Exercise 34. Consider the safe Boolean MRSW construction shown in Fig. **??**. True or false: if we replace the safe Boolean SRSW register array with an array of safe M-valued SRSW registers, then the construction yields a safe M-valued MRSW register. Justify your answer.

Solution *true:* If we replace the safe Boolean SRSW register array with an array of safe M-valued SRSW registers, then the construction does yield a safe M-valued MRSW register.

The proof is almost exactly the same. For non-overlapping method calls, each table$[i]$ holds the most recently written value, which is returned by the next read() call. For overlapping method calls, the reader may return any value, because the component registers are safe.

Exercise 35. Consider the safe Boolean MRSW construction shown in Fig. **??**. True or false: if we replace the safe Boolean SRSW register array with an array of regular Boolean SRSW registers, then the construction yields a regular Boolean MRSW register. Justify your answer.

Solution *true:* If we replace the safe Boolean SRSW register array with an array of regular Boolean SRSW registers, then the construction does yield a regular Boolean MRSW register.

The proof is almost exactly the same. For non-overlapping method calls, each table$[i]$ holds the most recently written value, which is returned by the next read() call. For overlapping method calls, the reader may return either the new value or the old value, because the component registers are regular.

Exercise 36. Consider the atomic MRSW construction shown in Fig. **??**. True or false: if we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register. Justify your answer.

Solution *true:* If we replace the safe SRSW register array with an array of regular Boolean SRSW registers, then the construction does yield a regular MRSW register.

The proof is almost exactly the same. For non-overlapping method calls, each table$[i]$ holds the most recently written value, which is returned by the next read() call. For overlapping method calls, the reader may return either the new value or the old value, because the component registers are regular.

Exercise 37. Give an example of a quiescently-consistent register execution that
is not regular.

```
P  |−write(0)−| |−write(1)−| |−write(2)−|
Q            |−read(0)−| |−read(0)−| |−read(0)−|
```

Exercise 38. Consider the safe Boolean MRSW construction shown in Fig. **??**. True or false: if we replace the safe Boolean SRSW register array with an array of *regular* M-valued SRSW registers, then the construction yields a regular M-valued MRSW register. Justify your answer.

Solution [] *true:* If we replace the safe Boolean SRSW register array with an array of regular M-valued SRSW registers, then the construction does yield a regular M-valued MRSW register.

If the read() call does not overlap any write() call, then each table[i] holds the most recently written value, which is returned by the read() call. Otherwise, suppose the read() call overlaps one or more write() calls. Let v_0 be the value written by the latest preceding write() call, and let $v_1, \ldots, v_k$ be the sequence of values written by overlapping write() calls. We claim the read returns one of the v_i.

- If thread i reads table[i] after the writer writes v_i to table[i] and before it writes v_{i+1}, then the read return v_i.
- If thread i reads table[i] while the writer writes v_{i+1} to table[i], then because table[i] is regular the read returns either v_i or v_{i+1}.

Exercise 39. Consider the regular Boolean MRSW construction shown in Fig. **??**. True or false: if we replace the safe Boolean MRSW register with a safe M-valued MRSW register, then the construction yields a regular M-valued MRSW register. Justify your answer.

Solution *False:* If we replace the safe Boolean MRSW register shown in Figure **??** with a safe M-valued MRSW register, then the construction does not yield a regular M-valued MRSW register.

A read overlapping a write can return an arbitrary value in the range $0 \ldots M$, but regularity requires that it return either the old value or the new value. The Binary construction works only because there are only two possible values, 0 or 1, so an arbitrary value is either the old or new value.

Exercise 40. Does Peterson's two-thread mutual exclusion algorithm work if we replace shared atomic registers with regular registers?

Solution Yes it does, because regular registers act differently only when two reads are done concurrently with a write. Consider the case where a write to flag by a thread overlaps the other thread reading in the **while** loop. If thread A is writing flag i in the lock() method, a thread B reading this field might read the new value of *true*, which keeps it in the loop, and later reads a temporary flicker of the value to its old value, *false*. This would allow B to acquire the lock, which is acceptable since the writing thread A would have set itself to be the victim as its next step, which would have allowed B to acquire the lock anyway. A would get then get caught as expected in the **while** loop, and only one thread would get in. If we look at the write to flag in the unlock() method, we conclude that a read could flicker to a *true* value even after a previous read had noticed the new value of *false*. But this would never happen, since if the reading thread firrst reads value *false* it must be reading in the **while** loop, and it would break out of it, causing the second read not to occur. Finally, we look at the victim value. If this value flickers then on the first read of the new value the condition in the **while** loop will be *false*, and the second read could not occur, just as in the previous case. Therefore, the Peterson Lock does work when regular registers are used.

Exercise 41. Consider the following implementation of a Register<> in a distributed, message-passing system. There are n processors $P_0, \ldots, P_{n-1}$ arranged in a ring, where P_i can send messages only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link.

Each processor keeps a copy of the shared register.

- To read a register, the processor reads the copy in its local memory.
- A processor P_i starts a `write()` call of value v to register x, by sending the message "P_i: write v to x" to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_j: write v to x," for $i \neq j$, then it writes v to its local copy of x, and forwards the message to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_i: write v to x," then it writes v to its local copy of x, and discards the message. The `write()` call is now complete.

Give a short justification or counterexample.
If `write()` calls never overlap,

- Is this register implementation regular?
- Is it atomic?

If multiple processors call `write()`,

- Is this register implementation atomic?

Solution Terminology: the *logical register* is the virtual register physically distributed over multiple processors. A *register* is one of the registers used in every processor to build the logical register.

When only a single writer exists then the logical register is regular: if one processor reads its local register while another processor's write is in progress, then it will see either the new or the old value, depending on how far around the ring the write message has propagated.

The logical register is not Atomic. Processor P_0 starts writing, and processor P_1 updates its local register. The next two method calls overlap the write, but do not overlap one another.

1. P_1 reads the new value.
2. P_2 reads the old value.

This history is not linearizable, because P_2's call must be linearized after P_1's, but P_2 saw the old value

Exercise 42. You learn that your competitor, the Acme Atomic Register Com-
pany, has developed a way to use Boolean (single-bit) atomic registers to construct an efficient *write-once* single-reader single-writer atomic register. Through your spies, you acquire the code fragment shown in Fig. 1, which is unfortunately missing the code for `read()`. Your job is to devise a `read()` method that works for this class, and to justify (informally) why it works. (Remember that the register is write-once, meaning that your read will overlap at most one write.)

Solution [] Think of the register as having three copies of the value, which we call left (low-order), middle, and right (high). Think of the writer as writing one bit at a time from left to right, and the reader as reading one bit at a time from right to left. Because the reader and writer move in opposite directions, they overlap on only one of the three copies.

- If the left and middle copies are the same, then the overlap occurred on the right copy, so take the left copy.
- If the right and middle copies are the same, then the overlap occurred on the left copy, so take the right copy.
- if all three copies are different, then the overlap occurred on the middle copy, so take the right copy.

Exercise 43. Prove that the safe Boolean MRSW register construction from safe Boolean SRSW registers illustrated in Fig. **??** is a correct implementation of a regular MRSW register if the component registers are regular SRSW registers.

Solution If A's read() call does not overlap any write() call, then the read() call returns the value of s_table [A], which is the most recently written value. For overlapping method calls, the reader may return any value written in that interval, because the component registers are regular.

```
class AcmeRegister implements Register{
  // N is the total number of threads
  // Atomic multi−reader single−writer registers
  private BoolRegister [] b = new BoolMRSWRegister[3 * N];
  public void write(int x) {
    boolean[] v = intToBooleanArray(x);
    // copy v[i] to b[i] in ascending order of i
    for (int i = 0; i < N; i++)
      b[i].write(v[i]);
    // copy v[i] to b[N+i] in ascending order of i
    for (int i = 0; i < N; i++)
      b[N+i].write(v[i]);
    // copy v[i] to b[2N+i] in ascending order of i
    for (int i = 0; i < N; i++)
      b[(2*N)+i].write(v[i]);
  }
  public int read() {
    // missing code
  }
}
```

Figure 1: Partial acme register implementation.

Exercise 44. A monotonic counter is a data structure $c = c_1 \dots c_m$ (i.e., c is composed of the individual digits c_j, $j > 0$) such that $c^0 \leq c^1 \leq c^2 \leq \dots$, where $c^0, c^1, c^2, \dots$ denote the successive values assumed by c.

If c is not a single digit, then reading and writing c may involve several separate operations. A read of c which is performed concurrently with one or more writes to c may obtain a value different from any of the versions c^j, $j \geq 0$. The value obtained may contain traces of several different versions. If a read obtains traces of versions c^{i_1}, ..., c^{i_m}, then we say that it obtained a value of $c^{k,l}$ where $k =$ minimum (i_1, ..., i_m) and $l =$ maximum (i_1, ..., i_m), so $0 \leq k \leq l$. If $k = l$, then $c^{k,l} = c^k$ and the read obtained is a consistent version of c.

Hence, the correctness condition for such a counter simply asserts that if a read obtains the value $c^{k,l}$, then $c^k \leq c^{k,l} \leq c^l$. The individual digits c_j are also assumed to be atomic digits.

Give an implementation of a monotonic counter, given that the following theorems are true:

Theorem 0.0.1. If $c = c_1 \dots c_m$ is always written from right to left, then a read from left to right obtains a sequence of values $c_1^{k_1,\ell_1}, \dots,\ c_m^{k_m,\ell_m}$ with $k_1 \leq \ell_1 \leq k_2 \leq \dots \leq k_m \leq \ell_m$.

Lemma 0.0.2. Let $c = c_1 \dots c_m$ and assume that $c^0 \leq c^1 \leq \dots \leq c_m$

1. If $i_1 \leq \dots \leq i_m \leq i$ then $c_1^{i_1} \dots c_m^{i_m} \leq c^i$.

2. If $i_1 \geq \dots \geq i_m \geq i$ then $c_1^{i_1} \dots c_m^{i_m} \geq c^i$.

Theorem 0.0.3. Let $c = c_1 \cdots c_m$ and assume that $c^0 \leq c^1 \leq \dots$, where the digits c_i are atomic.

1. If c is always written from right to left, then a read from left to right obtains a value $c^{k,l} \leq c^l$.

2. If c is always written from left to right, then a read from right to left obtains a value $c^{k,l} \geq c^l$.

Note that if a read of c obtains traces of version $c^j, j \geq 0$, then:

- The beginning of the read preceded the end of the write of c^{j+1}.

- The end of the read followed the beginning of the write of c^j.

Furthermore, we say that a read (write) of c is performed from left to right if for each $j > 0$, the read (write) of c_j is completed before the read (write) of c_{j+1} has begun. Reading or writing from right to left is defined in the analogous way.

Finally, always remember that subscripts refer to individual digits of c while superscripts refer to successive values assumed by c.

Solution We keep two copies *cl* and *c*2 of the counter. The writer first writes to *c*2 from left to right, and then to *cl* from right to left. The reader first reads *cl* from left to right and then reads *c*2 from right to left.

time	A	B	
0	flag $[A] = true$	flag $[B] = true$	both start doorways
1	read **label**$[B]$ is 0	read **label**$[A]$ is 0	
2		**label**$[B] = 1$	B leaves doorway
3		read flag $[A]$ is *true*	B starts waiting
4	**label**$[A] = 1$	read **label**$[A]$ is 25!	overlapping calls
5	read flag $[B]$ is *true*		A starts waiting
6	read **label**$[B]$ is 1		A starts waiting
7	enters critical section	enters critical section	

Figure 2: Bad execution for unsafe bakery lock

Exercise 45. Prove Theorem 0.0.1 of Exercise 44. Note that since $k_j \leq \ell_j$, you need only to show that $\ell_j \leq k_{j+1}$ if $1 \leq j < m$.

(2) Prove Theorem 0.0.3 of Exercise 44, given that the Lemma is true.

Solution For a full solution see the following paper:

Lamport, L. 1990. Concurrent reading and writing of clocks. ACM Trans. Comput. Syst. 8, 4 (Nov. 1990), 305-310.DOI= `http://doi.acm.org/10.1145/128733.128736`.

Exercise 46. We saw safe and regular registers earlier in this chapter. Define a *wraparound* register that has the property that there is a value v such that adding 1 to v yields 0, not $v + 1$.

If we replace the Bakery algorithm's shared variables with either (a) flickering, (b) safe, (c) or wraparound registers, then does it still satisfy (1) mutual exclusion, (2) first-come-first-served ordering?

You should provide six answers (some may imply others). Justify each claim.

Solution Note that in lectures, "usafe" registers are called *safe*, and "flickering" registers are called *regular*.

The Bakery lock does not provide mutual exclusion using unsafe registers. Suppose we have two threads, A and B, both with labels 0. In the execution shown in Figure 2, A and B proceed in parallel, except that B enters its waiting loop first, and happens to read **label**$[A]$ at the same time A writes to it. B decides that A has a later label (when in fact A's label is earlier), and both A and B enter the critical section at the same time.

Even though the lock does not provide mutual exclusion, it does provide first-come-first-served (FCFS) in the following sense. If A finishes its doorway before B starts its doorway, then $(\textbf{label}[A], A) << (\textbf{label}[B], B)$. B's read of **label**$[A]$ does not overlap A's write, so B will read the correct value, take a larger value, and then be blocked until A leaves the critical section.

The Bakery lock using flickering registers provides both FCFS and mutual exclusion. The argument for FCFS is the same as for unsafe registers.

Suppose A and B are concurrently in the critical section. Let labeling$_A$ and labeling$_B$ be the last respective sequences of acquiring new labels prior to entering the critical section. Suppose that $(\textbf{label}[A], A) << (\textbf{label}[B], B)$. When B successfully completed the test in its waiting section, it must have read that $\mathsf{flag}[A]$ was *false* or that $(\textbf{label}[B], B) << (\textbf{label}[A], A)$.

If B read that $\mathsf{flag}[A]$ was *false*, then that read preceded or overlapped A's write to $\mathsf{flag}[A]$, which preceded A's read of $\mathsf{flag}[B]$ and write to $\mathsf{flag}[A]$, implying that $\textbf{label}[A] > \textbf{label}[B]$, a contradiction.

So B observed that $(\textbf{label}[B], B) << (\textbf{label}[A], A)$. Since A never wrote such a value, B must have read $\textbf{label}[A]$ at the time A was updating it. But B must have seen either the value being written, or the previous value, both of which are less than or equal to $\textbf{label}[B]$, a contradiction.

Wraparound registers do not provide either FCFS or mutual exclusion. It fails to be FCFS because the label written by a later doorway is not necessarily larger than the labels it read. Essentially the same counterexample as the unsafe register case shows that the lock fails to provide mutual exclusion.