Pointer Manipulation and Indexing

In this chapter, we continue our exploration of how C is converted into assembly language by the compiler, focusing our attentions on pointers. Let's start with a simple C example:

```
unsigned x;
unsigned *p;
unsigned t;
void pops () {
   p = &x;
   t = *p;
   *p = t+x;
}
```

The corresponding assembler file is:

```
        .file   "p.c"
        .intel_syntax
        .text
.globl _pops
        .def    _pops;  .scl    2;      .type   32;     .
endef
_pops:
        mov     DWORD PTR _p, OFFSET FLAT:_x
        mov     eax, DWORD PTR _p
        mov     eax, DWORD PTR [eax]
        mov     DWORD PTR _t, eax
        mov     edx, DWORD PTR _p
        mov     eax, DWORD PTR _x
        add     eax, DWORD PTR _t
        mov     DWORD PTR [edx], eax
        ret
        .comm   _x, 16   # 4
        .comm   _p, 16   # 4
        .comm   _t, 16   # 4
```

Most of the lines starting with period are familiar from previous chapters. One exception is the **.comm line,** which is used to declare variables that are not initialized. The "comm" here stands for common, and reflects the fact that such variables can be declared in different units, and if labeled as extern, they all refer to the same location. The value of 16 is the alignment (higher than needed, but certainly high enough), and the 4 is the number of bytes (appropriate on the 32-bit machine we are using for both **int** and ***int** values.

Now let's look at the code of the function, which is what is interesting and new:

```
        mov     DWORD PTR _p, OFFSET FLAT:_x
```

This is the translation of the C statement **p = &x**. What is being moved to the pointer **p** is not the value of **x** (that could certainly not be done in one instruction anyway, since that would be a memory to memory move, and in any case it's not want we want. Rather it is the address of variable **x** that is moved there. This is a constant which is known by the time the program is loaded into memory, although as a programmer we certainly do not know it as we write the program. The somewhat mysterious **OFFSET FLAT:** incantation tells the assembler to figure out the (4-byte) address where the variable **x** will end up when the program is loaded. Even the assembler does not have all the information to figure this out, since a program may be in several pieces and the assembler does not know where each piece will go. It is up to the loader to figure this out, so in fact all the assembler does with the **OFFSET FLAT:** reference is to make a note in the object file and it is the loader that will finally fill in the right value in the generated instruction. This is one of the respects in which object code (which ends up in a **.o** file after assembly) is not pure machine code.

OK, so now that we have the proper address stored in variable **p**, the next C statement is **t = *p** which loads the value of ***p** and copies it to **t**. That takes the following three instructions:

```
        mov     eax, DWORD PTR _p
        mov     eax, DWORD PTR [eax]
        mov     DWORD PTR _t, eax
```

The first of these **mov** instructions moves the value of the pointer **p** into the **eax** register. Now **eax** contains the address of the variable **x** as set in the previous instruction. The next instruction is the one that dereferences the pointer. What we need now is the value stored in the memory location whose address is stored in the eax register. That's exactly what the notation **DWORD PTR [..]** does. The square brackets say "don't get the value of this register, instead get the value of the memory location whose address is in the register". The **DWORD PTR** part of this reference says get a four byte integer at that address.

Note: looking at the second **mov** instruction, it seems redundant to have to say **DWORD PTR**, since it is obvious that we must be getting four bytes, since we are loading the value into **eax** which is a four byte register. In fact, it *is* redundant in this case, and the gcc assembler is perfectly happy if you write the last instruction as:

```
        mov     eax, [eax]
```

However, the rules about when you can leave out **DWORD PTR** are a little tricky, and the gcc compiler always generates this phrase even if it is not necessary in some cases. You should probably follow the same convention when you are writing assembly language by hand.

The next C statement demonstrates the reverse operation of storing a value indirectly via a pointer. The statement ***p = t+x** generates the following four instructions:

```
        mov     edx, DWORD PTR _p
        mov     eax, DWORD PTR _x
        add     eax, DWORD PTR _t
        mov     DWORD PTR [edx], eax
```

The first move instruction retrieves the value of **p** into register **edx**. The second loads the value of **x** into **eax** and then the third instruction adds the value of **t**, so that the right hand side **t+x** is now in **eax**. Finally the fourth instruction shows that the **[…]** notation can be used as a destination as well as a source. This fourth instruction has the effect of storing the 4-byte value in **eax** into four bytes of memory at the address given in the **edx** register.

The only remaining instruction is the ret to return to the caller, which after the previous chapter should no longer be a mystery ☺

As usual, we are seeing pretty junky code in unoptimized mode, let's add the **–O2** optimization switch and see what happens. Now we get for the same C program, the following assembly language program generated:

```
        .file   "p.c"
        .intel_syntax
        .text
        .p2align 4,,15
.globl _pops
        .def    _pops;  .scl    2;      .type   32;     .
endef
_pops:
        mov     eax, DWORD PTR _x
        mov     DWORD PTR _p, OFFSET FLAT:_x
        mov     DWORD PTR _t, eax
        add     eax, eax
        mov     DWORD PTR _x, eax
        ret
        .comm   _x, 16   # 4
        .comm   _p, 16   # 4
        .comm   _t, 16   # 4
```

Hmm! That's a bit *too* clever. Trace through this code, it has the right effect, but completely eliminated the pointer references. Essentially it compiled the program as though the C function had been written:

```
   void pops () {
      p = &x;
      x = t+t;
   }
```

Indeed this has the same exact result. It is a good reminder that the optimizer is allowed to make any transformation it likes on a program, provided that the effect is the same. In this case, the compiler new that the pointer **p** pointed to **x**, so it could replace references to ***p** by **x** without changing the meaning of the program. It also shows why we often find it easier to read (and debug) unoptimized code, and in these chapters, where we are trying to figure out what the compiler generates for various C programs, the unoptimized code will usually be more instructive. That's especially true of little test programs we write to demonstrate features in the language.

An interesting aside is that this is also a problem with benchmarking programs. Suppose we modified our little test program here to be one that would supposedly test the speed of pointer accesses, something like:

```
unsigned x;
unsigned *p;
unsigned t;
unsigned j;
void pops () {
   p = &x;
   for (j = 1 ; j < 1000000000 ; j++) {
      t = *p;
      *p = t+x;
   }
}
```

If we run this benchmark program in unoptimized mode, that's a bit deceptive, since the quality of generated code is horrible in **–O0** mode, and would not reflect the real performance of an application where we would compile in **–O2** mode if we are interested in speed. On the other hand, if we run this benchmark program in **–O2** mode, the clever optimizer will remove the pointer operations completely, undermining the intention of the test. In some cases the optimizer can even remove a loop entirely if it can figure out that the loop is useless. Writing benchmarks so that they do not get defeated by optimizers is a difficult task. This task is made all the more difficult by the activities of compiler writers in putting more and more clever optimizations into their compilers, sometimes specifically aimed at breaking specific benchmark programs.

This basic **[..]** notation corresponding to the fundamental notion of using the value in a register as the address of the location in memory containing the data is actually all that we need to translate programs containing pointers. For example, here is a program which sums the elements of an array using pointer notation.

```
unsigned a[5] = {1,2,3,4,5};
unsigned sum = 0;
unsigned *ptr;
void csum () {
  for (ptr=a; ptr<&a[5]; ptr++) {
    sum += *ptr;
  }
}
```

The corresponding assembly language in unoptimized mode is

```
        .file   "s.c"
        .intel_syntax
```

```
.globl _a
        .data
        .align 4
_a:
        .long   1
        .long   2
        .long   3
        .long   4
        .long   5
.globl _sum
        .bss
        .align 4
_sum:
        .space 4
        .text
.globl _csum
        .def    _csum;  .scl    2;      .type   32;     .
endef
_csum:
        mov     DWORD PTR _ptr, OFFSET FLAT:_a
L2:
        cmp     DWORD PTR _ptr, OFFSET FLAT:_a+20
        jae     L3
        mov     eax, DWORD PTR _ptr
        mov     eax, DWORD PTR [eax]
        add     DWORD PTR _sum, eax
        add     DWORD PTR _ptr, 4
        jmp     L2
L3:
        ret
        .comm   _ptr, 16         # 4
```

Let's look in detail at the generated code for the function. The first mov instruction sets ptr to point to a, and is familiar from our previous example. L2 marks the start of the loop. The two instructions at the start of the loop are worth looking at:

```
        cmp     DWORD PTR _ptr, OFFSET FLAT:_a+20
        jae     L3
```

Here we are comparing the value of the pointer, which is just a 32-bit unsigned integer, with the address **a+20**. Note that the plus here does not correspond to the generation of an add instruction. This addition is done by the assembler/loader working together. The reference is to an address computed by adding **20** to whatever address is assigned to the array **a**. When the program is finally loaded the actual constant in the instruction will be the four byte unsigned integer value obtained by doing this addition. That's what we want since **&a[5]** is the address just past the end of the array, and since each of the five elements occupies four bytes, we add **20 = 5*4**. Since these are simply 32-bit unsigned values being compared, the normal **jae** (jump above or equal) instruction (which is testing the **ZF** and

CF flags set by the **cmp** instruction in an appropriate manner) can be used to terminate the loop after the pointer has traveled through the array.

The next three instructions perform the addition of the next element to the sum. The final instruction in the body of the loop:

```
add     DWORD PTR _ptr, 4
```

implements the increment of the pointer (**ptr++**). Since the elements of our array are four byte unsigned integers, 4 is the right constant to add to get to the next element.

Just for fun, let's see what the function code looks like if we optimize:

```
_csum:
        mov     ecx, OFFSET FLAT:_a
        mov     DWORD PTR _ptr, ecx
        cmp     ecx, OFFSET FLAT:_a+20
        jae     L7
        mov     eax, DWORD PTR _sum
        .p2align 4,,7
L5:
        mov     edx, DWORD PTR [ecx]
        add     eax, edx
        lea     edx, [ecx+4]
        mov     DWORD PTR _sum, eax
        mov     ecx, edx
        cmp     edx, OFFSET FLAT:_a+20
        jb      L5
        mov     DWORD PTR _ptr, edx
L7:
        ret
```

Ha! gcc is not that clever after all. If it was really clever, it would have figured out that we did not need a loop at all, and done the addition to get the value of 15 for sum. Essentially it could have replaced the function by the equivalent C function:

```
   void csum () {
     ptr = &a[5];
     sum = 20;
   }
```

That would have been a perfectly valid optimization, which would of course have run much faster, but that optimization is (so far) beyond the reach of gcc. It's by no means beyond the reach of compilers in general, and a technique known as symbolic interpretation where you essentially run as much of the program as you can at compile time would have achieved this optimization. This is another caution to anyone trying to write performance benchmarks for compilers!

As it is, gcc certainly improved the code by removing memory loads, and reducing the number of jumps in the loop from two to one. The optimized code introduces one more instruction:

```
        lea     edx, [ecx+4]
```

There are two interesting things about this line. First the notation **[ecx+4]** shows that indexing can be more powerful that we saw in our simple examples. For example, suppose we write:

```
        mov     eax, [ebx+4]
```

The meaning of this is that the register **eax** should be loaded not with the memory location contained in **ebx**, but rather the memory location four bytes higher. The effect is similar to the two instruction sequence:

```
        add     ebx, 4
        mov     eax, [ebx]
```

except that when we use the notation **[ebx+4]** the value in **ebx** is not changed, and we avoid an addition instruction. The addition here is done by the machine as part of the execution of the **mov** instruction. No separate addition instruction is required. The methods of referencing memory in an instruction are called addressing modes. So far we have seen three addressing modes that the ia32 architecture implements:

direct memory reference (e.g. DWORD PTR _a)
register indirect (e.g. [EBX])
register indirect with offset (e.g. [EBX+4])

Going back to the new instruction:

```
        lea     edx, [ecx+4]
```

we have not only a new addressing mode, but also a new opcode, namely **lea**.
The mnemonic **lea** stands for load effective address. For both the **mov** and **lea** instructions, the right operand can use one of the three addressing modes to address memory. In the case of the **mov** instruction, the result is to access this location in memory and reference the value stored there. In the case of an **lea** instruction, there is no memory reference, instead the *address itself* is loaded into register.

For the simple addressing modes, the lea instruction is not that interesting since we have other ways of doing things. These two instructions have the same effect:

```
        lea     edx, DWORD PTR _a
        mov     edx, OFFSET FLAT: _a
```

as do these two instructions:

```
        lea     edx, [ebx]
        mov     edx, ebx
```

However, for more complicated addressing modes (and we will see some even more interesting ones later in this chapter), lea is definitely interesting. The lea instruction we are studying here:

```
        lea     edx, [ecx+4]
```

would otherwise take two instructions

```
        mov     edx, ecx
        add     edx, 4
```

Furthermore, the lea instruction does not clobber the flags, which is also useful in some code sequences (though this is not significant in this particular case).

Addressing members of a struct

The *register plus offset indirect* addressing mode is implemented on nearly all architectures. One reason for this is that it is particularly useful in addressing components of a struct. Consider the following linked list example:

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

  struct node *h;
  unsigned val;
  unsigned result;

  void find () {
    while (h) {
      if (h->val == val) result = 1;
      else h = h->next;
    }
    result = 0;
  }
```

In this example, we declare a simple node structure with a forward pointer and a value field containing an unsigned integer. The function **find** searches the linked list , and sets **result** to zero (false) if the element is not found, and to one (true) if the element is found. Now let's look at the generated assembly code for the function.

```
_find:
L2:
        cmp     DWORD PTR _h, 0
        je      L3
        mov     eax, DWORD PTR _h
        mov     eax, DWORD PTR [eax]
        cmp     eax, DWORD PTR _val
        jne     L4
        mov     DWORD PTR _result, 1
        jmp     L5
L4:
        mov     eax, DWORD PTR _h
        mov     eax, DWORD PTR [eax+4]
        mov     DWORD PTR _h, eax
L5:
        jmp     L2
L3:
        mov     DWORD PTR _result, 0
        ret
```

The interesting instructions are the ones which access a field of the **struct h**. The fields of a **struct** are simply laid out in memory in sequence, so a node has 8 bytes. The first four bytes are the **val** field, and the last four bytes are the **next** field. The loading of the **val** field is accomplished with the instructions:

```
        mov     eax, DWORD PTR _h
        mov     eax, DWORD PTR [eax]
```

The first **mov** instruction loads the current pointer value into register **eax**. Then the second **mov** instruction loads the word referenced by this address. Since we have defined **val** to be the first field, a pointer to the **struct** points right at the val **field**. To access the next field, we again have two instructions:

```
        mov     eax, DWORD PTR _h
        mov     eax, DWORD PTR [eax+4]
```

and here we see the value of the **register+offset** indirect addressing mode. By adding the **+4** into the indexing reference the second **mov** instruction here loads the word four bytes past the start of the **struct**, which contains the **next** field. Since **structs** are a very heavily used data structure, the ability to access any field without using extra addressing instructions is very valuable. Note that this approach to addressing **struct** fields requires that all fields of the **struct** have a fixed length which is known at compile time. That's always the case in C, since all declared datatypes have a fixed length.

More Complex Addressing Modes

Earlier in this chapter we mentioned that there are some more complicated addressing modes in the ia32 architecture. Let's see if we can get gcc to demonstrate them to us. Consider the following C program:

```
   char *p;
   unsigned q;
   char result;
   void sindex () {
      result = p[q];
   }
```

Here is the resulting unoptimized assembly code from gcc for the function sindex which indexes into the string pointed to by p, obtains the character with index q (since strings are indexed from 0, this is the q+1'th character in the string), and stores it in result:

```
_sindex:
        mov     eax, DWORD PTR _p
        add     eax, DWORD PTR _q
        mov     al, BYTE PTR [eax]
        mov     BYTE PTR _result, al
        ret
```

That's too bad, gcc is too stupid to illustrate the utility of an addressing mode we have not seen yet, but wait, let's give it another chance, here is the same code generated in optimized mode with the –**O2** switch set:

```
        mov     eax, DWORD PTR _q
        mov     edx, DWORD PTR _p
        mov     al, BYTE PTR [eax+edx]
        mov     BYTE PTR _result, al
        ret
```

That's better! Now we can see the new addressing mode in the third instruction. This is called **register+register indirect** mode. The meaning of the third **mov** instruction here is to compute the address by adding the contents of the **eax** and **edx** registers, and then to load the byte that is at the memory address corresponding to this sum. The addition of the two registers here is part of the execution of the **mov** instruction, and so we avoid the need for a separate add instruction.

At first sight, the optimized code does not seem that much of an improvement. It still has five instructions, with three memory references. However, on close inspection there is an important difference. The five instructions in the unoptimized case have to be done one at a time in sequence (for example, you can't add to **eax** in the second instruction until the first instruction has loaded the value into **eax**). On the other hand in the optimized code, the first two **mov** instructions are completely independent, and indeed modern implementations of the ia32 architecture will execute the first two **mov** instructions in parallel, thus saving time.

OK, let's get one step more complicated by modifying this little indexing program to index into an array of unsigned words:

```
   unsigned *p;
   unsigned q;
   unsigned result;
   void sindex () {
        result = p[q];
    }
```

The difference here is that the subscript q now counts words instead of bytes. That means that we have to multiply q by 4 before adding it to the address p. This time, we will go straight to the optimized code generated by gcc:

```
_sindex:
        mov     eax, DWORD PTR _q
        mov     edx, DWORD PTR _p
        mov     eax, DWORD PTR [edx+eax*4]
        mov     DWORD PTR _result, eax
        ret
```

Interesting, no additional instructions! Instead we see the form **[edx+eax*4]** in the third instruction, which multiplies the value in eax (the subscript value) by 4 before adding it to edx. Both the multiplication (which is really just a 2-bit shift so it is cheap, really almost free) and the addition are done as part of the execution of the third mov instruction so they are essentially free. This is called **register+register scaled indirect** mode, and it is designed specifically for scaled references to array elements as in this example. The only allowed scaling factors are 2,4, or 8, which correspond to simple shifts and also to the most typical cases of array values.

Note that these addressing modes are not essential, in that we could manage without them if we had to by using explicit shift or add instructions to compute the required address. Indeed many of the modern **RISC** (reduced instruction set computers) have a very limited set of addressing modes. The ia32 is considered a **CISC** (complex instruction set computer), and one of the characteristics that makes it more complex is the complex addressing modes. We don't call these complex because they are hard to understand (though that may be the case), but rather because more is going on in each instruction. Part of the idea of **RISC** design is to streamline instructions so that less happens in each instruction. The downside is that we may need more instructions. The upside is that we can build the processor to execute **RISC** type instructions much faster. The net result may be faster execution even if we have to execute more of the simple instructions.

Summary of Addressing Modes

The following is a summary of the addressing modes supported by the ia32. This list has all the cases we have seen so far, plus a few combinations that are also allowed (we could probably find gcc programs to demonstrate these additional combinations, but that's probably not worth the effort).

Mode	**Example**
Direct address	`DWORD PTR _a`
Register indirect	`[EBX]`
Register indirect with offset	`[EBX + 4]`

Register plus register indirect	`[EAX + EBX]`
Register plus register indirect with offset	`[EAX + EBX + 12]`
Register indirect with scaling	`[4 * EAX]`
Register indirect with scaling and offset	`[4 * EAX + 8]`
Register + register indirect with scaling	`[EAX + 8 * EBX]`
Register + register indirect with scaling and offset	`[EAX + 2 * ECX + 3]`

Remember when you read this chart that all the operations in the example are performed as part of the addressing computation in a single instruction. Looking at the last example, we see that this can require two additions and a shift, in addition to the actual operation to be performed. No wonder they call the ia32 a complex instruction set computer!

The LEA Instruction Revisited

Earlier on, we looked at the **lea** (load effective address) instruction which had the form:

```
lea     edx, {memory address}
```

Here {memory address} can be any of the possibilities in the chart we just gave. The result is to compute the address and place the address into the target register (**edx** in the above example). The **lea** instruction does not actually reference memory, since only the address is stored, not the contents of the address in memory. This means that **lea** can be used for all kinds of interesting operations:

```
lea     edx, [eax+ebx]        # edx = eax + ebx
lea     edx, [eax+4]          # edx = eax + 4
lea     edx, [eax+4*ebx]      # edx = eax + 4*ebx
lea     edx, [eax+4*ebx+2]    # edx = eax + 4*ebx + 2
```

Here studying these examples is a good way of making sure you understand how the addressing modes work. Furthermore, particularly in optimized mode, gcc makes a lot of use of these instructions, since it is very nice to be able to do all this work in a single instruction.