Towards rigorous relaxed memory models

Peter Sewell

University of Cambridge

joint work with:

Mark Batty, Magnus Myreen, Scott Owens, Tom Ridge, and Susmit Sarkar
(University of Cambridge)

Jade Alglave, Luc Maranget, and Francesco Zappa Nardelli
(INRIA)

`http://www.cl.cam.ac.uk/~pes20/weakmemory/`

Intel DTS Workshop on Hardware & Software Co-Design & Co-Verification
Haifa   8 September 2009

We’re not in Kansas anymore

Traditional assumption:

> multiprocessors are *sequentially consistent*:
> accesses by multiple threads to a shared memory
> occur in a global-time linear order.

We’re not in Kansas anymore

Traditional assumption:

> multiprocessors are *sequentially consistent*:
> accesses by multiple threads to a shared memory
> occur in a global-time linear order.

False!

Multiprocessors (and compilers) incorporate optimisations:
local store buffers, shadow register files, cache hierarchies,...

- unobservable by single-threaded programs;
- sometimes observable by concurrent code.

Only a *relaxed* (or *weakly consistent*) view of the memory.

A Simple Example

Initial: [x]$=$0 $\wedge$ [y]$=$0	
proc 0	proc 1
MOV [x]$\leftarrow$\$1 MOV EAX$\leftarrow$[y] (0)	MOV [y]$\leftarrow$\$1 MOV EBX$\leftarrow$[x] (0)
Allow: EAX$=$0 $\wedge$ EBX$=$0	

- One can't view the execution in global time
  (at this level of abstraction)
- Observable on dual core Intel Core2 (630 / 100,000)
- Plausible microarchitectural explanation
- If that's allowed, then what *else* might be?

Our Plan

We’ve been looking at the memory models of x86, Power, ARM, and C++ (and Ševčík and Aspinall looked at Java).

Our Plan

We’ve been looking at the memory models of x86, Power, ARM, and C++ (and Ševčík and Aspinall looked at Java).

$\boxed{\text{The vendor specs and language standards are } \textbf{all} \text{ flawed}}$

Our Plan

We've been looking at the memory models of x86, Power, ARM, and C++ (and Ševčík and Aspinall looked at Java).

$\boxed{\text{The vendor specs and language standards are } \textbf{all} \text{ flawed}}$

So, we try to establish usable models, for programming and as a basis for software verification.

Architectures

Hardware manufacturers document *architectures*:

- loose specifications;
- claimed to cover a wide range of past and future processor implementations.

Architectures should:

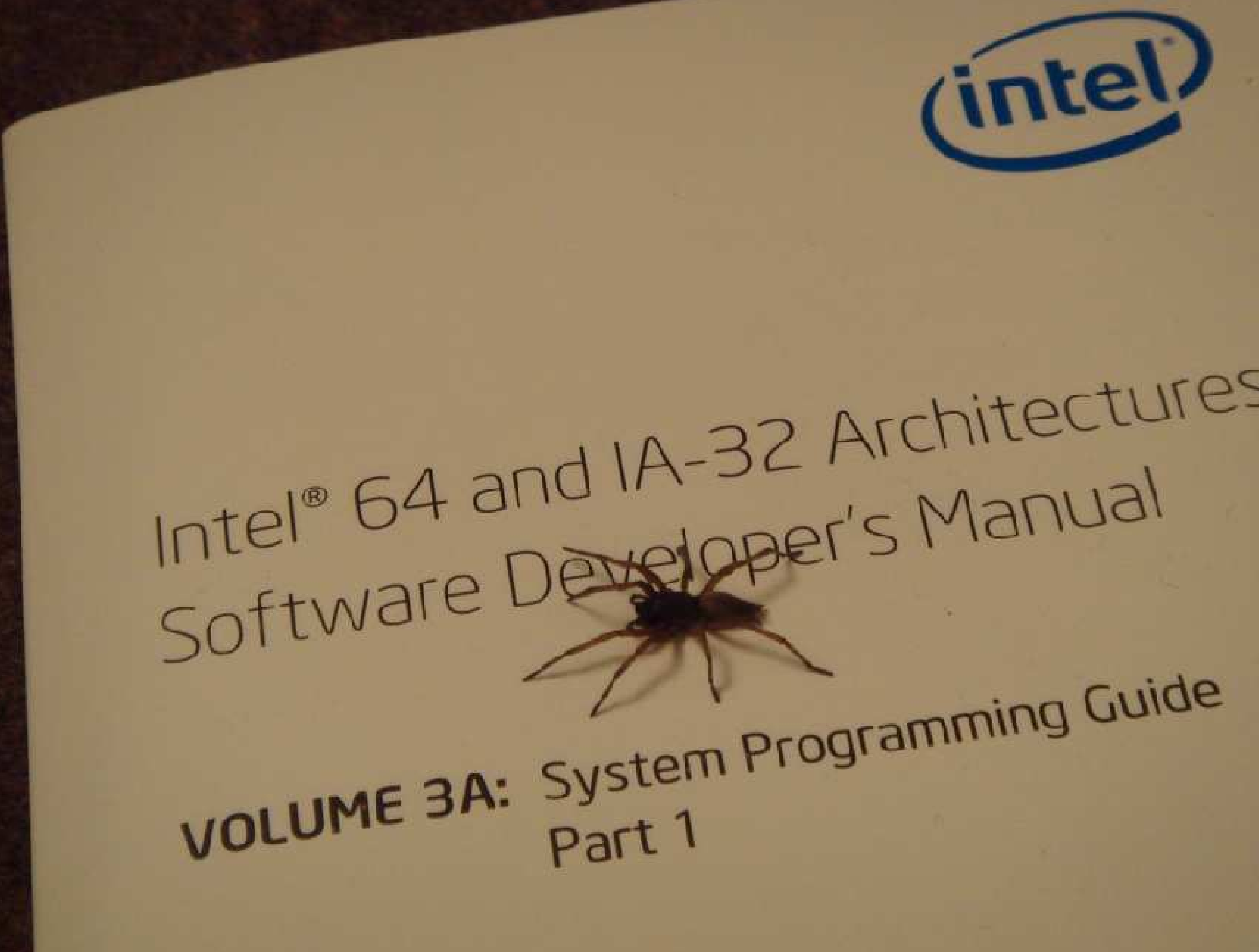
- reveal enough for effective programming;
- without revealing sensitive IP; and
- without unduly constraining future processor design.

Examples:

Intel 64 and IA-32 Architectures SDM,

AMD64 Architecture Programmer's Manual,

Power ISA specification,...



Intel® 64 and IA-32 Architectures
Software Developer's Manual

VOLUME 3A: System Programming Guide
Part 1

In practice

Architectures described by *informal prose*:

> *In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.*

(Intel SDM, Nov. 2006, vol 3a, 10-5)

As we shall see, such descriptions sometimes are:

1) vague; 2) incomplete; 3) unsound.

Also, they cannot be used to *test programs* or to *test processor implementations*.

Era of Vagueness (before Aug. 2007, e.g. Intel SDM rev. 22)

- Linux kernel mailing list (Nov. 1999)
- Simple programming questions, micro-architectural debate
  - *speculation*
  - *ordering*
  - *causality*
  - *retire*
  - *cache*
- Resolved only by appeal to an oracle

Era of Causality: Ambiguity

IWP/SDM rev. 26/AMD 3.14/x86-CC (Aug 2007 – Oct. 2008)

- 10 *litmus tests*, e.g.:

proc:0	proc:1	proc:2
MOV [x]$\leftarrow$\$1	MOV EAX$\leftarrow$[x]	MOV EBX$\leftarrow$[y]
	MOV [y]$\leftarrow$\$1	MOV ECX$\leftarrow$[x]
Forbid: 1:EAX$=$1 $\wedge$ 2:EBX$=$1 $\wedge$ 2:ECX$=$0		

- 8 *‘principles’*, e.g.:

  “Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation”

Ambiguity: “causality”

Era of Causality: Weakness

Independent reads of independent writes

Initial: [x]=0 ∧ [y]=0			
proc 0	proc 1	proc 2	proc 3
MOV [x]←$1	MOV [y]←$1	MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV ECX←[y] (1) MOV EDX←[x] (0)
Final: EAX=1 ∧ EBX=0 ∧ ECX=1 ∧ EDX=0			
cc : Allow; tso : Forbid			

- proc2: see write x before write y
- proc3: see write y before write x
- AMD: yes!
- Intel: ???
- real hardware: unobserved

Weakness: adding MFENCEs does not recover SC (which was assumed in a Sun implementation of the JMM)

Era of Causality: Unsoundness

Initial: [x]=0 ∧ [y]=0	
proc 0	proc 1
MOV [x]←\$1	MOV [y]←\$2
MOV EAX←[x] (1)	MOV [x]←\$2
MOV EBX←[y] (0)	
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	

(Thanks to Paul Loewenstein)

Observed on real hardware, but not allowed by ‘principles’:

“Stores are not reordered with other stores”

and

“In a multiprocessor system, stores to the same location have a total order”

Second Era of Causality

SDM rev. 29–31 (Nov. 2008 – now)

- Not unsound in the previous sense
- Not weak in the IRIW sense

  "Any two stores are seen in a consistent order by processors other than those performing the stores."

- But... still a bit ambiguous, and *the view by those processors is left entirely unspecified!*

proc:0	proc:1
MOV [x]$\leftarrow$\$1 MOV EAX$\leftarrow$[x]	MOV [x]$\leftarrow$\$2 MOV EBX$\leftarrow$[x]
Forbid: 0:EAX$=$2 $\wedge$ 1:EBX$=$1	

Modern Age (x86-TSO)

- Folk wisdom: x86 has a relatively strong architecture.

  I _like_ PC's. Almost every other architecture decided to be lazy in hw, and put the onus on the software to tell it what was right. The PC platform hardware competition didn't allow for the "let's recompile the software" approach, so the hardware does it all for you.

  —Linus Torvalds, linux-arch list, 7 March 2006
- Litmus tests consistent with total store ordering (TSO)
- Suggestive rev29
- Suggestive vendor comments

Simple Plan: x86-TSO

Specify a TSO-based programmer-visible architecture
(adapting to x86 as appropriate)

Separate instruction semantics and memory model
(specify and test 30-odd instructions, with all addressing modes, including instruction decoding)

Define both abstract machine and axiomatic versions of MM

In HOL (mechanized logic)

Intel and AMD Memory Model Descriptions

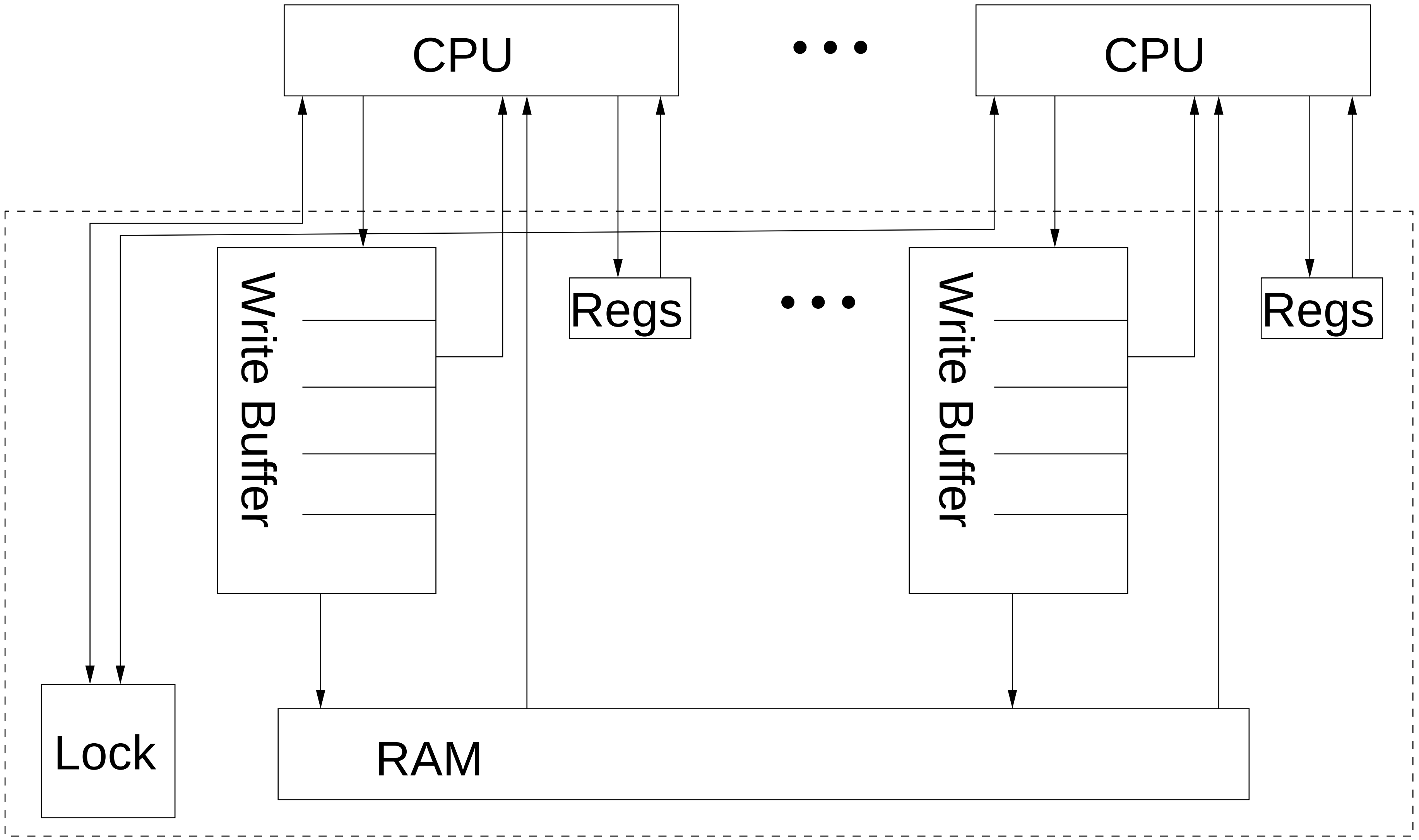
Ages past to Nov 2006	Intel manuals, rev 22	Extremely **vague**
Aug 2007	Intel White Paper v1.0	Moderately clear except for “causality” (x86-CC)
Sep 2007	AMD manual, rev 3.14	Arguably **too weak** for programmers
Feb 2008	Intel manuals, rev 26	**Unsound** w.r.t. current hardware
Nov 2008 to date	Intel manuals, rev 29–31	Somewhat less clear (esp. causality) **Sound** (as far as we know) w.r.t. current hardware Surprisingly **weak**
Now	Our Proposal	Based on x86 “folk knowledge” (x86-TSO) **Sound** (as far as we know) w.r.t. current hardware **Strong** enough to program to

Focus: Not *All* of x86

Basic user-code scenario:

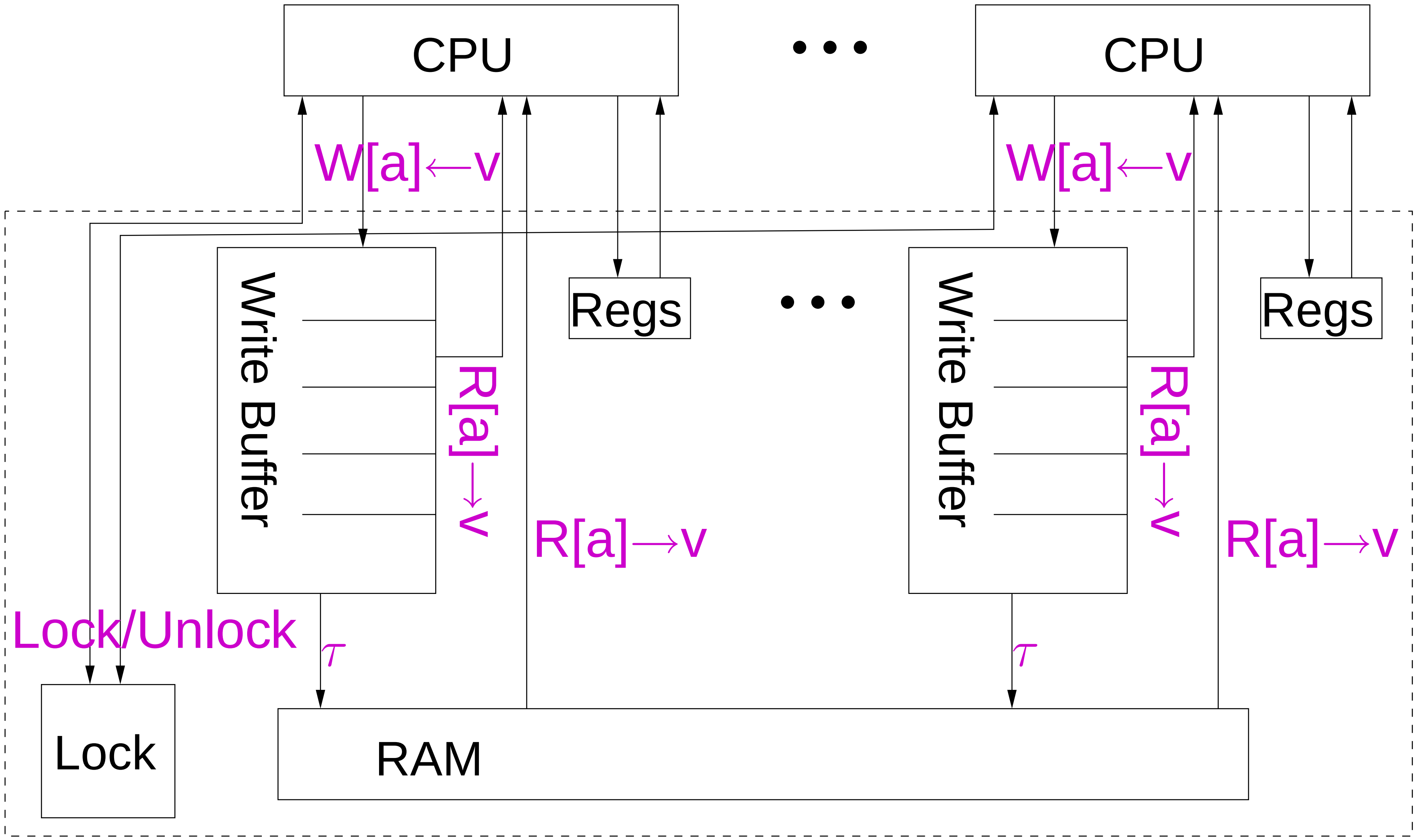
- coherent write-back memory
- no exceptions
- no misaligned accesses
- no ‘non-temporal’ operations
- no self-modifying code
- no page-table changes

Abstract Machine



- Transition relation: $s \xrightarrow{l} s$
- $\|$ w/ instruction semantics machine

Abstract Machine



- Transition relation: $s \xrightarrow{l} s$
- $\|$ w/ instruction semantics machine

Abstract Machine

A tool to specify the *programmer-visible behaviour* only.

The internal structure should be easy to understand, but may be (is!) not much like the actual h/w.

Force of the model: programmers can assume that nothing (of the internal optimisations of processors) *except* FIFO write buffers is *visible*

Barriers and Locks

- MFENCE
  - flush local write buffer
- LOCK prefix (for CMPXCHG etc.)
  - flush local write buffer
  - globally lock memory

Initial: [x]$=$0 $\wedge$ [y]$=$0	
proc 0	proc 1
MOV [x]$\leftarrow$\$1 MFENCE MOV EAX$\leftarrow$[y]	MOV [y]$\leftarrow$\$1 MFENCE MOV EBX$\leftarrow$[x]
Forbid: EAX$=$0 $\wedge$ EBX$=$0	

Axiomatic Memory Model

- $\approx$ SPARCv8
- predicate on potential executions
- $\exists$memory_order.
- Partial order on memory events
  - that is a total order on writes
- Consistent with program order
  - load-load
  - store-store
  - load-store
- Equivalent to abstract machine
- Useful for metatheory

Side-by-side Comparison

notTblocked =
$(s.L = \textsf{None}) \lor (s.L = \textsf{Some}\ \ p)$
not_blocked =
$\neg(\exists v'.\, \text{MEM}\ (a, v')b)$

Read from memory

$$\frac{\text{not\_blocked}\ \ s\ p \land (s.M\ \ a = \textsc{Some}\ \ v) \land \text{no\_pending}\ (s.B\ \ p)a}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location\_mem}\ a)v)} s}$$

Read from write buffer

$$\frac{\text{not\_blocked}\ \ s\ p \land (\exists b_1\ b_2.(s.B\ \ p\ = b_1 +\!\!+[(a,v)] +\!\!+ b_2) \land \text{no\_pending}\ \ b_1\ \ a)}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location\_mem}\ a)v)} s}$$

Read from register

$$\frac{(s.R\ p\ r = \textsc{Some}\ \ v)}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location\_reg}\ p\ r)v)} s}$$

Write to write buffer

$$\frac{\mathbf{T}}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ \mathrm{W}\ (\textsc{Location\_mem}\ a)v)} s \oplus \{\!|B := s.B \oplus (p \mapsto [(a,v)] +\!\!+(s.B\ \ p))|\!\}}$$

Write from write buffer to memory

$$\frac{\text{not\_blocked}\ \ s\ p \land (s.B\ \ p\ = b +\!\!+[(a,v)])}{s \xrightarrow{\textsc{Tau}} s \oplus \{\!|M := s.M \oplus (a \mapsto \textsc{Some}\ \ v); B := s.B \oplus (p \mapsto b)|\!\}}$$

Write to register

$$\frac{\mathbf{T}}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ \mathrm{W}\ (\textsc{Location\_reg}\ p\ r)v)} s \oplus \{\!|R := s.R \oplus (p \mapsto ((s.R\ \ p) \oplus (r \mapsto \textsc{Some}\ \ v)))|\!\}}$$

Barrier

$$\frac{(b = \textsc{Mfence}) \implies (s.B\ \ p = [\,])}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Barrier}\ b)} s}$$

Lock

$$\frac{(s.L = \textsc{None}) \land (s.B\ \ p = [\,])}{s \xrightarrow{\textsc{Lock}\ p} s \oplus \{\!|L := \textsc{Some}\ \ p|\!\}}$$

Unlock

$$\frac{(s.L = \textsc{Some}\ \ p) \land (s.B\ \ p = [\,])}{s \xrightarrow{\textsc{Unlock}\ p} s \oplus \{\!|L := \textsc{None}|\!\}}$$

reads_from_map_candidates =
$\forall(ew, er) \in rfmap.(er\ \in\ \text{reads}\ \ E) \land (ew\ \in\ \text{writes}\ \ E) \land$
$(\text{loc}\ \ ew = \text{loc}\ \ er) \land (\text{value\_of}\ \ ew = \text{value\_of}\ \ er)$
check_rfmap_written =
$\forall(ew, er) \in (X.rfmap).$
if $ew\ \in\ \text{mem\_accesses}\ \ E$ **then**
$ew\ \in\ \text{maximal\_elements}\ (\text{previous\_writes}\ \ E\ \ er\ \ X.memory\_order\ \cup$
$\text{previous\_writes}\ \ E\ \ er\ (\text{po\_iico}\ \ E))$
$X.memory\_order$
else
$ew\ \in\ \text{maximal\_elements}\ (\text{previous\_writes}\ \ E\ \ er\ (\text{po\_iico}\ \ E))(\text{po\_iico}\ \ E)$
check_rfmap_initial =
$\forall er \in (\text{reads}\ \ E\ \setminus\ \text{range}\ \ X.rfmap).$
$(\exists l.(\text{loc}\ \ er = \textsf{Some}\ \ l) \land (\text{value\_of}\ \ er = X.initial\_state\ l)) \land$
$(\text{previous\_writes}\ \ E\ \ er\ \ X.memory\_order\ \cup$
$\text{previous\_writes}\ \ E\ \ er\ (\text{po\_iico}\ \ E) = \{\})$
valid_execution =
$\text{partial\_order}\ \ X.memory\_order\ (\text{mem\_accesses}\ \ E) \land$
$\text{linear\_order}\ (X.memory\_order|_{(\text{mem\_writes}\ \ E)})(\text{mem\_writes}\ \ E) \land$
$\text{finite\_prefixes}\ \ X.memory\_order\ (\text{mem\_accesses}\ \ E) \land$
$(\forall ew \in (\text{mem\_writes}\ \ E).$
$\text{finite}\{er \mid er\ \in\ E.events \land (\text{loc}\ \ er = \text{loc}\ \ ew) \land$
$(er, ew) \notin\ X.memory\_order \land$
$(ew, er) \notin\ X.memory\_order\}) \land$
$(\forall er \in (\text{mem\_reads}\ \ E).\forall e \in (\text{mem\_accesses}\ \ E).(er, e) \in\ \text{po\_iico}\ \ E \implies$
$(er, e) \in\ X.memory\_order) \land$
$(\forall ew_1\ ew_2 \in (\text{mem\_writes}\ \ E).(ew_1, ew_2) \in\ \text{po\_iico}\ \ E \implies$
$(ew_1, ew_2) \in\ X.memory\_order) \land$
$(\forall ew \in (\text{mem\_writes}\ \ E).\forall er \in (\text{mem\_reads}\ \ E).\forall ef \in (\text{mfences}\ \ E).$
$(ew, ef) \in\ \text{po\_iico}\ \ E \land (ef, er) \in\ \text{po\_iico}\ \ E \implies$
$(ew, er) \in\ X.memory\_order) \land$
$(\forall e_1\ e_2 \in (\text{mem\_accesses}\ \ E).\forall es \in (E.atomicity).$
$(e_1\ \in\ es \lor e_2\ \in\ es) \land (e_1, e_2) \in\ \text{po\_iico}\ \ E$
$\implies$
$(e_1, e_2) \in\ X.memory\_order) \land$
$(\forall es \in (E.atomicity).\forall e \in (\text{mem\_accesses}\ \ E\ \setminus\ es).$
$(\forall e' \in (es\ \cap\ \text{mem\_accesses}\ \ E).(e, e') \in\ X.memory\_order) \lor$
$(\forall e' \in (es\ \cap\ \text{mem\_accesses}\ \ E).(e', e) \in\ X.memory\_order)) \land$
$X.rfmap\ \in\ \text{reads\_from\_map\_candidates}\ \ E \land$
$\text{check\_rfmap\_written}\ \ E\ X \land$
$\text{check\_rfmap\_initial}\ \ E\ X$

Theorems

- Linear axiomatic executions
- Event annotated machine
- Abstract machine/axiomatic model equivalence
- Infinite executions (and liveness)

Tools

- MEMEVENTS, evaluating all executions of litmus tests
- LITMUS, running litmus tests on real h/w
- X86SEM, testing instruction semantics on real h/w

Comparing Models

	IWP	rev-29	test on hardware	*x86-TSO*
IRIW	allowed	forbidden	not observed	*forbidden*
Loewenstein	forbidden	allowed	observed	*allowed*
two stores	forbidden	allowed	not observed	*forbidden*

(testing is subject to the obvious limitations)

(x86-TSO is also consistent with h/w for various other tests)

Open Questions

- write-buffer progress properties?
- mixed-size accesses?
- ...all the rest of x86?

Related Work

- TSO
  - Burckhardt and Musuvathi (CAV 2008)
  - Sun TSOtool, Hangel et al. (ISCA 2004)
  - Roy et al. (CAV 2006)
  - Boudol and Petri (POPL 2009)
- x86
  - Burckhardt et al. (Tech. report)
- Equivalence proof
  - Seungjoon Park (PhD thesis 1996)

Power ISA 2.06 and ARM v7

Key concept: actions being *performed*.

> A *load* by a processor (P1) *is performed* with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).

Performed

> A *load* by a processor (P1) *is performed* with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

This is *subjunctive*: it refers to a hypothetical store by P2.

A memory model should define whether a particular execution is allowed: it is awkward to make a definition that *explicitly quantifies over such hypothetical variant executions*.

The Java Memory Model(s)

Java has integrated multithreading, and it attempts to specify the precise behaviour of concurrent programs.

By the year 2000, the initial specification was shown:

- to allow unexpected behaviours;
- to prohibit common compiler optimisations,
- to be challenging to implement on top of a weakly-consistent multiprocessor.

Superseded around 2004 by the JSR-133 memory model.

JSR-133

- Goal 1: data-race free programs are sequentially consistent;
- Goal 2: all programs satisfy some memory safety and security requirements;
- Goal 3: common compiler optimisations are sound.

JSR-133: Unsoundness

The model is intricate, and *fails to meet Goal 3.*
[Ševčík and Aspinall]

Some optimisations may generate code that exhibits more behaviours than those allowed by the un-optimised source.

As an example, JSR-133 allows `r2=1` in the optimised code below, but forbids `r2=1` in the source code:

x = y = 0	
`r1=x`	`r2=y`
`y=r1`	`x=(r2==1)?y:1`

$\xrightarrow{\textit{HotSpot optimisation}}$

x = y = 0	
`r1=x`	`x=1`
`y=r1`	`r2=y`

The C++ memory model

C++ was originally designed without thread support.

Ongoing effort (almost completed) to provide semantics for threads in the next revision of the standard.

The model gives semantics only to data-race free programs.

Some (easy to fix) ambiguities are manifest if the model is formalised in mathematical language.

Suspicion: most of the specification is (informally) axiomatic, but there is one appeal to compiler concepts

Danger: very complex semantics for *low-level atomics*.

Loose Specifications

Architectures are the key interface between h/w and s/w.

They are necessarily *loose specifications*

But informal prose is a *terrible* way to express loose specifications: ambiguous, untestable, and usually wrong.

Instead, architectures should be mathematically rigorous, clarifying precisely just *how* loose one wants them to be.

(common misconception: precise = tight ?)

A Rigorous Memory Model should be:

1. precise;

   mathematical language, ideally expressed in a proof assistant

2. testable;

   compute one/all of the executions; algorithm derived from the statements of the model

3. accurate with respect to implementations;

   allow all the behaviours observed in practice

4. loose enough for future implementations;

   but this point should not be over-emphasized!

5. strong enough for programmers;

   so that reasonable programs can be shown to behave as intended

6. integrated with the semantics of the rest of the system;

7. accessible.

   to programmers, hardware architects, language designers and implementors

The End

Intel Principles

P1 Reads are not reordered with other reads.

P3 Writes are not reordered with older reads.

P2 Writes to memory are not reordered with other writes[, with the exception of writes executed with the CLFLUSH instruction,streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD),string operations (see Section 7.2.4.1)].

P4 Reads may be reordered with older writes to different locations but not with older writes to the same location.

P8 Reads or writes cannot be reordered with [I/O instructions,] locked instructions[, or serializing instructions].

P11 Reads cannot pass LFENCE and MFENCE instructions.

P12 Writes cannot pass SFENCE and MFENCE instructions.

In a multiple-processor system, the following ordering principles apply:

P13 Individual processors use the same ordering principles as in a single-processor system.

P10 Writes by a single processor are observed in the same order by all processors.

P14 Writes from an individual processor are NOT ordered with respect to the writes from other processors.

P5 Memory ordering obeys causality (memory ordering respects transitive visibility).

P9 Any two stores are seen in a consistent order by processors other than those performing the stores.

P7 Locked instructions have a total order.

Proof Techniques

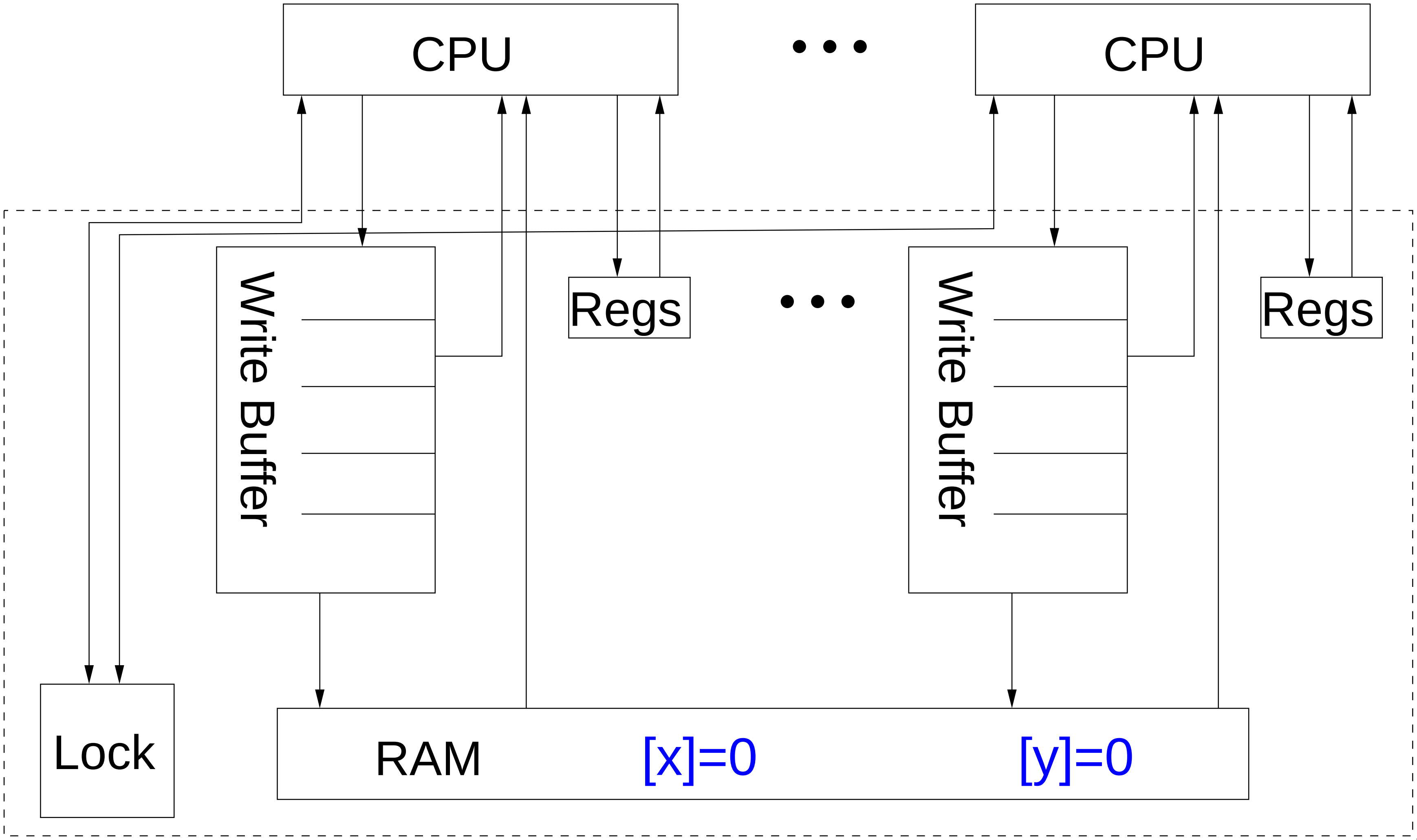
- Correlate events and labels

write event	W[a]$\leftarrow$v and τ
read event	R[a]$\rightarrow$v
mfence event	mfence label
set of locked events	proper L and U label bracketing

- Rule out pathological instructions
- Extend *stream-like* P.O. to *stream-like* T.O.
- Induction
- Explicit constructions, no transitive closure

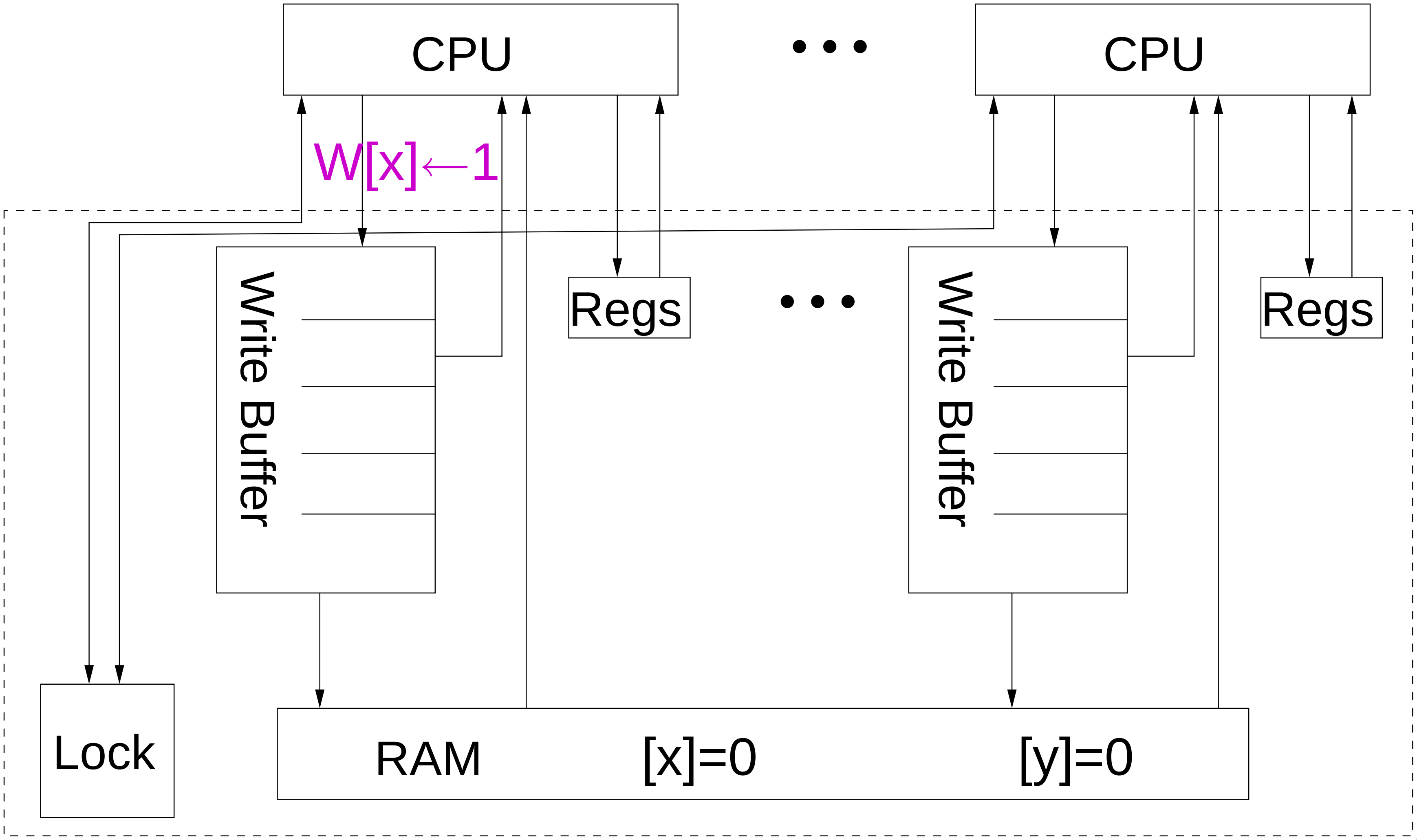
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[y] (0)	MOV [y]←$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



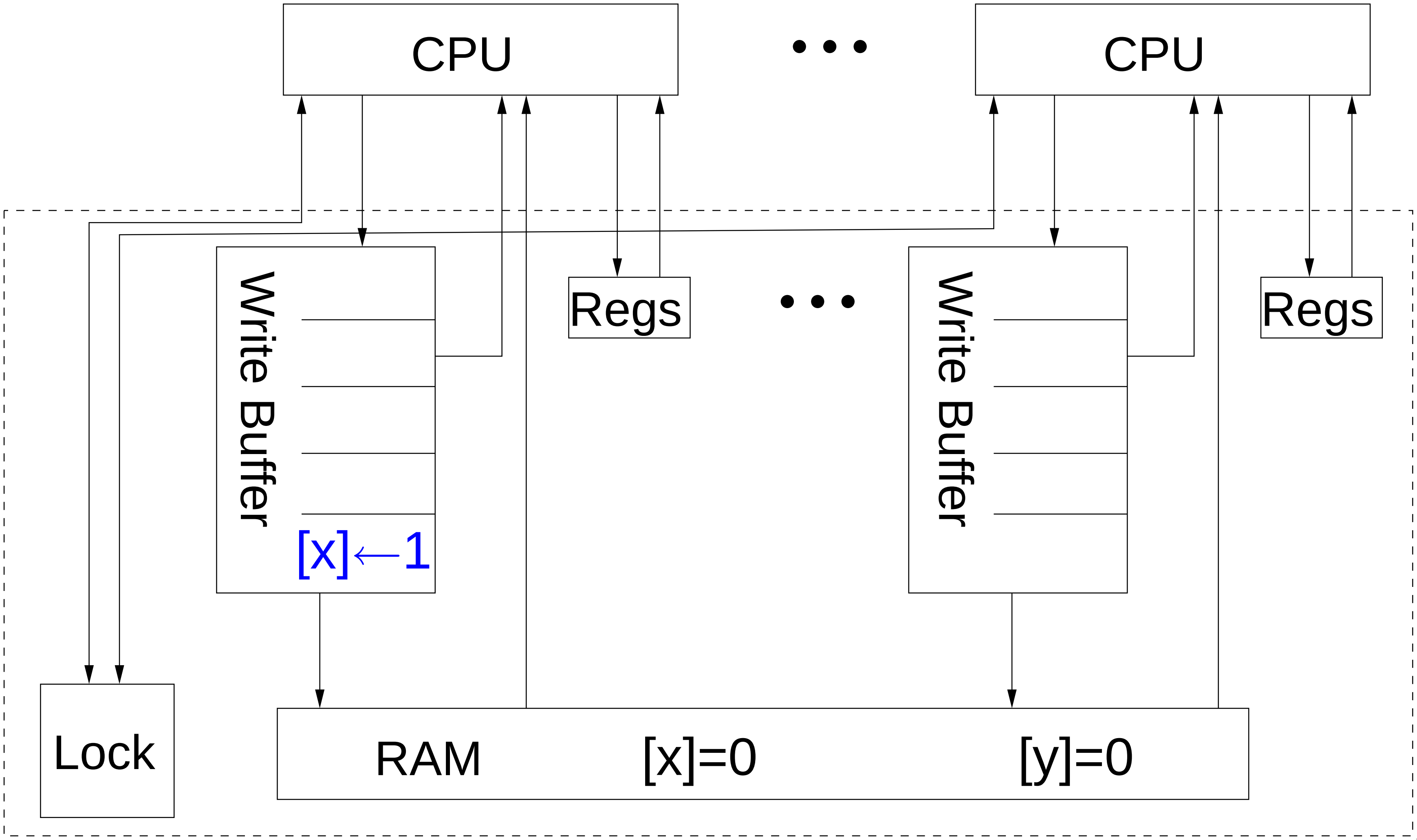
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[y] (0)	MOV [y]←$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



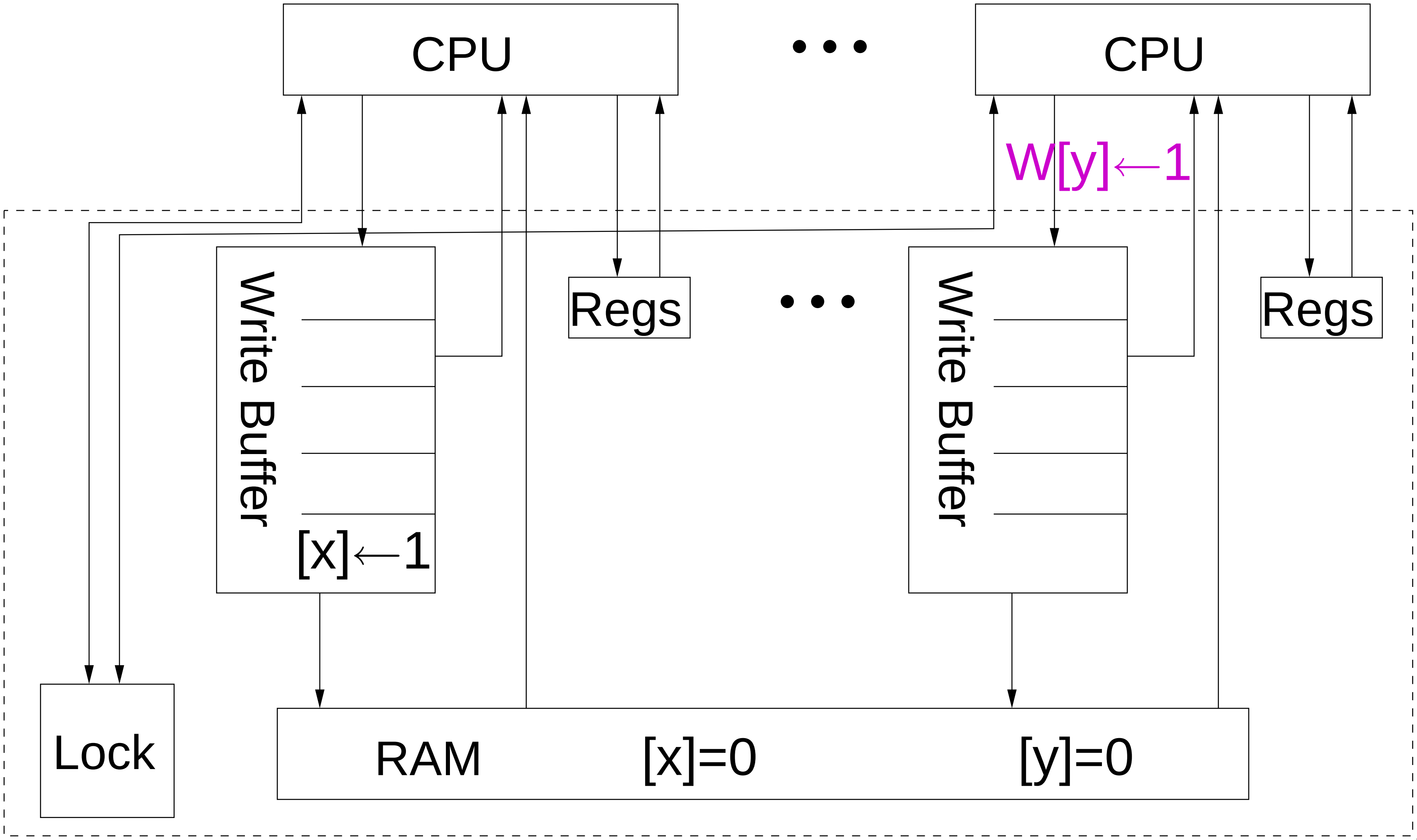
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[y] (0)	MOV [y]←\$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



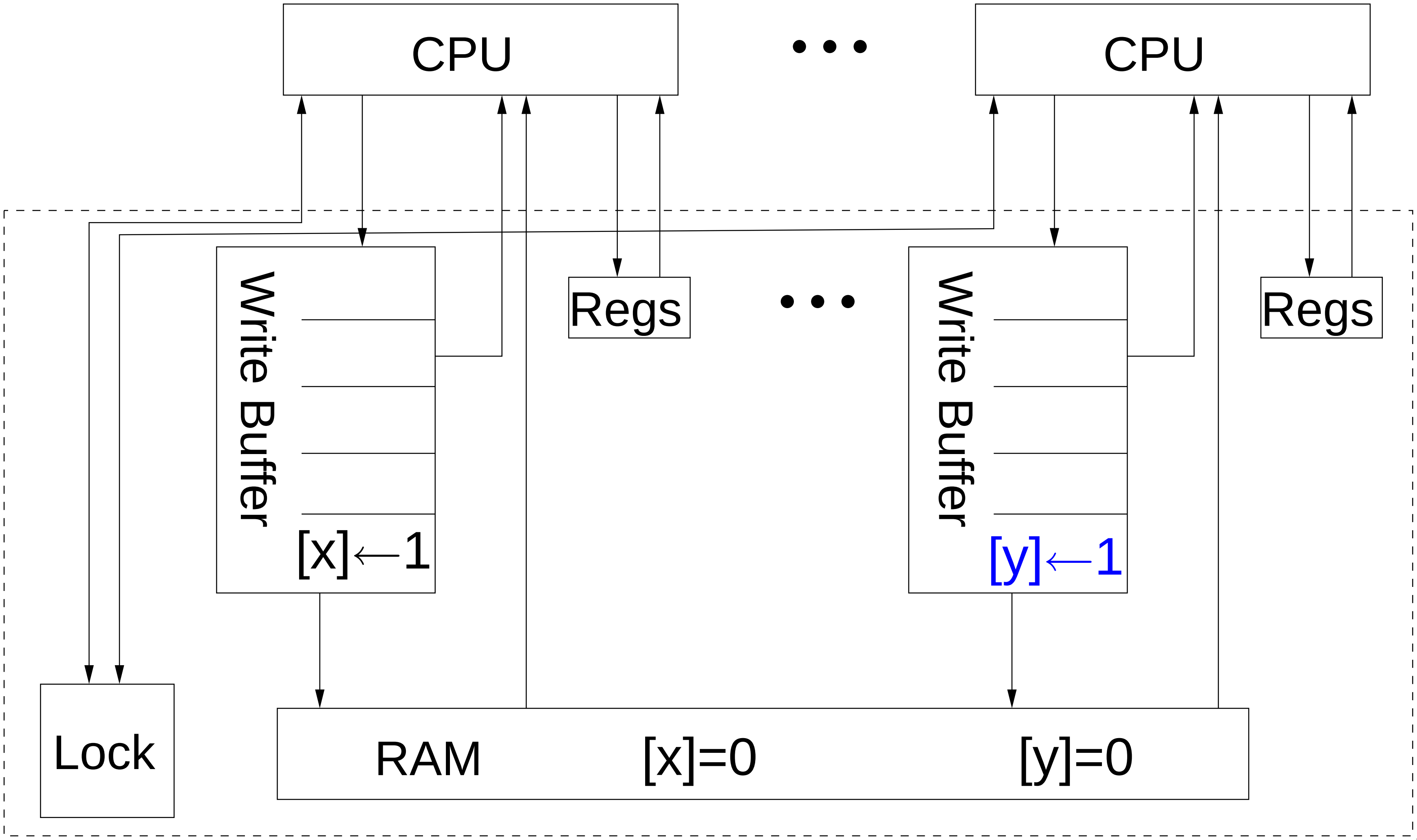
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[y] (0)	MOV [y]←\$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



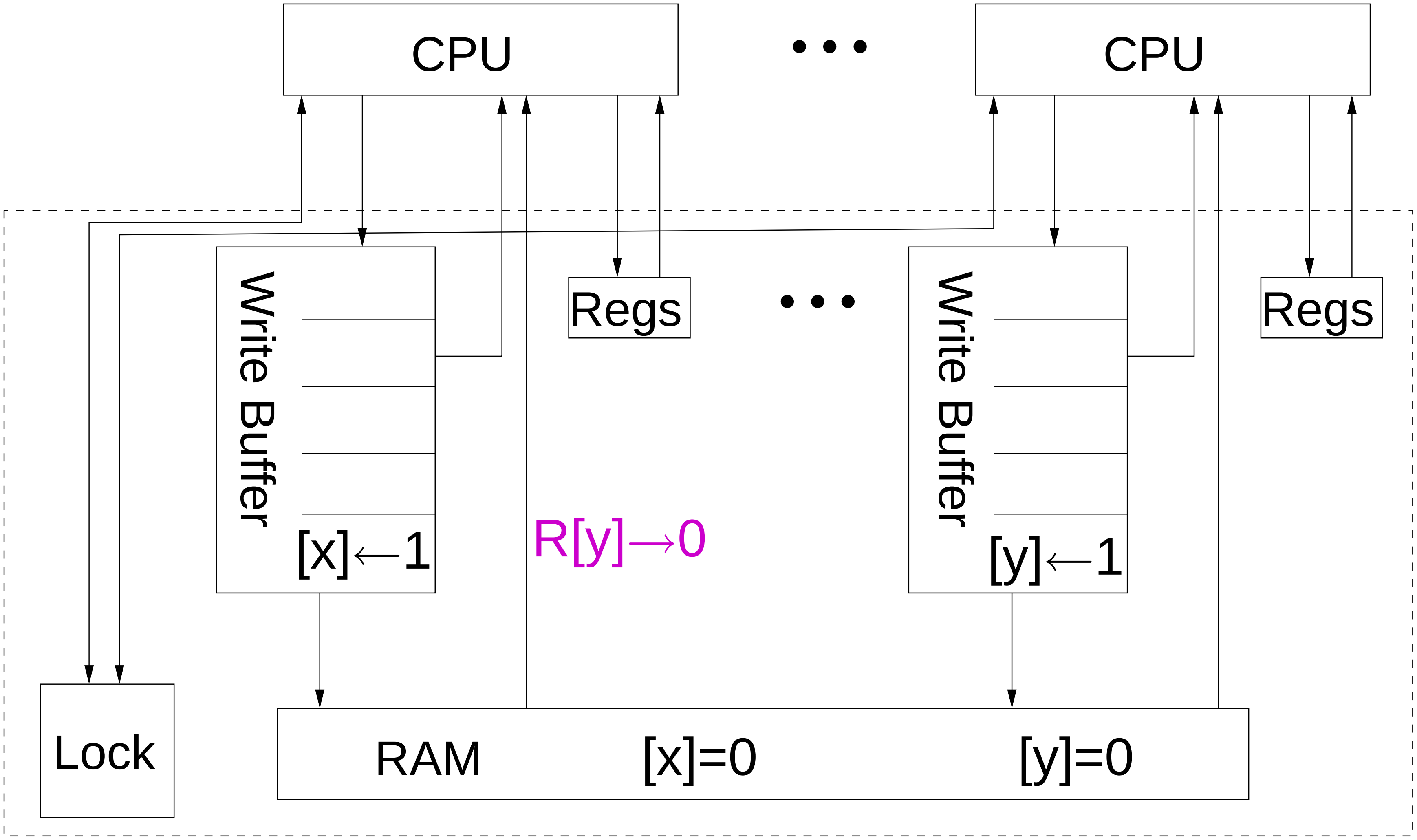
Litmus Test Example

Initial: $[x]=0 \wedge [y]=0$	
MOV $[x] \leftarrow \$1$ MOV EAX$\leftarrow[y]$ (0)	MOV $[y] \leftarrow \$1$ MOV EBX$\leftarrow[x]$ (0)
Allow: EAX$=0 \wedge$ EBX$=0$	



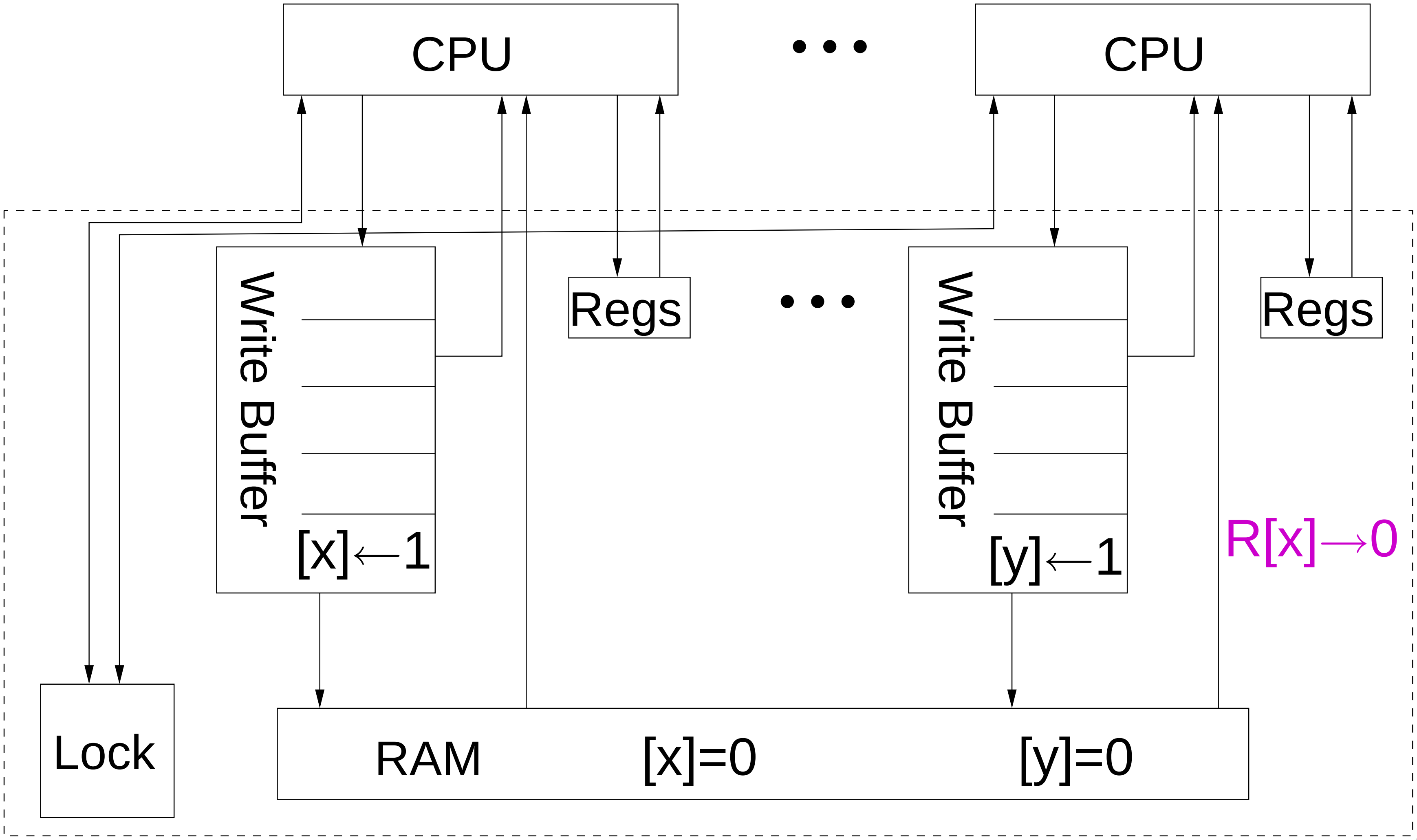
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[y] (0)	MOV [y]←$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



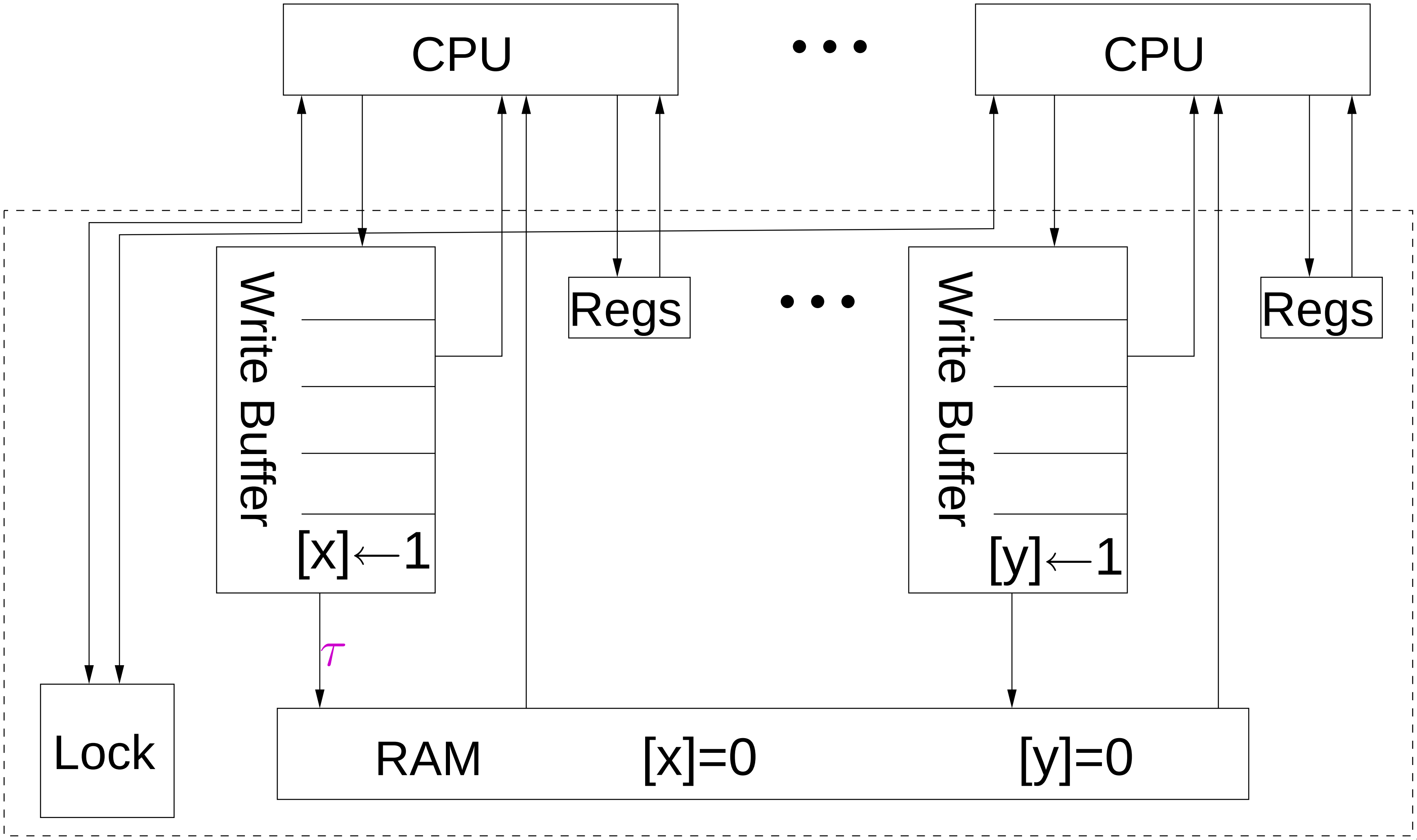
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[y] (0)	MOV [y]←\$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



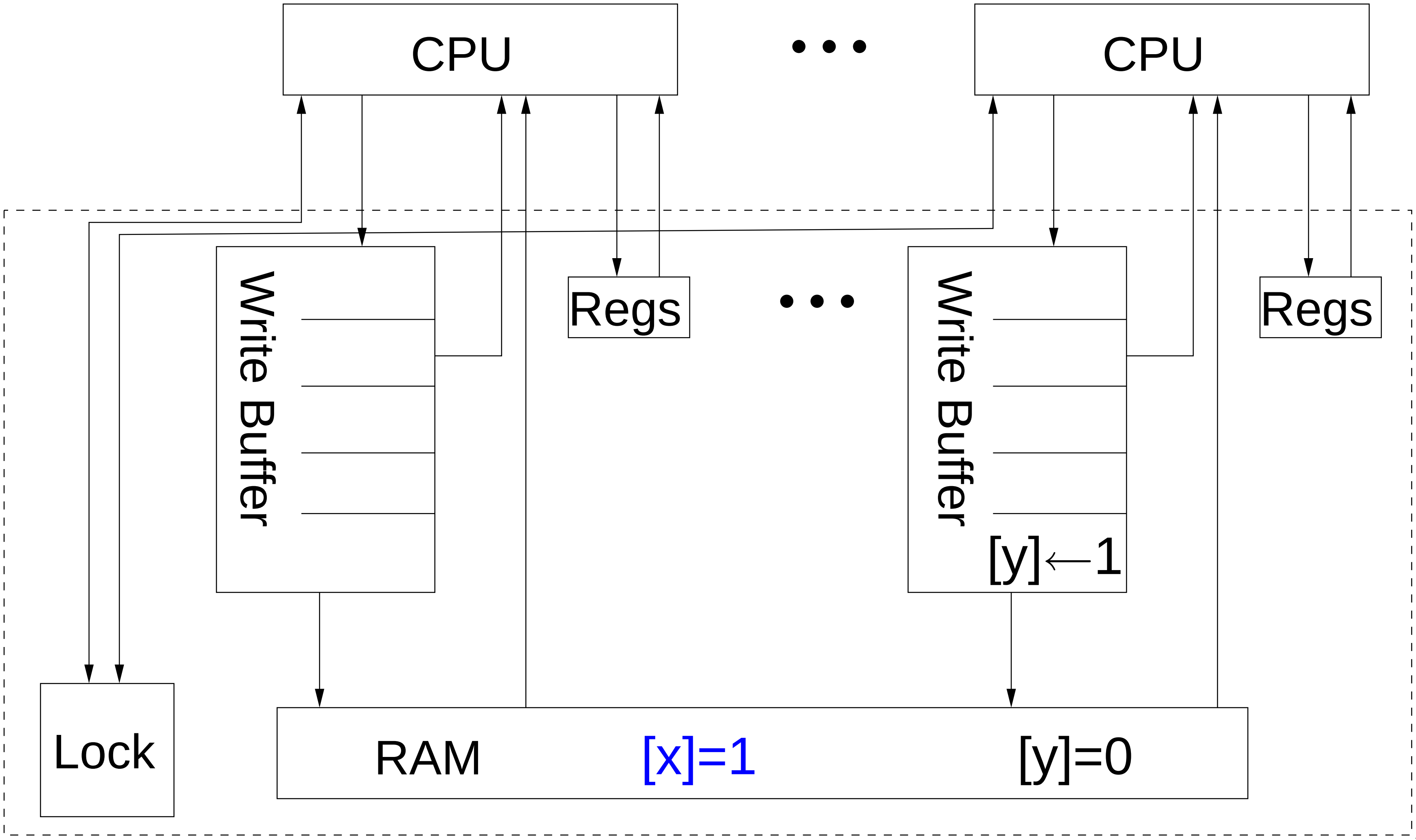
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[y] (0)	MOV [y]←\$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



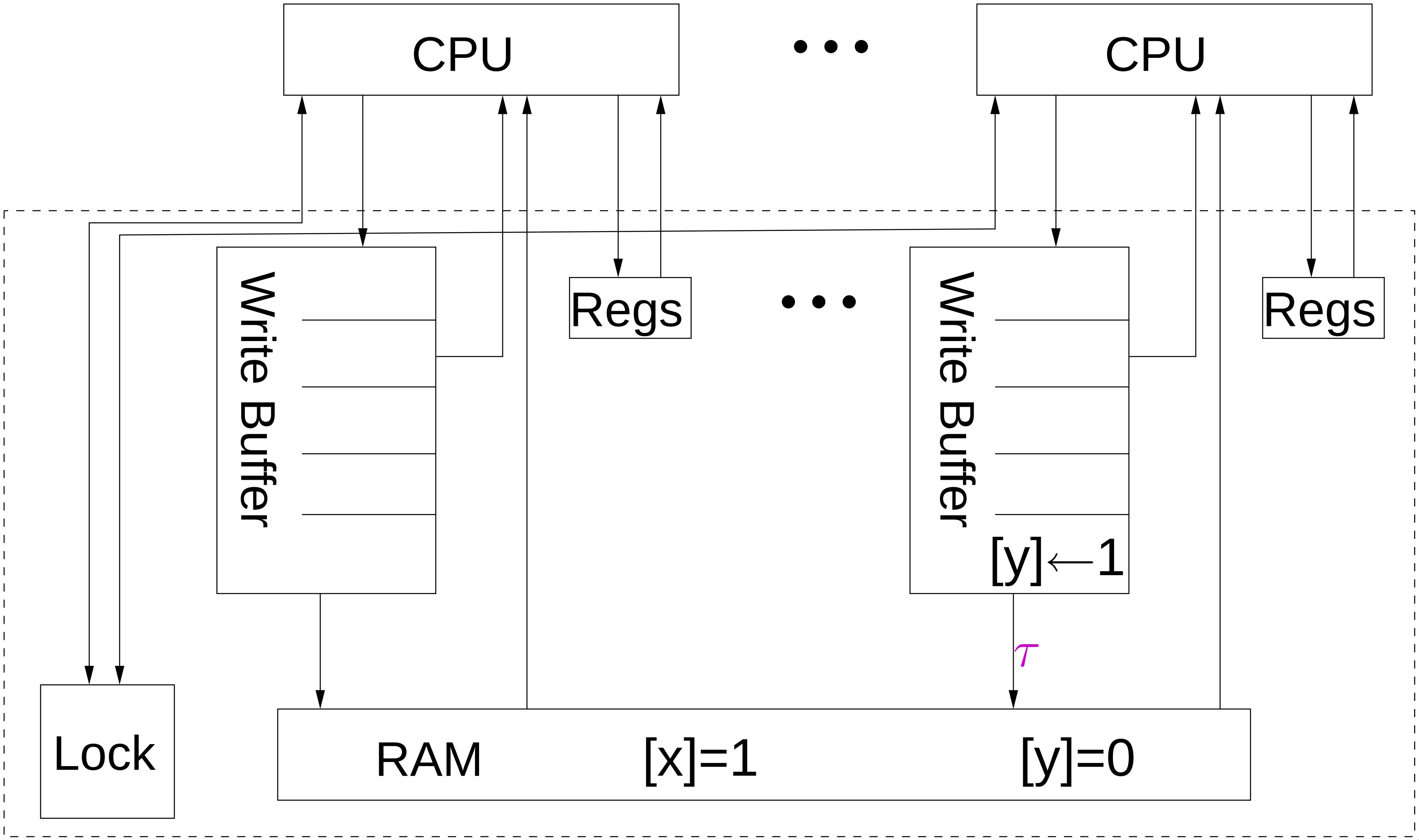
Litmus Test Example

Initial: $[x]=0 \wedge [y]=0$	
MOV $[x] \leftarrow \$1$	MOV $[y] \leftarrow \$1$
MOV EAX$\leftarrow [y]$ (0)	MOV EBX$\leftarrow [x]$ (0)
Allow: EAX$=0 \wedge$ EBX$=0$	



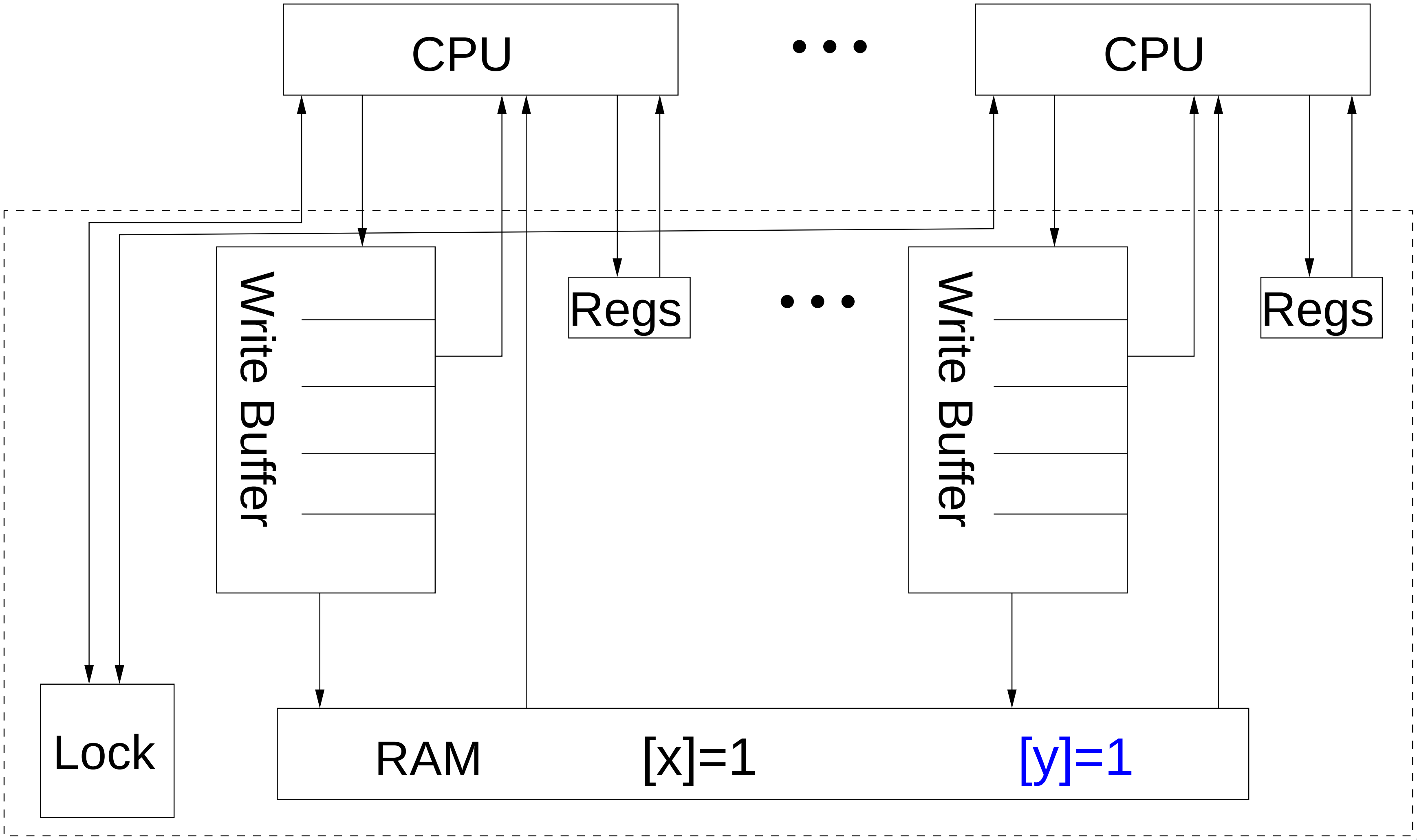
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[y] (0)	MOV [y]←$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



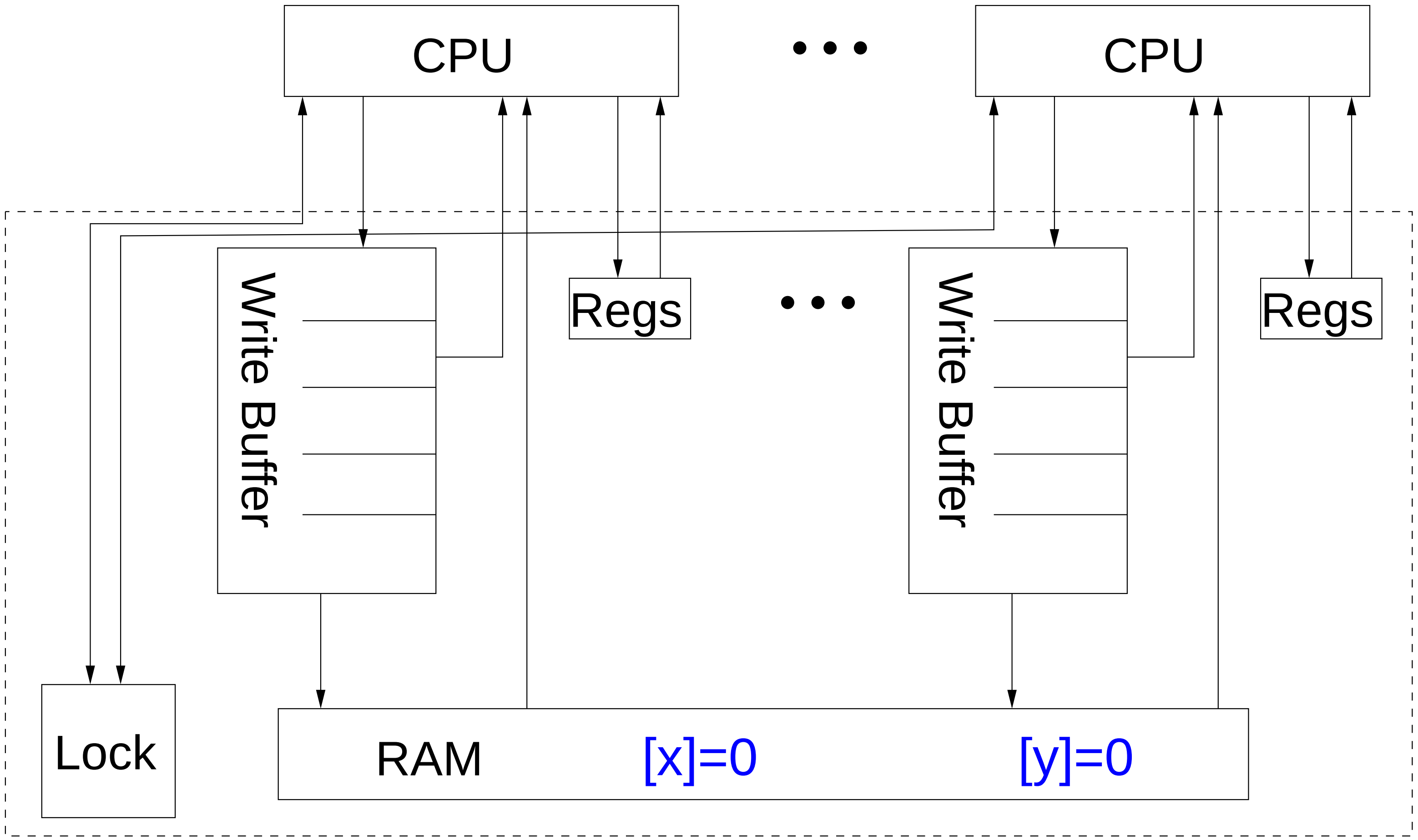
Litmus Test Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[y] (0)	MOV [y]←\$1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	



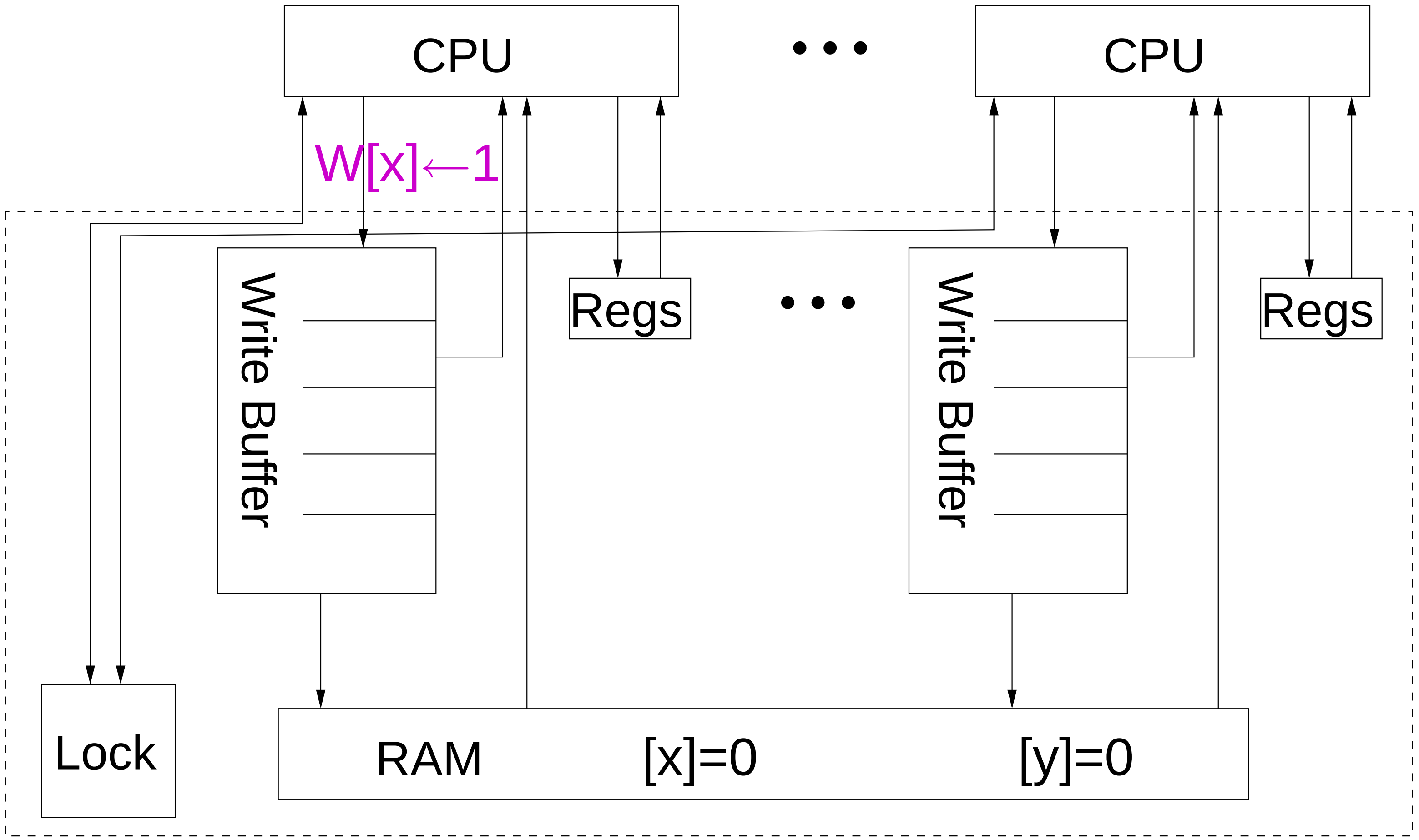
CC Unsound Example

Initial: $[x]=0 \land [y]=0$	
MOV $[x] \leftarrow \$1$ MOV $EAX \leftarrow [x]$ (1) MOV $EBX \leftarrow [y]$ (0)	MOV $[y] \leftarrow \$2$ MOV $[x] \leftarrow \$2$
Final: $EAX=1 \land EBX=0 \land [x]=1$	
cc : Forbid; tso : Allow	



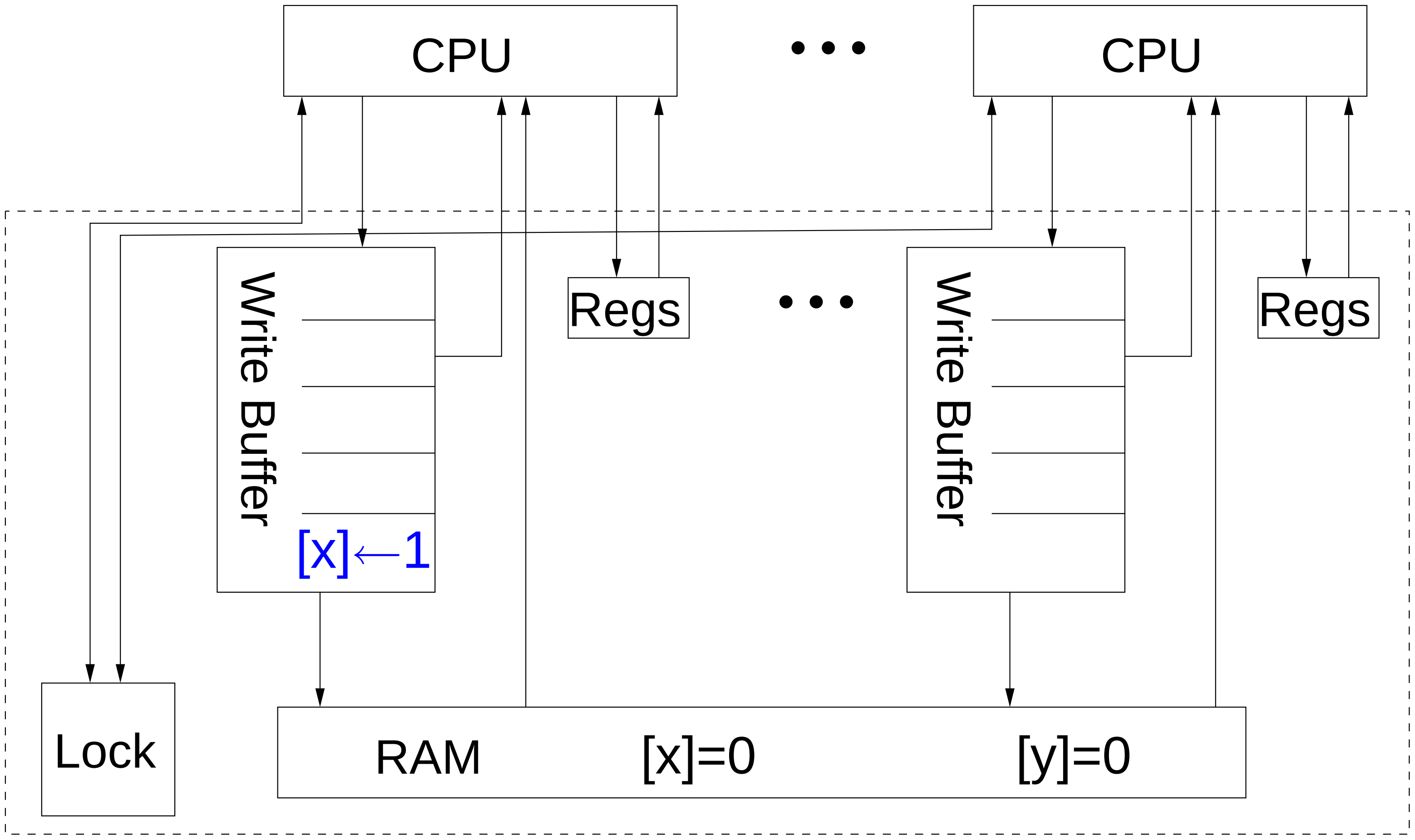
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←\$2 MOV [x]←\$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



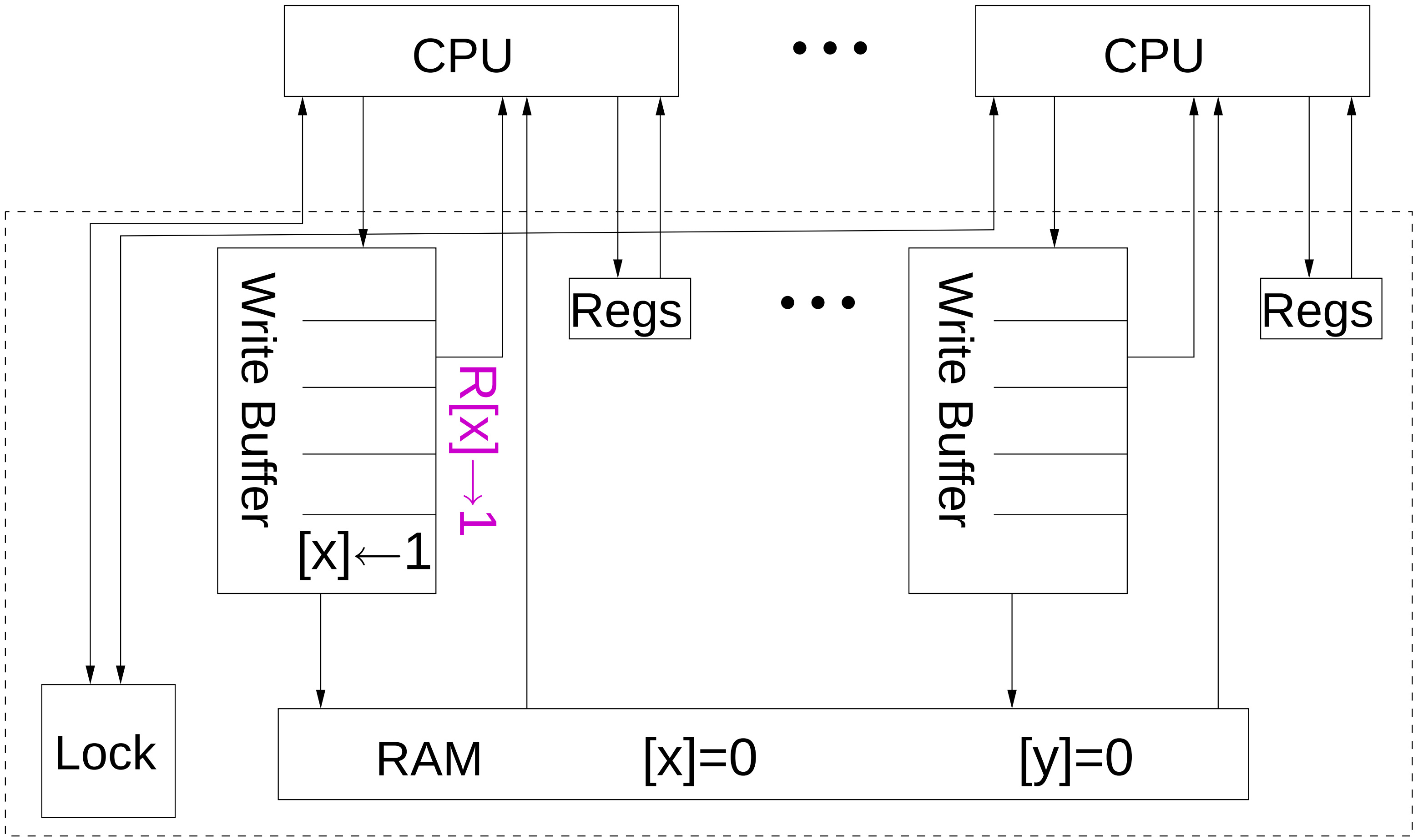
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



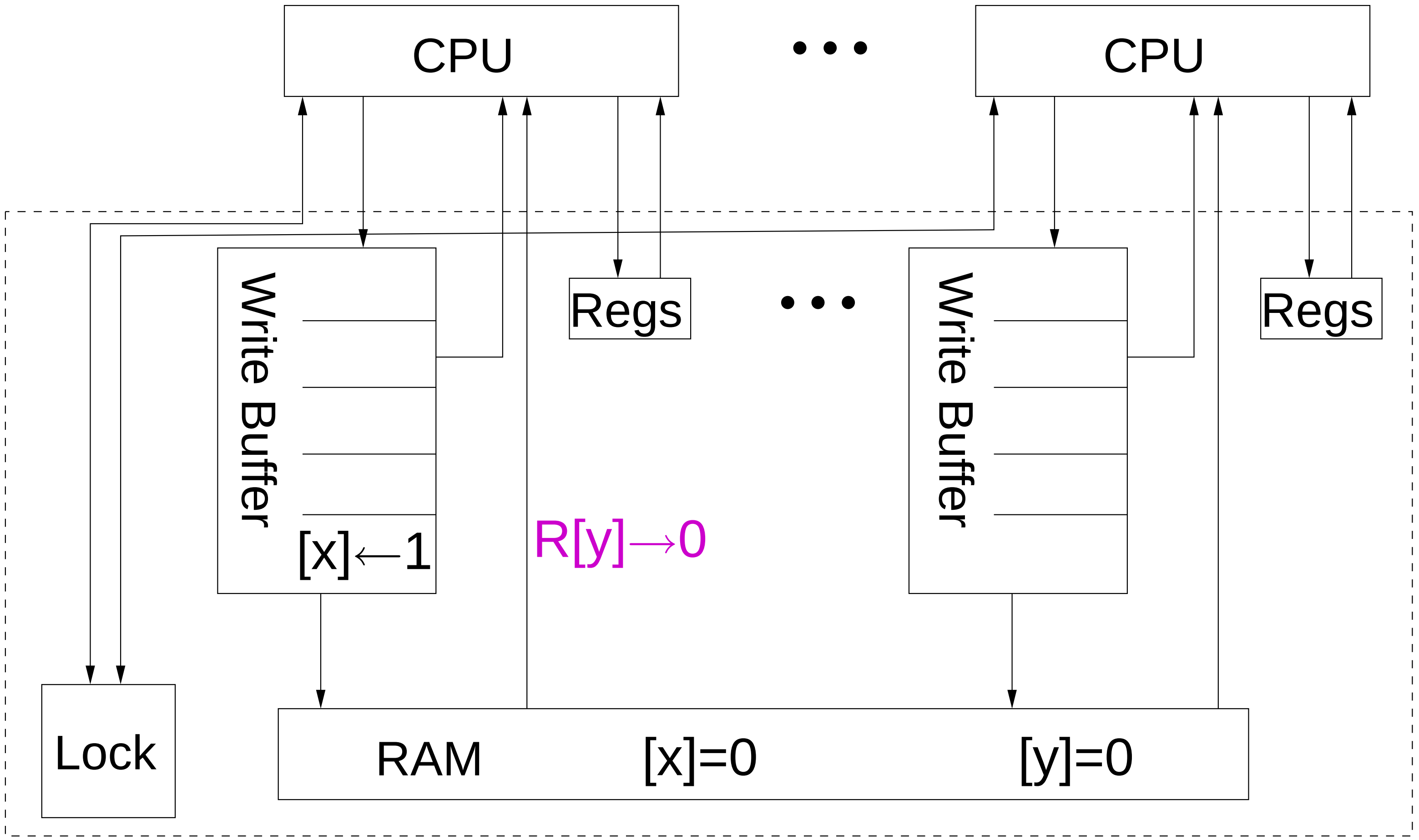
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



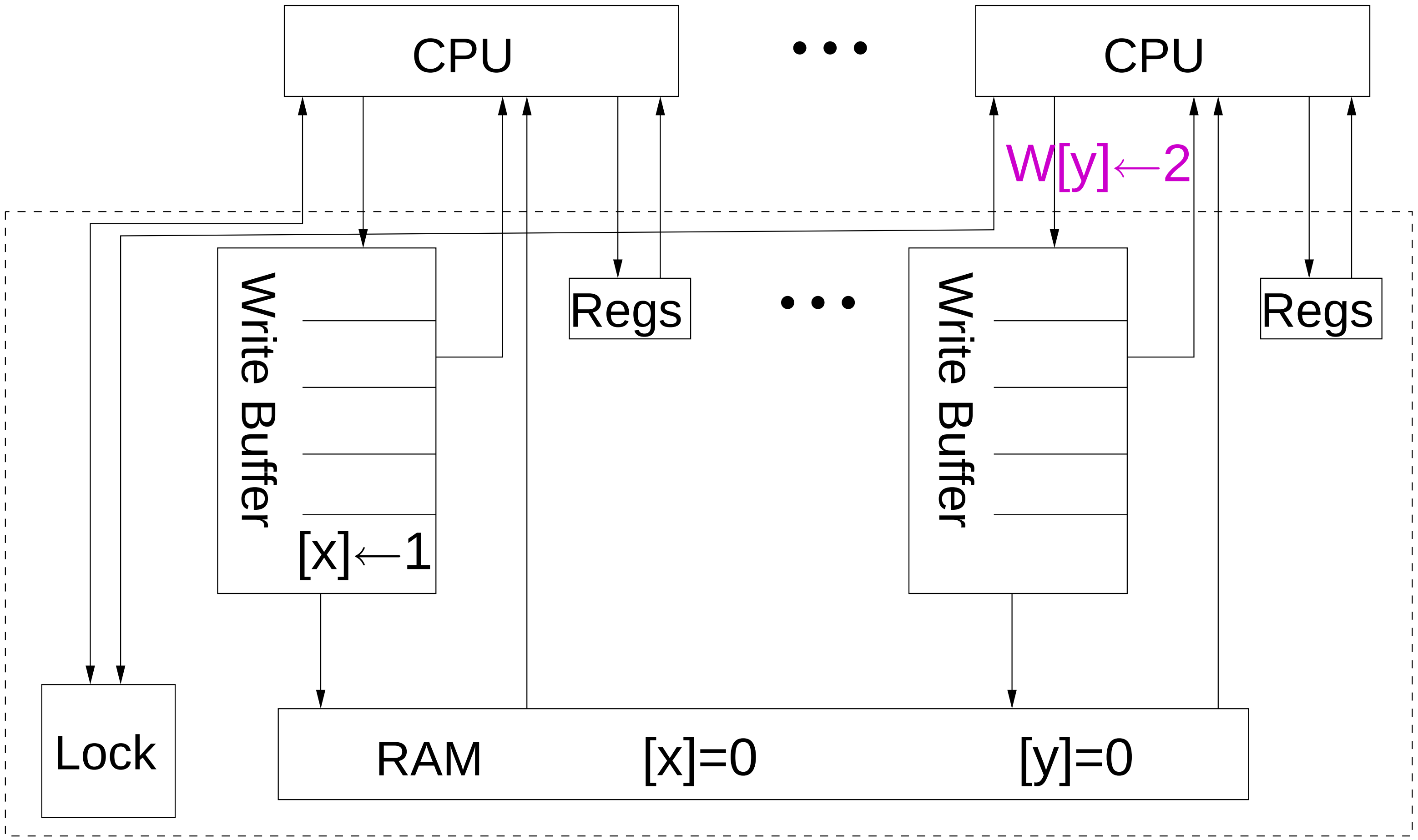
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



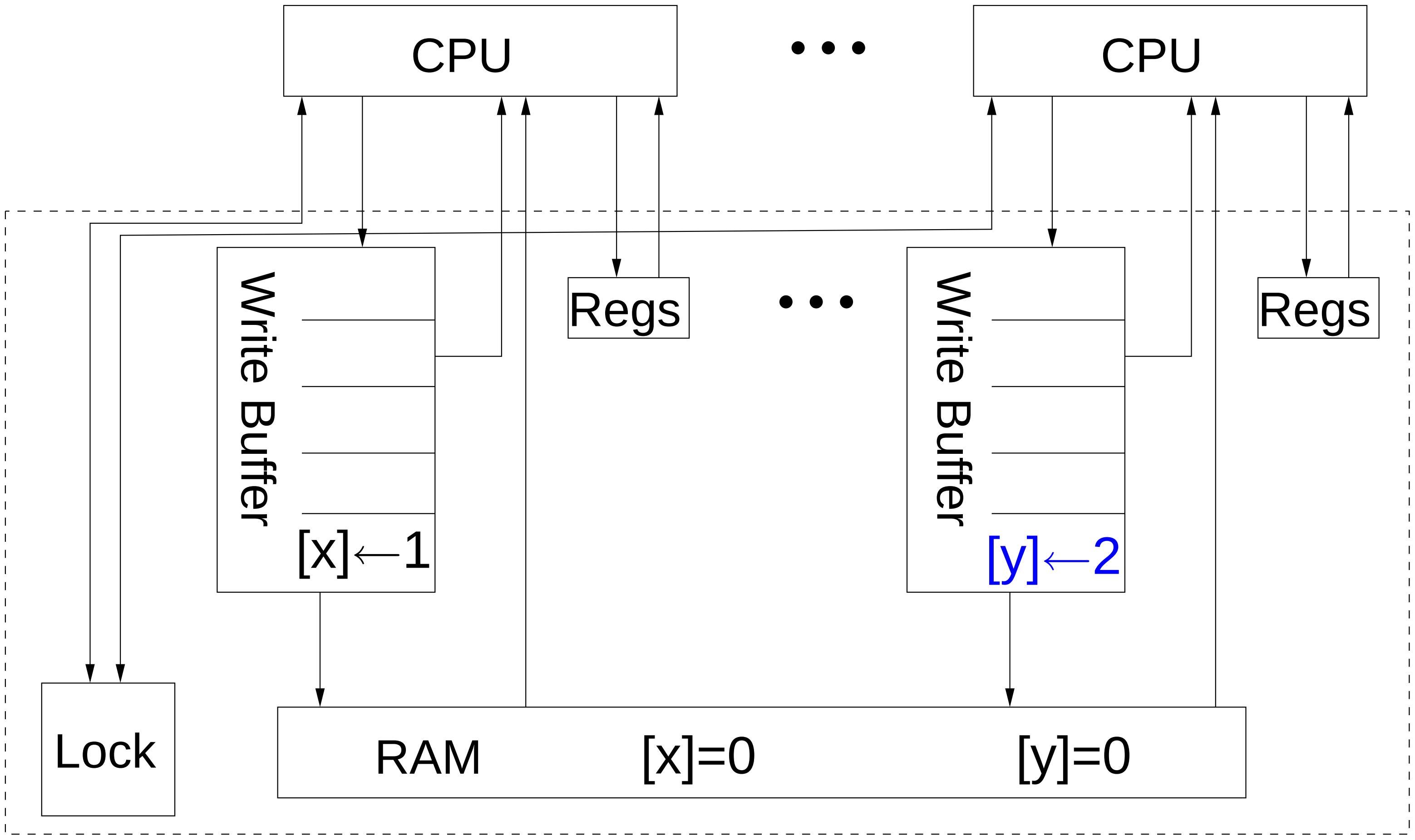
CC Unsound Example

Initial: [x]$=$0 $\wedge$ [y]$=$0	
MOV [x]$\leftarrow$\$1 MOV EAX$\leftarrow$[x] (1) MOV EBX$\leftarrow$[y] (0)	MOV [y]$\leftarrow$\$2 MOV [x]$\leftarrow$\$2
Final: EAX$=$1 $\wedge$ EBX$=$0 $\wedge$ [x]$=$1	
cc : Forbid; tso : Allow	



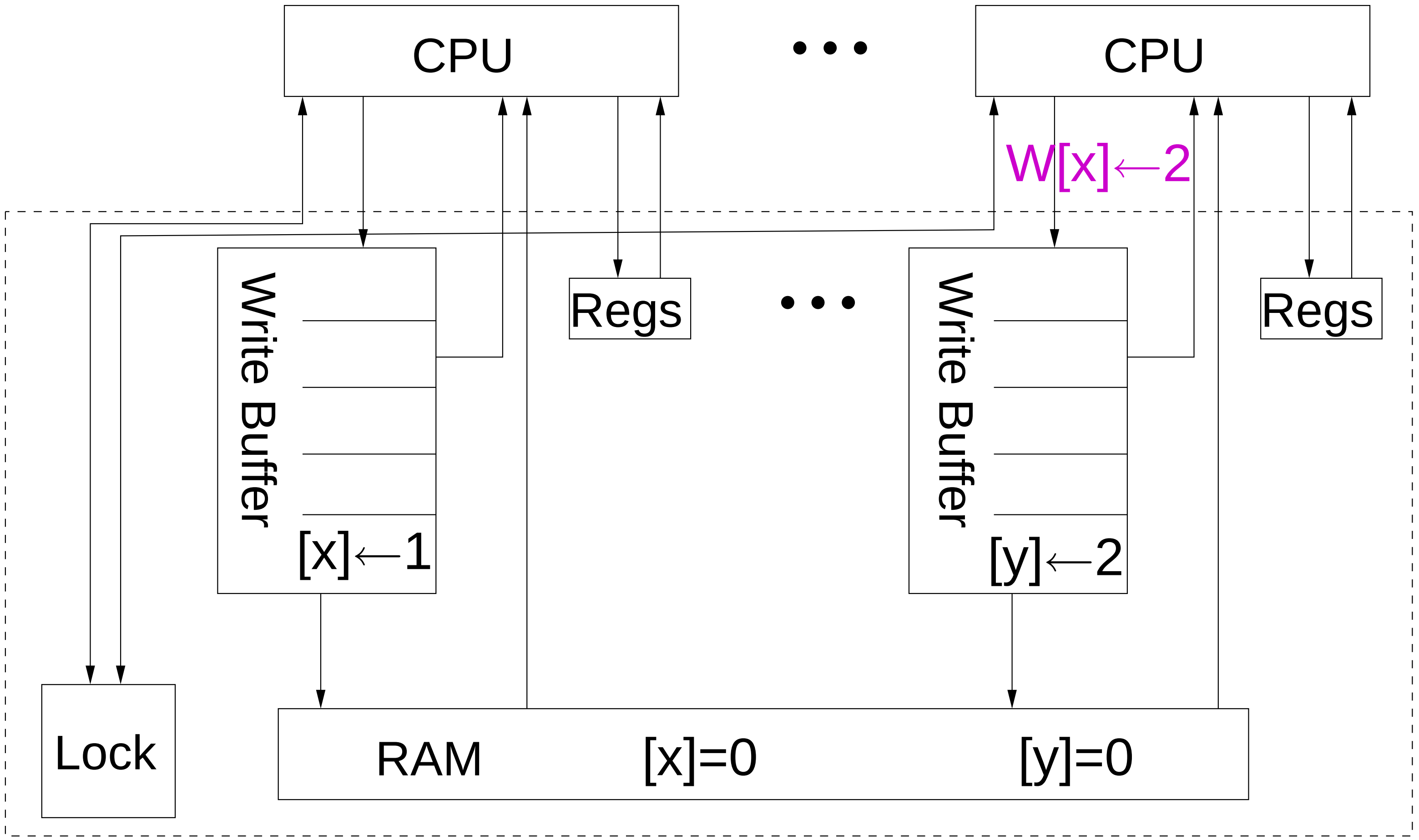
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←\$2 MOV [x]←\$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



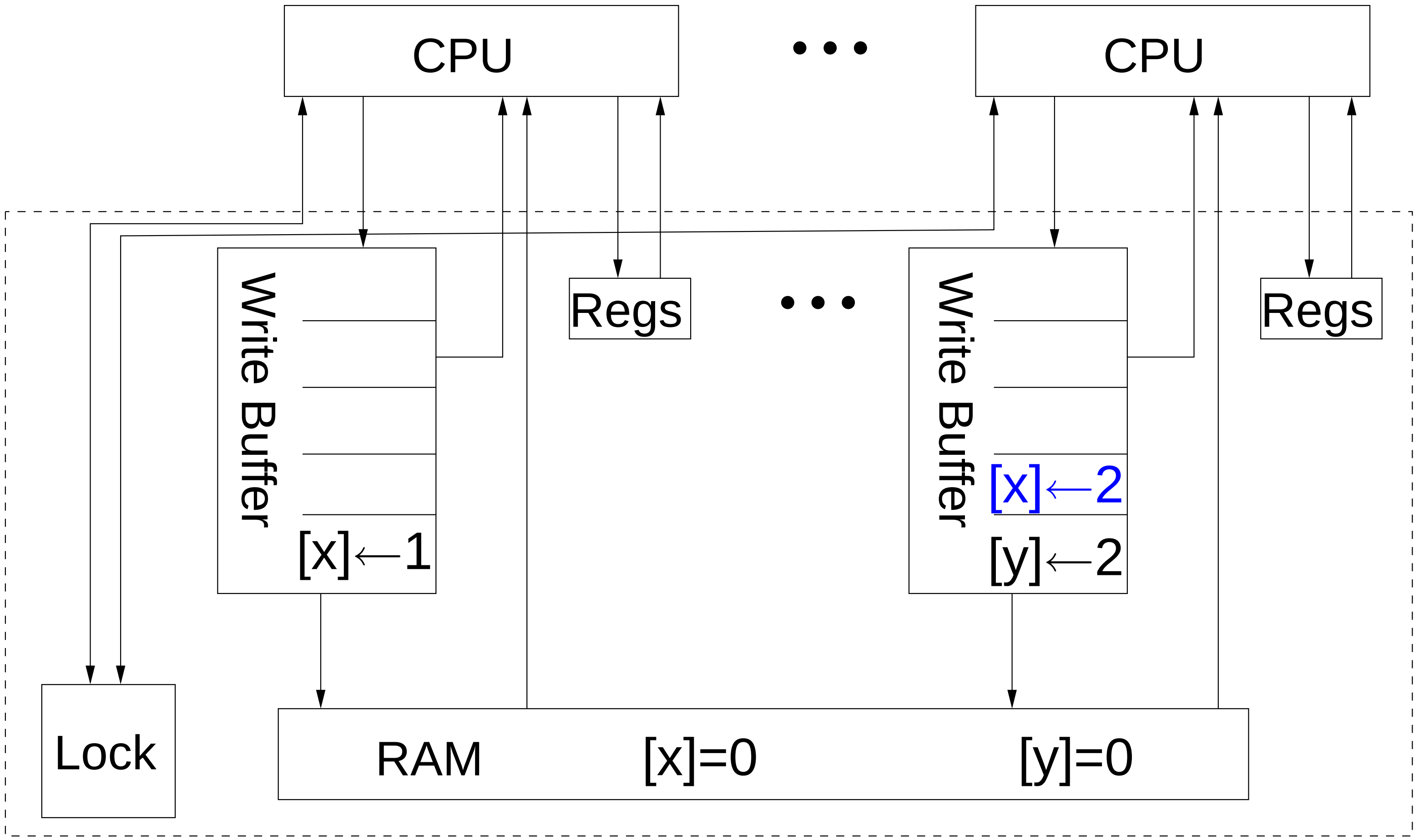
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



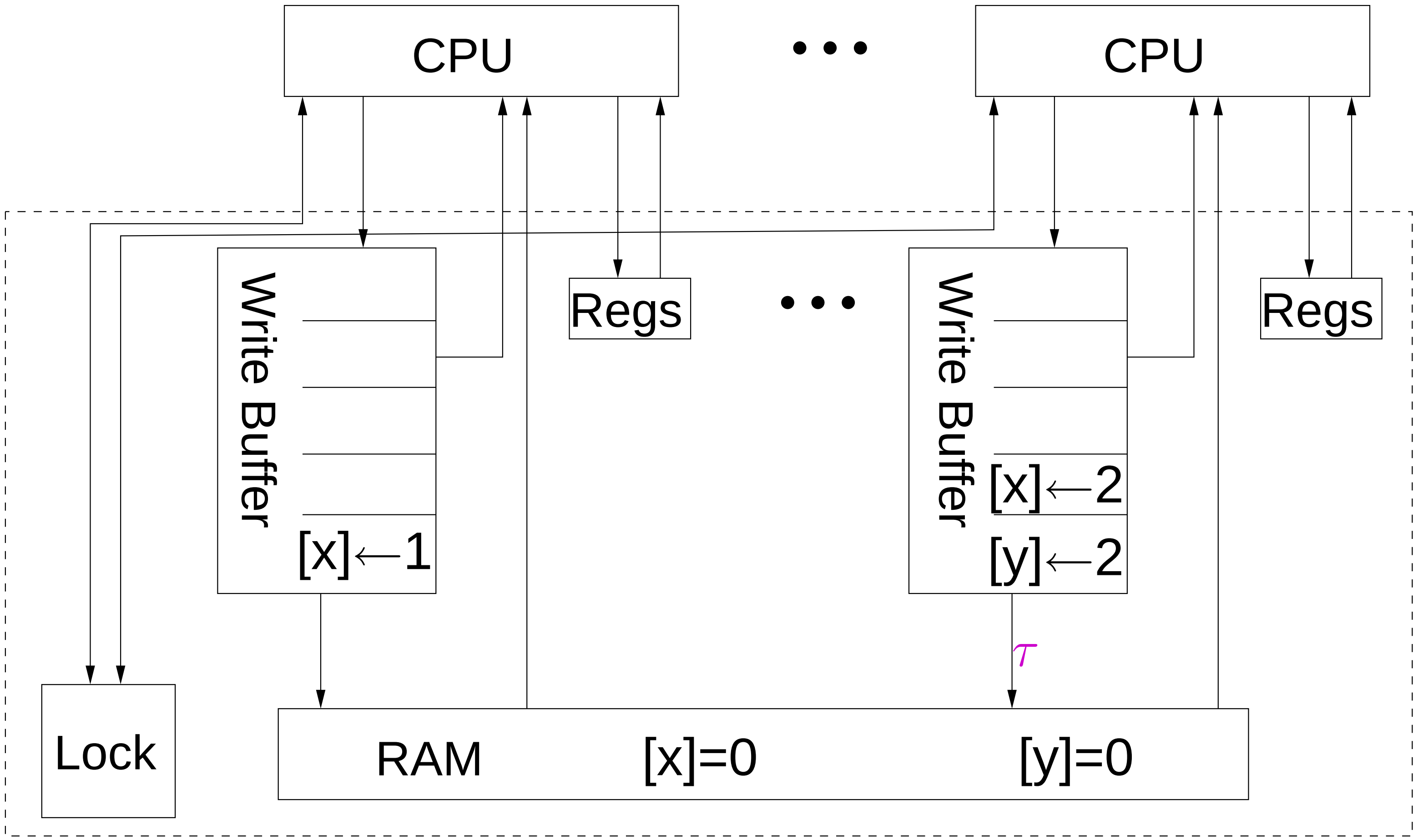
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



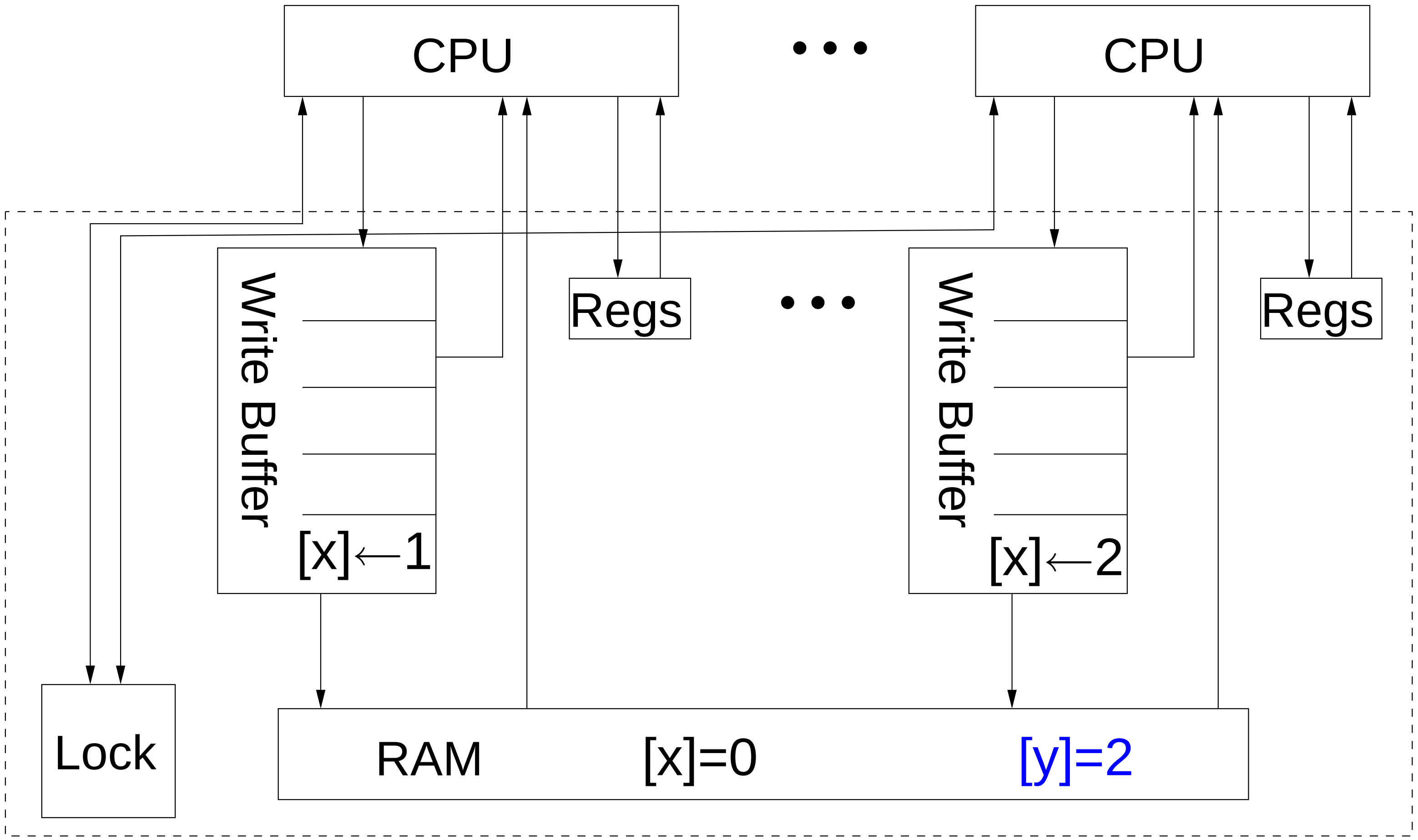
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←\$2 MOV [x]←\$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



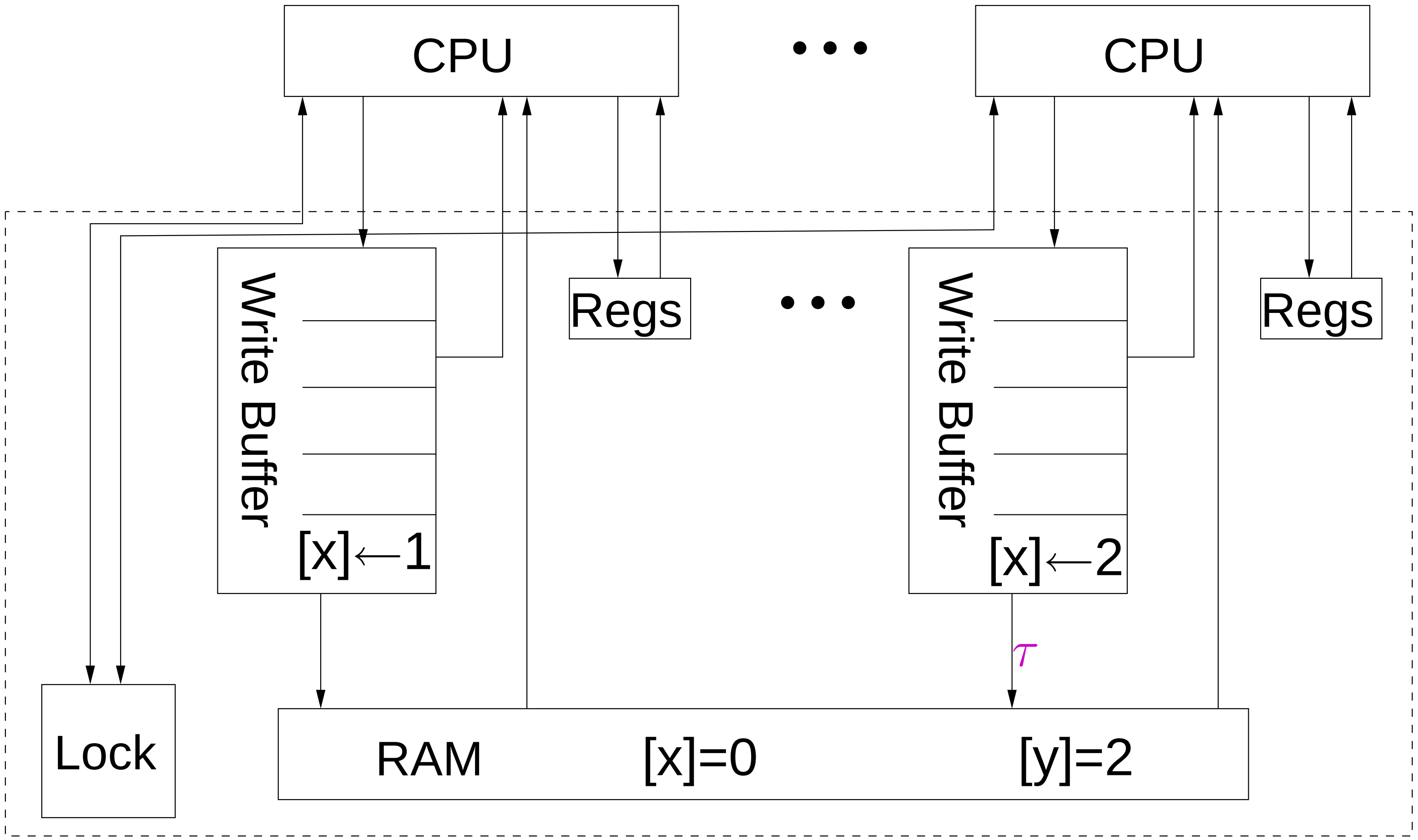
CC Unsound Example

Initial: [x]=0 $\wedge$ [y]=0	
MOV [x]$\leftarrow$\$1	MOV [y]$\leftarrow$\$2
MOV EAX$\leftarrow$[x] (1)	MOV [x]$\leftarrow$\$2
MOV EBX$\leftarrow$[y] (0)	
Final: EAX=1 $\wedge$ EBX=0 $\wedge$ [x]=1	
cc : Forbid; tso : Allow	



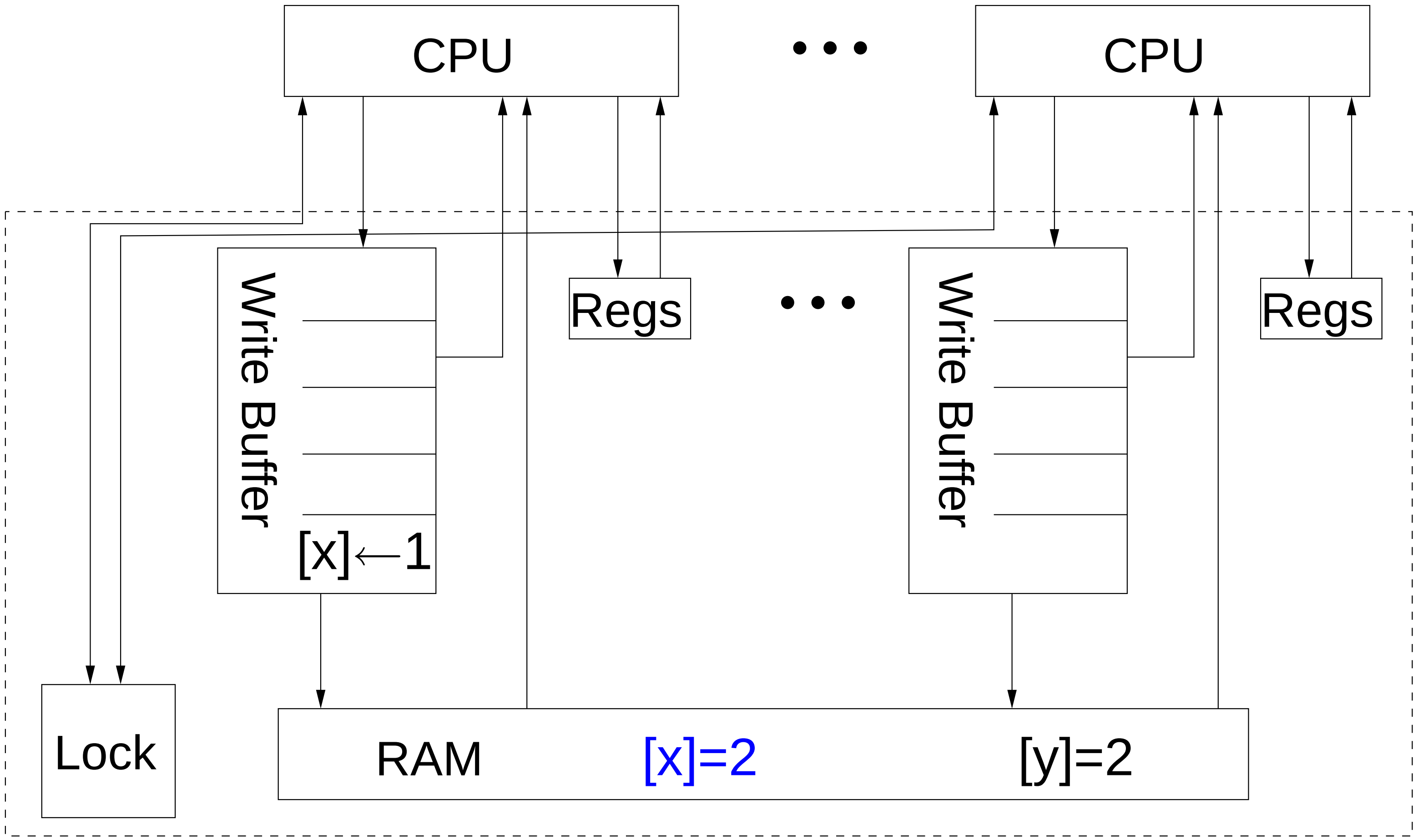
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←\$2 MOV [x]←\$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



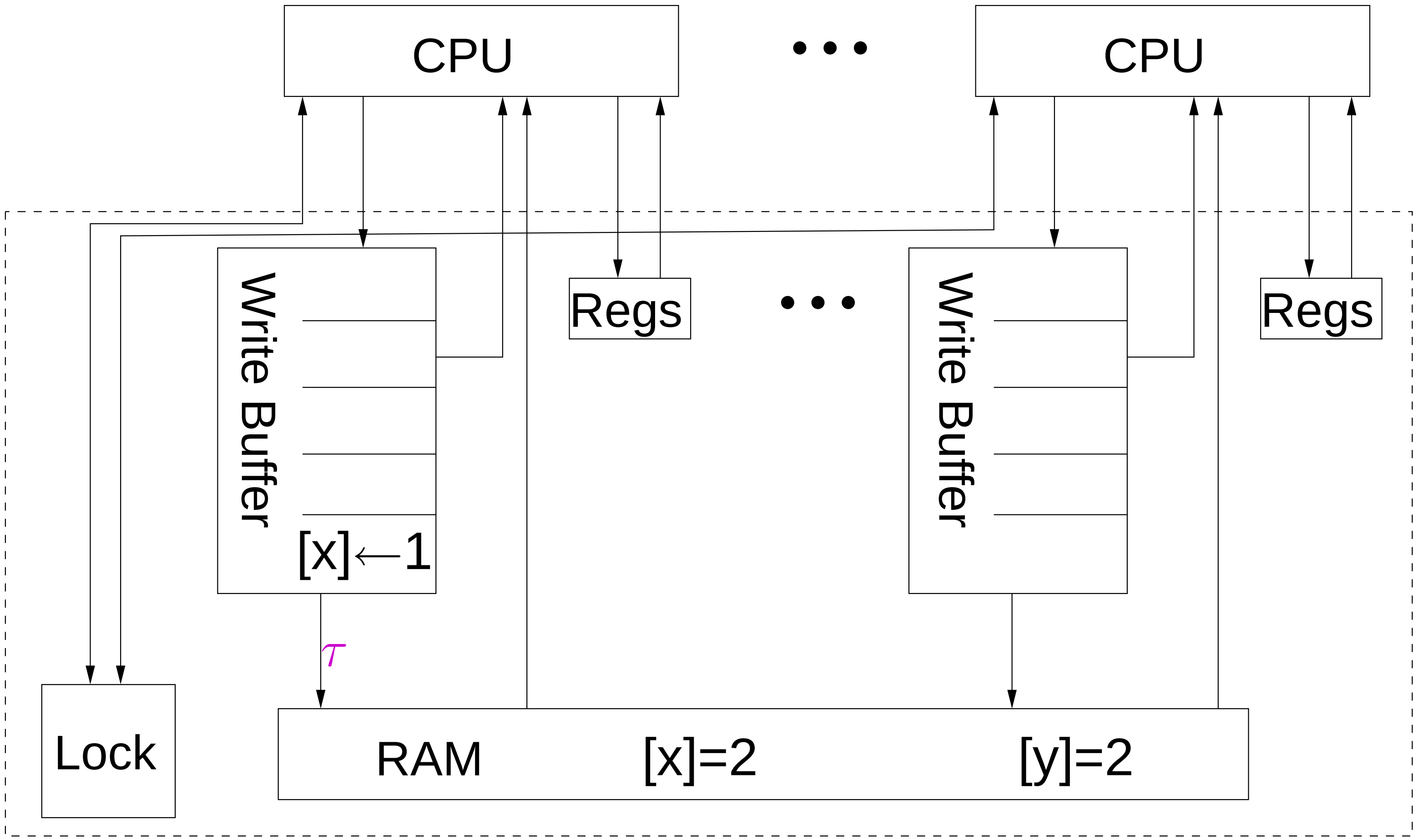
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←\$1	MOV [y]←\$2
MOV EAX←[x] (1)	MOV [x]←\$2
MOV EBX←[y] (0)	
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



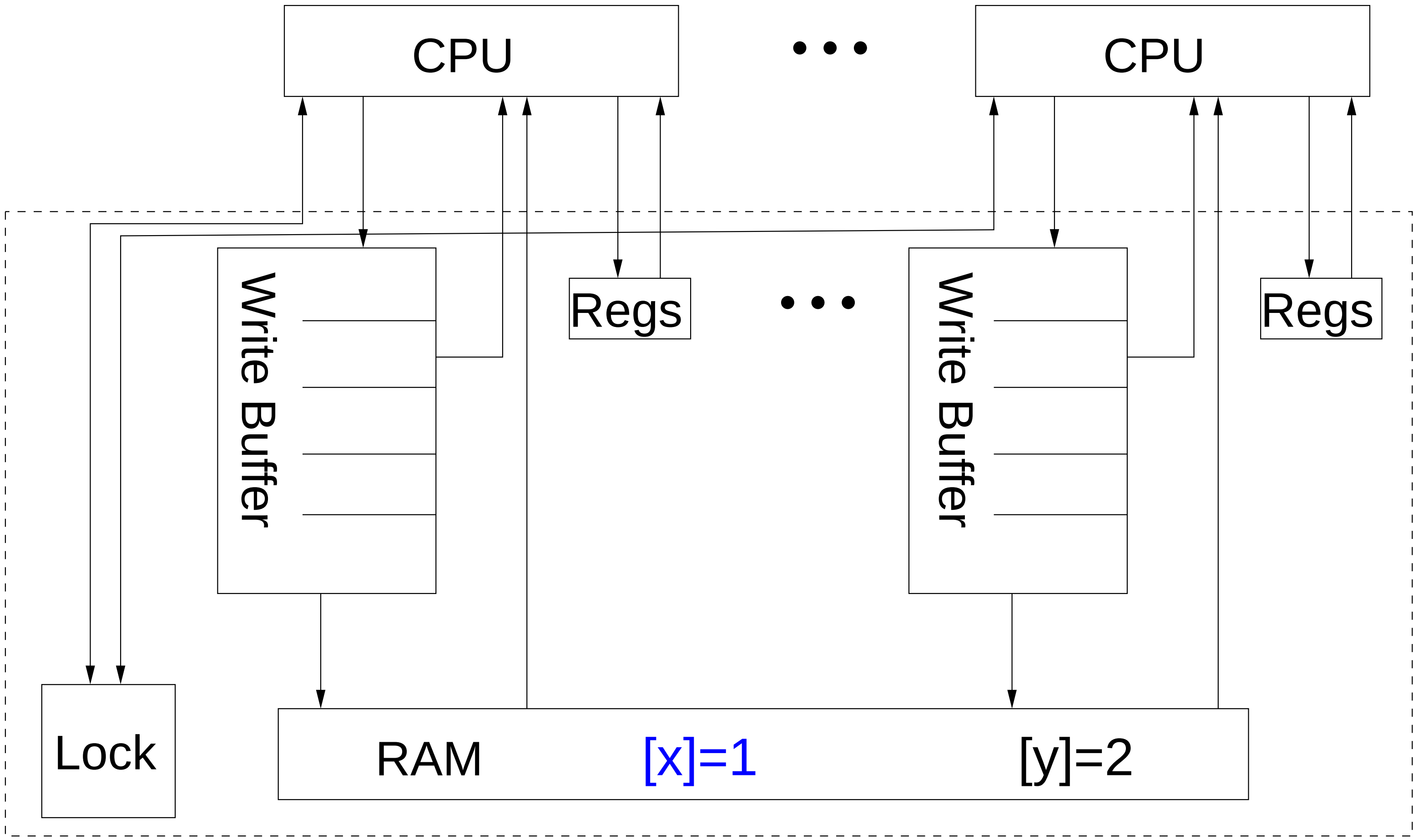
CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



CC Unsound Example

Initial: [x]=0 ∧ [y]=0	
MOV [x]←$1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←$2 MOV [x]←$2
Final: EAX=1 ∧ EBX=0 ∧ [x]=1	
cc : Forbid; tso : Allow	



CPU
• • •
CPU
Write Buffer
Regs
• • •
Write Buffer
Regs
Lock
RAM
[x]=1
[y]=2