Department of informatics
Computer Science degree
Computer Architecture

ILP Exercises II

3.1 What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.48 if no new instruction execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle-branch delay slot.

```
Loop:  LD   F2,0 (RX)
I0:    DIVD F8,F2,F0
I1:    MULTD F2,F6,F2
I2:    LD   F4,0(Ry)
I3:    ADDD F4,F0,F4
I4:    ADDD F10,F8,F2
I5:    ADD1 Rx,Rx,#8
I6:    ADDI Ry,Ry,#8
I7:    SD   F4,0(Ry)
I8:    SUB  R20,R4,Rx
I9:    BNZ  R20, Loop
```

Latencies beyond single cycle	
Memory LD	**+4**
Memory SD	+**1**
Integer ADD, SUB	+0
Branches	+1
ADDD	+1
MULTD	**+5**
DIVD	+12

Figure 3.48 Code and latencies for Exercises 3.1 through 3.6.

3.2 Think about what latency numbers really mean-they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 3.48 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with **<stall** > inserted where necessary to accommodate stated latencies. *(Hint:* An instruction with latency **t2** requires two <**stall** > cycles to be inserted into the code sequence. Think of it this way: A one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency 1 + N has N extra stall cycles.

3.3 Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

3.4 In the multiple-issue design of Exercise 3.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are neither identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction N + I begins execution in Execution Pipe 1 at the same time that instruction N begins in Pipe 0, and N + 1 happens to require a shorter execution latency than N, then N + 1 will complete before N (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 3.48 that demonstrate this hazard.

3.5 Reorder the instructions to improve performance of the code in Figure 3.48. Assume the two-pipe machine in Exercise 3.3 and that the out-of-order completion issues of Exercise 3.4 have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?

3.6 Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not living up to its potential.

a) In your reordered code from Exercise 3.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?
b) Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 3.5.
c) What speedup did you obtain? (For this exercise, just color the N + 1 iteration's instructions green to distinguish them from the Nth iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.).