Addressing

If you are familiar with C pointers and pointer arithmetic, you will notice many things here that look familiar. In fact, C/C++ pointers were designed to mimic assembly language in order to make them efficient to implement. In other words, if you don't really understand pointers in C, after you see how they really work, you will probably be able to understand them, finally.

Memory Address versus Memory Contents

In high-level languages, such as C, the identifier refers to the contents of a memory location. Thus, when you write:

```
A = 3;      /* assigns the value of three to the memory location you named A */
x = A;      /* the memory location x gets the value used at location A or three */
p = &A;     /* P now holds the address of the memory location you named A */
```

Unless you use & symbol, you are referring to the contents. The contents of that location change during the execution of your program, but the address will be the same throughout.

As you remember, we can write something like this:

```
A   DB   24
B   DW   1234
```

Now when we want to get the contents and address we do:

```
mov  bx, B           ; moves the contents of B to bx
mov  bx, OFFSET B    ; moves the address of B to bx
```

or the keyword **OFFSET** is the equivalent of C's **&**

Address Arithmetic

In C, we have something like:

char name[10];

This reserves ten bytes of memory and the first one is called name[0]. In assembly language we can do it a number of ways:

```
A DB 0Ah, 1Ah, 2Ah, 3Ah, 6 dup (?)
```

Or

```
A DB 0Ah
  DB 1Ah
  DB 2Ah
  DB 3Ah
  DB 6 dup (?)
```

Now the location A holds 0Ah, etc. Notice that the other locations do not have a name (they are *anonymous*), but we can get to them with **address arithmetic** A + 3 refers to the byte containing 3Ah. This means that A + 3 points to the byte **[A + 3]**. This gives us two forms:

```
mov   al, A          ; moves the contents of A to al
mov   al, [A + 3]    ; moves the address of A plus 3 to al (which holds 3Ah!)
```

Additionally, if p is a pointer variable in C, we can say that [p] is the equivalent of *p.

Notice that the following are not the same thing!!!!!

```
mov   al, A + 3      ; moves 3Ah into al
mov   al, [A] + 3    ; moves 0Ah plus 3 (0Dh) into al
```

A + 3 in assembly language is the exact opposite of what it is in C! I recommend that you used the notation [a + 3].

If you to get the address of pointed to by [a + 3], you would use:

```
mov   ax, OFFSET [A + 3]   ; moves address of where the byte 3Ah is into ax
```

Suppose we have the following definitions of arrays:

```
A   DB   0Ah, 1Ah, 2Ah, 3Ah, 4Ah, 5Ah, 6Ah, 7Ah
B   DB   0Bh, 1Bh, 2Bh, 3Bh
C   DB   0Ch, 1Ch, 2Ch, 3Ch, 4Ch, 5Ch
```

In memory we would have:

Label	A:								B:				C:					
Contents	0A	1A	2A	3A	4A	5A	6A	7A	0B	1B	2B	3B	0C	1C	2C	3C	4C	5C
Offset	A B-8 C-12	A+1 B-7 C-11	A+2 B-6 C-10	A+3 B-5 C-9	A+4 B-4 C-8	A+5 B-3 C-7	A+6 B-2 C-6	A+7 B-1 C-5	A+8 B C-4	A+9 B+1 C-3	A+10 B+2 C-2	A+11 B+3 C-1	A+12 B+4 C	A+13 B+5 C+1	A+14 B+6 C+2	A+15 B+7 C+3	A+16 B+8 C+4	A+17 B+9 C+5

It is important to notice that the offset can be a positive or negative number and that there is nothing preventing this. Array-bound checking is only accomplished in high-level languages with the addition of additional code that you normally don't see! This is why in C, if you exceed the boundary of an array, you don't get an error message unless you attempt to use memory that is not available.

Whenever you use OFFSET, you are referring to a word! For mov instructions, you must have a sixteen bit register as the destination!

Always remember that when addressing an array element, what you are counting is the number of bytes from the beginning of the array.

Rules for Address Expressions

An address and a number are two different types of objects. An address represents a physical location in member. A number is simply a bit pattern that has no inherent data type! We don't know if it represents years, days, minutes, seconds, oranges, airplanes, characters or whatever.

The most important difference is when a program is loaded into a different location in memory, the addresses change but the numbers do not!

Legal Address Arithmetic

Symbols (identifiers) are addresses if the label memory locations. Normally, that is when they are in front of something like DB or DW in the data segment or when followed by a colon in the code segment. Additionally, symbols can be EQUated to addresses are addresses. (However, symbols EQUated to constant numbers are just simply ordinary numbers.)

If A and B are addresses and *n* a ordinary number, then we can legally do:

- A + *n* yields an **address**
- A - *n* yields an **address**
- A - B yields an **ordinary number**
- (A) yields an **address**
- any expression involving only ordinary numbers yields an ordinary number.

Every address has a particular data type (word or byte) and every address express retains the data type of the base address.

Some examples are:

A + 14	address
B - A	number
CW - AW	number
AW + (B - A)	address Remember B - A is a number that puts this into the form of A + *n*

There is also a special assembler symbol, the dollar sign. **NOTE:** This is not '$', the quotes make it a character. The assembler symbol represents the next location that code will be assembled into.

Here is an example:

```
INCLUDE pcmac.inc
        .data
msg     DB      'This is a test', 10, 13, '$'
msglen  DW      $ - msg
msg1    DB      'Its length is $'
        .code
        EXTRN   PutDec : NEAR
main    PROC
        _Begin
        _PutStr msg
        _putStr msg1
        mov     ax, msglen
        call    PutDec
        _PutCh  10, 13
        _Exit   0
main    ENDP
        END     main
```

This says that the length is 17, which is longer than the string that appears on the screen. This is because in memory, the 10, 13, and '$' each occupy one byte and is included in the length.

Byte Swapping

When we write code that will store a word, such as:

```
AWord    DW    1234h
```

produces memory that looks like this: | 34h | 12h |

When moving data from memory (or the opposite direction), the data is put into the correct format.

Words in registers have bytes in the normal order **Words in memory have their bytes swapped** **Moving to or from memory swaps bytes**

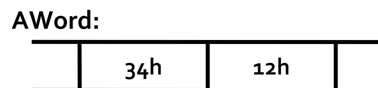
Once a label for a variable is defined with a size, it keeps that size of word or byte that it was give, unless you **cast** or **coerce** or **override** it:

```
Aword        DW        1234h
    ...
mov          al, BYTE PTR AWord        ; al now has 34h
```

If we declare

```
AWord            DW      1234h
```

The 80X86 CPU is said to be byte swapped because it stores the low-order byte first (i.e. in the lower address. In the above example, 34h goes into the first byte labelled AWord and 12h goes into the second.

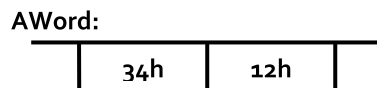


It is almost we had written:

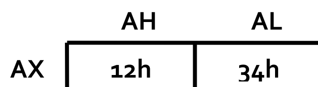
```
AWord            DB      34h, 12h
```

Therefore   AWord DW   1234h   and   mov   AWord, 1234h

Both produces



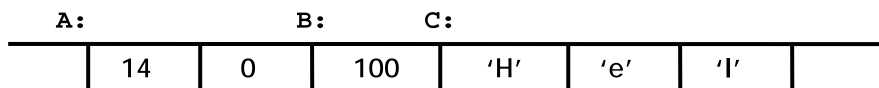
which is swapped back by mov AX, AWord



Variable size can be changed by prefixing a variable with DWORD PTR, WORD PTR, or BYTE PTR which forces it to be interpreted as a double word, word, or byte, respectively. Changing types in this way is called a type override or cast or coercion. The reason for the keyword PTR is that the address in the instruction, a pointer to the data, is what is being coerced.

Thus, if

```
A       DW      14
B       DB      100
C       DB      'Hello'
        ...
        mov     ax, WORD PTR B          ;sets al to 100, ah to 'H' = 72
        mov     al, BYTE PTR A          ;sets al = 14
```



Another example:

```
D       DB      -7, 14, 22
E       DW      434, -18, 26
        ...
        mov     al, D+4                 ;sets al = 01h
        mov     ax, E+3                 ;sets ax = 1AFFh
        mov     ax, E -2                ;sets ax = 160Eh
        mov     al, BYTE PTR E+3        ;sets al = 0FFh = -1
        mov     ax, WORD PTR D+2        ;sets ax = 0B216h
```