CS 433 Midterm Exam: Oct 10, 2022

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 **13 pages** (including this one). **All students** should solve **problems 1, 2A, and 3 through 5**. Only **graduate students should solve problem 2B and 2C**. Please budget your time appropriately. Good luck!

Problem	**1**	**2**	**3**	**4**	**5**	**Total**
Points T3	4	16	9	18	9	56 (undergrads)
T4	4	26	9	18	9	66 (graduates)
Score						

Problem 1 [4 points]

Assume 90% of a sequential program's execution time can be parallelized.

Part A [2 points]

What speedup (from parallelism) is required on the parallelizable section to achieve an overall speedup of 4X for the full program? You may express your answer in terms of an equation with all variables explicitly substituted. You are not required to perform numerical calculations.

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 [16 points for undergraduates, 26 points for graduates]

This problem concerns Tomasulo's algorithm. Consider running the following loop on an architecture specified below.

```
1.   LP:  L.D      F0, 0(R1)
2.        ADD.D    F0, F0, F6
3.        DIV.D    F2, F2, F0
4.        L.D      F0, 8(R1)
5.        DIV.D    F4, F0, F8
6.        S.D      F4, 16(R1)
7.        DADDI    R1, R1, #-24
8.        BNEZ     R1, LP
```

Functional Unit Type	**Cycles in EX**	**Number of Functional Units**
Integer	1	1
FP Adder	5	1
FP Divider	15	1

1) Assume that you have unlimited reservation stations.
2) Memory accesses use the integer functional unit to perform effective address calculation during the EX stage. For stores, memory is accessed during the EX stage (Tomasulo's algorithm without speculation) or commit stage (Tomasulo's algorithm with speculation). All loads access memory during the EX stage. Loads and Stores stay in EX for 1 cycle.
3) Functional units are not pipelined.
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 its EX stage in the cycle after the CDB broadcast.
6) Only one instruction can write to the CDB in one clock cycle. Branches and stores do not need the CDB.
7) 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.
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).
9) Assume that the BNEZ occupies the integer functional unit for its computation and spends one cycle in EX.

Part A [16 points]

Complete the following table using Tomasulo's algorithm but without assuming any hardware speculation on branches. That is, an instruction after a branch cannot issue until the cycle after the branch completes its EX. Assume a ***single-issue*** machine. Fill in the cycle numbers in each pipeline stage for the first several instructions of the loop as given below, assuming the branch is always taken. The entries for the first instruction are filled in for you. Explain the reasons for any stalls.

Instruction	**IS**	**EX**	**WB**	**Reason for Stalls**
Iteration 1				
1. LP: L.D F0, 0(R1)	1	2	3	
2. ADD.D F0, F0, F6	2	4-8	9	Data Dependence F0 (from 1)
3. DIV.D F2, F2, F0	3	22-36	37	Data Dependence F0 (from 2); FP div unit occupied (from 5)
4. L.D F0, 8(R1)	4	5	6	No stalls
5. DIV.D F4, F0, F8	5	7-21	22	Data dependence F0 (from 4)
6. S.D F4, 16(R1)	6	23	--	Data dependence F4 (from 5)
7. DADDI R1, R1, #-24	7	8	10	CDB occupied (from 2)
8. BNEZ R1, LP	8	11	--	Data dependence R1 (from 7)
Iteration 2				
9. L.D F0, 0(R1)	12	13	14	Control dependence

Grading: ½ a point for each correct entry. Cascading errors will not be penalized additionally as long as the relevant dependencies are still observed.

ONLY GRADUATE STUDENTS SHOULD SOLVE THE FOLLOWING PARTS B and C FOR PROBLEM 2.

Part B [4 points]

To improve the performance of the above loop execution, you are tasked with adding support for speculative execution. However, instead of using a reorder buffer (ROB) to store uncommitted (potentially speculative) values, your colleague proposes to build on the concept of renaming used in Tomasulo's algorithm and leverage the available large set of physical registers. The idea is to hold potentially speculative, uncommitted values that usually reside in the ROB in an extended set of physical registers. During instruction issue, a renaming process maps the names of the logical (i.e., architectural) registers (R0 to R31, F0 to F31) to this extended physical register set (P0 to P255), allocating a new unused register for the destination from a queue of free P-registers.

Using this idea, work out the register renaming necessary to enable the speculative execution of (part of) the second loop iteration in the table below. Assume that at the time the sequence in the table begins, registers R0 to R31 are mapped to P0 to P31 respectively and registers F0 to F31 are mapped to P32 to P63 respectively. Assume P64 to P255 are all on the free list and allocation of free registers is done in order from P64 to P255 (the lowest numbered one first), followed by any others that may be freed up in the execution below.

Instruction	**Instruction with renamed registers**	**Changes to the rename map**
BNEZ R1, LP	BNEZ P1, LP	
Iteration 2		
L.D F0, 0(R1)	L.D P64, 0(P1)	F0 → P64
ADD.D F0, F0, F6	ADD.D P65, P64, P38	F0 → P65
DIV.D F2, F2, F0	DIV.D P66, P34, P65	F2 → P66
L.D F0, 8(R1)	L.D P67, 8(P1)	F0 → P67
DIV.D F4, F0, F8	DIV.D P68, P67, P40	F4 → P68

Grading: 1 point for each correct row. No cascading errors.

Part C [6 points]

Suppose that the first load instruction - L.D F0, 0(R1) - in the second loop iteration raises an exception. Assume the exception is raised after all the instructions in the table have arrived in the reorder buffer. Also assume the hardware provides precise exceptions.

When should the above exception be handled? For precise exceptions, the instructions after the load must be squashed before the exception is handled. Explain what steps the hardware needs to take to restore the correct architectural state after these instructions are squashed. If there is any additional information that the hardware needs to track to enable this, state the information clearly. Illustrate your answer by applying it to at least one of the squashed instructions.

Solution:
The exception must be handled only when the load is at the head of the reorder buffer. [1 point.]

To restore the correct architectural state, we need to restore the mappings from the architectural to the physical registers to the same state as when the load entered the reorder buffer. [1 point.] To achieve this, the hardware needs to keep track of the previous mapping before it makes a new mapping for the destination register of a newly arrived instruction. [1 point.] To squash the instructions, the hardware needs to restore this old mapping in reverse program order (starting from the tail of the reorder buffer towards the head). [1 point.] It also needs to send the new (current) mapped register into the free list. [1 point.]

For example, when the last divide instruction is issued, the hardware needs to keep track of the old mapping for F4 (which in this case is T36). When the exception is handled, it starts by restoring the mapping of F4 to T36 and frees up T68 (the current mapping). [1 point.]

Grading: 6 points for a complete solution. Partial credit allocation is as described in the solution.

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? 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 direction: N,N,N,…,N,T. This 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 counter 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 only one incorrect prediction - the first time it is not-taken.
Branch 1 : 98 correct 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
       BNEZ      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. Assume the machine does NOT support delayed branches.

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 best performance on 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
        BNEZ   R1, R3, Loop
```

Please write unrolled loop below

```
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
        BNEZ 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)		BNEZ R1, R3, Loop

Grading:
8 points for unrolling the loop correctly, following the break-down below:

- 1 points repeating first 5 loop body instructions 4 times
- 2 = 4 x ½ points for correct offset adjustments in each of the 4 unrolled iterations
- 2 = 4 x ½ points for correct use of temporary registers in each of the 4 unrolled iterations
- 2 points for using only 2 DADDUI instructions to update indexes correctly
- 1 point for using only one correct branch instruction at the end

10 points for scheduling the unrolled loop correctly while minimizing the overall amount of cycles, following the break-down below:

- 2 points for scheduling all the L.D correctly.
- 2 points for scheduling all the S.D correctly.
- 2 points for scheduling all the MUL.D that multiply from the load value correctly.
- 2 points for scheduling all the ADD.D that add from the load value correctly.
- 1 point for scheduling all the DADDIU correctly.
- 1 point for scheduling the BNEZ branch correctly

If the sequence of unrolled instructions was incorrect, points for scheduling of instructions are still awarded as long as the scheduling remains consistent.
If instructions are scheduled correctly, but not optimally with respect to the total number of cycles, we deduct the points for the scheduled instructions that cause the number of cycles to unnecessarily increase, but do not penalize cascading shifts for when following instructions are scheduled.
Furthermore, we do not deduct points for an altered scheduling of DADDIU R1, R1, #32 or DADDIU R2, R2, #32 as long as offsets remain consistent and latencies are honored.

Problem 5 [9 points]

Consider the following code fragment from an if-then-else statement of the form

```
if (A == 0) A = B;
else A = A  + 4;
```

where A is at 0(R3) and B is at 0(R2):

```
1.          LD        R1, 0(R3)     ; load A
2.          BNEZ      R1, L1        ; test A
3.          LD        R1, 0(R2)     ; then clause
4.          J         L2            ; skip else
5.    L1:   DADDI     R1, R1, #4    ; else clause
6.    L2:   SD        R1, 0(R3)     ; store A
```

The machine on which the code will run does not have hardware speculation but does support compiler speculation where speculative instructions are marked with a speculation bit and poison bits are provided to deal with exceptions on speculative instructions. You are told that if two loads appear one after another in program order and happen to miss in the cache, then the machine can pipeline them in the memory system with significant performance boost. (We will learn about this type of load optimization soon in class. It is not necessary to know how it works to solve this problem.)

Given the above, you modify the above code as below to exploit compiler speculation and move the two loads next to each other for a possible performance boost. The second load is speculative (denoted by (s)).

```
      LD        R1, 0(R3)     ; load A
      (s)LD     R14, 0(R2)    ; speculative load B
      BEQZ      R1, L3        ; other branch of the if
      DADDI     R14, R1, #4   ; else clause
L3:   SD        R14, 0(R3)    ; store A
```

Part A [4 points]

Assume A = 0 at the beginning of the execution and that the speculative load above incurs an exception. Fill the following table for the execution of your modified code above, showing the instructions executed (in program order), the speculative bit for the instruction (0 or 1), and the state of the poison bits for the different registers after the instruction is executed (0 or 1). State if and when (i.e., at which instruction) the exception incurred by the speculative load is handled and why. The first entry is filled for you.

Instruction	Speculative bit	Poison bits			
		R1	R2	R3	R14
LD R1, 0(R3)	0	0	0	0	0
(s)LD R14,0(R2)	1	0	0	0	1
BEQZ R1,L3	0	0	0	0	1
SD R14, 0(R3)	0	0	0	0	0 or 1 are accepted

Solution:
See above. The exception is handled at the store because it is the first non-speculative instruction that sees the poison bit.

Grading:
1 point for each row. 1 point for identifying where the speculation is handled and why. We accepted a poison bit value of 0 or 1 for R14 for the SD instruction as the execution takes the exception at this point. Some students added the DADDI instruction to the table that is not being executed; however, in this instance, we did not deduct points for this kind of error.

Part B [5 points]

Now assume that A!=0 at the start of the execution of your modified code and redo the following table with all of the other assumptions and instructions of Part A:

Instruction	Speculative bit	Poison bits			
		R1	R2	R3	R14
LD R1, 0(R3)	0	0	0	0	0
(s)LD R14,0(R2)	1	0	0	0	1
BEQZ R1,L3	0	0	0	0	1
DADDI R14, R1, #4	0	0	0	0	0
SD R14, 0(R3)	0	0	0	0	0

Solution: See above. The exception does not need to be handled since the poison bit is overwritten by the DADDI instruction.

Grading: 1 point for each correct row. 1 point for identifying that the exception does not need to be handled and why.