A Formal Model of a Large Memory that Supports Efficient Execution

Warren A. Hunt, Jr. and Matt Kaufmann
Dept. of Computer Science, University of Texas, Austin, TX 78701
Email: {hunt,kaufmann}@cs.utexas.edu

***Abstract*—The validation and application of formal processor models benefits fundamentally from both efficient execution and automated reasoning about the models. We present a memory model written in the ACL2 logic, with both reasoning support and a runtime environment, that accomplishes these objectives. Our memory model provides a space-efficient implementation for an address space of 2^{48} bytes, and is used in our development of an ISA model for x86 instructions. We define and prove invariants, and we use them to prove useful lemmas and to formally verify absence of run-time simulator errors. Our memory model also supports efficient execution through constant-time read and write access in an applicative setting.**

I. Introduction

We describe a model of memory suitable for specifying and simulating a 64-bit microprocessor instruction-set architecture (ISA). The model is formalized in the logic of the ACL2 theorem prover [1], [2]. Our contribution is the formal specification and mechanical verification that our implementation provides a single, large, uniformly-addressed memory with space-efficient, high-speed (constant-time) performance. We desire high performance because of our interest in validating a (uni)processor model by simulating and comparing with expected results. As far as we know, our verified memory model is more time and space efficient than other models of a large memory formalized using the language of a theorem prover.

Microprocessor specifications require a model of its memory and its memory operations. Our model provides a memory of 2^{48} bytes; this is the address space defined by contemporary x86 implementations. Actually, some x86 implementations define a 52-bit address space, but such implementations require the use of the x86 memory management unit to access physical memory locations larger than 2^{48} bytes. If the need arises, we expect to be able to parameterize our model to offer larger (or smaller) memory address spaces.

Likely, every microprocessor design in the last 40 years has been modeled, and necessarily every such model includes a memory model, often written in C or Verilog. Our effort is focused on memory models that are (1) defined formally, (2) scale up to very large memories, (3) provide high-speed simulation, and (4) support mechanized reasoning. The memory model we present here defines four read (`rmXY`) operations and four write (`wmXY`) operations with the following interface signatures:

```
rm08: addr * mem → byte      wm08: addr * byte  * mem → mem
rm16: addr * mem → word      wm16: addr * word  * mem → mem
rm32: addr * mem → dword     wm32: addr * dword * mem → mem
rm64: addr * mem → qword     wm64: addr * qword * mem → mem
```

In this paper, we specify and verify a memory model satisfying the four (numbered) properties above and then we use this memory model to implement the eight memory functions just identified. In particular, Section V discusses classic read-over-write properties. Various microprocessor memory models can be layered on top of our memory model. Microprocessors providing virtual memory or other memory access mechanisms require a model of the physical memory; our focus here is the formalization of the physical memory interface. The complete source code and theorems for our memory model and its use in a partial x86 ISA specification may be found elsewhere [3].

Our efforts in this area started with the FM8501 and FM8502 microprocessors [4], [5], which included complete memory models whose performance was linear in the address size; thus, these models were not practical for simulating large memories. Our FM9001 microprocessor model [6] included a tree-based memory model that provided constant-time, tree-based accesses. Anthony Fox has developed a tree-based memory using HOL for his ARM microprocessor model, with a focus on program verification performance measured in tens of accesses per second [7]; by comparison, our performance is measured in hundreds of thousands of accesses per second (see Section VI).

Jared Davis used ACL2 to implement a tree-based 64-bit memory [8], which could make several hundred thousand accesses per second running on an Intel Pentium 4 in 2006. David Hardin (personal communication) reports 350,000 bytes/second on a 2.4 GHz Intel Core 2 Duo, using a version of Davis's model incorporated into an AAMP7 model [9]. The memory models of Davis and Hardin provide less than 1% of the memory performance we present here.

We begin by providing background on ACL2, the system that we are using for memory modeling. In Section III we present our two-level memory model. Section IV discusses invariants on our model and their role in efficient execution and fundamental properties. In Section V we present our higher-level read and write operations for bytes, words, doublewords, and quadwords, together with formally verified read-over-write properties of our memory model. Because we are using this memory model to support the modeling of microprocessor specifications, Section VI provides some mem-

978-0-9835678-2-0/12/$31.00 ©2012 FMCAD Inc.

ory access/update benchmark data. We conclude by observing that our memory implementation has been verified to operate correctly while providing sufficient performance to be used as the foundation of an ISA simulator.

II. ACL2 PRELIMINARIES

ACL2 [1] is a freely available system that provides a theorem prover and a programming language, both of which are based on a first-order logic of recursive functions [10], [11]. The logic is compatible with Common Lisp — indeed, "ACL2" is an acronym that might be written as "ACL^2" and stands for "A Computational Logic for Applicative Common Lisp" — and thus an executable image can be built on any of seven Common Lisp implementations. As a result, ACL2 provides efficient execution by way of Common Lisp compilers.

The initial theory for ACL2 contains axioms for primitive functions such as `car` (the head of a list or first component of a pair) and `cdr` (the tail of a list or second component of a pair). It also contains axioms for Common Lisp functions, such as `ash` (arithmetic shift), and it introduces axioms for user-supplied definitions.

ACL2 provides a top-level read-eval-print loop. Arbitrary ACL2 expressions may be submitted for evaluation. Of special interest are *events*, including definitions and theorems; these modify the logical database for subsequent proof and evaluation. For example, our memory model defines `n45p` to return true on 45-bit natural number inputs.

Links to numerous papers that apply ACL2, as well as detailed hypertext documentation and installation instructions, may be found on the ACL2 home page [2]. In the remainder of this section we briefly introduce aspects of ACL2 that are referenced in the remainder of this paper.

A. ACL2 basics

As is the case for Lisp, the syntax of ACL2 is generally case-insensitive and is based on prefix notation: (`function` `argument`$_1$ ... `argument`$_k$). For example, the term denoting the sum of `x` and `y` is `(+ x y)`. A semicolon (';') begins a comment to the end of the line, generally shown in *italics* in this paper. Other ACL2 syntax used in this paper will probably make sense from the context, but we say a bit here about local variables, which may be introduced using `let` for parallel binding or `let*` for sequential binding. The term

```
(let ((x1 t1)
      (x2 t2)
      ...)
  (f ... x1 ... x2 ...))
```

binds variable x_1 to the value of term t_1, variable x_2 to the value of term t_2, and so on, before evaluating the indicated call of `f`. `Let*` is similar but has a sequential semantics: each binding applies to subsequent bindings. The following log illustrates the difference between the parallel bindings of `let` and the sequential bindings of `let*`.

```
ACL2 !>(let ((x 3))
         (let ((x (1+ x))  ; x is bound to 4
               (y x))      ; y is bound to old  x: 3
           (list x y)))  ; return list of  x and  y
(4 3)
ACL2 !>(let ((x 3))
         (let* ((x (1+ x)) ; x is bound to 4
                (y x))     ; y is bound to new  x: 4
           (list x y)))  ; return list of  x and  y
(4 4)
ACL2 !>
```

Functions in ACL2 may return multiple values. Logically, a multiple-value return is just a return value that is a list; but the implementation can avoid building list objects. Syntactic restrictions enforce proper use of multiple values. The primitives `mv` and `mv-let` create and bind multiple values, respectively, as we now illustrate (see [12] for details). The following function takes two numbers and uses the if-then-else primitive to return two values: the smaller and larger of those numbers, respectively. Note that here and throughout the paper, we avoid using the Lisp `defun` command, showing instead just the logical axiom added by the definition. For complete definitions, including `declare` forms that can improve efficiency and specify *guards* (*cf.* Section II-B), see the associated technical report [3].

```
Definition.
(min-max x y)
= (if (< x y) (mv x y) (mv y x))
```

The next function exponentiates the smaller of two numbers to the power of the larger.

```
Definition.
(expt-min-max x y)
= (mv-let (smaller bigger)
          (min-max x y)
          (expt smaller bigger))
```

Then for example:

```
ACL2 !>(expt-min-max 2 5)
32
ACL2 !>(expt-min-max 5 2)
32
ACL2 !>
```

B. Definitions and guards

The logic of ACL2 is untyped. However, ACL2 definitions may specify preconditions, known as *guards*. Consider for example the following definition of a function that returns the reciprocal of the difference of its inputs.

```
Definition.
(f x y)
= (/ (- x y))
Guard:
(and (rationalp x)
     (rationalp y)
     (not (equal x y)))
```

When this form is submitted, ACL2 performs *guard verification*, a static check (using the theorem prover) that for every function call that takes place during evaluation, the arguments satisfy the guard of that function. The example above generates the following two proof obligations, each under the hypothesis of the above guard: `x` and `y` are distinct rational numbers.

- The indicated subtraction requires that its arguments, `x` and `y`, are rationals.
- The indicated reciprocal operation requires that its argument, `(- x y)`, is a non-zero rational.

ACL2 easily discharges these proof obligations. Subsequently, any call of `f` will be evaluated in Common Lisp using the above code. Indeed, guards provide a link between the ACL2 logic and the host Lisp implementation, by allowing the use of Common Lisp evaluation in a way that avoids runtime errors.

Note that while guards are important for supporting evaluation by the host Lisp, they are irrelevant logically. For example, the ACL2 logic includes the following axiom, which implies that the reciprocal of a non-number or zero is zero.

```
Axiom. completion-of-unary-/
(equal (/ x)
       (if (and (acl2-numberp x)
                (not (equal x 0)))
           (/ x)
         0))
```

Thus, for example, one can prove `(equal (/ 0) 0)` with the ACL2 theorem prover. An attempt to evaluate `(/ 0)` (the reciprocal of zero) in the ACL2 read-eval-print loop will, by default, result in an error that reports a guard violation.

C. Single-threaded objects (stobjs)

Our memory model uses ACL2 single-threaded objects, or *stobjs* [13]. The first-order logic of ACL2 represents stobjs using linear lists, without side-effects. But for execution, ACL2 enforces syntactic *single-threadedness* restrictions on function definitions involving stobjs. so that they provide constant-time access and update using arrays, which can be made resizable. ACL2 provides detailed stobj documentation [12]; here we use an example to convey key ideas.

The following ACL2 event specifies a single-threaded object, `st`, which has a single field, `store`, which in turn is an array of 31-bit non-negative integers, initially all 0. Although `store` initially has length 8, it can be resized to arbitrary lengths.

```
(defstobj st
  (store :type (array (unsigned-byte 31) (8))
         :initially 0
         :resizable t))
```

Logically, `st` is just a one-element list whose unique element, `store`, is itself just a list. The following theorem makes this claim formally, where `store-length` is a function introduced by the above `defstobj` event, returning the number of entries in `store`. Throughout this paper and also in our ACL2 development, we give theorems descriptive names.

```
Theorem. store-length-computes-len
(implies
 (stp st)  ; st satisfies its recognizer
 (and (consp st)            ; st is a list
      (null (cdr st))       ; st has only one member
      (equal (len (car st)) ; list-length of  store
             (store-length st))))
```

However, the implementation guarantees that no list construction is performed when updating `store`, and unlike linear list operations, every access to `store` is done in constant time with an array indexing operation.

D. About proofs

Our presentation below focuses on formalization and proof highlights, avoiding proof details. Full ACL2 input scripts may be found elsewhere [3].

Our proofs of the read-over-write lemmas in Section V-B take advantage of the GL symbolic simulation package [14]. That package requires the experimental "hons" extension, ACL2(h), of the ACL2 theorem prover [15], [12], which we therefore used for this effort.

III. Memory Structure, Access, and Update

Our memory model is based on an array of 64-bit quadwords, providing the illusion of a memory containing 2^{48} bytes. The model includes read and write operations, `memi` and `!memi`, for quadwords (64 bits). Later, in Section V, we build on these primitives to define byte-addressed reads and writes for various sizes: byte (8 bits), word (16 bits), doubleword (32 bits), and quadword (64 bits).

The correctness of our model is captured by the following standard property of arrays. We briefly discuss its proof in Subsection IV-C.

```
Theorem. memi-!memi
(implies
  (and (x86-64p x86-64)        ; Memory OK
       (n45p i)                ; Read address OK
       (n45p j))               ; Write address OK
  (equal (memi i               ; Read address
               (!memi j        ; Write address
                      v        ; Value to write
                      x86-64)) ; Initial memory
         (if (equal i j)       ; For equal addresses
             v                 ; the read value is v
           (memi i x86-64))))) ; else, unchanged
```

Our memory model is implemented using a data structure with three fields; see Fig. 1. Although the memory is conceptually an array of 2^{48} bytes, we choose our data structure for space efficiency. Our choice of 27 bits is somewhat arbitrary, but intended to balance the size of `mem-table` — 2^{27} (134M) entries — with the size of the `mem-array`, which initially consists of (somewhat arbitrarily) 100 pages, each containing 2^{21} bytes (2MB).

- The memory address table, `mem-table`, is indexed by the top (most significant) 27 bits of a 45-bit quadword address. Its valid entries are 45-bit addresses.
- The memory array, `mem-array`, is indexed by 45-bit quadword addresses from `mem-table`. Its entries are the memory quadword values.
- A 45-bit quadword address, `mem-array-next-addr`, points to the next free two-megabyte section ("page") of `mem-array`.

We use the ACL2 *stobj* mechanism (see Section II) to implement these fields.

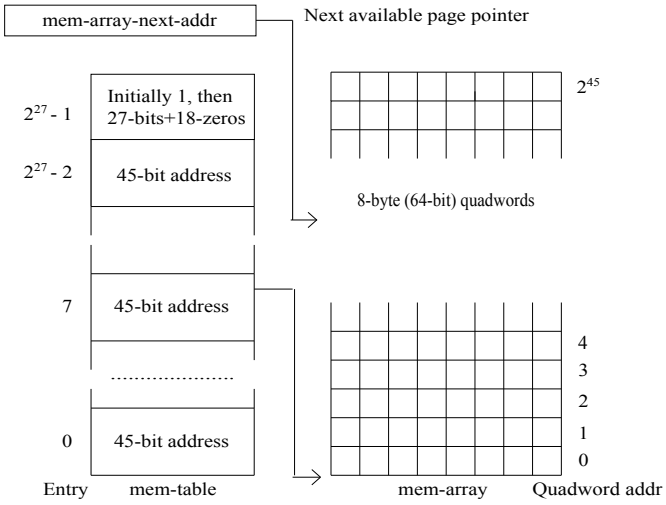

Fig. 1. Memory System

```
(defstobj x86-64
 ; some fields elided
  (mem-table
   :type
   (array (unsigned-byte 45)
          (*mem-table-size*))  ; 2^27
   ;; either 1 or a memory page address
   :initially 1
   :resizable nil)
  (mem-array  ; resizable array of quadwords
   :type (array (unsigned-byte 64)
                (*initial-mem-array-length*))
   :initially 0
   :resizable t)
  (mem-array-next-addr
   :type  ; natural number < 2^45
   (integer 0 35184372088832)
   :initially 0)
)
```

The first field defines `mem-table` as an array of 2^{27} entries where each entry is constrained to be a 45-bit natural number, initially 1. The second field defines a memory array of 2^{45}, unsigned 64-bit integers (quadwords), with its initial entries all being `0`. This array has (an initial length of) `*initial-mem-array-length*` entries (arbitrarily set to $100 * 2^{18}$); but since it is declared resizable, it will be extended automatically as necessary. The third field, a 45-bit integer named `mem-array-next-addr`, tracks the space allocated in `mem-array`.

We initialize `mem-table` values to `1` so we can distinguish which memory table entries are valid. The valid entries in `mem-table` are unique 45-bit addresses that are aligned to two-megabyte boundaries; that is, the bottom (least significant) 18 bits of these addresses are all zero. This choice results in each `mem-table` entry pointing to the start of a two megabyte "page" in `mem-array`. In our implementation, `mem-array` is initially allocated an amount of memory corresponding to a positive integral number of two-megabyte pages. When the demand for memory exceeds the available memory pages, `mem-array` is dynamically extended (until the underlying operating system fails to be able to allocate memory).

Our memory implementation writes to `mem-table` whenever a write is presented for which the corresponding two-megabyte page has no entry in `mem-table`, following a process that can be thought of as a one-level paging scheme. Suppose for example that we start with an empty memory and perform three writes, as shown below.

- **First write to memory: at quadword address** $7 * 2^{18} + 345$. (See Fig. 1.) The corresponding page index into `mem-table` is 7, selected by right shifting the quadword address by 18 bits. Since this is the first write, our memory write function will see that `mem-table` has value 1 at index 7, indicating that the corresponding two-megabyte page has no index in `mem-table`. Index 7 will then obtain the value of `mem-array-next-addr`, $0 * 2^{18}$, which corresponds to the first available "page" in `mem-table`. Also, `mem-array-next-addr` is bumped up to the next page address, $1 * 2^{18}$. An address into `mem-array` is then constructed by combining the page address of $0 * 2^{18}$ with the original low 18 address bits, in this case 345, to obtain $0 * 2^{18} + 345$. We write the given quadword data to that address of `mem-array`.
- **Second write to memory: at quadword address** $23 * 2^{18} + 12$. Following the steps above, we find an invalid entry at index 23, which we replace by the current value of `mem-array-next-addr`, $1 * 2^{18}$. (And, `mem-array-next-addr` is then bumped up by 2^{18}, to $2 * 2^{18}$.) We then write the quadword data into `mem-array` at index $1 * 2^{18} + 12$.
- **Third write to memory: at quadword address** $7 * 2^{18} + 5$. This time we find a valid entry in `mem-table`, namely at index 7 as placed by the first write. So we write the quadword data into `mem-array` at index $0 * 2^{18} + 5$.

In summary, a `mem-table` entry for the top 27 bits of an address serves as the base index for the address where we will write a quadword to `mem-array`. The full index for writing into `mem-array` is the sum (performed by the logical inclusive 'or' function `logior`) of the base index and the bottom 18 bits of the original address. We expect that it would be easy to remove 17 of those 18 bits, leaving just one "valid" bit, and we may do that in the future; but we liked the simplicity of using `logior`.

The same addressing scheme is used when reading, but `mem-array` is never extended on reads. If there is an appropriate `mem-table` entry, then the value returned will be found in `mem-array` using the scheme described above. Otherwise, the default value `0` is returned.

Our primitive memory read and write functions, `memi` and `!memi`, are defined as described above. In particular, note that `!memi` calls a function `add-page` when necessary to extend the available memory and obtain a `mem-array` address from `mem-array-next-addr`.

```
Definition.
(memi i x86-64)
= (let* ((i-top27  ; right shift 18 bits
          (ash i -18))
         (addr (mem-tablei i-top27 x86-64)))
    (if (eql addr 1)  ; page is not present
        *default-mem-value*
      (let ((index (logior addr
                           (logand #x3ffff i))))
        (mem-arrayi index x86-64))))

Definition.
(!memi i v x86-64)
= (let* ((i-top27 (ash i -18))
         (addr (mem-tablei i-top27 x86-64)))
    (mv-let
     (addr x86-64)
     (if (eql addr 1)  ; if page is not present
         (add-page i-top27 x86-64)  ; add a page
       (mv addr x86-64))
     (!mem-arrayi (logior addr (logand #x3ffff i))
                  v
                  x86-64)))

Definition.
(add-page i x86-64)
= (let* ((addr (mem-array-next-addr x86-64))
         (len (mem-array-length x86-64))
         (x86-64
          (if (eql addr len)  ; must resize!
            (resize-mem-array
             (min (* *mem-array-resize-factor* len)
                  *2^45*)
             x86-64)
            x86-64))
         (x86-64  ; Add next new page.
                (!mem-array-next-addr
                 (+ addr *2^18*)
                 x86-64))
         (x86-64 (!mem-tablei i addr x86-64)))
    (mv addr x86-64))
```

The question remains of whether this scheme always works. The next section addresses this question.

IV. Memory invariant and its consequences

In this section we introduce our invariant on the memory. We then sketch how it supports proofs of properties that support efficient execution. Finally, we show how our invariant supports the proof of the key property of our memory model.

A. The memory invariant

Recall our two-level memory, where `mem-table` is an array that contains 2^{18}-aligned addresses indexing into `mem-array`, a resizable array containing 64-bit data, where those addresses are below the (2^{18}-aligned) address limit, `mem-array-next-addr`. Our invariant, stated informally below, incorporates these and other properties. We write `table-max-index` to denote the maximum index into `mem-table`, i.e., one less than the length of `mem-table`, and we write `mem-array-length` to denote the current length of `mem-array`.

1) `mem-array-next-addr` $\leq$ `mem-array-length`.
2) `*initial-mem-array-length*` $\leq$ `mem-array-length`.
3) `#x3ffff` $\&$ `mem-array-length` $=$ 0, i.e., `mem-array-length` is 2^{18}-aligned.
4) `mem-array-next-addr` = $2^{18} * k$, where k is the number of valid entries in `mem-table` (entries not equal to 1).
5) Every valid entry in `mem-table` is 2^{18}-aligned and is less than `mem-array-next-addr`.
6) There are no duplicate valid entries in `mem-table`.
7) The value is 0 in `mem-array` at every index at or exceeding `mem-array-next-addr`.

The function `good-memp` formalizes our memory invariant, as described informally by these seven clauses. The invariant on our stobj is the conjunction of basic structural properties, represented by the stobj recognizer `x86-64p-pre`, and our memory invariant.

```
Definition.
(x86-64p x86-64)
= (and (x86-64p-pre x86-64)
       (good-memp x86-64))
```

The following theorem formalizes invariance for our basic memory write operation. We have also proved such theorems for the higher-level memory write operations presented in Section V.

```
Theorem. x86-64p-!memi
(implies (and (x86-64p x86-64) (n45p i) (n64p v))
         (x86-64p (!memi i v x86-64))))
```

B. Guard verification using the invariant

Section II discussed the role of *guards* in supporting efficient execution. In this section we illustrate the important role played by our invariant for verifying guards, using as a key example our basic memory read function, `memi`.

Recall that `memi` reads the quadword at address `i` from the memory of our `x86-64` stobj. Its guard is given as follows.

```
(and (n45p i)  ; 45-bit quadword address
     (x86-64p x86-64))
```

The interesting case for reading a 45-bit quadword address is that its top 27 bits index into a valid entry of `mem-table`, which is an index into `mem-array`. A corresponding proof obligation arises from guard verification for function `memi`; it states that the corresponding index into `mem-array` is in bounds.

```
(implies
 (and (x86-64p x86-64)
      (< i 2^45)
      (<= 0 i)
      (integerp i)
 ; The next conjunct says that we have a valid
 ; mem-table entry.
      (not (equal (nth (ash i -18)
                       (nth *mem-tablei* x86-64))
                  1)))
 ; We conclude that the index into  mem-array,
 ; as represented by the  logior call below, is
 ; less than the length of  mem-array.
 (< (logior (logand #x3ffff i) ; low 18 bits of i
            (nth (ash i -18)   ; top 27 bits of i
                 (nth *mem-tablei* x86-64)))
    (len (nth *mem-arrayi* x86-64))))
```

In order for this formula to be a theorem, the hypothesis `(x86-64p x86-64)` must be sufficiently strong. We have checked mechanically with ACL2 that this is indeed the case.

C. A fundamental read-over-write lemma

Recall the key property `memi-!memi` from the start of Section III, which characterizes the effect of a quadword write on the memory. It is crucial for proving analogous properties of higher-level read and write functions, as discussed in Section V. That property naturally breaks into two lemmas. One of those is the following, for the case that the address for reading is the same as the address that is written.

```
Theorem. memi-!memi-same
(implies (x86-64p x86-64)
         (equal (memi i (!memi i v x86-64))
                v))
```

With suitable hints and lemmas, ACL2 proves this theorem. But among the lemmas applied in its proof, as reported by the prover, the following is one that is critical, as the proof fails without it. Note that it corresponds to Clause 4 of our invariant.

```
Theorem. logand-mem-array-next-addr
(implies (good-memp x86-64)
         (equal (logand #x3ffff
                        (nth *mem-array-next-addr*
                             x86-64))
                0))
```

We now consider the other case, that is, where the addresses are distinct.

```
Theorem. memi-!memi-different
(implies (and (not (equal i j))
              (n45p i)
              (n45p j)
              (x86-64p x86-64))
         (equal (memi i (!memi j v x86-64))
                (memi i x86-64)))
```

Consider what happens if two mem-table indices contain the same value. For example, suppose that quadword addresses `i` and `j` are 0 and 2^{18}, respectively, yet the corresponding mem-table entries at indices 0 and 1 both have value 0. Also suppose that all values stored in `mem-array` are 0, and that the value `v` is 1. Then, even though `i` and `j` are distinct, the equality displayed above is false, as `(memi i (!memi j v x86-64))` is 1 yet `(memi i x86-64)` is 0.

It is here that the memory invariant saves us. Specifically, Clause 6 prohibits duplicate entries in `mem-table`. We prove that every operation on the memory preserves the memory invariant.

V. Using the memory model: reads and writes

We have seen functions `memi` and `!memi` for reading and writing quadwords (8-byte natural numbers) at quadword-aligned memory addresses. But for our intended application of modeling the x86 ISA [3], we also require functions that read and write bytes, words, doublewords, and quadwords at arbitrary addresses. We tour those below and then discuss theorems relating reads and writes.

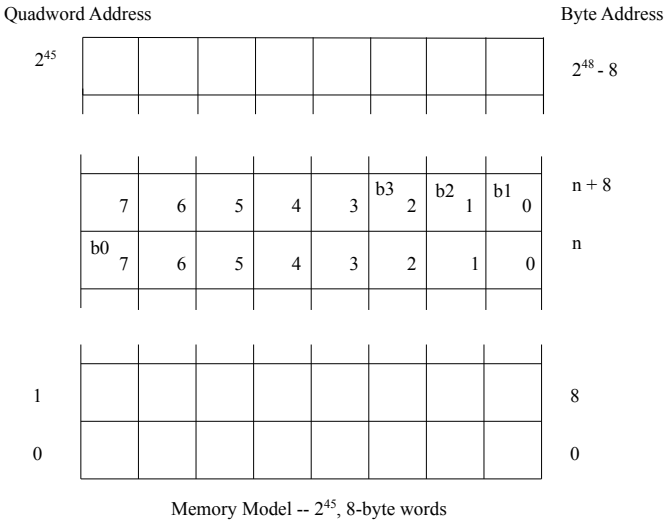

Fig. 2. Misaligned memory access

For byte reads the memory is read once. But for non-aligned accesses of more than one byte, it may be necessary to read the memory twice because the access may be split across two (64-bit) quadwords. This possibility is illustrated in Fig. 2 for a doubleword (four bytes: b0, b1, b2, b3) stored at address `n+7`. Here, `n` is the address of a byte on a quadword boundary (`n` is a multiple of 8). We use little-endian format, which requires that the least significant byte appear at address `n+7`, the next byte at `n+8`, the next byte at `n+9`, and the most significant byte at `n+10`.

A. Read and write functions

We implement byte, word, doubleword, and quadword read and write operations using the primitive quadword memory-read and memory-write functions, `memi` and `!memi`. The following function reads a single byte from memory.

```
Definition.
(rm08 addr x86-64)
= (let* ((byte-num     (n03 addr))
         (qword-addr   (ash addr -3))
         (qword        (memi qword-addr x86-64))
         (shift-amount (ash byte-num 3))
         (shifted-qword (ash qword
                             (- shift-amount))))
    (n08 shifted-qword))
Guard:[1]
(and (n48p addr)
     (x86-64p x86-64))
```

The following lemma is critical in order to verify guards for the above function. Specifically, it is used in the proof of the guard obligation from the definition of `rm08` for the call `(ash qword (- shift-amount))`, which states that `qword` is an integer, where `qword` is `(memi qword-addr x86-64)`.

[1]We show a formula that is logically equivalent to the guard.

```
Theorem. memi-is-unsigned-byte-64
(implies (and (x86-64p x86-64)
              (n45p addr))
         (n64p (memi addr x86-64)))
```

But why does this lemma hold? Clause 5 of our invariant (Section IV-A) is crucial, and is a consequence of hypothesis `(x86-64p x86-64)`:

> Every valid entry in `mem-table` is 2^{18}-aligned and is less than `mem-array-next-addr` (Fig.**??**).

Indeed, if we tell ACL2 to ignore ("disable") two rewrite rules corresponding to this property, the proof fails.

We turn now from reading to writing a byte.

```
Definition.
(wm08 addr byte x86-64)
= (let* ((byte-num       (n03 addr))
         (qword-addr     (ash addr -3))
         (qword          (memi qword-addr x86-64))
         (shift-amount   (ash byte-num 3))
         (byte-mask      (ash #xff shift-amount))
         (qword-masked   (logand (lognot byte-mask)
                                 qword))
         (byte-to-write  (ash byte shift-amount))
         (qword-to-write (logior qword-masked
                                 byte-to-write)))
    (!memi qword-addr qword-to-write x86-64))
```

It will be important to maintain our invariant after doing a write, so that the guards (which include our invariant) are met for subsequent memory operations. We therefore prove the following lemma.

```
Theorem. x86-64p-wm08
(implies (and (x86-64p x86-64)
              (n48p addr)
              (n08p byte))
         (x86-64p (wm08 addr byte x86-64)))
```

Reads and writes of more than one byte are built up in layers. For example, here is the function for reading four bytes, which invokes the two-byte read function when the addresses cross a quadword boundary. Notice that the call of `n48p` in the guard leaves room to read four bytes.

```
Definition.
(rm32 addr x86-64)
= (let ((byte-num (n03 addr)))
    (cond
     ((<= byte-num 4)
      (let* ((qword-addr   (ash addr -3))
             (qword        (memi qword-addr x86-64))
             (shift-amount (ash byte-num 3))
             (shifted-qword (ash qword
                                 (- shift-amount))))
        (n32 shifted-qword)))
     (t  ; byte-num is 5, 6, or 7
      (let* ((word0 (rm16 addr x86-64))
             (word1 (rm16 (n48+! 2 addr) x86-64)))
        (logior (ash word1 16) word0)))))
```

B. Read-over-write theorems

The following theorem characterizes the effect of reading a byte from address `i` after writing a byte, `v`, at address `j`. The result, of course, is `v` if `i` equals `j`; otherwise the write does not affect the value returned by the read. The proof relies on the lemma `memi-!memi`, which takes advantage of the invariant, `(x86-64p x86-64)`; see Section IV-C.

```
Theorem. rm08-wm08
(implies (and (x86-64p x86-64)
              (n48p i) (n48p j) (n08p v))
         (equal (rm08 i (wm08 j v x86-64))
                (if (equal i j)
                    v
                  (rm08 i x86-64))))
```

The corresponding lemma for two bytes is a bit more complex, as the cases for the resulting read depend on how the two address regions overlap.

```
Theorem. rm16-wm16
(implies
 (and (x86-64p x86-64)
      (natp i) (n48p (1+ i))
      (natp j) (n48p (1+ j))
      (n16p v))
 (equal (rm16 i (wm16 j v x86-64))
        (cond ((equal i j)
               v)
              ((equal j (1+ i))
               (logior (* *2^8* (logand #xff v))
                       (rm08 i x86-64)))
              ((equal i (1+ j))
               (logior (ash (logand #xff00 v) -8)
                       (* *2^8*
                          (rm08 (+ 1 i) x86-64))))
              (t
               (rm16 i x86-64)))))
```

Our approach to proving this theorem is to reduce it to the preceding theorem for single-byte reads and writes. Thus, we characterize two-byte reads in terms of single-byte reads, and similarly for writes. Here is the relevant lemma for writes.

```
Theorem. wm16-as-wm08
(implies
 (and (x86-64p x86-64)
      (natp addr)
      (n48p (1+ addr))
      (n16p word))
 (equal (wm16 addr word x86-64)
        (let* ((x86-64
                (wm08 addr
                      (logand word #xff)
                      x86-64))
               (x86-64
                (wm08 (+ 1 addr)
                      (ash (logand word #xff00) -8)
                      x86-64)))
          x86-64)))
```

Recall that `wm08` is defined in terms of the primitive quadword write operation, `!memi`. Since the proof of the lemma above involves reasoning about successive writes, it is not surprising that the following is critical for its proof.

```
Theorem. !memi-!memi-same
(implies (x86-64p x86-64)
         (equal (!memi addr v1
                       (!memi addr v2 x86-64))
                (!memi addr v1 x86-64)))
```

VI. Efficient execution

We have fabricated some memory-copy tests to get an idea of the performance of our memory implementation. The Lisp runs reported below used a 3.5 GHz Intel Xeon processor.

```
(defun copy (from to count x86-64)
  (declare (type (unsigned-byte 29) count)
           (type (unsigned-byte 45) from to)
           (xargs :guard
                  (and (< (+ from count) *2^45*)
                       (< (+ to count) *2^45*)
                       (x86-64p x86-64))
                  :stobjs (x86-64)))
  (if (zpf count)
      x86-64
    (let* ((value (memi from x86-64))
           (x86-64 (!memi to value x86-64)))
      (copy (1+ from) (1+ to) (1- count) x86-64))))
```

Function `copy-test`, called below, writes a 1 at each address below its first argument, `addr`, and then calls `(copy 0 addr addr x86-64)`. Note that the first two runs copy 1 GB (either 128 quadwords copied 1M times or 128K quadwords copied 1K times), while the third copies 1 GB (128M quadwords) ten times, to amortize the memory initialization with ones.

```
(time$  ; 2.9 seconds
 (copy-test 128              (* 1024 1024) x86-64))

(time$  ; 2.8 seconds
 (copy-test (* 128 1024)     1024          x86-64))

(time$  ; 29.9 seconds (for 10x memory ops)
 (copy-test (* 128 1024 1024) 10           x86-64))
```

The copying of approximately 350M bytes/second corresponds to 700M memory byte accesses per second, which we find encouraging. However, this is slower by about a factor of 9 than the analogous three runs of a corresponding C program compiled with `gcc -O3`, with times of 0.330 seconds, 0.320 seconds, and 3.260 seconds, respectively.

VII. Conclusion

Our formal memory model has been proven to provide the illusion of a complete 2^{48}-byte memory. Our implementation provides time- and space-efficient, constant-time memory read/write operations, thus supporting validation of ISA simulators. We represent our large memory by using an expandable collection of pages indexed by a table of page pointers, so that we can provide the illusion of having a memory containing 2^{48} bytes. We have assured that our memory model is correct throughout the entire address range by proving requisite properties of our model. This kind of modeling is important for large-scale memory systems as they cannot be practically built nor tested for all manner of configurations.

Our memory model permits 350M bytes per second to be copied from one part of memory to another part, which we believe exceeds the performance of all other theorem-prover-based memory models for such large address spaces. When used within our evolving x86 microprocessor ISA specification, our model provides sufficient performance to allow binary code to be executed at more than 500,000 instructions per second [3]. We expect to use this or a similar memory model as we move forward with our microprocessor modeling efforts [16].

Acknowledgments

This material is based upon work supported by DARPA under contract number N66001-10-2-4087 and by ForrestHunt, Inc. We thank Jared Davis, Shilpi Goel, Sandip Ray, Anna Slobodova, and Sol Swords for useful feedback on drafts of this paper. We also thank Shilpi Goel for drawing the figures and David Hardin for providing performance data on an AAMP7 memory model. Finally, we thank the referees for providing useful feedback on our submission.

References

[1] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Boston, MA: Kluwer Academic Publishers, Jun. 2000.

[2] M. Kaufmann and J. S. Moore, "ACL2 home page," see URL http://www.cs.utexas.edu/users/moore/acl2.

[3] M. Kaufmann and W. A. Hunt, Jr., "Towards a formal model of the x86 ISA," Department of Computer Sciences, University of Texas at Austin, Tech. Rep. TR-12-07, May 2012.

[4] W. A. Hunt, Jr., *FM8501: A Verified Microprocessor*, ser. LNAI. Springer-Verlag, 1994, vol. 795.

[5] ——, "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, pp. 429–460, 1989.

[6] W. A. Hunt, Jr. and B. Brock, "A Formal HDL and Its Use in the FM9001 Verification," in *Mechanized Reasoning and Hardware Design*, ser. Prentice-Hall International Series in Computer Science, C. A. R. Hoare and M. J. C. Gordon, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1992, pp. 35–48.

[7] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *ITP 2010*, ser. LNCS, no. 6172. Springer, 2010.

[8] J. C. Davis, "Memories: Array-like records for ACL2," in *Proceeding of the* 6th International Workshop on the ACL2 Theorem Prover and its Applications, no. ISBN: 0-9788493-0-2. ACM, 2006.

[9] D. Hardin, E. W. Smith, and W. D. Young, "A Robust Machine Code Proof Framework for Highly Secure Applications," in *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, P. Manolios and M. Wilding, Eds. ACM, Jul. 2006, pp. 11–20.

[10] M. Kaufmann and J. S. Moore, "Structured Theory Development for a Mechanized Logic," *Journal of Automated Reasoning*, vol. 26, no. 2, pp. 161–203, 2001.

[11] ——, "A Precise Description of the ACL2 Logic," 1997, see URL http://www.cs.utexas.edu/users/moore/publications/km97.ps.gz.

[12] ——, "ACL2 documentation," see URL http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html.

[13] R. S. Boyer and J. S. Moore, "Single-threaded Objects in ACL2," in *Practical Aspects of Declarative Languages (PADL)*, ser. LNCS, S. Krishnamurthy and C. R. Ramakrishnan, Eds., vol. 2257. Springer-Verlag, 2002, pp. 9–27.

[14] S. Swords and J. Davis, "Bit-blasting ACL2 theorems," in *Proceeding 10th International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, D. Hardin and J. Schmaltz, Eds., vol. 70, 2011, pp. 84–102.

[15] R. S. Boyer and W. A. Hunt, Jr., "Function Memoization and Unique Object Representation for ACL2 Functions," in *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*. ACM, Aug. 2006, pp. 81–89.

[16] S. Goel, W. A. Hunt, Jr., and M. Kaufmann, "Towards an ACL2 model of the x86 ISA," in preparation.