

CSSE3001/7000 2006

Prac 2: More SPARC Assembly

Last Update: 11/03/2007 03:02 PM

Author: [Simon Leung](#)

Preparations

You will need to have a good understanding of Prac 1 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: Arithmetic Operations

Using your favourite text editor, create a file called `perimeter.c` with the following code:

```
#include <stdio.h>

int length;
int width;
int perimeter;

int main()
{
    printf("Please enter length: ");
    scanf("%d", &length);
    printf("\nPlease enter width: ");
    scanf("%d", &width);
    perimeter = (length + width) * 2;
    printf("\nThe perimeter is %d.\n", perimeter);
    return (0);
}
```

Compile this code to observe the assembly output. Answer the following questions:

1. Which assembly instruction performs the addition?
2. Which assembly instruction performs the multiplication by 2?
3. Modify the C code above, or otherwise, obtain the assembly instruction for multiplication in general. What are the possible reasons that the compiler chooses answer to Q2 instead of this instruction for multiplication by 2?
4. Investigate the order of operators, ie., how does the program handle without the brackets for addition?

Task 2: Loops

Using your favourite text editor, create a file called `sum.c` with the following code:

```
#include <stdio.h>

int i;
int sum=0;

int main()
{
    for (i=1; i<10; i++)
    {
        sum = sum + i;
    }
}
```

```

    printf("Sum is %d\n", sum);
    return (0);
}

```

Compile this code to observe the assembly output. Notice the use of `cmp` and `b1` to perform the loop operations. Answer the following questions:

1. Which register is used to keep track of the index variable `i`?
2. How will the assembly code be different if the condition was `i<=10`? Confirm your answer with experiment.
3. What is the reason to have the `bge` instruction before entering the loop?
4. How are other loops constructed in assembly? Try the loops introduced in lecture (while loops, do ... while loops)

Task 3: (Type) Casts

Using your favourite text editor, create a file called `magic.c` with the following code:

```

#include <stdio.h>

int i=2;
float j;

int main()
{
    printf("The integer number is %d\n", i);
    j=(float)i;
    printf("The floating point number is %f\n", j);
    return (0);
}

```

Compile this code to observe the assembly output. Answer the following questions:

1. Do the microprocessor simply copy directly between integer registers and floating point registers?
2. Identify the assembly instructions associated with floating point numbers. What do they do?
3. How costly is type casting? When will it affect the performance of the program?

Task 4: Some Performance Measurements and Indicators

Recall from lectures, total time taken is a good indicator of the overall performance of the program. This part will show you a couple of methods in obtaining the execution time.

Using your favourite text editor, modify your `sum.c` file (rename to `sum2.c`) to:

```
#include <stdio.h>

int i;
int sum=0;

int main()
{
    for (i=1; i<=65535; i++)
    {
        sum = sum + i;
    }
    printf("Sum is %d\n", sum);
    return (0);
}
```

(Yes, this is a very silly way of calculating a sum, but for illustration purpose it isn't so bad) Compile the program into the binary. You can use the `time` function to see how long it takes to run this program: `time ./sum2`

If you would like to measure the time taken for certain part(s) of your program, take a look at the following code:

```
#include <stdio.h>
#include <sys/time.h>

int num;
long useconds;
struct timeval start_time; /*special pre-defined structure for times in microsecond and seconds*/
struct timeval end_time;

int main()
{
    printf("Please enter a number: ");
    gettimeofday(&start_time, NULL); /*Get the start time*/
    scanf("%d", &num);
    gettimeofday(&end_time, NULL); /*Get the end time*/
    useconds = (end_time.tv_sec - start_time.tv_sec)*1000000 + (end_time.tv_usec - start_time.tv_usec);
    printf("You have entered %d.\n",num);
    printf("You took %d microseconds to think of this number.\n",useconds);
    return (0);
}
```

When you run this program, the time needed to complete the `scanf` command is recorded, including the time it took for the user to respond. Examine closely the syntax to calculate the microseconds and seconds.

1. Using the syntax help in this code and modify the `SUM` program above. How much time does it need to complete the loop (ONLY) to calculate the sum, excluding the initialisations and the `printf` statements?

2. Run the modify code at least 10 times. Is the result the same each time? Why or why not?
3. If you use this method to measure performance, what other data about the server should you record?
4. Is the time indicated the time SOLELY used by the for loop? Why or why not?

Sometime, you would like to check out the number of times the functions are called or used in your program. In such cases, `prof` is very useful.

Using your favourite text editor, enter the following code as `power.c` file:

```
#include <stdio.h>

int power(int base, int n)
{
    int i, p; /*delay loop for illustration only*/
    for (i=1; i<=65535; i++)
    {
        p = p + i;
    }
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}

int main()
{
    int i;

    for (i=0; i<10; ++i)
        printf("%d %d %d\n", i, power(i,2), power(i,3));
    return (0);
}
```

Now compile the program with the addition of the `-pg` flag (on the step where the binary file is produced). Alternatively, you can use the command `cc -pg -o power power.c` to bypass the generation of assembly code. Now run the program once. Once it is completed, it will generate a `mon.out` file, which has the information about the last program run.

Now enter command `prof ./power`. Examine the output and answer the following questions:

1. How many times was the power function called? Why?
2. How long does it take to complete each call of the power function? Can you confirm this number (and the unit) with an earlier code fragment?

3. Which function consumed the most time to complete?

Have you wondered how many assembly instructions were executed during runtime? Let's take a look at a (free) tool which counts the number of instructions executed in a program.

Using your favourite text editor, enter the following code as `instr_count.c` file:

```
#include <stdio.h>

int main(int argc, int *argv[])
{
    int a=1, b=2, c;
    c = a + b;
    /*printf("The Sum is %d.\n",c);*/
    return (0);
}
```

Now compile the program as per normal procedure.

1. You need to add the following path to your `PATH` environment variable so that the tools can be found:

```
/opt/local/pkg/cad/csse3001/shade-32/bin/
eg. export PATH=$PATH:/opt/local/pkg/cad/csse3001/shade-32/bin/
```

2. Now you will need to run the special program by typing `ifreq`. It will not give you any messages afterwards, so just run the `instr_count` you have compiled previously as per normal, ie. `./instr_count`. Now type `quit` or `ctrl-D` to exit.
3. How many instructions (opcode) were executed for the program? Does the number seem right? Take a look at the assembler listing of the program. Surely, the number is way off. This is because the executable version of the program will contain all the linked standard libraries, so we need to somehow isolate the part of the program which is related to the source code.
4. Take a look at the assembly code generated (`instr_count.s`). You will notice that the bulk of the code is a few `mov` and `st` instructions and the `add` instruction.
5. Using the command `dis ./instr_count` (which disassemble the binary program), find the part of the listing where you see the similar list of instructions, and note the range of addresses which relates to the same part. (It is most likely between 10620 to 10654, but it could be different for your experiment.)
6. We can now tell `ifreq` to only check out the range of addresses we are interested in, by invoking `ifreq` with this command:

```
ifreq -t, +t10620,10654
```

Note the spaces of the command above. `-t` means ignore all addresses, `+txxx,yyy` means to track addresses `xxx` to `yyy`.

Now investigate the effect of the `printf` statement with the steps above. How does `printf` affect the instruction counts? the final output? linked libraries?

Task 5: Embedding Assembly Instruction in C code.

Sometimes, it may be useful to embed assembly instructions into the C code as a function. Generally, this is done to get maximum performance for a particular processor. However, the result is that the code is not portable anymore, and it is bound to a specific type of processor. It is also much more difficult to maintain, as most programmers will avoid touching/editing any assembly code. Beware, sometimes assembly code may NOT be the fastest solution available.

Using your favourite text editor, enter the following code as `inline.c`:

```
#include <stdio.h>

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);
}
int add3(int a, int b, int c)
{ volatile int result=0;
  result=a+b+c;
  return (result);
}
```

And take a look at the assembly code generated:

```
! 5 int main()
! 6 { int i;
! 7 i=add3(1,3,5);

mov 1,%00
mov 3,%01
call add3
mov 5,%02
mov %00,%01
```

From this part of the assembly code, it looks like the sum is returned via the `%00` register and the parameters (1, 3 and 5) are placed in `%00`, `%01` and `%02`. (Recalled that the `mov` instruction after `call` instruction is executed before the `call` jump is completed) This is further confirmed by the return code on the `add3` procedure:

```
! 24 return (result);

ba .L98
mov %i5,%i4
```

```

! block 2
.L98:
mov %i4,%i0
jmp %i7+8
restore

```

From the move instruction, the result is in %i0, which after the restore command, is in the %o0 due to moving register windows.

The add3 procedure is compiled into the following assembly code:

```

! 10 }
! 11
! 12 int add3(int a, int b, int c)
! 13 {
! 14 volatile int result=0;

mov %g0,%i5

! 16 result=a+b+c;

ld [%fp+68],%o1
ld [%fp+72],%o0
add %o1,%o0,%o1
ld [%fp+76],%o0
add %o1,%o0,%i5

```

So as you can see, %i5 is used to (temporarily) house the value of result during this procedure. So technically, if we want to replace the C code `result=a+b+c` we must have the final result stored into %i5, and this is what we are going to do. Comment out the `result=a+b+c` line and we will replace it with assembly code directly:

```

....
int add3(int a, int b, int c)
{ volatile int result=0;
  /* result=a+b+c; */

  __asm("\n\
add %i0,%i1,%i5\n\
add %i5,%i2,%i5\n\
");

  return (result);
....

```

In theory, this should produce the same result as the original C code. For those who are familiar with scripting language, you will recognise the `\n` tags which is used to generate new lines. If you are guessing that it is passing these assembly code as text to the assembly file output (.s), then you are totally correct.

However, if you peruse the assembly code generated, you will see:

```
! 14 volatile int result=0;

st %g0,[%fp-8]
!#ASM

add %i0,%i1,%i5
```

Notice how the compiler decided that the result variable should now be stored in the heap (memory location `%fp-8`)? It is no longer in the `%i5` register, so the assembly code we inserted does not work. Try it, the output is now zero. So we will modify our assembly code fragment to use this location:

```
__asm("\n\
add %i0,%i1,%i5\n\
add %i5,%i2,%i5\n\
st %i5,[%fp-8]\n\
");
```

It would now work.

Try writing assembly code to replace some of the lines of C code that you have done above.

It is very clumsy to put "inline assembly" into C code this way. For each compiler, there is a certain method in writing the inline assembly. Unfortunately, it is NOT unified and is different on each compiler.

Summary

In this prac, we have covered:

- arithmetic operations;
- loops and their structures;
- type casting;
- performance measures;
- embedding assembly in C.

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 nested loops in C and see the assembly code generated by the assembler. Try initialising different arrays and see how they are handled.

Throughout this prac we have been using the C compiler from SUN, designed by SUN and optimised for their SPARC architecture. Most of you would be familiar with gcc, a free GNU compiler. It is also available on agave as long as you don't modify the PATH variable at the start of the prac. It is very interesting to compare the listing produced by various compilers, to see how code is handled, which one is more efficient etc. For your reference, simply replace "cc" with "gcc" using the same flags.
