

## Prac 3: Embedding Assembly and Introduction to Loop Unrolling

Last Update: 16/03/2007 11:26:00 AM

Author: [Simon Leung](#)

---

### Preparations

You will need to have a good understanding of Prac 1 and 2 in order to benefit the most from this prac.

Don't forget you need to use the very useful reference document "[The SPARC Architecture Manual, version 9](#)" (also [available internally](#)) to look up information. Use it smartly and carefully, looking at the relevant sections and use the search function of the PDF reader to help you navigate in this book.

### Setup

For this prac (and the later ones), we will use compilers from SUN (Sun Workshop 6). You will need to setup your path to the SUN compiler installed in agave, so make sure \$PATH has these entries:

```
/opt/SUNWspro/bin:/usr/ccs/bin:/opt/local/bin:/usr/bin\  
:/sbin:/usr/sbin:/opt/local/sbin:/usr/openwin/bin.
```

For example, the commands:

- `PATH=/opt/SUNWspro/bin:/usr/ccs/bin:/opt/local/bin\  
:/usr/bin:/sbin:/usr/sbin:/opt/local/sbin:/usr/openwin/bin`
- `export PATH`

will achieve the desired result.

### Procedure

#### Task 1: A Quick Word on Architecture Dependence

We have used the Sun compiler (cc) and the GNU C compiler (gcc) in the last two pracs to investigate the correlations between C code constructs and the assembly code generated. The assembly code ran without any problems. However, the instructions generated are from the basic, earliest instruction set for SPARC. How do we take advantage of the newer architecture?

Copy your "Hello World" with parameter code into another file, say `hello3.c`. Now compile this file with an additional flag, `-xarch=v9` to indicate that the assembly is targeted at the SPARC V9 architecture. Display the two assembly files, one with the target architecture flag and one without, side by side. Answer the following questions:

1. Is the overall program flow any different between the two assembly listings?
2. Identify all the "new" instructions used by the new assembly listing. Which ones look familiar to you? Using the Architecture Manual, find out what those instructions mean.
3. What is the purpose and use of the `%icc` register? Is it specific to V9?
4. Why is there a need for two `sethi`, three `or` and one `sllx` instructions in certain parts of the assembly code?
5. Which one is faster in your opinion and why? How will you test your predictions?

#### Task 2: Inline Assembly with Sun Compiler

Recall in the last prac, we have used the syntax

```
__asm( " " );
```

to insert assembly instructions directly into the C code. The problem is that we cannot predict which registers will be assigned by the compiler. As it turns out, Sun recommends that everyone should do all coding in high-level languages, and does not support this `__asm` directive as extensively as GCC. Sun provides another construct, called "inline templates", to allow programmers to embed assembly code via function calls. Going back to our addition example from prac 2, let's illustrate how this inline template works.

Recall the following code (call it `inline2.c`), but since we are going to write the addition function in assembly, we will need to declare the function first:

```
#include <stdio.h>  
  
extern int add3(int a, int b, int c);  
  
int main()
```

```

{ int i;
  i=add3(1,3,5);
  printf("The sum is %d!\n",i);
  return (0);
}

```

This is the restriction placed on us by the Sun Compiler. The assembly code should be placed within regular function calls, and in this case, add3.

Now using your favourite text editor, enter the following code as inline2.i1:

```

/*Inline insert for inline3.c*/
.inline add3,12
  add %o0,%o1,%o0
  add %o0,%o2,%o0
.end

```

As you have probably guessed, following the .inline directive is the name of the function we declared earlier. The number (12) indicates that the number of bytes of the parameters of this function (3 integers, each 4 bytes long). The parameters (integers 1, 3 and 5 designated for a, b and c) are passed on registers %o0, %o1, %o2 respectively, and the result is stored in %o0.

To compile this program, you need to run `cc -s inline2.c inline2.i1`. Take a look at the position where the inline assembly is inserted.

Be aware that this is really NOT a function call - there is no return statement! The assembly code will be inserted directly in the position where the "function" is referred, so this is closer to macros rather than function calls.

If you need to pass more than 6 parameters, the remaining ones can be found on the stack. For 32-bit codes, [%sp+0x5c] is the next parameter, for 64-bit codes, [%sp+0x8af] is the next one. Another restriction placed by Sun is that you should only alter registers %o0 to %o5, and %f0 to %f31. Other registers should be left alone, or properly restored before the inline code is finished.

What if the parameters are floating point numbers? In such cases, it will depend on code length. For 32-bit codes, due to compatibility reasons, the floating point number parameters will be stored in the integer registers as before. For 64-bit codes, since all of their architecture has native 64-bit floating point support, the parameters will be stored in the floating point registers directly. As a result, if you want to use the floating point operations and registers for 32-bit codes, you must store the content of the %ox (which has a fp number) to memory stack, and load that content to the floating point registers %fx.

- Write a C program to calculate the product of two floating point numbers, and use the floating point multiplication assembly instruction as the inline function, on 32-bit codes.

### Task 3: Inline Assembly using GCC Compiler

GCC supports inline assembly functions using a different syntax (as mentioned before, this is different for each compiler and assembler). What gcc allows us to do is specify C operands within our assembly code. This syntax also leaves the compiler to allocate registers, something the compiler is usually better at than humans! In order to understand the syntax, the following template may be useful:

```

asm ( assembler template
      : output operands          /* optional */
      : input operands          /* optional */
      : list of clobbered registers /* optional */
      );

```

The first line specifies the instruction(s) to be used, while the following lines specify the operands to be used. The final line specifies the "clobbered registers" which are those registers which have had their contents modified. It is then up to the compiler to restore the values appropriately (if required elsewhere in the code). The ':'s are used to separate these sections. Operands (parameters) are specified using %, and do not confuse them with the register names!

Enter the following code in a file gccinline.c:

```

#include <stdio.h>

int main()
{
  int i;
  int x,y,z;
  x=34;y=13;z=25;

  asm("add %2, %1, %0;"
      : "=r" (x)
      : "r" (y), "r" (z)
      );

  printf("The sum is %d!\n",x);
  return (0);
}

```

Compile the code by typing `gcc inlinegcc.c` and run the compiled code (you may want to open another xterm for this part). Examine the output and the assembly listing, answer the following questions:

- In the asm syntax, which variable(s) is/are in the output operand section? Which variable(s) is/are in the input operand section?
- Obviously, %0, %1, %2 are the parameters passed into the assembly instruction. Which variable, x, y & z are %0, %1 & %2 corresponding to?
- What are the registers that gcc has chosen for the %0, %1 & %2 parameters?

"r" is an example of a *constraint* that we pass onto the compiler. Here, "r" tells the compiler to use any of the available registers to store the operands. For the "=r", it is a constraint which tells the compiler that this register is write-only.

- Why is the "=r" constraint only applicable to x, and not y or z?
- Try changing the order of the operands to make sure you understand how it works.

In the code section above, there is no clobbered registers list. Now add the line : "%0" so that the asm section looks like:

```
asm("add %2, %1, %0;"
    : "=r" (x)
    : "r" (y), "r" (z)
    : "%0"
);
```

How is the assembly listing changed? What does "clobbered" list mean? Can you now use the register %0 in your asm section without causing problems?

This first example only uses a single assembly instruction. In order for inline assembly to be useful, we need to be able to use multiple instructions within one asm block. In order to do this, something like the following should be used:

```
#include <stdio.h>

int main()
{
    int i;
    int x,y,z;
    i = 0;
    x=34;y=13;z=25;
    asm("\n
        add %2, %1, %0; \t
        mov %0, %%0; \t
        add %3, %%0, %0; \n"
        : "=r" (i)
        : "r" (x), "r" (y), "r" (z)
        : "%0"
    );
    printf("X is %d\nY is %d\nZ is %d\nI is %d\n",x,y,z,i);
    return (0);
}
```

Run this code, working out what happens within the assembly block. Note that:

1. %%0 tells the compiler that we want to use register %0 directly in the assembly code, so it will not confuse with the parameter syntax;
2. the operands for each instruction are the opposite way around from what we have seen in MIPS - the output register is the last one in the list.

- Would this code compile using the Sun Compiler cc?

You might like to refer to the gcc manual (info gcc) or online at [http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc\\_4.html#SEC933ore](http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_4.html#SEC933ore) for more information on the syntax of gcc assembly. There are many more options available to you - some of them might prove useful later in semester.

- Translate your C program from the previous section to now use gcc inline assembly instead of Sun CC syntax. Examine the output code produced by the compiler. How is it different from that produced by the Sun compiler?

#### Task 4: Introduction to Loop Unrolling

Using the time measuring code from last prac, find out the time needed to complete the following simple loop:

```
for (i=100000; i>0; i=i-1)
    x[i] = x[i] + 5;
```

Of course, declare the necessary variables and initialise the values in the array x to some value first. This simple loop is used to add a scalar value (5) to a vector (x).

Now replace the loop with the following code which is functionally equivalent:

```
for (i=100000; i>0; i=i-10)
{
    x[i] = x[i] + 5;
    x[i-1] = x[i-1] + 5;
    x[i-2] = x[i-2] + 5;
    x[i-3] = x[i-3] + 5;
    x[i-4] = x[i-4] + 5;
    x[i-5] = x[i-5] + 5;
    x[i-6] = x[i-6] + 5;
    x[i-7] = x[i-7] + 5;
    x[i-8] = x[i-8] + 5;
    x[i-9] = x[i-9] + 5;
}
```

How much time does it take to complete the new code?

Perform an instruction count (as shown in the last prac) and compare the total number of instructions executed for these two loops.

Examine the answers of the questions above, and look at the assembly listing, answer the following questions:

- In your opinion, why does the second loop execute faster than the first one?
- Is the assembly code produced by the compiler optimal? Can you improve it? What would you change in the assembly code?
- Under what conditions is this technique of "unrolling" the loop possible?

## Summary

In this prac, we have covered:

- compiler architecture flag;
- inline assembly in Sun Compiler;
- inline assembly in GCC;
- Introduction to loop unrolling.

## Suggested Exercises

You should reinforce your understanding of these concepts by modifying the code and try to make some mistakes and redundant steps. As a suggestion, experiment with writing a slightly more complex assembly function and embedded in your code. This is most interesting when you combined with loop unrolling. Study how the arrays was accessed in the unrolled code in the last section, and write an inline assembly function to run the loop to see how much more time you can save on the loop

---