



## COMP2300

- Home
- StudyAt
- Assessment
- Schedule
- Lecture Notes
- Tute / Labs
- Assignments
- Announcements
- Discussion
- Reading material
- Old Exams
- Links
- PeANUt@Home
- Getting Help

## Quick Links

- ANU Home
- FEIT Home
- DCS Home
- Search



## COMP2300 Laboratory 09

### Caches, SPARC Assembly and Linking and Loading

Semester 1, 2009      Week 11 (18 - 22 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.

Also for this session, there is a submitable laboratory exercise which is due by 09 am Tuesday 26 May (week 12), which will contribute up to 1% of your assessment (in the Tute/Lab mark).

#### 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

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-associate 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. Note that this is a cache version of *Belady's anomaly* (Lecture M2). 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. Assume that the cache line size is 4 bytes, and the cache is fully associative (note: this is not a realistic cache!), and the replacement policy is least recently used. Calculate the *cache hit rate* for (a)  $N = 213$ , (b)  $N = 214$ , (c)  $N = 215$  and (d)  $N = 216$ . *Hint: calculate the relative size of  $a[i]$  to the cache first.*

Explain whether a *random* replacement policy perform better or worse in each case.

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 that the replacement policy

plays no role in this case).

3. Suppose we now had a line size of 16 bytes. How would this affect the rate (consider the situation in Q2 where you had a 0%, 50% and 100% hit 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. *Hint: imagine writing the above loop in PeANUt; what other memory accesses than the array itself would you have?*
5. Reconsider Question 1 for the random replacement policy. For parts (c) and (d), consider the following argument.  
 For  $N=2^{15}$ , we expect half of the array will be resident in cache with a random replacement policy. Thus half the memory accesses will hit, resulting in a 50% hit rate.  
 For  $N=2^{15}$ , we expect 1/4 of the array will be resident; thus we expect a 25% hit rate.

*Evidently, this argument was so persuasive that 2 PhD students and 100 undergraduates did not challenge it when it was proposed in 2007!*

Is this argument correct? What tools do you have in order to test it? If it is correct, develop reasons to support the argument. If not correct, what are the reasons why this is the case.

*No answers will be published on this question. Its up to you and your class to resolve it!*

### Laboratory Exercises

If you have read this document beforehand, you should be able to complete the exercise to the end of the [section on branching](#) in one hour of your session. If need be, the rest of the exercise should be completed for homework.

### 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
```

- Copy the files /dept/dcs/comp2300/public/lab9/\* into your lab9 folder.

We will start by looking at a very simple example, example1.S. Open your file in your favorite text editor.

The program relies on the C pre-processor (cpp) to expand macros, in an equivalent manner to concise macros in PeANUt (except we can use it for register names as well as numbers). The suffix convention .S instructs gcc to run cpp over the text, before invoking the assembler (as).