CS 433 Midterm Exam – Oct 15, 2018
Professor Sarita Adve

Time: 2 Hours

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

Name	**Solution**
NetID	
Category	Undergraduate Graduate

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 **11 *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!

Problem	Maximum Points	Received Points
1	4	
2	11	
3	9	
4	14	
5	4	
6	7 **(for graduates only)**	
Total	42 for undergraduates 49 for graduates	

Problem 1 [4 points]

Assume 90% of a program can be executed in parallel.

Part A [2 points]

What speedup is required on the parallel section to achieve an overall speedup of 4X for the full program?

Solution: Let S be the speedup on the parallel section and T be the original total time.

.9T/S + .1T = T/4

.9/S + .1 = .25

.9/S = .15

S = .9 / .15 = 6. 6X speedup is required on the parallel section.

Grading: 2 points for the correct formula. 0 points for the final answer without showing the work.

Part B [2 points]

What is the maximum possible speedup achievable on the above program through parallelization?

Solution: In the limit, the parallel section reduces to 0, so speedup = T / 0.1T = 10X.

Grading: 2 points for the correct formula.

Problem 2 [11 points]

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

Consider the following architecture.

Functional Unit Type	Cycles in EX	Number of Functional Units
Integer	1	1
FP Add/Subtract	4	1
FP Multiply	5	1
FP Divide	16	1

- 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 "---"

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

Grading: ¼ point for each IS, EX, WB and CM entries; ¼ point bonus if all entries are correct (total of 8 points). 1/2 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 [9 points]

Consider a loop that is entered multiple times in a program. Each time it is entered, the loop performs 10 iterations. Each iteration executes five branches with the following outcomes (Branch 1 occurs before Branch 2 which occurs before Branch 3 which occurs before Branch 4 which occurs before Branch 5 in each iteration):

Iteration	1	2	3	4	5	6	7	8	9	10
Branch1	T	N	N	N	N	N	N	N	N	N
Branch2	T	T	T	N	N	N	T	T	T	T
Branch3	T	N	T	N	T	N	T	N	T	N
Branch4	T	T	N	N	N	T	T	T	T	N
Branch5	T	T	T	T	T	T	T	T	T	N

When Branch 5 is not taken at iteration 10, the program counter leaves the loop. Assume there are no other conditional branches in the program besides the ones above. (There may be unconditional jumps, but assume those are ignored by branch predictors.) Also assume that each branch has its own predictor entries; i.e., there is no aliasing among the multiple branches on the predictor entries.

Of all the dynamic branch predictors studied in class, state the predictor that will give the **best** misprediction rate for each of the following branches (don't worry about gshare or tagged hybrid predictors). State the misprediction rate of that predictor. In a case where multiple predictors will give the best misprediction rate, give the cheapest such predictor. Explain why. (No credit without explanation.) Focus on the steady state; i.e., ignore the first few times the loop is executed when evaluating your misprediction rate.

Part A [3 points]

Branch 1

Solution:
Branch 1 is always opposite the previous branch, which is branch 5. A global (1,1) predictor will always predict correctly, after the first execution of the loop.

Half credit for saying 2-bit predictor.

Part B [3 points]

Branch 2

Solution:

Branch 2's outcome follows the pattern established by the previous 3 branches (branches 4 and 5 from the previous iteration and branch 1 from the same iteration). Therefore, a (3,1) correlating predictor is best.

Partial credit (1 point) for saying 1-bit predictor

Part C [3 points]

Branch 5 – for this part and this part alone, do not consider correlating predictors

Solution:

Branch 5 is almost always taken. We need to use a 2-bit predictor. It will have one misprediction each time the code exits the loop. In contrast, a 1 bit predictor will have two mispredictions– at the entry and exit of each loop invocation.

Grading:

3 points for each part. Half credit for a predictor which is more expensive, but attains the minimal error rate. Half credit if the explanation gives an incorrect prediction rate. No credit without explanation.

Problem 4: [14 points]

Assume an in-order single-issue pipeline with multiple functional units. IF, ID, and WB take 1 cycle as usual, branches are resolved in ID, and all needed forwarding paths are provided. There is a fully pipelined, 3 cycle FP add unit (i.e., 3 cycles elapse from the time an FP add enters its functional unit to the time its result is available for a later instruction). There is a fully pipelined 5 cycle FP multiplier, and a 1 cycle integer ALU for all non-branch integer instructions. Loads and stores execute for 1 cycle in the integer ALU and spend 12 cycles in MEM. Consider the following code on the above machine:

```
loop:
   LD.D      F4,     0(R1)
   LD        R2, 4096(R1)
   LD.D      F10,    0(R2)
   MUL.D     F6, F4, F10
   ADD.D     F8, F8, F6
   DADDI     R1, R1, #8
   BNEQ      R1, R3, loop   // branch if R1 does not equal R3
```

Part (A) [4 points]
Give one pair of instructions with a RAW dependence that requires the most separation between them to avoid stalling? What is the minimum number of instructions that would need to be between this pair in program order to avoid the stall? Explain your answer.

Solution:
A load and an instruction that uses the load's result require the most separation. Any such pair is acceptable. Each of these pairs requires 12 instructions of separation. The load produces a result after 1 cycle of EX and 12 of MEM, and in this code all the instructions that need the result of the load will use that result in EX.

Grading:
2 points for any correct pair with correct explanation. 2 points for the number of instructions needed for separation. 1 point partial credit on the separation if the correct forwarding (MEM->EX) is mentioned.

Part (B) [2 points]
Consider the two dependent instructions from Part (A). Just for this part, ignore all other dependences in the loop body; i.e., assume that dependences for all other instructions magically do not require any stalls. Now consider unrolling the loop to remove any stalls between the instructions chosen in Part (A). What is the minimum number of original iterations that the new unrolled loop would need to have to eliminate these stalls? Remember to explain your answer.

Solution:
From part (a), a loop must be at least 14 instructions long to avoid stalling. There are 5 instructions of loop body which will be multiplied by unrolling, and two instructions of loop overhead which will not. Having two of the original iterations in the unrolled loop is insufficient (2*5+2 = 12 < 14), 3 iterations is enough.

Grading:
2 points for correct count, but only with explanation. 1 point partial credit for finding the number of instructions needed in the unrolled loop.

Part (C) [8 points]
Now consider the original loop with all the original dependences and stalls as in Part (A). Software pipeline the loop to run in as few cycles as possible, using the start-up code given below. You only have to produce the code for the steady state. Do not write or consider the performance of the clean-up code. Do not modify the start-up code. *Do not unroll the loop.*

Startup code:

```
LD      R2,  4096(R1)        // iteration 1
LD.D    F4,     0(R1)        // iteration 1
LD.D    F10,    0(R2)        // iteration 1
LD      R2,  (8+4096)(R1)    // iteration 2
DADDI   R1,  R1,  #16
```

Your steady-state code here:

Solution:

```
loop:
   MUL.D    F6, F4, F10          // iteration i
   LD.D     F4, -8(R1)           // iteration i +1
   LD.D     F10, 0(R2)           // iteration i+1
   LD       R2, 4096(R1)         // iteration i+2
   ADD.D    F8, F8, F6           // iteration i
   DADDI    R1, R1, #8
   BNEQ     R1, R3, loop
```

This avoids using load results for as long as possible. 8 stall cycles remain. (Multiply depends on both loads.)

Grading: 1 point for listing all the instructions from iteration i (1/2 point deducted for missing instructions), 1 point for listing all the instructions from iteration i+1 (1/2 point deducted for missing instruction), 1 point for listing instruction from iteration i+2, 1/2 point each for listing the DADDI and BNEQ instructions. 1/2 point each for listing correct offset and immediate for the loads and DADDI instructions. 2 points for correct ordering of instructions (1/4 point deducted for each incorrect order).

Problem 5 [4 Points]

Consider the following format for predicated MIPS instructions:

(pT) 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:

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 != R0)

pF = !(R8!= 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.

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) ADDI R2, R2, #1
(4) (pF) SW R2, 0(R7)
(5) (pT) DIV.D F0, F0, F2
(6) (pT) ADD.D F0, F4, F2
(7) (pT) S.D 0(R8), F0

L2: . . .

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.)

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.