A Tiny Guide to Programming in 32-bit x86 Assembly Language

by Adam Ferrari, *ferrari@virginia.edu*
(*with changes by* Alan Batson, *batson@virginia.edu*
and Mike Lack, *mnl3j@virginia.edu*)

1. Introduction

This small guide, in combination with the material covered in the class lectures on assembly language programming, should provide enough information to do the assembly language labs for this class. In this guide, we describe the basics of 32-bit x86 assembly language programming, covering a small but useful subset of the available instructions and assembler directives. However, real x86 programming is a large and extremely complex universe, much of which is beyond the useful scope of this class. For example, the vast majority of real (albeit older) x86 code running in the world was written using the 16-bit subset of the x86 instruction set. Using the 16-bit programming model can be quite complex—it has a segmented memory model, more restrictions on register usage, and so on. In this guide we'll restrict our attention to the more modern aspects of x86 programming, and delve into the instruction set only in enough detail to get a basic feel for programming x86 compatible chips at the hardware level.

2. Registers

Modern (i.e 386 and beyond) x86 processors have 8 32-bit general purpose registers, as depicted in Figure 1. The register names are mostly historical in nature. For example, EAX used to be called the "accumulator" since it was used by a number of arithmetic operations, and ECX was known as the "counter" since it was used to hold a loop index. Whereas most of the registers have lost their special purposes in the modern instruction set, by convention, two are reserved for special purposes—the stack pointer (ESP) and the base pointer (EBP).

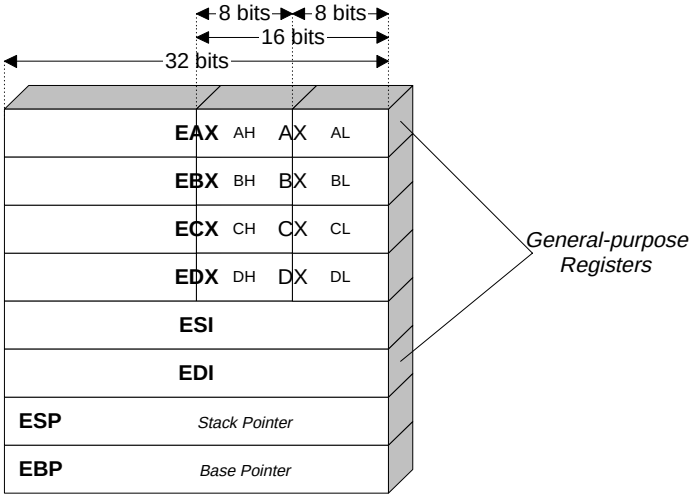

Figure 1. The x86 register set.

In some cases, namely EAX, EBX, ECX, and EDX, subsections of the registers may be used.

For example, the least significant 2 bytes of EAX can be treated as a 16-bit register called AX. The least significant byte of AX can be used as a single 8-bit register called AL, while the most significant byte of AX can be used as a single 8-bit register called AH. It is important to realize that these names refer to the same physical register. When a two-byte quantity is placed into DX, the update affects the value of EDX (in particular, the least significant 16 bits of EDX). These "sub-registers" are mainly hold-overs from older, 16-bit versions of the instruction set. However, they are sometimes convenient when dealing with data that are smaller than 32-bits (e.g. 1-byte ASCII characters).

When referring to registers in assembly language, the names are not case-sensitive. For example, the names EAX and eax refer to the same register.

3. Memory and Addressing Modes

3.1. Declaring Static Data Regions

You can declare static data regions (analogous to global variables) in x86 assembly using special assembler directives for this purpose. Data declarations should be preceded by the `.DATA` directive. Following this directive, the directives `DB`, `DW`, and `DD` can be used to declare one, two, and four byte data locations, respectively. Declared locations can be labeled with names for later reference - this is similar to declaring variables by name, but abides by some lower level rules. For example, locations declared in sequence will be located in memory next to one another. Some example declarations are depicted in Figure 2.

```
.DATA
var      DB 64    ; Declare a byte containing the value 64. Label the
                  ; memory location "var".
var2     DB ?     ; Declare an uninitialized byte labeled "var2".
         DB 10    ; Declare an unlabeled byte initialized to 10. This
                  ; byte will reside at the memory address var2+1.
X        DW ?     ; Declare an uninitialized two-byte word labeled "X".
Y        DD 3000  ; Declare 32 bits of memory starting at address "Y"
                  ; initialized to contain 3000.
Z        DD 1,2,3 ; Declare three 4-byte words of memory starting at
                  ; address "Z", and initialized to 1, 2, and 3,
                  ; respectively. E.g. 3 will be stored at address Z+8.
```

Figure 2. Declaring memory regions

The last example in Figure 2 illustrates the declaration of an array. Unlike in high level languages where arrays can have many dimensions and are accessed by indices, arrays in assembly language are simply a number of cells located contiguously in memory. Two other common methods used for declaring arrays of data are the `DUP` directive and the use of string literals. The `DUP` directive tells the assembler to duplicate an expression a given number of times. For example, the statement "`4 DUP(2)`" is equivalent to "`2, 2, 2, 2`". Some examples of declaring arrays are depicted in Figure 3.

```
bytes    DB 10  DUP(?)  ; Declare 10 uninitialized bytes starting at
                        ; the address "bytes".
arr      DD 100 DUP(0)  ; Declare 100 4 bytes words, all initialized to 0,
                        ; starting at memory location "arr".
str      DB 'hello',0   ; Declare 5 bytes starting at the address "str"
                        ; initialized to the ASCII character values for
                        ; the characters 'h', 'e', 'l', 'l', 'o', and
                        ; '\0'(NULL), respectively.
```

Figure 3. Declaring arrays in memory

3.2. Addressing Memory

Modern x86-compatible processors are capable of addressing up to 2^{32} bytes of memory; that is, memory addresses are 32-bits wide. For example, in Figure 2 and Figure 3, where we used labels to refer to memory regions, these labels are actually replaced by the assembler with 32-bit quantities that specify addresses in memory. In addition to supporting referring to memory regions by labels (i.e. constant values), the x86 provides a flexible scheme for computing and referring to memory addresses:

X86 Addressing Mode Rule - Up to two of the 32-bit registers and a 32-bit signed constant can be added together to compute a memory address. One of the registers can be optionally pre-multiplied by 2, 4, or 8.

To see this memory addressing rule in action, we'll look at some example `mov` instructions. As we'll see later in Section 4.1, the `mov` instruction moves data between registers and memory. This instruction has two operands—the first is the destination (where we're moving data *to*) and the second specifies the source (where we're getting the data *from*). Some examples of `mov` instructions using address computations that obey the above rule are:

- `mov eax, [ebx]` *; Move the 4 bytes in memory at the address contained in EBX into EAX*
- `mov [var], ebx` *; Move the contents of EBX into the 4 bytes at memory address "var"*
  ; (Note, "var" is a 32-bit constant).
- `mov eax, [esi-4]` *; Move 4 bytes at memory address ESI+(-4) into EAX*
- `mov [esi+eax], cl` *; Move the contents of CL into the byte at address ESI+EAX*
- `mov edx, [esi+4*ebx]` *; Move the 4 bytes of data at address ESI+4*EBX into EDX*

Some examples of incorrect address calculations include:

- `mov eax, [ebx-ecx]` *; Can only **add** register values*
- `mov [eax+esi+edi], ebx` *; At most **2** registers in address computation*

3.3. Size Directives

In general, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction in which it is referenced. For example, in all of the above instructions, the size of the memory regions could be inferred from the size of the register operand—when we were loading a 32-bit register, the assembler could infer that the region of memory we were referring to was 4 bytes wide. When we were storing the value of a one byte register to memory, the assembler could infer that we wanted the address to refer to a single byte in memory. However, in some cases the size of a referred-to memory region is ambiguous. Consider the following instruction:

```
mov [ebx], 2
```

Should this instruction move the value 2 into the single byte at address EBX? Perhaps it should move the 32-bit integer representation of 2 into the 4-bytes starting at address EBX. Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct. The size directives `BYTE PTR`, `WORD PTR`, and `DWORD PTR` serve this purpose. For example:

```
• mov BYTE PTR [ebx], 2    ; Move 2 into the single byte at memory location EBX
• mov WORD PTR [ebx], 2    ; Move the 16-bit integer representation of 2 into the 2 bytes starting at
                           ; address EBX
• mov DWORD PTR [ebx], 2   ; Move the 32-bit integer representation of 2 into the 4 bytes starting at
                           ; address EBX
```

4. Instructions

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control-flow. In this section, we will look at important examples of x86 instructions from each category. This section should not be considered an exhaustive list of x86 instructions, but rather a useful subset.

In this section, we will use the following notation:

- `<reg32>` - means any 32-bit register described in Section 2, for example, `ESI`.
  `<reg16>` - means any 16-bit register described in Section 2, for example, `BX`.
  `<reg8>` - means any 8-bit register described in Section 2, for example `AL`.
  `<reg>` - means any of the above.
- `<mem>` - will refer to a memory address, as described in Section 3, for example `[EAX]`, or `[var+4]`, or `DWORD PTR [EAX+EBX]`.
- `<con32>` - means any 32-bit constant.
  `<con16>` - means any 16-bit constant.
  `<con8>` - means any 8-bit constant.
  `<con>` - means any of the above sized constants.

4.1. Data Movement Instructions

Instruction: `mov`

Syntax:

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

Semantics: The `mov` instruction moves the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

Examples:

```
mov eax, ebx            ; transfer ebx to eax
mov BYTE PTR [var], 5   ; store the value 5 into the byte at
                        ; memory location "var"
```

Instruction: **`push`**

Syntax:
```
push <reg32>
push <mem>
push <con32>
```

Semantics: The `push` instruction places its operand onto the top of the hardware supported stack in memory. Specifically, `push` first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

Examples:
```
push eax    ; push the contents of eax onto the stack
push [var]  ; push the 4 bytes at address "var" onto the stack
```

Instruction: **`pop`**

Syntax:
```
pop <reg32>
pop <mem>
```

Semantics: The `pop` instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location). Specifically, `pop` first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4.

Examples:
```
pop edi     ; pop the top element of the stack into EDI.
pop [ebx]   ; pop the top element of the stack into memory at the
            ; four bytes starting at location EBX.
```

Instruction: **`lea`**

Syntax:
```
lea <reg32>,<mem>
```

Semantics: The `lea` instruction places the *address* specified by its second operand into the register specified by its first operand. Note, the *contents* of the memory location are not loaded—only the effective address is computed and placed into the register. This is useful for obtaining a "pointer" into a memory region.

Examples:
```
lea eax, [var]          ; the address of "var" is places in EAX.
lea edi, [ebx+4*esi]    ; the quantity EBX+4*ESI is placed in EDI.
```

4.2. Arithmetic and Logic Instructions

Instruction: **`add, sub`**

Syntax:
```
add <reg>,<reg>     sub <reg>,<reg>
add <reg>,<mem>     sub <reg>,<mem>
add <mem>,<reg>     sub <mem>,<reg>
add <reg>,<con>     sub <reg>,<con>
add <mem>,<con>     sub <mem>,<con>
```

Semantics: The `add` instruction adds together its two operands, storing the result in its first operand. Similarly, the `sub` instruction subtracts its second operand from its first. Note, whereas both operands may be registers, at most one operand may be a memory location.

Examples:
```
add eax, 10       ; add 10 to the contents of EAX.
sub [var], esi    ; subtract the contents of ESI from the 32-bit
                  ; integer stored at memory location "var".
```

```
add BYTE PTR [var], 10  ; add 10 to the single byte stored
                        ;  at memory address "var".
```

Instruction: **`inc, dec`**

Syntax:
```
inc <reg>          dec <reg>
inc <mem>          dec <mem>
```

Semantics: The `inc` instruction increments the contents of its operand by one, and similarly `dec` decrements the contents of its operand by one.

Examples:
```
dec eax                 ; subtract one from the contents of EAX.
inc DWORD PTR [var]     ; add one to the 32-bit integer stored at
                        ; memory location "var".
```

Instruction: **`imul`**

Syntax:
```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

Semantics: The `imul` instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).
The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. first) operand must be a register.
The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

Examples:
```
imul eax, [var]   ; multiply the contents of EAX by the 32-bit
                  ; contents of the memory location "var". Store
                  ; the result in EAX.
imul esi, edi, 25 ; multiply the contents of EDI by 25. Store the
                  ; result in ESI.
```

Instruction: **`idiv`**

Syntax:
```
idiv <reg32>
idiv <mem>
```

Semantics: The `idiv` instruction is used to divide the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX. This instruction must be used with care. Before executing the instruction, the appropriate value to be divided must be placed into EDX and EAX. Clearly, this value is overwritten when the `idiv` instruction is executed.

Examples:
```
idiv  ebx   ; divide the contents of EDX:EAX by the contents of
            ; EBX. Place the quotient in EAX and the remainder
            ; in EDX.
idiv DWORD PTR [var] ; same as above, but divide by the 32-bit
                     ; value stored at memory location "var".
```

Instruction: `and, or, xor`

Syntax:
```
and <reg>,<reg>  or <reg>,<reg>  xor <reg>,<reg>
and <reg>,<mem>  or <reg>,<mem>  xor <reg>,<mem>
and <mem>,<reg>  or <mem>,<reg>  xor <mem>,<reg>
and <reg>,<con>  or <reg>,<con>  xor <reg>,<con>
and <mem>,<con>  or <mem>,<con>  xor <mem>,<con>
```

Semantics: These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Examples:
```
and   eax, 0fH    ; clear all but the last 4 bits of EAX.
xor   edx, edx    ; set the contents of EDX to zero.
```

Instruction: `not`

Syntax:
```
not <reg>
not <mem>
```

Semantics: Performs the logical negation of the operand contents (i.e. flips all bit values).

Examples:
```
not BYTE PTR [var]; negate all bits in the byte at the memory
                  ; location "var".
```

Instruction: `neg`

Syntax:
```
neg <reg>
neg <mem>
```

Semantics: Performs the arithmetic (i.e. two's complement) negation of the operand contents.

Examples:
```
neg eax     ; negate the contents of EAX.
```

Instruction: `shl, shr`

Syntax:
```
shl <reg>,<con8>    shr <reg>,<con8>
shl <mem>,<con8>    shr <mem>,<con8>
shl <reg>,cl        shr <reg>,cl
shl <mem>,cl        shr <mem>,cl
```

Semantics: These instructions shift the bits in their first operand's contents left and right (`shl` and `shr`, respectively), padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater then 31 are performed modulo 32.

4.3. Control Flow Instructions

In this section, we will refer to labeled locations in the program text as `<label>`. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example, consider the code fragment in Figure 4. The second instruction in this code fragment is labeled "`begin`". Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name "`begin`" instead of having to

refer to the memory address as an integer.

```
          mov esi, [ebp+8]
begin:    xor ecx, ecx
          mov eax, [esi]
```

Figure 4. A labeled code location

Instruction: **`jmp`**

Syntax: `jmp <label>`

Semantics: Transfers program control flow to the instruction at the memory location indicated by the operand.

Examples:
```
jmp begin
```

Instruction: **`j`*`CC`***

Syntax:

`je <label>`	- Jump when equal
`jne <label>`	- Jump when not equal
`jz <label>`	- Jump when last result was zero
`jg <label>`	- Jump when greater than
`jge <label>`	- Jump when greater than or equal to
`jl <label>`	- Jump when less than
`jle <label>`	- Jump when less than or equal to

Semantics: These instructions are conditional jumps that are based on the status of a set of **condition codes** that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the `jz` instruction performs a jump to the specified operand label if the result of the last arithmetic operation (e.g. `add`, `sub`, etc.) was zero. Otherwise, control proceeds to the next instruction in sequence after the `jz`. These conditional jumps are the underlying support needed to implement high-level language features such as "if" statements and loops (e.g. "while" and "for").

A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, `cmp` (see below). For example, conditional branches such as `jle` and `jne` are based on first performing a `cmp` operation on the desired operands.

Examples:
```
cmp eax, ebx      ; if the contents of eax are less than or equal
jle done          ; to the contents of EBX, jump to the code
                  ; location labeled "done".
```

Instruction: **`cmp`**

Syntax: `cmp <reg>,<reg>`

```
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
cmp <mem>,<con>
```

Semantics: Compares the two specified operands, setting the condition codes in the machine status word appropriately. In fact, this instruction is equivalent to the `sub` instruction, except the result of the subtraction is discarded.

Examples:
```
cmp DWORD PTR [var], 10 ; if the 4 bytes stored at memory location
jeq loop                ; “var” equal the 4-byte integer value 10,
                        ; then jump to the code location labeled
                        ; loop
```

Instruction: `call`

Syntax: `call <label>`

Semantics: This instruction implements a subroutine call that operates in cooperation with the subroutine return instruction, `ret`, described below. This instruction first pushes the current code location onto the hardware supported stack in memory (see the `push` instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. The added value of this instruction (as compared to the simple `jmp` instruction) is that it saves the location to return to when the subroutine completes.

Examples:
```
call my_subroutine
```

Instruction: `ret`

Syntax: `ret`

Semantics: In cooperation with the `call` instruction, the `ret` instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the `pop` instruction for details). It then performs an unconditional jump to the retrieved code location.

Examples:
```
ret
```

5. Basic Program Structure

Given the above repertoire of instructions, you are in a position to examine the basic skeletal structure of an assembly language subroutine suitable for linking into C++ code. Unlike C++, which is often used for the development of complete software systems, assembly language is most often used in cooperation with other languages such as Fortran, C, and C++. Commonly, most of a project is implemented in the more convenient high-level language, and assembly language is used sparingly to implement extremely low-level hardware interfaces or performance-critical “inner loops.” Thus, in addition to understanding how to program in assembly language, it is equally important to understand how to link assembly language code into high-level language programs.

Before examining the linkage conventions, we must first examine the basic structure of an assembly language file. To do this, we can compare a very simple assembly language file to an equivalent C++ file. In Figure 5 we see two files, one in C++, the other in x86 assembly. Each file

includes a function (albeit an ugly one) to return the integer value 2.

```
int var = 2;

extern "C" int returnTwo();
int returnTwo()
{
   return var;
}
```

```
.486
MODEL FLAT

.DATA
var   DD    2

.CODE
PUBLIC _returnTwo
_returnTwo PROC
   mov   eax, [var]
   ret
ENDP _returnTwo
END
```

Figure 5. Equivalent functions that return the value 2, implemented in C++ (left) and x86 (right).

The top of the assembly file contains two directives that indicate the instruction set and memory model we will use for all work in this class (note, there are other possibilities—one might use only the older 80286 instruction set for wider compatibility, for example).

Next, where in the C++ file we find the declaration of the global variable "`var`", in the assembly file we find the use of the `.DATA` and `DD` directives (described in Section 3.1) to reserve and initialize a 4-byte (i.e. integer-sized) memory region labeled "`var`".

Next in each file, we find the declaration of the function named `returnTwo`. In the C++ file we have declared the function to be `extern "C"`. This declaration indicates that the C++ compiler should use C naming conventions when labeling the function `returnTwo` in the resulting object file that it produces. In fact, this naming convention means that the function `returnTwo` should map to the label `_returnTwo` in the object code. In the assembly code, we have labeled the beginning of the subroutine `_returnTwo` using the `PROC` directive, and have declared the label `_returnTwo` to be public. Again, the result of these actions will be that the subroutine will map to the symbol `_returnTwo` in the object code that the assembler generates.

The function bodies are straight-forward. As we will see in more detail in Section 6, return values for functions are placed into EAX by convention, hence the instruction to move the contents of "`var`" into EAX in the assembly code.

Given these equivalent function definitions, use of either version of the function is the same. A sample call to the function `returnTwo` is depicted in Figure 6. This C++ code could be linked to either definition of the function and would produce the same results (note, we could not link to both definitions, or the linker would produce a "multiply defined symbol" error. The mechanics of

linking programs in the Borland environment available in the lab are covered in Section 7.

```
#include <iostream.h>
extern “C” int returnTwo();
int main()
{
   if(returnTwo()!=2) {
      cerr << “Does not compute!\n”;
   }
}
```

Figure 6. Calling `returnTwo` from C++.

6. Subroutine Calling Convention

6.1. What is a Calling Convention?

In Section 5 we saw a simple example of a subroutine defined in x86 assembly language. In fact, this subroutine was quite simple—it did not modify any registers except EAX (which was needed to return the result), and it did not call any other subroutines. In practice, such simple function definitions are rarely useful. When more complex subroutines are combined in a single program, a number of complicating issues arise. For example, how are parameters passed to a subroutine? Can subroutines overwrite the values in a register, or does the caller expect the register contents to be preserved? Where should local variables in a subroutine be stored? How should results be returned from functions?

To allow separate programmers to share code and develop libraries for use by many programs, and to simplify the use of subroutines in general, programmers typically adopt a common *calling convention*. The calling convention is simply a set of rules that answers the above questions without ambiguity to simplify the definition and use of subroutines. For example, given a set of calling convention rules, a programmer need not examine the definition of a subroutine to determine how parameters should be passed to that subroutine. Furthermore, given a set of calling convention rules, high-level language compilers can be made to follow the rules, thus allowing hand-coded assembly language routines and high-level language routines to call one another.

In practice, even for a single processor instruction set, many calling conventions are possible. In this class we will examine and use one of the most important conventions: the C language calling convention. Understanding this convention will allow you to write assembly language subroutines that are safely callable from C and C++ code, and will also enable you to call C library functions from your assembly language code.

6.2. The C Calling Convention

The C calling convention is based heavily on the use of the hardware-supported stack. To understand the C calling convention, you should first make sure that you fully understand the `push`, `pop`, `call`, and `ret` instructions—these will be the basis for most of the rules. In this calling convention, subroutine parameters are passed on the stack. Registers are saved on the stack, and local variables used by subroutines are placed in memory on the stack. In fact, this stack-centric implementation of subroutines is not unique to the C language or the x86 architecture. The vast majority of high-level procedural languages implemented on most processors have used similar calling convention.

The calling convention is broken into two sets of rules. The first set of rules is employed by the caller of the subroutine, and the second set of rules is observed by the writer of the subroutine (the "callee"). It should be emphasized that mistakes in the observance of these rules quickly result in fatal program errors; thus meticulous care should be used when implementing the call convention in your own subroutines.

6.2.1. The Caller's Rules

The caller should adhere to the following rules when invoking a subroutine:

1. Before calling a subroutine, the caller should save the contents of certain registers that are designated *caller-saved*. The caller-saved registers are EBX, ECX, EDX. If you want the contents of these registers to be preserved across the subroutine call, push them onto the stack.
1. To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in inverted order (i.e. last parameter first)—since the stack grows down, the first parameter will be stored at the lowest address (this inversion of parameters was historically used to allow functions to be passed a variable number of parameters).
2. To call the subroutine, use the `call` instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.
3. After the subroutine returns, (i.e. immediately following the `call` instruction) the caller must remove the parameters from stack. This restores the stack to its state before the call was performed.
4. The caller can expect to find the return value of the subroutine in the register EAX.
5. The caller restores the contents of caller-saved registers (EBX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

6.2.2. The Callee's Rules

The definition of the subroutine should adhere to the following rules:

1. At the beginning of the subroutine, the function should push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

   ```
   push  ebp
   mov   ebp, esp
   ```

   The reason for this initial action is the maintenance of the *base pointer*, EBP. The base pointer is used by convention as a point of reference for finding parameters and local variables on the stack. Essentially, when any subroutine is executing, the base pointer is a "snapshot" of the stack pointer value from when the subroutine started executing. Parameters and local variables will always be located at known, constant offsets away from the base pointer value. We push the old base pointer value at the beginning of the subroutine so that we can later restore the appropriate base pointer value for the caller when the subroutine returns. Remember, the caller isn't expecting the subroutine to change the value of the base pointer. We then move the stack pointer into EBP to obtain our point of reference for accessing parameters and local variables.
2. Next, allocate local variables by making space on the stack. Recall, the stack grows down, so to make space on the top of the stack, the stack pointer should be decremented. The amount by which the stack pointer is decremented depends on the number of local variables needed. For

example, if 3 local integers (4 bytes each) were required, the stack pointer would need to be decremented by 12 to make space for these local variables. I.e:

```
sub esp, 12
```

As with parameters, local variables will be located at known offsets from the base pointer.

3. Next, the values of any registers that are designated *callee-saved* that will be used by the function must be saved. To save registers, push them onto the stack. The callee-saved registers are EDI and ESI (ESP and EBP will also be preserved by the call convention, but need not be pushed on the stack during this step).

   After these three actions are performed, the actual operation of the subroutine may proceed. When the subroutine is ready to return, the call convention rules continue:

4. When the function is done, the return value for the function should be placed in EAX if it is not already there.
5. The function must restore the old values of any callee-saved registers (EDI and ESI) that were modified. The register contents are restored by popping them from the stack. Note, the registers should be popped in the inverse order that they were pushed.
6. Next, we deallocate local variables. The obvious way to do this might be to add the appropriate value to the stack pointer (since the space was allocated by subtracting the needed amount from the stack pointer). In practice, a less error-prone way to deallocate the variables is to move the value in the base pointer into the stack pointer, i.e.:

   ```
   mov esp, ebp
   ```

   This trick works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.
7. Immediately before returning, we must restore the caller's base pointer value by popping EBP off the stack. Remember, the first thing we did on entry to the subroutine was to push the base pointer to save its old value.
8. Finally, we return to the caller by executing a `ret` instruction. This instruction will find and remove the appropriate return address from the stack.

It might be noted that the callee's rules fall cleanly into two halves that are basically mirror images of one another. The first half of the rules apply to the beginning of the function, and are therefor commonly said to define the *prologue* to the function. The latter half of the rules apply to the end of the function, and are thus commonly said to define the *epilogue* of the function.

6.2.3. Call Convention Example

The above rules may seem somewhat abstract on first examination. In practice, the rules become simple to use when they are well understood and familiar. To start the process of better understanding the call convention, we now examine a simple example of a subroutine call and a subroutine definition.

In Figure 7 a sample function call is depicted. Note how the caller pushes the parameters onto

the stack in inverted order before the call. The `call` instruction is used to jump to the beginning

```
; Want to call a function “myFunc” that takes three
; integer parameters. First parameter is in EAX.
; Second parameter is the constant 123. Third
; parameter is in memory location “var”

   push  [var]  ; Push last parameter first
   push  123
   push  eax    ; Push first parameter last

   call _myFunc ; Call the function (assume C naming)

; On return, clean up the stack. We have 12 bytes
; (3 parameters * 4 bytes each) on the stack, and the
; stack grows down. Thus, to get rid of the parameters,
; we can simply add 12 to the stack pointer

   add   esp, 12

; The result produced by “myFunc” is now available for
; use in the register EAX. No other register values
; have changed
```

Figure 7. Example function call, caller’s rules obeyed

of the subroutine in anticipation of the fact that the subroutine will use the `ret` instruction to return when the subroutine completes. When the subroutine returns, the parameters must be removed from the stack. A simple way to do this is to add the appropriate amount to the stack pointer (since the stack grows down). Finally, the result is available in EAX.

Relative to the caller’s rules, the callee’s rules are somewhat more complex. An example subroutine implementation that obeys the callee’s rules is depicted in Figure 8. The subroutine prologue performs the standard actions of saving a snapshot of the stack pointer in EBP (the base pointer), allocating local variables by decrementing the stack pointer, and saving register values on the stack.

In the body of the subroutine we can now more clearly see the use of the base pointer illustrated. Both parameters and local variables are located at constant offsets from the base pointer for the duration of the subroutines execution. In particular, we notice that since parameters were placed onto the stack before the subroutine was called, they are always located below the base pointer (i.e. at higher addresses) on the stack. The first parameter to the subroutine can always be found at memory location [EBP+8], the second at [EBP+12], the third at [EBP+16], and so on. Similarly, since local variables are allocated after the base pointer is set, they always reside above the base pointer (i.e. at lower addresses) on the stack. In particular, the first local variable is always located at [EBP-4], the second at [EBP-8], and so on. Understanding this conventional use of the base pointer allows us to quickly identify the use of local variables and parameters within a function body.

The function epilogue, as expected, is basically a mirror image of the function prologue. The caller’s register values are recovered from the stack, the local variables are deallocated by reset-

ting the stack pointer, the caller's base pointer value is recovered, and the ret instruction is used to return to the appropriate code location in the caller.

```
.486
MODEL FLAT
.CODE
PUBLIC _myFunc
_myFunc PROC
   ; *** Standard subroutine prologue ***
   push ebp     ; Save the old base pointer value.
   mov ebp, esp ; Set the new base pointer value.
   sub esp, 4   ; Make room for one 4-byte local variable.
   push edi     ; Save the values of registers that the function
   push esi     ; will modify. This function uses EDI and ESI.
                ; (no need to save EAX, EBP, or ESP)

   ; *** Subroutine Body ***
   mov eax, [ebp+8] ; Put value of parameter 1 into EAX
   mov esi, [ebp+12]; Put value of parameter 2 into ESI
   mov edi, [ebp+16]; Put value of parameter 3 into EDI

   mov [ebp-4], edi ; Put EDI into the local variable
   add [ebp-4], esi ; Add ESI into the local variable
   add eax, [ebp-4] ; Add the contents of the local variable
                    ; into EAX (final result)

   ; *** Standard subroutine epilogue ***
   pop esi      ; Recover register values
   pop edi
   mov esp, ebp ; Deallocate local variables
   pop ebp      ; Restore the caller's base pointer value
   ret
ENDP _myFunc
END
```

Figure 8. Example function definition, callee's rules obeyed

A good way to visualize the operation of the calling convention is to draw the contents of the nearby region of the stack during subroutine execution. Figure 9 depicts the contents of the stack during the execution of the body of `myFunc` (depicted in Figure 8). Notice, lower addresses are depicted lower in the figure, and thus the "top" of the stack is the bottom-most cell. This corresponds visually to the intuitive statement that the x86 hardware stack "grows down." The cells depicted in the stack are 32-bit wide memory locations, thus the memory addresses of the cells are 4 bytes apart. From this picture we see clearly why the first parameter resides at an offset of 8 bytes from the base pointer. Above the parameters on the stack (and below the base pointer), the `call` instruction placed the return address, thus leading to an extra 4 bytes of offset from the base pointer to the first parameter.

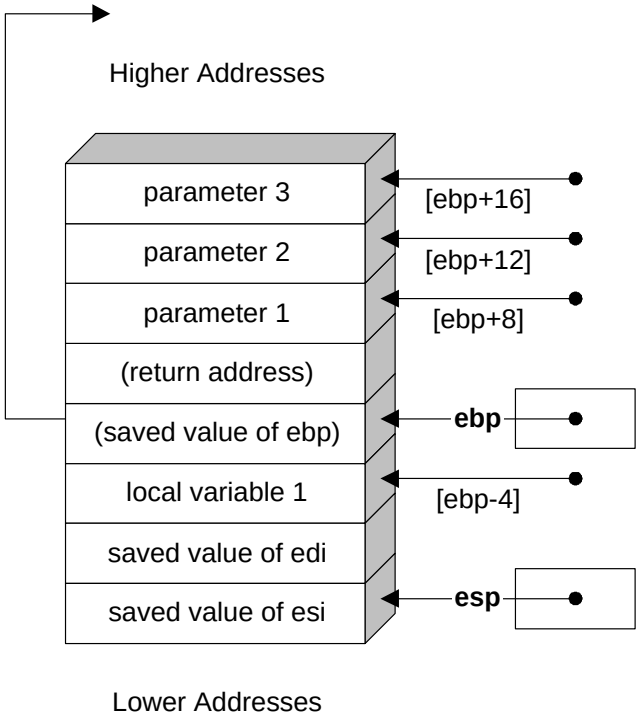

Figure 9. A picture of the stack in memory during the execution of the body of `myFunc`.

7. Using the Borland Environment

Thus far in this guide we have only covered programming concepts, not actual programming environment mechanics such as how to use the Borland x86 assembly language tools available in the lab. In this section we describe the basic steps involved in assembling, linking, and debugging combined C++/assembly language programs in the Borland environment. This quick guide assumes familiarity with the Borland C++ Integrated Development Environment (IDE), which you have used in other computer science classes.

7.1. Assembly and Linking

You can write assembly language files in the same way that you compose new C++ files in the Borland environment. From the **File** menu, select **New**, then select **Text Edit**. When you save your file, use the extension ".asm" instead of the familiar ".cpp". This will let the Borland environment know that it's dealing with assembly code instead of C++.

You can add assembly language source files to a project in the same way that you add C++ files to a project. In the Project window (select **Project** from the **View** menu to display the Project Window), right-click on the appropriate target and select **Add Node**. Your assembly file will not appear on the default list of files (only .cpp files appear). Select ".asm" in the **Files of Type** drag-down menu, then select your .asm file.

NOTE: Do not name your assembly language file with the same name as your project file (e.g. sort.ide, sort.cpp, sort.asm) as this will confuse the linker. It would be better to name your assem-

bly language file in a way that suggests what it does (e.g. sort.ide, sort.cpp, bubblesort.asm).

Before you can compile and link your program, you'll need to set some of the target options for the project. Right-click on the target in your Project window and select **Target Expert**. In the Target Expert dialogue window, set **Target Type** to Application (.exe), set **Platform** to Win32, set **Target Model** to Console, un-select all Libraries, Frameworks, and Controls, and select Static linking. The multithreaded option should be de-selected by default—if not, turn this option off.

You are now ready to compile your combined C++/assembly program. From the Project window, select **Build all**. Errors will appear as usual in the Message window. Once all compile and link errors are resolved, you can try running your application from the command prompt (i.e. from outside the Borland IDE). If your program does not run as expected, you can debug it in the Borland IDE using the interactive debugger.

7.2. Debugging

The debugging process is for the most part similar to your experience using the debugger on C++ code. However, when debugging assembly code, we are generally interested in examining lower-level program states. In addition to looking at variable (i.e. memory) contents, we also wish to see the contents of registers, the current machine instruction, the state of the stack, and so on.

To debug your assembly language function, set a breakpoint at the call site within your C++ code. Once the program suspends execution and is about to step into your assembly language code, select **CPU** from the **View** menu. Notice that the usual **statement step over** and **statement step into** buttons have been replaced with **instruction step over** and **instruction step into** buttons. Within the CPU window you can examine the register contents, the stack contents, memory, and your assembly language instructions. Now you can use the **instruction step into** button (or the F7 key) to step through your instructions one at a time. After each instruction has executed, you can examine the complete state of the CPU and verify that your code is executing as you expected, or find out why it isn't.