Chapter 1

The 32 bit x86 C Calling Convention

...

This chapter was derived from a document written by Adam Ferrari and later updated by Alan Batson, Mike Lack and Anita Jones

1.1 What is a Calling Convention?

At the end of the previous chapter, 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.

1.2 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.

1.3 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 EAX, ECX, EDX. If you want the contents of these registers to be preserved across the subroutine call, push them onto the stack.
2. 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).
3. 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.
4. 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.
5. The caller can expect to find the return value of the subroutine in the register EAX.
6. The caller restores the contents of caller-saved registers (EAX, ECX, EDX) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

1.4 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:

Listing 1.1: x86 callee code, part 1

```
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:

Listing 1.2: x86 callee code, part 2

```
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 EBX, 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 (EBX, 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.:

Listing 1.3: x86 callee code, part 3

```
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

Listing 1.4: Example function call, caller's rules obeyed

```
; 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
```

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.

1.5 Calling 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 Listing 1.4 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 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 Listing 1.5. 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

Listing 1.5: Example function definition, callee's rules obeyed

```
global myFunc

section .text

myFunc:
        ; *** 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
```

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 resetting 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.

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 1.1 depicts the contents of the stack during the

execution of the body of myFunc (myFunc is depicted in Listing 1.5). 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.

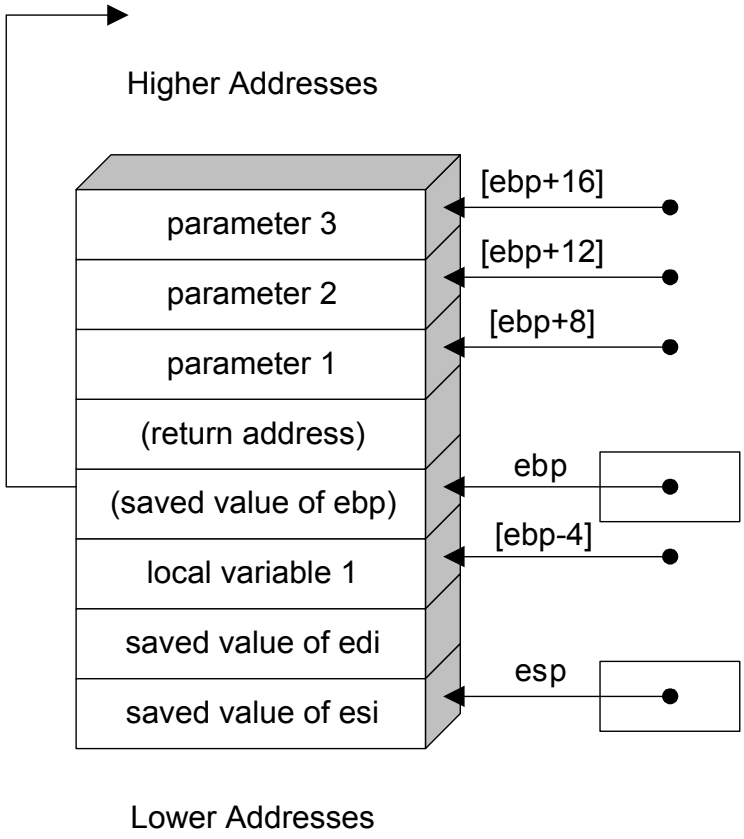

Figure 1.1: A picture of the stack in memory during the execution of the body of myFunc

The assembly code for `myFunc()` was shown above in Listing 1.5. The C++ code to call that subroutine is shown in Listing 1.6.

Listing 1.6: Example C++ code to invoke a 3-parameter x86 subroutine

```
#include <iostream>
using namespace std;

extern "C" int myFunc(int,int,int);

int main() {
  int x = 3;
  cout << "myFunc() returned: "
       << myFunc(x,5,10) << endl;
  return 0;
}
```