CS 433 Midterm Exam – Oct 21, 2019

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	7	
2	15	
3	9	
4	18	
5	9	
6	6	
Total	58 for undergraduates 64 for graduates	

Problem 1 [7 points]

Identify all the data dependences (potential hazards) in the code below. Specify the associated instructions, registers, and whether the dependence is a RAW, WAW, or WAR. State whether the dependence will cause a stall. Consider the 5-stage MIPS pipeline. Branches are resolved in the ID stage. All stages take 1 cycle. Assume full forwarding. If the identified pair does not form a dependence at all, you will get negative points (so don't give all possible instruction pairs!).

1: LD R1, 0(R6)

2: LD R2, 4(R6)

3: ADD R3, R2, R1

4: SUB R2, R2, R1

5: BEQZ R2, dest

Solution
The dependences are:

1->3 RAW (R1)
1->4 RAW (R1)
2->3 RAW (R2) stall
2->4 RAW (R2)
2->4 WAW (R2)
3->4 WAR (R2)
4->5 RAW (R2) stall (EX -> ID)

Grading:
1 point for each dependence with correct stall information
½ point for a correct dependence with incorrect stall information
-¼ points for incorrect listings or listing RAR relationships

Problem 2 [15 points]

This problem concerns Tomasulo's algorithm (with reservation stations) with the reorder buffer as discussed in detail in the lecture notes, with the following changes/additions/ clarifications.

Functional Unit Type	Cycles in EX (NON-PIPELINED)	Number of Functional Units
Integer	1	1
FP Add/Subtract	3	1
FP Divide	6	1

1) Assume dual issue and dual commit.
2) Assume unlimited reservation stations.
3) Loads use the integer function unit to perform effective address calculation during the EX stage. They also access memory during the EX stage. Loads stay in EX for 1 cycle.
4) 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.
5) An instruction waiting for data on the CDB can move to EX in the cycle after the CDB broadcast.
6) Only one instruction can write to the CDB in one clock cycle. Branches/stores do not need the CDB.
7) When there is a conflict for a functional unit or the CDB, assume that the oldest (by program order) instruction gets access, while others are stalled.
8) Assume that the result from the integer functional unit is also broadcast on the CDB and forwarded to dependent instructions through the CDB (just like any floating point instruction).

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). For the last column, enter all the reasons that the instruction experiences a stall (leave blank if there is no stall). The first row is filled for you.

No	Instruction	IS	EX	WB	CM	Reason for stall
1	L.D F0, 0(R1)	1	2	3	4	---
2	ADD.D F0, F0, F3	1	4-6	7	8	RAW due to F0 from (1)
3	DIV.D F8, F0, F6	2	11-16	17	18	RAW due to F0 from (2) Structural Hazard due to (6)
4	L.D F6, 8(R1)	2	3	4	18	---
5	ADD.D F4, F6, F2	3	7-9	10	19	RAW due to F6 from (4) Structural hazard due to (2)
6	DIV.D F4, F6, F2	3	5-10	11	19	RAW due to F6 from (4)
7	L.D F6, 16(R1)	4	5	6	20	---

Grading: .5 points for each entry in the table. Cascading errors are not penalized.

Problem 3 [9 points]

Consider the following piece of code:

```
        DADDI  R1, R0, #100

L1:     DADDI  R1, R1, #-1
        BEQZ   R1, END      -- Branch 1
        DADDI  R12, R0, #2

L2:     DADDI  R12, R12, #-1
        BNEZ   R12, L2      -- Branch 2

        J          L1

END: ...
```

Assume R0 stores 0. Branch 1 is executed 100 times and branch 2 is executed a total of 198 times. For each branch, how many correct predictions will occur if we use the following prediction schemes (as discussed in class)? Assume separate prediction bits for each branch (i.e., no aliasing). Assume at the beginning of execution, the last branch was not taken. Please explain your answers.

Part A [3 points]

1-bit predictor initialized to T (taken).

Solution:

Branch 1: N, N, N, …, N, T is predicted correctly every time except the first and last.

Branch 2 alternates T, N, T, N, ... and thus is predicted incorrectly every time except the first.

Branch 1: 98 correct predictions, Branch 2: 1 correct prediction.

Grading: ½ point for each number and 1 point for each explanation.

Part B [3 points]

2-bit saturating predictor initialized to 10 (taken).

Solution:

Branch 1 is predicted correctly every time except the first and last.

Branch 2's predictor alternates between 10 and 11, and thus predicts correctly every other time.

Branch 1: 98 correct predictions, Branch 2: 99 correct predictions.

Grading: ½ point for each number and 1 point for each explanation.

Part C [3 points]

(1, 1) global correlating predictor, initialized to T/T.

Solution:

When Branch 1 is executed, the last branch is always not taken, so performance is the same as with a 1-bit predictor. Branch 2 is always taken when the last branch was not taken and not taken when the last branch was taken. Thus, there is one incorrect prediction - the first time it is not taken.

Branch 1: 98 correction predictions, Branch 2: 197 correct predictions.

Grading: ½ point for each number and 1 point for each explanation.

Problem 4: [18 points]

Consider the following code fragment:

```
Loop:  LD.D      F1, 0(R1)
       LD.D      F2, 0(R2)
       MUL.D     F3, F1, F10
       ADD.D     F4, F3, F2
       SD.D      F4, 0(R1)
       DADDUI    R1, R1, #8
       DADDUI    R2, R2, #8
       BNE       R1, R3, Loop
```

Consider a pipeline with the following latencies: 3 cycles between an FP multiply and its consumer, 1 cycle between an FP add and its consumer, and 0 cycles between all other pairs. Thus, there should be three stall cycles between the multiply and addition in the above code for correct operation. Assume that all functional units are pipelined.

Unroll the above loop 4 times and write the resulting code to the left of the table on the next page (the above loop is repeated on the next page for your convenience). You have access to temporary registers T0…T63. Assume that the total number of iterations for the original loop is a multiple of 4. Schedule the unrolled loop for a VLIW machine where each VLIW instruction can contain one memory reference, one FP operation, and one integer operation. Write the scheduled instructions in the table on the next page to minimize the number of stalls. You may use L for L.D, M for MUL.D, etc.

```
Loop:  LD.D      F1, 0(R1)
       LD.D      F2, 0(R2)
       MUL.D     F3, F1, F10
       ADD.D     F4, F3, F2
       SD.D      F4, 0(R1)
       DADDUI    R1, R1, #8
       DADDUI    R2, R2, #8
       BNE       R1, R3, Loop
```

```
Loop:  LD.D F1, 0(R1)
       LD.D F2, 0(R2)
       MUL.D F3, F1, F10
       ADD.D F4, F3, F2
       SD.D F4, 0(R1)

       LD.D T1, 8(R1)
       LD.D T2, 8(R2)
       MUL.D T3, T1, F10
       ADD.D T4, T3, T2
       SD.D T4, 8(R1)

       LD.D T5, 16(R1)
       LD.D T6, 16(R2)
       MUL.D T7, T5, F10
       ADD.D T8, T7, T6
       SD.D T8, 16(R1)

       LD.D T9, 24(R1)
       LD.D T10, 24(R2)
       MUL.D T11, T9, F10
       ADD.D T12, T11, T10
       SD.D T12, 24(R1)

       DADDUI R1, R1, #32
       DADDUI R2, R2, #32
       BNE R1, R3, Loop
```

Mem	FP ALU	Integer ALU
LD.D F1, 0(R1)		
LD.D T1, 8(R1)	MUL.D F3, F1, F10	
LD.D T5, 16(R1)	MUL.D T3, T1, F10	
LD.D T9, 24(R1)	MUL.D T7, T5, F10	
LD.D F2, 0(R2)	MUL.D T11, T9, F10	DADDUI R1, R1, #32
LD.D T2, 8(R2)	ADD.D F4, F3, F2	
LD.D T6, 16(R2)	ADD.D T4, T3, T2	
LD.D T10, 24(R2)	ADD.D T8, T7, T6	
SD.D F4, -32(R1)	ADD.D T12, T11, T10	DADDUI R2, R2, #32
SD.D T1, -24(R1)		
SD.D T5, -16(R1)		
SD.D T9, -8(R1)		BNE R1, R3, Loop

Grading: 9 points for code on left, 9 points for table. Minimum score is 0 for both.
Deductions (only for first error, no cascading errors):

- -2 points for badly ordered loads.
- -2 points for inconsistent load/store addresses.
- -2 points for badly ordered MULs or ADDs.
- -2 points for late stores.
- -2 points for late branches/integer operations.
- -2 points for missing or wrong-path operations.
- -2 points if instruction uses wrong functional unit.
- -2 points if R1/R2 are updated every iteration.
- -2 points for inconsistent DADDUI offsets.
- -2 points for violating a RAW dependency.
- -1 point if temporary registers are out of bounds (not from T0-T63).
- -1 point if branch delay slot used but not stated in assumptions.
- No deduction for not using temporary registers.

If you did not write the unrolled code, we immediately deducted ½ points. Then we took your VLIW scheduled code and graded it according to the above deductions.

Problem 5: [9 points]
Consider the classic five stage pipeline as discussed in class, with the following extensions/clarifications:

- The MEM stage takes 4 cycles but is pipelined.
- Branches are resolved in the decode stage, and have one branch delay slot.
- All instructions take one cycle in the EX stage. Address calculation for memory instructions occurs in the EX stage as in the basic pipeline.
- Assume all forwarding paths as needed.

Consider the loop below. It calculates the sum of a list of numbers stored in an indirect representation. Instead of storing the number, the locations 0(R1), 4(R1), 8(R1), etc. contain the addresses of the values. This is similar to the memory access pattern of sparse matrix codes. The sum is accumulated in F1.

```
loop:
1)      LD R3, 0(R1)
2)      LD.D F2, 0(R3)
3)      ADD.D F1, F1, F2
4)      SUBIU R2, R2, #1
5)      BNEZ R2, loop
6)      ADDIU R1, R1, #4
```

Part A [3 points] List all the stalls incurred by the above loop and the reason for the stalls. If a dependence results in multiple stall cycles, indicate the number of cycles.

Solution:
1->2: 4 stall cycles due to RAW on R3.
2->3: 4 stall cycles due to RAW on F2.
4->5: 1 stall cycle due to RAW on R2

Grading: ½ point for identifying each stall. ½ point for the correct number of cycles. 3 points total.

Part B [4 points]
Software pipeline the loop to minimize the stalls. Assume infinite registers are available. Only the most efficient solution will fetch a perfect score. Only provide the steady state code. Do not worry about start-up and finish-up code. (Do not unroll the loop for this part.). The original loop is included at the top of this page for convenience.

Solution:

```
loop:
        LD.D F2, 0(R3)
        LD R3, 0(R1)
        SUBIU R2, R2, #1
        ADDIU R1, R1, #4
        BNEZ R2, loop
        ADD.D F1, F1, F2
```

Grading: 2 points if the solution appears to "spread out" the two loads and the add in different iterations. 3 points for a mostly correct solution. 4 points for a perfect solution.

Part C [1 point]
What is the advantage of using software pipelining over loop unrolling for the above code?

Solution: There are fewer static instructions in a software pipelined loop and so there is less pressure on the instruction cache.

Grading: 1 point for identifying fewer static instructions as the reason.

Part D [1 point]
What is the advantage of using loop unrolling over software pipelining for the above code?

Solution: There are fewer dynamic instructions because the loop overhead is reduced.
Grading: 1 point for correct answer.

ONLY GRADUATE STUDENTS SHOULD SOLVE THE NEXT PROBLEM.

Problem 6: [6 points]

You are a member of a team designing an out-of-order processor with dynamic scheduling and speculative execution. Your initial design was just reviewed by the circuit implementation team, and it turns out that you have some spare transistor budget! (A rare occurrence in practice.)

Your processor currently has a small 2-bit saturating counter-based branch predictor which performs moderately well. It has 8 Integer Functional Units and 4 Floating Point Units (FPUs), 256KB of on-chip caches, 4 reservation stations for the Integer Units, and 2 reservation stations for FPUs. The Reorder Buffer has 8 entries. The processor has a 25 stage pipeline.

The applications you care for have a small code size and work on small data sets in the range of 64 KB. These applications spend most of their time in loops whose iterations are independent of each other, but typically have only a limited amount of ILP within a single iteration (within the current processor implementation).

You can use the extra transistors in (possibly several of) the following ways:

1. Improve the branch predictor accuracy.
2. Add more reservation stations to your Tomasulo's Algorithm-based Dynamic Scheduler.
3. Add more FPUs and Integer Units.
4. Add more Reorder Buffer entries.

Some of these may be desirable additions, while others may not be too beneficial given the current configuration. There is a meeting coming up to discuss the proposed additions. Which of the above four additions should you support and which ones should you oppose (you can support/oppose multiple of these)? You need to justify your choices to receive credit.

Solution:

1. Improve the branch predictor accuracy: This is a desirable addition. The problem states that the branch predictor performs only moderately well and the processor has a long pipeline making branch mispredictions expensive. Thus, improving branch prediction accuracy is quite likely to increase performance.

2. Add more reservation stations to your Tomasulo's Algorithm-based Dynamic Scheduler: This is a desirable addition. More reservation stations would mean a larger window within which the processor can search for ready instructions to execute, thus it can discover more parallelism and keep execution units busy. This would lead to better performance, especially since our application needs to discover parallelism across loop iterations.

3. Add more FPUs and Integer Units: This doesn't seem to be a good addition. The current machine already has enough FUs and we should try to improve other aspects of the processor. Adding more FUs

won’t help if the processor is unable to discover enough parallelism in the instruction stream to keep them busy.

4. Add more Reorder Buffer entries: This is a desirable addition. The current configuration has very few ROB entries. A large ROB helps to mask out the effects of long latency instructions and help search for parallelism within a larger window (this goes together with (3)).

Grading:
1.5 points for correctly analyzing each part. 6 points total. Give partial credit (0.5 points) if student gives valid reason for why improvement can be avoided.