

An Assembly Language Primer

(C) 1983 by David Whitman

TABLE OF CONTENTS

Introduction.....	2
The Computer As A Bit Pattern Manipulator.....	2
Digression: A Notation System for Bit Patterns.....	4
Addressing Memory.....	6
The Contents of Memory: Data and Programs.....	7
The Dawn of Assembly Language.....	8
The 8088.....	9
Assembly Language Syntax.....	12
The Stack.....	14
Software Interrupts.....	15
Pseudo-Operations.....	17
Tutorial.....	18

INTRODUCTION

Many people requesting CHASM have indicated that they are interested in *learning* assembly language. They are beginners, and have little idea just where to start. This primer is directed to those users. Experienced users will probably find little here that they do not already know.

Being a primer, this text will not teach you everything there is to know about assembly language programming. Its purpose is to give you some of the vocabulary and general ideas which will help you on your way.

I must make a small caveat: I consider myself a relative beginner in assembly language programming. A big part of the reason for writing CHASM was to try and learn this branch of programming from the inside out. I think I've learned quite a bit, but it's quite possible that some of the ideas I relate here may have some small, or even large, flaws in them. Nonetheless, I have produced a number of working assembly language programs by following the ideas presented here.

THE COMPUTER AS A BIT PATTERN MANIPULATOR.

We all have some conception about what a computer does. On one level, it may be thought of as a machine which can execute BASIC programs. Another idea is that the computer is a number crunching device. As I write this primer, I'm using my computer as a word processor.

I'd like to introduce a more general concept of just what sort of machine a computer is: a bit pattern manipulator.

I'm certain that everyone has been introduced to the idea of a **bit**. (Note: Throughout this primer, a word enclosed in **asterisks** is to be read as if it were in italics.) A bit has two states: on and off, typically represented with the symbols "1" and "0". In this context, DON'T think of 1 and 0 as numbers. They are merely convenient shorthand labels for the state of a bit.

The memory of your computer consists of a huge collection of bits, each of which could be in either the 1 or 0 (on or off) state.

3

At the heart of your computer is a microprocessor chip, named the 8088 by Intel, who makes the chip. What this chip can do is manipulate the bits which make up the memory. The 8088 likes to handle bits in chunks, and so we'll introduce special names for the two sizes of bit chunks the 8088 is most happy with. A **byte** will refer to a collection of eight bits. A **word** consists of two bytes, or equivalently, sixteen bits.

A collection of bits holds a pattern, determined by the state of it's individual bits. Here are some typical byte long patterns:

10101010 11111111 00001111

If you've had a course in probability, it's quite easy to work out that there are 256 possible patterns that a byte could hold. Similarly, a word can hold 65,536 different patterns.

All right, now for the single most important idea in assembly language programming. Are you sitting down? These bit patterns can be used to represent other sets of things, by mapping each pattern onto a member of the other set. Doesn't sound like much, but IBM has made **BILLIONS** off this idea.

For example, by mapping the patterns a word can hold onto the set of integers, you can represent either the numbers from 0 to 65535 or -32768 to 32767, depending on the exact mapping you use. You might recognize these number ranges as the range of possible line numbers, and the possible values of an integer variable, in BASIC programs. This explains these somewhat arbitrary seeming limits: BASIC uses words of memory to hold line numbers and integer variables.

As another example, you could map the patterns a byte can hold onto a series of arbitrarily chosen little pictures which might be displayed on a video screen. If you look in appendix G of your BASIC manual, you'll notice that there are *exactly* 256 different characters that can be displayed on your screen. Your computer uses a byte of memory to tell it what character to display at each location of the video screen.

4

Without getting too far ahead of myself, I'll just casually mention that there are about 256 fundamental operations that the 8088 microprocessor chip can carry out. This suggests another mapping which we'll discuss in more detail later.

The point of this discussion is that we can use bit patterns to represent anything we want, and by manipulating the patterns in different ways, we can produce results which have significance in terms of what we're choosing to represent.

DIGRESSION: A NOTATION SYSTEM FOR BIT PATTERNS

Because of their importance, it would be nice to have a convenient way to represent the various bit patterns we'll be

talking about. We already have one way, by listing the states of the individual bits as a series of 1's and 0's. This system is somewhat clumsy, and error prone. Are the following word patterns identical or different?

1111111011111111

1111111101111111

You probably had trouble telling them apart. It's easier to tell that they're different by breaking them down into more manageable pieces, and comparing the pieces. Here are the same two patterns broken down into four bit chunks:

1111 1110 1111 1111

1111 1111 0111 1111

Some clown has given the name *nybble* to a chunk of 4 bits, presumably because 4 bits are half a byte. A nybble is fairly easy to handle. There are only 16 possible nybble long patterns, and most people can distinguish between the patterns quite easily.

Each nybble pattern has been given a unique symbol agreed upon by computer scientists. The first 10 patterns were given symbols "0" through "9", and when they ran out of digit style symbols, they used the letters "A" through "F" for the last six patterns. Below is the "nybble pattern code":

0000 = 0	0001 = 1	0010 = 2	0011 = 3
0100 = 4	0101 = 5	0110 = 6	0111 = 7
1000 = 8	1001 = 9	1010 = A	1011 = B
1100 = C	1101 = D	1110 = E	1111 = F

Using the nybble code, we can represent the two similar word patterns given above, with the following more manageable shorthand versions:

FEFF FF7F

Of course, the assignment of the symbols for the various nybble patterns was not so arbitrary as I've tried to make it appear. A perceptive reader who has been exposed to binary numbers will have noticed an underlying system to the assignments. If the 1's and 0's of the patterns are interpreted as actual *numbers*, rather than mere symbols for bit states, the first 10 patterns correspond to binary numbers whose decimal representation is the symbol assigned to the pattern. The last six patterns receive the symbols "A" through "F", and taken together, the symbols 0 through F constitute the digits of the *hexadecimal* number system. Thus, the symbols assigned to the different nybble patterns were born out of historical prejudice in thinking of the computer as strictly a number handling machine. Although this is an important interpretation of these symbols, for the time being it's enough to merely think of them as a shorthand way to write down bit patterns.

Because some nybble patterns can look just like a number, it's often necessary to somehow indicate that we're talking about a pattern. In BASIC, you do this by adding the characters &H to the beginning of the pattern: &H1234. A more common convention is to just add the letter H to the end of the pattern: 1234H. In both conventions, the H is referring to hexadecimal.

Eventually you'll want to learn about using the hexadecimal number system, since it is an important way to use bit patterns. I'm not going to discuss it in this primer, because a number of books have much better treatments of this topic than I could produce. Consider this an advanced topic you'll want to fill in later.

ADDRESSING MEMORY

As stated before, the 8088 chip inside your computer can manipulate the bit patterns which make up the computer's memory. Some of the possible manipulations are copying patterns from one place to another, turning on or turning off certain bits, or interpreting the patterns as numbers and performing arithmetic operations on them. To perform any of these actions, the 8088 has to know what part of memory is to be worked on. A specific location in memory is identified by it's *address*.

An address is a pointer into memory. Each address points to the beginning of a byte long chunk of memory. The 8088 has the capability to distinguish 1,048,576 different bytes of memory.

By this point, it probably comes as no surprise to hear that addresses are represented as patterns of bits. It takes 20 bits to get a total of 1,048,576 different patterns, and thus an address may be written down as a series of 5 nybble codes. For example, DOS stores a pattern which encodes information about what equipment is installed on your IBM PC in the word which begins at location 00410. Interpreting the address as a hex number, the second byte of this word has an address 1 greater than 00410, or 00411.

The 8088 isn't very happy handling 20 bits at a time. The biggest chunk that's convenient for it to use is a 16 bit word. The 8088 actually calculates 20 bit addresses as the combination of two words, a segment word and an offset word. The combination process involves interpreting the two patterns as hexadecimal numbers and adding them. The way that two 16 bit patterns can be combined to give one 20 bit pattern is that the two patterns are added out of alignment by one nybble:

0040	4 nybble segment
0010	4 nybble offset

00410	5 nybble address

Because of this mechanism for calculating addresses, they will often be written down in what may be called segment:offset form. Thus, the address in above calculation could be written:

0040:0010

MEMORY CONCERNS: DATA AND PROGRAMS

The contents of memory may be broken down into two broad classes. The first is **data**, just raw patterns of bits for the 8088 to work on. The significance of the patterns is determined by what the computer is being used for at any given time.

The second class of memory contents are **instructions**. The 8088 can look at memory and interpret a pattern it sees there as specifying one of the 200 some fundamental operations it knows how to do. This mapping of patterns onto operations is called the **machine language** of the 8088. A machine language **program** consists of a series of patterns located in consecutive memory locations, whose corresponding operations perform some useful process.

Note that there is no way for the 8088 to know whether a given pattern is meant to be an instruction, or a piece of data to operate on. It is quite possible for the chip to accidentally begin reading what was intended to be data, and interpret it as a program. Some pretty bizarre things can occur when this happens. In assembly language programming circles, this is known as "crashing the system".

THE DAWN OF ASSEMBLY LANGUAGE

Unless you happen to be an 8088 chip, the patterns which make up a machine language program can be pretty incomprehensible. For example, the pattern which tells the 8088 to flip all the bits in the byte at address 5555 is:

```
F6 16 55 55
```

which is not very informative, although you can see the 5555 address in there. In ancient history, the old wood-burning and vacuum tube computers were programmed by laboriously figuring out bit patterns which represented the series of instructions desired. Needless to say, this technique was incredibly tedious, and very prone to making errors. It finally occurred to these ancestral programmers that they could give the task of figuring out the proper patterns to the computer itself, and assembly language programming was born.

Assembly language represents each of the many operations that the computer can do with a *mnemonic*, a short, easy to remember series of letters. For example, in boolean algebra, the logical operation which inverts the state of a bit is called "not", and hence the assembly language equivalent of the preceding machine language pattern is:

```
NOTB [5555]
```

The brackets around the 5555 roughly mean "the memory location addressed by". The "B" at the end of "NOTB" indicates that we want to operate on a byte of memory, not a word.

Unfortunately, the 8088 can't make head nor tail of the string of characters "NOTB". What's needed is a special program to run on the 8088 which converts the string "NOTB" into the pattern F6 16.

This program is called an assembler. A good analogy is that an assembler program is like a meat grinder which takes in assembly language and gives out machine language.

Typically, an assembler reads a file of assembly language and translates it one line at a time, outputting a file of machine language. Often times the input file is called the *source file* and the output file is called the *object file*. The machine language patterns produced are called the *object code*.

9

Also produced during the assembly process is a *listing*, which summarizes the results of the assembly process. The listing shows each line from the source file, along with the shorthand "nybble code" representation of the object code produced. In the event that the assembler was unable to understand any of the source lines, it inserts error messages in the listing, pointing out the problem.

The primeval assembly language programmers had to write their assembler programs in machine language, because they had no other choice. Not being a masochist, I wrote CHASM in BASIC. When you think about it, there's a sort of circular logic in action here. Some programmers at Microsoft wrote the BASIC interpreter in assembly language, and I used BASIC to write an assembler. Someday, I hope to use the present version of CHASM to produce a machine language version, which will run about a hundred times faster, and at the same time bring this crazy process full circle.

THE 8088

The preceding discussions have (I hope) given you some very general background, a world view if you will, about assembly and machine language programming. At this point, I'd like to get into a little more detail, beginning by examining the internal

structure of the 8088 microprocessor, from the programmer's point of view. This discussion is a condensation of information which I obtained from "The 8086 Book" which was written by Russell Rector and George Alexy, and published by Osborne/McGraw-Hill. Once you've digested this, I'd recomend going to The 8086 Book for a deeper treatment. To use the CHASM assembler, you're going to need The 8086 Book anyway, to tell you the different 8088 instructions and their mnemonics.

Inside the 8088 are a number of *registers* each of which can hold a 16 bit pattern. In assembly language, each of the registers has a two letter mnemonic name. There are 14 registers, and their mnemonics are:

AX BX CX DX SP BP SI DI CS DS SS ES PC ST

Each of the registers are a little different and have different intended uses, but they can be grouped into some broad classes.

The *general purpose* registers (AX BX CX DX) are just that. These are registers which hold patterns pulled in from memory which are to be worked on within the 8088. You can use these registers for just about anything you want.

Each of the general purpose registers can be broken down into two 8 bit registers, which have names of their own. Thus, the CX register is broken down into the CH and CL registers. The "H" and "L" stand for high and low respectively. Each general purpose register breaks down into a high/low pair.

The AX register, and it's 8 bit low half, the AL register, are somewhat special. Mainly for historical reasons, these registers are referred to as the 16 bit and 8 bit *accumulators*. Some operations of the 8088 can only be carried out on the contents of the accumulators, and many others are faster when used in

conjunction with these registers.

Another group of registers are the **segment** registers (CS DS SS ES). These registers hold segment values for use in calculating memory addresses. The CS, or code segment register, is used every time the 8088 accesses memory to read an instruction pattern. The DS, or data segment register, is used for bringing data patterns in. The SS register is used to access the stack (more about the stack later). The ES is the extra segment register. A very few special instructions use the ES register to access memory, plus you can override use of the DS register and substitute the ES register, if you need to maintain two separate data areas.

The **pointer** (SP BP) and **index** (DI SI) registers are used to provide indirect addressing, which is an very powerful technique for accessing memory. Indirect addressing is beyond the scope of this little primer, but is discussed in The 8086 Book. The SP register is used to implement a stack in memory. (again, more about the stack later) Besides their special function, the BP, DI and SI registers can be used as additional general purpose registers. Although it's physically possible to directly manipulate the value in the SP register, it's best to leave it alone, since you could wipe out the stack.

Finally, there are two registers which are relatively inaccessible to direct manipulation. The first is the **program*

*counter**, PC. This register always contains the offset part of the address of the next instruction to be executed. Although you're not allowed to just move values into this register, you **can** indirectly affect it's contents, and hence the next instruction to be executed, using operations which are equivalent to BASIC's GOTO and GOSUB instructions. Occasionally, you will see the PC referred to as the **IP**, which stands for instruction pointer.

The last register is also relatively inaccessible. This is the *status* register, ST. This one has a *two* alternate names, so watch for FL (flag register) and PSW (program status word). The latter is somewhat steeped in history, since this was the name given to a special location in memory which served a similar function on the antique IBM 360 mainframe.

The status register consists of a series of one bit *flags* which can affect how the 8088 works. There are special instructions which allow you to set or clear each of these flags. In addition, many instructions affect the state of the flags, depending on the outcome of the instruction. For example, one of the bits of the status register is called the Zero flag. Any operation which ends up generating a bit pattern of all 0's automatically sets the Zero flag on.

Setting the flags doesn't seem to do much, until you know that there a whole set of conditional branching instructions which cause the equivalent to a BASIC GOTO if the particular æmãg pattern they look for is set. In assembly language, the only way to make a decision and branch accordingly is via this flag testing mechanism.

Although some instructions implicitly affect the flags, there are a series of instructions whose *only* effect is to set the flags, based on some test or comparison. It's very common to see one of these comparison operations used to set the flags just before a conditional branch. Taken together, the two instructions are exactly equivalent to BASIC's:

```
IF (comparison) THEN GOTO (linenumber)
```

ASSEMBLY LANGUAGE SYNTAX

In general, each line of an assembly language program translates to a set of patterns which specify one fundamental operation for the 8088 to carry out.

Each line may consist of one or more of the following parts:

First, a label, which is just a marker for the assembler to use. If you want to branch to an instruction from some other part of the program, you put a label on the instruction. When you want to branch, you refer to the label. In general, the label can be any string of characters you want. A good practice is to use a name which reminds you what that particular part of the program does. CHASM will assume that any string of characters which starts in the first column of a line is intended to be a label.

After the label, or if the text of the line starts to the right of the first column, at the beginning of the text, comes an instruction mnemonic. This specifies the operation that the line is asking for. For a list of the 200-odd mnemonics, along with the instructions they stand for, see *The 8086 Book*.

Most of the 8088 instructions require that you specify one or more **operands**. The operands are what the operation is to work on, and are listed after the instruction mnemonic.

There are a number of possible operands. Probably the most common are registers, specified by their two letter mnemonics.

Another operand type is **immediate data**, a pattern of bits to be put somewhere or compared or combined with some other pattern. Generally immediate data is specified by its nybble code representation, marked as such by following it with the letter "H". Some assemblers allow alternate ways to specify immediate data which emphasize the pattern's intended use. CHASM recognizes five different ways to represent immediate data.

A memory location can be used as an operand. We've seen one way to do this, by enclosing its address in brackets. (You can now see why the brackets are needed. Without them, you couldn't distinguish between an address and immediate data.) If you've asked the assembler to set aside a section of memory for data

(more on this latter), and put a label on the request, you can specify that point in memory by using the label. Finally, there are a number of indirect ways to address memory locations, which you can read about in The 8086 Book.

The last major type of operands are labels. Branching instructions require an operand to tell them where to branch **to**. In assembly language, you specify locations which may be branched to by putting a label on them. You can then use the label as an operand on branches.

Often times, the order in which the operands are listed can be important. For example, when moving a pattern from one place to another, you need to specify where the pattern is to come from, and where it's going. The convention in general use is that the first operand is the **destination** and the second is the **source**. Thus, to move the pattern in the DX register into the AX register, you would write:

```
MOV AX,DX
```

This may take some getting used to, since when reading from left to right it seems reasonable to assume that the transfer goes in this direction as well. However, since this convention is pretty well entrenched in the assembly language community, CHASM goes along with it.

The last part of an assembly language line is a **comment**. The comment is totally ignored by the assembler, but is **vital** for humans who are attempting to understand the program. Assembly language programs tend to be very hard to follow, and so it's particularly important to put in lots of comments so that you'll remember just what it was you were trying to do with a given piece of code. Professional assembly language programmers put a comment on **every** line of code, explaining what it does, plus devoting many entire lines for additional explanations. For an example of a professional assembly language program, you should examine the BIOS source listing given in the IBM Technical Reference manual. Over **half** the text consists of comments!

Since the assembler ignores the comments, they cost you nothing in terms of size or speed of execution in the resulting machine language program. This is in sharp contrast to BASIC, where each remark slows your program down and eats up precious memory.

Generally, a character is set aside to indicate to the assembler the beginning of a comment, so that it knows to skip over. CHASM follows a common convention of reserving the semi-colon (;) for marking comments.

THE STACK

I've been dropping the name **stack** from time to time. The stack is just a portion of memory which has been temporarily set aside to be used in a special way.

To get a picture of how the stack works, think of the spring loaded contraptions you sometimes see holding trays in a cafeteria. As each tray is washed, the busboy puts it on top of the stack in the contraption. Because the thing is spring loaded, the whole stack sinks down from the weight of the new tray, and the top of the stack ends up always being the same height off the floor. When a customer takes a tray off the stack, the next one rises up to take it's place.

In the computer, the stack is used to hold data patterns, which are generally being passed from one program or subroutine to another. By putting things on the stack, the receiving routine doesn't need to know a particular address to look for the information it needs, it just pulls them off the top of the stack.

There is some jargon associated with use of the stack. Patterns are **pushed** onto the stack, and **popped** off. Accordingly, there are a set of PUSH and POP instructions in the 8088's repertoire.

Because you don't need to keep track of where the patterns are actually being kept, the stack is often used as a scratch pad area, patterns being pushed when the register they're in is needed for some other purpose, then popped out when the register is free. It's very common for the first few instructions of a subroutine to be a series of pushes to save the patterns which

are occupying the registers its about to use. This is referred to as *saving the state* of the registers. The last thing the subroutine will do is pop the patterns back into the registers they came from, thus *restoring the state* of the registers.

15

Following the analogy of the cafeteria contraption, when you pop the stack, the pattern you get is the last one which was pushed. When you pop a pattern off, the next-to-last thing pushed automatically moves to the top, just as the trays rise up when a customer removes one. Everything comes off the stack in the reverse order of which they went on. Sometimes you'll see the phrase "last in, first out" or *LIFO stack*.

Of course, there are no special spring loaded memory locations inside the computer. The stack is implemented using a register which keeps track of where the top of the stack is currently located. When you push something, the pointer is moved to the next available memory location, and the pattern is put in that spot. When something is popped, it is copied from the location pointed at, then the pointer is moved back. You don't have to worry about moving the pointer because it's all done automatically with the push and pop instructions.

The register set aside to hold the pointer is SP, and that's why you don't want to monkey with SP. You'll recall that to form an address, two words are needed, an offset and a segment. The segment word for the stack is kept in the SS register, so you should leave SS alone as well. When you run the type of machine language program that CHASM produces, DOS will automatically set the SP and SS registers to reserve a stack capable of holding 128 words.

SOFTWARE INTERRUPTS

I have been religiously avoiding talking about the various individual instructions the 8088 can carry out, because if I didn't, this little primer would soon grow into a rather long book. However, there's one very important instruction, which when you read about it in The 8088 Book, won't seem particularly useful. This section will discuss the *software interrupt* instruction, and why it's so important.

The 8088 reserves the first 1024 bytes of memory for a series of 256 *interrupt vectors*. Each of these two word long interrupt vectors is used to store the segment:offset address of a location in memory. When you execute a software interrupt instruction, the the 8088 pushes the location of the next instruction of your program onto the stack, then branches to the memory location pointed at by the vector specified in the interrupt.

16

This probably seems like a rather awkward way to branch around in memory, and chances are you'd never use this method to get from one part of your program to another. The way these instructions become important is that IBM has pre-loaded a whole series of useful little (and not so little) machine language routines into your computer, and set the interrupt vectors to point to them. All of these routines are set up so that after doing their thing, they use the location pushed on the stack by the interrupt instruction to branch back to your program.

Some of these routines are a part of DOS, and documentation for them can be found in Appendix D of the DOS manual. The rest of them are stored in ROM (read only memory) and comprise the *BIOS*, or basic input/output system of the computer. Details of the BIOS routines can be found in Appendix A of IBM's Technical Reference Manual. IBM charges around \$40 for Technical Reference, but the information in Appendix A alone is easily worth the money.

The routines do all kinds of useful things, such as run the disk

drive for you, print characters on the screen, or read data from the keyboard. In effect, the software interrupts add a whole series of very powerful operations to the 8088 instruction set.

A final point is that if you don't like the way that DOS or the BIOS does something, the vectored interrupt system makes it very easy to substitute your own program to handle that function. You just load your program and reset the appropriate interrupt vector to point at your program rather than the resident routine. This is how all those RAM disk and print spooler programs work. The programs change the vector for disk drive or printer support to point to themselves, and carry out the operations in their own special way.

To make things easy for you, one of the DOS interrupt routines has the function of resetting interrupt vectors to point at new code. Still another DOS interrupt routine is used to graft new code onto DOS, so that it doesn't accidentally get wiped out by other programs. The whole thing is really quite elegant and easy to use, and IBM is to be complimented for setting things up this way.

PSEUDO-OPERATIONS

Up to this point, I've implied that each line of an assembly language program gets translated into a machine language instruction. In fact, this is not the case. Most assemblers recognize a series of *pseudo-operations* which are handled as embedded commands to the assembler itself, not as an instruction in the machine language program being built. Almost invariably you'll see the phrase "pseudo-operation" abbreviated down to *pseudo-op*. Sometimes you'll see *assembler directive*, which means the same thing, but just doesn't seem to roll off the

tongue as well as pseudo-op.

One very common pseudo-op is the **equate**, usually given mnemonic **EQU**. What this allows you to do is assign a name to a frequently used constant. Thereafter, anywhere you use that name, the assembler automatically substitutes the equated constant. This process makes your program easier to read, since in place of the somewhat meaningless looking pattern, you see a name which tells you what the pattern is for. It also makes your program easier to modify, since if you decide to change the constant, you only need to do it once, rather than all over the program.

The only other type of pseudo-op I'll talk about here are those for setting aside memory locations for data. These pseudo-ops tend to be quite idiosyncratic with each assembler. CHASM implements two such pseudo-ops: DB (declare byte) and DS (declare storage). DB is used to set aside small data areas, which can be initialized to any pattern, one byte at a time. DS sets up relatively large areas, but all the locations are filled with the same initial pattern.

If you put a label on a pseudo-op which sets aside data areas, most assemblers allow you to use the label as an operand, in place of the actual address of the location. The assembler automatically substitutes the address for the name during the translation process.

Some assemblers have a great number of pseudo-ops. CHASM implements a couple more, which aren't discussed here.

To conclude this primer, this section will walk through the process of writing, assembling, and running a very simple program.

The program will perform the function filled by the BASIC command CLS, that is, it will clear the video screen and move the cursor to the upper left hand corner. In fact, this is a useful little program, since the DOS environment doesn't provide any method of clearing the screen.

There is a BIOS routine called VIDEO_IO which provides an interface to the screen. Access to VIDEO_IO is through software interrupt number 16, and documentation can be found on pages A-43 and A-44 of Technical Reference. VIDEO_IO actually performs 15 different screen handling functions. We specify which function we want, along with information needed by the individual function, in the 8088 registers. Our entire program will be made up of putting the proper patterns into the registers, then activating VIDEO_IO with an interrupt.

To clear the screen, we'll use VIDEO_IO's scroll up function. What this does is move a portion of the screen up, filling the vacated space with blanks. We have to tell VIDEO_IO what portion of the screen to scroll, and how far to scroll it. We can get the proper patterns into the right registers using the MOV instruction, MOVing the patterns in as immediate data. Here's the code to do this:

```

MOV AH,6      ;this specifies we want a scroll
              ;the CH/CL register pair specifies the row and
              ;column of the upper left hand corner of the region
              ;to be scrolled
MOV CH,0      ;row = 0
MOV CL,0      ;column = 0
              ;the DH/DL pair does the same for the lower
              ;right corner.
MOV DH,24     ;row = 24
MOV DL,79     ;column = 79
              ;BH specifies what color to fill with
MOV BH,7      ;we'll use black
              ;AL specifies how far to scroll.
MOV AL,0      ;pattern 0 means to blank out the whole region.
INT 16        ;call video_io

```

Notice that none of the lines starts at the left margin (column 1). If they did, CHASM would think that the instruction mnemonic was meant to be a label, and would get very confused.

Since the bit patterns are meant to represent numbers, I've chosen to write down the immediate data as decimal numbers. CHASM will automatically translate into the proper patterns. Notice that since each of the high/low register pairs can be accessed as a single 16 bit register, I could have moved the patterns for both halves in at the same time. I did it this way for clarity. Note also the profusion of comments.

The second half of the program has to move the cursor to the upper left. Again, all that's necessary is to load the registers and execute the interrupt:

```

MOV AH,2      ;specifies that we want to position the cursor.
              ;the DH/DL pair specifies the row and column of
              ;where we want the cursor.
MOV DH,0      ;row = 0
MOV DL,0      ;column = 0
              ;BH specifies which display page
MOV BH,0      ;put the cursor on page 0
INT 16        ;call video_io

```

There's one last detail. We have to warn the 8088 that it's come to the end of our program, or it'll just keep executing whatever random patterns are in memory after our stuff. Remember "crashing the system"? One of DOS's vectored interrupts handles program termination, returning you to DOS. The last instruction is:

```
INT 32      ;return to DOS
```

After writing the program, we must now create a text file which contains the lines of our program. This is done using a text editor, such as EDLIN, which comes on your DOS disk. At this point, you can either copy the above lines into a file using an editor, or use the file CLS.ASM, which was included on your CHASM disk. CLS.ASM contains the above lines already entered for you, if you'd rather not bother making your own file at the moment.

It's now time to assemble the program. From DOS, you start CHASM up by typing it's name:

```
A> CHASM
```

CHASM will respond by printing a hello screen, and ask you to press a key when you're done reading it. When you do so, CHASM will ask you some questions:

```
Source code file name? [.asm]
```

Type in the name of the file which has your assembly language program text in it, then press return.

```
Direct listing to Printer (P), Screen (S), or Disk (D)?
```

CHASM wants to know where to send the listing produced during the assembly process. If you have a printer, turn it on then press P. If you don't have a printer, press S.

The last question is:

```
Name for object file? [xxx.com]
```

CHASM is asking for the name you'd like to give to the machine language program which is about to be produced. Just press enter here. (We'll accept CHASM's default name)

21

At this point CHASM will start accessing the disk drive, reading in your program a line at a time. A status line will appear at the bottom of your screen, telling you how far along the translation has gotten. For this program, the whole process takes about 2 1/2 minutes.

If the listing went to your printer, CHASM automatically returns you to DOS when it's finished. If it went to the screen, CHASM waits for you to press a key to indicate that you're done reading. Near the bottom of the listing will be the message:

```
XXX Diagnostics Offered  
YYY Errors Detected
```

If both numbers are 0, everything went fine. If not, look up on the listing for error messages, which will point out the offending lines. At this point, don't worry too much about what the error messages say, just fix the line in your input file to look like the text developed above. Once you manage to get an assembly with no errors, you're ready to go on.

Your disk will now contain machine language program whose name is that of your input file, with an extension of .COM. Check this by typing DIR to get a directory listing. Not only will this confirm that the file is really there, it fills up your screen, to give us something to clear.

To run the machine language program, you just type it's name, with or without the .COM extension. (Note: even though you don't need to *enter* the it, the file has to have the .COM extension for DOS to recognize it as a machine language program.) If

everything was done right, the screen will clear, and then the DOS prompt, A>, will appear.

That's the entire process, from start to finish. At this point you should have enough of a start to be able to digest CHASM's documentation and The 8086 Book, then begin to write your own programs. Good Luck!