serial printer. The 8086 sends your printer control information through one port address and sends the data through a different port address.

The port instructions can be either with AL or AX. AL and AX are the only registers you can use, AL for a byte transfer and AX for a word transfer. There are two forms to the input instruction:

```
    in   al, port_address
or
    in   al, dx
```

port_address is a constant and is limited to 0 - 255. It is a one byte constant. DX is the only register than can be used with the second form. DX can contain any number from 0 - 65535. The

---

1. There is memory address 0000 and there is port address 0000; they are two entirely different things. You can reach memory address 0000 with the DS:SI pair 0000:0000, but DS:SI can't get to port address 0000. The instruction "out 0, al" writes the contents of al to port address 0000, but the OUT instruction can't get anywhere near memory address 0000.

---

following two pieces of code do the same thing:

```
    in   al, 155

    mov  dx, 155
    in   al, dx
```

This code moves one byte of data from port 155 to register AL.

The output instruction is similar. We have:

```
    out  port_address, al
or
    out  dx, al
```

where port_address is a constant 0-255 and DX can contain a number 0 - 65535.

```
    out  97, ax
```

moves a word from AX to port addresses 97-98. Notice that both the IN and the OUT instructions follow the DESTINATION, SOURCE ordering found in all the other 8086 instructions.

The "in ax, port_address" form is not all that useful. If you look at the addresses for the video ports, for COM1 and COM2, you will see that they are all larger than 255. Also, most equipment can be at several different addresses. A modem can be at COM1, COM2, COM3, or COM4. If the port address is hard coded into the instructions, you need 4 subprograms to handle the 4 different addresses. It is easier to find out where the modem is, then use:

```
modem_data_address dw   ?

    mov  dx, modem_data_address
    in   al, dx

and
    mov  dx, modem_data_address
    out  dx, al
```

If you aren't planning on writing device drivers this is for background information only, since all standard i/o is done through DOS or BIOS interrupts.

One last thing is the parity flag. It has been sitting on the screen with all the other flags. What is it for? When you send information over a modem, there is a large possibility of line interference. If it is text, and an occasional screw up is not too bad, using a parity check may be enough.{2} Parity can be even or odd. Even means that an even number of bits are set to 1; odd means that an odd number of bits are set to 1. This is not whether the number is even or odd, it is whether the number of 1

---

2. It is NEVER enough if you are transferring binary data or executable files.

BITS is even or odd. Here's a short list.

NUMBER	BINARY	PARITY
6	0110	even
7	0111	odd
8	1000	odd
9	1001	even
10	1010	even

8 is an even number but has odd parity, 9 is an odd number but has even parity. 6 has two 1 bits (even), 7 has three 1 bits (odd), 8 has one 1 bit (odd), 9 has two 1 bits (even) and 10 has two 1 bits (even).

How does this help us? If there is a chance of 1/100 of screwing up a single bit in a byte, there is only a chance of 1/10,000 of screwing up two bits in a byte. What this means is that of every 100 errors, 99 of them will will be detectable because of a change in parity (by changing a bit from 0 to 1 or from 1 to 0) and only 1 of them will go undetected because two changes will keep the same parity. This is a little obscure, so make sure you understand why before continuing.

This doesn't help us much yet, because we don't know what the parity was originally. 9 has even parity, 7 has odd parity. What communications programs do is FORCE the parity. They can either force it even or force it odd.

The standard ASCII characters end at 127 - that is, 0111 1111 (7F) is the highest legal number you can transmit. The left bit is unused, so we can use it to force the parity. We are going to force it even, but forcing it odd uses the same technique.

The steps are:

(1) Find the parity of the byte.

(2) If it is even, leave it alone, if it is odd, put a 1 in the left hand bit. The parity is now even.

If both the sending and receiving program have agreed on even parity, then on the receiving end, the program:

(1) Checks for even parity. If the parity is odd, it is an error.

(2) Puts a 0 back in the left hand bit. The original number is restored.

We are going to make a loop with both parts and use show_regs to watch the parity flag. Here's the program:

```
; - - - - - - - - - - ENTER DATA BELOW THIS LINE

error_banner  db  "Whoa! We have a screw up.", 13, 10, 0
```

```
; - - - - - - - - - - - ENTER DATA ABOVE THIS LINE

; - - - - - - - - - - - ENTER CODE BELOW THIS LINE
    mov  ax_byte, 0A3h            ; al binary
    mov  bx_byte, 0A2h            ; unsigned half registers
    mov  cx_byte, 0A2h            ; unsigned half registers
    mov  dx_byte, 0A3h            ; dl binary
    lea  ax, ax_byte
    call set_reg_style

comm_loop:
    mov  ax, 0
    call set_count                ; reset count to 0
    mov  bx, 0                    ; clear registers
    mov  cx, 0
    mov  dx, 0
    call show_regs                ; (1)

    ; the sending program

    call get_unsigned_byte
    and  al, 7Fh              ; 0111 1111 - make sure al < 128
    mov  bl, al                   ; copy to bl
    call show_regs_and_wait       ; (2)

    and  al, al                   ; check parity
    call show_regs_and_wait       ; (3)

    jpe  do_nothing
    or   al, 80h                  ; if parity odd, set left bit

do_nothing:
    and  al, al                   ; check parity for show_regs
    call show_regs_and_wait       ; (4)

    ; the receiving program

    mov  dl, al                   ; transmit from al to dl
    mov  cl, dl                   ; copy to cl
    call show_regs_and_wait       ; (5)

    and  dl, dl
    jpe  zero_left_bit            ; is parity even?
    mov  ax, error_banner
    call print_string

zero_left_bit:
    and  dl, 7Fh              ; 0111 1111  binary, left bit 0
    mov  cl, dl                   ; copy to cl
    call show_regs_and_wait       ; (6)
    jmp  comm_loop

; - - - - - - - - - - - ENTER CODE ABOVE THIS LINE
```

First, we set the register style so AL and DL are binary, BL and CL are unsigned. We'll use AL and BL for sending, CL and DL for receiving. Next, we reset the counter to 0 so we can see where we

are, and zero AX, BX, CX, and DX. We get a byte and make sure that it is 127 or less,{3} then send a copy to BL. BL stays unchanged for the rest of the loop. We test the parity. and if it is odd, change it. Finally, we send it off to DL.

On the receiving end, we test the parity. If it is odd, we print an error message. Then we zero the left hand bit. It is faster to zero the left hand bit than to check to see if it needs to be zeroed, so we do it for all data. The result is put in CL for comparison. AL and DL are shown in binary form so you can count the number of 1 bits in the byte.

3. The following could happen if we didn't take this step. We get a number > 128 with odd parity. 131 (1000 0011 83h) is an example. The sending program sees that it is odd parity, so it puts a 1 in the left hand bit. But there is already a 1 in the left hand bit, so the parity doesn't change, it is odd. The receiving program gets the byte, tests it, notices that it is odd, and sends back a transmission error. Since the sending program didn't see anything wrong, it just sends the same byte again. It will never get through correctly. In real life, the sending program would probably test the byte to see if it was greater than 127 and print an error if it was.

SUMMARY

POSSIBLE I/O INSTRUCTIONS

```
in  ax, constant (= port address 0 - 255)
in  ax, DX

in  al, constant (= port address 0 - 255)
in  al, DX

out constant, ax  (constant = port address 0 - 255)
out DX, ax

out constant, al  (constant = port address 0 - 255)
out DX, al
```

In these instructions DX holds a port address and can be from 0 - 65535.

PARITY

Parity is whether the number of 1 bits in a byte (or word) is even or odd. The 8086 sets the parity flag after most arithmetic and all logical operations. If parity is even, the flag is 1, if parity is odd, the flag is 0.

CHAPTER 19 - STRINGS

Sometimes we want to deal with long strings of information. Here long means hundreds or thousands of bytes, not tens of bytes. The 8086 provides a group of instructions to move and compare strings. These instructions have a rigid structure, but with a little bit of effort we can get them to work easily for us. We will start with SCAS, since it is simple, yet embodies all the rigid features of these instructions.

SCAS (scan string) compares either a byte to AL or a word to AX. The byte or word must be in memory, and the register must be AL or AX. SCAS also increments or decrements the pointer. First, the size:

```
    scasb
```

compares a byte to AL, while:

```
    scasw
```

compares a word to AX. But where's the pointer? You have no choice, it's DI. Not only is it DI, but it MUST be ES:DI. The ES segment is coded into the 8086 microcode; the DI register is coded into the 8086 microcode; there is nothing you can do to change it. What about incrementing or decrementing? In the flags register, there is a flag called the direction flag. It is set manually by the program. If DF = 0, SCAS increments DI; if DF = 1, SCAS decrements DI.{1} The equivalent software for the instruction would be:

```
              (scasb)                    (scasw)
DF = 0        cmp  al, es:[di]           cmp  ax, es:[di]
              pushf                      pushf
              add  di, 1                 add  di, 2
              popf {2}                   popf


DF = 1        cmp  al, es:[di]           cmp  ax, es:[di]
              pushf                      pushf
              sub  di, 1                 sub  di, 2
              popf                       popf
```

---

1. Every time you have called show_regs DF has been there; it doesn't show 0 and 1, it shows + and - (+ = 0 , - = 1).

2. The microcode doesn't really push and pop the flags. This is only to indicate that the order of operations is (1) get the byte (word) from the string, (2) compare and set the flags, and finally (3) increment (decrement) the pointer without changing any of the flags.

Thus, at the end of the end of the instruction, DI is in a new place and the flags are set according to the compare result. DI is incremented by a byte for the byte instructions; it is incremented by a word for the word instructions. The same pattern holds true for decrementing DI.

We set the direction flag with the instruction STD (set direction flag) and we clear the direction flag by using CLD (clear direction flag). It only needs to be set or cleared once, and this should be done before starting the operation. DF is only changed by those specific instructions from the program - it can't be changed by any arithmetical or logical operation on the chip.

If you have a string and you are looking for a specific number, (27 for instance), you simply put that number in AL (or AX) and run a loop. If:

```
long_string  db 5000 dup (?)
```

contains data and we want to look for a 27d then the operation is:

```
    lea  di, long_string{3}
    mov  al, 27
    cld

search_loop:
    scasb
    jne  search_loop
```

on exiting, DI will point 1 PAST the matching byte (word). You move back one byte (word) to find the match. Why would anyone want this instruction? With a 0 in AL, it will find the end of a C (0d terminated) string quickly. Also, that number 27 is no accident. 27d is the ASCII escape character. For a lot of hardware, 27d indicates that the bytes that follow are not ASCII characters but are technical information. For instance, on my printer the sequence (27d, 65d, 0d, 11d) sets tabs every 11 columns. SCAS can find where these substrings are so the program can operate on them.

In order to use string instructions, we need strings to work on. The one we will use is called CH1STR.OBJ. It is in \XTRAFILE. It is an object file that contains one data segment containing a string of lower case characters. The string is several thousand bytes long, it is terminated by 0, and it contains ONLY the lower case letters (a-z). It is the first draft of part of chapter 0 with all punctuation, numbers, spaces, carriage returns etc. deleted. All upper case letters have been converted to lower case so we don't have to worry about the difference between A and a, Q and q.

The name of the array in CH1STR.OBJ is CH1STR and it is defined:

---

3. Assuming that long_string's segment address is in ES.

```
    PUBLIC  ch1str
```

so that you can access it with the SEG and OFFSET operators. First, let's find out how long the string is.{4}

```
MYPROG1.ASM
; - - - - - - - - - -
STRINGSTUFF  SEGMENT  PUBLIC 'DATA'
EXTRN  ch1str:BYTE
STRINGSTUFF  ENDS
; - - - - - - - - - -
;- - - - - - - - - - PUT CODE BELOW THIS LINE

    mov  ax, seg ch1str      ; segment address of ch1str
    mov  es, ax

    mov  di, offset ch1str   ; offset address of ch1str
    mov  al, 0               ; try to match zero
    cld                      ; increment (DF = 0)

string_end_loop:
    scasb
    jne  string_end_loop

    dec  di                  ; back up one
    mov  ax, di
    sub  ax, offset ch1str
    call print_unsigned

;- - - - - - - - - - PUT CODE ABOVE THIS LINE
```

In all these string operations, we need to be careful about boundary conditions. What if there is no valid data? What if there is one valid item? What if the string is empty?

On exiting the loop, DI will point 1 past the first 0d, so we need to back up one to point to the first 0d. Then subtracting the starting position will give us the count.{5} Try it out and find out how long it is. Since we now have 3 object modules, the link instruction must read:

```
    link  myprog+ch1str+asmhelp ;
```

assuming that you name your program myprog.asm. Save the result because we will need to use this number several times.

---

4. Just to keep it from being too easy, I have put garbage both in front of the string and behind the string. That means that the string length is shorter than the length of the object file and the string does not start with the first byte of the object file.

5. If the first byte in the string is 0d, we move one, then move back one which gives the length zero.

You will notice that we have gotten the segment address by using the SEG operator. You don't need to know the name of the segment. The segment doesn't even have to be PUBLIC. As long as the VARIABLE is either in the same file or is in another file and PUBLIC, the linker will find the correct segment address and put it there.

To make things a little more complicated, we will make another infinite loop. This time you will enter a character, and the program will find the first occurance of that character. We need to add some error checking here. Since you will probably be dreaming about taking your next vacation in Hawaii while you are entering the data, a few characters that don't exist in the string (things like G $? ~) might creep in. It would be possible to run way past the end of the string before you found that character. We'll put the length of the string (from the last program) in CX, have a regular loop so we can't go too far, and jump out of the loop if we find a match.

MYPROG2.ASM

```
; - - - - - - - - - - -
STRINGSTUFF  SEGMENT  PUBLIC 'DATA'
EXTRN  ch1str:BYTE
STRINGSTUFF  ENDS
; - - - - - - - - - - -
;- - - - - - - - - - - PUT CODE BELOW THIS LINE

    mov  ax, seg ch1str
    mov  es, ax

outer_loop:
    call get_ascii_byte          ; returns character in al
    mov  cx, $$$$$$$             ; enter string length here

    mov  di, offset ch1str
    cld                          ; increment (DF = 0)

string_end_loop:
    scasb
    je   after_loop              ; if equal, we found the char
    loop string_end_loop

    mov  ax, 0                   ; we fell through the loop
    call print_unsigned
    jmp  outer_loop

after_loop:
    mov  ax, di                  ; move for printing
    sub  ax, offset ch1str       ; number of bytes
    call print_unsigned

    jmp  outer_loop


;- - - - - - - - - - - PUT CODE ABOVE THIS LINE
```

Those dollar signs are the place to enter the exact length of the string that you got from the last program. This time we jump out of the loop if we find a match; DI will be 1 past the matching character, but this will give us the right count (if we find the character in the first space, we increment once). If we can't find a match we fall through the loop and print a 0. Remember to link all 3 modules when you run the program. Run the program and then we'll move forward.

This type of thing is so common with string operations that there is a special prefix for SCAS and all other string operations which makes the coding simpler. It has several forms:

```
    rep       decrement cx ; repeat if cx is not zero
    repe      decrement cx ; repeat if cx not zero and zf = 1
    repz      decrement cx ; repeat if cx not zero and zf = 1
    repne     decrement cx ; repeat if cx not zero and zf = 0
    repnz     decrement cx ; repeat if cx not zero and zf = 0
```

REP is for the move instructions which we will see later - it won't work here. For each prefix, if either (or both) of the conditions is not true, the repitition stops. For instance, with REPE, if cx is zero, and/or if the comparison was not equal (so the zero flag was not set), the instruction will stop. For our program, the coding is:

```
    repne scasb
```

That's it. That replaces the whole inner loop. Here is our new coding of the last program.

MYPROG3.ASM

```
; - - - - - - - - - -
STRINGSTUFF  SEGMENT  PUBLIC 'DATA'
EXTRN  ch1str:BYTE
STRINGSTUFF  ENDS
; - - - - - - - - - -
;- - - - - - - - - - PUT CODE BELOW THIS LINE

    mov  ax, STRINGSTUFF
    mov  es, ax
    cld                          ; increment (DF = 0)

outer_loop:
    call get_ascii_byte          ; returns character in al
    mov  cx, $$$$$$$             ; enter string length here
    lea  di, ch1str              ; address of string

    repne  scasb

    je   found_the_char          ; an equal comparison
    mov  ax, 0                   ; we didn't find a match
    call print_unsigned
    jmp  outer_loop

found_the_char:
    mov  ax, di                  ; move for printing
```

```
    sub  ax, offset ch1str        ; number of bytes
    call print_unsigned
    jmp  outer_loop

;- - - - - - - - - - PUT CODE ABOVE THIS LINE
```

There are two possibilities for exiting the 'repne scasb' instruction. Either we found an equal comparison or we exhausted all the characters in ch1str. If we found an equal comparison, JE will send us to the print routine. Otherwise we print a 0 because we finished the loop without finding anything.

STOS

We can ask the operating system to allocate memory for us while the program is running.{6} When you get it, however, it will contain trash. The fast way to clear it is to use STOS (store to string). The instruction is:

```
    stosb
or:
    stosw
```

The equivalent action (not counting changing the value of DI) is:

```
    mov  es:[di], ax    ; or AL for byte moves
```

Once again (1) the pointer is the ES:DI pair, which is mandatory, and (2) DI is incremented or decremented (by 1 for byte, by 2 for word) depending on the status of DF, the direction flag. The instruction moves a byte (a word) from the AL (AX) register to the memory address pointed to by ES:DI. We can use the REP{7} instruction to speed things up a bit. If we have a 11,872 word block of memory, we can clear it with the following instructions:

```
; - - - - - - -
DATASTUFF  SEGMENT
my_bufferdw  11872 dup (?)
DATASTUFF  ENDS
; - - - - - - -

    mov  ax, seg my_buffer
    mov  es, ax
    cld                           ; increment (DF = 0)

    mov  ax, 0               ; clear the buffer with 0s
    mov  di, offset my_buffer
    mov  cx, 11872
    rep  stosw
```

6. Cf. You-know-who's Programmer's Guide to You-know-what or "DOS Programmer's Reference."

7. There is no comparison here, so REPE or REPNE doesn't make any sense.

That's as fast as it gets. Why does the STOS instruction use AX? Because that's the register that port i/o uses. If you are writing a communications program, you need speed. You can have the following:

```
; - - - - - - - - -
DATASTUFF  SEGMENT
port_address   dw   0F2A8h        ; this address is legal but
                                  ; there's nothing there.
input_buffer  db   4000h     dup (?)
output_buffer  db   4000h     dup (?)
DATASTUFF  ENDS
; - - - - - - - - -

    mov  ax, DATASTUFF
    mov  es, ax
    cld                           ; increment (DF = 0)
    mov  di, offset input_buffer
    mov  dx, port_address

input_loop:
    in   al, dx
    stosb
    jmp  input_loop
; - - - - - - - - -
```

A real program would be much more complicated because we would have to check to see if data was ready to come in and we might need to check the data for errors. Also we would occasionally have to clear the buffer. The port address F2A8h is just an arbitrary address. It's a legal address but there's nothing there.

We should write a program, so let's input a character and have it fill the screen. We'll leave the last line of the screen alone so you can see your input. Move your cursor to the last line before beginning the program.

```
; - - - - - ENTER CODE BELOW THIS LINE

    mov  ax, 0B800h          ; or 0B000h for a monochrome card
    mov  es, ax
    cld                      ; increment (DF = 0)


outer_loop:
    call get_ascii_byte      ; AL = fill char from input
    mov  ah, 07h             ; black background, white letters
    sub  di, di              ; set di to zero
    mov  cx, 1920            ; 24 lines X 80 chars
    rep  stosw
    jmp  outer_loop

; - - - - - ENTER CODE ABOVE THIS LINE
```

If you have a monochrome card, the segment address is 0B000h. If

you have a color card and are in text mode, the segment address should be 0B800h. This fills the first 24 lines with the input character. The STOS instruction has no effect on the cursor.

LODS

The opposite of STOS is LODS (load string) It moves a byte (word) from the string to the AL (AX) register. This time, for a change, we use the SI register as a pointer, and the default register is DS.{8} As always, SI is incremented or decremented by a byte (word) depending on the setting of DF, the direction flag. The two possibilities are:

```
    lodsb
and
    lodsw
```

The equivalent action (not counting changing the value of SI) is:

```
    mov  ax, [si]  ; or AL for byte moves
```

This is an instruction for people that write device drivers. You could use it if you are sending a string of characters to the printer, but that's about it. Code for doing that would have the following form:

```
; - - - - - - - -
buffer   db   1000 dup (?)
; - - - - - - - -
    lea  si, buffer   ; the buffer must be in the ds segment
    cld                     ; increment

out_loop:
    lodsb
    and  al, al             ; if 0, end of string
    jz   quit_loop

    mov  dl, al             ; move character to dl {9}
    mov  ah, 5              ; int 21h function 5
    int  21h                ; print a character
    jmp  out_loop

quit_loop:
    ; continue with the program
```

If you actually run this program, many printers will not print anything until they get an end of line signal (10d, 13d).

---

8. Register DS can be overriden. We'll talk about that in the second part of this chapter.

9. Int 21h (AH = 5) prints one character from DL to the printer. Why it's DL and not AL is a mystery.

By this time you may have become annoyed by the fact that there is no instruction for moving data from one place in memory to another. That is, you can't have:

```
    mov  variable2, variable1
```

Instead you have to have:

```
    mov  ax, variable1
    mov  variable2, ax
```

There is one instruction, however, where you can move a BLOCK of data from one place in memory to another. It is called MOVS. As usual with these instructions, there are two forms.

```
    movsb
```

moves a byte from DS:SI to ES:DI and increments or decrements SI and DI by one, depending on the setting of DF, the direction flag. Notice that either both are incremented or both are decremented. You can't have one pointer incrementing while the other one is decrementing.

```
    movsw
```

moves a word from DS:SI to ES:DI and increments or decrements SI and DI by two, depending on the setting of DF, the direction flag. This requires the same amount of setup as all the other routines we have looked at so far, so it is not efficient to use it for just a few bytes. For 30 or so, it is very efficient. This has the equivalent effect (except for changing DI and SI) as:

```
    mov  WORD PTR es:[di], ds:[si]   ; or BYTE PTR for bytes
```

We are going to write some subroutines which copy strings from one place in memory to another.{1} But first we need to review text strings.

The text string world is divided into Pascal and C. A Pascal string has its length in the first byte and the first character in the second byte. Since the length is in one byte, the string length may only be 0 to 255. You read the first byte of the string to get the length.

A C string can have any length. The end of a C string is marked by a byte with the value 0d. This is not the character '0', it is the number 0 (0hex). In order to find the end of a C string, you need to check each character to see if it is 0h.

We'll do the Pascal string first. We are going to pass the

---

1. For the technically minded, these routines will be only half of a real life subroutine, since they assume that the two strings do not overlap. In robust subroutines, these routines would be the section for when the destination address is lower than the source address.

addresses of the strings. If you have the Pascal call:

```
    move_pascal_string (from_string, to_string) ;
```

The first thing you need to know is that Pascal pushes things on the stack from left to right. In other words, Pascal will generate the following code:

```
    lea  ax, from_string
    push ax
    lea  ax, to_string
    push ax
    call move_pascal_string
```

We will start by assuming both near data (all data is in DS), and near subroutines. After setting up BP, the stack will look like this:{2}

```
         from_string address          bp + 6
         to_string address            bp + 4
         old IP                       bp + 2
  bp ->  old BP                       bp + 0
```

Here's the subroutine. Remember, PUSHREGS and POPREGS are macros:

```
; ----------
move_pascal_string  proc  near

    FROM_PTR  EQU  [bp+6]
    TO_PTR    EQU  [bp+4]

    push bp                         ; set up bp
    mov  bp, sp
    pushf                           ; push the flags
    PUSHREGS  cx, si, di, es        ; push the registers
    push ds                         ; move ds to es
    pop  es

    mov  si, FROM_PTR               ; load pointers
    mov  di, TO_PTR
    cld                             ; clear DF (increment)

    sub  cx, cx                     ; zero cx
    mov  cl, [si]                   ; length to cl
    inc  cx                         ; increment count by one

    rep  movsb                      ; the actual move

    POPREGS cx, si, di, es
    popf                            ; pop the flags
    pop  bp
    ret (4)                         ; pop pointers and return
```

---

2. If you forgot about BP, go back to the chapter on subroutines.

```
move_pascal_string endp
; ----------
```

The count is increased by one since we need to move not only the text, but the count itself. If the length is 0, we still need to move one byte - the count byte. The value in DS is moved to ES with a PUSH and a POP. You cannot move directly from one segment register to another. Also, at the return we POP 4 bytes (2 words) to get the pointers off the stack. Remember, in Pascal, it is the subroutine's responsibility to get rid of the arguments from a subroutine call.

You will notice that this time we saved the flags register. Why? Because we are clearing DF. When we return from the subroutine, we want DF to be exactly the same as it was on entry to the subroutine. The calling program may have DF set in some special way and we don't want to interfere with that.

There are three flags which I will call 'hard' flags. Once they are set they do not change. These are (1) TF, the trap flag, (2) IEF, the interrupt enable flag, and (3) DF, the direction flag. The 'soft' flags are CF, OF, ZF, etc. If you call a subroutine you expect CF, OF, ZF etc. to be unreliable, but you expect these three 'hard' flags to remain the same. TF is the domain of a debugger, so it is none of your business. IEF is only of interest to you if you are writing an interrupt procedure.{3} The third one, DF, is your concern. If you use DF in a subroutine, you MUST save the flags to ensure that the DF flag has the same value at the return that it had on entry.

Now for the C subroutine. If we have a C subroutine call:

```
    move_c_string ( from_string, to_string ) ;
```

C pushes things on the stack from right to left (the exact opposite of Pascal). The C complier will generate the following code.

```
    lea  ax, to_string
    push ax
    lea  ax, from_string
    push ax
    call move_pascal_string
    add  sp, 4
```

After setting up BP, the stack will look like this:

```
         to_string address            bp + 6
         from_string address          bp + 4
         old IP                       bp + 2
  bp ->  old BP                       bp + 0
```

---

3. If you do an interrupt procedure you don't have to worry because INT automatically saves the flags while clearing IEF, and IRET restores the flags on exiting.

Here's the C subroutine:

```
; ----------
move_c_string  proc  near

    FROM_PTR  EQU  [bp+4]
    TO_PTR    EQU  [bp+6]

    push bp                         ; set up bp
    mov  bp, sp
    pushf                           ; push the flags
    PUSHREGS  ax, si, di, es        ; push the registers
    push ds                         ; move ds to es
    pop  es

    mov  si, FROM_PTR               ; load pointers
    mov  di, TO_PTR
    cld                             ; clear DF (increment)

move_loop:
    lodsb                           ; source to al
    stosb                           ; al to destination
    and  al, al                     ; check for 0
    jnz  move_loop


    POPREGS ax, si, di, es
    popf                            ; pop the flags
    pop  bp
    ret

move_c_string endp
; ----------
```

We set up the routine the same way, but we cannot use MOVSB. We need to check each individual byte to see if it is 0 hex, so we move it to AL, move it from AL to the destination, and then check AL for 0. Also note that we did not pop the addresses off the stack with the return statement, since in C it is the calling program's responsibility to do that. If you look at the calling code above, you will see:

```
    add  sp, 4
```

which gets rid of the two pointers from the stack. Remember, the stack grows downward, so you ADD to decrease the size of the stack.

Let's do the same Pascal program again, but this time use long pointers, that is, give both the segment and offset of the string. This means that we will be able to move from any place in memory to any place in memory. Here is the calling code.

```
    mov  ax, segment from_string
```

```
    push ax
    mov  ax, offset from_string
    push ax
    mov  ax, segment to_string
    push ax
    mov  ax, offest to_string
    push ax
    call move_pascal_string
```

We will still keep it a near subroutine. After setting up BP, the stack will look like this:

```
         from_string segment          bp + 10
         from_string offset           bp + 8
         to_string segment            bp + 6
         to_string offset             bp + 4
         old IP                       bp + 2
  bp ->  old BP                       bp + 0
```

Here's the subroutine:

```
; ----------
move_pascal_string  proc  near

    FROM_PTR  EQU  [bp+8]
    TO_PTR    EQU  [bp+4]

    push bp                         ; set up bp
    mov  bp, sp
    pushf                           ; push the flags
    PUSHREGS  cx, si, di, ds, es    ; push the registers

    lds  si, FROM_PTR               ; load pointers
    les  di, TO_PTR
    cld                             ; clear DF (increment)

    sub  cx, cx                     ; zero cx
    mov  cl, [si]                   ; length to cl
    inc  cx                         ; increment count by one

    rep  movsb                      ; the actual move

    POPREGS cx, si, di, ds, es
    popf                            ; pop the flags
    pop  bp
    ret (8)                         ; pop pointers and return

move_pascal_string endp
; ----------
```

This takes slightly less code since we load SI and DS at the same time (with LDS -load DS) and we load DI and ES at the same time (with LES - load ES). Remember, 8086 instructions which move an offset:segment pair always have the offset in low memory and the segment in high memory; the offset is the first two bytes and the segment is the next two bytes.

We changed the EQU statements, and the return statement is now:

```
    ret (8)
```

so we take 8 bytes (4 words) off the stack, but the rest is the same.

CMPS

The final instruction in this group is CMPS, and as usual, it comes in two varieties.

```
    cmpsb
```

compares the byte addressed by DS:SI to the byte addressed by ES:DI. It is the same as the CMP instruction. It moves both bytes into the 8086, subtracts the DI byte from the SI byte and sets the flags. The two bytes in memory remain unchanged. You can look at the flags to see which byte is larger, or if they are equal. As usual, both SI and DI are incremented or decremented by one, depending on the setting of DF, the direction flag.

```
    cmpsw
```

compares the word addressed by DS:SI to the word addressed by ES:DI. It is the same as the CMP instruction. It moves both words into the 8086, subtracts the DI word from the SI word and sets the flags. The two words in memory remain unchanged. You can then look at the flags to see which word is larger, or if they are equal. Both SI and DI are incremented or decremented by two, depending on the setting of DF, the direction flag. This instruction has the same effect on the flags as:

```
    push ax
    mov  ax, ds:[si]    ; or AL for bytes
    cmp  ax, es:[di]    ; performs ( DS:[si] - ES:[DI] )
    pop  ax
```

What use is this instruction? It is possible to use this for word find, and we will do that later, but it is a little unsophisticated for that. It is great for data verification, however.

When you use the DISKCOMP utility in DOS which compares two floppy disks, it reads each of the disks sector by sector, and then compares them. A sector is 512 bytes. The code for this utility looks like this:

```
; - - - - - DATA - - - - -
error_message db   "Sectors are not the same", 0
disk1_buffer  db   512 dup (?)
disk2_buffer  db   512 dup (?)

; - - - - - CODE - - - - -
```

```
get_next_sector:

; the code for reading one sector from each disk goes here.
; then we have the code to compare the two sets of data.

    mov  si, offset disk1_buffer
    mov  di, offset disk2_buffer

    mov  cx, 256                ; 512 / 2 = 256
    repe cmpsw
    je   get_next_sector

    lea  ax, error_message   ; we had an unequal comparison
    call print_string
    jmp  get_next_sector

; - - - - - - - - - -
```

We do a word compare since it takes only half as many steps. If there is an unequal comparison at any time, the REPE instruction will terminate the loop. We can test for this inequality with JE or JNE. In this example we assume that DS and ES have the same segment address.

Any time you need to verify data, this is the instruction to use.

We are going to build a word search program. It is not very valuable since 'a' will not match 'A', but it is a good exercise to look at CMPS. We will use ch1str.obj, the file we used at the beginning of the chapter, as the text file and you can try to find individual words in the file. Remember, the file is continuous characters (no spaces), and all characters are small. If you didn't save the file length, you will have to run that program again to find the length of the file.

Here's the word_search program:

```
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
EXTRN  ch1str:BYTE
entry_banner     db  13,10, "Enter a word for a word search", 0
no_match_banner  db  "There was no match", 0
input_buffer     db   80 dup (?)
letter_count     dw   ?
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       mov   ax, seg ch1str      ; load es register
       mov   es, ax
       cld                       ; clear DF (increment)

big_loop:
       ; get a word for the word search
       mov   ax, offset entry_banner
       call  print_string
       mov   ax, offset input_buffer
```

```
        call  get_string

        ; find the end of string
        mov   al, 0               ; compare with 0
        mov   bx, offset input_buffer
        mov   cx, 0               ; letter count
letter_count_loop:
        cmp   al, [bx]            ; compare to 0
        je    end_of_count_loop
        inc   cx                  ; increment count
        inc   bx                  ; increment pointer
        jmp   letter_count_loop
end_of_count_loop:
        jcxz  big_loop      ; if cx = 0, string is empty so redo
        mov   letter_count, cx   ; store our count

        ; look for word match
        mov   di, offset ch1str
        mov   cx, $$$$            ; $$$$ = length of ch1str
        sub   cx, letter_count    ; calculate last possible match

word_search_loop:
        push  di                  ; start of search
        push  cx                  ; count for ch1str
        mov   si, offset input_buffer
        mov   cx, letter_count
        repe  cmpsb               ; the actual comparison
        je    found_it           ; if equal, we have a match

        ; no match. are we finished?
        pop   cx
        pop   di
        inc   di               ; move to next starting address
        loop  word_search_loop

        ; we fell through. finished, but no match
        mov   ax, offset no_match_banner
        call  print_string
        jmp   big_loop

found_it:
        pop   cx                  ; clear cx off the stack
        pop   di                  ; start of the match
        mov   si, offset input_buffer
        mov   cx, 25              ; move 25 characters to buffer
transfer_loop:
        mov   al, es:[di]
        mov   [si], al
        inc   si
        inc   di
        loop  transfer_loop
        mov   BYTE PTR [si], 0    ; end of a C string

        mov   ax, offset input_buffer
        call  print_string
        jmp   big_loop
; + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

The code is so long that the whole assembler file has been put on disk so you don't have to do all the typing. The pathname is \XTRAFILE\COMPARE.ASM. All you need to do is enter the length of ch1str in the MOV instruction where the dollar signs are:

```
       mov   cx, $$$$           ; $$$$ = length of ch1str
```

Link with 'link compare+ch1str+\asmhelp'. You enter a text string and the program looks for an exact match in ch1str. Here is how the program is structured.

First, the program prompts you to enter a string. The program then counts the number of bytes in the string. It must have a non-zero length or the program will prompt you again for a string. The program then starts at the beginning of the text. It saves a copy of the pointer to the start of the comparison so if we fail we can start over again at the next character. The actual comparison is:

```
    repe cmpsb
```

If that makes it through all the letters in the search string, REPE will quit because CX = 0, not because we have an unequal character. If the comparison failed we pop DI (the text pointer) and start at the next character.

If there is a match, we move 25 characters (starting with the matching characters) from the text to the buffer. It is necessary to move these because when you call print_string, the string must be in the DATASTUFF segment, and ch1str isn't. We haven't used MOVSB here because ES and DS are in the wrong place. For 25 characters there is only a marginal advantage to setting up for MOVS. Finally, the 25 characters are printed. If there is no match, a message to that effect is printed.

The text in ch1str is the first draft of chapter 1, but just for interest, I have hidden eight C keywords and eight of your favorite Middle English words in the text.{4} See if you can find them.

SEGMENT OVERRIDES

Here are the string instructions and the override rules for each one.

LODS moves a byte or word from DS:[si] to AL or AX. You may use CS:[si], SS:[si] or ES:[si].

STOS moves a byte (or a word) from AL (or AX) to ES:[di]. NO

4. Two hints. You might find four of these Middle English words in the name of a boutique. The other four of the Middle English words are some of your favorite monosyllabic words.

OVERRIDES ARE ALLOWED.

SCAS compares AL (or AX) to the byte (or word) pointed to by ES:[di]. NO OVERRIDES ARE ALLOWED.

MOVS moves a byte (or a word) from DS:[si] to ES:[di]. You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE ES:[di].

CMPS compares the byte (or a word) from DS:[si] to ES:[di]. You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE ES:[di].

Looking at the whole group, you may override DS:[si], but you may not override ES:[di]. The form of the override is strict. We will take MOVS as an example. Till now, the instructions were written:

```
movsb     ; byte move
movsw     ; word move
```

If you want to do an override, the syntax is:

```
movs BYTE PTR ES:[di], SS:[si]
movs WORD PTR ES:[di], SS:[si]
```

If you write:

```
movsb     ES:[di], SS:[di]
```

you will get an assembler error. Here are all the legal forms:

```
LODS
    lodsb
    lodsw
    lods BYTE PTR SS:[si]         ; or CS:[si], DS:[si], ES:[si]
    lods WORD PTR SS:[si]         ; or CS:[si], DS:[si], ES:[si]

STOS
    stosb
    stosw
    stos BYTE PTR ES:[di]         ; no override allowed
    stos WORD PTR ES:[di]         ; no override allowed

SCAS
    scasb
    scasw
    scas BYTE PTR ES:[di]         ; no override allowed
    scas WORD PTR ES:[di]         ; no override allowed

MOVS
    movsb
    movsw
    movs BYTE PTR ES:[di], SS:[si]     ;or CS, DS, ES:[si]
    movs WORD PTR ES:[di], SS:[si]     ;or CS, DS, ES:[si]

CMPS
    cmpsb
```

```
    cmpsw
    cmps BYTE PTR SS:[si], ES:[di]     ;or CS, DS, ES:[si]
    cmps WORD PTR SS:[si], ES:[di]     ;or CS, DS, ES:[si]
```

Just because you can do overrides with these instructions doesn't mean that you should. In fact, there is a problem. If you are using the REP instruction with an override:

```
    rep movs  WORD PTR ES:[di], SS:[si]
```

and the 8086 gets a hardware interrupt,{5} the 8086 forgets the override. What this means is that one moment you are moving data from the SS segment, and the next moment you are moving data from the same offset, but in the DS segment. This just won't do. Thus the rule is:

    NEVER USE AN OVERRIDE WITH A REP/REPE/REPNE INSTRUCTION

This actually is no hardship. Using the override adds time to the instruction. All you need to do is change the segment addresses for the duration of the string instruction, and the code will run faster. Of course, there is the setup time, but the break even point is say, 20 repeats. Here is what you would do if you needed an SS segment override:

```
    push ds        ; save old DS
    push ss        ; move SS to DS
    pop  ds        ; the same as an SS:[di] override

    rep  movsb

    pop  ds        ; get old DS back
```

The other possibility is to use LOOP instead of REP. It is slower, but better slower and reliable than faster and unreliable.

```
    rep  movs BYTE PTR ES:[di], SS:[si]
```

is the same as:

```
repeat_loop:
    movs BYTE PTR ES:[di], SS:[si]
    loop repeat_loop
```

There are even three forms of the LOOP instruction: LOOP, LOOPE, LOOPNE which are the exact counterparts to REP, REPE, REPNE.

---

5. Which can be caused by such rare occurances as your pressing a key on the keyboard or one of the 18 timer interrupts that happen each second.

SUMMARY

LODS (load from string) moves a byte or word from DS:[si] to AL or AX, and increments (or decrements) SI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si]. This performs the same action (except for changing SI) as:

```
mov  ax, DS:[SI]              ; or AL for bytes
```

The allowable forms are:

```
lodsb
lodsw
lods BYTE PTR SS:[si]         ; or CS:[si], DS:[si], ES:[si]
lods WORD PTR SS:[si]         ; or CS:[si], DS:[si], ES:[si]
```

STOS (store to string) moves a byte (or a word) from AL (or AX) to ES:[di], and increments (or decrements) DI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). NO OVERRIDES ARE ALLOWED. This performs the same action (except for changing DI) as:

```
mov  ES:[DI], ax              ; or AL for bytes
```

The allowable forms are:

```
stosb
stosw
stos BYTE PTR ES:[di]         ; no override allowed
stos WORD PTR ES:[di]         ; no override allowed
```

SCAS compares AL (or AX) to the byte (or word) pointed to by ES:[di], and increments (or decrements) DI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). NO OVERRIDES ARE ALLOWED. This sets the flags the same way as:

```
cmp  ax, ES:[DI]              ; or AL for bytes
```

The allowable forms are:

```
scasb
scasw
scas BYTE PTR ES:[di]         ; no override allowed
scas WORD PTR ES:[di]         ; no override allowed
```

MOVS moves a byte (or a word) from DS:[si] to ES:[di], and increments (or decrements) SI and DI, depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE

ES:[di]. Though the following is not a legal instruction, it signifies the equivalent action to MOVS (not including changing DI and SI):

```
    mov  WORD PTR ES:[DI], DS:[SI]     ; or BYTE PTR for bytes
```

The allowable forms are:

```
    movsb
    movsw
    movs BYTE PTR ES:[di], SS:[si]     ;or CS, DS, ES:[si]
    movs WORD PTR ES:[di], SS:[si]     ;or CS, DS, ES:[si]
```

CMPS compares the byte (or a word) at DS:[si] to the one at ES:[di], and increments (or decrements) SI and DI, depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE ES:[di]. Although the following is not a legal action, it signifies the equivalent action to CMPS (not including changing DI and SI):

```
    cmp  WORD PTR DS:[SI], ES:[DI]     ; or BYTE PTR for bytes
```

The allowable forms are:

```
    cmpsb
    cmpsw
    cmps BYTE PTR SS:[si], ES:[di]     ;or CS, DS, ES:[si]
    cmps WORD PTR SS:[si], ES:[di]     ;or CS, DS, ES:[si]
```

The string instructions may be prefixed by REP/REPE/REPNE which will repeat the instructions according to the following conditions:

```
    rep       decrement cx ; repeat if cx is not zero
    repe      decrement cx ; repeat if cx not zero AND zf = 1
    repz      decrement cx ; repeat if cx not zero AND zf = 1
    repne     decrement cx ; repeat if cx not zero AND zf = 0
    repnz     decrement cx ; repeat if cx not zero AND zf = 0
```

Here, 'e' stands for equal, 'z' is zero and 'n' is not. These repeat instructions should NEVER be used with a segment override, since the 8086 will forget the override if a hardware interrupt occurs in the middle of the REP loop.

'HARD' FLAGS

IEF, TF and DF are 'hard' flags. Once they are set they remain in the same setting. If you use DF, the direction flag, in a subroutine, you must save the flags upon entry and restore the flags on exiting to make sure that DF has not been altered.

CHAPTER 20 - CONTROL STRUCTURES

Control structures are things like 'for', 'do', 'while' and 'if' that you have in high level languages. It is possible to do these things in assembler language if you want. They are included here so if you ever want to use them at this level, you will have a template on how to set them up.

All the examples will use integers only. If you have real numbers:

```
    if ( variable < 3.891 )
         x = 4 * y ;
```

it is too difficult to implement on the 8086. Of course, if you have an 8087, it's a piece of cake.

IF

IF is the easiest and it will illustrate how we are going to set up our labels. Let's take that first example. Since it is a waste of space to define variables over and over, we will assume that all variables are words, not bytes. The examples will all be written in C.

```
    if (variable < 7)
    {
         x = 4 + y ;
    }
```

The assembler code looks like this:

```
if1: ;----------------
    cmp  variable, 7
    jge  end_if1

    mov  x, 4
    mov  ax, y
    add  x, ax
end_if1: ;----------------
```

All labels will look like variations of this. The control words will be the labels.{1} They will have a unique number for each block so you can write as many different blocks as you want; the label that signals the end of the block will be the word 'end_' followed by the control word. The semicolon with the line is a

---

1. The words 'if' and 'endif' are Microsoft macro directives, so if you use them without a tag number, you will get some bizarre code indeed.

---

comment, and it is used for visual separation.

By using a different number each time, you may nest as deeply as you want, though the code will be pretty unreadable:

```
    if   (variable < 7 )
    {
         if ( x > 9 )
         {
              if ( z == 0 )
              {
                   y = 12 ;
              }
              q = 5 ;
         }
         r = 7 ;
    }
```

becomes:

```
if1: ;---------------
    cmp  variable, 7
    jge  end_if1

if2: ;---------------
    cmp  x, 9
    jle  end_if2

if3: ;---------------
    cmp  z, 0
    jne  end_if3

    mov  y, 12
end_if3: ;-----------
    mov  q, 5
end_if2: ;-----------
    mov  r, 7
end_if1: ;-----------
```

With the Microsoft assembler, you don't need to have the labels start at the beginning of the line, but they MUST be the first thing on the line. You can indent them:

```
if1: ; --------------
    cmp  variable, 7
    jge  end_if1

    if2: ;-------------
         cmp  x, 9
         jle  end_if2

         if3: ;------------
              cmp  z, 0
              jne  end_if3

              mov  y, 12
```

```
         end_if3: ;--------
         mov  q, 5
    end_if2: ;---------
    mov  r, 7
end_if1: ;---------
```

You will notice that they are all compares and jumps using reverse logic. If the condition is NOT true, then we jump. If it is true, we just go on. What about IF ELSE? It looks almost the same.

```
    if ( j < 12 )
    {
         y = 19 ;
    }
    else
    {
         z = 25 ;
    }
```

We get:

```
if16: ;----------------

    cmp  j, 12
    jge  else16

    mov  y, 19
    jmp  end_if16

else16:
    mov  z, 25
     end_else16:
end_if16: ;---------------
```

We jump to another part inside the block, and throw in a JMP after the IF part.

WHILE

WHILE is the same as IF except that at the end of the block, we jump back to the beginning.

```
    while ( j < 20 )
    {
         j = j + 1 ;
    }

while25: ;------------
    cmp  j, 20
    jge  end_while25

    inc  j
    jmp  while25
end_while25: ;-----------
```

DO WHILE

DO WHILE always goes through the code once. The decision process is at the end.

```
    do
    {
         k = k + 7 ;
    } while (k < 0) ;


do_while74: ;------------
    add  k, 7

    cmp  k, 0
    jl   do_while74
end_do_while74: ;------------
```

Notice that here the jump is a positive decision. If the condition is fulfilled, we jump back to the start, otherwise we fall through. What do you do with 'break' and 'continue'? {2} Let's put them in:

```
    do
    {
         k = k + 7 ;
         if ( y < 9 )
         {
              continue ;
         }
         j = j - 6 ;
         if ( z == 5 )
         {
              break ;
         }
         y = y * 2 ;
    } while ( j < 17) ;
```

In assembler language, this becomes:

```
do_while143: ;----------
    add  k, 7

    if144: ;-----------
         cmp  y, 9
         jge  end_if144

         jmp  continue143
    end_if144: ;-----------

    sub  j, 6
```

---

2. For those of you who are not C people, BREAK breaks you out of the innermost loop. CONTINUE skips all the code till the end of the loop.

```
    if145: ;-----------
         cmp  z, 5
         jne  end_if145

         jmp  break143
    end_if145: ;-----------

    sal  y, 1           ; shift left 1 = multiply by 2
continue143:
    cmp  j, 17
    jl   do_while143
end_do_while143:
break143:
```

In DO WHILE, CONTINUE is the label just before the decision process. BREAK is always the first label after the block. We can just use the label that marks the end of our block for the break. Instead of:

```
    jmp  break143
```

we can have:

```
    jmp  end_do_while143
```

From now on, we'll put in the continue and break, even if they aren't used. Once you are used to it, you can drop the break label and use the end of block label. Going back to the WHILE statement, we have:

```
while25: ;------------
    cmp  j, 20
    jge  end_while25

    inc  j
continue25:
    jmp  while25
end_while25: ;-----------
break25:
```

FOR

The FOR statement in C is more sophisticated than in other languages, but we will keep the example simple. It has 3 parts:

```
    for ( i = 11 ; i < 20 ; i = i + 1 )
```

can be looked at as:

```
    for ( initialize ; test ; update )
```

The reason that it is sophisticated is that it can have any number of things in the first and third parts.

```
    for ( i = 11, j = 27  z = 4 ; i < 20 ; k = k + 1, j = j/2)
```

is legitimate. Therefore we need to set aside a block for the initialize part, a block for the test part, and a block for the update part. One question is whether we want the initialization inside the FOR label. I'm not doing it, but it is a possibility. Here's the C code.

```
    for ( i = 11 ; i < 20 ; i = i + 1 )
    {
         k = k + 9
    }
```

And its corresponding assembler code:

```
init217: ;---------------

    mov  i, 11
    end_init217: ;------
for217: ; ---------------
test217:
    cmp  i, 20
    jge  end_for217
    end_test217: ;-----

    add  k,9

continue217:
update217:
    inc  i
    end_update217: ; ----
    jmp  test217
end_for217: ;-------------
break217:
```

My that's a lot of labels. In this example we would get rid of most of them because there is so little code, but if the block contained a lot of code they would be a help, not a hindrance. Remember, a label generates no code, so this makes no difference in the size of the object file. Some of the labels which are not targets of jumps could be made into comments.

```
    end_update217: ;----
```

can become:

```
    ; end_update217 ----
```

Whichever way looks better to you. Using comments instead of labels gives you a slightly faster compile time. Just make sure you don't change one that IS a target of a jump.

SWITCH

Finally there is the switch statement. If you don't know what it is, skip this section because it will probably be confusing.

```
    switch ( k )
    {
              case 'A':      x = x + 9 ;
                             break ;

              case '&':      y = y * 2 ;
                             break ;

              case 'j':      z = z - 7 ;
                             break ;
    }
```

We get the following:

```
switch82: ;-----------------

    cmp  k, 'A'
    jne  test82_2
    jmp  case82_1
test82_2
    cmp  k, '&'
    jne  test82_3
    jmp  case82_2
test82_3
    cmp  k, 'j'
    jne  end_test82
    jmp  case82_3
end_test82:
    jmp  deafult82
    ; ----------

case82_1:
    add  x, 9
    jmp  break82

case82_2:
    sal  y, 1           ; multiply by 2
    jmp  break82

case82_3:
    sub  z, 7
    jmp  break82

default82:
    jmp  break82

end_switch82: ;------------
break82:
```

It may look like there are unnecessary jumps at the beginning, but it is likely that in a real program some of the case statements would be more than +127 bytes away from the conditional jumps, so you need JMP, which can go anywhere in the segment.

CHAPTER 21 - .COM FILES

All the programs that we have made so far have been .EXE files. That means that the file extension has always been .EXE after linking. When you have and .EXE file, the program loader makes certain adjustments to the machine code at run time. These adjustments are the actual segment addresses of the segments.

There is another type of executable file, and that is a .COM file. When the loader puts a .COM file into memory, it makes no adjustments, it simply reads the file directly from disk into memory. Therefore, a .COM file loads faster. However, there is a restriction:

> All code, data, and the stack must be in a single segment. This effectively limits a .COM program to 65536 bytes of code+data+stack. {1}

In general, it is easier for program development to keep code and data separate, but we can mix them together. Let's look at the template for a .COM file. It is called COMTEMP.ASM.

```
; com file template
; put name here
; * * * * * * * * * * * * * * * *
INCLUDE \PUSHREGS.MAC

COMSEG  SEGMENT  PUBLIC  'CODE'

        ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG, ss:COMSEG
; - - - - - - - - - -
main    proc  NEAR

        ORG 100h
start:

; - - - - START CODE BELOW THIS LINE

; - - - - END CODE ABOVE THIS LINE

        ret

main    endp

; - - - - START SUBROUTINES BELOW THIS LINE

; - - - - END SUBROUTINES ABOVE THIS LINE
```

---

1. Actually, it is possible to get around this restriction with suitable fiddling, but by that time you have made the program much more complicated so you have lost any advantage that you had by not using an .EXE file.

```
; - - - - START DATA BELOW THIS LINE

; - - - - END DATA ABOVE THIS LINE

Stack_area  dw  500 dup (?)

COMSEG ENDS
; * * * * * * * * * * * * * * *
END  start
```

We will take the things in order. First, there is the line:

```
    ORG  100h
```

This effectively tells the assembler to put 100h (256d) bytes of zeros at the beginning. Also notice that the setup code that we had in a .EXE file is missing; we start with the code immediately. This all has to do with the PSP (program segment prefix).{2}

PSP

You will remember from the chapter describing the .EXE template file that we wrote:

```
    push ds
    sub  ax, ax
    push ax
```

because upon entry to an .EXE file, DS contains the segment address of the PSP, and at offset 0000 (that is, the first byte of the segment) there is a machine instruction for an orderly exit from the program. In an .EXE file, the PSP is somewhere in memory, put there by the loader. In a .COM file, the PSP is the first 100h (256d) bytes of the segment. The loader fills in the PSP and then reads the file directly from disk. That means that the machine instruction for an orderly exit is at 0000 of the current segment. Notice that we have a NEAR procedure. A .COM file has a near return which will stay in the same segment. Normally we would have to write:

```
    sub  ax, ax
    push ax
```

to put the return address 0000 on the stack, but for a .COM file the loader pushes 0000 on the stack before giving control to the program. Why the loader provides this service for a .COM file but not for an .EXE file is a mystery. In any case, with a .COM file, you don't need to push the return address on the stack, since it's there already.

---

2. If you want to know what the PSP (program segment prefix) is exactly, consult either of the two books on hardware and interrupts. The PSP is always exactly 256 bytes long.

We have the assembler directive:

```
       ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG, ss:COMSEG
```

The reason for including all the segments is (1) the loader actually loads the segment address into all four segments and (2) the assembler will use the natural segment for all instructions. The BP instructions will refer to SS, the data instructions will refer to DS, the string move instructions (SCAS, CMPS, MOVS) will refer to ES. That means that the assembler will not use segment overrides. This helps avoid something called phase errors which will be explained at the end.

Right after the ORG instruction is start: (the first instruction executed in the program.)

```
         ORG 100h       ; 256d
    start:
```

This is inflexible. After reading the program into memory, the loader ALWAYS starts the program at 100h. All .COM files start execution at 100h. Period.

There is space for subroutines, space for data, and finally space for the stack. The order of subroutines and data can be changed. The stack space is technically not necessary, but it is a good reminder to leave it there. All .COM files take up a whole 65536 byte segment in memory, no matter how short they are.{3} The stack is at the very end of the segment, so if your program is 200 bytes long, you have 65336 bytes of stack space.

Let's make a simple program. It is the famous "Hello,World" program.

```
; - - - - START CODE BELOW THIS LINE
    mov  dx, offset mr_happy_face      ; int 21h, function 9
    mov  ah, 9
    int  21h
; - - - - END CODE ABOVE THIS LINE

; - - - - START DATA BELOW THIS LINE
mr_happy_face db  "Hello, world!", 13, 10, "$"
; - - - - END DATA ABOVE THIS LINE
```

This program prints 'Hello World!' and a new line. The dollar sign signifies the end of the string for this interrupt. It is simple enough, and it gives us the chance to look at the extra step needed to make a .COM file. Assemble it, and link it. When you link it, you will get a warning that there is no stack segment. For .COM files, this warning is unimportant. We now have an .EXE file (which, by the way, won't run correctly). How do we

---

3. This is a slightly abridged explaination. For the real details, consult Microsoft's "The MS-DOS Encyclopedia".

make it a .COM file?

Among the programs that you got with DOS is one called EXE2BIN. It takes an .EXE file, and if possible, converts it into a .COM file. You simply write the name of the file you want converted and the name of the converted file. Both of these names must have the full file extension:

```
exe2bin  programA.exe  programA.com
```

You will now have programA.com as a .COM file. If we now write:

```
C> programA
```

Will the loader load the .COM file or the .EXE file? The DOS order for execution is .COM files first, then .EXE files, then .BAT files. DOS will execute the .COM file. Try it.

That was pretty easy. Now, as a technical exercise, we are going to link together three different files. Here's the first file:

```
; prog1.asm
; * * * * * * * * * * * * * * *
INCLUDE \PUSHREGS.MAC
COMSEG  SEGMENT  PUBLIC  'CODE'

       ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG, ss:COMSEG

PUBLIC message1
EXTRN  subroutine_a:NEAR, message3:BYTE
; - - - - - - - - - -
main   proc  NEAR

       ORG 100h
start:
       mov   ah, 9                ; int 21h, ah = 9
       mov   dx, offset message1
       int   21h

       mov   ah, 9                ; int 21h, ah = 9
       mov   dx, offset message3
       int   21h

       call  subroutine_a
       ret

main   endp

message1 db  "This is from the main program.", 13, 10, "$"

COMSEG ENDS
; * * * * * * * * * * * * * * *
END  start
; ----------
```

Here is the second file:

```
; prog2.asm
```

```
    ; * * * * * * * * * * * * * * * *
    INCLUDE \PUSHREGS.MAC
    COMSEG  SEGMENT  PUBLIC  'CODE'

           ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG, ss:COMSEG

    PUBLIC message2, subroutine_a
    EXTRN  subroutine_b:NEAR, message3:BYTE
    ; - - - - - - - - - - -
    subroutine_a   proc  NEAR

           PUSHREGS ax, dx
           mov   ah, 9                ; int 21h, ah = 9
           mov   dx, offset message2
           int   21h

           mov   ah, 9                ; int 21h, ah = 9
           mov   dx, offset message3
           int   21h

           call  subroutine_b
           POPREGS ax, dx
           ret

    subroutine_a   endp

    message2 db  "This is from subroutine A.", 13, 10, "$"

    COMSEG ENDS
    ; * * * * * * * * * * * * * * * *
    END
    ; ----------
```

And here is the third file:

```
    ; prog3.asm
    ; * * * * * * * * * * * * * * * *
    INCLUDE \PUSHREGS.MAC
    COMSEG  SEGMENT  PUBLIC  'CODE'

           ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG, ss:COMSEG

    PUBLIC message3, subroutine_b
    EXTRN  message1:BYTE, message2:BYTE
    ; - - - - - - - - - - -
    subroutine_b   proc  NEAR

           PUSHREGS ax, dx
           mov   ah, 9                ; int 21h, ah = 9
           mov   dx, offset message1
           int   21h

           mov   ah, 9                ; int 21h, ah = 9
           mov   dx, offset message2
           int   21h

           POPREGS ax, dx
```

```
          ret

    subroutine_b   endp

    message3 db  "This is from subroutine B.", 13, 10, "$"

    COMSEG ENDS
    ; * * * * * * * * * * * * * * * *
    END
    ; ----------
```

If you look at them, you will see that all they do is print messages; sometimes from their own file, sometimes from external files. The subprograms all have different names since they call each other. Of course, they have both PUBLIC and EXTRN statements. Only prog1 has:

```
    start:
```

since that is where the program execution will start, and only prog1 has:

```
     ORG 100h
```

since having it in the other files would leave unnecessary blank spaces in the other programs. Assemble the programs. When you link the programs, prog1 MUST be the first on the line:

```
    link prog1+prog2+prog3
    link prog1+prog3+prog2
```

are both ok, but:

```
    link prog2+prog1+prog3
```

will not work since the linker, exe2bin, and the loader are counting on the starting instruction being at 100h. If you change the order, you will either get complaints from EXE2BIN or the program won't run correctly. Now with:

```
    exe2bin  prog1.exe  prog1.com
```

you have a .COM file. Try it out.

Any .COM file can also be made into an .EXE file. We will make a simple program in .COM format, and then add the necessary things to make it an .EXE format. First, here's the .COM format.

```
; commode.asm
; * * * * * * * * * * * * * * * *
COMSEG  SEGMENT  PUBLIC  'CODE'

       ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG, ss:COMSEG

main   proc  NEAR
```

```
       ORG 100h
start:
       ; get video mode int 10h, function 0Fh
       mov   ah, 0Fh
       int   10h

       ; al = display mode
       mov   bl, 10              ; divide by 10
       mov   si, offset ones     ; right hand digit
       mov   cx, 2               ; 2 digit answer
division_loop:
       mov   ah, 0               ; clear ah
       div   bl                  ; al/bl, remainder in ah
       add   ah, '0'             ; change to ascii
       mov   [si], ah
       dec   si                  ; one byte to the left
       loop  division_loop

       ; display string
       mov   dx, offset message  ; int 21h, service 9
       mov   ah, 9
       int   21h

       ret

main   endp

; - - - - START DATA BELOW THIS LINE
message      db  "The  current video mode is  "
ones         db  ?
             db  ".",  13, 10, "$"
; - - - - END DATA ABOVE THIS LINE

COMSEG ENDS
; * * * * * * * * * * * * * * *
END  start
; - - - - - - - - - - -
```

This program gets the video mode which is a number which tells you what mode the monitor is operating in. To find out what the number means, consult either of those two hardware books. It gets the mode through an interrupt. The mode is returned in AL. It then puts the number in a string and prints the string. We cannot link with asmhelp.obj, so it takes all this work is simply to output a number.

We'll call this commode.asm. Assemble, link, use exe2bin, and run it. Now let's make the .EXE counterpart. Here it is.

```
; exemode.asm
; * * * * * * * * * * * * * * *
STACKSEG     SEGMENT  STACK  'STACK'

       dw    20 dup (?)

STACKSEG  ENDS
```

```
; - - - - - - - - - -
COMSEG  SEGMENT  PUBLIC  'CODE'

        ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG

main    proc  FAR

start:
        push  ds                  ; same as before
        sub   ax, ax
        push  ax

        push  cs                  ; ds = cs
        pop   ds

        ; get video mode int 10h, service 15
        mov   ah, 15
        int 10h

        ; al = display mode
        mov   bl, 10              ; divide by 10
        mov   si, offset ones     ; right hand digit
        mov   cx, 2               ; 2 digit answer
division_loop:
        mov   ah, 0               ; clear ah
        div   bl                  ; al/bl, remainder in ah
        add   ah, '0'             ; change to ascii
        mov   [si], ah
        dec   si                  ; one byte to the left
        loop  division_loop

        ; display string
        mov   dx, offset message  ; int 21h, service 9
        mov   ah, 9
        int   21h

        ret

main    endp

; - - - - START DATA BELOW THIS LINE
message      db  "The  current video mode is  "
ones         db  ?
             db  ".",  13, 10, "$"
; - - - - END DATA ABOVE THIS LINE

COMSEG ENDS
; * * * * * * * * * * * * * * *
END  start
```

This is almost the same. We have put in a small stack segment. Here's the different part:

```
;----------
        ASSUME  cs:COMSEG, ds:COMSEG, es:COMSEG

main    proc  FAR
```

```
start:
       push  ds                  ; same as before
       sub   ax, ax
       push  ax

       push  cs                  ; ds = cs
       pop   ds

;----------
```

SS is no longer in the ASSUME statement, since it now refers to the stack segment. CS, DS, and ES still refer to COMSEG. The procedure is now a FAR procedure. We have taken the ORG out, since that would simply waste 100h (256) bytes of space. Then we have the normal .EXE startup except the data is now in COMSEG, so we move CS to DS. That's it. We'll call this one exemode.asm. Assemble, link and run it. It should give you the same result.

Here is the listing for both executable files:

```
    COMMODE  COM       65  10-21-89  11:11p
    EXEMODE  EXE      631  10-21-89  11:10p
```

Notice how much bigger the .EXE file is. That is because the .EXE file has a bunch of information for the loader. Also, if you run both programs, the .COM file will start a little quicker. Those are the only advantages.

PHASE ERRORS

The assembler generates code in two steps. On the first pass, it calculates the address of each variable and machine instruction without actually writing code. For instance, if there is the instruction:

```
    mov  ax, variable1
```

The assembler will allocate 4 bytes for the instruction. If variable1 has already been defined, but is in ES, then the assembler will allocate 5 bytes; 1 for the segment override and 4 for the instruction itself. If, however, variable1 has not been defined yet (it appears later in the code), then the assembler will assume that it is in DS and allocate 4 bytes. If it turns out that it is later defined to be in ES, then when the assembler generates code, it will write 5 bytes, 1 for the override and 4 for the instruction. But this means that EVERYTHING after this instruction will have been shifted one byte, so EVERYTHING after the instruction will be at the wrong address. The assembler will detect this and print out a PHASE ERROR. This means that the machine code is garbage.

By having all four segment registers in the ASSUME statement of a .COM file, you guarantee that the assembler will not generate segment overrides.

```
    add  dx, [bp]
```

will have BP relative to SS.

```
    sub  variable1, si
```

will have variable1 relative to DS. This will go a long way towards eliminating errors in a .COM file.

CHAPTER 22 - BCD NUMBERS

This chapter covers numbers which are in BCD format, both packed and unpacked. You will probably never need to write any programs on the 8086 that need these instructions, so you can either do this chapter because it is the only time that you will run into them, or you can skip the chapter because you will never see them again. My advice would be to go through it anyways so you know what the capabilities of the 8086 are. The programs in this chapter are more advanced so will be more of a challenge to your understanding.

BCD stands for binary coded decimals. In their unpacked form, each byte stands for a single decimal digit. If we take a number like 831974 and put it in memory, the bytes will look like this:

```
08h
03h
01h
09h
07h
04h
```

With high memory at the top and low memory at the bottom. Notice that the top number is not ASCII '8' (hex 38), but the number 8 (hex 08). The same holds true for all the bytes; they are not ASCII characters, they are numbers.

Why would we want to have numbers like this? They use up more space (about 2 1/2 times as much), and the arithmetic operations are slower (a multiplication on the 8086 can be several HUNDRED times slower). They are not used for scientific operations. They are used in business for billing. In a typical billing operation, the number is entered from a terminal and stored. At the end of the month, the number is printed on one line of the bill, and then added to the total. If it were converted to an integer, it would be necessary to do one conversion during data entry, then another conversion during printing. This way, the only time it will be converted is if it is used in some arithmetic.

One excuse for using BCD numbers is that they are more accurate. It is true that they have no rounding errors, but neither do integers. Integers and BCD numbers can have the same accuracy.

The typical form for BCD numbers is 18 digits. If you want to convert an 18 digit number to a standard integer, it takes about 25 multiplications. That is a lot of multiplications to convert one number. Similarly, it takes about 25 divisions to get a number out of integer form back into a BCD number. This is a big waste of time for a business situation. We keep the numbers in decimal form and use them occasionally for arithmetic.

Actually, you would have to be crazy to use an 8086 for BCD

numbers. If you have a PC and use BCD numbers habitually, an 8087 coprocessor will work on BCD numbers at lightning speed. An investment of $175 will save you countless hours of grief. Also, it is next to impossible to do BCD division on the 8086, but takes no longer than normal to do BCD division on the 8087.

That being said, just consider this chapter an academic exercise. It will give you the information so if the question should arise at a party, you will be able to say how BCD arithmetic works on the 8086.

All 8086 numbers have the least significant digit in low memory and the most significant digit in high memory. This includes packed and unpacked BCD numbers. In the unpacked form, each byte represents a digit, and has a value from 0 to 9. Here are some numbers and their unpacked BCD encoding (in hex):

3986149	27	961728	74610
03h	02h	09h	07h
09h	07h	06h	04h
08h		01h	06h
06h		07h	01h
01h		02h	00h
04h		08h	
09h			

This wastes a lot of space. Someone early on noticed that the upper half byte is completely unused. You can cut the space consumption in half if you put two digits in each byte - one in the lower half byte, and the other in the upper half byte. This packing always starts from the least significant digit and goes to the most significant digit. Here are the same numbers in packed form.

3986149	27	961728	74610
03h	27h	96h	07h
98h		17h	46h
61h		28h	10h
49h			

This is a considerable saving in space. The 8087 (not 8086) standard for these numbers is 10 byte long numbers. The low order 9 bytes contain 18 digits. The last byte is 00h if the number is positive and 80h if the number is negative. Here are an 18 digit positive number and an 18 digit negative number, along with their 10 byte BCD encoding.

+137486298374691552	-581726405829645298
00	80
13	58
74	17
86	26
29	40
83	58
74	29
69	64
15	52
52	98

Let's work with the packed BCD numbers first.

PACKED BCD NUMBERS

The 8086 can manipulate packed BCD numbers. There is a subprogram in asmhelp.obj that gets a BCD number from the keyboard and one that prints a BCD number on the screen. Let's do some input and output first.

```
; - - - - - - - - - - START DATA BELOW THIS LINE
variable1dw   ?    ; first four bcd digits
variable2dw   ?    ; second four bcd digits
variable3dw   ?    ; third four bcd digits
variable4dw   ?    ; fourth four bcd digits
variable5dw   ?    ; last two bcd digits and sign
; - - - - - - - - - - END DATA ABOVE THIS LINE

; - - - - - - - - - - START CODE BELOW THIS LINE

outer_loop:
    lea  ax, variable1
    call get_bcd

    mov  ax, variable5
    call print_hex
    mov  ax, variable4
    call print_hex
    mov  ax, variable3
    call print_hex
    mov  ax, variable2
    call print_hex
    mov  ax, variable1
    call print_hex

    lea  ax, variable1
    call print_bcd
    loop outer_loop

; - - - - - - - - - - END CODE ABOVE THIS LINE
```

This gets a 10 byte BCD number, prints it out in hex (with high memory on top), and then prints the BCD number out again. Commas

are allowed. Printing the number in hex form allows you to see what the numbers look like internally.

There are two instructions for BCD numbers - DAA (decimal adjust for addition), and DAS (decimal adjust for subtraction). We'll use each one on individual bytes to see how they work. Let's try DAA.

```
; - - - - - - - - - - START CODE BELOW THIS LINE
    mov  ax_byte, 0C4h  ; half registers, hex
    mov  bx_byte, 094h  ; half regs signed, hex
    mov  dx_byte, 91h   ; half registers, signed
    lea  ax, ax_byte
    call set_reg_style

    sub  cx, cx         ; clear cx for clarity
outer_loop:
    mov  ax, 0              ; clear the registers
    mov  bx, 0
    mov  dx, 0
    call set_count
    call show_regs

    call get_hex_byte       ; byte for al
    mov  dx, ax             ; copy to dx
    push ax                 ; save al
    call get_hex_byte       ; byte for bl
    mov  bl, al
    mov  bh, bl             ; copy to bh
    pop  ax                 ; restore al
    call show_regs_and_wait
    add  al, bl             ; normal add
    mov  dx, ax             ; copy to dx
    call show_regs_and_wait
    daa                     ; make adjustment
    mov  dx, ax             ; copy to dx
    call show_regs_and_wait
    jmp  outer_loop

; - - - - - - - - - - END CODE ABOVE THIS LINE
```

Since the program works with both BCD adjustments and integer arithmetic, DX shows a signed integer copy of AX and BH shows a signed integer copy of BL. A copy of AX is moved to DX every time an operation is performed on AL.

The idea of this subprogram is to enter hex numbers that look like decimal numbers - e.g. 65h, 78h, 08h, 29h. You can enter numbers that contain A-F if you want to see what happens with bad data. Each half byte of a BCD number should look like a decimal number. You will notice that the actual addition is done by ADD. The alteration is done by DAA, which assumes that you have just added two legitimate BCD numbers, and the result is in AL. The register MUST be AL. What DAA does will be discussed in a moment.

The DAA instruction looks at AL as being two half bytes. If the result from the lower half byte addition is 10 or over, the 8086 subtracts 10 and then adds 1 to the upper half byte.{1} For instance, if the result is 17, it subtracts 10, to leave 7 in the lower half byte, and adds 1 to the upper half byte. After it has done this, it looks at the upper half byte. If the upper half byte is now 10 or over, it subtracts 10 and sets the carry flag for future additions.{2}

While the whole thing sounds a little confusing, if you look at the bytes in hex, they will act in the same way as if you were doing normal decimal addition with pencil and paper except all the carries are done at one time. CF will be set if the hundreds digit is 1 and will be cleared if the hundreds digit is 0. Use the previous program a few times to get the hang of what is going on. When you feel confident, we'll move on to subtraction.

SUBTRACTION

DAS (decimal adjust for subtraction) is similar to DAA. It makes an adjustment after the subtraction itself. We'll use the same program as before, making two alterations. Where you have ADD, change it to SUB ; where you have DAA, change it to DAS. That's all.

```
add  ->   sub
daa  ->   das
```

Run the program, and put in a number of examples. If the lower number is larger than the top number, there will be a borrow.

DAS works the opposite of DAA. If there has been a borrow into the low half byte, it adds 10 to the low half byte and subtracts 1 from the high half byte. It then looks at the high half byte.

---

1. The technical description is a little confusing, so if you get muddled up, just forget it. Here it goes. (AF is the auxilillary flag). The 8086 checks for either (1) the low half byte > 9 (that is, between 10 and 15) or (2) AF = 1. AF is set on a byte addition when there is a carry out of the low half byte, that is, the result is greater than 16 for the low half byte. If either of these events has occured, the 8086 ADDS 6 to the low half byte. Why? Let 10 + x represent the result of the addition, where x < 10. We then have:

$$10 + x + 6 = 10 + 6 + x = 16 + x$$

but 16 is 0001 0000 (10h), the first bit in the high byte, so this has the effect of leaving the part less than 10 in the low half byte and adding 1 (the carry) to the high half byte.

2. This is exactly the same logic as in the last footnote, except that the 8086 adds 60h to the high byte. This has the effect of shifting the excess out of AL altogether. The 8086 then sets the carry flag.

If there has been a borrow into it, the 8086 adds 10 to the high half byte and sets the carry flag.

Do a few more examples with the program to make sure you understand what's happening. It will look just like what you do with pencil and paper, except that with pencil and paper you do the borrow before you do the subtraction and here the borrow is done afterwards if it is needed.

ADDITION AND SUBTRACTION

We have a number of possibilities with addition and subtraction. Here they are. The sign of the numbers is in parentheses and the operation is in between.

```
(+) + (+)      (+) + (-)      (+) - (+)      (+) - (-)
(-) + (+)      (-) + (-)      (-) - (+)      (-) - (-)
```

The subroutines we use are going to assume that (1) both numbers are the same sign, and (2) for subtraction, the larger number is on top and the smaller number is on the bottom. There should be a section of the program that decides whether addition or subtraction should be used, and which number is on top. Then comes the addition or subtraction. We will write the first part later. For now, when you input numbers, both numbers must have the same sign, and for subtraction, the larger number must be first. Here's the addition subroutine.

```
; - - - - - START SUBROUTINE BELOW THIS LINE
_bcd_addition proc near

    RESULT_ADDRESS      EQU  [bp + 8]
    BOTTOM_ADDRESS      EQU  [bp + 6]
    TOP_ADDRESS         EQU  [bp + 4]

    push bp
    mov  bp, sp
    PUSHREGS  ax, bx, cx, si, di

    mov  si, TOP_ADDRESS
    mov  bx, BOTTOM_ADDRESS
    mov  di, RESULT_ADDRESS
    mov  cx, 9          ; 9 bytes of BCD numbers
    clc                 ; clear the carry flag

add_loop:
    mov  al, [si]       ; move and add bytes
    adc  al, [bx]       ; add two numbers and the carry
    daa                 ; bcd adjust
    mov  [di], al       ; store result
    inc  si             ; increment pointers
    inc  bx
    inc  di
    loop add_loop

    jnc  continue_addition   ; BCD overflow if CF = 1
```

```
    lea  ax, overflow_message
    call print_string

continue_addition:
    mov  al, [si]       ; sign of top addend to result
    mov  [di], al

    POPREGS   ax, bx, cx, si, di
    pop  bp
    ret

_bcd_addition endp
; - - - - END SUBROUTINE ABOVE THIS LINE
```

AL contains the first number. We use the carry flag (ADC) for carrying from byte to byte. The carry flag is cleared before the first addition since we don't want a carry there. The INC instruction was designed by Intel so that it would not alter the carry flag just so we could use it in situations like this.

If CF = 1 upon exiting the loop, the result was too large and we had overflow. In this case we print a message to that effect.

The result [di] can be stored in the same place as one of the numbers. The addition of each byte is completed before the result for that byte is stored, so this won't interfere with the addition. We assume that the sign of the result is the sign of the top addend. (If this goes into a working program, it will only be used if both numbers have the same sign).

This is designed as a C subroutine. The calling program pushes things on the stack and it has the responsibility of popping them off the stack on return from the call.

Here's the calling routine:

```
; - - - - - ENTER DATA BELOW THIS LINE
bcd_num1 dt   ?
bcd_num2 dt   ?
bcd_num3 dt   ?
overflow_message   db  "We had overflow.", 0
; - - - - - ENTER DATA ABOVE THIS LINE
; - - - - - ENTER CODE BELOW THIS LINE

outer_loop:
    lea  ax, bcd_num1        ; get 2 numbers for addition
    call get_bcd
    lea  ax, bcd_num2
    call get_bcd

    lea  ax, bcd_num3        ; push addresses for subroutine
    push ax
    lea  ax, bcd_num2
    push ax
    lea  ax, bcd_num1
    push ax
```

```
    call _bcd_addition
    add  sp, 6                    ; 3 pushes = 6 bytes

    lea  ax, bcd_num1         ; print both numbers and result
    call print_bcd
    lea  ax, bcd_num2
    call print_bcd
    lea  ax, bcd_num3
    call print_bcd
    loop outer_loop

; - - - - - ENTER CODE ABOVE THIS LINE
```

This is the main program. The other one should be in the subroutine section. This program is straightforward. We get two numbers, call the subroutine, and print the numbers and the result. Try some numbers. The results you get will always have the sign of the top number and will have the same numerical result as adding two unsigned numbers.

The subtraction routine is the same as the addition but we need to make some changes because it is subtraction:

```
    ADC  ->   SBB
    DAA  ->   DAS  (decimal adjust for subtraction)

    add_loop:      ->   subtract_loop
    loop add_loop  ->   loop subtract_loop

    jmp  continue_addition   ->   jmp  continue_subtraction
    continue_addition:       ->   continue_subtraction:
```

Also, we want to change the subroutine name and call

```
    _bcd_addition       ->   _bcd_subtraction
    call _bcd_addition  ->   call _bcd_subtraction
```

Do all these changes, run the program, but make sure the top number is larger or you will get strange results.

We now have an addition routine that only works with the right numbers and a subtraction routine that is very touchy about the numbers that are put in. How can these things help us? In fact they are almost all we need. The only thing else we need is a preliminary subroutine to organize what we do. To see why we have some orginizational problems, take out a pencil and paper and do the following additions and subtractions. Do these with a pencil and paper, not a pocket calculator:

```
    (+15)     +    (+27)              (+15)     +    (-27)
    (+15)     -    (+27)              (+15)     -    (-27)
    (-15)     +    (+27)              (-15)     +    (-27)
    (-15)     -    (+27)              (-15)     -    (-27)
    (+27)     +    (+15)              (+27)     +    (-15)
    (+27)     -    (+15)              (+27)     -    (-15)
    (-27)     +    (+15)              (-27)     +    (-15)
    (-27)     -    (+15)              (-27)     -    (-15)
```

There are only four possible answers: +12, -12, +42 and -42. We had 16 different additions and subtractions, yet we got only 4 possible answers and only 2 possible absolute values. We could have a different subroutine for each one, but 16 subprograms is a LOT of code, so it's easier to do it with 2 subprograms. We merely need to order the numbers and pick the correct subroutine.

Here is the BCD driving routine. We'll get a number and then we'll get an operation, either addition or subtraction. If it is subtraction, when we get the second number, we REVERSE the sign. We don't even check what it is; plus becomes minus and minus becomes plus. What we have now is the ADDITION of two signed numbers. We XOR the two signs. If the result is 0, they both are the same sign and we can go to the addition subroutine. If the signs are different we need to subtract the smaller from the larger. We find out which one is larger and then call the subtraction routine. It is going to take you a little time to read this code.

```
; - - - - - START DATA BELOW THIS LINE
top_number    dt   ?
bottom_number dt   ?
result        dt   ?
sign_mask     db   ?
sign_message       db   "Enter either + or -.", 0
overflow_message   db   "We had overflow.", 0
; - - - - - END DATA ABOVE THIS LINE

; - - - - - START CODE BELOW THIS LINE
outer_loop:
    lea  ax, top_number
    call get_bcd

sign_loop:
    ; get either a '+' or a '-'
    lea  ax, sign_message    ; prompt for operation
    call print_string
    call get_ascii_byte      ; operation type in al
    cmp  al, '+'
    jne  check_for_minus
    mov  sign_mask, 00h      ; for XOR of bottom number sign
    jmp  get_second_number
check_for_minus:
    cmp  al, '-'
    jne  sign_loop           ; if not a minus then redo
    mov  sign_mask, 80h      ; for XOR of bottom sign

get_second_number:
    lea  ax, bottom_number
    call get_bcd
    ; XOR bottom sign with sign mask
    mov  al, sign_mask
    xor  BYTE PTR (bottom_number + 9), al   ; sign byte

    ; same sign or different signs?
    mov  ah, BYTE PTR (top_number + 9)      ; sign byte
```

```
    xor  ah, BYTE PTR (bottom_number + 9)   ; different?
    jnz  which_is_larger          ; if different, subtract

    ; same sign, so add
    lea  ax, result               ; push parameters and add
    push ax
    lea  ax, bottom_number
    push ax
    lea  ax, top_number
    push ax
    call _bcd_addition
    add  sp, 6                    ; adjust the stack
    jmp  print_the_numbers

which_is_larger:
    lea  si, top_number + 8       ; top digits
    lea  di, bottom_number + 8
    mov  cx, 9                    ; 9 bytes of digits
check_for_greater_loop:
    mov  al, [si]                 ; top number
    cmp  al, [di]                 ; bottom number
    ja   top_is_more
    jb   bottom_is_more
    dec  si                       ; equal, so continue
    dec  di
    loop check_for_greater_loop

    ; we fell through so they are the same
    ; leave the top number on top
top_is_more:
    lea  ax, result
    push ax
    lea  ax, bottom_number
    push ax
    lea  ax, top_number
    push ax
    call _bcd_subtraction
    add  sp, 6                    ; adjust the stack
    jmp  print_the_numbers

bottom_is_more:
    lea  ax, result
    push ax
    lea  ax, top_number
    push ax
    lea  ax, bottom_number
    push ax
    call _bcd_subtraction
    add  sp, 6                    ; adjust the stack

print_the_numbers:
    lea  ax, top_number
    call print_bcd
    lea  ax, bottom_number
    call print_bcd
    lea  ax, result
    call print_bcd
```

```
    jmp  outer_loop
; - - - - - END CODE ABOVE THIS LINE
```

The sign_mask is either 00h if it is addition or 80h if it is subtraction. 00h XOR sign_byte will leave the sign byte unchanged. 80h XOR sign_byte will reverse the sign of the sign byte.

Thus, as we did in the earlier multiplication and division routines, we sometimes change the sign of a number (bottom_number). In a real routine we would either have to make a copy of it when we change the sign or change the sign back at the end. Here we leave it with the sign change (if any) so you can see what is happening internally. If you want to restore the number, you can alter the code a little:

```
    print_the_numbers:
         mov  al, sign_mask                       ; add this
         xor  BYTE PTR (bottom_number + 9), al    ; add this
         lea  ax, top_number
```

This will restore the bottom number to its original form ASSUMING THAT THE RESULT HAS NOT BEEN STORED THERE.

It is possible to get -0 as a result. This is a legally defined number: +0 = -0.

There are three places where we have 'BYTE PTR'. This is because the variables are defined as 10 byte long objects and the assembler will complain if we don't put in the 'BYTE PTR'.

Finally, if we want to use the typical 'destination, source' style for the 8086, it is built in. Here it is for addition:

```
    lea  ax, top_number
    push ax
    lea  ax, bottom_number
    push ax
    lea  ax, top_number
    push ax
```

We put the address of 'top_number' where the result address goes. The routines store a byte only after all calculations are done on that particular byte, so there is no interference from doing this.

The first thing to talk about is the 8086 mnemonics. The four instructions for unpacked BCD numbers are:

```
    AAA       ASCII adjust for addition
    AAD       ASCII adjust for division
    AAM       ASCII adjust for multiplication
    AAS       ASCII adjust for subtraction.
```

Even though all four instructions have ASCII as part of their mnemonic, they have NOTHING to do with ASCII numbers. These instructions operate on unpacked BCD numbers. They always give results which are unpacked BCD numbers. Because of side effects of the 8086 microcode, the add and subtract instructions do something unusual, but this will be covered a little later.

Just like the packed BCD instructions, these four unpacked instructions use the normal arithmetic operations and adjust the results to compensate for their being unpacked BCD numbers. Let's start with addition. The following program is like the BCD program except that we use AAA (ascii adjust for addition) instead of DAA (decimal adjust for addition).

```
; - - - - - - - - - - START CODE BELOW THIS LINE
       mov   ax_byte, 0C4h       ; half registers, hex, hex
       mov   bx_byte, 94h        ; half regs, signed, hex
       mov   dx_byte, 91h        ; half regs, signed, signed
       lea   ax, ax_byte
       call  set_reg_style

       mov   cx, 0                     ; clear cx for clarity
outer_loop:
       mov   ax, 0                     ; clear the registers
       mov   bx, 0
       mov   dx, 0
       call  set_count
       call  show_regs

       call  get_hex_byte              ; byte for al
       mov   dx, ax                    ; copy for dx
       push  ax                        ; save al
       call  get_hex_byte              ; byte for bl
       mov   bl, al
       mov   bh, bl                    ; copy to bh
       pop   ax                        ; restore al
       call  show_regs_and_wait
       add   al, bl                    ; normal add
       mov   dx, ax                    ; copy to dx
       call  show_regs_and_wait
       aaa                             ; make adjustment
       mov   dx, ax                    ; copy to dx
       call  show_regs_and_wait
       jmp   outer_loop
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

Since this is a mixture of unpacked BCD instructions and normal integer instructions, a copy of AX (which is showing hex) is moved to DX (which is showing signed) each time AX is changed. Also, a copy of BL is put in BH.

For the rest of this chapter, I will refer to the one's digit, the ten's digit and the hundred's digit. In the number 346, 3 is the hundred's digit, 4 is the ten's digit, and 6 is the one's digit. In the number 928, 9 is the hundred's digit, 2 is the ten's digit, and 8 is the one's digit.

If two valid unpacked BCD numbers are added, the result will be between 0 and 18. This result MUST be in AL. AAA sets CF if this result is greater than 9 (i.e. there was a carry). AAA leaves the one's digit in AL and adds the ten's digit to AH. (The ten's digit can only be 0 for 0 - 9 or 1 for 10 - 18).

Run this. The standard input should be hex numbers between 00h and 09h. When you feel comfortable with this, stop for a minute and read on.

We now come to the peculiarity of this instruction. It isn't operating on the whole byte, only the lower half byte. Technically, if the LOWER HALF BYTE of each of the two numbers which are added is a legitimate BCD digit, then at the end of AAA the above results will be true and the UPPER HALF BYTE of AL will be set to 0. If you put anything in the upper half byte, it will be added by ADD, but will be blanked out (zeroed) by AAA.{1}

As a side effect of blanking out (zeroing) the upper half byte, it is possible to use the ASCII numbers '0' (30h) to '9' (39h) as addends. The result (after AAA) will still be a number between 00h and 09h with the carry flag either set or cleared, and AH incremented if appropriate. Don't use it this way. The multiplication and division instructions REQUIRE that the numbers be between 00h and 09h, so all numbers should be changed into unpacked BCD on data entry. Here is a partial list of things you can add to get the result 07h:

03h + 04h	'c' + 'd'	'+' + 't'
'3' + '4'	's' + 't'	'c' + '4'
'C' + 'D'	'#' + '$'	'S' + '$'

As you can see, the addition instruction is pretty indiscriminate about what kind of data you can put in and still get a legitimate unpacked number out. It is your job to make sure that you have

---

1. They used the same microcode as for the beginning of DAA. If the low half byte of AL is greater than 9, or if there was a carry out of the low half byte (if AF, the auxillary flag was set), then the 8086 adds 6 to AL, which shifts the excess out of the low half byte. It then zeros the high half byte, sets the carry flag, and increments AH. If you don't understand this, go back and give the footnote on DAA a try.

legitimate unpacked data upon data entry.

If this is just a side effect of blanking the upper half byte, why did they name the instruction "ascii adjust for addition"? It's just that impish sense of humor of those Intel engineers. If you think of these instructions as having anything to do with ASCII numbers, you will only confuse yourself. These are instructions on unpacked BCD numbers, period.

The other thing to notice is that this instruction increments the AH register if there is a carry, so whenever you use AAA, it is going to trash the AH register. Count on it. If AH contains important data, store it somewhere else.

SUBTRACTION

When you have gotten used to how this works, look at subtraction. The subtraction routine is the same as the addition routine except that (1) ADD is replaced by SUB and (2) AAA is replaced by AAS (ascii adjust for subtraction). If you generate a borrow, the carry flag will be set and AH will be decremented by 1.

If you have two legitimate unpacked BCD numbers, then after subtraction the result will be from +9 to -9. This result must be in AL. After AAS (ascii adjust for subtraction), (1) if AL was from 0 to +9, it will stay the same and CF will be cleared (CF=0) or (2) if AL was -1 to -9, AAF will borrow 1 from AH (considering it a ten's digit), and add 10 to AL. This will give a number from +1 to +9. It will also set the carry flag (CF=1) to signal a borrow. This is what you do with pencil and paper except that you always do the borrow BEFORE the subtraction and AAS always does the borrow AFTER the subtraction. Once again, this operates on the LOW HALF BYTE, so what is in the upper half byte of the numbers is irrelevant. It will be zeroed by AAS.

There is all sorts of data which will generate a legitimate unpacked result; some of it is actually legitimate input. It too trashes the AH register, so beware. Run this program till you see what is going on with individual numbers.

MULTIPLICATION

There is also an instruction for multiplication. It assumes that AL contains the result of the multiplication of two unpacked BCD numbers. That is, 0 <= AL <= 81. After AAM (ascii adjust for multiplication), AH will contain the 10's digit and AL will contain the 1's digit. If AL is 75, then after AAM, AH will be 7 and AL will be 5. If AL is 48, then after AAM, AH will be 4 and AL will be 8. (If AL is 183, an illegal result, then after AAM, AH will be 18 and AL will be 3).

We are going to use a similar program for multiplication, but AX and BL will be half byte unsigned; BH and DX will be half byte hex. Every time we change AX, we will copy it to DX. Here is the program:

```
; - - - - - - - - - - START CODE BELOW THIS LINE
       mov   ax_byte, 0A2h       ; half registers, unsigned
       mov   bx_byte, 0C2h       ; half regs, hex, unsigned
       mov   dx_byte, 0C4h       ; half regs, hex
       lea   ax, ax_byte
       call  set_reg_style

       mov   cx, 0               ; clear cx for clarity
outer_loop:
       mov   ax, 0               ; clear the registers
       mov   bx, 0
       mov   dx, 0
       call  set_count
       call  show_regs

       call  get_hex_byte        ; byte for al
       mov   dx, ax              ; copy to dx
       push  ax                  ; save al
       call  get_hex_byte        ; byte for bl
       mov   bl, al
       mov   bh, bl              ; copy to bh
       pop   ax                  ; restore al
       call  show_regs_and_wait
       mul   bl                  ; unsigned multiplication
       mov   dx, ax              ; copy to dx
       call  show_regs_and_wait
       aam                       ; make adjustment
       mov   dx, ax              ; copy to dx
       call  show_regs_and_wait
       jmp   outer_loop
 ; + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

All you need to change from the addition program is the register style, ADD -> MUL and AAA -> AAM.

Try this out. Notice that after MUL, what we get in DX is garbage. This number is a pure unsigned number, not a BCD number. Just to underline how important it is to have legitimate unpacked BCD data, try a few multiplications where the upper half byte is non-zero and see what happens.

DIVISION

AAD (ascii adjust for division) is slightly different. What we want to do is divide a number from 0 - 99 by a number from 1 to 9 (0 gives a zero divide interrupt). Therefore, AAD multiplies what is in AH by 10 and adds it to what is in AL. It clears AH for the coming unsigned division.{2} If AH contains 7 and AL contains 2, then after AAD, AL will be 72 and AH will be 0. If AH is 4 and AL is 6, then after AAD, AL will be 46 and AH will be 0. (If AH is 14 and AL is 7 - an illegal situation - then after AAD, AL will

---

2. Remember that for unsigned byte division, we set AH to 0. This was back in the first chapter on division.

be 147 and AH will be 0)

After you have done AAD, you are ready for regular unsigned division. After division, the quotient will be in AL and the remainder will be in AH. Here's the program:

```
; - - - - - - - - - - START CODE BELOW THIS LINE
       mov   ax_byte, 0A2h      ; half registers, unsigned
       mov   bx_byte, 0C2h      ; half regs, hex, unsigned
       mov   dx_byte, 0C4h      ; half regs, hex
       lea   ax, ax_byte
       call  set_reg_style

       mov   cx, 0              ; clear cx for clarity
outer_loop:
       mov   ax, 0               ; clear the registers
       mov   bx, 0
       mov   dx, 0
       call  set_count
       call  show_regs

       call  get_hex             ; word
       mov   dx, ax              ; copy to dx
       push  ax                  ; save al for later
       call  get_hex_byte        ; byte for bl
       mov   bl, al
       mov   bh, bl              ; copy to bh
       pop   ax                  ; get back al
       call  show_regs_and_wait
       aad                       ; adjust for division
       mov   dx, ax              ; copy to dx
       call  show_regs_and_wait
       div   bl                  ; unsigned division
       mov   dx, ax              ; copy to dx
       call  show_regs_and_wait
       jmp   outer_loop

; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

The differences from the multiplication routine are (1) we get a TWO byte hex number for the dividend and (2) AAD comes first, then DIV. That's all.

We need to enter a TWO byte unpacked BCD number in order to get grist for the mill. 87 is 0807h, 93 is 0903h, 42 is 0402h. This will give us two unpacked digits so we have something to put in AH.

Run this program and enter legitimate (and if you want, illegitimate) data. Notice that even with legitimate data, it is possible to get a quotient that is larger than 9. (47 / 3 is 15, remainder 2). Don't worry about this. We will avoid this problem in the real program.

PACKING AND UNPACKING

Making an i/o routine is such a bother that we will enter data with 'get_bcd' and output data with 'print_bcd'. To do this, we need to pack and unpack the data. The calls in C would be:

```
    unpack_bcd (&packed_number, &unpacked_number) ;
    pack_bcd (&unpacked_number, &packed_number) ;{3}
```

The thing to be operated on is the first variable and the result is the second variable. First, here's the unpacking routine:

```
; - - - - - - - - - -
unpack_bcd proc near

       UNPACK_UNPACKED_ADDRESS  EQU    [bp + 6]
       UNPACK_BCD_ADDRESS       EQU    [bp + 4]


       push  bp
       mov   bp, sp
       PUSHREGS  ax, bx, cx, si, di

       mov   si, UNPACK_BCD_ADDRESS
       mov   di, UNPACK_UNPACKED_ADDRESS

       ; start at top to unpack
       add   si, 9               ; bcd sign
       add   di, 18              ; unpacked sign
       mov   al, [si]            ; move sign to unpacked
       mov   [di], al
       dec   si                  ; move down to next byte
       dec   di

       mov   cx, 9               ; 9 bcd data bytes
       ; unpack high byte first, then low byte
unpack_loop:
       push  cx                  ; save counter
       mov   al, [si]            ; bcd byte to al
       mov   bl, al              ; copy to bl
       mov   cl, 4
       ror   al, cl              ; high half byte to low half
       and   al, 0Fh             ; blank upper half byte
       mov   [di], al            ; al is high half of bcd byte
       dec   di                  ; result pointer
       and   bl, 0Fh             ; blank upper half byte
       mov   [di], bl            ; bl is low half bcd byte
       dec   di                  ; result pointer
       dec   si                  ; source pointer
       pop   cx                  ; restore counter
       loop  unpack_loop

       POPREGS  ax, bx, cx, si, di
       pop   bp
       ret
```

---

3. That '&' means that we are passing the addresses, not the values.

```
unpack_bcd  endp
; - - - - - - - - - - - - - - - - -
```

This is pretty straightforward. First it moves the sign. Then, taking a BCD byte, the routine divides it in two parts, rotating the high half byte to the proper position, storing it, DECREMENTING the pointer and storing the low half byte. The upper half byte of each unpacked byte is zeroed. Notice that by starting at the top, it is possible to unpack a number in place. Diagram what is happening to make sure you believe that.

Here's the packing routine:

```
; - - - - - - - - - - - - - - - - -
pack_bcd     proc near

       PACK_BCD_ADDRESS       EQU     [bp + 6]
       PACK_UNPACKED_ADDRESS  EQU     [bp + 4]

       push  bp
       mov   bp, sp
       PUSHREGS  ax, bx, cx, si, di

       mov   si, PACK_UNPACKED_ADDRESS
       mov   di, PACK_BCD_ADDRESS

       ; start at bottom to pack
       mov   cx, 9               ; 9 bytes of bcd data
pack_loop:
       push  cx                  ; save counter
       mov   al, [si + 1]        ; high unpacked byte
       mov   cl, 4
       ror   al, cl              ; move to high half byte
       or    al, [si]            ; OR low half with high half
       mov   [di], al            ; move to BCD
       inc   di                  ; adjust pointers
       add   si, 2
       pop   cx                  ; restore counter
       loop  pack_loop

       ; si and di are now in the right place for the sign
       mov   al, [si]
       mov   [di], al

       POPREGS  ax, bx, cx, si, di
       pop   bp
       ret

pack_bcd endp
; - - - - - - - -
```

This does the reverse process, rotating the high byte to the proper place, then ORing the two together to form a packed BCD byte. Notice that by starting at the BOTTOM it is possible to pack a number in place. Diagram the action to make sure you believe this.

These two routines allow us to use 19 byte unpacked BCD numbers. Here's the multiplication routine for a 19 byte (18 data bytes and 1 sign byte) unpacked number by a 1 byte unpacked number:

```
; - - - - -
unpacked_multiply  proc near

       PUSHREGS  ax, bx, cx, dx, si, di
       mov   si, offset multiplicand
       mov   di, offset result
       mov   dh, 0                    ; clear dh for carry
       mov   bl, multiplier_copy
       mov   cx, 18                   ; 18 data bytes

mult_loop:
       mov   al, [si]                 ; multiplicand to al
       mul   bl                       ; multiply
       add   al, dh                   ; add old carry
       aam                            ; adjust al
       mov   [di], al                 ; partial result
       mov   dh, ah                   ; save carry
       inc   si                       ; adjust pointers
       inc   di
       loop  mult_loop

       mov   extra_byte, ah           ; extra byte
       POPREGS  ax, bx, cx, dx, si, di
       ret

unpacked_multiply  endp
; - - - - -
```

This is a clone of what we had in the chapter on multiple word multiplication but is byte multiplication instead of word multiplication. Refer back to that chapter to understand the mechanics of the process. We store the partial result and save the high byte for addition with the next multiplication. At the end we save the 19th byte for printout.

We are cheating a little. we have:

```
       mul   bl                       ; multiply
       add   al, dh                   ; add old carry
       aam                            ; adjust al
```

when we should have:

```
       mul   bl                       ; multiply
       aam                            ; adjust al
       add   al, dh                   ; add old carry
       aaa                            ; adjust al
```

The reason this works is that the maximum multiplication is 9X9=81. The maximum addition is 81+9 = 90. AAM will work correctly with any number 99 or less, so we save a step; we do one adjustment instead of two.

Now, let's look at the driver for the program:

```
; + + + + + START DATA BELOW THIS LINE
multiplier_message  db    "Enter a number from -9 to +9", 0
bcd_in              dt    ?
bcd_out             dt    ?
multiplicand        db    19 dup (?)
multiplier          db    ?
multiplier_copy     db    ?
result              db    19 dup (?)
extra_byte          db    ?
result_sign         db    ?
; + + + + + END DATA ABOVE THIS LINE


; + + + + + START CODE BELOW THIS LINE
outer_loop:
       mov   ax, offset bcd_in
       call  get_bcd
       call  print_bcd           ; reprint for clarity
       lea   cx, multiplicand
       push  cx                  ; unpacked address
       push  ax                  ; bcd_in address
       call  unpack_bcd
       add   sp, 4               ; adjust the stack
       mov   al, multiplicand + 18      ; sign byte
       mov   result_sign, al     ; either 00h or 80h

enter_multiplier:
       lea   ax, multiplier_message
       call  print_string
       call  get_signed_byte
       ; check for valid multiplier
       cmp   al, 9               ; > +9 ?
       jg    enter_multiplier
       cmp   al, -9              ; < -9?
       jl    enter_multiplier

       ; adjust multiplier sign
       mov   multiplier, al
       mov   multiplier_copy, al
       and   al, 80h             ; sign bit set?
       jz    do_the_multiplication
       ; negative multiplier, so adjust
       neg   multiplier_copy     ; make positive
       xor   result_sign, 80h    ; reverse sign of result

do_the_multiplication:
       call  unpacked_multiply

       mov   al, result_sign   ; transfer sign to result
       mov   result + 18, al

       ; pack the result
       mov   ax, offset bcd_out  ; bcd number
       push  ax
```

```
        mov   ax, offset result   ; unpacked number
        push  ax
        call  pack_bcd
        add   sp, 4               ; adjust the stack

        ; print multiplicand, multiplier, extra byte and result
        mov   ax, offset bcd_in
        call  print_bcd
        mov   al, multiplier
        call  print_signed_byte
        mov   al, extra_byte
        call  print_hex_byte
        mov   ax, offset bcd_out
        call  print_bcd

        jmp   outer_loop
; - - - - - END CODE ABOVE THIS LINE
```

We enter a multiplicand, unpack it, enter a number from -9 to +9 and save a copy of its absolute value as well as the sign the result will be. We adjust the sign of the result after the multiplication. No matter what you enter, the sign will be correct, and if you put the 19th byte in front of the BCD number which you have printed, the absolute value will be correct. We are multiplying with a COPY of the multiplier, so we can reprint the actual multiplier; it hasn't changed. At the end we pack the result and print everything. The order of printout is multiplicand, multiplier, extra byte and result.

Notice that we are not passing the parameters for unpacked_multiply on the stack. This is pure whim. A rule for when to pass on the stack is (1) If the subroutine doesn't know where the parameters will be located in memory, you MUST pass them on the stack but (2) if the subroutine knows for certain where the parameters will be, it can fetch them itself.

DIVISION

Here is the division routine for 19 byte unpacked numbers:

```
; - - - - - - - -
unpacked_divide  proc near

        PUSHREGS  ax, bx, cx, si, di

        mov   bl, divisor_copy
        mov   si, offset dividend + 17  ; start at the top
        mov   di, offset quotient + 17
        mov   cx, 18              ; 18 numeric bytes
        mov   ah, 0               ; clear ah for division

div_loop:
        mov   al, [si]            ; dividend byte to al
        aad                       ; adjust for unpacked number
        div   bl                  ; bl is divisor
        mov   [di], al            ; move partial quotient
        dec   si                  ; decrement pointers
```

```
       dec   di
       loop  div_loop

       mov   remainder, ah       ; final remainder

       POPREGS  ax, bx, cx, si, di
       ret

unpacked_divide  endp
; - - - - -
```

It is pretty simple; it too is a clone of the multiple word division process. Go back to multiple word division if you don't understand it. We keep the previous remainder for the next division. Here is the driver:

```
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
divisor_message db "Enter a number from -9 to +9 (but not 0).",0
bcd_in          dt     ?
bcd_out         dt     ?
quotient_sign   db     ?
remainder_sign  db     ?
divisor         db     ?
divisor_copy    db     ?
dividend        db     19 dup (?)
quotient        db     19 dup (?)
remainder       db     ?
; + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
outer_loop:
       mov   ax, offset bcd_in
       call  get_bcd
       call  print_bcd           ; reprint for clarity
       lea   cx, dividend
       push  cx                  ; unpacked address
       push  ax                  ; bcd_in address
       call  unpack_bcd
       add   sp, 4               ; adjust the stack
       mov   al, dividend + 18   ; sign byte
       mov   quotient_sign, al   ; either 00h or 80h
       mov   remainder_sign, al  ; ditto

enter_divisor:
       lea   ax, divisor_message
       call  print_string
       call  get_signed_byte
       ; check for valid divisor
       cmp   al, 0               ; 0?
       je    enter_divisor
       cmp   al, 9               ; > +9 ?
       jg    enter_divisor
       cmp   al, -9              ; < -9?
       jl    enter_divisor

       ; adjust divisor, quotient sign
       mov   divisor, al
```

```
        mov   divisor_copy, al
        and   al, 80h             ; sign bit set?
        jz    do_the_division
        ; negative divisor, so adjust
        neg   divisor_copy        ; make positive
        xor   quotient_sign, 80h  ; reverse sign of quotient

do_the_division:
        call  unpacked_divide

        mov   al, quotient_sign   ; transfer sign to quotient
        mov   quotient + 18, al
        test  remainder_sign, 0FFh  ; positive or negative?
        jz    pack_the_quotient
        neg   remainder             ; make the remainder negative

pack_the_quotient:
        mov   ax, offset bcd_out  ; bcd number
        push  ax
        mov   ax, offset quotient ; unpacked number
        push  ax
        call  pack_bcd
        add   sp, 4               ; adjust the stack

        ; print dividend, divisor, quotient and remainder
        mov   ax, offset bcd_in
        call  print_bcd
        mov   al, divisor
        call  print_signed_byte
        mov   ax, offset bcd_out
        call  print_bcd
        mov   al, remainder
        call  print_signed_byte

        jmp   outer_loop
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

Enter a BCD number, unpack it. Enter a signed number, do range checking, and if it is ok, store it (along with a copy, which we make sure is positive). Calculate the sign of the quotient and remainder. Do the division, then print the dividend, divisor, quotient and remainder. (which have been sign adjusted). The remainder is a signed byte.

Like unpacked_multiply, unpacked_divide fetches the parameters directly from memory rather than from the stack.

SUMMARY

PACKED BCD INSTRUCTIONS

DAA (decimal adjust for addition) adjusts AL, assuming that it contains the result of a legitimate packed BCD addition. It treats AL as two independent half-bytes. If the result of the lower half-byte is 10 or over, it subtracts 10 from the lower half-byte and adds the carry to the upper half byte. It then looks at the upper half byte. If its result is 10 or over, DAA subtracts 10 from the upper half byte and sets the carry flag. Otherwise the carry flag is cleared.

DAS (decimal adjust for subtraction) adjusts AL, assuming that it contains the result of a legitimate packed BCD subtraction.It treats AL as two independent half-bytes. If the result of the lower half-byte is -1 or less, it adds 10 to the lower half-byte and borrows 1 from the upper half byte. It then looks at the upper half byte. If its result is -1 or less, DAS adds 10 to the upper half byte and sets the carry flag to indicate a borrow. Otherwise the carry flag is cleared.

UNPACKED BCD INSTRUCTIONS

AAA (ascii adjust for addition) adjusts AL, assuming that it contains the result of a legitimate unpacked BCD addition. If the lower half-byte has generated a result 10 or over, it subtracts 10, carries 1 to AH, and sets the carry flag. If the result is 9 or less, it clears CF. In either case it zeroes the upper half-byte of AL.

AAD (ascii adjust for division) PREPARES AL and AH for division. It assumes that AH contains the 10's digit and AL contains the 1's digit of a two byte unpacked BCD number. It multiplies AH by 10 and adds it to AL, thus making a single integer between 0 and 99. It zeroes AH in preparation for division.

AAM (ascii adjust for multiplication) adjusts AL, assuming that it contains the result of a legitimate BCD multiplication. It divides the result by 10, putting the quotient in AH and the remainder in AL.

AAS (ascii adjust for subtraction) adjusts AL, assuming that it contains the result of a legitimate unpacked BCD subtraction. If the lower half-byte has generated a result -1 or less, it borrows 1 from AH, adds 10 to AL, and sets the carry flag. If the result is 0 or more, it clears CF. In either case it zeroes the upper half-byte of AL

CHAPTER 23 - XLAT

The 800 pound gorilla in the computer field is, of course, IBM. It can go its own way and other companies have to adjust to keep themselves in line with what IBM is doing.

You have been using ASCII characters since the first time you used BASIC (or whatever your first high-level language was). Every character has a unique number which represents it.

character	ASCII encoding
A	65d
a	97d
?	63d
0	48d

IBM has its own encoding for mainframe computers. It is called EBCDIC (pronounced ebb'-sih-dick).{1} It is a spinoff of the coding on punch cards. You remember punch cards? This coding is entirely different from ASCII. Here are some examples.

character	ASCII code	EBCDIC code
a	97d	129d
?	63d	111d
0	48d	240d
H	72d	200d
I	73d	201d
J	74d	209d
K	75d	210d

You can see that there is no relationship between the two encodings. Also, notice that while the alphabet is a continuous section of ASCII coding, there are breaks in the EBCDIC code (I=201, J=209).

All PCs use ASCII, so if we want to transfer text from a PC to an IBM mainframe computer, we need to change ASCII -> EBCDIC going to the mainframe and change EBCDIC -> ASCII coming from the mainframe. This is the responsibility of the communications program that runs the modem, so you will never have to do it yourself. Intel has provided an instruction to help the communications program do this translation. It is called XLAT.

In order to use XLAT, you need a translation table. This is a 256 byte array where each element of the array contains the result you want. Looking at the data above:

---

1. Which stands for Extended Binary Coded Decimal Interchange Code.

CHARACTER	ASCII TO EBCDIC TABLE	EBCDIC TO ASCII TABLE
a	array1 [97] = 129	array2 [129] = 97
?	array1 [63] = 111	array2 [111] = 63
0	array1 [48] = 240	array2 [240] = 48
H	array1 [72] = 200	array2 [200] = 72
I	array1 [73] = 201	array2 [201] = 73
J	array1 [74] = 209	array2 [209] = 74
K	array1 [75] = 210	array2 [210] = 75

We have two different tables here. Array1 takes the ASCII encoding and gives back the EBCDIC encoding. Array2 takes the EBCDIC encoding and gives back the ASCII encoding. For each character, the appropriate table gives the correct translation from one encoding to another. All we need now is the translation instruction. Put the address of the translation table in BX. This table should be in the DS segment, but DS may be overriden:

```
    mov  bx, offset ascii_to_ebcdic_table
```

Put the character you want translated in al:

```
    mov  al, character

translate:

    xlat
```

To translate a 20 byte string of ASCII data into EBCDIC, you might have the following code:

```
;----------
    mov  di, offset ebcdic_string
    mov  ax, seg ebcdic_string
    mov  es, ax

    mov  si, offset ascii_string

    mov  bx, offset ascii_to_ebcdic_table
    mov  cx, 20                   ; translate 20 bytes
    cld                           ; clear DF (increment)

translation_loop:
    lodsb                         ; ascii to al
    xlat                          ; translate
    stosb                         ; al to ebcdic
    loop translation_loop
; ----------
```

Since this is ASCII to EBCDIC, if AL contained 63 before XLAT, then after XLAT AL would contain 111. If AL contained 73 before XLAT, then after XLAT it would contain 201. If AL contained 97 before XLAT, after XLAT it would contain 129.

If we wanted to go the other direction we would have to make the EBCDIC string the source string, make the ASCII string the

destination string, and use the other table:

```
    mov  bx, offset ebcdic_to_ascii_table
```

The rest of the code would be the same.

Since this is done by the communications program, we won't concern ourselves with ASCII <-> EBCDIC any more, but we will use XLAT in two slightly different ways.

First, let's categorize characters. Some things are Whitespace (that is, tabs, newlines, spaces, form feeds, etc.) Some characters are octal, decimal, punctuation, hex, etc. There is a pre-existing table called translation_table in the subdirectory XTRAFILE. Its pathname is \xtrafile\transtbl.obj. It has all 256 ascii characters coded in the following way:

```
WHITESPACE    EQU  80h     ; 1000 0000
PUNCTUATION   EQU  40h     ; 0100 0000
ALPHABETIC    EQU  20h     ; 0010 0000
OCTAL         EQU  10h     ; 0001 0000
DECIMAL       EQU  08h     ; 0000 1000
HEX           EQU  04h     ; 0000 0100
BOX_CHAR      EQU  02h     ; 0000 0010
GREEK_CHAR    EQU  01h     ; 0000 0001
```

If the character is whitespace, then the leftmost bit is set. If it is a greek character (ascii 224 - 239 on the PC) then the rightmost bit is set. If it is more than one thing, then the appropriate bits are set. For instance, '6' is octal, decimal and hex, so it's encoding is:

```
    '6'  0001 1100
```

'a' is both alphabetic and hex, so it's encoding is:

```
    'a'  0010 0100
```

The following program inputs a character, and finds out whether it is punctuation, a letter, etc. If it is none of the eight things, then the program prints that nothing was found. It is the same block of code over and over, so you might want to do only part, or you might want to cut it out with a word processor and insert it in the template file (don't forget to delete the page headers and page numbers).

```
; + + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
EXTRN  translation_table:BYTE  ;\xtrafile\transtbl.obj

whitespace_banner   db "It is whitespace." , 0
punctuation_banner  db "It is punctuation." , 0
alphabet_banner     db "It is alphabetic." , 0
octal_banner        db "It is octal." , 0
decimal_banner      db "It is decimal." , 0
```

```
hex_banner          db "It is hex." , 0
drawing_banner      db "It is a box drawing character." , 0
greek_banner        db "It is a Greek character." , 0
nothing_banner      db "No match was found." , 0

dirty_flag          db ?
; + + + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + + + START CODE BELOW THIS LINE
       WHITESPACE    EQU  80h
       PUNCTUATION   EQU  40h
       ALPHABETIC    EQU  20h
       OCTAL         EQU  10h
       DECIMAL       EQU  08h
       HEX           EQU  04h
       BOX_CHAR      EQU  02h
       GREEK_CHAR    EQU  01h

       ; set up the xlat table
       mov   ax, seg translation_table
       mov   es, ax
       mov   bx, offset translation_table

outer_loop:
       mov   dirty_flag, 0       ; marker for success
       call get_ascii_byte       ; input a byte to al
       xlat es:[bx]              ; do the translation

       test  al, WHITESPACE
       jz    punct_check
       push  ax                  ; save translation in al
       mov   ax, offset whitespace_banner
       call  print_string
       pop   ax
       mov   dirty_flag, 1       ; set the dirty flag

punct_check:
       test  al, PUNCTUATION
       jz    alpha_check
       push  ax                  ; save translation in al
       mov   ax, offset punctuation_banner
       call  print_string
       pop   ax
       mov   dirty_flag, 1       ; set the dirty flag

alpha_check:
       test  al, ALPHABETIC
       jz    octal_check
       push  ax                  ; save translation in al
       mov   ax, offset alphabet_banner
       call  print_string
       pop   ax
       mov   dirty_flag, 1       ; set the dirty flag

octal_check:
       test  al, OCTAL
       jz    decimal_check
```

```
        push  ax                  ; save translation in al
        mov   ax, offset octal_banner
        call  print_string
        pop   ax
        mov   dirty_flag, 1       ; set the dirty flag

decimal_check:
        test  al, DECIMAL
        jz    hex_check
        push  ax                  ; save translation in al
        mov   ax, offset decimal_banner
        call  print_string
        pop   ax
        mov   dirty_flag, 1       ; set the dirty flag

hex_check:
        test  al, HEX
        jz    drawing_check
        push  ax                  ; save translation in al
        mov   ax, offset hex_banner
        call  print_string
        pop   ax
        mov   dirty_flag, 1       ; set the dirty flag

drawing_check:
        test  al, BOX_CHAR
        jz    greek_check
        push  ax                  ; save translation in al
        mov   ax, offset drawing_banner
        call  print_string
        pop   ax
        mov   dirty_flag, 1       ; set the dirty flag

greek_check:
        test  al, GREEK_CHAR
        jz    nothing_check
        push  ax                  ; save translation in al
        mov   ax, offset greek_banner
        call  print_string
        pop   ax
        mov   dirty_flag, 1       ; set the dirty flag

nothing_check:
        cmp   dirty_flag, 0       ; was anything found?
        je    print_nothing_banner
        jmp   outer_loop
print_nothing_banner:
        mov   ax, offset nothing_banner
        call  print_string
        jmp   outer_loop

; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE

you need to:

    link prog1+transtbl+\asmhelp ;
```

The program is long, but straightforward. Input a character and get its encoding. Test for each characteristic. If it is found, print the appropriate message and set the dirty_flag to indicate something was printed. At the end, if nothing was printed, print the failure message.

Notice that the translation table is in ES and we are using a segment override for it. If you look at the EXTRN statement for 'translation_table', you will see that even though we are using ES, it is declared EXTRN in a segment with an:

```
ASSUME ds:DATASTUFF
```

statement. How can we get away with this? The assembler never deals with 'translation table' directly. The only thing it does is put the offset in BX. We put the segment override in ourselves with:

```
xlat es:[bx]
```

so the assembler never has to decide whether a segment override is necessary or which segment override to use.

WORD SEARCH

When doing the mock word search program in the chapter on string instructions, I mentioned that it really wouldn't cut the mustard when it comes to real word searches. Why? If we are looking for "when" we also want to find "When". If we are looking for " searches ", we also want to find " searches,", that is, punctuation should not interefere unless we want it to, and capitals should not interefere unless we want them to. With the aid of a translation table, we will make a word search program which uses the following rules. In the SEARCH string (the string that defines what you are looking for):

(1) Any small letter will match either a small or large letter.
(2) A capital letter will match only a capital letter.
(3) A blank will match any whitespace or punctuation.
(4) A punctuation mark will only match itself.

With these rules "Why" must start with a capital 'W' to be a match, but 'h' and 'y' may be either capital or small. " some," may have any whitespace (including a carriage return) in front, but must hava a comma ',' at the end.

This program has two data files. \XTRAFILE\SRCHTBL.OBJ contains the translation table. It is called "wordsearch_table" and is in DATASTUFF, so will be in our normal DS segment. In order to have text to search I have included an object file that is the text of a chapter from a book. (The object file text includes carriage returns). The text is a C string - it is terminated by a 0.

The book was written by C.D. Huffam, and is the autobiographical account of his dual life as a writer and lecturer. The book is

called "A Tale of Two C.D.s". The object file with the text is \XTRAFILE\TWOTALE.OBJ. It is in a private segment and will use ES as a segment register. There is also a straight text file which you can print out so you can see what is in the object file. It is \XTRAFILE\TWOTALE.DOC.

Here's the program. The explaination is at the end.

```
; + + + + + + + + + + + + + + + START DATA BELOW THIS LINE
EXTRN  tale_text:BYTE, wordsearch_table:BYTE

entry_message    db   13,10, "Enter a word for a word search", 0
no_match_message db   "There was no match", 0
input_buffer     db   80 dup (?)
text_file_length dw   ?
letter_count     dw   ?
; + + + + + + + + + + + + + + + END DATA ABOVE THIS LINE

; + + + + + + + + + + + + + + + START CODE BELOW THIS LINE

       ; find the length of the text file
       mov   ax, seg tale_text      ; load es register
       mov   es, ax

       mov   di, offset tale_text   ; offset to di
       mov   bx, di                 ; copy to bx
       mov   al, 0                  ; try to match zero
       cld                          ; clear DF (increment)

string_end_loop:
       scasb                        ; search for zero
       jne   string_end_loop

       dec   di                  ; one too many , so decrement
       sub   di, bx                 ; finish - start = length
       mov   text_file_length, di     ; length of text_file


big_loop:
       ; get a word for the word search
       mov   ax, offset entry_message
       call  print_string
       mov   ax, offset input_buffer
       call  get_string

       ; find the end of string
       mov   al, 0               ; compare with 0
       mov   bx, offset input_buffer
       mov   cx, 0               ; letter count
letter_count_loop:
       cmp   al, [bx]            ; compare to 0
       je    end_of_count_loop
       inc   cx                  ; increment count
       inc   bx                  ; increment pointer
       jmp   letter_count_loop
end_of_count_loop:
       cmp   cx, 0               ; if 0, string is empty
```

```
       je    big_loop            ; so start again
       mov   letter_count, cx

       ; look for word match. In this program, the text string
       ; is referenced by si and the search string is referenced
       ; by di.

       mov   si, offset tale_text
       mov   cx, text_file_length    ; length of file
       sub   cx, letter_count        ; last possible match
       inc   cx                      ; +1 for boundary condition

       ; set up translation table ( it is in DATASTUFF )
       mov   bx, offset wordsearch_table


word_search_loop:
       push  si                  ; save a copy
       push  cx                  ; save a copy
       mov   di, offset input_buffer
       mov   cx, letter_count


letter_loop:
       mov   al, es:[si]         ; text to al
       cmp   al, [di]            ; same as search string?
       je    next_letter
       xlat                      ; if not, translate
       cmp   al, [di]            ; allowable substitute?
       jne   new_start           ; if not, start at new place
next_letter:
       inc   di                  ; move to next letter
       inc   si
       loop  letter_loop

       ; we fell through, so we found a complete match
       jmp    found_it

       ; no match. are we finished?
new_start:
       pop   cx
       pop   si
       inc   si                  ; move to next character
       loop  word_search_loop

       ; we fell through. finished, but no match
       mov   ax, offset no_match_message
       call  print_string
       jmp   big_loop

found_it:
       pop   cx                  ; take cx off the stack
       pop   si                  ; start of the match

       ; move 25 characters to buffer for printing
       mov   di, offset input_buffer
       mov   cx, 25
```

```
character_move:
       mov   al, es:[si]
       mov   [di], al
       inc   si                    ; increment pointers
       inc   di
       loop  character_move

       mov   BYTE PTR [di], 0      ; end of string
       mov   ax, offset input_buffer
       call  print_string
       jmp   big_loop
; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

You need to:

```
    link prog2+twotale+srchtbl+\asmhelp ;
```

to get asmhelp and the two data files in the program.

This program is very similar to the search program in the chapter on strings. However, because of where the files are, the pointers have been changed around. Therefore, it is safer if you simply cut out the program with a word processor and paste it into the template file rather than try to modify the prevoius search program.{2}

It is assumed that you did the string match program. The logic is the same and will not be covered again. First we input a search string. Then starting at the beginning of the text to be search we check till we find the first match. If we find a match, we print out 25 characters starting with the first character of the match. If no match is found, a message to that effect is printed.

The character match is a two step process. The character from the text is put in AL. It is compared with the search character for an EXACT match. If they match, we are done. If not, we use XLAT on AL (the character from the text) which will translate to its allowable substitute. In fact, all this is just: (1) all capital letters become small, (2) all punctuation becomes spaces, and (3) all whitespace becomes spaces. Once again, we compare AL with the search character. If we have a match, ok. If not, we start over.

The text is in ES, the translation table is in DS, so it is inconvenient to use the string instructions in this program.

Try to match a word at the beginning of the line, end of the line, with and without punctuation and with and without capitals. If you go across a line break, you need to substitute two blanks in the search string for CRLF (13,10).

---

2. You should understand what is going on in the code before you run these programs. I didn't write the code for myself, I wrote it for you. If you run it but don't understand it, it won't help you a bit.

Suppose you are not interested in all 256 values of the translation table. Let's say that you only want to have a translation table for the numbers from 0 to 99. Can you still use this? Yes, but you need to put in some range checking to make sure that you have valid data.

```
    MAX_VALUE EQU 99

    mov  al, data_byte       ; byte to al
    cmp  al, MAX_VALUE       ; too large?
    ja   data_error          ; report error
    xlat
```

This insures that any data that is out of range is not translated. Therefore the translation table only needs to be 100 bytes long (0 - 99).

If you want more than 256 elements in the translation table you need to use words, not bytes, and you cannot use XLAT. You can make your own code to do the same thing.

```
    MAX_VALUE EQU 999
    my_translation_table     dw   1000 dup (?)
```

if you put the translation data into the table, you can then have the following code:

```
    mov  bx, offset my_translation_table

    ; - - - - - translation block
    mov  si, data_word       ; word to si
    cmp  si, MAX_VALUE       ; too large?
    ja   data_error
    shl  si, 1         ; SI x 2 = number of BYTES into table
    mov  ax, [bx+si]         ; base + offset
    ; - - - - - end of translation block
```

XLAT is about twice as fast as this last code, so when you have a choice always use XLAT.

SUMMARY

XLAT

BX holds the address of a 256 byte array called a translation table. AL holds the character to be translated. If x is the value in AL before XLAT, then after XLAT, AL=array[x].

CHAPTER 24 - MISCELLANEOUS INSTRUCTIONS

There are a few more assembler instructions which have not been covered. Some are seldom used and some are used in special circumstances. This chapter gives a brief explanation of them.

XCHG

XCHG, the exchange instruction, switches the contents of two registers or of a register and a memory location.

```
xchg ax, bx
xchg al, dl
xchg variable1, si
xchg ch, variable2
xchg [si], ax
xchg bh, [di]
```

It operates on either a word or a byte. It cannot switch two memory locations, and it does not operate on the segment registers, only on the 8 arithmetic registers.

ESC

The escape instruction is not a complete instruction, it is the beginning of an instruction. It signals the 8086 that the first two bytes of the instruction contain a co-processor instruction. The 8087 is a mathematics co-processor. It reads the instructions at the same time as the 8086, and when it sees an escape instruction meant for it, it performs that operation. The 8086 does nothing unless there is a memory address involved. In that case the 8086 calculates the address and gives the address to the 8087.

```
fld  DWORD PTR [si]
fmul st, st(3)
fisub WORD PTR [di]
```

are all 8087 instructions that the assembler codes with the escape code. You will never code the escape instruction yourself.

WAIT (FWAIT)

The 8087 operates independently of the 8086. They can both perform operations at the same time, but it is possible for them to interfere with each other. If both the 8086 and 8087 are reading from or writing to memory, or if one is reading from while the other is writing to memory, the read/write will be corrupted. In order to stop this, whenever you access memory with the 8087, you need to put in a WAIT instruction. WAIT (or FWAIT

which is the same thing), suspends operation of the 8086 until the co-processor is finished. It would look like this:

```
fstp DWORD PTR [bx]
fwait
```

The 8086 will wait until the 8087 is finished storing into memory.

There must also be a wait between 8087 instructions. This is to keep the 8087 from starting a second instruction before it is finished with the first one. The instruction execution is done by the 8087, but the timing is done by the 8086. If you had the folllowing code:

```
fmul st, st(3)
fadd st, st(2)
fsub st, st(4)
```

the 8086 would be past the third instruction before the 8087 had time to finish doing the first instruction. The proper coding is:

```
fmul st, st(3)
fwait
fadd st, st(2)
fwait
fsub st, st(4)
```

This should concern you only if you program the 8087. It is outside the realm of this book. Remember, WAIT and FWAIT are the same instruction.

LOCK

LOCK is not really of concern to a PC programmer. On some systems it is possible to have more than one 8086 that has access to the same memory. The problem then arises as it did with the 8087 that there is the possibility of two 8086s doing read/write operations to memory at the same time. This will result in corrupted data. LOCK allows an 8086 to lock out other 8086s. It will be the only one allowed to read to or write to memory during the next instruction. This is mostly of concern to people who write code for peripheral devices which have DMA (direct memory access).

LOOPE/LOOPZ LOOPNE/LOOPNZ

We have used the loop instruction, but these are special cases. Remember, the general loop instruction decrements CX by 1, and if the result is not zero, it jumps to the top of the loop. In special circumstances, you not only want to check the counter in CX, but you also want to check the result of some other operation.

```
test  ax, 5
loope outer_loop
```

will loop if cx is not zero AND ax = 5

```
    test   ax, 5
    loopne outer_loop
```

will loop if cx is not zero AND ax is not 5.

HALT

HALT actually halts the operation of the 8086. It can only be started again by a reset or an interrupt. If you write:

```
    cli       ; clear interrupt flag
    halt
```

Then normal interrupts can't happen and you have effectively shut down the system. One place this might be of use is if you have a copy protection scheme and you detect that someone has violated it. It halts the system and you need to reset to start again. Another place might be if you are writing a game - someone inadvertently enters the "Dungeon of Darkness" and you shut the system down.

CMC

CMC (complement the carry flag) toggles the carry flag. If it is off this turns it on, and if it is on, this turns it off. Why this instruction exists is a complete mystery to me. Why would you want it as part of the instruction set?

LAHF

LAHF (load AH from flags) stores half of the flags in AH. The register looks like this:

```
    7 6 5 4 3 2 1 0
    S Z   A   P   C
```

Where these are the Sign, Zero, Auxillary, Parity and Carry flags. This is a leftover from earlier Intel chips. All these flags are testable so you dont need to look at them, and if you want to save them, then:

```
    pushf
```

does the trick. Notice that AH does not contain DF, the direction flag or IEF, the interrupt enable flag, which are things you CAN'T test with a jump instruction. To test for them you need to:

```
    pushf
    pop  ax
```

Then AX contains all the flags.

SAHF

SAHF (store AH to flags) is the counterpart to the above instruction. It puts AH into the low half of the flags register. The comments about LAHF apply to this instruction also.

NOP

And finally, NOP does absolutely nothing. It is there in case you need to fill space, either because you have taken out an instruction or for reasons of alignment. It also allows a debugger to put the single byte INT (int 3) in the code and then replace it with NOP when the breakpoint is no longer desired.

CHAPTER 25 - WHAT DOES IT ALL MEAN?

What does it all mean? Not a whole lot, actually. We can now save memory space and we can save a lot of time. But with the incessant march of technology these things mean less and less. A few years ago, when 64k or 128k was a lot of memory and memory was expensive, having a 20k program instead of a 40k program was a significant advantage. Now it means almost nothing unless it is a memory resident program. What about disk space? Just a while back we were operating with two 360k floppy disks and a hard disk was too expensive. Nowdays everyone has a 20meg hard disk.{1} And speed? Those programs that were slow on an 8088 now seem o.k. on an 80386. Those programs that were unbearably slow on the 8088 died a quick death and are no longer around.

Compilers are better and they have more subroutines available. They are also easier to program than going to the assembler level. What this chapter is about is when NOT to use the standard compiler functions and subroutines.

First, you should understand that all compiler subroutines are general purpose subroutines. They need to be all things to all people. Imagine what a vehicle would be like if we gave the designer the following specification:

> We want to be able to drive to the store for groceries. It should be fuel efficient. In case we want to go into the mountains it should be an all terrain vehicle. We also want to be able to haul a roomful of furniture from coast to coast. Oh yes, and we want to be able to race it at Le Mans.

Being universal requires a lot of code and it slows things down. Whether this extra code and time is too much is a question you need to decide for yourself. First, here are some examples of size. This is a C program that does almost nothing:

```
#include <stdio.h>
main()
{
        int     x ;
        x = 27 ;                        /* line 1 */
        scanf  ( "%d", &x ) ;           /* line 2 */
        printf ( "%d\n", x ) ;          /* line 3 */
}
```

---

1. Which has led to one of my pet peeves. All installation programs for compilers and word processors dump EVERYTHING on the hard disk. This gives us subdirectories that have 50 files in them, and we don't have the foggiest notion of what any of the files are for. If these installation programs would only prompt us by type of file to find out what we want to install and want to leave off the hard disk, we would all be better off.

I have made 3 programs from this. LINE1.C has line1 only. LINE2.C has lines 1 and 2. LINE3.C has lines 1, 2 and 3. For those non C people, scanf is an input function, printf is an output function. Guess how big each program is. Here's the directory listing.

```
LINE1    EXE     3176   6-22-90   8:48a
LINE2    EXE     7170   6-22-90   8:49a
LINE3    EXE     9134   6-22-90   8:49a
```

It takes 3000 bytes to start a C program (this is the startup module) 4000 bytes more to enter something and 2000 bytes extra to print something. That first 3000 bytes is unavoidable if you are writing in a high level language. If you are doing a lot of general purpose i/o, these extra amounts aren't too bad. There are two cases where you might want to use your own i/o routines. First, if you have something simple or secondly, if you have something special, you want to do your own i/o.

If you don't need all that flexibility, you are better off doing your own i/o. Here are two files that write a text screen to a disk file.

```
COPYSCRN EXE    10454   6-10-90   9:30a
INTSCRN  COM      445   6-12-90   7:40p
```

They both do the same thing except that the .COM file is a little more sophisticated. Notice the difference in size. Speed really doesn't play a part here because what they do is so simple that it takes just a second in any case. The program was so simple that it only took an hour or two to write, so I didn't lose any time by writing it in assembler.

The other case is when you have a specific idea of what the screen should look like. You want control of the whole screen all the time. This includes all word processors, databases, programming environments, etc. They all take charge of the screen because some DOS functions are too slow. If you remember from the ZOOM chapter, there is a radical difference between what you can do and what DOS can do. Even though these large programs are written in C, they all bypass the C i/o functions. That does not mean that they go down to the assembler level, however.

INTERRUPTS

You have done a few interrupts. They call the standard DOS or BIOS functions. Remember, they do this by going into low memory (the first 1k of memory) and getting the address of the subprogram that handles that particular interrupt. However, you do not need to call these interrupts from the assembler level. All modern compilers support interrupt calls in the language. If yours doesn't, you need a more recent compiler. Before going on with this chapter you need to read your compiler documentation about interrupts. TURBO Pascal has INTR, QuickBASIC and QuickC have INT86. Read the documentation now.

Have you read it? No cheating is allowed, because you won't understand the rest of this if you haven't read it.

Though technically C is a structure and Pascal is a record, they are actually arrays where each array element has a specific name. The interrupt routine reads all these values into the corresponding register, calls the interrupt, then reads the register values back into the array. Some languages have one array for the input and another for the output. Int 21h is special so QuickC has a special function called INTDOS. It is the same as using Int 21h.

The order of registers in the array is arbitrary and language dependent. For TurboPascal it is (AX, BX, CX, DX, BP, SI, DI, DS, ES, FLAGS). You enter values in the registers specified by the interrupt, and then call the interrupt. The routine does the rest.

```
INTR ( int_no: byte, var the_regs: Registers)
```

This will push the interrupt number, then the array address. On entry to the interrupt call and after initializing BP, we will have:

```
        int_no              bp + 6
        array_address       bp + 4
        old IP              bp + 2
bp -> old BP                bp
```

What follows is not the exact code, but is similar to what the Pascal routine does:

```
; - - - - - - - - - -
intr proc near

     push bp
     mov  bp, sp
     push ax   ; save all registers except SP, DS, SS, CS
     push bx
     push cx
     push dx
     push si
     push di
     push bp   ; this is OUR bp
     push es

     ; insert the interrupt number in the interrupt
     mov  al, [bp+6]  ; AL now contains the interrupt number
     lea  si, interrupt_spot  ; where the interrupt is
     mov  cs:[si+1], al       ; insert it in the interrupt

     ; change all the registers
     mov  si, [bp+4]     ; array address is DS:SI
     mov  ax, [si]
     mov  bx, [si+2]
     mov  cx, [si+4]
     mov  dx, [si+6]
     mov  bp, [si+8]
```

```
        mov  di, [si+12]
        mov  es, [si+16]

        ; special manipulation for DS and SI
        push ds             ; save ds
        push si             ; save si
        push ax             ; temp save of ax from array
        mov  ax, [si+14]        ; ds from array to ax
        mov  si, [si+10]        ; si from array to si
        mov  ds, ax             ; now move ax to ds
        pop  ax                 ; restore ax

        ; call the interrupt
interupt_spot:
        int  0          ; dummy number for the interrupt

        ; special needs for SI and DS
        ; our SI and DS are at the top of the stack
        ; save values of flags, si and ds from interrupt
        pushf           ; value from interrupt
        push si         ; value from interrupt
        push ds         ; value from interrupt
        add  sp, 6      ; get to our si and ds
        pop  si         ; our si
        pop  ds         ; our ds
        sub  sp, 10     ; sp is where it was a moment ago.
        mov  [si], ax   ; DS:SI points to array
        mov  [si+2], bx
        mov  [si+4], cx
        mov  [si+6], dx
        mov  [si+8], bp
        mov  [si+12], di
        mov  [si+16], es
        pop  [si+14]    ; ds from the interrupt
        pop  [si+10]    ; si from the interrupt
        pop  [si+18]    ; flags from the interrupt
        add  sp, 4      ; skip our DS and SI (already in regs)

        pop  es
        pop  bp         ; this is OUR bp
        pop  di
        pop  si
        pop  dx
        pop  cx
        pop  bx
        pop  ax

        mov  sp, bp
        pop  bp

        ret (4)         ; clear arguments off the stack
        ; - - - - - - - - - - - - - - - - - - - -
```

This should test your insight into using code. DS and SI are needed for moving data, so we use some kludges to get it to work.

There are two things here that you shouldn't normally do. First,

we are inserting the interrupt number directly in the machine code. Secondly, we are playing around with the value of SP. These are rare exceptions and shouldn't occur in your own code unless absolutely necessary.

The first thing you are going to say is, "Gosh, that's a lot of code for one interrupt." True, especially when the interrupt is interrupt 12h. Here's int 12h inside of our template file:

```
; - - - - - START CODE HERE
     int  12h       ; machine memory  (return in ax)
     call print unsigned
; - - - - - END CODE HERE
```

It finds out how much memory your computer has and returns the number of kbytes in AX. But how much extra time does using this Pascal interrupt routine take? About 700 clocks or about .0002 seconds (that's right, 2 ten thousandths) on the slowest machine. How many times will you call it during a program? Only one time. There is no point in going down to the assembler level to write a program that saves you .0002 seconds. In Pascal, you would write:

```
INTR ( $12 , the_regs) ;
```

and be done with it. No big loss of time and no trouble at all.

In fact, as far as I can see, there is no reason for doing any interrupts from the assembler level. You may want to do a whole subprogram that contains interrupts, but if you just need one or two interrupts, it is easier to work from inside the high-level language.

This includes the i/o we were talking about a minute ago. Yoy can write a screen program inside a high level language using arrays. Just think of a screen as a 80X25 array. If a two dimensional array is too slow you need to go to a one dimensional array. All interrupts that tell what kind of video card is in the computer, what mode the screen is in, etc. can be done from the high-level language. The most you need assembler for (depending on the language) is moving the text array into video memory. You want a bunch of help screens? Put all the help screens in a single file and use the interrupt for random access file read to read a screen when you need it.{2}

Anything else? Yes, we still have the need for speed. There are certain types of operations like block moves of data, word searches and sorting of arrays that are characterized by large amounts of data and/or large amounts of computation. If you think you see a way to use registers effectively for one of these

---

2. What you actually want to do is have the first block of data in the file tell you where each screen is and how long its data is. Then the first 2 bytes or words of the screen data should say the dimensions of the screen data (12 X 25, 17 X 3, etc.). This will allow you to store and use screens of any size.

things, you probably can beat a compiled version of the subprogram. Then the only question is whether or not it is worth the trouble.

We have used the words "fast" and "slow" ambiguously so far, but now it is time to quantify them. Before you get the numbers, you need to know one thing about memory. People always talk about the "data bus". What is it? It is a group of wires connecting the 80x86 chip to memory. The 8088 has 8 wires, the 8086, 80286 and 80386/SX have 16 wires, and the 80386 has 32 wires. That means that the 8088 can transfer 8 bits of information at one time, the 8086 et. al. can transfer 16 bits at a time and the 80386 can transfer 32 bits at a time. This means one byte, two byte and four byte transfers respectively. This also means that the memory bytes are ordered a little differently. You will never notice it externally, but here is the different internal ordering.

The 8088 has all bytes one after the other. All memory read/writes are done with the same 8 wires:

```
              8088
        MEMORY ADDRESSES

              00005
              00004
              00003
              00002
              00001
              00000
data lines   ||||||||   (8 bits)
```

(All our examples will use absolute memory locations starting at 00000). The chips with a 16 bit data bus have all the even locations on the first 8 wires and the odd locations on the other 8 wires. They come in pairs - first even then odd:

```
                   8086
             MEMORY ADDRESSES

              00006     00007
              00004     00005
              00002     00003
              00000     00001
data lines   ||||||||  ||||||||   (16 bits)
```

When one of these chips reads or writes, it can read/write either the left or the right byte or the whole word. What it cannot do is read the right byte from one pair along with the left byte from another pair. If you want to read the word at 00005:00006, the 8086 must:

1) read the 00005 byte.
2) read the 00006 byte.
3) join them together.

This takes longer than just a single word read.

The true 80386 has a 32 bit data bus. This allows it to read 4 bytes at a time, and its physical memory structure looks like this:

```
                          80386
                      MEMORY ADDRESSES

              00010     00011     00012     00013
              0000C     0000D     0000E     0000F
              00008     00009     0000A     0000B
              00004     00005     00006     00007
              00000     00001     00002     00003
data lines   ||||||||  ||||||||  ||||||||  ||||||||  (32 bits)
```

Instead of memory pairs, we now have memory quadruplets. As long as a word is totally inside of one quadruplet, the read/write time will be unaffected. If the read/write crosses the boundary (as we did above), the read/write time will be affected in the same way. The 80386 can also read 4 byte data quickly as long as the total data is inside of one memory quadruplet.

In the 8086 family, data can always be read across these boundaries but it takes more time. (On the IBM 370, on the other hand, there are instructions that REQUIRE that data be aligned along 32 bit boundaries).

This means you should order your data in the following way in the data segment:

```
    QWORD DATA
    DWORD DATA
    TBYTE DATA      ; this is for the 8087
    WORD DATA
    BYTE DATA       ; all strings, etc.
```

This insures that any read/write for that type of data will always be as fast as possible. If the segment definition has no alignment type, it will start on a paragraph boundary - i.e. every 16 bytes, and will work with anything. {3}

In addition, if you ever subtract a number from SP to provide for a temporary data area, it should always be an even number. If SP is at an odd address instead of an even address, it takes longer for PUSHes and POPs. Also, when you define the size of the stack segment, it should be an even number of bytes.

Having said that, it is now time for you to see the speeds of

---

3. The alignment type is a word after the word SEGMENT which says how the segment should be aligned. The following:

```
        DATASTUFF SEGMENT BYTE PUBLIC 'DATA'
```

says the segment can be aligned at any byte. The allowable forms are BYTE, WORD, DWORD, PARA, PAGE (256 bytes). If there is no explicit type, the default is PARA.

instructions. Read the introduction to APPENDIX III, then glance at the times to get the general idea of how fast times are. Come back to this chapter when you are comfortable with what the times look like.

Have you read APPENDIX III? If not, do it before going on.

The compiled languages all have one thing in common. They tell you that if you are writing a subroutine, you need to return from the subroutine with DS, BP, SS, and SP unchanged. They don't say a thing about any of the other registers. One thing this tells us is that they are doing everything from memory locations, not register locations. If you have taken a good look at the execution times, you will have noticed the phenomenal difference in time between a "memory, register" addition and a "register, register" addition.

Now, if all you are going to do is move a number to a register, add it, and move it out again, a compiler can do it as fast as you can. But, if you run into a situation where you can use three or four registers at the same time, you can cut the execution time drastically. Compilers really can't use registers as efficiently as we can (yet). This is an ideal spot for using assembly language.

The old adage that 10% of the code uses 90% of the computer time is appropriate here. You now know about assembler language, and you know what you want to do with it, so go out and enjoy. But before you do, try to slog your way through the next chapter on "simplified" segment definitions and linking to high level languages.

CHAPTER 26 - SIMPLIFYING THE TEMPLATE

By the time you have finished this chapter your assembler files will look cleaner. Unfortunately there is some heavy sledding before we get there.

EXITING A PROGRAM

Till now, we have exited most programs with CTRL-C; otherwise the program has done a return. A return to what? It has been returning to a section of code that does INT 20h, one of the ways of quitting a program when everything is in order. Notice the "everything is in order" in the last sentence. What happens if you have 2 files open, you are off in some subroutine, and you have things so hopelessly confused that you might as well give up? Can you call INT 20h? The answer is no for two reasons. First, you need CS to point to the PSP (program segment prefix). and you don't know where the PSP is. Secondly, you need to close files. Now, it is possible to make some code to do this, but why bother. We have a special interrupt for this:

```
INT 21h function 4Ch
AH = 4Ch
AL = return code
```

This will close all files, get you out of the program, and give a return code that is usable by the calling program. Here's a small program. Use template.asm and call this TEST4CH.ASM.

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - -
CODESTUFF    SEGMENT   PUBLIC  'CODE'

   ASSUME cs:CODESTUFF, ds:DATASTUFF
   EXTRN   get_unsigned_byte:NEAR

main   proc far

start:
        mov  ax, DATASTUFF    ; load ds
        mov  ds,ax

        call get_unsigned_byte    ; value is in al
        mov  ah, 4Ch              ; int 21h, function  4Ch
        int  21h

main   endp
CODESTUFF    ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

We have revised the CODESTUFF segment so there is only one EXTRN statement and the beginning code:

```
    push ds
    sub  ax, ax
    push ax
```

is gone. Since we will never again do a return to the PSP, there is no need for this code anymore.

The program gets a single byte to use as the exit code and then exits using int 21h function 4Ch. Get this assembled and linked. It should ask for a number and then exit. But where is that number? It is available through the operating system. Make the following batch file. It runs TEST4CH.EXE and then looks at the error code. Unfortunately ERRORLEVEL is not available as an exact number to a batch file, so we are checking to see if the return code was above a certain level.

```
    -----------------  DO4CH.BAT -----------------------
    test4ch
    ECHO OFF
    IF ERRORLEVEL   1  ECHO The return code was over 0
    IF ERRORLEVEL  51  ECHO The return code was over 50
    IF ERRORLEVEL 101  ECHO The return code was over 100
    IF ERRORLEVEL 151  ECHO The return code was over 150
    IF ERRORLEVEL 201  ECHO The return code was over 200
    ECHO ON
    ----------------------------------------------------
```

Here's one run of the batch file:

```
    >do4ch
    >int4ch

    The PC Assembler Helper   Version 1.01
    Copyright (C) 1989  Chuck Nelson   All rights reserved.
    Enter a number from 0 to 255  172

    >ECHO OFF
    The return code was over 0
    The return code was over 50
    The return code was over 100
    The return code was over 150
```

This is what happens to DO4CH.BAT with a return code of 172.

From now on, always use INT 21h function 4Ch to exit.

SEGMENTS

Our major simplification has to do with segment names. Before we go on with segment simplification, here are the rules the linker uses. If you don't remember them, you should review Chapter 10 before going on.

During the link process, the linker will combine any segments which:

1) have the same name.
2) are declared PUBLIC.
3) have the same class name (type).

The linker processes object modules from left to right on the command line. The classes will be ordered in the ordering in which they were encountered (including the empty class type). Within each class, the segments will be ordered in the ordering in which they were encountered.

If we have all these rules, how do high-level languages manage to combine their data and code correctly? The answer is that they use standardized segment definitions. Here are the basic ones for our data:

```
;-------------------------------------------------
_DATA  SEGMENT WORD PUBLIC 'DATA'
_DATA  ENDS
;-------------------------------------------------
;-------------------------------------------------
CONST  SEGMENT WORD PUBLIC 'CONST'
CONST  ENDS
;-------------------------------------------------
;-------------------------------------------------
_BSS  SEGMENT WORD PUBLIC 'BSS'
_BSS  ENDS
;-------------------------------------------------
;-------------------------------------------------
STACK  SEGMENT PARA STACK 'STACK'
STACK  ENDS
;-------------------------------------------------
```

If all the code will fit in one segment we can use a single segment name:

```
;-------------------------------------------------
_TEXT  SEGMENT WORD PUBLIC 'CODE'
_TEXT  ENDS
;-------------------------------------------------
```

otherwise we can make independent segments, each with an independant name:

```
;-------------------------------------------------
name_TEXT  SEGMENT WORD PUBLIC 'CODE'
name_TEXT  ENDS
;-------------------------------------------------
```

where the "name" can be anything, but the "_TEXT" remains invariable. Any subroutine calls within the segment can be NEAR, while any calls to a different segment should be FAR.

The "WORD" in these definitions says that when the linker combines segments into a larger segment, each subsegment must start at an even address (a word boundary). This has to do with the speed of word fetches from memory that we discussed in the last chapter. "WORD" is fine for 16 bit data busses, but for a

80386 you actually want "DWORD" so things are correctly aligned with a 32 bit data bus. "PARA" means paragraph and that means aligned with a segment starting address (every 16 bytes). Everything will work with "PARA".

For reasons of convenience, compilers put different types of data in different segments. For you, there is no reason to use more than one segment, and that is:

```
;--------------------------------
_DATA  SEGMENT WORD PUBLIC 'DATA'
_DATA  ENDS
;--------------------------------
```

Compilers use these different segments because they can. If they were constrained to use only one segment name, they could do it with no problem. What is in these different segments?

```
_DATA     standard initialized data
CONST     data constants
_BSS      uninitialized static data
STACK     room for the SS:SP stack
```

So what do these things mean?

_DATA

The _DATA segment stores all initialized data which exists from the time the program starts till the time that the program ends.

In C:

```
static int     x = 5;
```

In Pascal:

```
const
     my_salary  : real = 52.77
```

These variables have a specific value at the start of the program, even before the first instruction is executed. This value may change during the program. The variable exists during the whole program.

_BSS

The _BSS segment stores all uninitialized data which exists from the time the program starts till the time that the program ends.

In C:

```
static int     x ;
```

In Pascal, any variable declared outside a procedure but without an initial value will be in _BSS. In compiled BASIC, everything except dynamic arrays is in the _BSS. These variables have an indeterminate value at the start of the program and exist during the whole program.

CONST

CONST takes all constants which are longer than 2 bytes. If the compiler is on its toes, anything one or two bytes long will be coded into the machine instructions since this is much faster. What is a constant? It is anything that has a value but doesn't have a variable name:

```
value = 275.29 ;
printf ( "Mr. Yellow: 'Read my lips - no new taxis!'\n");
result =  value / 27.619 ;
file_ptr = fopen ( "stuff.doc", "r+") ;
```

All the numbers and all the text strings need to be stored somewhere. They are stored in the CONST segment and given an internal name by the compiler so they can be used at the appropriate location. They are not available in other parts of the program.{1} These constants are sometimes called literals.

STACK

BASIC does not use the stack in the same way as Pascal and C. In BASIC it is used only for passing variables between subroutines. In C and Pascal, most variables are temporary. They come into existance at the beginning of the subroutine and they disappear upon leaving the subroutine. When you call the subroutine again, the values these variables have are indeterminate. These variables all exist on the stack relative to BP, the base pointer. This is why you can have recursion in C and Pascal but not in BASIC.

As I said, you don't need to put your different types of data in different segments. It can all go into _DATA.

GROUPS

We now come to the bizarre. You will notice that when the linker links all these object modules together, it will have four distinct segments with each segment having a distinct class name. We will get:

```
_DATA     'DATA'
CONST     'CONST'
_BSS      'BSS'
STACK     'STACK'
```

The problem here is that we want to set DS at the beginning of

---

1. There is an exception. Some compilers check to make sure that there are no duplicates of the constant. These compilers give all duplicates the same address so there is only one copy of any one constant such as 0, 1, etc.

the program so that it will reference all the data. How are we going to do this? The warped minds of electrical engineers and computer scientists spent hours and hours trying to find the most obscure way possible to unify data addressing and they came up with GROUPS.

You can tell the linker that you want data from distinct segments to be referenced by the offset from the beginning of the lowest segment in memory that belongs to the group. Read this about five or ten times to get the hang of it. You tell the linker that a bunch of different segments belong to a group. It will find the segment which is lowest in memory and then whenever you ask for the GROUP offset, the linker will calculate the offset from the beginning of this first segment.

The way you define a group is with a name, the word "GROUP", and then a list of those segments in the file which belong to the group:

```
DGROUP  GROUP _DATA, CONST, _BSS, STACK
```

Note that it is the segment names, not the class names. DGROUP is the standard name for the data group. If the assembler gives the linker the correct information, the linker will adjust all offsets relative to the beginning of the group. The only limit on a group is that the distance from the first byte of the group to the last byte of the group must be 65535 bytes or less. This is because all the group segments must reside in one physical segment in memory.

It is not even necessary for all the segments in a block of memory to belong to the group. Consider the following ordering of segments in memory.

```
_DATA
DATASTUFF
CONST
CODESTUFF
_BSS
EVENMORESTUFF
STACK
```

As long as the distance from one end of _DATA to the other end of STACK is 65535 bytes or less, the linker will adjust the offsets in _DATA, CONST, _BSS and STACK relative to the start of DGROUP and the linker will adjust the offsets of DATASTUFF, CODESTUFF and EVENMORESTUFF relative to their respective segment starting addresses. I didn't say that this was good programming, I only said that it was possible.

Thoroughly confused? You're not alone. Just remember, in all compiled languages, we are going to combine these four types of segments into a single group where offsets are relative to the very beginning of the data.

Before getting you even more confused, let's take a look at what we have so far. Make sure you actually do all of the following

examples. Use template.asm and at the very top, put in the following segments:

```
; - - - - - - - - - -
SEG1  SEGMENT 'STUFF'
          db  100  dup (?)
seg1_data db  ?
          db  899 dup (?)
SEG1 ENDS
; - - - - - - - - - -
SEG3  SEGMENT 'STUFF'
          db  300  dup (?)
seg3_data db  ?
          db  699 dup (?)
SEG3 ENDS
; - - - - - - - - - -
SEG5  SEGMENT 'STUFF'
          db  500  dup (?)
seg5_data db  ?
          db  499 dup (?)
SEG5 ENDS
; - - - - - - - - - -
```

Call this program QGROUP1.ASM. These segments are 1000 bytes long, and the data names are 100, 300 and 500 bytes into their respective segments. Because these segments will be paragraph aligned, the second and third segments will start 1008 bytes (16 X 63) after the proceeding one. You need to tell the assembler that these are in a group and give the proper ASSUME statement. We'll call this QGROUP:

```
QGROUP  GROUP  SEG1, SEG3, SEG5
ASSUME cs:CODESTUFF, ds:DATASTUFF, ds:QGROUP
```

Here's some code:

```
; + + + + + + + + + + + + START CODE BELOW THIS LINE
    lea     ax, seg1_data
    call    print_unsigned
    lea     ax, seg3_data
    call    print_unsigned
    lea     ax, seg5_data
    call    print_unsigned
; + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

As you can see, all we are doing is putting the addresses into AX and then printing them as unsigned numbers. Here's the output:

```
00100
01308
02516
```

Remember, each segment is starting 1008 bytes after the start of the previous one. Here's the same program with a few extra segments thrown in. Call it QGROUP2.ASM.:

```
; - - - - - - - - - -
```

```
SEG1  SEGMENT  'STUFF'
          db  100  dup (?)
seg1_data db  ?
          db  899 dup (?)
SEG1 ENDS
; - - - - - - - - - -
SEG2  SEGMENT  'STUFF'
          db  200  dup (?)
seg2_data db  ?
          db  799 dup (?)
SEG2 ENDS
; - - - - - - - - - -
SEG3  SEGMENT  'STUFF'
          db  300  dup (?)
seg3_data db  ?
          db  699 dup (?)
SEG3 ENDS
; - - - - - - - - - -
SEG4  SEGMENT  'STUFF'
          db  400  dup (?)
seg4_data db  ?
          db  599 dup (?)
SEG4 ENDS
; - - - - - - - - - -
SEG5  SEGMENT  'STUFF'
          db  500  dup (?)
seg5_data db  ?
          db  499 dup (?)
SEG5 ENDS
; - - - - - - - - - -
```

This is almost the same thing but we have added two more segments. We are NOT going to join these two segments into the group. Here's the GROUP and ASSUME statements:

```
QGROUP  GROUP  SEG1, SEG3, SEG5
ASSUME  ds:SEG2, ds:SEG4
ASSUME cs:CODESTUFF, ds:DATASTUFF, ds:QGROUP
```

Make sure the ASSUME statements are in that order or things may get confused. We also add some code:

```
; + + + + + + + + + + + + START CODE BELOW THIS LINE
    lea     ax, seg1_data
    call    print_unsigned
    lea     ax, seg2_data
    call    print_unsigned
    lea     ax, seg3_data
    call    print_unsigned
    lea     ax, seg4_data
    call    print_unsigned
    lea     ax, seg5_data
    call    print_unsigned
; + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

This shows the addresses of all five variables. Here's the new output:

```
00100
00200
02316
00400
04532
```

As you can see, the GROUPed segments have their offsets relative to the beginning of the group while the others have their offsets relative to the beginning of the segment.

Make a copy of QGROUP1.ASM and call it QGROUP?.ASM. Leave the segment definitions, group definitions, ASSUME statements and code the same, but add six more lines of code at the end:

```
        ; compare the offsets
        mov     ax, offset seg1_data
        call    print_unsigned
        mov     ax, offset seg3_data
        call    print_unsigned
        mov     ax, offset seg5_data
        call    print_unsigned
    ; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

After the three LEAs we now do 3 OFFSETS. Assemble and link this. Here's the output:

```
00100
01308
02516
00100
00300
00500
```

Wait a minute! Those last three numbers should be the same as the first three numbers. That's right, folks. This is a known error in the MASM assembler. In fact the Turbo Assembler copies this mistake when it is in "MASM" mode but does it right when it is in "IDEAL" mode. A86 does it right all the time. Here is the output from the same source file when assembled by A86:

```
00100
01308
02516
00100
01308
02516
```

You have told the assembler to calculate all offsets relative to the beginning of the group and MASM is ignoring you every time you use the OFFSET operator. The code fix for this is to use an override when you use OFFSET:

```
         ; compare the offsets
        mov     ax, offset QGROUP:seg1_data
        call    print_unsigned
        mov     ax, offset QGROUP:seg3_data
```

```
        call    print_unsigned
        mov     ax, offset QGROUP:seg5_data
        call    print_unsigned
    ; + + + + + + + + + + + + + + + + END CODE ABOVE THIS LINE
```

Better yet, use LEA whenever possible. If you do use OFFSET with groups, you need to go through the text file with a word search to make sure that all OFFSETs have a group override. This is a subtle error and it is very hard to find if you are not looking for it.

This system is designed so that we can have 64k of data and stack, all of which is addressable with DS without changing DS's value. What happens if you have more data than that? One thing for sure is that you don't have more than 64k of individually named variables. Either that or you have some huge calluses on your typing fingers.

What you do have is arrays. If you run into space problems, you move the least used or the biggest arrays into their own segments. You can have one segment per array if you want. The standardized high-level language names for these segments is:

```
    ; - - - - - - - - - - - - - - - - - - - -
    FAR_DATA  SEGMENT  PARA  'FAR_DATA'
    FAR_DATA  ENDP
    ; - - - - - - - - - - - - - - - - - - - -
    FAR_BSS   SEGMENT  PARA  'FAR_BSS'
    FAR_BSS   ENDP
    ; - - - - - - - - - - - - - - - - - - - -
```

Once again, the '_DATA' is for initialized data while the '_BSS' is for uninitialized data. Use only the 'FAR_DATA' kind.{2} You will notice that these segments are NOT PUBLIC. Although an assembler will unify all segments with the same definition that are in the same file, the linker will not unify segments from different files which are not PUBLIC. If we create 4 different .ASM files, each with one segment:

```
    ; FARDATA1.ASM
    PUBLIC data1
    ; - - - - - - - - - - - - - - - - - - - -
    FAR_DATA  SEGMENT  PARA  'FAR_DATA'
    data1     db   1A67h dup (0)
    FAR_DATA  ENDS
    ; - - - - - - - - - - - - - - - - - - - -

    ; FARDATA2.ASM
    PUBLIC data2
```

---

2. A high-level language has the right to set all the data of a 'BSS' segment to zero as part of its startup routine. Whether it does so or not depends on what it has told the linker. If you put initialized data into either a '_BSS' or a 'FAR_BSS' segment, it might easily wind up zero after startup.

```
    ; - - - - - - - - - - - - - - - - - - - - - -
    FAR_DATA  SEGMENT  PARA  'FAR_DATA'
    data2     db   0D4A8h dup (0)
    FAR_DATA  ENDS
    ; - - - - - - - - - - - - - - - - - - - - - -

    FARDATA3.ASM
    PUBLIC data3
    ; - - - - - - - - - - - - - - - - - - - - - -
    FAR_DATA  SEGMENT  PARA  'FAR_DATA'
    data3     db   200h dup (0)
    FAR_DATA  ENDS
    ; - - - - - - - - - - - - - - - - - - - - - -

    FARDATA4.ASM
    PUBLIC  data4
    ; - - - - - - - - - - - - - - - - - - - - - -
    FAR_DATA  SEGMENT PARA  'FAR_DATA'
    data4     db   8716h dup (0)
    FAR_DATA  ENDS
    ; - - - - - - - - - - - - - - - - - - - - - -
```

and link these with TEMPLATE.OBJ and ASMHELP, we will get the following .MAP file:

```
     Start  Stop   Length Name                   Class
     00000H 01A66H 01A67H FAR_DATA              FAR_DATA
     01A70H 0EF17H 0D4A8H FAR_DATA              FAR_DATA
     0EF20H 0F11FH 00200H FAR_DATA              FAR_DATA
     0F120H 17835H 08716H FAR_DATA              FAR_DATA
     17840H 1823FH 00A00H STACKSEG              STACK
     18240H 1875DH 0051EH DATASTUFF             DATA
     18760H 1A02FH 018D0H CODESTUFF             CODE

    Program entry point at 1876:0000
```

The numbers in the segment definitions were in hex so you could read the .MAP file more easily. We have created four different FAR_DATAs - one for each variable.

The idea here is to leave DS alone if possible and use ES:SI or ES:DI for your manipulation of the array.

```
    mov  ax, seg data1
    mov  es, ax
    mov  si, offset data1
```

Of course, if you using two different FAR_DATA arrays from two different segments at the same time, you will probably need to use DS temporarily. This is the kind of thing you need to plan before you start a program which contains large arrays.

You have now seen all possible segments for any Microsoft language and for Turbo C.{3} These are:

```
_DATA  SEGMENT WORD PUBLIC 'DATA'
CONST  SEGMENT WORD PUBLIC 'CONST'
_BSS   SEGMENT WORD PUBLIC 'BSS'
STACK  SEGMENT PARA STACK 'STACK'
_TEXT  SEGMENT WORD PUBLIC 'CODE'

name_TEXT SEGMENT WORD PUBLIC 'CODE'
FAR_DATA  SEGMENT PARA  'FAR_DATA'
FAR_BSS   SEGMENT PARA  'FAR_BSS'

DGROUP  GROUP  _DATA, CONST, _BSS, STACK
```

We have another problem on our road to simplification. We want DS to have the address of the start of DGROUP. How do we do it? Well, before we had:

```
mov  ax, DATASTUFF
mov  ds, ax
```

DATASTUFF was a segment. We do the same thing for groups:

```
mov  ax, DGROUP
mov  ds, ax
```

We use a group name instead of a segment name. This means that our ultimate code segment will look like this

```
; - - - - - - - - - -
_TEXT  SEGMENT WORD PUBLIC 'CODE'

     DGROUP  GROUP  _DATA, CONST, _BSS, STACK
     ASSUME cs:_TEXT, ds:DGROUP

start:
     mov  ax, DGROUP
     mov  ds, ax

     ; - - - - - - - - - - -
     ; the program goes here
     ; - - - - - - - - - - -

     mov  ah, 4Ch
```

---

3. If you are using Turbo PASCAL, then there are only two segments possible. They are:

```
DATA  SEGMENT WORD PUBLIC
CODE  SEGMENT BYTE PUBLIC
```

There is no class name. You can substitute DSEG for DATA and CSEG for CODE if you want. Turbo Pascal has no DGROUP.

```
        mov  al, ?          ; replace ? with error code
        int  21h

    _TEXT  ENDS
    ; - - - - - - - - - - -
```

Say, if all this stuff is standardized text, why are we forced to type all this drivel over and over again. The answer is that we aren't. All the segment information has a shorthand. Here's how it works. Every shorthand symbol starts with a dot. The assembler will then generate the desired text.{4} This is from MASM 5.0 on, so if you have an earlier assembler you'll have to write the full text.

To start out, use the two starting directives DOSSEG (with no dot) and .MODEL. MODEL will be explained later.{5}

```
        DOSSEG
        .MODEL Medium
```

For now, 'medium' is what we want.

From that point, if you want a data segment, you just write .DATA, if you want code, you write .CODE. Every time that the assembler sees a segment directive it will close any segment that is open and start the segment indicated by the directive. (You can always reopen a segment). Here is what replaces the directives:

```
    DIRECTIVE               REPLACEMENT TEXT

    .DATA               _DATA  SEGMENT WORD PUBLIC 'DATA'
    .CONST              CONST  SEGMENT WORD PUBLIC 'CONST'
    .DATA?              _BSS   SEGMENT WORD PUBLIC 'BSS'
    .STACK [size]       STACK  SEGMENT PARA STACK 'STACK'
    .CODE               _TEXT  SEGMENT WORD PUBLIC 'CODE'

    .CODE [name]        name_TEXT SEGMENT WORD PUBLIC 'CODE'
    .FARDATA [name]     FAR_DATA  SEGMENT PARA  'FAR_DATA'
    .FARDATA? [name]    FAR_BSS   SEGMENT PARA  'FAR_BSS'
```

The [name] in brackets will be explained in a minute. The [size] after the stack declaration allows you to customize the size of the stack. Without any size, the declaration

```
    .STACK
```

will allocate 1k of memory for the stack. A size allocates a

---

4. It really generates no text. It is just that the assembler will generate the same machine code as if that text had been generated.

5. DOSSEG tells the assembler to tell the linker that the .EXE file should have the standard segment order. It is not necessary but it doesn't hurt.

specific number of bytes:

```
    .STACK 2000h
```

You can make it anything you want, but make sure it is an even number and remember that the limit for all four parts of DGROUP is 64k.

To see how the names work, we need some text files. Here is a complete main file:

```
    ; FARDATA.ASM   - driver module
    DOSSEG
    .MODEL  medium
    EXTRN  data2_routine:FAR, data3_routine:FAR
    .STACK  200h
    .FARDATA
         data1     db   0100h dup (0)
    .CODE
    main:
         mov  ax, DGROUP
         mov  ds, ax
         call data2_routine
         call data3_routine
         mov  ax, 4C00h
         int 21h
    END  main
```

It has some data and some code though it doesn't really do anything. We will use this along with two other files for the examples. Here is FARDATA2:

```
    ; FARDATA2.ASM
    DOSSEG
    .MODEL  medium
    PUBLIC data2_routine
    .FARDATA
         data2     db   0200h dup (0)
    .CODE
    data2_routine  proc
         ret
    data2_routine endp
    END
```

Notice that data2_routine doesn't have a FAR or NEAR. That's being taken care of by the memory model. Data2_routine's type does need to be declared EXTRN in the main module. The third routine has similar code. Here is the .MAP file when they are combined:

```
     Start  Stop   Length Name                   Class
     00000H 00013H 00014H FARDATA_TEXT          CODE
     00014H 00014H 00001H FARDATA2_TEXT         CODE
     00016H 00016H 00001H FARDATA3_TEXT         CODE
     00020H 0011FH 00100H FAR_DATA              FAR_DATA
     00120H 0031FH 00200H FAR_DATA              FAR_DATA
```

```
00320H 0061FH 00300H FAR_DATA                FAR_DATA
00620H 00620H 00000H _DATA                   DATA
00620H 0081FH 00200H STACK                   STACK
```

You can see the FAR_DATAs there, but where did the FARDATA3_TXT come from? The assembler decided that we wanted independent code segments and gave each one the name of the assembler file it came from. Since all the object files in a program must have unique names, these segment names should also be unique. If we change the .MODEL from MEDIUM to COMPACT without touching anything else, then we get:

```
Start  Stop   Length Name                   Class
00000H 00022H 00023H _TEXT                   CODE
00030H 0012FH 00100H FAR_DATA                FAR_DATA
00130H 0032FH 00200H FAR_DATA                FAR_DATA
00330H 0062FH 00300H FAR_DATA                FAR_DATA
00630H 00630H 00000H _DATA                   DATA
00630H 0082FH 00200H STACK                   STACK
```

If we now put a name after the .FARDATA directive, it will give the segment a unique name. Putting:

```
.FARDATA  jake_the_snake
```

in FARDATA2.ASM, along with name changes in the other modules results in the following .MAP file:

```
Start  Stop   Length Name                   Class
00000H 00022H 00023H _TEXT                   CODE
00030H 0012FH 00100H HACKSAW                 FAR_DATA
00130H 0032FH 00200H JAKE_THE_SNAKE          FAR_DATA
00330H 0062FH 00300H HULKSTER                FAR_DATA
00630H 00630H 00000H _DATA                   DATA
00630H 0082FH 00200H STACK                   STACK
```

We are doing a number of interrelated things here, so let's try to unify what is going on. You have seen both NEAR and FAR routines in the Tutor. A NEAR routine alters IP and restores IP on the return. A FAR routine alters both CS and IP and restores them on the return.

When we passed addresses of data, we have almost always passed just the offset of the data. That is because the data has almost always been in the DATASTUFF SEGMENT, and the value of DS has been known. In Chapter 19 we did "move_pascal_string" which was a subroutine where we passed both the segment and offset of the data. These are our two choices for passing addresses:

```
OFFSET           1 word
SEGMENT:OFFSET   2 words
```

This gives us four basic possiblilities for program structure:

SUBROUTINE CALL	DATA ADDRESSES PASSED AS
NEAR	OFFSET
FAR	OFFSET
NEAR	SEGMENT:OFFSET
FAR	SEGMENT:OFFSET

Each of these structural possibilities has a name called a MODEL name. They are:

SUBROUTINE CALL	ADDRESSES PASSED AS	MODEL NAME
NEAR	OFFSET	SMALL
FAR	OFFSET	MEDIUM
NEAR	SEGMENT:OFFSET	COMPACT
FAR	SEGMENT:OFFSET	LARGE

You tell the assembler which model you are working with by using the .MODEL directive:

```
.MODEL  medium
```

The assembler will then make either NEAR or FAR the default type. This can be overridden if you have explicitly given a NEAR or FAR:

```
my_proc  procedure
```

will generate the correct subroutine calls and returns for that model, while:

```
my_proc1  procedure  near
my_proc2  procedure  far
```

will remain unaltered.

At the assembler level, you need to code the address passing yourself, but if you have a MODEL and you are connected to a high-level language (with the same .MODEL type), the high-level language will pass all addresses as stated above.

The advantage of this system is that using the .MODEL directive and appropriate EQU statements and MACROS (which we have not covered), it is possible to write a single subroutine which can then be assembled in all four model configurations. Coding this is non-trivial, but when you have done more programming you will see how to deal with the stack using EQUs and MACROs.

For now, you want to stay with data addresses which are passed by offset only. This is much easier. These are the SMALL and MEDIUM models. Whether you choose NEAR or FAR procedures doesn't affect much except where parameters are on the stack (because of that extra CS).

Are these .MODELS important? They are nice, but not particularly vital. What happens is that in a manual you see a sample program like this:

```
DOSSEG
.MODEL medium
.STACK
.DATA
variable1 dw 25
.CODE
sample proc
mov  ax, variable1
ret
 sample  endp
ENDS
END
```

and you start comparing the size of this to the size it would be if you used the standard segment definitions. I have news for you. This is not a legitimate program. A legitimate program is a page or two long.{6} Also, at least to my way of thinking, you want visual separation between segments. The above is a disordered presentation of segments. We want order in our programs and the segment headers provide a visual structure. In the text file for ASMHELP, (which is about 3600 lines long), the SEGMENT declarations occupy about 20 lines. This is about 0.5% of the total length of the file.

If you are going to assemble a file in multiple models, then it is worthwhile to use the .MODEL directives, otherwise it is optional depending more on your concept of what looks clear than any major difference.

6. Perhaps you want to scan \COMMENTS\MISHMASH.DOC which contains some real subroutines. They are all long.

SUMMARY

To exit a program, use INT 21h Function 4Ch

```
    mov   ah, 4Ch         ; exit program
    mov   al, ?           ; replace ? with error code
    int   21h
```

A GROUP is a group of segments whose data will be referenced by the offset from the beginning of the group. You declare a group with:

```
    DGROUP  GROUP  _DATA, CONST, _BSS, STACK
```

MASM calculates OFFSETS incorectly with groups, so you should either use LEA or the DGROUP override:

```
    lea  ax, variable1
    mov  ax, offset DGROUP:variable1
```

To get the address of DGROUP in DS you need:

```
    ASSUME ds:DGROUP
```

and:

```
    mov  ax, DGROUP
    mov  ds, ax
```

The standardized segment definitions, along with their simplified directives are:

```
    DIRECTIVE               REPLACEMENT TEXT

    .DATA               _DATA  SEGMENT WORD PUBLIC 'DATA'
    .CONST              CONST  SEGMENT WORD PUBLIC 'CONST'
    .DATA?              _BSS   SEGMENT WORD PUBLIC 'BSS'
    .STACK [size]       STACK  SEGMENT PARA STACK 'STACK'
    .CODE               _TEXT  SEGMENT WORD PUBLIC 'CODE'

    .CODE [name]        name_TEXT SEGMENT WORD PUBLIC 'CODE'
    .FARDATA [name]     FAR_DATA  SEGMENT PARA  'FAR_DATA'
    .FARDATA? [name]    FAR_BSS   SEGMENT PARA  'FAR_BSS'
```

In addition you have the different model names:

```
    SUBROUTINE CALL      ADDRESSES PASSED AS        .MODEL NAME

         NEAR               OFFSET                   SMALL
         FAR                OFFSET                   MEDIUM
         NEAR               SEGMENT:OFFSET           COMPACT
         FAR                SEGMENT:OFFSET           LARGE
```

THE PC ASSEMBLER HELPER

"The PC Assembler Helper" is an object module (ASMHELP.OBJ) which is designed as a companion for "The PC Assembler Tutor". It can also be used as an aid in the development of assembler programs and subprograms. It allows for input to and output from the assembler level, as well as displaying the data in all the 8086 registers. Part 1 will give a discription of all callable subroutines and part 2 will explain how to correctly link a program to ASMHELP.OBJ.

THE SUBROUTINES

There are a number of routines for displaying data and for inputting data. They all follow a standard format.

(1) All subroutines which display output on the monitor at the current cursor position start with the word "print_". On the screen, all hex output is followed by an 'H'. All signed output has a + or a -. Unsigned output has no sign. All binary output is distinguishable because it is either 8 digits or 16 digits long. ASCII output is followed by a single asterisk '*' under normal conditions. If one or more of the ASCII characters cannot be printed (i.e. if it is less than 33d or it is 127d or 255d){1} then it will be displayed as a two digit hex number instead of a single character. A double asterisk '**' is then used to signal the presence of a hex number in an ASCII format. The only time there might be confusion is if one of the characters is also an asterisk.

(2) All subroutines which get input from the keyboard start with the word "get_". They all display prompts to tell you what kind of input is desired.

(3) One byte input or output is passed through register AL. One word (two byte) input or output is passed through register AX. Any input or output that is longer than two bytes is passed by reference. The offset address of the data is put into AX before calling the subroutine.

Each subroutine returns with all 8086 registers (including the flags register) unchanged from when the subroutine was called. As an example, if "variable3" is the name of a variable (in memory) which we want to print as a signed number, the proper way to set up for the calls is:

---

1 In this chapter, as in all others, 'd' stands for decimal, 'h' for hex.

```
ONE BYTE:
    mov  al, variable3
    call print_signed_byte

ONE WORD:
    mov  ax, variable3
    call print_signed

FOUR BYTES:
    lea  ax, variable3
    call print_signed_4byte

EIGHT BYTES:
    lea  ax, variable3
    call print_signed_8byte
```

For a byte or a word we use the data itself, while for data larger than a word we pass the data by reference. The same applies for getting input. Here are the subroutines:

get_num
: Enters a number 5 digits or less, either signed or unsigned. It does no checking to see if the number is too positive or too negative. Returns a two byte value in AX.

print_num
: Prints a word value (from AX) in its signed, unsigned, hex, ASCII, and binary representation.

All the other input routines do strict error checking. If an illegal character is entered or if the input is out of range, the subroutine will ask for new input until it receives valid input.

get_string
: Enters a string of 79 characters or less, and puts a 00h at the end of the string (a C string with a maximum total length of 80 bytes). The address of the string must be in AX before making the call.

print_string
: Displays a 00h terminated string (a C string) on the monitor. The address of the first byte of the string must be in AX before making the call.

get_ascii_byte
: Enters a single ascii character and returns it in AL.

print_ascii_byte
: Displays one byte (from AL) as an ascii character.

get_ascii
: Enters one or two characters and returns it (them) in AX.

print_ascii
    Displays the value in AX as two characters.

get_bcd
    Enters a one to eighteen digit signed number as a 10 byte bcd number. Commas are allowed. The address of the bcd variable must be in AX before calling the routine.

print_bcd
    Displays a 10 byte bcd number as a one to eighteen digit signed number with commas. The address of the bcd number must be in AX before calling the subroutine.

get_binary_byte
    Enters a one to eight digit binary number and returns it in AL.

print_binary_byte
    Displays a one byte number (from AL) as an eight digit binary number.

get_binary
    Enters a one to sixteen digit binary number and returns it in AX.

print_binary
    Displays a one word number (from AX) as a sixteen digit binary number.

get_hex_byte
    Enters a one or two digit hex number and returns it in AL.

print_hex_byte
    Displays a one byte number (in AL) as two hex digits.

get_hex
    Enters a one to four digit hex number and returns it in AX.

print_hex
    Displays a one word number (from AX) as a four digit hex number.

get_signed_byte
    Enters a one byte signed number (-128 to +127) and returns it in AL.

print signed_byte
    Displays a one byte number (from AL) as a signed number

(-128 to +127).

get_signed
Enters a one word signed number (-32768 to +32767) and returns it in AX.

print_signed
Displays a one word number (from AX) as a signed number (-32768 to +32767).

get_signed_4byte
Enters a 4 byte signed number (-2,147,483,648 to +2,147,483,647). Commas are allowed. The address of the 4 byte number must be in AX before calling the subroutine.

print_signed_4byte
Displays a 4 byte signed number (-2,147,483,648 to +2,147,483,647) with commas. The address of the 4 byte number must be in AX before calling the subroutine.

get_signed_8byte
Enters an 8 byte signed number (-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807). Commas are allowed. The address of the 8 byte number must be in AX before calling the subroutine. The screen prompt will show the last 3 negative digits as -807 instead of -808, but this is because of lack of screen space.

print_signed_8byte
Displays an 8 byte signed number (-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807) with commas. The address of the 8 byte number must be in AX before calling the subroutine.

get_unsigned_byte
Enters a one byte unsigned number (0 to 255) and returns it in AL.

print_unsigned_byte
Displays a one byte number (from AL) as an unsigned number (0 to 255).

get_unsigned
Enters a one word unsigned number (0 to 65535) and returns it in AX.

print_unsigned
Displays a one word number (from AX) as an unsigned number (0 to 65535).

get_unsigned_4byte
Enters a 4 byte unsigned number (0 to 4,294,967,295). Commas are allowed. The address of the 4 byte number must be in AX before calling the subroutine.

print_unsigned_4byte
    Displays a 4 byte unsigned number (0 to 4,294,967,295) with commas. The address of the 4 byte number must be in AX before calling the subroutine.

get_unsigned_8byte
    Enters an 8 byte unsigned number (0 to 18,446,744,073,709,551,615). Commas are allowed. The address of the 8 byte number must be in AX before calling the subroutine.

print_unsigned_8byte
    Displays an 8 byte unsigned number (0 to 18,446,744,073,709,551,615) with commas. The address of the 8 byte number must be in AX before calling the subroutine.

Some of the above routines allow commas to be used for data entry. These routines strip the commas before looking at the number, so the following numbers all give the equivalent input:

```
2134875
2,134,875
21,,,34875
2,1,3,4,8,7,5
21,34,87,5
```

show_regs
    Displays the 8086 registers on the top of the screen. Each call to show_regs increments a resettable counter so you can know where you are in the program. The counter starts with an initial value of 0. It also scrolls the screen up if the cursor is past line 19.

show_regs_and_wait
    The same as show_regs except that it waits for you to press ENTER before continuing.

set_count
    Sets the counter in show_regs to the value in AX. The new counter value (incremented by 1) will appear the next time show_regs is called.

set_blue
    If you have a color monitor which is displaying color text, it sets the background to blue.

get_continue
    Waits for you to press ENTER before continuing the program. It enters no data. See also set_timer below.

set_timer
    In order to allow use with a debugger, it is possible to set a timer so the print functions keep control of the screen for a specific amount of time. To use set_timer, you put a number from 1 to 5 in AL, and call set_timer. This will cause a 1 to 5 second delay every time a print function or

show_regs is called. A 0 in AL will reset the timer to 0. A number larger than 5 in AL will cause ASMHELP to wait for you to press the ENTER key every time a print function or show_regs is called.

kill_timer
Resets the timer to 0.

set_reg_style
Sets the display style of the individual registers. The correct order of the definition is:

AX, BX, CX, DX, SI, DI, BP, SP

The style values are:

	FULL REGISTER OR RIGHT HALF REGISTER
signed	= 1d 1h
unsigned	= 2d 2h
binary	= 3d 3h
hex	= 4d 4h
ascii	= 5d 5h

plus (if applicable)

	LEFT HALF REGISTER AND HALF REG. BIT
signed	= 144d 90h
unsigned	= 160d A0h
binary	= 176d B0h
hex	= 192d C0h
ascii	= 208d D0h

set_reg_style makes a COPY of the information, which is used the next (and subsequent) times that show_regs is called. The next several paragraphs explain how this works. AX must contain the address of the 8 byte style definition array before calling set_reg_style.

REGISTER DISPLAY STYLE

The 8086 contains a number of different registers. Some of these always contain addresses and are always displayed in hex. These are CS, DS, ES, SS and IP. They are always in hex and cannot be changed. Also, each flag has an unchangeable style which will be explained later.

This still leaves us with AX, BX, CX, DX, SI, DI, BP and SP. Any of these registers can be displayed in any style desired. In addition, AX, BX, CX and DX can each be broken into two half registers, and each half register has an independent style. The default style is full register, unsigned. If you call show_regs without setting a style, the eight abovementioned registers will all display full, unsigned numbers.

In order to change the style, you need to set up an 8 byte style definition array in your data segment. The correct definition for this is the following:

```
ax_byte  db   2
bx_byte  db   2
cx_byte  db   2
dx_byte  db   2
si_byte  db   2
di_byte  db   2
bp_byte  db   2
sp_byte  db   2
```

This is the order that ASMHELP.OBJ expects. It is also the order that the registers appear on the screen. The number 2 is the default value for each byte. Once you have defined the 8 bytes, you may alter any of them that you want to.

The possible styles are signed, unsigned, binary, hex, and ascii. If you look at a byte:

```
76543210
HLLL0RRR
```

the leftmost bit (80h or 128d) signals a half or a full register. If it is 1, you get a half register, if it is 0, you get a full register. bits 4,5 and 6 represent the left half register (if appropriate), while bits 0,1 and 2 represent either the full register or the right half register. For SI, DI, BP and SP, only bits 0, 1 and 2 are significant. For AX, BX, CX and DX, all bits except bit 3 are significant.

The code for bits 0, 1 and 2 is the following:

```
signed    = 1d          1h
unsigned  = 2d          2h
binary    = 3d          3h
hex       = 4d          4h
ascii     = 5d          5h
```

This is the correct code for either a full register or the right half register.

If you want half registers, you must add 80h (128d) and the code for the left half register. We have for the left half register:

```
                DECIMAL                HEX

signed    = 16d  + 128d           10h + 80h
unsigned  = 32d  + 128d           20h + 80h
binary    = 48d  + 128d           30h + 80h
hex       = 64d  + 128d           40h + 80h
ascii     = 80d  + 128d           50h + 80h
```

which is the same as the above codes but shifted left 4 bits. Since the 128d and the left half register code always appear in

tandem, we may simply add them together. This gives us:

	DECIMAL	HEX
signed	= 144d	90h
unsigned	= 160d	A0h
binary	= 176d	B0h
hex	= 192d	C0h
ascii	= 208d	D0h

Here are some examples:

```
signed left + hex right           = 144 + 4 = 148
ascii left  + unsigned right      = 208 + 2 = 210
binary left + ascii right         = 176 + 5 = 181
```

Simply move the constant to the appropriate byte:

```
mov  cx_byte, 210        ; ascii left, unsigned right
mov  ax_byte, 5          ; full register, ascii
mov  si_byte, 1          ; full register signed
mov  dx_byte, 148        ; signed left, hex right
mov  bx_byte, 4          ; full register, hex
```

After you have changed all the styles you want to, put the address of ax_byte (the first byte of the array) in ax and call set_reg_style:

```
lea  ax, ax_byte
call set_reg_style
```

ASMHELP.OBJ uses the address in AX and stores a COPY of those 8 bytes. The next time you call show_regs, it will use this copy to define the styles. If you make a mistake, it will default to unsigned.

Registers are displayed the same way as with the 'print_' subroutines. Unsigned numbers have no sign. Signed numbers have a + or -, hex has an 'H', binary is either 8 or 16 digits, and ascii has a single asterisk '*' unless one of the characters is not printable (00d to 32d, 127d and 255d) in which case the non-printable character will be written as a two digit hex number, and a double asterisk will signal the event.

Finally, the flags are displayed as follows:

OF, IEF, TF, ZF, AF, and CF are blank if they are not set and have an X if they are set.

DF has a + or a - to indicate increment or decrement.

SF has a + or a - to indicate the sign.

PF has E (for even) or O (for odd).

LINKING

In order to use ASMHELP.OBJ you must have some standard segments in your program. The standard code segment is defined as:

```
CODESTUFF SEGMENT PUBLIC 'CODE'
```

and the standard data segment is:

```
DATASTUFF SEGMENT PUBLIC 'DATA'
```

In addition you need a stack segment:

```
STACKSEG SEGMENT STACK 'STACK'
```

which should have at least 100 bytes for ASMHELP to use.

ASMHELP.OBJ expects that when you call one of its routines the following conditions are met:

(1) It is a near call, and CS is set to the CODESTUFF segment.

(2) DS is set to the DATASTUFF segment. Any data which is longer than one word long and is being transferred must be in the DATASTUFF segment and its offset address must be in AX. For one byte or one word data, the data itself is in AX/AL.

(3) There is available stack space.

In order to link properly, any subroutine which is called must have an EXTRN statement.

```
EXTRN  show_regs:NEAR
```

If the above conditions hold, then simply link your object file with asmhelp.obj:

```
C> link myfile.obj+asmhelp.obj
```

and the subroutines will be usable.

ALIASES

The subroutine names were chosen so their functions would be completely clear. However, they tend to be long so if you use them with some frequency it would be easier to use aliases. Make a redefinition macro file and then include it at the beginning of the program. For instance, if you have:

```
call print_unsigned_8byte
```

you might want the EQU statement

```
    prt_u8  EQU  print_unsigned_8byte
```

and then rewrite the call:

```
    call prt_u8
```

Include redefinitions which are reasonable to you and put them in a file REDEF.MAC.

```
    prt_s8    EQU  print_signed_8byte
    get_u4    EQU  get_unsigned_4byte
    show_rw   EQU  show_regs_and_wait
```

Then all you need on the first line of your program is:

```
    include REDEF.MAC
```

and the assembler will do the work for you.

MEMORY RESIDENT SHOW_REGS

If you are not using a debugger, it is possible to have the show_regs portion of The Assembler Helper resident in memory. This means that once it is installed, it will stay there till you turn the machine off or reset the machine. The name of the program is HELPMEM.COM and it is in \XTRAFILE. It operates exactly the same way as show_regs except you get to it by using INT 3. At that point you can set TF to do single stepping.

You load it into memory by typing:

```
    >helpmem
```

and it will wait for you to send interrupts.

The first time you do INT 3, you will see the normal show_regs screen. The count is reset to 0 each time you have an INT 3. There will also be two menu lines:

```
----------
0=clear TF ; X=set TF ; A=regs from ax ; B=set count ; C=continue
1=ax ; 2=bx ; 3=cx ; 4=dx ; 5=si ; 6=di ; 7=bp ; 8=sp
----------
```

This is pretty clear. 0 clears TF and X sets TF. If you press B you will be prompted for a new count number. C lets you continue. You can set the registers individually. Each register has a number; ax is 1, bx is 2, etc. When you press the number, you will be prompted for a HEX style code. This is the same style code you have been using all the time.

Finally, if you want to transfer the style information from your program, do the following:

```
mov  ax, offset ax_byte
int  3
```

Load the address of ax_byte in AX before the interrupt, and then press selection A. The 8 bytes located at the address in AX will be transfered to HELPMEM. The advantage of this is that you can run ASMHELP concurrently, and they will both show the same register styles.

When you move into single step mode, you get a different prompt line:

```
----------
0 = clear TF and continue  ;  2 = menu  ;  other keys = continue
----------
```

0 clears TF, 2 sends you to the above menu, and any other key will continue the program with TF set.

As is true with all debuggers, you should not single step through subroutine calls since the subroutine might have hundreds or thousands of steps. In addition, the subroutines in ASMHELP store the flags. This means that they may store the flags with the trap flag set. If you clear the trap flag while inside ASMHELP, when the code exits ASMHELP it will POP the flags with the trap flag set. If this happens, keep pressing 0 until you get out of single step mode.

The memory resident version works fine for single stepping as long as there are no interrupts or subroutine calls. The show_regs in ASMHELP works fine inbetween subroutine calls but requires a lot of coding for single stepping. Therefore, a strategy for using these two programs in conjunction is:

1) use the same register style definition array for both programs.

2) when there is only 8086 code with no interrupts or subroutine calls, use INT 3 and single step.

3) When there are interrupts and subroutines interspersed with the 8086 instructions, switch to ASMHELP.

The screens should look the same except the count will be different.

I/O

All data i/o is done to the current screen at the current cursor position. As long as you are in a text mode, ASMHELP should write to whatever is current.

SHOW_REGS is a different matter. The only allowed modes are 2, 3 and 7. If you enter in modes 2, 3 or 7, it will keep the same mode. If it is in any other mode, the video card will be forced to mode 3.

The memory resident version does almost the same thing. Every time it is called, it checks for modes 2, 3 or 7, and if the mode is not one of these, changes the mode to mode 3. If the mode is 3, then HELPMEM looks at the first character on page 0 and uses the attribute of that character for all its display operations. Whatever display attribute is in the upper left hand corner of page 0 will be the display attribute for everything.

The register display occupies the first 10 lines of page 0 of whatever mode you are in. This is written directly to memory and cannot be changed. Both versions change the page to page 0 upon entry. If you are on another page, after a call to show_regs you will be on page 0 at its current cursor position.

If you use ASMHELP in a programming environment or with a debugger there may be conflicts as to who controls the screen display. HELPMEM.COM should not be used in a programming environment and cannot be used with a debugger, since both HELPMEM and the debugger try to use the same interrupts.

THE 8086 INSTRUCTION SET

Before I go through the instructions I need to say something about the structure of the instructions. For the majority of instructions, two bits in the first instruction byte and the whole second byte determine whether it is a byte or word instruction and whether it is:

1. register, register
2. register, memory
3. memory, register

These refer to the 8 arithmetic registers, not the segment registers. "Memory" is any allowable addressing mode. Therefore, it is simpler to abbreviate this as:

```
reg/mem, reg/mem
```

This will always mean:

1. either a byte or word operation is allowed
2. any of the above 3 possibilities is allowed

unless specifically stated otherwise. Another possibility is the operations with constants:

```
add  ax, 7
xor  sign_flag, 80h
```

These follow the same form. There are two possibilities:

1. register, constant
2. memory, constant

where this may be a byte or a word, "register" is any arithmetic register, and "memory" is any allowable addressing mode. These will be abbreviated:

```
reg/mem, constant
```

If we are taliking about a segment register, they will be abbreviated:

```
segreg
```

If anything does not follow this pattern, it will be explicitly stated.

DESTINATION,SOURCE

The standard way of writing instructions is with the destination on the left and the source on the right:

```
    add  destination, source
```

The register or memory location on the left ALWAYS gets the result of the operation. The thing on the right is the second operand (if any).

ADDRESSING MODES

These are the natural (default) segments of all addressing modes:

(1) DS

```
variable + (constant)
[bx] + (constant)
[si] + (constant)
[di] + (constant)
[bx+si] + (constant)
[bx+di] + (constant)
```

(2) SS

```
[bp] + (constant)
[bp+si] + (constant)
[bp+di] + (constant)
```

Where the constant is optional. Segment overrides may be used. The segment overrides are:

SEGMENT OVERRIDE	MACHINE CODE (hex)
CS:	2E
DS:	3E
ES:	26
SS:	36

These default segments apply to all instructions except the string instructions, which will be explained individually.

THE INSTRUCTION SET

AAA (ascii adjust for addition) adjusts AL, assuming that it contains the result of a legitimate unpacked BCD addition. If the lower half-byte has generated a result 10 or over, it subtracts 10, carries 1 to AH, and sets the carry flag. If the result is 9 or less, it clears CF. In either case it zeroes the upper half-byte of AL.

AAD (ascii adjust for division) PREPARES AL and AH for division. It assumes that AH contains the 10's digit and AL contains the 1's digit of a two byte unpacked BCD number. It multiplies AH by 10 and adds it to AL, thus making a single integer between 0 and 99. It zeroes AH in preparation for unsigned division.

AAM (ascii adjust for multiplication) adjusts AL, assuming that it contains the result of a legitimate BCD multiplication. It divides the result by 10, putting the quotient in AH and the remainder in AL. If AL contains 73 before AAM, then afterwards AH will contain 7 and AL will contain 3

AAS (ascii adjust for subtraction) adjusts AL, assuming that it contains the result of a legitimate unpacked BCD subtraction. If the lower half-byte has generated a result -1 or less, it borrows 1 from AH, adds 10 to AL, and sets the carry flag. If the result is 0 or more, it clears CF. In either case it zeroes the upper half-byte of AL.

ADC (add with carry) adds two integers, either signed or unsigned, and adds in the carry from the previous arithmetic operation. It is used for multiple word (byte) addition.

```
adc  reg/mem, reg/mem
adc  reg/mem, constant
```

ADD adds two integers, either signed or unsigned.

```
add  reg/mem, reg/mem
add  reg/mem, constant
```

AND ands two bytes or words. A bit remains set only if both its respective source and destination bits were set.

```
and  reg/mem, reg/mem
and  reg/mem, constant
```

CALL calls a procedure. The two forms we have used are NEAR and FAR calls.

```
call set_regs
call FAR PTR calculate_array
```

There are two other forms where the addresses of the subprograms are in memory and changable. You should only use these forms if you are writing an operating system or a compiler, since they are an invitation to disaster.

CBW sign extends the signed byte in AL to AH:AL in preparation for signed division. It is used alone without any register specification.

```
cbw
```

CLC clears the carry flag (CF = 0).

CLD clears the direction flag (increments string pointers).

CLI clears the interrupt flag (disables interrupts). All maskable interrupts must wait till the flag is set again before they will be processed.

CMC changes the value of the carry flag. If CF =1, then CF = 0; if CF = 0, then CF = 1.

CMP compares two values. It is the same as SUB except it does not alter the "destination" variable. Its job is to set the flags as if a subtraction had been performed, in preparation for a conditional jump instruction.

```
cmp  reg/mem, reg/mem
cmp  reg/mem, constant
```

CMPS compares the byte (or word) at DS:[si] with the one at ES:[di], (calculating [si] - [di]) and sets the flags accordingly. It increments (or decrements) SI and DI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE ES:[di]. Notice that SI and DI are reversed from their position in the other string instructions.

```
cmpsb
cmpsw
cmps BYTE PTR SS:[si], ES:[di]     ;or CS, DS, ES:[si]
cmps WORD PTR SS:[si], ES:[di]     ;or CS, DS, ES:[si]
```

CWD sign extends the signed integer in AX to DX:AX in preparation for signed word division. No register specification is used.

```
cwd
```

DAA (decimal adjust for addition) adjusts AL, assuming that it contains the result of a legitimate packed BCD addition. It treats AL as two independent half-bytes. If the result of the lower half-byte is 10 or over, it subtracts 10 from the lower half-byte and adds the carry to the upper half byte. It then looks at the upper half byte. If its result is 10 or over, DAA subtracts 10 from the upper half byte and sets the carry flag.

Otherwise the carry flag is cleared.

DAS (decimal adjust for subtraction) adjusts AL, assuming that it contains the result of a legitimate packed BCD subtraction.It treats AL as two independent half-bytes. If the result of the lower half-byte is -1 or less, it adds 10 to the lower half-byte and borrows 1 from the upper half byte. It then looks at the upper half byte. If its result is -1 or less, DAA adds 10 to the upper half byte and sets the carry flag to indicate a borrow. Otherwise the carry flag is cleared.

DEC decrements the byte or word by 1. It does not alter CF.

```
dec  reg/mem
```

DIV performs unsigned division. If byte division, the unsigned double byte must be in AH:AL. Afterwards, AL has the quotient and AH has the remainder. If word division, the unsigned word must be in DX:AX. Afterwards, AX has the quotient, and DX the remainder. Division by a constant is not allowed. The DIV instruction does not mention DX:AX or AH:AL. They are understood.

```
div  reg/mem
```

ESC (escape) signals to the 8086 that the bytes starting with the ESC byte are a coprocessor instruction. The 8086 will calculate a memory address (if appropriate) and give it to the coprocessor. The 8086 will then go on to the next instruction.

HLT (halt) halts the machine. It can be restarted with a reset, a non-maskable interrupt, or (if IEF is set) with a maskable interrupt.

IDIV performs signed integer division. For byte division, the value in AL must be sign extended to AH (with CBW), giving the double signed byte AH:AL, after division, AL contains the quotient and AH contains the remainder. For word division, the value in AX must be sign extended to DX (with CWD), giving the double signed word DX:AX, after division, AX contains the quotient and DX contains the remainder. Division by a constant is not allowed. IDIV cannot perform multiple word division. To do that, you need to make the numbers positive and use DIV, adjusting the signs afterwards. IDIV does not mention the AL or AX register, it is understood.

```
    idiv reg/mem
```

IMUL performs signed integer multiplication. The multiplicand must be in AX (or AL for bytes). After the multiplication, the result is in DX:AX for words and AH:AL for bytes. If DX (or AH for bytes) contains significant information, then CF is set (=1). If all the information is in AX (or AL for bytes) then CF = 0. Multiplication by a constant is not allowed. The IMUL instruction does not mention AX (or AL); it is understood.

```
    imul mem/reg
```

IN allows you to move data from a port address to AX (for a word) or AL (for a byte). There are two forms:

```
    IN   al, port_number
    IN   al, dx
```

Where 'port_number' is a CONSTANT that is hard coded into the machine instruction and is from 0 - 255. The port address in DX may be from 0-65535. It must be the DX register.

INC increments a register or a variable by 1. It does not effect CF.

```
    inc  reg/mem
```

INT (interrupt) sends the program to a subprogram whose address is located in the interrupt vector table in low memory. For any interrupt number I, the 8086 goes to segment 0000, offset (I X 4). It loads IP with the first two bytes, then loads CS from the next two bytes. The next instruction executed will be at the new CS:IP. Before loading the new CS and IP, it pushes (1) the flags, (2) the old CS and (3) the old IP in that order.

```
    int  3
    int  21h
```

INTO checks OF. If OF = 1, INTO generates interrupt 4. Otherwise it does nothing. It is used for error handling of signed numbers.

```
    into
```

IRET (interrupt return) is a special return instruction for interrupt routines. It pops (1) the old IP, (2) the old CS, and (3) the old flags in that order.

```
    iret
```

JUMPS ----------

There are two kinds of jumps. JMP is unconditional and can jump anywhere in the segment. All other jumps are conditional and are limited jumps from -128 to +127 bytes from the END of the jump instruction.{1}

THESE ARE THE JUMP INSTRUCTIONS FOR SIGNED NUMBERS

```
    jg        ; jump if greater
    jnle      ; jump if not (less or equal)

    jl        ; jump if less
    jnge      ; jump if not (greater or equal)

    je        ; jump if equal
    jz        ; jump if zero

    jge       ; jump if greater or equal
    jnl       ; jump if not less

    jle       ; jump if less or equal
    jng       ; jump if not greater

    jne       ; jump if not equal
    jnz       ; jump if not zero
```

THESE ARE THE JUMP INSTRUCTIONS FOR UNSIGNED NUMBERS

```
    ja        ; jump if above
    jnbe      ; jump if not (below or equal)

    jb        ; jump if below
    jnae      ; jump if not (above or equal)

    je        ; jump if equal
    jz        ; jump if zero

    jae       ; jump if above or equal
    jnb       ; jump if not below

    jbe       ; jump if below or equal
    jna       ; jump if not above
```

1 There is also a long jump which can jump anywhere in memory (from 00000 to FFFFF). Except under special circumstances, using this is truly lousy programming practice. The long jump is for the Olympics, not for quality programming.

```
    jne       ; jump if not equal
    jnz       ; jump if not zero
```

THESE JUMPS CHECK A SINGLE FLAG

These come in opposite pairs

```
    jc        ; jump if the carry flag is set
    jnc       ; jump if the carry flag is not set

    jo        ; jump if the overflow flag is set
    jno       ; jump if the overflow flag is not set

    jp or jpe ; jump if parity flag is set (parity is even)
    jnp or jpo  ;jump if parity flag is not set (parity is odd)


    js        ; jump if the sign flag is set (negative )
    jns       ; jump if the sign flag is not set (positive or 0)
```

THIS CHECKS THE CX REGISTER

```
    jcxz      ; jump if cx is zero
```

This is used before entry to a loop to make sure the loop counter is not set to 0.

INDIRECT JUMPS are jumps where the information for the jump is stored in memory (or in a register for an in-segment jump). These forms are dangerous and should be used only when writing things like multi-tasking operating systems. Have you written one lately?

END OF JUMPS -

LAHF (load AH from flags) loads the lower half of the flags register into AH.

LDS (load DS) loads the first two bytes of the memory address into the named arithmetic register and the next two bytes into DS. The register may be any arithmetic register, but this is designed for the 4 addressing registers: BX, SI, DI and BP.

```
         lds  reg, memory_address
```

LEA calculates an offset address and puts it in the named register.

```
        lea  ax, [bp+si]+145
```

LES (load ES) loads the first two bytes into the named arithmetic register and the next two bytes into ES. The register may be any arithmetic register, but this is designed for the 4 addressing registers: BX, SI, DI and BP.

```
        les  reg, memory_address
```

LOCK is for use if there is more than one 8086 operating in a computer. LOCK allows one 8086 to be the only one which can read from or write to memory during the following instruction.

LODS moves a byte or word from DS:[si] to AL or AX, and increments (or decrements) SI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si].

```
    lodsb
    lodsw
    lods BYTE PTR SS:[si]         ; or CS, DS, ES:[si]
    lods WORD PTR SS:[si]         ; or CS, DS, ES:[si]
```

LOOP/LOOPE/LOOPNE are conditional jumps (-128 to + 127 bytes). They will jump back to the start of the loop (which is identified by a label), under the following conditions:

```
    loop      decrement cx ; jump back if cx is not zero
    loope     decrement cx ; jump back if cx not zero AND zf = 1
    loopz     decrement cx ; jump back if cx not zero AND zf = 1
    loopne    decrement cx ; jump back if cx not zero AND zf = 0
    loopnz    decrement cx ; jump back if cx not zero AND zf = 0
```

Here, 'e' stands for equal, 'z' is zero and 'n' is not.

```
    loop some_label
```

MOV is the general instruction for moving bytes or words on the 8086. The forms are:

```
    reg/mem, reg/mem
    reg/mem, constant
    segreg, reg/mem
    reg/mem, segreg
```

Notice that you may not (1) move a constant to a segment register, (2) move a segment register to another segment

register, or (3) move from memory to memory.

MOVS moves a byte (or a word) from DS:[si] to ES:[di], and increments (or decrements) SI and DI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE ES:[di].

```
movsb
movsw
movs BYTE PTR ES:[di], SS:[si]     ;or CS, DS, ES:[si]
movs WORD PTR ES:[di], SS:[si]     ;or CS, DS, ES:[si]
```

MUL performs unsigned integer multiplication. The multiplicand must be in AX (or AL for bytes). After the multiplication, the result is in DX:AX (or AH:AX for bytes). If DX (or AH for bytes) contains significant information, then CF = 1. If all the significant information is in AX (or AL for bytes) then CF = 0. Multiplication by a constant is not allowed. The MUL instruction does not mention AX (or AL). It is understood.

```
mul  reg/mem
```

NEG performs '0 - number' on a number and sets the flags accordingly.

```
neg  reg/mem
```

NOP (no operation) does absolutely nothing. It can fill up space which was previously occupied by a different instruction.

NOT performs bitwise logical NOT on a word or a byte.

```
not  reg/mem
```

OR performs bitwise logical OR on a byte or word

```
or   reg/mem, reg/mem
or   reg/mem, constant
```

OUT moves a byte (or word) from AL (or AX) to the named port. There are two forms:

```
    out  port_number, ax
    out  dx, ax
```

where 'port_number' is a CONSTANT which is coded into the machine code. The constant may be from 0-255. DX (and it must be DX) may hold a port number from 0-65535.

POP pops a WORD off the stack. Technically, POP takes the word at SS:SP and puts it into the named register or memory location, then ADDS 2 to SP.

```
    pop  mem/reg
    pop  segreg
```

POPF pops a WORD off the top of the stack into the flags register. Its technical operation is the same as for POP.

PUSH pushes a WORD on the stack from the named register or memory location. Technically, PUSH subtracts 2 from SP, then moves the word to SS:SP.

```
    push reg/mem
    push segreg
```

PUSHF pushes the flags register on the stack. Its technical operation is the same as for PUSH.

RCR (rotate through carry right) and RCL (rotate through carry left) rotate the bits of a register or of memory data right and left respectively. The bit which is shoved off the register (or data) is placed in CF and CF is placed on the other side of the register (or data). There are two fixed forms. (1) rotate 1 bit and (2) rotate by the number in CL.

```
    rcr  reg/mem, 1
    rcl  reg/mem, cl
```

REP. The string instructions may be prefixed by REP/REPE/REPNE which will repeat the instructions according to the following conditions:

```
    rep       decrement cx ; repeat if cx is not zero
    repe      decrement cx ; repeat if cx not zero AND zf = 1
    repz      decrement cx ; repeat if cx not zero AND zf = 1
    repne     decrement cx ; repeat if cx not zero AND zf = 0
```

```
    repnz     decrement cx ; repeat if cx not zero AND zf = 0
```

Here, 'e' stands for equal, 'z' is zero and 'n' is not. These repeat instructions should NEVER be used with a segment override, since the 8086 will forget the override if a hardware interrupt occurs in the middle of the REP loop.

```
    rep    movsb
    repe   scasb
    repne  cmpsw
```

RET returns from the subroutine to the calling program. The return will be either NEAR or FAR depending on the type of procedure it is in. It may take the parameters off the stack (for Pascal) or leave them on the stack (for C).

```
    ret (6)        ; Pascal - take 6 bytes off the stack
    ret            ; C - do nothing
```

ROR (rotate right) and ROL (rotate left) rotate the bits of a register or memory data right and left respectively. The bit which is shoved off one end is moved to the other end. CF indicates whether the last bit moved from one end to the other was a 1 or a 0. There are two fixed forms. (1) rotate 1 bit and (2) rotate by the number in CL.

```
    ror  reg/mem, 1
    rol  reg/mem, cl
```

SAHF (store AH into flags) puts AH into the low byte of the flags register.

SAL (shift arithmetic left) and SHL (shift logical left) are exactly the same instruction. They move bits left. 0s are placed in the low bit. Bits are shoved off the register or memory data on the left side, and CF indicates whether the last bit shoved off was a 1 or a 0. It is used for multiplying an unsigned number by powers of 2. There are two fixed forms. (1) shift 1 bit and (2) shift by the number in CL.

```
    shl  reg/mem, 1
    sal  reg/mem, cl
```

SAR (shift arithmetic right) shifts bits right. The high (sign) bit stays the same throughout the operation. Bits are shoved off the register or memory location on the right side. CF indicates whether the last bit shoved off was a 1 or a 0. It is used (with

difficulty) for dividing a signed number by powers of 2. There are two fixed forms. (1) shift 1 bit and (2) shift by the number in CL.

```
sar  reg/mem, 1
sar  reg/mem, cl
```

SBB (subtract with borrow) subtracts one integer from another and then subtracts 1 more if CF is set. It works with both signed and unsigned numbers and is used for multiple word arithmetic.

```
sbb  reg/mem, reg/mem
sbb  reg/mem, constant
```

SCAS compares AL (or AX) to the byte (or word) pointed to by ES:[di], and increments (or decrements) DI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). NO OVERRIDES ARE ALLOWED.

```
scasb
scasw
scas BYTE PTR ES:[di]    ; no override allowed
scas WORD PTR ES:[di]    ; no override allowed
```

SHR (shift logical right) does the same thing as SHL but in the opposite direction. Bits are shifted right. 0s are placed in the high bit. Bits are shoved off the register or memory location on the right side and CF indicates whether the last bit shoved off was a 0 or a 1. It is used for dividing an unsigned number by powers of 2. There are two fixed forms. (1) shift 1 bit and (2) shift by the number in CL.

```
shr  reg/mem, 1
shr  reg/mem, cl
```

STC sets the carry flag (CF = 1).

STD sets the direction flag (DF = 1, which decrements).

STI sets the interrupt flag (IEF = 1; interrupts enabled).

STOS (store to string) moves a byte (or a word) from AL (or AX) to ES:[di], and increments (or decrements) DI depending on the setting of DF, the direction flag (by 1 for bytes and by 2 for words). NO SEGMENT OVERRIDES ARE ALLOWED.

```
stosb
```

```
    stosw
    stos BYTE PTR ES:[di]    ; no override allowed
    stos WORD PTR ES:[di]    ; no override allowed
```

SUB subtracts one integer from another. It works for both signed and unsigned numbers.

```
    sub  reg/mem, reg/mem
    sub  reg/mem, constant
```

TEST performs logical AND, but does not alter the value of the "destination" variable. Its purpose is to set the flags for conditional jumps.

```
    test reg/mem, reg/mem
    test reg/mem, constant
```

WAIT forces the 8086 to wait until the coprocessor is finished with its instruction before the 8086 can go on to the next instruction.

XCHG exchanges the values in two registers or memory locations.

```
    xchg reg/mem, reg/mem
```

XLAT. If BX contains the address of a 256 byte translation table, then XLAT goes to (BX + AL), takes the byte there and puts it into AL. DS is the normal segment, but segment overrides are allowed.

XOR performs logical exclusive OR on two numbers.

```
    xor  reg/mem, reg/mem
    xor  reg/mem, constant
```

APPENDIX III - INSTRUCTION SPEED AND FLAGS

This appendix contains a list of all the 8086 instructions along with their relatives speed and the flags they affect. These speed numbers are from INTEL and are in clock ticks.{1} If you have a 25 mhz clock, then it is doing 25 million ticks per second. If you have a 4.77 mhz clock, then it is doing 4.77 million ticks per second. Instead of calling them ticks, I'll be calling them clocks.

In order to understand these numbers, you need a little extra information. In order to do an instruction, ANY microprocessor must:

(1) calculate any memory addresses.
(2) fetch the two operands (or the single operand if it is an instruction like NOT).
(3) process the instruction and
(4) put the result in the destination.

While (3) will require the same amount of time whether the operands are in memory or in registers, (1), (2) and (4) are all different depending on where the operands are. Some machines allow both operands to be in memory, but the 8086 does not. Therefore, (1) is either NO memory addresses (if everything is in registers) or ONE memory address.

Calculating a memory address involves (a) calculating the offset from the beginning of the segment and (b) adding it to the segment starting address. Part (a) is different depending on the addressing mode. In terms of the speed of calculating (a) and (b), the order is:

(i) one pointer
(ii) named variable (+ constant)
(iii) two pointers
(iv) one pointer + constant
(v) two pointers + constant

The reason that (ii) contains both "variable" and "variable + constant" is that the addition (variable + constant) is done by the assembler, not the 8086. By the time the 8086 sees it, it is a single number. The constants in both (iv) and (v) need to be added by the 8086, and this takes extra time. According to INTEL, here is the time required for calculating all possible memory addresses. Note that some pointers are marginally faster than other pointers (this is trivial - don't worry about it).

---

1 All the speed numbers are from "Programmer's Pocket Reference Guide", (c)1980, 1982 Intel Corporation.

	ADDRESSING MODE	CLOCKS (EA)
(i)	[bx], [si], [di], [bp]	5
(ii)	variable (+ constant)	6
(iii)	[bp+di] or [bx+si]	7
	[bp+si] or [bx+di]	8
(iv)	([bx] or [si] or [di] or [bp]) + constant	9
(v)	([bp+di] or [bx+si]) + constant	11
	([bp+si] or [bx+di]) + constant	12

The most complicated memory address takes 2.4 times longer to calculate than the simplest address. These calculation times will be noted by EA (calculate Effective Address). Remember, if both operands are in registers, this calculation does not have to be done.

In order for you to see how all of this works, we'll use ADD as an example. Don't start using the table till you understand this example.

On the 8086, we normaly have the following possibilities for source and destination:

```
register, register
register, memory
memory, register
register, constant
memory, constant
```

In Appendix II, we simply combined them as:

```
reg/mem, reg/mem
reg/mem, constant
```

We can't do this here because they all have different times. For ADD they are:

ADD	CLOCKS
register, register	3
register, memory	9 + EA
memory, register	16 + EA
register, constant	4
memory, constant	17 + EA

Notice how much faster using a register is. The EA stands for "calculate Effective Address" and is the number from the list above. If you have:

```
add  ax, bx
```

that is "register, register", and it will take 3 clocks to execute. If you have:

```
add [bx+di+9], 17
```

that is "memory, constant" and will take 17 + EA. What is EA here? According to the above list, [BX+DI+CONSTANT] takes 12

cycles, so 17 + EA is 17 + 12 is 29, so this will take 29 clocks. That's right. The one instruction is almost 10 times slower than the other. If you can move things into some registers, do a number of calculations, and then move them back to memory, you can save a lot of time. Let's do a few examples to make sure that you see all of them:

INSTRUCTION	TYPE	TIME
add variable1, bl	memory, register	16 + EA = 22
add bl, variable1	register, memory	9 + EA = 15
add [si], di	memory, register	16 + EA = 21
add di, [si]	register, memory	9 + EA = 14

Examples 1 and 2 are the same except that source and destination have been switched. The same applies to examples 3 and 4. Notice that when the 8086 has to fetch the variable and then put the result back in memory, it is significantly slower than when it just gets the variable from memory and puts the result in a register.

INSTRUCTION	TYPE	TIME
add ax, cx	register, register	3
add di, 1876	register, constant	4
add variable1, 199	memory, constant	17 + EA = 23

To show you how the different types of EA effect the time, let's do all 3 types of "source, destination" that involve memory. First, "memory, register":

INSTRUCTION	TIME
add [bx], ax	16 + EA = 21
add variable1, ax	16 + EA = 22
add [bp+di], ax	16 + EA = 23
add [bp+si], ax	16 + EA = 24
add [bx+9], ax	16 + EA = 25
add [bp+di+294], ax	16 + EA = 27
add [bx+di+294], ax	16 + EA = 28

Now let's do the same things but go "register, memory":

INSTRUCTION	TIME
add ax, [bx]	9 + EA = 14
add ax, variable1	9 + EA = 15
add ax, [bp+di]	9 + EA = 16
add ax, [bp+si]	9 + EA = 17
add ax, [bx+9]	9 + EA = 18
add ax, [bp+di+294]	9 + EA = 20
add ax, [bx+di+294]	9 + EA = 21

And finally we have "memory, constant":

INSTRUCTION	TIME
add [bx], 177	17 + EA = 22
add variable1, 177	17 + EA = 23
add [bp+di], 177	17 + EA = 24
add [bp+si], 177	17 + EA = 25
add [bx+9], 177	17 + EA = 26
add [bp+di+294], 177	17 + EA = 28
add [bx+di+294], 177	17 + EA = 29

Is this everything you need to know before looking at the list? Not quite. Most of the 8086 family has a 16 bit data bus. That means that there are 16 wires connecting the processor to memory, and the processor reads 1 word (2 bytes) at a time. These memory reads ALWAYS start at an even location 1472d, 88026d, 198752d, etc. If you are reading one byte, it makes no difference whether it is at an even or odd location. If you are reading a word at an even location, then everything is normal. If you are reading a word at an ODD location, however, the processor must:

1) start reading at the first even location that contains the variable.
2) read the next even location (which contains the last part of the variable).
3) join the parts together.

As an example, let's take a word at address 21957 (i.e. 21957-21958). The processor will:

1) read the high byte from the word at 21956
2) read the low byte from the word at 21958
3) join them together. It now has 21957-21958.

The processor can do this, but it takes extra time (4 extra clock ticks), so our speed listing will also contain the following notice:

WORDS WHICH ARE AT ODD ADDRESSES NEED 4 EXTRA CLOCKS

Thus for:

```
add  ax, variable1        (9 + EA = 15)
```

if the address is an even location, the instruction will require 15 clocks. If it is an odd location, the instruction will require 19 clocks (4 extra ticks). It is worth your while to keep words at even locations if at all possible.

INSTRUCTION SPEEDS AND FLAGS

REGISTER:
One of the arithmetic registers - either word (AX, BX, CX, DX, SI, DI, BP, SP for word operations) or byte (AH, AL, BH, BL, CH, CL, DH or DL for byte operations).

AX or AL:
AX and AL are considered special on the 8086. Sometimes there is a special AX/AL form of the instruction, (such as for ADD). This form will be shorter. It will only be noted if it is FASTER (such as for MOV) or if it is the only form allowed. It will either say (AX/AL) or (AX only) depending on whether both words and bytes are allowed or only words are allowed. Though both multiplication and division require the use of the (AX/AL) register, the AX/AL is understood, and is not mentioned in the instruction.

SEGREG:
One of the 4 segment registers - CS, DS, ES or SS.

MEMORY:
Either a byte or word in memory. It may be addressed with any possible addressing mode, and the extra time needed to calculate the address in memory (EA - calculate Effective Address) is the following:

	ADDRESSING MODE	CLOCKS (EA)
(i)	[bx], [si], [di], [bp]	5
(ii)	variable (+ constant)	6
(iii)	[bp+di] or [bx+si]	7
	[bp+si] or [bx+di]	8
(iv)	([bx] or [si] or [di] or [bp]) + constant	9
(v)	([bp+di] or [bx+si]) + constant	11
	([bp+si] or [bx+di]) + constant	12

EXTRA TIME:
1) WORDS WHICH ARE AT ODD ADDRESSES NEED 4 EXTRA CLOCKS.
2) A segment override adds 2 clocks to the instruction.

FLAGS

Following the instruction mnemonic is information in square brackets that indicates how the instruction effects the flags register. There are three possibilities

1) The instruction may alter the value of a flag in a particular way depending on the result. For AND, the sign, zero and parity flags [SZP] will be set according to the result.

2) The instruction may set a flag to a specific number (either 1 or 0). For AND, the overflow flag and the carry flag are cleared [(OC=0)].

3) The instruction may unreliably alter a flag. That means that the instruction might change the flag, but that it gives you no information. This kind of flag cannot be trusted after this operation. For AND, the auxillary flag is unreliable [?A?].

The information will always be displayed in this order. The flags which are reliably set according to the result will be listed first [SZP]. Next, any flags which are either set or cleared will be put inside parentheses followed by an equal sign followed by the value of the flag (all inside of the parentheses) [SZP,(OC=0)]. Finally, any unrelaible flags will be put between question marks [SZP,(OC=0),?A?]. Each part will be separated by commas. If no flags are changed by the instruction, the brackets will have [none] written between them.

Each flags will be indicated by a single letter. The letters are:

O	overflow flag	0 = no overflow, 1 = overflow
D	direction flag	direction of movement for string instructions 0 = upwards, 1 = downwards
I	interrupt enable	0 = no ints, 1 = ints o.k.
T	trap flag	trap next instruction? 0 = no trap, 1 = trap
S	sign flag	0 = positive, 1 = negative
Z	zero flag	0 = non-zero, 1 = zero
A	auxillary flag	carry out of bottom half register? 0 = no, 1 = yes
P	parity flag	0 = odd, 1 = even
C	carry flag	0 = no carry, 1 = carry

********** THE INSTRUCTIONS **********

INSTRUCTION	TIMING
AAA [AC,?OSZP?]	4 clocks
AAD [SZP,?OAC?]	60 clocks
AAM [SZP,?OAC?]	83 clocks
AAS [AC,?OSZP?]	4 clocks

ADC [OSZAPC]	see ADD	
ADD [OSZAPC]	register, register	3
	register, memory	9 + EA
	memory, register	16 + EA
	register, constant	4
	memory, constant	17 + EA
AND [SZP,(OC=0),?A?]	see ADD	
CALL [none]	near call	19
	far call	28
	near call (reg-ind)	16 {2}
	near call (mem-ind)	21 + EA
	far call (mem-ind)	37 + EA
CBW [none]	2 clocks	
CLC [(C=0)]	2 clocks	
CLD [(D=0)]	2 clocks	
CLI [(I=0)]	2 clocks	
CMC [C]	2 clocks	
CMP [OSZAPC]	register, register	3
	register, memory	9 + EA
	memory, register	9 + EA
	register, constant	4
	memory, constant	10 + EA
CMPS [OSZAPC]	22 clocks	
CWD [none]	5 clocks	
DAA [SZAPC,?O?]	4 clocks	
DAS [SZAPC,?O?]	4 clocks	
DEC [OSZAP]	word register	2
	byte register	3
	word/byte memory	15 + EA
DIV [?OSZAPC?]	byte register	80 - 90 {3}
	word register	144 - 162

2 These three last ones are indirect calls. They get the address of the subroutine from a register (reg-ind) or from memory (mem-ind).

3 The smaller the numbers, the faster this operation can be accomplished. This applies for signed and unsigned multiplication and division. They all show a range of values rather than a specific value.

	byte memory	(86 - 96) + EA
	word memory	(150 - 168) + EA
ESC [none]	memory	8 + EA
	coproc. register	2
	This only makes sense if you know about coprocessors.	
HLT [none]	2 clocks	
IDIV [none]	byte register	101 - 112
	word register	165 - 184
	byte memory	(107 - 118) + EA
	word memory	(171 - 190) + EA
IMUL [OC,?SZAP?]	byte register	80 - 98
	word register	128 - 154
	byte memory	(86 - 104) + EA
	word memory	(134 - 160) + EA
IN [none]	(AX/AL), port#	10
	(AX/AL), dx	8
INC [OSCAP]	word register	2
	byte register	3
	word/byte memory	15 + EA
INT [(IT=0) {4}]	51 clocks {5}	
INTO [{6}]	overflow	54
	no overflow	4
IRET [{7}]	24 clocks	
J(condition) [none]	This includes all conditional jumps (JAE, JZ, JNO, JLE, JP, etc.) with the exception of JCXZ.	
	jump	16
	no jump	4

4 Although this sets the trap flag and interrupt flag to 0, it doesn't do it to YOUR flags, it does it to the flags that the interrupt will see. Your flags are safely stored on the stack and will return unaltered at the end of the interrupt.

5 There is one exception. INT 3, when coded as the single byte trap interrupt, is 52 clocks.

6 If there is overflow, it pushes your flags and sets (IT=0) for the interrupt. If there was no overflow, it does nothing. In either case your own flags will remain unaffected.

7 This puts the copy of your old flags back in the flags register. The flags will be the same as they were when you called the interrupt.

JCXZ [none]	jump	18
	no jump	6
JMP [none]	same segment	15
	different segment	15
	near (reg-ind)	11 {8}
	near (mem-ind)	18 + EA
	far (mem-ind)	24 + EA
LAHF [none]	4 clocks	
LDS [none]	16 + EA	
LES [none]	16 + EA	
LEA [none]	2 + EA	
LOCK [none]	2 clocks	
LODS [none]	12 clocks	
LOOP [none]	jump	17
	no jump	5
LOOPE/LOOPZ [none]	jump	18
	no jump	6
LOOPNE/LOOPNZ [none]	jump	19
	no jump	5
MOV [none]	register, register	2
	register, memory	8 + EA
	memory, register	9 + EA
	register, constant	4
	memory, constant	10 + EA
	(AX/AL) <-> memory	10 {9}

---

8 These last three are indirect jumps. The information about where to jump to is coming from a register (reg-ind) or from memory (mem-ind).

9 This is a special instruction which moves a directly addressed variable to or from AX (or AL for bytes). Pointers are not allowed, only the forms:

```
mov  ax, variable1
mov  variable1, ax
```

This takes 10 clocks instead of 14 or 15 for the other form. Whether this form gets used is up to the assembler, not you. Fortunately, MASM, TurboAssembler and A86 all use this form when appropriate.

	segreg <-> register	2 {10}
	segreg, memory	8 + EA
	memory, segreg	9 + EA
MOVS [none]	11 clocks	
MUL [OC,?SZAP?]	byte register	70 - 77
	word register	118 - 133
	byte memory	(76 - 83) + EA
	word memory	(124 - 139) + EA
NEG [OSZAPC] {11}	register	3
	memory	16 + EA
NOP [none]	3 clocks	
NOT [none]	register	3
	memory	16 + EA
OR [SZP,(OC=0),?A?]	see ADD	
OUT [none]	(AX/AL), port#	10
	(AX/AL), dx	8
POP [none]	register	8
	segreg	8
	memory	17 + EA
POPF [{12}]	8 clocks	
PUSH [none]	register	11
	segreg	10
	memory	16 + EA
PUSHF [none]	10 clocks	
RCL [OC]	register by 1 bit	2
	memory by 1 bit	15 + EA
	register by # in CL	8 + (4 * #) {13}
	memory by # in CL	20 + EA + (4 * #)
RCR [OC]	see RCL	

10 MOV, PUSH and POP are the only instructions that can alter the segment registers (other than CALLs and JMPs).

11 (NEG number) sets the flags the same as (SUB 0, number).

12 POPF resets the flags register by POPping a word of the stack and using the values stored in that word.

13 Thus, if you rotate right by 3 bits add (4 * 3), if you rotate by 7 bits add (4 * 7), if you rotate by 2 bits add (4 * 2). As you can see, this can cost a lot of time if you are rotating more than 3 or 4 bits.

REP [none]	2 clocks	
RET [none]	near ret	8 {14}
	near ret (#)	12
	far ret	18
	far ret (#)	17
ROL [OC]	see RCL	
ROR [OC]	see RCL	
SAHF [{15}]	4 clocks	
SAL/SHL [OSZPC,?A?]	see RCL	
SAR [OSZPC,?A?]	see RCL	
SBB [OSZAPC]	see ADD	
SCAS [OSZAPC]	15 clocks	
SEGMENT OVERRIDE [none]	2 clocks	
SHR [OSZPC,?A?]	see RCL	
STC [(C=1)]	2 clocks	
STD [(D=1)]	2 clocks	
STI [(I=1)]	2 clocks	
STOS [none]	11 clocks	
SUB [OSZAPC]	see ADD	
TEST [SZP,(OC=0),?A?]	register, register	3
	register, memory	9 + EA
	memory, register	9 + EA
	(AX/AL), constant	4
	register, constant	5
	memory, constant	11 + EA
WAIT [none]	3 clocks minimum, then check every 5 clocks	
XCHG [none]	register, register	4
	(AX only), register	3

14 The # here indicates that you pop things off the stack as you would in a Pascal program:

```
ret (18)
ret (6)
```

15 Alters the values of the SZAPC flags according to the values in the AH register.

	register, memory	17 + EA
XLAT [none]	11 clocks	
XOR [SZP,(OC=0),?A?]	see ADD	

DEBUGGERS

Debuggers are designed to take you through your program step by step. This allows you to see which instruction is being executed at each moment, and this in turn allows you to pinpoint the spot where the program starts doing something you don't want it to do. In the beginning, debuggers like DEBUG would look at each instruction and give you the text mnemonic for that instruction. Each individual instruction has exactly one text mnemonic.

This was better than nothing, but it suffered from several problems. First, it substituted the address of a variable or subroutine for the variable or subroutine name. A name like 'result' became 0A84h, a hex number. Secondly, in a high level language, you know what the source code looks like, but you don't have the slightest idea what the machine code looks like, and you don't care. It is nearly impossible to tell where you are in a PASCAL program by looking at the machine code. You are interested in what is happening, not in how the compiler generated instructions for your code.

The general solution to these problems is a SYMBOLIC debugger. While the compiler or assembler is generating the object file, it keeps track of what LINE you are on in the text file. Each instruction is indexed by the text line where it is generated. This information is passed along to the linker, which processes it if it is asked to do so . Here is a program which was assembled once with the symbolic information and once without the information:

```
MIT      OBJ     1978   7-23-90   4:31p
OHNE     OBJ      952   7-23-90   4:31p
```

MIT.OBJ has the symbolic information, OHNE.OBJ doesn't. As you can see, this debugging information can take up as much space as the rest of the file.

When you start up the program under the debugger, the debugger reads both the executable file and the TEXT file into memory and displays the appropriate text line for each machine instruction.

In order to use a symbolic debugger, you need to:

1) tell the compiler to generate the debugger information
2) tell the linker to process the information
3) leave your text files UNALTERED.

For MASM, the instructions are:

1) masm /zi filename1 ;
2) link /co filename1+... ;

For TASM, the instructions are:

1) tasm /zi filename1 ;
2) tlink /v filename1+... ;

where the dots indicate that you may be linking more than one file.

It is not necessary for all the object files to have the debugging information. The debuggers will read the text files for those parts which have the debugging information and will generate code like DEBUG for those parts which don't have any information.

If you have a large program you probably want to concentrate on the section that is causing the problems. You set a breakpoint (which is a command to stop execution at a certain spot) for the start of this section, run the program, and then single step through the section. That one section is the only place that needs the symbolic information.

Both Codeview and Turbo Debugger do much the same thing. Here is the Turbo screen:

```
************************** TURBO SCREEN {1} ***************************
  File   View   Run   Breakpoints   Data   Window   Options        READY
.Module: debugtst  File: debugtst.asm 74.................................1.
.                                                                        .
.  start: push  ds               ; set up for return   .Registers......3. .
.         sub   ax,ax                                    .  ax 5C94   .c=0. .
.         push  ax                                       .  bx 0000   .z=1. .
.                                                        .  cx 0000   .s=0. .
.         mov   ax, DATASTUFF    ; load ds               .  dx 0000   .o=0. .
.         mov   ds,ax                                    .  si 0000   .p=1. .
.                                                        .  di 0000   .a=0. .
.  outer_loop:                                           .  bp 0000   .i=1. .
.          lea     ax, multiplicand        ; load multi. .  sp 09FA   .d=0. .
.          call    get_unsigned_8byte                    .  ds 4AD6   .   . .
.          call    print_unsigned_8byte                  .  es 4A26   .   . .
.          call    get_unsigned            ; unsigned w. .  ss 4A36   .   . .
.          mov     multiplier, ax                        .  cs 4B27   .   . .
.                                                        .  ip 0019   .   . .
.                                                        ..................
.          lea     si, multiplicand        ; load pointers                 .
.........................................................................
.Watches..................................................................2.
.multiplier,d                 23700                                      .
.multiplicand                 qword 00000042E843515D                     .
.........................................................................
F1-Help F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run
****************************** END TURBO ******************************
```

Here's the Codeview screen:

1. Turbo Debugger Version 1.5

```
************************ CODEVIEW SCREEN {2} ***************************
.File  View  Search  Run  Watch  Options  Language  Calls  Help .
F8=Trace F5=Go
.......................... debugtst.ASM ..................................
73:     main   proc far                                          . AX = 0000
74:                                                              . BX = 0000
75:     start: push  ds              ; set up for return         . CX = 0000
76:            sub   ax,ax                                       . DX = 0000
77:            push  ax                                          . SP = 0A00
78:                                                              . BP = 0000
79:            mov   ax, DATASTUFF    ; load ds                  . SI = 0000
80:            mov   ds,ax                                       . DI = 0000
81:                                                              . DS = 44FD
82:     outer_loop:                                              . ES = 44FD
83:             lea     ax, multiplicand        ; load multipli. SS = 450D
84:             call    get_unsigned_8byte                       . CS = 4601
85:             call    print_unsigned_8byte                     . IP = 0000
86:             call    get_unsigned            ; unsigned word.
87:             mov     multiplier, ax                           .   NV UP
88:                                                              .   EI PL
89:                                                              .   NZ NA
90:             lea     si, multiplicand        ; load pointers. .   PO NC
.....................................................................
Microsoft (R) CodeView (R)  Version 2.2
(C) Copyright Microsoft Corp. 1986-1988.  All rights reserved.
>
***************************** END CODEVIEW *****************************
```

As you can see, they look almost the same and they operate similarly. The Borland people (Turbo Debugger) have gone an extra step and have made it possible to use Turbo Debugger with both Borland and Microsoft high-level languages.

I/O AND ASMHELP

All programs do output. These two debuggers deal with this by reserving a section of video memory for the program's output. You put output in one place in memory and the debugger puts output in a different place. As long as things are being done in an orderly manner, when your screen is doing output your screen is visible and when the debugger has charge its screen is showing.

The problem here is that your program may take only 1/100 of a second to write to the screen. This will appear as a flash on the screen. You won't see anything. Both debuggers allow you to toggle back and forth between the debugger screen and your screen. This is tedious, but it works.

In order to alleviate the problem, ASMHELP has a timer. You

2. Microsoft (R) CodeView (R) Version 2.2
   (C) Copyright Microsoft Corp. 1986-1988.
   All rights reserved.

activate the timer with:

```
    mov  al, #          ; # is a number from 1 to 5
    call set_timer
```

where you put a number from 1 to 5 in AL. You deactivate it with:

```
    call kill_timer
```

You put a number from 1 to 5 in AL before the call to 'set_timer'. This will create a 1 to 5 second delay. From that point on, every time a print function in ASMHELP is called, it will show the screen for the specified number of seconds. If you put a number larger than 5 in AL:

```
    mov  al, 120
    call set_timer
```

ASMHELP will require you to press the ENTER key before continuing. This allows you to view the user screen as much time or as little time as you want.

The idea here is for you to set the timer at the beginning of your program. Then the rest of the program will have the extended viewing time for the print functions.

D86

D86 is what I'd call a semi-symbolic debugger. It does not use line information - it uses a symbol table. This is a list of symbols in the program along with where they appear in the code. This is much better than DEBUG but is not as valuable as using the text files. It also means that it can't work that well with high-level languages. Here is the D86 output from the same piece of code:

```
************************** D86 SCREEN {3} **************************
START:                                B  Set permanent breakpoints
# 0000 PUSH DS                        F  Find MARKed memory bytes
  0001 SUB AX,AX                      G  Go, until address(es) reached
  0003 PUSH AX                        J  Jump within this code segment
  0004 MOV AX,0253F                   L  List disassembly to LST file
  0007 MOV DS,AX                      O  set operating system-call traps
OUTER_LOOP:                           Q  Quit debugging session
  0009 MOV AX,8                       W  Write program and SYM to disk
  000C CALL 0151A
  000F CALL 01253                     Alt-F9    restore trashed screen
  0012 CALL 069B                      Shift-F7  set MARK at CS:IP
  0015 MOV MULTIPLIER,AX              Home      Jump last-trap/prog-start
  0018 MOV SI,8
  001B MOV BX,012
```

```
AX 0929            1:
BX 001A  IP 0000   2:
CX 0000  CS 2593   3:
DX 0000  SS 249F   4:
SI 0010  DS 253F   5:
DI 0000  ES 248F   6:
BP 42D9  SP 09F2   6: 001A 06B1 F202 0015 0000 248F

g 0015
****************************** END D86 ******************************
```

As you can see, D86 has lost some of the information, but you can follow this as long as you have a printout of the text file at your side.

D86 has some weaknesses. It makes no allowance for user output, so all user output is either erased immediately or is mixed with the information on the debugger screen. It has no key to let you jump over subroutines (you either need to go through the whole subroutine or set a breakpoint after every subroutine call). Finally, it alters the counter which keeps track of where you are in the program.

When using a debugger you want to scroll through the program to find places where you want to stop, and then set breakpoints. If you scroll with D86, it will change the program location. If you don't know the EXACT location where you were before you started scrolling, you can't get back to continue debugging the program. You need to reload the file and start over again. Think of what this would mean if you had been working on a program for 15 minutes. Why D86 does this is completely beyond me.

USING TASM

This is absolutely the easiest chapter in the book. The default mode for TASM is something called MASM mode. It imitates what MASM does, warts and all. When you are finished with the Tutor, you can decide whether you want to use "IDEAL" mode.

The following are the minor differences from MASM.

===== CHAPTER 1

To assemble the file myprog.asm use:

```
>tasm myprog
```

You don't need commas or semicolons or anything. Then if you are using TLINK, use:

```
>tlink myprog+\asmhelp
```

The batch file for this would be:

```
>tlink %1+\asmhelp
```

No commas or semicolons.

===== CHAPTER 10

To get your list file put three commas after the filename:

```
.tasm myprog, , ,
```

This will produce MYPROG.LST. You will notice that the list files look similar.

Page 94 - This is the TURBO listing for variable2:

```
     42 000E  8E C0               mov   es,ax
     43
     44 0010  26: 8B 0E 0000r     mov   cx, variable2
     45 0015  26: 89 0E 0000r     mov   variable2, cx
     46
     47 001A  CB                          ret
```

This is the TURBO listing for variable3:

```
     42 000E  8E C0               mov   es,ax
     43
     44 0010  2E: 8B 0E 0000r     mov   cx, variable3
     45 0015  2E: 89 0E 0000r     mov   variable3, cx
     46
     47 001A  CB                          ret
     48
```

This is the TURBO listing for variable4:

```
     42 000E  8E C0               mov   es,ax
     43
     44 0010  36: 8B 0E 0000r     mov   cx, variable4
     45 0015  36: 89 0E 0000r     mov   variable4, cx
     46
     47 001A  CB                      ret
     48
```

Except for the line numbers at the left, they look exactly the same as what is in the text. TASM is calculating the segment overrides.

PAGE 99 - This is the TURBO listing for calls similar to the ones on page 99:

```
     89 0AFF  E8 FEC8            call    near_routine
     90 0B02  9A 00000AF6sr      call    far_routine
     91 0B07  E8 0000e           call    near_external_routine
     92 0B0A  9A 00000000se      call    far_external_routine
     93 0B0F  E8 0000e           call    get_unsigned
```

and here is the MASM output for the same file:

```
    0AFF  E8 09CA R             call    near_routine
    0B02  9A 0AF6 ---- R        call    far_routine
    0B07  E8 0000 E             call    near_external_routine
    0B0A  9A 0000 ---- E        call    far_external_routine
    0B0F  E8 0000 E             call    get_unsigned
```

Why is that first number different (FEC8 vs. 09CA)? Turbo knows the correct value so it is inserting it into the code. MASM is passing the information on to the linker for calculation. The first way makes the linking faster.

Some of these numbers are reversed for display purposes. The actual code is the same.

===== CHAPTER 21

You don't need to use EXE2BIN after TLINK. TLINK will make a .COM file (if possible) from object files if you use the /t option with the following commands:

```
    >tlink /t  myprog
    >tlink /t prog1+prog2+prog3
```

you will get .COM files as output.

USING A86

This is intended as a guide to get A86 to work for all the programs in the Tutor. It only covers things that won't work or things A86 does differently. It does not cover any of the benign options like local labels.

===== CHAPTER 1

You need to take your text editor and change the last line of the template files. At the moment they read:

```
END start
```

You may either eliminate the line or change it to:

```
END main
```

If you don't, you will get this error message:

```
 END start~40 Conflicting Multiple Definition Not Allowed~
```

Do this for all the original template files, because this line will always cause an error. You will find out why when you get to Chapter 10.

You should also insert the word PUBLIC just before the line with ASSUME. Do this in all the template files.

```
PUBLIC
ASSUME cs:CODESTUFF, ds:DATASTUFF
```

You will find out why in Chapter 15.

A86 has some unusual defaults, so you need to change them when you assemble a file. If your file is named myprog.asm, the command line you want to use for the rest of this tutorial is:

```
>a86 +DOSX myprog.asm
```

You must use the file extension .asm when you call the program. You can make a batch file (let's call it QASM.BAT) which has:

```
a86 +DOSX  %1.asm
```

Then all you need to do is:

```
>qasm myprog
```

Those four options on the command line are:

```
    +D   numbers with leading 0s are default decimal
    +O   make an object file
    +S   don't create an ancillary symbol table
    +X   require specific declaration of external information
```

They should be used from now on. When you are through with all the chapters, you can decide whether you want to continue using them all. If you are using D86, you need to eliminate the 'S'. The declaration should be +DOX since you will need the symbol table.

===== CHAPTER 2

Please read all of chapter 2 before you read these comments.

The defaults and definitions are the same with two exceptions. First, any number that begins with a '0' is hex by default. That is, 22 is a decimal number and 022 is a hex number. This has some serious side effects. Take the number:

```
    0847d
```

This is 847, right? No. It is 0847Dh = 34,637. In Chapter 7 we will use these binary numbers:

```
    01001011b
    00000100b
```

These should be 75 and 4 respectively, but A86 evaluates them as 01001011Bh = 268500992 and 0100Bh = 4107. Since half of the binary numbers you use will start with a 0, half of your binary numbers will be wildly incorrect. In order to correct the situation, you need to put a +D on the command line. This is the 'D' of +DOSX. The D will insure that all numbers starting with 0 will be decimal. This gets rid of all the side effects. Alternatively, you can put:

```
    RADIX 10
```

at the top of your files before any of the code or data. This has the same effect. Of course, if you really do want hex numbers, this can be overridden with a specific 'h' after any number.

There is another major problem with data definitions. Let's say that you want the following definitions:

```
    weight    dw    ?
    cost      dw    20
    profit    dw    ?
```

but you forget to put in some of the initializations in your text file (a situation which is not unknown):

```
    weight    dw
```

```
cost      dw
profit    dw    ?
```

All other assemblers will generate an error. You must either put in an initial value or you must say that you want no initial value (by putting in a question mark). Not only does A86 not generate an error, it does something strange. If there is neither initialization nor a question mark, A86 will calculate an address for the variable but WILL ALLOCATE NO SPACE FOR IT. What this means is that in our above example, 'weight', 'cost', and 'profit' will all be at the same memory address. 'Weight' and 'cost' will have the same address with no space allocation. 'Profit' will be at the same memory address too, but will have space allocated for it.

This is all done silently, so you don't know that this is being done. There is no way to shut this off. You might be able to find that this is happening by looking at the symbol table, but that means that you suspect that it is happening. What you need to do whenever you use A86 is:

> ALWAYS CHECK ALL DATA TO MAKE SURE THAT IT IS INITIALIZED OR HAS A QUESTION MARK

===== CHAPTER 10

We now come to some actual differences. A86 does not produce a list file. You can read the justification for this in your documentation. Does this make a difference? It would be nice to have one, but we can normally live without it. Just follow what is happening with the MASM listing.

p94 - ASSUME statements. Both TASM and MASM keep track of how you want the segments to be referenced. A86 doesn't. It simply ignores any ASSUME statements. This means that any segment overrides must be coded into the text file with:

```
mov  cx, ss:variable4
mov  cx, es:variable2
mov  cx, ds:variable1
mov  cx, cs:variable3
```

A86 assumes that all code uses CS (which it must), and that all data uses DS unless there is a specific segment override. Is this a big imposition? Not really. If you use a segment register other than DS, it is normally for dealing with arrays, and in those cases you need a segment override anyways. You will find out about that in the next chapter.

p97 - END statements. END is not necessary in A86. It knows that the end has come when the file is finished. The default starting point for an A86 file is the label 'main'. If A86 sees a label name after an END statement, then it RENAMES that word 'main', and renames all occurances of that word in the file 'main'. It does not change the default starting name. This means that if you

have a statement like:

```
    END start
```

and you also have the label 'main' in your file, A86 will change all occurances of 'start' to 'main', causing multiple use of the same label.

======== Chapter 15

p154 - You do not need PUSHREGS and POPREGS. With A86, PUSH is exactly the macro PUSHREGS:

```
    PUSHREGS ax, bx, cx, dx, si
    PUSH     ax, bx, cx, dx, si
```

These are exactly the same. POP shows the actual order that things will be popped, while POPREGS has the reverse order so that you can use a word processor to copy the PUSHREGS part. For the above PUSH declaration, this is the correct POP:

```
    POP     si, dx, cx, bx, ax
    POPREGS ax, bx, cx, dx, si
```

From now on, whenever you see PUSHREGS, change it to PUSH, and whenever you see POPREGS, change it to POP but reverse the order of registers so that it is the actual order in which things are POPped. You can delete the line:

```
    include \pushregs.mac
```

from your template file.

; - - - - - - - - - -

Please read the whole chapter before reading the rest of these comments.

For some reason, A86 makes ALL normal variables and labels PUBLIC unless you specifically tell it not to. The whole thrust of programming for the last 20 years (PASCAL, C, and all structured languages), has been to make NOTHING public unless specifically requested, so this is an unwelcome default setting. You turn it off by naming any normal (but not local) variable PUBLIC. In fact, if you use the word PUBLIC alone:

```
    PUBLIC
```

without any name after it, A86 will turn this off. That is why you put it in all the template files. Always have this in all your assembler files.

A86 also assumes that any variable that has no data declaration is EXTRN. This is a bad policy. What will happen is the following. You are using the variable 'last_occurance' and you

misspell it 'last_ocurrance'. A86 sees no data declaration so assumes that it is an external variable and assembles the file. Three hours later you link the 6 modules of your program together and you get a link error. The linker cannot find 'last_ocurrance' anywhere.

Just as you should need to explicitly name things PUBLIC, you should need to explicitly name things EXTRN. You do this by putting +X (forced external declarations) on the command line. This is the 'X' in the '+DOSX' that you have been using on the command line.

===== CHAPTER 21

You need to change the ORG statement so it is in front of main

```
ORG 100h
main proc near
```

This is because you are using main as the starting address, so it must be at 100h. This 100h is optional with A86. If it is making a .COM file it automatically starts at 100h. A86 is designed to give you a .COM file directly, so all you need to do with a single file is:

```
>a86 +DS   myfile.asm
```

and you will get MYFILE.COM as the output file. In the example where we make a .COM file from multiple files, these files are set up to be .OBJ files. If you want to make them into a .COM file directly, you need to take out all PUBLIC, EXTRN and END statements (and adjust PUSHREGS and POPREGS) and then do:

```
>a86 +DS  prog1.asm prog2.asm prog3.asm
```

making sure that prog1.asm is first. The output file should be prog1.com.

===== CHAPTER 26

You need to read all of Chapter 26 before reading any of the following.

A86 will work fine with any standard segment definitions. It also has two special segment statements of its own. The first:

```
CODE SEGMENT
```

will generate the following:

```
_TEXT SEGMENT PUBLIC 'CODE'
```

The second statement:

```
DATA SEGMENT
```

is NOT a segment declaration. It generates NO segment, it allocates NO space and it initializes NO data. What is it? I'll explain how it operates, but I have no idea why it exists.

The first time you use the 'DATA SEGMENT' instruction in a file it is followed by ORG:

```
DATA SEGMENT
ORG 1000h
```

ORG relocates the counter for where the assembler is allocating space in memory. From this point on, every time you define variables:

```
time         dw   ?
distance     dw   ?
space        dw   ?
```

A86 generates an distinct address for each variable. However, it ALLOCATES NO SPACE. This is just an address that the code can use when it refers to the variables. If you have the following variables:

```
year        dw    1990
prime       dw    73
cost        dw    5167
```

A86 will generate address for these variables but (1) will allocate NO space, (2) will do NOTHING with those initializing values and (3) WILL NOT COMPLAIN that you have tried using initialized data in a DATA SEGMENT statement. A86 will do the equivalent of:

```
year        dw    ?
prime       dw    ?
cost        dw    ?
```

This data is technically in the CODE SEGMENT. If the code is longer than that initial ORG number, some of the code will be in the same place as some of the data and if you write to the data you will overwrite the code. A86 will not complain.

For instance, we had ORG 1000h (4096d). If the code is 6000 bytes long, the last 2000 bytes of code will share the space with the data. This leads to either bad data or bad code or both.

Why have this? Well, when DOS loads a .COM file into memory, it uses ALL of memory. If I have:

```
INTSCRN.COM     541   07/23/90  17:45
```

a .COM file that is 541 bytes long, DOS will allocate about 550,000 bytes for it on my computer. 549,459 bytes of this are wasted space. It is A86's idea to put data in this wasted space. There is no reason for this, so don't do it. If you have a .COM file, put all your data in the CODE SEGMENT. If you have an .OBJ

file you need to create a legitimate SEGMENT for the DATA. Its name cannot be:

```
DATA  SEGMENT  PUBLIC 'DATA'
```

If I have the following program:

```
; - - - - - - - - - - - - - - -
DATA SEGMENT PUBLIC
data1   db  "This is the output string", 13, 10, "$"
data2   db  100 dup (0)
ENDS

CODE SEGMENT

main:
        mov     ax, seg DATA
        mov     ds, ax

        mov     ah, 09h         ; print string
        lea     dx, data1
        int     21h

        mov     ax, 4C00h       ; exit
        int     21h
; - - - - - - - - - - - - - - -
```

and I try to make an object file with it, I get the following error:

```
      mov  ax, seg DATA~61 Overlapping Local Not Allowed~
```

The ability to put the segment starting address in the segment register is a minimal requirement for doing anything. This effectively eliminates the possibility of using DATA as the name for a segment. Therefore, those of you who use Turbo Pascal need to use:

```
DSEG  SEGMENT PUBLIC
```

as an alternative. This will work with no problems.

USING BASIC - 1

So you like using BASIC. I can't blame you. BASIC has several very nice features.

(1) Its graphics interface is easy to use and is very powerful. If you want to draw lines, circles, patterns or other graphics images, you can do it. Your only limitations are the aspect ratio of the screen and the possible screen colors. These limitations have to do with hardware, not with BASIC.

(2) Its string handling capabilities are unparalleled. If you want to have an array of strings like the following:

```
DIM  STRING.ARRAY$ (50, 25)  ' 1250 elements
```

and some of the elements are only a couple of characters long but some of the strings are several hundred characters long, BASIC is the ONLY language that can handle it correctly. In fact, if you wanted one of the strings to be 1000 characters (it's allowed in the most recent BASIC), then you would need 1.25 megabytes for this array in C or in PASCAL. This would probably be too big for the computer. BASIC, on the other hand, implements this without wasting any space in memory.

BASIC has a few things that require a little more work than with other languages. It has strings, integers, and floating point numbers but it is missing characters and unsigned numbers - these can be implemented by using integers (if you are careful). This is not all that limiting.

If BASIC is so nice, why do I think that you should have some experience with a structured language before doing assembler? It all has to do with data INTEGRITY. The question is, are you sure that you haven't inadvertantly screwed up your data? There are two ways that this might happen. You need to know how BASIC works to understand the problem.

A variable name in BASIC consists of a name followed by a type identifier. The identifiers are '%', '!', '$', '#' (double precision) and possibly '&' (long integer). The name:

```
CURRENT.TEMPERATURE
```

is NOT a variable inside of BASIC while:

```
CURRENT.TEMPERATURE%
CURRENT.TEMPERATURE!
CURRENT.TEMPERATURE$
```

---

are all valid variables. In fact, all three can be in the same program, as we shall see shortly. I just said that that first example is not a valid variable inside of BASIC, but that is exactly what you have been writing since the first time that you wrote a BASIC program. What's going on here? Well, when the BASIC interpreter or compiler reads the text file, it reads the name and the identifier. If the type identifier is missing, it tacks the default identifier on to the end of the name.

What's the default type identifier? The answer is '!'. BASIC maintains a default list. The legal first characters in a name are A-Z. For each legal first letter, there is an entry in the list which says what the default type for that letter is. When you start the program, the type list looks like this:

```
!!!!!!!!!!!!!!!!!!!!!!!!!
```

When the interpreter sees a name without an identifier, it takes the first character of the name and goes to the default-type list, gets the appropraiate type identifier for that first character, and adds the identifier on to the end of the name. Externally (in your text file) there may be names without type identifiers, but internally (inside the interpreter/compiler) ALL variables have the type identifier after the name.

Though the list starts out with all '!'s, you can change this list with a DEF statement. Here are some DEF statements and how they change the list. I am assuming that we are always starting at the beginning of the program (with all '!'s).

```
(1)
DEFINT A-Z     ' This is probably your favorite
%%%%%%%%%%%%%%%%%%%%%%%%%

(2)
DEFINT A-E
%%%%%!!!!!!!!!!!!!!!!!!!!

(3)
DEFINT A-E
DEFSTR F-K
DEFDBL L-R
%%%%%$$$$$$#######!!!!!!!!
```

This last one is a recipe for disaster, but it's legal.

Have you understood everything so far? Then let's go on. This ability to NOT use the type identifier leads to laziness. The usual thing is to always identify strings with a '$' but to be rather lax about the distinction between integers and floating point numbers.

Now we get to the problem. Look at the following code:

```
10       DEFINT A-Z
20       FOR I = 1 to 5
30            LARGE.NUMBER! = 300.0 * I
40            SMALL.NUMBER! = LARGE.NUMBER / 7
50            PRINT  LARGE.NUMBER!, SMALL.NUMBER!
60            NEXT I
```

If you can't see the error, I forgot to put the '!' after LARGE.NUMBER in line 40. The problem here is that BASIC is not going to complain. When it sees LARGE.NUMBER without a type identifier, it will go to the default list (because of the DEFINT statement, all defaults are integer) and put a % after it. BASIC has created a SECOND variable with the same initial name, but a different type. This is not because you wanted it to, but rather because you forgot a '!'. Lets do a BASIC program. Here's some code:

```
******************  NINJA.BAS ******************

10                        ' place for defint statement
20      '
30      ' enter all the data
40      NINJA% = 20
50      NINJA! = 4.7134E+13
60      NINJA$ = "Hey dude!"
70      '
80      NINJA$ (0) = "howdy! "
90      FOR I% = 1 TO 5
100         NINJA$ (I%) = NINJA$ ( I% - 1 ) + "doody! "
110         NINJA! (I%) = I% * 25.31
120         NINJA% (I%) = I% * 3
130     NEXT I%
140     '
150     ' print all the data
160     PRINT   NINJA%, NINJA!, NINJA$
170     FOR I% = 1 TO 5
180         PRINT NINJA% (I%), NINJA! (I%), NINJA$ (I%)
190     NEXT I%
200     PRINT  NINJA , NINJA (3)
210     '
220     END

*********************************************************
```

I actually want you to run this through your BASIC. I have defined SIX different NINJAs. They are:

```
1)   NINJA%
2)   NINJA% ( )
3)   NINJA!
4)   NINJA! ( )
5)   NINJA$
6)   NINJA$ ( )
```

where the parentheses indicate an array. If I had gotten carried away, I could have included 4 more NINJAs for double precision,

double precision array, long integer and long integer array.

Running this through my computer I get:

```
20           4.7134E+13  Hey dude!
3            25.31       howdy! doody!
6            50.62       howdy! doody! doody!
9            75.93       howdy! doody! doody! doody!
12           101.24      howdy! doody! doody! doody! doody!
15           126.55      howdy! doody! doody! doody! doody! doody!
4.7134E+13   75.93
```

Notice that I have 3 different array types and 3 different variable types. As you can see from the bottom line (which prints the defaults), the default here is '!'. If we now put DEFINT A-Z on line 10:

```
    10   DEFINT A-Z
```

everything will stay the same except the bottom line which now is:

```
    20        9
```

which are the integers. Finally, if we have DEFSTR A-Z

```
    10   DEFSTR A-Z
```

we get:

```
    Hey dude!    howdy! doody! doody! doody!
```

as the bottom line of output. If you didn't know all of this before, make sure you have assimilated this before going on.

PASCAL and C

Both PASCAL and C require you to account for every variable. You need to specifically state what kind of variable it is and you need to enter it in a list of variables used in that block of code:

```
    int    ch, variable1, variable2, variable3[60] ;
    char   long_array[4027] ;
```

A name can be used in only one way in any block of code. If you try to use it differently, the compiler will generate an error. Though people moan and groan about having to specifically list all the variables used in a block of code, over time they get used to it. It offers two advantages.

(1) No variable can be inadvertently used in an incorrect way (used as a string when you wanted an integer, used as an integer when you wanted a floating point number, etc.).

(2) There can be no variables which are misspelled, are left

over from a previous version of the code or appear by mistake, without both the compiler and you knowing about them.

Slightly modifying names is one of my major problems. I might use 'long_array' and 'longarray' in different places in the C code, though I want the same thing. C will catch this, BASIC won't. Every time BASIC encounters a slightly different name it simply creates a new variable (and normally sets it to 0). In programs larger than a few pages, this situation is almost impossible to detect or control.

THE RANGE OF A VARIABLE

When talking about PASCAL and C I kept using the word BLOCK. Both languages have a modular (or block) design. The philosophy behind this is that when you write a program you want to divide a task into a number of distinct subtasks, and you don't want these subtasks to interfere with each other. You put each of these in a subprogram. In BASIC you can put sections of a program in subprograms too. This is good programming practice. There is a difference, however. In C and PASCAL, a variable is valid ONLY in the block in which it is declared (unless you take specific steps to make it universally valid). If you try to use this variable in a different block of code the compiler will generate an error - it literally has no information about where this variable is. In BASIC, all names are valid everywhere in a file. If on page 1 you have:

```
210 CURRENT.PRICE! = 427.11
```

you may want to use this as a constant number throughout the program. But if in a subprogram on page 23 you have:

```
12060     CURRENT.PRICE! = 0
```

you have changed the value. You're saying to yourself "But why would I change a variable like "CURRENT.PRICE!"? You probably wouldn't. But you will have dozens and dozens of variables around with names like I, J, K, COUNT, START, STARTING.NUMBER, STARTING.VALUE, TOTAL, RESULT and LAST.VALUE (whatever their type). It is exactly these that will be corrupted in BASIC but will be reliable in C or PASCAL.

```
*******************************************
1920      MINIMUM! = RATE! * 5.0
1930      MAXIMUM! = RATE! * 7.32
1940      COUNT% = (MAXIMUM! - MINIMUM!) / 3.0
1950      CALL CHECK.ANSWER
1960      PRINT COUNT%
*******************************************
```

In this section of code, did 'CALL CHECK.ANSWER' change any of the four variables MINIMUM!, MAXIMUM!, RATE! or COUNT%? In BASIC, we hope so, but we can't be sure. In PASCAL or C we KNOW that it

didn't change any code. This is the difference between a structural engineer HOPING that a bridge won't fall down and KNOWING that a bridge won't fall down.

In fact, the PASCAL and C programmers will probably be annoyed when they start using assembler because in assembler, ALL names are valid everywhere. More importantly, however, these programmers will make great efforts to insure that all constants stay constants and that each variable is used for one job only.

Finally, some BASIC programmers still do most of their programming using GOTO statements instead of CALLs. This is the antethesis of modular programming and is called spaghetti programming. The code winds up as an intertwined mess. If you are one of these people, once you start writing assembler code it will be impossible to figure out what the code is doing once it is longer than a page or two.

All that being said, if you feel that your programming style is clear and orderly, have a fun time learning assembler. When you are done with all the chapters, read the second BASIC appendix on the problems associated with linking a non-BASIC subprogram to a BASIC program.

BASIC II - INTERFACING BASIC WITH ASSEMBLER

Have you finished reading all the chapters? If not, go back and do them, then come back to this when you are done. This chapter assumes you know about segments, subroutines, and the general information about linking subroutines to high-level languages.

In order to do this appendix I had to dust off my old QuickBASIC 3.0. If you have QuickBASIC 4.x, some things will have been updated. If you have TurboBASIC, the subroutine conventions are different. However, the structure will be the same. You will have to consult your manual for exact details. If you are trying to this with the interpreter that came with DOS, I have a simple comment -> Forget it! I won't go into the details, but the BASIC interpreter is so much slower (about 10 times slower) and so much more difficult to use with assembler that I won't even cover it. There is no reason not to use one of the compiled BASICs if BASIC is your language of choice. This material only covers how to deal with COMPILED BASIC.

In BASIC, all individual numeric data, strings, "static" arrays and the stack must fit into one 64k segment. The word 'segment' here has the same meaning as in assembler. Both the DS register and the SS register are set to this segment, and must stay set to this segment whenever BASIC has control of the program. "Dynamic" arrays can be located somewhere else in memory.

You allocate a "static" array with a constant number as a dimension:

```
DIM  array1! (277), array2% (346), array3$ (500)
```

and you allocate a "dynamic" array by using a variable to dimension the array:

```
length1% = 277
length2% = 346
length3$ = 500

DIM  array1!(length1%), array2%(length2%), array3$(length3%)
```

Even though the first and second dimension statements produce the same size and type arrays, the first ones must be located inside DS and the second ones can be located outside of DS.

"Static" means that once the array is defined, its length and number of dimensions cannot be changed for the rest of the program. It will occupy a specific amount of space for the rest of the program. "Dynamic" means that you can change the length of the array whenever you want to. You do this with:

```
REDIM  array1! (495)
```

BASIC does this by deallocating space for the old array and then reallocating space for the new array. All the old information is lost. There are certain restrictions. You cannot change the number of dimensions in an array (if it starts out with 2 dimensions like DIM A!(47,63), it must always have 2 dimensions).

In order to understand BASIC's memory strategy, we need to look at strings, the reason for it all.{1} The limit for a single string is 32,767 bytes. If the total amount of data you can have in the DS segment is only 65536 bytes, how does BASIC allocate memory so you can have long strings without runnung out of space? It uses only as much space as it needs. Let's define 3 strings (the dots will indicate a space):

```
mystring$ = "You.say.either"
yourstring$ = "And.I.say.either"
ourstring$ = "Let's.call.it.off"
```

After defining these three strings one after the other, memory will look like this:

```
17150
|You.say.eitherAnd.I.say.eitherLet's.call.it.off|
```

(For clarity, the memory image will be between the '|'s and each row will be 50 bytes long. The next row down would start at 17200). For our example we will assume that this data starts at memory location 17150.

There is no empty space. How does BASIC know where and how long mystring$ is? It has something called a string descriptor. This is a two word (4 byte) block, also in DS, which says exactly where and how long the string is. The first word is the length and the second word is the location (offset) in DS.

From BASIC's view, we have:

```
STRING          DESCRIPTOR
                length:location

mystring$       14:17150        ->      |You.say.either|
yourstring$     16:17164        ->      |And.I.say.either|
ourstring$      17:17180        ->      |Let's.call.it.off|
```

Now let's change one of the strings:

```
yourstring$ = "But.oh!,.If.we.call.the.whole.thing.off"
```

We now have a problem. The current "yourstring$" is only 16 bytes long, but the new one is 39 bytes long. What does BASIC do? It (1) deallocates the space for the old "yourstring", (2) allocates new space for the new string and (3) updates the string

---

1. This is an outline of what BASIC does, but it will not include the parts of memory management that you will never see.

descriptor. Memory will now look like this:

```
    17150
    |You.say.either               Let's.call.it.offBut|
    |.oh!,.If.we.call.the.whole.thing.off|
```

and the descriptors will now look like this:

```
    STRING         DESCRIPTOR
                 length:location

    mystring$      14:17150       ->        |You.say.either|
    yourstring$    39:17197       ->        |But.oh!,.if.we...
    ourstring$     17:17180       ->        |Let's.call.it.off|
```

BASIC is aware that there is an empty block of space and has a strategy for dealing with empty spaces, though each BASIC has its own strategy. We don't know exactly WHEN it will take action, but we do know WHAT action it will take. At some point BASIC will decide that it has too many empty spaces in memory and will REORGANIZE the segment. This is known as GARBAGE COLLECTION. Exactly how this is done is up to the person who wrote the BASIC compiler/interpreter.

After reorganization, the addresses of ALMOST ALL strings and MANY dynamic arrays will have changed. The string locations themselves will have changed, but the string descriptors will still be in the same place in DS, and they will have been updated. Here is the new memory:

```
                          12724
                          |You.say.eitherLet's.call.i|
     |t.offBut.oh!,.If.we.call.the.whole.thing.off|
```

and here are the updated descriptors:

```
    STRING         DESCRIPTOR
                 length:location

    mystring$      14:12724       ->        |You.say.either|
    yourstring$    39:12755       ->        |But.oh!,.if.we...
    ourstring$     17:12738       ->        |Let's.call.it.off|
```

The strings have been moved several thousand bytes from where they were just a second ago. The information that was in the string descriptors a second ago is no longer valid. Old information about dynamic arrays is also unreliable. This means that if you have a subroutine written in assembler, you must get any address information at the time the subroutine is called. We'll come back to this later.

Let's go on to data input and output. When you first started

doing BASIC, you did i/o using only:

```
WRITE #1, my.data!
```

Perhaps you you do it differently now, perhaps not. In any case, you need to know about i/o speed and how different file i/o works. Here's the simplest file output:

```
**********************************
DIM   large.array! (10000)

FOR i% = 1 to 10000
     large.array! (i%) = 2.1
NEXT i%

OPEN "2-1.doc" for output as # 1
PRINT time$
FOR i% = 1 to 10000
     WRITE #1, large.array! (i%)
NEXT i%
PRINT time$
CLOSE #1
**********************************
```

Of course, to make it a challenge we are going to write an array of 10,000 numbers. How long does it take?{2} For this output it took 38 seconds. The same program, inputting the same data with:

```
INPUT #1, large.array(i%)
```

took 49 seconds. These are fairly large amounts of time. But wait, it gets worse. Let's change one line of the above program:

```
large.array! (i%) = 2.1678319E+19
```

This is a different constant which is put into each element of the array. How long does output take now? 59 seconds. And input? a whopping 79 seconds! What's going on here?

When you do i/o with INPUT #, WRITE # or PRINT #, it is exactly like doing i/o to the screen. For output, BASIC converts the binary numbers into TEXT and then writes the TEXT to the disk. When it does input, it reads the TEXT from the disk and converts the TEXT into a binary number. Here is the beginning of the output file from the first example:

```
2.1
2.1
2.1
2.1
```

---

2. All times from now on are with a slower PC with a slower hard disk, but an 8087. Since these are floating point numbers, your results should be slower if you don't have an 80x87, while if you have an 80386 with an 80387 and a fast hard disk, your times will be much faster.

```
    2.1
```

Each data item has been converted into "2.1" + CHR$(13) + CHR$(10). These last two things are a carriage return on the IBM PC. That's (5 bytes X 10000 items) plus 1 byte for the end of file marker, or 50001 bytes:

```
    2-1      DOC    50001   6-29-90  12:39p
```

Here's the beginning of the output file from the second example:

```
    2.167832E+19
    2.167832E+19
    2.167832E+19
    2.167832E+19
    2.167832E+19
```

Each data item has been converted into "2.167832E+19" + CHR$(13) + CHR$(10). That's (14 bytes * 10000 items) plus 1 byte for the end of file marker, or 140001 bytes:

```
    2-1E19   DOC   140001   6-29-90  12:47p
```

These files are unnecessarily large, and i/o is slow: if you don't have an 8087 and you are doing floating-point i/o, it can be slower still.

Can we do it faster? Yes. Using GET and PUT, we get a certain number of bytes from the disk, then transfer them to the array. Some of you have never used random access i/o, so this is a brief summary.

When you open a file as text (as we did in the above examples), BASIC divides the text by looking for carriage returns. When you open a file as a random access file, you are telling BASIC that you want to divide the file into distinct blocks of information. It may be text or it may be something else - BASIC doesn't care. If you say nothing, BASIC assumes that you want the blocks to be 128 bytes long, but the length can be anything.

In the example that we will do, we will use 1024 byte blocks because that is exactly 2 disk sectors long, so the disk can read information easily and efficiently. If we had a block length of 4 bytes, the disk would have to do 10000 disk writes; that would be very slow and be hard on the disk. Here's how we open the file:

```
    OPEN "packed.doc" for RANDOM as #1 LEN = 1024
```

This will be a random access file and the block length will be 1024 bytes. When you tell it to read or write, it will do it 1024 bytes at a time. That is getting faster.

Where is the block of data that it is going to write to disk? Here life starts getting complicated, so I hope you have understood everything that we have done so far. When you open a file, BASIC assigns it a BUFFER. The buffer has a fixed length (either 128 bytes or the length you have designated), and is

located somewhere in the DS data segment along with the numbers and strings. Like a string, it is relocatable. We need a way to pin it down. The easy and nice way would be if it were an array and we cound address it like an array:

```
buffer#1 (45) = 20
```

We are not that lucky. The only thing you can do is overlay a template on the buffer, and work from the template. This template MUST be made up of strings. We make up the template with a FIELD statement.

```
FIELD #1, 1024 AS out.string$
```

The FIELD statement starts out with the file # followed by a list of strings and the length of each string.

```
FIELD #1, 100 AS string1$, 200 AS string2$, 300 AS string3$
```

The total length of the strings may be shorter than the buffer, but may not be longer than the buffer. What does the FIELD statement do? The first thing that it does is set the string descriptor for all of these strings. Let's say that at the moment file #1 buffer is at 46217:

STRING	DESCRIPTOR length:location
string1$	100:46217
string2$	200:46317
string3$	300:46517

The first string starts at the first byte of the buffer. The second string starts right where the first string ends and the third string starts right where the second string ends. This is true for any FIELD statement, no matter how many strings are defined. Because of the way BASIC does memory management, if it moves the buffer, it will also update these string descriptors to point to the same relative places in the buffer. These string descriptors are on auto pilot.

Suppose now that we have the following string:

```
"Let's get physical"
```

and we want to write it to disk as string1$. All we need to do is:

```
string1$ = "Let's get physical"
```

Right? No, that's very, very, very wrong. What you have just done is alter the string descriptor of string1$ to point to an entirely different place in memory. The string descriptors are now:

string1$	18:58902
string2$	200:46317

```
    string3$       300:46517
```

BASIC deallocated the space for string1, reallocated it somewhere else in memory, and changed the file descriptor. Not only is string1 in a different place in memory, but BASIC may think that part of the file #1 buffer is actually empty space, and the next time it reorganizes memory, who knows what is going to happen. From the moment you define strings in a FIELD statement until the time you close the corresponding file, you can NEVER have them on the left side of an equal sign. Having them on the left side is sure to change the file descriptor.

How are we going to transfer data to these strings? There are three special operators in BASIC - LSET, MID$ and RSET. Their job is to put something into a string without altering the string length or location (i.e. without altering the string descriptor).

```
    LSET string1$ = "Let's get physical"
    MID$ (string1$,17) = "Let's get physical"
    RSET string1$ = "Let's get physical"
```

LSET will insert the string at the very left of string1, RSET will insert the string at the very right of string1, and MID$ will insert the string starting at the 17th byte of string1.

This is the strategy for all random access i/o in BASIC. We:

1) open a file as RANDOM and declare a block size.
2) define some "fixed length" strings inside the buffer with a FIELD statement.
3) insert data in the strings using LSET, RSET or MID$. This is true whether the data is strings or numbers.

There's only one problem left. For LSET, RSET and MID$, the thing on the RIGHT side of the equal sign must be a string. You can't have:

```
    LSET string1$ = number!
```

It's illegal. To counter this, BASIC has some pseudo-functions. Let's take integers as an example:

```
    a.string$ = MKI$ (number%)
    number% = CVI (a.string$)
```

MKI$ doesn't actually DO anything. It just tells BASIC that it is o.k. to move two bytes from "number%" to "a.string$". The bytes are binary data and are moved unaltered. Similarly, CVI tells BASIC that it is alright to move two bytes of binary data from "a.string$" to "number%". We are tricking BASIC into moving binary data from one data type to another. This is simply data movement, and there is no data conversion. The forms are:

NUMERIC DATA MOVE	TO STRING	FROM STRING
integer <-> string	MKI$	CVI
long integer <->string	MKL$	CVL

single precision <-> string	MKS$	CVS
double precision <-> string	MKD$	CVD

In contrast, the functions STR$ and VAL convert text representations to binary representations and binary representations to text representations. This is the same as what happens with PRINT and INPUT. Here's a program:

```
********************************************
number! = 2.1678319E+19
binary.string$ = MKS$ (number!)
text.string$ = STR$ (number!)
PRINT  LEN(text.string$), LEN(binary.string$)
PRINT  text.string$, binary.string$
********************************************
```

and here's the output:

```
 13            4
 2.167832E+19 nl _
```

You probably won't be able to see all of that last output on your printer because it is four bytes long and the number is:

```
6E6C965F hex or   110, 108, 150, 95 decimal
```

The third byte is outside of ASCII 33-127, the standard ASCII characters.

STR$ gives us the text representation of the number, while MKS$ stuffs the binary representation of a number into a string. In the opposite direction, VAL gives us the numeric value of a text string (if it has a numeric representation), while CVS stuffs 4 binary bytes from a string into a single precision number.

STR$	from binary value to text representation
VAL	from text representation to binary value

Note that STR$ can convert ANY type of number to a text string and VAL can convert a text string to ANY type of number, while CVI, CVL, CVS, CVD, MKI$, MKL$, MKS$, and MKD$ can only stuff a specific type of number into a string or a string into a specific type of number.

We want our output program to stuff the binary value from a single precision number to selected bytes of a string. To stuff a floating-point number into string1$ above, all we need to do is:

```
LSET string1 = MKS$ ( number!)
```

The following program has a single string which is the size of the entire buffer, and we are going to stuff the single precision numbers in one at a time with MID$.

```
********************************************************
number% = 10240
DIM   large.array! (number%)
```

```
FOR i% = 1 to 10240
     large.array! (i%) = 2.1678319e+19
NEXT i%

OPEN "packed.doc" for RANDOM as #1 LEN = 1024
FIELD #1 , 1024 AS out.string$

PRINT time$
k% = 0
record.count% = 0
FOR i% = 1 to 40
     record.count% = record.count% + 1
     spot% = 1
     FOR j% = 1 to 256
        k% = k% + 1
        MID$ (out.string$,spot%,4) = MKS$ (large.array!(k%))
        spot% = spot% + 4
     NEXT j%
     PUT #1, record.count%
NEXT i%
PRINT time$
CLOSE #1
*********************************************************
```

The array length has been increased slightly so that we have an exact number of blocks. We use MID$ to make sure that the string descriptor for out.string$ does not get changed. Each file write will be (256 numbers * 4 bytes/number) 1024 bytes long. We start with the first record and increase the record number by 1 each time we write. Does this increase the speed any? Well, this takes 11 seconds.

TYPE	OUTPUT	INPUT
num <-> text	38 - 59 sec	49 - 79 secs
num <-> bin. string	11 sec	11 sec

I didn't show you the equivalent input routines but here are the times they took. Note that the complexity of the single precision number has no effect on the last (the binary) routine. Also, the last routine does not suffer if there is no 8087. If you are running an 80286 with a fast hard disk, this last routine should only take a second or two. Here are the file sizes:

```
2-1      DOC    50001   6-29-90  12:39p
2-1E19   DOC   140001   6-29-90  12:47p
PACKED   DOC    40960   6-29-90   1:08p
```

The first two are the different sizes depending on whether the constant was 2.1 or 2.1678319E+19. The last one is for our last routine. Notice that it is more compact.

Can we do any better than 11 seconds? Yes, but we need to take over disk i/o and we need to know a few more things before we do that.

LOCATION OF DATA

BASIC is designed to pass subroutines the location of the data, not the data itself. This is called passing by reference. Though it is possible to pass the data itself, there are certain problems with the stack if you do.{3} We will always pass the addresses.

All single numeric variables are in the DS segment. BASIC passes the offset address of these variables in DS (1 word).

All strings are in the DS segment. Their string descriptors are also in the DS segment. BASIC always passes the offset address of the STRING DESCRIPTOR. This, in fact, is what we want. We need to know both where the string is and how long it is. If we write past the end of the string we may destroy BASIC's memory management system.

Static arrays are in the DS segment but dynamic arrays can be anywhere. If we want to write a general purpose routine with arrays, we need to handle them no matter where they are.

BASIC has a special function called VARPTR that tells you where a variable is in memory. Here's a program that uses it for a couple of variables:

```
**********************************************************
' check out the use of varptr
n% = 5000
p% = 50
DIM  b!(800),a!(900)
DIM  d!(n%), c!(p%)

mystring$ = "What's up, doc?"
addressA! = varptr (n%)
addressB! = varptr (p%)
address1! = varptr (a!(0))
address2! = varptr (b!(0))
address3! = varptr (c!(0))
address4! = varptr (d!(0))
address5! = varptr (mystring$)
PRINT addressA!, addressB!
PRINT address1!, address2!, address3!, address4!, address5!
**********************************************************
```

It gives us the addresses of all sorts of things. a!() and b!() are static arrays, so they should be in the DS segment. c!() and d!() are dynamic arrays, so they might be anywhere. Remember, the DS segment is from offset 0 to offset 65535. Let's see where they

---

3. If you make a mistake and pass a single precision number instead of an integer, you will pass 4 bytes instead of 2. From that moment on the stack will have 2 extra bytes on it and you won't know where they came from.

are:

```
     6230         6232
     9438         6234         87616        67584        13062
```

The individual numbers are in DS, and the two static arrays are in DS, but c!() and d!() are outside of DS. These numbers tell us the address relative to the start of DS, but we don't know where DS is at the moment. Where exactly are these arrays? It would be tedious to pass the subroutine these numbers because they are floating-point numbers and would be very difficult to deal with.

QuickBASIC has a function called PTR86. It is in an external object file called INT86.OBJ.{4} This object file has the routines that you need if you want to do interrupts from BASIC itself. We'll come back to that. PTR86's job is to take the floating-point number which we got from VARPTR, add the starting address of the DS segment to get an absolute address in memory, and then calculate both a segment and an offset for that address in memory. The segment will always be the highest segment that contains the first byte of the variable or array and the offset will always be a number from 0 to 15.

In order to use an object file from inside of QuickBASIC you need to put it in a library file and then load the library file when starting QuickBASIC.

Building the library file is quite easy. QuickBASIC comes with a program called BUILDLIB.EXE which builds the library for you. For now, you need only INT86.OBJ and PREFIX.OBJ in your library.{5} Put these two things in every library that you build from now on. PREFIX.OBJ insures proper segment ordering in the executable file.

```
    >buildlib  int86+prefix
```

This will create a library with the default name USERLIB.EXE. To load a library with this default file name, just put '/l' on the command line:

```
    >qb  /l
```

If you have given the library a different name like XQRTYF.EXE, then put that name after the '/l':

```
    >qb /lXQRTYF.EXE
```

These object files will now be loaded and their subroutines will be usable from inside BASIC.

---

4. PTR86 has been replaced by VARSEG in QuickBASIC 4.0.

5. Both of these object files come with your QuickBASIC.

If you now load the user library along with BASIC, you can look at the segments and offsets of the arrays. Here is a program with a couple of arrays:

```
*****************************************
k% = 12000
DIM   array1!(k%), array2!(k%), array3!(12000)

value1! = VARPTR (array1!(0))
value2! = VARPTR (array2!(0))
value3! = VARPTR (array3!(0))
PRINT value1!, value2!, value3!
*****************************************
```

Each array is 12001 * 4 (bytes) or 48004 bytes long. The first two have been defined as dynamic, so they can be anywhere in memory as long as they're after the start of DS. The third one is static, so it must be entirely in DS. Here's the output:

```
 68032         116064        6252
```

Remember, these offsets are not relative to the start of memory, they are relative to the start of the DS segment. The DS segment only goes up to offset 65535 so the first two are outside of DS while the third one is totally inside (6252 + 48004 = 54006 which is less than 65536). PTR86 is going to give us a SEGMENT:OFFSET pair relative to the start of memory.{1} The form for the call is:

```
CALL  PTR86 (segment%, offset%, value!)
```

where value! is the number returned by VARPTR.

We'll add the following code to the bottom of the above program:

```
*************************************
CALL  PTR86 (  seg1% , off1%, value1! )
CALL  PTR86 (  seg2% , off2%, value2! )
CALL  PTR86 (  seg3% , off3%, value3! )

PRINT  seg1%, off1%
PRINT  seg2%, off2%
PRINT  seg3%, off3%
*************************************
```

What does the program print now?

```
 68032         116064        6252
```

1. If you are using QuickBasic 4.0 or later this has become simplified. You can just use VARSEG and VARPTR. These will give you a segment offset pair which is usable in a subroutine call.

```
 19125         0
 22127         0
 15263         12
```

PTR86 has calculated the segments. The first two offsets are 0 and the third one is 12. Even though the third array is in the DS segment, PTR86 has recalculated the segment to find the highest segment which contains the first byte of the data. If you want to do a little calculation, you can figure out that while this program was running, DS was set to segment 14873.

VARPTR and PTR86 can be used to do this calculation for any array element, not just array(0). Here's VARPTR:

```
value1! = VARPTR (array1!(0))
value2! = VARPTR (array1!(196))
value3! = VARPTR (array1!(2781))
PRINT value1!, value2!, value3!
```

And here's the output:

```
68032         68816         79156
```

From now on, we will have VARPTR inside of the PTR86 call:

```
CALL  PTR86 (  seg1% , off1%, VARPTR (array1!(0)) )
CALL  PTR86 (  seg2% , off2%, VARPTR (array2!(0)) )
CALL  PTR86 (  seg3% , off3%, VARPTR (array3!(0)) )
```

It is clearer and uses less space. Of course, we can use this for any element in the array, not just the beginning of the array:

```
CALL  PTR86 (  seg1% , off1%, VARPTR (array1!(0)) )
CALL  PTR86 (  seg2% , off2%, VARPTR (array1!(5076)) )
CALL  PTR86 (  seg3% , off3%, VARPTR (array1!(1983)) )
```

We now have all the ammunition we need to do a quicker disk write. We can pass the offset address of any numeric data in the DS segment, we can pass the string descriptor address of any string, and we can pass the SEGMENT:OFFSET pair of any array anywhere.

Before doing our disk write program, I need to say something about subroutine calls. You notice that when I show a subroutine call I am always showing what numeric type I am passing. There is a reason for this. Let's step back from assembler for a minute and do a BASIC program with a subroutine. Here's the program:

```
**********************************************************
floatA! = 27.925
floatB! = 16.96
integerA% = 300
integerB% = 140
```

```
    CALL CheckTheNumbers (integerA%, integerB%, floatA!, floatB!)
    PRINT floatA!, floatB!, integerA%, integerB%
    END

    SUB  CheckTheNumbers ( int1%, int2%, flt1!, flt2!) STATIC
            int1% = int1% + 7
            int2% = int2% * 45
            flt1! = flt1! + 19.0
            flt2! = flt2! * 43.0
    END SUB
    ************************************************************
```

This gives the following output:

```
    46.925        729.28        307           6300
```

This is nothing earthshaking. Now let's change one line in the program:

```
    CALL CheckTheNumbers (floatA!, floatB!, integerA%, integerB%)
```

In the call statement I have put single precision numbers where the integers were and integers where the single precision numbers were. Now let's look at the output:

```
    Overflow.
    ENTER to debug, SPACEBAR to edit
```

We had an overflow. Here's where the debugger said the error was:

```
              int2% = int2% * 45
```

But int2% received the floating point number for floatB!, and that is 16.96. Since 16.96 * 45 = 763.2, where was the overflow? What the subroutine saw was not 16.96 but the first two binary bytes of floatB! (since we passed the address of floatB!). It thought these were an integer, performed a multiplication and got an error because the result was too big for an integer. Now, change the value in floatB! to:

```
    floatB! = 1.696E-29
```

and run the program again. Your results should be:

```
     27.92501      1.693756E-29  0             16792
```

These numbers have no relation to either what we started out with or what we did. Why? Because the subroutine mixed binary information from single precision numbers with binary information from integers and came up with unmitigated garbage. It was not only doing that, it was also writing 4 bytes of information into a 2 byte integer. Both flt1! and flt2! were writing past the end of the data and overwriting something else.

There is NEVER any checking between a subroutine and the calling program to see that the correct numeric types are being passed (integers, long integers, single precision, double precision). In

C and Pascal this checking is done by the compiler at compile time and the compiler will howl if you try to do something like this. In BASIC (my BASIC at least), this checking is not being done. Therefore, it is IMPERATIVE that you make sure that you pass the correct data types.

Having given that warning, we are going to build an assembler subprogram that opens a file for writing, then writes a block of data from memory to disk. The form of the call will be:

```
CALL BlockToDisk ("filename$"+CHR$(0), seg%, offset%, # of bytes)
```

Notice that there MUST be a 0 after the filename. When you use this function, you must always have:

```
    filename$ = "my.file.name" + CHR$(0)
```

The disk interrupt that is used in this subroutine expects a 'C' string (terminated by a number 0, not an ASCII character '0'). If you don't do it, you will almost certainly get an error.

This is followed by the segment of the first byte of data, the offset of the first byte of data, and the number of BYTES (not array elements) to write.

Which way does BASIC load the arguments to a subroutine? From left to right, just like PASCAL. Also, in BASIC, ALL subroutine calls are far calls. Therefore, BASIC will do the following when it calls BlockToDisk:

```
        PUSH address of file_descriptor
        PUSH address of block segment
        PUSH address of block offset
        PUSH address of length
        CALL FAR PTR BlockToDisk
```

There are two things to notice here. First, these are all ADDRESSES of the data, not the data items themselves. Upon entry to the subroutine and initialization of BP, the stack will look like this:

```
        address of file_descriptor    bp+12
        address of block segment      bp+10
        address of block offset       bp+8
        address of length             bp+6
        old CS                        bp+4
        old IP                        bp+2
   BP-> old BP                        bp
```

Secondly, the name of the subroutine does not have any periods. BASIC allows periods '.' but does not allow underscores '_' while assembler allows underscores but doesn't allow periods. (Periods have a special meaning in assembler; they are used in structures).

In order to go on from here, you need a book about interrupts. This information is from "DOS Programmer's Reference" by Terry

Dettmann, but if you have "The Peter Norton Programmer's Guide to The IBM PC", that's fine too. I'm going to give only partial information about these interrupts and you should have complete information.

Here's the program. The explaination will come afterwards.

```
*****************************************************
; BASOUT.ASM
include \pushregs.mac
PUBLIC  BLOCKTODISK
DGROUP  GROUP _DATA
; - - - - - - - - - - - - - - - - - - - - -
_DATA  SEGMENT PUBLIC 'DATA'
file_handle     dw      ?
error_message   db      "Disk i/o error #"
error_byte      db      "   ", 13, 10, "$"
_DATA  ENDS
; - - - - - - - - - - - - - - - - - - - - -
_TEXT  SEGMENT 'CODE'
        ASSUME cs:_TEXT, ds:DGROUP
; - - - - - - - - - - - - - - - - - - - - -
print_error  proc far
        mov     ah, 9           ; print error message
        mov     dx, offset DGROUP:error_message
        int     21h
        ret
print_error  endp
; - - - - - - - - - - - - - - - - - - - - - - -
; BlockToDisk ( filename , array SEG, array OFF, # of bytes)
; this is for BASIC

        DESCRIPTOR_LOCATION  EQU    [bp + 12]
        SEGMENT_LOCATION     EQU    [bp + 10]
        OFFSET_LOCATION      EQU    [bp + 8]
        LENGTH_LOCATION      EQU    [bp + 6]

BLOCKTODISK proc far
        push    bp
        mov     bp, sp
        PUSHREGS    ax, bx, cx, dx, si, ds

        ; open a new file or truncate an old one
        mov     si, DESCRIPTOR_LOCATION
        mov     ah, 3Ch         ; open new or truncate old
        mov     cx, 0           ; normal file attribute
        mov     dx, [si+2]      ; [si] =length, [si+2] =location
        int     21h
        jnc     write_the_file  ; ok if CF=0, error if CF=1

        mov     error_byte, '1' ; cannot open
        call    print_error
        jmp     exit

write_the_file:
        mov     file_handle, ax ; store handle for later use
        mov     bx, ax          ; file_handle to bx
```

```
        mov     ah, 40h         ; int 21h ah = 40h, write block
        mov     si, LENGTH_LOCATION
        mov     cx, [si]        ; # of bytes into CX
        push    ds              ; save BASIC's DS
        mov     si, OFFSET_LOCATION
        mov     dx, [si]        ; offset to DX
        mov     si, SEGMENT_LOCATION
        mov     ds, [si]        ; segment to DS
        int     21h
        pop     ds              ; restore BASIC's DS
        jnc     normal_exit     ; ok if CF=0, error if CF=1
        mov     error_byte, '2' ; bad file write
        call    print_error

normal_exit:
        mov     ah, 3Eh         ; close the file
        mov     bx, file_handle
        int     21h

exit:
        POPREGS    ax, bx, cx, dx, si, ds
        mov     sp, bp
        pop     bp
        ret     (8)      ; pop 4 words (8 bytes off the stack)

BLOCKTODISK  endp
; - - - - - - - - - - - - - - - - - - - - - - - - - -
_TEXT  ENDS
END
************************************************************
```

This is using the standardized segment names so the data will be in the DS segment. Notice the DGROUP GROUP declaration. Also notice that in the 'print_error' subroutine, we have done an offset override with 'offset DGROUP:error_message'. You need to do this every time to avoid errors with the offset addressing whenever you are using the DGROUP GROUP directive. Go back to the discussion of simplified segment directives if you don't remember this.

The main program has 3 interrupts. The first one opens a new file or truncates an old one to zero length. The file will be usable for reading and/or writing:

```
    Int  21h   function 3Ch
    AH = 3Ch
    CX = 0     0 indicates a normal file
    DS:DX      address of ASCII filename (terminated by 0)

    Returns:
              AX = file handle if CF = 0
         or:  AX = error code if CF = 1
```

This filename can be any legitimate pathname specification, and must be terminated by a zero. Like all the disk interrupts we will see in this chapter, if there is an error, the interrupt will set CF = 1. Otherwise it will clear CF = 0. If CF = 0 you

can go on; if CF = 1, there was an error and you need to terminate the subroutine and do some error reporting. The file handle is a number from 0 to 65535 which the operating system gives your program to uniquely identify that open file. There is no other open file in the system which has that number. Guard it carefully because it is your ONLY access to the file.

The second interrupt writes a block of data to disk.

```
    Int 21h   function 40h
    AH = 40h
    BX = file handle
    CX = number of bytes to write
    DS:DX = address of first byte of data

    Returns:
         AX = actual number of bytes written if CF = 0
         AX = error code if CF = 1
```

This too sets the carry flag if there was an error and clears it if there wasn't. It is limited to writing 65535 bytes at a time, but the largest array we can have is 65535 bytes (actually 65534 since all data types have an even number of bytes), so this is no problem.

The third interrupt closes the file.

```
    Int 21h   function 3Eh
    AH = 3Eh
    BX = File handle
```

Also, the print-error subroutine has an interrupt

```
    Int 21h   function 09h
    AH = 9
    DS:DX = first byte of string.
```

This string must be terminated by a dollar sign '$' (of all things). The message is on two lines so we can insert an error number into the middle of the message. This is a quick and dirty interrupt for string printing.

All interrupt numbers and function numbers are hex. This is standard for interrupts. If things go wierd, always check first to make sure that you have a hex number and not a decimal number.

The data has an 'ASSUME ds:DGROUP' statement.

Like Pascal, BASIC requires that the CALLED subroutine pop the arguments off the stack, so we pop 4 extra words (8 extra bytes) with:

```
    ret  (8)
```

Assemble this program and put the object file in a library with the other object files by using BUILDLIB.EXE. Now all we need is a BASIC program to use this. Here it is:

```
************************************************************
DIM   large.array! (10000)

FOR i% = 1 to 10000
      large.array! (i%) = 2.167832E+19
NEXT i%

filename$ = "blocktxt.doc" + CHR$ (0)
length% = 40000 - 65536
PRINT time$
CALL PTR86 (segment%, offset%, VARPTR (large.array!(1)) )
CALL BlockToDisk ( filename$ , segment% , offset%, length% )
PRINT
PRINT time$
************************************************************
```

There is an extra PRINT statement there which I will explain later. We are starting at large.array!(1) because that is where we started with the other programs. why are we subtracting 65536? Because BASIC has a limit of -32768 to +32767, so we store 40000 as its modular equivalent (mod 65536).

How long does the disk write take? From 2 to 3 seconds, and much of that time was spent opening and truncating the file. This is significantly better than the other ways of doing i/o. In fact, the limits of this routine are the limits of your system. It is literally impossible to do disk i/o any faster than this.

Try using a filename that doesn't have a CHR$(0) at the end. You should get an error message. In my BASIC, here is the output:

```
19:50:23
Disk i/o error #1
19:50:23
```

Now remove that lone PRINT statement (the next to the last line). Here's my output:

```
19:50:00
D9:50:00 error #1
1
```

For the QuickBASIC 3.0 environment, BASIC thinks that it has complete control of screen i/o, so it is not doing its i/o in a standard way and is overwriting the error message. If you are going to do any screen i/o from assembler, you will have to think of a way to live in harmony with BASIC.{2} We simply trick BASIC into writing an empty line where the message was. This may not always work correctly, especially if the window is scrolling up.

---

2. The easiest way to do this is to save the whole screen image and cursor location, do what you want using the whole screen, and then restore the screen and the cursor before returning.

This whole program only involved interrupts. There is nothing intrinsically assembler-like in its capabilities. In fact, we'll do its disk read counterpart entirely in BASIC.

Let's do something that requires assembler language. The BASIC program:

```
    ****************************************
    FOR i% = 1 to 10000
         to.array!(i%) = from.array!(i%)
    NEXT i%
    ****************************************
```

get's the job done, but is isn't all that fast. It requires about 5.5 seconds. This is a natural for assembler. Dive down into the assembler level, move the string, and come back up for air. Our BASIC program will be:

```
    ****************************************
    n% = 10000
    DIM from.array! (n%), to.array! (n%)

    PRINT time$
    FOR i% = 1 to 10000
         to.array!(i%) = from.array!(i%)
    NEXT i%
    PRINT time$
   FOR j% = 1 to 50
    cnt% = 40000 - 65536     'count
    CALL PTR86 (from.seg%, from.off%, VARPTR( from.array!(1)) )
    CALL PTR86 (to.seg%, to.off%, VARPTR ( to.array!(1)) )
    CALL BlockMove(from.seg%, from.off%, to.seg%, to.off%, cnt%)
   NEXT j%
    PRINT time$
    ****************************************
```

We are doing 50 repeats of the bottom section of code so you will be able to average the time. Here's the assembler program:

```
; - - - - - - - - - -
include /pushregs.mac
_TEXT SEGMENT PUBLIC 'CODE'
        ASSUME cs:_TEXT
        PUBLIC BlockMove
; - - - - - - - - - -
; BlockMove ( from.seg, from.off, to.seg, to.off, byte.count)
; for BASIC
; MOVSW is from DS:[SI] to ES:[DI]

        FROM_SEG_ADDRESS        EQU     [bp+14]
        FROM_OFFSET_ADDRESS     EQU     [bp+12]
        TO_SEG_ADDRESS          EQU     [bp+10]
        TO_OFFSET_ADDRESS       EQU     [bp+8]
        BYTE_COUNT_ADDRESS      EQU     [bp+6]
; - - - - - - - - - -
BlockMove proc far
        push    bp
        mov     bp, sp
```

```
        PUSHREGS  ax, bx, cx, dx, si, di, es, ds

        mov     si, TO_SEG_ADDRESS
        mov     es, [si]        ; to_seg to ES
        mov     si, TO_OFFSET_ADDRESS
        mov     di, [si]        ; to_offset to DI
        mov     si, BYTE_COUNT_ADDRESS
        mov     cx, [si]        ; byte count to CX
        mov     si, FROM_SEG_ADDRESS
        mov     ax, [si]        ; temporary storage for new DS
        mov     si, FROM_OFFSET_ADDRESS
        mov     si, [si]        ; from_offset to SI
        mov     ds, ax          ; now move from_seg to DS
        sub     bx, bx          ; clear BX
        shr     cx, 1           ; divide by 2, remainder in CF
        rcl     bx, 1           ; move CF to low bit of BX
        cld                     ; clear DF (go up)
        rep     movsw           ; the block move (count in CX)
        and     bx, bx          ; one extra byte?
        jz      exit
        movsb                   ; move one last byte

exit:
        POPREGS  ax, bx, cx, dx, si, di, es, ds
        mov     sp, bp
        pop     bp
        ret     (10)

BlockMove endp
; - - - - - - - - - -
_TEXT ENDS
END
; - - - - - - - - - -
```

This is a string block move using MOVSW. The count is the number of BYTES, not the number of array elements. CX contains the byte count. It is divided by 2 so we can move words, and if there is a remainder (i.e. if the number was odd), BX is set to 1. We move words instead of bytes, and afterwards we check BX to see if we need to move 1 byte more. This routine takes about 1/8 second instead of 5.5 seconds. This is a considerable savings in time. There is a small problem, however. If the FROM block and the TO block overlap (e.g. move 400 bytes from array!(11) to array!(26)), then the data may be compromised. To be exact, if the start of the FROM data is below the start of the TO data, the data will be screwed up. The general solution of this for BASIC is in BLKMOVE.ASM, which is in a file called MISHMASH.DOC which is in \XTRAFILE.

Finally, here's the disk read done entirely in BASIC. Once again you need your DOS interrupt book.

```
    ******************************************************
    ' READBLK.BAS
    ' reads a block from the disk into memory

    DIM in.regs%(9), out.regs%(9)
```

```
    DIM big.array! (10000)

    AX% = 0
    BX% = 1
    CX% = 2
    DX% = 3
    BP% = 4
    SI% = 5
    DI% = 6
    FLGS% = 7
    DS% = 8
    ES% = 9

    filename$ = "blocktxt.doc" + CHR$(0)

    PRINT time$
    ' open an existing file for reading
    in.regs%(AX%) = &H3D00
    in.regs%(DX%) = SADD (filename$)
    CALL INT86 (&H21,VARPTR (in.regs%(0)),VARPTR (out.regs%(0)))

    IF  (out.regs%(FLGS%) AND &H0001)  <> 0  THEN
         PRINT  "Can't open the file."
         GOTO ExitProgram
    END IF

    ' set the i/o pointer to 0
    file.handle% = out.regs%(AX%)
    in.regs%(AX%) = &H4200
    in.regs%(BX%) = file.handle%
    in.regs%(CX%) = 0
    in.regs%(DX%) = 0
    CALL INT86 (&H21, VARPTR(in.regs%(0)), VARPTR(out.regs%(0)))

    IF  (out.regs%(FLGS%) AND &H0001)  <> 0   THEN
         PRINT  "File pointer error"
         GOTO  CloseFile
    END IF

    in.regs%(AX%) = &H3F00
    in.regs%(BX%) = file.handle%
    in.regs%(CX%) = 40000 - 65536
    CALL PTR86 ( segment%, offset%, VARPTR (big.array!(1) ))
    in.regs%(DX%) = offset%
    in.regs%(DS%) = segment%
    CALL INT86X (&H21,VARPTR(in.regs%(0)),VARPTR(out.regs%(0)))
    IF  (out.regs%(FLGS%) AND &H0001)  <> 0  THEN
         PRINT  "Disk read error"
    END IF


CloseFile:
    in.regs%(AX%) = &H3E00
    in.regs%(BX%) = file.handle%
    CALL INT86 (&H21, VARPTR(in.regs%(0)), VARPTR(out.regs%(0)))

    PRINT time$
```

```
ExitProgram:
    END
    ******************************************************
```

This shows the use of INT86 in BASIC. We have two 10 element integer arrays (0 - 9). One is used for putting the data into the registers before the interrupt call and the other is used for getting the data out of the registers after the interrupt. At the top we have substituted variable names for the array elements they represent. This is the only way to make sense of things in BASIC. What is the difference between INT86 and INT86X? INT86X also changes ES and DS.

We need a different file opening call because the last one TRUNCATED the file. This one just opens a pre-existing file and we put 00 in AL to signal a file read:

```
    INT 21h   Function 3Dh
    AH = 3dh  ; open a pre-existing file
    AL = 0    ; file read
    DS:DX     ; pointer to a 00h terminated string

    Returns
         AX = file handle if CF = 0
         AX = error code if CF = 1
```

We use SADD to get the filename offset in DS because this is exactly what DOS wants. SADD is a function that gives the offset of a string (the string itself) relative to the DS segment. It gives no length information.

At every step along the way we check CF to make sure it is 0 and not 1. We need to make sure that the file pointer is at the beginning of the file. This is:

```
    INT 21h   Function 42h
    AH = 42h  ; move file pointer
    AL = 0    ; count from beginning of the file
    CX:DX = 0 ; 4 byte offset from beginning of file

    Returns
         DX:AX = new file-pointer location if CF = 0
         AX = error code if CF = 1
```

Then we do the block read:

```
    INT 21h   Function 3Fh
    AH = 3Fh  ; read a block from disk
    BX = file handle
    CX = byte count
    DS:DX = pointer to first byte of block

    Returns
         AX = # of bytes read (can be less than CX) if CF = 0
         AX = error code if CF = 1
```

Finally, we close the file:

```
    INT 21h   Function 3Eh
    AH = 3Eh  ; close a file
    BX = file handle

    Returns
         nothing if CF = 0 (close was successful)
         AX = error code if CF = 1
```

All you need to know about CF is that when the flags are represented as a word (2 bytes) CF = &H0001.

Is this faster than our other access methods? Our worst case before took 79 seconds and this one takes 1 second. This is certainly worth using for large disk reads. We don't need to go down to the assembler level, either.

What's the difference between this and bload? Bload requires that the file has already been stored from memory. When you use BSAVE, the binary information is written to disk, but the first seven bytes is BLOAD information. The first byte (0FDh) is a signature byte. The next six bytes are three words. (1) the segment where the data came from, (2) the offset where the data came from, and (3) the length of the data. Of course, having these seven bytes in the front makes the file incompatible with everything else in the world. It even makes it difficult to load the information into BASIC the first time, since this seven byte header is missing in the original data unless the data came from a BASIC file.

So what sorts of things are candidates for assembler subroutines? Things that are cumbersome in BASIC. If you want the top byte of a number, that's difficult. If you want to rotate the bits of a number, that's extremely hard. Shifting bits is hard. Practically everything involving unsigned numbers is problematic. For every assembler instruction, if you can't do it easily in BASIC you should make a subroutine that does it in assembler. How about one that does unsigned division? Another subroutine that you might want to make is one that returns both the quotient and the remainder from signed division. This cuts the work in half if you need both of them.

Well, use BASIC any way you want, but most of all, have fun!

SUMMARY

BASIC strings are defined by STRING DESCRIPTORS. A string descriptor is a 4 byte block that contains the LENGTH of the string and its LOCATION in the DS segment. Though you may modify individual bytes of a string from the assembler level, you may not alter the length without interfering with BASIC's memory management system.

BASIC passes all arguments by reference. That is it sends the offset address of the data instead of the data itself. The ruiles are:

1) If it is a single piece of numeric data, the offset is relative to the DS segment.

2) If it is a string, the address is the address of the STRING DESCRIPTOR which contains both the length and location of the string.

3) To reference an array, use VARPTR (array(0)) to get the offset and then PTR86 to convert this to a SEGMENT:OFFSET pair which is usable by the assembler subroutine.

4) If you pass a single array element array(x) instead of using VARPTR, BASIC will pass the location of that element, but the element might be separated from the rest of the array, so only pass an individual element if you want the element itself and not the array.{3}

SADD (stringname$)
SADD [Microsoft's string address function] is used to pass the offset address of stringname$ relative to BASIC's DS segment. It should only be used with 00h terminated strings since this gives no length information. It can, however, be used in conjunction with LEN (stringname$).

number! = VARPTR (variable)
In older BASICs, VARPTR gives the offset address of "variable" relative to the first byte of the DS segment. This variable can be anywhere in memory from DS:0000 to the end of memory, and the number returned will be a single precision number in the range 0 to 1,048,576.

VARSEG and VARPTR
In more recent BASICs, using a combination of VARSEG and

---

3. The rule here is that if the array itself is outside of the DS segment (if it is a dynamic array), BASIC will make a copy of the element inside of DS before the CALL, give you the address of the COPY, and return the copy to its appropriate place in the array after the CALL. This copy can be hundreds of thousands of bytes away from the actual array. If you want the element itself this works properly, but if you want the array, the address will be the wrong address.

VARPTR has supplanted the use of PTR86.

PTR86 (segment%, offset%, VARPTR (variable))
PTR86 [Microsoft's segmentation scheme] takes the result provided by VARPTR and adds it to DS to come up with a total address. It then converts this absolute address into a SEGMENT:OFFSET pair where the segment is the highest segment that contains the first byte of the variable from VARPTR and the offset is a number from 0 to 15 which is the offset of this variable in this segment.

RET (x)
When executing a return, all called subroutines must pop the arguments passed to them by BASIC. The number of BYTES popped is twice the number of arguments (as long as you are passing addresses and not actual data).

CALL MY_ROUTINE (arg1, arg2, arg3, etc)
Arguments are always PUSHed on the stack from left to right, so this call will:

```
PUSH address of arg1
PUSH address of arg2
PUSH address of arg3
etc.
```

in that order.

INT86 (interrupt.number%, in.reg.array%(9), out.reg.array%(9))
INT86 executes a DOS interrupt (interrupt.number%). The integers in in.reg.array% are put into the arithmetic registers before the call and the arithmetic registers are put into out.reg.array after the call. INT86X does the same thing but also changes the DS and ES segment registers. Consult your BASIC manual for the proper ordering of the registers in the array. Neither of these changes CS, SS or SP.

MISHMASH

This document contains several assembler programs. It has no page breaks and no footnotes so you can cut the programs directly out of the text with a word processor.

BLOCK MOVE

The first subroutine does a block move from one place in memory to another. It is designed so the sorce block and the target block can be overlapping. It first calculates the total address of the sorce block and the target block. If the sorce block is below the target block the move starts at the top of the source block and moves down. If the source block is above the target block the move starts at the bottom of the source block and moves up. This makes sure that overlapping data will not be clobbered.

This calculates the full 20 bit address. It was designed for BASIC; BASIC sometimes requires the full 20 bit address. For many languages, all you need to do is look at the offset addresses since segments cannot overlap. This is NOT true of something called the "HUGE" mode, where you need to calculate the full 20 bit address.

++++++++++++++++++++ << START OF PROGRAM >> ++++++++++++++++++++

```
include /pushregs.mac
_TEXT SEGMENT PUBLIC 'CODE'
        ASSUME cs:_TEXT
        PUBLIC BlockMove
; - - - - - - - - - -
; BlockMove ( from.seg, from.off, to.seg, to.off, byte.count)
; for BASIC
; MOVSW is from DS:[SI] to ES:[DI]

        FROM_SEG_ADDRESS        EQU     [bp+14]
        FROM_OFFSET_ADDRESS     EQU     [bp+12]
        TO_SEG_ADDRESS          EQU     [bp+10]
        TO_OFFSET_ADDRESS       EQU     [bp+8]
        BYTE_COUNT_ADDRESS      EQU     [bp+6]
; - - - - - - - - - -
BlockMove proc far
        push    bp
        mov     bp, sp
        PUSHREGS  ax, bx, cx, dx, si, di, es, ds

        ; AX:BX is the total FROM address
        ; DX:DI is the total TO address
        ; (FROM address > TO address) ->  upwards
        ; (FROM address < TO address) ->  downwards

        ; calculate 20 bit total address
        sub     ax, ax          ; zero AX
        mov     si, FROM_SEG_ADDRESS
        mov     bx, [si]        ; from_seg to BX
        sub     dx, dx          ; zero DX
        mov     si, TO_SEG_ADDRESS
        mov     di, [si]        ; to_seg to DI

        mov     cx, 4           ; shift 4 bytes
shift_loop:
        shl     bx, 1
```

```
        rcl     ax, 1           ; carry from BX -> AX
        shl     di, 1
        rcl     dx, 1           ; carry from DI -> DX
        loop    shift_loop

        ; AX:BX and DX:DI now contain the total address of the
        ; segment start. Now add the offsets.
        mov     si, FROM_OFFSET_ADDRESS
        add     bx, [si]
        adc     ax, 0
        mov     si, TO_OFFSET_ADDRESS
        add     di, [si]
        adc     dx, 0

        ; AX:BX and DX:DI are now the total addresses of the first
        ; byte to be moved. First compare AX and DX and go to the
        ; appropriate routine depending on which address is higher.
        ; If AX and DX are the same, then compare BX and DI and go
        ; to the appropriate routine. If BX = DI, the block is being
        ; moved onto itself, so just exit (there is no work to be done).

        cmp     ax, dx
        ja      bottom_to_top   ; FROM is higher
        jb      top_to_bottom   ; TO is higher
        cmp     bx, di
        ja      bottom_to_top   ; FROM is higher
        jb      top_to_bottom   ; TO is higher
        jmp     exit

bottom_to_top:
        mov     si, TO_SEG_ADDRESS
        mov     es, [si]        ; to_seg to ES
        mov     si, TO_OFFSET_ADDRESS
        mov     di, [si]        ; to_offset to DI
        mov     si, BYTE_COUNT_ADDRESS
        mov     cx, [si]        ; byte count to CX
        mov     si, FROM_SEG_ADDRESS
        mov     ax, [si]        ; temporary storage for new DS
        mov     si, FROM_OFFSET_ADDRESS
        mov     si, [si]        ; from_offset to SI
        mov     ds, ax          ; now move from_seg to DS
        sub     bx, bx          ; clear BX
        shr     cx, 1           ; divide by 2, remainder in CF
        rcl     bx, 1           ; move CF to low bit of BX
        cld                     ; clear DF (go up)
        rep     movsw           ; the block move (count in CX)
        and     bx, bx          ; one extra byte?
        jz      exit
        movsb                   ; move one last byte
        jmp     exit

top_to_bottom:
        mov     si, TO_SEG_ADDRESS
        mov     es, [si]        ; to_seg to ES
        mov     si, TO_OFFSET_ADDRESS
        mov     di, [si]        ; to_offset to DI
        mov     si, BYTE_COUNT_ADDRESS
        mov     cx, [si]        ; byte count to CX
        mov     si, FROM_SEG_ADDRESS
        mov     ax, [si]        ; temporary storage for new DS
        mov     si, FROM_OFFSET_ADDRESS
        mov     si, [si]        ; from_offset to SI
        mov     ds, ax          ; now move from_seg to DS
        add     si, cx          ; move to top of block
        sub     si, 2           ; we were 1 word too far
        add     di, cx          ; move to top of block
        sub     di, 2           ; we were 1 word too far
        sub     bx, bx          ; clear BX
        shr     cx, 1           ; divide by 2, remainder in CF
```

```
        rcl     bx, 1           ; move CF to low bit of BX
        std                     ; set DF (go down)
        rep     movsw           ; the block move (count in CX)
        and     bx, bx          ; one extra byte?
        jz      exit
        inc     si              ; top byte of word
        inc     di              ; top byte of word
        movsb                   ; move one last byte

exit:
        POPREGS  ax, bx, cx, dx, si, di, es, ds
        mov      sp, bp
        pop     bp
        ret     (10)

BlockMove endp
; - - - - - - - - - - -
_TEXT ENDS
END
```

++++++++++++++++++++++ << END OF PROGRAM >> ++++++++++++++++++++++

MULTIPLICATION AND DIVISION

The following are routines for multiple word multiplication and division. They are the core routines. There must be an intermediate routine which prepares the information correctly for the core routine and then calls the core routine. Among other things, these intermediate routines must:

1) deal with signed numbers. They must convert any negative numbers into positive numbers and keep track of the signs. Then they must alter the signs of the results if necessary.

2) make copies of numbers for the core routine when the core routine will destroy or alter the number during the calculation.

3) make decisions about valid results for the multiplication routines. If we multiply two numbers of length N words, then the result can be N + N words long. What do you want to do if the result is over N words long? It is your decision.

4) transfer the results back to the programs if necessary.

These are the things we did in chapter 16, and they are necessary here as well. In all the following routines you need to pay attention to the lengths. Some lengths are in BYTES and some lengths are in WORDS. Make sure you know which is which.

BLOCK MULTIPLICATION

The first multiplication program uses block multiplication. This is simply the multiple word multiplication that you did in chapters 13 and 16. This time, instead of multiplying n X 1 words, we will be multiplying n X n words. The most important thing that this routine does is minimize its work. If n = 100 words, then it is possible for the routine to do 10,000 multiplications. This takes a lot of time. If we have two 100 word numbers but the first one is 127,911 and the other one is 4,926,948,187,062 the first number has significant information in two words and the second number has significant information in three words. We only need to multiply 3 X 2 = 6 words instead of 10,000 words. As you can see, this will cut the time by a factor of over 1000. This routine requires that the result be distinct from either the multiplicand or multiplier and be n + n (2n) words long.

First we clear the area for the result. The next section finds the highest non-zero word of both the multiplicand and multiplier. If either is 0 the result is 0, so we exit (the result is cleared and is 0). After that comes the multiplication proper. We multiply the complete multiplicand by one multiplier word, then cycle to the next multiplier word and so on. We add each DX:AX pair to the temporary result and propagate any carry that results from the addition. The result cannot be larger than N + N words, so we will never propagate past the result area. This is as fast as you can multiply numbers on the 8086.

```
++++++++++++++++++++ << START OF PROGRAM >> ++++++++++++++++++++++

; block multiplication using standard 8086 multiplication
; block_multiply ( length , multiplicand, multiplier, temp_result )

; length is the number of WORDS
; length is a number, but the others are addresses. The temp_result
; space must be (2 X length), and must be distinct from the other
; varibles since it will be overwritten by the routine. This is
; a far routine for C, and after setting up BP, we have:
;
;       TEMP_RESULT_ADDRESS     EQU   [bp + 12]
;       MULTIPLIER_ADDRESS      EQU   [bp + 10]
;       MULTIPLICAND_ADDRESS    EQU   [bp + 8]
;       DATA_LENGTH             EQU   [bp + 6]


INCLUDE \pushregs.mac
; - - - - - - - - - - - - - - - - - - - - -
DATASTUFF SEGMENT PUBLIC 'DATA'
multiplicand_top_address      dw   ?
multiplier_top_address        dw   ?
temp_bottom_address           dw   ?
current_multiplier_address    dw   ?
DATASTUFF  ENDS
; - - - - - - - - - - - - - - - - - - - - -
CODESTUFF SEGMENT PUBLIC 'CODE'

     PUBLIC  block_multiply
     ASSUME CS:CODESTUFF, DS:DATASTUFF

     TEMP_RESULT_ADDRESS     EQU   [bp + 12]
     MULTIPLIER_ADDRESS      EQU   [bp + 10]
     MULTIPLICAND_ADDRESS    EQU   [bp + 8]
     DATA_LENGTH             EQU   [bp + 6]

; - - - - - - - - - - -
block_multiply proc far

     push  bp
     mov   bp, sp
     pushf               ; save DF value
     PUSHREGS ax, bx, cx, dx, si, di, es
     push  ds            ; es = ds
     pop   es

     ; clear temp_result
     mov   di, TEMP_RESULT_ADDRESS
     mov   cx, DATA_LENGTH
     shl   cx, 1         ; 2 X LENGTH is buffer length
     mov   ax, 0         ; zero for clearing
     cld                 ; upwards
     rep   stosw         ; store ax

     ; find the highest multiplicand word which is non-zero
     mov   di, MULTIPLICAND_ADDRESS
     mov   dx, DATA_LENGTH
```

```
     mov   cx, dx       ; cx = length in words
     mov   bx, dx
     dec   bx           ; first word is at offset 0
     shl   bx, 1        ; bx = top word
     add   di, bx       ; di = address of top word
     std                ; downwards
                        ; ax is still 0
     repe  scasw        ; continue as long as es:[di] is 0
     jne   first_top_found    ; found non-zero word
     jmp   exit_mult    ; multiplicand is 0 so result is 0

first_top_found:
     add   di, 2                          ; we went 2 too far
     mov   multiplicand_top_address, di   ; address of top non-zero word

     ; no registers have been modified except di and cx
     ; use the same ax, bx and dx values as before for multiplier.

     ; find the highest non-zero multiplier word
     mov   di, MULTIPLIER_ADDRESS
     add   di, bx                 ; di = address of top word
     mov   cx, dx                 ; cx = length in words
                                  ; ax is still 0
     repe  scasw                  ; continue as long as es:[di] is 0
     jne   second_top_found       ; found non-zero word
     jmp   exit_mult              ; multiplier is 0 so result is 0

second_top_found:
     add   di, 2                          ; we went 2 too far
     mov   multiplier_top_address, di     ; address of top non-zero word

     ; the multiplication  ********************
     mov   ax, TEMP_RESULT_ADDRESS
     mov   temp_bottom_address, ax        ; start at bottom
     mov   si, MULTIPLIER_ADDRESS
     mov   current_multiplier_address, si  ; save address

outer_multiplication_loop:
     ; set up the registers
     mov   cx, [si]                 ; move current multiplier to cx
     mov   di, MULTIPLICAND_ADDRESS
     mov   bx, temp_bottom_address

inner_multiplication_loop:
     mov   ax, cx                   ; multiplier word to ax
     mul   WORD PTR [di]            ; multiplicand - result in DX:AX
     add   [bx], ax                 ; low word of multiplication
     adc   [bx+2], dx               ; high word of multiplication
     jnc   no_more_carry            ; extra work if CF=1
     mov   si, 4
     ; keep propagating the carry till CF = 0
propagate_carry:
     add   WORD PTR [bx+si], 1
     jnc   no_more_carry
     add   si, 2                    ; next word
     jmp   propagate_carry

no_more_carry:
     add   bx, 2         ; next word of temp result
     add   di, 2         ; next word of multiplicand
     cmp   di, multiplicand_top_address  ; finished?
     ja    next_multiplier_word
     jmp   inner_multiplication_loop

next_multiplier_word:
     mov   si, current_multiplier_address
     add   si, 2
     cmp   si, multiplier_top_address
     ja    exit_mult               ; end of multiplication
```

```
     mov   current_multiplier_address, si  ; save address
     add   temp_bottom_address, 2   ; increment for next start
     jmp   outer_multiplication_loop

     ; end of the multiplication  *************

exit_mult:
     POPREGS ax, bx, cx, dx, si, di, es
     popf              ; restore DF value
     mov   sp, bp
     pop   bp
     ret               ; a C return, so don't pop arguments.

block_multiply  endp
; - - - - - - - - - - -
CODESTUFF ENDS
END

++++++++++++++++++++++ << END OF PROGRAM >> ++++++++++++++++++++++
```

If you understand all of this you can go on. The next one is even more difficult.

BINARY MULTIPLICTION

This is how the 8086 does multiplication internally. It is a series of shifts and additions. We can do the same thing with base 10 numbers.

```
               24763 X 275

                    24763  ---
                    24763   |
                    24763   5
                    24763   |
                    24763  ---
                   247630  ___
                   247630   |
                   247630   |
                   247630   7
                   247630   |
                   247630   |
                   247630  ---
                  2476300  ---
                  2476300   2

                 6,809,825
```

In the base 10 system this is tedious. In the base 2 system this works well. You either do NO addition or you do 1 addition. We start at the bottom and add (either once or not at all), then shift the whole number left one bit. We repeat this cycle till we are finished with the whole multiplier. Once again, the pivotal operation is finding the highest non-zero word before starting. This is about 5 times slower than the first method. The only reason that it is here is to prepare you for the binary division routine.

We need to reserve an extra word above the multiplicand. If the multiplicand is 6 words long, we need 7 words for the multiplicand. The 6th word will shift into that 7th word 1 bit at a time. At the end of our 16 bit cycle, all words will have shifted up one word.

As the multiplication progresses, the bottom words of the multiplicand will be 0 so we don't bother to add these 0 words.

We load the multiplier into DX one word at a time. We then check this word one bit at a time. If the bit is 1 we add, if the bit is 0 we do nothing. We shift the multiplicand left 1 bit each time, whether we add or not.

```
++++++++++++++++++++ << START OF PROGRAM >> ++++++++++++++++++++

; binary multiplication using shifts and addition
; binary_multiply ( length , multiplicand, multiplier, temp_result )

; length is the number of WORDS
; length is a number, but the others are addresses. The temp_result
; space and the multiplicand space must be ((2 X length)+1) WORDS,
; and must be distinct from the calling variables since they will be
; overwritten by the routine. This is a far routine for C, and after
; setting up BP, we have:

;      TEMP_RESULT_ADDRESS     EQU   [bp + 12]
;      MULTIPLIER_ADDRESS      EQU   [bp + 10]
;      MULTIPLICAND_ADDRESS    EQU   [bp + 8]
;      DATA_LENGTH             EQU   [bp + 6]


include \pushregs.mac
; - - - - - - - - - - - - - - - - - - -
DATASTUFF SEGMENT PUBLIC 'DATA'
multiplicand_length          dw     ?
multiplier_length            dw     ?
lowest_non_zero_word         dw     ?
DATASTUFF ENDS
; - - - - - - - - - - - - - - - - - - -
CODESTUFF SEGMENT PUBLIC 'CODE'

     PUBLIC binary_multiply
     ASSUME  cs:CODESTUFF, ds:DATASTUFF

     TEMP_RESULT_ADDRESS     EQU   [bp + 12]
     MULTIPLIER_ADDRESS      EQU   [bp + 10]
     MULTIPLICAND_ADDRESS    EQU   [bp + 8]
     DATA_LENGTH             EQU   [bp + 6]

; - - - - - - - - - - -
binary_multiply proc far

     push  bp
     mov   bp, sp
     pushf               ; save DF value
     PUSHREGS ax, bx, cx, dx, si, di, es
     push  ds            ; es = ds
     pop   es

     ; clear temp buffer
     mov   di, TEMP_RESULT_ADDRESS
     mov   cx, DATA_LENGTH
     shl   cx, 1         ; 2 X LENGTH is buffer length
     mov   ax, 0
     cld                 ; upwards
     rep   stosw         ; store ax

     ; find the highest word which is non-zero
     mov   di, MULTIPLICAND_ADDRESS
     mov   dx, DATA_LENGTH
     mov   cx, dx        ; cx = length in words
     mov   bx, dx
     dec   bx
     shl   bx, 1         ; bx = top word
     add   di, bx        ; di = address of top word
     std                 ; downwards
```

```
                        ; ax is still 0
     repe  scasw
     jne   first_top_found    ; found non-zero word
     jmp   exit_mult    ; multiplicand is 0 so result is 0

first_top_found:
      ; we went 2 too far + 2 for length + 2 extra for bit shift
     add   di, 6
     sub   di, MULTIPLICAND_ADDRESS
     shr   di, 1                      ; divide by 2
     mov   multiplicand_length, di   ; length in WORDS

     ; no registers have been modified except di and cx
     ; use the same ax, bx and dx values as before for multiplier.

     ; find the highest non-zero word
     mov   di, MULTIPLIER_ADDRESS
     add   di, bx         ; di = address of top word
     mov   cx, dx         ; cx = length in words
                          ; ax is still 0
     repe  scasw
     jne   second_top_found    ; found non-zero word
     jmp   exit_mult      ; multiplier is 0 so result is 0

second_top_found:
     ; we went 2 too far + 2 for length
     add   di, 4
     sub   di, MULTIPLIER_ADDRESS
     mov   multiplier_length, di        ; length in BYTES

     ; the multiplication  ********************
     mov   lowest_non_zero_word, 0

multiplicand_loop:
     mov   ax, lowest_non_zero_word      ; # of words shifted
     cmp   ax, multiplier_length         ; length in bytes
     jb    multiply_a_word
     jmp   exit_mult                     ; we are through
     ; ax still has lowest word count
multiply_a_word:
     mov   si, MULTIPLIER_ADDRESS
     add   si, ax                ; calculate where multiplier is
     mov   dx, [si]              ; this is current multiplier word
     mov   cx, 16                ; 16 adds and shifts
add_and_shift_loop:
     push  cx
     shr   dx, 1                   ; add if low bit is 1
     jnc   skip_the_addition
     mov   ax, lowest_non_zero_word     ; offset count
     mov   si, MULTIPLICAND_ADDRESS
     add   si, ax
     mov   bx, TEMP_RESULT_ADDRESS
     add   bx, ax
     mov   cx, multiplicand_length       ; length in words
     clc
inner_add_loop:
     mov   ax, [si]
     adc   [bx], ax
     inc   si               ; doesn't affect the carry flag
     inc   si
     inc   bx
     inc   bx
     loop  inner_add_loop
     adc   WORD PTR [bx], 0     ; one last carry is possible

skip_the_addition:
     ; shift one bit to the left
     mov   si, MULTIPLICAND_ADDRESS
     add   si, lowest_non_zero_word
```

```
     mov   cx, multiplicand_length         ; length in words
     clc
shift_1_loop:
     rcl   WORD PTR [si], 1
     inc   si                    ; doesn't affect carry flag
     inc   si
     loop  shift_1_loop

     pop   cx
     loop  add_and_shift_loop

     add   lowest_non_zero_word, 2          ; move up one word
     jmp   multiplicand_loop


     ; end of the multiplication  *************

exit_mult:
     POPREGS ax, bx, cx, dx, si, di, es
     popf              ; restore DF value
     mov   sp, bp
     pop   bp
     ret               ; a C return, so don't pop arguments.

binary_multiply  endp
; - - - - - - - - - - -
CODESTUFF ENDS
END
```

++++++++++++++++++++++ << END OF PROGRAM >> ++++++++++++++++++++++

BINARY DIVISION

This is by far the hardest to understand. The binary division routine is the opposite of the multiplication routine. We move the dividend to the remainder area since it will be modified during the routine. We shift the divisor one word past the top of the dividend (to make sure that the divisor starts out larger than the dividend) and then start the shift-subtract cycle. We shift right 1 bit and then take a look at the two numbers. If the divisor is larger than the dividend we do nothing and put a 0 bit in the quotient. If the divisor is smaller, we put a 1 bit in the quotient and subtract the divisor from the dividend. At the end, what is left of the dividend is our remainder

As usual, we only use only as many words as necessary, both for the numbers and the individual subtractions.

This is about 5 times slower than the block multiplication. It is possible to approach the speed of the block multiplication routine by using block division routine which guesses and then modifies its guess, but it would be almost impossible to understand what the code does, so I won't show it to you.

++++++++++++++++++++ << START OF PROGRAM >> ++++++++++++++++++++

```
; binary division using shifts and subtraction
; binary_divide ( length , dividend, divisor, quotient, remainder)

; length is the number of WORDS
; length is a number, but the others are addresses. The divisor and
; remainder space will be overwritten one word past the highest non-
; zero word by the subroutine. The remainder space is cleared one word past
; its length. This is a far routine for C, and after setting up BP, we have:

     OUR_DIVIDEND_ADDRESS EQU   [bp + 14]   ; same as remainder address
     REMAINDER_ADDRESS    EQU   [bp + 14]
```

```
     QUOTIENT_ADDRESS     EQU   [bp + 12]
     DIVISOR_ADDRESS      EQU   [bp + 10]
     DIVIDEND_ADDRESS     EQU   [bp + 8]
     DATA_LENGTH          EQU   [bp + 6]

include \pushregs.mac
; - - - - - - - - - - - - - - - - - - - - -
DATASTUFF SEGMENT PUBLIC 'DATA'

dividend_length               dw    ?
divisor_length                dw    ?
     ; - - - -
top_divisor_address           dw    ?
bottom_divisor_address        dw    ?
top_dividend_address          dw    ?
bottom_dividend_address       dw    ?
current_quotient_address      dw    ?
     ; - - - -
shift_count                   dw    ?
quotient_bit                  dw    ?



DATASTUFF ENDS
; - - - - - - - - - - - - - - - - - - - - -
CODESTUFF SEGMENT PUBLIC 'CODE'

     PUBLIC  binary_divide
     ASSUME  cs:CODESTUFF, ds:DATASTUFF

; - - - - - - - - - -
binary_divide proc far

     push bp
     mov  bp, sp
     pushf                    ; save DF value
     PUSHREGS ax, bx, cx, dx, si, di, es
     push ds              ; es = ds
     pop  es

     ; clear quotient
     mov   ax, 0                   ; zero for clearing
     mov   dx, DATA_LENGTH         ; store for later
     mov   cx, dx
     mov   di, QUOTIENT_ADDRESS
     cld                           ; upwards
     rep   stosw
     ; move dividend to remainder area
     mov   si, DIVIDEND_ADDRESS
     mov   di, REMAINDER_ADDRESS   ; our new dividend area
     mov   cx, dx                  ; DATA_LENGTH
     rep   movsw                   ; upwards
     mov   [di], ax                ; extra 0 above dividend space

     ; find the highest divisor word which is non-zero
     ; dx  still has DATA_LENGTH
     mov   bx, dx            ; dx = DATA_LENGTH
     dec   bx
     shl   bx, 1             ; bx = top word (in # of bytes)
     mov   di, DIVISOR_ADDRESS
     mov   bottom_divisor_address, di    ; save for later
     add   di, bx            ; di = address of top word
     mov   cx, dx            ; cx = length in words
     std                     ; downwards
     ; ax is still 0
     repe  scasw             ; look for nonzero
     jne   first_top_found   ; left loop because unequal?
     int   0                 ; divisor is 0 so divide error
```

```
first_top_found:
     add   di, 2                         ; we went 2 too far
     mov   top_divisor_address, di       ; store for later
     sub   di, DIVISOR_ADDRESS
     add   di, 2                         ; actual length
     mov   divisor_length, di            ; length in BYTES

     ; no registers have been modified except di and cx
     ; use the same ax, bx and dx values as before for dividend.
     ; find the highest non-zero dividend word
     ; ax is still 0 (from above)
     mov   di, OUR_DIVIDEND_ADDRESS
     mov   bottom_dividend_address, di   ; save for later
     add   di, bx      ; di = address of top word
     mov   cx, dx      ; dx = length in words
     repe  scasw       ; downwards
     jne   second_top_found        ; equal on exit?
     jmp   exit_div    ; dividend = 0 so quotient is 0, remainder is 0

second_top_found:
     ; add 2 for overshoot & 2 for calculating length
     ; top dividend address is just past top of dividend
     add   di, 4
     mov   top_dividend_address, di   ; this is correct
     sub   di, OUR_DIVIDEND_ADDRESS
     mov   dividend_length, di    ; length in BYTES

     ; if dividend length < divisor length, we are done
     cmp   di, divisor_length
     jae   shift_divisor
     jmp   exit_div

shift_divisor:
     ; figure out shift count.
     ; change divisor length from bytes to words
     ; di is still dividend length
     mov   ax, di                  ; dividend_length
     mov   dx, divisor_length
     sub   ax, dx                  ; amount of shift
     add   bottom_divisor_address, ax   ; current bottom
     add   bottom_dividend_address, ax  ; current bottom
     add   ax, 2                   ; 2 extra bytes for shift
     mov   shift_count, ax         ; save shift count
     shr   dx, 1                   ; divisor length - BYTES to WORDS
     mov   cx, dx                  ; cx is amount of data to shift
     inc   dx                      ; one word extra for shift
     mov   divisor_length, dx      ; new divisor_length  (WORDS)
     ; prepare pointers for the shift
     mov   si, top_divisor_address
     mov   di, si                  ; destination pointer
     add   di, ax                  ; add the shift
     mov   top_divisor_address, di ; new top of divisor
     rep   movsw                   ; downwards
     ; zero bottom of divisor
     mov   ax, 0
     mov   cx, shift_count
     shr   cx, 1                   ; shift count in words
     rep   stosw

     ; set up quotient info
     mov   ax, QUOTIENT_ADDRESS
     add   ax, shift_count
     sub   ax, 2                   ; address of top word
     mov   current_quotient_address, ax
     mov   quotient_bit, 0001h     ; bit to rotate

     ; ***** the division  ****************

division_loop:
```

```
     cmp   shift_count, 0    ; if 0, we are done
     ja    do_shift_16
     jmp   exit_div

do_shift_16:
     ; ++++++++++ SHIFT AND SUBTRACT LOOP ++++++
     mov   cx, 16
shift_16_loop:
     push  cx               ; save counter

     ; +++++++++  SHIFT +++++++++
     ; shift divisor one bit to the right
     ror   quotient_bit, 1
     mov   si, top_divisor_address
     mov   cx, divisor_length         ; length in words
     clc                              ; clear CF
shift_1_loop:
     rcr   WORD PTR [si], 1
     dec   si                ; doesn't affect carry flag
     dec   si
     loop shift_1_loop

     ; +++++++++ CHECK FOR SKIP SUBTRACTION +++++++
     ; skip subtraction if dividend < divisor
     mov   di, top_divisor_address
     mov   si, top_dividend_address
     mov   cx, divisor_length
     std                     ; decrement pointers
     repe  cmpsw             ; cmp dividend, divisor
     jb    skip_subtraction  ; dividend < divisor

     ; +++++++++++++++ SUBTRACTION ++++++++++++++++
     ; OR 1 into quotient
     mov   si, current_quotient_address
     mov   dx, quotient_bit
     or    [si], dx

     mov   si, bottom_divisor_address
     mov   di, bottom_dividend_address
     mov   cx, divisor_length      ; words
     clc                           ; clear CF
subtraction_loop:
     mov   dx, [si]
     sbb   [di], dx
     inc   si
     inc   si
     inc   di
     inc   di
     loop  subtraction_loop

     ; dividend >= divisor, so we have no final borrow

     ; ++++++++++ AFTER SUBTRACTION ++++++++++++++++
skip_subtraction:
     pop   cx
     loop  shift_16_loop

     ; reset the pointers and counters for the outer loop
     sub   shift_count, 2
     sub   top_divisor_address, 2
     sub   top_dividend_address, 2
     sub   bottom_divisor_address, 2
     sub   bottom_dividend_address, 2
     sub   current_quotient_address, 2
     jmp   division_loop

     ; end of the division  *************

exit_div:
```

```
     POPREGS ax, bx, cx, dx, si, di, es
     popf              ; restore DF value
     mov  sp, bp
     pop  bp
     ret            ; a C return, so don't pop arguments.

binary_divide  endp
; - - - - - - - - - -
CODESTUFF ENDS
END

++++++++++++++++++++++ << END OF PROGRAM >> ++++++++++++++++++++++
```

FILELIST

This is the list of all the files, grouped by the compressed file which they are in.

==========	DISK1-A.COM
APP1.DOC	Appendix I is a list of all ASMHELP subroutines.
ASMHELP.OBJ	A module with i/o routines for use with assembler programs.
ASMLINK.BAT	A batch file for linking modules to ASMHELP.OBJ
HELPMEM.COM	A memory resident file which can show the state of the registers using INT1 and INT3.
PUSHREGS.MAC	A file with two macros for multiple PUSHes and POPs.
SETREGS.EXE	A demonstration program for showing how to set the registers with 'set_reg_style'.
TEMPLATE.ASM	A template file for making normal .OBJ files which link to ASMHELP.OBJ. It contains EXTRN statements for ALL of the subroutines in ASMHELP.OBJ.
BBSINFO.DOC	An informational file for use if this compressed file is electronically transmitted.

==========	DISK1-B.EXE
A86.DOC	Information for those who are using the A86 assembler.
ADD1.ASM	An addition program for use in Chapter 5.
APP2.DOC	Appendix II contains a list of all 8086 instructions along with a description of what they do.
APP3.DOC	Appendix III is a list of both the speed of all 8086 instructions and how each instruction affects the flags.
CH1STR.OBJ	A file containing text for use in Chapter 19.
COMPARE.ASM	An assembler file for text comparison used in Chapter 19.
COMTEMP.ASM	A template file for making .COM files.
DEBUGGER.DOC	Discusses what debuggers do and shows the same example from CODEVIEW, TURBO DEBUGGER and D86.
DISK1MAK.BAT	A batch file which creates subdirectories for disk1 and transfers all of the unpacked files from DISK1-A.COM and DISK1-B.EXE to the disk.
FILELIST.DOC	The list of all files, what they are, and where they belong.
INTO.COM	A memory resident program which indicates when INTO has been invoked.
INTRO1.DOC	Introduction to The PC Assembler Helper and The PC Assembler Tutor.
INTRO2.DOC	A general explaination of assemblers, the assemble times of different assemblers, the Table Of Contents and the Index.
MISHMASH.DOC	This contains four subroutines. It is for people

	who have finished the Tutor. The subroutines are: a general block move, a block multiplication, a binary multiplication and a binary division.
PSEUDDBG.COM	A memory resident program which shows when either of the two debugger interrupts (INT1 or INT3) has been invoked.
README1.DOC	An identification file.
SRCHTBL.OBJ	An XLAT translation table for use in a word search program. This is from Chapter 23.
SUBTEMP1.ASM	A template file for use with subroutines. It contains the program entry point.
SUBTEMP2.ASM	A template file for use with subroutines. It doesn't contain the program entry point.
TASM.DOC	Information for those who are using the Turbo Assembler.
TEMP1.ASM	A beginning template file. It contains EXTRN statements for a few of the subroutines in ASMHELP.OBJ.
TEMP2.ASM	A beginning template file. It contains EXTRN statements for a more of the subroutines in ASMHELP.OBJ.
TRANSTBL.OBJ	A translation table for showing the characteristics of all ASCII characters 0-255. This is used in Chapter 23.
TWOTALE.OBJ	An object file containing text for use in a word search program. This is used in Chapter 23.
TWOTALE.DOC	A text file showing the contents of the previous file.

The contents of these two compressed files become DISK1. Their file structure is as follows:

```
\
    INTRO1.DOC
    INTRO2.DOC
    README1.DOC

\APPENDIX
    APP1.DOC
    APP2.DOC
    APP3.DOC

\ASMHELP
    ASMHELP.OBJ
    ASMLINK.BAT
    SETREGS.EXE

\COMMENTS
    A86.DOC
    BBSINFO.DOC
    DEBUGGER.DOC
    FILELIST.DOC
    MISHMASH.DOC
    TASM.DOC

\TEMPLATE
```

```
    COMTEMP.ASM
    PUSHREGS.MAC
    SUBTEMP1.ASM
    SUBTEMP2.ASM
    TEMP1.ASM
    TEMP2.ASM
    TEMPLATE.ASM

\XTRAFILE
    ADD1.ASM
    CH1STR.OBJ
    COMPARE.ASM
    HELPMEM.COM
    INTO.COM
    PSEUDDBG.COM
    SRCHTBL.OBJ
    TRANSTBL.OBJ
    TWOTALE.OBJ
    TWOTALE.DOC
```

The following files (except the BASIC files) are the chapters of the Tutor. There are no subdirectories on disks 2-4 so they can be unpacked directly to floppy disk. The contents of the chapters is listed in the Table of Contents in INTRO2.DOC.

```
========== DISK2.EXE
BAS1.DOC      For BASIC programmers. It should be read before
              starting to learn assembler.
BAS2-1.DOC
BAS2-2.DOC    For BASIC programmers. These two files should be
              read after finishing the Tutor. They discuss
              BASIC's memory management, disk i/o and
              interfacing with .OBJ files.
CHAP0-1.DOC
CHAP0-2.DOC
CHAP1.DOC
CHAP2.DOC
CHAP3.DOC
CHAP4.DOC
CHAP5.DOC
CHAP6.DOC
CHAP7.DOC
CHAP8.DOC
README2.DOC

========== DISK3.EXE
CHAP09.DOC
CHAP10-1.DOC
CHAP10-2.DOC
CHAP11-1.DOC
CHAP11-2.DOC
CHAP12.DOC
CHAP13.DOC
CHAP14.DOC
CHAP15-1.DOC
CHAP15-2.DOC
```

CHAP15-3.DOC
CHAP16.DOC
CHAP17.DOC
CHAP18.DOC
README3.DOC

========== DISK4.EXE
CHAP19-1.DOC
CHAP19-2.DOC
CHAP20.DOC
CHAP21.DOC
CHAP22-1.DOC
CHAP22-2.DOC
CHAP23.DOC
CHAP24.DOC
CHAP25.DOC
CHAP26.DOC
README4.DOC

========== UPDATE INFORMATION FOR TUTOR AND HELPER

November 1989 - Original release

Aug 1990 - version 1.01
Correction of errata and minor revisions. The following are new items.
HELPER
    set_timer and kill_timer
TUTOR
    Table of Contents and Index
    A86.DOC
    APP3.DOC
    BAS1.DOC, BAS2-1.DOC, BAS2-2.DOC
    CHAP25.DOC, CHAP26.DOC
    DEBUGGER.DOC
    INTRO2.DOC
    MISHMASH.DOC
    TASM.DOC