CS 433 Midterm Exam – Oct 12, 2016
Professor Sarita Adve

Time: 2 Hours

Please print your name and NetID and circle the appropriate category in the space provided below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetID</td>
<td></td>
</tr>
<tr>
<td>Category</td>
<td>Undergraduate Graduate</td>
</tr>
</tbody>
</table>

Instructions

1. No books, papers, notes, or any other typed or written materials are allowed. No calculators or other electronic materials are allowed.

2. Please do not turn in loose scrap paper. Limit your answers to the space provided if possible. If this is not possible, please write on the back of the same sheet. You may use the back of each sheet for scratch work.

3. In all cases, show your work. No credit will be given if there is no indication of how the answer was derived. Partial credit will be given even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.

4. If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution. In all cases, clearly state any assumptions that you make in your answers.

5. This exam has 6 problems and 14 pages (including this one). All students should solve problems 1 through 5. Only graduate students should solve problem 6. Please budget your time appropriately. Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Maximum Points</th>
<th>Received Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7 (for graduates only)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>53 for undergraduates 60 for graduates</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1 [6 points]
Assume that we make an enhancement to a computer that reduces the execution time of part of a program’s execution by a factor of 10. Assume the enhancement is used 50% of the total execution time, measured as a percentage of the execution time when the enhancement is in use.

Part (A) [3 points]
What is the overall speedup we have obtained from the enhancement?

Solution:
Let old = the execution time without the enhancement.
Let new = the execution time with the enhancement.
old = 0.5 new + 0.5 × 10 new = 5.5 new
speedup = old/new = 5.5

Grading: 1.5 points for each of the two equations. No deduction for calculation mistakes.

Part (B) [3 points]
What percentage of the original execution time has been converted to enhanced mode?

Solution:
Let the original execution take 100 units of time. Let x be the percentage of the original execution time that was enhanced. The time taken by the enhanced version can be split as follows:
For the unenhanced part: 100 – x
For the enhanced part: x/10
We know that these two parts take the same amount of time in the enhanced version. So:
(100 – x) = x /10
1000 – 10x = x
1000 = 11x
1000/11 = x = 91%

Grading: 3 points for equation (1). No deduction for calculation mistakes.
Problem 2 [11 points]

This problem concerns Tomasulo’s algorithm with dual issue and hardware speculation.

Consider the following architecture.

<table>
<thead>
<tr>
<th>Functional Unit Type</th>
<th>Cycles in EX</th>
<th>Number of Functional Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP Add/Subtract</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FP Divide</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

- Functional units are not pipelined.
- Memory accesses use the integer functional unit to perform effective address calculation during a single cycle EX stage. Stores access memory during the commit stage while loads access memory during EX. Stores do not need the CDB or the WB stage.
- Assume that integer instructions also follow Tomasulo’s algorithm (analogous to the floating point instructions). So the result from the integer functional unit is also broadcast on the CDB and forwarded to dependent instructions through the CDB. An exception is that addresses generated for memory accesses are not broadcast on the CDB.
- If an instruction moves to its WB stage in cycle x, then an instruction that is waiting on the same functional unit (due to a structural hazard) can start executing in cycle x.
- An instruction waiting for data on the CDB can move to EX in the cycle after the CDB broadcast.
- Only one instruction can write to the CDB in one clock cycle.
- If there is a conflict for a functional unit or the CDB, the oldest (by program order) instruction gets access.
- Branches use the integer unit with one cycle in EX. They do not need the CDB or the WB stage.
- Assume a perfect branch predictor. That is, an instruction can issue before a prior branch has completed execution, but it cannot issue in the same cycle as the branch issues (the earliest it can issue is the cycle immediately after the branch issues). Any other instruction pair can issue in the same cycle.
- There are unlimited reorder buffer entries and reservation stations.
- Two instructions can commit per cycle.

Complete the blank entries in the following table using the above specifications. For each instruction, fill in the cycle numbers in each pipeline stage (CM stands for commit). Then indicate where its source operands are read from (use RF for register file, ROB for reorder buffer, and CDB for common data bus) – the source of the first operand should appear first. In
the last two columns, indicate if the instruction is stalled for structural or data hazards. If it is stalled, indicate which instructions cause those stalls – give all such instructions even if the dependence due to one of them is resolved before the other. Some entries are filled for you. You don’t have to fill entries with “---”

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IS</th>
<th>EX</th>
<th>WB</th>
<th>CM</th>
<th>Source of operand1, source of operand2</th>
<th>Stalls due to structural hazards</th>
<th>data hazards</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 L.D F0, 0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2 DIV.D F2, F0, F6</td>
<td>1</td>
<td>4</td>
<td>20</td>
<td>21</td>
<td>CDB, RF</td>
<td>No</td>
<td>Yes: 1</td>
</tr>
<tr>
<td>3 L. D F6, 8(R1)</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>21</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>4 ADD.D F4, F2, F6</td>
<td>2</td>
<td>21</td>
<td>25</td>
<td>26</td>
<td>---</td>
<td>No</td>
<td>Yes: 2,3</td>
</tr>
<tr>
<td>5 MUL.D F8, F6,F4</td>
<td>3</td>
<td>26</td>
<td>31</td>
<td>32</td>
<td>CDB, CDB</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6 S.D F4, 16(R1)</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>32</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>7 DADDUI R1, R1, -32</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>33</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>8 BNEZ R1, target</td>
<td>4</td>
<td>7</td>
<td>-</td>
<td>33</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>9 ADD.D F10, F2, F6</td>
<td>5</td>
<td>25</td>
<td>29</td>
<td>34</td>
<td>CDB, ROB</td>
<td>Yes: 4</td>
<td>Yes: 2</td>
</tr>
</tbody>
</table>

**Grading:** ¼ point for each IS, EX, WB and CM entries; ¼ point bonus if all entries are correct (total of 8 points). ½ point for each correct source operand (total 2 points). ¼ point for each stall entry; points given only if all instructions causing a stall are mentioned (total 1 point).

**Note:** Cascading errors were not penalized.

Total 11 points
Problem 3 [10 points]

Consider code with three static branch instructions, B1, B2, and B3. During an execution of this code, the global history for these branches (i.e., the exact sequence in which the three branches were executed and their direction) is as follows, where T stands for the taken direction and N stands for not-taken:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
</tbody>
</table>

Thus, the execution starts with branch B1 being taken, then B2 is not taken, then B1 is not taken, etc. Assume that before the above execution, the outcome of all previous branches and the initial state of all predictors is not taken (N).

Assume that a correlating predictor of the form (2,1) is used for branch B1. Assume the state of the predictor is recorded in the form W/X/Y/Z where,

- W = state for the case where the last branch and the branch before the last are both TAKEN
- X = state for the case where the last branch is TAKEN and the branch before the last is NOT TAKEN
- Y = state for the case where the last branch is NOT TAKEN and the branch before the last is TAKEN
- Z = state for the case where the last branch and the branch before the last are both NOT TAKEN

Fill the blank entries in the two tables below assuming the (2,1) correlating predictor for branch B1 uses local branch history in Table 1 and global branch history in Table 2. (The global history above is repeated on the next page for convenience.)

<table>
<thead>
<tr>
<th>Table 1: Assume the predictor uses LOCAL branch history</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch B1 invocation #</strong></td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
Table 2: Assume the predictor uses GLOBAL branch history

<table>
<thead>
<tr>
<th>Branch B1 invocation #</th>
<th>History used for prediction</th>
<th>Prediction for B1</th>
<th>Actual direction of B1</th>
<th>New state of the predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N N</td>
<td>N</td>
<td>T</td>
<td>N/N/N/T</td>
</tr>
<tr>
<td>2</td>
<td>T N</td>
<td>N</td>
<td>N</td>
<td>N/N/N/T</td>
</tr>
<tr>
<td>3</td>
<td>T T</td>
<td>N</td>
<td>T</td>
<td>T/N/N/T</td>
</tr>
<tr>
<td>4</td>
<td>T T</td>
<td>T</td>
<td>T</td>
<td>T/N/N/T</td>
</tr>
<tr>
<td>5</td>
<td>T N</td>
<td>N</td>
<td>T</td>
<td>T/N/T/T</td>
</tr>
</tbody>
</table>

The global history from the previous page is repeated below for convenience:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>T</td>
<td>N</td>
<td>N</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
</tbody>
</table>

Grading: 5 points for each table. 0.5 points for each blank entry of first row. For other rows, 0.5 for history, 0.25 for prediction, and 0.25 for new state.
Problem 4 [18 Points]

In this problem, we will use a single-issue, in-order MIPS pipeline similar to those studied in class, but with the following specification.

- The integer functional unit performs integer addition (including effective address calculation for loads/stores), subtraction, and logic operations.
- There is full forwarding and bypassing, including forwarding from the end of an FU to the MEM stage for stores.
- Loads and stores complete in one cycle. That is, they spend one cycle in the MEM stage after the effective address calculation.
- There are as many registers, both FP and integer, as you need.
- Branches are resolved in ID and there is one branch delay slot.
- If multiple instructions finish their EX stages in the same cycle, then we will assume they can all proceed to the MEM stage together. Similarly, if multiple instructions finish their MEM stages in the same cycle, then we will assume they can all proceed to the WB stage together. In other words, for the purpose of this problem, you are to ignore structural hazards on the MEM and WB stages.
- This problem explores the ability of the compiler to schedule code as efficiently as possible for such a pipeline. We will consider the following code.

<table>
<thead>
<tr>
<th>Functional Unit Type</th>
<th>Cycles in EX</th>
<th>Number of Functional Units</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>FP Add/Subtract</td>
<td>3</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>FP/Integer Multiplier</td>
<td>8</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>FP/Integer Divider</td>
<td>24</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

Loop: 

L.D F4, 0 (R1)  
MUL.D F8, F4, F0  
L.D F6, 0 (R2)  
ADD.D F10, F6, F2  
ADD.D F12, F8, F10  
S.D F12, 0 (R3)  
DADDUI R1, R1, #8  
DADDUI R2, R2, #8  
DADDUI R3, R3, #8  
DSUB R5, R4, R1  
BNEZ R5, Loop
Part A [6 points]

Rewrite the above loop, but let every row take a cycle. If an instruction can’t be issued on a given cycle (because the current instruction has a dependency that will not be resolved in time), write STALL instead, and move on to the next cycle (row) to see if it can be issued then. *Explain the cause of all stalls*, but don’t reorder instructions. How many cycles elapse before the second iteration begins? Show your work (i.e., write instructions in order as they would be executed and write STALL in between instructions for each cycle where there would be a stall). Remember there is 1 branch delay slot.

**Solution:**

Loop: 

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F4, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>stall RAW F4</td>
<td></td>
</tr>
<tr>
<td>MUL.D F8, F4, F0</td>
<td></td>
</tr>
<tr>
<td>L.D F6, 0(R2)</td>
<td></td>
</tr>
<tr>
<td>stall RAW F6</td>
<td></td>
</tr>
<tr>
<td>ADD.D F10, F6, F2</td>
<td></td>
</tr>
<tr>
<td>stall RAW F8, F10</td>
<td></td>
</tr>
<tr>
<td>stall RAW F8, F10</td>
<td></td>
</tr>
<tr>
<td>stall RAW F8</td>
<td></td>
</tr>
<tr>
<td>stall RAW F8</td>
<td></td>
</tr>
<tr>
<td>ADD.D F12, F8, F10</td>
<td></td>
</tr>
<tr>
<td>stall RAW F12</td>
<td></td>
</tr>
<tr>
<td>S.D F12, 0(R3)</td>
<td></td>
</tr>
<tr>
<td>DADDUI R1, R1, #8</td>
<td></td>
</tr>
<tr>
<td>DADDUI R2, R2, #8</td>
<td></td>
</tr>
<tr>
<td>DADDUI R3, R3, #8</td>
<td></td>
</tr>
<tr>
<td>DSUB R5, R4, R1</td>
<td></td>
</tr>
<tr>
<td>stall RAW R5 since branch resolved in ID</td>
<td></td>
</tr>
<tr>
<td>BNEZ R5, Loop</td>
<td></td>
</tr>
<tr>
<td>NOP – stall for branch delay</td>
<td></td>
</tr>
</tbody>
</table>

20 cycles elapse before the next iteration begins.

**Grading:** 1 point for each sequence of stalls. ½ point partial credit for indicating that there is a stall between a pair of instructions, but with an incorrect number of cycles. Negative ½ point for each unnecessary sequence of stalls.
Part B [6 points]

Now reschedule the loop to compute the same results as quickly as possible. You can change immediate values and memory offsets and reorder instructions, but don’t change anything else. Show any stalls that remain. How many cycles elapse before the second iteration begins? Show your work.

**Solution:**

Loop:
- L.D F4, 0(R1)
- L.D F6, 0(R2)
- MUL.D F8, F4, F0
- ADD.D F10, F6, F2
- DADDUI R1, R1, #8
- DADDUI R2, R2, #8
- DADDUI R3, R3, #8
- DSUB R5, R4, R1
- stall RAW F8
- stall RAW F8
- ADD.D F12, F8, F10
- BNEZ R5, Loop
- S.D F12, -8(R3)

13 cycles elapse before the second iteration begins

**Grading:** Full points for any correct sequence with minimum number of stalls. Partial credit only if the sequence does the same computation and reduces some stalls. Deduct \( \frac{1}{2} \) point for each error (e.g., incorrect index), and deduct \( \frac{1}{2} \) point for each stall in excess of 2.
Part C [6 points]

Now unroll the loop the minimum number of times needed to eliminate all stalls (with rescheduling). Show the unrolled and rescheduled loop. You can, and should, remove redundant instructions. How many original iterations of the loop are in an iteration of your new unrolled loop? How many cycles elapse before the next iteration of the unrolled loop begins? Don’t worry about start-up or clean-up code outside the unrolled loop. Show your work.

Solution: Note that in the solutions below, the registers used could be different and there is some flexibility in scheduling the instructions.

Loop:

L.D F4, 0(R1)
L.D F6, 0(R2)
MUL.D F8, F4, F0
L.D F14, 8(R1)
L.D F16, 8(R2)
MUL.D F18, F14, F0
ADD.D F10, F6, F2
ADD.D F20, F16, F2
DADDUI R1, R1, #16
DADDUI R2, R2, #16
DADDUI R3, R3, #16
ADD.D F12, F8, F10
DSUB R5, R4, R1
ADD.D F22, F18, F20
S.D F12, -16(R3)
BNEZ R5, Loop
S.D F22, -8(R3)

There are two original iterations in an iteration of the new loop. 17 cycles elapse before the next iteration of the new loop begins.

Grading: 1 point for the correct iteration count. Deduct ½ point for every error or stall cycle. Give partial credit (3 points) if three iterations are used instead of two and the solution is correct with three iterations.
Problem 5 [8 Points]

Consider the following format for predicated MIPS instructions:

\[(pT) \text{ADD R1, R2, R3}\]

where the ADD instruction is predicated on the predicate register pT. Assume a set of 1-bit predicate registers.

Assume compare instructions that set a pair of predicate registers to complementary values:

\[\text{CMP.NE pT, pF = R8, R0}\]

The above compare sets the 1-bit predicate registers, pT and pF, based on the "not equal" (NE) comparison relation as follows:

\[pT = (R8 \neq R0)\]
\[pF = !(R8\neq R0)\]

So pT is true if R8 is not equal to R0, and pF is the complement of pT.

For the following problem, you can assume the availability of any comparison relation with two operands; e.g., .LE for less than or equal to and .GT for greater than.

Part A [4 points]

Using the predicated instructions described above, write the three basic blocks of the following code fragment as a single basic block; i.e., eliminate all branches using predicated instructions.

```
SUB R1, R13, R14
BLT R1, R4, L1  //branch if R1 < R4
ADDI R2, R2, #1
SW R2, 0(R7)
J L2
L1:  DIV.D F0, F0, F2
    ADD.D F0, F4, F2
    S.D F0, 0(R8)
L2:   ...
```

Solution:

The code fragment with predicated instructions is as below

1. SUB R1, R13, R14
2. CMP.LT pT, pF = R1, R4
3. \((pF) \text{ADDI R2, R2, #1}\)
4. \((pF) \text{SW 0(R7), R2}\)
5. \((pT) \text{DIV.D F0, F0, F2}\)
6. \((pT) \text{ADD.D F0, F4, F2}\)
7. \((pT) \text{S.D 0(R8), F0}\)

Grading: 1/2 point for correctly translating each of the 8 instructions in the original code. (Note that for the jump instruction, J, the correct translation is to not have any instruction.)
Part B [4 points]

What are all the RAW data dependences in your new code?

**Solution:**
The RAW data dependences are:
(i) RAW dependency due to R1 between SUB and CMP.
(ii) RAW dependency due to pF between CMP and ADDI.
(iii) RAW dependency due to pF between CMP and SW.
(iv) RAW dependency due to R2 between ADDI and SW.
(v) RAW dependency due to pT between CMP and DIV.D.
(vi) RAW dependency due to pT between CMP and ADD.D.
(vii) RAW dependency due to pT between CMP and S.D.
(viii) RAW dependency due to F0 between ADD.D and S.D.

**Grading:** 1/2 point for each correct dependency.
ONLY GRADUATE STUDENTS SHOULD SOLVE THE NEXT PROBLEM.

Problem 6 [7 points]

Consider the following loop:

```
LOOP:  ADD  RA, R*, R*
       SRA  RB, RA
       ADD  RC, RB, R*
       SLA  RD, RC
       SUB  RE, RD, R*
       BNE  RF, RE LOOP
```

You need not concern yourself with the actual action of the loop or whether it is doing anything useful. Focus only on the dependences in the loop. Here all registers marked as $R^*$ indicate that there are no dependences for them and you need not care about them. Note that there are no RAW dependences across loop iterations. Assume the loop executes 10 times.

Part A [2 points]

Consider a machine that uses Tomasulo’s algorithm for dynamic scheduling with renaming. Suppose the machine does not support speculation and so stalls on all branches until they are resolved. Suppose the machine can fetch, decode, and issue (to the reservations stations) an unbounded number of instructions per cycle and has infinite functional units. How large an instruction window must the machine support to best exploit ILP in this code? Assume that the instruction window size can only be a multiple of six. Recall that the instruction window refers to the total number of instructions in flight at a time. Justify your answer and clearly state any assumptions you make.

Solution:
Due to the long dependence, instructions of a given iteration are serialized and none of them can execute out of order. Using an instruction window larger than the loop iteration will not be of benefit since the machine does not use speculation and the direction of the branch is dependent on the rest of the loop computation. Therefore, an instruction window size of 6 would be appropriate.

Grading: 2 points for the correct size and justification.
**Part B [2 points]**

Suppose now that the machine in part (A) supports speculation and has a magical branch predictor with 100% accuracy. How would your answer to part (A) change?

**Solution:** Each iteration of the loop is independent of the others, so all iterations can be executed in parallel. So to achieve maximum performance, an issue window of 60 should be used. **Grading:** 2 points for the correct size and justification.

**Part C [3 points]:**

You are now told of some new research that does not require stalling on RAW dependences (e.g., value prediction allows speculation on values, and triggers a rollback if the speculation was wrong). Now how would your answer to part (B) change? Do you expect to see a change in IPC from part (B) to part (C)? Again, be sure to justify your answers.

**Solution:**
The answer to part (B) would remain the same. Instruction window size would still be 60 for the same reason as above. Since dependences in each iteration are broken via value prediction, IPC would be expected to increase provided that the value predictor has a reasonable prediction rate.

**Grading:** 1.5 points for the statement about instruction window size, 1.5 points for the statement about IPC. The comment about value prediction rate is not required. 3 points total.