An ACL2 Proof of Write Invalidate Cache Coherence

J Strother Moore[1]

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
moore@cs.utexas.edu

Abstract. As a pedagogical exercise in ACL2, we formalize and prove the correctness of a write invalidate cache scheme. In our formalization, an arbitrary number of processors, each with its own local cache, interact with a global memory via a bus which is snooped by the caches.

1 Ongoing Industrial Applications of ACL2

The ACL2 theorem proving system is finding use in industrial-scale verification projects. Two significant projects which have been reported previously are

- the mechanical verification of the floating-point division microcode for the AMD-K5^{TM}[6], and
- the ACL2 modeling of the Motorola CAP digital signal processor and its use to prove that a pipeline hazard detection predicate was correct and that several DSP microcode applications were correct [1].

The abstract of a recent talk given by David Russinoff of Advanced Micro Devices, Inc., summarizes the current AMD work with ACL2:

> Formal design verification at AMD has focused on the elementary arithmetic floating point operations, beginning with the **FDIV** and **FSQRT** instructions of the AMD-K5^{TM} processor, and continuing with the **FADD**, **FSUB**, **FMUL**, **FDIV**, and **FSQRT** instructions of the AMD-K7^{TM} processor, which is currently under development.
>
> Design-level mathematical models of all of these operations have been rigorously proved to comply with behavioral specifications derived from IEEE Standard 754 and the Intel Pentium Family User's Manual. Every step of each proof (with one minor exception in the case of **FSQRT**) has been formally encoded in the logic of ACL2 and mechanically checked with the ACL2 prover.
>
> In this talk, we shall briefly describe the results of this project:
>
> - a reusable general theory of floating point representation, rounding, and logical operations on bit vectors;
> - an automatic translator from AMD's RTL language (essentially a subset of Verilog) to ACL2;

- several design flaws that were exposed by our analysis and ultimately corrected after surviving extensive testing;
- the proofs of correctness of the operations listed above.

This monumental work is reported in [8]. To corroborate the ACL2 RTL translation, AMD executed the ACL2 translation on a test suite of 80 million floating point problems and compared the results to their standard RTL simulation. The bugs found by Russinoff's proofs were not uncovered by this extensive test suite.

ACL2 is being used to model microprocessors at several industrial sites. For example, at Rockwell-Collins, Inc., ACL2 is being used experimentally to provide an executable model of JEM1, the world's first silicon Java Virtual Machine [2].

In addition, [9] describes an ACL2 model of a microprocessor with multiple, out-of-order instruction issue with a reorder buffer, speculative execution and exceptions. Proofs are being done to relate this model to a more conventional ISA model. While this work is not industrial scale, the microprocessor is more complicated than many academic models studied.

Finally, ACL2 is being used at EDS, Inc., in the verification of "renovation rules" used in COGEN 2000^{TM}, an in-house, proprietary suite of tools used at EDS CIO Services to renovate legacy COBOL code that is not "Year 2000 compliant." Roughly speaking, the problem is how to use given fixed-width data fields to encode the dates in a 100-year window so that commonly used relations are correctly and efficiently implemented. Matt Kaufmann, in [4], describes how he used ACL2 to verify that certain rules were correct. In fact, he describes an environment in ACL2 that can be used conveniently to verify newly proposed transformation rules and to simplify date manipulation expressions.

2 What is ACL2?

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." The logic is both an applicative programming language and a first order mathematical logic[5]. Technically, the programming language is an extension of a subset of applicative Common Lisp. In addition, "ACL2" is the name we use for the implemented system[1]. The system provides an execution environment for the programming language and a theorem proving environment for the logic.

The theorem prover's behavior is determined by rules in its data base. The rules are determined by the theorems the system has proved already. The user can guide the system to deep proofs by presenting it with an appropriate sequence of lemmas to prove. The user is not responsible for soundness, since no rule can be entered into the system's data base until it (or more accurately, its corresponding formula) has been proved as a theorem.

Collections of definitions and theorems can be assembled into "books." The user can instruct the system to include a book into the data base, thereby adding all the (non-local) rules contained in the book. Books thus provide both a scoping mechanism and a way to take advantage of the work of others.

3 A Write Invalidate Cache Example

In the rest of this paper we present a formal model of a write invalidate cache scheme and prove it correct. Write invalidate schemes are known not to scale efficiently to large numbers of processors. But this is a simple problem that is familiar to many readers. Furthermore, at first sight, it may not seem to lend itself to Common Lisp modeling. By choosing this example, we hope to arose the reader's curiosity while illustrating ACL2.

Our model is based on the discussion on page 658 of [3]. Our model includes an arbitrary number of processors, each with its own local cache connected via a bus to one global memory. Fundamentally, a cache is a table of "cache lines", each of which is associated with an address and contains a value and a flag indicating whether the cache line is valid – i.e., whether the value for the given address is consistent with the value assigned by the global memory. Each processor can send its cache read and write requests, receiving some response. The cache's behavior on a read request depends on whether it contains a valid cache line for the requested address. If it does, it responds with the associated value. If it does not, the cache sends a read request on the bus, waits for the reply, constructs a new cache line containing the resulting value, and then responds to the processor with the value obtained from memory. The cache's behavior on a write request is to update (or create) the appropriate cache line, send a write request on the bus, and respond to the processor with the value written. All caches snoop the bus. Read requests are ignored. Write requests cause the other caches to invalidate the corresponding cache line, if any. We model the read/write actions of the individual processors as interleaved atomic actions.

To specify this system we construct a cache-free model in which the interleaved actions are played directly against the global memory. We prove that the response to every read/write action is the same in the two models. The proof requires less than 10 seconds on a Sun Ultra 2 (177 MHz).

For pedagogical purposes, we have divided our work on this problem into three books, discussed in turn below. These books are available at http://www.-cs.utexas.edu/users/moore/publications/write-invalidate-cache/index.html.

4 Utilities

In the `"utilities"` book we define some generic functions and predicates for dealing with problems of this sort. Fundamental to our formalization is the notion of an *association list*. Each element in an association list (or *alist*) is a pair consisting of a *key* and a *datum*. The key is said to be *bound* to the datum. If no key in an alist is bound twice, we say the alist has *unique keys*. The function `fetch` fetches the datum associated with a key in an alist. The function `deposit` binds a given key to a given datum in a given alist.

A *memory* is an alist binding addresses to values.

A *cache* is an alist binding addresses to pairs of the form (*value* *flag*). Such pairs are called *lines*. A line is said to be *valid* if *flag* is on. A cache is *ok* with respect to a memory if every valid cache line has as its value the value of the corresponding address in the memory.

A *named cache* is a processor identifier and a cache. In a slight abuse of terminology, we call a list of such pairs simply a *caches* list. Note that a caches list is itself an alist in which each key is a processor identifier and each datum a cache.

An *event* is a pair consisting of a processor identifier and an action. We call the processor identifier of an event the *agent*. An *action* is a list either of the form `(READ` *addr*`)` or `(WRITE` *addr val*`)`. A list of events is *appropriate* with respect to a caches list if each agent has an associated cache, i.e., if the set of agents of the events is a subset of the keys of the caches list.

The concepts mentioned above are formalized with functions named appropriately. To save space we do not exhibit those definitions here.

The `"utilities"` book contains fifteen theorems relating these concepts in various ways. Most of the theorems tell us how the various concepts are affected by `deposit`s. For example,

```
(defthm  cache-okp-deposit2
  (implies (and (cache-okp cache mem)
                (unique-keysp cache))
           (cache-okp (deposit addr (list any nil) cache)
                      (deposit addr val mem)))) .
```

Informally, this theorem tells us that if *cache* is ok with respect to *mem* (i.e., every line with a true flag contains the correct value), and the cache has unique keys, then invalidating the (first) line for *addr* produces a cache that is ok with respect to a memory in which *addr* has been changed. We do not mention the others but they are stated entirely in terms of the concepts enumerated above, plus ACL2 primitives.

ACL2 requires less than 4 seconds to admit all the definitions and prove all the theorems in the `"utilities"` book. This is called *certifying* the book. No hints are required, but the order in which the theorems are proved is important.

5 Cache System

In the `"system"` book, we define our model of the write invalidate cache system. A cache system state, *csys*, is a pair consisting of a caches list and a memory. We say that *p* is a processor of *csys* if *p* is bound in the caches list of *csys*. We define a *good cache system state* with

```
(defun good-csysp (csys)
  (and (unique-keysp (caches csys))
       (every-cache-unique-keysp (caches csys))
       (every-cache-okp (caches csys) (mem csys)))) .
```

The semantics of an action by a processor on its cache and the memory is formalized by

```
(defun do-action (action cache mem)
  (let ((op (car action))
        (addr (cadr action))
        (val (caddr action)))
    (case op
      (READ
       (let* ((line (fetch addr cache))
              (oldval (car line))
              (validp  (cadr line)))
         (cond
          ((and line validp)
           (mv oldval cache nil))
          (t (let ((memval (fetch addr mem)))
               (mv memval
                   (deposit addr (list memval t) cache)
                   (list 'READ addr)))))))
      (otherwise ; WRITE
       (mv val
           (deposit addr (list val t) cache)
           (list 'WRITE addr val))))))
```

This function returns three results packaged together using ACL2's "multiple values" facility. The first of the three values is the response to the processor. The second is the new version of the local cache. The third is the message sent to the bus. We now paraphrase the definition above. Recall that an action is of the form `(READ` *addr*`)` or `(WRITE` *addr val*`)`. Consider first the case where the operation is `READ`. If the cache has the corresponding line and it is valid, then respond with the value, do not change the cache, and send no message. Otherwise, respond with the value from memory, change the cache accordingly, and send the `READ` request on the bus. (In an implementation, they are done in the opposite temporal order, but that is not relevant here.) In the case where the operation is a `WRITE`, respond with the written value, change the cache accordingly and send the `WRITE` request on the bus.

Here is how a cache snoops the bus:

```
(defun snoop (msg cache)
  (cond
   ((null msg) cache)
   (t (let ((op (car msg))
            (addr (cadr msg)))
        (case op
          (READ cache)
          (otherwise ; WRITE
           (let* ((line (fetch addr cache))
                  (val (car line))
  (validp (cadr line)))
             (cond ((and line validp)
                    (deposit addr
                             (list val nil)
```

```
                        cache))
                   (t cache)))))))) .
```

We can paraphrase this: If there is no message on the bus, do not change the cache. If the message is a `READ`, do not change the cache. Otherwise (the message is a `WRITE`), if the cache contains a lined marked valid, invalidate it.

We similarly define `(new-mem` *msg mem*`)` to describe how memory changes in response to a message on the bus.

Here is how the system state, *csys*, is changed by a single *action* performed by a processor *p*.

```
(defun step-csys (p action csys)
  (let ((cache (fetch p (caches csys))))
    (mv-let (response cache' msg)
            (do-action action cache (mem csys))
            (mv response
                (csys (deposit p
                               cache'
                               (snoop-others p
                                             msg
                                             (caches csys)))
                      (new-mem msg (mem csys)))))))
```

This function returns two values. The first is the response of *p*'s cache to the action. The second is a modified system state. We compute this as follows. First, do the action on *p*'s cache, obtaining three results which are bound to the variables *response*, *cache*' and *msg*, respectively.[1] The first is the response of the cache to the action, the second is the new cache for *p*, and the third is the message sent to the bus. The modified system state is then built with `csys` from a modified list of caches and a modified memory. The modified list of caches is obtained by letting the other caches snoop the message and then depositing *p*'s new cache into the *p* slot. The modified memory is obtained via `new-mem`. We leave the simple subroutine `snoop-others` to the reader; it calls `snoop` on every cache in the caches except *p*'s.

Finally, here is how we run a sequence of events.

```
(defun run-csys (events csys)
  (cond ((endp events) nil)
        (t (let ((p (car (car events)))
                 (action (cadr (car events))))
             (mv-let (response csys')
                     (step-csys p action csys)
                     (cons (cons p response)
                           (run-csys (cdr events) csys'))))))))
```

Recall than an event consists of a processor identifier and an action. The function above returns a history of every processor that performed an action and the

[1] In this paper we sometimes use primed variable names, as in *cache*'. ACL2 does not permit such names. Our actual text uses a caret instead of a prime.

response to the action. It should be obvious how this is done: If the list of events is empty, return the empty list. Otherwise, step the system once with the indicated processor and action. Obtain two values, a response and a new state. Pair the processor and its response and cons that pair onto the result of running the rest of the events on the new state.

Also in this book we prove two key theorems. Both are invariants about the state, say *csys*', produced as the second value by `step-csys` on some state *csys*. The first invariant is that if *csys* is a good state then so is *csys*'. The second invariant is that if *events* is appropriate with respect to the caches in *csys*, it is appropriate with respect to those in *csys*'. One can regard our formalization and proof of these invariants as simple discipline: if a system is thought to enjoy an invariant, say so and prove it. In fact, we use both invariants in our correctness proof below.

ACL2 requires the user to state seven lemmas to lead it to the proofs of these two invariants. ACL2 uses less than 4 seconds to certify the `"system"` book.

6 Correctness

To specify what it is for the cache system to be correct, we define a model in which the processors interact directly with the memory. In this cache-free model, the state is simply the memory. An action evokes a response from memory and possibly changes memory, as described by the two values returned by the following function.

```
(defun step-mem (action mem)
  (let ((op (car action))
        (addr (cadr action))
        (val (caddr action)))
   (case op
     (READ (mv (fetch addr mem) mem))
     (otherwise ; WRITE
      (mv val (deposit addr val mem))))))
```

If the action is a `READ`, the response is the associated value in the memory and no change is visited upon the memory. If the action is a `WRITE`, the response is the value written and the memory is changed by depositing that value at the associated address.

To run a sequence of events against a memory we use:

```
(defun run-mem (events mem)
  (cond ((endp events) nil)
        (t (let ((p (car (car events)))
                 (action (cadr (car events))))
             (mv-let (response mem')
                     (step-mem action mem)
                     (cons (cons p response)
                           (run-mem (cdr events) mem')))))))) .
```

The correctness of the cache system is given by

```
(defthm  cache-system-correct
  (implies (and (good-csysp csys)
                (appropriate-eventsp events (caches csys)))
           (equal (run-csys events csys)
                  (run-mem events (mem csys))))) ,
```

which may be paraphrased as follows. Suppose *csys* is a good cache system state and every agent in *events* is a processor in the system. Then running *events* in the cache system *csys* produces the same history of processor/responses as running the same *events* in the simple shared memory model, starting from the initial memory in *csys*.

We now illustrate how to interact with ACL2 to lead it to interesting proofs. The main idea is to use ACL2 to help us decide how to proceed. We start by asking it to prove the conjecture above, without actually expecting it to succeed! However, it is helpful when trying prove theorems about functions like **`run-csys`** and **`run-mem`** to "disable" the two step functions, **`step-csys`** and **`step-mem`**, because they introduce case analysis and make the failed proof attempt hard to decipher. By "disable" we mean to attempt the proof without using the definitions of those two functions. This will help us identify what we need to prove about them. We similarly disable **`good-csysp`** during the proof attempt.

The proof attempt proceeds by an induction on the structure of *events*. In the inductive step, the variable *csys* above is replaced by the cache system state returned as the second value of **`step-csys`**. The two previously mentioned invariants in the **`"system"`** book are sufficient to relieve the **`good-csysp`** and **`appropriate-eventsp`** hypotheses of the induction hypothesis. Nevertheless, the proof attempt runs on for many seconds and eventually starts causing a lot of garbage collections. We abort the proof attempt and really look at the output for the first time.

A subgoal near the beginning of the aborted proof attempt reads[2]

```
(IMPLIES (AND ...
              (GOOD-CSYSP CSYS)
              (BOUND P (CACHES CSYS))
              ...)
         (EQUAL (MV-NTH 0 (STEP-CSYS P ACTION CSYS))
                (MV-NTH 0 (STEP-MEM ACTION (MEM CSYS))))) .
```

A little further down is another subgoal with similar hypotheses and the conclusion:

```
         (EQUAL (RUN-MEM EVENTS
                         (MEM (MV-NTH 1 (STEP-CSYS P ACTION CSYS))))
                (RUN-MEM EVENTS
                         (MV-NTH 1 (STEP-MEM ACTION (MEM CSYS))))) .
```

[2] In the actual output, the variable `EVENTS3` is used for `P` and `EVENTS5` is used for `ACTION`. We have changed the names to make the formulas more suggestive.

These two subgoals suggest the need for the following two lemmas.

```
(defthm mv-nth-0-step-csys
  (implies (and (good-csysp csys)
                (bound p (caches csys)))
           (equal (mv-nth 0 (step-csys p action csys))
                  (mv-nth 0 (step-mem action (mem csys)))))) .
```

and

```
(defthm mv-nth-1-step-csys
  (implies (and (good-csysp csys)
                (bound p (caches csys)))
           (equal (mem (mv-nth 1 (step-csys p action csys)))
                  (mv-nth 1 (step-mem action (mem csys)))))) .
```

What do these two formidable looking conjectures say? The first hypothesis of each lemma requires that *csys* be a good cache system state. The second hypothesis requires that *p* be one of the processors in *csys*. The two lemmas equate a left-hand side term with a right-hand side term. To read the left-hand sides, it is helpful to know that `mv-nth` is the ACL2 function used to retrieve a value from a "multiple values" tuple. Also, recall that `step-csys` returns two values, the response of the processor's cache to an action and a new cache system state, while `step-mem` returns the cache-free response and the new memory. But now it is easy to interpret the two lemmas. The first says that the response of processor *p*'s cache to an action is the same as memory's response. The second says that the memory produced by the cache system is that of the simple system.

These two lemmas are easy to prove, by expanding the definitions of the two step functions and using the results in the `"utilities"` book.

With these two lemmas in the data base, ACL2 proves the correctness theorem. ACL2 requires less than 2 seconds to certify the `"correctness"` book. The total time to certify all three books is 9.05 seconds.

7 Conclusions

The simplicity of this example hides several important observations. First, we are talking here about an "infinite state" system: there are an arbitrary number of processors, a cache can be arbitrarily large, addresses and data values are arbitrarily large, and the memory is arbitrarily large. The proof is made *easier* by these infinities, not harder.

Second, interaction with the theorem prover helps the informed user find proofs. Here is some advice for the new user. Simple theorems are usually proved quickly. Keep ACL2 on a "short leash." Either it succeeds within a few seconds or it should be aborted. Treat the first response as "yes, I believe the fact you just told me." Treat the second as "no, I don't believe it." In the case of a "no," look at the output to determine what obvious fact you know that ACL2 does not. Sometimes you will think "But I've already told it this fact!" Most likely, you did, but it is unable to use that "old" fact because some hypothesis could not be relieved or some term does not actually match. Given what you've told it, what is it missing? Once you realize what ACL2 is missing, formulate the new fact as a lemma and get ACL2 to say "yes" by continuing this dialog. When the system says "yes" to the lemma, return to the original conjecture again and see if ACL2 agrees with it now. Unfortunately, the ACL2 interface does not make it at all obvious that such a structured dialog is being conducted. We illustrate this dialog approach in the source files for these books, available on the web.

Third, the proof described here takes virtually no time. The "bottleneck," if there is one, is the time it takes the user to model the cache system and explain why it is correct. To the extent that the explanation is simple, the proof is simple and quick. In this case, the explanation is simple: The cache system is always in a good state with appropriate events. These are the two invariants in the `"system"` book. Furthermore, in such a state, the response and new memory produced by an action in the cache system are the same as those produced by the simple system. These are the two lemmas noted in the `"correctness"` book. These facts are obvious to anyone familiar with the design. Stating them requires familiarity with the language of ACL2 and the user's own model of the cache system. Their proofs are easily constructed by the dialog method described above.

References

1. B. Brock, M. Kaufmann, and J S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, M. Srivas and A. Camilleri (eds.), Springer-Verlag, November, 1996, pp. 275–293.
2. D. A. Greve and M. M. Wilding Stack-based Java a back-to-future step", Electronic Engineering Times, Jan. 12, 1998, pp. 92.
3. J. Hennessy and D. Paterson, *Computer Architecture A Quantitative Approach, Second Edition*, Morgan Kaufmann Publishers, Inc., San Francisco, 1996.
4. M. Kaufmann. ACL2 Support for Verification Projects. In *15th International Conference on Automated Deduction (CADE)* (to appear, 1998).
5. M. Kaufmann and J S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. In *IEEE Transactions on Software Engineering* **23**(4), April, 1997, pp. 203–213.
6. J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5$_K$86 Floating-Point Division Algorithm, *IEEE Transactions on Computers* (to appear). See URL http://devil.ece.utexas-.edu:80/~lynch/divide/divide.html for a preliminary draft.
7. J S. Moore. Symbolic Simulation: An ACL2 Approach. 1998. (submitted for publication)
8. D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithms of the AMD-K7^{TM} Processor URL `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
9. J. Sawada, W. Hunt, Jr., Processor Verification with Precise Exceptions and Speculative Execution, *Computer Aided Verification 1998*, Lecture Notes in Computer Science, Springer Verlag, 1998 (to appear).