CO21BTECH11002
Aayush Kumar
Theory Assignment 3

1)

The statement is true. If we replace the atomic SRSW register array with regular SRSW registers, then the construction does yield an atomic MRSW register. If the read and write calls are non-overlapping, each table[i] will hold the most recently written value in them, which will be returned by the read() call. If they are overlapping, read() may return any value.

2)

The strongest property that this 64-bit register satisfies is “safe”. Consider the following example: Initially all the bits in both the registers are filled with 0. A writer starts writing, and it fills all the bits in both the registers with 1. But when the writer completes writing in the first register, the reader starts reading and reads 1s in the first register and 0s in the second register. This value read by the reader is neither the old value in register nor the new value. Hence, this register does not satisfy atomic or regular property.

3)

1.

Consider the following construct:

```
class Consensus <T> {
        StickyBit s ;
        T decide (T V) {
                s . write ( V ) ;
                T r = s . read ( )
                return  r;
        }
}
```

Since write() is called before any read() calls, validity is given. The sticky bit changes its value only once (after the first write() call), hence, consensus is given. Thus, if the StickyBit class is wait free then our consensus protocol is also wait free.

2.
Consider the following construct:

```
class Consensus <T > {
        T n = #threads;
        T bits = #bits;
        T tID = threadID;
        StickyBit [] s = new StickyBit [ bits ];
        MRSWRegister [] reg = new MRSWRegister [2 * n ]; // Initially 0.
        T decide ( T value ) {
                T v = value ;
                reg [ n + tID ] = value ;
                reg [ tID ] = 1;
                for (int i = 0; i < bits ; i ++) {
                        bool b = (v & (1<<i)) ; // ith bit of v
                        s [ i ]. write ( b ) ;
                        if ( s [ i ]. read () != b ) {
                                for (int j = 0; j < n ; j ++) {
                                        if ( reg [ j ] == 1 && bits [0: i ] match in reg [ n + j ] and
                                s ) {
                                        v = reg [ n + j ];
                                        }
                                }
                        }
                }
                return binaryToInt( s );
        }
}
```

The function decide() begins by marking the current thread as having submitted a value, which is done by setting a_reg[tID] to 1 and a_reg[n + tID] to the proposed value. After this, it proceeds to change all sticky bits to a specified value, denoted as 'v'. If it encounters a situation where a sticky bit is already set to a different value, the function searches for another thread that has proposed the same value as the already set sticky bits. It then assists that particular thread in completing the assignment of sticky bits. The function continues this process until all sticky bits have been successfully assigned, and once this is accomplished, it returns their final values.

It maintains consistency by returning a value derived from the sticky bit array after attempting a write operation on each array element. The presence of bounded loops ensures wait-freedom.

We need to demonstrate the validity of Consensus. To do so, let's consider value_i[k], which represents the k'th bit of the value proposed by thread i. Now, suppose we have two threads, i and j, and two indices, k and l, such that s[k] = value_i[k] and s[l] = value_j[l], where value_i[k] is not equal to value_j[k], and value_i[l] is not equal to value_j[l]. This situation implies that thread j must have had a value v with bits [0..l-1] that do not match those of s.

However, this scenario is impossible because, during each iteration of the outer loop, we check whether the bits match. If they do not match, the value v is updated to match the value of a thread with matching bits. Therefore, it is impossible for thread j to have bits [0..l-1] that do not match those of s.

As a result, all threads return the same value, demonstrating that the protocol is valid.

4)

Second step is the faulty step. For implementing the peek() method, we need more than getAndSet() and getAndIncrement() methods.