CSSE3001/7000 2007

Prac 1: Introduction to SPARC Assembly

Last Update: 06/03/2007 10:36 AM

Author: Simon Leung

Preparations

Read the excerpts from Tanenbaum (5th Edition) to learn more about the UltraSPARC III processor architecture: section 1.4.2, 3.5.2, 5.1.6, 5.4.9 (also available internally). This is a very good introduction to the UltraSPARC III and you should read this thoroughly before attempting this prac. In particular, you should study the concept of moving register windows (fairly unique to the SPARC architecture).

Another very useful reference document is "The SPARC Architecture Manual, version 9" (also available internally). This is a very comprehensive (and long) book about SPARC, so do not read it from page 1 till the end. 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.

We will be using the School's SUN server, agave.students.itee.uq.edu.au, so make sure you have access (remote or otherwise) and an account on this server.

Also, you will need some basic UNIX skills and C programming skills for this prac.

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: What is the specification of the machine that we are using?

First of all, let us find out more about agave. Enter the command:

`/sbin/uname -a.`

It will tell us the O/S version, name, kernel version, processor architecture etc. The last term means that

this server is a SUN Netra 1280 model server.

Now, investigate the commands `top` and `psrinfo` (hint: `man psrinfo`), and answer the following questions:

1. How much physical memory does agave have?
2. How much virtual memory does agave have?
3. How many physical processors are there in agave?
4. What is the name of these processors?
5. What is the clock speed of these processors?

If you run these commands in a different Unix system, you will get information about that specific machine.

Task 2: Hello world in SPARC assembly.

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

```
#include <stdio.h>
int main()
{
  printf("Hello, world!\n");
  return (0);
}
```

As expected, the compiler will compile this code into binary program. When run, it will print "Hello, world!" and a carriage return to the screen.

We will compile this program in two steps:

1. *cc -S hello.c*
2. *cc -o hello hello.s*

The first command will produce a file called *hello.s*, which is assembler code generated by the compiler. The high-level C code is compiled into lower-level assembler code. The second command will use the results from step 1 and generate the executable binary for the machine to run. The compiler can achieved this in one step and skip the intermediate output, if the assembler code is not required. Here, we are interested in computer architecture, so we should examine the assembly code.

If you have done CSSE1000 before, some of the listing in *hello.s* will look familiar to you.

1. "!" are comment markers;
2. tab positions help you read the code: labels are left-aligned, and commands are at least one tab position to the right;
3. commands preceded with a full stop (eg. `.section`) are directives;
4. other lines are assembly code (eg. `save %sp,-104,%sp`).

In order to understand the assembly code, we will start by discussing some of the more important lines (not necessarily in order of appearance).

```
    sethi    %hi(.L95),%o0
    or       %o0,%lo(.L95),%o0
    ...
.L95
    .ascii   "Hello, world!\n\000"
    .type    .L95,#object
    .size    .L95,15
```

The `sethi` and `or` commands load the address of the memory which stores the ASCII string into output register 0 (`%o0`). Look up in the reference the meaning of `$hi`, `%lo`, `sethi` and `.ascii`. Also note that the assembly commands in SPARC are of the format:
`instruction` *`source1`*`,`*`source2(opt)`*`,`*`destination`*

```
    call     printf
```

1. How many bits are the instructions and registers used in SPARC?

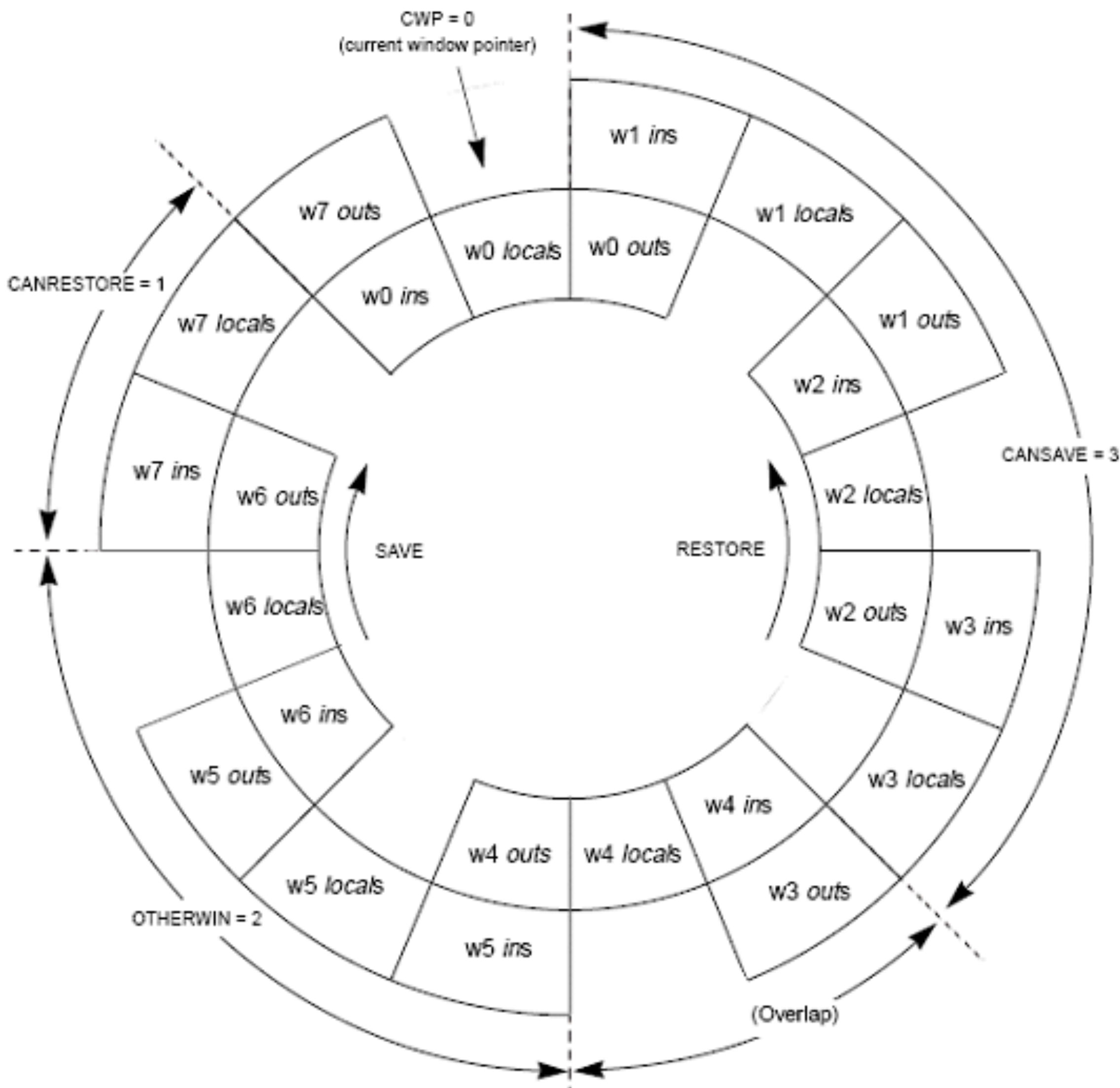
Once the address is loaded into the register, the *printf* procedure is called to print the text. These are the core functionalities of the program.

Moving Register Windows & Procedure Calls

The rest of the code is for managing the moving register windows, maintaining the values of the registers for other programs and proper handling and return. You need to be familiar with the SPARC architecture to understand the following discussion.

```
main:
    save     %sp,-104,%sp
```

This command creates (or slides) a new register window for the assembly instructions after it. It also puts a value into the new stack pointer (`%sp`) which is 104 bytes higher than the original stack pointer, so there is now new 104 bytes of stack to use. Note that because of the moving register window, the caller's stack pointer (the one who called the hello program) is now the frame pointer for the current procedure (the hello program). The following diagram illustrates how the register windows are structured: Each block in the diagram is a set of 8 registers. As the CWP is changed, a different set of registers is visible to the programmer, with some registers in common. Read the description in the references, and ask a tutor to explain if you still don't understand it!




```
    st      %g0,[%fp-4]
```

This command stores a value of zero to the contents of the address location `%fp-4`. This ensures the top element of the current stack is empty, possibly signifying that the current procedure has finished using the current stack.

```
    mov     %g0,%i0
    jmp     %i7+8
    restore
```

The program, upon its successful completion, must return a value of zero to the operating system. So a zero is loaded into input register 0 (%i0). The jmp instruction is the equivalent of a procedure return instruction, the program counter is set to the return address. (%sp is the stored PC, +8 because when the procedure is called, the PC is still pointing to the call instruction. Upon return, we need to skip over the call instruction and the delay slot.) The restore instruction closes (or slides back) the new register window created by the previous save instruction.

Data Definitions

```
    .section  ".rodata1",#alloc
    .align 4
.L95
    .ascii   "Hello, world!\n\000"
    .type    .L95,#object
    .size    .L95,15
```

The `.section` directive here tells the assembler that the following lines define the Read-Only Data (rodata) section of the code, and the `.ascii` directive allocates memory with the string "Hello, world!" and a carrriage return chararcter. The `.align` directive increases the memory address to the nearest multiple (of 4 here), so that data is aligned to a memory fetch.

```
    .section  ".bss",#alloc,#write
Bbss.bss:
    .skip     0
    .type     Bbss.bss,#object
    .size     Bbss.bss,0
```

The `.section` directive here defines the read-write data (bss) that is initialised to zero. The `.skip` skips a number of memory addresses, useful to leave spaces in the memory. Here, there is no data of this type and no skipping necessary.

```
    .section  ".data",#alloc,#write
Ddata.data:
    .skip     0
    .type     Ddata.data,#object
    .size     Ddata.data,0
```

The `.section` directive here defines the programs initialised read-write data (data). Again, not used in this program.

Back to the start of the program,

```
    .section  ".data",#alloc,#write
```

The `.section` directive here tells the assembler that the following section contains executable code (of course, until the next `.section` directive).

The other directives are not critical to the operation of the program, and they are fairly self-explanatory to understand.

Task 3: Effects of Variable Declarations

Modify your hello world program to include a (useless) variable declaration:

```
#include <stdio.h>
int main()
{
  int i=0;
  printf("Hello, world!\n");
  return (0);
}
```

When you examine the assembly output, it will now contain the line

```
    st  %g0,[%fp-8]
```

The memory location pointed by value `%fp-8` is now written by zero. In essence, the variable `i` is equivalent to memory location `%fp-8`.

Try initialising different types of variables, and see what happens.

Task 4: Interactive Hello World Program

The following modified hello world program will accept parameters at the command line. Try to run it with no parameter, one parameter and multiple parameters:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int count = argc;
  if (count > 1) {
    int i;
    for (i = 1; i < count; i++) {
      printf("Hello, %s\n", argv[i]);
    }
  }
  else {
    printf("Hello, world!\n");
  }
  return (0);
}
```

This program will be compiled into two pages worth of assembly instructions! Before you proceed any further, take a close look at the assembly intermediate output. Which commands, directives, and sections are familiar to you?

The first difference is found at the start of the assembly listing:

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

This, as you may have guessed, is related to the parameters passed to the program. Unlike a stack, which is usually found in `%fp-x` locations (notice the minus), the input parameters `%i0` and `%i1` are stored into other locations (probably setup by the operating system) outside the stack so they would not be affected by the program and remain accessible by the rest of the code.

Further down, we see:

```
! 5 int count = argc;

    ld [%fp+68],%i4
```

When we initialise the variable count, the compiler decided that we need to load (again!) from the memory location `%fp+68` into an input register `%i4`.

```
! 6 if (count > 1) {

    mov %i4,%l0
    cmp %l0,1
    ble .L95
    nop
```

The compiler also recognised that `count` is a local variable, so it copies the value from the input register into a local register `%l0`. A comparison (`cmp`) is made and branch if count is less than or equal to 1. When will it branch?

The `nop` operation is to fill in the branch delay slot, which is ALWAYS executed whether branch is taken or not. More on the branch delay slot in later lectures and pracs. For now, think of it as a by-product of pipelining and that whatever instruction is in that slot will be executed and then the branch will complete.

So in the case where there are no parameters entered on the command line, the program will branch to

the section labelled `.L95:`

```
! 10 }
! 11 }
! 12 else {
! 13 printf("Hello, world!\n");

    sethi   %hi(.L104),%o0
    or      %o0,%lo(.L104),%o0
    call    printf
```

which is familiar to you.

In the case where parameters are entered, the program will NOT branch and proceed to the next part:

```
! block 2
.L96:

! 7 int i;
! 8 for (i = 1; i < count; i++) {

    mov 1,%i5
    cmp %i5,%l0
    bge .L99
```

As you can see, variable `i` is set to input register `%i5` and is used to keep the loop index. It is compared to `%l0`, which is `count`. Where is the assembly instruction corresponding to `i++`? Further on...

```
! 9 printf("Hello, %s\n", argv[i]);

    ld   [%fp+72],%l2
    sll  %i5,2,%l1
    ld   [%l2+%l1],%o1
    call printf
    mov  %l3,%o0
    add  %i5,1,%i5
    cmp  %i5,%i4
    bl   .L97
```

The first instruction of this listing, the contents of the second parameter (which is a pointer to a memory location) is loaded into local register `%l2`. The next instruction, the variable `i` is changed to the byte address equivalent and stored in `%l1`. When these two values are combined (at each iteration), it will point to the corresponding parameters entered by the user. The parameter is then passed to the `printf` procedure via output register `%o1`. `i++` is done after the printing using the `add` instruction and another comparison is made to see if the loop has to be repeated.

You should now examine the rest of the assembly code to see how it differs and resembles the previous code.

If you analyse the code carefully, you would find that many of the assembly instructions are redundant, and the loops and code generated are NOT optimal. You should try to manually optimise the assembly code to see if you can maintain the functionalities of the program but cut down the number of instructions required. At this stage, you may want to take a look at the optimisations provided by the compiler, but the code generated is much less readable.

Summary

In this prac, we have covered:

- the steps needed to identify a Unix machine;
- Sun compiler and its compiler flags;
- assembly commands associated with hello world program and some directives;
- moving register windows;
- procedure calls and stacks;
- parameters;
- branching in assembly.

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 multiple printf statements in C and see the assembly code generated by the assembler. Try initialising different variables at different part of the C code and see how it is 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.