CS 740, Fall 1995
Homework Assignment 6

Assigned: Nov. 28
Due: Mon., Dec. 11, 9:00AM

Policy

You may work in a group of up to 3 people in solving the problems for this assignment. However, you must prepare your own writeup, and you must identify your collaborators in your writeup.

You only need to hand in a hard copy document for this assignment. Formatted text is preferred to hand written. If some question arises during grading regarding the correctness of your code for the programming portion, you may be asked to submit an electronic version.

The files you need for this assignment are available in the directory:

```
/afs/cs.cmu.edu/academic/class/15740-f95/public/asst/asst6
```

The problems in this assignment all make use of the model checker SMV. This program has been installed in the class AFS directory, although you may need to compile it for your CPU type. A user's manual for SMV is included in the assignment directory.

The problems in this assignment require running SMV many times, making incremental changes to the files (this is the easiest way to gain an understanding of the program). For each problem, show the fragment of the SMV file you are modifying, and indicate the response produced by SMV, i.e., that the verification succeeded, or describing the counterexample it produced. You *should not* include the detailed trace file. A verbal or pictorial explanation would be more useful.

One useful technique for maintaining multiple versions of an SMV description is to create a single "master" copy containing C-style conditional compilation directives. You then generate the different versions by running the C preprocessor using the appropriate set of command line arguments. For example, the version of `snoopy.smv` shown in the assignment directory was generated from the master version `snoopy.master`. The appropriate incantations are stored in the `Makefile`, including some good parameters for running SMV.

1. Exploring the Write Invalidate Protocol

The file `snoopy.smv` is an SMV description of a write-invalidate cache protocol, as was described in class. This protocol is similar to the one described in Section 8.3 of Hennessy and Patterson but is somewhat more complete in its details.

Task 1: Tracing Interesting Behaviors

When SMV is run on the cache model, it simply returns a message stating that the verification succeeded. Often we'd like to be convinced that the protocol has the potential to do interesting

things, as well. The best way to do this is to purposely run SMV with an incorrect specification that where the desired behavior is a counterexample.

Show the following behaviors can actually occur. Give the temporal logic formulas you use to generate the behaviors, and describe the sequence generated:

1. An `invalidate` operation occurs on the bus.
2. A cache performs a write back due to a write miss by its processor.
3. A cache performs a write back due to a write miss by some other processor.
4. A single cache gains ownership first of block A and later of block B.

2. Synchronization Operations

The (extended) MIPS instruction set provides a pair of instructions to support processor synchronization: load linked `LL` and store conditional `SC`. These are discussed in Sect. 8.5 of Hennessy & Patterson.

Load linked has the same format as the load instruction, except that it causes the effective address to be stored in a special "link" register. From this point on, if any other processor changes the value stored at the physical address indicated by the link register, then the register will be cleared.

Store conditional has the same format as the store instruction, but it has significantly different behavior. The store will take place only if the effective address matches the link register value. Otherwise, the store will be aborted. [The behavior is not defined if the effective address differs from that of the most recent load linked instruction.] The success or failure of the operation is indicated by setting the source register (register `Rt`) to 1 or 0.

Here's an example of code to acquire a lock, adapted from p. 698:

```
# busy wait until lock available
Loop:  ll   R2, lock     # Get lock value & set link register
       bne  R2, R0, Loop # If nonzero, try again
       li   R2, 1
       sc   R2, lock     # Attempt to grab lock
       beq  R2, R0, Loop # Store failed, restart busy wait
```

The code to release the lock is simply:

```
       sw   R0, lock     # Set lock to 0
```

In a typical shared memory multiprocessor system, the kernels running on the different processors access any shared state only within their "critical sections," where entry to the critical section requires first acquiring a lock. The behavior of each kernel can then be viewed as follows:

```
while(1) {
  ... Operate on private state ...
  acquire(lock);
  /* Critical section */
  ... Operate on shared state  ...
  release(lock);
}
```

Task 2: Synchronization Considerations

In implementing the load linked operation, we have a choice of treating it like a normal processor read, and therefore asking for a read-only copy of the cache line, or as a special form of read, where we obtain an exclusive copy in anticipation of the upcoming store conditional.

Discuss the trade-offs between the two alternatives. Assume that a number of processors are competing for a lock using the code shown above.

Task 3: Implementing Synchronization Operations

Describe how the cache protocol could be extended to support these two operations. Indicate what additional registers and signals would be required, any changes to the protocol state diagram, and how the state values would be updated.

Modify the SMV description of the bus to implement the operations. To simplify the modeling of processor/cache interactions, you can assume that the success or failure of the store conditional is implemented as a Boolean signal.

Create an extended processor model in which a processor performs a load linked followed by a store conditional. Verify that the store conditional will succeed if and only if no other cache could have obtained an exclusive copy of the block in the interim. **Advice:** Instead of trying to encode all this as a temporal logic formula, you might want to create a separate module that "watches" the system behavior detects whether the appropriate events have occurred.

2.1. Task 4: Fairness Considerations

Our cache protocol is designed to be fair: no processor will be prevented from completing a read or write indefinitely. Show that this does not guarantee that the lock acquisition code will be fair, i.e., describe a scenario in which a processor will be denied access to its critical section indefinitely. Do you expect this to be a problem in practice? What steps could be taken to eliminate, or at least minimize, this problem?

3. Read Broadcast

In the current design, if a number of processors want to read the same cache block, they must each perform a separate bus read. It would make sense instead to let the caches make use of any tranfers

on the bus that would enable them to satisfy any pending reads.

Task 5: Implementing Read Broadcast

Describe how the state diagram should be altered to implement this capability. The idea is to let one cache make the request and to have the others "piggy-back" off the resulting bus transaction. Implement all of the conditions in which a cache could opportunistically satisfy a pending read.

Modify the SMV to take advantage of the broadcast capability of the bus.

Use the counterexample facility to demonstrate a case where all three processors read a block with a single bus transaction.

How would this modification affect the behavior of a set of processors contending for access to their critical sections, as shown in the code above?