########################## Calling Conventions ##########################

cdecl - In cdecl, subroutine arguments are passed on the stack. Integer values and memory addresses are returned in the EAX register, floating point values in the ST0 x87 register. Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved. The x87 floating point registers ST0 to ST7 must be empty (popped or freed) when calling a new function, and ST1 to ST7 must be empty on exiting a function. ST0 must also be empty when not used for returning a value.

syscall - This is similar to cdecl in that arguments are pushed right-to-left. EAX, ECX, and EDX are not preserved. The size of the parameter list in doublewords is passed in AL.

pascal - the parameters are pushed on the stack in left-to-right order (opposite of cdecl), and the callee is responsible for balancing the stack before return.

stdcall - The stdcall[4] calling convention is a variation on the Pascal calling convention in which the callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order, as in the _cdecl calling convention. Registers EAX, ECX, and EDX are designated for use within the function. Return values are stored in the EAX register.

Microsoft X64 Calling Convention - The Microsoft x64 calling convention[12][13] is followed on Windows and pre-boot UEFI (for long mode on x86-64). It uses registers RCX, RDX, R8, R9 for the first four integer or pointer arguments (in that order), and XMM0, XMM1, XMM2, XMM3 are used for floating point arguments. Additional arguments are pushed onto the stack (right to left). Integer return values (similar to x86) are returned in RAX if 64 bits or less. Floating point return values are returned in XMM0. Parameters less than 64 bits long are not zero extended; the high bits are not zeroed.

When compiling for the x64 architecture in a Windows context (whether using Microsoft or non-Microsoft tools), there is only one calling convention – the one described here, so that stdcall, thiscall, cdecl, fastcall, etc., are now all one and the same.

In the Microsoft x64 calling convention, it is the callers responsibility to allocate 32 bytes of "shadow space" on the stack right before calling the function (regardless of the actual number of parameters used), and to pop the stack after the call. The shadow space is used to spill RCX, RDX, R8, and R9,[14] but must be made available to all functions, even those with fewer than four parameters.

The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile (caller-saved).[15]

The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile (callee-saved).[15]

For example, a function taking 5 integer arguments will take the first to fourth in registers, and the fifth will be pushed on the top of the shadow space. So when the called function is entered, the stack will be composed of (in ascending order) the return address, followed by the shadow space (32 bytes) followed by the fifth parameter.

In x86-64, Visual Studio 2008 stores floating point numbers in XMM6 and XMM7 (as well as XMM8 through XMM15); consequently, for x86-64, user-written assembly language routines must preserve XMM6 and XMM7 (as compared to x86 wherein user-written assembly language routines did not need to preserve XMM6 and XMM7). In other words, user-written assembly language routines must be updated to

save/restore XMM6 and XMM7 before/after the function when being ported from x86 to x86-64.

###################### C function calling calling- cdecl ########################

The C calling convention, often referred to as cdecl (C declaration), is a commonly used calling convention for functions in the C programming language. It defines a set of rules for how function calls and returns are managed in a program, particularly in relation to the stack, registers, and parameter passing. Below is a detailed description of the cdecl calling convention:

Responsibilities of the Caller:

1. **Push Arguments:** The caller is responsible for pushing function arguments onto the stack, in reverse order (right-to-left). This means the last argument is pushed first.

2. **Prepare Return Address:** The caller pushes the return address onto the stack. This is the address where control should return after the callee function finishes execution.

3. **Call the Function:** The caller transfers control to the callee by jumping to the functions entry point. The exact mechanism for this depends on the architecture.

4. **Save Registers:** If the caller wants to preserve certain registers across the function call, it is responsible for saving their values before making the call and restoring them afterward.

Responsibilities of the Callee:

1. **Allocate Stack Frame:** The callee is responsible for allocating space on the stack for its local variables and any bookkeeping information. The size of this space is determined by the callees needs.

2. **Save the Base Pointer:** The callee saves the current value of the base pointer (if it uses it) onto the stack. The base pointer is used to reference parameters and local variables.

3. **Setup the Stack Frame:** The callee sets up its stack frame by adjusting the stack pointer to point to the newly allocated space for local variables.

4. **Execute the Function:** The callee executes its code and processes the provided arguments and local variables.

5. **Restore the Stack Pointer:** After the function is finished, the callee restores the stack pointer to its original value, effectively deallocating the space used for the stack frame.

6. **Restore the Base Pointer:** If the callee uses the base pointer, it restores its value from the stack.

7. **Clean Up Arguments:** The callee removes the function arguments from the stack. This is especially important in cdecl, where the caller is responsible for cleaning up the stack after the call.

8. **Return the Result:** The callee places the result of the function (if any) in the designated register or memory location.

Returning from the Callee:

1. **Set the Return Value:** The callee places the return value in the appropriate register or memory location.

2. **Restore Registers:** If any registers were saved by the caller before the call, the caller restores them after the call.

3. **Return to Caller:** The callee transfers control back to the caller using the stored return address.

Use of Base Pointer and Stack Pointer:

In cdecl, the base pointer (BP or EBP) is often used as a frame pointer to access function parameters and local variables. The stack pointer (SP or ESP) is adjusted to allocate space for local variables and is restored after the function call. The base pointer allows easy access to function parameters and local variables, even if the stack pointer changes due to function call operations.

Note: Its important to remember that while cdecl is a common calling convention, the specifics can vary depending on the architecture and compiler. Different calling conventions might be used on different platforms or with different compilers. Always refer to your compilers documentation for accurate and up-to-date information.

################################## SCHEME ####################################### ######## https://redirect.cs.umbc.edu/~chang/cs313.s02/stack.shtml #########

UMBC CMSC 313, Computer Organization & Assembly Language, Spring 2002, Section 0101
C Function Call Conventions and the Stack
[Revised 10/18/2001 for better compatibility with Netscape. -RC]

In this page we will review how a stack frame is set up and taken down when a C function call is made. The details are accurate for the platform and compiler we are using, gcc on Linux running on an Intel Pentium chip. There are many possible ways to set up a stack frame --- a different compiler, processor or operating system could use a different set of conventions.

############ A typical stack frame ############

```
ESP ==>                                              0x7FFFD0F8

                 Callee saved registers
                 EBX, ESI & EDI    (as needed)


                 temporary storage


                 local variable #2         [EBP - 8]

                 local variable #1         [EBP - 4]

EBP ==>     Callers EBP

                 Return Address
```

```
Argument #1              [EBP + 8]

Argument #2               [EBP + 12]

Argument #3              [EBP + 16]

Caller saved registers
EAX, ECX & EDX    (as needed)
```

Figure 1 on the right is what a typical stack frame might look like. In these diagrams, the stack grows upward and smaller numbered memory addresses are on top.

This would be the contents of the stack if we have a function foo with the prototype:

```
int foo (int arg1, int arg2, int arg3) ;
```

and foo has two local int variables. (We are assuming here that sizeof(int) is 4 bytes). The stack would look like this if say the main function called foo and control of the program is still inside the function foo. In this situation, main is the "caller" and foo is the "callee".

The ESP register is being used by foo to point to the top of the stack. The EBP register is acting as a "base pointer". The arguments passed by main to foo and the local variables in foo can all be referenced as an offset from the base pointer.

The convention used here is that the callee is allowed to mess up the values of the EAX, ECX and EDX registers before returning. So, if the caller wants to preserve the values of EAX, ECX and EDX, the caller must explicitly save them on the stack before making the subroutine call. On the other hand, the callee must restore the values of the EBX, ESI and EDI registers. If the callee makes changes to these registers, the callee must save the affected registers on the stack and restore the original values before returning.

Parameters passed to foo are pushed on the stack. The last argument is pushed first so in the end the first argument is on top. Local variables declared in foo as well as temporary variables are all stored on the stack.

Return values of 4 bytes or less are stored in the EAX register. If a return value with more than 4 bytes is needed, then the caller passes an "extra" first argument to the callee. This extra argument is address of the location where the return value should be stored. I.e., in C parlance the function call:

```
x = foo(a, b, c) ;
```

is transformed into the call:

```
foo(&x, a, b, c) ;
```

Note that this only happens for function calls that return more than 4 bytes.

Lets go through a step-by-step process and see how a stack frame is set up and taken down during a function call.

############ The callers actions before the function call ############

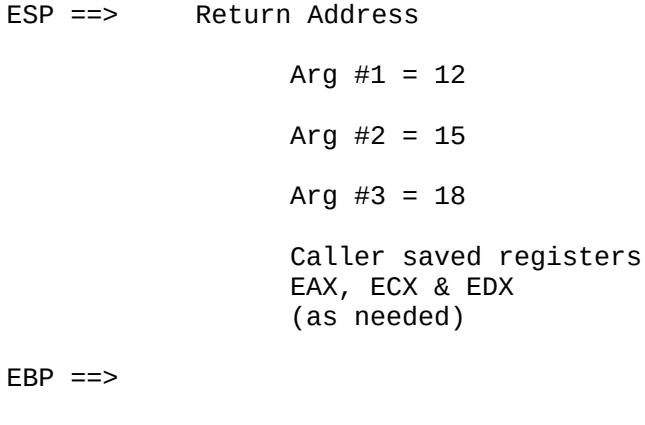


Fig. 2

In our example, the caller is the main function and is about to call a function foo. Before the function call, main is using the ESP and EBP registers for its own stack frame.

First, main pushes the contents of the registers EAX, ECX and EDX onto the stack. This is an optional step and is taken only if the contents of these 3 registers need to be preserved.

Next, main pushes the arguments for foo one at a time, last argument first onto the stack. For example, if the function call is:

```
a = foo(12, 15, 18) ;
```

The assembly language instructions might be:

```
push    dword 18
push    dword 15
push    dword 12
```

Finally, main can issue the subroutine call instruction:

```
call    foo
```

When the call instruction is executed, the contents of the EIP register is pushed onto the stack. Since the EIP register is pointing to the next instruction in main, the effect is that the return address is now at the top of the stack. After the call instruction, the next execution cycle begins at the label named foo.

Figure 2 shows the contents of the stack after the call instruction. The red line in Figure 2 and in subsequent figures indicates the top of the stack prior to the instructions that initiated the function call process. We will see that after the entire function call has finished, the top of the stack will be restored to this position.

The callees actions after function call
When the function foo, the callee, gets control of the program, it must do 3 things: set up its own stack frame, allocate space for local storage and save the contents of the registers EBX, ESI and EDI as needed.

So, first foo must set up its own stack frame. The EBP register is currently pointing at a location in mains stack frame. This value must be preserved. So, EBP is pushed onto the stack. Then the contents of ESP is transferred to EBP. This

allows the arguments to be referenced as an offset from EBP and frees up the stack register ESP to do other things. Thus, just about all C functions begin with the two instructions:

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

The resulting stack is shown in Figure 3. Notice that in this scheme the address of the first argument is 8 plus EBP, since mains EBP and the return address each takes 4 bytes on the stack.

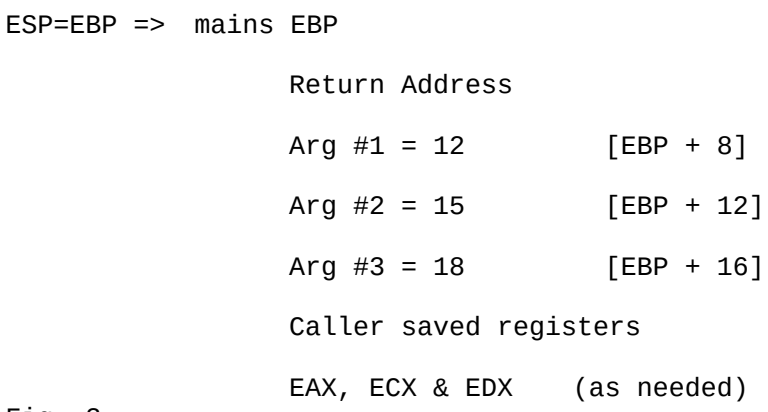


Fig. 3

In the next step, foo must allocate space for its local variables. It must also allocate space for any temporary storage it might need. For example, some C statements in foo might have complicated expressions. The intermediate values of the subexpressions must be stored somewhere. These locations are usually called temporary, because they can be reused for the next complicated expression. Lets say for illustration purposes that foo has 2 local variables of type int (4 bytes each) and needs an additional 12 bytes of temporary storage. The 20 bytes needed can be allocated simply by subtracting 20 from the stack pointer:

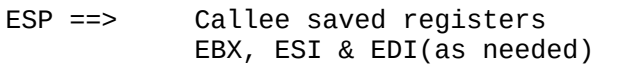
```
        sub     esp, 20
```

The local variables and temporary storage can now be referenced as an offset from the base pointer EBP.

Finally, foo must preserve the contents of the EBX, ESI and EDI registers if it uses these. The resulting stack is shown in Figure 4.

The body of the function foo can now be executed. This might involve pushing and popping things off the stack. So, the stack pointer ESP might go up and down, but the EBP register remains fixed. This is convenient because it means we can always refer to the first argument as [EBP + 8] regardless of how much pushing and popping is done in the function.

Execution of the function foo might also involve other function calls and even recursive calls to foo. However, as long as the EBP register is restored upon return from these calls, references to the arguments, local variables and temporary storage can continue to be made as offsets from EBP.



```
            temporary storage         [EBP - 20]

            local variable #2         [EBP - 8]

            local variable #1         [EBP - 4]

EBP==>      mains EBP

            Return Address

            Arg #1 = 12               [EBP + 8]

            Arg #2 = 15               [EBP + 12]

            Arg #3 = 18               [EBP + 16]

            Caller saved registers
            EAX, ECX & EDX (as needed)
```

Fig. 4

The callees actions before returning

```
ESP ==>     Arg #1 = 12

                  Arg #2 = 15

                  Arg #3 = 18

                  Caller saved registers
                  EAX, ECX & EDX (as needed)

EBP ==>
```

Fig. 5

Before returning control to the caller, the callee foo must first make arrangements for the return value to be stored in the EAX register. We already discussed above how function calls with return values longer than 4 bytes are transformed into a function call with an extra pointer parameter and no return value.

Secondly, foo must restore the values of the EBX, ESI and EDI registers. If these registers were modified, we pushed their original values onto the stack at the beginning of foo. The original values can be popped off the stack, if the ESP register is pointing to the correct location shown in Figure 4. So, it is important that we do not lose track of the stack pointer ESP during the execution of the body of foo --- i.e., the number of pushes and pops must be balanced.

After these two steps we no longer need the local variables and temporary storage for foo. We can take down the stack frame with these instructions:

```
        mov     esp, ebp
        pop     ebp
```

The result is a stack that is exactly the same as the one shown in Figure 2. The return instruction can now be executed. This pops the return address off the stack and stores it in the EIP register. The result is the stack shown in Figure 5.

The i386 instruction set has an instruction "leave" which does exactly the same thing as the mov and pop instructions above. Thus, it is very typical for C functions to end with the instructions:

```
leave
ret
```

The callers actions after returning

After control of the program returns to the caller (which is main in our example), the stack is as shown in Figure 5. In this situation, the arguments passed to foo is usually not needed anymore. We can pop all 3 arguments off the stack simultaneously by adding 12 (= 3 times 4 bytes) to the stack pointer:

```
add     esp, 12
```

The caller main should then save the return value which was placed in EAX in some appropriate location. For example if the return value is to be assigned to a variable, then the contents of EAX could be moved to the variables memory location now.

Finally, the main function can pop the values of the EAX, ECX and EDX registers if their values were preserved on the stack prior to the function call. This puts the top of the stack at the exact same position as before we started this entire function call process. (Recall that this position is indicated by a red line in Figures 2-5.)

Examples

Understanding these conventions lets you do two things: 1) write assembly language programs that can be called from a C program, and 2) call standard C functions from your own assembly language programs.

As an example of the first case, we have a function arrayinc written in assembly language that adds one to each element of an integer array. The array is passed to arrayinc as the only argument. Here are the files:

- The assembly language program that implements arrayinc: arrayinc.asm.
- The C function that calls the arrayinc function: arraytest.c.
- A transcript of the UNIX commands & output: arraytest.txt.

Notice that the C program treats arrayinc as any other function that is implemented elsewhere. It really doesnt care if the function is implemented in C or in assembly language.

The second situation, calling C functions from assembly language programs, is commonly used to invoke C input/output routines. Here you must decide if your main function will behave like a C function or as an a "normal" assembly language program.

An example of the first case, we call printf from an assembly language program. The entry point for the assembly language program is labeled main and must be declared global. This program must behave like a C function. It must set up and take down the stack frame and preserve the registers according to the C function call convention. The identifier printf must be declared external to get the linker to do the right thing. We also use gcc to do the final linking and loading (instead of using ld), since gcc knows which library contains the printf function. Here are the files:

- An assembly language program that calls printf (version 1): printf1.asm.

A transcript of the UNIX commands & output: printf1.txt.

In the second example, the assembly language program is a "normal" one. The entry point is labeled "_start" and the program exits using a Linux kernel system call. In this case, we need to give gcc the "-nostartfiles" option for the linking to work correctly.

An assembly language program that calls printf (version 2): printf2.asm.
A transcript of the UNIX commands & output: printf2.txt.

A good way to understand how C handles function calls is to examine the assembly language code generated by the compiler. We can use "gcc -S" to tell the gcc compiler to create a file with extension ".s" that contains the assembly language code. The gcc compiler does normally generate assembly language code, but without the -S option it does not save it to a file. Unfortunately the assembly code in the .s file is in AT&T-style syntax. This can be converted to Intel-style syntax using an "intel2gas" command.

Here are the files from a simple example.

The original C program: cfunc.c.
The assembly output in AT&T style: cfunc.s.
The assembly output converted to Intel style: cfunc.asm.
A transcript of the UNIX commands & output: cfunc.txt.

For a more complicated example, look at the assembly code generated by gcc for a program with nested function calls:

The original C program: cfunc2.c
The converted assembly language output in Intel style: cfunc2.asm.

Finally, we have an example with a C function with a return value with size more than 4 bytes.

The original C program: cfunc3.c.
The converted assembly language output in Intel style: cfunc3.asm. The comments in this file were added afterwards.

Last Modified: 13 Mar 2002 21:23:05 EST by Richard Chang
to Spring 2002 CMSC 313 Section Homepage