

COMP2300 Laboratory 09

Caches, SPARC Assembly and Linking and Loading

Semester 1, 2007 Week 11 (14 - 20 May)

As well as the Preparation Exercises for this class, you are expected to have revised lectures M3--M5 and O2 (of course it would be a very good idea to bring your notes with you to the session!). It will also be useful to have read this document beforehand.

Objectives

This lab has the following aims.

- To deepen your understanding of caches.
- To give you a brief introduction to the assembly language of a real machine, in this case SPARC assembly.
- To understand load objects and symbol tables.

Preparation Exercises

Complete the following questions on a separate sheet of paper, with your name and student number clearly written. Please ensure your writing is legible. Hand in to your tutor at the *beginning of your tutorial / laboratory session*.

1. Define the terms *direct-mapped cache* and *k-way set-associative cache*. What are typical values for *k*?
2. What is a *cache line*, and what lengths are these typically?
3. What is a *branch delay slot*, and why were they introduced to RISC processors?
4. Write the SPARC assembly language instruction that sets register `%o2` to the sum of registers `%i1` and `%i2`.
5. Suppose you wished to write an assembly language program to call the C function `int f(int x, int y)`. Which registers would you use to pass the parameters, and which would hold the return value?

Tutorial Questions on Caches

1. Consider a program which repeatedly accesses all elements in an `int` array of length *N* in a cyclic fashion (from first to last element), executing on a computer with a 64 KB data cache. For example, the computation could be:

```
while (1) {
    int s = 0, i;
    for (i = 0; i < N ; i++)
        s += a[i];
}
```

Suppose the first iteration has already been completed. Assuming that the cache line size is 4 bytes, and the cache is fully associative, and the replacement policy is random, calculate the *cache hit rate* for (a) $N = 2^{13}$, (b) $N = 2^{14}$, (c) $N = 2^{15}$ and (d) $N = 2^{16}$.

Explain would a *least recently used* policy perform better or worse?

Recall that the *cache hit rate* is calculated as the percentage of (in this case `int`) load/store operations that occur when the data is in the cache.

2. Repeat the above for a direct-mapped cache (note the replacement policy plays no role in this case).
3. Repeat Q2 for a line size of 16 bytes. Does this affect the rate? If so, does this necessarily mean the increase in line size would affect the computer's execution rate?
4. An assumption in the above is that there is no significant amounts of data access other than to the array. Discuss the extent you expect this to hold on both SPARC and Intel x86 architectures.

Laboratory Exercises

SPARC Assembly Language

As discussed in lectures the SPARC architecture is a load/store architecture. All arithmetic and logical operations are carried out between operands located in registers. Load and store instructions are provided to load and store register contents from memory. The machine has 32 registers available to the programmer at any one time. These registers are logically divided into four sets:

- *global registers* (`%g0-%g7` in SPARC assembly) are for global register data, ie data that has meaning for the entire program.
- *local registers* (`%l0-%l7`) are for local function variables.
- *out registers* (`%o0-%o7`) are used as temporaries and in passing arguments to functions
- *in registers* (`%i0-%i7`) are used as temporaries and in receiving arguments from calling functions

Two of the out registers `%o6` and `%o7` have special use and should not be used (`%o6` is the stack pointer or `%sp`, while `%o7` is the called function's return address). Also the first global register (`%g0`) is special in that it always returns zero when read and discards anything written to it.

While the default for C programs files is to append a `.c` to the end of the file name, for assembly programs it is common to append a `.s`. Also, since compiling a C program is a two step process that first involves translation of the C program to assembly and then from assembly to machine code and linkage, we can use the C compiler both to produce assembly code from an existing C program, and to produce machine code and link our assembly programs.

```
gcc -S program.c           # to produce assembly listings from C code
gcc program.s -o executable # to compile and link assembly code
```

We will start by looking at a very simple example:

```
/* This program computes the expression:
   y = (x-1)*(x-7)/(x-11)
*/
define(a2,1)
define(a1,7)
define(a0,11)

define(x_i,9)

define(x_r,10)
define(y_r,11)

.global main
main:
save    %sp, -96, %sp
mov     x_i, %x_r      ! initialize x (=x_i)
sub     %x_r, a2, %o0  ! (x - a2) into %o0
sub     %x_r, a1, %o1  ! (x - a1) into %o1
call    .mul
nop     ! result in %o0
sub     %x_r, a0, %o1  ! (x - a0) into %o1, the divisor
call    .div
nop     ! result in %o0
mov     %o0, %y_r      ! store it in y

mov     1, %g1         ! exit request number (=1)
ta     0               ! trap to system
```

For convenience the program is written using the `m4` macro preprocessor (do `man m4` on `iwaki` to learn about this). We will only make very limited use of it. Notably in the above example the `m4` preprocessor is used in an equivalent manner to concise macros in PeANUt, allowing us to, e.g. replace the variable `a2` with `1` (`define(a2,1)`) etc.

- A copy of the above code is available as [example1.m4](#). Save this to a file of the same name on `iwaki`. Run it through the `m4` preprocessor using the following command:

```
m4 example1.m4 > example1.s
```

and inspect the output file `example1.s` using an editor. This will make clear what the `m4` preprocessor has done.

Things to note about the assembly code:

- The linker will look for for an address (pointing to a block of code) labelled `main`. This is declared to be `.global` in an analogous manner to the `global` definition in PeANUt and for the same reason.
- Thus the first user program instruction to be executed is `save %sp, -96, %sp`. This command provides space,

on the stack, to save ALL general-purpose registers (if required).

- Most SPARC instructions take three operands: two registers and a literal constant, or three registers:

```
op regrs1, regor_imm, regrd
```

Where the content of the first or source register reg_{rs1} is combined with the literal or the contents of the second source register reg_{rs2} to produce a result that is stored in the destination register reg_{rd} . The number of bits available in the instruction word for storing a literal constant allows for values from -4096 to +4095.

- `mov` moves either a register or literal from the source to the destination register.

```
mov regrs1, regrd
```

- `clr` sets the value of a register to zero

```
clr regrd
```

- `sub` is obviously subtract, while `add` is addition.
- The original SPARC Instruction Set Architecture (ISA) had no integer multiplication or division instructions. Instead these operations are accomplished by a series of bit shifts and additions. This is, however, all available in system provided functions we access via the `call .mul` and `call .div` statements (we will come back to this point at the very end of this lab!).
- Notice how the operands for the multiplication and division are placed in registers `%o0` and `%o1`, and the result is retrieved from `%o0`.
- Notice also that after each function call we have a *branch delay slot* that has been filled via a `nop` or "No Operation" instruction. This consumes one execution cycle (in one of the integer pipelines) but does nothing.
- Finally the last two instructions return us to the operation system. The trap instruction `ta` calls the operating system with the service request encoded into register `%g1` (not so different to how we terminated a PeANUt program).

- Compile and run the program on iwaki as follows:

```
me@iwaki> m4 example1.m4 >! example1.s
me@iwaki> gcc example1.s -o example1.exe
me@iwaki> example1.exe
```

A problem, that you may now have noticed, is that we have no means of knowing if the program is correct since it produces no output! Just like in PeANUt, input/output is an involved topic since it involves translation of the number to an ASCII representation etc etc. At this stage we could use the debugger (`gdb`) to run the code and examine the contents of the various registers as the calculation proceeds. This in itself would be a valuable exercise, but I will let you pursue that in your leisure time. Instead we, will do basic integer I/O through a user defined function that in C corresponds to:

```
void myprt(int value){
    printf( "Integer value is %d \n",value);
}
```

We can use this function in the assembly in the following manner:

```
/* This program computes the expression:
   y = (x-1)*(x-7)/(x-11)
*/
define(a2,1)
define(a1,7)
define(a0,11)

define(x_i,9)

define(x_r,10)
define(y_r,11)

.global main
main:
save    %sp, -96, %sp
mov     x_i, %x_r      ! initialize x (=x_i)
sub     %x_r, a2, %o0  ! (x - a2) into %o0
sub     %x_r, a1, %o1  ! (x - a1) into %o1
call    .mul
nop     ! result in %o0
sub     %x_r, a0, %o1  ! (x - a0) into %o1, the divisor
call    .div
nop     ! result in %o0
mov     %o0, %y_r      ! store it in y

! MYPRT FUNCTION CALL
mov     %y_r, %o0      ! move what we want to print to %o0
```

```

        call    myprt,0          ! call myprt
        nop                    ! remember to have a nop
! END MYPRT FUNCTION CALL

        mov     1, %g1          ! exit request number (=1)
        ta     0                ! trap to system

! BEGIN DETAILS OF MYPRT FUNCTION
.section    ".rodata"
        .align 8
.LLC0:
        .asciz "Integer value is %d \n"

.section    ".text"
        .align 4
        .global myprt
        .type  myprt,#function
        .proc  020
myprt:
        save   %sp, -96, %sp
        sethi  %hi(.LLC0), %o1
        or     %o1, %lo(.LLC0), %o0
        mov    %i0, %o1
        call   printf,0
        nop
        ret
        restore

```

You should note the following:

- The call to `myprt` from the `main` program and how the value of the register to be printed is passed to the function.
- How the character string associated with `printf()` is stored in function `myprt()`.
- How the starting address for this character string, defined by `.LLC0`, is loaded into register `%o0` via the `sethi` (set high 22 bits of an immediate constant) and subsequent `or` instructions.

- Why can the address of the print string not be set using a single instruction? (it requires both the `sethi` and `or` instructions outlined above).
- A copy of this version of our program is available in [example2.m4](#). Save this on `iwaki`, and then use `m4/gcc` to preprocess/compile the code. Run the executable and ensure that it prints out the correct result (`-8`).
- Insert additional calls to `myprt` to print out the values of various intermediates (ensure you get the results you expect).

Branch Delay Slots

As mentioned in lectures, a branch delay slot is a characteristic of RISC machines. In our program the `call` instruction is an example of an operation that has a branch delay slot and these are currently filled with `nop` instructions.

- Remove the `nop` instructions from immediately after the `call .mul` AND the `call .div` lines. Build the code and run it. Verify that the results are now wrong!

If we know that the instruction immediately after `call` instructions is always executed BEFORE jumping to whatever address corresponds to the called function, we can restructure the code to try and move suitable instructions from before the `call` instruction into this *branch delay slot*. Obviously these moved instructions cannot be instructions that determine whether or not the branch is taken!

- Restructure the code such that there are NO `nop` instructions between the initial `save` instruction, and the `mov %o0, %y_r` instruction. While ensuring the the code also produces the correct result!

Filling delay slots and removing `nop` instructions is important as it both reduces the size of the executable, and saves operations (therefore speeds up the code). As we will see later, this is normally done by the compiler, providing you use a compiler flag that allows the compiler to use a moderate level of optimisation.

Branching

Like in PeANUt branch instructions test for condition codes in order to determine if a branch condition exists. Branch instructions take the following form:

```
bicc label
```

where b_{icc} stands for one of the branches testing the integer condition codes. A few of the possible branch conditions are given in the following table:

Assembly	Action
ba	branch always
bn	branch never (why does this exist?)
bl	branch on less than zero
ble	branch on less or equal to zero
be	branch on equal to zero
bne	branch on not equal to zero
bge	branch on greater or equal to zero
bg	branch on greater than zero

To set the condition code we can use the following compare instruction

```
cmp reg_rsl, reg_or_imm
```

for example we might construct a conditional branch in the following manner:

```
sub %x_r, 1, %x_r ! x_r = x_r - 1
cmp %x_r, %g0    ! compare x_r with %g0 (which is always equal to zero)
bgt loop        ! branch to loop if x_r > 0
nop             ! note we also have a branch delay slot
```

Note that a good assembly programmer would try to fill the branch delay slot with a useful instruction.

The following C program computes the value of the simple polynomial for integer values of x from 0 to 10.

```
#define A2 1
#define A1 7
#define A0 11
void myprt(int value);
int main(){
    int x, y;
    x = 0;
    do {
        y = ((x - A2) * (x - A1)) / (x - A0);
        myprt(y);
        x++;
    } while (x < 11);
    return 0;
}
void myprt(int value){
    printf("Integer value is %d \n", value);
}
```

The above C code is available in file [example3.c](#).

- Compile and run example [example3.c](#) making sure that the results are what you expect them to be!
- Modify your `example2.m4` code to reproduce the results from [example3.c](#).
- Now modify `myprt` in `example2.m4` to also print x using the equivalent of the following C statement:

```
printf("Value of y = %d for x = %d \n", y, x);
```

Generating Assembly from C Code

File [example3.c](#) contains the C code given above.

- Generate assembly for this C code using

```
gcc -S example3.c
```

A few differences from what we have done will be evident:

- Lots of header info for the `main()` function (that we ignored)
- Slightly different stack pointer offsets in the `save` function (it requires more space).
- Use of the frame pointer `%fp` with offsets to point to stack are where the local registers may be saved.
- Use of `%g0` to initialize `x` to zero.
- Use of `umul` instead of `mul`, in this case it produces the same result but is slightly faster.
- Lots of jumps at the end, and a return value for function `main()`.

- Now generate assembly code but tell the compiler that it can attempt to optimise the code:

```
gcc -O -S example3.c
```

inspect the resulting assembly code.

In the last case did you notice how:

- All the `nop` instructions have been removed.
- The use of stack to implement local variables has been removed (i.e. what was using `%fp`).

So far we have been using the GNU C compiler `gcc`. Of course Sun also have their own C compiler. This is called `cc` and is located in the directory `/opt/SUNWspro/bin`.

- Now generate assembly using the Sun C compiler and no optimisation:

```
/opt/SUNWspro/bin/cc -x00 -S example3.c
```

inspect this and compare the result with that obtained with high level optimization

```
/opt/SUNWspro/bin/cc -x05 -S example3.c
```

Note how the `myprt` has been in-lined
- Now compile using the `-fast` flag:

```
/opt/SUNWspro/bin/cc -fast -S example3.c
```

This will give some warning messages about not being able to run the resulting code on non-UltraSPARC architectures. If you now inspect the result you will notice that the calls to `.mul` and `.div` have been removed. UltraSPARC does actually have multiply and divide instructions - but the original SPARC architecture did not! So these instructions are only used if you tell the compiler that you never want to run the executable on older machines.

Finally you should note that given an executable image it is possible to disassemble this into assembly code. The command is:

```
/usr/ccs/bin/dis
```

Do `man dis` on `iwaki`.

- Compile the C version of `example3.c`

```
gcc example3.c -o example3.exe
```

Run the executable and make sure it works. Then disassemble it using the following command

```
/usr/ccs/bin/dis example3.exe > example3.dis
```

Inspect the resulting file using an editor. Compare the code for `_start` with the pseudo-code given in the O2 lecture. You should be able to locate the `main` program and function `myprt` and relate the resulting assembly to what is given via

```
gcc -S example3.c
```

Symbol Tables

Consider the following version of the `swap.c` program discussed in lectures. The file has been modified to include a function that counts the number of times the function `swap()` is called.

```
/* swap.c */
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void incr()
{
    static int count=0;
    count++;
}

void swap()
{
    int temp;

    incr();
    bufp1 = &buf[1];
    temp = *bufp0;
```

```

*bufp0 = *bufp1;
*bufp1 = temp;
}

```

- For each symbol that is defined and referenced in `swap.o` indicate if it will have a symbol table entry in the `.symtab` section in module `swap.o`. If so, indicate the symbol type (function, object (=data), or undefined (=notype, used for externally defined symbols)), the symbol binding (local or global (external symbols must also be global)) and the section (`.text`, `.data` or `.bss`) it occupies in that module.

symbol	has <code>.symtab</code> entry?	symbol type	symbol binding	section
buf				
bufp0				
bufp1				
swap				
temp				
incr				
count				

- Use the utility program `readelf -S -s swap.o` (on Linux, note that ``Ndx'` signifies section index) or `elfdump swap.o` (on `iwaki`) to verify your answers. You can produce the file `swap.o` via the command `gcc -c swap.c`.
- Your colleague sees the above code and suggests that you should replace the line

```
void incr()
```

with

```
static void incr()
```

why is this a good idea? (HINT, use `readelf/elfdump` to compare the symbol tables obtained for `swap.o` with and without this modification).

A Challenge!

Consider the following program, which consists of two object modules:

```

/* file foo.c */
void p2(void);
int main()
{
    p2();
    return 0;
}

/* file bar.c */
#include <stdio.h>
char main;
void p2()
{
    printf("0x%x\n", main);
}

```

When this program is compiled and executed on an x86 linux system, it prints the string `"0x55\n"` and terminates normally, even though `p2` never initializes variable `main`. Explain exactly why this value is printed out? Interestingly the same code fails to link on `iwaki`, even though we are still compiling with `gcc` - why is this different?

Last modified: Wed May 9 17:00:48 EST 2007