Computer Architecture - tutorial 2

Context, Objectives and Organization

Covered Lectures 3 (compiler optimizations and pipelining) and 4 (pipelining).

The goals of the quantitative exercises in this tutorial are: to familiarize students with writing MIPS assembly code (E1), and to provide examples that demonstrate the principles and performance implications of pipelining (E2 and E3). The discussion elaborate on the role of the compiler in modern computer architecture (D1).

This tutorial consists of two activities. The first involves quantitative problem solving. The exercises are extracted from the H&P book 3rd edition.

E1: H&P 2.8 pg. 164, groups of 2 – 15 min

Problem

Consider the following fragment of C code:

```c
for (i=0; i<100; i++)
    {A[i] = B[i] + C;}
```

Assume that A and B are arrays of 64-bit integers, and C and i are 64-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 0, 5000, 1500, and 2000 for A, B, C, and i, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop.

1. Write the code for MIPS. How many instructions are required dynamically?
2. How many memory-data references will be executed?
3. What is the code size in bytes?

E2: H&P (2e) 3.4 pg. 217 and Example pg. 137, groups of 2 – 10 min

Problem

Consider an unpipelined machine with a 10ns clock time. For a particular workload, instructions take on average 4.4 clock cycles to execute. Now consider a pipelined version of this machine. Due to clock skew the machine adds 1ns of overhead to the clock. Assume that this overhead is fixed and each pipe stage is balanced and takes 10ns in the five stage pipeline. Plot the speedup of the pipelined machine versus the unpipelined machine as the number of pipeline stages is increased from five stages to 20 stages, considering only the impact of the pipelining overhead and assuming that the work can be evenly divided as the stages are increased (which is not generally true). Also plot the “perfect” speedup that would be obtained if there was no overhead.
E3: individual – 10 mins

Problem
Consider the following MIPS code fragments, each containing two instructions. For each code fragment identify the type of hazard that exists between the two instructions and the registers involved.

a.
LD R1, 0(R2)
DADD R3, R1, R2

b.
LD R1, 0(R2)
DADD R2, R4, R3

c.
MULT R1, R2, R3
MULT R4, R5, R6

d.
DADD R1, R2, R3
SD 2000(R0), R1

e.
DADD R1, R2, R3
SD 2000(R1), R4

D1: Discussion - groups of 4 - 15 min
For some of the compiler optimizations listed on Figure 2.24 page 124 do some of the following:

• Show examples of the optimization.
• Discuss situations where optimization may be useful.
• Discuss limitations and drawbacks, if any, of the optimization.
• Discuss the complexity of implementing the optimization in a real compiler.
Figure 2.24 is reproduced on the following page for convenience.

_Nigel Topham 2010, thanks to Marcelo Cintra._
<table>
<thead>
<tr>
<th>Optimization name</th>
<th>Explanation</th>
<th>Percentage of total number of optimizing transforms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-level</strong></td>
<td>At or near the source level; processor independent</td>
<td></td>
</tr>
<tr>
<td>Procedure integration</td>
<td>Replace procedure call by procedure body</td>
<td>N.M.</td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td>Within straight-line code</td>
<td></td>
</tr>
<tr>
<td>Common sub-expression elimination</td>
<td>Replace two instances of the same computation with one copy</td>
<td>18%</td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Replace instances of a variable with a constant</td>
<td>22%</td>
</tr>
<tr>
<td>Stack-height reduction</td>
<td>Rearrange expression tree to minimize resource usage</td>
<td>N.M</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td>Within a function</td>
<td></td>
</tr>
<tr>
<td>Global CSE</td>
<td>Same as CSE, but across a whole function</td>
<td>15%</td>
</tr>
<tr>
<td>Copy propagation</td>
<td>Replace instances of a variable A that has been assigned X, with X.</td>
<td>11%</td>
</tr>
<tr>
<td>Code motion</td>
<td>Move loop-invariant code out of a loop</td>
<td>16%</td>
</tr>
<tr>
<td>Induction variable elimination</td>
<td>Simplify/eliminate iterator computations in loops</td>
<td>2%</td>
</tr>
<tr>
<td><strong>Processor-dependent</strong></td>
<td>Depends on processor knowledge</td>
<td></td>
</tr>
<tr>
<td>Strength reduction</td>
<td>E.g. multiply to shift/add</td>
<td>N.M</td>
</tr>
<tr>
<td>Pipeline scheduling</td>
<td>Reorder instructions to improve pipeline performance</td>
<td>N.M.</td>
</tr>
<tr>
<td>Branch offset optimization</td>
<td>Choose shortest branch displacement that reaches target</td>
<td>N.M</td>
</tr>
</tbody>
</table>