CS 740, Fall 1997
Assignment 6:
Part III: Cache Protocol Verification with SMV

Assigned: Fri., Nov. 21
Due: Fri., Dec. 5

Logistics

You may work in a group of up to 3 people in solving the problems for this assignment. You should turn in a single report for your entire group, identifying all of the group members.

Any clarifications and revisions to the assignment will be posted on the class bboard and Web page. There is no electronic handin for this assignment.

In the following, *AFS740* refers to the directory:

```
/afs/cs.cmu.edu/academic/class/15740-f97/public
```

The files you need for this assignment are in the directory *AFS740*/`asst/asst6/cache`.

The problems in this assignment all make use of the model checker SMV. This program has been installed as *AFS740*/`bin/smv` for many CPU types. You may need to do the installation for your machine. Refer to the file *AFS740*/`README` for instructions on how to do this. A user's manual (`smvmanual.r2.2.ps`) for SMV and a man page (`smv.1`) 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.

Exploring the Cache Protocol

The file `snoopy.smv` is an SMV description of a write-invalidate cache protocol, as was described in class on November 19 and 21. This protocol is similar to the one described in Section 8.3 of Hennessy and Patterson but is somewhat more complete in its details.

Problem 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 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 operation due to a write miss by some other processor.
4. A single cache gains ownership first of block A and later of block B.

1. 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   $3, lock     # Get lock value & set link register
       bne  $3, $0, Loop # If nonzero, try again
       li   $3, 1
       sc   $3, lock     # Attempt to grab lock
       beq  $3, $0, Loop # Store failed, restart busy wait
```

The code to release the lock is simply:

```
       sw   $0, 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);
}
```

Problem 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.

Problem 3: Maintaining Link Register

The MIPS instruction set manual states the following caveat: "A cache miss that occurs between LL [load-linked] and SC [store-conditional] may cause SC to fail, so no load or store instruction should occur between LL and SC." From this statement, what can you infer about the types of memory transactions that the cache controller must monitor to properly maintain the link register? Is this caveat necessary for a bus-based shared memory machine? What about for a network-based machine?

Problem 4: Implementing Synchronization Operations

Describe how the cache protocol, as described in the Lecture of November 19, 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.

Problem 5: Modifying SMV Version of Protocol

Modify the SMV description of the bus to implement the operations. Here are some tips:

- The repetoire of processor instructions should be expanded to include `load-linked` and `store-conditional`.
- The processor model should not generate instruction sequences that are considered invalid (e.g,, a store-conditional to an address other than that of the preceding load-linked).
- The current Boolean-valued `stall` signal from the cache to the processor should be expanded to a `status` signal which can have values `stall`, `succeed`, and `fail` to indicate the success or failure of store-conditional.

- As always, you need only model the control operations supporting these instructions and not any transfer of data.

Show that your modified model satisfies the basic liveness and safety conditions from before.

Also, demonstrate that it is possible for store-conditional to succeed using the SMV counterexample facility.

Problem 6: Verifiying LL/SC

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 only if no other cache has 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 and detects whether the appropriate events have occurred.