CS 433 Midterm Exam: Oct 18, 2021

Professor Sarita Adve

Time: **2 Hours**

Please print your Name and NetID and circle your course section below.

Name:	Instructor	
NetID:		
Section:	T3 (Undergraduate)	T4 (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 **5 problems** and **16 pages** (including this one). **All students** should solve **problems 1 through 4**. Only **graduate students should solve problem 5**. Please budget your time appropriately. Good luck!

Problem	**1**	**2**	**3**	**4**	**Graduate students 5**	**Total**
Points	6	17	26	14	8	63 (undergrads) 71 (graduates)
Score						

Problem 1 [6 Points]

Suppose an important program you run has the following characteristics.

Instruction Type	% of Execution Time
Load from memory	12%
Store to memory	6%
FP Multiplication	17%
Other	65%

Part A [3 points]

You are considering upgrading your machine to one of two possible configurations. M1 reduces the contribution of loads to the execution time by 3$\times$ and that of stores by 2$\times$. M2 reduces the contribution of (only) floating-point multiplications to the execution time by 10$\times$. What is the overall speedup of the program for each upgrade? You may express your answer in terms of an equation with all variables explicitly substituted. You are not required to perform numerical calculations.

$$T_{M1} = 0.12\frac{1}{3} + 0.06\frac{1}{2} + 0.17 + 0.65 = 0.04 + 0.03 + 0.82 = 0.89$$

$$S_{M1} = \frac{1}{0.89} \approx 1.124 = 12.4\%$$

$$T_{M2} = 0.12 + 0.06 + 0.17\frac{1}{10} + 0.65 = 0.017 + 0.83 = 0.847$$

$$S_{M2} = \frac{1}{0.847} \approx 1.181 = 18.1\%$$

Grading:

1.5 points for the speedup equation for M1.
1.5 points for the speedup equation for M2.

Part B [3 points]

What is the maximum possible speedup if all memory accesses could be sped up infinitely? What is the maximum speedup if all (and only) floating-point multiplications could be sped up infinitely? You may express your answer in terms of an equation with all variables explicitly substituted. You are not required to perform numerical calculations.

$$S_{M1} = \frac{1}{0.17+0.65} = \frac{1}{0.82} \approx 1.220 = 22.0\%$$

$$S_{M2} = \frac{1}{0.12+0.06+0.65} = \frac{1}{0.83} \approx 1.205 = 20.5\%$$

Grading:

1.5 points for the speedup equation for M1.
1.5 points for the speedup equation for M2.

Problem 2 [17 Points]

Consider running the following code on a machine with the assumptions below:

```
1.  loop:   L.D      F1,  0(R5)
2.          S.D      F0,  0(R5)
3.          ADD.D    F0,  F0,  F1
4.          DADDIU   R4,  R4,  #-1
5.          DADDIU   R5,  R5,  #8
6.          BNEZ     R4,  loop
```

- The machine has a **dual-issue**, out-of-order processor with hardware speculation, **a reorder buffer with 8 entries**, and the following functional units:

Functional Unit Type	Number of Functional Units	Cycles in EX
Integer ALU	1	1
FP Adder	1	4 (not pipelined)

- 2 instructions can be issued and committed each cycle.
- Loads use the Integer ALU for effective address calculation and for memory access during the EX stage. It takes a single cycle for the ALU to do both the address calculation and memory access.
- Stores use the integer ALU for effective address calculation in the EX stage and access memory in the CM stage (1 cycle for each).
- Branches use the Integer ALU for all computation in the EX stage. Branch direction and target are predicted perfectly.
- Instructions following a branch cannot issue in the same cycle as a branch.
- Branches do not have a branch delay slot.
- Only one instruction can write to the CDB in each clock cycle.
- 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.
- If an instruction cannot move from EX to its WB stage in cycle x, it continues occupying the functional unit it was using.
- An instruction waiting for data from the CDB can move to its EX stage in the cycle after the CDB broadcasts the data.
- Branches and stores do not need the CDB.
- Whenever there is a conflict for a functional unit or the CDB, assume that the oldest, by program order, of the conflicting instructions gets access, while others are stalled.
- Assume that the result from the Integer ALU is also broadcast on the CDB and forwarded to dependent instructions through the CDB, just like any floating point instruction.

The following table shows the first 11 instructions of the above code. Fill in the cycles each instruction is in each stage. Note **all reasons for all stalls**, including the type of hazard; the functional unit or register it is dependent on; and the instruction it is dependent on. **Note any stalls due to commit ordering and/or limited commit width.** Write "**None**" if there are no stalls. Some entries are already written for you.

#	Instruction	IS	EX	WB	CM	Reasons for Stalls
1	L.D F1, 0(R5)	1	2	3	4	None
2	S.D F0, 0(R5)	1	3	—	4	Structural INT from 1
3	ADD.D F0, F0, F1	2	4–7	8	9	RAW F1 from 1
4	DADDIU R4, R4, #-1	2	4	5	9	Structural INT from 2 In-order commit
5	DADDIU R5, R5, #8	3	5	6	10	Structural INT from 4 In-order commit Commit width
6	BNEZ R4, *loop*	3	6	—	10	RAW R4 from 4 In-order commit
7	L.D F1, 0(R5)	4	7	9	11	RAW R5 from 5 Structural CDB from 3 Commit width
8	S.D F0, 0(R5)	4	9	—	11	Structural INT from 7 RAW R5 from 5 In-order commit
9	ADD.D F0, F0, F1	5	10–13	14	15	RAW F0 from 3 RAW F1 from 7
10	DADDIU R4, R4, #-1	5	10	11	15	Structural INT from 6, 7, 8 In-order commit
11	DADDIU R5, R5, #8	10	11	12	16	ROB Full from 3 In-order commit Commit width

Grading:

1/2 point for each entry. ½ point if all entries are correct.

Cascading errors will not be penalized as long as the relevant dependencies are still observed.

Problem 3 [26 Points]

Consider a single-issue, in-order five stage pipeline similar to those studied in class, but with the following specification:

Functional Unit	Cycles in EX	Number of Functional Units	Pipelined
Integer	1	1	Yes
FP Add/Subtract	3	1	Yes
FP/Integer Multiplier	8	1	Yes
FP/Integer Divider	24	1	No

- 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 a functional unit 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.
- While the hardware has full forwarding and bypassing, it is the responsibility of the compiler to schedule such that the operands of each instruction are available when needed by each instruction.
- 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. Consider the following code (also repeated on the next pages for reference):

```
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)
        DADDIU   R1,   R1,    #8
        DADDIU   R2,   R2,    #8
        DADDIU   R3,   R3,    #8
        DSUB     R5,   R4,    R1
        BNEZ     R5,   loop
```

Part A [6 Points]

Rewrite the above loop (repeated below for reference), but let every row take a cycle (each row can be an instruction or a stall). 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. Assume that a NOP is scheduled in the branch delay slot (effectively stalling 1 cycle after the branch). *Explain the cause of all stalls*, but don't reorder instructions. How many cycles elapse before the second iteration begins?

```
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)
       DADDIU   R1,   R1,   #8
       DADDIU   R2,   R2,   #8
       DADDIU   R3,   R3,   #8
       DSUB     R5,   R4,   R1
       BNEZ     R5,   loop
```

```
loop:     L.D      F4,    0(R1)
          stall    RAW    F4
          MUL.D    F8,    F4,    F0
          L.D      F6,    0(R2)
          stall    RAW    F6
          ADD.D    F10,   F6,    F2
          stall    RAW    F8, F10
          stall    RAW    F8, F10
          stall    RAW    F8
          stall    RAW    F8
          ADD.D    F12,   F8,    F10
          stall    RAW    F12
          S.D      F12,   0(R3)
          DADDIU   R1,    R1,    #8
          DADDIU   R2,    R2,    #8
          DADDIU   R3,    R3,    #8
          DSUB     R5,    R4,    R1
          stall    RAW    R5 (branch resolved in ID)
          BNEZ     R5,    loop
          NOP      stall for branch delay
```

20 cycles elapse before the second iteration begins.

Grading:

1 point for each sequence of stalls; ½ point partial credit for identifying that a stall is needed 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.

```
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)
       DADDIU    R1,   R1,    #8
       DADDIU    R2,   R2,    #8
       DADDIU    R3,   R3,    #8
       DSUB      R5,   R4,    R1
       BNEZ      R5,   loop
```

```
loop:       L.D       F4,    0(R1)
            L.D       F6,    0(R2)
            MUL.D     F8,    F4,    F0
            ADD.D     F10,   F6,    F2
            DADDIU    R1,    R1,    #8
            DADDIU    R2,    R2,    #8
            DADDIU    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 ½ point for each error e.g. incorrect index.
Deduct ½ 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. Assume a very large number of iterations for the original loop. Show your work.

```
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)
       DADDIU   R1,    R1,    #8
       DADDIU   R2,    R2,    #8
       DADDIU   R3,    R3,    #8
       DSUB     R5,    R4,    R1
       BNEZ     R5,    loop
```

Note that in the solution 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
          DADDIU    R1,   R1,   #16
          DADDIU    R2,   R2,   #16
          DADDIU    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)
```

The loop has an unroll factor of 2: there are two iterations of the original loop in a single iteration of the new loop. 17 cycles elapse before the next iteration of the unrolled loop begins.

Grading:

1 point for the correct iteration count.
Deduct ½ point for each 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.

Part D [8 Points]

Consider a VLIW processor in which one instruction can support two memory operations (load or store), one integer operation (addition, subtraction, comparison, or branch), one floating point add or subtract, and one floating point multiply or divide. There is no branch delay slot. Now unroll the original loop four times (i.e., four original iterations in one new iteration), and schedule it for this VLIW processor to take as few stall cycles as possible. How many cycles do the four iterations take to complete? Use the table template on the next page to show your work.

```
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)

            L.D         F14,    8(R1)
            MUL.D       F18,    F14,    F0
            L.D         F16,    8(R2)
            ADD.D       F20,    F16,    F2
            ADD.D       F22,    F18,    F20
            S.D         F22,    8(R3)

            L.D         F24,    16(R1)
            MUL.D       F28,    F24,    F0
            L.D         F26,    16(R2)
            ADD.D       F30,    F26,    F2
            ADD.D       F32,    F28,    F30
            S.D         F32,    16(R3)

            L.D         F34,    24(R1)
            MUL.D       F38,    F34,    F0
            L.D         F36,    24(R2)
            ADD.D       F40,    F36,    F2
            ADD.D       F42,    F38,    F40
            S.D         F42,    24(R3)

            DADDIU      R1,     R1,     #32
            DADDIU      R2,     R2,     #32
            DADDIU      R3,     R3,     #32
            DSUB        R5,     R4,     R1
            BNEZ        R5,     loop
```

#	MEMORY 1	MEMORY 2	INTEGER	FP ADD/SUB	FP MUL/DIV
1	L.D F4, 0(R1)	L.D F6, 0(R2)			
2	L.D F14, 8(R1)	L.D F16, 8(R2)			
3	L.D F24, 16(R1)	L.D F26, 16(R2)		ADD.D F10, F6, F2	MUL.D F8, F4, F0
4	L.D F34, 24(R1)	L.D F36, 24(R2)		ADD.D F20, F16, F2	MUL.D F18, F14, F0
5				ADD.D F30, F26, F2	MUL.D F28, F24, F0
6				ADD.D F40, F36, F2	MUL.D F38, F34, F0
7			DADDIU R1, R1, #32		
8			DADDIU R2, R2, #32		
9			DADDIU R3, R3, #32		
10			DSUB R5, R4, R1		
11				ADD.D F12, F8, F10	
12				ADD.D F22, F18, F20	
13	S.D F12, -32(R3)			ADD.D F32, F28, F30	
14	S.D F22, -24(R3)			ADD.D F42, F38, F40	
15	S.D F32, -16(R3)				
16	S.D F42, -8(R3)		BNEZ R5, loop		
17					
18					
19					
20					

Grading:

1 point for scheduling all the L.D correctly.
1 point for scheduling all the ADD.D that add from the load value correctly.
1 point for scheduling all the MUL.D that multiply from the load value correctly.
1 point for scheduling all the ADD.D that add from the ADD.D and MUL.D correctly.
1 point for scheduling all the DADDIU correctly.
½ point for scheduling the DSUB correctly.
1 point for scheduling all the stores correctly.
½ point for scheduling the branch correctly.
1 point for correctly unrolling the loop with proper register renaming and offset.s

Do not take off points for *DADDIU R1, R1, #32* or *DADDIU R2, R2, #32* as long as they are scheduled on cycle 5 or later.
Do not take off points for *DADDIU R3, R3, #32* as long as it is scheduled on or before cycle 12.
Do not take off points for *DSUB R5, R4, R1* as long as it is scheduled after *DADDIU R1, R1, #32* and there is one cycle between the DSUB and the branch, since the branch needs its value in ID.

Problem 4 [14 Points]

Consider a piece of code with three static branch instructions, B1, B2, and B3. During an execution of this code, the global history for these branches (i.e., their execution sequence and direction) is as follows:

Branch	B1	B2	B1	B2	B3	B1	B1	B2	B1
Direction	T	N	N	T	T	T	T	N	T

T stands for Taken; N stands for Not-taken. Thus, the execution starts with branch B1 being taken, then B2 is not taken, etc. Before the above execution, the outcome of all previous branches and the initial state of all predictors is not taken (N). Assume a (2, 1) correlating predictor is used for branch B1. Assume the state of the predictor is recorded in the form W/X/Y/Z where:

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

Assume the history at a branch is recorded in the form H_{-2} H_{-1} where H_{-1} is the last branch and H_{-2} is the branch before the last one.

Fill the 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. (Recall that by local history for B1, we mean the history for only branch B1.)

Table 1: Assume the predictor uses LOCAL branch history

Branch B1 invocation #	**History used for prediction**	**Prediction for B1**	**Actual direction of B1**	**New Predictor State**
1	**N N**	N	**T**	N/N/N/T
2	N T	N	**N**	N/N/N/T
3	T N	N	**T**	N/N/T/T
4	N T	N	**T**	N/T/T/T
5	T T	N	**T**	T/T/T/T

Table 2: Assume the predictor uses GLOBAL branch history

Branch B1 invocation #	**History used for prediction**	**Prediction for B1**	**Actual direction of B1**	**New Predictor State**
1	**N N**	N	**T**	N/N/N/T
2	T N	N	**N**	N/N/N/T
3	T T	N	**T**	T/N/N/T
4	T T	T	**T**	T/N/N/T
5	T N	N	**T**	T/N/T/T

ONLY GRADUATE STUDENTS SHOULD SOLVE PROBLEM 5

Problem 5 [8 Points]

In this problem, we try to understand the implications of the Reorder Buffer (ROB) size on performance. Consider a processor implementing the ROB scheme described in class. Recall each instruction goes through issue (IS), writeback (WB), execute (EX), and commit (CM).

- Assume IS, WB, and CM each take one cycle (once all the conditions for these stages are met), as discussed in class.
- Assume our machine can fetch and commit 4 instructions each cycle.
- Assume a branch misprediction is handled when the branch instruction reaches the head of the ROB. It involves flushing that ROB entry and all entries following that entry.
- For now, assume there are no memory accesses (this will change in part C).

Part A [2 Points]

Suppose we have a perfect branch predictor and there is no data dependency between instructions. We have infinite execution units of each type and infinite reservation stations. All instructions take one cycle in the EX stage. What is the maximum achievable IPC? What is the minimum ROB size required to guarantee that IPC?

Each instruction holds its ROB entry for 4 cycles (Issue, Execute, Writeback, and Commit). Therefore, we must have a minimum ROB size of 16 to avoid any stalls. In that case, the throughput would be 4 instructions per cycle.

Grading:

1 point for correct IPC.
1 point for correct ROB size.

Part B [2 Points]

Suppose different FUs have different latencies in the EX stage, as given by the following table. Everything else is the same as in the previous part. What is the minimum size of the ROB required now to avoid any issue stalls due to a full ROB?

Functional Unit	Cycles
Integer ALU	1
FP Adder	5
FP Multiplier	10

If we issue an instruction to the FP multiplier, it would occupy its ROB entry for 13 cycles, and would also block the instructions following it from committing. During that period, we could have issued 52 instructions in all. Thus we need a minimum ROB size of 52 to avoid stalls. In that case we would get a throughput of 4 instructions per cycle.

Grading:

1 point for realizing that FP multiplier is the bottleneck.
1 point for correct ROB size.

Part C [2 Points]

In addition to the latencies above, now every 10th instruction is a load instruction. Assume the address calculation and cache/memory access parts of the load both happen in the EX stage. The hit rate in the data cache is 95% and the misses are uniformly spaced through the instruction stream. A hit takes 1 cycle in the EX stage. However, upon a miss, the data has to be fetched from the memory and this results in 100 cycles in the EX stage. What is the ROB size required now to avoid any issue stalls?

A load instruction that misses in the cache would occupy its ROB entry for $1 + 100 + 1 + 1 = 103$ cycles. During this time, we could have issued 412 instructions. This is the size of the ROB required to continuously issue 4 instructions each cycle.

Grading:

1 point for calculating the correct latency of loads.
1 point for correct ROB size.

Part D [2 Points]

Now additionally assume we don't have perfect branch prediction anymore. Instead, we have a predictor with an accuracy of 95%. Assume every 8th instruction is a branch and the mispredictions are uniformly spaced through the instruction stream.

After a misprediction, how many instructions are issued before the next misprediction is encountered? In light of this result, do you think we need a ROB of the size you derived in Part C? Why/why not? If not, what do you think would be a good ROB size to have?

We would issue $8 \times 20 = 160$ instructions before a mispredict is encountered. Given this result, it doesn't make sense to have a ROB of size 412, because once we have a cache miss, soon after we would run into a branch mispredict, and all the work done after that would be wasted anyway. A reasonable ROB size would be 160.

Grading:

1 point for stating that a smaller ROB is sufficient.
1 point for correct ROB size.