COS 471A, COS 471B/ELE 375 Midterm Solution

Prof: David August
TAs: Jonathan Chang
Easwaran Raman
Mike Wawrzoniak
Fall 2005

Please write your answers clearly in the space provided. For partial credit, show all work. State all assumptions. You have 1 hour and 20 minutes for this exam. This midterm is closed book. Only one two-sided, handwritten 8.5x11 sheet is allowed. Put your name on every page. Write out and sign the Honor Code pledge before turning in the test. “I pledge my honor that I have not violated the Honor Code during this examination.”

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/150</td>
</tr>
<tr>
<td>2</td>
<td>/150</td>
</tr>
<tr>
<td>3</td>
<td>/300</td>
</tr>
<tr>
<td>4</td>
<td>/200</td>
</tr>
<tr>
<td>5</td>
<td>/200</td>
</tr>
<tr>
<td>Total</td>
<td>/1000</td>
</tr>
</tbody>
</table>

Name:

Course (circle one): COS471B/ELE375 COS471A

Honor Code:
1 WiMPY Precision IEEE 754

Define the WiMPY precision IEEE 754 floating point format to be:

\[
\begin{array}{c}
X \\
XXX \\
XXX
\end{array}
\]

Sign Exponent Mantissa

where each 'X' represents one bit. Convert each of the following decimal numbers to WiMPY floating point:

1. 0.0

\[
0.00000000
\]

2. -0.25

\[
-0.25 = (-0.1)_2 = -1.0 \times 2^{-2} \implies E = -2 + 3 = 1, N = 0000
\]

\[
10010000
\]

3. -0.0625

\[
-0.0625 = (-0.0001)_2 = -0.01 \times 2^{-2} \implies E = 0, N = 0100
\]

\[
10000100
\]

(Done same)

4. 0.005

The smallest number that can be represented:

\[
0.0001 \times 2^{-2} = (0.000001)_2 = 0.015625
\]

\[
0.00000000
\]

The closest number to 0.005 that we can represent is 0
2 Math is useful (occasionally)

We're going to look at how binary arithmetic can be unexpectedly useful. For this problem, all numbers will be 8-bit, signed, and in 2's complement.

In some architectures (such as the PowerPC), there is an instruction `adde rX=rY, rZ`, which performs the following:

\[ rX = rY + rZ + CA \]

where `CA` is the carry flag. There is also a negation instruction, `neg rX=rY` which performs:

\[ rX = 0 - rY \]

Both `adde` and `neg` set the carry flag. `gcc` will often use these instructions in the following sequence in order to implement a feature of the C language:

\[
\begin{align*}
\text{neg } r1 &= r0 \\
adde r2 &= r1, r0
\end{align*}
\]

Explain, simply, what the relationship between `r0` and `r2` is (Hint: `r2` has exactly two possible values), and what C operation it corresponds to. Be sure to show your reasoning.

\[
\begin{align*}
\text{Consider } \neg \ y_1 &= y_0 \\
\text{If } y_0 &= 00000000 \text{ (assuming 8 bits)} \\
\therefore y_1 &= 11111111 + 1 \\
&= 0 \quad \text{with } CA = 1. \\
\text{If } y_0 \neq 0, \ CA = 0 \text{ after neg, and } y_1 = -y_0
\end{align*}
\]

Thus:

\[
\begin{align*}
y_2 &= y_1 + y_0 + CA \\
&= -y_0 + y_0 + CA \\
&= CA. \quad (CA \text{ is the carry before addle})
\end{align*}
\]

Thus:

\[
\begin{align*}
y_2 &= 1 \quad \text{if } y_0 = 0 \\
&= 0, \quad \text{if } y_0 \neq 0 \\
0 \land y_2 &= (y_0 = 0)
\end{align*}
\]
3 Compilers are useful (occasionally)

A common loop in scientific programs is the so-called SAXPY loop:

```c
for (i=0; i<N; i++) {
    Y[i] = A*X[i] + Y[i]
}
```

1. For \( A=3 \) and \( N=100 \), convert the C code given above into MIPS assembly. You may assume that \( X \) and \( Y \) are 32-bit integer arrays. Try to avoid using pseudo-ops as they will hinder the next parts of this question. Also, avoid introducing false dependences. To aid your wonderful TAs as they grade your work, please use \( $s0 \) to hold the value of \( i \). You may also assume that \( X \) and \( Y \) begin at addresses \( 0x1000 \) and \( 0x2000 \), respectively.

   **Hint:** Since \( A=3=2^1 \), you can easily perform the multiplication without using the `mult` instruction.

   ```mips
   add $s0, $2000, $2010  # i = 0
   add $t0, $t0, $2000, 100  # n = 100
   loop:
   slt $t1, $t0, $t0  # i < n?
   beq $t1, $2000, exit  # if i >= n, end loop
   sll $t2, $t0, 2  # i * 4
   lw $t3, 0x2000($t2)  # Y[i]
   lw $t4, 0x1000($t2)  # X[i]
   add $t5, $t4, $t4  # 2 * X[i]
   add $t5, $t5, $t4  # 3 * X[i]
   add $t5, $t5, $t3  # 3 * X[i] + Y[i]
   sw $t5, 0x2000($t2)  # Store into Y[i]
   addi $s0, $s0, 1  # i + 1
   j loop
   exit:
   
   # static instructions = 13
   # Dynamic instructions = 2 + 11 * 100 + 2 * 100
   
   before loop
   
   # and now, after 100 iterations
   
   # static instructions = 13
   # Dynamic instructions = 2 + 11 * 100 + 2 * 100
   ```

2. How many static instructions (total instructions in memory) does the snippet have? How many dynamic instructions (total instructions executed) does the snippet have?

3. Suppose we wanted to do more than one SAXPY loop:

   ```c
   for (i=0; i<N; i++) {
       Y[i] = A*X[i] + Y[i]
   }
   for (i=0; i<N; i++) {
       Z[i] = A*X[i] + Z[i]
   }
   ```

   Using your computations from parts (1) and (2), how many dynamic and how many static instructions will this snippet have?

   ```plaintext
   # static instructions = 2 \times 13 - 1 = 25  \quad \text{(No need to repeat)}
   # Dynamic ins = 2 + 2 \times \left( 11 \times 100 + 2 \right) = 2207
   ```
4. There are two optimizations that can be performed to this code. The first is loop fusion, which combines two similar loops into one loop:

```c
for (i=0; i<\text{\textbackslash n}; i++) {
    Y[i] = A*X[i] + Y[i]
    Z[i] = A*X[i] + Z[i]
}
```

What would the dynamic and static instruction count be for this snippet of code? How much would you save over unoptimized code? You do not need to write out the assembly.

- Dynamic ins. before loop = 2
- A ins. inside loop = 11 + 6

Total = 19

Savings = \frac{25 - 19}{25} = 0.24

5. Another optimization we can do is common sub-expression elimination. This means that we can factor out repeated computation:

```c
for (i=0; i<\text{\textbackslash n}; i++) {
    T = A*X[i];
    Y[i] = T + Y[i]
    Z[i] = T + Z[i]
}
```

Convert this snippet into MIPS assembly. Assume that $Z$ starts at address 0x3000. Use $s1$ to store the value of $T$. How many dynamic and how many static instructions does this snippet have? How much do you save with constant sub-expression elimination (over loop fusion)?

6. Annotate your assembly from part (4) with data dependence arcs (flow, anti, and output - not control). What is the dependence height (the longest chain of dependences)?

The longest chain within one loop iteration is:

$\text{\textbackslash n}$

Dependence height = 5
4 Pipeline Performance

Consider a DLX machine with a 5 stage pipeline with a cycle time of 10ns. Assume that you are executing a program where a fraction, $f$, of all instructions immediately follow a load upon which they are dependent.

1. With forwarding enabled, what is the total execution time for $N$ instructions, in terms of $f$?

   When pipeline is filled,
   
   \[(1-f)N\] instructions take 1 cycle
   
   \[f N\] instructions take 2 cycles (including 1 cycle
   
   \[\text{Load-use stall})\]

   Total cycles: \[(1-f)N + 2fN + \frac{4}{N}\]

   \[\text{# cycles to fill the pipeline}\]

   Total time: \[\frac{N(1+f)+4}{10} (N(1+f)+4)\]

2. Consider a scenario where the MEM stage, along with its pipeline registers, needs 12ns. There are now two options: add another MEM stage so that there are MEM1 and MEM2 stages or increase the cycle time to 12ns so that the MEM stage fits within the new cycle time and the number of pipeline stages remain unaffected. For a program mix with the above characteristics, when is the first option better than the second. Your answer should be based on the value of $f$.

   Let us ignore the fill cycles to simplify calculations.

   Option 1: Time: \[\text{cycles: ((1-f)N + 3fN) x 10} = (1+2f)\times 10N\]

   \[\text{additional load-use delay}\]

   Option 2: Time: \[((1-f)N + 2fN) x 12\]

   Option 1 is better when \[(1+2f)\times 10 N < (1+f)\times 12 N\]

   \[10 + 20f \leq 12 + 12f\]

   \[f \leq \frac{1}{4}\]

3. Embedded processors have two different memory regions - a faster scratchpad memory and a slower normal memory. Assume that in the 6 stage machine (with MEM1 and MEM2 stages), there is a region of memory that is faster and for which the correct value is obtained at the end of the MEM1 stage itself while the rest of the memory needs both MEM1 and MEM2 stages. For the sake of simplicity assume that there are two load instructions \texttt{load.fast} and \texttt{load.slow} that indicate which memory region is accessed.

   If 40% of the fraction $f$ mentioned above get their value from the fast memory, how does the answer to the previous question change?

   Option 1: Time: \[10\left((1-f)N + 0.4 \times 2fN + 0.6 \times 5fN\right)\times 10\]

   \[= (1+1.6f)\times 10 N\]

   Option 1 is better when \[(1+1.6f)\times 10 N \leq (1+f)\times 12 N\]

   \[10N + 16f \leq 12N + 12f\]

   \[4f \leq 2\]

   \[f \leq \frac{1}{2}\]
5 Processor in Trouble

On the figure on the next page several lines are marked with an "X". For each one:

- Describe in words the negative consequence of cutting this line relative to the working, unmodified processor.
- Provide a snippet of code that will fail.
- Provide a snippet of code that will still work.

1. - Cannot write to register file. \Rightarrow R-type and lw will fail.

   Example code snippet that will fail
   \[\text{add } \$s1, \$s2, \$s3\]

   Code snippet that will not fail
   \[\text{sw } \$s1, 0(\$s2)\]

2. - Forwarding of an operand fails.

   Snippet that will fail
   \[\text{add } \$s1, \$t0, \$t1\]
   \[\text{add } \$s1, \$s1, \$s1\]

   Snippet that will not fail
   \[\text{add } \$s1, \$t0, \$t1\]
   \[\text{add } \$s1, \$t2, \$s1\]

   \(\Rightarrow\) Second operand forwarded correctly

3. - Jumping to a branch target will not work.

   Code that fails
   \[\text{add } \$s1, \$2e80, 2\]
   \[\text{add } \$s2, \$2e10, 2\]
   \[\text{beq } \$s1, \$s2, \text{exit}\]

   Code that will still work
   \[\text{add } \$s1, \$2e80, 10\]
   \[\text{add } \$s2, \$2e10, 20\]
   \[\text{beq } \$s1, \$s2, \text{exit}\]