Memory Layout and Access Chapter Four

Chapter One discussed the basic format for data in memory. Chapter Three covered how a computer system physically organizes that data. This chapter discusses how the 80x86 CPUs access data in memory.

4.0 Chapter Overview

This chapter forms an important bridge between sections one and two (Machine Organization and Basic Assembly Language, respectively). From the point of view of machine organization, this chapter discusses memory addressing, memory organization, CPU addressing modes, and data representation in memory. From the assembly language programming point of view, this chapter discusses the 80x86 register sets, the 80x86 memory addressing modes, and composite data types. This is a pivotal chapter. If you do not understand the material in this chapter, you will have difficulty understanding the chapters that follow. Therefore, you should study this chapter carefully before proceeding.

This chapter begins by discussing the registers on the 80x86 processors. These processors provide a set of general purpose registers, segment registers, and some special purpose registers. Certain members of the family provide additional registers, although typical application do not use them.

After presenting the registers, this chapter describes memory organization and segmentation on the 80x86. Segmentation is a difficult concept to many beginning 80x86 assembly language programmers. Indeed, this text tends to avoid using segmented addressing throughout the introductory chapters. Nevertheless, segmentation is a powerful concept that you must become comfortable with if you intend to write non-trivial 80x86 programs.

80x86 memory addressing modes are, perhaps, the most important topic in this chapter. Unless you completely master the use of these addressing modes, you will not be able to write reasonable assembly language programs. Do not progress beyond this section of the text until you are comfortable with the 8086 addressing modes. This chapter also discusses the 80386 (and later) extended addressing modes. Knowing these addressing modes is not that important for now, but if you do learn them you can use them to save some time when writing code for 80386 and later processors.

This chapter also introduces a handful of 80x86 instructions. Although the five or so instructions this chapter uses are insufficient for writing real assembly language programs, they do provide a sufficient set of instructions to let you manipulate variables and data structures – the subject of the next chapter.

4.1 The 80x86 CPUs:A Programmer's View

Now it's time to discuss some real processors: the 8088/8086, 80188/80186, 80286, and 80386/80486/80586/Pentium. Chapter Three dealt with many hardware aspects of a computer system. While these hardware components affect the way you should write software, there is more to a CPU than bus cycles and pipelines. It's time to look at those components of the CPU which are most visible to you, the assembly language programmer.

The most visible component of the CPU is the register set. Like our hypothetical processors, the 80x86 chips have a set of on-board registers. The register set for each processor in the 80x86 family is a superset of those in the preceding CPUs. The best place to start is with the register set for the 8088, 8086, 80188, and 80186 since these four processors have the same registers. In the discussion which follows, the term "8086" will imply any of these four CPUs.

Intel's designers have classified the registers on the 8086 into three categories: general purpose registers, segment registers, and miscellaneous registers. The general purpose registers are those which may appear as operands of the arithmetic, logical, and related instructions. Although these registers are "general purpose", every one has its own special purpose. Intel uses the term "general purpose" loosely. The 8086 uses the segment registers to access blocks of memory called, surprisingly enough, segments. See "Segments on the 80x86" on page 151 for more details on the exact nature of the segment registers. The final class of 8086 registers are the miscellaneous registers. There are two special registers in this group which we'll discuss shortly.

4.1.1 8086 General Purpose Registers

There are eight 16 bit general purpose registers on the 8086: ax, bx, cx, dx, si, di, bp, and sp. While you can use many of these registers interchangeably in a computation, many instructions work more efficiently or absolutely require a specific register from this group. So much for general purpose.

The ax register (*Accumulator*) is where most arithmetic and logical computations take place. Although you can do most arithmetic and logical operations in other registers, it is often more efficient to use the ax register for such computations. The bx register (*Base*) has some special purposes as well. It is commonly used to hold indirect addresses, much like the bx register on the x86 processors. The cx register (*Count*), as its name implies, counts things. You often use it to count off the number of iterations in a loop or specify the number of characters in a string. The dx register (*Data*) has two special purposes: it holds the overflow from certain arithmetic operations, and it holds I/O addresses when accessing data on the 80x86 I/O bus.

The si and di registers (*Source Index* and *Destination Index*) have some special purposes as well. You may use these registers as pointers (much like the bx register) to indirectly access memory. You'll also use these registers with the 8086 string instructions when processing character strings.

The bp register (*Base Pointer*) is similar to the bx register. You'll generally use this register to access parameters and local variables in a procedure.

The sp register (*Stack Pointer*) has a very special purpose – it maintains the *program stack*. Normally, you would not use this register for arithmetic computations. The proper operation of most programs depends upon the careful use of this register.

Besides the eight 16 bit registers, the 8086 CPUs also have eight 8 bit registers. Intel calls these registers al, ah, bl, bh, cl, ch, dl, and dh. You've probably noticed a similarity between these names and the names of some 16 bit registers (ax, bx, cx, and dx, to be exact). The eight bit registers are not independent. al stands for "ax's L.O. byte." ah stands for "ax's H.O. byte." The names of the other eight bit registers mean the same thing with respect to bx, cx, and dx. Figure 4.1 shows the general purpose register set.

Note that the eight bit registers do not form an independent register set. Modifying al will change the value of ax; so will modifying ah. The value of al exactly corresponds to bits zero through seven of ax. The value of ah corresponds to bits eight through fifteen of ax. Therefore any modification to al or ah will modify the value of ax. Likewise, modifying ax will change *both* al and ah. Note, however, that changing al will not affect the value of ah, and vice versa. This statement applies to bx/bl/bh, cx/cl/ch, and dx/dl/dh as well.

The si, di, bp, and sp registers are only 16 bits. There is no way to directly access the individual bytes of these registers as you can the low and high order bytes of ax, bx, cx, and dx.

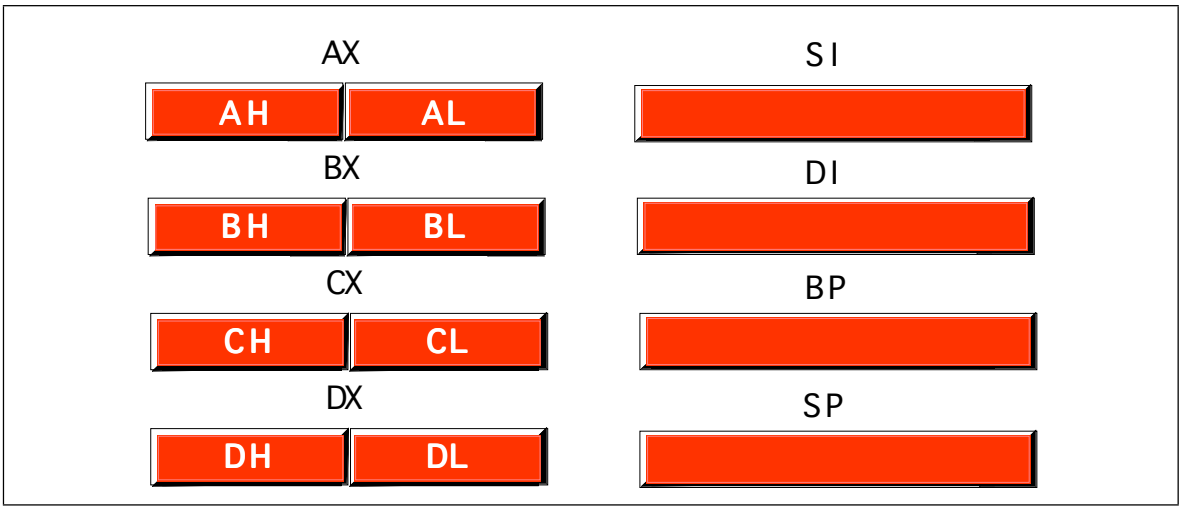

Figure 4.1 8086 Register Set

4.1.2 8086 Segment Registers

The 8086 has four special *segment registers*: cs, ds, es, and ss. These stand for *Code Segment*, *Data Segment*, *Extra Segment*, and *Stack Segment*, respectively. These registers are all 16 bits wide. They deal with selecting blocks (segments) of main memory. A segment register (e.g., cs) points at the beginning of a segment in memory.

Segments of memory on the 8086 can be no larger than 65,536 bytes long. This infamous "64K segment limitation" has disturbed many a programmer. We'll see some problems with this 64K limitation, and some solutions to those problems, later.

The cs register points at the segment containing the currently executing machine instructions. Note that, despite the 64K segment limitation, 8086 programs can be longer than 64K. You simply need multiple code segments in memory. Since you can change the value of the cs register, you can switch to a new code segment when you want to execute the code located there.

The data segment register, ds, generally points at global variables for the program. Again, you're limited to 65,536 bytes of data in the data segment; but you can always change the value of the ds register to access additional data in other segments.

The extra segment register, es, is exactly that – an extra segment register. 8086 programs often use this segment register to gain access to segments when it is difficult or impossible to modify the other segment registers.

The ss register points at the segment containing the 8086 *stack*. The stack is where the 8086 stores important machine state information, subroutine return addresses, procedure parameters, and local variables. In general, you do not modify the stack segment register because too many things in the system depend upon it.

Although it is theoretically possible to store data in the segment registers, this is never a good idea. The segment registers have a very special purpose – pointing at accessible blocks of memory. Any attempt to use the registers for any other purpose may result in considerable grief, especially if you intend to move up to a better CPU like the 80386.

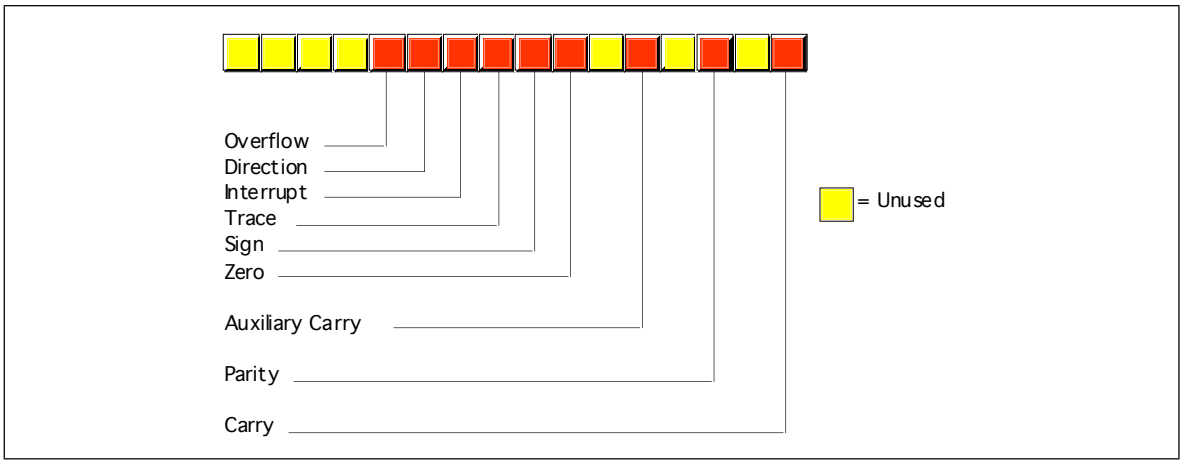

Figure 4.2 8086 Flags Register

4.1.3 8086 Special Purpose Registers

There are two special purpose registers on the 8086 CPU: the instruction pointer (ip) and the flags register. You do not access these registers the same way you access the other 8086 registers. Instead, the CPU generally manipulates these registers directly.

The ip register is the equivalent of the ip register on the x86 processors – it contains the address of the currently executing instruction. This is a 16 bit register which provides a pointer into the current code segment (16 bits lets you select any one of 65,536 different memory locations). We'll come back to this register when we discuss the control transfer instructions later.

The flags register is unlike the other registers on the 8086. The other registers hold eight or 16 bit values. The flags register is simply an eclectic collection of one bit values which help determine the current state of the processor. Although the flags register is 16 bits wide, the 8086 uses only nine of those bits. Of these flags, four flags you use all the time: zero, carry, sign, and overflow. These flags are the 8086 *condition codes.* The flags register appears in Figure 4.2.

4.1.4 80286 Registers

The 80286 microprocessor adds one major programmer-visible feature to the 8086 – protected mode operation. This text will not cover the 80286 protected mode of operation for a variety of reasons. First, the protected mode of the 80286 was poorly designed. Second, it is of interest only to programmers who are writing their own operating system or low-level systems programs for such operating systems. Even if you are writing software for a protected mode operating system like UNIX or OS/2, you would not use the protected mode features of the 80286. Nonetheless, it's worthwhile to point out the extra registers and status flags present on the 80286 just in case you come across them.

There are three additional bits present in the 80286 flags register. The I/O Privilege Level is a two bit value (bits 12 and 13). It specifies one of four different privilege levels necessary to perform I/O operations. These two bits generally contain 00b when operating in *real mode* on the 80286 (the 8086 emulation mode). The NT (*nested task*) flag controls the operation of an interrupt return (IRET) instruction. NT is normally zero for real-mode programs.

Besides the extra bits in the flags register, the 80286 also has five additional registers used by an operating system to support memory management and multiple processes: the

machine status word (msw), the global descriptor table register (gdtr), the local descriptor table register (ldtr), the interrupt descriptor table register (idtr) and the task register (tr).

About the only use a typical application program has for the protected mode on the 80286 is to access more than one megabyte of RAM. However, as the 80286 is now virtually obsolete, and there are better ways to access more memory on later processors, programmers rarely use this form of protected mode.

4.1.5 80386/80486 Registers

The 80386 processor dramatically extended the 8086 register set. In addition to all the registers on the 80286 (and therefore, the 8086), the 80386 added several new registers and extended the definition of the existing registers. The 80486 did not add any new registers to the 80386's basic register set, but it did define a few bits in some registers left undefined by the 80386.

The most important change, from the programmer's point of view, to the 80386 was the introduction of a 32 bit register set. The ax, bx, cx, dx, si, di, bp, sp, flags, and ip registers were all extended to 32 bits. The 80386 calls these new 32 bit versions eax, ebx, ecx, edx, esi, edi, ebp, esp, eflags, and eip to differentiate them from their 16 bit versions (which are still available on the 80386). Besides the 32 bit registers, the 80386 also provides two new 16 bit segment registers, fs and gs, which allow the programmer to concurrently access six different segments in memory without reloading a segment register. Note that all the segment registers on the 80386 are 16 bits. The 80386 did not extend the segment registers to 32 bits as it did the other registers.

The 80386 did not make any changes to the bits in the flags register. Instead, it extended the flags register to 32 bits (the "eflags" register) and defined bits 16 and 17. Bit 16 is the debug resume flag (RF) used with the set of 80386 debug registers. Bit 17 is the Virtual 8086 mode flag (VM) which determines whether the processor is operating in virtual-86 mode (which simulates an 8086) or standard protected mode. The 80486 adds a third bit to the eflags register at position 18 – the alignment check flag. Along with control register zero (CR0) on the 80486, this flag forces a trap (program abort) whenever the processor accesses non-aligned data (e.g., a word on an odd address or a double word at an address which is not an even multiple of four).

The 80386 added four control registers: CR0-CR3. These registers extend the msw register of the 80286 (the 80386 emulates the 80286 msw register for compatibility, but the information really appears in the CRx registers). On the 80386 and 80486 these registers control functions such as paged memory management, cache enable/disable/operation (80486 only), protected mode operation, and more.

The 80386/486 also adds eight *debugging* registers. A debugging program like Microsoft Codeview or the Turbo Debugger can use these registers to set breakpoints when you are trying to locate errors within a program. While you would not use these registers in an application program, you'll often find that using such a debugger reduces the time it takes to eradicate bugs from your programs. Of course, a debugger which accesses these registers will only function properly on an 80386 or later processor.

Finally, the 80386/486 processors add a set of test registers to the system which test the proper operation of the processor when the system powers up. Most likely, Intel put these registers on the chip to allow testing immediately after manufacture, but system designers can take advantage of these registers to do a power-on test.

For the most part, assembly language programmers need not concern themselves with the extra registers added to the 80386/486/Pentium processors. However, the 32 bit extensions and the extra segment registers are quite useful. To the application programmer, the *programming model* for the 80386/486/Pentium looks like that shown in Figure 4.3

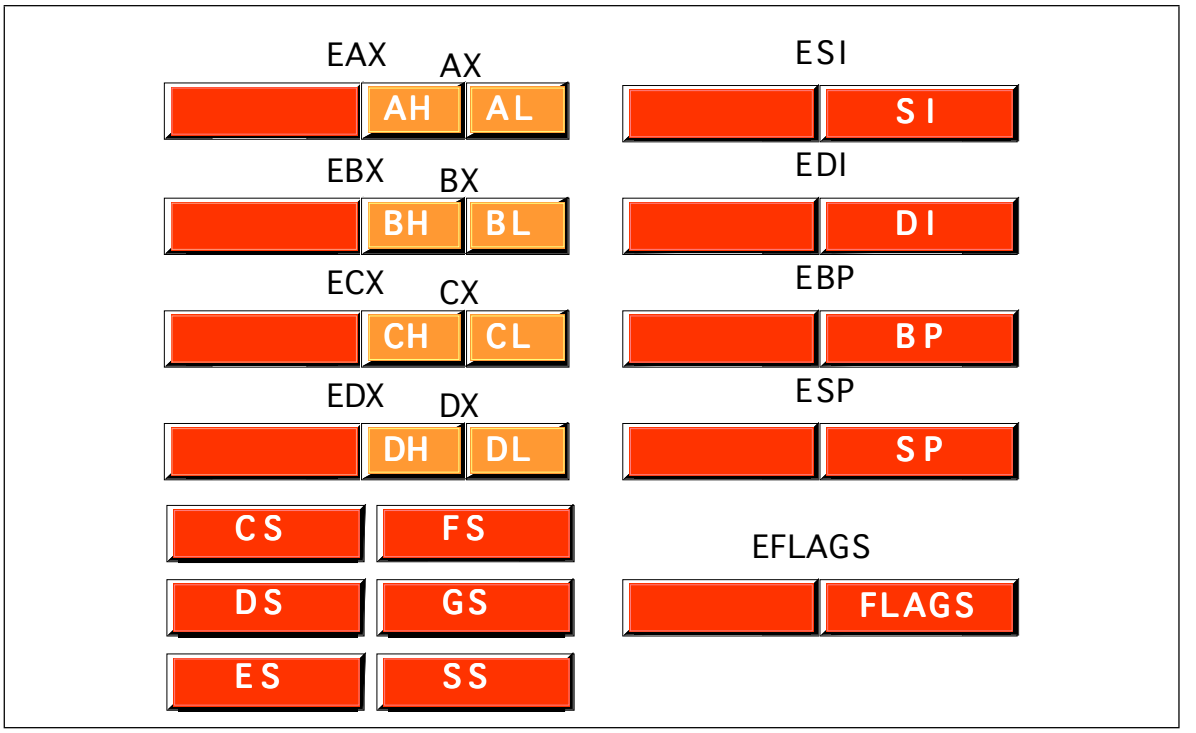

Figure 4.3 80386 Registers (Application Programmer Visible)

4.2 80x86 Physical Memory Organization

Chapter Three discussed the basic organization of a Von Neumann Architecture (VNA) computer system. In a typical VNA machine, the CPU connects to memory via the bus. The 80x86 selects some particular memory element using a binary number on the address bus. Another way to view memory is as an array of bytes. A Pascal data structure that roughly corresponds to memory would be:

```
Memory : array [0..MaxRAM] of byte;
```

The value on the address bus corresponds to the index supplied to this array. E.g., writing data to memory is equivalent to

```
Memory [address] := Value_to_Write;
```

Reading data from memory is equivalent to

```
Value_Read := Memory [address];
```

Different 80x86 CPUs have different address busses that control the maximum number of elements in the memory array (see “The Address Bus” on page 86). However, regardless of the number of address lines on the bus, most computer systems do *not* have one byte of memory for each addressable location. For example, 80386 processors have 32 address lines allowing up to four gigabytes of memory. Very few 80386 systems actually have four gigabytes. Usually, you’ll find one to 256 megabytes in an 80x86 based system.

The first megabyte of memory, from address zero to 0FFFFFh is special on the 80x86. This corresponds to the entire address space of the 8088, 8086, 80186, and 80188 microprocessors. Most DOS programs limit their program and data addresses to locations in this range. Addresses limited to this range are named *real addresses* after the 80x86 *real mode*.

4.3 Segments on the 80x86

You cannot discuss memory addressing on the 80x86 processor family without first discussing segmentation. Among other things, segmentation provides a powerful memory management mechanism. It allows programmers to partition their programs into modules that operate independently of one another. Segments provide a way to easily implement object-oriented programs. Segments allow two processes to easily share data. All in all, segmentation is a really neat feature. On the other hand, if you ask ten programmers what they think of segmentation, at least nine of the ten will claim it's terrible. Why such a response?

Well, it turns out that segmentation provides one other nifty feature: it allows you to extend the addressability of a processor. In the case of the 8086, segmentation let Intel's designers extend the maximum addressable memory from 64K to one megabyte. Gee, that sounds good. Why is everyone complaining? Well, a little history lesson is in order to understand what went wrong.

In 1976, when Intel began designing the 8086 processor, memory was very expensive. Personal computers, such that they were at the time, typically had four thousand bytes of memory. Even when IBM introduced the PC five years later, 64K was still quite a bit of memory, one megabyte was a tremendous amount. Intel's designers felt that 64K memory would remain a large amount throughout the lifetime of the 8086. The only mistake they made was completely underestimating the lifetime of the 8086. They figured it would last about five years, like their earlier 8080 processor. They had plans for lots of other processors at the time, and "86" was not a suffix on the names of any of those. Intel figured they were set. Surely one megabyte would be more than enough to last until they came out with something better[1].

Unfortunately, Intel didn't count on the IBM PC and the massive amount of software to appear for it. By 1983, it was very clear that Intel could not abandon the 80x86 architecture. They were stuck with it, but by then people were running up against the one megabyte limit of 8086. So Intel gave us the 80286. This processor could address up to 16 megabytes of memory. Surely more than enough. The only problem was that all that wonderful software written for the IBM PC was written in such a way that it couldn't take advantage of any memory beyond one megabyte.

It turns out that the maximum amount of addressable memory is not everyone's main complaint. The real problem is that the 8086 was a 16 bit processor, with 16 bit registers and 16 bit addresses. This limited the processor to addressing 64K chunks of memory. Intel's clever use of segmentation extended this to one megabyte, but addressing more than 64K at one time takes some effort. Addressing more than 256K at one time takes a *lot* of effort.

Despite what you might have heard, segmentation is not bad. In fact, it is a really great memory management scheme. What is bad is Intel's 1976 implementation of segmentation still in use today. You can't blame Intel for this – they fixed the problem in the 80's with the release of the 80386. The real culprit is MS-DOS that forces programmers to continue to use 1976 style segmentation. Fortunately, newer operating systems such as Linux, UNIX, Windows 9x, Windows NT, and OS/2 don't suffer from the same problems as MS-DOS. Furthermore, users finally seem to be more willing to switch to these newer operating systems so programmers can take advantage of the new features of the 80x86 family.

With the history lesson aside, it's probably a good idea to figure out what segmentation is all about. Consider the current view of memory: it looks like a linear array of bytes. A single index (address) selects some particular byte from that array. Let's call this type of addressing *linear* or *flat* addressing. Segmented addressing uses two components to specify a memory location: a segment value and an offset within that segment. Ideally, the segment and offset values are independent of one another. The best way to describe

1. At the time, the iapx432 processor was their next big product. It died a slow and horrible death.

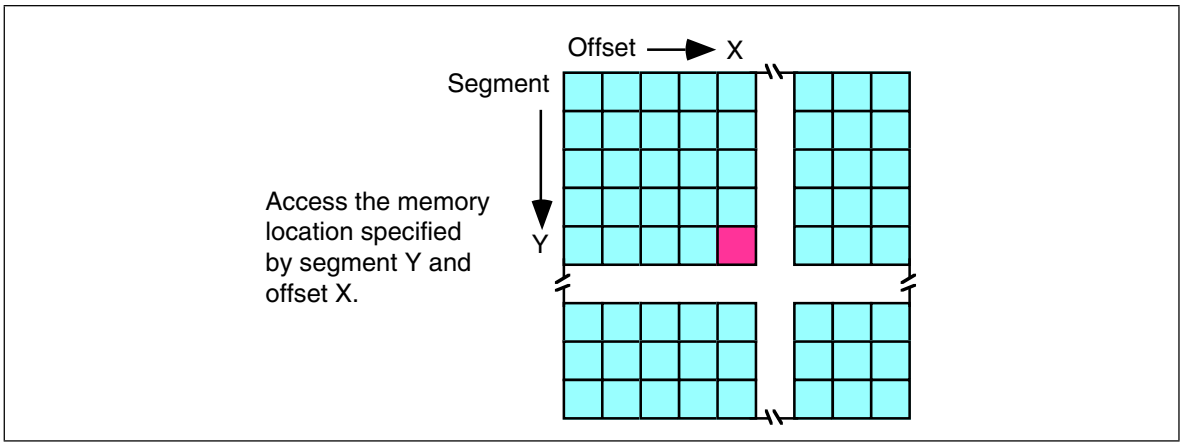

Figure 4.4 Segmented Addressing as a Two-Dimensional Process

segmented addressing is with a two-dimensional array. The segment provides one of the indices into the array, the offset provides the other (see Figure 4.4).

Now you may be wondering, "Why make this process more complex?" Linear addresses seem to work fine, why bother with this two dimensional addressing scheme? Well, let's consider the way you typically write a program. If you were to write, say, a SIN(X) routine and you needed some temporary variables, you probably would not use global variables. Instead, you would use local variables inside the SIN(X) function. In a broad sense, this is one of the features that segmentation offers – the ability to attach blocks of variables (a segment) to a particular piece of code. You could, for example, have a segment containing local variables for SIN, a segment for SQRT, a segment for DRAW-Window, etc. Since the variables for SIN appear in the segment for SIN, it's less likely your SIN routine will affect the variables belonging to the SQRT routine. Indeed, on the 80286 and later operating in *protected mode*, the CPU can *prevent* one routine from accidentally modifying the variables in a different segment.

A full segmented address contains a segment component and an offset component. This text will write segmented addresses as *segment*:*offset*. On the 8086 through the 80286, these two values are 16 bit constants. On the 80386 and later, the offset can be a 16 bit constant or a 32 bit constant.

The size of the offset limits the maximum size of a segment. On the 8086 with 16 bit offsets, a segment may be no longer than 64K; it could be smaller (and most segments are), but never larger. The 80386 and later processors allow 32 bit offsets with segments as large as four gigabytes.

The segment portion is 16 bits on all 80x86 processors. This lets a single program have up to 65,536 different segments in the program. Most programs have less than 16 segments (or thereabouts) so this isn't a practical limitation.

Of course, despite the fact that the 80x86 family uses segmented addressing, the actual (*physical*) memory connected to the CPU is still a linear array of bytes. There is a function that converts the segment value to a physical memory address. The processor then adds the offset to this physical address to obtain the actual address of the data in memory. This text will refer to addresses in your programs as *segmented addresses* or *logical addresses.* The actual linear address that appears on the address bus is the *physical address* (see Figure 4.4).

On the 8086, 8088, 80186, and 80188 (and other processors operating in *real* mode), the function that maps a segment to a physical address is very simple. The CPU multiplies the segment value by sixteen (10h) and adds the offset portion. For example, consider the segmented address[2]: 1000:1F00. To convert this to a physical address you multiply the seg-

2. All segmented addresses in this text use the hexadecimal radix. Since this text will always use the hex radix for addresses, there is no need to append an "h" to the end of such values.

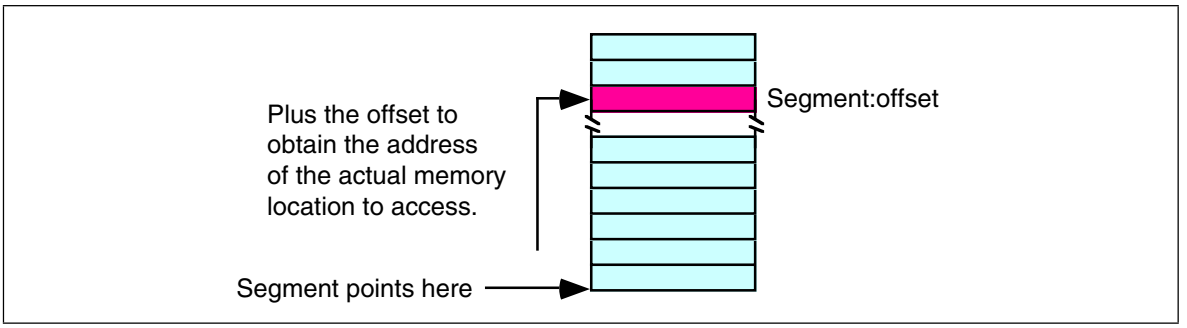

Figure 4.5 Segmented Addressing in Physical Memory

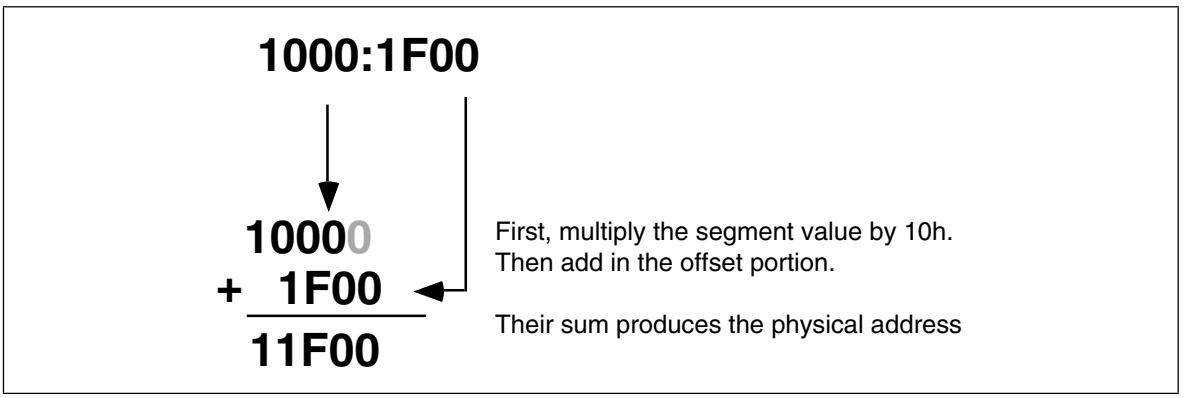

Figure 4.6 Converting a Logical Address to a Physical Address

ment value (1000h) by sixteen. Multiplying by the radix is very easy. Just append a zero to the end of the number. Appending a zero to 1000h produces 10000h. Add 1F00h to this to obtain 11F00h. So 11F00h is the physical address that corresponds to the segmented address 1000:1F00 (see Figure 4.4).

Warning: A very common mistake people make when performing this computation is to forget they are working in hexadecimal, not decimal. It is surprising to see how many people add 9+1 and get 10h rather than the correct answer 0Ah.

Intel, when designing the 80286 and later processors, did not extend the addressing by adding more bits to the segment registers. Instead, they changed the function the CPU uses to convert a logical address to a physical address. If you write code that depends on the "multiply by sixteen and add in the offset" function, your program will only work on an 80x86 processor operating in real mode, and you will be limited to one megabyte of memory[3].

In the 80286 and later processors, Intel introduced *protected mode segments*. Among other changes, Intel completely revamped the algorithm for mapping segments to the linear address space. Rather than using a function (such as multiplying the segment value by 10h), the protected mode processors use a *look up table* to compute the physical address. In protected mode, the 80286 and later processors use the segment value as the index into an array. The contents of the selected array element provide (among other things) the starting address for the segment. The CPU adds this value to the offset to obtain the physical address (see Figure 4.4).

Note that your applications cannot directly modify the segment descriptor table (the lookup table). The protected mode operating system (UNIX, Linux, Windows, OS/2, etc.) handles that operation.

3. Actually, you can also operate in V86 (virtual 86) mode on the 80386 and later, but you will still be limited to one megabyte addressable memory.

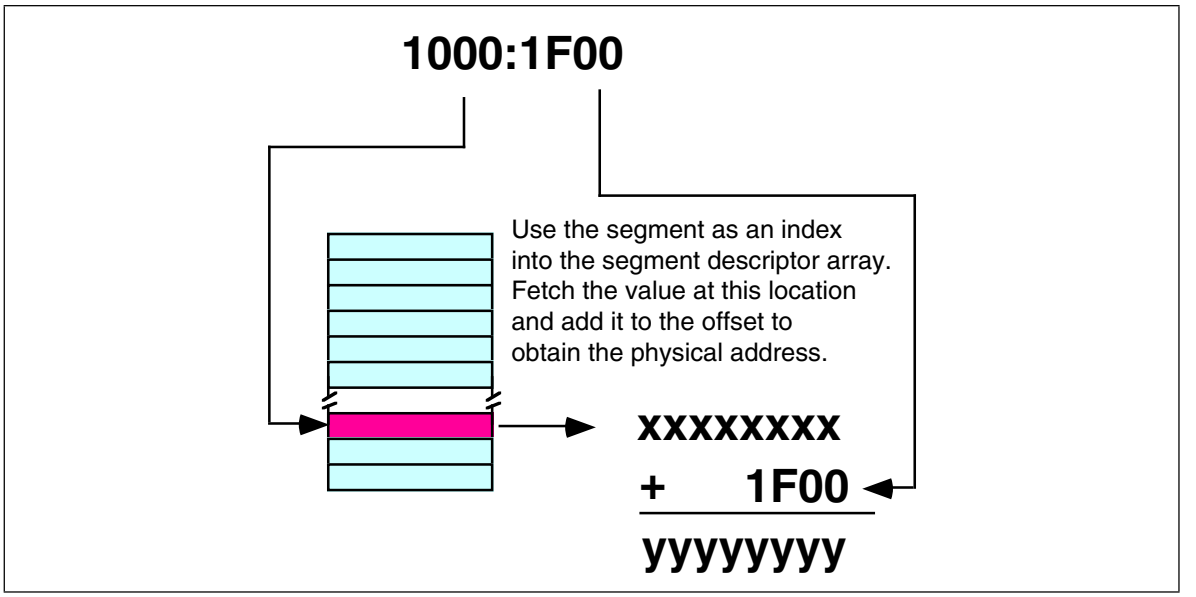

Figure 4.7 Converting a Logical Address to a Physical Address in Protected Mode

The best programs never assume that a segment is located at a particular spot in memory. You should leave it up to the operating system to place your programs into memory and not generate any segment addresses on your own.

4.4 Normalized Addresses on the 80x86

When operating in real mode, an interesting problem develops. You may refer to a single object in memory using several *different* addresses. Consider the address from the previous examples, 1000:1F00. There are several different memory addresses that refer to the same physical address. For example, 11F0:0, 1100:F00, and even 1080:1700 all correspond to physical address 11F00h. When working with certain data types and especially when comparing pointers, it's convenient if segmented addresses point at different objects in memory when their bit representations are different. Clearly this is not always the case in real mode on an 80x86 processor.

Fortunately, there is an easy way to avoid this problem. If you need to compare two addresses for (in)equality, you can use *normalized* addresses. Normalized addresses take a special form so they are all unique. That is, unless two normalized segmented values are exactly the same, they do not point at the same object in memory.

There are many different ways (16, in fact) to create normalized addresses. By convention, most programmers (and high level languages) define a normalized address as follows:

- The segment portion of the address may be any 16 bit value.
- The offset portion must be a value in the range 0..0Fh.

Normalized pointers that take this form are very easy to convert to a physical address. All you need to do is append the single hexadecimal digit of the offset to the segment value. The normalized form of 1000:1F00 is 11F0:0. You can obtain the physical address by appending the offset (zero) to the end of 11F0 yielding 11F00.

It is very easy to convert an arbitrary segmented value to a normalized address. First, convert your segmented address to a physical address using the "multiply by 16 and add in the offset" function. Then slap a colon between the last two digits of the five-digit result:

1000:1F00 ⇒ 11F00 ⇒ 11F0:0

Note that this discussion applies only to 80x86 processors operating in real mode. In protected mode there is no direct correspondence between segmented addresses and physical addresses so this technique does not work. However, this text deals mainly with programs that run in real mode, so normalized pointers appear throughout this text.

4.5 Segment Registers on the 80x86

When Intel designed the 8086 in 1976, memory was a precious commodity. They designed their instruction set so that each instruction would use as few bytes as possible. This made their programs smaller so computer systems employing Intel processors would use less memory. As such, those computer systems cost less to produce. Of course, the cost of memory has plummeted to the point where this is no longer a concern but it was a concern back then[4]. One thing Intel wanted to avoid was appending a 32 bit address (segment:offset) to the end of instructions that reference memory. They were able to reduce this to 16 bits (offset only) by making certain assumptions about which segments in memory an instruction could access.

The 8086 through 80286 processors have four segment registers: cs, ds, ss and es. The 80386 and later processors have these segment registers plus fs and gs. The cs (code segment) register points at the segment containing the currently executing code. The CPU always fetches instructions from the address given by cs:ip. By default, the CPU expects to access most variables in the data segment. Certain variables and other operations occur in the stack segment. When accessing data in these specific areas, no segment value is necessary. To access data in one of the extra segments (es, fs, or gs), only a single byte is necessary to choose the appropriate segment register. Only a few control transfer instructions allow you to specify a full 32 bit segmented address.

Now, this might seem rather limiting. After all, with only four segment registers on the 8086 you can address a maximum of 256 Kilobytes (64K per segment), not the full megabyte promised. However, you can change the segment registers under program control, so it is possible to address any byte by changing the value in a segment register.

Of course, it takes a couple of instructions to change the value of one of the 80x86's segment registers. These instructions consume memory and take time to execute. So saving two bytes per memory access would not pay off if you are accessing data in different segments all the time. Fortunately, most consecutive memory accesses occur in the same segment. Hence, loading segment registers isn't something you do very often.

4.6 The 80x86 Addressing Modes

Like the x86 processors described in the previous chapter, the 80x86 processors let you access memory in many different ways. The 80x86 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 80x86 addressing modes is the first step towards mastering 80x86 assembly language.

When Intel designed the original 8086 processor, they provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. Note that the 80386 retained all the modes of the previous processors; the new modes are just an added bonus. If you need to write code that works on 80286 and earlier processors, you will not be able to take advantage of these new modes. However, if you intend to run your code on 80386sx or higher processors, you can use these new modes. Since many programmers still need to write programs that run on 80286 and earlier machines[5], it's important to separate the discussion of these two sets of addressing modes to avoid confusing them.

4. Actually, small programs are still important. The smaller a program is the faster it will run because the CPU has to fetch fewer bytes from memory and the instructions don't take up as much of the cache.

5. Modern PCs rarely use processors earlier than the 80386, but embedded system still use the older processors.

4.6.1 8086 Register Addressing Modes

Most 8086 instructions can operate on the 8086's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 8086 `mov` (move) instruction:

```
mov     destination, source
```

This instruction copies the data from the *source* operand to the *destination* operand. The eight and 16 bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 8086 `mov` instructions:

```
mov     ax, bx      ;Copies the value from BX into AX
mov     dl, al      ;Copies the value from AL into DL
mov     si, dx      ;Copies the value from DX into SI
mov     sp, bp      ;Copies the value from BP into SP
mov     dh, cl      ;Copies the value from CL into DH
mov     ax, ax      ;Yes, this is legal!
```

Remember, the registers are the best place to keep often used variables. As you'll see a little later, instructions using the registers are shorter and faster than those that access memory. Throughout this chapter you'll see the abbreviated operands *reg* and *r/m* (register/memory) used wherever you may use one of the 8086's general purpose registers.

In addition to the general purpose registers, many 8086 instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, you may not specify cs as the destination operand, second, only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of cs to ds, you'd have to use some sequence like:

```
mov     ax, cs
mov     ds, ax
```

You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses. But more on that, later. Throughout this text you'll see the abbreviated operand *sreg* used wherever segment register operands are allowed (or required).

4.6.2 8086 Memory Addressing Modes

The 8086 provides 17 different ways to access memory. This may seem like quite a bit at first[6], but fortunately most of the address modes are simple variants of one another so they're very easy to learn. And learn them you should! The key to good assembly language programming is the proper use of memory addressing modes.

The addressing modes provided by the 8086 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed. Variations on these five forms provide the 17 different addressing modes on the 8086. See, from 17 down to five. It's not so bad after all!

4.6.2.1 The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al,ds:[8088h] loads the al register with a copy of the byte at memory loca-

6. Just wait until you see the 80386!

MASM Syntax for 8086 Memory Addressing Modes

Microsoft's assembler uses several different variations to denote indexed, based/indexed, and displacement plus based/indexed addressing modes. You will see all of these forms used interchangeably throughout this text. The following list some of the possible combinations that are legal for the various 80x86 addressing modes:

```
disp[bx], [bx][disp], [bx+disp], [disp][bx], and [disp+bx]

[bx][si], [bx+si], [si][bx], and [si+bx]

disp[bx][si], disp[bx+si], [disp+bx+si], [disp+bx][si], disp[si][bx], [disp+si][bx],
[disp+si+bx], [si+disp+bx], [bx+disp+si], etc.
```

MASM treats the "[ ]" symbols just like the "+" operator. This operator is commutative, just like the "+" operator. Of course, this discussion applies to all the 8086 addressing modes, not just those involving BX and SI. You may substitute any legal registers in the addressing modes above.

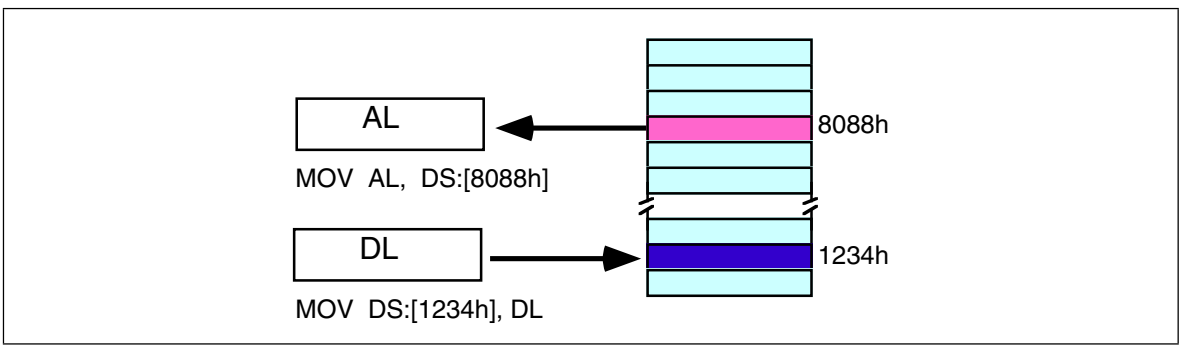

Figure 4.8 Displacement Only (Direct) Addressing Mode

tion 8088h[7]. Likewise, the instruction mov ds:[1234h],dl stores the value in the dl register to memory location 1234h (see Figure 4.8)

The displacement-only addressing mode is perfect for accessing simple variables. Of course, you'd probably prefer using names like "I" or "J" rather than "DS:[1234h]" or "DS:[8088h]". Well, fear not, you'll soon see it's possible to do just that.

Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the mov opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors (see the previous chapter). There are some minor differences, however. First of all, a displacement is exactly that– some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). Don't worry if this doesn't make a lot of sense right now. You'll get an opportunity to study segments to your heart's content a little later in this chapter. For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words on the 8086 processors[8] (see Figure 4.9).

By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a *segment override prefix* before your address. For example, to access location 1234h in the extra segment (es) you would use an instruction of the form mov ax,es:[1234h]. Likewise, to access this location in the code segment you would use the instruction mov ax, cs:[1234h]. The ds: prefix in the previous examples is *not* a segment override. The CPU uses the data segment register by default. These specific examples require ds: because of MASM's syntactical limitations.

7. The purpose of the "DS:" prefix on the instruction will become clear a little later.
8. And double words on the 80386 and later.

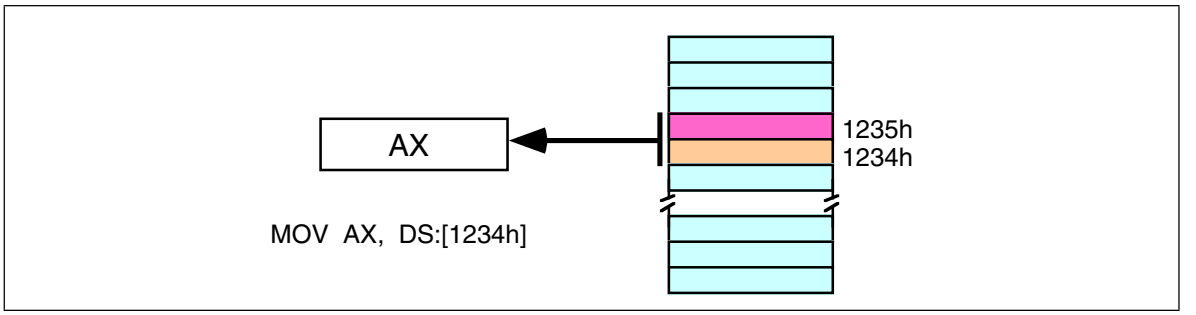

Figure 4.9 Accessing a Word

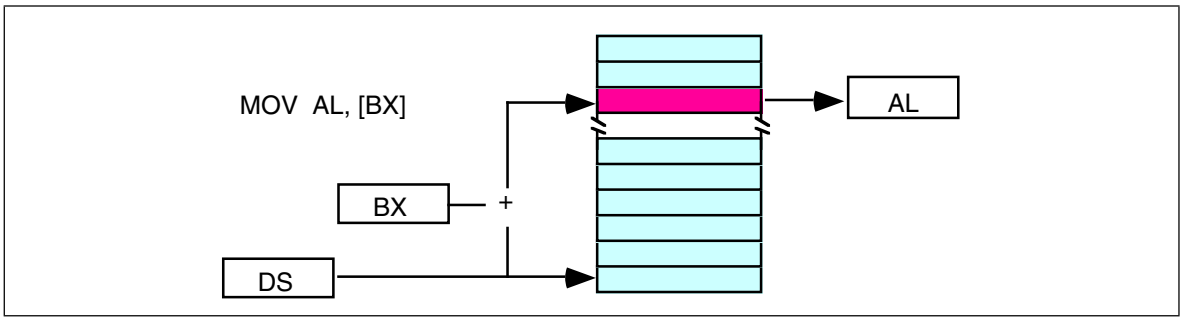

Figure 4.10 [BX] Addressing Mode

4.6.2.2 The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the ds segment by default. The [bp] addressing mode uses the stack segment (ss) by default.

You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
mov     al, cs:[bx]
mov     al, ds:[bp]
mov     al, ss:[si]
mov     al, es:[di]
```

Intel refers to [bx] and [bp] as *base addressing modes* and `bx` and `bp` as *base registers* (in fact, bp stands for base pointer). Intel refers to the [si] and [di] addressing modes as *indexed addressing modes* (si stands for *source index*, di stands for *destination index*). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.

Note: the [si] and [di] addressing modes work exactly the same way, just substitute si and di for bx above.

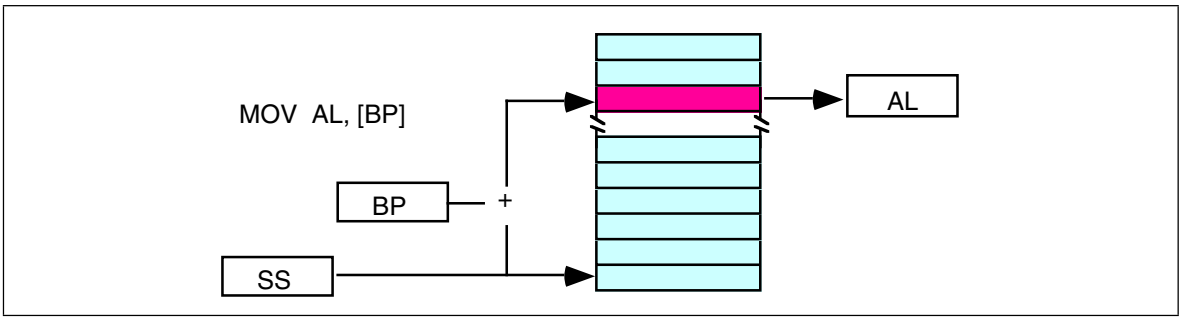

Figure 4.11 [BP] Addressing Mode

4.6.2.3 Indexed Addressing Modes

The indexed addressing modes use the following syntax:

```
mov       al, disp[bx]
mov       al, disp[bp]
mov       al, disp[si]
mov       al, disp[di]
```

If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

```
mov       al, ss:disp[bx]
mov       al, es:disp[bp]
mov       al, cs:disp[si]
mov       al, ss:disp[di]
```

You may substitute si or di in Figure 4.12 to obtain the [si+disp] and [di+disp] addressing modes.

Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel's literature does not differentiate between these modes with or without the constant. If you look at how the hardware works, this is a reasonable definition. From the programmer's point of view, however, these addressing modes are useful for entirely

Based vs. Indexed Addressing

There is actually a subtle difference between the based and indexed addressing modes. Both addressing modes consist of a displacement added together with a register. The major difference between the two is the relative sizes of the displacement and register values. In the indexed addressing mode, the constant typically provides the address of the specific data structure and the register provides an offset from that address. In the based addressing mode, the register contains the address of the data structure and the constant displacement supplies the index from that point.

Since addition is commutative, the two views are essentially equivalent. However, since Intel supports one and two byte displacements (See "The 80x86 MOV Instruction" on page 166) it made more sense for them to call it the based addressing mode. In actual use, however, you'll wind up using it as an indexed addressing mode more often than as a based addressing mode, hence the name change.

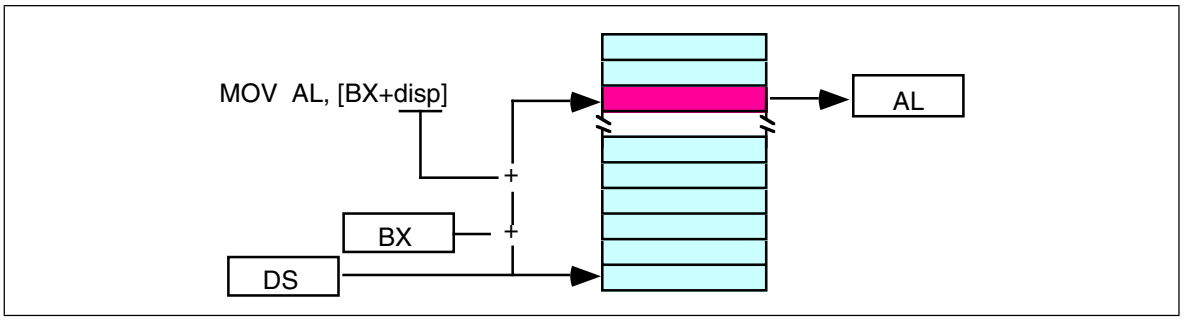

Figure 4.12 [BX+disp] Addressing Mode

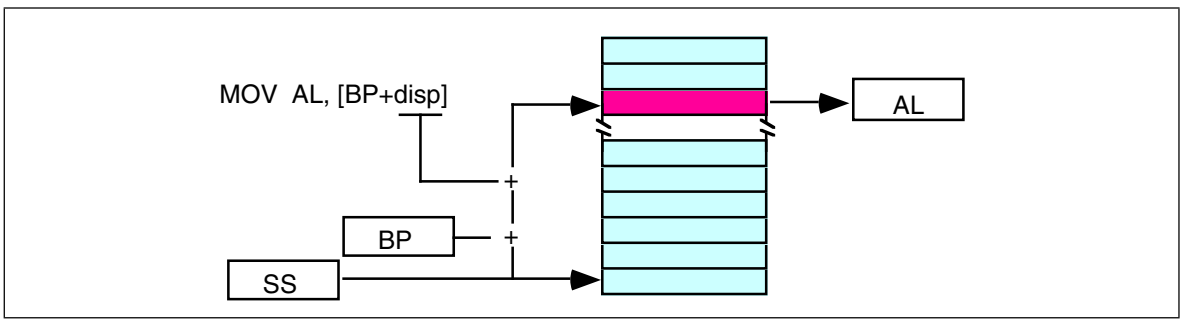

Figure 4.13 [BP+disp] Addressing Mode

different things. Which is why this text uses different terms to describe them. Unfortunately, there is very little consensus on the use of these terms in the 80x86 world.

4.6.2.4 Based Indexed Addressing Modes

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

```
mov       al, [bx][si]
mov       al, [bx][di]
mov       al, [bp][si]
mov       al, [bp][di]
```

Suppose that bx contains 1000h and si contains 880h. Then the instruction

```
mov       al,[bx][si]
```

would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D.

The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.

You substitute di in Figure 4.12 to obtain the [bx+di] addressing mode. You substitute di in Figure 4.12 for the [bp+di] addressing mode.

4.6.2.5 Based Indexed Plus Displacement Addressing Mode

These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes (see Figure 4.12 and Figure 4.12).

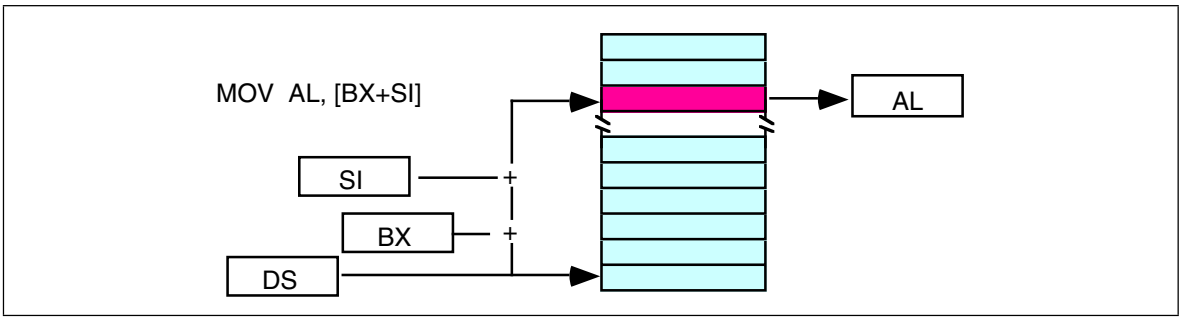

Figure 4.14 [BX+SI] Addressing Mode

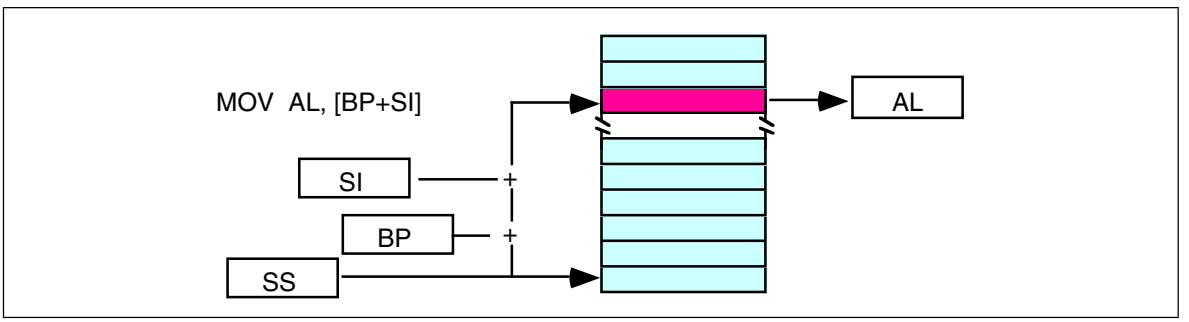

Figure 4.15 [BP+SI] Addressing Mode

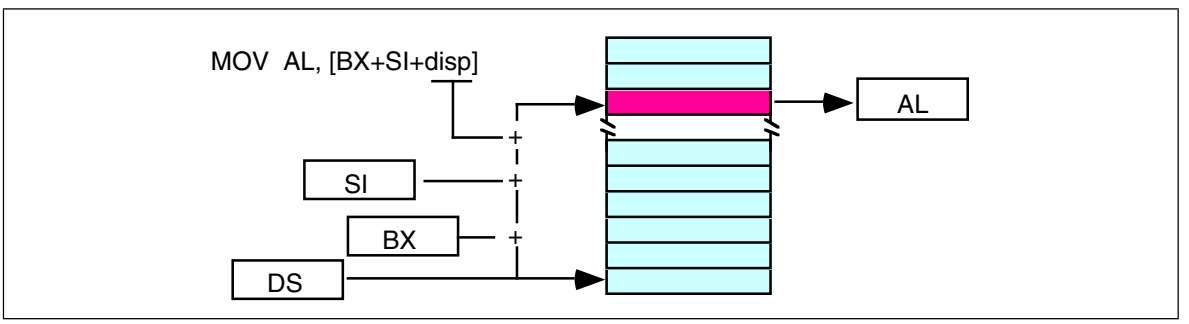

Figure 4.16 [BX + SI + disp] Addressing Mode

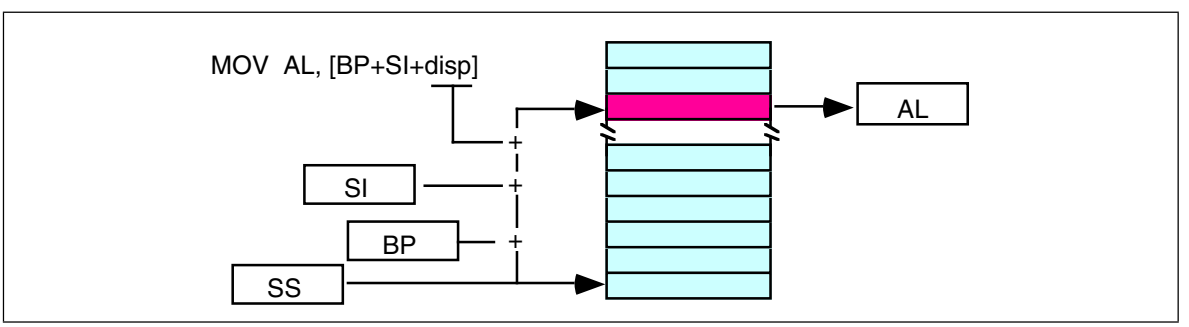

Figure 4.17 [BP + SI + disp] Addressing Mode

```
mov       al, disp[bx][si]
mov       al, disp[bx+di]
mov       al, [bp+si+disp]
mov       al, [bp][di][disp]
```

You may substitute di in Figure 4.12 to produce the [bx+di+disp] addressing mode. You may substitute di in Figure 4.12 to produce the [bp+di+disp] addressing mode.

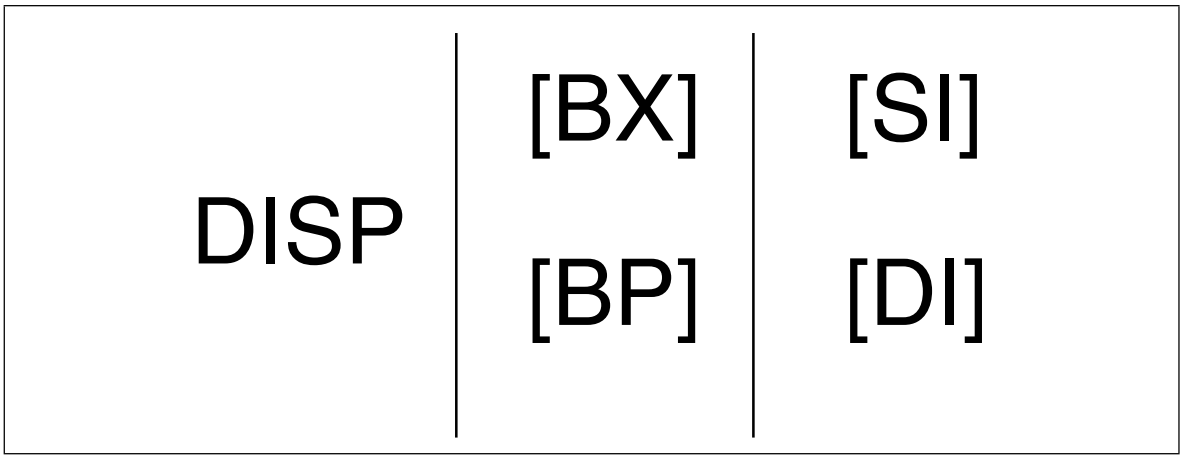

Figure 4.18 Table to Generate Valid 8086 Addressing Modes

Suppose bp contains 1000h, bx contains 2000h, si contains 120h, and di contains 5. Then mov al,10h[bx+si] loads al from address DS:2130; mov ch,125h[bp+di] loads ch from location SS:112A; and mov bx,cs:2[bx][di] loads bx from location CS:2007.

4.6.2.6 An Easy Way to Remember the 8086 Memory Addressing Modes

There are a total of 17 different legal memory addressing modes on the 8086: disp, [bx], [bp], [si], [di], disp[bx], disp[bp], disp[si], disp[di], [bx][si], [bx][di], [bp][si], [bp][di], disp[bx][si], disp [bx][di], disp[bp][si], and disp[bp][di][9]. You could memorize all these forms so that you know which are valid (and, by omission, which forms are invalid). However, there is an easier way besides memorizing these 17 forms. Consider the chart in Figure 4.12.

If you choose zero or one items from each of the columns and wind up with at least one item, you've got a valid 8086 memory addressing mode. Some examples:

- Choose disp from column one, nothing from column two, [di] from column 3, you get disp[di].
- Choose disp, [bx], and [di]. You get disp[bx][di].
- Skip column one & two, choose [si]. You get [si]
- Skip column one, choose [bx], then choose [di]. You get [bx][di]

Likewise, if you have an addressing mode that you *cannot* construct from this table, then it is not legal. For example, disp[dx][si] is illegal because you cannot obtain [dx] from any of the columns above.

4.6.2.7 Some Final Comments About 8086 Addressing Modes

The *effective address* is the final offset produced by an addressing mode computation. For example, if bx contains 10h, the effective address for 10h[bx] is 20h. You will see the term effective address in almost any discussion of the 8086's addressing mode. There is even a special instruction *load effective address* (lea) that computes effective addresses.

Not all addressing modes are created equal! Different addressing modes may take differing amounts of time to compute the effective address. The exact difference varies from processor to processor. Generally, though, the more complex an addressing mode is, the longer it takes to compute the effective address. Complexity of an addressing mode is directly related to the number of terms in the addressing mode. For example, disp[bx][si] is

9. That's not even counting the syntactical variations!

more complex than [bx]. See the instruction set reference in the appendices for information regarding the cycle times of various addressing modes on the different 80x86 processors.

The displacement field in all addressing modes *except* displacement-only can be a signed eight bit constant or a signed 16 bit constant. If your offset is in the range -128…+127 the instruction will be shorter (and therefore faster) than an instruction with a displacement outside that range. The size of the value in the register does not affect the execution time or size. So if you can arrange to put a large number in the register(s) and use a small displacement, that is preferable over a large constant and small values in the register(s).

If the effective address calculation produces a value greater than 0FFFFh, the CPU ignores the overflow and the result *wraps around* back to zero. For example, if bx contains 10h, then the instruction mov al,0FFFFh[bx] will load the al register from location ds:0Fh, not from location ds:1000Fh.

In this discussion you've seen how these addressing modes operate. The preceding discussion didn't explain *what you use them for*. That will come a little later. As long as you know how each addressing mode performs its effective address calculation, you'll be fine.

4.6.3 80386 Register Addressing Modes

The 80386 (and later) processors provide 32 bit registers. The eight general-purpose registers all have 32 bit equivalents. They are eax, ebx, ecx, edx, esi, edi, ebp, and esp. If you are using an 80386 or later processor you can use these registers as operands to several 80386 instructions.

4.6.4 80386 Memory Addressing Modes

The 80386 processor generalized the memory addressing modes. Whereas the 8086 only allowed you to use bx or bp as base registers and si or di as index registers, the 80386 lets you use almost any general purpose 32 bit register as a base or index register. Furthermore, the 80386 introduced new *scaled indexed* addressing modes that simplify accessing elements of arrays. Beyond the increase to 32 bits, the new addressing modes on the 80386 are probably the biggest improvement to the chip over earlier processors.

4.6.4.1 Register Indirect Addressing Modes

On the 80386 you may specify *any* general purpose 32 bit register when using the register indirect addressing mode. [eax], [ebx], [ecx], [edx], [esi], and [edi] all provide offsets, by default, into the data segment. The [ebp] and [esp] addressing modes use the stack segment by default.

Note that while running in 16 bit real mode on the 80386, offsets in these 32 bit registers must still be in the range 0…0FFFFh. You cannot use values larger than this to access more than 64K in a segment[10]. Also note that you must use the 32 bit names of the registers. You cannot use the 16 bit names. The following instructions demonstrate all the legal forms:

```
mov     al, [eax]
mov     al, [ebx]
mov     al, [ecx]
mov     al, [edx]
mov     al, [esi]
mov     al, [edi]
mov     al, [ebp]    ;Uses SS by default.
```

10. Unless, of course, you're operating in protected mode, in which case this is perfectly legal.

```
mov     al, [esp]    ;Uses SS by default.
```

4.6.4.2 80386 Indexed, Base/Indexed, and Base/Indexed/Disp Addressing Modes

The indexed addressing modes (register indirect plus a displacement) allow you to mix a 32 bit register with a constant. The base/indexed addressing modes let you pair up two 32 bit registers. Finally, the base/indexed/displacement addressing modes let you combine a constant and two registers to form the effective address. Keep in mind that the offset produced by the effective address computation must still be 16 bits long when operating in real mode.

On the 80386 the terms *base register* and *index register* actually take on some meaning. When combining two 32 bit registers in an addressing mode, the first register is the base register and the second register is the index register. This is true regardless of the register names. Note that the 80386 allows you to use the *same* register as both a base and index register, which is actually useful on occasion. The following instructions provide representative samples of the various base and indexed address modes along with syntactical variations:

```
mov     al, disp[eax]          ;Indexed addressing
mov     al, [ebx+disp]         ; modes.
mov     al, [ecx][disp]
mov     al, disp[edx]
mov     al, disp[esi]
mov     al, disp[edi]
mov     al, disp[ebp]          ;Uses SS by default.
mov     al, disp[esp]          ;Uses SS by default.
```

The following instructions all use the base+indexed addressing mode. The first register in the second operand is the base register, the second is the index register. If the *base* register is esp or ebp the effective address is relative to the stack segment. Otherwise the effective address is relative to the data segment. Note that the choice of index register does not affect the choice of the default segment.

```
mov     al, [eax][ebx]         ;Base+indexed addressing
mov     al, [ebx+ebx]          ; modes.
mov     al, [ecx][edx]
mov     al, [edx][ebp]         ;Uses DS by default.
mov     al, [esi][edi]
mov     al, [edi][esi]
mov     al, [ebp+ebx]          ;Uses SS by default.
mov     al, [esp][ecx]         ;Uses SS by default.
```

Naturally, you can add a displacement to the above addressing modes to produce the base+indexed+displacement addressing mode. The following instructions provide a representative sample of the possible addressing modes:

```
mov     al, disp[eax][ebx]     ;Base+indexed addressing
mov     al, disp[ebx+ebx]      ; modes.
mov     al, [ecx+edx+disp]
mov     al, disp[edx+ebp]      ;Uses DS by default.
mov     al, [esi][edi][disp]
mov     al, [edi][disp][esi]
mov     al, disp[ebp+ebx]      ;Uses SS by default.
mov     al, [esp+ecx][disp]    ;Uses SS by default.
```

There is one restriction the 80386 places on the index register. You cannot use the esp register as an index register. It's okay to use esp as the base register, but not as the index register.

4.6.4.3 80386 Scaled Indexed Addressing Modes

The indexed, base/indexed, and base/indexed/disp addressing modes described above are really special instances of the 80386 *scaled indexed addressing modes.* These addressing modes are particularly useful for accessing elements of arrays, though they are not limited to such purposes. These modes let you multiply the index register in the addressing mode by one, two, four, or eight. The general syntax for these addressing modes is

```
                disp[index*n]
                [base][index*n]
or
                disp[base][index*n]
```

where "base" and "index" represent any 80386 32 bit general purpose registers and "n" is the value one, two, four, or eight.

The 80386 computes the effective address by adding disp, base, and index*n together. For example, if ebx contains 1000h and esi contains 4, then

```
                mov al,8[ebx][esi*4]            ;Loads AL from location 1018h
                mov al,1000h[ebx][ebx*2]        ;Loads AL from location 4000h
                mov al,1000h[esi*8]             ;Loads AL from location 1020h
```

Note that the 80386 extended indexed, base/indexed, and base/indexed/displacement addressing modes really are special cases of this scaled indexed addressing mode with "n" equal to one. That is, the following pairs of instructions are absolutely identical to the 80386:

```
mov    al, 2[ebx][esi*1]            mov       al, 2[ebx][esi]
mov    al, [ebx][esi*1]             mov       al, [ebx][esi]
mov    al, 2[esi*1]                 mov       al, 2[esi]
```

Of course, MASM allows lots of different variations on these addressing modes. The following provide a small sampling of the possibilities:

```
disp[bx][si*2], [bx+disp][si*2], [bx+si*2+disp], [si*2+bx][disp],
disp[si*2][bx], [si*2+disp][bx], [disp+bx][si*2]
```

4.6.4.4 Some Final Notes About the 80386 Memory Addressing Modes

Because the 80386's addressing modes are more orthogonal, they are much easier to memorize than the 8086's addressing modes. For programmers working on the 80386 processor, there is always the temptation to skip the 8086 addressing modes and use the 80386 set exclusively. However, as you'll see in the next section, the 8086 addressing modes really are more efficient than the comparable 80386 addressing modes. Therefore, it is important that you know *all* the addressing modes and choose the mode appropriate to the problem at hand.

When using base/indexed and base/indexed/disp addressing modes on the 80386, without a scaling option (that is, letting the scaling default to "*1"), the first register appearing in the addressing mode is the base register and the second is the index register. This is an important point because the choice of the default segment is made by the choice of the base register. If the base register is ebp or esp, the 80386 defaults to the stack segment. In all other cases the 80386 accesses the data segment by default, *even if the index register is ebp*. If you use the scaled index operator ("*n") on a register, that register is always the index register regardless of where it appears in the addressing mode:

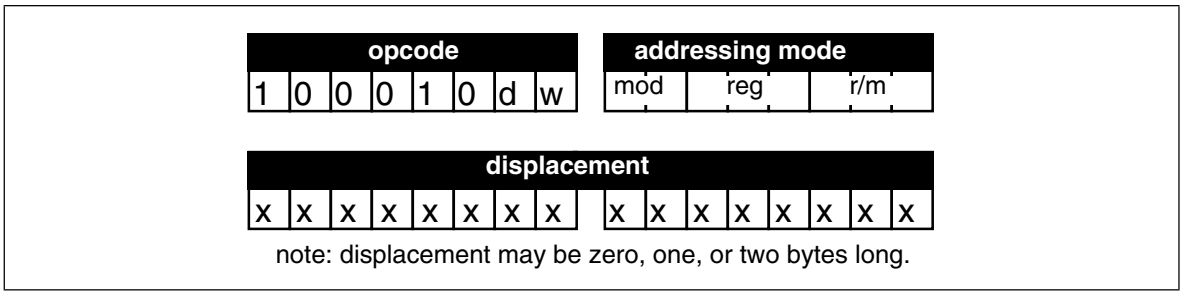

Figure 4.19 Generic MOV Instruction

```
[ebx][ebp]          ;Uses DS by default.
[ebp][ebx]          ;Uses SS by default.
[ebp*1][ebx]        ;Uses DS by default.
[ebx][ebp*1]        ;Uses DS by default.
[ebp][ebx*1]        ;Uses SS by default.
[ebx*1][ebp]        ;Uses SS by default.
es:[ebx][ebp*1]     ;Uses ES.
```

4.7 The 80x86 MOV Instruction

The examples throughout this chapter will make extensive use of the 80x86 mov (move) instruction. Furthermore, the mov instruction is the most common 80x86 machine instruction. Therefore, it's worthwhile to spend a few moments discussing the operation of this instruction.

Like it's x86 counterpart, the mov instruction is very simple. It takes the form:

```
mov     Dest,Source
```

Mov makes a copy of *Source* and stores this value into *Dest*. This instruction does not affect the original contents of *Source.* It overwrites the previous value in *Dest*. For the most part, the operation of this instruction is completely described by the Pascal statement:

```
Dest := Source;
```

This instruction has many limitations. You'll get ample opportunity to deal with them throughout your study of 80x86 assembly language. To understand why these limitations exist, you're going to have to take a look at the machine code for the various forms of this instruction. One word of warning, they don't call the 80386 a CISC (Complex Instruction Set Computer) for nothing. The encoding for the `mov` instruction is probably the most complex in the instruction set. Nonetheless, without studying the machine code for this instruction you will not be able to appreciate it, nor will you have a good understanding of how to write optimal code using this instruction. You'll see why you worked with the x86 processors in the previous chapters rather than using actual 80x86 instructions.

There are several versions of the mov instruction. The mnemonic[11] mov describes over a dozen different instructions on the 80386. The most commonly used form of the mov instruction has the following binary encoding shown in Figure 4.19.

The opcode is the first eight bits of the instruction. Bits zero and one define the *width* of the instruction (8, 16, or 32 bits) and the *direction* of the transfer. When discussing specific instructions this text will always fill in the values of *d* and *w* for you. They appear here only because almost every other text on this subject requires that *you* fill in these values.

Following the opcode is the addressing mode byte, affectionately called the "mod-reg-r/m" byte by most programmers. This byte chooses which of 256 different pos-

11. Mnemonic means *memory aid.* This term describes the English names for instructions like MOV, ADD, SUB, etc., which are much easier to remember than the hexadecimal encodings for the machine instructions.

sible operand combinations the generic mov instruction allows. The generic mov instruction takes three different assembly language forms:

```
mov         reg, memory
mov         memory, reg
mov         reg, reg
```

Note that at least one of the operands is always a general purpose register. The *reg* field in the mod/reg/rm byte specifies that register (or one of the registers if using the third form above). The *d* (direction) bit in the opcode decides whether the instruction stores data into the register (d=1) or into memory (d=0).

The bits in the reg field let you select one of eight different registers. The 8086 supports 8 eight bit registers and 8 sixteen bit general purpose registers. The 80386 also supports eight 32 bit general purpose registers. The CPU decodes the meaning of the *reg* field as follows:

Table 23: REG Bit Encodings

reg	w=0	16 bit mode w=1	32 bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

To differentiate 16 and 32 bit register, the 80386 and later processors use a special opcode prefix byte before instructions using the 32 bit registers. Otherwise, the instruction encodings are the same for both types of instructions.

The r/m field, in conjunction with the mod field, chooses the addressing mode. The mod field encoding is the following:

Table 24: MOD Encoding

MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16 bit signed displacement (in 16 bit mode) or a 32 bit signed displacement (in 32 bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

The mod field chooses between a register-to-register move and a register-to/from-memory move. It also chooses the size of the displacement (zero, one, two, or four bytes) that follows the instruction for memory addressing modes. If MOD=00, then you have one of the addressing modes without a displacement (register indirect or base/indexed). Note the special case where MOD=00 and r/m=110. This would normally correspond to the [bp]

addressing mode. The 8086 uses this encoding for the displacement-only addressing mode. This means that *there isn't a true [bp] addressing mode on the 8086.*

To understand why you can use the [bp] addressing mode in your programs, look at MOD=01 and MOD=10 in the above table. These bit patterns activate the disp[reg] and the disp[reg][reg] addressing modes. "So what?" you say. "That's not the same as the [bp] addressing mode." And you're right. However, consider the following instructions:

```
mov     al, 0[bx]
mov     ah, 0[bp]
mov     0[si], al
mov     0[di], ah
```

These statements, using the indexed addressing modes, perform the same operations as their register indirect counterparts (obtained by removing the displacement from the above instructions). The only real difference between the two forms is that the indexed addressing mode is one byte longer (if MOD=01, two bytes longer if MOD=10) to hold the displacement of zero. Because they are longer, these instructions may also run a little slower.

This trait of the 8086 – providing two or more ways to accomplish the same thing – appears throughout the instruction set. In fact, you're going to see several more examples before you're through with the mov instruction. MASM generally picks the best form of the instruction automatically. Were you to enter the code above and assemble it using MASM, it would still generate the register indirect addressing mode for all the instructions except mov ah,0[bp]. It would, however, emit only a one-byte displacement that is shorter and faster than the same instruction with a two-byte displacement of zero. Note that MASM does not require that you enter 0[bp], you can enter [bp] and MASM will automatically supply the zero byte for you.

If MOD does not equal 11b, the r/m field encodes the memory addressing mode as follows:

Table 25: R/M Field Encoding

R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

Don't forget that addressing modes involving bp use the stack segment (ss) by default. All others use the data segment (ds) by default.

If this discussion has got you totally lost, you haven't even seen the worst of it yet. Keep in mind, these are just *some* of the 8086 addressing modes. *You've still got all the 80386 addressing modes to look at*. You're probably beginning to understand what they mean when they say *complex* instruction set computer. However, the important concept to note is that you can construct 80x86 instructions the same way you constructed x86 instructions in Chapter Three – by building up the instruction bit by bit. For full details on how the 80x86 encodes instructions, see the appendices.

4.8 Some Final Comments on the MOV Instructions

There are several important facts you should always remember about the mov instruction. First of all, *there are no memory to memory moves*. For some reason, newcomers to assembly language have a hard time grasping this point. While there are a couple of instructions that perform memory to memory moves, loading a register and then storing that register is almost always more efficient. Another important fact to remember about the mov instruction is that there are many different mov instructions that accomplish the same thing. Likewise, there are several different addressing modes you can use to access the same memory location. If you are interested in writing the shortest and fastest possible programs in assembly language, you must be constantly aware of the trade-offs between equivalent instructions.

The discussion in this chapter deals mainly with the generic mov instruction so you can see how the 80x86 processors encode the memory and register addressing modes into the mov instruction. Other forms of the mov instruction let you transfer data between 16-bit general purpose registers and the 80x86 segment registers. Others let you load a register or memory location with a constant. These variants of the mov instruction use a different opcode. For more details, see the instruction encodings in Appendix D.

There are several additional mov instructions on the 80386 that let you load the 80386 special purpose registers. This text will not consider them. There are also some string instructions on the 80x86 that perform memory to memory operations. Such instructions appear in the next chapter. They are not a good substitute for the mov instruction.

4.9 Laboratory Exercises

It is now time to begin working with actual 80x86 assembly language. To do so, you will need to learn how to use several assembly-language related software development tools. In this set of laboratory exercises you will learn how to use the basic tools to edit, assemble, debug, and run 80x86 assembly language programs. These exercises assume that you have already installed MASM (Microsoft's Macro Assembler) on your system. If you have not done so already, please install MASM (following Microsoft's directions) before attempting the exercises in this laboratory.

4.9.1 The UCR Standard Library for 80x86 Assembly Language Programmers

Most of the programs in this textbook use a set of standard library routines created at the University of California, Riverside. These routines provide standardized I/O, string handling, arithmetic, and other useful functions. The library itself is very similar to the C standard library commonly used by C/C++ programmers. Later chapters in this text will describe many of the routines found in the library, there is no need to go into that here. However, many of the example programs in this chapter and in later chapters will use certain library routines, so you must install and activate the library at this time.

The library appears on the companion CD-ROM. You will need to copy the library from CD-ROM to the hard disk. A set of commands like the following (with appropriate adjustments for the CD-ROM drive letter) will do the trick:

```
c:
cd \
md stdlib
cd stdlib
xcopy r:\stdlib\*.* . /s
```

Once you've copied the library to your hard disk, there are two additional commands you must execute before attempting to assemble any code that uses the standard library:

```
set include=c:\stdlib\include
set lib=c:\stdlib\lib
```

It would probably be a good idea to place these commands in your autoexec.bat file so they execute automatically every time you start up your system. If you have not set the include and lib variables, MASM will complain during assembly about missing files.

4.9.2 Editing Your Source Files

Before you can assemble (compile) and run your program, you must create an assembly language source file with an editor. MASM will properly handle any ASCII text file, so it doesn't matter what editor you use to create that file as long as that editor processes ASCII text files. Note that most word processors *do not* normally work with ASCII text files, therefore, you should not use a word processor to maintain your assembly language source files.

MS-DOS, Windows, and MASM all three come with simple text editors you can use to create and modify assembly language source files. The EDIT.EXE program comes with MS-DOS; The NOTEPAD.EXE application comes with Windows; and the PWB (Programmer's Work Bench) comes with MASM. If you do not have a favorite text editor, feel free to use one of these programs to edit your source code. If you have some language system (e.g., Borland C++, Delphi, or MS Visual C++) you can use the editor they provide to prepare your assembly language programs, if you prefer.

Given the wide variety of possible editors out there, this chapter will not attempt to describe how to use any of them. If you've never used a text editor on the PC before, consult the appropriate documentation for that text editor.

4.9.3 The SHELL.ASM File

Although you can write an assembly language program completely from scratch within your text editor of choice, most assembly language programs contain a large number of statements common to every assembly language program. In the Chapter Four directory on the companion CD-ROM there is a "SHELL.ASM" text file. The SHELL.ASM file is a skeleton assembly language file[12]. That is, it contains all the "overhead" instructions necessary to create a working assembly language program with the exception of the instructions and variables that make up that specific program. In many respects, it is comparable to the following Pascal program:

```
program shell(input,output);
begin
end.
```

Which is to say that SHELL.ASM is a valid program. You can assemble and run it but it won't do very much.

The main reason for the SHELL.ASM program is that there are lots of lines of code that must appear in an empty assembly language program just to make the assembler happy. Unfortunately, to understand what these instructions mean requires considerable study. Rather than put off writing any programs until you understand everything necessary to create your first program, you're going to blindly use the SHELL.ASM file without questioning what any of it means. Fear not. Within a couple chapters it will all make sense. But for now, just type it in and use it exactly as it appears. The only thing you need to know about SHELL.ASM right away is where to place your code in this file. That's easy to see, though; there are three comments in the file telling you where to put your variables (if any), subroutine/procedures/functions (if any), and the statements for your main pro-

12. This file is available on the companion CD-ROM.

gram. The following is the complete listing of the SHELL.ASM file for those who may not have access to the electronic version:

```
                        .xlist
                        include     stdlib.a
                        includelib  stdlib.lib
                        .list

dseg                    segment     para public 'data'

; Global variables go here:

dseg                    ends

cseg                    segment     para public 'code'
                        assume      cs:cseg, ds:dseg

; Variables that wind up being used by the standard library routines.
; The MemInit routine uses "PSP" and "zzzzzzseg" labels. They must be
; present if you intend to use getenv, MemInit, malloc, and free.

                        public      PSP
PSP                     dw          ?

;-----------------------------------------
; Here is a good place to put other routines:
;-----------------------------------------------------------------
; Main is the main program. Program execution always begins here.

Main                    proc
                        mov         cs:PSP, es      ;Save pgm seg prefix
                        mov         ax, seg dseg    ;Set up the segment
registers
                        mov         ds, ax
                        mov         es, ax

                        mov         dx, 0
                        meminit
                        jnc         GoodMemInit

                        print
                        db          "Error initializing memory
manager",cr,lf,0
                        jmp         Quit
GoodMemInit:

;*****************************************************************************
; Put your main program here.
;*****************************************************************************

Quit:                   ExitPgm
Main                    endp
cseg                    ends

; Allocate a reasonable amount of space for the stack (2k).

sseg                    segment     para stack 'stack'
stk                     db          256 dup ("stack ")
sseg                    ends

; zzzzzzseg must be the last segment that gets loaded into memory!

zzzzzzseg               segment     para public 'zzzzzz'
LastBytes               db          16 dup (?)
zzzzzzseg               ends
                        end         Main
```

Although you're supposed to simply accept this code as-is and without question, a few explanations are in order. The program itself begins with a pair of "include" and "includelib" statements. These statements tell the assembler and linker that this code will be using some of the library routines from the "UCR Standard Library for 80x86 Assembly Language Programmers." This library appears on the companion CD-ROM.

Note that text beginning with a semicolon (";") is a comment. The assembler ignores all the text from the semicolon to the end of the line. As with high level languages, comments are very important for explaining the operation of your program. In this example, the comments point out some important parts of the SHELL.ASM program[13].

The next section of interest is the line that begins with `dseg segment` .... This is the beginning of your global data area. This statement defines the beginning of a data segment (*dseg* stands for data segment) that ends with the `dseg ends` statement. You should place all your global variables between these two statements.

Next comes the code segment (it's called *cseg*) where the 80x86 instructions go. The important thing to note here is the comment "Put your main program here." For now, you should ignore everything else in the code segment except this one comment. The sequences of assembly language statements you create should go between the lines of asterisks surrounding this comment. Have no fear; you'll learn what all these statements mean in the next two chapters. Attempting to explain them now would simply be too much of a digression.

Finally come two additional segments in the program: `sseg` and `zzzzzzseg`. These segments are absolutely necessary (the system requires sseg, the UCR Standard Library requires `zzzzzzseg`). You should not modify these segments.

When you begin writing a new assembly language program you should *not* modify the SHELL.ASM file directly. You should first make a copy of SHELL.ASM using the DOS copy command. For example, you might copy the file to PROJECT1.ASM and then make all your modifications to this file. By doing this you will have an undisturbed copy of SHELL.ASM available for your next project.

There is a special version of SHELL.ASM, X86.ASM, that contains some additional code to support programming projects in this chapter. Please see the programming projects section for more details.

4.9.4 Assembling Your Code with MASM

To run MASM you use the ML.EXE (MASM and Link) program. This file is typically found in a directory with a name like C:\MASM611\BIN. You should check to see if your path includes this directory. If not, you should adjust the DOS shell path variable so that it includes the directory containing ML.EXE, LINK.EXE, CV.EXE, and other MASM-related programs.

MASM is a DOS-based program. The easiest way to run it is from DOS or from a DOS box inside Windows. The basic MASM command takes the following form:

```
ml {options} filename.asm
```

Note that the ML program requires that you type the ".asm" suffix to the filename when assembling an assembly language source file.

Most of the time, you will only use the "/Zi" option. This tells MASM to add symbolic debugging information to the .EXE file for use by CodeView. This makes the executable file somewhat larger, but it also makes tracing through a program with CodeView (see "Debuggers and CodeView™" on page 173) considerably easier. Normally, you will always use this option during development and skip using it when you want to produce an EXE file you can distribute.

Another useful option, one you would normally use without a filename, is "/?"– the help command. ML, if it encounters this option, will display a list of all the options ML.EXE accepts. Most of these options you will rarely, if ever, use. Consult the MASM documentation for more details on MASM command-line options.

13. By the way, when you create a program using SHELL.ASM it's always a good idea to delete comments like "Insert your global data here." These comments are for the benefit of people reading the SHELL.ASM file, not for people reading your programs. Such comments look really goofy in an actual program.

Typing a command of the form "ML /Zi mypgm.asm" produces two new files (assuming there were no errors): mypgm.obj and mypgm.exe. The OBJ (object code file) is an intermediate file the assembler and *linker* use. Most of the time you can delete this if you program consists of a single source file. The mypgm.exe file is the executable version of the program. You can run this program directly from DOS or run it through the CodeView debugger (often the best choice).

4.9.5 Debuggers and CodeView™

The SIMx86 program is an example of a very simple debugging program. It should come as no surprise that there are several debugger programs available for the 80x86 as well. In this chapter you will learn the basic operation of the CodeView debugger. CodeView is a professional product with many different options and features. This short chapter cannot begin to describe all the possible ways to use the CodeView debugger. However, you will learn how to use some of the more common CodeView commands and debugging techniques.

One major drawback to describing a system like CodeView is that Microsoft constantly updates the CodeView product. These updates create subtle changes in the appearance of several screen images and the operation of various commands. It's quite possible that you're using an older version of CodeView than the one described in this chapter, or this chapter describes an older version of CodeView than the one you're using (This Chapter uses CodeView v4.0). Well, don't let this concern you. The basic principles are the same and you should have no problem adjusting for version differences.

Note: this chapter assumes you are running CodeView from MS-DOS. If you are using a Windows version, the screens will look slightly different.

4.9.5.1 A Quick Look at CodeView

To run CodeView, simply type the following command at the DOS command line prompt:

```
c:> CV program.exe
```

Program.exe represents the name of the program you wish to debug (the ".exe" suffix is optional). CodeView requires that you specify a ".EXE" or ".COM" program name. If you do not supply an executable filename, CodeView will ask you to pick a file when you run it.

CodeView requires an executable program name as the command line parameter. Since you probably haven't written an executable assembly language program yet, you haven't got a program to supply to CodeView. To alleviate this problem, use the SHELL.EXE program found in the Chapter Four subdirectory. To run CodeView using SHELL.EXE just use the command "CV SHELL.EXE". This will bring up a screen which looks something like that in Figure 4.20.

There are four sections to the screen in Figure 4.20: the *menu bar* on the first line, the *source1* window, the *command* window, and the help/ status line. Note that CodeView has many windows other than the two above. CodeView remembers which windows were open the last time it was run, so it might come up displaying different windows than those above. At first, the Command window is the active window. However, you can easily switch between windows by pressing the F6 key on the keyboard.

The windows are totally configurable. The Windows menu lets you select which windows appear on the screen. As with most Microsoft windowing products, you select items on the menu bar by holding down the alt key and pressing the first letter of the menu you wish to open. For example, pressing alt-W opens up the Windows menu as shown in Figure 4.21.

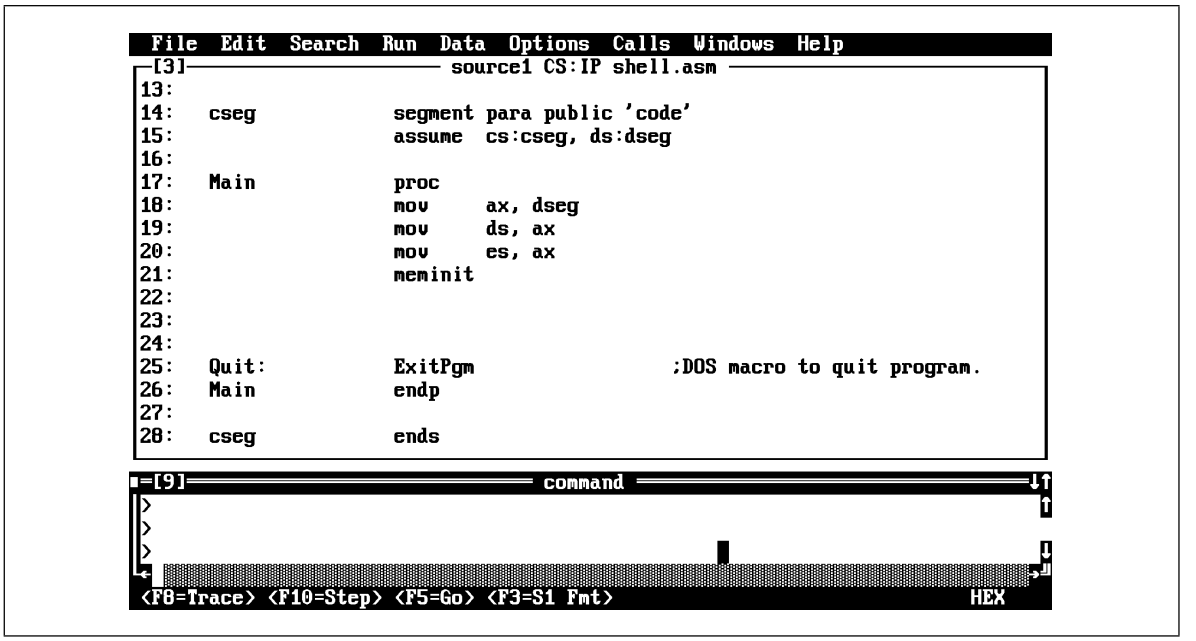

Figure 4.20 CodeView Debugger: An Initial Window

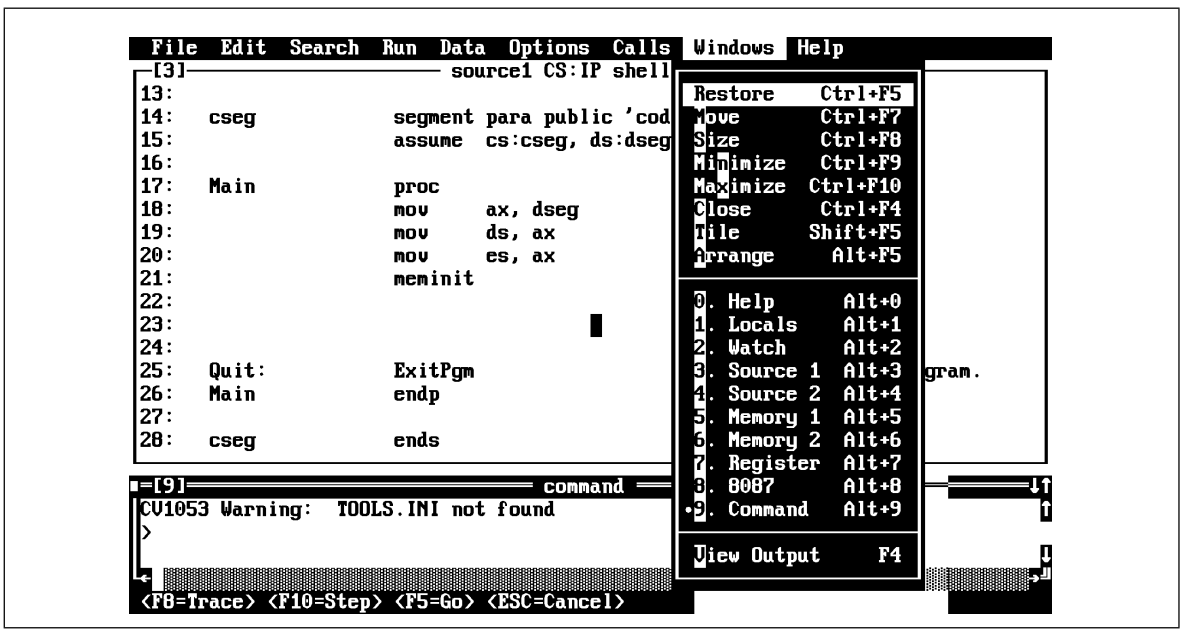

Figure 4.21 CodeView Window Menu (alt-W)

4.9.5.2 The Source Window

The Source1 and Source2 items let you open additional source windows. This lets you view, simultaneously, several different sections of the current program you're debugging. Source windows are useful for source level debugging.

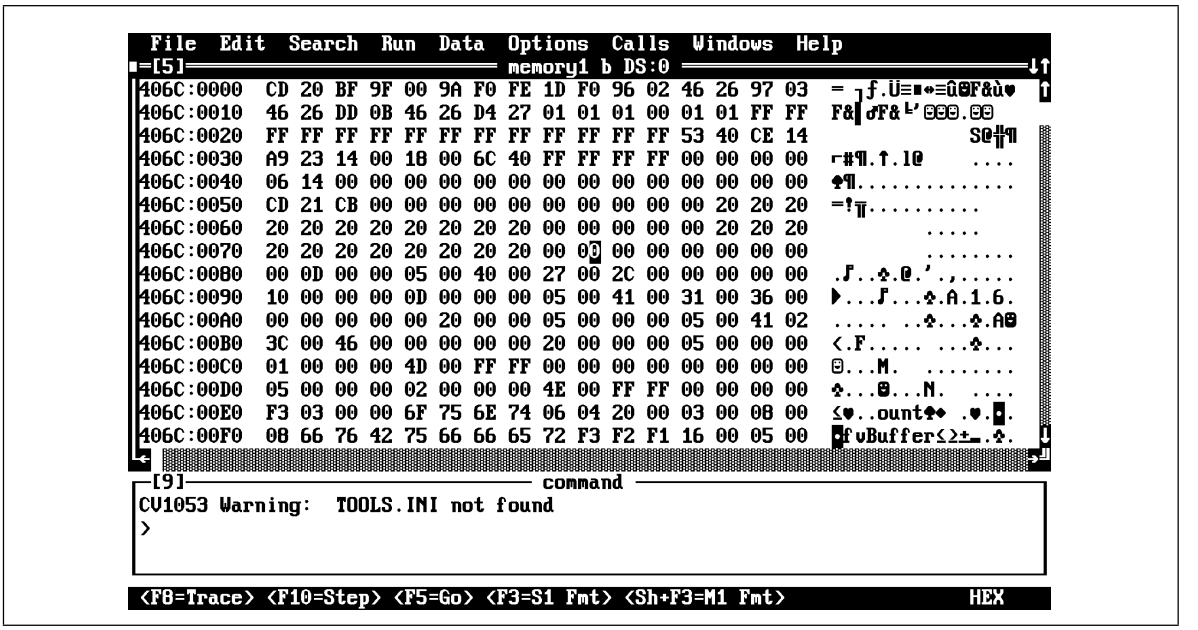

Figure 4.22 A Memory Display

4.9.5.3 The Memory Window

The Memory item lets you open a memory window. The memory windows lets you display and modify values in memory. By default, this window displays the variables in your data segment, though you can easily display any values in memory by typing their address.

Figure 4.22 is an example of a memory display.

The values on the left side of the screen are the segmented memory addresses. The columns of hexadecimal values in the middle of the screen represent the values for 16 bytes starting at the specified address. Finally, the characters on the right hand side of the screen represent the ASCII characters for each of the 16 bytes at the specified addresses. Note that CodeView displays a period for those byte values that are not printable ASCII characters.

When you first bring up the memory window, it typically begins displaying data at offset zero in your data segment. There are a coup.le of ways to display different memory locations. First, you can use the PgUp and PgDn keys to scroll through memory[14]. Another option is to move the cursor over a segment or offset portion of an address and type in a new value. As you type each digit, CodeView automatically displays the data at the new address.

If you want to modify values in memory, simply move the cursor over the top of the desired byte's value and type a new hexadecimal value. CodeView automatically updates the corresponding byte in memory.

CodeView lets you open multiple Memory windows at one time. Each time you select Memory from the View memory, CodeView will open up another Memory window. With multiple memory windows open you can compare the values at several non-contiguous memory locations on the screen at one time. Remember, if you want to switch between the memory windows, press the F6 key.

Pressing Shift-F3 toggles the data display mode between displaying hexadecimal bytes, ASCII characters, words, double words, integers (signed), floating point values, and

14. Mouse users can also move the thumb control on the scroll bar to achieve this same result.

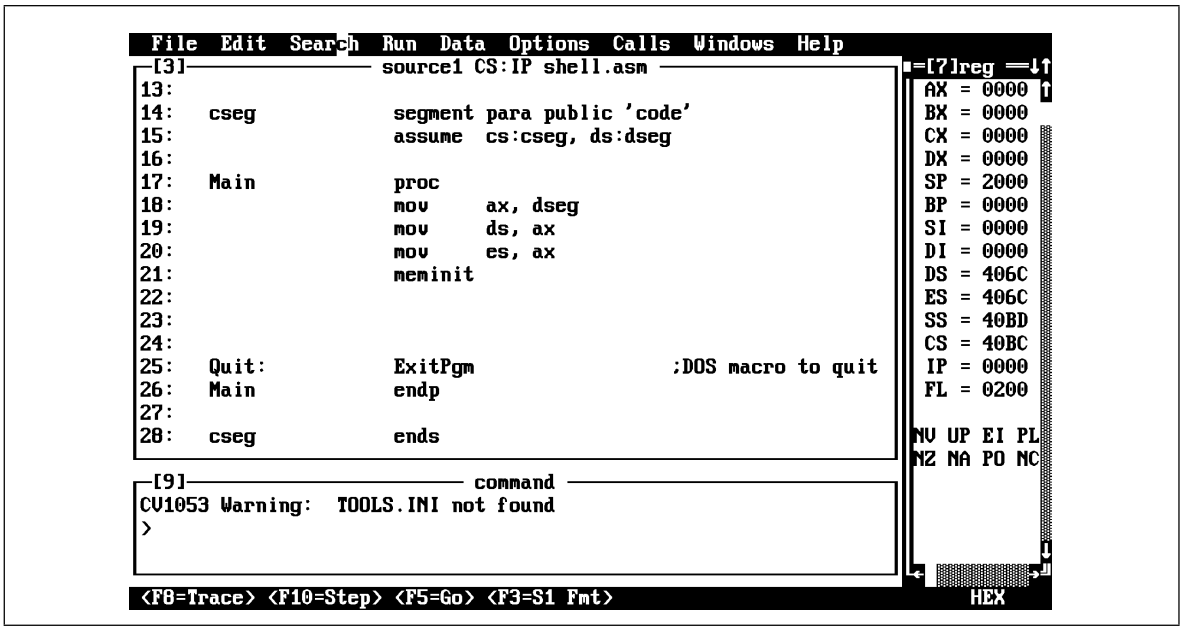

Figure 4.23 The Register Window

other data types. This is useful when you need to view memory using different data types. You only have the option of displaying the contents of the entire window as a single data type; however, you can open multiple memory windows and display a different data type in each one.

4.9.5.4 The Register Window

The Register item in the Windows menu displays or hides the 80x86 registers window. This windows displays the current 80x86 register values (see Figure 4.23).

To change the value of a register, activate the register window (using F6, if it is not already selected) and move the cursor over the value you wish to change. Type a new value over the desired register's existing value. Note that FL stands for *flags*. You can change the values of the flags in the flags register by entering a new value after the FL= entry. Another way to change the flags is to move the cursor over one of the flag entries at the bottom of the register window and press an alphabetic key (e.g., "A") on the keyboard. This will toggle the specified flag. The flag values are (0/1): overflow=(OV/NV), direction=(DN/UP), interrupt=(DI/EI), sign=(PL/NG), zero=(NZ/ZR), auxiliary carry=(NA/AC), parity=(PO/PE), carry=(NC/CY).

Note that pressing the F2 key toggles the display of the registers window. This feature is quite useful when debugging programs. The registers window eats up about 20% of the display and tends to obscure other windows. However, you can quickly recall the registers window, or make it disappear, by simply pressing F2.

4.9.5.5 The Command Window

The **Command** window lets you type textual commands into CodeView. Although almost every command available in the command window is available elsewhere, many operations are easier done in the command window. Furthermore, you can generally execute a sequence of completely different commands in the command window faster than switching between the various other windows in CodeView. The operation of the command window will be the subject of the next section in this chapter.

4.9.5.6 The Output Menu Item

Selecting **View Output** from the **Windows** menu (or pressing the F4 key) toggles the display between the CodeView display and the current program output. While your program is actually running, CodeView normally displays the program's output. Once the program turns control over to CodeView, however, the debugging windows appear obscuring your output. If you need to take a quick peek at the program's output while in CodeView, the F4 key will do the job.

4.9.5.7 The CodeView Command Window

CodeView is actually two debuggers in one. On the one hand, it is a modern window-based debugging system with a nice mouse-based user interface. On the other hand, it can behave like a traditional command-line based debugger. The command window provides the key to this split personality. If you activate the command window, you can enter debugger commands from the keyboard. The following are some of the more common CodeView commands you will use:

A *address*	Assemble
BC *bp_number*	Breakpoint Clear
BD *bp_number*	Breakpoint Disable
BE *bp_number*	Breakpoint Enable
BL	Breakpoint List
BP *address*	Breakpoint Set
D *range*	Dump Memory
E	Animate execution
E*x Address*	Enter Commands (x= " ", b, w, d, etc.)
G {*address*}	Go (address is optional)
H *command*	Help
I *port*	Input data from I/O port
L	Restart program from beginning
MC *range address*	Compare two blocks of memory
MF *range data_value(s)*	Fill Memory with specified value(s)
MM *range address*	Copy a block of memory
MS *range data_value(s)*	Search memory range for set of values
N $Value_{10}$	Set the default radix
O *port value*	Output value to an output port
P	Program Step
Q	Quit
R	Register
R*xx value*	Set register *xx* to *value*
T	Trace
U *address*	Unassemble statements at *address*

In this chapter we will mainly consider those commands that manipulate memory. Execution commands like the breakpoint , trace, and go commands appear in a later chapter. Of course, it wouldn't hurt for you to learn some of the other commands, you may find some of them to be useful.

4.9.5.7.1 The Radix Command (N)

The first command window command you must learn is the RADIX (base selection) command. By default, CodeView works in decimal (base 10). This is very inconvenient for assembly language programmers so you should always execute the radix command upon entering CodeView and set the base to hexadecimal. To do this, use the command

```
N 16
```

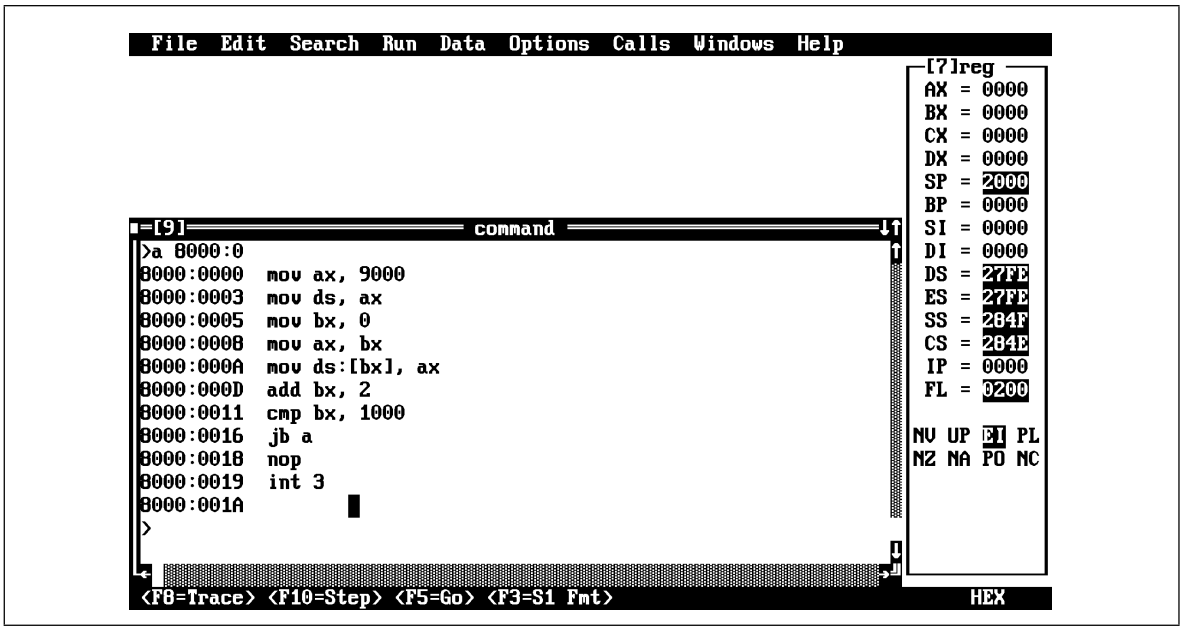

Figure 4.24 The Assemble Command

4.9.5.7.2 The Assemble Command

The CodeView command window **Assemble** command works in a fashion not unlike the SIM886 assemble command. The command uses the syntax:

```
A address
```

Address is the starting address of the machine instructions. This is either a full segmented address (*ssss:oooo*, *ssss* is the segment, *oooo* is the offset) or a simple offset value of the form *oooo*. If you supply only an offset, CodeView uses CS' current value as the segment address.

After you press Enter, CodeView will prompt you to enter a sequence of machine instructions. Pressing Enter by itself terminates the entry of assembly language instructions. Figure 4.24 is an example of this command in action.

The Assemble command is one of the few commands available *only* in the command window. Apparently, Microsoft does not expect programmers to enter assembly language code into memory using CodeView. This is not an unreasonable assumption since CodeView is a a high level language source level debugger.

In general, the CodeView Assemble command is useful for quick *patches* to a program, but it is no substitute for MASM 6.x. Any changes you make to your program with the assemble command will not appear in your source file. It's very easy to correct a bug in CodeView and forget to make the change to your original source file and then wonder why the bug is still in your code.

4.9.5.7.3 The Compare Memory Command

The Memory Compare command will compare the bytes in one block of memory against the bytes in a second block of memory. It will report any differences between the two ranges of bytes. This is useful, for example, to see if a program has initialized two arrays in an identical fashion or to compare two long strings. The compare command takes the following forms:

```
MC start_address end_address second_block_address
```

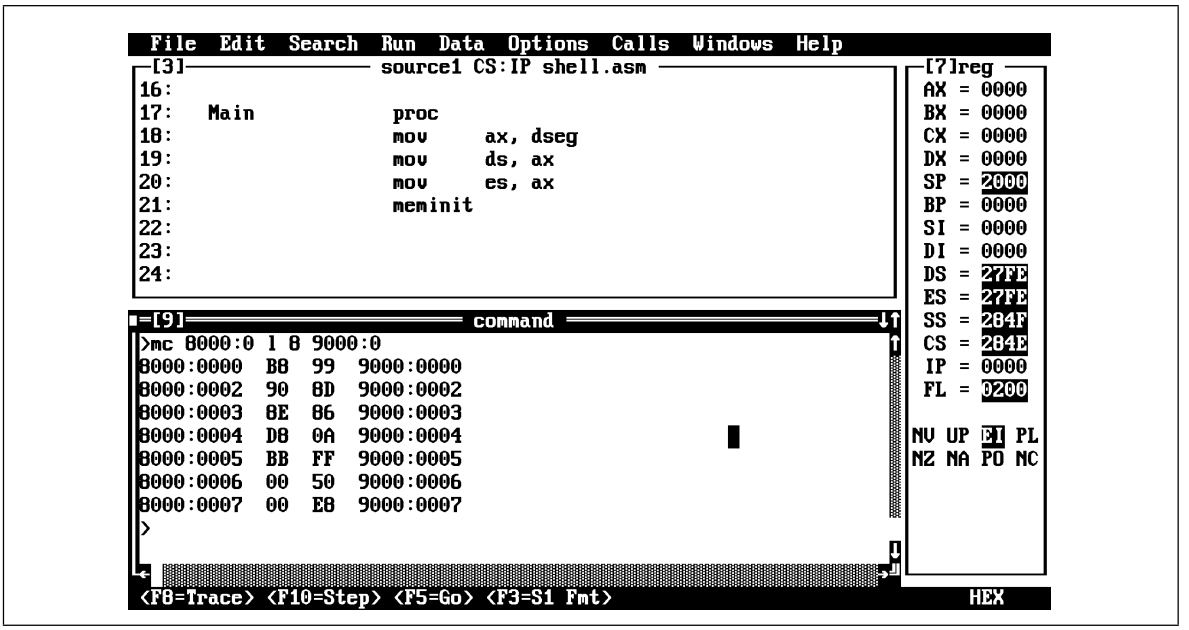

Figure 4.25 The Memory Compare Command

```
MC start_address L length_of_block second_block_address
```

The first form compares the bytes from memory locations *start_address* through *end_address* with the data starting at location *second_block_address.* The second form lets you specify the size of the blocks rather than specify the ending address of the first block. If CodeView detects any differences in the two ranges of bytes, it displays those differences and their addresses. The following are all legal compare commands:

```
MC 8000:0 8000:100 9000:80
MC 8000:100 L 20 9000:0
MC 0 100 200
```

The first command above compares the block of bytes from locations 8000:0 through 8000:100 against a similarly sized block starting at address 9000:80 (i.e., 9000:80..180).

The second command above demonstrates the use of the "L" option which specifies a length rather than an ending address. In this example, CodeView will compare the values in the range 8000:0..8000:1F (20h/32 bytes) against the data starting at address 9000:0.

The third example above shows that you needn't supply a full segmented address for the *starting_address* and *second_block_address* values. By default, CodeView uses the data segment (DS:) if you do not supply a segment portion of the address. Note, however, that if you supply a starting and ending address, they must both have the same segment value; you must supply the same segment address to both or you must let both addresses default to DS' value.

If the two blocks are equal, CodeView immediately prompts you for another command without printing anything to the command window. If there are differences between the two blocks of bytes, however, CodeView lists those differences (and their addresses) in the command window.

In the example in Figure 4.25, memory locations 8000:0 through 8000:200 were first initialized to zero. Then locations 8000:10 through 8000:1E were set to 1, 2, 3, ..., 0Fh. Finally, the Memory Compare command compared the bytes in the range 8000:0…8000:FF with the block of bytes starting at address 8000:100. Since locations 8000:10…8000:1E were different from the bytes at locations 8000:110…8000:11E, CodeView printed their addresses and differences.

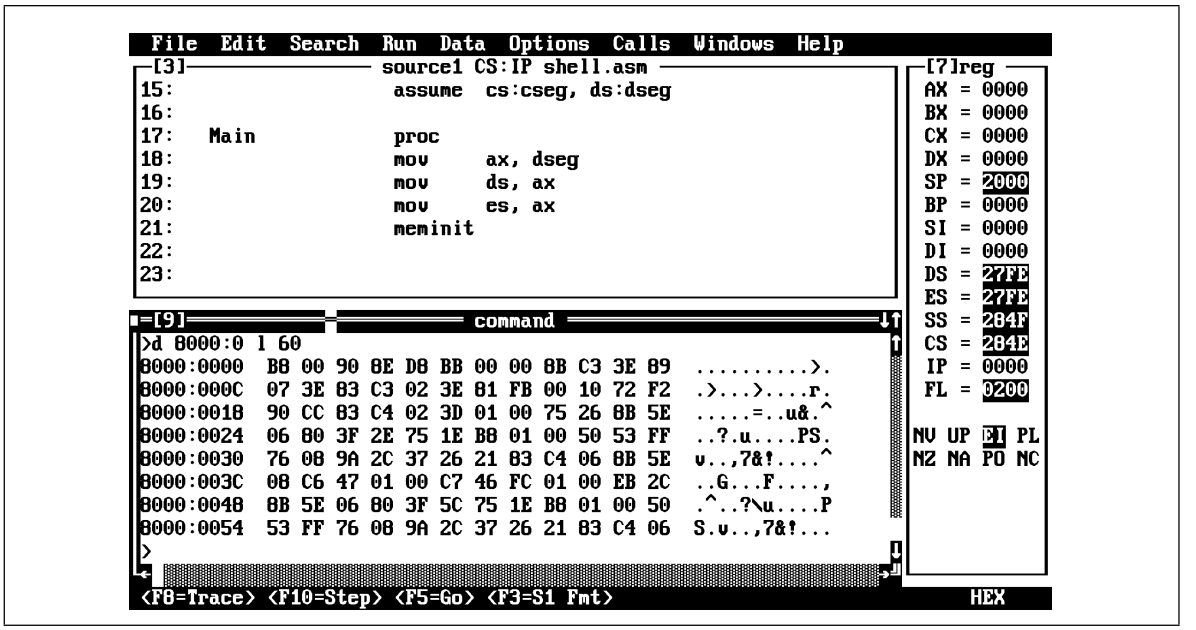

Figure 4.26 The Memory Dump Command

4.9.5.7.4 The Dump Memory Command

The Dump command lets you display the values of selected memory cells. The Memory window in CodeView also lets you view (and modify) memory. However, the Dump command is sometimes more convenient, especially when looking at small blocks of memory.

The Dump command takes several forms, depending on the type of data you want to display on the screen. This command typically takes one of the forms:

```
D starting_address ending_address
D starting_address L length
```

By default, the dump command displays 16 hexadecimal and ASCII byte values per line (just like the Memory window).

There are several additional forms of the Dump command that let you specify the display format for the data. However, the exact format seems to change with every version of CodeView. For example, in CodeView 4.10, you would use commands like the following:

```
DA address_range          Dump ASCII characters
DB address_range          Dump hex bytes/ASCII (default)
DI address_range          Dump integer words
DIU address_range         Dump unsigned integer words
DIX address_range         Dump 16-bit values in hex
DL address_range          Dump 32-bit integers
DLU address_range         Dump 32-bit unsigned integers
DLX address_range         Dump 32-bit values in hex
DR address_range          Dump 32-bit real values
DRL address_range         Dump 64-bit real values
DRT address_range         Dump 80-bit real values
```

You should probably check the help associated with your version of CodeView to verify the exact format of the memory dump commands. Note that some versions of CodeView allow you to use MDxx for the memory dump command.

Once you execute one of the above commands, the "D" command name displays the data in the new format. The "DB" command reverts back to byte/ASCII display. Figure 4.26 provides an example of these commands.

If you enter a dump command without an address, CodeView will display the data immediately following the last dump command. This is sometimes useful when viewing memory.

4.9.5.7.5 The Enter Command

The CodeView Memory windows lets you easily display and modify the contents of memory. From the command window it takes two different commands to accomplish these tasks: Dump to display memory data and Enter to modify memory data. For most memory modification tasks, you'll find the memory windows easier to use. However, the CodeView Enter command handles a few tasks easier than the Memory window.

Like the Dump command, the Enter command lets you enter data in several different formats. The commands to accomplish this are

```
EA-    Enter data in ASCII format
EB-    Enter byte data in hexadecimal format
ED-    Enter double word data in hexadecimal format
EI-    Enter 16-bit integer data in (signed) decimal format
EIU-   Enter 16-bit integer data in (unsigned) decimal format.
EIX-   Enter 16-bit integer data in hexadecimal format.
EL-    Enter 32-bit integer data in (signed) decimal format
ELU-   Enter 32-bit integer data in (unsigned) decimal format.
ELX-   Enter 32-bit integer data in hexadecimal format.
ER-    Enter 32-bit floating point data
ERL-   Enter 64-bit floating pont data
ERT-   Enter 80-bit floating point data
```

Like the Dump command, the syntax for this command changes regularly with different versions of CodeView. Be sure to use CodeView's help facility if these commands don't seem to work. *MExx* is a synonym for *Exx* in CodeView.

Enter commands take two possible forms:

```
Ex starting_address
Ex starting_address list_of_values
```

The first form above is the *interactive Enter command.* Upon pressing the key, Codeview will display the starting address and the data at that address, then prompt you to enter a new value for that location. Type the new value followed by a space and CodeView will prompt you for the value for the next location; typing a space by itself skips over the current location; typing the enter key or a value terminated with the enter key terminates the interactive Enter mode. Note that the EA command does not let you enter ASCII values in the interactive mode. It behaves exactly like the EB command during data entry.

The second form of the Enter command lets you enter a sequence of values into memory a single entry. With this form of the Enter command, you simply follow the starting address with the list of values you want to store at that address. CodeView will automatically store each value into successive memory locations beginning at the starting address. You can enter ASCII data using this form of Enter by enclosing the characters in quotes. Figure 4.27 demonstrates the use of the Enter command.

There are a couple of points concerning the Enter command of which you should be aware. First of all, you cannot use "E" as a command by itself . Unlike the Dump command, this does not mean "begin entering data after the last address." Instead, this is a totally separate command (Animate). The other thing to note is that the current display mode (ASCII, byte, word, double word, etc.) and the current entry mode are not independent. Changing the default display mode to word also changes the entry mode to word, and vice versa.

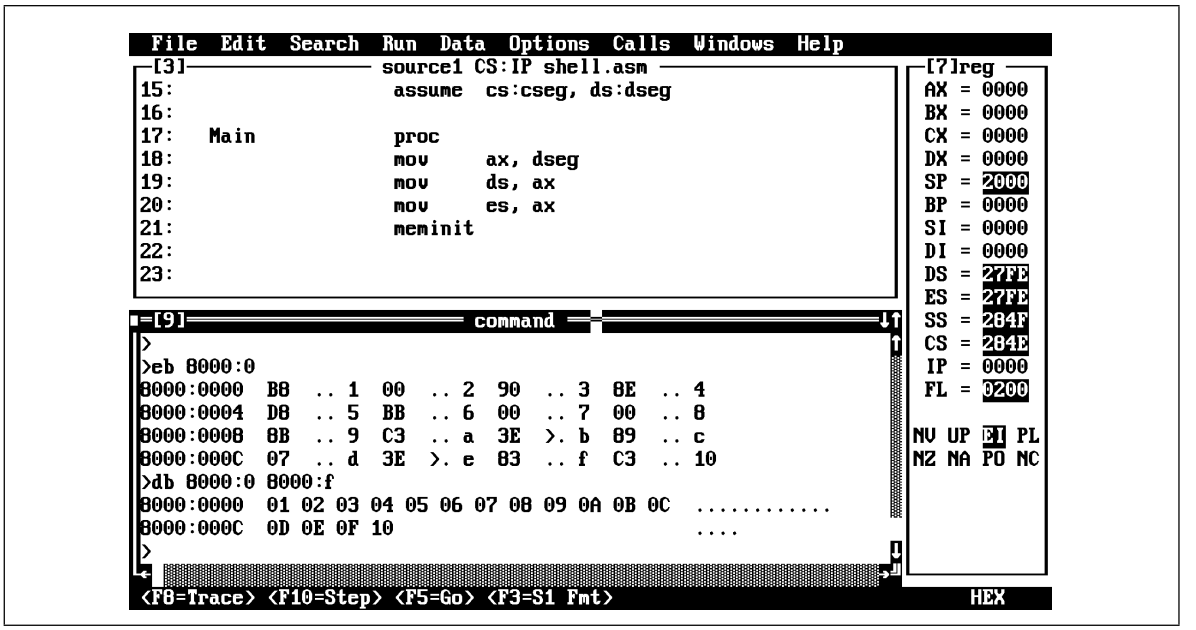

Figure 4.27 The Enter Command

4.9.5.7.6 The Fill Memory Command

The Enter command and the Memory window let you easily change the value of individual memory locations, or set a range of memory locations to several different values. If you want to clear an array or otherwise initialize a block of memory locations so that they all contain the same values, the Memory Fill command provides a better alternative.

The Memory Fill command uses the following syntax:

```
MF starting_address ending_address values
MF starting_address L block_length values
```

The Memory Fill command fills memory locations *starting_address* through *ending_address* with the byte values specified in the *values* list. The second form above lets you specify the block length rather than the ending address.

The *values* list can be a single value or a list of values. If *values* is a single byte value, then the Memory Fill command initializes all the bytes of the memory block with that value. If *values* is a list of bytes, the Fill command repeats that sequence of bytes over and over again in memory. For example, the following command stores 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5... to the 256 bytes starting at location 8000:0

```
F 8000:0 L 100 1 2 3 4 5
```

Unfortunately, the Fill command works only with byte (or ASCII string) data. However, you can simulate word, doubleword, etc., memory fills breaking up those other values into their component bytes. Don't forget, though, that the L.O. byte always comes first.

4.9.5.7.7 The Move Memory Command

This Command window operation copies data from one block of memory to another. This lets you copy the data from one array to another, move code around in memory, reinitialize a group of variables from a saved memory block, and so on. The syntax for the Memory Move command is as follows:

```
MM starting_address ending_address destination_address
MM starting_address L block_length destination_address
```

If the source and destination blocks overlap, CodeView detects this and handles the memory move operation correctly.

4.9.5.7.8 The Input Command

The Input command lets you read data from one of the 80x86's 65,536 different input ports. The syntax for this command is

```
I port_address
```

where *port_address* is a 16-bit value denoting the I/O port address to read. The input command reads the byte at that port and displays its value.

Note that it is not a wise idea to use this command with an arbitrary address. Certain devices activate some functions whenever you read one of their I/O ports. By reading a port you may cause the device to lose data or otherwise disturb that device.

Note that this command only reads a single byte from the specified port. If you want to read a word or double-word from a given input port you will need to execute two successive Input operations at the desired port address and the next port address.

This command appears to be broken in certain versions of CodeView (e.g., 4.01).

4.9.5.7.9 The Output Command

The Output command is complementary to the Input command. This command lets you output a data value to a port. It uses the syntax:

```
O port_address output_value
```

Output_value is a single byte value that CodeView will write to the output port given by *port_address*.

Note that CodeView also uses the "O" command to set options. If it does not recognize a valid port address as the first operand it will think this is an Option command. If the Output command doesn't seem to be working properly, you've probably switched out of the assembly language mode (CodeView supports BASIC, Pascal, C, and FORTRAN in addition to assembly language) and the port address you're entering isn't a valid numeric value in the new mode. Be sure to use the N 16 command to set the default radix to hexadecimal before using this command!

4.9.5.7.10 The Quit Command

Pressing Q (for Quit) terminates the current debugging session and returns control to MS-DOS. You can also quit CodeView by selecting the Exit item from the File menu.

4.9.5.7.11 The Register Command

The CodeView Register command lets you view and change the values of the registers. To view the current values of the 80x86 registers you would use the following command:

```
R
```

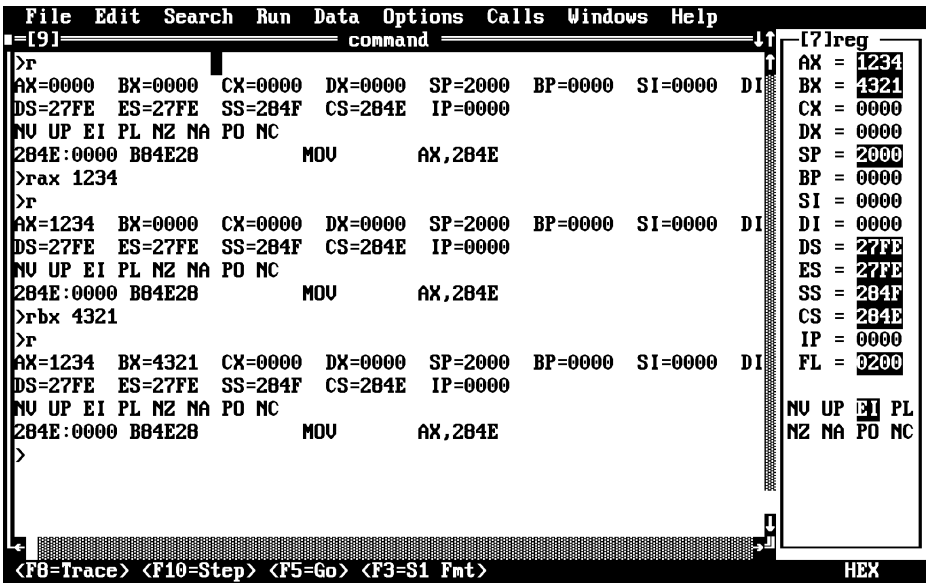

Figure 4.28 The Register Command

This command displays the registers and disassembles the instruction at address CS:IP.

You can also change the value of a specific register using a command of the form:

```
Rxx
-or-
Rxx = value
```

where *xx* represents one of the 80x86's register names: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, ES, SS, IP, or FL. The first version ("`Rxx`") displays the specified register and then prompts you to enter a new value. The second form of this command above immediately sets the specified register to the given value (see Figure 4.28).

4.9.5.7.12 The Unassemble Command

The Command window Unassemble command will disassemble a sequence of instructions at an address you specify, converting the binary machine codes into (barely) readable machine instructions. The basic command uses the following syntax:

`U` *address*

Note that you must have a source window open for this instruction to operate properly!

In general, the Unassemble command is of little use because the Source window lets you view your program at the source level (rather than at the disassembled machine language level). However, the Unassemble command is great for disassembling BIOS, DOS, TSRs, and other code in memory.

4.9.5.8 CodeView Function Keys

CodeView uses the function keys on the PC's keyboard for often-executed operations. The following table gives a brief description of the use of each function key.

Table 26: Function Key Usage in CodeView

Function Key	Alone	Shift	Ctrl	Alt
F1	Help	Help contents	Next Help	Prev Help
F2	Register Window			
F3	Source Window Mode	Memory Window Mode		
F4	Output Screen		Close Window	
F5	Run			
F6	Switch Window	Prev Window		
F7	Execute to cursor			
F8	Trace	Prev History	Size window	
F9	Breakpoint			
F10	Step instrs, run calls.	Next History	Maximize Window	

The F3 function key deserves special mention. Pressing this key toggles the source mode between *machine language* (actually, disassembled machine language), *mixed*, and *source*. In source mode (assuming you've assembled your code with the proper options) the source window displays your actual source code. In mixed mode, CodeView displays a line of source code followed by the machine code generated for that line of source code. This mode is primarily for high level language users, but it does have some utility for assembly language users as you'll see when you study macros. In *machine mode*, CodeView ignores your source code and simply disassembles the binary opcodes in memory. This mode is useful if you suspect a bug in MASM (they do exist) and you're not sure than MASM is assembling your code properly.

4.9.5.9 Some Comments on CodeView Addresses

The examples given for addresses in the previous sections are a little misleading. You could easily get the impression that you have to enter an address in hexadecimal form, i.e., *ssss:oooo* or *oooo*. Actually, you can specify memory addresses in many different ways. For example, if you have a variable in your assembly language program named *MyVar*, you could use a command like

```
D Myvar
```

to display the value of this variable[15]. You do not need to know the address, nor even the segment of that variable. Another way to specify an address is via the 80x86 register set. For example, if ES:BX points at the block of memory you want to display, you could use the following command to display your data:

```
D ES:BX
```

CodeView will use the current values in the `es` and `bx` registers as the address of the block of memory to display. There is nothing magical about the use of the registers. You can use them just like any other address component. In the example above, `es` held the segment value and `bx` held the offset— very typical for an 80x86 assembly language program.

15. This requires that you assemble your program in a very special way, but we're getting to that.

However, CodeView does not require you to use legal 80x86 combinations. For example, you could dump the bytes at address cx:ax using the dump command

```
D CX:AX
```

The use of 80x86 registers is not limited to specifying source addresses. You can specify destination addresses and even lengths using the registers:

```
D CX:AX L BX ES:DI
```

Of course, you can mix and match the use of registers and numeric addresses in the same command with no problem:

```
D CX:AX L 100 8000:0
```

You can also use complex arithmetic expressions to specify an address in memory. In particular, you can use the addition operator to compute the sum of various components of an address. This works out really neat when you need to simulate 80x86 addressing modes. For example, if you want to see which byte is at address 1000[bx], you could use the command:

```
D BX+1000 L 1
```

To simulate the [BX][SI] addressing mode and look at the word at that address you could use the command:

```
DIX BX+SI L 1
```

The examples presented in this section all use the Dump command, but you can use this technique with any of the CodeView commands. For more information concerning what constitutes valid CodeView address, as well as a full explanation of allowable expression forms, please consult the CodeView on-line help system.

4.9.5.10 A Wrap on CodeView

We're not through discussing CodeView by any means. In particular, we've not discussed the execution, single stepping, and breakpoint commands which are crucial for debugging programs. We will return to these subjects in later chapters. Nonetheless, we've covered a considerable amount of material, certainly enough to deal with most of the experiments in this laboratory exercise. As we need those other commands, this manual will introduce them.

Of course, there are two additional sources of information on CodeView available to you— the section on CodeView in the "Microsoft Macro Assembler Programmer's Guide" and the on-line help available inside CodeView. In particular, the on-line help is quite useful for figuring out how a specific command works inside CodeView.

4.9.6 Laboratory Tasks

The Chapter Four subdirectory on the companion CD-ROM contains a sample file named EX4_1.ASM. Assemble this program using MASM (do not use the /Zi option for the time being). **For your lab report:** include a print-out of the program. Describe what the program does. Run the program and include a print-out of the program's output with your lab report.

Whenever you assemble a program MASM, by default, writes one byte of data to the file for every instruction byte and data variable in the program, even if that data is uninitialized. If you declare large arrays in your program the EXE file ML produces will be quite large as well. Note the size of the EX4_1.EXE program you created above. Now reassemble the program using the following command:

```
ml EX4_1.asm /link /exepack
```

ML passes the "/link /exepack" option on to the linker. The exepack option tells the linker to pack the EXE file by removing redundant information (in particular, the unini-

tialized data). This often makes the EXE file much smaller. **For your lab report:** after assembling the file using the command above, note the size of the resulting EXE file. Compare the two sizes and comment on their difference in your lab report.

Note that the EXEPACK option only saves disk space. It does not make the program use any less memory while it is running. Furthermore, you cannot load programs you've packed with the EXEPACK option into the CodeView debugger. Therefore, you should not use the EXEPACK option during program development and testing. You should only use this option once you've eliminated all the bugs from the program and further development ceases.

Using your editor of choice, edit the x86.asm file. Read the comments at the beginning of the program that explain how to write x86 programs that assemble and run on the 80x86 CPU. **For your lab report:** describe the restrictions on the x86 programs you can write.

The EX4_2.ASM source file is a copy of the x86.ASM file with a few additional comments in the main program describing a set of procedures you should follow. Load this file into your text editor of choice and read the instructions in the main program. Follow them to produce a program. Assemble this program using ML and execute the resulting EX4_2.EXE program file. **For your lab report:** include a print-out of your resulting program. Include a print-out of the program's output when you run it.

Trying loading EX4_2.EXE into CodeView using the following DOS Window command:

```
cv EX4_2
```

When CodeView runs you will notice that it prints a message in the command window complaining that there is "no CodeView information for EX4_2.EXE." Look at the code in the source window. Try and find the instructions you place in the main program. **For your lab report:** contrast the program listing appearing in the CodeView source window with that produced on the Emulator screen of the SIMx86 program.

Now reassemble the EX4_2.asm file and load it into CodeView using the following DOS commands:

```
ml /Zi EX4_2.asm
cv EX4_2
```

For your lab report: describe the difference in the CodeView source window when using the /Zi ML option compared to the CodeView source window without this option.

4.10 Programming Projects

Note: You are to write these programs in 80x86 assembly language code using a copy of the X86.ASM file as the starting point for your programs. The 80x86 instruction set is almost a superset of the x86 instruction set. Therefore, you can use most of the instructions you learned in the last chapter. Read the comments at the beginning of the x86.ASM file for more details. **Note in particular that you cannot use the label "C" in your program because "C" is a reserved word in MASM.** Include a specification document, a test plan, a program listing, and sample output with your program submissions.

1) The following projects are modifications of the programming assignments in the previous chapter. Convert those x86 programs to their 80x86 counterparts.

1a. The x86 instruction set does not include a multiply instruction. Write a short program that reads two values from the user and displays their product (hint: remember that multiplication is just repeated addition).

1b. Write a program that reads three values from the user: an address it puts into BX, a count it puts into CX, and a value it puts in AX. It should write CX copies of AX to successive words in memory starting at address BX (in the data segment).

1c. Write the generic logic function for the x86 processor (see Chapter Two). Hint: add ax, ax does a shift left on the value in ax. You can test to see if the high order bit is set by checking to see if ax is greater than 8000h.

1d. Write a program that scans an array of words starting at address 1000h and memory, of the length specified by the value in cx, and locates the maximum value in that array. Display the value after scanning the array.

1e. Write a program that computes the two's complement of an array of values starting at location 1000h. CX should contain the number of values in the array. Assume each array element is a two-byte integer.

1f. Write a simple program that *sorts* the words in memory locations 1000..10FF in ascending order. You can use a simple *insertion sort* algorithm. The Pascal code for such a sort is

```
for i := 0 to n-1 do
       for j := i+1 to n do
                 if (memory[i] > memory[j]) then
                 begin
                           temp := memory[i];
                           memory[i] := memory[j];
                           memory[j] := temp;
                 end;
```

For the following projects, feel free to use any additional 80x86 addressing modes that might make the project easier to write.

2) Write a program that stores the values 0, 1, 2, 3, ..., into successive words in the data segment starting at offset 3000h and ending at offset 3FFEh (the last value written will be 7FFh). Then store the value 3000h to location 1000h. Next, write a code segment that sums the 512 words starting at the address found in location 1000h. This portion of the program cannot assume that 1000h contains 3000h. Print the sum and then quit.

4.11 Summary

This chapter presents an 80x86-centric view of memory organization and data structures. This certainly isn't a complete course on data structures, indeed this topic appears again later in Volume Two. This chapter discussed the primitive and simple composite data types and how to declare and use them in your program. Lots of additional information on the declaration and use of simple data types appears in "MASM: Directives & Pseudo-Opcodes" on page 355.

The 8088, 8086, 80188, 80186, and 80286 all share a common set of registers which typical programs use. This register set includes the general purpose registers: ax, bx, cx, dx, si, di, bp, and sp; the segment registers: cs, ds, es, and ss; and the special purpose registers ip and flags. These registers are 16 bits wide. These processors also have eight 8 bit registers: al, ah, bl, bh, cl, ch, dl, and dh which overlap the ax, bx, cx, and dx registers. See:

- "8086 General Purpose Registers" on page 146
- "8086 Segment Registers" on page 147
- "8086 Special Purpose Registers" on page 148

In addition, the 80286 supports several special purpose memory management registers which are useful in operating systems and other system level programs. See:

- "80286 Registers" on page 148

The 80386 and later processors extend the general purpose and special purpose register sets to 32 bits. These processors also add two additional segment registers you can use in your application programs. In addition to these improvements, which any program can take advantage of, the 80386/486 processors also have several additional system level registers for memory management, debugging, and processor testing. See:

- "80386/80486 Registers" on page 149

The Intel 80x86 family uses a powerful memory addressing scheme known as *segmented addressing* that provides simulated two dimensional addressing. This lets you group logically related blocks of data into segments. The exact format of these segments depends on whether the CPU is operating in *real mode* or *protected mode*. Most DOS programs operate in real mode. When working in real mode, it is very easy to convert a *logical* (segmented) address to a linear *physical* address. However, in protected mode this conversion is considerably more difficult. See:

- "Segments on the 80x86" on page 151

Because of the way segmented addresses map to physical addresses in real mode, it is quite possible to have two different segmented addresses that refer to the same memory location. One solution to this problem is to use normalized addresses. If two normalized addresses do not have the same bit patterns, they point at different addresses. Normalized pointers are useful when comparing pointers in real mode. See:

- "Normalized Addresses on the 80x86" on page 154

With the exception of two instructions, the 80x86 doesn't actually work with full 32 bit segmented addresses. Instead, it uses *segment registers* to hold default segment values. This allowed Intel's designers to build a much smaller instruction set since addresses are only 16 bits long (offset portion only) rather than 32 bits long. The 80286 and prior processors provide four segment registers: `cs`, `ds`, `es`, and `ss`; the 80386 and later provide six segment registers: `cs`, `ds`, `es`, `fs`, `gs`, and `ss`. See:

- "Segment Registers on the 80x86" on page 155

The 80x86 family provides many different ways to access variables, constants, and other data items. The name for a mechanism by which you access a memory location is *addressing mode.* The 8088, 8086, and 80286 processors provide a large set of memory addressing modes. See:

- "The 80x86 Addressing Modes" on page 155
- "8086 Register Addressing Modes" on page 156
- "8086 Memory Addressing Modes" on page 156

The 80386 and later processors provide an expanded set of register and memory addressing modes. See:

- "80386 Register Addressing Modes" on page 163
- "80386 Memory Addressing Modes" on page 163

The most common 80x86 instruction is the `mov` instruction. This instruction supports most of the addressing modes available on the 80x86 processor family. Therefore, the `mov` instruction is a good instruction to look at when studying the encoding and operation of 80x86 instructions. See:

- "The 80x86 MOV Instruction" on page 166

The `mov` instruction takes several generic forms, allowing you to move data between a register and some other location. The possible source/destination locations include: (1) other registers, (2) memory locations (using a general memory addressing mode), (3) constants (using the immediate addressing mode), and (4) segment registers.

The `mov` instruction lets you transfer data between two locations (although you cannot move data between two memory locations see the discussion of the mod-reg-r/m byte).

4.12 Questions

1) Although the 80x86 processors always use segmented addresses, the instruction encodings for instructions like "`mov AX, I`" only have a 16 bit offset encoded into the opcode. Explain.

2) Segmented addressing is best described as a *two dimensional addressing scheme*. Explain.

3) Convert the following logical addresses to physical addresses. Assume all values are hexadecimal and real mode operation on the 80x86:

   a) 1000:1000 b) 1234:5678 c) 0:1000 d) 100:9000 e) FF00:1000

   f) 800:8000 g) 8000:800 h) 234:9843 i) 1111:FFFF j) FFFF:10

4) Provide *normalized* forms of the logical addresses above.

5) List all the 8086 memory addressing modes.

6) List all the 80386 (and later) addressing mode that are not available on the 8086 (use generic forms like disp[reg], do no enumerate all possible combinations).

7) Besides memory addressing modes, what are the other two major addressing modes on the 8086?

8) Describe a common use for each of the following addressing modes:

   a) Register b) Displacement only c) Immediate

   d) Register Indirect e) Indexed f) Based indexed

   g) Based indexed plus displacement h) Scaled indexed

9) Given the bit pattern for the generic MOV instruction (see "The 80x86 MOV Instruction" on page 166) explain why the 80x86 does not support a memory to memory move operation.

10) Which of the following MOV instructions are *not* handled by the generic MOV instruction opcode? Explain.

    a) mov ax, bx b) mov ax, 1234 c) mov ax, I

    d) mov ax, [bx] e) mov ax, ds f) mov [bx], 2

11) Assume the variable "I" is at offset 20h in the data segment. Provide the binary encodings for the above instructions.

12) What determines if the R/M field specifies a register or a memory operand?

13) What field in the REG-MOD-R/M byte determines the size of the displacement following an instruction? What displacement sizes does the 8086 support?

14) Why doesn't the displacement only addressing mode support multiple displacement sizes?

15) Why would you *not* want to interchange the two instructions "mov ax, [bx]" and "mov ax,[ebx]"?

16) Certain 80x86 instructions take several forms. For example, there are two different versions of the MOV instruction that load a register with an immediate value. Explain why the designers incorporated this redundancy into the instruction set.

17) Why isn't there a true [bp] addressing mode?

18) List all of the 80x86 eight bit registers.

19) List all the 80x86 general purpose 16 bit registers.

20) List all the 80x86 segment registers (those available on all processors).

21) Describe the "special purposes" of each of the general purpose registers.

22) List all the 80386/486/586 32 bit general purpose registers.

23) What is the relationship between the 8, 16, and 32 bit general purpose registers on the 80386?

24) What values appear in the 8086 flags register? The 80286 flags register?

25) Which flags are the condition codes?

26) Which extra segment registers appear on the 80386 but not on earlier processors?