CS433: Computer Architecture – Fall 2019
Homework 4
Total Points: Undergraduates (32 points), Graduates (50 points)
Undergraduate students should only do Problem 1. Graduate students should solve all problems.
Due Date: October 15, 2019 at 11:00 am (See course information handout for more details)

Problem 1 [32 points]:

Consider the following architecture.

Functional Unit Type	Cycles in EX	Number of Functional Units	Pipelined
Integer	1	1	No
FP Add/Subtract	3	1	Yes
FP/Integer Multiplier	6	1	Yes
FP/Integer Divider	24	1	No

- In this problem we will use the 5-stage MIPS pipeline.
- The integer functional unit performs integer addition (including effective address calculation for loads/stores), subtraction, logic operations and branch operations.
- There is full forwarding and bypassing, including forwarding from the end of an FU 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.
- 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 in the MEM and WB stages.

The following code implements the DAXPY operation, $Y = aX + Y$, for a vector length 100. Initially, R1 is set to the base address of array X and R2 is set to the base address of Y. Assume

initial value of R3 = 0. The *DADDUI* instruction before the loop is initialization code and should not be included in the answer to any of the questions.

```
        DADDIU R4, R1, #800
FOO:    L.D F2, 0(R1)
        MUL.D F4, F2, F0
        L.D F6, 0(R2)
        ADD.D F6, F4, F6
        S.D F6, 0(R2)
        DADDIU R1, R1, #8
        DADDIU R2, R2, #8
        DSLTU R3, R1, R4 // set R3 to 1 if R1 < R4
        BNEZ R3, FOO
```

Part A [6 points]

Consider the role of the compiler in scheduling the code. Rewrite this loop, but let every row take a cycle (each row can be an instruction or a stall). If an instruction can't be issued in 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 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 all stalls, but don't reorder instructions. How many cycles elapse before the second iteration begins? Show your work.

Solution:

```
FOO:    L.D F2, 0(R1)
        (1) STALL RAW F2
        MUL.D F4, F2, F0
        L.D F6, 0(R2)
        (2) STALL RAW F4, F6
        (3) STALL RAW F4
        (4) STALL RAW F4
        (5) STALL RAW F4
        ADD F6, F4, F6
        (6) STALL RAW F6
        S.D. F6, 0(R2)
        DADDUI R1,R1, #8
```

DADDUI R2, R2, #8
DSLTU R3, R1, R4
(7) STALL RAW R3
BENZ R3, FOO
NOP

17 cycles.

Grading: 1 point for each of stalls (1), (6), and (7). No credit without any explanation for the stall. Partial credit of ½ point if more than one stall cycle is indicated for the corresponding instruction.

2 points total for stalls (2) to (5). Partial credit is awarded as follows, assuming there is at least one correct explanation for the stall (e.g., at least one of the F4 or F6 dependence is listed for stall 2): 1 point if a stall is listed between these instructions, but the number of stalls is incorrect; ½ point if at least some of the reasons for the stalls are correct.

1 point for NOP (branch delay slot).
Negative ½ point for each unnecessary sequence of stalls.

Part B [6 points]

Now reschedule the loop. You can change immediate values and memory offsets. You can 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.

Solution:

FOO: L.D F2, 0(R1)
L.D F6, 0(R2)
MUL.D F4, F2, F0
DADDUI R1, R1, #8
DADDUI R2, R2, #8
DSLTU R3, R1, R4
STALL
STALL
ADD F6, F4, F6
BNEZ R3, FOO
S.D F6, -8(R2)

11 cycles

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), and deduct ½ point for each stall in excess of 2.

Part C [6 points]

Now unroll and reschedule the loop the minimum number of times needed to eliminate all stalls. You can remove redundant instructions. How many times did you unroll the loop? How many cycles elapse before the next iteration of the loop begins? Don't worry about clean-up code. Show your work.

Solution:

```
FOO:  L.D F2, 0(R1)
      L.D F6, 0(R2)
      MUL.D F4, F2, F0
      L.D F14, 8(R1)
      L.D F16, 8(R2)
      MUL.D F18, F14, F0
      DADDUI R1, R1, #16
      DADDUI R2, R2, #16
      ADD F6, F4, F6
      DSLTU R3, R1, R4
      S.D F6, -16(R2)
      ADD F16, F18, F16
      BNEZ R3, FOO
      S.D F16, -8(R2)
```

There are two original iterations in an iteration of the new loop. 14 cycles elapse before the next iteration.

Grading: 1 point for the correct iteration count. Deduct ½ point for every error or stall cycle. Give partial credit (2 points) if use three iterations 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 loop four times, and schedule it for this VLIW to take as few stall cycles as possible. How many cycles do the four iterations take to complete? Use the following table template to show your work.

Solution:

MEM 1	MEM 2	INTEGER	FP ADD	FP MUL
L.D F2, 0(R1)	L.D F6, 0(R2)			

L.D F14, 8(R1)	L.D F16, 8(R2)			
L.D F24, 16(R1)	L.D F26, 16(R2)			MUL.D F4, F2, F0
L.D F34, 24(R1)	L.D F36, 24(R2)			MUL.D F18, F14, F0
				MUL.D F28, F24, F0
				MUL.D F38, F34, F0
		DADDUI R1, R1, #32		
		DADDUI R2, R2, #32		
		DSLTU R3, R1, R4	ADD F6, F4, F6	
			ADD F16, F18, F16	
S.D F6, -32(R2)			ADD F26, F28, F26	
S.D F16, -24(R2)			ADD F36, F38, F36	
S.D. F26, -16(R2)				
S.D F36, -8(R2)		BENZ R3, FOO		

14 cycles with unrolling 4 times.

Grading scheme: 0.5 point for each correct row in the above table.1 point if the scheduled code is correct and takes14 cycles.

Part E: Software Pipelining (6 points)

Provide the steady-state code for a software pipelined version of the loop given in this question. Your code should give the minimum number of stalls using the minimum number of static instructions. Assume the loop will have at least four iterations. You do not have to show the start-up or finish-up code (i.e., prolog or epilog).

Solution:

(1) S.D F6, -24(R2) // x-3 instruction
(2) ADD.D F6, F4, F7 // x-2 instruction
(3a) L.D F7, -8(R2) // x-1 instruction
(3b) MUL.D F4, F2, F0 // x-1 instruction
(4) DADDIU R1, R1, #8
(5) DSLTU R3, R1, R4 // set R3 to 1 if R1 < R4
(6) L.D F2, -8(R1) // x instruction
(7) BNEZ R3, FOO
(8) DADDIU R2, R2, #8

Grading scheme:

3 points for correct offsets of instruction (1), (3a) and (6).

0.5 point for (1), 0.5 for (2), 0.25 for (3a), 0.25 for (3b), 0.5 for (6).

1 point if instructions at number (4), (5), (6), (7) and (8) are given in an order with no stall.

Alternate solution 1:

(1) ADD.D F6, F4, F6 // x-1 instruction
(2) L.D F2, 0(R2) // x instruction
(3) S.D F6, -8(R2) // x-1 instruction
(4) MUL.D F4, F2, F0 // x instruction
(5) DADDIU R1, R1, #8
(6) DSLTU R3, R1, R4 // set R3 to 1 if R1 < R4
(7) L.D F6, -8(R1) // x instruction
(8) BNEZ R3, FOO
(9) DADDIU R2, R2, #8

Grading scheme:

3 points for correct offsets of instruction (2), (3) and (7).

1.5 for x instructions, 0,5 for x-1 instructions.

1 point if instructions at number (5), (6), (7), (8) and (9) are given in an order with no stall.

Alternate solution 2:

(1) S.D F6, 0(R2) // x instruction
(2) ADD.D F6, F4, F7 // x+1 instruction
(3) MUL.D F4, F2, F0 // x+2 instruction
(4) L.D F2, 24(R1) // x+3 instruction
(5) DADDIU R1, R1, #8
(6) DSLTU R3, R1, R4 // set R3 to 1 if R1 < R4 – see note below for loop bound
(7) L.D F7, 16(R2) // x+2 instruction
(8) BNEZ R3, FOO
(9) DADDIU R2, R2, #8

Note: Since this solution performs loads for future iterations, the loop bound (R4) needs to be reduced by 3 to avoid erroneous and out-of-bounds accesses.

Grading scheme:

3 points for correct offsets of instruction (1), (4) and (7).

1.5 for x+2 instructions, 0.5 for x+1 and x+3 instructions.

1 point if instructions at number (5), (6), (7), (8) and (9) are given in an order with no stall.

NOTE: ONLY GRADUATE STUDENTS SHOULD SOLVE THE NEXT TWO PROBLEMS.

Problem 2 [10 points]

Consider the following C code fragment:

```
for (i = 0; i < 100; i++) {
        if (c == 0) {
                …
                c = …;
                … // code I
        }
        else {
                …
                c = …;
                … // code II
        }
        … // code III
}
```

The above translates to the MIPS fragment below. R5 and R6 store variables i and c, respectively.

```
Init:   MOV.D  R5, R0      // i = 0
If:     BNEZ    R6, Else   // Branch1 (c == 0?)
        ...                // Code I = 10 instructions; contains a write to R6
        J Join
Else:
        ...                // Code II = 100 instructions; contains a write to R6

Join:   ...                // Code III = 10 instructions

Loop:   DADDI  R5, R5, #1     // i++
        DSUBI   R7, R5, #100
        BNEZ    R7, If        // Branch2 (i == 100?)
        J Done
```

Suppose the segments “Code I” (if part), “Code II” (else part), and Code III (common part) contain 10, 100, and 10 assembly instructions respectively. You did a profile run of this program and found that on average, Branch1 is taken once in 100 iterations of the “for loop”.

Your boss suggests that you perform one of the following two transformations to speed up the above code: (1) Loop unrolling with an unrolling factor of 2. (2) Trace scheduling.

Which one of these would be more effective and why? Show the code with the more effective transformation applied. If you use trace scheduling, then include any repair code and branches into and out of it. Assume that only the values of c and i may need repair. Assume that registers R10 and higher are free for your use.

Solution:

You should perform trace scheduling. Loop unrolling would increase the code size significantly because of the huge else statement. Further, because of the if-else statement, loop unrolling will not provide any additional longer straight-line code snippet for the compiler to schedule. On the other hand, trace scheduling will be able to combine the "if" and "join" parts of the code together to provide a longer fragment of straight-line code. The large else part is moved out in repair code.

```
Init:   MOV.D R5, R0              // i = 0
        Trace: MOV.D R10, R6      // Save the old value of c
        ....                      // Code I
        ....                      // Code III
        BNEZ R10, Repair

Loop:   DADDI R5, R5, #1    // i++
        DSUBI R7, R5, #100
        BNEZ R7, Trace      // i==100?
        J Done

Repair: MOV.D R6, R10
        ....      // Code II
        ....      // Code III
J Loop
```

The above trace can be further increased by duplicating the loop index manipulation in the trace and repair parts.

Grading:

2 points for choosing/rejecting each option.

3 points for the trace (1 point for saving the old value of c, 1 point for combining code I and code III in the trace, 1 point for the correct branch to repair code).

3 points for the repair code (1 point for restoring c, 1 point for combining code II and III, 1 point for the jump back to the trace).

2 points for a fully correct answer.

If you answered Loop Unrolling, then you will be graded out of 4 for the unrolled code. 1 point for the unrolled loop body, 1 point for correct register usage, 1 point for correct branch index manipulation, and 1 point for a fully correct unrolled loop.

Problem 3 [8 points]

The example on page H-30 of the textbook uses a speculative load instruction to move a load above its guarding branch instruction. Read appendix H in the text for this problem and apply the concepts to the following code:

```
        instr.1             ; arbitrary instruction
        instr.2             ; next instruction in block
        . . .               ; intervening instructions
        BEQZ R1, null       ; check for null pointer
        L.D F2, 0(R1)       ; load using pointer
        ADD.D F4, F0, F2    ; dependent ADD.D
        . . .
        . . .
null:   . . .               ; handle null pointer
```

Part A [4 points]

Write the above code using a speculative load (sL.D) and a speculation check instruction (SPECCK) to preserve exception behavior. Where should the load instruction move to best hide its potentially long latency?

Solution:

The speculative load instruction defers the hardware response to a memory access fault if one occurs. In combination with the speculation check instruction this allows the load to be moved above the branch. Because the load may have long latency, it should be moved as early in the program as possible, in this case to the position of first instruction in the basic block. If the speculation check finds no deferred exceptions, computation can proceed.

```
        sL.D F2, 0(R1)
        instr. 1
```

```
        instr. 2
        ...
        BEQZ R1, null
        SPECCK 0(R1) ; check for exception deferred by sL.D on 0(R1)
        ADD.D F4, F0, F2
        ...
null:   ...
```

Grading:

3 points for explanation.

1 point for correct code.

Part B [4 points]

Assume a speculation check instruction that branches to the recovery code. Assume that the speculative load instruction defers both terminating and non-terminating exceptions. Write the above code speculating on both the load and the dependent add. Use a speculative load, a non-speculative add, a check instruction, and the block of recovery code. How should the speculated load and the add be scheduled with respect to each other?

Solution:

Potentially, this problem will have several different solutions. Only one is provided here.

With a speculation check instruction that can branch to the recovery code, instructions dependent on the load can also be speculated. Now, if the load fails because of an exception for high latency (e.g., page fault), rather than one that is a fatal error (e.g., a memory protection access violation), the speculated use instruction can take as an operand an incorrect value from the register that is the target of the delayed load. The speculation check instruction can distinguish these types of exceptions, terminating the program in the event of a protection violation and branching to recovery code for the case of a page fault, which will yield correct load behavior given sufficient time.

```
        sL.D F2, 0(R1)
        instr. 1
        instr. 2
        ...
        ADD.D F4, F0, F2 ; ADD.D speculated far from load for latency
        BEQZ R1, null
```

```
        SPECCK 0(R1), recover ; check for exception deferred by sL.D on 0(R1) and branch to
                              "recover" on exception

back:   ...
        ... ; etc.

recover: L.D F2, 0(R1)
        ADD.D F4, F0, F2
        JUMP back ; return to original path
        ...

null:   ...
```

Note: Although repair code would be needed in the "null" section of the code for correct behavior (the values of F2 and F4 need to be restored), the question explicitly does not ask for it.

Grading:

3 points for explanation.

1 point for correct code that follows from explanation.