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.

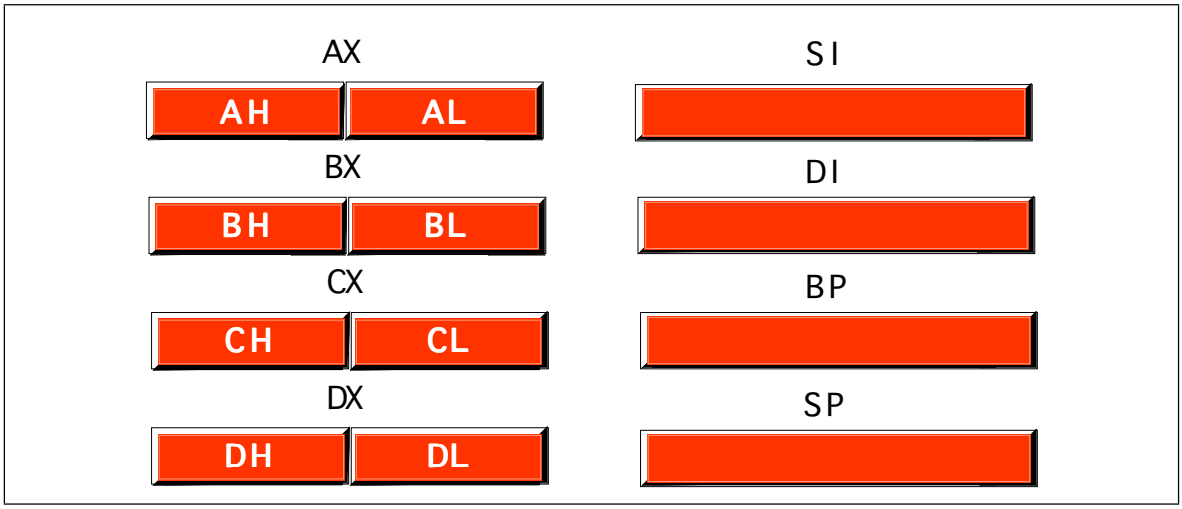

Figure 4.1 8086 Register Set

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.

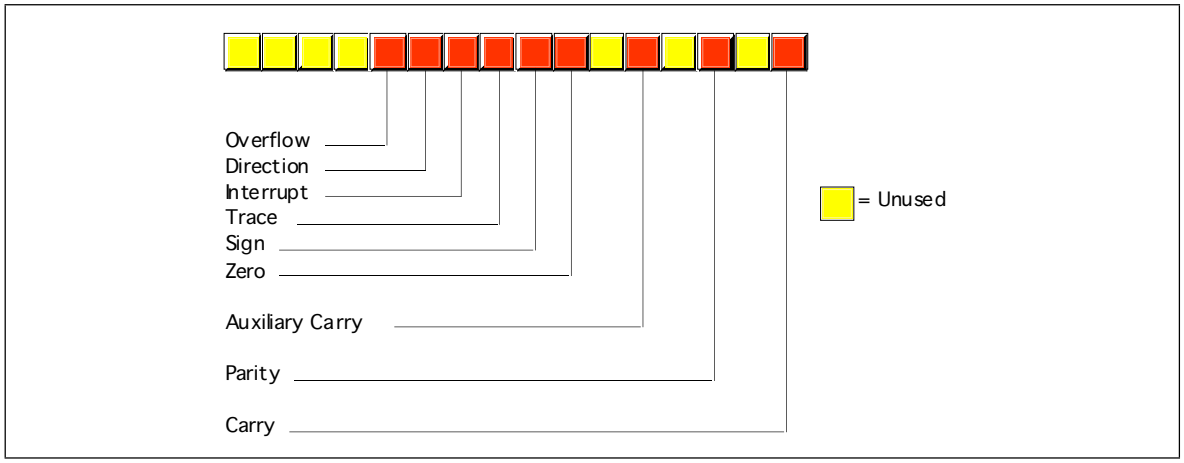

Figure 4.2 8086 Flags Register

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

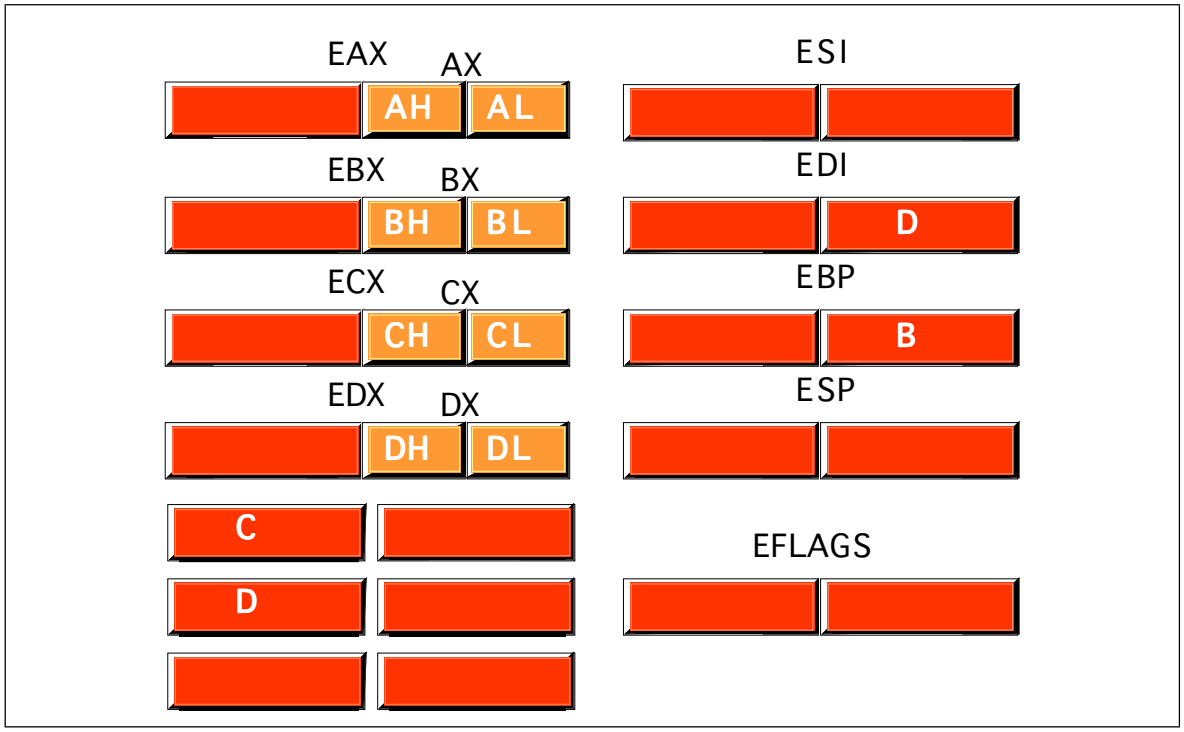

Figure 4.3 80386 Registers (Application Programmer Visible)

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 80s 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.

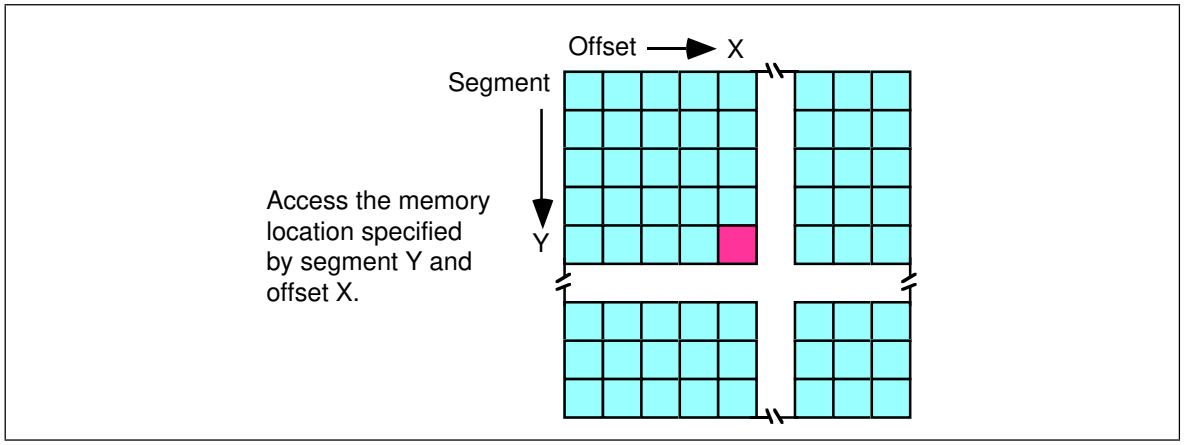

Figure 4.4 Segmented Addressing as a Two-Dimensional Process

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.

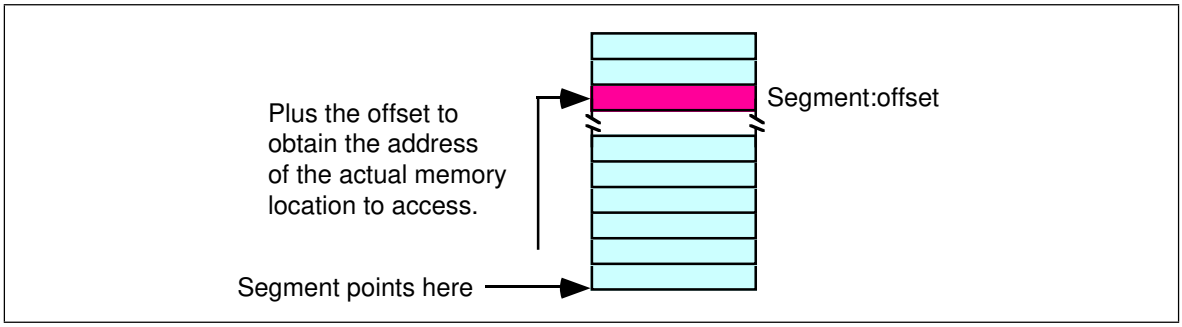

Figure 4.5 Segmented Addressing in Physical Memory

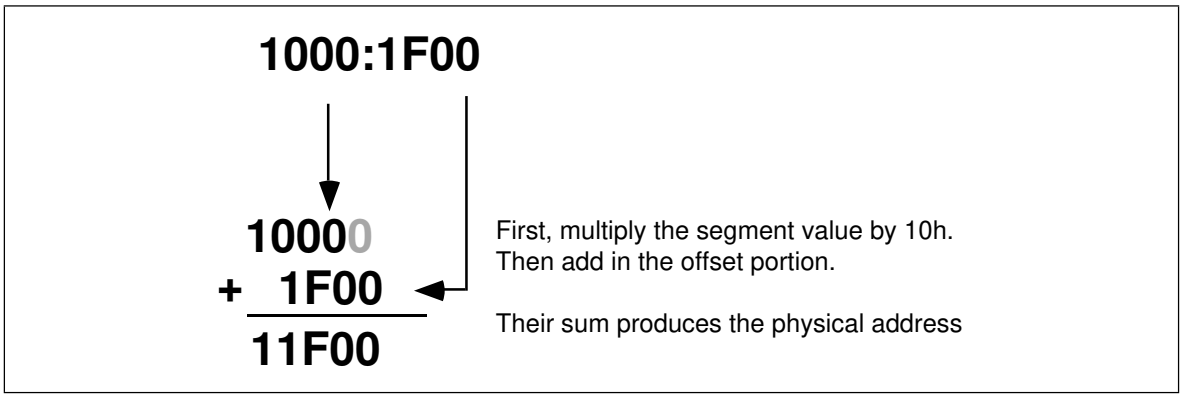

Figure 4.6 Converting a Logical Address to a Physical Address

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.

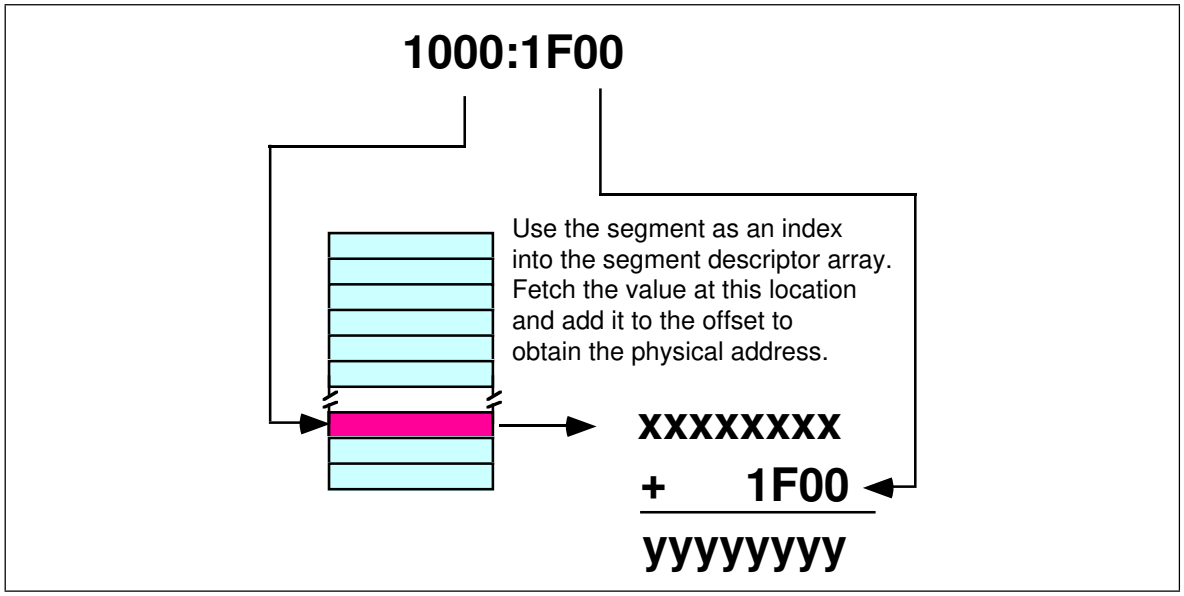

Figure 4.7 Converting a Logical Address to a Physical Address in Protected Mode

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.

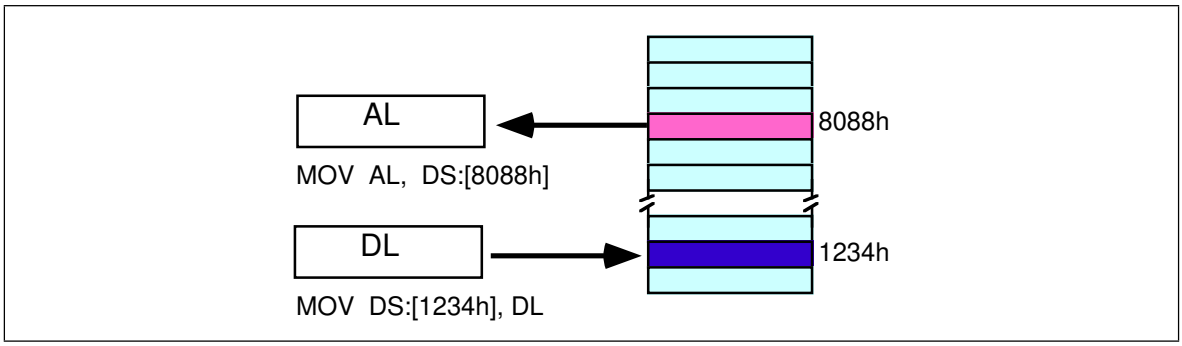

Figure 4.8 Displacement Only (Direct) Addressing Mode

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.

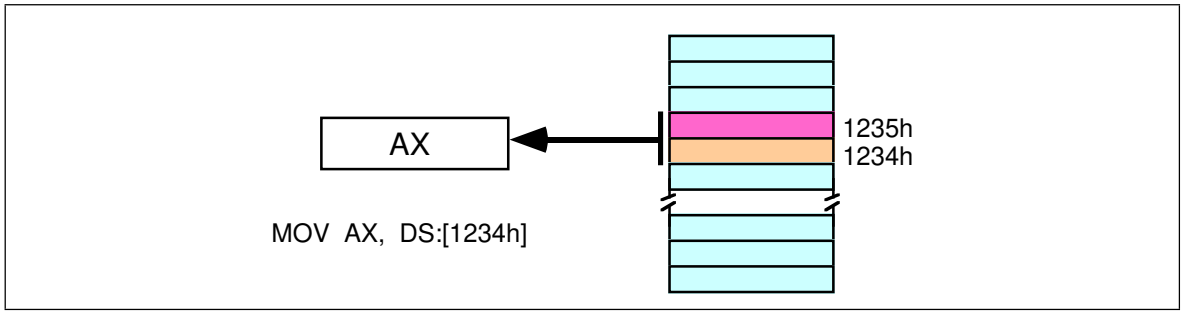

Figure 4.9 Accessing a Word

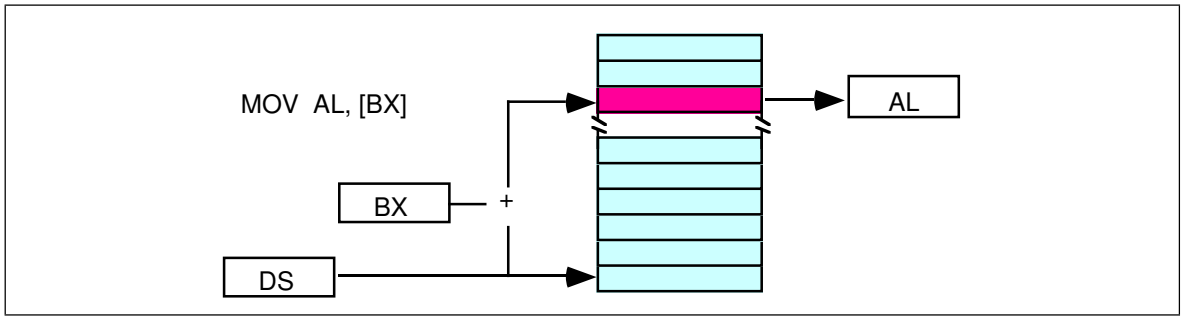

Figure 4.10 [BX] Addressing Mode

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.

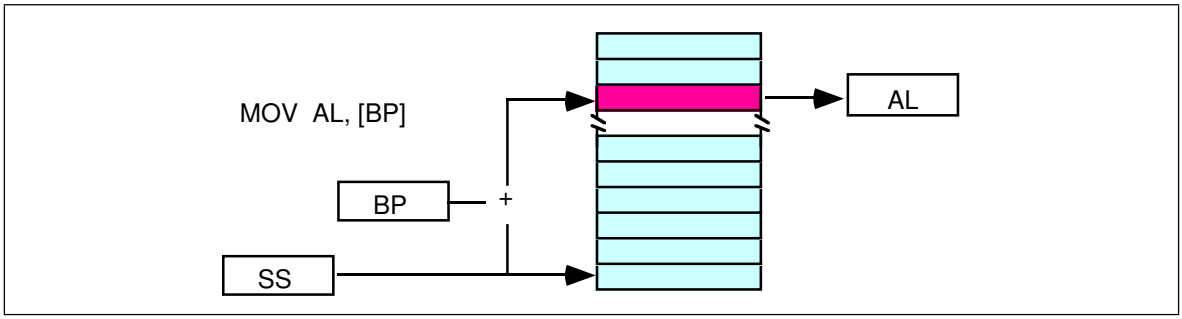

Figure 4.11 [BP] Addressing Mode

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.

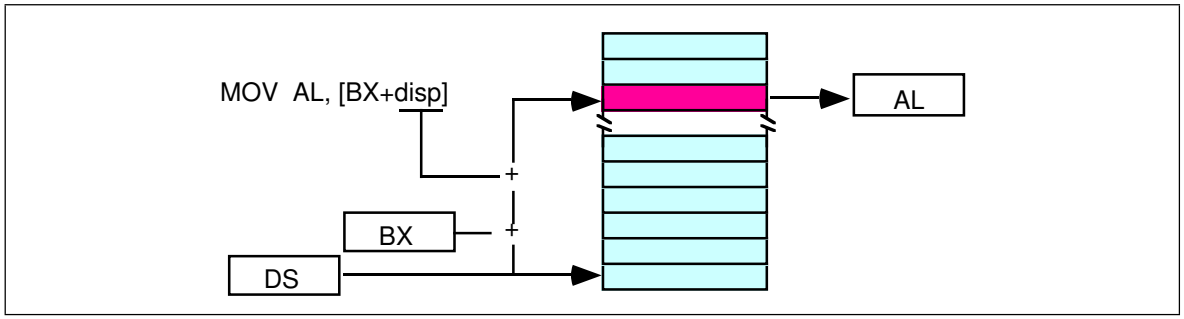

Figure 4.12 [BX+disp] Addressing Mode

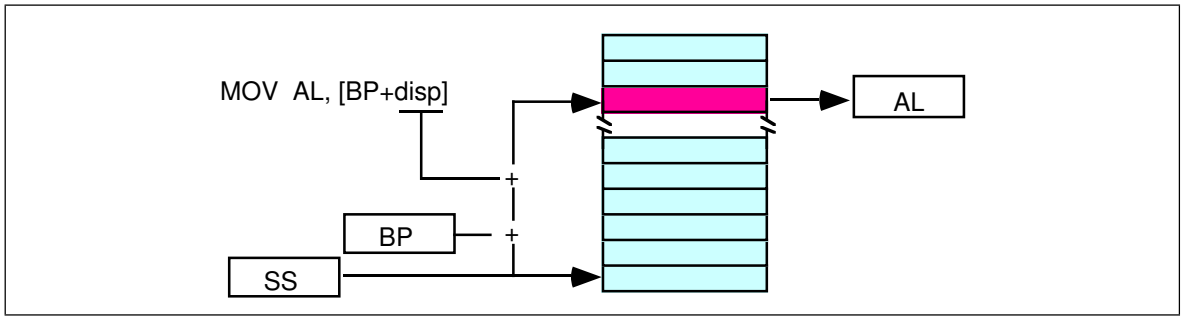

Figure 4.13 [BP+disp] Addressing Mode

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).

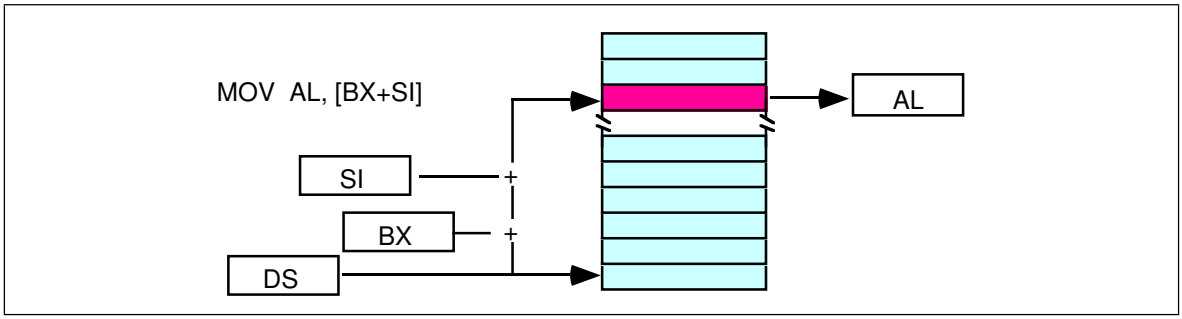

Figure 4.14 [BX+SI] Addressing Mode

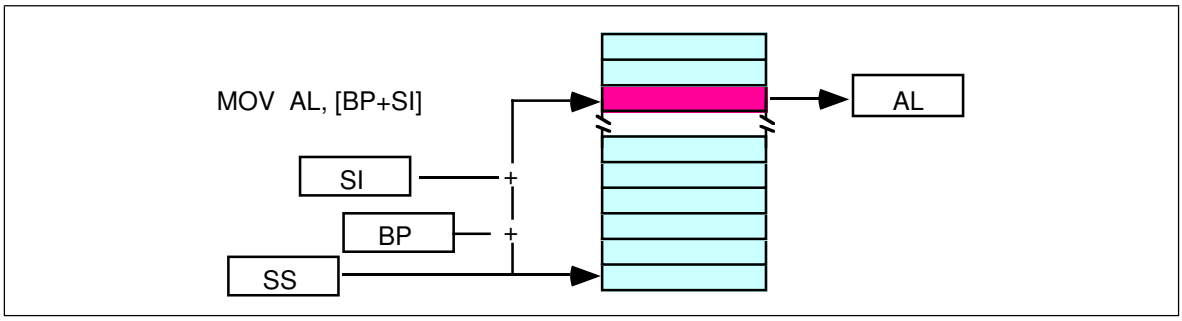

Figure 4.15 [BP+SI] Addressing Mode

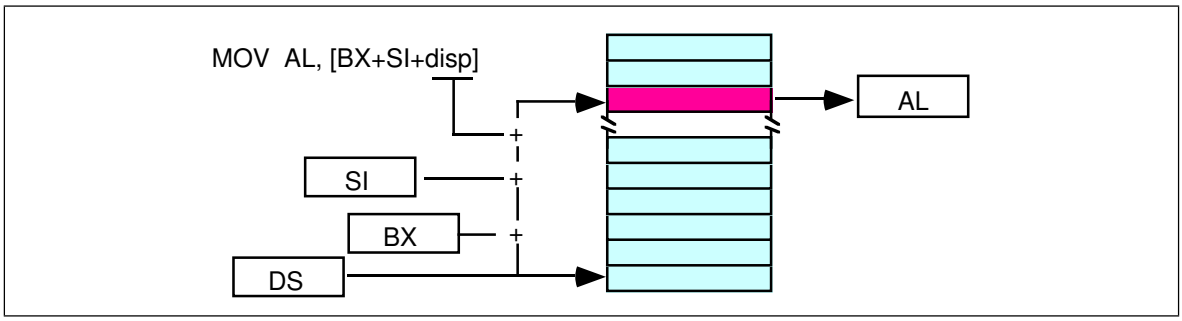

Figure 4.16 [BX + SI + disp] Addressing Mode

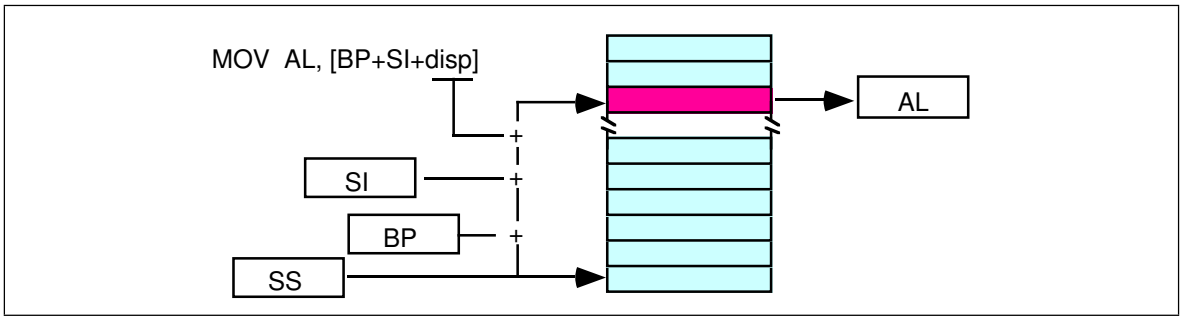

Figure 4.17 [BP + SI + disp] Addressing Mode

```
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.

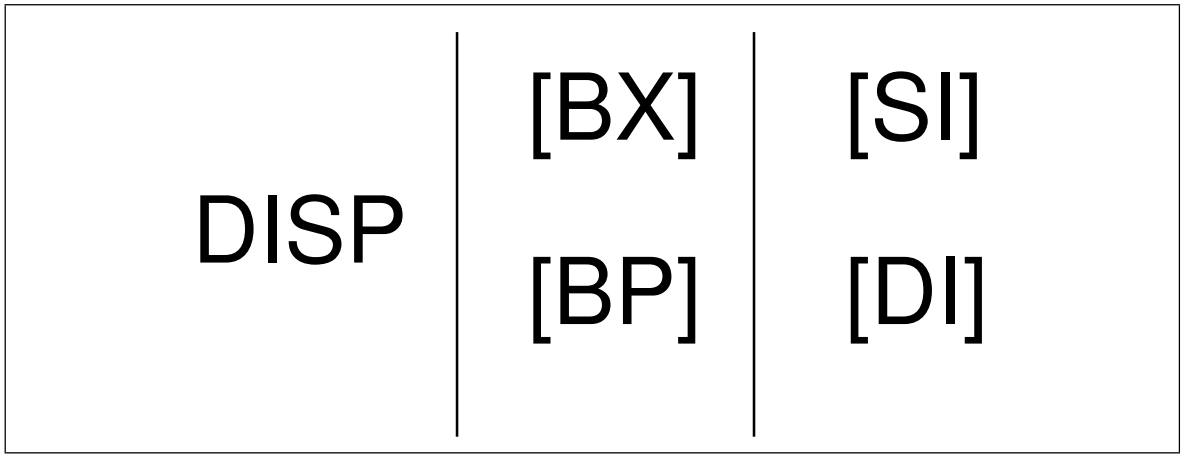

Figure 4.18 Table to Generate Valid 8086 Addressing Modes

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:

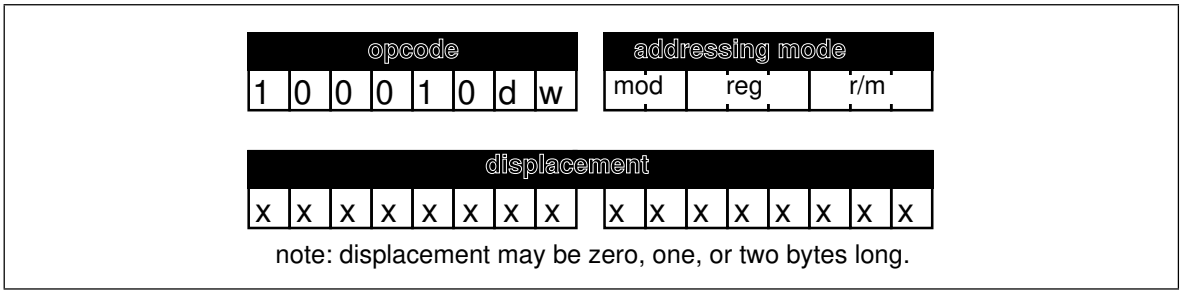

Figure 4.19 Generic MOV Instruction

```
[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.