CS433: Computer Architecture – Fall 2020
Homework 4
Total Points: Undergraduates (32 points), Graduates (50 points)
Undergraduate students: only Problem 1. Graduate students: all problems.
<u>This homework is only for practice. Please do not submit solutions. They will not be graded.</u>

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.

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.

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.

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.

MEM 1	MEM 2	INTEGER	FP ADD	FP MUL

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).

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.

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?

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?