Written_sync.ans

1. sol

Because WAR and WAW can be resolved through optimization, e.g renaming.

For WAR, `add r0 r1 r2` / `lw r1 *` , apply renaming, `add R10 R1 R2` / `lw R11 *`

there will be no longer a dependency problem.

For WAW, `add r0 r1 r2` / `lw r0 *` , apply renaming, `add R10 R1 R2` / `lw R11 *`

there will be no longer a dependency problem.

2. sol

PROS: for the condition that one processor writes to a cache while many others are needing the data, update is much more efficient than invalidation. Because it will eliminate the time for other processor to reload the new data from memory.

CONS: for the condition that one processor writes to a cache many times before another one need it, updating will be a big waste. Because every time the processor write to the cache, updating is executed once, while in fact only the last update is useful. Thus, it would be a big waste of time of updating a cache again and again uselessly. And for this condition, invalidation will only be executed once, which is very efficient.

3. sol

Without cache coherence, turn can be 2 for P1 and 1 for P2 simultaneously. If P1 don't have cache of flag2 and P2 don't have cache of flag1 originally, the two while loop conditions may meet at the same time. Thus, the program doesn't guarantee mutually exclusive executions of the critical sections.

4. sol

As sequential memory consistency means that every write must be seen on all processors before any succeeding read or write can be issued, the output can only be 1.

5. sol

(1) For the processor firstly entering the code, the return value of test_and_set would be unlocked, and the location is locked. If the multiprocessor supports sequential memory consistency, the state of location (locked) should be visible to others before they issue it. Thus, other processors won't entering critical section until the first processor unlocks the location.

(2) Considering the above situation, as I have pointed out, if the multiprocessor supports sequential memory consistency, the first while loop is already enough for other processors to stay waiting. Thus, the second loop is redundant.

(3) Yes, Dr. Foobar is right. If we take away the second while loop, also considering the above situation, when the first processor is in critical section, other processors is

"locked" and execute "*test_and_set(location, locked)*" continually, which means write to location again and again. As we all know, to support sequential memory consistency, write operation is a very high consuming work, much more for this continually write operation from many processors. But with the second loop, most of the time other processors just need to read the value of the location, which help improve the performance.

6. sol

The code denotes how to emulate:

```
L victim = *location;
compare_and_swap(location, victim, value);
return victim;
```