CS 433 Midterm Exam: Mar 12, 2025

Professor Sarita Adve

Time: 2 Hours

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

Name:	Instructor	
NetID:		
Section:	S3 (Undergraduate)	S4 (Graduate)

Instructions

1. **Please write your Name and NetID on the top right corner of all pages in this exam.**
2. No books, papers, notes, or any other typed or written materials are allowed. No calculators or other electronic materials are allowed.
3. 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.
4. *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.*
5. 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.
6. 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!

Problem		**1**	**2**	**3**	**4**	**5**	**6**	**Total**
Points	**S3**	5	15	12	14	4	0	50
	S4	5	15	12	14	4	6	56
Score								

Name:
NetID:

Problem 1 [5 points]

A program contains three non-overlapping regions that respectively constitute the following percentages of the total execution time of the program on a given machine:

- **Region A:** 30% of the program execution time
- **Region B:** 40% of the program execution time
- **Region C:** 30% of the program execution time

Part (A) [3 points]

An enhancement is proposed that **speeds up region A by 3X, region B by 5X, and slows down region C by 2X** (i.e., the execution time for region C doubles). Using the information provided, calculate the speedup for the entire 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:

$$Speedup = \frac{1}{\frac{0.3}{3}+\frac{0.4}{5}+(0.3\times 2)} = \frac{1}{0.1+0.08+0.6} = \frac{1}{0.78}$$

Grading:
3 points for setting up the speedup equation. -0.5 for each fraction/speedup value incorrectly substituted.

Part (B) [2 points]

The enhancement from Part (A) is modified such that it provides infinite speedups for both Region A and Region B and is unchanged for Region C (i.e., slowdown from the original by 2X). Calculate the speedup for the entire program. Again, you are not required to perform numerical calculations.

Solution:

$$Speedup = \frac{1}{\frac{0.3}{\infty}+\frac{0.4}{\infty}+(0.3\times 2)} = \frac{1}{0+0+0.6} = \frac{1}{0.6}$$

Grading:
2 points for setting up the speedup equation. -0.5 for each fraction/speedup value incorrectly substituted.

Name:
NetID:

Problem 2 [15 points]

This problem concerns Tomasulo's algorithm with **dual-issue and hardware speculation**. Consider the following architecture specification:

Functional Unit Type	Cycles in EX	Number of Functional Units
Integer	1	1
FP Adder	3	1
FP Divider	6	1

1. Assume that you have unlimited reservation stations and reorder buffer entries.
2. Functional units are ***not*** pipelined.
3. Two instructions can commit each cycle.
4. Loads use the integer functional 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.
5. 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.
6. An instruction waiting for data on the CDB can move to its EX stage in the cycle after the CDB broadcast.
7. Only one instruction can write to the CDB in one clock cycle. Stores and branches do not need to write to the CDB (i.e., they skip the WB stage).
8. Whenever there is a conflict for a resource like a functional unit or CDB (i.e., multiple instructions are ready to use the resource in the same cycle), assume that the oldest (by program order) of the conflicting instructions gets access, while others are stalled.
9. 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).

Based on the above specification, fill in the cycle numbers in each pipeline stage for each instruction in the following table (CM stands for the commit stage). If a stall occurs at any stage for an instruction, describe the reason for the stall in the last column. The reason should include the type of hazard; the register, functional unit, etc. that caused the dependence; and the instruction number (#) on which the given instruction is dependent. Additionally, note any stalls due to commit ordering.

Name:
NetID:

#	Instruction	IS	EX	WB	CM	Reason for Stalls
1	LD.D F0, 0(R1)	1	2	3	4	
2	ADD.D F0, F0, F3	1	4-6	7	8	RAW on F0 (from #1)
3	DIV.D F8, F0, F6	2	11-16	17	18	RAW on F0 (from #2) FP DIV occupied (by #6)
4	LD.D F6, 8(R1)	2	3	4	18	In-order commit
5	ADD.D F4, F6, F2	3	7-9	10	19	RAW due to F6 (from #4) FP ADD occupied (by #2) In-order commit
6	DIV.D F4, F6, F2	3	5-10	11	19	RAW on F6 (from #4) In-order commit
7	LD.D F6, 16(R1)	4	5	6	20	In-order commit

Grading:
½ point for each entry (each stage for each instruction and reason for instruction are worth ½ point). 30 entries worth ½ point each = 15 points total. Cascading errors will not be penalized additionally as long as the relevant dependencies are still observed.

Name:
NetID:

Problem 3 [12 points]

Consider a loop that is entered several times within a program. The loop contains 4 branches: branch 1, 2, 3, 4, where branch 1 occurs before branch 2 which occurs before branch 3 which occurs before branch 4 in each iteration. Each time the loop is entered in the program, the loop performs 8 iterations, and each iteration executes all four branches with the following outcomes:

	Iteration							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Branch 1	N	T	T	N	N	T	T	N
Branch 2	T	T	T	T	T	N	N	N
Branch 3	T	N	N	T	T	T	T	N
Branch 4	T	T	T	T	T	T	T	N

When Branch 4 is not taken at iteration 8, the program leaves the loop.

Assume any other branches in the program (outside of the loop) do not affect local histories or prediction entries of any of the above branches, and every time the loop is entered (i.e., iteration 1), the global branch history is *all not-taken*. Assume the predictor tables have infinite storage and the loop occurs enough times that the initial state of the predictors at the beginning of the program does not matter.

For each of the three branches below, describe the predictor with the best (lowest) misprediction rate, explain why that predictor works well for the specific branch, and give the state of that predictor at the end of the 1000^{th} invocation of the loop. When giving the state for a history based predictor, indicate which history a given prediction corresponds to. For example, for a (2, 1) history based predictor, the state can be represented as W/X/Y/Z, where each of W, X, Y, and Z is the prediction corresponding to a specific history pattern.

Consider only local and global correlating predictors, saturating counters, and static predictions. History may not be longer than 2 branches, and counters may not be larger than 2 bits. For full credit, you should give the simplest predictor that achieves the same misprediction rate, where counter based predictors are considered simpler than history based and global history is simpler than local history.

Part (A) Branch 1 [4 points]:

Solution:
A local (2,1) predictor will predict every branch correctly. This works well because the decision of branch 1 is highly correlated with its previous results. The predictor state is T/T/N/N, assuming the first entry is for history NN, the second is for history NT, third for history TN, and fourth for history TT.

Part (B) Branch 2 [4 points]:

Solution:
A 1-bit counter makes 2 incorrect predictions per iteration. The branch has long runs of the same decision, and a 1-bit counter costs a minimal number of mispredicts when switching from taken to not taken. The final predictor state at the end of an iteration is 0 (not taken).

Part (C) Branch 3 [4 points]:

Solution:
A global (2,1) predictor will predict every branch correctly. This works well because the decision is highly correlated with the decision of earlier branches in the same iteration. The predictor state is N/T/T/N (with the same correspondence to history bits as above).

Grading:
For each case, 2 points for the correct predictor and justification and 2 points for the correct state.

Name:
NetID:

Problem 4 [14 points]

Consider the following variation of the simple five stage pipeline (IF ID EX MEM WB) discussed in class:

1. The MEM stage takes 4 cycles for each memory instruction, but is pipelined. For simplicity, assume that all other instructions spend one cycle in MEM like we discussed in class.
2. All instructions take one cycle in the EX stage. Address calculation for memory instructions occurs in the EX stage.
3. There is full forwarding and bypassing.
4. For simplicity, ignore any structural hazards in the WB stage. If multiple instructions finish their MEM stages in the same cycle, then assume they can all proceed to the WB stage together (assuming there are no WAW hazards to be resolved).
5. For simplicity, assume that instructions can proceed to the WB stage out of order (assuming other hazards are resolved). This is similar to the multicycle operations extension of the 5 stage pipeline we studied in the lecture relating to Appendix C.
6. Branches are resolved in the decode stage as discussed in class, **and have one branch delay slot.**

Consider the loop below. It calculates the sum of a set of values stored with an indirect representation. Instead of storing the values, 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 register F1.

```
loop:     LD        R3, 0(R1)          // I1
          LD.D      F2, 0(R3)          // I2
          ADD.D     F1, F1, F2         // I3
          SUBI      R2, R2, #1         // I4
          BNEZ      R2, loop           // I5
          ADDI      R1, R1, #4         // I6
```

Name:
NetID:

Part A [3 points]

List all the stalls incurred by the above loop (within a single iteration) 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:

1/2 point for identifying each stall. 1/2 point for the correct number of cycles. 3 points total. -½ point for any other stall wrongly identified.

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.)

Solution:

```
loop:     LD.D      F2, 0(R3)       // i-1 instruction
          LD        R3, 0(R1)       // i instruction
          SUBI      R2, R2, #1
          ADDI      R1, R1, #4      // i instruction
          BNEZ      R2, loop
          ADD.D     F1, F1, F2      // i-1 instruction
```

Grading:

2 points if the solution appears to "spread out" the two loads and the add instruction enough (across two iterations) to eliminate the RAW stalls. -0.5 if stalls are not fully eliminated for each RAW hazard.
1 point for resolving the branch stall by spacing the sub instruction and the branch.
0.5 if the branch delay slot is filled with a suitable instruction.
0.5 bonus point for a perfect solution.

Part C [5 points]

Now consider the original code (without software pipelining) again. Consider unrolling the (original) loop. What is the minimum number of original iterations that you would need to include in the unrolled loop to minimize the stalls in the above code? Show the unrolled code. Assume you have infinite registers.

Solution:

Four iterations of the original loop are needed in each iteration of the new loop.

```
loop:     LD R3, 0(R1)
          LD R4, 4(R1)
          LD R5, 8(R1)
          LD R6, 12(R1)
          SUBI R2, R2, #4
          LD.D F2, 0(R3)
          LD.D F4, 0(R4)
          LD.D F6, 0(R5)
          LD.D F8, 0(R6)
          ADDI R1, R1, #16
          ADD.D F1, F1, F2
          ADD.D F1, F1, F4
          ADD.D F1, F1, F6
          BNEZ R2, loop
          ADD.D F1, F1, F8
```

Grading:

1 point for identifying the correct number of iterations.
2 points for correct unrolling of the code for the identified number of iterations (correct immediate offsets, renamed registers, etc). Partial credit of 1 point if at least half of the unrolled address arithmetic and registers are correct.
2 points for scheduling for minimal stalls for the identified number of iterations, including for the delay slot. Partial credit of 1 point if stalls are reduced but not eliminated.

Part D [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 registers as the reason.

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

Solution:
Loop overhead instructions such as branch and loop variable increment are reduced in the loop unrolling code, so fewer total dynamic instructions to execute.

Grading:
1 point for correct answer.

Name:
NetID:

Problem 5 [4 points]

Consider the following format for predicated (also known as guarded or conditional) 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, eliminate all branches/jumps in the following code fragment (i.e., write the three basic blocks of the following code fragment as a single basic block).

```
     SUB  R1, R13, R14
     BLT  R1, R4, L1          // branch if R1 < R4
     ADDI R2, R2, #1
     ST   R2, 0(R7)
     J    L2

L1:  DIV.D F0, F0, F2
     ADD.D F0, F4, F2
     ST.D  F0, 0(R8)

L2:  ...
```

Name:
NetID:

Space to solve Problem 5

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) ST R2, 0(R7)
      (5) (pT) DIV.D F0, F0, F2
      (6) (pT) ADD.D F0, F4, F2
      (7) (pT) ST.D F0, 0(R8)

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.

Name:
NetID:

NOTE: ONLY GRADUATE STUDENTS SHOULD SOLVE PROBLEM 6.

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 station entries shared among all the Integer Units, and 2 reservation station entries shared among all the FPUs. The Reorder Buffer has 8 entries. The processor has a 25 stage pipeline.

The application you are concerned about has a small code size and works on small data sets that fit in the processor cache. The application spends most of its time in loops where the iterations are independent of each other, but a given iteration has only a limited amount of ILP.

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

1. Improve the branch predictor accuracy.
2. Add more reservation station entries 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 are unlikely to be very 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.

Space to solve Problem 6

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 station entries 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. The initial design had fewer reservation station entries than available functional units. More reservation station entries will allow all functional units to be better utilized. 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. Instead, we should devote extra transistors to first improve other aspects that enable the processor to discover enough parallelism in the instruction stream to keep the available FUs busy.

4. Add more Reorder Buffer entries:
   This is a desirable addition. The current configuration has very few ROB entries relative to the number of FUs available. A large ROB helps to mask out the effects of long latency instructions and help search for parallelism within a larger window to utilize the FUs better (this goes together with (2)).

Grading:
1.5 points for correctly analyzing each part. 6 points total. Give partial credit (1 point) if the student gives a valid reason for an opposing viewpoint.