A Formal Model of a Large Memory that Supports Efficient Execution

Warren A. Hunt, Jr. and Matt Kaufmann

Presented at FMCAD, October, 2012

Computer Science Department
University of Texas
1 University Way, M/S C0500
Austin, TX 78712-0233

{hunt, kaufmann}@cs.utexas.edu

Outline

Introduction

To enable the modeling and analysis of industrial-sized systems:

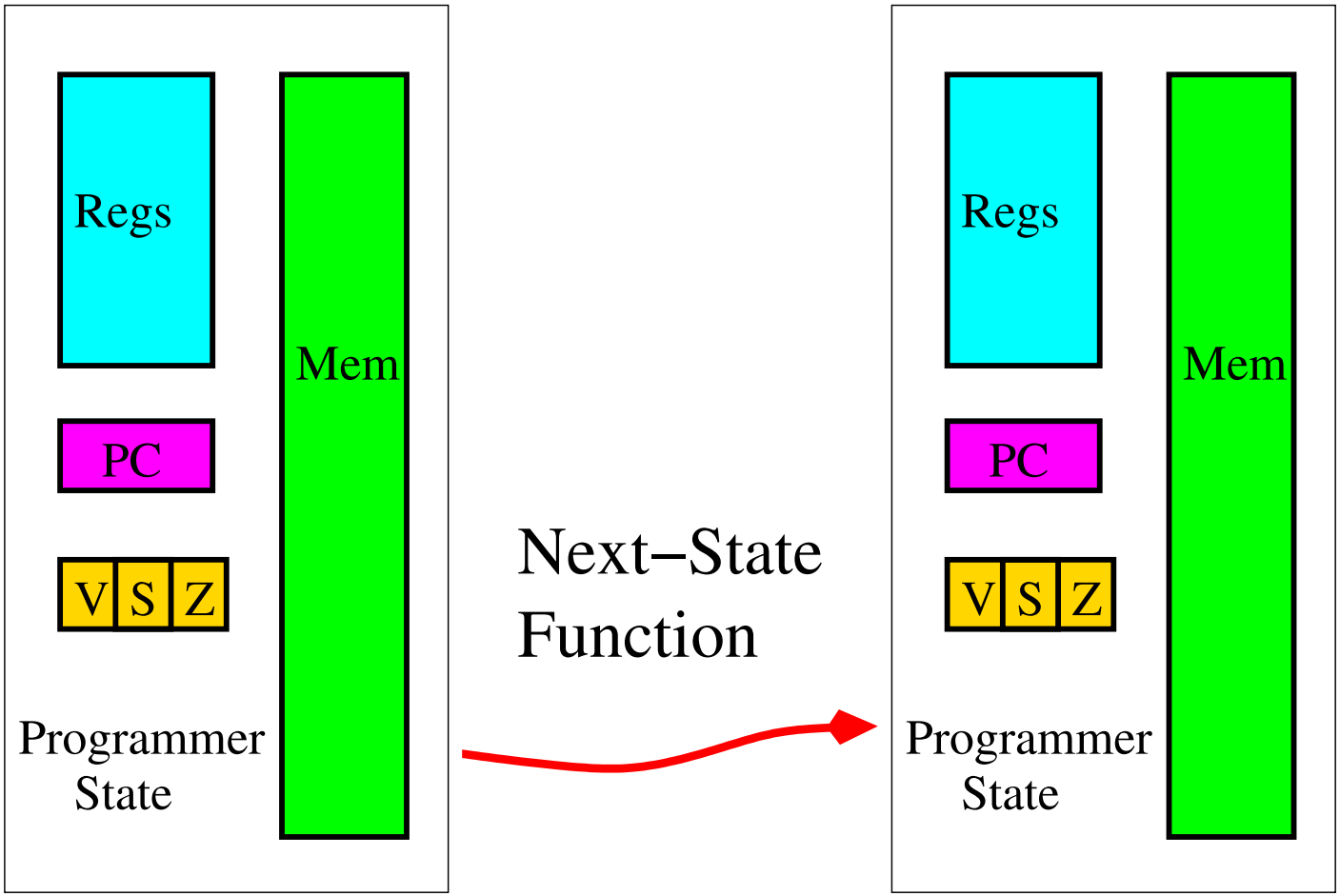
- We are developing ISA models suitable for code analysis.
- We have defined a 2^{48}-byte (281 TB) memory model.
- We have proved key properties of the model.

We discuss our approach for modeling large *physical* memories suitable for ISA modeling and analysis.

Motivation: Create a formally-correct memory model that is *efficient* to execute for a large number of memory writes. Why? Efficient co-simulation! (More later...)

Memory Model Rationale

Our memory model supports our ISA-specification efforts.
Example: X86. BUT NOTE: This talk is about a memory model!



Core Technology: ACL2

- First-order logic with induction and recursion
  - Atomic data objects: numbers, characters, strings, symbols
  - Constructor, `CONS`, for pairs and lists
  - Sophisticated quotation and abbreviation mechanisms
  - Functions (31 primitive functions, 200+ defined functions)
- Efficient execution – models are often validated by co-simulation
  - *Guards* (preconditions) enable use of Common Lisp for execution.
  - We use *STOBJs* (single-threaded objects) for constant-time array read/write operations with applicative semantics.
- Critical feature is the overall capacity of the ACL2 system (proofs, execution, output, database management, documentation, ...)
- In use at AMD, Centaur (Nano), IBM, and Rockwell-Collins
- See `http://www.cs.utexas.edu/users/moore/acl2/` for more info (download, documentation, tutorials, applications, ...).

The ACL2 Theorem Prover

Associated with the ACL2 Logic, is the ACL2 theorem prover.

- Rewriter-based theorem-prover
- Simplifier includes:
  - Clausification
  - Rewriting
  - Linear arithmetic solver
  - Term "type" analysis
- Other processes include destructor elimination, generalization, induction
- "Proof checker" for goal-directed reasoning (as for tactic-based proof assistants)
- Clause processors
  - Extensions to the ACL2 proving process
  - Can be verified or "trusted" (for external tools)
  - ACL2(h) example: Symbolic simulation using BDDs and AIGs

Underlying Memory Model

We have developed a 2^{48}-byte (2^{45}-quadword) memory model.

Our effort is focused on memory models that are:

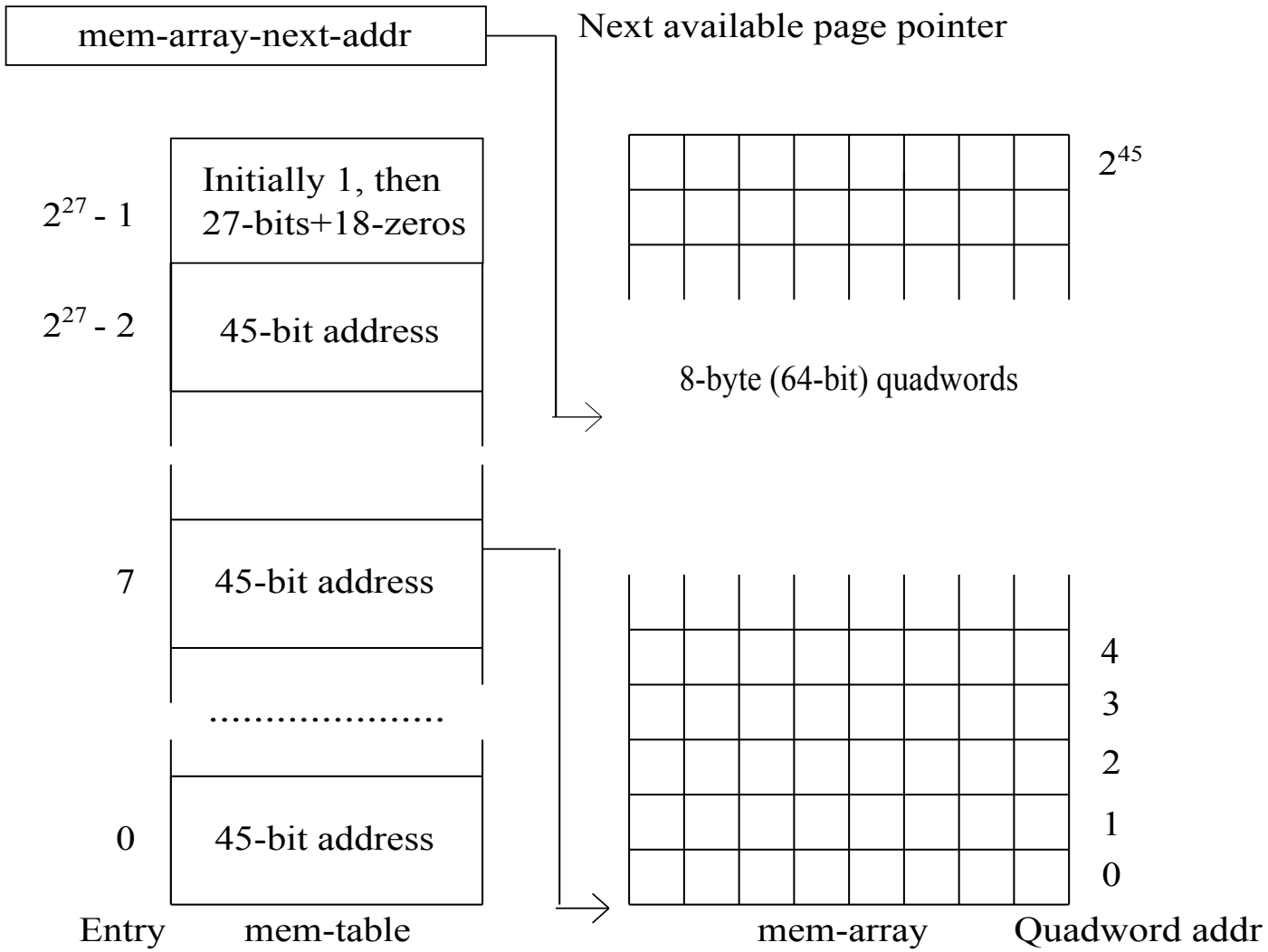
- defined formally,
- scale up to very large memories
- provide high-speed simulation, and
- support mechanized reasoning.

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
```

Memory Model Organization

We use three fields of an ACL2 STOBJ to define our memory model.



Memory Model Invariant

Our two-level memory contains aligned addresses indexing into `mem-array`, a resizable array containing 64-bit data, where those addresses are below the (aligned) address limit, `mem-array-next-addr`.

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 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 aligned and is less than `mem-array-next-addr`.
6. The value is 0 in `mem-array` at every index at or exceeding `mem-array-next-addr`.
7. There are no duplicate valid entries in `mem-table`.

Memory Model Invariant, continued

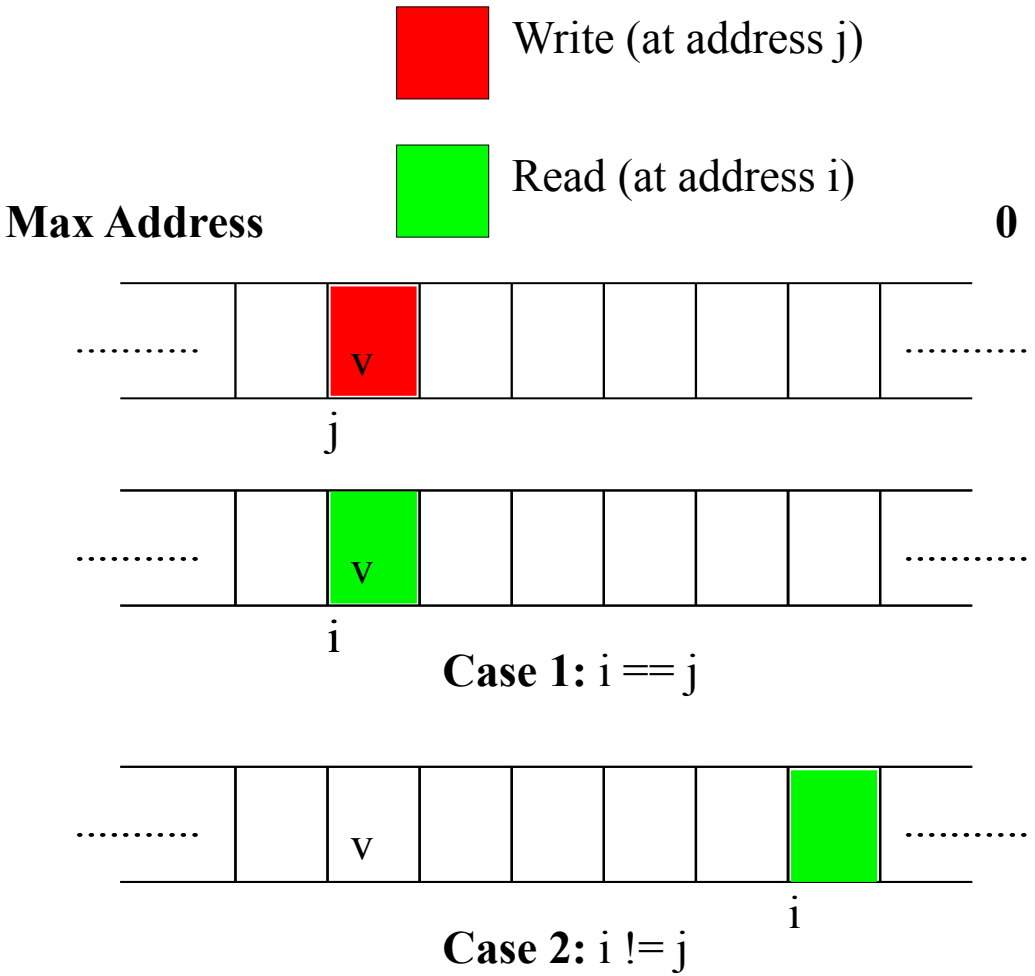
The function good-memp formalizes our memory invariant, as described informally by the previous seven clauses.
(Note: We replaced x86-64 by st for readability.)

```
Definition.
(stp st)
= (and (stp-pre st)  ; E.g., rip is a 64-bit natural
       (good-memp st))
```

The following theorem formalizes the preservation of our invariant by our basic memory write operation, !memi, which updates the three STOBJ fields.

```
Theorem. stp-!memi
(implies (and (stp st)
              (n45p i)    ; quadword address (45-bit natural)
              (n64p v))   ; 64-bit natural
         (stp (!memi i v st))))
```

Memory Read-Over-Write Theorem Diagram



Memory Model Based on 64-bit Words

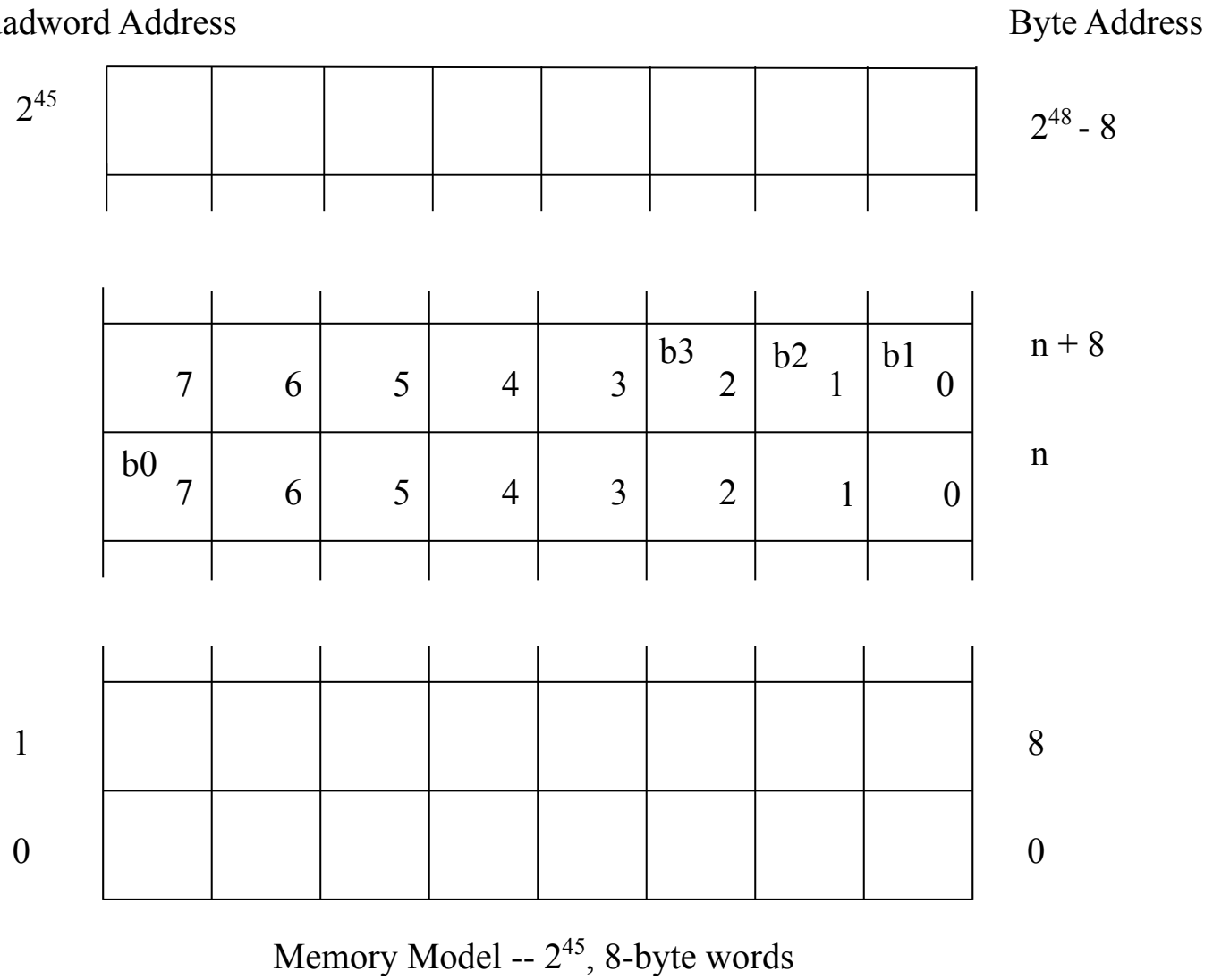
Our memory model is based on an array of 64-bit quadwords, providing (the illusion of a) memory containing 2^{48} bytes.[1]

```
Theorem. memi-!memi
(implies
  (and (stp st)                    ; 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
                      st))         ; Initial memory
         (if (equal i j)           ; For equal addresses
             v                     ; the read value is v
           (memi i st)))))         ; else, unchanged
```

[1] Recent work eliminates the hypotheses of this theorem.

Memory Operations for Various Widths

We provide byte, two-byte, four-byte, and eight-byte memory operations.



Memory Model -- 2^{45}, 8-byte words

Byte Memory Operations

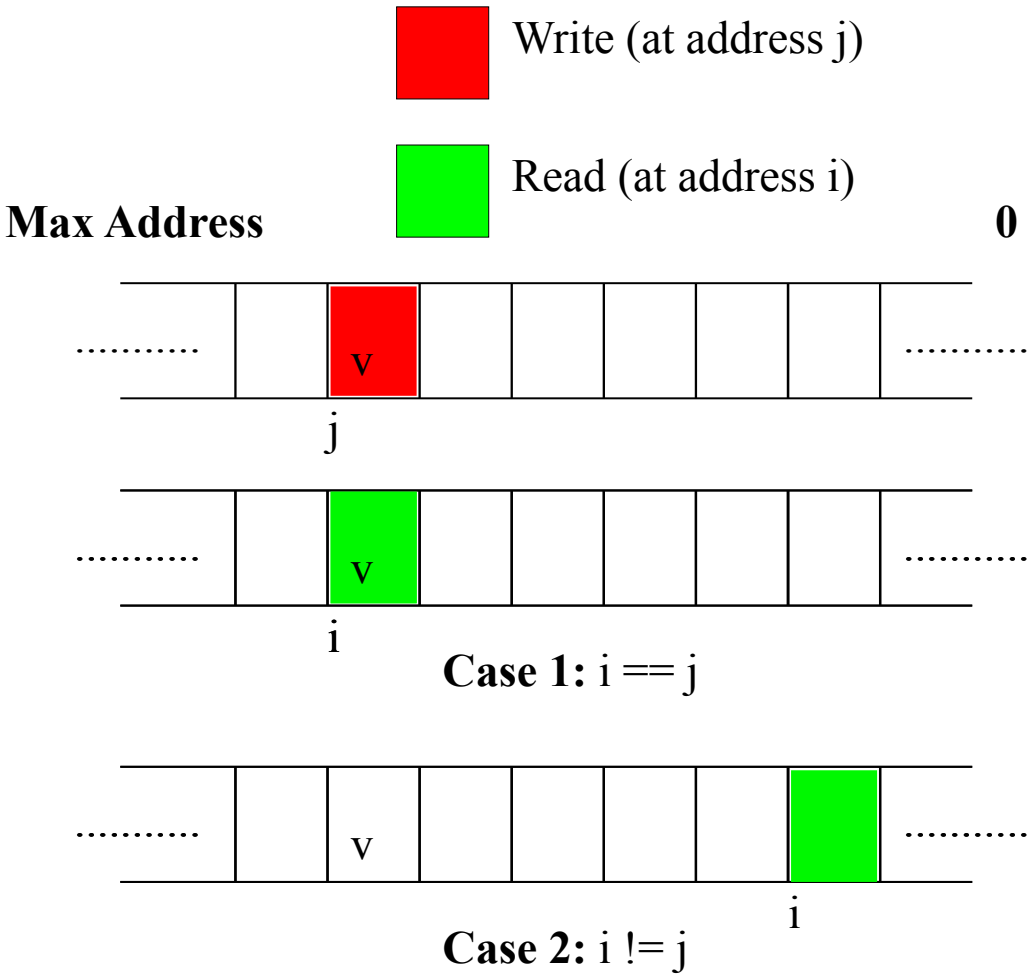
```
Definition.
(rm08 addr st)  ; grab a quadword, and then a byte from it
= (let* ((byte-num      (n03 addr))
         (qword-addr    (ash addr -3))
         (qword         (memi qword-addr st))
         (shift-amount  (ash byte-num 3))
         (shifted-qword (ash qword (- shift-amount))))
    (n08 shifted-qword))
Definition.
(wm08 addr byte st)  ; write appropriate byte into a quadword
= (let* ((byte-num      (n03 addr))
         (qword-addr    (ash addr -3))
         (qword         (memi qword-addr st))
         (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 st))
```

Four-Byte Memory Read Operation

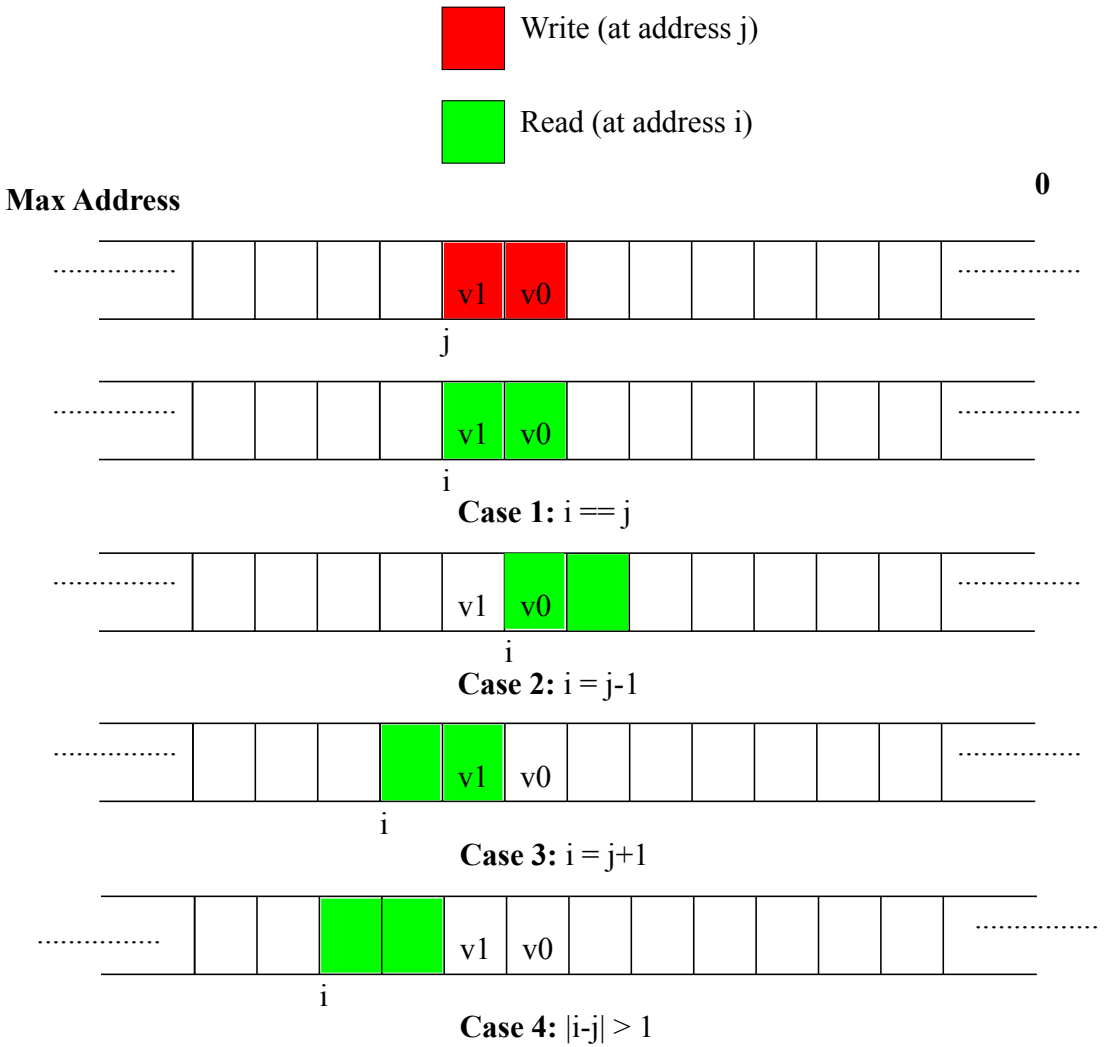
```
Definition.
(rm32 addr st)
= (let ((byte-num (n03 addr)))
    (cond
     ((<= byte-num 4)  ; E.g., if 4 then read bytes 4,5,6,7
      (let* ((qword-addr    (ash addr -3))
             (qword         (memi qword-addr st))
             (shift-amount (ash byte-num 3))
             (shifted-qword (ash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

Note that for two-or-more-byte memory operations, we may need to access two words of the underlying, 64-bit memory.

Memory Read-Over-Write Theorem Diagram: 8 bits



Memory Read-Over-Write Theorem Diagram: 16 bits



Memory Read-Over-Write Theorems

```
Theorem. rm16-wm16
(implies
 (and (stp st)
      (natp i) (n48p (1+ i))
      (natp j) (n48p (1+ j))
      (n16p v))
 (equal
  (rm16 i (wm16 j v st))
  (cond ((equal i j)
         v)
        ((equal j (1+ i))
         (logior (* *2^8* (logand #x00ff v))
                 (rm08 i st)))
        ((equal i (1+ j))
         (logior (ash (logand #xff00 v) -8)
                 (* *2^8*
                    (rm08 (+ 1 i) st))))
        (t
         (rm16 i st)))))
```

```
Theorem. rm08-wm08
(implies
 (and (stp st)
      (n48p i) (n48p j) (n08p v))
 (equal (rm08 i (wm08 j v st))
        (if (equal i j)
            v
          (rm08 i st)))))

                                ; write is +1 from read

                                ; write is -1 from read

                                ; Otherwise, no overlap,
                                ; read unaffected by write
```

Memory Model Performance

Our verified 2^{48}-byte memory implementation allows rapid simulation.

```
(defun copy (from to count st)
  (declare (xargs :guard (and (< (+ from count) *2^45*)
                              (< (+ to count) *2^45*)
                              (stp st))
                  :stobjs (st)))
  (if (zpf count)
      st
    (let* ((value  (memi from st))
           (st (!memi to value st)))
      (copy (1+ from) (1+ to) (1- count) st))))
```

When copying one GByte of data, we can move about 400 MBytes/sec – this is about 15% of what we observe with structurally-similar C code.

Ongoing Work

Extend the ACL2 system ("abstract stobjs"):

- Allow STOBJ (array) semantics to include associative lookup.
- Enable our symbolic simulation to include STOBJs.
- Avoid expensive invariant checks (done as guard checks).
- Eliminate hypotheses in read-over-write theorems.

Extend our memory model:

- Defined and verified properties of a 2^{52}-byte memory model
- Developing co-simulation environment for model validation
- Modeling processor segment and paging mechanisms

Conclusion

We continue to expand our hardware and software modeling and analysis capabilities.

- We have developed a 64-bit data and 48-bit address memory model.
- We have verified memory operation properties (read-over-write theorems).
- Our memory model provides a foundation for our processor modeling, and supports:
  - processor model validation by co-simulation; and
  - code proofs.

We perform all of our work in an environment where we can prove or disprove theorems about our models.