cdecl versus *stdcall* on x86 architecture: But really a discussion of how function calls use registers and stack

Brian R. Hall
Department of Software Technology
Champlain College
Burlington, VT, USA
hall@champlain.edu

1 Introduction

The *cdecl* and *stdcall* calling conventions (protocols) define the process (management of the stack, registers, etc.) of procedure calls. *cdecl* is typically the default in compilers such as Visual Studio's C compiler and GCC. WinAPI requires the use of *stdcall*, and when using an assembler such as MASM the protocol can be set via the .MODEL directive. The two protocols are similar, but differ in the way parameters are removed from the stack upon procedure completion. This document explains and illustrates the difference.

First, understand that when a function (procedure in assembly) is called, the function is allocated a chunk of memory on the stack at run-time, which is called a *stack frame*. A stack frame is used to store the function's variables and parameters. It is worth saying that every function call has its own stack frame, including main, and in the case of recursion every call has its own stack frame.

The issue at hand is what happens when the function is done running and we return whence the call occurred. The parameters pushed on the stack as part of the call's stack frame need to be cleaned up. This is where *calling conventions* come into play, which determine the way clean up happens and other details such as the order in which parameters are passed. Two common calling protocols are *cdecl* and *stdcall*.

Figure 1: Cleanup Responsibilities

cdecl	***stdcall***
caller	callee
calling function	called function

2 Call and Return Instructions

Two of the assembly instructions at play in this scenario are *call* and *ret*.

- CALL: pushes location (address) of next instruction to the stack and transfers to new destination.
- RET: pops from top of stack, which hopefully is the location (address) of next instruction that was pushed with CALL. Can be written two ways:
  (1) ret
  (2) ret *count*
  count being added to the %esp register after completion of the return.

3 *cdecl*

The *cdecl* protocol is the most common across system platforms because it is based on the C language. C supports variadic functions (variable argument lists), which means that the caller must clean up the stack after the function call because the callee has no way of knowing how many arguments it has received and thus cannot clean up the stack. This means cleanup code is written every time the function is called.

4 *stdcall*

The *stdcall* protocol, based on the *pascal* calling convention, is mainly used by the Windows API. *stdcall* expects that argument lists are fixed which means the called function can know how many parameters have been sent, and thus can clean up the stack. The advantage of this protocol is that the cleanup code is written once, in the called function. The result is slightly smaller code that is potentially slightly faster.

As an aside not specifically discussed in this document, you can define functions as *stdcall* beyond the realm of the Windows API (e.g., GCC and LLVM/clang) by defining *stdcall* in the following way:
#define stdcall __attribute__((stdcall)) or **int __attribute__((__stdcall__)) func( )**

5 *cdecl* Example

The following code example illustrates the *cdecl* calling convention. The following assembly code is written in Xcode 6.1.1, which uses modified LLVM/clang and expects AT&T syntax. The program illustrates a function that adds two numbers.

Figure 2: *cdecl* Code Example

```
.data
num1: .long 2
num2: .long 4

.text
.globl _main, _sum
_main:

mov $10, %eax
dec %eax
dec %eax
mov $5, %ebx

push num2
push num1
call _sum
add $8, %esp

add %ebx, %eax
dec %eax
dec %eax
ret


_sum:
push %ebp
mov %esp, %ebp
push %ebx

mov 8(%ebp), %ebx
mov 12(%ebp), %eax
add %ebx, %eax
pop %ebx
pop %ebp
ret

.end
```

Let us examine what this code does.

◇ This example assumes you understand the following:

  – Stack grows down, with addresses descending.
  – Every memory slot holds 4 bytes (32 bits), with each byte represented by 2 hexadecimal digits. This should make sense as every four bits can be represented by 1 hex digit.
  – Values are stored in memory in *little endian* form. This means the least significant byte is stored at the start of the address and the most significant byte is stored at the end. Example: if we wanted to store the hexadecimal value 9BFB3701 in memory it would be stored as 01 37 FB 9B.
  – The letter "l" added to the instructions in the example such as "mov" becoming "movl" is just the assembler re-writing the instructions to explicitly indicate the values are of type *long*.
  – Pass-by-value versus pass-by-reference. This example is based on pass-by-value.
  – In AT&T syntax registers are prefixed with %

◇ This example explains the under-the-hood assembly implementation of a C-style function call. If the function call illustrated in this example were written in C++, the function parts might look like this:

  – Variable declarations: **int num1 = 2, num2 = 4;**
  – Prototype: **int sum(int num1, int num2);**
  – Call: **int answer = sum(num1, num2);**
  – Implementation: **int sum(int num1, int num2){ return num1 + num2; }**

◇ First is the .data segment (lines 1-3), which defines two variables num1 and num2. Both are of type long (4 bytes) with num1 set to the integer value 2 and num2 set to the integer value 4.

◇ Next is the .text segment, which contains the executable instructions for the program. In .text, two global procedures are declared (line 6): _main (lines 7-22) and _sum (lines 25-35).

◇ _main is the entry point and where execution begins.

◇ Lines 9-12 are simply meant to simulate other activities going on in the program prior to a procedure call.

  1. Line 9: moves the value 10 to the %eax register
  2. Line 10: decrements %eax by 1
  3. Line 11: decrements %eax by 1
  4. Line 12: moves the value 5 to the %ebx register
  5. %eax holds the value 8 and %ebx holds the value 5

◇ *%eax* and *%ebx* are general purpose registers that will be used in the in the _sum procedure (clobbered), so at some point we will need to save the values the registers contain if we want to have them available after the procedure ends. In this example, we are going to care about saving %ebx, but do not care about saving %eax.

◇ Next, it is time to prepare for the procedure. *cdecl* pushes parameters on the stack in reverse order. So, given a high-level function call such as **sum(num1, num2);** num2 will be pushed first, then num1 (lines 14-15).

◇ Two very important registers to keep track of in this process are the *%eip* and *%esp* registers. The %eip register, known as the *instruction pointer register*, holds the address of the next instruction to be fetched, decoded, and executed. The %esp register, known as the *stack pointer register*, holds the address that represents the top of the stack (the address of the item most recently pushed to the stack).

◇ Figure 3 illustrates the state of the program prior to the parameters being pushed to stack. Notice that %eip holds the address 0x1f90, which contains the next instruction to be executed "pushl 0x2004". When we step forward and execute the instruction several things will happen:

1. %eip will be incremented to 0x1f96 (the next instruction to be fetched and executed)
2. the value contained in address 0x2004 will be pushed to the stack (if we looked at memory location 0x2004 it would be the value 04 00 00 00)
3. %esp will be decremented to account for the value pushed on the stack

Figure 3: Prior to Parameters Pushed

stack

	address	Value in address (little endian)
	0xbffffa48	7C FA FF BF
	0xbffffa44	74 FA FF BF
	0xbffffa40	01 00 00 00
esp →	0xbffffa3c	01 37 FB 9B

Instructions in memory

```
       0x1f84: movl  $0xa, %eax
       0x1f89: decl  %eax
       0x1f8a: decl  %eax
       0x1f8b: movl  $0x5, %ebx
eip -> 0x1f90: pushl 0x2004
       0x1f96: pushl 0x2000
       0x1f9c: calll 0x1fa9        ; sum
       0x1fa1: addl  $0x8, %esp
       0x1fa4: addl  %ebx, %eax
       0x1fa6: decl  %eax
       0x1fa7: decl  %eax
       0x1fa8: retl
       0x1fa9: pushl %ebp
       0x1faa: movl  %esp, %ebp
       0x1fac: pushl %ebx
       0x1fad: movl  0x8(%ebp), %ebx
       0x1fb0: movl  0xc(%ebp), %eax
       0x1fb3: addl  %ebx, %eax
       0x1fb5: popl  %ebx
       0x1fb6: popl  %ebp
       0x1fb7: retl
```

◇ Figure 4 illustrates the state of the program after the two parameters have been pushed to the stack.

◇ The next instruction (see %eip in Figure 4) to execute is stored in 0x1f9c and is the instruction "calll 0x1fa9". As stated in Section 2, CALL does several things:

1. Pushes the location of the next instruction on the stack. Explanation: after the sum procedure is done running we want to be able to pick up where we left off in the main procedure, which means we need to save the address of the first instruction after the call. In our example that means the value 0x1fa1 is pushed on the stack.
2. Execution then transfers to the address called, in this case 0x1fa9.
3. %eip and %esp are updated accordingly.

◇ Figure 5 illustrates the state of the program after the CALL instruction has been executed.

Figure 4: After Parameters Pushed

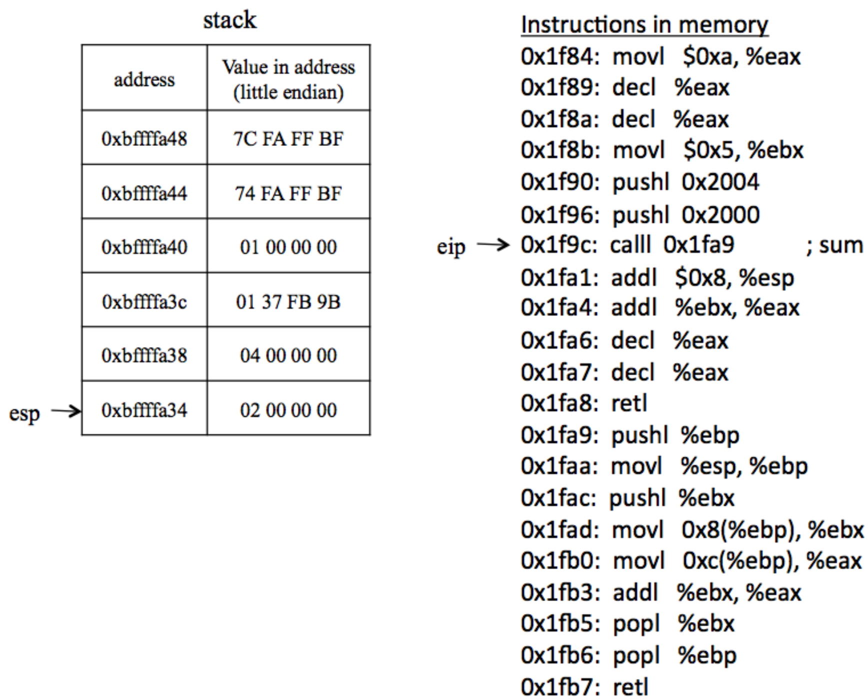
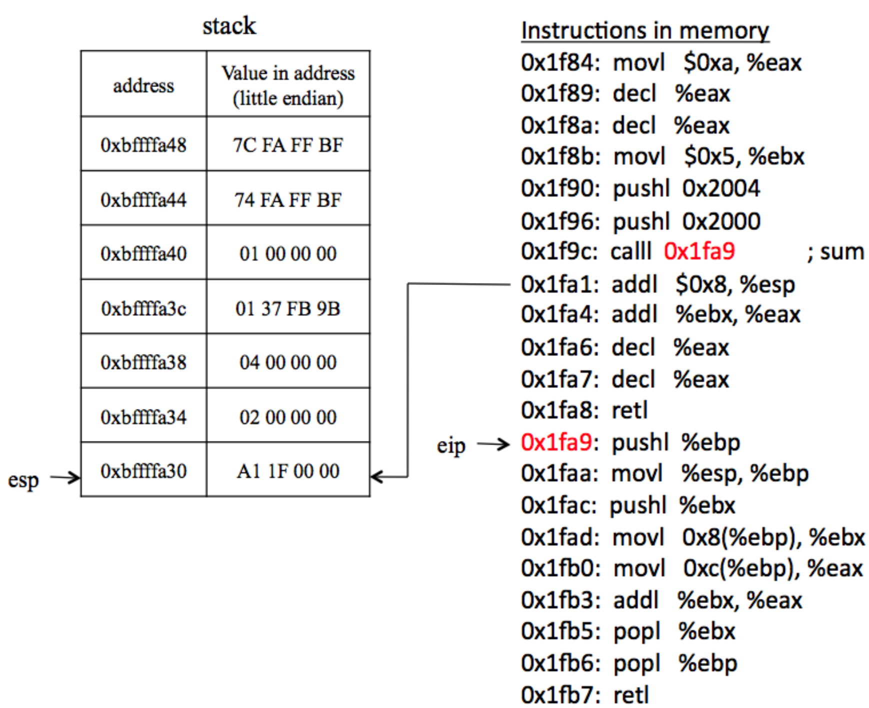


Figure 5: After CALL Instruction



◇ Alright, remember that snippet about stack frames in the Introduction. It is time to set up the new frame so we can refer to the arguments (the values 2 and 4) in the frame and also not have to worry about other things being added to the stack (frame setup could have happened first based on memory needed, see Section 7). This introduces another register called *%ebp* also known as the *base/frame pointer register*. Typically, %ebp points to the boundary of the currently executing stack frame. It is important to note that %esp should not be used to refer to arguments in the frame because %esp will change if other values are pushed to the stack in the procedure (as we are about to see in our example).

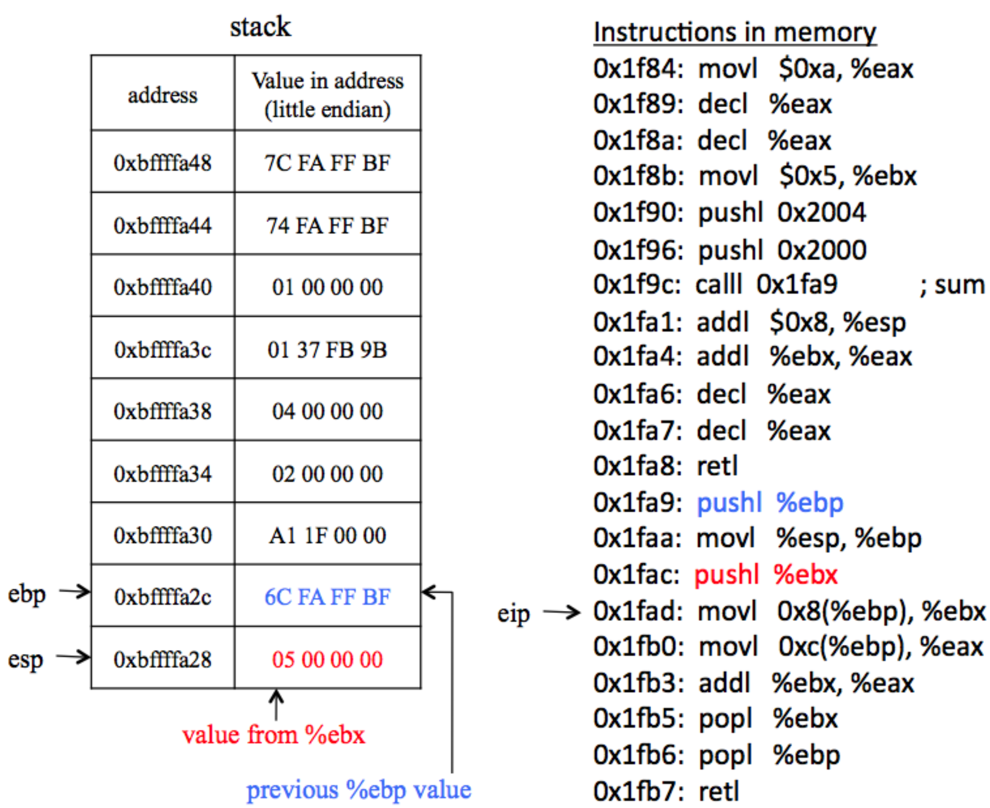
◇ Looking back at the code in Figure 2, the first line in the _sum procedure (line 26) is "push %ebp". This will push the value of %ebp onto the stack. Explanation: We are saving the boundary of the previous stack frame so that we can use %ebp to store the boundary of our new stack frame (of the _sum procedure). When we are done with the _sum procedure we will be able to reset %ebp to its previous value.

◇ Line 27 in the code is the instruction "mov %esp, %ebp". This effectively makes %ebp hold the same value as what is currently in %esp. In other words, they point to the same address: the boundary of the _sum stack frame and the top of the stack.

◇ Earlier in this section we mentioned that for this example we wanted to hang onto the value that was stored in %ebx (5) so it is available when we get back to main. Now is the time to save the value, since we are about to use %ebx to help with our sum calculation. So, we push %ebx to the stack (line 28). Also, remember we did not care to save the previous value in %eax.

◇ Figure 6 illustrates the state of the program after setting up the stack frame and saving %ebx.

Figure 6: After Establishing Frame and Saving %ebx



◇ The time has come to do the math. We need to move our first argument, which is saved at 0xbffffa34 to a register such as %ebx. This is now easy since we know the frame boundary (%ebp) points to the old 4-byte %ebp value and that another 4-byte value (the next instruction address) lies between the boundary and the beginning of our arguments. We can add 8 bytes to %ebp to access the value we need, hence "mov 8(%ebp), %ebx" (line 30). This moves the value 2 to %ebx.

◇ We know the second parameter was pushed first, so it should be 4 bytes further from the first, so we add 12 bytes to %ebp, hence "mov 12(%ebp), %eax" (line 31). This moves the value 4 to %eax.

◇ The next step is to add the two values together, which is accomplished by "add %ebx, %eax" (line 31). So, %eax now holds the value 6. The value is saved in %eax since that register is by default used for return values. Explanation: %eax will not be destroyed when the _sum procedure ends.

◇ The addition is done and we want to prepare to return to main, so we should put things back the way they were before we left main.

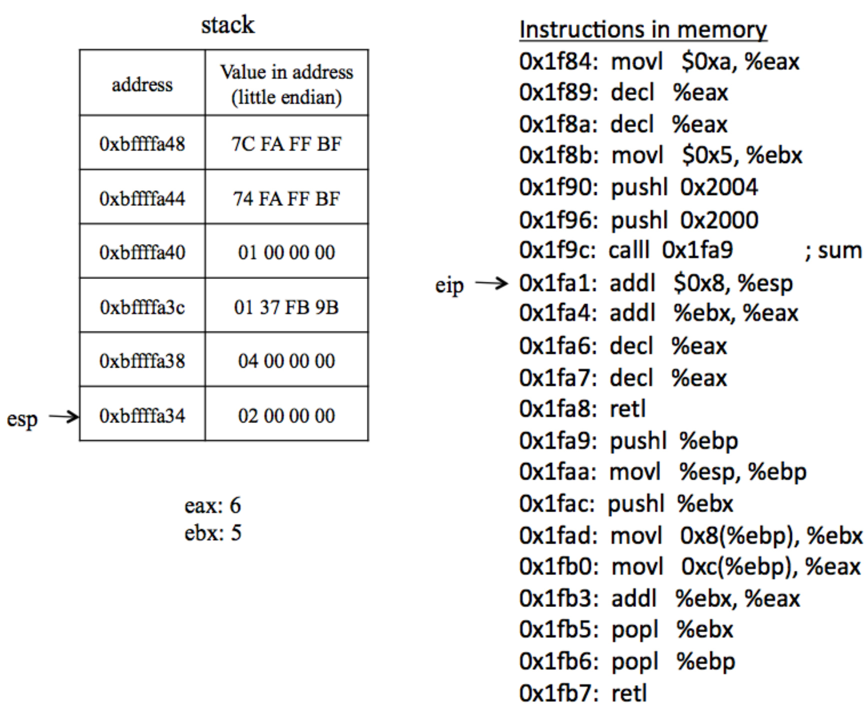
◇ Restore %ebx with the value we wanted to save. The value is on the top of the stack so we just "pop %ebx" (line 33), which removes the value 5 from the stack, places it in %ebx, then adjusts %esp.

◇ Restore %ebp with the previous base pointer address. Again, this is done by popping it off the stack and placing the value in %ebp (line 34).

◇ Return (line 35). As stated in Section 2, RET pops from the top of stack, which should be the address of the next instruction, which was pushed with CALL.

◇ Figure 7 illustrates the state of the program after the return from the _sum procedure.

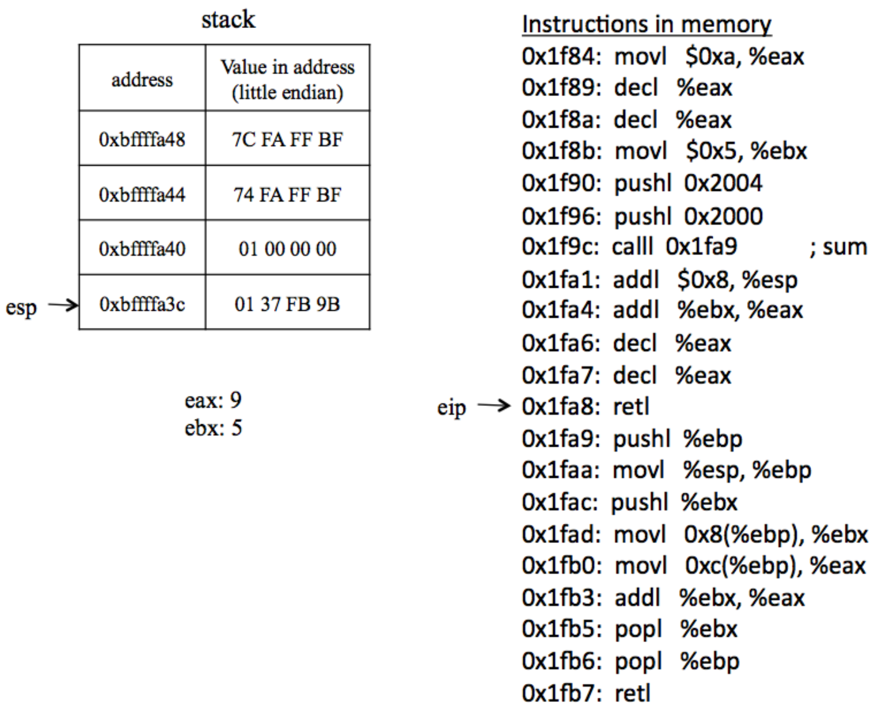
Figure 7: After Return from Sum Procedure



◇ The _sum procedure is done, but we cannot leave the parameters on the stack or else it will interfere with subsequent tasks in the program.

- ◇ The *cdecl* convention requires the calling function (in this case _main) to clean up the stack. We know that two 4 byte parameters are sitting on the stack, so to clean up we add 8 to %esp (line 17), moving the stack pointer 8 bytes and back to where %esp was prior to the CALL. The cleanup instruction is "add $8, %esp".
- ◇ Lines 19-22 continue on with more instructions in _main. %ebx is added to %eax resulting in the value 11 (line 19). %eax is then decremented twice for a final value of 9 (lines 20-21). The program ends with the RET on line 22.
- ◇ Figure 8 illustrates the state of the program just before the final return in the _main procedure.

Figure 8: Just Before _main Return



6 The *stdcall* Difference

stdcall is different from *cdecl* in one significant way: the called function can perform the cleanup of the parameters on the stack instead of the calling function. In our example, that means the cleanup can happen as part of the _sum procedure instead of in _main.

As we stated in Section 2 of this document, the RET instruction can have an optional count stated. If used, the count is added to the %esp register after completion of the return. Our code (Figure 2), implemented in Windows-land/MASM would be nearly identical (aside from the AT&T to Intel syntax translation) with two small changes:

1. Remove line 17 "add $8, %esp"
2. Modify line 35 to "ret 8"

Again, the advantage is that the cleanup code is not needed in _main or anywhere and everywhere we call the _sum procedure. The cleanup code only occurs once, at the end of the procedure, and is a shorter instruction.

7 What About a C++ Implementation?

Given a through review of the previous sections in this document and an understanding some of the basics of assembly, registers, and stack, what might this program look like via a modern C++ compiler? The following figures show sample code written in C++ (in Xcode 6.1.1) and the matching assembly code. This implementation also shows how each function (main and sum) gets their own stack frame, which is why %rpb is being pushed, set, and popped in both functions. Note that %rbp and %rsp are the 64-bit versions of the 32-bit %ebp and %esp registers.

c++ code:

```
int sum(int num1, int num2){
    return num1 + num2;
}


int main()
{

    int num1=2, num2=4, answer;

    answer=sum(num1, num2);

    return 0;
}
```

main function:

```
2   0x100000f10:  pushq   %rbp
3   0x100000f11:  movq    %rsp, %rbp
4   0x100000f14:  subq    $0x10, %rsp
5   0x100000f18:  movl    $0x0, -0x4(%rbp)
6   0x100000f1f:  movl    $0x2, -0x8(%rbp)
7   0x100000f26:  movl    $0x4, -0xc(%rbp)
8   0x100000f2d:  movl    -0x8(%rbp), %edi
9   0x100000f30:  movl    -0xc(%rbp), %esi
10  0x100000f33:  callq   0x100000ef0                ; sum(int, int) at main.cpp:5
11  0x100000f38:  movl    $0x0, %esi
12  0x100000f3d:  movl    %eax, -0x10(%rbp)
13  0x100000f40:  movl    %esi, %eax
14  0x100000f42:  addq    $0x10, %rsp
15  0x100000f46:  popq    %rbp
16  0x100000f47:  retq
```

sum function:

```
0x100000ef0:  pushq   %rbp
0x100000ef1:  movq    %rsp, %rbp
0x100000ef4:  movl    %edi, -0x4(%rbp)
0x100000ef7:  movl    %esi, -0x8(%rbp)
0x100000efa:  movl    -0x4(%rbp), %esi
0x100000efd:  addl    -0x8(%rbp), %esi
0x100000f00:  movl    %esi, %eax
0x100000f02:  popq    %rbp
0x100000f03:  retq
```

Though this is slightly different than our previous example, the assembly should make sense. In the main function, lines 2-4 set up the stack frame. Lines 5-7 are the variables being moved (saved/pushed) to the stack. Using %rbp as the reference point, lines 8-9 move the variables (2, 4) to the registers %edi and %esi. Line 10 is the procedure call for the sum function. The numbers will be added and returned via %eax (see sum function). The registers are then cleared, as the value in %eax is saved to the stack (lines 11-13). The last task is to clean up all those values that were placed on the stack and reset %rbp, which happens on lines 14-15. Finally, the return happens on line 16. The process is almost identical to our previous example.

8 Alternatives

This document does not to go into detail about alternatives to *cdecl* and *stdcall*, but several do exist and might be more useful in certain situations or on other architectures. One common alternative is *fastcall* (for multiple platforms) which uses registers to store several parameters instead of stack memory, which eliminates the overhead of cleanup. *fastcall* is well-suited for an architecture with more registers at its disposal such as ARM, but *fastcall* and its variants (e.g., *vectorcall*) certainly have a place in the world of x86/x86-64.