Memory Structure

Segment and Offset

Actual addresses on the IBM PC are given as a *pair* of 16-bit numbers, the *segment* and the *offset*, written in the form segment:offset. We have ignored the segment because it was a constant in one of the four *segment* registers, specifically, the DS register, which we loaded with the statements:

```
mov    ax,@data
mov    ds, ax
```

The rest of the time, only the offset was contained in individual instructions.

Things on the 80x86 are based on a 16-bit word. Therefore, addresses are based on a 16-bit word. As we know, 2^{16} is 65,536. It might seem that a segment of 2^{16} and an offset of 2^{16}, would give us 2^{32}, or approximately 4GB of memory. Before the 8086 was invented, microprocessors had a maximum of 2^{16} or 64KB of memory. The designers know that was insufficient, and the decision was made that 1 megabyte of memory would be more than would ever be needed. Part of the reason was to hold down cost; less address lines means lower system costs. 1MB is 2^{20} or four more bits. This did not fit into things easily. They decided to make the segment register assumed to be 20 bits and only the upper 16 bits would be specified. That means the four zeros must be added to the right side of the segment. What really happens looks like this if we want to convert the segment:offset of 13a5:3327 to an actual physical address that all computers need:

```
segment     1 3 a 5 0
offset    + 3 3 2 7
address     1 6 d 7 7
```

On 386 and later machines, segment had a size of 64 KB. Therefore, to get the actual physical address, we can calculate it in the following way:

Physical address = Segment address x 64 K + Offset address

Memory Model

We have always used SMALL memory model because we needed to force all memory references to near pointers and near calls/jumps. There are actually six models, TINY, SMALL, COMPACT, MEDIUM, LARGE, and HUGE. This directive simply sets the assembler to use the proper sized addresses. It is useful to be able to separately specify the data and code address as near or far. The TINY model puts everything into one segment: the code, the data, and the stack. The HUGE uses multiply code segments (FAR calls) and multiple data (FAR calls) and multiple data segments (FAR pointers), including segments that are larger than 64K, which requires special treatment. That leaves:

size	Code Segments	Data Segments
SMALL	one (near calls)	one (near pointers)
COMPACT	one (near calls)	multiple (far pointers)
MEDIUM	multiple (far calls)	one (near pointers)
LARGE	multiple (far calls)	multiple (far pointers)

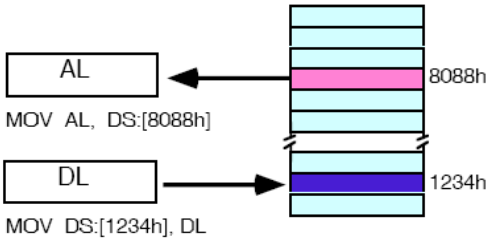
When there are multiple code segments, the PROCs are to return with far ret instruction, and should be referenced in other files as EXTRN *Label* : FAR.
As a general rule, if you are combining assembly programs with high-level languages, you should use the same model in assembly as the high-level language, where the model is much more important!
In order to allow a true small mode program, the assemblers automatically group .DATA and .STACK segments into a single segment. Since the stack data must come as the end of the program, the segments always occur in the order .CODE, .DATA and .STACK.

Memory Addressing Modes

The 80X86 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. 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-only, 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.

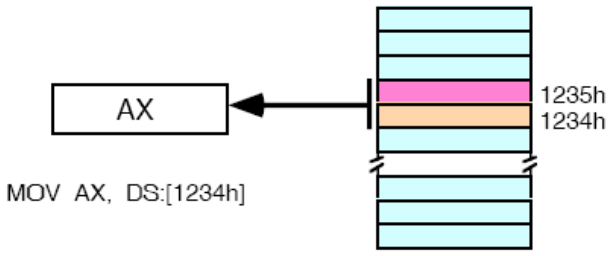
The Displacement Only Addressing Mode

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 below)



Displacement Only (Direct) Addressing Mode

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]".
On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). You can think of the displacement-only addressing mode as a direct addressing mode. The examples that follow will typically access bytes in memory. Don't forget; however, that you can also access words on the 8086 processors (see Figure below).



Accessing a word

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 Mode

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 shown by the following instructions:

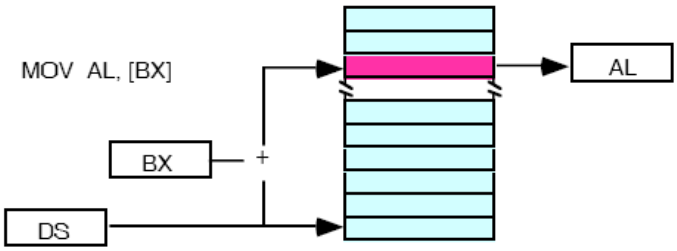
```
mov al, [bx]
mov al, [bp]
mov al, [si]
mov al, [di]
```

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:

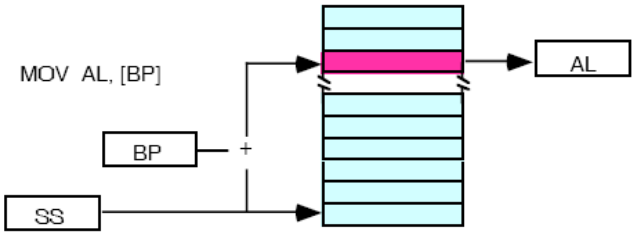
```
mov al, cs:[bx]
mov al, ds:[bp]
mov al, ss:[si]
mov al, es:[di]
```

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.

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



[bx] Addressing Mode



[bp] Addressing Mode

Indexed Addressing Mode

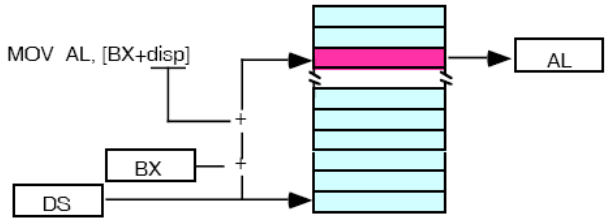
The indexed addressing modes use the following syntax:

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

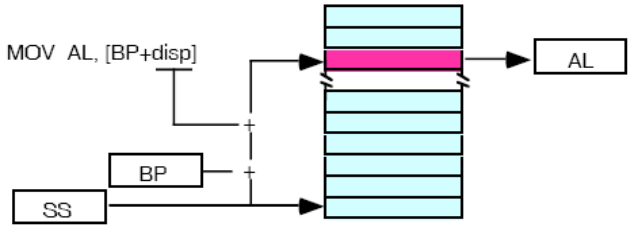
If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.
The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing 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 to obtain the [si+disp] and [di+disp] addressing modes.



[bx + disp] Addressing Mode



[bp + disp] Addressing Mode

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, 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.

Based Indexed Addressing Mode

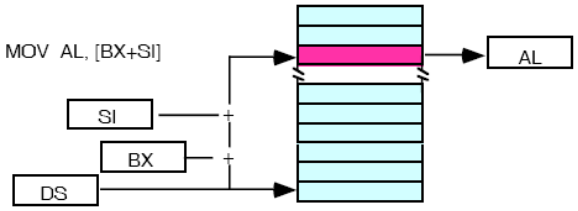
The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx/bp) and an index register (si/di). The allowable addressing modes are:

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

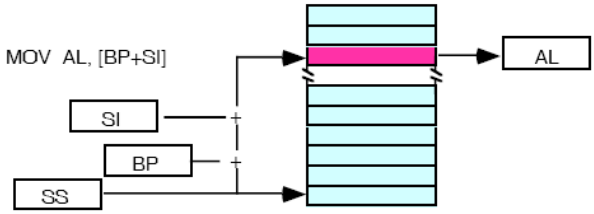
Suppose that bx contains 1000h and si contains 880h. Then the instruction mov al,[bx][si] would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D. The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default. You substitute di to obtain the [bx+di] addressing mode and di for the [bp+di] addressing mode.

Based Indexed plus Displacement Addressing Mode

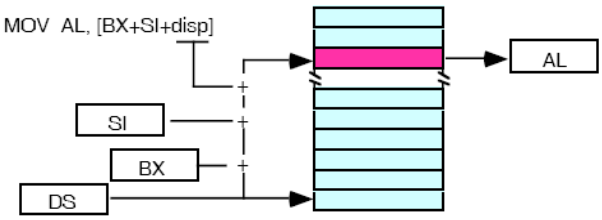
These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes:



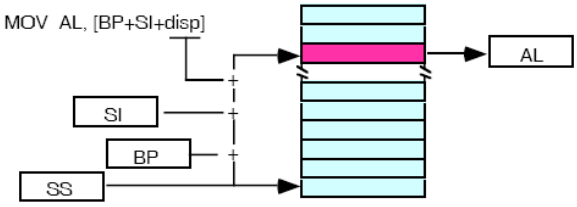
[BX + SI] Addressing Mode



[BP + SI] Addressing Mode



[BX + SI + disp] Addressing Mode



[BP + SI + disp] Addressing Mode

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

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

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.

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 below: 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.

DISP	[BX]	[SI]
	[BP]	[DI]

Table to generate valid 8086 Addressing Modes