More Arithmetic Operations

In this chapter, we are going to investigate further into how arithmetic operations are handled in assembly language. As usual, we will use gcc as our tutor.

Multiplication

Let’s try a small example to see what code is generated for multiplication operations:

```
  unsigned mult (unsigned a) {
    return a * 8;
  }
```

The generated assembly language for this function is:

```
_mult:
        mov     eax, DWORD PTR [esp+4]
        sal     eax, 3
        ret
```

Well that does not look exactly like a multiply instruction. What has happened here is that we chose a rather special multiplier. 8 is a power of 2, and the compiler has noticed this and replaced the multiply operation by a shift instruction. The **sal** instruction shifts bits left in a register. Zero bits enter at the right hand end, and the bits at the left hand end simply get discarded. Shifting left is equivalent to multiplying by powers of 2 (for an analogy consider that to multiply by 1000 in decimal, you just add three zeroes at the end.
Certainly it is nice to learn the shift instruction, but we were interested in learning how to multiply. Let’s try a number that is definitely not a power of 2:

```
  unsigned mult (unsigned a) {
    return a * 135;
  }
```

The generated assembly language for this new version is:

```
_mult:
        mov     edx, DWORD PTR [esp+4]
        mov     eax, edx
        sal     eax, 3
        add     eax, edx
        mov     edx, eax
        sal     edx, 4
        sub     edx, eax
        mov     eax, edx
        ret
```

Still no multiply instruction—what the heck is going on? Instead of a multiply we have a complicated series of shift, add and subtract instructions. The answer is that the compiler has noticed that **135 = 128+8–1**, and it is computing **a*135** by using the formula:

```
   a*128 = (a*8 + a)*16 – (a*8 + a)
```

That sounds horribly inefficient until you remember that multiplying by a power of 2 is cheap since we can use a shift. Trace out the instructions in the above code and you will see that it does indeed succeed in multiplying the argument by 135. Here are comments for the code that make this clear:

```
        mov     eax, edx        # copy value of a
        sal     eax, 3          # a * 8
        add     eax, edx        # a * 9
        mov     edx, eax        # copy a * 9
        sal     edx, 4          # a * 144
        sub     edx, eax        # a * 135
```

Well OK, this gets the right result, but what's going on here? Is there no multiply instruction on this machine. Let's try once more, forcing the compiler to handle a case where it does not know the value of the multiplier:

```
  unsigned mult (unsigned a, unsigned b) {
    return a * b;
  }
```

It is hard to see how gcc can play games with this one, and indeed when we compile this with gcc, we get:

```
_mult:
        mov     eax, DWORD PTR [esp+4]
        imul    eax, DWORD PTR [esp+8]
        ret
```

Finally—a multiply instruction. The **imul** instruction multiplies its two operands and places the product in the target register. So now we have an interesting question. If there is indeed the convenient imul instruction available, how come gcc did not use it for multiplying by 135 and generate a much simpler program in that case:

```
_mult:
        mov     eax, DWORD PTR [esp+4]
        imul    eax, 135
        ret
```

The interesting answer is that the above is a perfectly valid use of the imul instruction, and would have worked just fine. So how come gcc generated the long sequence of shifts and adds. The answer is that, surprisingly enough, the sequence of shifts and adds is faster. If you think about the way long multiplication works, it's a rather tedious process that goes digit by digit. It's not such a surprise to find that the imul instruction is rather slow and takes many clocks. The gcc compiler has lots of knowledge about speed trade offs like this, and it tries to generate the most efficient code, even if it is not the most obvious code. This is yet another reason to leave generation of assembly language to compilers! For interest let's try a really big constant multiplier:

```
  unsigned mult (unsigned a) {
    return a * 1277359;
  }
```

Now we get:

```
_mult:
        mov     ecx, DWORD PTR [esp+4]
        mov     edx, ecx
        sal     edx, 2
        add     edx, ecx
        mov     eax, edx
        sal     eax, 4
        sub     eax, edx
        mov     edx, eax
        sal     edx, 5
        add     eax, edx
        add     eax, eax
        add     eax, ecx
        mov     edx, eax
        sal     edx, 7
        add     eax, edx
        add     eax, eax
        add     eax, ecx
        ret
```

Surprisingly, gcc still figures out that this long sequence is faster than an **imul** instruction. It is actually quite a challenge to figure out exactly why the above sequence works, and if we were writing code by hand, it is hard to imagine we would take the trouble to do this transformation. Surely there must be some huge constant where it does not pay to use shifts and adds. How about:

```
 unsigned mult (unsigned a) {
   return a * 1277359401;
 }
```

Surely it cannot be worth generating the even longer sequence required by this mysterious constant? And sure enough, this time gcc finally gives up trying to be clever and figures out that an **imul** instruction is faster in this case:

```
_mult:
        mov     eax, DWORD PTR [esp+4]
        imul    eax, eax, 1277359401
        ret
```

Who would have thought that generating multiply instructions could be that complicated? Is this effort really worth while? The answer is definitely yes. Most multiplications by constants in real programs are of small constants, so the gain in figuring out how to avoid the generation of slow **imul** instructions is definitely worth it.