The CALL/RET Instructions and the Stack

In the previous chapter, we explained most of the lines of the assembler file that is generated by the compiler for a C file, with one important exception which is the "ret" instruction at the end of the function. Let's just review a simple example:

```
unsigned a = 6;
void f() {a = a + a;}
void g() {f(); f();}
```

The corresponding assembler file is:

```
     .file     "x.c"
     .intel_syntax
.globl _a
     .data
     .align 4
_a:
     .long     6
     .text
.globl _f
     .def _f;  .scl 2;   .type     32;  .endef
_f:
     mov  eax, DWORD PTR _a
     add  DWORD PTR _a, eax
     ret
.globl _g
     .def _g;  .scl 2;   .type     32;  .endef
_g:
     call _f
     call _f
     ret
```

Most of the lines in this file are familiar from the previous chapter. The **.file** directive gives the name of the original source file for informational purposes. The **.intel_sytax l**ine tells the assembler we are using Intel rather than AT&T syntax. The **.globl** lines are used to tell the assembler that the corresponding symbols should be marked as globally accessible from other files. The **.data** directive marks the start of the data in the program, and **.text** marks the start of the program. The **.align** directive makes sure that the variable is on a 4-byte boundary, for efficiency purposes. The **.long** directive actually generates four bytes of data containing the specified initial value. The label lines, such as **_a:** associate assembler labels with specific addresses in memory. The **mov** instruction loads a word into a register. The **add** instruction adds the contents of a register to a word in memory.

The previous paragraph identifies every line in the assembler output (see previous chapter if you need to consult the more detailed descriptions), with two notable exceptions, the **call** instruction and the **ret** instruction. In general terms, we know what these do. The **call**

instruction clearly transfers control to another procedure, and the **ret** instruction returns to the instruction following the **call**. For rough understanding of what the assembler means, this is an adequate level of explanation. That's even a sufficient explanation for writing assembler yourself. You can just code the **call** and the **ret** following the above example and your code will work fine.

However, that's not good enough! We are aiming here at a complete understanding of how the machine works with no magic at all. It's all very well to say that the ret instruction returns past the call, but the question remains of how this works. Remember that the machine executes instructions in isolation, it does not keep a memory of all the instructions it has executed, so it can't use some algorithm such as "check the list of instructions you have executed, find the most recent call instruction, and return to the instruction that follows the call." That's fine as a description of what is supposed to happen, but it is definitely not fine as a description of how things actually work at the machine level.

When an instruction is executed, it has only the current machine environment available, consisting of the current register values (including the instruction pointer EIP) and the current contents of memory.

So the **ret** instruction must be able to do its job by consulting only the register and memory state at the point when it is executed. From this we can conclude that somehow the **call** instruction must figure out the address of the following instruction, and save that address in either a register or a memory location in such a way that it is accessible to the **ret** instruction. So far, so good, the only issue that remains is how exactly does the **call** instruction save this information, and how does the **ret** instruction retrieve the information stashed away by the **call**.

Using a register to hold the return point

One way this might be accomplished is to put the address in a register. Let's redesign the ia32 architecture to use that approach. Let's pick a register, say EDI, which will contain the so called "return point", that is the address of the instruction immediately following the call. Then all the ret instruction would have to do is to copy that value back into the EIP register, to cause the instruction following the call to be executed next.

The ia32 does not in fact work that way, but it is a common approach. If you have a Mac, which uses the PowerPC, and you look at the design of that machine, you will find that this is exactly how the PowerPC works. The PowerPC has 32 registers, labeled %0 to %31, and register %31 is used to stash the return point on a call. The equivalent of a ret instruction simply has to copy this value into the program counter to achieve the return.

To understand why the ia32 does not use this technique, consider one little problem with this approach. Look again at the generated assembler language for function **_g**. Like all functions this also ends with a **ret** instruction. If call sets EDI to the return point, and ret uses this return point, we have a problem. The problem is that the call within function g will clobber the saved return point. Although this technique of using a fixed register for the return works fine for one level of call, it does not work so nicely for the case of multiple calls. One way around this would be to temporarily save the return point in some other

register in the higher level function. That's quite reasonable on a machine like the PowerPC with lots of registers, but it's bad news on the ia32 with its rather miserable set of 8 registers (only 7 of which can be freely used). That means that in practice we would have to resort to having the high level function store the return point in memory, and retrieve it for the return.

How the ia32 Really Does Call/Ret

The designers of the architecture, faced with this dilemma, made the decision to avoid using registers for saving the return point, and instead to use a memory location. That sounds good in general terms. All we have to do is to have the call instruction save the return point in memory, and then the ret instruction has to retrieve this value from memory to know where to return to.

Great! Problem solved! Oops, wait a moment, there is one pesky detail we did not address. Where shall we put this value in memory? If we choose a fixed location, say location zero, then we are stuck with the problem of multiple level calls again, since a low level call will destroy the return point required by the high level call. Let's think for a moment about this problem. The way **call** and **ret** work is that control always returns to the most recently executed **call** when the **ret** is executed, or in other words, we want to always access the most recently stored value. If there is a sequence of calls, then the return points will be accessed in reverse order, or first in/last out. Does this recall some standard data structure? It should! What we need here is a stack, since the behavior of a stack is exactly what we want here. The **call** needs to push a return point value onto the stack and the **ret** needs to pop off the value.

Now, we have to decide how the machine should implement the stack. The usual way to implement a stack at the machine level is to use a sequence of locations in memory in conjunction with a stack pointer value that tells us the location in memory of the top of the stack. The push operation then means move the stack pointer one way and store a value, and the pop operation means retrieve the value and restore the stack pointer by moving it in the opposite direction. It doesn't matter which dircection we use for the push as long as the pop knows to use the reverse direction.

Let's suppose we implement a stack with a sequence of words building down in memory so that the deepest (first pushed) value is at a high memory address, and the top of stack value (last pushed) is at a low memory address. A picture of the stack at some point in time might look like (assuming location 00100000 is the location of the deepest word, or in other words the starting point of the stack):

```
Location 00100000  1st word pushed onto stack
Location 000FFFFC  2nd word pushed onto stack
Location 000FFFF8  3rd word pushed onto stack (stack top)
```

Here the stack pointer would have the value **000FFFF8** pointing to the most recently pushed value on the stack. If we push one more word onto this stack, we do it by subtracting 4 (4 bytes per word) from the stack pointer, and storing the new value, making the stack look like this:

```
Location 00100000  1st word pushed onto stack
Location 000FFFFC  2nd word pushed onto stack
Location 000FFFF8  3rd word pushed onto stack
Location 000FFFF4  4th word pushed onto stack (stack top)
```

And now the stack pointer has the value **000FFFF4**. If we then pop off this top value, we get the value we just stored, and the stack returns to its previous state:

```
Location 00100000  1st word pushed onto stack
Location 000FFFFC  2nd word pushed onto stack
Location 000FFFF8  3rd word pushed onto stack (stack top
```

This kind of stack structure is exactly what we need for storing return points. If we push the return point values on to the stack when a call is executed, then we just have the ret instruction pop off the most recent value to use as the new value of the EIP register. The only remaining detail is where to store the stack pointer, and you can probably guess the answer. Remember in the last chapter, we said there was a register called ESP, and we said that for reasons we would explain later, you must not store anything directly in this register. Well now it's later, and we can learn that in fact ESP is used precisely as a stack pointer for managing the return point stack.

We now have all the ingredients to describe the exact operation of the **call** and **ret** instructions, removing the remaining mystery from the assembler files we have seen so far. The **call** instruction works in the following steps.

> First step the EIP register past the call. The call instruction certainly knows how long it is, so it has the necessary information to perform this operation. Indeed all instructions routinely perform this operation, since it is part of the basic instruction flow.
>
> Second step, subtract 4 from the current value of ESP
>
> Third step, using the address that is now in ESP after subtracting 4, copy the value of the EIP register (the return point) into the word addressed by ESP.
>
> Set the EIP register to contain the address given in the call instruction, which is the function to be called. This will result in the processor executing the first instruction of that routine immediately after the call instruction.

Now the corresponding execution of the **ret** instruction is as follows:

> Copy the value addressed by the current value in ESP (the return point) into the EIP register, so that the next instruction to be executed will be the instruction following the most recently executed call instruction.
>
> Add 4 to ESP to pop this return point off the stack, so that the next ret instruction will retrieve the previous return point.

That's all there is to it. By executing these precise steps, the **call** and **ret** instructions achieve the desired result of properly nested calls. The only important point is to make sure at the beginning of the execution of the program that ESP is set to an appropriate value, which is the ending address of an area of memory set aside for return points. The size of this area needs to correspond to at least the deepest level of call in the program. In practice, we have plenty of memory available on modern machines. For instance if we set aside an area of 64K bytes for this purpose, that's enough for over 16,000 levels of calls, which should be enough for virtually any program, even one that uses recursive routines.

An Example (with Recursion)

Speaking of recursion, one nice advantage of using a stack in this way is that it takes care of recursion without any further special considerations. Let's modify our sample program a bit:

```
unsigned a = 2;
void f() {if (a) {a--; f();}}
void g() {f();}
```

The corresponding assembly language, considering just the code for the two functions f and g is:

```
_f:
     cmp  DWORD PTR _a, 0
     je   L2
     dec  DWORD PTR _a
     call _f
L2:
     ret

_g:
     call _f
     ret
```

When we execute this program by calling the function g, we expect g to call f, and then f should call itself recursively, with three calls to f in all. On the third call to f, a has counted down to zero, and f should return up the tree of calls to f, finally returning to g, and from there to the caller of g. Let's trace the execution of the program. We will assume that ESP contains the value **0001000** on entry to g. This value is set to an appropriate value by the operating system, and generally we don't need to worry about it. This means that on entry to g, the stack looks like:

Location 00100000 return point past call of g ← ESP

Now, g executes its **call** instruction, pushing a new return point onto the stack, which now looks like this:

```
Location 00100000  return point past call of g
Location 000FFFFC  return point past call of f in g   ← ESP
```

Now function f gets control. Variable a has the value 2, so it is decremented to 1, and we get a recursive call to f, leaving the stack looking like:

```
Location 00100000  return point past call of g
Location 000FFFFC  return point past call of f in g
Location 000FFFF8  return point past call of f in f   ← ESP
```

Function f gets control again. Variable a now has the value 1, so it is decremented to 0, and we get another recursive call to f, leaving the stack looking like:

```
Location 00100000  return point past call of g
Location 000FFFFC  return point past call of f in g
Location 000FFFF8  return point past call of f in f
Location 000FFFF4  return point past call of f in f   ← ESP
```

Function f gets control once more. Variable a now has the value 0, so the if condition is false, and the **ret** instruction in f is executed. This ret picks up the return point from the stack, and increments ESP, leaving the stack looking like:

```
Location 00100000  return point past call of g
Location 000FFFFC  return point past call of f in g
Location 000FFFF8  return point past call of f in f   ← ESP
```

Note that when this book was written, it was quite easy to write the above paragraph by cutting and pasting the paragraph above. Stacks are like that, a push and a pop leave the stack the way it was. We are now back in f, immediately following the **call** instruction, and the instruction is another **ret**, so again we pick off the top return point, and the stack now returns to:

```
Location 00100000  return point past call of g
Location 000FFFFC  return point past call of f in g   ← ESP
```

You can see how we are retracing our steps, we are once again at the ret instruction immediately following the call in function f, and so once again we pop the top return point off the stack, which now returns to:

```
Location 00100000  return point past call of g        ← ESP
```

Now we are back in function g immediately following the call instruction. Function g then executes its ret instruction picking off the remaining return point on the stack to return to the caller of g, and we are done.

If we knew the exact locations of all the instructions and how long they are, we could lay out this example using exact return point locations on the stack (and of course that is the way the machine does things), but the whole point of using an assembler is that we don't need to worry about such things. The assembler takes care of making sure that appropriate space is allocated for each instruction.

Optimizing the Code

It's always interesting to see what the gcc optimizer will do with little example programs, and in this case, it is quite instructive. Here again is our test program:

```
unsigned a = 2;
void f() {if (a) {a--; f();}}
void g() {f();}
```

The corresponding assembly language, considering just the code for the two functions f and g with the –O2 optimization switch set is:

```
_f:
     mov  eax, DWORD PTR _a
     jmp  L3
     .p2align 4,,7
L5:
     dec  eax
     mov  DWORD PTR _a, eax
L3:
     test eax, eax
     jne  L5
     ret
     .p2align 4,,15

_g:
     jmp  _f
```

Well, there is quite a bit going on here. First, quite impressively, the gcc compiler has looked at our recursive implementation of f and figured out that it would be more efficient to use a loop to decrement a. So the recursion has been replaced by a loop, and we avoid three function calls. Of course if it was really, really clever, it could have optimized all the way to:

```
_f:
     mov  DWORD PTR _a, 0
     ret
```

But that's asking a bit much, and in any case, it is not clear that it would be such a useful optimization since it is a bit peculiar to write such awful code whose only purpose is to set a variable to zero in any case ☺ The general optimization of replacing recursion by a loop is however a useful one, because, especially for those who are really comfortable with

recursion, it is often clearer to express things recursively. Furthermore, there are languages which have recursion but no loops (any loop can always be expressed in recursive form), and in such languages, this optimization becomes really important. Note that it is not only a time optimization, but also a space optimization, since the stack space is not required. Consider a little modification of the program namely setting a to have an initial value of 1000000000 (one thousand million). Now the unoptimized program requires nearly four gigabytes of stack space, which you probably can't get, since that's the maximum memory on the machine, but the optimized program requires no stack space and can execute in under a second of machine time on a modern machine. This general optimization is called "tail recursion removal". Tail recursion is the name we give to the loop-like construct in which the very last thing a function does is to call itself again, and such a call can always be replaced by a jump.

Now let's look at the other optimization that is interesting and instructive. If you understand this optimization, you really understand how the return point stack works. In the unoptimized program, the code for function g looked like:

```
_g:
     call _f
     ret
```

In the optimized program, the same function generates:

```
_g:
     jmp  _f
```

The **call**/**ret** sequence has been replaced by a single unconditional jump instruction which simply transfers control to function f without saving a return point. That can't be right! Or can it? Well let's thing what happens here. When g passes control to f, the top entry on the return stack is the return point past the call instruction which called g. Now when f reaches its return point, it would expect to return past the call instruction in g, but instead it pops the top return point off the stack and returns directly past the call instruction that called g. Well that's **perfectly** fine. That's all the ret after the call in the unoptimized version does anyway, so it is just fine to return directly skipping the unnecessary intermediate step. In fact it is always valid to replace a **call**/**ret** instruction by a **jmp**, and indeed this is basically what tail recursion removal is all about.

Infinite Recursion

In the world of loops, we have to worry about infinite loops. In the world of recursion, we similarly have to worry about infinite recursion. Let's write a really nasty little function:

```
   void f() {f();}
```

In unoptimized mode, the generated code for this function is:

```
_f:
     call _f
```

```
ret
```

What happens if we execute this function? Each time we call f, it pushes another junk return point on to the stack and calls itself again. Why junk? Well there is never any way of getting to the **ret** instruction, so these return points are useless. So we have an infinite loop. But wait, it's worse than that, as the recursion executes, more and more return points are being pushed onto the stack, and the value in ESP is constantly decreasing. Remember that code and data share the same memory. Sooner or later, disaster will strike when the ESP value points to the memory locations containing the code of function f. One of two things will happen. On modern versions of the ia32 chip, it is possible for the operating system to protect code from being destroyed by memory writes. This is a useful defense both against evil programs (for example viruses) that do their dirty work by deliberately clobbering code, and also proves useful in catching bugs like this. If the code is memory protected, the processor will detect the attempt to modify it, and automatically call an operating system routine that will boot you off the machine.

If the code is not protected, the effect can be truly mysterious, since in this case the code gets destroyed, and it is almost impossible to figure out exactly what will happen. The moral of this story is to be careful to avoid infinite recursion!

With the material in this chapter, we have now got to the point of complete understanding of some simple ia32 machine language programs. Now we will expand the kind of C code that we ask gcc to compile, and see what other mysteries this architecture has in store for us.