CS 433: Computer Architecture – Fall 2022
Homework 3

Total Points: Undergraduates (69 points), Graduates (79 points)
Undergraduate students should solve only the first 4 problems.
Graduate students should solve all problems.
Due Date: September 29, 2022 at 10:00 pm CT
(See course information slides for more details)

Directions:

- **All students must write and sign the following statement at the end of their homework submission. "I have read the honor code for this class in the course information handout and have done this homework in conformance with that code. I understand fully the penalty for violating the honor code policies for this class." No credit will be given for a submission that does not contain this signed statement.**
- **On top of the first page of your homework solution, please write your name and NETID, your partner's name and NETID, and whether you are an undergrad or grad student.**
- **Please show all the work that you used to arrive at your answer. Answers without justification will not receive credit. Errors in numerical calculations will not be penalized. Cascading errors will usually be penalized only once.**
- **See course information slides for more details.**

Problem 1 [39 points]

This problem concerns Tomasulo's algorithm. Consider the following architecture specification:

Functional Unit Type	**Cycles in Ex**	**Number of Functional Units**
Integer	1	1
FP Adder	5	1
FP Multiplier	8	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) Branches do not have a delay slot.
(10) Assume that the BNEQZ occupies the integer functional unit for its computation and spends one cycle in EX.

Part A [9 points]

For this part, assume a single-issue machine. Fill in the table below with the cycle number where each instruction occupies the given stage. If a stall occurs, describe the reason for the stall. The reason should include the type of hazard; the register, functional unit, etc. that caused the dependence; and the instruction on which the given instruction is dependent (use the IS stage cycle number to identify the instruction). The first entry has been filled for you.

Instruction		IS	EX	WB	Reasons for stall
L.D	F0, 0(R1)	1	2	3	
ADD.D	F2, F0, F4	2	4–8	9	RAW on F0 (from 1)
MUL.D	F4, F2, F6	3	17–24	25	RAW on F2 (from 2) Structural hazard on MUL (from 7)
ADD.D	F6, F8, F10	4	9–13	14	Structural hazard for adder (from 2)
DADDI	R1, R1, #8	5	6	7	
L.D	F1, 0(R2)	6	7	8	
MUL.D	F1, F1, F8	7	9–16	17	RAW on F1 (from 6)
ADD.D	F6, F3, F5	8	14–18	19	Structural hazard for adder (from 4)
DADDI	R2, R2, #8	9	10	11	

Grading: ¼ point for each entry. 2 points for fully correct answer. Cascading errors will not be penalized additionally as long as the relevant dependencies are still observed.

Part B [2 points]

Would pipelining any of the functional units reduce the total execution time of the above code segment? Explain your answer.

Solution: Yes. The uses of the multiplier have no data dependence so they could issue back-to-back (or as soon as possible).

Grading: 2 points with correct explanation. No points for just a "yes" with no explanation.

Part C [2 points]

Would adding another multiplier functional unit be more advantageous than pipelining the multiplier for the above code segment? Assume that pipelining the multiplier results in 8 stages with 1 cycle per stage. Explain your answer.

Solution: No. With single issue the total throughput would be the same. You could not issue any faster than back-to-back anyway.

Grading: 2 points with correct explanation. No points for just a "no" with no explanation.

Part D [18 points]

For this part, assume ***hardware speculation*** and ***dual-issue*** added to the Tomasulo pipeline you used for part A. That is, assume that an instruction can issue even before the branch has completed (or started) its execution (as with perfect branch and target prediction). However, assume that an instruction after a branch cannot issue in the same cycle as the branch; the earliest it can issue is in the cycle immediately after the branch (to give time to access the branch history table and/or buffer). Any other pair of instructions can issue in the same cycle. Assume that a store calculates its target address in EX and performs its memory access during the Commit stage. Recall that stores and branches (BNEZ) do not write back.

Additionally, assume that your reorder buffer has ***12 entries*** (at the beginning of execution the ROB is empty). Furthermore, ***two instructions can commit each cycle***.

Fill in the cycle numbers in each pipeline stage for each instruction in the first two iterations of the loop represented below, assuming the branch is always taken. If a stall occurs, describe the reason for the stall. The reason should include the type of hazard; the register, functional unit, etc. that caused the dependence; and the instruction on which the given instruction is dependent (use the IS stage cycle number to identify the instruction). Additionally, note any stalls due to commit ordering. The entries for the first two instructions of the first iteration are filled in for you. CM stands for the commit stage.

Instruction	IS	EX	WB	CM	Reason for Stalls
Iteration 1					
LP: L.D F0, 0(R1)	1	2	3	4	
ADD.D F0, F0, F6	1	4–8	9	10	RAW on F0 (from 1.1)
DIV.D F2, F2, F0	2	20–34	35	36	RAW on F0 (from 1.2) FP DIV unit occupied (from 3.1)
L.D F0, 8(R1)	2	3	4	36	In-order commit
DIV.D F4, F0, F8	3	5–19	20	37	RAW on F0 (from 2.2) In-order commit
S.D F4, 16(R1)	3	4	—	37	In-order commit
DADDI R1, R1, #-24	4	5	6	38	In-order commit
BNEZ R1, LP	4	7	—	38	RAW on R1 (from 4.1) In-order commit
Iteration 2					
LP: L.D F0, 0(R1)	5	8	10	39	RAW on R1 (from 4.1) INT unit occupied (from 4.2) CDB occupied (from 1.2) In-order commit
ADD.D F0, F0, F6	5	11–15	16	39	RAW on F0 (from 5.1) In-order commit
DIV.D F2, F2, F0	6	50–64	65	66	RAW on F2 (from 2.1) FP DIV unit occupied (from 7.1)
L.D F0, 8(R1)	6	10	11	66	INT unit occupied (from 5.1) CDB occupied (from 5.1) In-order commit
DIV.D F4, F0, F8	7	35–49	50	67	FP DIV unit occupied (from 2.1) In-order commit
S.D F4, 16(R1)	11	12	—	67	ROB full (from 1.2) In-order commit
DADDI R1, R1, #-24	37	38	39	68	ROB full (from 2.1 and 2.2) In-order commit
BNEZ R1, LP	37	40	—	68	RAW on R1 (from 37.1) In-order commit

Grading: ¼ point for each entry. 0.5 points for a fully correct answer. Cascading errors will not be penalized additionally as long as the relevant dependencies are still observed.

Part E [6 Points]

For the code in Part D, which of the following optimizations will improve the total execution time, in terms of number of cycles: triple issue, three instructions committed per cycle, or increasing the reorder buffer size to 14 entries? Explain why (consider each of these as independent optimizations).

Solution: Triple issue causes the fifth instruction to be fetched in the second cycle, but EX starts in the fifth cycle again. Since DIV causes the same bottleneck as before, there will not be any overall improvement.

Triple commit allows the second loop to finish by the end of the 67th cycle, reducing the overall runtime by 1 cycle. (It also allows the first loop to end in 37 cycles.)

Increased reorder buffer again causes more instructions to be fetched, but they cannot commit until DIV instructions complete and commit. Hence there will not be any improvement.

Grading: 2 points each for observing DIV does not let performance improve for both triple issue and increased ROB size. 2 points for observing 1 cycle improvement per iteration for triple commit.

Part F [2 Points]

For the code in Part D, assume a floating point divide by 0 incurs an exception. In which clock cycle will the system invoke a jump to the interrupt service routine if F8 used in the fifth instruction has the value 0? Assume that the exception is identified as soon as EX begins and the instruction with the exception gives up the execution unit in the same cycle (i.e., it is available for another instruction in the next cycle). Explain your answer.

Solution: Interrupt identified at 5th cycle. DIV of instruction 3 runs from the 10th until the 24th cycle. DIV and L.D commit at the 26th cycle. The system invokes a jump to the ISR in cycle 27.

Grading: 2 points for the answer and justification for 27th cycle. No points without explanation.

Problem 2: Dynamic Branch Prediction [14 points]

Consider the following MIPS code. The register R0 is always 0.

```
        DADDI   R1, R0, R0

L1:     DADDI   R2, R0, R0

L2:     DADDI   R2, R2, #1
        DSUBI   R3, R2, #3
        BNEQZ   R3, L2              -- Branch 1

        DADDI   R1, R1, #1
        DSUBI   R4, R1, #4
        BNEQZ   R4, L1              -- Branch 2
```

Each table below refers to only one branch. For instance, branch 1 will be executed 12 times. Those 12 times should be recorded in the table for branch 1. Similarly, branch 2 is executed 4 times.

Part A [4 points]

Assume that 1-bit branch predictors are used. When the processor starts to execute the above code, both predictors contain value N (Not taken). What is the number of correct predictions? Use the following tables to record the prediction and action of each branch. Several entries are filled in for you.

Branch 1:

Step	Branch 1 Prediction	Actual Branch 1 Action
1	N	T
2	T	T
3	T	N
4	N	T
5	T	T
6	T	N
7	N	T
8	T	T
9	T	N
10	N	T
11	T	T
12	T	N

Branch 2:

Step	Branch 2 Prediction	Actual Branch 2 Action
1	N	T
2	T	T
3	T	T
4	T	N

Solution: Total of 6 correct predictions, 4 for Branch 1 and 2 for Branch 2.

Grading: -1/2 for not stating the correct number of predictions. -1/2 for each mistake in the table, cascading errors will not be penalized. Minimum possible score is 0.

Part B [4 Points]

Now assume that 2-bit saturation counters are used. When the processor starts to execute the above code, both counters contain value 00. What is the number of correct predictions? Use the following tables to record the prediction and action of each branch. You have to follow the 2-bit saturation counters taught in class for branch prediction.

Branch 1:

Step	Counter Value	Branch 1 Prediction	Actual Branch 1 Action
1	0 0	N	T
2	0 1	N	T
3	1 0	T	N
4	0 1	N	T
5	1 0	T	T
6	1 1	T	N
7	1 0	T	T
8	1 1	T	T
9	1 1	T	N
10	1 0	T	T
11	1 1	T	T
12	1 1	T	N

Branch 2:

Step	Counter Value	Branch 2 Prediction	Actual Branch 2 Action
1	0 0	N	T
2	0 1	N	T
3	1 0	T	T
4	1 1	T	N

Solution: 6 total correct predictions, 5 for Branch 1 and 1 for Branch 2.

Grading: -1/2 for not stating the correct number of predictions. -1/2 for each mistake in the table, cascading errors will not be penalized. Minimum possible score is 0.

Part C [6 points]

Now assume that 2 level *global* correlating predictors of the form (2, 1) are used. (Note that global here means that the history used captures the history of all previous branches. It does not mean that there is only one set of prediction bits for all branches.) When the processor starts to execute the above code, the outcome of the previous two branches is not taken (N). Also assume that the initial state of predictors of all branches is not taken (N). What is the number of correct predictions? Use the following table to record your steps. Record the "New State" of predictors in the form W/X/Y/Z where,

W - state corresponds to the case where the last branch and the branch before the last are both TAKEN.

X - state corresponds to the case where the last branch is TAKEN and the branch before the last is NOT TAKEN.

Y - state corresponds to the case where the last branch is NOT TAKEN and the branch before the last is TAKEN.

Z - state corresponds to the case where the last branch and the branch before the last are both NOT TAKEN.

Note: The state of the predictor at position W/X/Y/Z can be *N* for predict-not-taken or *T* for predict-taken.

Branch 1:

Step	Branch 1 Prediction	Actual Branch 1 Action	New State
1	N	T	N/N/N/T
2	N	T	N/T/N/T
3	N	N	N/T/N/T
4	T	T	N/T/N/T
5	N	T	T/T/N/T
6	T	N	N/T/N/T
7	T	T	N/T/N/T
8	N	T	T/T/N/T
9	T	N	N/T/N/T
10	T	T	N/T/N/T
11	N	T	T/T/N/T
12	T	N	N/T/N/T

Branch 2:

Step	Branch 2 Prediction	Actual Branch 2 Action	New State
1	N	T	N/N/T/N
2	T	T	N/N/T/N
3	T	T	N/N/T/N
4	T	N	N/N/N/N

Solution: 6 total correct predictions, 4 for Branch 1 and 2 for Branch 2.

Grading: -1/2 for not stating the correct number of predictions. -1/2 for each mistake in the table, cascading errors will not be penalized. Minimum possible score is 0.

Problem 3 [8 Points]

Part A [6 points]

Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for conditional branches only. Assume that the misprediction penalty is always four cycles and buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume that the base CPI without branch stalls is one cycle.

Solution: For this problem we are given the base CPI without branch stalls. From this we can compute the number of stalls given by no BTB and with the BTB:

CPI_1 is CPI with no BTB, CPI_2 is CPI with BTB and CPI represents the CPI base.

$$Speedup = \frac{CPI_1}{CPI_2} = \frac{(CPI + Stalls_1)}{(CPI + Stalls_2)}$$

$$Stalls_1 = branch\, frequency \times branch\, penalty = 0.15 \times 2 = 0.30$$

$Stalls_2$ is calculated from the following table.

BTB Result	BTB Prediction	Penalty (cycles)	CPI Penalty
Miss	N/A	3	$15\% \times 10\% \times 3 = 0.045$
Hit	Correct	0	$15\% \times 90\% \times 90\% \times 0 = 0$
Hit	Incorrect	4	$15\% \times 90\% \times 10\% \times 4 = 0.0$

$$Stalls_2 = 0.045 + 0 + 0.054 = 0.099$$

$$Speedup = \frac{(1+0.30)}{(1+0.099)} = 1.183$$

Grading: 1 point for calculating CPI with no branch prediction. 1 point for correct penalty for each of the three cases: BTB miss, BTB hit (correct prediction) and BTB hit (incorrect prediction). 2 points for correct use of the speedup formula.

Part B [2 points]

Now consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch. What is the penalty or benefit in clock cycles when an unconditional branch is found in the buffer? Explain.

Solution: Storing the target instruction of an unconditional branch effectively removes one instruction. If there is a BTB hit in instruction fetch and the target instruction is available, then that instruction is fed into decode in place of the branch instruction. The penalty is –1 cycle. In other words, it is a performance gain of 1 cycle.

Grading: 2 points for correct penalty/gain with correct reason.

Problem 4 [8 points]

This problem concerns the implications of the reorder buffer size on performance. Consider a processor implementing Tomasulo's algorithm with reservation stations and the reorder buffer scheme described in detail in the lecture notes. Assume infinite processor resources unless stated otherwise; e.g., infinite execution units and infinite reservation stations. Assume a perfect branch predictor and assume there are no data dependences in the instruction stream we are considering. Assume the maximum instruction fetch rate is 12 instructions per cycle. (The other stages in the pipeline have no constraints; e.g., the processor can decode an unbounded number of instructions per cycle.)

Part A [2 points]

Suppose all instructions take one cycle to execute and the processor has an infinite reorder buffer. What is the average instructions-per-cycle rate or IPC for this processor?

Solution: The average IPC would be 12, since it is limited only by the fetch rate. There is no reason for any stall since there are no data dependencies or branch mispredicts and we have infinite resources.

Grading: 2 points for saying IPC is 12 and why it is so.

Part B [2 points]

Consider the system in part (a) except that now every 48th instruction is a load that misses in the cache such that completing the load takes 500 cycles. What is the average instructions-per-cycle or IPC for this processor?

Solution: The average IPC would again be 12. The ROB would mask out the latencies associated with missing load instructions, and would allow us to keep fetching and issuing 12 instructions each cycle. The misses would introduce a lag of up to 500 cycles between the fetch and commit, but the average throughput would still be 12 instructions each cycle.

Grading: 2 points for saying IPC is 12 and why it is so.

Part C [4 points]

Consider the system in part (b) except that now the reorder buffer size is 48 entries. What is the average IPC for this processor? If the IPC is less than 12, then what is the smallest reorder buffer size for which the IPC will be 12 again (assume the reorder buffer size can only be a multiple of 12).

Solution: We can no longer get an IPC of 12, since the limited ROB size will cause stalls. Suppose that at cycle 1, we issue a load that misses. We would keep fetching and issuing instructions until the ROB becomes full, so we would issue 47 more instructions and then stall, until cycle 500, when the instruction at the head of the ROB completes and is ready to commit (along with the other 47 instructions). At that point, we would issue the next missing load, and this cycle would repeat. Thus, every 500 cycles, we are able to commit 48 instructions, and the IPC is 48/500.

To obtain an average IPC of 12, we need to be able to overlap the execution of 12 instructions per cycle on average during the 500 cycles that the load is stalled. These instructions cannot commit until the load commits. This requires an ROB size of 12*500=6000 instructions.

Grading: 2 points for realizing that since commit is in order, a long load miss at the head of the ROB implies that fetch and retire will stall once the ROB fills up, until the load miss completes. 1 point for getting that 48 instructions complete in 500 cycles for an IPC of 48/500. 1 point for getting that for an average IPC of 12, we need to have 12 instructions per cycle times 500 cycles = 6000 instructions in the ROB.

NOTE: ONLY GRADUATE STUDENTS SHOULD SOLVE THE NEXT PROBLEM.

Problem 5 - GRADUATE STUDENTS PROBLEM [10 points]

In class, we studied the use of a reorder buffer to maintain precise interrupts. With this mechanism, an instruction does not update the register file with its newly computed value until it is committed. Instead, the new value is stored in the reorder buffer. This requires later instructions to possibly read source operand values from the reorder buffer.

An alternative is to use a history buffer. This mechanism updates the register file as soon as the instruction computes a new value, but it stores the previous value of the register in the history buffer. On an interrupt, appropriate old values are restored.

Consider using the above history buffer idea to maintain precise interrupts with the standard Tomasulo algorithm design (with reservation stations) as covered in class (no reorder buffer). Explain the modifications to the Tomasulo pipeline for this purpose.

Hint: As with the reorder buffer, we need to split the Write stage into Complete and Commit.

Separate your answer into the following parts. You need only give the conceptual changes from the basic Tomasulo algorithm (e.g., at the level discussed in the lecture notes for the reorder buffer).

Part A [2 points] Explain how the fields of the history buffer would be different from the reorder buffer.

Solution: History buffer is similar to the reorder buffer, but it will have the old value instead of the new value of the destination register.

Grading: 2 points for this change. Incorrect changes will result in negative ½ point for each change.

Part B [1 point] Describe the changes to the Issue stage.

Solution: Allocate history buffer entry (HB). So in the issue stage, as before, we read the old value of the destination if available or point to the reservation station (RS) that will get the last value (as indicated by the result status register).

Grading: 1 point for this change. Incorrect changes will result in negative ½ point for each change.

Part C [1 point] Describe the changes to the Execute stage.

Solution: No changes.

Grading: 1 point for stating there are no changes. Any incorrect change will result in -½ for each change.

Part D [2 points] Describe the changes to the Complete stage.

Solution: This stage does everything in the write stage of the basic pipeline: write result to the CDB; all RS and registers waiting on this tag copy the CDB value. Additionally, any old value field in HB waiting on this tag also copies the value.

Grading: 2 points for the last sentence, negative ½ for any incorrect change.

Part E [2.5 points] Describe the changes to the Commit stage (for all instructions, including for stores and branches).

Solution: When the instruction reaches the head of the history buffer, it leaves the buffer and bumps the head.

For a store, no change.

For a branch: if mispredict, go through all the entries in HB from the youngest one first. For each entry, update the destination register with the old value field and flush the entry.

Grading: 1 point for describing normal. 1.5 points for describing branch mispredict. Negative ½ for any incorrect change.

Part F [1.5 points] How is an interrupt handled?

Solution: Same as for the branch mispredict case above.

Grading: 1.5 points for correct answer.