**"AZƏRBAYCAN HAVA YOLLARI"**

**QAPALI SƏHMDAR CƏMİYYƏTİ**

**MİLLİ AVİASİYA AKADEMİYASI**

**About Of "Memory Handling İn Assembly Language" from**

**"OS-2" Lesson**

**COURSE WORK**

Department: Computer Systems and Programming

Faculty: Aerospace

Course: III

Specialty: Computer Engineering

Group: 1459i

Student: Babayev Mehrali

Teacher: Shahmaliyev Mamed

Head of Department: Melikov Agasi\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Bakı – 2022**

**CONTENTS**

1. **Introduction3**
2. The 80x86 CPUs:A Programmer’s View4
3. **8086 General Purpose Registers4**
4. 8086 Segment Registers 6
5. 8086 Special Purpose Registers 8
6. **80286 Registers** **8**
7. 80386/80486 Registers9
8. 80x86 Physical Memory Organization11

**9. The 80x86 Addressing Modes** **12**

**10. 8086 Register Addressing Modes 12**

**11. 8086 Memory Addressing Modes14**

**12. MASM Syntax for 8086 Memory Addressing Modes14**

**13. The Displacement Only Addressing Mode15**

**14. The Register Indirect Addressing Modes16**

**15. Indexed Addressing Modes17**

**16. Based Indexed Addressing Modes18**

**17. An Easy Way to Remember the 8086 Memory Addressing Modes19**

**18. Some Final Comments About 8086 Addressing Modes19**

**19. 80386 Register Addressing Modes20**

**20. 80386 Memory Addressing Modes21**

**21. Register Indirect Addressing Modes21**

**22. 80386 Indexed, Base/Indexed and Base/Indexed/Disp Addressing Mode..22**

**23. Some Final Notes About the 80386 Memory Addressing Modes24**

**24. Source Used .....................................................................................................25**

**Introduction**

This course work discusses memory addressing, memory organization, CPU addressing modes, and data representation in memory. From the assembly language programming point of view, this course work discusses the 80x86 register sets, the 80x86 memory addressing modes, and composite data types.

This course work 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.

80x86 memory addressing modes are, perhaps, the most important topic in this course work. This course work 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 course work 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.

**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.

**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.

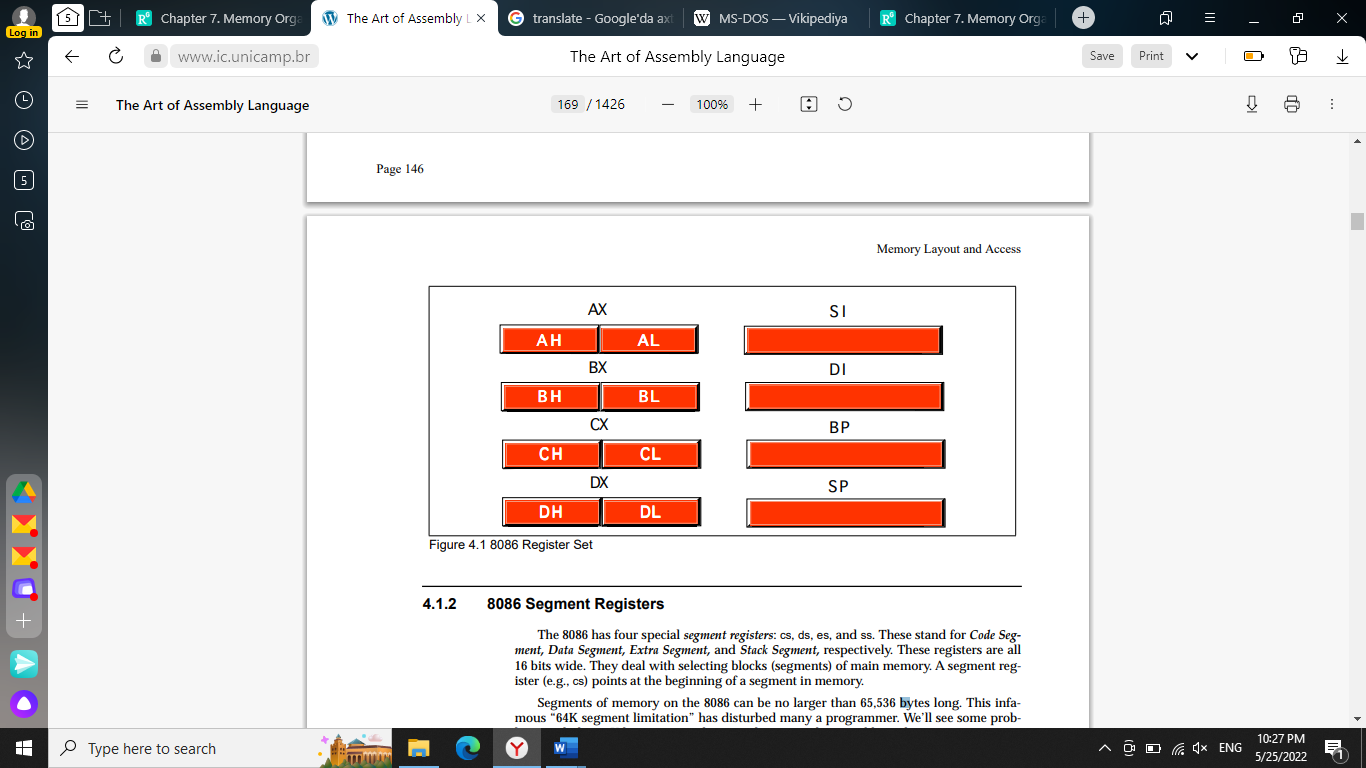
The bp register (Base Pointer) is similar to the bx register. You’ll generally use this register to access parameters and local variables in a procedure.

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.



**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.

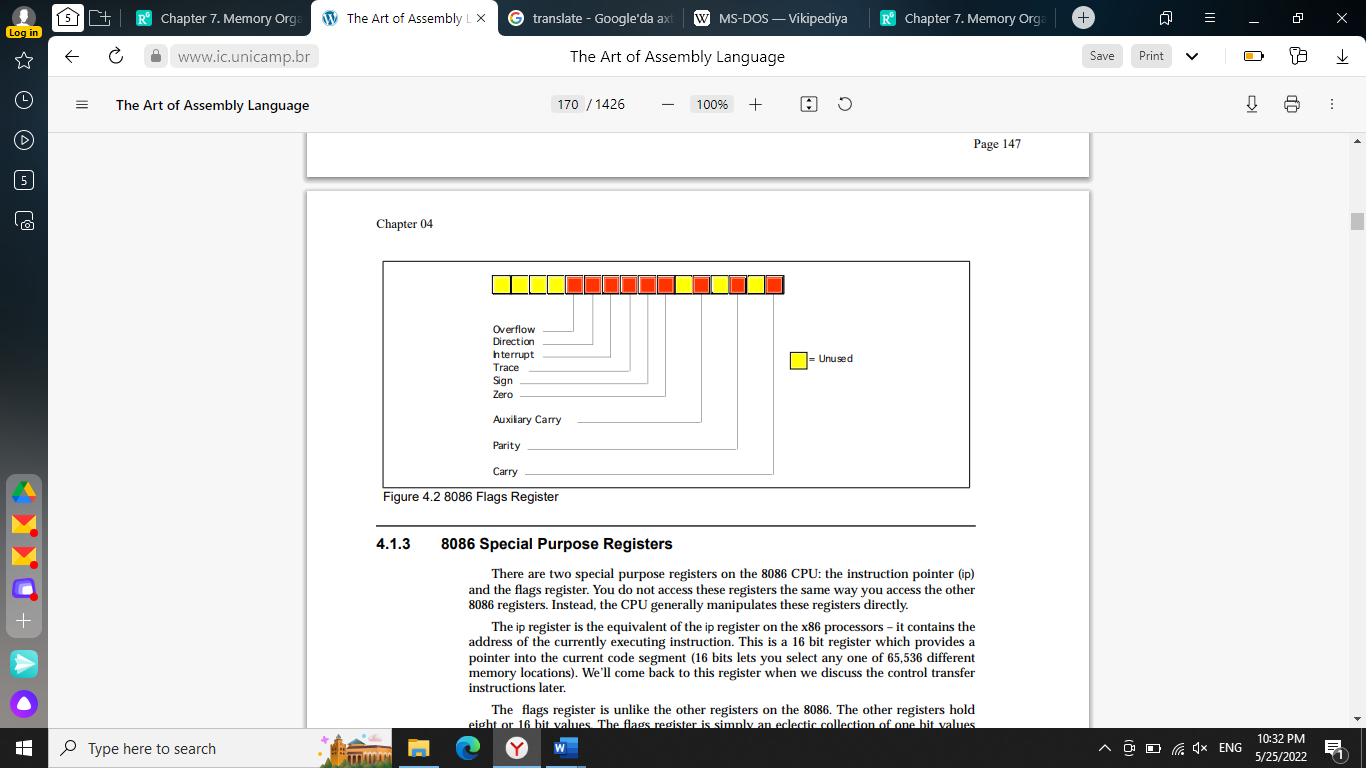
The cs register points at the segment containing the currently executing machine instructions. Note that, despite the 64K segment limitation, 8086 programs can be longer than 64K. You simply need multiple code segments in memory. Since you can change the value of the cs register, you can switch to a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



**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.

**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.

**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.

**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.

**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 machines5, it’s important to separate the discussion of these two sets of addressing modes to avoid confusing them.

**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 ; 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).

**8086 Memory Addressing Modes**

The 8086 provides 17 different ways to access memory. This may seem like quite a bit at first, 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!

**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]

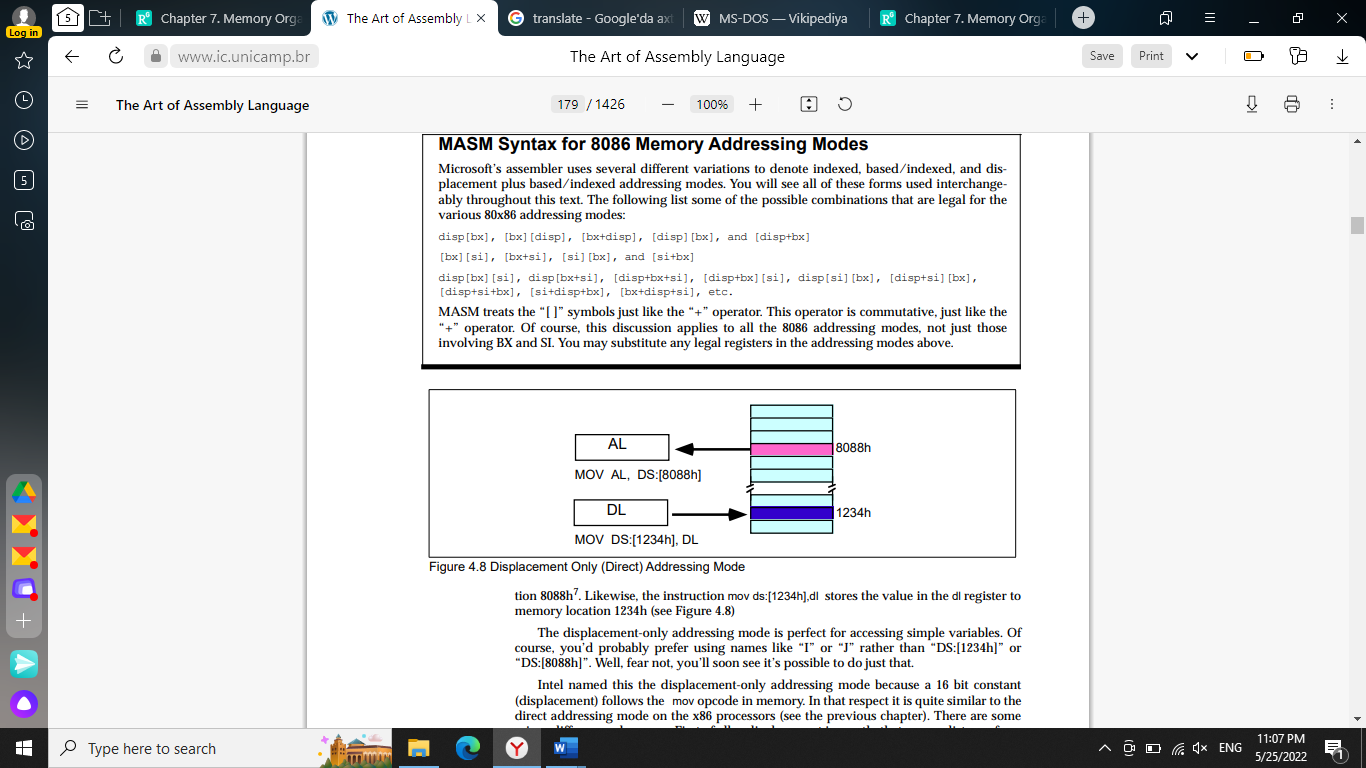
[bx][si], [bx+si], [si][bx], and [si+bx]

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.

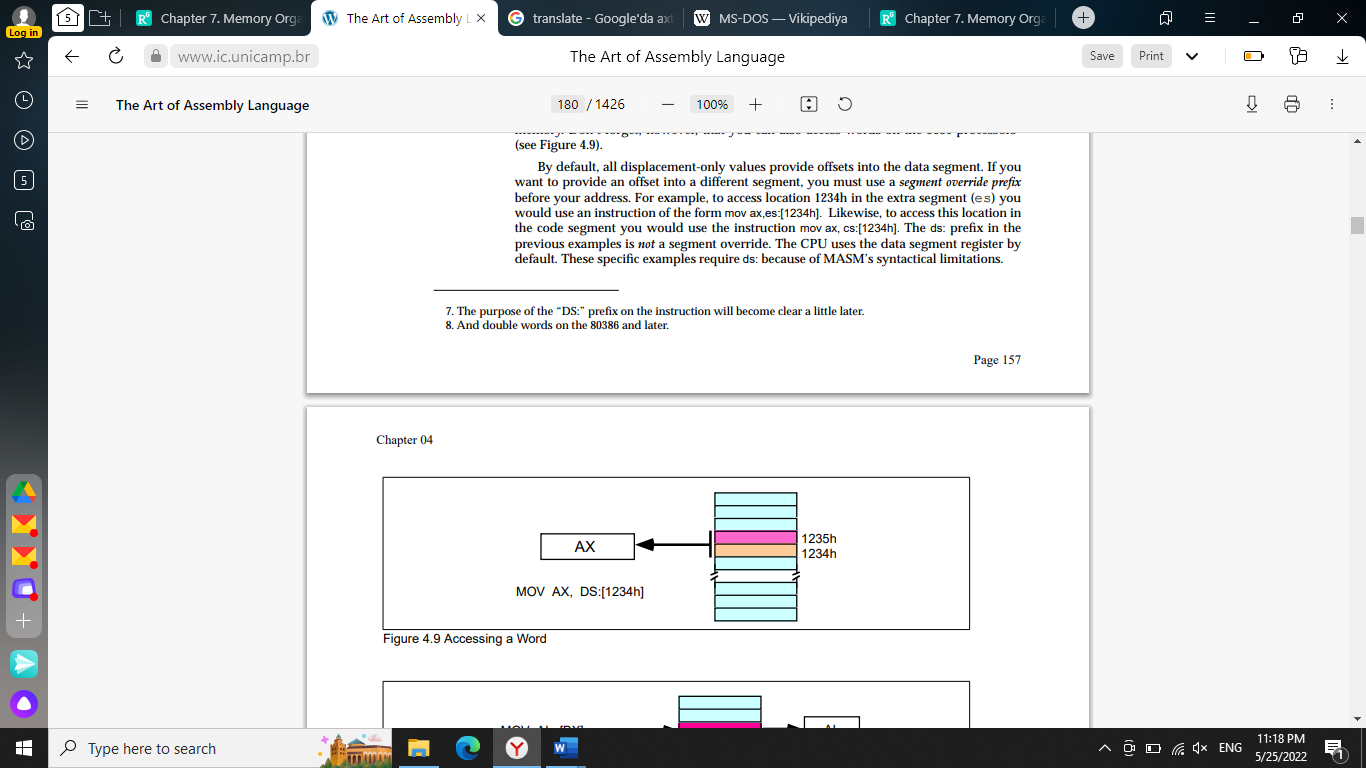
**The Displacement Only Addressing Mode**

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 location 8088h . 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 (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.



**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.

**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]*

**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]*

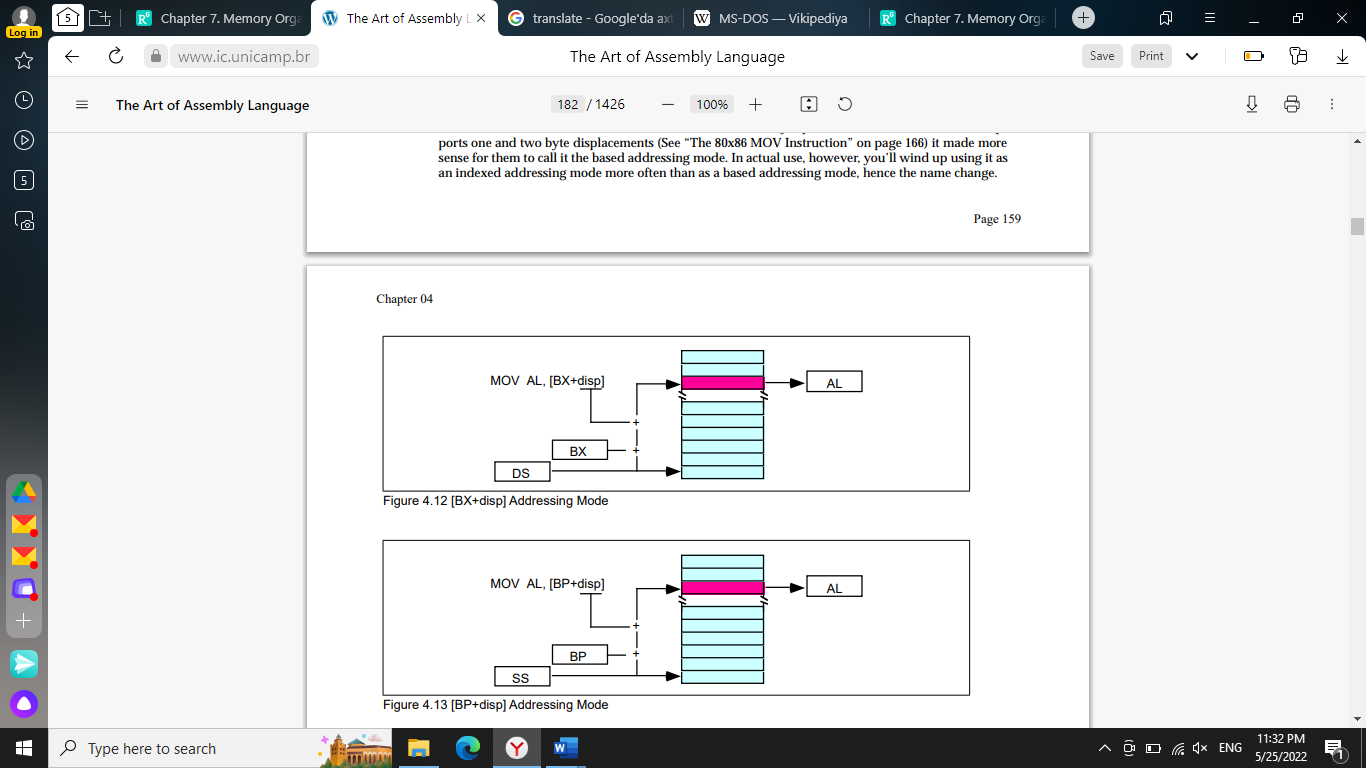
*mov al, [bp][si]*

*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.



**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]. 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.

**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 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.

**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.

**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.

**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. 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.*

**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.

**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:

*[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.*

**SOURCE USED**

1. <https://www.ic.unicamp.br/~pannain/mc404/aulas/pdfs/Art%20Of%20Intel%20x86%20Assembly.pdf>
2. <https://www.researchgate.net/publication/344938164_Chapter_7_Memory_Organization_and_Assembly_Language_Programming>
3. <http://www.ece.utep.edu/courses/web3376/Notes_files/ee3376-assembly.pdf>
4. https://www2.southeastern.edu/Academics/Faculty/kyang/2009/Fall/CMPS293&290/ClassNotes/CMPS293&290ClassNotesChap03.pdf