# ECSE 425
# Computer Organization and Architecture

## Assignment 1

**Due: February 1, 2007 at 4:00 PM in the Trottier Assignment Box**

- You may make use of reasonable assumptions of your own for data that might be missing in the problem texts, provided that they are explicitly and clearly stated.

- You may submit a partial answer to a problem. The grading will account for this.

- Their main purpose of this assignment is to help you attain an understanding of the course material (the primary goal) and, in addition, to help you prepare for the exams.

### Question 1: IC Cost Model (10%)

We developed in class (Chap. 1 of the text) a formula that allowed us to estimate the "die yield", that is, the average fraction of working dies cut out of a wafer, as a function of defect density, die area, and a parameter α that reflected the manufacturing process.

The parameter α depends strongly on the complexity of the technology (e.g. the number of steps of a fabrication process). This was our first exposure to the necessity of thinking of computer architecture in terms of tradeoffs. For example, we might think of implementing the same circuit with different processes, i.e. different α's. It is natural to think that a process with a low α will cause the die to be larger owing to a lower integration density.

For the sake of argument, and all other things being equal, assume that the die area is inversely proportional to α. The nominal chip occupies 1 cm$^2$ for an alpha equal to 3. The number of defects per cm$^2$ is 0.6. For simplicity, take "wafer yield" to be 100%. Find the number of working chips (before packaging and final test) you can get from a 21 cm diameter wafer for alpha equal to 2 and alpha equal to 4.

### Question 2: CPU Performance and Ahmdahl's Law (15%)

In this question we investigate the effect on CPU performance when an instruction that immediately follows another instruction depends on the result of the first instruction. For example, a "load" instruction which loads a value from memory into a register might be followed by an ALU instruction which uses this value from the register. As we will see in Appendix A, a technique called pipelining overlaps the execution of instructions so that the second instruction starts to execute before the first one is finished. If the output from the first instruction is not available in time to be used as input to the second instruction then the machine has to stall the execution of the second instruction until the result from the first is available.

Assume that in a particular machine, load instructions create 1 clock cycle stalls when the instruction that immediately follows depends on the result of the load (this is called a *hazard*). Statistics show that 25% of the instruction mix are loads and that hazards occur for 50% of loads. Furthermore, all "branch" instructions create 1 clock stalls, and 10% of the instruction mix are branches.

A combination of architectural and software techniques (to be explored later in Chap. 3) would make it possible to reduce the stall rate for loads from 50% to 25%, and reduce the stall rate for branches from 100% to 35%. Assume all other types of instructions take 1 clock cycle to execute (while loads and branches also take 1 clock cycle when they do not have to stall).

What is the overall speedup that would result from the application of the enhancements? Find the speedup using two different approaches. In the first approach, use the CPU time equation directly. For the second approach, use Ahmdahl's law as a different way to get the same answer.

**Question 3: CPI (15%)**

In this question, we are concerned *only* with cache performance. Recall that a cache is a small, fast memory that is used to buffer all accesses between the CPU and the main memory (we will study this more in depth in Chapter 5). A portion of the main memory is mirrored in the cache. When the CPU needs to read from (load) or write to (store) the main memory it instead goes to the cache. If the data it is trying to read or write is in the cache then this is called a *hit*. A *miss* occurs when the data is not in the cache. A miss means there will be a performance penalty to access the data from the slower main memory and bring it into the cache.

Suppose that you have a machine with a cache and an instruction mix as follows. The data below assumes a perfect cache, that is, the hit rate is 1.

| Type | Frequency | Clock Cycles |
|---|---|---|
| ALU ops | 40% | 1 |
| Loads | 30% | 2 |
| Stores | 10% | 1 |
| Branches | 20% | 2 |

With an imperfect cache, measurements show that instruction fetches have a miss rate of 6%, load/stores, which reference the memory, have a miss rate of 10%, and that the miss penalty is 40 cycles.

(a) Find the CPI for each instruction type.
(b) Find the CPI for the overall machine.
(c) How much faster is the machine that does not have any cache misses?

**Question 4: Tradeoffs (15%)**

Suppose you are considering a design change to an instruction set. The current design is a pure load/store architecture like MIPS (which means that all data accesses to memory are only through explicit load and store instructions). The case we consider yields an instruction mix identical to that in the previous question. Assume that the cache is imperfect as described in the previous question.

You propose to your design team leader to add a new instruction that makes it possible to eliminate 50% of the branches. These new instructions would have a basic clock cycle count of 1.

Your team leader replies that it is apparent that the design is more complicated, with the consequences that the machine clock cycle time would be increased by a factor *x*. To make a convincing argument to the hardware design team, determine the maximum tolerable increase *x* in clock cycle time which would yield a design which performs better than the original design.

**Question 5: More Trade Offs (15%)**

(a) Consider a machine with separate caches for instructions and data. The CPI is 2 when in the ideal case all memory references including instruction fetches are cache hits. The only data accesses are by Loads and Stores and these form 40% of all instructions. For a real machine, the miss penalty is 30 clock cycles and the miss rate is 2% for the data cache for both read and writes. For the instruction cache the miss penalty is also 30 clock cycles but the miss rate is 2.5%. What is the speed-up factor for the ideal machine when all memory references are cache hits, relative to the real machine?

(b) Consider adding an enhancement to the real machine that would increase the clock cycle time by 5% but would save 30% of the Load/Stores. Is the enhancement worth being implemented assuming it is free of cost?

**Question 6: Performance (30%)**

Using compiler optimization can greatly improve the execution times of your programs. In this question, we compare the performance of a machine with and without compiler optimization. We use two benchmark programs for this performance comparison.

(a) Write the following two benchmark programs in C:

Program A fills an N x N array of double precision numbers with random values in the range [-π, π]. Program A then replaces each array entry with its cosine.

Program B fills two N X N arrays in the same way as Program A (i.e., with cosines of random doubles), then transposes one of the arrays. Finally, compute the matrix product of the two arrays in a third array.

In order to generate random numbers, use both `srand()` and `rand()`, as well as RAND_MAX (you can use the current time as the seed). You can find more information about the random functions using the UNIX "man" command, e.g. `man srand`. The array sizes for both programs will have to be large (N = hundreds to thousands, as explained later), therefore make sure to declare all arrays as *global* variables. Your programs do not have to be efficient (and don't bother using dynamic memory allocation for the arrays), just be correct.

Compile each program with the GNU compiler, both with and without level-2 optimization:

```
gcc progA.c -lm -o progA_noopt
gcc progA.c -O2 -lm -o progA_opt
gcc progB.c -lm -o progB_noopt
gcc progB.c -O2 -lm -o progB_opt
```

Run each program with the UNIX time command:

```
prompt$ time some_program
90.7u 12.9s 2:39 65%
```

Make sure to read p. 26 of the textbook to understand what is returned by the time command. Adjust the array dimension N so the user CPU times are between 1 and 10 seconds. You may have to choose different values of N for programs A and B in order to get reasonable CPU times.

(b) Repeat the time measurement 10 times for each program and tabulate the resulting user CPU times and system CPU times. Calculate the mean and standard deviations of the CPU times. Please use the format of the sample table given below (or a very similar one) for your tabulation. Make sure to write down the values of N for programs A and B. Also, specify your operating system name and version, and the GNU compiler version used. Which CPU time is more affected by optimization, system CPU time or user CPU time? Why?

```
Table of CPU times in sec (N=3000 for prog A, N=1000 for prog B)
------------------------------------------------------------------
      | progA_noopt  | progB_noopt  | progA_opt    | progB_opt
Run # |--------------+--------------+--------------+--------------
      | user  system | user  system | user  system | user  system
------+--------------+--------------+--------------+--------------
  1   | 5.61  1.23   |              |              |
  2   | 5.42  1.31   |              |              |
  .   |              |      .       |      .       |       .
  .   |              |      .       |      .       |       .
      |              |              |              |
 10   | 5.50  1.19   |              |              |
------+--------------+--------------+--------------+--------------
mean  | 5.51  1.24   |              |              |
s.d.  | 0.06  0.04   |              |              |
------------------------------------------------------------------
```

(c) For parts (c) and (d), consider only the user CPU time as a program's execution time. With the tabulated mean of user CPU times, we can now compare the performance of your machine with and without optimization. Calculate the weighted arithmetic mean execution times for the machine with and without optimization. What is the speedup in execution time with optimization?

(d) Performance comparison can also be done by looking at MFLOPS (millions of floating-point operations per second) ratings rather than execution times. The obvious way of calculating native MFLOPS ratings is

MFLOPSnative = # floating point operations / (execution time in secs x $10^6$)

Since some types of floating-point operations take longer to execute than others, we assign different weights to different types of floating-point operations in order to make MFLOPS comparisons more fair (the weights are called normalization factors). Assume the following:

| FP operation | Normalization factor |
|---|---|
| add/sub | 1 |
| mult/div | 4 |
| trig | 8 |

MFLOPSnormalized = # normalized floating point operations / (execution time in secs x $10^6$)

Estimate the number of floating-point operations in both programs (divide them into the three types in the table above). Calculate both the native and normalized MFLOPS ratings for the machine with and without optimization.