7 Memory Consistency [65 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

	Thread T0		**Thread T1**
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`	Instr. T1.1	`*flag = 1;`
Instr. T0.2	`while(*flag == 0);`	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

`X`, `Y`, and `flag` have been allocated in main memory, while `a` and `b` are contained in processor registers. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a *single* instruction.

(a) [15 points] What is the final value of `Y[0]` in the SC processor, after both threads finish execution? Explain your answer.

2.

Explanation. `Y[0]` is set equal to 1 by instruction T1.0. Then, it will be incremented by instruction T0.3. The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag`. Thread 0 will remain in instruction T0.2 until `flag` is set by T1.1, i.e., after `Y[0]` is initialized. So, instruction T0.3 must be executed after instruction T1.0, causing `Y[0]` to be first set to 1 and then incremented.

(b) [15 points] What is the final value of `b` in the SC processor, after both threads finish execution? Explain your answer.

0 or 1.

Explanation. There are *at least* two possible sequentially-consistent orderings that lead to *at most* two different values of `b` at the end:
Ordering 1: T1.0 → T0.1 - Final value = 1.
Ordering 2: T0.1 → T1.0 - Final value = 0.

With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called RC, that implements *weak consistency.* As discussed in the lecture, the weak consistency model has no need to guarantee a strict order of memory operations. For this question, consider a very weak model where there is *no* guarantee on the ordering of instructions as seen by different cores.

(c) [15 points] What is the final value of `Y[0]` in the RC processor, after both threads finish execution? Explain your answer.

1 or 2.

Explanation. Since there is no guarantee of a strict order of memory operations, as seen by different cores, instruction T1.1 could complete before or after instruction T1.0, from the perspective of the core that executes T0. If instruction T1.1 completes before instruction T1.0, from the perspective of the core that executes T0, instruction T0.3 could complete before or after instruction T1.0. Thus, there are three possible weakly-consistent orderings that lead to different values of `Y[0]` at the end:
Ordering 1 (from the perspective of T0): T1.0 → T1.1 → T0.3 - Final value = 2.
Ordering 2 (from the perspective of T0): T1.1 → T1.0 → T0.3 - Final value = 2.
Ordering 3 (from the perspective of T0): T1.1 → T0.3 → T1.0 - Final value = 1.

After several months spent debugging her code, the programmer learns that the new processor includes a memory_fence() instruction in its ISA. The semantics of memory_fence() is as follows for a given thread that executes it:

1. Wait (stall the processor) until *all* preceding memory operations from the thread complete in the memory system and become visible to other cores.
2. Ensure *no* memory operation from any later instruction in the thread gets executed before the memory_fence() is retired.

(d) [20 points] What *minimal* changes should the programmer make to the program above to ensure that the final value of Y[0] on RC is the same as that in part (a) on SC? Explain your answer.

Use memory fences before T1.1 and after T0.2.

Explanation. The memory fence before instruction T1.1 stalls thread 1 until instruction T1.0 has completed, i.e., ensures that Y[0] is initialized to 1 before the flag is set. Thread 0 waits in the loop T0.2 until the flag is set. The memory fence after instruction T0.2 ensures that instruction T0.3 will not happen until the memory fence is retired. Thus, instruction T0.3 will also complete *after* the flag is set. The modified code will be as follows:

	Thread T0		**Thread T1**
Instr. T0.0	a = X[0];	Instr. T1.0	Y[0] = 1;
Instr. T0.1	b = a + Y[0];		**memory_fence();**
Instr. T0.2	while(*flag == 0);	Instr. T1.1	*flag = 1;
	memory_fence();	Instr. T1.2	X[1] *= 2;
Instr. T0.3	Y[0] += 1;	Instr. T1.3	a = 0;