Symbolic Simulation of x86-64 Instructions Using Congruence-Based Rewriting

A **Failed** but **Interesting** Experiment on the `x86isa` Model
Useful

Shilpi Goel

ACL2 Seminar

March, 2016

This is a Weird Talk

I'm going to talk about a plan for symbolic simulation on the `x86isa`

- that didn't work
- how it would have worked, had it worked
- why it didn't work
- why it is still good to remember this plan

I don't have a success story to convince you that this plan is not useless.
Instead, I'll try to show its worth by describing an experiment that failed.

What I'll Cover in This Talk

- Classical approach that employs *equality-based rewriting* to perform symbolic simulation using interpreter-based models in ACL2
- Why I wanted a different approach for the `x86isa`
  - Overview of the x86 paging system
- An alternative approach that employs *congruence-based rewriting*
  - Overview of congruence-based reasoning in ACL2
- Why this alternative approach failed for the `x86isa`
  - But this approach is a solution for problems like...

Quick Overview: Interpreter-Based Model in ACL2

An interpreter-based model typically has the following main components:

1. **State**
   - Registers, Memory, Flags, etc.
2. **Instruction Semantic Functions**
   - `semantic-fn(x86)` → `x86'`
   - Specification of each instruction (ADD, SUB, etc.)
3. **Step Function**
   - `step(x86)` → `x86'`
   - Fetches, decodes, and executes one instruction
4. **Run Function**
   - `run(n,x86)` → `x86'`
   - Calls `step` `n` times or till an error occurs, whichever comes first

Equality-Based Rewriting for Symbolic Simulation

```
(defthm step-opener
  (implies <hyps>
           (equal (step x86)
                  (top-level-opcode-execute pc ... x86))))
```

```
(defthm run-opener-no-error
  (implies (and (not (ms x86)) (not (zp n)))
           (equal (run n x86)
                  (run (1- n) (step x86)))))
```

```
(defthm run-opener-error-or-end
  (implies (or (ms x86) (zp n))
           (equal (run n x86) x86)))
```

Example: Symbolic Simulation of CLC instruction

```
(run 1 x86)
;; Using run-opener-no-error
=
(run 0 (step x86))
;; Using step-opener
=
(run 0 (top-level-opcode-execute pc ... x86))
;; Opening up top-level-opcode-execute
=
(run 0 (clc-semantic-fn pc ... x86))
;; Opening up clc-semantic-fn
=
(run 0 (!rip (1+ pc) (!cf 0 x86)))
;; Using run-opener-error-or-end
=
(!rip (1+ pc) (!cf 0 x86))
```

Example: Symbolic Simulation of CLC instruction

```
(run 1 x86)
=
(!rip (1+ pc) (!cf 0 x86))
```

We usually reason about *projections* from this symbolic expression.

```
<preconditions> ⇒ (read *pc* (run 1 x86)) = (1+ pc)
```

`x86isa`: Modes of Operation

1. **Programmer-level Mode:** provides the same interface to the x86 state as is provided by an OS to application programs
2. **System-level Mode:** provides the same interface to programs as is provided by the processor

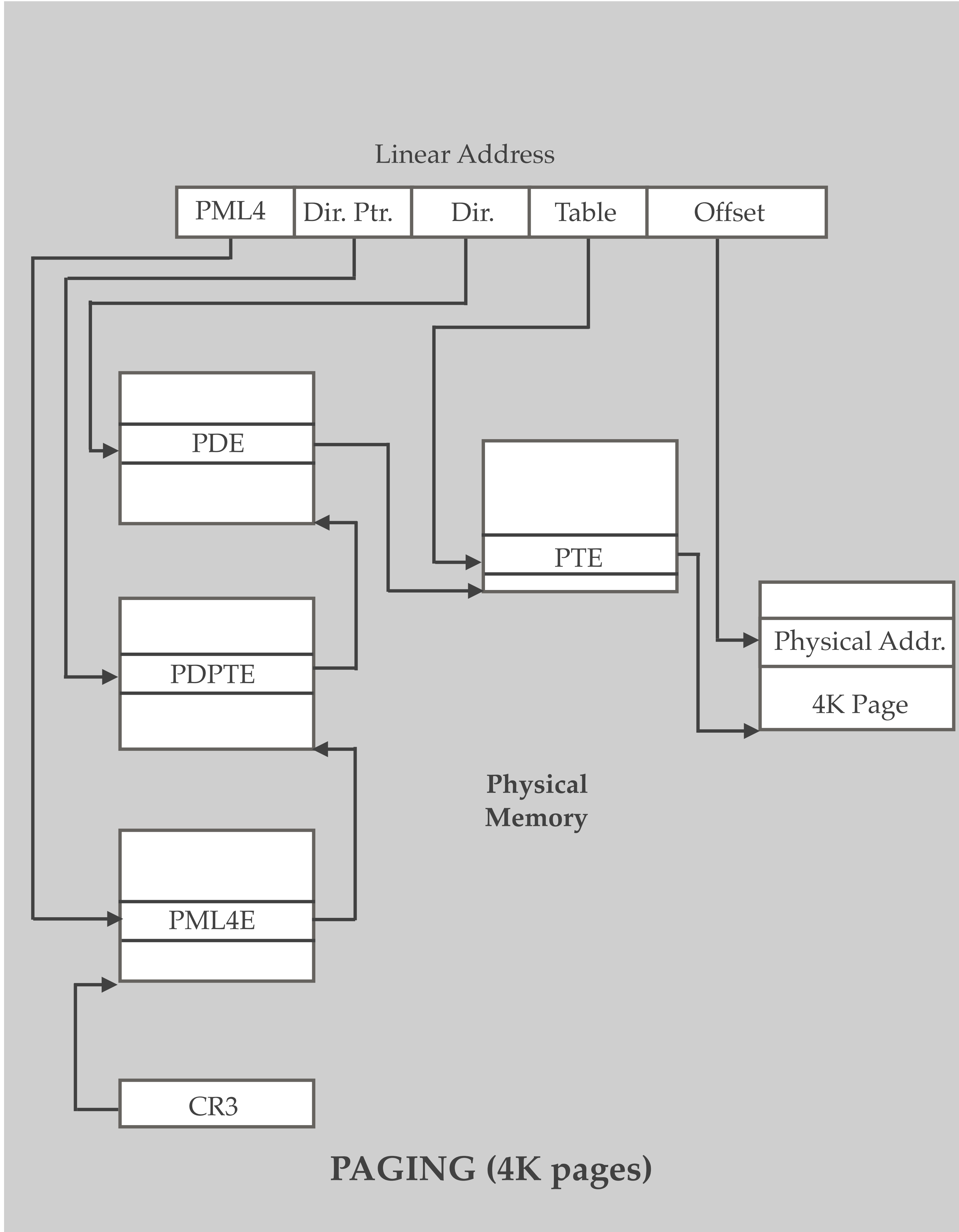
For this talk, the main difference between these modes is the view of the memory.

Programmer-level mode traffics in ***linear memory.***

- Address translation is not a part of the model.

System-level mode traffics in ***physical memory.***

- Address translation (**paging**) is a part of the model.


Linear Address
PML4
Dir. Ptr.
Dir.
Table
Offset
PDE
PTE
Physical Addr.
4K Page
PDPTE
Physical
Memory
PML4E
CR3
PAGING (4K pages)

Programmer-Level Mode: CLC instruction

```
(run 1 x86)
...
=
(!rip (1+ pc) (!cf 0 x86))
```

```
= (mv-nth 2 (rm08 pc x86))
```

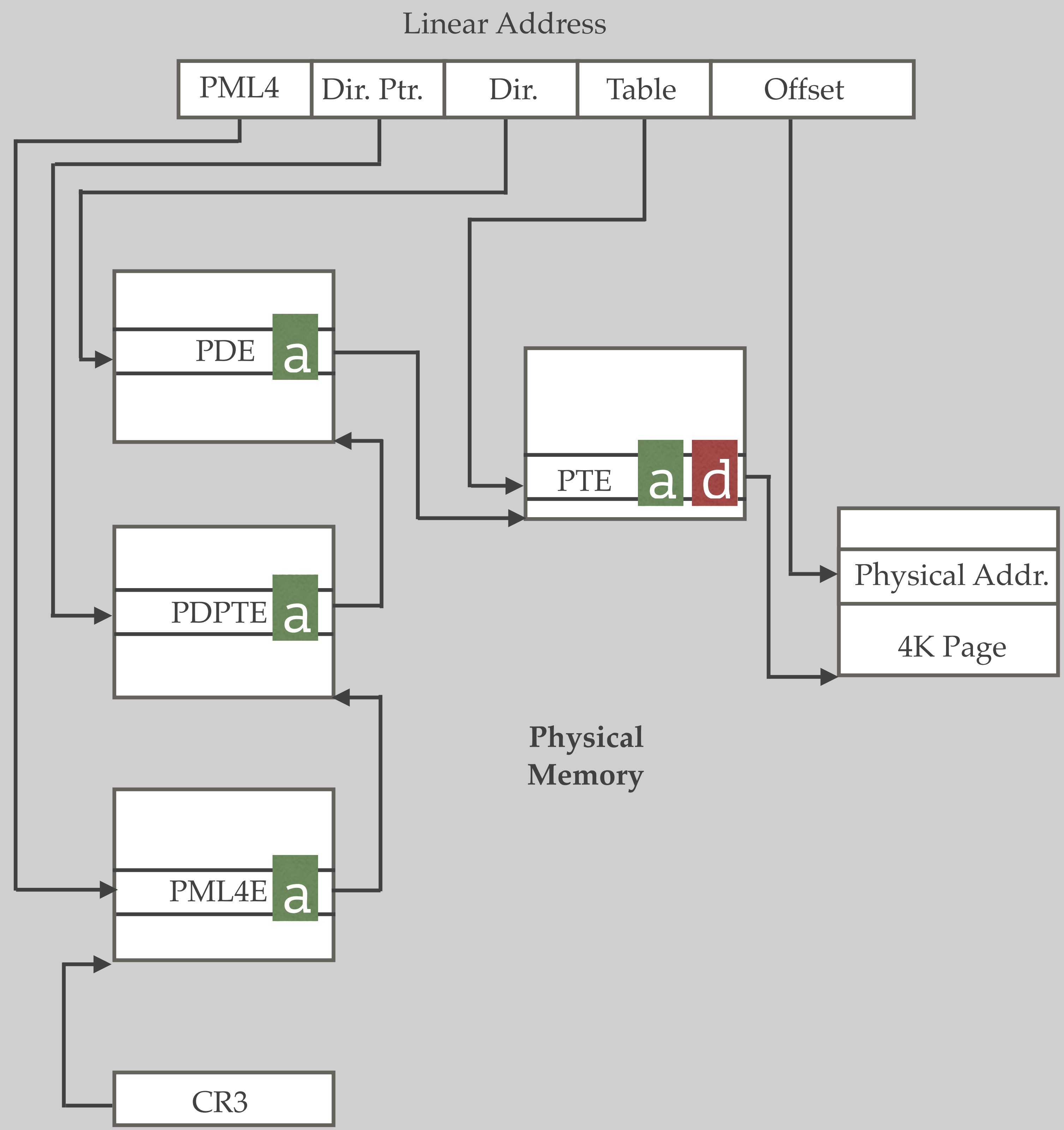
Reads from the memory do not modify the x86 state.

System-Level Mode: CLC instruction

```
(run 1 x86)
...
=
(!rip (1+ pc) (!cf 0 (mv-nth 2 (rm08 pc x86))))

                                ≠ x86
```

Every memory access modifies the x86 state.



PAGING (4K pages)

Accessed and Dirty Bits

- Address translation is done by traversing the paging data structures, which produces on-the-fly updates — A & D bits.
- For programs other than those that swap pages to/from the physical memory, writes to these bits are just “side effects”.
  - A & D bits do not affect the address translation.
- I don’t want to see x86 symbolic expressions cluttered with writes that don’t affect a program’s execution.
  - Having these writes hanging around slows down rewriting too.

`step` Opener Lemma in System-Level Mode

`step`: reads an instruction from the memory, decodes it, and dispatches control to the appropriate instruction semantic function.

```
(defthm step-opener-in-programmer-level-mode
  (implies (and (programmer-level-mode x86)
                (other-hyps))
           (equal (step x86)
                  (top-level-opcode-execute pc ... x86))))
```

```
(defthm step-opener-in-system-level-mode
 (implies (and (not (programmer-level-mode x86))
               (other-hyps))
          (xlate-equiv (step x86)
                       (top-level-opcode-execute pc ... x86))))
```

`xlate-equiv`

Two x86 states are **translate-equivalent x86 states** or `xlate-equiv` if:

1. the paging structures contained in the memory of the two states must be equal, modulo the accessed and dirty bits.

2. all other components, including the rest of the memory, of the two states must be exactly equal.

`step` Opener Lemma in System-Level Mode

```
(defthm step-opener-in-system-level-mode
 (implies (and (not (programmer-level-mode x86))
               (other-hyps))
          (xlate-equiv (step x86)
                       (top-level-opcode-execute pc ... x86))))
```

But on its own, the above lemma doesn't do us much good; ACL2 replaces the conclusion by

```
(iff
 (xlate-equiv (step x86) (top-level-opcode-execute pc ... x86))
 t)
```

We want ACL2 to treat `xlate-equiv` the same way it treats `iff` and `equal`; we want to "hang" rewriting on it.

Overview of Congruence-Based Rewriting

- **Equivalence Relations**
  - `defequiv, :rule-classes :equivalence`
- **Equiv-Rewrite (or Driver) Rules**
  - `:rule-classes :rewrite`
- **Congruence Rules:**
  - `defcong, :rule-classes :congruence`
- **Refinement Rules**
  - `defrefinement, :rule-classes :refinement`

Equivalence Relations

Functions like `xlate-equiv` must be equivalence relations.

- `defequiv`, `:rule-classes :equivalence`

```
(defequiv <equiv>):

(defthm <equiv>-is-an-equivalence
  (and (booleanp (<equiv> x y))
       (<equiv> x x)
       (implies (<equiv> x y)
                (<equiv> y x))
       (implies (and (<equiv> x y) (<equiv> y z))
                (<equiv> x z)))
  :rule-classes :equivalence)
```

Equiv-Rewrite Rules

Rules like `step-opener-in-system-level-mode` are equiv-rewrite rules (or driver rules). They rewrite terms using your equivalence relations.

- `:rule-classes :rewrite`

```
(defthm step-opener-in-system-level-mode
 (implies (and (not (programmer-level-mode x86))
               (other-hyps))
          (xlate-equiv (step x86)
                       (top-level-opcode-execute pc...x86))))
```

Congruence Rules

These rules tell ACL2 where the equiv-rewrite rules can be applied — they tell ACL2 to interpret an equiv-rewrite rule as hanging on the new equivalence relation, and not `iff`.

- `defcong, :rule-classes :congruence`

```
(defcong equiv1 equiv2 (fn x1 ... xk ... xn) k):

(defthm congruence-rule-example
 (implies (equiv1 xk xk-equiv)
          (equiv2 (fn x1... xk        ... xn)
                  (fn x1... xk-equiv ... xn)))
:rule-classes :congruence)
```

Refinement Rules

These rules allow `equiv1`-rewrite rules to be used in place of `equiv2`-rewrite rules. `Equal` is a refinement of all equivalence relations.

- `defrefinement, :rule-classes :refinement`

```
(defrefinement equiv1 equiv2):

(defthm refinement-rule-example
  (implies (equiv1 x y) (equiv2 x y))
:rule-classes :refinement)
```

Congruence-Based Rewriting

<DEMO>

x86isa: Symbolic Simulation in System-Level Mode

```
(defthm run-opener-no-error
  (implies (and (not (ms x86)) (not (zp n)))
           (equal (run n x86)
                  (run (1- n) (step x86)))))
```

```
(defthm step-opener-in-system-level-mode
  (implies (and (not (programmer-level-mode x86))
                (other-hyps))
           (xlate-equiv (step x86)
                        (top-level-opcode-execute pc ... x86))))
```

```
(defthm run-and-xlate-equiv
  (implies (xlate-equiv x86-1 x86-2)
           (xlate-equiv (run n x86-1) (run n x86-2)))
  :rule-classes :congruence)
```

`(run 1 x86) = (run 0 (step x86))`

Using `run-opener-no-error`, `equal` *context*

```
(run 1 x86) = (run 0 (step x86)) = ?
```

We could rewrite the above to

```
(run 0 (top-level-opcode-execute pc ... x86))
```

using

```
(defthm step-opener-in-system-level-mode
  (implies (and (not (programmer-level-mode x86))
                (other-hyps))
           (xlate-equiv (step x86)
                        (top-level-opcode-execute pc ... x86))))

(defthm run-and-xlate-equiv
  (implies (xlate-equiv x86-1 x86-2)
           (xlate-equiv (run n x86-1) (run n x86-2)))
  :rule-classes :congruence)
```

only if we could switch to the `xlate-equiv` context from the `equal` context.

To switch to the `xlate-equiv` context from the `equal` context, we need the following type of rules:

```
(defthm <accessor-fn>—and-xlate-equiv
  (implies (xlate-equiv x86-1 x86-2)
           (equal (<accessor-fn> ... x86-1))
                  (<accessor-fn> ... x86-2))))
  :rule-classes :congruence)
```

```
(read *pc* (run 1 x86))
=
(read *pc* (run 0 (step x86)))
=
(read *pc* (run 0 (top-level-opcode-execute pc ... x86)))
=
```

and so on.

Key to this Plan

```
(defthm <accessor-fn>—and-xlate-equiv
  (implies (xlate-equiv x86-1 x86-2)
           (equal (<accessor-fn> ... x86-1))
                  (<accessor-fn> ... x86-2))))
  :rule-classes :congruence)
```

But, the following isn't a theorem!

```
(defthm memi-and-xlate-equiv
  (implies (xlate-equiv x86-1 x86-2)
           (equal (memi phy-addr x86-1))
                  (memi phy-addr x86-2))))
  :rule-classes :congruence)
```

What if `phy-addr` is the physical address that contains the accessed and dirty bits?

A Problem for this Solution

An entire field of the state needs to be ignored during reasoning.

- E.g., a field which records information during program execution.
- **The field's reader should not appear in `run` and its supporters.**
- **The field's writer may appear in `run` and its supporters.**

```
(defthm <accessor-fn>–and-xlate-equiv
  (implies (xlate-equiv x86-1 x86-2)
           (equal (<accessor-fn> ... x86-1))
                  (<accessor-fn> ... x86-2))))
  :rule-classes :congruence)
```

Remember: Warnings!

Pay attention to warnings when doing congruence-based rewriting.

- Double-rewrite `[:doc double-rewrite]`
- Replacing iff by equal `[:doc congruence]`

Symbolic Simulation of x86-64 Instructions Using Congruence-Based Rewriting

A **Failed** but **Interesting** Experiment on the `x86isa` Model
Useful

Shilpi Goel

ACL2 Seminar

March, 2016