CS252 Graduate Computer Architecture
Memory Models Practice Solutions

October 27, 2007

Sequential Consistency, Synchronization, and Relaxed Memory Models

Cy D. Fect wants to run the following code sequences on processors P1 and P2, which are part of a two-processor MIPS64 machine. The sequences operate on memory values located at addresses A, B, C, D, E, F, and G, which are all sequentially located in memory (e.g. B is 8 bytes after A). Initially, M[A], M[B], M[C], M[D], and M[E] are all 0. M[F] is 1, and M[G] is 2. For each processor, R1 contains the address A (all of the addresses are located in a shared region of memory). Also, remember that for a MIPS processor, R0 is hardwired to 0. In the below sequences, a semicolon is used to indicate the start of a comment.

P1

```
ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2
LD   R3,40(R1) ;R3=F
SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B
SD   R4,24(R1) ;D=R4
```

P2

```
ADDI R2,R0,#1  ;R2=1
SD   R2,8(R1)  ;B=R2
LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3
LD   R4,0(R1)  ;R4=A
SD   R4,32(R1) ;E=R4
```

Problem 1.A **Sequential Consistency**

If Cy's code runs on a system with a sequentially consistent memory model, what are the possible results of execution? List all possible results in terms of values of `M[C]`, `M[D]`, and `M[E]` (since the values in the other locations will be the same across all possible execution paths).

We know that a given set of results can only be sequentially consistent if it is possible to produce that set by having the operations of each individual processor occur in program order, and interleaving the operations of all the processors in some fashion. So, we can determine all possible sets of sequentially consistent results for the above program by experimenting with different interleavings of the two code sequences. However, because it is possible for two different execution paths to produce the same set of results, we will only explicitly list an interleaving of the code sequences if it generates a set of results that we do not yet have.

The first possibility that we will consider is having P1 execute all of its code, and then having P2 execute all of its code. This is shown below. In each dynamic instruction trace for this problem, we append "P1:" or "P2:" before each instruction to indicate which processor is executing that operation. Also, within each comment, we append "P1." or "P2." before each register name for clarity (in order to distinguish between the two processor's register files).

```
P1: ADDI  R2,R0,#1  ;P1.R2=1
P1: SD    R2,0(R1)  ;M[A]=P1.R2 -> M[A]=1
P1: LD    R3,40(R1) ;P1.R3=M[F] -> P1.R3=1
P1: SD    R3,16(R1) ;M[C]=P1.R3 -> M[C]=1
P1: LD    R4,8(R1)  ;P1.R4=M[B] -> P1.R4=0
P1: SD    R4,24(R1) ;M[D]=P1.R4 -> M[D]=0
P2: ADDI  R2,R0,#1  ;P2.R2=1
P2: SD    R2,8(R1)  ;M[B]=P2.R2 -> M[B]=1
P2: LD    R3,48(R1) ;P2.R3=M[G] -> P2.R3=2
P2: SD    R3,16(R1) ;M[C]=P2.R3 -> M[C]=2
P2: LD    R4,0(R1)  ;P2.R4=M[A] -> P2.R4=1
P2: SD    R4,32(R1) ;M[E]=P2.R4 -> M[E]=1
```

The above execution order results in `M[C]=2`, `M[D]=0`, and `M[E]=1`.

Now we can consider allowing P1 to be the first processor to start execution, but then allowing P2 to execute some or all of its code before P1 completes. As stated before, we won't list redundant traces—e.g. if we allow P1 to execute all of its code except the last instruction (the store into `D`), then execute all of P2's code, and finally execute P1's last instruction, we will end up with the same results as above. This is because once P1 executes its next-to-last instruction (the load from `B`), P2 can no longer affect anything P1 does (because the value that P1 stores to `D` will come from its local register file). However, if we execute all but the last two instructions for P1 before switching to P2, we achieve a different set of results, as shown below.

```
P1: ADDI  R2,R0,#1  ;P1.R2=1
```

```
P1: SD    R2,0(R1)  ;M[A]=P1.R2 -> M[A]=1
P1: LD    R3,40(R1) ;P1.R3=M[F] -> P1.R3=1
P1: SD    R3,16(R1) ;M[C]=P1.R3 -> M[C]=1
P2: ADDI  R2,R0,#1  ;P2.R2=1
P2: SD    R2,8(R1)  ;M[B]=P2.R2 -> M[B]=1
P2: LD    R3,48(R1) ;P2.R3=M[G] -> P2.R3=2
P2: SD    R3,16(R1) ;M[C]=P2.R3 -> M[C]=2
P2: LD    R4,0(R1)  ;P2.R4=M[A] -> P2.R4=1
P2: SD    R4,32(R1) ;M[E]=P2.R4 -> M[E]=1
P1: LD    R4,8(R1)  ;P1.R4=M[B] -> P1.R4=1
P1: SD    R4,24(R1) ;M[D]=P1.R4 -> M[D]=1
```

By delaying the last two instructions for P1 long enough for P2 to store a value of 1 into `B`, we obtain the results: `M[C]=2`, `M[D]=1`, and `M[E]=1`.

Now let's consider delaying P1's store into `C` until P2 has completed its store into `C`.

```
P1: ADDI  R2,R0,#1  ;P1.R2=1
P1: SD    R2,0(R1)  ;M[A]=P1.R2 -> M[A]=1
P1: LD    R3,40(R1) ;P1.R3=M[F] -> P1.R3=1
P2: ADDI  R2,R0,#1  ;P2.R2=1
P2: SD    R2,8(R1)  ;M[B]=P2.R2 -> M[B]=1
P2: LD    R3,48(R1) ;P2.R3=M[G] -> P2.R3=2
P2: SD    R3,16(R1) ;M[C]=P2.R3 -> M[C]=2
P2: LD    R4,0(R1)  ;P2.R4=M[A] -> P2.R4=1
P2: SD    R4,32(R1) ;M[E]=P2.R4 -> M[E]=1
P1: SD    R3,16(R1) ;M[C]=P1.R3 -> M[C]=1
P1: LD    R4,8(R1)  ;P1.R4=M[B] -> P1.R4=1
P1: SD    R4,24(R1) ;M[D]=P1.R4 -> M[D]=1
```

This results in `M[C]=1`, `M[D]=1`, and `M[E]=1`.

Finally, consider what happens when we execute all of P2's code and then all of P1's code.

```
P2: ADDI  R2,R0,#1  ;P2.R2=1
P2: SD    R2,8(R1)  ;M[B]=P2.R2 -> M[B]=1
P2: LD    R3,48(R1) ;P2.R3=M[G] -> P2.R3=2
P2: SD    R3,16(R1) ;M[C]=P2.R3 -> M[C]=2
P2: LD    R4,0(R1)  ;P2.R4=M[A] -> P2.R4=0
P2: SD    R4,32(R1) ;M[E]=P2.R4 -> M[E]=0
P1: ADDI  R2,R0,#1  ;P1.R2=1
P1: SD    R2,0(R1)  ;M[A]=P1.R2 -> M[A]=1
P1: LD    R3,40(R1) ;P1.R3=M[F] -> P1.R3=1
P1: SD    R3,16(R1) ;M[C]=P1.R3 -> M[C]=1
P1: LD    R4,8(R1)  ;P1.R4=M[B] -> P1.R4=1
P1: SD    R4,24(R1) ;M[D]=P1.R4 -> M[D]=1
```

This results in `M[C]=1`, `M[D]=1`, and `M[E]=0`.

There are no other possible sequentially consistent results. To see why this is so, first consider whether M[D] and M[E] can both have final values of 0. In order for M[D] to end up with a value of 0, P1 must execute its last load (the load from B) before P2 executes its first store (the store into B). Sequential consistency requires that if P1 has executed its last load, then it must also have executed its first store (the store into A), which is earlier in program order. Now note that for M[E] to end up with a value of 0, P2 must execute its last load (the load from A) before P1 executes its first store (the store into A). Again, sequential consistency requires that if P2 has executed its last load, then it must have executed its first store (the store into B). When we put all of these requirements together, we see that they cannot all be true simultaneously: in order for M[D] to end up with a value of 0, P1 must execute its last load, and therefore its first store, before P2 executes its first store; in order for M[E] to end up with a value of 0, P2 must execute its last load, and therefore its first store, before P1 executes its first store. Thus, M[D] and M[E] can not both end up with values of 0.

The only other possibilities that we did not obtain were: M[C]=1, M[D]=0, M[E]=1; and M[C]=2, M[D]=1, M[E]=0. We can use the same reasoning as in the last paragraph to see why these results cannot occur. In order for M[D] to be 0, P1 must execute its last load before P2 executes its first store. This means that P1 has already completed its store to C and P2 has not yet executed its store to C. Thus, in this situation the final value of M[C] will always be the value that P2 stores to it, which will be 2. Similarly, in order for M[E] to be 0, P2 must execute its last load before P1 executes its first store. This means that P2 has already completed its store to C and P1 has not yet executed its store to C. Thus, in this situation the final value of M[C] will always be the value that P1 stores to it, which will be 1.

The set of all possible sequentially consistent results is given in the following table.

Set of Results	M[C]	M[D]	M[E]
1	1	1	0
2	1	1	1
3	2	0	1
4	2	1	1

Problem 1.B **Generalized Synchronization**

Assume now that Cy's code is run on a system that does not guarantee sequential consistency, but that memory dependencies are not violated for the accesses made by any individual processor. The system has a MEMBAR memory barrier instruction that guarantees the effects of all memory instructions executed before the MEMBAR will be made globally visible before any memory instruction after the MEMBAR is executed.

Add MEMBAR instructions to Cy's code sequences to give the same results as if the system were sequentially consistent. Use the minimum number of MEMBAR instructions.

For the remainder of this problem, we essentially play the role of a compiler that needs to guarantee a sequentially consistent result on a machine that does not guarantee sequential consistency in the hardware. Note that the machine in this part is the same as the machine in Part E (which uses weak ordering), except that this machine only has coarse-grain memory barrier instructions.

The most straightforward approach to guaranteeing a sequentially consistent set of results is to insert a memory barrier between every pair of instructions that can be reordered. This approach will ensure that instructions execute in program order. The following table shows the result of this approach. Note that memory barriers do not need to be inserted between a load of a value and a store of that same value because data dependencies enforce instruction execution in program order for that situation.

P1

```
ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2
MEMBAR
LD   R3,40(R1) ;R3=F
SD   R3,16(R1) ;C=R3
MEMBAR
LD   R4,8(R1)  ;R4=B
SD   R4,24(R1) ;D=R4
```

P2

```
ADDI R2,R0,#1  ;R2=1
SD   R2,8(R1)  ;B=R2
MEMBAR
LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3
MEMBAR
LD   R4,0(R1)  ;R4=A
SD   R4,32(R1) ;E=R4
```

The above placement of memory barriers ensures that instructions will execute in program order, as none of the memory operations can move across the barriers. However, the problem asked for the minimum number of MEMBAR instructions. We can try to take away one or more memory barriers. Consider the following code with one of the MEMBARs removed.

P1

```
ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2
MEMBAR
LD   R3,40(R1) ;R3=F
SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B
SD   R4,24(R1) ;D=R4
```

P2

```
ADDI R2,R0,#1  ;R2=1
SD   R2,8(R1)  ;B=R2
MEMBAR
LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3
MEMBAR
LD   R4,0(R1)  ;R4=A
SD   R4,32(R1) ;E=R4
```

Does the above code guarantee a sequentially consistent set of results? Consider the following dynamic reordering of instructions for P1, where the last load is moved ahead of its preceding store.

```
P1                                P2
ADDI R2,R0,#1  ;R2=1              ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2              SD   R2,8(R1)  ;B=R2
MEMBAR                            MEMBAR
LD   R3,40(R1) ;R3=F              LD   R3,48(R1) ;R3=G
LD   R4,8(R1)  ;R4=B              SD  R3,16(R1) ;C=R3
SD   R3,16(R1) ;C=R3              MEMBAR
SD   R4,24(R1) ;D=R4              LD   R4,0(R1)  ;R4=A
                                  SD   R4,32(R1) ;E=R4
```

Suppose P1 executes all of its instructions up to and including its last load (the load from B). If P2 now executes all of its instructions, and then P1 executes the rest of its instructions, this will result in M[C]=1, M[D]=0, and M[E]=1. But we already determined in Part A that this is not a sequentially consistent set of results. Thus, we must replace the removed MEMBAR instruction. For similar reasons, we cannot remove the last MEMBAR for P2. However, we can try removing P1's first memory barrier.

```
P1                                P2
ADDI R2,R0,#1  ;R2=1              ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2              SD   R2,8(R1)  ;B=R2
LD   R3,40(R1) ;R3=F              MEMBAR
SD   R3,16(R1) ;C=R3              LD   R3,48(R1) ;R3=G
MEMBAR                            SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B              MEMBAR
SD   R4,24(R1) ;D=R4              LD   R4,0(R1)  ;R4=A
                                  SD   R4,32(R1) ;E=R4
```

Consider the following dynamic reordering of instructions for P1, where its first store is moved right before the memory barrier.

```
P1                                P2
ADDI R2,R0,#1  ;R2=1              ADDI R2,R0,#1  ;R2=1
LD   R3,40(R1) ;R3=F              SD   R2,8(R1)  ;B=R2
SD   R3,16(R1) ;C=R3              MEMBAR
SD   R2,0(R1)  ;A=R2              LD   R3,48(R1) ;R3=G
MEMBAR                            SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B              MEMBAR
SD   R4,24(R1) ;D=R4              LD   R4,0(R1)  ;R4=A
                                  SD   R4,32(R1) ;E=R4
```

Suppose P1 executes its first three instructions (through the store to C), then P2 executes all of its instructions, and finally P1 executes the remainder of its instructions. This will result in M[C]=2, M[D]=1, and M[E]=0. But this is not a sequentially consistent set of results, as determined in Part A. So we cannot remove the first MEMBAR from P1's code. Similarly, we cannot remove the first MEMBAR from P2's code. Thus, the minimum number of MEMBAR instructions is given by our original code sequences:

```
P1                                P2
ADDI R2,R0,#1  ;R2=1              ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2              SD   R2,8(R1)  ;B=R2
MEMBAR                            MEMBAR
LD   R3,40(R1) ;R3=F              LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3              SD   R3,16(R1) ;C=R3
MEMBAR                            MEMBAR
LD   R4,8(R1)  ;R4=B              LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4              SD   R4,32(R1) ;E=R4
```

The above solution is acceptable. However, a performance optimization is possible. Note that the first loads for each processor access locations that are never modified by either processor (F and G). Thus, executing these loads early does not violate sequential consistency. The problem with our last example of reordering P1's first store occurred because we stored a value to C before we stored a value to A—i.e. the first two stores for each processor cannot be interchanged. This means that we can move the first memory barriers for each processor, as shown below.

```
P1                                P2
ADDI R2,R0,#1  ;R2=1              ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2              SD   R2,8(R1)  ;B=R2
LD   R3,40(R1) ;R3=F              LD   R3,48(R1) ;R3=G
MEMBAR                            MEMBAR
SD   R3,16(R1) ;C=R3              SD   R3,16(R1) ;C=R3
MEMBAR                            MEMBAR
LD   R4,8(R1)  ;R4=B              LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4              SD   R4,32(R1) ;E=R4
```

The above code sequences guarantee sequential consistency and also allow the first load for each processor to execute early, possibly improving performance if there is a cache miss. The same optimization is not possible for the second memory barrier of each processor, because in that case, it is the last load (from B or A) that is the critical operation which cannot be reordered.

Problem 1.C **Total Store Ordering**

Now consider a machine that uses finer-grain memory barrier instructions. The following instructions are available:

- MEMBAR_{rr} guarantees that all read operations initiated before the MEMBAR_{rr} will be seen before any read operation initiated after it.
- MEMBAR_{rw} guarantees that all read operations initiated before the MEMBAR_{rw} will be seen before any write operation initiated after it.
- MEMBAR_{wr} guarantees that all write operations initiated before the MEMBAR_{wr} will be seen before any read operation initiated after it.
- MEMBAR_{ww} guarantees that all write operations initiated before the MEMBAR_{ww} will be seen before any write operation initiated after it.

There is no generalized MEMBAR instruction as in Part B of this problem.

In total store ordering (TSO), a read may complete before a write that is earlier in program order if the read and write are to different addresses and there are no data dependencies. For a machine using TSO, insert the minimum number of memory barrier instructions into the code sequences for P1 and P2 so that sequential consistency is preserved.

The most straightforward approach to guaranteeing sequential consistency is to insert a MEMBAR_{wr} after a write if the next memory instruction that occurs after the write in the dynamic execution may be a read (assuming different addresses and no data dependencies). For example, in the following sample code sequence:

```
SD   R2, 0(R1)
SD   R3, 8(R1)
LD   R4, 16(R1)
```

it is possible to have the following dynamic reorderings in TSO:

```
SD   R2, 0(R1)                    LD   R4, 16(R1)
LD   R4, 16(R1)        or         SD   R2, 0(R1)
SD   R3, 8(R1)                    SD   R3, 8(R1)
```

Thus, if we place a MEMBAR_{wr} after each store, we can prevent both reorderings. But we can optimize this policy for TSO by noticing that we only need to insert a memory barrier after a store if the next instruction *is* a load in the static program. Consider the following code:

```
SD   R2, 0(R1)
SD   R3, 8(R1)
MEMBAR_wr
LD   R4, 16(R1)
```

A single memory barrier suffices to prevent any reorderings in TSO, because *all* stores before the barrier must complete before any loads after the barrier. Adopting this policy results in the following sequences for the code given in this problem.

P1

```
ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2
MEMBAR_wr
LD   R3,40(R1) ;R3=F
SD   R3,16(R1) ;C=R3
MEMBAR_wr
LD   R4,8(R1)  ;R4=B
SD   R4,24(R1) ;D=R4
```

P2

```
ADDI R2,R0,#1  ;R2=1
SD   R2,8(R1)  ;B=R2
MEMBAR_wr
LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3
MEMBAR_wr
LD   R4,0(R1)  ;R4=A
SD   R4,32(R1) ;E=R4
```

As in Part B, suppose we try taking away P1's second memory barrier, resulting in the following code.

```
P1                                  P2
ADDI R2,R0,#1  ;R2=1                ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                SD   R2,8(R1)  ;B=R2
MEMBAR_wr                           MEMBAR_wr
LD   R3,40(R1) ;R3=F                LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B                MEMBAR_wr
SD   R4,24(R1) ;D=R4                LD   R4,0(R1)  ;R4=A
                                    SD   R4,32(R1) ;E=R4
```

Does this guarantee sequential consistency? No, because the same reordering that we saw in Part B is also possible here. If we move P1's last load (the load from B) ahead of its preceding store (the store of C), then it is possible to obtain results that are not sequentially consistent. The same holds true for removing P2's second memory barrier. However, we can again try removing P1's first memory barrier.

```
P1                                  P2
ADDI R2,R0,#1  ;R2=1                ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                SD   R2,8(R1)  ;B=R2
LD   R3,40(R1) ;R3=F                MEMBAR_wr
SD   R3,16(R1) ;C=R3                LD   R3,48(R1) ;R3=G
MEMBAR_wr                           SD   R3,16(R1) ;C=R3
LD   R4,8(R1)  ;R4=B                MEMBAR_wr
SD   R4,24(R1) ;D=R4                LD   R4,0(R1)  ;R4=A
                                    SD   R4,32(R1) ;E=R4
```

This code does guarantee sequential consistency. Note that the only possible reordering consists of moving P1's first load (the load from F) ahead of its first store (the store to A). As discussed in Part B, this does not violate sequential consistency. Similarly, we can remove P2's first memory barrier, which results in the following sequences.

```
P1                                  P2
ADDI R2,R0,#1  ;R2=1                ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                SD   R2,8(R1)  ;B=R2
LD   R3,40(R1) ;R3=F                LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                SD   R3,16(R1) ;C=R3
MEMBAR_wr                           MEMBAR_wr
LD   R4,8(R1)  ;R4=B                LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4                SD   R4,32(R1) ;E=R4
```

Problem 1.D **Partial Store Ordering**

In partial store ordering (PSO), a read or a write may complete before a write that is earlier in program order if they are to different addresses and there are no data dependencies. For a machine using PSO, insert the minimum number of memory barrier instructions from Part C into the code sequences for P1 and P2 so that sequential consistency is preserved.

The most straightforward approach to guaranteeing sequential consistency is to insert a MEMBAR_{wr} after a write if the next memory instruction that occurs after the write in the dynamic execution may be a read, and to insert a MEMBAR_{ww} after a write if the next memory instruction that occurs after the write in the dynamic execution may be a write (assuming different addresses and no data dependencies). For example, in the following code sequence:

```
SD   R2, 0(R1)
LD   R4, 8(R1)
SD   R3, 16(R1)
```

the following dynamic reorderings are possible in PSO:

```
LD   R4, 8(R1)                    LD   R4, 8(R1)
SD   R2, 0(R1)          or        SD   R3, 16(R1)
SD   R3, 16(R1)                   SD   R2, 0(R1)
```

In order to guarantee sequential consistency, we need to insert the following barriers:

```
SD   R2, 0(R1)
MEMBARwr
MEMBARww
LD   R4, 8(R1)
SD   R3, 16(R1)
```

Adopting this policy prevents the reordering allowed in PSO, and results in the following sequences for the code given in the problem.

P1

```
ADDI R2,R0,#1   ;R2=1
SD   R2,0(R1)   ;A=R2
MEMBARwr
LD   R3,40(R1) ;R3=F
SD   R3,16(R1) ;C=R3
MEMBARwr
LD   R4,8(R1)   ;R4=B
SD   R4,24(R1) ;D=R4
```

P2

```
ADDI R2,R0,#1   ;R2=1
SD   R2,8(R1)   ;B=R2
MEMBARwr
LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3
MEMBARwr
LD   R4,0(R1)   ;R4=A
SD   R4,32(R1) ;E=R4
```

This is the same initial code that we started with in our solution to Part C because there are no consecutive store operations, and since data dependencies prevent the last two stores for each

processor from being moved above their preceding loads, the MEMBAR_{wr} instructions also prevent later stores from being moved before earlier stores. For the same reasons given in Part C, we cannot remove the second memory barrier for either processor. We need to reconsider whether we can remove the first memory barrier for P1, which would result in the following code:

P1	P2
ADDI R2,R0,#1 ;R2=1	ADDI R2,R0,#1 ;R2=1
SD R2,0(R1) ;A=R2	SD R2,8(R1) ;B=R2
LD R3,40(R1) ;R3=F	**MEMBAR_{wr}**
SD R3,16(R1) ;C=R3	LD R3,48(R1) ;R3=G
MEMBAR_{wr}	SD R3,16(R1) ;C=R3
LD R4,8(R1) ;R4=B	**MEMBAR_{wr}**
SD R4,24(R1) ;D=R4	LD R4,0(R1) ;R4=A
	SD R4,32(R1) ;E=R4

We showed in Part B that if P1 reorders its first load and second store above its first store (so that the store to A is now the last instruction before the memory barrier), then a set of results can be obtained that is not sequentially consistent. This reordering is also possible in the above code, which necessitates that we keep both memory barriers for each processor:

P1	P2
ADDI R2,R0,#1 ;R2=1	ADDI R2,R0,#1 ;R2=1
SD R2,0(R1) ;A=R2	SD R2,8(R1) ;B=R2
MEMBAR_{wr}	**MEMBAR_{wr}**
LD R3,40(R1) ;R3=F	LD R3,48(R1) ;R3=G
SD R3,16(R1) ;C=R3	SD R3,16(R1) ;C=R3
MEMBAR_{wr}	**MEMBAR_{wr}**
LD R4,8(R1) ;R4=B	LD R4,0(R1) ;R4=A
SD R4,24(R1) ;D=R4	SD R4,32(R1) ;E=R4

This answer is acceptable. However, we can use a similar performance optimization to the one used in Part B. Since the first memory barrier for each processor only needs to prevent the processor's first two stores from being reordered, we can change the type of barrier that we use.

P1	P2
ADDI R2,R0,#1 ;R2=1	ADDI R2,R0,#1 ;R2=1
SD R2,0(R1) ;A=R2	SD R2,8(R1) ;B=R2
MEMBAR_{ww}	**MEMBAR_{ww}**
LD R3,40(R1) ;R3=F	LD R3,48(R1) ;R3=G
SD R3,16(R1) ;C=R3	SD R3,16(R1) ;C=R3
MEMBAR_{wr}	**MEMBAR_{wr}**
LD R4,8(R1) ;R4=B	LD R4,0(R1) ;R4=A
SD R4,24(R1) ;D=R4	SD R4,32(R1) ;E=R4

The above sequences also guarantee sequentially consistent results, but allow the first load for each processor to execute early. (The $\texttt{MEMBAR}_{ww}$ for each processor can also be placed after its first load, and the result will still be correct.)

Problem 1.E **Weak Ordering**

In weak ordering (WO), a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies. For a machine using WO, insert the minimum number of memory barrier instructions from Part C into the code sequences for P1 and P2 so that sequential consistency is preserved.

The most straightforward approach to guaranteeing sequential consistency is to insert a $\texttt{MEMBAR}_{wr}$ after a write if the next memory instruction that occurs after the write in the dynamic execution may be a read, to insert a $\texttt{MEMBAR}_{ww}$ after a write if the next memory instruction that occurs after the write in the dynamic execution may be a write, to insert a $\texttt{MEMBAR}_{rr}$ after a read if the next memory instruction that occurs after the read in the dynamic execution may be a read, and to insert a $\texttt{MEMBAR}_{rw}$ after a read if the next memory instruction that occurs after the read in the dynamic execution may be a write (assuming different addresses and no data dependencies). This policy prevents the reordering allowed in WO, and results in the following sequences.

P1	P2
`ADDI R2,R0,#1 ;R2=1`	`ADDI R2,R0,#1 ;R2=1`
`SD R2,0(R1) ;A=R2`	`SD R2,8(R1) ;B=R2`
MEMBAR$_{wr}$	**MEMBAR$_{wr}$**
`LD R3,40(R1) ;R3=F`	`LD R3,48(R1) ;R3=G`
`SD R3,16(R1) ;C=R3`	`SD R3,16(R1) ;C=R3`
MEMBAR$_{wr}$	**MEMBAR$_{wr}$**
`LD R4,8(R1) ;R4=B`	`LD R4,0(R1) ;R4=A`
`SD R4,24(R1) ;D=R4`	`SD R4,32(R1) ;E=R4`

Again, because of data dependencies and the fact that stores and loads alternate in the code, the only types of memory barriers needed for the above sequences are $\texttt{MEMBAR}_{wr}$ instructions. For this part, we don't even need to attempt removing any of the above memory barriers. Because weak ordering allows all of the reorderings that are allowed by partial store ordering, and we already determined in Part D that we couldn't remove any memory barriers, we immediately know that the above sequences include the minimum number of memory barrier instructions. The same performance optimization that we used in Part D can also be used here:

P1	P2
`ADDI R2,R0,#1 ;R2=1`	`ADDI R2,R0,#1 ;R2=1`
`SD R2,0(R1) ;A=R2`	`SD R2,8(R1) ;B=R2`
MEMBAR$_{ww}$	**MEMBAR$_{ww}$**
`LD R3,40(R1) ;R3=F`	`LD R3,48(R1) ;R3=G`
`SD R3,16(R1) ;C=R3`	`SD R3,16(R1) ;C=R3`
MEMBAR$_{wr}$	**MEMBAR$_{wr}$**

```
LD   R4,8(R1)  ;R4=B              LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4              SD   R4,32(R1) ;E=R4
```

Problem 1.F **Release Consistency**

Release consistency (RC) distinguishes between *acquire* and *release* synchronization operations. An *acquire* must complete before any reads or writes following it in program order, while a read or a write before a *release* must complete before the *release*. However, reads and writes before an *acquire* may complete after the *acquire*, and reads and writes after a *release* may complete before the release. Consider the following modified versions of the original code sequences.

```
P1                                P2
ADDI R2,R0,#1  ;R2=1              ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2              SD   R2,8(R1)  ;B=R2
L1:ACQUIRE R2,56(R1);SWAP(R2,H)   L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                        BNEZ R2,L1
LD   R3,40(R1) ;R3=F              LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3              SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0             RELEASE R0,56(R1);H=0
LD   R4,8(R1)  ;R4=B              LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4              SD   R4,32(R1) ;E=R4
```

In the above sequences, the *acquire* and *release* operations modify memory location H, which is located sequentially after G. The *acquire* operation performs a read and a write, while the *release* operation performs a write. For a machine using RC, insert the minimum number of memory barrier instructions from Part C into the above code sequences for P1 and P2 so that sequential consistency is preserved.

The most straightforward approach to guaranteeing sequential consistency is to insert a MEMBAR_{wr} after a write if the next memory instruction that occurs after the write in the dynamic execution may be a read, to insert a MEMBAR_{ww} after a write if the next memory instruction that occurs after the write in the dynamic execution may be a write (not including *release* operations), to insert a MEMBAR_{rr} after a read if the next memory instruction that occurs after the read in the dynamic execution may be a read, and to insert a MEMBAR_{rw} after a read if the next memory instruction that occurs after the read in the dynamic execution may be a write (not including *release* operations—all of this assumes different addresses and no data dependencies). Additionally, we need to insert a MEMBAR_{wr} or MEMBAR_{ww} after a write if the next instruction may be an *acquire* operation, and we need to insert a MEMBAR_{rr} or MEMBAR_{rw} after a read if the next instruction may be an *acquire* operation. We also need to insert a MEMBAR_{wr} after a *release* operation if the next instruction may be a read (not including *acquire* operations), and a MEMBAR_{ww} after a *release* operation if the next instruction may be a write (not including *acquire* or *release* operations).

Obviously for long code sequences, the above policy can be very complicated. However, for the code in this problem, adopting this policy is rather straightforward, resulting in the following sequences. (As a side note, the problem did not specify the initial value in location H.

Obviously, if this initial value is not 0, then both processors will spin forever trying to get the lock. However, that only occurs in the dynamic execution—since you are essentially playing the role of a compiler for this problem, you have to insert memory barriers statically regardless of what may happen at runtime. So the solution to this part is not affected by the initial value in H—although the compiler should initialize it to 0.)

P1

```
ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2
MEMBAR_wr
MEMBAR_ww
L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1
LD   R3,40(R1) ;R3=F
SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0
MEMBAR_wr
LD   R4,8(R1)  ;R4=B
SD   R4,24(R1) ;D=R4
```

P2

```
ADDI R2,R0,#1  ;R2=1
SD   R2,8(R1)  ;B=R2
MEMBAR_wr
MEMBAR_ww
L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1
LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0
MEMBAR_wr
LD   R4,0(R1)  ;R4=A
SD   R4,32(R1) ;E=R4
```

Because of data dependencies, no reorderings are possible in the above code. Now we can consider whether all of the memory barriers are necessary. Suppose we remove the last memory barrier for P1.

```
P1                                          P2
ADDI R2,R0,#1  ;R2=1                        ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                        SD   R2,8(R1)  ;B=R2
MEMBAR_wr                                   MEMBAR_wr
MEMBAR_ww                                   MEMBAR_ww
L1:ACQUIRE R2,56(R1);SWAP(R2,H)             L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                                  BNEZ R2,L1
LD   R3,40(R1) ;R3=F                        LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                        SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0                       RELEASE R0,56(R1);H=0
LD   R4,8(R1)  ;R4=B                        MEMBAR_wr
SD   R4,24(R1) ;D=R4                        LD   R4,0(R1)  ;R4=A
                                            SD   R4,32(R1) ;E=R4
```

This allows the following dynamic reordering of instructions.

```
P1                                          P2
ADDI R2,R0,#1  ;R2=1                        ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                        SD   R2,8(R1)  ;B=R2
MEMBAR_wr                                   MEMBAR_wr
MEMBAR_ww                                   MEMBAR_ww
L1:ACQUIRE R2,56(R1);SWAP(R2,H)             L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                                  BNEZ R2,L1
LD   R4,8(R1)  ;R4=B                        LD   R3,48(R1) ;R3=G
SD   R4,24(R1) ;D=R4                        SD   R3,16(R1) ;C=R3
LD   R3,40(R1) ;R3=F                        RELEASE R0,56(R1);H=0
SD   R3,16(R1) ;C=R3                        MEMBAR_wr
RELEASE R0,56(R1);H=0                       LD   R4,0(R1)  ;R4=A
                                            SD   R4,32(R1) ;E=R4
```

Can this produce results that are not sequentially consistent? No. As we showed in Part B, this type of dynamic reordering will only generate incorrect results if P1 loads a value from B before P2 begins execution, then P2 executes at least through its store to C, and then P1 completes execution. But this can never happen in the above code, because only one processor can be in its critical section (the block between the ACQUIRE and RELEASE operations) at any given time. Thus, even if P1 loads a value from B and then P2 starts execution, P2 will spin endlessly until P1 completes its RELEASE instruction—which will cause sequentially consistent results to be generated. For the same reason, we can remove the last memory barrier from P2.

Note that as we remove memory barriers and allow instructions to be reordered, the only reordering that we can do involves moving an instruction between the ACQUIRE and RELEASE operations, because an instruction after the ACQUIRE cannot move before the ACQUIRE, and an instruction before the RELEASE cannot move after the RELEASE. Within an ACQUIRE-RELEASE block, instructions can be reordered arbitrarily (as long as dependencies are preserved). However, because only one processor can be in its ACQUIRE-RELEASE block at a

given time, that restricts the execution of the other processor. Observing these facts leads us to try the following code, which has *no* memory barriers .

```
P1                                      P2
ADDI R2,R0,#1  ;R2=1                    ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                    SD   R2,8(R1)  ;B=R2
L1:ACQUIRE R2,56(R1);SWAP(R2,H)         L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                              BNEZ R2,L1
LD   R3,40(R1) ;R3=F                    LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                    SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0                   RELEASE R0,56(R1);H=0
LD   R4,8(R1)  ;R4=B                    LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4                    SD   R4,32(R1) ;E=R4
```

Does the above code guarantee sequentially consistent results? From Part A, we know that the following results for (`M[C]`, `M[D]`, `M[E]`) are not sequentially consistent:

{(1, 0, 0), (1, 0, 1), (2, 0, 0), (2, 1, 0)}

First let's consider whether the first and third results can occur in the above code—i.e. `M[D]=0` and `M[E]=0`. As we discussed in Part A, in order for `M[D]` to end up with a value of 0, P1 must execute its last load before P2 executes its first store; in order for `M[E]` to end up with a value of 0, P2 must execute its last load before P1 executes its first store. Can both of these occur simultaneously in the above code? They can only occur if, in the dynamic execution, the last load for each processor is moved above its own first store. However, this is only possible if all of the memory operations are moved within the `ACQUIRE-RELEASE` blocks, as shown by the following possible dynamic reordering.

```
P1                                      P2
ADDI R2,R0,#1  ;R2=1                    ADDI R2,R0,#1  ;R2=1
L1:ACQUIRE R2,56(R1);SWAP(R2,H)         L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                              BNEZ R2,L1
LD   R4,8(R1)  ;R4=B                    LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4                    SD   R4,32(R1) ;E=R4
SD   R2,0(R1)  ;A=R2                    SD   R2,8(R1)  ;B=R2
LD   R3,40(R1) ;R3=F                    LD   R3,48(R1) ;R3=G
SD   R3,16(R1) ;C=R3                    SD   R3,16(R1) ;C=R3
RELEASE R0,56(R1);H=0                   RELEASE R0,56(R1);H=0
```

Note that when all of the memory operations are within `ACQUIRE-RELEASE` blocks, one processor must completely execute all of its memory instructions before the other processor can begin executing its memory instructions. However, this is equivalent to having one processor completely execute all of its instructions, and then having the other processor completely execute all of its instructions—which is guaranteed to produce a sequentially consistent result.

Now we can consider whether the second set of results—`M[C]=1`, `M[D]=0`, and `M[E]=1`—can occur. This requires P1 to execute its last load (from `B`) before P2 executes its first store (to `B`),

and it also requires P2 to execute its store to C before P1 executes its store to C. This can only be accomplished if, in the dynamic execution, P1's last load is moved above its store to C and P2's store to C is executed before P1's store to C. Consider the following dynamic reordering.

```
P1                                      P2
ADDI R2,R0,#1  ;R2=1                    ADDI R2,R0,#1  ;R2=1
SD   R2,0(R1)  ;A=R2                    SD   R2,8(R1)  ;B=R2
L1:ACQUIRE R2,56(R1);SWAP(R2,H)         L1:ACQUIRE R2,56(R1);SWAP(R2,H)
BNEZ R2,L1                              BNEZ R2,L1
LD   R3,40(R1) ;R3=F                    LD   R3,48(R1) ;R3=G
LD   R4,8(R1)  ;R4=B                    SD   R3,16(R1) ;C=R3
SD   R3,16(R1) ;C=R3                    RELEASE R0,56(R1);H=0
RELEASE R0,56(R1);H=0                   LD   R4,0(R1)  ;R4=A
SD   R4,24(R1) ;D=R4                    SD   R4,32(R1) ;E=R4
```

P1's load from B will execute before its store to C. However, there is no possible way that we can have P1 execute that load before P2 executes its store to B, and also have P2 execute its store to C before P1 executes its store to C. This is because if P1 loads from B, then it is in the critical section—either P2 has not yet entered the critical section (meaning that P1 must execute its store to C before P2 can execute its store to C), or P2 has already exited the critical section (meaning that P2 has already executed its store to B). There is no reordering of instructions for P1 or P2 that will change this fact. We can use similar reasoning to show that it is not possible to obtain the results M[C]=2, M[D]=1, and M[E]=0.

Thus, we do not need to insert any memory barriers into the given code in order to guarantee sequentially consistent results.