ECE 411, Exam 2

- This exam has 5 problems. Make sure you have a complete exam before you begin.
- Write your name on every page in case pages become separated during grading.
- You will have three hours to complete this exam.
- Write all of your answers on the exam itself. If you need more space to answer a given problem, continue on the back of the page, but clearly indicate that you have done so.
- This exam is closed-book but 1-page of cheat sheet is allowed. You may also use a calculator.
- DO NOT do anything that might be perceived as cheating. The minimum penalty will be a grade of zero.
- Show all of your work on all problems. Correct answers that do not include work demonstrating how they were generated may not receive full credit, and answers that show no work cannot receive partial credit.

- The exam is meant to test your understanding. Ample time has been provided. So, be patient. Read the questions/problems carefully before you answer.

- **Good luck!**

Problem 1 (21 pts): __________
Problem 2 (18 pts): __________
Problem 3 (14 pts): __________
Problem 4 (19 pts): __________
Problem 5 (12 pts): __________

Total (84 pts): __________
1. Data Hazards (21 pts)

Assuming 5-stage pipeline, each instruction takes ONE cycle in every stage. Cache hit rate is 100%. There are one integer adder, one floating point adder and one floating point multiplier. The memory address calculation is done in the EX stage for a memory access instruction. SGTI does the comparison in EX stage.

```
LOOP:     // upon entry into loop,
     // F0 = a (constant), F3 = 0, R1 = 0
I1       L.D F1, 0(R1)     ; load X(i)
I2       MUL.D F2, F1, F0  ; multiply a*X(i)
I3       ADD.D F3, F3, F2  ; add a*X(i) to F3
I4       MUL.D F2, F1, F1  ; multiply X(i)*X(i)
I5       S.D F2, 0(R1)     ; store to memory
I6       ADDI R1, R1, 8d   ; if R1 > 800, R2 = 1
I7       SGTI R2, R1, 800  ; if R1 > 800, R2 = 1
I8       BEQZ R2, LOOP    ; branch
```

a) Identify all RAW, WAW, WAR dependencies in the loop shown above. Write down all the dependencies within a single iteration only. Use the following notation. For example, to indicate a dependency between I1 and I2 through F3 register: I1->I2 (F3). (3 points)

```
Solution: This question asks for dependencies rather than hazards, so dependencies that do not cause a hazard should also be included.

RAW: I1->I2(F1), I2->I3(F2), I4->I4(F2), I6->I7(R1), I7->I8(R2), I2->I5(R2)(NO HAZARD)
WAR: I3->I4(F2), I5->I6(R1), I1->I6(R1)(NO HAZARD)
WAW: I2->I4(F2)
```

Correctly answering dependencies for only the instructions that are adjacent to each other get nearly full credit.

b) How many cycles does single iteration take without forwarding? (3 points)

```
Solution: 8 instr + 5(hazards) * 2(bubble) + 4(optional for filling up pipeline) + 1(optional for bubble after branch since branch resolves in ID)
```

The most important part is 8 + 5 * 2(*3 if assuming non transparent register file)
c) How many cycles does single iteration take only with EX (end) to EX (beginning) forwarding? Note that memory instructions will not be able to use this forwarding mechanism. (3 points)

Solution: Tricky part starts here and carries to the later parts:
Look closely at instruction 8: BEQZ R2, LOOP, since we assume branch always RESOLVE in ID (stated during the exam), this instruction will never forward to its EX stage until it sees the new R2 value so it can use it to resolve in ID. This way, none of the mentioned forwarding techniques will improve this situation. In other words, the 2(3 for non transparent) bubble between I7 and I8 will always be there.

EX to EX will solve I2->I3(F2), I4->I4(F2), I6->I7(R1), so you will have a -6(-9 for non transparent register file) cycles improvement compare to what you have in b.

d) How many cycles does single iteration take only with MEM (end) to EX (beginning) forwarding? Assume all the instructions will be using this scheme, i.e. non-memory instructions will still be forwarded from MEM to EX. (3 points)

Solution: Note: Even for instructions that benefit the most from a EX to EX forwarding, they can still use the MEM to EX forwarding by having one bubble inserted, in this question, I1->I2(F1), I2->I3(F2), I4->I4(F2), I6->I7(R1) will have their bubble number reduced to 1, which results in a -4(-8 for non-transparent) in total cycle number.

e) How many cycles does single iteration take when you implement the combination of both forwarding schemes b) and c)? Instructions will use the fastest forwarding possible. (3 points)

Solution: One bubble left for the load-use case (I1->I2) and 2(3) bubbles left for I7->I8.

f) Reorder the instructions to achieve the least number of cycles per iteration, assuming full forwarding and one branch delay slot. Note: a branch delay slot is an instruction slot that gets executed without the effects of the preceding branch instruction. (6 points)

Solution: Partial credits are given leniently if reordering has shown any improvements to the original code

LOOP: // upon entry into loop,
I1 L.D F1, 0(R1) ; load X(i)
I6 ADDI R1, R1, 8d
I7 SGTI R2, R1, 800 ; if R1 > 800, R2 = 1
I2 MUL.D F2, F1, F0 ; multiply a*X(i)
I3 ADD.D F3, F3, F2 ; add a*X(i) to F3
I4 MUL.D F2, F1, F1 ; multiply X(i)*X(i)
I8 BEQZ R2, LOOP
I5 S.D F2, 0(R1) ; store to memory (BRANCH DELAY SLOT)

Some students put I1 into branch delay slot without explaining that they need to add it to the prolog as well for the first iteration.
2. Control Hazards (18 pts)

You have written an assembly code to compute 5! for MP1. Below code is one way to implement the factorial program and it was modified to compute 7!. Note that there are two main branch instructions (Branch 1 and Branch 2) other than the infinite loop. Please answer the following questions. Assume R0 is set to 0 initially.

SEGMENT CodeSegment:
  LDR R1, R0, SEVEN
  LDR R2, R0, NEGONE
  LDR R3, R0, NEGTWO
  ADD R4, R1, R3 ; inner counter
  ADD R5, R0, R4 ; outer counter
  ADD R6, R0, R1 ; initial set

LOOP:
  ADD R6, R6, R1
  ADD R4, R4, R2
  BRp Loop ; Branch 1
  ADD R1, R0, R6
  ADD R5, R5, R2
  ADD R4, R0, R5
  BRp LOOP ; Branch 2

HALT:
  BRnzp HALT ; Infinite loop

SEVEN:   DATA2  4x0007
NEGONE:  DATA2  4xFFFF
NEGTWO:  DATA2  4xFFFE

a) What are the actual behaviors (Taken: T, Not Taken: N) of each branches? (2 points)

- Branch 1: TTTTTTTNTNTNN
- Branch 2: TTTTN
b) What are the misprediction rates (# of mispredictions /total number of branch execution) for each branch prediction techniques for Branch 1? For 2-bit predictor, there are total 4 states - Strongly Taken: ST. Weakly Taken: WT, Weakly Not Taken: WN, Strongly Not Taken: SN. (6 points)

- Static Predict Taken – 5/15
- Static Predict Not Taken – 10/15
- 1-bit Predictor (Initial: T) – 7/15
- 1-bit Predictor (Initial: N) – 8/15
- 2-bit Predictor (Initial: WT) – 5/15
- 2-bit Predictor (Initial: SN) – 7/15

c) How about Branch 2? (6 points)

- Static Predict Taken – 1/5
- Static Predict Not Taken – 4/5
- 1-bit Predictor (Initial: T) – 1/5
- 1-bit Predictor (Initial: N) – 2/5
- 2-bit Predictor (Initial: WT) – 1/5
- 2-bit Predictor (Initial: SN) – 3/5

d) From the result you have chosen from above, which branch prediction techniques would you choose for both Branch 1 and Branch 2? Note that you can only choose ONE method. (2 points)

Solution: Static predict taken.

e) How would the performance be different if you implement global predictor? Will it improve or worsen the performance or no effect? Why or why not? (2 points)

Solution: In this case, it doesn't improve better but we can see the close correlation with Branch 1 and Branch 2. For the iterations except the last one, Branch 2 will always be taken when Branch 1 was not Taken. We can use this correlation to form the global predictor, however, the misprediction rate will not improve from the best predictor we chose above (1-bit with initial taken state).
3. Software ILP Techniques (14 pts)

Consider the following code:

```c
int i, int j, double A[256][256];
for (i = 0; i < 64; i++) {
    for (j = 0; j < 64; j++) {
        x += A[2*i][2*j];
    }
}
```

Assume the following:
- Array A entries contain double words.
- Only accesses to array locations generate loads to the data cache. The rest of the variables are allocated in registers.
- The array A is stored in row major form.
- The program is running on a machine with an L1 data cache that has 32 words per cache line.
- Memory is word addressable.
- Assume the array A starts at a cache line boundary.
- Assume an infinite data cache that is initially empty.

a) How many L1 data cache misses occur for the above code? You must explain how you derived this number to receive any credit. (3 points)

Solution: Each cache line holds 32/2=16 elements of the matrix. Each inner loop will read data from 0 to 2*63=126. Each inner loop invocation therefore results in 126/16=8 cache misses. There are a total of 64 invocations of the inner loop. Therefore there are a total of 8*64=512 L1 misses.

b) Suppose our machine has a data prefetch instruction with the format prefetch (array[index]). This prefetches the entire cache line containing the word array[index] into the data cache. Assume it takes one cycle for the processor to execute this instruction. The processor can then execute subsequent instructions. The data will be loaded into the cache after 16*execution_time_of (x += A[2*i][2*j]).

Consider inserting prefetch instructions for the inner loop of the above code (i.e., ignore the outer loop for this part). Explain why we may need to unroll the inner loop to insert prefetches. What is the minimum number of times you would need to unroll the loop for this purpose? The number of unrolled loops includes the original loop. (4 points)

Solution: If we attempt to insert prefetch instructions without unrolling the inner loop, we would be prefetching superfluously since each prefetch brings in an entire cache line. To take advantage of prefetching, we would like every iteration of the loop to access as much data as possible in the prefetched cache line. We are told that each cache line contains 32 words (i.e., 16 double words).
However, the program uses a strided access pattern where only the even array entries are used; which translates to only half of a cache line being used. Thus, we should unroll the loop $16/2 = 8$ times.

c) Unroll the inner loop for the number of times identified in the previous part and insert the minimum number of software prefetches to minimize execution time. You do not need to worry about startup and cleanup code (epilog and prolog) and do not introduce any new loops. Again, for this part, you may ignore the outer loop. (4 points)

Solution:

```c
for (i = 0; i < 64; i++) {
    for (j = 0; j < 64; j+=8) {
        prefetch(A[2*i][2*(j+16)]);
        x += A[2*i][2*j];
        x += A[2*i][2*(j+1)];
        x += A[2*i][2*(j+2)];
        x += A[2*i][2*(j+3)];
        x += A[2*i][2*(j+4)];
        x += A[2*i][2*(j+5)];
        x += A[2*i][2*(j+6)];
        x += A[2*i][2*(j+7)];
    }
}
```

d) Consider your solution to part c). How many of the data cache misses from the original code now have their latency fully hidden? For this part, consider both the inner and outer loops. You must explain how you derived this number to receive any credit. If an incorrect solution to part c) trivializes this problem or makes it too difficult, no credit will be given. (3 points)

Solution: The first two iterations of the new inner loop will still result in cache misses because that data is not being prefetched. There are a total of 64 invocations of the inner loop. Therefore there are a total of 128 L1 misses using prefetches. Since there were originally 512 L1 misses, 384 of these L1 misses have their latency fully hidden due to the prefetches.
4. Hardware ILP Techniques (19 pts)

You are building out-of-order 6-stage pipeline (Instruction Fetch, Instruction Decode, Execute, Memory, Writeback, Retire). Instruction Fetch and Decode will be done in order, but from Execute to Retire stage, datapath handles whichever instruction that is ready, i.e. out-of-order. If multiple instructions are ready, execute in program order. Consider the given code and specifications below and answer the following questions.

// I is for Instruction Fetch, D is for Data
I1: LDR R1, R0, DATA ; I: Cache miss, D: Cache miss
I2: ADD R1, R1, R1 ; I: Cache hit
I3: LDR R2, R0, R1 ; I: Cache hit, D: Cache miss
I4: LDR R3, R0, DATA2 ; I: Cache hit, D: Cache hit
I5: AND R4, R2, R3 ; I: Cache hit
I6: MUL R5, R1, R2 ; I: Cache hit
I7: NOT R7, R7, R0 ; I: Cache hit
I8: ADD R2, R0, R2 ; I: Cache hit
I9: LDR R3, R0, R7 ; I: Cache miss, D: Cache miss
I10: ADD R5, R3, R5 ; I: Cache hit
I11: STR R5, R0, R2 ; I: Cache hit, D: Cache miss
...

DATA: XXXX
DATA2: XXXX

<Specifications>
- There are 3 integer units and 2 load and 1 store units
- Instruction queue can hold up to 6 instructions
- ID, WB, Retire stages take 1 cycle each
- In IF and MEM stage, it takes 1 cycle on a cache hit and 10 cycles on a cache miss
- MUL takes 10 cycles, ADD/AND/NOT takes 2 cycles, LDR/STR takes 1 cycle in EX stage
- Assume data forwarding from MEM to EX stage
- Assume register renaming allowed

a) With given constraints, what is the minimum size of the reservation station to prevent it from being a bottleneck? In other words, how many entries (1 entry can hold 1 instruction) does the reservation station require to have in order to ensure the datapath doesn’t stall due to reservation capacity? Assume infinite size of reorder buffer. Use the tables in the next page to compute the answer. (6 points)

b) Assume unlimited reservation stations. What is the minimum number of entries for reorder buffer to avoid being a bottleneck? Is it related with the reservation station size? Why or why not? (3 points)
Solution: a), b) There can be two assumptions possible depending on where the LD/ST unit is being used. 1) multiple LD/ST units are used in the EX stage in order to calculate the memory addresses simultaneously, but holds to send to MEM stage in order one at a time, 2) multiple LD/ST units not only handles multiple instructions in the EX stage but also in MEM stage so that non-dependent memory instructions can access memory simultaneously.

1) First assumption (no multiple instructions in MEM stage)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Retire</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R1 R0 DATA</td>
<td>1-10</td>
<td>11</td>
<td>12</td>
<td>13-22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>ADD R1 R1 R1</td>
<td>11</td>
<td>12</td>
<td>23-24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>LDR R2 R0 R1</td>
<td>12</td>
<td>13</td>
<td>26</td>
<td>27-36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>LDR R3 R0 DATA2</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>AND R4 R2 R3</td>
<td>14</td>
<td>15</td>
<td>37-38</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>MUL R5 R1 R2</td>
<td>15</td>
<td>16</td>
<td>37-46</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>NOT R7 R7</td>
<td>16</td>
<td>17</td>
<td>18-19</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>ADD R2 R0 R2</td>
<td>17</td>
<td>18</td>
<td>37-38</td>
<td>49</td>
<td>50</td>
<td>51</td>
</tr>
<tr>
<td>LDR R3 R0 R7</td>
<td>18-27</td>
<td>28</td>
<td>29</td>
<td>37-46</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>ADD R5 R3 R5</td>
<td>28</td>
<td>29</td>
<td>49-50</td>
<td>51</td>
<td>52</td>
<td>53</td>
</tr>
<tr>
<td>STR R5 R0 R2</td>
<td>29</td>
<td>30</td>
<td>50</td>
<td>52-61</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Reservation Station</th>
<th>Entries</th>
<th>Reorder Buffer</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R1 R0 DATA</td>
<td>11</td>
<td>1</td>
<td>1-24</td>
<td>1</td>
</tr>
<tr>
<td>ADD R1 R1 R1</td>
<td>12-22</td>
<td>1</td>
<td>11-27</td>
<td>2</td>
</tr>
<tr>
<td>LDR R2 R0 R1</td>
<td>13-25</td>
<td>2</td>
<td>12-38</td>
<td>3</td>
</tr>
<tr>
<td>LDR R3 R0 DATA2</td>
<td>14</td>
<td>3</td>
<td>13-25</td>
<td>4</td>
</tr>
<tr>
<td>AND R4 R2 R3</td>
<td>15-36</td>
<td>3</td>
<td>14-49</td>
<td>5</td>
</tr>
<tr>
<td>MUL R5 R1 R2</td>
<td>16-36</td>
<td>4</td>
<td>15-50</td>
<td>6</td>
</tr>
<tr>
<td>NOT R7 R7</td>
<td>17</td>
<td>5</td>
<td>16-26</td>
<td>7</td>
</tr>
<tr>
<td>ADD R2 R0 R2</td>
<td>18-36</td>
<td>5</td>
<td>17-51</td>
<td>8</td>
</tr>
<tr>
<td>LDR R3 R0 R7</td>
<td>28</td>
<td>4</td>
<td>18-48</td>
<td>9</td>
</tr>
<tr>
<td>ADD R5 R3 R5</td>
<td>29-48</td>
<td>4</td>
<td>28-62</td>
<td>6</td>
</tr>
<tr>
<td>STR R5 R0 R2</td>
<td>30-49</td>
<td>5</td>
<td>29-61</td>
<td>7</td>
</tr>
</tbody>
</table>
2) Second assumption (multiple memory instructions in MEM stage allowed)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Retire</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R1 R0 DATA</td>
<td>1-10</td>
<td>11</td>
<td>12</td>
<td>13-22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>ADD R1 R1 R1</td>
<td>11</td>
<td>12</td>
<td>23-24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>LDR R2 R0 R1</td>
<td>12</td>
<td>13</td>
<td>26</td>
<td>27-36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>LDR R3 R0 DATA2</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>AND R4 R2 R3</td>
<td>14</td>
<td>15</td>
<td>37-38</td>
<td>47</td>
<td>48</td>
<td>49</td>
</tr>
<tr>
<td>MUL R5 R1 R2</td>
<td>15</td>
<td>16</td>
<td>37-46</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>NOT R7 R7</td>
<td>16</td>
<td>17</td>
<td>18-19</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>ADD R2 R0 R2</td>
<td>17</td>
<td>18</td>
<td>37-38</td>
<td>49</td>
<td>50</td>
<td>51</td>
</tr>
<tr>
<td>LDR R3 R0 R7</td>
<td>18-27</td>
<td>28</td>
<td>29</td>
<td>30-39</td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>ADD R5 R3 R5</td>
<td>28</td>
<td>29</td>
<td>49-50</td>
<td>51</td>
<td>52</td>
<td>53</td>
</tr>
<tr>
<td>STR R5 R0 R2</td>
<td>29</td>
<td>30</td>
<td>50</td>
<td>52-61</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reservation Station</th>
<th>Entries</th>
<th>Reorder Buffer</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R1 R0 DATA</td>
<td>11</td>
<td>1-24</td>
<td>1</td>
</tr>
<tr>
<td>ADD R1 R1 R1</td>
<td>12-22</td>
<td>11-27</td>
<td>2</td>
</tr>
<tr>
<td>LDR R2 R0 R1</td>
<td>13-25</td>
<td>12-38</td>
<td>3</td>
</tr>
<tr>
<td>LDR R3 R0 DATA2</td>
<td>14</td>
<td>13-18</td>
<td>4</td>
</tr>
<tr>
<td>AND R4 R2 R3</td>
<td>15-36</td>
<td>14-49</td>
<td>5</td>
</tr>
<tr>
<td>MUL R5 R1 R2</td>
<td>16-36</td>
<td>15-50</td>
<td>6</td>
</tr>
<tr>
<td>NOT R7 R7</td>
<td>17</td>
<td>16-26</td>
<td>7</td>
</tr>
<tr>
<td>ADD R2 R0 R2</td>
<td>18-36</td>
<td>17-51</td>
<td>8</td>
</tr>
<tr>
<td>LDR R3 R0 R7</td>
<td>28</td>
<td>18-41</td>
<td>8</td>
</tr>
<tr>
<td>ADD R5 R3 R5</td>
<td>29-48</td>
<td>28-62</td>
<td>6</td>
</tr>
<tr>
<td>STR R5 R0 R2</td>
<td>30-49</td>
<td>29-61</td>
<td>7</td>
</tr>
</tbody>
</table>

c) Now, let’s assume unlimited execution units (both integer units and load/store units). What is
the minimum size of the reservation station and reorder buffer to run the program without any
stalls other than hardware delay (e.g. memory latency, execution time of each operations)? (3
points)

- Integer, LD/ST units: Unlimited
- Reservation station: 5
- Reorder buffer: 8 or 9

Solution: Since the execution units were not the bottleneck of the above instructions, the
number of reservation stations and the reorder buffer would remain the same.
d) Let’s say you have 4-entries in reservation station and 8-entries in reorder buffer. How many cycles does the above program take to run? (5 points)
- Integer units: 3, LD unit: 2, ST unit: 1
- Reservation station: 4
- Reorder buffer: 8

For this problem, the solution for only first assumption from a) is provided.

<table>
<thead>
<tr>
<th>&lt;Cycle&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
</tr>
<tr>
<td>LDR R1, R0, DATA</td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
</tr>
<tr>
<td>LDR R2, R0, R1</td>
</tr>
<tr>
<td>LDR R3, R0, DATA2</td>
</tr>
<tr>
<td>AND R4, R2, R3</td>
</tr>
<tr>
<td>MUL R5, R1, R2</td>
</tr>
<tr>
<td>NOT R7, R7, R0</td>
</tr>
<tr>
<td>ADD R2, R0, R2</td>
</tr>
<tr>
<td>LDR R3, R0, R7</td>
</tr>
<tr>
<td>ADD R5, R3, R5</td>
</tr>
<tr>
<td>STR R5, R0, R2</td>
</tr>
</tbody>
</table>

e) Explain why the number of bypass paths varies quadratically with issue width. (2 points)
5. VLIW (12 pts)

a) List one advantage and one disadvantage of a VLIW design over your basic pipeline for MP3. (2 points)

Solution:
Advantage: More instructions can occupy the same stage of the pipeline at the same time, leading to increased throughput.

Disadvantage: Need a more complex compiler to schedule instructions while taking account dependencies. Compatibility between different versions of the processor is hard to maintain and will require recompilation of programs. The need for NOPs can make programs take more memory.

b) If you were converting the MP3 basic pipeline to a VLIW design, what modifications must be made to the Instruction Decode (ID) stage to avoid structural hazards? (2 points)

Solution: All the instructions issued together must be able to access their operands. The number of register file ports, the number of sign/zero extension units, and the number of control signals would need to be increased proportional to the issue width.
c) Schedule single iteration of the loop below on a three-issue VLIW pipeline. The pipeline contains hardware to support the following combination of operations in parallel:

- 1 integer ALU operation or branch (addi or bxx)
- 1 floating point operation (mulf or addf)
- 1 data transfer operation (load or store)

Reorder the instructions to avoid as many pipeline stalls as possible. Assume all loads have a use latency of one cycle (e.g., if a load is executed in cycle 1, then a dependent instruction must wait until cycle 3 to begin its execution). Assume branches are predicted correctly. (4 points)

```
loop:
    load f0, 0(r1)
    load f2, 0(r2)
    mulf f4, f0, f6
    addf f8, f4, f2
    store f8, 0(r2)
    addi r1, r1, 4
    addi r2, r2, 4
    addi r3, r3, -1
    bne r3, loop ; branch if r3 != 0
```

Solution: There are several ways to reorder the instructions, so that execution finishes in 5 cycles. Below is one way to do it.

<table>
<thead>
<tr>
<th>Integer ALU/Branch</th>
<th>Floating Point ALU</th>
<th>Data Transfer</th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi r3, r3, -1</td>
<td>load f0, 0(r1)</td>
<td>addi r1, r1, 4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mulf f4, f0, f6</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>addf f8, f4, f2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>store f8, -4(r2)</td>
<td>4</td>
</tr>
<tr>
<td>bne r3, loop</td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Note: some students assumed that the use latency for load meant that no loads could be scheduled immediately following another load. In this case, the store, bne, addf, and second load would simply occur one cycle later.

d) What is the CPI of the loop using the schedule above? What is the speedup over the original code executed on a non-VLIW processor? (4 points)

Solution: The CPI is the number of cycles divided by the number of instructions, or 5/9. The CPI of the original code is 1 using the assumptions provided. Thus, the speedup is 9/5.