Using a Frame Pointer

In our examples so far, we have always compiled with **–fomit-frame-pointer**. That made the resulting code simpler, but in practice we usually do use a frame pointer. In the general case it is required, and it makes life a lot more convenient to use a frame pointer for all functions. So before that makes sense, we have to find out what exactly we mean by a frame pointer.

To motivate this, and follow up on the comment that a frame pointer is required, we first look at a case where we really cannot manage without one.

The C Function alloca

The C standard (and indeed all C compilers for a long time) have implemented a dynamic storage function called **alloca**. This is similar to **malloc** with one very important difference. The storage from **malloc** stays around until you do a corresponding **free** operation. However, in the case of **alloca**, the storage is automatically released when you return from the current function, that is, the function in which the **alloca** call occurred.

There are several possible ways of implementing **alloca**. One approach would be to use **malloc**, and then make sure that the function did an implicit **free** call before it returned. But with a little thought we can see a much simpler implementation, which is in fact the one that is intended by the designers of this facility. What other storage do we know about that is freed automatically on return from a function? The answer of course is the local variables of the function, and we achieve this by allocating them in the local stack frame. A simple add to the stack pointer releases the entire frame. If we have **alloca** allocate its storage in the local stack frame, then it will be automatically freed on the return. Let's look at a little example in C:

```
unsigned sum (unsigned s) {
  char *ptr;
  unsigned j;
  ptr = (char *)alloca(16);
  for (j=0; j<16; j++) ptr[j] = j;
  for (j=0; j<16; j++) s += ptr[j];
  return s;
}
```

The code generated for the alloca call might look like:

```
        sub     esp, 16
        mov     eax, esp
```

The purpose of this code is simply to allocate 16 bytes on the local stack frame by increasing its size by 16 bytes. After the **sub** instruction has increased the size, the new value of esp points to the allocated area, and is copied to **EAX** for further use.
We fudged a bit above, and said "might look like". In fact the actual code generated by gcc is:

```
sub     esp, 32
mov     eax, esp
add     eax, 15
shr     eax, 4
sal     eax, 4
```

What is happening here is that the specification of **alloca** requires that the allocated storage be maximally aligned, which on our machine means 16-byte aligned. The way this is achieved is to allocate 16 bytes more than is needed, and then the last three instructions adjust the address to insure that it is 16-byte aligned.

So far so good. We have in **EAX** the address of a properly aligned block of appropriate length (16 bytes in this case) storage allocated on the local stack. When the function returns, the local stack frame will be removed, by restoring the old value of **ESP**, and the **alloca** storage (along with all local variables) will be freed.

There is one huge problem. The code sequence above modified the value in **ESP**. But we rely on the value in **ESP** for addressing both our local variables and the arguments to the function. If we go changing **ESP** by arbitrary amounts, we are in big trouble, because we don't know where our local variables and parameters are any more relative to **ESP**. Note that in the case above, the argument to **alloca** is a static constant, but in the general case it can be a computed variable. After such an **alloca** call the stack looks like:

Arguments to the current function
Return point for call
Local variables for function
Alloca variable of unknown size
Alignment fill of unknown size

where **ESP** points to the bottom of this frame. But since the areas at the bottom of the frame are of unknown (at compile time) size, we have no idea how to address the local variables or the arguments, and we don't know how much to add to **ESP** to release the frame and get back to the return point at the end of the function.

This is a mess, which is cleaned up completely by the introduction of a frame pointer. The idea of a frame pointer is that we capture the value of **ESP** in a separate register on entry to the function. This saved value, which is called the frame pointer, is never changed throughout execution of the function, even if **ESP** is changed. On the ia32 architecture, it is conventional to use **EBP** to hold the frame pointer (in fact that's where its name comes from **BP** = Base Pointer, since this value points to the base of the stack frame).

Although the frame pointer is only absolutely required if a function has alloca calls, it is convenient to use a frame pointer for any function. To see the frame pointer in action, let's recompile a simple function we looked at in a previous chapter:

```
struct node { unsigned val; struct node *next; };
```

```
unsigned find (struct node *h, unsigned val) {
  struct node *p;
  p = h;
  while (p) {
    if (p->val == val) return 1;
    else p = p->next;
  }
  return 0;
```

With the frame pointer omitted, gcc generated:

```
_find:
        sub     esp, 8
        mov     eax, DWORD PTR [esp+12]
        mov     DWORD PTR [esp+4], eax
L2:
        cmp     DWORD PTR [esp+4], 0
        je      L3
        mov     eax, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [eax]
        cmp     eax, DWORD PTR [esp+16]
        jne     L4
        mov     DWORD PTR [esp], 1
        jmp     L1
L4:
        mov     eax, DWORD PTR [esp+4]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [esp+4], eax
        jmp     L2
L3:
        mov     DWORD PTR [esp], 0
L1:
        mov     eax, DWORD PTR [esp]
        add     esp, 8
        ret
```

Now we will recompile without the **–fomit-frame-pointer switch**, which will cause a frame pointer to be used. The default is to use a frame pointer always. It only gets omitted if we specifically use this switch to suppress it. Even that switch is only a request. If we use **alloca**, the compiler ignores the request, since otherwise it would be stuck! Here is the code with a frame pointer:

```
_find:
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     eax, DWORD PTR [ebp+8]
```

```
        mov     DWORD PTR [ebp-4], eax
L2:
        cmp     DWORD PTR [ebp-4], 0
        je      L3
        mov     eax, DWORD PTR [ebp-4]
        mov     eax, DWORD PTR [eax]
        cmp     eax, DWORD PTR [ebp+12]
        jne     L4
        mov     DWORD PTR [ebp-8], 1
        jmp     L1
L4:
        mov     eax, DWORD PTR [ebp-4]
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR [ebp-4], eax
        jmp     L2
L3:
        mov     DWORD PTR [ebp-8], 0
L1:
        mov     eax, DWORD PTR [ebp-8]
        leave
        ret
```

The critical instructions here are the initial entry instructions (which are part of the *prolog* of the function, that is the instructions generated at the start of the function to get things set up):

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

The first instruction is saving **EBP** on the stack. We need to do this, since we are about to clobber **EBP**. Our caller is also presumably using a frame pointer stored in **EBP**. We can't just go destroying that value, or we will be in bad shape when we return, since our caller will depend **on EBP** containing the right value. So part of the requirement in the **ABI** is that every function must return with the value in **EBP** unchanged. That's achieved in one of two ways. Either the function never touches **EBP** (that's how all the functions we compiled so far worked), or if **EBP** is changed it must be saved on entry, and restored on exit.

The second instruction copies the value in **ESP** into **EBP**. From now on all references to arguments and local variables will use this **EBP** value that does not change, rather than **ESP**. We can understand the code more clearly if we have the stack layout clearly in mind as we read it. As usual we will assume a starting value of **00100000** for the stack pointer.

```
Addr 00100000  copy of value of argument val   ← [EBP+12]
Addr 000FFFFC  copy of value of argument h     ← [EBP+8
```

```
Addr 000FFFF8  return point past call of find  ← [EBP+4]
Addr 000FFFF8  saved value of caller EBP       ← [EBP]
Addr 000FFFF4  value of local variable p       ← [EBP-4]
Addr 000FFFF4  local variable to hold result   ← [EBP-8]
```

Once we understand this stack frame layout, we can see that the code of the function is essentially unchanged. The value in **ESP** actually ends up pointing to the result variable, but we don't use **ESP** in our addressing calculations any more. This simplifies the code quite a bit. In particular, we don't need to worry about **PUSH** instructions changing the addressing, not to mention alloca calls with variable arguments!

In the above stack frame layout, you will notice that arguments are addressed using positive offsets off the frame pointer **EBP**, and local variables are addressed using negative offsets off the frame pointer. That's typical, and is fine, since the addressing mode we use (regiser indirect with offset) allows both positive and negative offsets.

One nice thing about this layout is that you can always immediately find the return point from any function by using the frame pointer value. The return point is just above the address referenced by the frame pointer, and this will always be the case, regardless of the number of arguments or local variables. Furthermore, the frame pointer points to the old saved frame pointer, so you can work your way back on the stack easily. This is very convenient for tools like the debugger for tracing their way through a call history. By call history, we mean the sequence of calls that got us to a particular point in the program execution.

One final thing to look at is the return sequence. How do we get rid of the frame and get back to the return point. We used to do this by adding a known constant to **ESP**, but the whole idea of using a frame pointer is to avoid the need of keeping track of the value in ESP. In fact the solution is simple, since **EBP** points to the top of the frame, we can simply execute

```
mov     esp, ebp
```

That removes the local frame, by copyiing the frame pointer value back into the stack pointer. This undoes all modifications to the stack pointer that occurred after the original setting of the frame pointer, including allocation of the local stack frame variables, as well as any changes due to alloca calls. After that instruction has executed, **EBP** now points back to the saved **EBP** value., and we simply do:

```
pop     ebp
```

to restore the old frame pointer. We haven't actually seen the **pop** instruction before, but it is easy to guess that it is the exact opposite of the push instruction. It removes a value from the stack placing it in the destination (in this case **EBP**) and adjusts the stack pointer by adding four after retrieving the value.

The **push** and **pop** instructions are useful in conjunction with one another if you run out of registers. For example, you can save a few registers using push instructions:

```
        push    eax
        push    ebx
        push    ecx
```

Now you can execute some code using these registers, and when you are done with them, the original values can be retrieved using pop instructions, which must be in the reverse order since this is a stack:

```
        pop     ecx
        pop     ebx
        pop     eax
```

Although it would work fine to use the sequence:

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

to return from a function when using a frame pointer, in fact, as you can see from the full code from the function above, the actual sequence used is:

```
        leave
        ret
```

The **leave** instruction is precisely equivalent to the mov/pop sequence, but is shorter and more efficient. It is provided since we know that this particular mov/pop sequence will be very common, since it will occur every time we return from a function.

We will end this chapter by looking at the full code for the function we introduced at the start of the chapter that calls **alloca**:

```
_sum:
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        sub     esp, 32
        mov     eax, esp
        add     eax, 15
        shr     eax, 4
        sal     eax, 4
        mov     DWORD PTR [ebp-4], eax
        mov     DWORD PTR [ebp-8], 0
L2:
        cmp     DWORD PTR [ebp-8], 15
        ja      L3
```

```
        mov     eax, DWORD PTR [ebp-4]
        mov     edx, DWORD PTR [ebp-8]
        add     edx, eax
        mov     eax, DWORD PTR [ebp-8]
        mov     BYTE PTR [edx], al
        lea     eax, [ebp-8]
        inc     DWORD PTR [eax]
        jmp     L2
L3:
        mov     DWORD PTR [ebp-8], 0
L5:
        cmp     DWORD PTR [ebp-8], 15
        ja      L6
        mov     eax, DWORD PTR [ebp-4]
        add     eax, DWORD PTR [ebp-8]
        movsx   eax, BYTE PTR [eax]
        add     DWORD PTR [ebp+8], eax
        lea     eax, [ebp-8]
        inc     DWORD PTR [eax]
        jmp     L5
L6:
        mov     eax, DWORD PTR [ebp+8]
        leave
        ret
```

As we can see, the introduction of the frame pointer has solved the problem of alloca moving ESP, since we use the frame pointer for addressing everywhere, and the generated code just does not care about the exact value in ESP at any point.