The Assignment

To hand-optimize part of the Insertion Sort algorithm for the SPARC architecture.

The Analysis

First thing I did was compare the output from gcc -S and gcc -S -O3 to see how they stacked up. Without optimizations there are 57 lines and with -O3 there are 31, already quite an improvement. It does not use save/restore, as required. Additionally, -O3 never calls the seperate insert() function, but instead inlines it directly into isort(). Then I tried running CPU and architecture specific optimizations with gcc -S -O3 -mcpu=ultrasparc, but that only changed the instruction names a bit without changing the instruction count or speed at all.

Analyzing isort() for unused registers was fairly simple. It uses %o0-o3 and none of %g. It loads its own data into the registers both before and after calling insert(), so practically nothing insert() does can affect isort()'s correct behavior. I did a quick test to prove this by throwing in -O3's insert() into the unoptimized assembly, and it worked without any complaints. -O3's output is in Source Listing 2.

The completely unoptimized insert() is quite horrible. The main loop, consisting of .LL7 and .LL9, uses 8 load and 2 store operators, totalling 10 memory accesses. A rolling analysis follows...

```
  insert:
      !#PROLOGUE# 0
      save  %sp, -112, %sp
      !#PROLOGUE# 1
```

Here it saves the input variables, i and key, for later use.

```
      st    %i0, [%fp+68]
      st    %i1, [%fp+72]
```

Start of the main loop. LL7 handles the conditions.

```
  .LL7:
```

Now load the previously saved input variable, i.

```
      ld    [%fp+68], %o0
```

Compare if i is <0, and branch to the end if so.

```
      cmp   %o0, 0
      bl    .LL10
      nop
```

Create an offset to num[i], then load it.

```
      sethi %hi(num), %o1
      or    %o1, %lo(num), %o0
      ld    [%fp+68], %o1
      mov   %o1, %o2
      sll   %o2, 2, %o1
```

```
        ld      [%o0+%o1], %o0
```

Load key, and compare to num[i]. If num[i]>key, branch to .LL9, else go to end.

```
        ld      [%fp+72], %o1
        cmp     %o0, %o1
        bg      .LL9
        nop
        b       .LL10
         nop
```

Welcome to Superfluous Branch Tags. This week's special:

```
  .LL10:
        b       .LL8
         nop
```

Continuation of the main loop. LL9 handles the assignments.

```
  .LL9:
```

Create an offset to num[i+1], then create an offset to num[i], then load num[i] and store in num[i+1].

```
        sethi %hi(num), %o1
        or      %o1, %lo(num), %o0
        ld      [%fp+68], %o2
        add     %o2, 1, %o1
        mov     %o1, %o2
        sll     %o2, 2, %o1
        sethi %hi(num), %o3
        or      %o3, %lo(num), %o2
        ld      [%fp+68], %o3
        mov     %o3, %o4
        sll     %o4, 2, %o3
        ld      [%o2+%o3], %o2
        st      %o2, [%o0+%o1]
```

Reload i, decrement i, save i.

```
        ld      [%fp+68], %o0
        add     %o0, -1, %o1
        st      %o1, [%fp+68]
```

Branch back to loop condition checks.

```
        b       .LL7
         nop
```

Post-loop actions are done in LL8.

```
  .LL8:
```

Create an offset to num[i+1]

```
        sethi %hi(num), %o1
        or      %o1, %lo(num), %o0
        ld      [%fp+68], %o2
        add     %o2, 1, %o1
        mov     %o1, %o2
        sll     %o2, 2, %o1
```

Load key, save key in num[i+1].

```
        ld      [%fp+72], %o2
        st      %o2, [%o0+%o1]
```

Done, so return and restore.

```
  .LL6:
        ret
        restore
```

All in all, this is a by-the-numbers what needs to be done and how can it be explicitly done step by step implementation. Nothing is incorrect or dangerous, but it is very slow. In contrast, -O3's main loop uses only 1 load and 1 store, but even that is not as sleek as possible.

The Implementation (see: Source Listing 1)

Looking at -O3 to see what can be optimized, not much really stands out as being unnecessary. The only thing I noticed was that it builds an offset from i by i<<2 (same as i*4), and then uses that offset for all subsequent loads and saves. Even so, it is still keeping i itself up to date. So, first thing was to only use i once to create the initial offset, then work with that. This shaved 1 instruction (`addcc %o0, -1, %o0`) off the main loop (.LL53).

Next thing I noticed was that the registers %o2, %o3, and %o4 served much of the same purpose in building offsets, but was being maintained seperately. The instructions "`add %o2, -4, %o2`" and "`add %o3, -4, %o3`" in .LL53 had to be redundant. Then I got stuck trying to clean it all up, and instead I looked around for a later version of gcc to see how that would do it.

Source Listing 3 has the insert() from v3.3.3 of gcc -O3 -mcpu=i686, which has a simple and clean 6 instruction main loop (.L58). One instruction better than what I had so far. It is different from SPARC in that i686 doesn't need to pre-calculate the offset into num[], so it can keep its i variable around in %edx for accessing the num[] array directly. Whether it actually is a single instruction behind the scenes I don't know, but it looks much cleaner.

Then I combined my single offset calculation with the i686 main loop ported to SPARC Assembly, and then had what you can see in Source Listing 1.

Runtimes:

Best case of i=0: 6 instructions

Better case of num[i]<=key: 12 instructions

All other cases: (i*6)+13 instructions

Used registers:

Register	***Purpose***
%o0	Holds i. Used twice, both reads.
%o1	Holds key. Used twice, both reads.
%o2	Holds i*4 for offset into num[], and gets gradually decremented by 4. Used as num[%o3+%o2].
%o3	Holds the base offset of num[0].
%o4	Holds the base offset of num[1].
%g2	Temporary register for holding values from num[] that are to be moved.

I also looked at isort() to see if I could reuse something from it's environment to speed things up, but there's only %o2 holding offset to num[j] and %o3 holding j. They might be useful for initialization, but I truly doubt the 6 instruction main loop can be any faster, regardless of how many registers are available or can be inherited from the caller.

The Testing

Not really sure what can be tested here except showing it gives the right output:

```
bash-2.03$ gcc insertion_sort.normal.s
bash-2.03$ ./a.out
Input:
 10, 18,-19, 18, 12,  8,  3, 42, 15, 20,  3, 17, 25, 11, 23, 12,  0, -8,-13,  9,
Output:
-19,-13, -8,  0,  3,  3,  8,  9, 10, 11, 12, 12, 15, 17, 18, 18, 20, 23, 25, 42,
```

The Conclusion

Beat gcc -O3's main loop by 2 instructions (a whopping 25%), which is as good as it gets.

- Tino Didriksen

Source Listing 1: The hand-optimized SPARC Assembly insert()

```
insert:
       !#PROLOGUE# 0
       !#PROLOGUE# 1
       cmp    %o0, 0
       bl     .LL9
       sethi  %hi(num), %o3
       or     %o3, %lo(num), %o3
       sll    %o0, 2, %o2
       ld     [%o3+%o2], %g2
       cmp    %g2, %o1
       ble    .LL9
       add    %o3, 4, %o4
.LL53:
       st     %g2, [%o4+%o2]
       addcc  %o2, -4, %o2
       bneg   .LL9
       ld     [%o3+%o2], %g2
       cmp    %g2, %o1
       bg     .LL53
.LL9:
       nop
       add    %o2, 4, %o2
       retl
       st     %o1, [%o3+%o2]
```

Source Listing 2: The insert() by gcc -O3

```
insert:
       !#PROLOGUE# 0
       !#PROLOGUE# 1
       cmp    %o0, 0
       bl     .LL9
       sethi  %hi(num), %g3
       or     %g3, %lo(num), %o3
       sll    %o0, 2, %o2
       ld     [%o3+%o2], %g2
       cmp    %g2, %o1
       ble    .LL9
       add    %o2, 4, %g2
       mov    %o3, %o4
       add    %g2, %o4, %o3
       ld     [%o2+%o4], %g2
       addcc  %o0, -1, %o0
.LL53:
       st     %g2, [%o3]
       add    %o2, -4, %o2
       bneg   .LL9
       add    %o3, -4, %o3
       ld     [%o2+%o4], %g2
       cmp    %g2, %o1
       bg,a   .LL53
       addcc  %o0, -1, %o0
.LL9:
       add    %o0, 1, %g2
       or     %g3, %lo(num), %g3
       sll    %g2, 2, %g2
       retl
       st     %o1, [%g3+%g2]
```

Source Listing 3: The insert() by v3.3.3 gcc -O3 -mcpu=i686

```
insert:
       pushl  %ebp
       movl   %esp, %ebp
       movl   8(%ebp), %edx
       movl   12(%ebp), %ecx
       testl  %edx, %edx
       js     .L55
       movl   num(,%edx,4), %eax
       cmpl   %ecx, %eax
       jle    .L55
       .p2align 4,,15
.L58:
       movl   %eax, num+4(,%edx,4)
       decl   %edx
       js     .L55
       movl   num(,%edx,4), %eax
       cmpl   %ecx, %eax
       jg     .L58
.L55:
       popl   %ebp
       movl   %ecx, num+4(,%edx,4)
       ret
```