3

MEMORY ACCESS AND ORGANIZATION

3.1 Chapter Overview

Earlier chapters in this text show you how to declare and access simple variables in an assembly language program. In this chapter you get the full picture on 80x86 memory access. You also learn how to efficiently organize your variable declarations to speed up access to their data. This chapter will also teach you about the 80x86 stack and how to manipulate data on the stack. Finally, this chapter will teach you about dynamic memory allocation and the *heap*.

3.2 The 80x86 Addressing Modes

The 80x86 processors let you access memory in many different ways. Until now, you've only seen a single way to access a variable, the so-called *displacement-only* addressing mode. In this section you'll see some additional ways your programs can access memory using 80x86 *memory addressing modes*. 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 toward mastering 80x86 assembly language.

When Intel designed the original 8086 processor, it provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor while retaining all the modes of the previous processors. However, in 32-bit environments like Windows, BeOS, and Linux, these earlier addressing modes are not very useful; indeed, HLA doesn't even support the use of these older, 16-bit-only addressing modes. Fortunately, anything you can do with the older addressing modes can be done with the new addressing modes as well (even better, as a matter of fact). Therefore, you won't need to bother learning the old 16-bit addressing modes when writing code for today's high-performance operating systems. Do keep in mind, however, that if you intend to work under MS-DOS or some other 16-bit operating system, you will need to study up on those old addressing modes (see the 16-bit edition of this book on the accompanying CD-ROM for details).

3.2.1 80x86 Register Addressing Modes

Most 80x86 instructions can operate on the 80x86's general purpose register set. By specifying the name of the register as an operand to the instruction, you can access the contents of that register. Consider the 80x86 `mov` (move) instruction:

```
mov( source, destination );
```

This instruction copies the data from the *source* operand to the *destination* operand. The 8-bit, 16-bit, and 32-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 80x86 `mov` instructions:

```
     mov( bx, ax );          // Copies the value from BX into AX
     mov( al, dl );          // Copies the value from AL into DL
     mov( edx, esi );        // Copies the value from EDX into ESI
     mov( bp, sp );          // Copies the value from BP into SP
     mov( cl, dh );          // Copies the value from CL into DH
     mov( ax, ax );          // Yes, this is legal!
```

The registers are the best place to keep variables. Instructions using the registers are shorter and faster than those that access memory. Of course, most computations require at least one register operand, so the register addressing mode is very popular in 80x86 assembly code. Throughout this chapter you'll see the abbreviated operands *reg* and *r/m* (register / memory) used wherever you may use one of the 80x86's general purpose registers.

3.2.2 80x86 32-Bit Memory Addressing Modes

The 80x86 provides hundreds of different ways to access memory. This may seem like quite a bit at first, but fortunately most of the addressing 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 80x86 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed. Variations on these five forms provide all the different addressing modes on the 80x86. See, from hundreds down to five. It's not so bad after all!

3.2.2.1 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 32-bit constant that specifies the address of the target location. Assuming that variable `J` is an `int8` variable appearing at address $8088, the instruction "mov(J, al);" loads the AL register with a copy of the byte at memory location $8088. Likewise, if `int8` variable `K` is at address $1234 in memory, then the instruction "mov(dl, K);" stores the value in the DL register to memory location $1234 (see Figure 3-1).

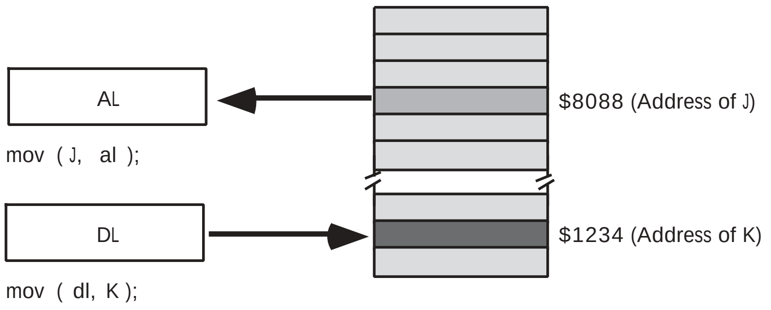


Figure 3-1: Displacement-Only (Direct) Addressing Mode.

The displacement-only addressing mode is perfect for accessing simple scalar variables.

Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the `mov` opcode in memory. On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address zero). The examples in this chapter will often access bytes in memory. Don't forget, however, that you can also access words and double words on the 80x86 processors by specifying the address of their first byte (see Figure 3-2).

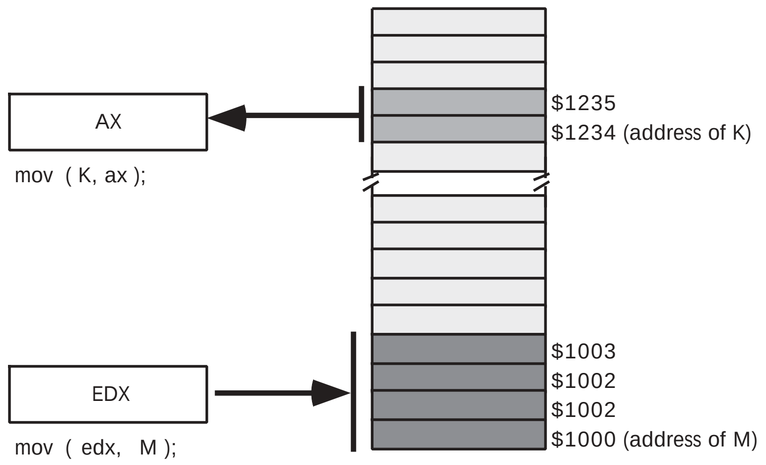


Figure 3-2: Accessing a Word or DWord Using the Displacement Only Addressing Mode.

3.2.2.2 The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. The term “indirect” means that the operand is not the actual address, but rather, the operand’s value specifies the memory address to use. In the case of the register indirect addressing modes, the value the register holds is the address of the memory location to access. For example, the instruction “mov(eax, [ebx]);” tells the CPU to store EAX’s value at the location whose address is in EBX (the square brackets around EBX tell HLA to use the register indirect addressing mode).

There are eight forms of this addressing mode on the 80x86; the following instructions are examples of these eight forms:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

These eight addressing modes reference the memory location at the offset found in the register enclosed by brackets (EAX, EBX, ECX, EDX, EDI, ESI, EBP, or ESP, respectively).

Note that the register indirect addressing modes require a 32-bit register. You cannot specify a 16-bit or 8-bit register when using an indirect addressing mode.[1] Technically, you could load a 32-bit register with an arbitrary numeric value and access that location indirectly using the register indirect addressing mode:

```
        mov( $1234_5678, ebx );
        mov( [ebx], al );     // Attempts to access location $1234_5678.
```

Unfortunately (or fortunately, depending on how you look at it), this will probably cause the operating system to generate a protection fault because it's not always legal to access arbitrary memory locations. As it turns out, there are better ways to load the address of some object into a register; you'll see how to do this shortly.

The register indirect addressing mode has many uses. You can use it to access data referenced by a pointer, you can use it to step through array data, and, in general, you can use it whenever you need to modify the address of a variable while your program is running.

The register indirect addressing mode provides an example of an *anonymous* variable. When using the register indirect addressing mode you refer to the value of a variable by its numeric memory address (e.g., the value you load into a register) rather than by the name of the variable. Hence the phrase "anonymous variable."

HLA provides a simple operator that you can use to take the address of a `static` variable and put this address into a 32-bit register. This is the "&" (address of) operator (note that this is the same symbol that C/C++ uses for the address-of operator). The following example loads the address of variable `J` into EBX and then stores EAX's current value into `J` using the register indirect addressing mode:

```
    mov( &J, ebx );             // Load address of J into EBX.
    mov( eax, [ebx] );          // Store EAX into J.
```

Of course, it would have been easier to store EAX's value directly into `J` rather than using two instructions to do this indirectly. However, you can easily imagine a code sequence where the program loads one of several different addresses into EBX prior to the execution of the "mov(eax, [ebx]);" statement, thus storing EAX into one of several different locations depending on the execution path of the program.

CAUTION *The "&" (address-of) operator is not a general address-of operator like the "&" operator in C/C++. You may only apply this operator to static variables.*[2] *You cannot apply it to generic address expressions or other types of variables. Later, you will learn about the "load effective address" instruction that provides a general solution for obtaining the address of some variable in memory.*

[1] Actually, the 80x86 does support addressing modes involving certain 16-bit registers, as mentioned earlier. However, HLA does not support these modes and they are not useful under 32-bit operating systems.

[2] Note: The term "static" here indicates a `static`, `read only`, or `storage` object.

3.2.2.3 Indexed Addressing Modes

The indexed addressing modes use the following syntax:

```
    mov( VarName[ eax ], al );
    mov( VarName[ ebx ], al );
    mov( VarName[ ecx ], al );
    mov( VarName[ edx ], al );
    mov( VarName[ edi ], al );
    mov( VarName[ esi ], al );
    mov( VarName[ ebp ], al );
    mov( VarName[ esp ], al );
```

`VarName` is the name of some variable in your program.

The indexed addressing mode computes an effective address[3] by adding the address of the variable to the value of the 32-bit register appearing inside the square brackets. Their sum is the actual address in memory the instruction accesses. So if `VarName` is at address $1100 in memory and EBX contains an eight, then “mov(VarName[ebx], al);” loads the byte at address $1108 into the AL register (see Figure 3-3).

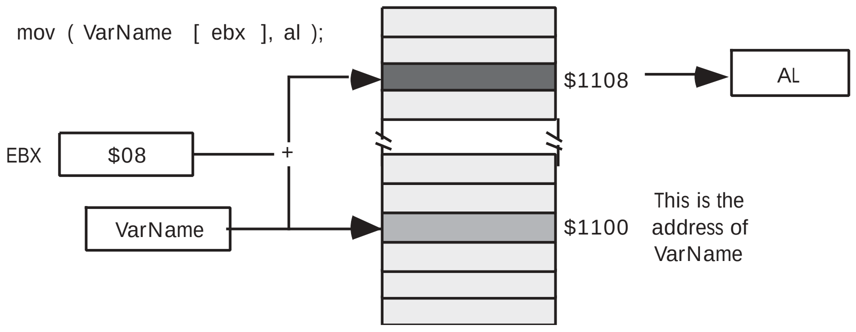


Figure 3-3: Indexed Addressing Mode.

The indexed addressing mode is really handy for accessing elements of arrays. You will see how to use this addressing mode for that purpose a little later in this book.

3.2.2.4 Variations on the Indexed Addressing Mode

There are two important syntactical variations of the indexed addressing mode. Both forms generate the same basic machine instructions, but their syntax suggests other uses for these variants.

The first variant uses the following syntax:

```
    mov( [ ebx + constant ], al );
    mov( [ ebx - constant ], al );
```

[3] The effective address is the ultimate address in memory that an instruction will access, once all the address calculations are complete.

These examples use only the EBX register. However, you can use any of the other 32-bit general purpose registers in place of EBX. This addressing mode computes its effective address by adding the value in EBX to the specified constant, or subtracting the specified constant from EBX (see Figures 3-4 and 3-5).

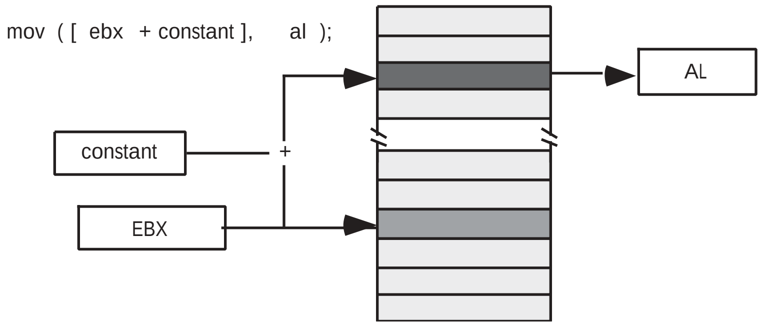


Figure 3-4: Indexed Addressing Mode Using a Register Plus a Constant.

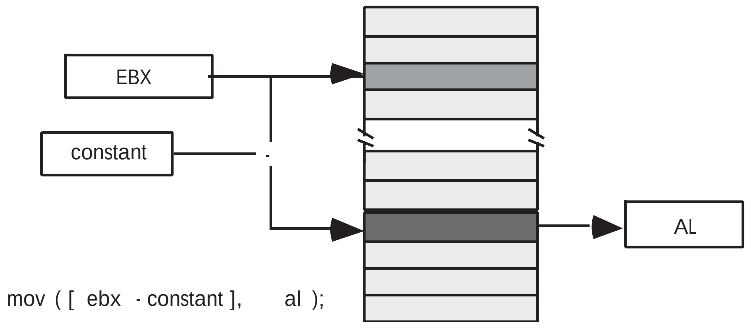


Figure 3-5: Indexed Addressing Mode Using a Register Minus a Constant.

This particular variant of the addressing mode is useful if a 32-bit register contains the base address of a multibyte object and you wish to access a memory location some number of bytes before or after that location. One important use of this addressing mode is accessing fields of a record (or structure) when you have a pointer to the record data. This addressing mode is also invaluable for accessing automatic (local) variables in procedures (see the chapter on procedures for more details).

The second variant of the indexed addressing mode is actually a combination of the previous two forms. The syntax for this version is the following:

```
    mov( VarName[ ebx + constant ], al );
    mov( VarName[ ebx - constant ], al );
```

Once again, this example uses only the EBX register. You may, however, substitute any of the 32-bit general purpose registers in lieu of EBX in these two examples. This particular form is quite useful when accessing elements of an array of records (structures) in an assembly language program (more on that in the next chapter).

These instructions compute their effective address by adding or subtracting the `constant` value from `VarName`'s address and then adding the value in EBX to this result. Note that HLA, not the CPU, computes the sum or difference of `VarName`'s address and `constant`. The actual machine instructions above contain a single constant value that the instructions add to the value in EBX at runtime. Because HLA substitutes a constant for `VarName`, it can reduce an instruction of the form

```
mov( VarName[ ebx + constant], al );
```

to an instruction of the form

```
mov( constant1[ ebx + constant2], al );
```

Because of the way these addressing modes work, this is semantically equivalent to

```
mov( [ebx + (constant1 + constant2)], al );
```

HLA will add the two constants together at compile time, effectively producing the following instruction:

```
mov( [ebx + constant_sum], al );
```

Of course, there is nothing special about subtraction. You can easily convert the addressing mode involving subtraction to addition by simply taking the two's complement of the 32-bit constant and then adding this complemented value (rather than subtracting the uncomplemented value).

3.2.2.5 Scaled Indexed Addressing Modes

The scaled indexed addressing modes are similar to the indexed addressing modes with two differences: (1) the scaled indexed addressing modes allow you to combine two registers plus a displacement, and (2) the scaled indexed addressing modes let you multiply the index register by a (scaling) factor of 1, 2, 4, or 8. The syntax for these addressing modes is

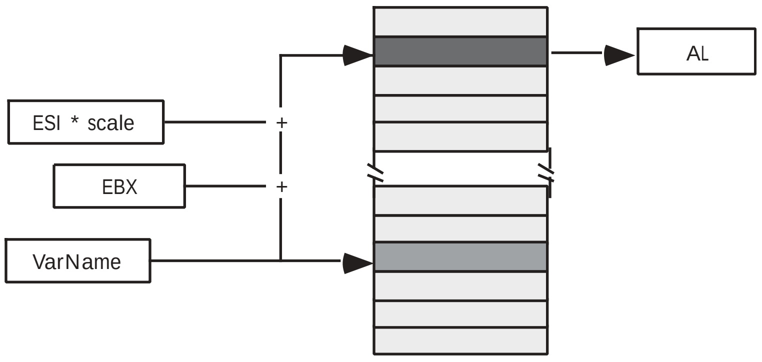
```
    VarName[ IndexReg32*scale ]
    VarName[ IndexReg32*scale + displacement ]
    VarName[ IndexReg32*scale - displacement ]


    [ BaseReg32 + IndexReg32*scale ]
    [ BaseReg32 + IndexReg32*scale + displacement ]
    [ BaseReg32 + IndexReg32*scale - displacement ]
```

```
VarName[ BaseReg32 + IndexReg32*scale ]
VarName[ BaseReg32 + IndexReg32*scale + displacement ]
VarName[ BaseReg32 + IndexReg32*scale - displacement ]
```

In these examples, `BaseReg32` represents any general purpose 32-bit register; `IndexReg32` represents any general purpose 32-bit register except ESP, and `scale` must be one of the constants: 1, 2, 4, or 8.

The primary difference between the scaled indexed addressing mode and the indexed addressing mode is the inclusion of the `IndexReg32*scale` component. These modes compute the effective address by adding in the value of this new register multiplied by the specified scaling factor (see Figure 3-6 for an example involving EBX as the base register and ESI as the index register).



mov (VarName [ebx + esi *scale], al);

Figure 3-6: The Scaled Indexed Addressing Mode.

In Figure 3-6, suppose that EBX contains $100, ESI contains $20, and `VarName` is at base address $2000 in memory. Then the following instruction:

```
mov( VarName[ ebx + esi*4 + 4 ], al );
```

will move the byte at address $2184 ($2000 + $100 + $20*4 + 4) into the AL register.

The scaled indexed addressing mode is useful for accessing elements of arrays whose elements are 2, 4, or 8 bytes each. This addressing mode is also useful for access elements of an array when you have a pointer to the beginning of the array.

CAUTION *Although this addressing mode contains two variable components (the base and index registers), don't get the impression that you use this addressing mode to access elements of a two-dimensional array by loading the two array indices into the two registers. Two-dimensional array access is quite a bit more complicated than this. The next chapter will consider multi-dimensional array access and discuss how to do this.*

3.2.2.6 Addressing Mode Wrap-Up

Well, believe it or not, you've just learned several hundred addressing modes! That wasn't hard now, was it? If you're wondering where all these modes came from, just consider the fact that the register indirect addressing mode isn't a single addressing mode, but eight different addressing modes (involving the eight different registers). Combinations of registers, constant sizes, and other factors multiply the number of possible addressing modes on the system. In fact, you need only memorize about two dozen forms and you've got it made. In practice, you'll use less than half the available addressing modes in any given program (and many addressing modes you may never use at all). So learning all these addressing modes is actually much easier than it sounds.

3.3 Run-Time Memory Organization

An operating system like Linux or Windows tends to put different types of data into different sections (or segments) of memory. Although it is possible to reconfigure memory to your choice by running the linker and specify various parameters, by default Windows loads an HLA program into memory using the organization appearing in Figure 3-7 (Linux is similar, though it rearranges some of the sections).

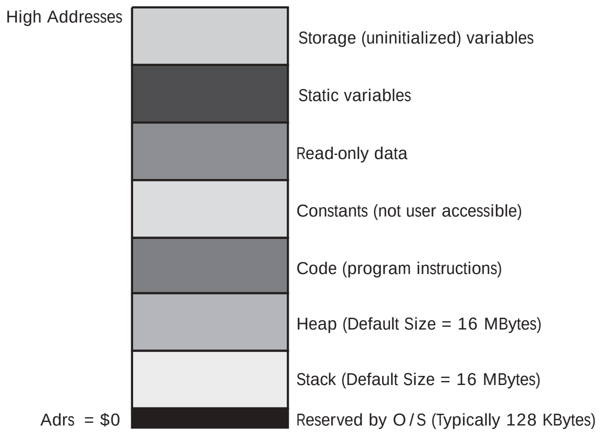

High Addresses
Storage (uninitialized) variables
Static variables
Read-only data
Constants (not user accessible)
Code (program instructions)
Heap (Default Size = 16 MBytes)
Stack (Default Size = 16 MBytes)
Adrs = $0
Reserved by O/S (Typically 128 KBytes)

Figure 3-7: HLA Typical Run-Time Memory Organization.

The operating system reserves the lowest memory addresses. Generally, your application cannot access data (or execute instructions) at these low addresses. One reason the OS reserves this space is to help trap NULL pointer references. If you attempt to access memory location zero, the operating system will generate a "general protection fault" meaning you've accessed a memory location that doesn't contain valid data. Because programmers often initialize pointers to NULL (zero) to indicate that the pointer is not pointing anywhere, an access of location zero typically means that the programmer has made a mistake and has not properly initialized a pointer to a legal (non-NULL) value. Also note that if you attempt to use one of the 80x86 16-bit addressing modes (HLA doesn't allow this, but were you to encode the instruction yourself and execute it . . .) the address will always be in the range 0..$1FFFE.[4] This will also access a location in the reserved area, generating a fault.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the stack section, the heap section, the code section, the `readonly` section, the `static` section, and the `storage` section. Each of these memory sections correspond to some type of data you can create in your HLA programs. The following sections discuss each of these sections in detail.

3.3.1 The Code Section

The code section contains the machine instructions that appear in an HLA program. HLA translates each machine instruction you write into a sequence of one or more byte values. The CPU interprets these byte values as machine instructions during program execution.

By default, when HLA links your program it tells the system that your program can execute instructions in the code segment and you can read data from the code segment. Note, specifically, that you cannot write data to the code segment. The operating system will generate a general protection fault if you attempt to store any data into the code segment.

Machine instructions are nothing more than data bytes. In theory, you could write a program that stores data values into memory and then transfers control to the data it just wrote, thereby producing a program that writes itself as it executes. This possibility produces romantic visions of *artificially intelligent* programs that modify themselves to produce some desired result. In real life, the effect is somewhat less glamorous.

Prior to the popularity of *protected mode operating systems*, like Windows and Linux, a program could overwrite the machine instructions during execution. Most of the time this was due to defects in a program, not because the program was artificially intelligent. A program would begin writing data to some array and fail to stop once it reached the end of the array, eventually overwriting the executing instructions that make up the program. Far from improving the quality of the code, such a defect usually causes the program to fail spectacularly.

Of course, if a feature is available, someone is bound to take advantage of it. Some programmers have discovered that in some special cases, using *self-modifying code* — that is, a program that modifies its machine instructions during execution — can produce slightly faster or slightly smaller programs. Unfortunately, self-modifying code is very difficult to test and debug. Given the speed of modern processors combined with their instruction set and wide variety of addressing modes, there is almost no reason to use self-modifying code in a modern program. Indeed, protected mode operating systems like Linux and Windows make it difficult for you to write self-modifying code.

[4] It's $1FFFE, not $FFFF, because you could use the indexed addressing mode with a displacement of $FFFF along with the value $FFFF in a 16-bit register.

HLA automatically stores the data associated with your machine code into the code section. In addition to machine instructions, you can also store data into the code section by using the following pseudo-opcodes:[5]

- byte
- word
- dword
- uns8
- uns16
- uns32
- int8
- int16
- in32
- boolean
- char

The following byte statement exemplifies the syntax for each of these pseudo-opcodes:

```
byte comma_separated_list_of_byte_constants ;
```

Here are some examples:

```
     boolean     true;
     char          'A';
     byte         0, 1, 2;
     byte         "Hello", 0
     word         0, 2;
     int8         -5;
     uns32         356789, 0;
```

If more than one value appears in the list of values after the pseudo-opcode, HLA emits each successive value to the code stream. So the first byte statement above emits three bytes to the code stream, the values zero, one, and two. If a string appears within a byte statement, HLA emits one byte of data for each character in the string. Therefore, the second byte statement above emits six bytes: the characters 'H', 'e', 'l', 'l', and 'o', followed by a zero byte.

Keep in mind that the CPU will attempt to treat data you emit to the code stream as machine instructions unless you take special care not to allow the execution of the data. For example, if you write something like the following:

```
          mov( 0, ax );
          byte 0,1,2,3;
          add( bx, cx );
```

[5] This isn't a complete list. HLA generally allows you to use any scalar data type name as a statement to reserve storage in the code section. You'll learn more about the available data types later in this text.

Your program will attempt to execute the 0, 1, 2, and 3 byte values as a machine instruction after executing the mov. Unless you know the machine code for a particular instruction sequence, sticking such data values into the middle of your code will almost always produce unexpected results. More often than not, this will crash your program. Therefore, you should never insert arbitrary data bytes into the middle of an instruction stream unless you understand exactly what you are doing. Typically when you place such data in your programs, you'll execute some code that transfers control around the data.

3.3.2 The Static Sections

The static section is where you will typically declare your variables. Although the static section syntactically appears as part of a program or procedure, keep in mind that HLA moves all static variables to the static section in memory. Therefore, HLA does not sandwich the variables you declare in the static section between procedures in the code section.

In addition to declaring static variables, you can also embed lists of data into the static declaration section. You use the same technique to embed data into your static section that you use to embed data into the code section: You use the byte, word, dword, uns32, and so on, pseudo-opcodes. Consider the following example:

```
static
    b:    byte := 0;
        byte 1,2,3;

    u:    uns32 := 1;
        uns32 5,2,10;

    c:    char;
        char 'a', 'b', 'c', 'd', 'e', 'f';

    bn: boolean;
        boolean true;
```

Data that HLA writes to the static memory segment using these pseudo-opcodes is written to the segment after the preceding variables. For example, the byte values 1, 2, and 3 are emitted to the static section after b's 0 byte. Because there aren't any labels associated with these values, you do not have direct access to these values in your program. You can use the indexed addressing modes to access these extra values (examples will appear a little later in this chapter).

In the examples above, note that the c and bn variables do not have an (explicit) initial value. However, if you don't provide an initial value, HLA will initialize the variables in the static section to all zero bits, so HLA assigns the NULL character (ASCII code zero) to c as its initial value. Likewise, HLA assigns false as the initial value for bn. In particular, you should note that your variable declarations in the static section always consume memory, even if you haven't assigned them an initial value. Any data you declare in a pseudo-opcode like byte will always follow the actual data associated with the variable declaration.

3.3.3 The Read-Only Data Section

The readonly data section holds constants, tables, and other data that your program cannot change during execution. You create read-only objects by declaring them in the readonly declaration section. The readonly section is very similar to the static section with three primary differences:

- The readonly section begins with the reserved word readonly rather than static.
- All declarations in the readonly section generally have an initializer.
- The system does not allow you to store data into a readonly object while the program is running.

Example:

```
readonly
     pi:              real32 := 3.14159;
     e:              real32 := 2.71;
     MaxU16:         uns16 := 65_535;
     MaxI16:         int16 := 32_767;
```

All readonly object declarations must have an initializer because you cannot initialize the value under program control.[6] For all intents and purposes, you can think of readonly objects as constants. However, these constants consume memory and other than you cannot write data to readonly objects, they behave like, and you can use them like, static variables. Because they behave like static objects, you cannot use a readonly object everywhere a constant is allowed; in particular, readonly objects are memory objects, so you cannot supply a readonly object and some other memory object as the operands to an instruction.[7]

Like the static section, you may embed data values in the readonly section using the byte, word, dword, and so on, data declarations, e.g.,

```
readonly
     roArray: byte := 0;
                    byte 1, 2, 3, 4, 5;
     qwVal: qword := 1;
               qword 0;
```

[6] There is one exception you'll see a little later in this chapter.

[7] mov is an exception to this rule because HLA emits special code for memory-to-memory move operations.

3.3.4 The Storage Section

The `readonly` section requires that you initialize all objects you declare. The `static` section lets you optionally initialize objects (or leave them uninitialized, in which case they have the default initial value of zero). The `storage` section completes the initialization coverage: You use it to declare variables that are always uninitialized when the program begins running. The `storage` section begins with the `storage` reserved word and contains variable declarations without initializers. Here is an example:

```
storage
     UninitUns32:     uns32;
     i:               int32;
     character:       char;
     b:               byte;
```

Linux and Windows will initialize all storage objects to zero when they load your program into memory. However, it's probably not a good idea to depend upon this implicit initialization. If you need an object initialized with zero, declare it in a `static` section and explicitly set it to zero.

Variables you declare in the `storage` section may consume less disk space in the executable file for the program. This is because HLA writes out initial values for `readonly` and `static` objects to the executable file, but uses a compact representation for uninitialized variables you declare in the `storage` section; note, however, that this behavior is OS and object-module format dependent. Because the `storage` section does not allow initialized values, you *cannot* put unlabeled values in the `storage` section using the `byte`, `word`, `dword`, and so on, pseudo-opcodes.

3.3.5 The @NOSTORAGE Attribute

The `@nostorage` attribute lets you declare variables in the static data declaration sections (i.e., `static`, `readonly`, and `storage`) without actually allocating memory for the variable. The `@nostorage` option tells HLA to assign the current address in a declaration section to a variable but do not allocate any storage for the object. That variable will share the same memory address as the next object appearing in the variable declaration section. Here is the syntax for the `@nostorage` option:

```
     variableName: varType; @nostorage;
```

Note that you follow the type name with "@nostorage;" rather than some initial value or just a semicolon. The following code sequence provides an example of using the `@nostorage` option in the `readonly` section:

```
readonly
     abcd: dword; nostorage;
               byte 'a', 'b', 'c', 'd';
```

In this example, *abcd* is a double word whose L.O. byte contains 97 ('a'), byte #1 contains 98 ('b'), byte #2 contains 99 ('c'), and the H.O. byte contains 100 ('d'). HLA does not reserve storage for the abcd variable, so HLA associates the following four bytes in memory (allocated by the byte directive) with abcd.

Note that the @nostorage attribute is only legal in the static, storage, and readonly sections (the so-called *static* declarations sections). HLA does not allow its use in the var section that you'll read about next.

3.3.6 The Var Section

HLA provides another variable declaration section, the var section, that you can use to create *automatic* variables. Your program will allocate storage for automatic variables whenever a program unit (i.e., main program or procedure) begins execution, and it will deallocate storage for automatic variables when that program unit returns to its caller. Of course, any automatic variables you declare in your main program have the same *lifetime*[8] as all the static, readonly, and storage objects, so the automatic allocation feature of the var section is wasted in the main program. In general, you should only use automatic objects in procedures (see the chapter on procedures for details). HLA allows them in your main program's declaration section as a generalization.

Because variables you declare in the var section are created at runtime, HLA does not allow initializers on variables you declare in this section. So the syntax for the var section is nearly identical to that for the storage section; the only real difference in the syntax between the two is the use of the var reserved word rather than the storage reserved word.[9] The following example illustrates this:

```
var
    vInt:       int32;
    vChar:      char;
```

HLA allocates variables you declare in the var section in the stack memory section. HLA does not allocate var objects at fixed locations within the stack segment; instead, it allocates these variables in an activation record associated with the current program unit. The chapter on procedures, later in this book, will discuss activation records in greater detail; for now it is important only to realize that HLA programs use the EBP register as a pointer to the current activation record. Therefore, any time you access a var object, HLA automatically replaces the variable name with "[EBP±displacement]". Displacement is the offset of the object in the activation record. This means that you cannot use the full scaled indexed addressing mode (a base register plus a scaled index register) with var objects because var objects already use the EBP register as their base register. Although you will not directly use the two register addressing modes often, the fact that the var section has this limitation is a good reason to avoid using the var section in your main program.

[8] The lifetime of a variable is the point from which memory is first allocated to the point the memory is deallocated for that variable.

[9] Actually, there are a few other, minor differences, but we won't deal with those differences in this text. See the HLA Reference Manual for more details.

3.3.7 Organization of Declaration Sections Within Your Programs

The static, readonly, storage, and var sections may appear zero or more times between the program header and the associated begin for the main program. Between these two points in your program, the declaration sections may appear in any order, as the following example demonstrates:

```
program demoDeclarations;

static
     i_static:     int32;

var
     i_auto:     int32;

storage
     i_uninit:     int32;

readonly
     i_readonly: int32 := 5;

static
     j:     uns32;

var
     k:     char;

readonly
     i2:     uns8 := 9;

storage
     c:     char;

storage
     d:     dword;

begin demoDeclarations;

     << code goes here >>

end demoDeclarations;
```

In addition to demonstrating that the sections may appear in an arbitrary order, this section also demonstrates that a given declaration section may appear more than once in your program. When multiple declaration sections of the same type (e.g., the three storage sections above) appear in a declaration section of your program, HLA combines them into a single group.

3.4 How HLA Allocates Memory for Variables

As you've seen, the 80x86 CPU doesn't deal with variables that have names like `I`, `Profits`, and `LineCnt`. The CPU deals strictly with numeric addresses it can place on the address bus like $1234_5678, $0400_1000, and $8000_CC00. HLA, on the other hand, does not force you refer to variable objects by their addresses (which is nice, because names are so much easier to remember). This abstraction (allowing the use of names rather than numeric addresses in your programs) is nice, but it does obscure what is really going on. In this section, we'll take a look at how HLA associates numeric addresses with your variables so you'll understand (and appreciate) the process that is taking place behind your back.

Take another look at Figure 3-7. As you can see, the various memory sections tend to be adjacent to one another. Therefore, if the size of one memory section changes, then this affects the starting address of all the following sections in memory. For example, if you add a few additional machine instructions to your program and increase the size of the code section, this may affect the starting address of the `static` section in memory, thus changing the addresses of all your static variables.[10] Keeping track of variables by their numeric address (rather than by their names) is difficult enough; imagine how much worse it would be if the addresses were constantly shifting around as you add and remove machine instructions in your program! Fortunately, you don't have to keep track of all of this, HLA does that bookkeeping for you.

HLA associates a current *location counter* with each of the three `static` declaration sections (`static`, `readonly`, and `storage`). These location counters initially contain zero and whenever you declare a variable in one of the static sections, HLA associates the current value of that section's location counter with the variable; HLA also bumps up the value of that location counter by the size of the object you're declaring. As an example, assume that the following is the only `static` declaration section in a program:

```
static
    b    :byte;              // Location counter = 0, size = 1
    w    :word;              // Location counter = 1, size = 2
    d    :dword;             // Location counter = 3, size = 4
    q    :qword;             // Location counter = 7, size = 8
    l    :lword;             // Location counter = 15, size = 16
    // Location counter is now 31.
```

Of course, the run-time address of each of these variables is not the value of the location counter. First of all, HLA adds in the base address of the `static` memory section to each of these location counter values (that we call *displacements* or *offsets*). Secondly, there may be other `static` objects in modules that you link with your program (e.g., from the HLA Standard Library), or even additional `static` sections in the same source file, and the linker has to merge the `static` sections together. Hence, these offsets may have very little bearing on the final address of

[10] Note that the operating system typically aligns the static section on a 4,096-byte boundary, so you many need to add a sufficient number of new instructions to cause the code section to grow in size across a 4K boundary before the static addresses actually change. This isn't necessarily true for all memory sections, however.

these variables in memory. Nevertheless, one important fact remains: HLA allocates variables you declare in a single `static` declaration section in contiguous memory locations. That is, given the declaration above, `w` will immediately follow `b` in memory, `d` will immediately follow `w` in memory, `q` will immediately follow `d`, and so on. Generally, it's not good coding style to assume that the system allocates variables this way, but sometimes it's convenient to do so.

Note that HLA allocates memory objects you declare in `readonly`, `static`, and `storage` sections in completely different regions of memory. Therefore, you cannot assume that the following three memory objects appear in adjacent memory locations (indeed, they probably will not):

```
static
     b     :byte;
readonly
     w     :word := $1234;
storage
     d     :dword;
```

In fact, HLA will not even guarantee that variables you declare in separate `static` (or whatever) sections are adjacent in memory, even if there is nothing between the declarations in your code (e.g., you cannot assume that `b`, `w`, and `d` are in adjacent memory locations in the following declarations, nor can you assume that they *won't* be adjacent in memory):

```
static
     b     :byte;
static
     w     :word := $1234;
static
     d     :dword;
```

If your code requires these variables to consume adjacent memory locations, you must declare them in the same `static` section.

Note that HLA handles variables you declare in the `var` section a little differently than the variables you declare in one of the `static` sections. We'll discuss the allocation of offsets to `var` objects in the chapter on procedures.

3.5 HLA Support for Data Alignment

In order to write fast programs, you need to ensure that you properly align data objects in memory. Proper alignment means that the starting address for an object is a multiple of some size, usually the size of object if the object's size is a power of two for values up to 16 bytes in length. For objects greater than 16 bytes, aligning the object on an 8-byte or 16-byte address boundary is probably sufficient. For objects less than 16 bytes, aligning the object at an address that is the next power of two greater than the object's size is usually fine.[11] Accessing

[11] An exception are the `real80` and `tbyte` (80-bit) types. These only need to be aligned on an address that is a multiple of eight bytes in memory.

data that is not aligned on at an appropriate address may require extra time; so if you want to ensure that your program runs as rapidly as possible, you should try to align data objects according to their size.

Data becomes misaligned whenever you allocate storage for different-sized objects in adjacent memory locations. For example, if you declare a byte variable, it will consume one byte of storage, and the next variable you declare in that declaration section will have the address of that byte object plus one. If the byte variable's address happens to be an address that is an even address, then the variable following that byte will start at an odd address. If that following variable is a word or double-word object, then its starting address will not be optimal. In this section, we'll explore ways to ensure that a variable is aligned at an appropriate starting address based on that object's size.

Consider the following HLA variable declarations:

```
static
    dw:   dword;
    b:    byte;
    w:    word;
    dw2:  dword;
    w2:   word;
    b2:   byte;
    dw3:  dword;
```

The first static declaration in a program (running under Windows, Linux, and most 32-bit operating systems) places its variables at an address that is an even multiple of 4096 bytes. Whatever variable first appears in the static declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of the sizes of all the preceding variables plus the starting address of that static section. Therefore, assuming HLA allocates the variables in the previous example at a starting address of 4096, HLA will allocate them at the following addresses:

```
                    // Start Adrs          Length
    dw:   dword;    //     4096                4
    b:    byte;     //     4100                1
    w:    word;     //     4101                2
    dw2:  dword;    //     4103                4
    w2:   word;     //     4107                2
    b2:   byte;     //     4109                1
    dw3:  dword;    //     4110                4
```

With the exception of the first variable (that is aligned on a 4K boundary) and the byte variables (whose alignment doesn't matter), all of these variables are misaligned. The `w`, `w2`, and `dw2` variables start at odd addresses, and the `dw3` variable is aligned on an even address that is not a multiple of four.

An easy way to guarantee that your variables are aligned properly is to put all the double-word variables first, the word variables second, and the byte variables last in the declaration:

```
static
     dw:    dword;
     dw2:   dword;
     dw3:   dword;
     w:     word;
     w2:    word;
     b:     byte;
     b2:    byte;
```

This organization produces the following addresses in memory:

```
                       // Start Adrs           Length
     dw:    dword;     //      4096               4
     dw2:   dword;     //      4100               4
     dw3:   dword;     //      4104               4
     w:     word;      //      4108               2
     w2:    word;      //      4110               2
     b:     byte;      //      4112               1
     b2:    byte;      //      4113               1
```

As you can see, these variables are all aligned at reasonable addresses.

Unfortunately, it is rarely possible for you to arrange your variables in this manner. While there are many technical reasons that make this alignment impossible, a good practical reason for not doing this is because it doesn't let you organize your variable declarations by logical function (that is, you probably want to keep related variables next to one another regardless of their size).

To resolve this problem, HLA provides the `align` directive. The `align` directive uses the following syntax:

```
align( integer_constant );
```

The integer constant must be one of the following small unsigned integer values: 1, 2, 4, 8, or 16. If HLA encounters the `align` directive in a `static` section, it will align the very next variable on an address that is an even multiple of the specified alignment constant. The previous example could be rewritten, using the `align` directive, as follows:

```
static
     align( 4 );
     dw:    dword;
     b:     byte;
     align( 2 );
     w:     word;
     align( 4 );
     dw2:   dword;
     w2:    word;
     b2:    byte;
```

```
    align( 4 );
    dw3:   dword;
```

If you're wondering how the `align` directive works, it's really quite simple. If HLA determines that the current address (location counter value) is not an even multiple of the specified value, HLA will quietly emit extra bytes of padding after the previous variable declaration until the current address in the `static` section is an even multiple of the specified value. This has the effect of making your program slightly larger (by a few bytes) in exchange for faster access to your data; given that your program will only grow by a few bytes when you use this feature, this is probably a good trade-off.

As a general rule, if you want the fastest possible access you should choose an alignment value that is equal to the size of the object you want to align. That is, you should align words to even boundaries using an "align(2);" statement, double words to four-byte boundaries using "align(4);", quad words to eight-byte boundaries using "align(8);", and so on. If the object's size is not a power of two, align it to the next higher power of two (up to a maximum of 16 bytes). Note, however, that you need only align `real80` (and `tbyte`) objects on an eight-byte boundary.

Note that data alignment isn't always necessary. The cache architecture of modern 80x86 CPUs actually handles most misaligned data. Therefore, you should only use the alignment directives with variables for whom speedy access is absolutely critical. This is a reasonable space/speed trade-off.

3.6 Address Expressions

Earlier, this chapter points out that addressing modes take a couple generic forms, including:

```
VarName[ Reg32 ]
VarName[ Reg32 + offset ]
VarName[ RegNotESP32*Scale ]
VarName[ Reg32 + RegNotESP32*Scale ]
VarName[ RegNotESP32*Scale + offset ]
and
VarName[ Reg32 + RegNotESP32*Scale + offset ]
```

Another legal form, which isn't actually a new addressing mode but simply an extension of the displacement-only addressing mode, is

```
VarName[ offset ]
```

This latter example computes its effective address by adding the constant offset within the brackets to the variable's address. For example, the instruction "mov(Address[3], AL);" loads the AL register with the byte in memory that is three bytes beyond the `Address` object (see Figure 3-8).

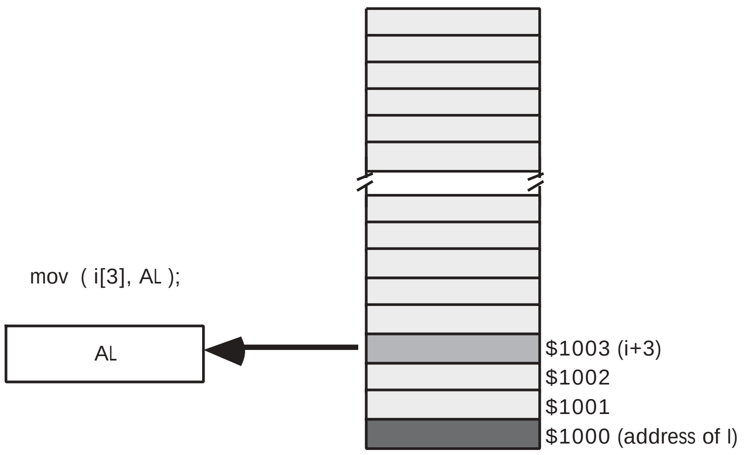


Figure 3-8: Using an Address Expression to Access Data Beyond a Variable.

Always remember that the `offset` value in these examples must be a constant. If `Index` is an `int32` variable, then `Variable[Index]` is not a legal address expression. If you wish to specify an index that varies at runtime, then you must use one of the indexed or scaled indexed addressing modes.

Another important thing to remember is that the offset in `Address[offset]` is a byte address. Despite the fact that this syntax is reminiscent of array indexing in a high level language like C/C++ or Pascal, this does not properly index into an array of objects unless `Address` is an array of bytes.

This text will consider an *address expression* to be any legal 80x86 addressing mode that includes a displacement (i.e., variable name) or an offset. In addition to the above forms, the following are also address expressions:

```
[ Reg32 + offset ]
[ Reg32 + RegNotESP32*Scale + offset ]
```

This book will *not* consider the following to be address expressions because they do not involve a displacement or offset component:

```
[ Reg32 ]
[ Reg32 + RegNotESP32*Scale ]
```

Address expressions are special because those instructions containing an address expression always encode a displacement constant as part of the machine instruction. That is, the machine instruction contains some number of bits (usually 8 or 32) that hold a numeric constant. That constant is the sum of the displacement (i.e., the address or offset of the variable) plus the offset. Note that HLA automatically adds these two values together for you (or subtracts the offset if you use the “-” rather than “+” operator in the addressing mode).

Until this point, the offset in all the addressing mode examples has always been a single numeric constant. However, HLA also allows a *constant expression* anywhere an offset is legal. A constant expression consists of one or more constant terms manipulated by operators such as addition, subtraction, multiplication, division, modulo, and a wide variety of other operators. Most address expressions, however, will only involve addition, subtraction, multiplication, and sometimes, division. Consider the following example:

```
mov( X[ 2*4+1 ], al );
```

This instruction will move the byte at address `X+9` into the AL register.

The value of an address expression is always computed at compiletime, never while the program is running. When HLA encounters the instruction above, it calculates 2*4+1 on the spot and adds this result to the base address of `X` in memory. HLA encodes this single sum (base address of `X` plus nine) as part of the instruction; HLA does not emit extra instructions to compute this sum for you at runtime (which is good, because doing so would be less efficient). Because HLA computes the value of address expressions at compiletime, all components of the expression must be constants because HLA cannot know the run-time value of a variable while it is compiling the program.

Address expressions are useful for accessing the data in memory beyond a variable, particularly when you've used the `byte`, `word`, `dword`, and so on, statements in a `static` or `readonly` section to tack on additional bytes after a data declaration. For example, consider the program in Listing 3-1.

Listing 3-1: Demonstration of Address Expressions.

```
program adrsExpressions;
#include( "stdlib.hhf" )
static
  i: int8; @nostorage;
    byte 0, 1, 2, 3;

begin adrsExpressions;

  stdout.put
  (
    "i[0]=", i[0], nl,
    "i[1]=", i[1], nl,
    "i[2]=", i[2], nl,
    "i[3]=", i[3], nl
  );

end adrsExpressions;
```

The program in Listing 3-1 will display the four values 0, 1, 2, and 3 as though they were array elements. This is because the value at the address of `i` is 0 (this program declares `i` using the `@nostorage` option, so `i` is the address of the next object in the `static` section, which just happens to be the value 0 appearing as part of the `byte` statement). The address expression "i[1]" tells HLA to fetch the byte appearing at `i`'s address plus one. This is the value one, because the `byte` statement in this program emits the value one to the static segment immediately after the value 0. Likewise for i[2] and i[3], this program displays the values 2 and 3.

3.7 Type Coercion

Although HLA is fairly loose when it comes to type checking, HLA does ensure that you specify appropriate operand sizes to an instruction. For example, consider the following (incorrect) program:

```
program hasErrors;
static
     i8:     int8;
     i16:    int16;
     i32:    int32;
begin hasErrors;

     mov( i8, eax );
     mov( i16, al );
     mov( i32, ax );

end hasErrors;
```

HLA will generate errors for these three mov instructions. This is because the operand sizes are incompatible. The first instruction attempts to move a byte into EAX, the second instruction attempts to move a word into AL and the third instruction attempts to move a double word into AX. The mov instruction, of course, requires both operands to be the same size.

While this is a good feature in HLA,[12] there are times when it gets in the way. Consider the following code fragments:

```
static
     byte_values: byte; @nostorage;
                    byte  0, 1;

     ...

          mov( byte_values, ax );
```

In this example let's assume that the programmer really wants to load the word starting at the address of byte_values into the AX register because they want to load AL with 0 and AH with 1 using a single instruction. HLA will refuse, claiming there is a type mismatch error (because byte_values is a byte object and AX is a word object). The programmer could break this into two instructions, one to load AL with the byte at address byte_values and the other to load AH with the byte at address byte_values[1]. Unfortunately, this decomposition makes the program slightly less efficient (which was probably the reason for using the single

[12] After all, if the two operand sizes are different this usually indicates an error in the program.

mov instruction in the first place). Somehow, it would be nice if we could tell HLA that we know what we're doing and we want to treat the byte_values variable as a word object. HLA's *type coercion* facilities provide this capability.

Type coercion[13] is the process of telling HLA that you want to treat an object as an explicit type, regardless of its actual type. To coerce the type of a variable, you use the following syntax:

```
(type newTypeName addressExpression)
```

The newTypeName item is the new type you wish to associate with the memory location specified by addressExpression. You may use this coercion operator anywhere a memory address is legal. To correct the previous example, so HLA doesn't complain about type mismatches, you would use the following statement:

```
mov( (type word byte_values), ax );
```

This instruction tells HLA to load the AX register with the word starting at address byte_values in memory. Assuming byte_values still contains its initial values, this instruction will load zero into AL and one into AH.

Type coercion is necessary when you specify an anonymous variable as the operand to an instruction that directly modifies memory (e.g., neg, shl, not, and so on). Consider the following statement:

```
not( [ebx] );
```

HLA will generate an error on this instruction because it cannot determine the size of the memory operand. The instruction does not supply sufficient information to determine whether the program should invert the bits in the byte pointed at by EBX, the word pointed at by EBX, or the double word pointed at by EBX. You must use type coercion to explicitly specify size of anonymous references with these types of instructions:

```
not( (type byte [ebx]) );
not( (type dword [ebx]) );
```

CAUTION *Do not use the type coercion operator unless you know exactly what you are doing and fully understand the effect it has on your program. Beginning assembly language programmers often use type coercion as a tool to quiet the compiler when it complains about type mismatches without solving the underlying problem. Consider the following statement (where* byteVar *is an 8-bit variable):*

```
mov( eax, (type dword byteVar) );
```

Without the type coercion operator, HLA complains about this instruction because it attempts to store a 32-bit register into an 8-bit memory location. A beginning programmer, wanting their program to compile, may take a shortcut and use the type coercion operator as shown in this instruction; this certainly quiets the compiler — it will no longer complain about a type mismatch. So the

[13] Also called *type casting* in some languages.

beginning programmer is happy. But the program is still incorrect; the only difference is that HLA no longer warns you about your error. The type coercion operator does not fix the problem of attempting to store a 32-bit value into an 8-bit memory location — it simply allows the instruction to store a 32-bit value *starting at the address specified by the 8-bit variable*. The program still stores four bytes, overwriting the three bytes following `byteVar` in memory. This often produces unexpected results including the phantom modification of variables in your program.[14] Another, rarer, possibility is for the program to abort with a general protection fault. This can occur if the three bytes following `byteVar` are not allocated in real memory or if those bytes just happen to fall in a read-only segment in memory. The important thing to remember about the type coercion operator is this: "If you cannot exactly state the affect this operator has, don't use it."

Also keep in mind that the type coercion operator does not perform any translation of the data in memory. It simply tells the compiler to treat the bits in memory as a different type. It will not automatically sign extend an 8-bit value to 32 bits nor will it convert an integer to a floating-point value. It simply tells the compiler to treat the bit pattern of the memory operand as a different type.

3.8 Register Type Coercion

You can also cast a register to a specific type using the type coercion operator. By default, the 8-bit registers are of type `byte`, the 16-bit registers are of type `word`, and the 32-bit registers are of type `dword`. With type coercion, you can cast a register as a different type *as long as the size of the new type agrees with the size of the register*. This is an important restriction that does not exist when applying type coercion to a memory variable.

Most of the time you do not need to coerce a register to a different type. As `byte`, `word`, and `dword` objects, registers are already compatible with all one, two, and four byte objects. However, there are a few instances where register type coercion is handy, if not downright necessary. Two examples include boolean expressions in HLA high-level language statements (e.g., `if` and `while`) and register I/O in the `stdout.put` and `stdin.get` (and related) statements.

In boolean expressions, HLA always treats `byte`, `word`, and `dword` objects as unsigned values. Therefore, without type coercion, the following `if` statement always evaluates false (because there is no unsigned value less than zero):

```
if( eax < 0 ) then

    stdout.put( "EAX is negative!", nl );

endif;
```

You can overcome this limitation by casting EAX as an `int32` value:

[14] If you have a variable immediately following `byteVar` in this example, the `mov` instruction will surely overwrite the value of that variable, whether or not you intend for this to happen.

```
if( (type int32 eax) < 0 ) then

    stdout.put( "EAX is negative!", nl );

endif;
```

In a similar vein, the HLA Standard Library `stdout.put` routine always outputs `byte`, `word`, and `dword` values as hexadecimal numbers. Therefore, if you attempt to print a register, the `stdout.put` routine will print it as a hex value. If you would like to print the value as some other type, you can use register type coercion to achieve this:

```
stdout.put( "AL printed as a char = '", (type char al), "'", nl );
```

The same is true for the `stdin.get` routine. It will always read a hexadecimal value for a register unless you coerce its type to something other than `byte`, `word`, or `dword`.

3.9 The Stack Segment and the PUSH and POP Instructions

This chapter mentions that all variables you declare in the `var` section wind up in the stack memory segment. However, `var` objects are not the only things in the stack memory section; your programs manipulate data in the stack segment in many different ways. This section introduces the `push` and `pop` instructions that also manipulate data in stack memory.

The stack segment in memory is where the 80x86 maintains the stack. The *stack* is a dynamic data structure that grows and shrinks according to certain needs of the program. The stack also stores important information about program including local variables, subroutine information, and temporary data.

The 80x86 controls its stack via the ESP (stack pointer) register. When your program begins execution, the operating system initializes ESP with the address of the last memory location in the stack memory segment. Data is written to the stack segment by "pushing" data onto the stack and "popping" or "pulling" data off of the stack. Whenever you push data onto the stack, the 80x86 decrements the stack pointer by the size of the data you are pushing, and then it copies the data to memory where ESP is then pointing. Therefore, the stack grows and shrinks as you push data onto the stack and `pop` data from the stack.

3.9.1 The Basic PUSH Instruction

Consider the syntax for the 80x86 `push` instruction:

```
push( reg16 );
push( reg32 );
push( memory16 );
push( memory32 );
```

```
pushw( constant );
pushd( constant );
```

The pushw and pushd operands are always two- or four-byte constants, respectively.

These six forms allow you to push word or dword registers, memory locations, and constants. You should specifically note that you cannot push byte values onto the stack.

The push instruction does the following:

```
ESP := ESP - Size_of_Register_or_Memory_Operand (2 or 4)
 [ESP] := Operand's_Value
```

Assuming that ESP contains $00FF_FFE8, then the instruction "push(eax);" will set ESP to $00FF_FFE4, and store the current value of EAX into memory location $00FF_FFE4 as Figures 3-9 and 3-10 show.

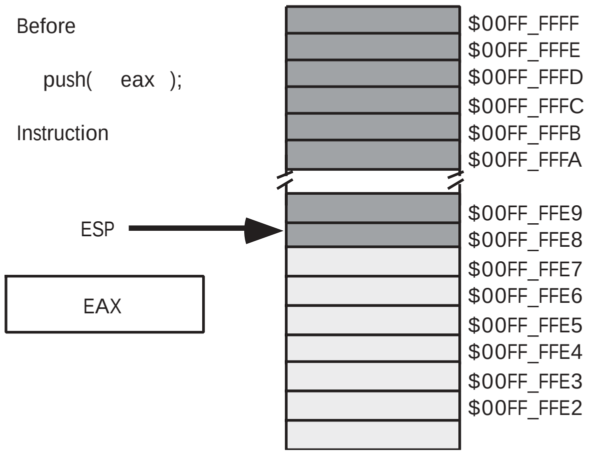


Figure 3-9: Before "PUSH(EAX);" Operation.

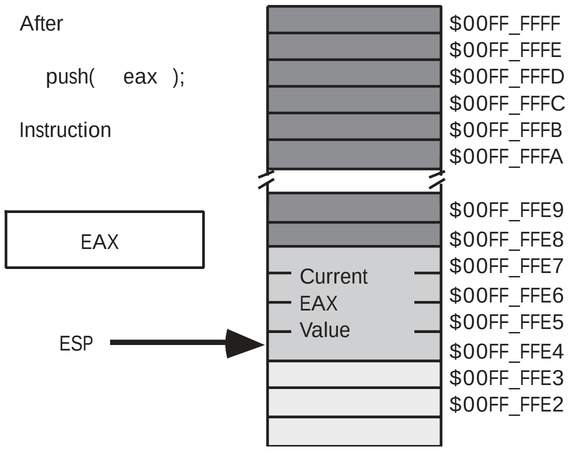


Figure 3-10: Stack Segment After "PUSH(EAX);" Operation.

Note that the "push(eax);" instruction does not affect the value of the EAX register.

Although the 80x86 supports 16-bit push operations, their primary use in is 16-bit environments such as DOS. For maximum performance, the stack pointer's value should always be an even multiple of four; indeed, your program may malfunction under Windows or Linux if ESP contains a value that is not a multiple of four and you make an operating system API call. The only practical reason for pushing less than four bytes at a time on the stack is because you're building up a double word via two successive word pushes.

3.9.2 The Basic POP Instruction

To retrieve data you've pushed onto the stack, you use the pop instruction. The basic pop instruction allows the following different forms:

```
pop( reg16 );
pop( reg32 );
pop( memory16 );
pop( memory32 );
```

Like the push instruction, the pop instruction only supports 16-bit and 32-bit operands; you cannot pop an 8-bit value from the stack. Also like the push instruction, you should avoid popping 16-bit values (unless you do two 16-bit pops in a row) because 16-bit pops may leave the ESP register containing a value that is not an even multiple of four. One major difference between push and pop is that you cannot pop a constant value (which makes sense, because the operand for push is a source operand while the operand for pop is a destination operand).

Formally, here's what the pop instruction does:

```
Operand := [ESP]
ESP := ESP + Size_of_Operand (2 or 4)
```

As you can see, the pop operation is the converse of the push operation. Note that the pop instruction copies the data from memory location [ESP] before adjusting the value in ESP. See Figures 3-11 and 3-12 for details on this operation.

Note that the value popped from the stack is still present in memory. Popping a value does not erase the value in memory; it just adjusts the stack pointer so that it points at the next value above the popped value. However, you should never attempt to access a value you've popped off the stack. The next time something is pushed onto the stack, the popped value will be obliterated. Because your code isn't the only thing that uses the stack (i.e., the operating system uses the stack as do subroutines), you cannot rely on data remaining in stack memory once you've popped it off the stack.

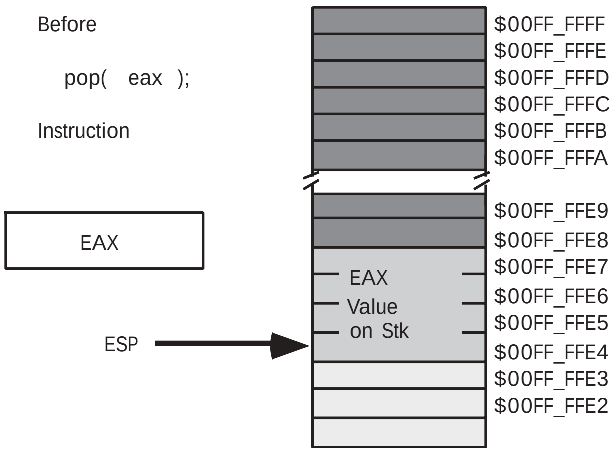


Figure 3-11: Memory Before a "POP(EAX);" Operation.

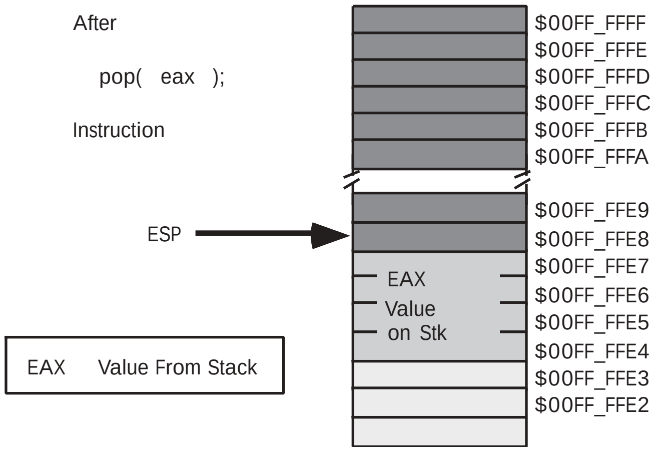


Figure 3-12: Memory After the "POP(EAX);" Instruction.

As Chapter One notes, HLA provides an extended syntax for the `mov` instruction that allows two memory operands (that is, the instruction provides a memory-to-memory move). HLA actually generates the following two instructions in place of such a `mov`:

```
// mov( src, dest );

        push( src );
        pop( dest );
```

This is the reason that the memory-to-memory form of the `mov` instruction only allows 16-bit and 32-bit operands — because `push` and `pop` only allow 16-bit and 32-bit operands.

3.9.3 Preserving Registers with the PUSH and POP Instructions

Perhaps the most common use of the `push` and `pop` instructions is to save register values during intermediate calculations. A problem with the 80x86 architecture is that it provides very few general purpose registers. Because registers are the best place to hold temporary values, and registers are also needed for the various addressing modes, it is very easy to run out of registers when writing code that performs complex calculations. The `push` and `pop` instructions can come to your rescue when this happens.

Consider the following program outline:

```
<< Some sequence of instructions that use the EAX register >>

<< Some sequence of instructions that need to use EAX, for a
     different purpose than the above instructions >>

<< Some sequence of instructions that need the original value in EAX >>
```

The `push` and `pop` instructions are perfect for this situation. By inserting a `push` instruction before the middle sequence and a `pop` instruction after the middle sequence above, you can preserve the value in EAX across those calculations:

```
<< Some sequence of instructions that use the EAX register >>
push( eax );
<< Some sequence of instructions that need to use EAX, for a
     different purpose than the above instructions >>
pop( eax );
<< Some sequence of instructions that need the original value in EAX >>
```

The `push` instruction above copies the data computed in the first sequence of instructions onto the stack. Now the middle sequence of instructions can use EAX for any purpose it chooses. After the middle sequence of instructions finishes, the `pop` instruction restores the value in EAX so the last sequence of instructions can use the original value in EAX.

3.9.4 The Stack Is a LIFO Data Structure

You can push more than one value onto the stack without first popping previous values off the stack. However, the stack is a *last-in, first-out* (LIFO) data structure, so you must be careful how you push and pop multiple values. For example, suppose you want to preserve EAX and EBX across some block of instructions. The following code demonstrates the obvious way to handle this:

```
push( eax );
push( ebx );
<< Code that uses EAX and EBX goes here >>
pop( eax );
pop( ebx );
```

Unfortunately, this code will not work properly! Figures 3-13 through 3-16 show the problem. Because this code pushes EAX first and EBX second, the stack pointer is left pointing at EBX's value on the stack. When the "pop(eax);" instruction comes along, it removes the value that was originally in EBX from the stack and places it in EAX! Likewise, the "pop(EBX);" instruction pops the value that was originally in EAX into the EBX register. The end result is that this code manages to swap the values in the registers by popping them in the same order that it pushes them.

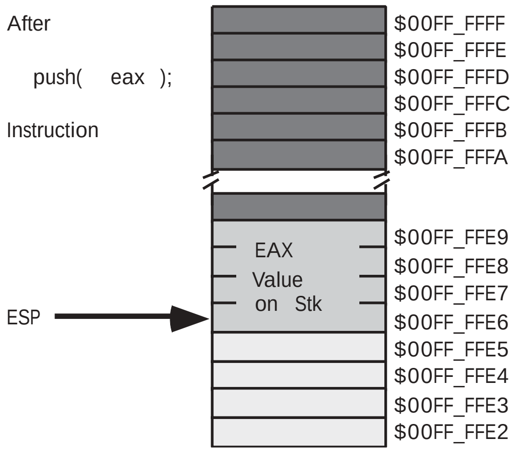


Figure 3-13: Stack After Pushing EAX.

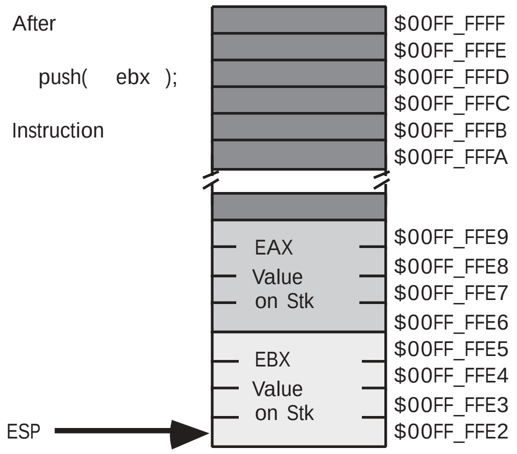


Figure 3-14: Stack After Pushing EBX.

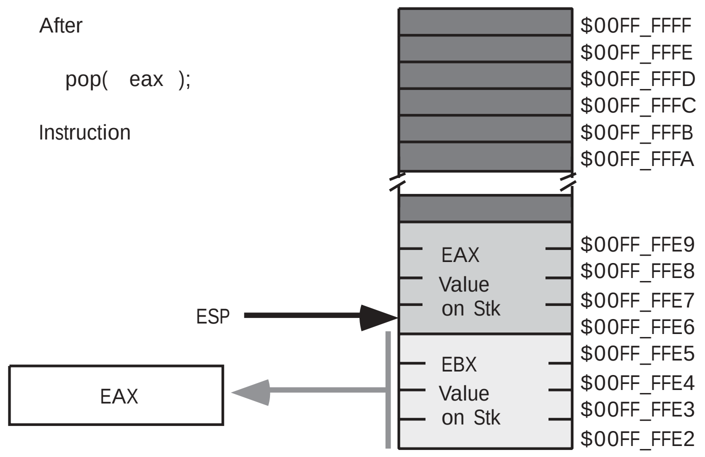


Figure 3-15: Stack After Popping EAX.

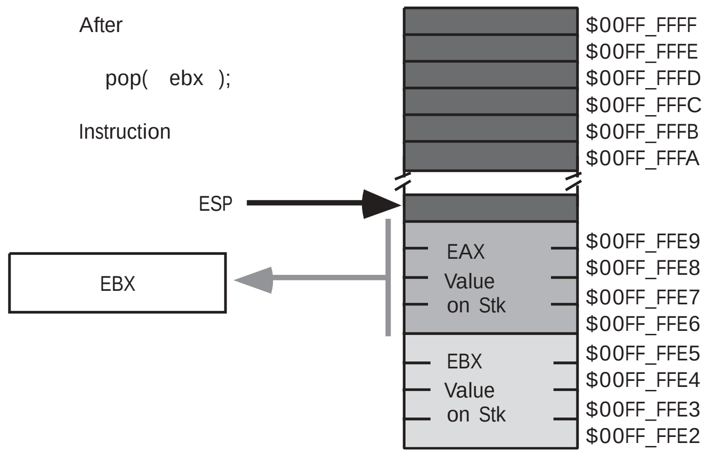


Figure 3-16: Stack After Popping EBX.

To rectify this problem, you must note that the stack is a LIFO data structure, so the first thing you must pop is the last thing you push onto the stack. Therefore, you must always observe the following maxim:

- Always pop values in the reverse order that you push them.

The correction to the previous code is

```
push( eax );
push( ebx );
<< Code that uses EAX and EBX goes here >>
pop( ebx );
pop( eax );
```

Another important maxim to remember is

- Always pop exactly the same number of bytes that you push.

This generally means that the number of pushes and pops must exactly agree. If you have too few pops, you will leave data on the stack, which may confuse the running program: If you have too many pops, you will accidentally remove previously pushed data, often with disastrous results.

A corollary to the maxim above is, “Be careful when pushing and popping data within a loop.” Often it is quite easy to put the pushes in a loop and leave the pops outside the loop (or vice versa), creating an inconsistent stack. Remember, it is the execution of the push and pop instructions that matters, not the number of push and pop instructions that appear in your program. At runtime, the number (and order) of the push instructions the program executes must match the number (and reverse order) of the pop instructions.

3.9.5 Other PUSH and POP Instructions

The 80x86 provides several additional push and pop instructions in addition to the basic push/pop instructions. These instructions include the following:

- pusha
- pushad
- pushf
- pushfd
- popa
- popad
- popf
- popfd

The pusha instruction pushes all the general-purpose 16-bit registers onto the stack. This instruction exists primarily for older 16-bit operating systems like DOS. In general, you will have very little need for this instruction. The pusha instruction pushes the registers onto the stack in the following order:

```
ax
cx
dx
bx
sp
bp
si
di
```

The pushad instruction pushes all the 32-bit (double-word) registers onto the stack. It pushes the registers onto the stack in the following order:

```
eax
ecx
edx
ebx
esp
```

```
ebp
esi
edi
```

Because the pusha and pushad instructions inherently modify the SP / ESP register, you may wonder why Intel bothered to push this register at all. It was probably easier in the hardware to go ahead and push SP / ESP rather than make a special case out of it. In any case, these instructions do push SP or ESP, so don't worry about it too much — there is nothing you can do about it.

The popa and popad instructions provide the corresponding "pop all" operation to the pusha and pushad instructions. This will pop the registers pushed by pusha or pushad in the appropriate order (that is, popa and popad will properly restore the register values by popping them in the reverse order that pusha or pushad pushed them).

Although the pusha/popa and pushad/popad sequences are short and convenient, they are actually slower than the corresponding sequence of push/pop instructions, this is especially true when you consider that you rarely need to push a majority, much less all the registers.[15] So if you're looking for maximum speed, you should carefully consider whether to use the pusha(d)/popa(d) instructions.

The pushf, pushfd, popf, and popfd instructions push and pop the (E)FLAGs register. These instructions allow you to preserve condition code and other flag settings across the execution of some sequence of instructions. Unfortunately, unless you go to a lot of trouble, it is difficult to preserve individual flags. When using the pushf(d) and popf(d) instructions it's an all-or-nothing proposition: You preserve all the flags when you push them; you restore all the flags when you pop them.

Like the pushad and popad instructions, you should really use the pushfd and popfd instructions to push the full 32-bit version of the EFLAGs register. Although the extra 16 bits you push and pop are essentially ignored when writing applications, you still want to keep the stack aligned by pushing and popping only double words.

3.9.6 Removing Data from the Stack Without Popping It

Once in a while you may discover that you've pushed data onto the stack that you no longer need. Although you could pop the data into an unused register or memory location, there is an easier way to remove unwanted data from the stack: Simply adjust the value in the ESP register to skip over the unwanted data on the stack.

Consider the following dilemma:

```
        push( eax );
        push( ebx );

        << Some code that winds up computing some values we want to keep
            into EAX and EBX >>
```

[15] For example, it is extremely rare for you to need to push and pop the ESP register with the PUSHAD / POPAD instruction sequence.

```
          if( Calculation_was_performed ) then

               // Whoops, we don't want to pop EAX and EBX!
               // What to do here?

          else

               // No calculation, so restore EAX, EBX.

               pop( ebx );
               pop( eax );

          endif;
```

Within the `then` section of the `if` statement, this code wants to remove the old values of EAX and EBX without otherwise affecting any registers or memory locations. How to do this?

Because the ESP register simply contains the memory address of the item on the top of the stack, we can remove the item from the top of stack by adding the size of that item to the ESP register. In the preceding example, we wanted to remove two double-word items from the top of stack. We can easily accomplish this by adding eight to the stack pointer (see Figures 3-17 and 3-18 for the details):

```
          push( eax );
          push( ebx );

          << Some code that winds up computing some values we want to keep
               into EAX and EBX >>

          if( Calculation_was_performed ) then

               add( 8, ESP );     // Remove unneeded EAX and EBX values from the
stack.

          else

               // No calculation, so restore EAX, EBX.

               pop( ebx );
               pop( eax );

          endif;
```

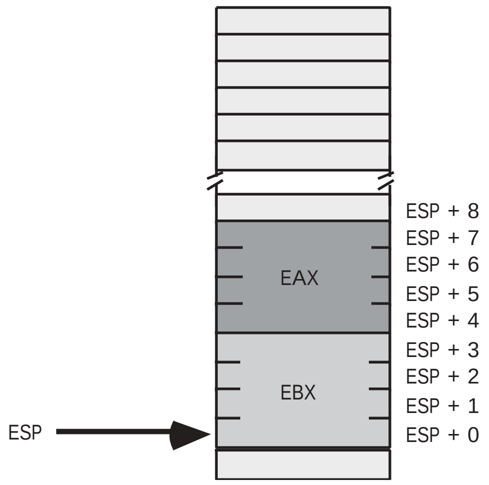


Figure 3-17: Removing Data from the Stack, Before ADD(8, ESP).

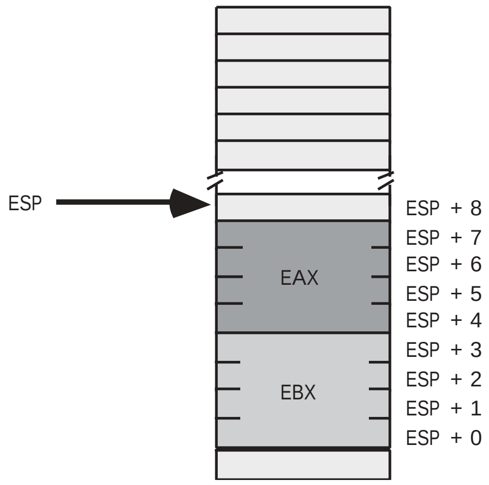


Figure 3-18: Removing Data from the Stack, After ADD(8, ESP).

Effectively, this code pops the data off the stack without moving it anywhere. Also note that this code is faster than two dummy pop instructions because it can remove any number of bytes from the stack with a single add instruction.

CAUTION *Remember to keep the stack aligned on a double-word boundary. Therefore, you should always add a constant that is an even multiple of four to ESP when removing data from the stack.*

3.9.7 Accessing Data You've Pushed on the Stack Without Popping It

Once in a while you will push data onto the stack and you will want to get a copy of that data's value, or perhaps you will want to change that data's value, without actually popping the data off the stack (that is, you wish to pop the data off the stack at a later time). The 80x86 "[reg$_{32}$ + offset]" addressing mode provides the mechanism for this.

Consider the stack after the execution of the following two instructions (see Figure 3-19):

```
        push( eax );
        push( ebx );
```

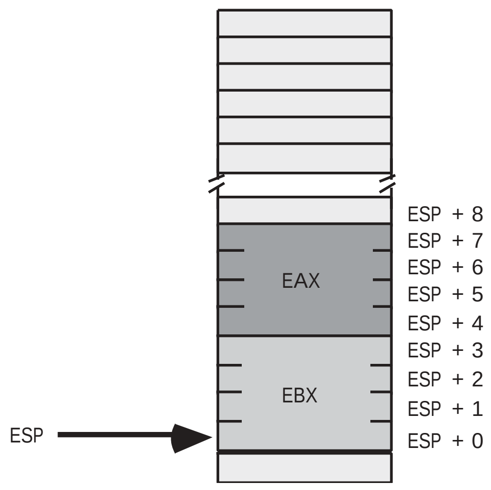


Figure 3-19: Stack After Pushing EAX and EBX.

If you wanted to access the original EBX value without removing it from the stack, you could cheat and pop the value and then immediately push it again. Suppose, however, that you wish to access EAX's old value, or some other value even farther up on the stack. Popping all the intermediate values and then pushing them back onto the stack is problematic at best and impossible at worst. However, as you will notice from Figure 3-19, each of the values pushed on the stack is at some offset from the ESP register in memory. Therefore, we can use the "[ESP + offset]" addressing mode to gain direct access to the value we are interested in. In the example above, you can reload EAX with its original value by using the single instruction

```
        mov( [esp+4], eax );
```

This code copies the four bytes starting at memory address ESP + 4 into the EAX register. This value just happens to be the previous value of EAX that was pushed onto the stack. You can use this same technique to access other data values you've pushed onto the stack.

CAUTION *Don't forget that the offsets of values from ESP into the stack change every time you push or pop data. Abusing this feature can create code that is hard to modify; if you use this feature throughout your code, it will make it difficult to push and pop other data items between the point you first push data onto the stack and the point you decide to access that data again using the "[ESP + offset]" memory addressing mode.*

The previous section pointed out how to remove data from the stack by adding a constant to the ESP register. That code example could probably be written more safely as:

```
push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
     into EAX and EBX >>

if( Calculation_was_performed ) then

     << Overwrite saved values on stack with new EAX/EBX values.
      (so the pops that follow won't change the values in EAX/EBX.) >>

     mov( eax, [esp+4] );
     mov( ebx, [esp] );

endif;
pop( ebx );
pop( eax );
```

In this code sequence, the calculated result was stored over the top of the values saved on the stack. Later on, when the program pops the values, it loads these calculated values into EAX and EBX.

3.10 Dynamic Memory Allocation and the Heap Segment

Although static and automatic variables are all simple programs may need, more sophisticated programs need the ability to allocate and deallocate storage dynamically (at runtime) under program control. In the C language, you would use the *malloc* and *free* functions for this purpose. C++ provides the *new* and *delete* operators. Pascal uses *new* and *dispose*. Other languages provide comparable facilities. These memory allocation routines share a couple of things in common: They let the programmer request how many bytes of storage to allocate, they return a *pointer* to the newly allocated storage, and they provide a facility for returning the storage to the system so the system can reuse it in a future allocation call. As you've probably guessed, HLA also provides a set of routines in the HLA Standard Library that handle memory allocation and deallocation.

The HLA Standard Library `malloc` and `free` routines handle the memory allocation and deallocation chores (respectively). The `malloc` routine uses the following calling sequence:

```
malloc( Number_of_Bytes_Requested );
```

The single parameter is a `double word` value specifying the number of bytes of storage you need. This procedure allocates storage in the *heap* segment in memory. The HLA `malloc` function locates an unused block of memory of the size you specify in the heap segment and marks the block as “in use” so that future calls to `malloc` will not allocate this same storage. After marking the block as “in use” the `malloc` routine returns a pointer to the first byte of this storage in the EAX register.

For many objects, you will know the number of bytes that you need in order to represent that object in memory. For example, if you wish to allocate storage for an `uns32` variable, you could use the following call to the `malloc` routine:

```
malloc( 4 );
```

Although you can specify a literal constant as this example suggests, it’s generally a poor idea to do so when allocating storage for a specific data type. Instead, use the HLA built-in *compile-time function*[16] `@size` to compute the size of some data type. The `@size` function uses the following syntax:

```
@size( variable_or_type_name )
```

The `@size` function returns an unsigned integer constant that is the size of its parameter in bytes. So you should rewrite the previous call to `malloc` as follows:

```
malloc( @size( uns32 ));
```

This call will properly allocate a sufficient amount of storage for the specified object, regardless of its type. While it is unlikely that the number of bytes required by an `uns32` object will ever change, this is not necessarily true for other data types, so you should always use `@size` rather than a literal constant in these calls.

Upon return from the `malloc` routine, the EAX register contains the address of the storage you have requested (see Figure 3-20).

[16] A compile-time function is one that HLA evaluates during the compilation of your program rather than at runtime.

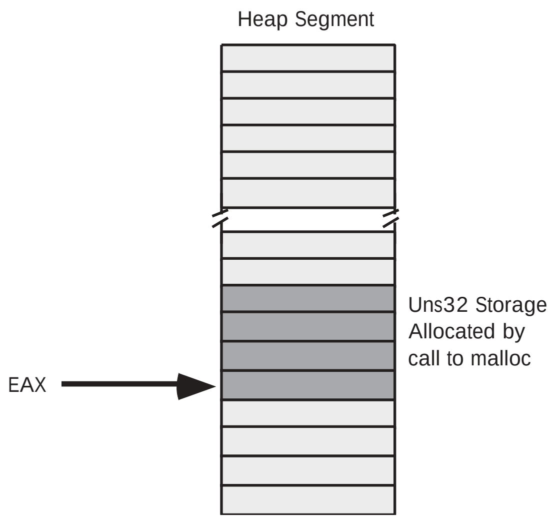


Figure 3-20: Call to malloc Returns a Pointer in the EAX Register.

To access the storage `malloc` allocates you must use a register indirect addressing mode. The following code sequence demonstrates how to assign the value 1234 to the `uns32` variable `malloc` creates:

```
malloc( @size( uns32 ));
mov( 1234, (type uns32 [eax]));
```

Note the use of the type coercion operator. This is necessary in this example because anonymous variables don't have a type associated with them, and the constant 1234 could be a `word` or `dword` value. The type coercion operator eliminates the ambiguity.

The `malloc` routine may not always succeed. If there isn't a single contiguous block of free memory in the heap segment that is large enough to satisfy the request, then the `malloc` routine will raise an `ex.MemoryAllocationFailure` exception. If you do not provide a `try..exception..endtry` handler to deal with this situation, a memory allocation failure will cause your program to stop. Because most programs do not allocate massive amounts of dynamic storage using `malloc`, this exception rarely occurs. However, you should never assume that the memory allocation will always occur without error.

When you are done using a value that `malloc` allocates on the heap, you can release the storage (that is, mark it as "no longer in use") by calling the `free` procedure. The `free` routine requires a single parameter that must be an address returned by a previous call to `malloc` (that you have not already freed). The following code fragment demonstrates the nature of the `malloc/free` pairing:

```
          malloc( @size( uns32));

               << use the storage pointed at by EAX >>
               << Note: this code must not modify EAX >>

          free( eax );
```

This code demonstrates a very important point: In order to properly free the storage that `malloc` allocates, you must preserve the value that `malloc` returns. There are several ways to do this if you need to use EAX for some other purpose; you could save the pointer value on the stack using `push` and `pop` instructions, or you could save EAX's value in a variable until you need to free it.

Storage you release is available for reuse by future calls to the `malloc` routine. The ability to allocate storage when you need it and then free the storage for other use when you are done with it improves the memory efficiency of your program. By deallocating storage once you are finished with it, your program can reuse that storage for other purposes, allowing your program to operate with less memory than it would if you statically allocated storage for the individual objects.

Several problems can occur when you use pointers. You should be aware of a few common errors that beginning programmers make when using dynamic storage allocation routines like `malloc` and `free`:

- Mistake #1: Continuing to refer to storage after you free it. Once you return storage to the system via the call to `free`, you should no longer access that storage. Doing so may cause a protection fault or, worse yet, corrupt other data in your program without indicating an error.
- Mistake #2: Calling `free` twice to release a single block of storage. Doing so may accidentally free some other storage that you did not intend to release or, worse yet, it may corrupt the system memory management tables.

The next chapter will discuss some additional problems you will typically encounter when dealing with dynamically allocated storage.

The examples thus far in this section have all allocated storage for a single unsigned 32-bit object. Obviously you can allocate storage for any data type using a call to `malloc` by simply specifying the size of that object as `malloc`'s parameter. It is also possible to allocate storage for a sequence of contiguous objects in memory when calling `malloc`. For example, the following code will allocate storage for a sequence of eight characters:

```
malloc( @size( char ) * 8 );
```

Note the use of the constant expression to compute the number of bytes required by an eight-character sequence. Because "@size(char)" always returns a constant value (one in this case), the compiler can compute the value of the expression "@size(char) * 8" without generating any extra machine instructions.

Calls to `malloc` always allocate multiple bytes of storage in contiguous memory locations. Hence the former call to `malloc` produces the sequence appearing in Figure 3-21.

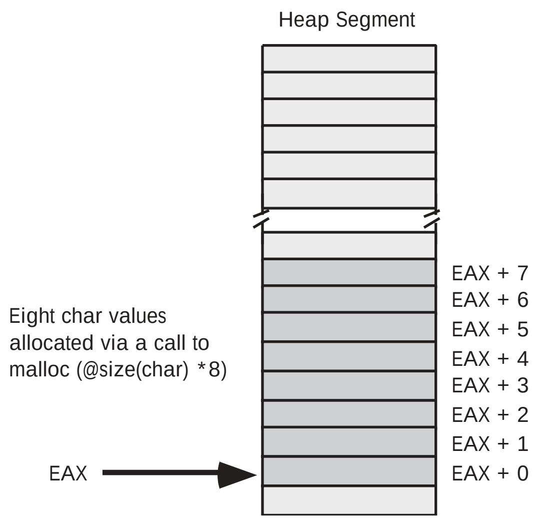


Figure 3-21: Allocating a Sequence of Eight-Character Objects Using Malloc.

To access these extra character values you use an offset from the base address (contained in EAX upon return from `malloc`). For example, "mov(ch, [eax + 2]);" stores the character found in CH into the third byte that `malloc` allocates. You can also use an addressing mode like "[EAX + EBX]" to step through each of the allocated objects under program control. For example, the following code will set all the characters in a block of 128 bytes to the NULL character (#0):

```
malloc( 128 );
for( mov( 0, ebx ); ebx < 128; add( 1, ebx ) ) do

     mov( 0, (type byte [eax+ebx]) );

endfor;
```

The next chapter discusses composite data structures (including arrays) and describes additional ways to deal with blocks of memory.

You should note that a call to `malloc` will actually allocate slightly more memory than you request. For one thing, memory allocation requests are generally of some minimum size (often a power of 2 between 4 and 16, though this is OS dependent). Furthermore, `malloc` requests also require a few bytes of overhead for each request (generally around 8 to 16 bytes) to keep track of allocated and free blocks. Therefore, it is not efficient to allocate a large number of small objects with individual calls to `malloc`. The overhead for each allocation may be greater than the storage you actually use. Typically, you'll use `malloc` to allocate storage for arrays or large records (structures) rather than small objects.

3.11 The INC and DEC Instructions

As the example in the last section indicates, indeed, as several examples up to this point have indicated, adding or subtracting one from a register or memory location is a very common operation. In fact, this operation is so common that Intel's engineers included a pair of instructions to perform these specific operations: the inc (increment) and dec (decrement) instructions.

The inc and dec instructions use the following syntax:

```
inc( mem/reg );
dec( mem/reg );
```

The single operand can be any legal 8-bit, 16-bit, or 32-bit register or memory operand. The inc instruction will add one to the specified operand; the dec instruction will subtract one from the specified operand.

These two instructions are slightly more efficient (they are smaller) than the corresponding add or sub instructions. There is also one slight difference between these two instructions and the corresponding add or sub instructions: they do not affect the carry flag.

As an example of the inc instruction, consider the example from the previous section, recoded to use inc rather than add:

```
        malloc( 128 );
        for( mov( 0, ebx ); ebx < 128; inc( ebx ) ) do

            mov( 0, (type byte [eax+ebx]) );

        endfor;
```

3.12 Obtaining the Address of a Memory Object

An earlier section of this chapter discusses how to use the address-of operator, "&", to take the address of a static variable.[17] Unfortunately, you cannot use the address-of operator to take the address of an automatic variable (one you declare in the var section); you cannot use it to compute the address of an anonymous variable, nor can you use this operator to take the address of a memory reference that uses an indexed or scaled indexed addressing mode (even if a static variable is part of the address expression). You may only use the address-of operator to take the address of a simple static object. Often, you will need to take the address of other memory objects as well; fortunately, the 80x86 provides the *load effective address* instruction, lea, to give you this capability.

The lea instruction uses the following syntax:

```
lea( reg32, Memory_operand );
```

[17] A static variable is one that you declare in the static, readonly, *or* storage of your program.

The first operand must be a 32-bit register; the second operand can be any legal memory reference using any valid memory addressing mode. This instruction will load the address of the specified memory location into the register. This instruction does not access or modify the value of the memory operand in any way.

Once you load the effective address of a memory location into a 32-bit general purpose register, you can use the register indirect, indexed, or scaled indexed addressing modes to access the data at the specified memory address. Consider the following code fragment:

```
static
    b:byte; @nostorage;
        byte 7, 0, 6, 1, 5, 2, 4, 3;
            .
            .
            .
    lea( ebx, b );
    for( mov( 0, ecx ); ecx < 8; inc( ecx )) do

        stdout.put( "[ebx+ecx]=", (type byte [ebx+ecx]), nl );

    endwhile;
```

This code steps through each of the eight bytes following the `b` label in the `static` section and prints their values. Note the use of the "[ebx+ecx]" addressing mode. The EBX register holds the base address of the list (that is, the address of the first item in the list), and ECX contains the byte index into the list.

3.13 For More Information

The CD-ROM that accompanies this book contains an older, 16-bit version of *The Art of Assembly Language Programming*. In that text you will find information about the 80x86's 16-bit addressing modes and segmentation. Please consult that documentation for more details.