Problem 3.1: Superscalar Processor

Problem 3.1.A

Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10

<table>
<thead>
<tr>
<th>Instr #</th>
<th>Instruction</th>
<th>Dest</th>
<th>Src1</th>
<th>Src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>LD F2, 0(R2)</td>
<td>T1</td>
<td>R2</td>
<td>0</td>
</tr>
<tr>
<td>I2</td>
<td>LD F3, 0(R3)</td>
<td>T2</td>
<td>R3</td>
<td>0</td>
</tr>
<tr>
<td>I3</td>
<td>FMUL F4, F2, F3</td>
<td>T3</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>I4</td>
<td>LD F2, 4(R2)</td>
<td>T4</td>
<td>R2</td>
<td>4</td>
</tr>
<tr>
<td>I5</td>
<td>LD F3, 4(R3)</td>
<td>T5</td>
<td>R3</td>
<td>4</td>
</tr>
<tr>
<td>I6</td>
<td>FMUL F5, F2, F3</td>
<td>T6</td>
<td>T4</td>
<td>T5</td>
</tr>
<tr>
<td>I7</td>
<td>FMUL F6, F4, F5</td>
<td>T7</td>
<td>T3</td>
<td>T6</td>
</tr>
<tr>
<td>I8</td>
<td>FADD F4, F4, F5</td>
<td>T8</td>
<td>T3</td>
<td>T6</td>
</tr>
<tr>
<td>I9</td>
<td>FMUL F6, F4, F5</td>
<td>T9</td>
<td>T8</td>
<td>T6</td>
</tr>
<tr>
<td>I10</td>
<td>FADD F1, F1, F6</td>
<td>T10</td>
<td>F1</td>
<td>T9</td>
</tr>
</tbody>
</table>

Renaming table

<table>
<thead>
<tr>
<th></th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
<th>I8</th>
<th>I9</th>
<th>I10</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T10</td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T1</td>
<td>T4</td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T6</td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T7</td>
</tr>
</tbody>
</table>

http://inst.eecs.berkeley.edu/~cs152/sp11
Problem 3.1.B

See the following table.

Problem 3.1.C

See the following table.
<table>
<thead>
<tr>
<th>Slot</th>
<th>Instruction</th>
<th>Cycle instruction entered ROB</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>dst</th>
<th>Cycle written back to ROB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>src1</td>
<td>cycle available</td>
<td>Src2</td>
<td>cycle available</td>
</tr>
<tr>
<td>T1</td>
<td>LD F2, 0(R2)</td>
<td>1</td>
<td>C</td>
<td>1</td>
<td>R2</td>
<td>1</td>
</tr>
<tr>
<td>T2</td>
<td>LD F3, 0(R3)</td>
<td>1</td>
<td>C</td>
<td>1</td>
<td>R3</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>FMUL F4, F2, F3</td>
<td>2</td>
<td>F2</td>
<td>6</td>
<td>F3</td>
<td>7</td>
</tr>
<tr>
<td>T4</td>
<td>LD F2, 4(R2)</td>
<td>2</td>
<td>C</td>
<td>2</td>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>T5</td>
<td>LD F3, 4(R3)</td>
<td>3</td>
<td>C</td>
<td>3</td>
<td>R3</td>
<td>3</td>
</tr>
<tr>
<td>T6</td>
<td>FMUL F5, F2, F3</td>
<td>3</td>
<td>F2</td>
<td>8</td>
<td>F3</td>
<td>9</td>
</tr>
<tr>
<td>T7</td>
<td>FMUL F6, F4, F5</td>
<td>4</td>
<td>F4</td>
<td>12</td>
<td>F5</td>
<td>14</td>
</tr>
<tr>
<td>T8</td>
<td>FADD F4, F4, F5</td>
<td>4</td>
<td>F4</td>
<td>12</td>
<td>F5</td>
<td>14</td>
</tr>
<tr>
<td>T9</td>
<td>FMUL F6, F4, F5</td>
<td>5</td>
<td>F4</td>
<td>18</td>
<td>F5</td>
<td>14</td>
</tr>
<tr>
<td>T10</td>
<td>FADD F1, F1, F6</td>
<td>5</td>
<td>F1</td>
<td>5</td>
<td>F6</td>
<td>23</td>
</tr>
<tr>
<td>T11</td>
<td>ADD R2, R2, 8</td>
<td>6</td>
<td>R2</td>
<td>6</td>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>T12</td>
<td>ADD R3, R3, 8</td>
<td>6</td>
<td>R3</td>
<td>6</td>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>T13</td>
<td>ADD R4, R4, -1</td>
<td>7</td>
<td>R4</td>
<td>7</td>
<td>C</td>
<td>7</td>
</tr>
<tr>
<td>T14</td>
<td>BNEZ R4, loop</td>
<td>7</td>
<td>R4</td>
<td>11</td>
<td>C</td>
<td>Loop</td>
</tr>
<tr>
<td>T15</td>
<td>LD F2, 0(R2)</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>R2</td>
<td>9</td>
</tr>
<tr>
<td>T16</td>
<td>LD F3, 0(R3)</td>
<td>8</td>
<td>C</td>
<td>8</td>
<td>R3</td>
<td>10</td>
</tr>
<tr>
<td>T17</td>
<td>FMUL F4, F2, F3</td>
<td>9</td>
<td>F2</td>
<td>14</td>
<td>F3</td>
<td>15</td>
</tr>
<tr>
<td>T18</td>
<td>LD F2, 4(R2)</td>
<td>9</td>
<td>C</td>
<td>9</td>
<td>R2</td>
<td>9</td>
</tr>
<tr>
<td>T19</td>
<td>LD F3, 4(R3)</td>
<td>10</td>
<td>C</td>
<td>10</td>
<td>R3</td>
<td>10</td>
</tr>
<tr>
<td>T20</td>
<td>FMUL F5, F2, F3</td>
<td>10</td>
<td>F2</td>
<td>16</td>
<td>F3</td>
<td>17</td>
</tr>
<tr>
<td>T21</td>
<td>FMUL F6, F4, F5</td>
<td>11</td>
<td>F4</td>
<td>20</td>
<td>F5</td>
<td>22</td>
</tr>
<tr>
<td>T22</td>
<td>FADD F4, F4, F5</td>
<td>11</td>
<td>F4</td>
<td>20</td>
<td>F5</td>
<td>22</td>
</tr>
<tr>
<td>T23</td>
<td>FMUL F6, F4, F5</td>
<td>12</td>
<td>F4</td>
<td>26</td>
<td>F5</td>
<td>22</td>
</tr>
<tr>
<td>T24</td>
<td>FADD F1, F1, F6</td>
<td>12</td>
<td>F1</td>
<td>27</td>
<td>F6</td>
<td>31</td>
</tr>
<tr>
<td>T25</td>
<td>ADD R2, R2, 8</td>
<td>13</td>
<td>R2</td>
<td>13</td>
<td>C</td>
<td>13</td>
</tr>
<tr>
<td>T26</td>
<td>ADD R3, R3, 8</td>
<td>13</td>
<td>R3</td>
<td>13</td>
<td>C</td>
<td>13</td>
</tr>
<tr>
<td>T27</td>
<td>ADD R4, R4, -1</td>
<td>14</td>
<td>R4</td>
<td>14</td>
<td>C</td>
<td>14</td>
</tr>
<tr>
<td>T28</td>
<td>BNEZ R4, loop</td>
<td>14</td>
<td>C</td>
<td>Loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3.1.D

15, 16, 17, 18, 19, 110 (see registers in blue in previous table)

27 cycles.

Problem 3.1.E

The behavior should repeat- should be obvious from the dependency graph (DAG) in Problem 3.1.D.

Problem 3.1.F

Yes/No -----------------------------------------------

An extra FP multiplier does not really help, because all FMUL instructions execute as soon as operands are ready. But an extra memory port helps, because dispatch of I4, I5 was delayed waiting for memory port.

Problem 3.1.G

The answer is 4 cycles.

Since the integer index/counter additions are relatively short, they can proceed to generate values for different loop iterations and load all values from memory saving them to renamed registers. After a large number of iterations, many iterations of the loop will be running in parallel. With each iteration depending only on the previous iteration for the FMULT on F1 value. Hence, the number of cycles is the latency of FMULT (3 + 1 cycle for writeback). At steady state, one iteration can complete every 4 cycles.
Problem 3.2: Register Renaming and Static vs. Dynamic Scheduling

### Problem 3.2.A

The following table shows the cycles in which instructions are decoded, issued, and written back. It starts with cycle 0 in which the first load has been decoded (and thus has just entered the issue stage). It is assumed that all instructions prior to the first load have already been completed. Although not shown below, there is a buffer that holds instructions that are waiting in the issue stage. Since there is no bypassing, an instruction must complete the write-back stage before a dependent instruction can issue. For example, as shown in the table, the second load is issued in cycle 2, executes for 2 cycles, and is written back in cycle 4. Thus, any instruction that depends on the load can issue no earlier than cycle 5.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decoded Instruction (Enters Issue)</th>
<th>Issued Instruction (Enters Execute)</th>
<th>WB Cycle For Issued Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L.S F0, 0(R1)</td>
<td>Stall</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L.S F1, 0(R2)</td>
<td>L.S F0, 0(R1)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>MUL.S F0, F0, F1</td>
<td>L.S F1, 0(R2)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>L.S F2, 0(R3)</td>
<td>Stall</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>L.S F3, 0(R4)</td>
<td>Stall</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>MUL.S F2, F2, F3</td>
<td>MUL.S F0, F0, F1</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>ADD.S F0, F0, F2</td>
<td>L.S F2, 0(R3)</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>S.S F0, 0(R5)</td>
<td>L.S F3, 0(R4)</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>Stall</td>
<td>Stall</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>Stall</td>
<td>Stall</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>MUL.S F2, F2, F3</td>
<td>Stall</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>Stall</td>
<td>Stall</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>Stall</td>
<td>Stall</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>Stall</td>
<td>Stall</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>Stall</td>
<td>Stall</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>ADD.S F0, F0, F2</td>
<td>Stall</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>Stall</td>
<td>Stall</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>Stall</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>S.S F0, 0(R5)</td>
<td>Stall</td>
<td></td>
</tr>
</tbody>
</table>

The number of cycles from the issue of the first load instruction (in cycle 1) until the issue of the final store instruction (in cycle 18) is 18 cycles, inclusive.
The new code sequence is given below. Originally there were two stall cycles after the second load instruction. Now these cycles will be filled by the third and fourth load instructions. The remaining instructions cannot be reordered due to data dependencies (except for the two multiply instructions, although doing that would hurt performance).

```
L S F0, 0 (R1)
L S F1, 0 (R2)
L S F2, 0 (R3)
L S F3, 0 (R4)
MUL.S F0, F0, F1
MUL.S F2, F2, F3
ADD.S F0, F0, F2
S . S F0, 0 (R5)
```

The following table shows the cycles in which the instructions in the above sequence are decoded, issued, and written back.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decoded Instruction (Enters Issue)</th>
<th>Issued Instruction (Enters Execute)</th>
<th>WB Cycle For Issued Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L.S F0, 0 (R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L.S F1, 0 (R2)</td>
<td>L.S F0, 0 (R1)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>L.S F2, 0 (R3)</td>
<td>L.S F1, 0 (R2)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>L.S F3, 0 (R4)</td>
<td>L.S F2, 0 (R3)</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>MUL.S F0, F0, F1</td>
<td>L.S F3, 0 (R4)</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>MUL.S F2, F2, F3</td>
<td>MUL.S F0, F0, F1</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>ADD.S F0, F0, F2</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>S.S F0, 0 (R5)</td>
<td>MUL.S F2, F2, F3</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ADD.S F0, F0, F2</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>S.S F0, 0 (R5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The number of cycles from the issue of the first load instruction (in cycle 1) to the issue of the final store instruction (in cycle 15) is 15 cycles, inclusive. Using static scheduling has enabled us to reduce the execution time of the sequence by 17%.

**Problem 3.2.C**

Fewer Registers

The new code sequence using only two floating-point registers is shown below. It is assumed that R6 holds the address of a memory location that can be used to store temporary values.

```
L.S   F0, 0(R1)
L.S   F1, 0(R2)
MUL.S F0, F0, F1
L.S   F1, 0(R3)
S.S   F0, 0(R6)
L.S   F0, 0(R4)
MUL.S F0, F0, F1
L.S   F1, 0(R6)
ADD.S F0, F0, F1
S.S   F0, 0(R5)
```

The following table shows the cycles in which the instructions in the above sequence are decoded, issued, and written back. For this problem, a store instruction is needed in the middle of the instruction sequence in order to spill a register. Although not explicitly stated in the problem, stores have the same latency as loads (two cycles), since they use the same functional unit. However, if you assume a different latency in your solutions, that is acceptable as long as you state it. Because the result of the store is not needed for several cycles after it completes (when the load restores the spilled value), it would take a very long latency for store instructions in order to delay the last load. We don’t have to worry about WAR hazards in the above sequence because instructions are issued in-order. Note that we can no longer execute the four original loads in sequence as in 3.2.B because of the lack of available registers. We can, however, execute the third load before saving the intermediate value from the first MUL instruction.
<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decoded Instruction (Enters Issue)</th>
<th>Issued Instruction (Enters Execute)</th>
<th>WB Cycle For Issued Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L.S F0, 0(R1)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L.S F1, 0(R2)</td>
<td>L.S F0, 0(R1)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>MUL.S F0, F0, F1</td>
<td>L.S F1, 0(R2)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>L.S F1, 0(R3)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S.S F0, 0(R6)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>L.S F0, 0(R4)</td>
<td>MUL.S F0, F0, F1</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>MUL.S F0, F0, F1</td>
<td>L.S F1, 0(R3)</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>L.S F1, 0(R6)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADD.S F0, F0, F1</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S.S F0, 0(R5)</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>S.S F0, 0(R6)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>L.S F0, 0(R4)</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>MUL.S F0, F0, F1</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>L.S F1, 0(R6)</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>ADD.S F0, F0, F1</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>S.S F0, 0(R5)</td>
<td></td>
</tr>
</tbody>
</table>

The number of cycles from the issue of the first load instruction (in cycle 1) to the issue of the final store instruction (in cycle 22) is 22 cycles, inclusive. Using only two floating-point registers results in a severe performance hit.
The table below shows the cycles in which the instructions in the original code sequence are decoded, issued, and written back on the single-issue machine with register renaming and out-of-order issue. The table also contains the rename table for the architectural registers.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decoded/Renamed Instruction (Enters Issue)</th>
<th>Rename</th>
<th>Issued Instruction (Enters Execute)</th>
<th>WB Cycle For Issued Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L.S T0, 0(R1)</td>
<td>T0</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L.S T1, 0(R2)</td>
<td>T0 T1</td>
<td>L.S T0, 0(R1)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>MUL.S T2, T0, T1</td>
<td>T2 T1</td>
<td>L.S T1, 0(R2)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>L.S T3, 0(R3)</td>
<td>T2 T1 T3</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L.S T4, 0(R4)</td>
<td>T2 T1 T3 T4</td>
<td>L.S T3, 0(R3)</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>MUL.S T5, T3, T4</td>
<td>T2 T1 T5 T4</td>
<td>MUL.S T2, T0, T1</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>ADD.S T6, T2, T5</td>
<td>T6 T1 T5 T4</td>
<td>L.S T4, 0(R4)</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>S.S T6, 0(R5)</td>
<td>T6 T1 T5 T4</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>MUL.S T5, T3, T4</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>ADD.S T6, T2, T5</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>S.S T6, 0(R5)</td>
<td></td>
</tr>
</tbody>
</table>

The number of cycles from the issue of the first load instruction (in cycle 1) to the issue of the final store instruction (in cycle 17) is 17 cycles, inclusive. This is one cycle better than executing this code on an in-order machine but not quite as good as the performance of the optimized code in 3.2.B, which only required 15 cycles. The difference in performance between the statically scheduled code and the dynamically scheduled code can be attributed to the fact that only a single instruction can be decoded at a time, which limits the hardware’s ability to find independent instructions to issue. The optimized version of the code from 3.2.B executing on this machine would not improve in performance over executing on an in-order machine – it would still take 15 cycles.
Note, that in cycle 5, we would get better performance if we issued the final load instruction rather than the MUL instruction. The machine doesn’t know that, so it issues the instruction that entered the ROB first.

**Problem 3.2.E**

Effect of Register Spills

The table below shows the cycles in which the instructions in the original code sequence are decoded, issued, and written back on the single-issue machine with register renaming and out-of-order issue.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Decoded/ Renamed Instruction (Enters Issue)</th>
<th>Rename</th>
<th>Issued Instruction (Enters Execute)</th>
<th>WB Cycle For Issued Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>L.S T0, 0(R1)</td>
<td>T0</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L.S T1, 0(R2)</td>
<td>T0</td>
<td>L.S T0, 0(R1)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>MUL.S T2, T0, T1</td>
<td>T2</td>
<td>L.S T1, 0(R2)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>L.S T3, 0(R3)</td>
<td>T2</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S.S T2, 0(R6)</td>
<td>T2</td>
<td>L.S T3, 0(R3)</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>L.S T4, 0(R4)</td>
<td>T4</td>
<td>MUL.S T2, T0, T1</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>MUL.S T5, T4, T3</td>
<td>T5</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L.S T6, 0(R6)</td>
<td>T5</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADD.S T7, T5, T6</td>
<td>T7</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S.S T7, 0(R5)</td>
<td>T7</td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>S.S T2, 0(R6)</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>L.S T4, 0(R4)</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>L.S T6, 0(R6)</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>MUL.S T5, T4, T3</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td>ADD.S T7, T5, T6</td>
<td>21</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td>S.S T7, 0(R5)</td>
<td>24</td>
</tr>
</tbody>
</table>
It now takes 22 cycles between issue of the first load instruction and issue of the last store instruction. That is the same performance as 3.2.C, and much worse than 3.2.D.

We managed to execute two instructions out of order, but we still couldn’t beat the in-order performance. The problem lies with the fact that we had to wait for the first store to issue before we could continue with the program. This is directly linked to having only two registers, thus having to store intermediate values.
Problem 3.3: Branch Prediction

loop:
    LW  R4, 0(R3)
ADDI R3, R3, 4
SUBI R1, R1, 1
b1:
    BEQZ R4, b2
ADDI R2, R2, 1
b2:
    BNEZ R1, loop

Figure 3.3-A: BP bits state diagram

Problem 3.3.A

Program

R2 contains the number of non-zero entries in the first n elements of array p.

Problem 3.3.B

2-bit branch prediction

There are 7 mispredicts (shown in italics).

<table>
<thead>
<tr>
<th>System State</th>
<th>Branch Predictor</th>
<th>Branch Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R3/R4</td>
<td>b1 bits</td>
</tr>
<tr>
<td>PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b1</td>
<td>4/1</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>4/1</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>8/0</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>8/0</td>
<td>11</td>
</tr>
<tr>
<td>b1</td>
<td>12/1</td>
<td>11</td>
</tr>
<tr>
<td>b2</td>
<td>12/1</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>16/0</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>16/0</td>
<td>11</td>
</tr>
<tr>
<td>b1</td>
<td>20/1</td>
<td>11</td>
</tr>
<tr>
<td>b2</td>
<td>20/1</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>24/0</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>24/0</td>
<td>11</td>
</tr>
<tr>
<td>b1</td>
<td>28/1</td>
<td>11</td>
</tr>
<tr>
<td>b2</td>
<td>28/1</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>32/0</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>32/0</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3.3-1
Problem 3.3.C  

Branch prediction with one global history bit

There are 9 mispredicts (shown in italics).

<table>
<thead>
<tr>
<th>PC</th>
<th>R3/R4</th>
<th>history bit</th>
<th>Branch Predictor</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>b1 bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>set 0</td>
<td>set 1</td>
</tr>
<tr>
<td>b1</td>
<td>4/1</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>4/1</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>8/0</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>8/0</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>b1</td>
<td>12/1</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>b2</td>
<td>12/1</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>16/0</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>16/0</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>b1</td>
<td>20/1</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>b2</td>
<td>20/1</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>24/0</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>24/0</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>b1</td>
<td>28/1</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>b2</td>
<td>28/1</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b1</td>
<td>32/0</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2</td>
<td>32/0</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3.3-2
Problem 3.3.D  
Branch prediction with two global history bits

There are 7 mispredicts (shown in italics).

<table>
<thead>
<tr>
<th>System State</th>
<th>Branch Predictor</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC R3/R4</td>
<td>b1 bits</td>
<td>b2 bits</td>
</tr>
<tr>
<td></td>
<td>bits</td>
<td>set 00</td>
</tr>
<tr>
<td>b1 4/1</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b2 4/1</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>b1 8/0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2 8/0</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b1 12/1</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b2 12/1</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>b1 16/0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2 16/0</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b1 20/1</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b2 20/1</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>b1 24/0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2 24/0</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b1 28/1</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>b2 28/1</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>b1 32/0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>b2 32/0</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.3-3
The first thing to notice is that the more history bits we have, the longer it takes to get any correct prediction since we have to “train” the predictor. These start-up costs go up as the number of history bits increase.

Another thing to notice is that the single history bit does not help at all (even after we get into a steady-state phase). In both the single history bit and no history cases, the b2 branch is predicted correctly once we get past the start-up phase (since b2 is always taken). The single bit of history does not help since this history is too “nearsighted”. The second history bit captures the alternating pattern of the b1 branch, and hence does not mispredict once it gets past the start-up phase. For large n then, the 2-bit history predictor is the best.

The final point of observation is that all the predictors mispredict the fall-through case (last b2 branch).

When the input is random, no prediction scheme will help predict whether b1 is taken or not. All three schemes will eventually predict b2 as always taken. However, the more history bits are used, the more sets need to be trained to predict the always taken for b2. Thus, the more history bits used, the more mispredicts of branch b2 will occur initially. The answer does not depend on the size of n. However, as n gets large, the start-up costs become insignificant among the three schemes.

The moral of the problem is: history bits are useful if there is a pattern among a sequence of branches. The longer this pattern is, the more history bits are needed to be able to recognize this pattern. If the pattern is not recognized, then global history bits can hurt because it take longer to train the branches that can be predicted correctly.
Problem 3.4: Importance of Features

For the following snippets of code, select the single architectural feature that will most improve the performance of the code. Explain your choice, including description of why the other features will not improve performance as much and your assumptions about the machine design. The features you have to choose from are: out-of-order issue with renaming, branch prediction, and superscalar execution. Loads are marked whether they hit or miss in the cache.

**Problem 3.4A**

```
ADD.D F0, F1, F8
ADD.D F2, F3, F8
ADD.D F4, F5, F8
ADD.D F6, F7, F8
```

Superscalar because the instructions have no dependencies so they could be issued in parallel (superscalar could deliver speedup).

The other two techniques will waste hardware because they will offer no speedup. Out-of-Order with renaming will not help because there are no dependencies. Branch prediction is useless since there are no branches.

Circle one:
- Out-of-Order Issue with Renaming
- Branch Prediction
- Superscalar

**Problem 3.4B**

```
loop:  ADD R3 R4 R0
       LD R4, 8(R4) # cache hit
       BNEQZ R4, LOOP
```

Branch prediction is necessary with a tight loop to prevent bubbles in the pipeline.

Superscalars will be limited not only by the branches, but also the WAR and RAW hazards. They will limit how much ILP it can achieve.

Out-of-order with renaming will be limited by the branches. Renaming will take care of the WAR hazard, but the RAW hazard will still limit how much improvement is possible.
Problem 3.4.C

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Notes</th>
<th>Out-of-order with renaming will let the third, fourth, and fifth instruction run while the cache miss is being handled. When the miss completes it only needs to do the second and sixth instruction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1 0(R2)</td>
<td># cache miss</td>
<td></td>
</tr>
<tr>
<td>ADD R2 R1 R1</td>
<td></td>
<td>Branch prediction won’t help with getting around the cache miss and there are enough hazards to limit a superscalar.</td>
</tr>
<tr>
<td>LD R1 0(R3)</td>
<td># cache hit</td>
<td></td>
</tr>
<tr>
<td>LD R3 0(R4)</td>
<td># cache hit</td>
<td></td>
</tr>
<tr>
<td>ADD R3 R1 R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1 R2 R3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Circle one:
- Out-of-Order Issue with Renaming
- Branch Prediction
- Superscalar