VIS : Technology Transfer Course

University of California, Berkeley

Lab # 11 : Project

Gitanjali Swamy and Dominique Borrione

5/16/96

1 Description of System

In this lab, we will be create and verify cache coherence properties in a multiprocessor system. This system is parametrized with N processors P_i. For our example $N = 2$. These processors may be READING, WRITING, or IDLE, depending on the instruction which may be READ_WORD, WRITE_WORD, or COMPUTE. If a processor is READING or WRITING, it has an associated address and value.

Each processor has a cache C_i associated with it. The cache state is one of $Ready, Rwait, Rgrant, Wwait, Wgrant$. The cache ($cache\_state = Ready$) reads the request from the processor and checks whether it already has the corresponding address. If the request is a *write_req* and the cache does not have EXCLUSIVE access to the address, then it forwards the request ($cache\_req = blk\_excl$) to the *Directory cum Main Mem cum arbiter D*. The cache moves to the $Wwait$ state. If the cache already has EXCLUSIVE access to the location, then it writes to it. This is referred to as a write HIT and following this the cache enter $cache\_state = Rgrant$. If the request is a *read_req* and the address is in INVALID in the cache, then it forwards the request ($cache\_req = blk\_rreq$) to the arbiter D. Otherwise it returns the value in the cache; this is referred to as aread HIT.

The arbiter serves one cache at a time. For instance it serves C_1 in state ONE. If C_1 sends a write request $cache\_req = blk\_excl$, the arbiter moves from state ONE to state ONESERVE and services the request by sending an *inval* to all other caches for the given address. The *inval* forces all other caches to invalidate data in given address. Following this the arbiter sends an *blk_ok* to C_1 to allow it EXCLUSIVE access to the address. If the cache requests a read ($cache\_req = blk\_rreq$), and if any other cache has EXCLUSIVE access to the address, the arbiter moves to state ONEWAIT and services the request by first sending a request for the data (*write_back_req*) to the cache with EXCLUSIVE access. All cache's always give priority to *write_back_req* or *inval* signals from the directory arbiter. The arbiter remains in ONEWAIT until the requested cache returns a *blk_ok* as well as the requested *back_data*. Once it receives this data, it first updates the *main_mem*, and then moves to state ONESERVE. It concludes the serve by returning the *blk_data* requested to C_1. Alternately, if no other cache has EXCLUSIVE access to the location, then arbiter moves directly to state ONESERVE and serves the request by returning the value in *main_mem*.

2 Description of Properties

Correctness1: "Safety Properties"

- Safety: If C_1 has exclusive access to some address, then it is either invalid in C_2 or not in C_2.
- The cache state stored in the memory and the cache actual state must be consistent.

Correctness2: "Coherency":

A memory system is *Coherent* iff

- A READ by P_i to X that follows a WRITE by P_i to X, with no intermediate WRITE by other P_k returns value A.
- A READ by P_i to X that follows a WRITE by P_k to X, with no intermediate WRITE returns value A.
- WRITE's to X by P_i, P_k are serialized (see correctness).

If you are having difficulty bringing up **VIS**, or do not have the sample files for this lab, or have any other problem, please ask for help.

3 Homework Exercise

Look at the example provided in session11. The processor is described in `processor.v`, the cache and controller in `cache_ctrl.v`, and the directory cum arbiter in `directory.v`. The entire system with all interaction is in `coherence.v`. Familiarize yourself with the example, and correlate the lecture material with the example provided. Identify the correspondence between the FSMs described in the lecture and the Verilog description.

4 Hands on Experience

Problem #1 Compile the verilog example `coherence.v` (main module) using vl2mv.

Problem #2 Read `coherence.mv` into VIS and compute the reached states. Kill the process if it takes more than 3 minutes (CNTRL-C). Try dynamic re-ordering on the example (do save the order !). Attempt to compute the reached states.

Problem #3 Read in given ordering (`coh-order`) and compute reached states.

Problem #4 Attempt to verify the properties using simulation.

Problem #5 Formulate the properties in CTL and model check them.

Problem #6 If the model checking fails, then correct the design.

Problem #7 Note that the arbiter is not efficient; if only C_1 makes requests, it still cycles through 2N states before servicing successive requests. Modify your arbiter to be more efficient.

Problem #8 Write a property to show that your arbiter is better. Show that this property fails on the old design, but passes on the new one.