EE557--FALL 2001

MIDTERM 2

Open books

Q1: /16
Q2: /12
Q3: /8
Q4: /8
Q5: /8
Q6: /8

TOTAL: /60

Grade: /25
QUESTION 1 (Tomasulo with ROB) 16 points

Consider the following loop.

```
LOOP   LD F0,0(R1)
       LD F2,-8(R1)
       LED F0,F2
       BFPT LESS
       SD 0(R1), F2
       SD -8(R1), F0
       LESS  SUBI R1,R1, 8
       BNEZ R1, LOOP
```

LED is a DLX instruction which compares FP values in F0 and F2. If F0 is less than or equal to F2, a FP status register is set to 1, otherwise it is set to 0. BFPT is also a DLX instruction which reads the value of the FP status register; if it is 1, then it branches. FP operands have size 8 bytes (double precision).

Concerning hardware behavior, please use the assumptions in the example in the class notes, with the following modifications:

1. One single LD/ST unit. The LD/ST unit takes 1 clock for address computation and 1 clock for the cache access if it hits. If the access misses the cache access takes 10 cycles instead of 1. The LD/ST unit (and RS) is busy until (not including) the cycle after write_result (LD) or exec_complete (SD).

2. One single integer/branch unit, which executes BFPT, SUBI and BNEZ in the program. The unit (and RS) is busy until (not including) the cycle after exec_complete (branches) or after write_result (SUBI). Execution takes one clock.

3. One single FP unit. The unit (and RS) is busy until (not including) the cycle after write_result (which updates the FP status register). The execution of the compare takes 8 cycles.

In all cases, we issue one instruction at a time and both the integer/branch unit and the FP unit have 4 reservation stations. There is a single CDB for both integer and FP operands. The RS are busy starting at issue, whereas the execution units are busy starting at exec_start.

The cache is blocking (1 access at a time), direct mapped and is empty at the start of the loop. Assume that the two memory locations accessed in the first iteration are in the same memory block.

Branches are predicted when they are issued and the order of instruction issues is kept in a ROB which is also used for register renaming. If the prediction was incorrect, it is corrected in the cycle following the execution of the branch (i.e. ROB is flushed and, the correct instruction is fetched so that it is available for issue in the next cycle.)
BFPT is predicted not taken in the first iteration.

If two instructions are ready to execute in the same cycle in an execution unit, then the older instruction executes first.

Show the schedule of the first iteration by filling the instruction status table. Each entry in the table shows the clock cycle when a given instruction went through a specific phase of processing.

1. Case 1. (8 points) 1 load buffer and 1 store buffer

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Exec start</th>
<th>Exec. complete</th>
<th>Write Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>LD F2, -8(R1)</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>LED F0, F2</td>
<td>15</td>
<td>18</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>BFPT LESS</td>
<td>16</td>
<td>27</td>
<td>27</td>
<td>-</td>
</tr>
<tr>
<td>SD 0(R1), F2</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>-</td>
</tr>
<tr>
<td>SD -8(R1), F0</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>-</td>
</tr>
<tr>
<td>SUBI R1, R1, #8</td>
<td>21</td>
<td>22</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>BNEZ R1, LOOP</td>
<td>22</td>
<td>24</td>
<td>24</td>
<td>-</td>
</tr>
</tbody>
</table>

2. Case 2. (8 points) 4 load buffers and 4 store buffers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Exec start</th>
<th>Exec. complete</th>
<th>Write Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>LD F2, -8(R1)</td>
<td>2</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LED F0, F2</td>
<td>3</td>
<td>17</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>BFPT LESS</td>
<td>4</td>
<td>26</td>
<td>26</td>
<td>-</td>
</tr>
<tr>
<td>SD 0(R1), F2</td>
<td>5</td>
<td>17</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>SD -8(R1), F0</td>
<td>6</td>
<td>19</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>SUBI R1, R1, #8</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>BNEZ R1, LOOP</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>-</td>
</tr>
</tbody>
</table>
QUESTION 2(software pipeline) 12 points

Apply software pipelining to the loop in question 1 so that no true data dependencies remain in the body of the loop (pipeline just enough to achieve that). Show the new loop body as well as the startup code (the code before the first iteration) and the clean up code (the code after the last iteration).

Answer: Unfortunately it is not possible to have a loop free of true dependencies without using additional registers.

Solution 1. No new register. Still one dependency (on FP status register)

<table>
<thead>
<tr>
<th>LD F0,0(R1)</th>
<th>LED F0,F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F2,-8(R1)</td>
<td>BFPT LESS</td>
</tr>
<tr>
<td></td>
<td>SD 0(R1), F2</td>
</tr>
<tr>
<td></td>
<td>SD -8(R1), F0</td>
</tr>
</tbody>
</table>

Startup code

LD F0, 0(R1)
LD F2, -8(R1)

loop:

LOOP  LED F0,F2
      BFPT LESS
      SD 0 (R1), F2
      SD -8 (R1), F0

LESS  SUBI R1,R1,8
      LD F0, 0(R1)
      LD F2, -8(R1)
      BNEZ R1, LOOP

cleanup code

LED F0,F2
BFPT EXIT
SD 8 (R1), F2
SD 0 (R1), F0
EXIT
Solution 2: No true dependency. Two more registers.

<table>
<thead>
<tr>
<th>LD F0,0(R1)</th>
<th>LED F0,F2</th>
<th>BFPT LESS</th>
<th>SD 0(R1), F6</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F2,-8(R1)</td>
<td>LED F0,F2</td>
<td>ADDDI F4,F0,0</td>
<td>SD -8(R1), F4</td>
</tr>
<tr>
<td></td>
<td>ADDDI F6,F2,0</td>
<td>ADDDI F4,F0,0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADDDI F6,F2,0</td>
<td>ADDDI F6,F2,0</td>
<td></td>
</tr>
</tbody>
</table>

Startup code

LD F0, 0(R1)
LD F2, -8(R1)
LED F0,F2
ADDDI F4,F0,0
ADDDI F6,F2,0
SUBI R1,R1,8
LD F0, 0(R1)
LD F2, -8(R1)

loop:

LOOP BFPT LESS
SD 0(R1), F6
SD 8(R1), F4
LESS LED F0,F2
ADDDI F4,F0,0
ADDDI F6,F2,0
SUBI R1,R1,8
LD F0, 0(R1)
LD F2, -8(R1)
BNEZ R1, LOOP

cleanup code

LOOP BFPT LESS
SD 0(R1), F6
SD 8(R1), F4
LESS LED F0,F2
LOOP BFPT EXIT
SD 0(R1), F2
SD -8(R1), F0
EXIT
QUESTION 3 8 points

An \((m,n)\) branch prediction scheme uses \(m\) bits of global history and \(n\) bit history counters. Consider again the loop in problem 1

Assume that BFPT is taken roughly half of the time. R1 is 256 at the beginning of the loop. All bits used in the prediction are zero at the beginning (zero means not taken). ASSUME THAT THE LOOP IS EXECUTED ONLY ONCE.

1. How many times is BNEZ executed?

-----------------------------32 times-----------------------------

2. What will be the misprediction rate for BNEZ with a \((0,1)\) predictor?

-----------------------------------2/32-----------------------------------

3. What will be the misprediction rate for BNEZ with a \((0,2)\) predictor?

---------------------------------3/32-----------------------------------

4. What will be the misprediction rate for BNEZ with a \((1,2)\) predictor?

---------------------------------5/32-----------------------------------
QUESTION 4 8 points

1. Assume that we have a predicated (conditional) store_double instruction (CSD) (which stores only if the FP status register is 1, otherwise has the same format and same semantic as a regular store), rewrite the code of problem 1 without using any new instruction except for CSD, so that the outcome of the program is the same.

Answer:

```
LOOP    LD F0,0(R1) 
        LD F2,-8(R1) 
        LED F2,F0 
        CSD 0(R1), F2 
        CSD -8(R1), F0 
        SUBI R1,R1, 8 
        BNEZ R1, LOOP 
```

2. Do you think it is a good idea to use conditional stores in this program? Please explain in the context of the architecture of problem 1.

Answer:

We have replaced one instruction with two speculative instructions. Let f be the fraction of times that the BFPT instruction branches. In the original code we execute 1 + 2f instructions on the average. In this code we execute 2 instructions always.

Thus, to be effective, the new code must be such that 1+2f > 2 or f > .5.

Unless it can be established that the two stores are executed more than 50% of the time, the new code is not a good idea.
QUESTION 5 8 points

Read problem 4.24 in your book. Then answer the following questions.

1. In your notes, the execution of Tomasulo algorithm is shown, cycle by cycle. Each frame is titled with a cycle number. The event described in problem 4.24 happens in one of the cycles. Please give the cycle number and the register involved.

Answer:

   cycle 4/ register F6

2. How would you propose to solve this problem SIMPLY. You can assume that the issue stage is clocked twice as fast as the rest of the machine,

Answer:

If we can clock the issue logic twice as fast, then the instruction can be issued in the first half cycle with the tag of the ROB buffer and then, in the second half cycle, the RS can grab the value off the CDB
QUESTION 6 8 points

First-level instruction caches are often direct-mapped, not only because direct-mapped caches are faster on a hit but also because they are better at handling loops than set-associative caches.

Let \( K \) be the size of the loop code in bytes, \( C \) be the size of the direct-mapped cache in bytes, and \( B \) be the block size in bytes. Define \( K' = B \times \text{ceiling}(K/B) \). \( K' \) is the total size of the instruction memory blocks holding the loop code. The instruction cache is accessed one instruction at a time.

Assume that the first instruction of the loop is stored in word 0 of blockframe 0 of the direct-mapped cache. Also consider the second iteration of the loop.

Find the instruction miss rate as a function of \( K' \), \( K \), \( C \), and \( B \).

Hint: Consider first \( K \leq C \), then \( C < K \leq 2C \), and finally \( 2C < K \).

1. \( K \leq C \):(2pts)

In this case the entire loop holds in cache and the miss rate is 0.

2. \( C < K \leq 2C \):(4pts)

In this case the first \((K' - C)/4\) instructions of the loop have been wiped out of the cache by the first iteration. We first get a total of \((K' - C)/B\) misses for the top instructions in the 2nd iteration plus \((K' - C)/B\) misses for the last instructions of the second iteration. The total number of accesses is \(K/4\).

Thus the miss rate is: \((2(K' - C)/B)/(K/4) = 8(K' - C)/(BK)\)

3. \( 2C < K \):(2pts)

Then the cache misses on every block in the loop.

There are \(K'/B\) blocks total and the number of instructions is \(K/4\).

Thus the miss rate is \((K'/B)/(K/4) = 4K'/BK\)