# Project Assignment – Assembler and cache optimization

## Course: Computer Systems Engineering – EDA331, Chalmers School of Technology

## Participants:

Jesper Josefsson – 860409-5276

Linus Oleander – 880613-4873

*2011-05-16*

# Background

The assignment demands that a Gauss elimination algorithm is translated from C-code into optimized assembler code. Thereafter, the memory architecture of a fictive MIPS-machine is optimized in order to yield minimal execution time and component cost.

Component prices, system configuration and a C-algorithm are given in the assignment. Choices include block size – 1, 2, 4 or 8 words; associativity – 1, 2 or 4; memory type – the slower type I or faster type II.

Greater cache sizes and bigger associativity require lower clock frequencies and cost more money. Faster memory is more expensive.

The aim of the assignment is to minimize the time · cost-product, measured in μsC$ (microsecondChalmersDollar).

# Implementation

## Assembler code

First, the C code was translated directly into MIPS assembler, including using integers as described in the original algorithm to control the for-loops. It was however discovered that major optimization could be achieved by using pointer arithmetic instead. This caused a major overhaul of the original assembler code.

Furthermore, all available delay slots were used and we ended up copying some code instead of using subroutines, which saved us a few cycles.

We feel that at this point, the bottleneck is in the original algorithm. Further optimization would therefore require using more 1337 mathematics.

The temporary T-registers were utilized first. This is to avoid having to push a large amount of registers to the stack.

Since the MIPS floating point instruction set seems to lack an addi-instruction, we wrote a zero and a one to memory and loaded them into the f0 and f1 floating-point registers for later use.

## System architecture

Since memory latencies in the assigment were given in nanoseconds and the simulator counts them in cycles, cycle equivalents had to be calculated for different memory types and clock speeds.

It was discovered early on that the default 128-byte cache was more than sufficient for the instruction cache. This is probably because of the assembler-level optimization, which enabled whole loops to reside entirely in the cache.

Consequently, most of the work was to be done with the data cache. The main method of optimization was large scale testing. When choosing which configurations to test we relied on previous experience, common sense and some initial probing samples.

From lab sessions and lectures, we know that a 2-way associative cache is an effective choice for data caching. It reduces collisions by letting us save blocks with the same index at two different locations in the cache. This lowers the miss-rate but doesn’t add as much overhead due to searching in the cache as a 4-way associative arrangement does.

Some sample tests confirmed that a direct-mapping cache was a bad choice for the data cache. Thus, we tested only n-way associative caches from thence on.

Other sample tests showed the very small caches suffered from poor performance and that very large caches with big associativity gave higher time-performance values because of spiraling cost. Further test cases could be eliminated.

A list of remaining system permutations was defined and large scale testing commenced.

In the 256 and 512 bytes caches, larger block sizes gave better results. This is presumably due to the high level of spatial locality present in the algorithm. Data is accessed in large, contiguous chunks.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cache | Cache size (bytes) | Block size (words) | Associativity | Write Buffer | Memory Type | Price (additional) |
| D | 512 | 8 | 2 | 0 | 2 | 1 |
| I | 128 | 8 | 1 | - | 2 | 0 |

Soon it was clear that very large write-buffers didn’t give enough of a performance boost to be worth the money. This was more pronounced with the faster memory. One can presume that write-buffers are more important when the memory is slower.

The final result was the following system:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total cost (C$) | Clock frequency (MHz) | Cycles | Time (micro seconds) | Time·Performance (μsC$) |
| 2 + 0.25 + 0.75 = 3 | 90 | 93694 | 1041 | 3123 |

Time and performance values: