ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures

Daniel Lustig Caroline Trippel Michael Pellauer* Margaret Martonosi
Princeton University *NVIDIA Research[1]
{dlustig,ctrippel,mrm}@princeton.edu mpellauer@nvidia.com

Abstract

Architectural heterogeneity is increasing: numerous products and studies have proven the benefits of combining cores and accelerators with varying ISAs into a single system. However, an underappreciated barrier to unlocking the full potential of heterogeneity is the need to specify and to reconcile differences in memory consistency models across layers of the hardware-software stack and among on-chip components.

This paper presents ArMOR, a framework for specifying, comparing, and translating between memory consistency models. ArMOR defines MOSTs, an architecture-independent and precise format for specifying the semantics of memory ordering requirements such as preserved program order or explicit fences. MOSTs allow any two consistency models to be directly and algorithmically compared, and they help avoid many of the pitfalls of traditional consistency model analysis. As a case study, we use ArMOR to automatically generate translation modules called shims that dynamically translate code compiled for one memory model to execute on hardware implementing a different model.

1. Introduction

Recent computing trends demonstrate a dramatic shift away from homogeneous multicores and towards increasing microarchitectural and architectural heterogeneity [18, 24, 48, 56, 63]. The GPGPU paradigm is one popular example, but the degree of heterogeneity used in practice has quickly moved beyond just GPUs. A current mobile system-on-chip (SoC) contains as many as five or six instruction sets among its CPUs, GPUs, DSPs, and accelerators [49]. This ISA diversity extends beyond the choice of opcodes; a modern SoC may likewise contain as many as five or six hardware memory consistency models, and each hardware component may serve as a target for any number of software memory models.

Memory model heterogeneity presents a number of challenges: how to compile from a given software model onto a given hardware model, how to design memory model-aware intermediate representations (e.g., LLVM IR, NVIDIA PTX), how to dynamically migrate code from one ISA to another, and so on. Due to the complexity of even properly defining memory models such as those used by C/C++11, Java, ARM, or Power, and due to the incompatible manner in which memory models are often defined, any solution to one of the above problems (e.g., any particular compiler) can only be built in a reliably correct manner through the use of formal analysis [3, 12, 15, 34, 40, 41, 51]. Informal analyses are prone to being either overconstrained or simply incorrect.

We present ARchitecture-independent Memory Ordering Requirements (ArMOR), a framework for specifying, reasoning about, and translating between memory consistency models. ArMOR defines memory ordering requirements (MORs) (fences, dependencies, or any other ordering enforcement mechanisms) in a self-contained, complete, and precise format known as a memory ordering specification table (MOST). MOSTs resemble standard reordering tables which indicate, e.g., whether load$\rightarrow$load, load$\rightarrow$store, store$\rightarrow$load, and store$\rightarrow$store orderings need to be maintained. The key contribution of MOSTs is that they also directly encode subtle details such as store multiple-copy atomicity, fence cumulativity, and so on. This added precision makes MOST-based analysis less prone to the types of under- or overconstraints that can result from relying on less systematic techniques.

As a case study which demonstrates the precision and flexibility of ArMOR, we use MOSTs to automatically derive self-contained translation modules called *shims* which dynamically adapt code compiled for one memory model to execute on hardware implementing another, *without recompilation or offline code analysis*. Shims operate by dynamically injecting fences or other enforcement mechanisms as needed into a code stream. Other use cases could include removing redundant fences to optimize performance or using programmable shims as a means of allowing memory-accessing IP blocks to be built independently of the reordering properties of the underlying infrastructure (e.g., the network-on-chip). In the future, we envision shims being used in JIT compilers [32, 44], dynamic

[1]Work performed while with VSSAD Group, Intel Corporation.

ISCA'15, June 13-17, 2015, Portland, OR USA
©2015 ACM. ISBN 978-1-4503-3402-0/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750378

	Loads	Stores
Loads	✓	✓
Stores	—	✓
Stores are multiple-copy atomic		

TSO PPO

	Loads	Stores
Loads	✓	✓
Stores	—	✓
A- and B-cumulative		

Power `lwsync`

(a) Traditional reordering tables for TSO PPO and Power `lwsync`. The two appear deceivingly similar.

Core 0	Core 1	Core 2	Core 3
mov [x], 1	mov rax, [x] mov rbx, [y]	mov rcx, [y] mov rdx, [x]	mov [y], 1
Outcome rax=1, rbx=0, rcx=1, rdx=0 **forbidden**			

`iriw` litmus test on x86

Core 0	Core 1	Core 2	Core 3
stw [x], 1	lwz r1, [x] lwsync lwz r2, [y]	lwz r3, [y] lwsync lwz r4, [x]	stw [y], 1
Outcome r1=1, r2=0, r3=1, r4=0 **observable**			

`iriw+lwsyncs` litmus test on Power

(b) The `iriw` litmus test on x86-TSO and Power. The outcome is observable on Power but not x86-TSO, indicating that the orderings enforced by TSO PPO and by `lwsync` differ.

	PO Ld	PO St
PO Ld	✓	✓
PO St	—	$\checkmark_M$
AC St	$\checkmark_M$	$\checkmark_M$

Partial TSO PPO MOST

	PO Ld	PO St
PO Ld	✓	✓
PO St	—	$\checkmark_N$
AC St	—	$\checkmark_N$

Partial `lwsync` MOST

(c) MOSTs add enough precision that the differences become clear. See Section 3 for details.

Figure 1: When specification formats differ, it can be unclear whether ordering requirements of different architectures are equivalent. In this example, the tables match, leading even experts prone to the pitfall of assuming TSO PPO and Power `lwsync` are equal [58]. However, multiple-copy atomicity and cumulativity differ in subtle but important ways (as seen with `iriw`).

	memory_ order_ relaxed	memory_ order_ release	memory_ order_ seq_cst
x86	mov	mov	xchg
Power	st	lwsync; st	sync; st
ARMv7	st	dmb; st	dmb; st
ARMv8	str	stl	stl
Itanium	st.rel	st.rel	st.rel; mf

Table 1: Mapping C11 low-level atomics with different ordering specifications onto hardware. The software constructs map onto different architectures in different ways [53].

binary translators [20, 65], dynamic code optimizers [43], and updates to reprogrammable microcode [30], with the latter used to fix implementation bugs [4].

When memory models are sufficiently compatible, we demonstrate that the performance overhead of implementing optimized ArMOR translation in hardware can be as low as 10-77%. Overall, our experiences with ArMOR and translation can inspire the designs of future ISAs to be truly portable across hardware memory models, and they can inspire future architectures in how to define and implement a set of memory ordering primitives that serve as a suitable back-end.

The rest of this paper is arranged as follows. Section 2 begins with a motivating example. Section 3 describes ArMOR's specification syntax, and Section 4 describes how to manipulate and compare MOSTs. Section 5 uses MOSTs to derive inter-MCM translation shims. Our experimental methodology and performance analysis results are given in Sections 6 and 7. Section 8 describes related work, and Section 9 concludes.

2. Motivating Example

Although many programmers write parallel code under the assumption of sequential consistency (SC), few software or hardware models today directly implement SC due to its performance cost. As a result, application programmers or library writers must explicitly specify additional consistency-related synchronization points, whether at coarse grain (e.g., function call or GPGPU kernel boundaries), medium grain (e.g., mutex operations), or fine grain (e.g., C11 low-level atomics or inline assembly). One key challenge in each case is determining how to map a given software synchronization primitive onto a sufficiently strong hardware primitive in the target architecture.

Table 1 shows how three flavors of C11 atomic store orderings map onto different architectures in very different ways. The corresponding table for loads is even more diverse, enforcing orderings through the use of features such as explicit dummy dependencies. Such decisions are sometimes shielded from application programmers writing in high-level languages, but they represent very real complexity for library and compiler writers. Unfortunately, the current most reliable method for determining such mappings requires the construction of complicated formal models and dense mathematical correctness proofs which may take years to complete [12]. In the meantime, programmers are forced to rely on bug-prone intuitive analysis to select primitives.

Figure 1 highlights some of what makes consistency models complicated. Figure 1a depicts a commonly-used manner of describing the total store ordering (TSO) consistency model used by SPARC and x86. In particular, this figure depicts *preserved program order* (PPO)—the set of orderings always enforced by default on a given architecture. The table in Figure 1a specifies whether an access of one type (the row heading) may be reordered with a subsequent access of another type (the column heading). Under TSO, stores may be reordered with later loads, but all other orderings are required. This figure also shows how `lwsync`, a fence on the Power architecture, can be defined in a similar way for orderings between operations before and after the fence.

The rest of Figure 1 describes how ordering specifications which appear similar on the surface may nevertheless differ in very subtle ways that make intuitive reasoning difficult. For example, consider the problem of mapping x86-TSO code onto the Power architecture [52]. Memory accesses on Power are reordered liberally by default; orderings on Power are only enforced through inter-instruction dependencies or explicit fences. Given the commonly-used tables in Figure 1a, it may appear that insertion of `lwsync` between every pair of accesses should be sufficient to restore all of the orderings required by TSO. However, this appearance is deceiving, as the two are in fact *not* equivalent.

The difference in strength between the default orderings of TSO and the orderings enforced by `lwsync` can be demonstrated explicitly by a litmus test called `iriw` (independent reads of independent writes), shown in Figure 1b. In particular, although TSO enforces orderings between the Core 0 store to `[x]` and the Core 1 load of `[y]` and between the Core 3 store to `[y]` and the Core 2 load of `[x]`, `lwsync` does not.

ArMOR avoids the pitfall of Figure 1a by improving the precision of the reordering tables themselves. We call these enhanced ordering tables *MOSTs*, or memory ordering specification tables. A partial example is shown in Figure 1c. Each cell in a MOST lists not just an ordering, but also the *strength* of the ordering (i.e., whether it is single-copy atomic, multiple-copy atomic, or neither). New rows and columns are introduced to directly address ordering enforced with respect to remote cores (as required by `iriw` above). The details of these new features are elaborated in Section 3. The highlight, however, is that by comparing cell-to-cell, the MOSTs clearly show that TSO PPO enforces more orderings than `lwsync`.

The ArMOR approach has numerous potential uses moving forward. As one example, we would use ArMOR directly in compiler backends in an effort to solve compilation problems such as the above. However, we leave the analysis of software constructs as future work. In this paper, we instead focus on the related problem of translating directly from one hardware model to another through the use of a *shim*. This case study fits within the hardware heterogeneity focus of the current paper, and it complements recent cross-ISA migration studies which focused on microarchitectural differences and memory layouts rather than consistency models [20, 65].

3. Memory Ordering Specification Tables

Memory ordering specification tables (MOSTs) describe the reordering behavior of memory consistency models at a precise and detailed level sufficient to support algorithmic analysis and automated comparisons and translation. Just as with traditional reordering tables, each cell in a MOST specifies whether instructions of the type in the row heading must maintain their ordering with subsequent instructions of the type in the column heading. Traditional reordering tables are most often used to define *preserved program order*, the set of orderings which are enforced by default. In contrast, we use MOSTs to define not just preserved program order, but also fences or any other type of ordering enforcement mechanism.

As two running examples, we will derive the MOSTs for TSO preserved program order and for Power `lwsync` step by step. Both were partially shown earlier in Figure 1c. The complete MOSTs will be given at the end of this section once all of the necessary notation and details have been presented.

Core 0	Core 1	Core 2
st [x], 1	ld r1, [x] fence st [y], r1	ld r2, [y] fence ld r3, [x]
Outcome r1=1, r2=1, r3=0: **Forbidden** if stores are single-or multiple-copy atomic **Allowed** if stores are non-multiple-copy-atomic		

Figure 2: The `wrc` litmus test with non-cumulative fences.

(a) Store→store

Abb.	Description
$\checkmark_S$	Single-copy atomic
$\checkmark_M$	Multiple-copy atomic
$\checkmark_N$	Non-atomic
—	Unordered

(b) Store→load

Abb.	Description
✓	Ordered
$\checkmark_L$	Locally ordered
—	Unordered

(c) Other

Abb.	Descrip.
✓	Ordered
—	Unordered

Figure 3: MOST strength levels used in this paper.

3.1. Store Atomicity

The first imprecision of traditional reordering tables is the fact that they do not address how orderings may have different *strengths*. In particular, stores may in general *perform* with respect to (i.e., become visible to) different cores in a system at different times. *Single-copy atomic* stores must become visible to all cores in the system at a single time [9]. Single-copy atomicity is uncommon, as it forbids even forwarding from a private local store buffer. *Multiple-copy atomic* stores must become visible to all cores *besides the issuing core* simultaneously. In other words, a multiple-copy atomic store cannot ever be visible to some but not all remote cores. TSO (used by SPARC and x86) falls into this category. *Non-multiple-copy-atomic* stores may become visible with respect to remote cores in any order and in any number of steps. Power and ARM fall into this category.

The effect of store atomicity (or a lack thereof) is commonly depicted by the write-to-read causality litmus test (`wrc`) of Figure 2. This test works as follows. If the core 1 load reads the value written by the core 0 store and then forwards it along to core 2, must core 2 have also seen the effect of the core 0 store? If the store from core 0 is multiple- or single-copy atomic, then core 2 must see the core 0 store before it sees the core 1 store. However, if the store from core 0 is not multiple-copy atomic, then the core 1 store may propagate to core 2 *before* the core 0 store does, *even though the core 0 store executed first*. This violates the intuitive notion of causality: even though the core 0 store *causes* the core 1 store value to exist, the core 0 store need not become visible to other cores before the core 1 store. Nevertheless, this execution remains a legal outcome on non-multiple-copy-atomic architectures.

To account for such strength differences in an architecture-independent manner, we introduce various *strength levels* into our MOST notation. Figure 3 summarizes the ordering strength levels used to describe MORs for architectures sur-

	Ld	St
Ld	✓	✓
St	—	$\checkmark_M$

(a) TSO (partial)

	Ld	St
Ld	✓	✓
St	—	$\checkmark_S$

(b) IBM 370/390/zSeries (partial)

Figure 4: The addition of explicit strength levels allows MOSTs to distinguish cases that would appear identical using traditional reordering tables.

Core 0	Core 1	Core 2
st [x], 1 ld r1, [x] ld r2, [y]	st [y], 1	ld r3, [y] ld r4, [x]
Outcome: r1=r3=1, r2=r4=0: Allowed		

Figure 5: TSO litmus test `n7` [45]. Although the first two instructions from core 0 access the same address, that store→load same-address ordering is *not* enforced from the point of view of other observers.

veyed in this paper. Additional (e.g., scoped) strength levels could easily be added if necessary.

As an example of the benefit of these strength levels, Figure 4 shows partial MOSTs for the TSO and IBM 370/390/zSeries memory models. With traditional reordering tables, the architectures would appear equivalent. With the improved precision of MOSTs, the difference in store→store ordering strength is made explicit.

3.2. Per-Address Orderings

Accesses from the same thread to the same address generally must maintain the ordering specified by program order. This property is sometimes called *coherence*[2]. There are exceptions; SPARC RMO and old Power models relax load→load orderings to the same address, while the behavior is forbidden yet observable on some GPUs [4, 7, 62, 59]. To address this in MOSTs, we explicitly distinguish accesses to the same address (SA) from those to different addresses (DA).

The notion of ordering strength from the previous subsection is also relevant to per-address orderings. In particular, a store→load ordering may need to be enforced locally to ensure that each load returns the value written by the latest store to the same address. However, the same store→load ordering may *not* need to be enforced from the point of view of any remote observers. This is highlighted in Figure 5. In this example, the core 2 load of `[x]` can occur after the core 0 load of `[x]` but before the core 0 store to `[x]` becomes visible to core 2. In other words, from the point of view of core 2, the core 0 store happens after the core 0 load of `[x]`. This highlights the need not just to specify that orderings must be enforced, but also to precisely specify their strength.

This amount of detail is enough to complete the MOST for TSO PPO, as shown in Figure 6a. In particular, store→store ordering has been marked as being multiple-copy atomic, and

[2]Coherence protocols often use stronger definitions of coherence (e.g., single writer or multiple readers), while other consistency model papers may use weaker notions such as total orders only on stores to the same address.

	Load to Diff. Address	Load to Same Address	Store
Load	✓	✓	✓
Store	—	$\checkmark_L$	$\checkmark_M$

(a) TSO PPO

	Load to Diff. Address	Load to Same Address	Store to Diff. Address	Store to Same Address
Load	—	—	—	✓
Store	—	$\checkmark_L$	—	$\checkmark_M$

(b) RMO PPO

Figure 6: Complete MOSTs for TSO and RMO PPO.

Core 0	Core 1	Core 2
(i) st [x], 1	(ii) r1 = ld [x] (iii) sync (iv) st [y], 2	(v) r2 = ld [y] (vi) st [y], 3

If the outcome is r1=1, r2=2:
Group A of (iii) = {(i),(ii)}
Group B of (iii) = {(iv),(v),(vi)}		

Figure 7: Since Power's `sync` is A- and B-cumulative, it includes accesses from other threads into its scope. Most [8, 28, 51] but not all [6] formalizations consider (vi) to be in group B.

store→load ordering is marked as being enforced, but only locally, if the instructions access the same address. Figure 6b also shows how the MOST for SPARC RMO clearly indicates that load→load ordering of accesses to the same address does not need to be enforced.

3.3. Fence Cumulativity

As seen earlier, non-multiple-copy-atomic architectures are by default prone to counterintuitive behaviors such as the non-causal outcome of `wrc` (Figure 2). To enable fences to restore causality, architectures such as ARM and Power define fences to enforce ordering with respect to accesses in threads other than the thread issuing the fence [8, 28]. This property is known as *fence cumulativity*. Cumulativity is difficult to define precisely, as can be seen from the variety of definitions in use [5, 6, 8, 28, 42, 51]. Nevertheless, they all share the same intuition. Cumulative fences are defined to enforce ordering with respect to instructions in each of two groups: group A is the set of instructions ordered before the fence, and group B is the set of instructions ordered after the fence. The base case is that groups A and B are the sets of instructions prior to and subsequent to the fence in program order, respectively. *A-cumulativity* (AC) requires that instructions (from any thread) that have performed prior to an access in group A are also members of group A. *B-cumulativity* (BC) requires that instructions (from any thread) that perform after a load that returns the value of a store in group B are also themselves in group B.

Figure 7 demonstrates the cumulativity of the Power `sync` fence (iii). In the base case, group A consists of (ii) and group B consists of (iv). Then, if (ii) reads from (i), (i) happens before (ii), and so since the fence is A-cumulative, (i) is included

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	—	$\checkmark_N$	$\checkmark_N$
AC St	—	—	—	$\checkmark_N$	$\checkmark_N$

(a) Power `lwsync`

	PO Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓
PO St	✓	✓	$\checkmark_S$	$\checkmark_S$
AC St	✓	✓	$\checkmark_S$	$\checkmark_S$

(b) Power `sync`

Figure 8: Incorporating cumulativity into MOST definitions

into group A of the `sync` instruction. Similarly, if (v) reads from (iv), then since (vi) happens after (v), (v) and (vi) are included in group B of the fence by B-cumulativity.

MORs address cumulativity by including A-cumulative (AC) and B-cumulative (BC) operations as explicit rows and columns in a MOST. Orderings of accesses related by cumulativity are specified in MOSTs in exactly the same way as for accesses related by program order, i.e., those in the same thread as the MOR in question. Figure 8 shows the MOSTs for both `lwsync` and `sync`. The fact that the `sync` fence (iii) enforced ordering from (i) to (iv) in Figure 7, for example, is captured by the $\checkmark_N$ entry in row (AC St) and column (PO St). From the point of view of (iii), (i) is related by A-cumulativity, and (iv) is later in program order.

3.4. Summary

By incorporating the details discussed above, MOSTs serve as a complete, precise, architecture-independent, and self-contained specification of the semantics of memory ordering requirements (MORs). To demonstrate the usefulness of this approach, the next section describes how to algorithmically compare the strengths of different MOSTs. Then, Section 5 describes a more advanced case study in which MOSTs are used to dynamically translate orderings of one architecture onto primitives of a different architecture.

An extended technical report version of this paper includes a gallery of example MOSTs for different architectures [39].

4. Comparing and Manipulating MOSTs

A key benefit of the MOST notation is that it allows for flexible, algorithmic comparison of MOSTs, *even those originally coming from different models*. This type of comparison forms a key component of compilers, mappers, or translators envisioned earlier in Section 1. This section describes how to perform such comparisons.

4.1. MOST Partition Refinement

Because different architectures emphasize different consistency model features, as described in Section 3, they may use distinct choices of rows and columns to define their MOSTs. To resolve this, before any MOST-MOST comparisons can occur, the rows and the columns of the MOSTs must be *refined* into matching partitions. The MOST refinement process has two steps. The first is to find the set of categories that should be used as the row and/or the column headings for the

	PO+ SA Ld	PO+ DA Ld	PO St
PO Ld	✓	✓	✓
PO St	$\checkmark_L$	—	$\checkmark_M$

$\xrightarrow{\text{Refine}}$

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	?	✓	?
AC Ld	?	?	?	?	?
PO St	$\checkmark_L$	—	?	$\checkmark_M$	?
AC St	?	?	?	?	?

(a) Because cumulativity is not explicitly addressed by the TSO PPO specification, the MOST must be refined in order to compare it with MOSTs from the Power architecture.

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	✓	✓	✓	✓	✓
AC Ld	✓	✓	✓	✓	✓
PO St	$\checkmark_L$	—	✓	$\checkmark_M$	✓
AC St	✓	✓	✓	$\checkmark_M$	$\checkmark_M$

(b) MOST for TSO PPO when properly refined to match the format of Power architecture MOSTs.

Figure 9: Using MOST partition refinement to compare TSO PPO and Power `lwsync`

refined MOSTs. Standard partition refinement techniques can be used to merge the row and/or column choices from different MOSTs into a finer-grained partition capturing both; thus we omit a full algorithmic description here [46].

The second step is to fill in the cells of the newly-refined MOST. In most cases, this simply requires duplicating the original contents of a cell that was refined into multiple "child" cells. However, if a particular MOST feature is architecture-specific, partition refinement can lead to scenarios in which the ordering strength of a particular cell is left unspecified. These cells can be filled in conservatively (i.e., by assuming the unspecified orderings are required, or by assuming they are not enforced) or using some external reasoning.

Figure 9 shows an example. The MOST for `lwsync` (Figure 8) is laid out differently from the MOST defining TSO PPO (Figure 6a), as TSO does not explicitly define its MOSTs in terms of cumulativity. In this case, we can reason that cumulativity follows implicitly from the $\checkmark_M$ store→store ordering strength of TSO, and therefore the cumulative ordering cells are in fact enforced.

4.2. MOST Comparison Operators

Once two MOSTs have been refined (if necessary) into the same layout of rows and columns, then a comparison of the two can be defined by comparing each pair of corresponding cells. The cell-by-cell comparison is defined by checking whether one strength level implies the other. For example, enforcement of single-copy atomic store→store ordering implies that multiple-copy atomic store→store ordering is also enforced, and hence that $\checkmark_S \geq \checkmark_M$. We define the full complement of comparison operations ($<,\leq,=,\neq,\geq,>$) analogously. Note that in general, this ordering is partial, not total.

Two MOSTs may also be combined to produce a single MOST representing enforcement of both orderings. This can

	PO+ SA Ld	PO+ DA Ld	BC Ld	PO St	BC St
PO Ld	—	—	—	—	—
AC Ld	—	—	—	—	—
PO St	—	—	✓	$\checkmark_{M-N}$	$\checkmark_{M-N}$
AC St	✓	✓	✓	$\checkmark_{M-N}$	$\checkmark_{M-N}$

Figure 10: Subtracting Power `lwsync` (Figure 8) from (a properly refined) TSO PPO (Figure 9b). The shaded cell highlights the ordering that distinguishes the two cases in Figure 1b.

occur if, e.g., there are two fences back-to-back in a program. We define this operation as the *join* operator ($\vee$). A join operation is intuitively similar to a max operation, except that the result may not be equal to any one of the inputs, because comparison is not totally ordered. Instead, the join produces a new MOST which is at least as strong as (in terms of $\geq$ above) each of the input MOSTs. The calculation of a join is also defined cell-by-cell; each cell in the result MOST must be an ordering strength which implies the strength levels in the corresponding cells of both input tables. In other words, if $A \vee B = C$, then C must satisfy $C \geq A$ and $C \geq B$.

Lastly, *subtraction* ($-$) produces a MOST which specifies the orderings which are enforced by the first MOST but not by the second. Conceptually, this corresponds to a scenario in which a certain set of orderings is required, but a particular MOR may only enforce some subset of those orderings; subtraction of these two MOSTs produces the set of required orderings that remain unenforced. Again, ArMOR calculates this in a cell-by-cell manner.

4.3. MOSTs Comparison Examples

As a relatively simple example, consider a comparison of the two MOSTs of Figure 8. By comparing each pair of corresponding cells in the table, it is clear that `lwsync` < `sync`: every cell in the `sync` MOST is at least as strong as the corresponding cell in the `lwsync` MOST, and some comparisons are strict. In this case, the join ($\vee$) of the two tables is equivalent to the `sync` MOST. On the other hand, consider the subtraction of TSO PPO (once properly refined) from Power `lwsync`. This result is shown in Figure 10. Not only does the subtraction operation show that `lwsync` clearly enforces fewer orderings than TSO requires, but it also shows exactly *which* orderings are unenforced.

A major benefit of the ArMOR approach is that the manipulations performed above are entirely algorithmic. In the next section, we will use these techniques to automatically derive the designs of consistency model translation modules called shims, given only the set of MOSTs used by the input and output memory consistency models.

5. ArMOR Case Study: Dynamic Inter-MCM Translation

Recent work has demonstrated the performance and/or power benefits of performing dynamic binary translation across ISAs and/or microarchitectures [20, 65]. However, this previous work focused on opcode-for-opcode translation and memory layout issues; it did not address memory consistency models. Inter-consistency model translation has only been studied for specific cases such as SC$\rightarrow$TSO 13b [21, 64]. In this section, we show how ArMOR fills this gap by deriving self-contained translation modules called *shims* which easily, automatically, and correctly translate between any pair of memory consistency models. Although translation results in some overhead, we envision this cost being outweighed by the benefits of migrating to faster or more power-efficient hardware.

5.1. Motivating Example

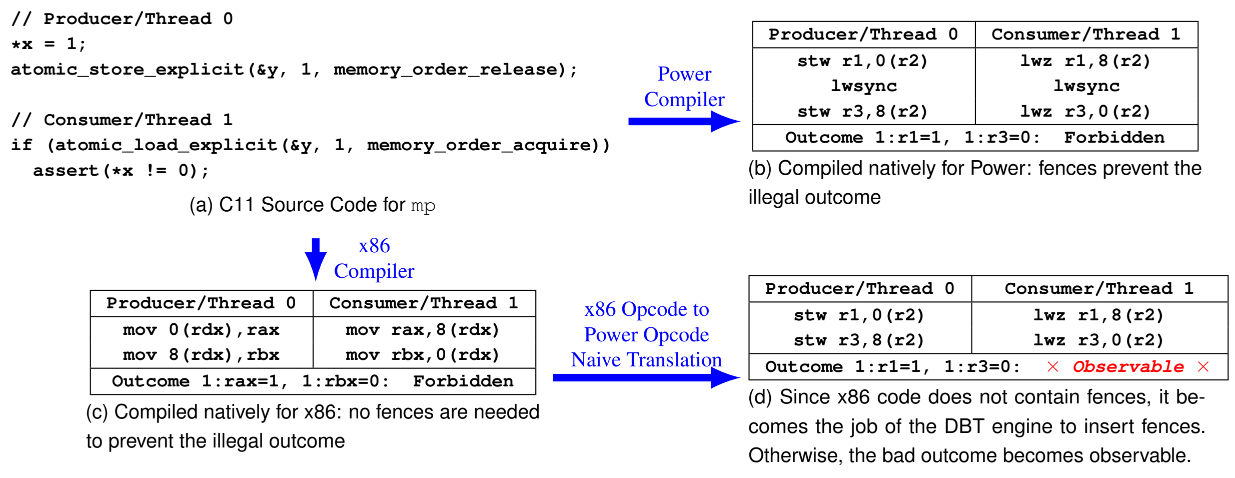
Figure 11a shows the source code for the `mp` (message passing) *litmus test*. For this test, the C11 memory ordering rules specify that if the consumer reads 1 from *y*, then it must also return 1 from *x*. In a traditional scenario, the compiler ensures that all of the C11 ordering rules within each thread are respected by the generated assembly code. This generally occurs by looking up the architecture-specific implementations of the software synchronization constructs in a pre-calculated table such as Table 1. On Power, the orderings are enforced by inserting `lwsync` fences, as shown in Figure 11b. On x86-TSO, as Figure 11c shows, no fences are needed.

Problems arise if one tries to perform naive binary translation of the x86 code to execute on the Power architecture. Opcode-for-opcode translation would produce the code in Figure 11d. Unfortunately, because the source x86 code lacks fences, the translated code also lacks fences, meaning that the extra enforcement required to prevent the bad outcome of the `mp` litmus test is missing. This demonstrates that *if cross-ISA binary translation techniques do not account for the consistency model, the resulting code could produce illegal outcomes*. The goal of this section is therefore to generate translator shims which automatically and dynamically determine where to insert MORs and which MORs to insert, *without requiring offline analysis of the code*.

5.2. Basic Operation

ArMOR translation takes place conceptually on a *stream*: an ordered sequence of memory operations (loads, stores, or fences) passing through some particular point in a processor or IP core. The specific format of the stream operations depends on the location where the translation is conducted. Streams may carry macroops, microops, or whatever other form operations may take at the chosen location. A stream may also carry implicit (via preserved program order) or explicit (via fences) ordering requirements on its memory operations. We refer to incoming (newer) operations as *upstream operations* and outgoing (older) operations as *downstream operations*.

A *shim* maps each incoming upstream operation into one or more downstream operations which are strong enough to enforce the memory ordering requirements of the upstream operation. To translate an explicit upstream MOR such as a



```
// Producer/Thread 0
*x = 1;
atomic_store_explicit(&y, 1, memory_order_release);

// Consumer/Thread 1
if (atomic_load_explicit(&y, 1, memory_order_acquire))
  assert(*x != 0);
```

(a) C11 Source Code for `mp`

Producer/Thread 0	Consumer/Thread 1
`stw r1,0(r2)`	`lwz r1,8(r2)`
`lwsync`	`lwsync`
`stw r3,8(r2)`	`lwz r3,0(r2)`
`Outcome 1:r1=1, 1:r3=0: Forbidden`	

(b) Compiled natively for Power: fences prevent the illegal outcome

Producer/Thread 0	Consumer/Thread 1
`mov 0(rdx),rax`	`mov rax,8(rdx)`
`mov 8(rdx),rbx`	`mov rbx,0(rdx)`
`Outcome 1:rax=1, 1:rbx=0: Forbidden`	

(c) Compiled natively for x86: no fences are needed to prevent the illegal outcome

Producer/Thread 0	Consumer/Thread 1
`stw r1,0(r2)`	`lwz r1,8(r2)`
`stw r3,8(r2)`	`lwz r3,0(r2)`
`Outcome 1:r1=1, 1:r3=0:` × ***Observable*** ×	

(d) Since x86 code does not contain fences, it becomes the job of the DBT engine to insert fences. Otherwise, the bad outcome becomes observable.

Figure 11: A compiler targeting either architecture directly would produce correct code. However, binary translation that does not account for differences in consistency models would lead to the invalid outcome becoming observable.

fence, the shim must emit zero or more downstream operations which combine to implement all of the ordering requirements specified by that fence. To handle implicit upstream ordering requirements, the shim must enforce any upstream PPO requirements that are not enforced by downstream PPO.

An overly-conservative (and hence low-performing) but correct baseline would be to insert the strongest possible fence between each pair of instructions. In most cases, this is sufficient to restore sequential consistency[3], let alone the requirements of the source architecture. However, this approach is overkill, as many inserted fences would be redundant and unnecessary. Instead, shims insert MORs lazily—just before they are actually needed.

Conceptually, shims are finite state machines in which downstream MOR insertion takes place while traversing certain state transitions. Specifically, shims only enforce particular orderings if the relevant upstream operations *have actually been observed* since a relevant earlier fence. We refer to such orderings as *pending*. Each FSM state represents a particular set of pending ordering requirements, and it does so in the form of a *pending ordering table*. Pending ordering tables are in a sense the inverses of MOSTs; rather than specifying which orderings are required, they specify the orderings that have not (yet) been enforced. As described in Figure 12, we depict pending orderings of a given strength with the lowercase equivalent of the uppercase notation from Figure 3.

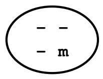
Given a state and an incoming upstream operation, the shim FSM generation algorithm calculates the MOR to emit (if any) based on the current state's pending orderings and then moves to a new state reflecting a new set of pending orderings. Pending orderings within columns matching incoming accesses need to be enforced by inserting a sufficiently strong fence or other MOR. Other pending orderings can be delayed lazily.

Lazy insertion is not the only possible design approach.

[3]This is not universally true; for example, Itanium unordered accesses cannot be made sequentially consistent [29].

	Loads	Stores
Loads	—	—
Stores	—	m

(a) Example pending orderings table.



(b) Equivalent FSM state representation

Level	Description
s	An ordering of strength $\checkmark_S$ is pending
m	An ordering of strength $\checkmark_M$ is pending
n	An ordering of strength $\checkmark_N$ is pending
✓	An ordering of strength ✓ is pending
—	No ordering is pending

(c) Pending orderings legend (derived from Fig. 3)

Figure 12: Pending ordering tables, which are used to describe states within shim FSMs

More eager insertion could make it easier to hide the latency of inserted fences, but it may also result in inserting a larger number of fences. Our experience is that the benefits of laziness outweigh the small potential latency hiding of eagerness.

5.3. Shim FSM Generation

The state transition function is given in Algorithm 1. Once MORs have been inserted downstream, the orderings they enforce can be subtracted (as defined in Section 4.2) from the current set of pending orderings. The upstream operation is then itself propagated downstream, and any orderings in the corresponding row which are enforced by PPO upstream but not downstream must be marked as pending.

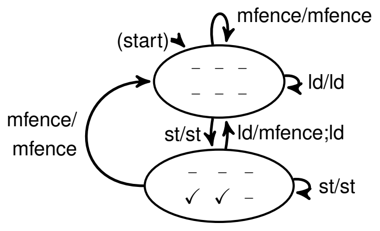
One subtlety in Algorithm 1 is the use of assumed pending ordering requirements, or "assumedReqs". This property handles the case in which accesses from certain rows and/or columns are not directly observable and must therefore be conservatively assumed to be pending ordering enforcement. The "assumedReqs" approach allows shims to translate correctly without needing to observe the execution of other cores. In particular, this addresses the challenge that cumulative fences (Section 3.3) enforce ordering restrictions with respect to accesses from other cores, even though cores may not always be

Algorithm 1 Shim FSM Transition Function

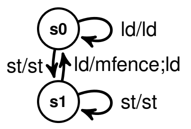
```
Function: NextState(currentState, op):
  if isFence(op) then
    orderingsToEnforce = op ∨ assumedReqs
    newOrderings = ∅ // Fences are not themselves in groups A or B
  else
    // Enforce pending orderings in relevant column(s)
    orderingsToEnforce = KeepColumn(currentState, op) ∨ assumedReqs
    // Mark relevant orderings in relevant row(s) pending
    newOrderings = KeepRow(upstreamPPO − downstreamPPO, op)
  end if
  insertedFence = WeakestSufficientFence(orderingsToEnforce)
  // propagated = pending and not enforced by inserted fence
  propagatedOrderings = state − insertedFence
  nextState = propagatedOrderings ∨ newOrderings
  return (insertedFence, nextState)
```

$$\text{KeepColumn}(s, col)[i][j] = \begin{cases} - & j \neq col \\ s[i][j] & j = col \end{cases}$$

$$\text{KeepRow}(s, row)[i][j] = \begin{cases} - & i \neq row \\ s[i][j] & i = row \end{cases}$$



(a) Automatically-generated shim FSM



(b) Previous work [21, 64].

Key:

x/y	On an incoming upstream operation x, send y downstream
x/y;z	On an incoming upstream operation x, send y followed by z downstream

Figure 13: Shim FSM for SC upstream and TSO downstream. ArMOR shims also allow upstream MORs to act as inputs.

able to directly observe such accesses.

Given MOSTs for a source and target architecture, ArMOR can generate shim state machines *offline and in advance*. Furthermore, the state space of the shim FSMs is small enough that the space can be explored completely in just seconds. As Section 7.1 shows, the FSMs generally end up with few states and hence can be implemented very cheaply.

We include one special-case optimization: we allow a store→store ordering of strength $\checkmark_S$ to be enforced by a MOR with enforcing store→store ordering of strength $\checkmark_M$ *if* store→store ordering of strength ✓is marked as pending. Intuitively, this captures the notion that the only difference between single- and multiple-copy atomicity is that forwarding from a local store buffer is permitted only in the latter. This optimization, which is used implicitly in previous work [21, 64], makes shims slightly lazier and hence improves performance.

As an example, Figure 13a shows the automatically-generated FSM for the shim translating between sequential consistency upstream and TSO downstream. This shim has two states, and it has the effect that a fence gets inserted between every store and subsequent load. As Figure 13b shows, this particular case has been studied before, and so it serves as a useful sanity check for our algorithm.

5.4. Design Considerations

Microarchitectural Placement. ArMOR requires that a stream be sorted into a legal visibility order so that preserved program order needs can be detected. This means that when shims are implemented in hardware, their placement within the pipeline matters. Placing a shim too early in the pipeline may render it unable to track same/different address dependencies (Section 3.2). Placing it at a later location through which non-memory instructions do not pass may prevent it from being able to observe information such as inter-instruction dependencies. In such cases, these unobservable dependencies would have to conservatively be included into "assumedReqs" (Algorithm 1) along with cumulativity.

Our hardware evaluation (Section 6.2) places a shim into the issue stage of a pipeline, as all necessary information is observable at that point. Our software dynamic binary translation-based evaluation operates the stream of instructions in their original program order.

Atomic Instructions. Atomic instructions such as compare-and-swap can be easily added into our model. If atomics are considered a separate class of instruction from loads and stores, they would simply form new MOST rows and columns. Alternatively, if atomics are treated as a bundled load-store pair, Algorithm 1 could be modified to look up multiple rows and columns for such instructions, rather than just one. Either solution is viable as long as the ordering implications of such operations are correctly specified.

Downstream Non-Fence MORs. Downstream MORs need not be fences. It is possible, for example, to use lightweight MORs such as ARM/Power address dependencies as well. Doing so would require more than just fence insertion; it would also require rewriting instruction operands, but dynamic binary translators already do this regularly [36].

Stream Interruptions. Streams may be interrupted by events such as context switches and hence lose their state. A conservative solution is to simply enforce a full fence instruction and then return to the start state. A more aggressive solution would be to jump to a state which marks all orderings not enforced by the downstream PPO as pending. In either solution, ArMOR's fundamental operation remains the same.

Dynamically Changing Partition Subsets. Algorithm 1 as shown assumes that the categorization of a given instruction into a particular row and column is based only on static properties of that instruction. However, this may not always be true. AMD GPUs contain a fence `s_waitcnt <n>` which only enforces ordering with respect to accesses other than the most recent *n*. If the classification of a memory access can change, the state transition function must be modified to account for such changes. In practice, few MORs are defined this way,

Property	Real System	Simulator
System	8-core	4-core
CPU	Xeon X7560	gem5 O3
Frequency	2.27 GHz	2.0 GHz
Pipeline	OoO	OoO
L1I Cache	32kB, private	32kB, private
L1D Cache	16kB, private	64kB, private
L2 Cache	256kB, private	2MB, shared
Cache coherence	MESI	MOESI_hammer
Memory timing model	N/A	Ruby

Table 2: System configurations

and so a cheaper option is to overapproximate the state and avoid the need for such dynamic reclassification altogether.

Speculation. ArMOR does not inhibit the use of speculative ordering enforcement techniques [21, 23], as long as these techniques maintain the architecturally-required behaviors.

6. Evaluation Methodology

In this section, we describe our evaluation of the ArMOR shims. We first provide a characterization of the breadth of ArMOR by generating shims for a number of upstream and downstream models. We then evaluate the performance of a subset of these models. We break our performance evaluation into two parts. We first implement ArMOR shims as software Pintools [36]. With near-native speeds, this approach allows for rapid exploration of various design possibilities. We then evaluate ArMOR in hardware by inserting shims into the gem5 O3 simulator pipeline.

6.1. Pintool-based Exploration

Software-based dynamic binary translation can be used by MOR designers to explore the performance impact of different hardware ordering requirements, fence implementations, or translation approaches prior to their being hardened into a processor. We use this approach to quantify the performance impact of statefulness in shims, and we explore some additional performance-oriented optimizations. We use Intel Pin [36] to implement our software shims. Because Pin executes on the x86 architecture and therefore has TSO as the downstream model, we use SC as the upstream model.

We evaluate three shim configurations. The first is the naive *stateless* case which always inserts a LOCKed instruction or `mfence` between each pair of memory instructions. The second is the *stateful* shim shown in Figure 13a. Third, the *ISA-assisted* scenario approximates the benefits of augmenting an ISA to track software- or compiler-provided information about accesses that do not need to enforce consistency. An increasing body of work has proven the benefits of providing hardware support for finer-grained specification of memory consistency behavior [19, 57]. Because we are constrained by Pin's need to execute on real hardware (which has no such ISA support), we instead present approximations which closely model the performance benefits of enabling such modifications.

The ISA-assisted scenario considers two ways in which the ISA can be augmented. First, certain accesses might be marked thread-private and hence not subject to reordering rules. Even relatively straightforward compiler analysis is able to classify as many as 81% of memory accesses [57] as private. We approximate this by inferring thread-privacy for all accesses to the stack. While this is not safe in general, our analysis reveals that it is safe for our benchmark suite[4]. This approximation classifies 75% of accesses as thread-private, very close to the percentage found by the previous work.

Second, we model the benefits of a compiler annotating memory accesses as being data-race-free, and thus not subject to any reordering constraints [2, 15]. For our pre-C/C++11 benchmark suite, all synchronization accesses occurred through libraries such as `libpthread` or inline assembly, with the remainder of the program accesses remaining data-race-free. Because library behavior may not be precisely known at compilation time, we chose to conservatively assume that all library code was annotated as potentially subject to races (and hence in need of shimming).

We run Pintool experiments on the real system from Table 2. We use benchmarks from PARSEC [13] with the native input set and four threads. We take three measurements for each scenario: the non-Pintool native runtime of the benchmark ("native"), the runtime of the benchmark with analysis enabled but fence insertion itself disabled ("instrumentation"), and the runtime with fence insertion enabled ("shim"). This allows us to roughly separate the overhead of the shim from the overheads of Pin itself. We use LOCK-prefixed `add` instructions as the primary downstream MOR; these are equivalent to `mfence` in strength but 28% faster in our experiments.

6.2. Hardware Simulation Methodology

While Pin offers opportunities for early exploration, hardware support can further accelerate translation. We use the gem5 simulator to implement a hardware shim within the issue queue of the gem5 O3 pipeline [14]. In the issue queue, the shim has enough information to properly track both instruction dependencies and preserved program order. As Figure 14 summarizes, the gem5 O3 pipeline is multiple-copy-atomic and always enforces load$\rightarrow$store ordering, while load$\rightarrow$load and/or store$\rightarrow$store ordering enforcement (of strength $\checkmark_S$) are optional. We adjust these options to implement a variety of downstream preserved program order settings.

At the gem5 O3 issue queue, regardless of the architecturally-defined fences, three downstream fences are available microarchitecturally: a load fence, a store fence, and a full fence. The MOSTs for these fences are also summarized in Figure 14. The fences are implemented microarchitecturally by treating an associated memory access as non-speculative. This requires that before the access executes, it must be at the head of the reorder buffer and the store buffer must be empty. The associated operation is also treated as a load and/or store

[4]There are cases in which worker threads access objects allocated by the main thread, but these are synchronized via `pthreads`.

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
Ld	✓	✓	✓	✓
St	✓	✓	$\checkmark_S$	$\checkmark_S$

(a) Full fence

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
Ld	✓	—	✓	✓
St	$\checkmark_L$	—	$\checkmark_M$	$\checkmark_M$

(b) PLO PPO

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
Ld	✓	—	✓	✓
St	$\checkmark_L$	—	$\checkmark_S$	$\checkmark_S$

(c) MSFence

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
Ld	✓	✓	✓	✓
St	$\checkmark_L$	—	$\checkmark_M$	—

(d) PSO PPO

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
Ld	✓	✓	✓	✓
St	✓	✓	$\checkmark_M$	—

(e) MLFence

	PO+ SA Ld	PO+ DA Ld	PO+ SA St	PO+ DA St
Ld	✓	—	✓	✓
St	$\checkmark_L$	—	$\checkmark_M$	—

(f) LSO PPO

Figure 14: Available downstream PPO and MORs of the gem5 O3 simulated CPU.

	Downstream								
Upstr.	**TSO**	**PLO**	**PSO**	**LSO**	**RMO**	**RMO+**	**PwrA**	**Pwr**	**ARM**
SC	2	1	2	1	1	2	2	1	1
TSO	-	2	2	4	3	5	4	1	1
PLO	-	-	2	2	2	4	3	1	1
PSO	-	2	-	2	3	3	2	1	1
LSO	-	-	-	-	2	2	2	1	1
RMO	-	-	-	-	-	-	1	1	1
PwrA	-	-	-	-	-	-	-	1	1
Pwr	-	-	-	-	-	-	-	-	1
ARM	-	-	-	-	-	-	-	1	-

Key for models not yet described:

RMO+	RMO variant with sixteen fences representing four independent choices of load→load, load→store, store→load, and store→store ordering
PwrA	A multiple-copy atomic variant of Power

Table 3: Number of states in shim for each pairing of upstream and downstream models. The performance of the shaded FSMs is evaluated in Section 7.3.

barrier to prevent subsequent memory microops of the relevant type(s) from executing until it has itself completed.

Table 2 gives specifications for our simulated system. Because the generated FSMs are small, we assume they can be updated in parallel with other pipeline operations with no incurred latency. We use PARSEC [13] benchmarks with the simsmall input set and 4 threads. We execute these benchmarks on four downstreams: TSO, PSO, PLO (partial load order, named by analogy to SPARC PSO), and LSO (load→store order enforced). We compare the performance for each case, including both stateless and stateful shims.

7. Performance Results

7.1. Shim FSM Generation

To demonstrate the full breadth of applicability of ArMOR, we automatically generate shim FSMs for various combinations of the upstream and downstream consistency models. A summary is shown in Table 3; the shaded subset highlights the scenarios evaluated in the next two sections. We also optimize away transient and redundant states, as in Figure 15, to reduce the implementation cost. These results show that shim FSMs are generally very small, and hence that they can be implemented in practice with little area cost.

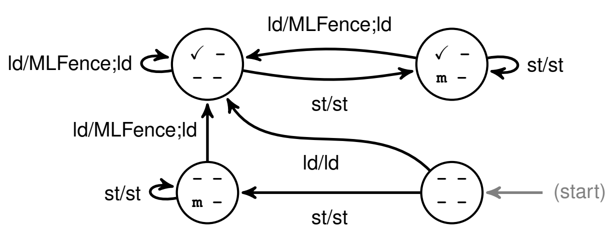


Figure 15: The shim FSM generated for SC on PLO (for visual clarity, not all pending ordering table cells are drawn). The bottom two states are transient (because there are no downward arrows to return to them from the top two states). The top two states are redundant because their behavior is identical. Therefore, this FSM reduces to a single state.

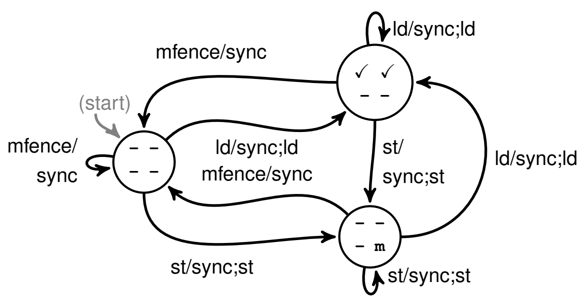


Figure 16: Automatically-generated shim FSM for TSO upstream and Power downstream (for visual clarity, not all pending ordering table cells are drawn).

Adding state does not help in every situation. Notably, Figure 16 shows that the FSM generated for TSO upstream and ARM/Power downstream minimizes to a trivial machine which always inserts `sync`. While it may seem to be overkill to insert a `sync` before each memory access, Figure 1b highlighted that anything weaker would permit behaviors such as `iriw` to become illegally observable. In fact, every multiple-copy-atomic upstream paired with a downstream allowing non-multiple-copy-atomic stores produces a FSM which is just as inefficient. This is not a shortcoming of ArMOR, but rather a fundamental difference in the behavior of stores on each architecture. We return to this observation in Section 7.4.

Lastly, we note that we attempted to translate to GPUs as well, but we were limited both by the incompleteness of current specifications as well as, more fundamentally, a lack of both multiple-copy atomicity and cumulative fences [4]. With neither feature, GPUs simply have no means by which to enforce implicit (e.g., on TSO) or explicit (e.g., Power `sync` or ARM `dmb`) cumulativity requirements, and hence they are unable to serve as downstream targets for translation.

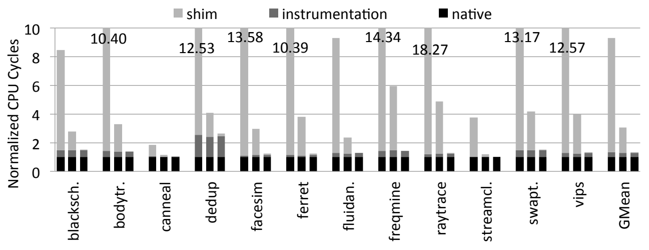


Figure 17: Performance overhead of ArMOR using dynamic binary translation and different levels of performance optimization. From left to right, the three bars represent the stateless, stateful, and ISA-assisted stateful cases, respectively.

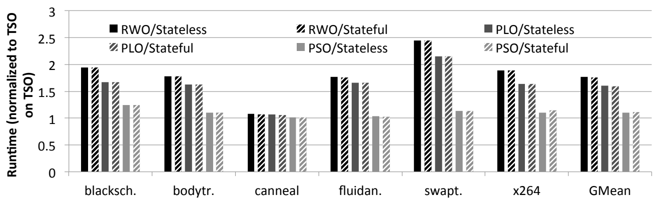


Figure 18: Simulated performance with x86-TSO software and varying hardware models.

7.2. DBT-Based Exploration

Figure 17 shows the performance of the three Pintool shim configurations of Section 6.1. We normalize to the runtime of each benchmark when it is compiled for x86-TSO and executed natively on x86-TSO hardware; this conservatively attributes the inherent overhead of SC vs. TSO to ArMOR as well. The stateless shim has a geomean overall performance cost of 9.33×. The stateful configuration improves this to 3.05×. Finally, making use of the ISA augmentations discussed in Section 6.1 reduces the total overhead to just 1.33×.

The instrumentation overhead was approximately the same for each Pintool—1.31× on average. This shows that the ArMOR shims themselves do not introduce significant overhead beyond the overhead of instrumentation itself—175% in the case of our conservative stateful configuration, but only 3% in the more aggressive ISA-assisted case. These numbers demonstrate that ArMOR translation can take place with low or, under the right conditions, even negligible overhead in practice beyond what is already needed to perform dynamic binary translation. They also demonstrate the value of using software-based DBT as a tool for exploring the design space of and profiling the use of synchronization in practice.

7.3. Performance of Hardware Implementation

Figures 18 and 19 show the overheads of running x86-TSO and x86-SC software, respectively, on cores with weaker hardware memory models. Stateful shims are shown with striped bars; however, the stateful FSMs for SC-on-PLO and SC-on-LSO are equivalent to the stateless cases, and so we draw only one bar for these. We normalize to x86-TSO hardware, just as we do for the Pintool results.

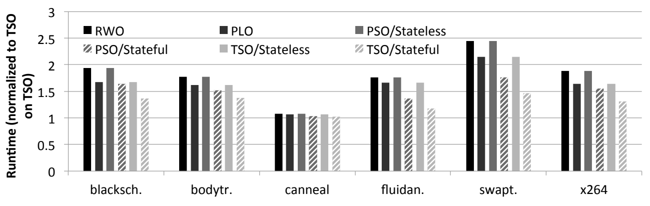


Figure 19: Simulated performance with x86-SC software and varying hardware models, normalized to x86-TSO software on x86-TSO hardware as described in the text.

For x86-TSO upstream, in some cases, the benefits of statefulness are negligible. These FSMs stay in a single state except when they see an upstream LOCKed instruction or fence. Since both are rare under TSO, the FSM mostly remains in the single state in practice. For this reason, implementations may choose to treat the lightly-used state as transient (as in Figure 15) and optimize it away to make the FSM even cheaper.

In all cases, the overheads remain well within an acceptable range for dynamic binary translation [10, 36]: even the worst case overhead of SC on RMO requires a geomean slowdown of only 77%. In the best case, TSO on PSO, the overheads are as small as 10%.

7.4. Takeaways

Our explorations both via Pintools and via simulation of hardware-supported shims have led to several major takeaways. First, *architectures should provide a way to optionally make stores multiple-copy atomic*. A multiple-copy atomic variant of Power, labeled “PwrA” in Table 3, does allow for more efficient FSMs. If the user is sure that no `iriw`-like behavior will occur, then multiple-copy atomicity can be disabled to improve performance; otherwise, it can be enabled to ensure safety. Notably, ARMv8 has taken this approach with new load-acquire and store-release opcodes [8]. ArMOR provides a rigorous methodology for performing this analysis.

Our second observation is that *the more downstream MORs (i.e., fence variations) are available, the more intelligent the translation can be*. The difference between “RMO” and “RMO+” in Table 3 is that the former implements only the three fences shown in Figure 14, while the latter implements sixteen possibilities, with one choice each for load→load, load→store, store→load, and store→store. Having finer-grained downstream fences allows for smarter fence choices and higher-performance implementations.

Third, *ISAs and intermediate representations should maintain consistency metadata even if it is redundant with respect to preserved program order*. In particular, ISAs with strong models carry little information about consistency, as it is mostly redundant. However, this makes translation much more difficult, as the overly-constrained preserved program orderings of a strong model like TSO are themselves costly and mostly unnecessary. Keeping consistency information in the ISA would provide numerous benefits (shown in Section 7.2 and previous work) at the cost of modest code size increase. In such a scenario, ArMOR can be used to remove upstream fences that become redundant under a stronger downstream model.

Finally, we note that *non-multiple-copy-atomic architectures cannot ignore cumulativity*. If they do, then there simply is no way to implement communication across more than two cores safely. While current hardware (e.g., GPUs) simply limits the amount of inter-thread communication that can take place, the increasingly heterogeneous hardware of the future will demand the ability to perform such many-threaded concurrent tasks. Fortunately, ArMOR provides a way to evaluate those needs early in the design process.

8. Related work

Memory Consistency Models. Adve and Gharachorloo provide a comprehensive survey [1] of early attempts to define various consistency models. Since then, there has been a progression [9, 55] in formalizing the definitions of memory models. Modern formalizations generally fall into two provably equivalent categories. Operational models [45, 51] define valid executions as those which are observable on a formally-defined abstract machine model. Axiomatic models [3, 40] define valid executions as those which satisfy the chosen set of memory ordering axioms. PipeCheck [38] extended memory model analysis to the microarchitecture space. Most of these models "hard-code" fence behavior into the model in some way, by defining fences in terms of barrier propagation and acknowledgment [51], store buffers [45], or other architecture-specific features. To the best of our knowledge, no existing model specifies fence types and ordering specifications in a way that is sufficiently general and architecture-independent that inter-architecture conversion can be safely performed dynamically in the way ArMOR does.

Software also plays a major role in consistency [15, 19, 41, 60]. There have been numerous academic proposals for software-level consistency models for heterogeneous systems [22, 31, 37, 50]. In this paper, we focus mostly on binary translation, although we do make use of software model concepts such as data race freedom [2].

Recent work has also explored the application of consistency models to non-volatile storage [47]. We see ArMOR as applicable to memory persistency model analysis as well.

Fence Insertion and/or Elimination. The work of Alglave et al. [6] has a goal similar to ours in that it studies how to restore the behavior of one architecture by inserting fences on a weaker architecture. Their definition of cumulativity is subtly different than the definition given in the Power architectural specification [28], and their proof-based method does not readily adapt to a modified definition. More critically, their solution is declarative: it specifies only a static correctness condition rather than a constructive dynamic translation method. Furthermore, their correctness condition depends partially on inserting fences between loads and their source stores. ArMOR makes no such assumption about identifying a load's source store, as such information is often simply unavailable.

Since the work of Shasha and Snir [54], researchers have considered topics such as verifying the insertion of fences to implement a stronger consistency model [17, 33] and/or the elimination of redundant fences [64]. Others focus on automatically determining where to insert fences [3, 27], and also on incorporating such methods into a compiler [35, 61].

Cross-ISA Translation. DeVuyst et al. [20] study heterogeneous-ISA code migration. They focus on laying out data in an architecture-independent manner, and they use compiler support and bursts of dynamic binary translation to smooth the migration process. They assume, however, that the source and target ISAs have identical consistency models; they do not address translation of memory ordering requirements.

Various case studies have studied translation in more specific contexts, including Baraz et al. [11] for x86 code on Itanium processors, Higham and Jackson [26] for SPARC to and from Itanium, and Gschwind et al. [25] from the "firm" model (similar to TSO) onto Power. Industry white papers [16] have also discussed this topic. None of these techniques, however, easily generalize to other architectures as ArMOR does.

9. Conclusion

ArMOR demonstrates the practicality of automating analysis and dynamic translation of memory consistency models. We foresee ArMOR's MOST notation being useful across a broad range of compilation and translation tasks including static compilation, JIT compilation, dynamic binary translation, and more. ArMOR highlights the pros and cons of different choices of fences and MORs, and we use ArMOR to provide insights that can assist in exploring memory system design tradeoffs in future heterogeneous systems.

Acknowledgments

We thank Santiago Cuellar, Yaosheng Fu, Yatin Manerkar, Tri Nguyen, Daniel Sorin, and the anonymous reviewers for their helpful feedback. The authors acknowledge the support of C-FAR (under the grant HR0011-13-3-0002), one of six SRC STARnet centers by MARCO and DARPA. In addition, this work was supported in part by the National Science Foundation (under the grant CCF-0916971). Michael Pellauer was employed at Intel VSSAD during much of the research for this paper, and Daniel Lustig was partially supported by an Intel Graduate Fellowship.

References

[1] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] S. Adve and M. Hill, "Weak ordering: a new definition," *ISCA*, 1990.

[3] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design (FMSD)*, vol. 41, no. 2, pp. 178–210, 2012.

[4] J. Alglave, M. Batty, A. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency: weak behaviours and programming assumptions," *ASPLOS*, 2015.

[5] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of Power and ARM machine code," *4th*

Workshop on Declarative Aspects of Multicore Programming (DAMP), 2009.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," *CAV*, 2010.

[7] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data-mining for weak memory," *ACM TOPLAS*, vol. 36, July 2014.

[8] ARM, "ARM architecture reference manual," 2013.

[9] Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," *ISCA*, 2006.

[10] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, "Analyzing parallel programs with Pin," *IEEE Computer*, vol. 43, no. 3, pp. 34–41, 2010.

[11] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems," *MICRO*, 2003.

[12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling C/C++ Concurrency: from C++11 to POWER," *POPL*, 2012.

[13] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comp. Arch. News*, vol. 39, no. 2, Aug. 2011.

[15] H.-J. Boehm and S. Adve, "Foundations of the C++ concurrency memory model," *PLDI*, 2008.

[16] Broadcom, "Migrating CPU specific code from the PowerPC to the Broadcom SB-1 processor," *White Paper SB-1-WP100-R*, 2002.

[17] S. Burckhardt, R. Alur, and M. M. K. Martin, "CheckFence: Checking consistency of concurrent data types on relaxed memory models," *PLDI*, 2007.

[18] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation—–a performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[19] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," *PACT*, 2011.

[20] M. DeVuyst, A. Venkat, and D. Tullsen, "Execution migration in a heterogeneous-ISA chip multiprocessor," *ASPLOS*, 2012.

[21] Y. Duan, A. Muzahid, and J. Torrellas, "WeeFence: Toward making fences free in TSO," *ISCA*, 2013.

[22] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-M. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," *ASPLOS*, 2010.

[23] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," *29th International Conference on Parallel Processing (ICPP)*, 1991.

[24] P. Greenhalgh, "big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011. [Online]. Available: http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf

[25] M. Gschwind, K. Ebcioğlu, E. Altman, and S. Sathaye, "Binary translation and architecture convergence issues for IBM System/390," *ICS*, 2000.

[26] L. Higham and L. Jackson, "Translating between Itanium and Sparc memory consistency models," *SPAA*, 2006.

[27] T. Q. Huynh and A. Roychoudhury, "Memory model sensitive bytecode verification," *Formal Methods in System Design (FMSD)*, vol. 31, 2007.

[28] IBM, "Power ISA version 2.07," 2013.

[29] Intel, "Intel Itanium architecture software developer's manual, revision 2.3," 2010.

[30] ——, "Intel 64 and IA-32 architectures software developer's manual," 2013.

[31] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: A hybrid memory model for accelerators," *ISCA*, 2010.

[32] Khronos Group, "OpenCL 2.0." [Online]. Available: http://www.khronos.org/opencl

[33] M. Kuperstein, M. Vechev, and E. Yahav, "Automatic inference of memory fences," *FMCAD*, 2012.

[34] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and efficient work-stealing for weak memory models," *PPoPP*, 2013.

[35] J. Lee and D. A. Padua, "Hiding relaxed memory consistency with compilers," *PACT*, 2000.

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *PLDI*, 2005.

[37] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," *HPCA*, 2013.

[38] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," *MICRO*, 2014.

[39] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "ArMOR: Defending against consistency model mismatches in heterogeneous architectures," *Princeton Computer Science Tech. Report TR-981-15*, 2015, (conference paper extension).

[40] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, "An axiomatic memory model for POWER multiprocessors," 2012.

[41] J. Manson, W. Pugh, and S. Adve, "The Java memory model," *POPL*, 2005.

[42] F. Z. Nardelli, P. Sewell, J. Sevcik, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave, "Relaxed memory models must be rigorous," 2009.

[43] NVIDIA, "NVIDIA Tegra K1: A new era in mobile computing," 2014. [Online]. Available: http://www.nvidia.com/content/pdf/tegra_white_papers/tegra_k1_whitepaper_v1.0.pdf

[44] ——, "CUDA C programming guide v5.5," 2013.

[45] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-TSO," *22nd Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[46] R. Paige and R. E. Tarjan, "Three partition refinement algorithms," *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973–989, 1987.

[47] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *ISCA*, 2014.

[48] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *ISCA*, 2014.

[49] Qualcomm, "Snapdragon S4 processors: System on chip solutions for a new mobile age," October 2011. [Online]. Available: https://developer.qualcomm.com/download/qusnapdragons4whitepaperfnlrev6.pdf

[50] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," *PLDI*, 2009.

[51] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER microprocessors," *PLDI*, 2011.

[52] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A verified compiler for relaxed-memory concurrency," *Journal of the ACM (JACM)*, vol. 60, no. 3, p. 22, 2013.

[53] P. Sewell *et al.*, "C/C++11 mappings to processors," http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[54] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *TOPLAS*, 1988.

[55] X. Shen, Arvind, and L. Rudolph, "Commit-Reconcile and Fences: A new memory model for architects and compiler writers," *ISCA*, 1999.

[56] A. L. Shimpi, "AMD announced K12 core: Custom 64-bit ARM design in 2016." [Online]. Available: http://www.anandtech.com/show/7990/amd-announces-k12-core-custom-64bit-arm-design-in-2016

[57] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," *ISCA*, 2012.

[58] D. Sorin, M. Hill, and D. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture, M. Hill, Ed. Morgan & Claypool Publishers, 2011.

[59] SPARC, "SPARC architecture manual, version 9," 1994.

[60] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: efficient hardware support for disciplined non-determinism," *ASPLOS*, 2013.

[61] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua, "Compiler techniques for high performance sequentially consistent Java programs," *PPoPP*, 2005.

[62] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[63] "Top500," http://www.top500.org, accessed: Jul. 28, 2014.

[64] V. Vafeiadis and F. Z. Nardelli, "Verifying fence elimination optimisations," *SAS*, 2011.

[65] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," *ISCA*, 2014.

Appendix

The following appendix presents a gallery of MOSTs and shim FSMs used in the evaluation of Sections 6 and 7. Because addresses may not all be resolved by the time instructions reach the issue queue, we ignore same-address orderings for this particular implementation.
The code which automatically generated these shims (and this appendix) is available online:
`https://github.com/daniellustig/armor`.

A.1 Summary

	TSO	PLO	PSO	LSO	RMO	RMO16	POWERA	POWER	ARM
SC	2	1	2	1	1	2	2	1	1
TSO	-	2	2	4	3	5	4	1	1
PLO	-	-	2	2	2	4	3	1	1
PSO	-	2	-	2	3	3	2	1	1
LSO	-	-	-	-	2	2	2	1	1
RMO	-	-	-	-	-	1	1	1	1
POWERA	-	-	-	-	1	1	-	1	1
POWER	-	-	-	-	-	-	-	-	1
ARM	-	-	-	-	-	-	-	1	-

Figure A.1.a: FSM node count summary.

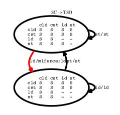
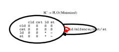
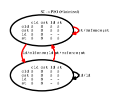
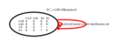
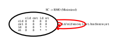
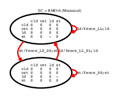
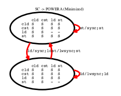
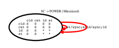
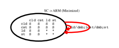
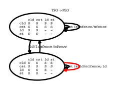
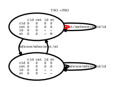
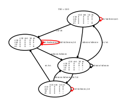
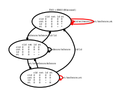
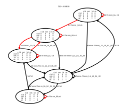
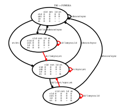
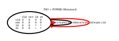
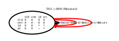
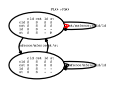
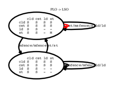
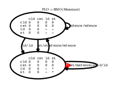
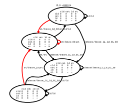
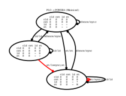
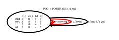
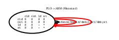
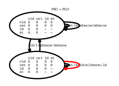
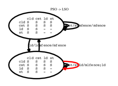
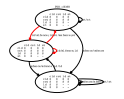
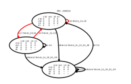
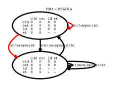
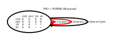
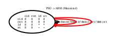
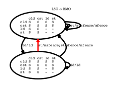
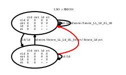
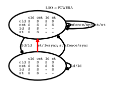
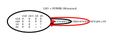
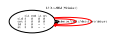
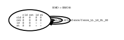
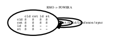
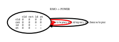
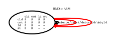
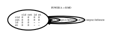
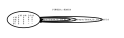
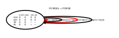
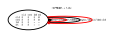
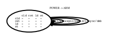
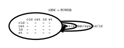
	TSO	PLO	PSO	LSO	RMO	RMO16	POWERA	POWER	ARM
SC									
TSO	-								
PLO	-	-							
PSO	-		-						
LSO	-	-	-	-					
RMO	-	-	-	-	-				
POWERA	-	-	-	-			-		
POWER	-	-	-	-	-	-	-	-	
ARM	-	-	-	-	-	-	-		-

Figure A.1.b: FSM summary.

A.2 Upstream MCMs

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.a: MOSTs for Upstream SC

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	M	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.b: MOSTs for Upstream TSO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	M	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.c: MOSTs for Upstream PLO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	-	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.d: MOSTs for Upstream PSO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	-	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.e: MOSTs for Upstream LSO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.f: MOSTs for Upstream RMO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

lwsync

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	N	S

sync

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.g: MOSTs for Upstream POWERA

PPO

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

lwsync

	ld	cld	st	cst
cld	S	S	S	S
cst	-	-	N	N
ld	S	S	S	S
st	-	-	N	N

sync

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.h: MOSTs for Upstream POWER

PPO

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

dmb

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.2.i: MOSTs for Upstream ARM

A.3 Downstream MCMs

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	M	S

msfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	S	S

mlfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	M	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.a: MOSTs for Downstream TSO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	M	S

msfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	S	S

mlfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	M	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.b: MOSTs for Downstream PLO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	-	S

msfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	S	S

mlfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	-	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.c: MOSTs for Downstream PSO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	-	S

msfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	S	S

mlfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	-	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.d: MOSTs for Downstream LSO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

msfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	S	S

mlfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	-	S

mfence

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.e: MOSTs for Downstream RMO

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

fence_LL_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	-	S
st	-	S	S	S

fence_LL_LS_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	S	S

fence_SL_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	S	S	S	S

fence_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	S	S

fence_LS_SL

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	S	S	-	S

fence_LL_SL_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	-	S
st	S	S	S	S

fence_LL_LS_SL_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

fence_LL

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	-	S
st	-	S	-	S

fence_LS_SL_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	S	S	S	S

fence_LS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	-	S

fence_LS_SS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	S	S

fence_SL

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	S	S	-	S

fence_LL_LS

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	-	S

fence_LL_LS_SL

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	-	S

fence_LL_SL

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	-	S
st	S	S	-	S

Figure A.3.f: MOSTs for Downstream RMO16

PPO

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

lwsync

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	N	S

sync

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.g: MOSTs for Downstream POWERA

PPO

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

lwsync

	ld	cld	st	cst
cld	S	S	S	S
cst	-	-	N	N
ld	S	S	S	S
st	-	-	N	N

sync

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.h: MOSTs for Downstream POWER

PPO

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

dmb

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.3.i: MOSTs for Downstream ARM

A.4 SC Upstream, TSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	S	-	S	-

Figure A.4.a: PPO of (Upstream SC - Downstream TSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	0
0	`SSSS SSSS SS-- SS--`	st	st	1
1	`SSSS SSSS SS-- SSS-`	ld	mlfence; ld	0
1	`SSSS SSSS SS-- SSS-`	st	st	1

Figure A.4.b: FSM Transition Table

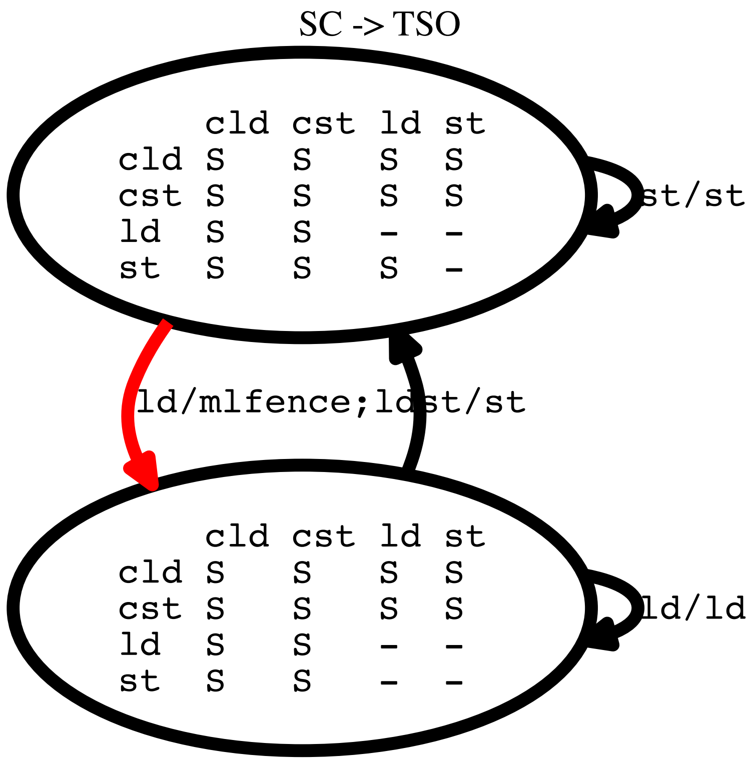


Figure A.4.c: FSM

A.5 SC Upstream, PLO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	-	-
st	S	-	S	-

Figure A.5.a: PPO of (Upstream SC - Downstream PLO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS*- SS*-`	ld	mlfence; ld	0
0	`SSSS SSSS SS*- SS*-`	st	st	0

Figure A.5.b: FSM Transition Table

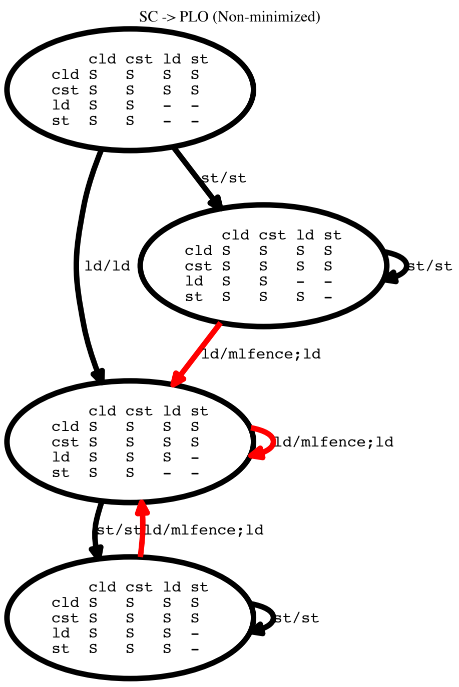


Figure A.5.c: FSM (Pre-minimization)

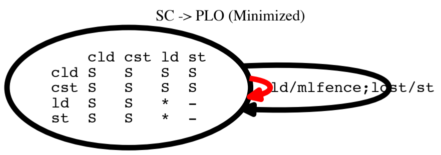


Figure A.5.d: FSM

A.6 SC Upstream, PSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	S	-	S	-

Figure A.6.a: PPO of (Upstream SC - Downstream PSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SSSS`	ld	mlfence; ld	1
0	`SSSS SSSS SS-- SSSS`	st	msfence; st	0
1	`SSSS SSSS SS-- SS-S`	ld	ld	1
1	`SSSS SSSS SS-- SS-S`	st	msfence; st	0

Figure A.6.b: FSM Transition Table

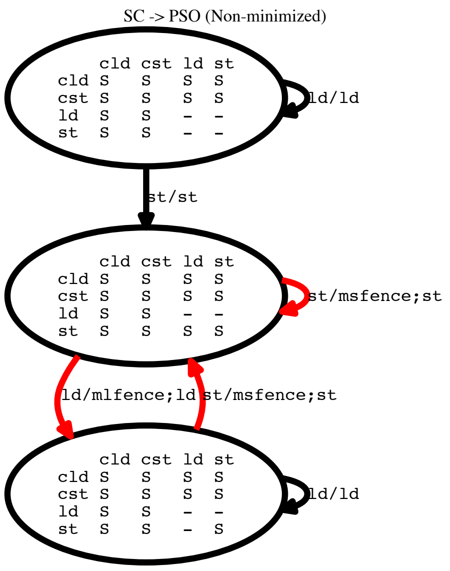


Figure A.6.c: FSM (Pre-minimization)

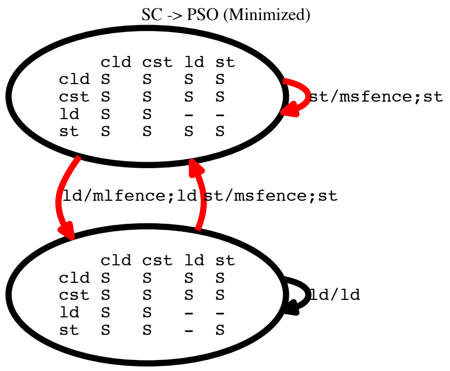


Figure A.6.d: FSM

A.7 SC Upstream, LSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	-	-
st	S	-	S	-

Figure A.7.a: PPO of (Upstream SC - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS*- SS*S`	ld	mlfence; ld	0
0	`SSSS SSSS SS*- SS*S`	st	msfence; st	0

Figure A.7.b: FSM Transition Table

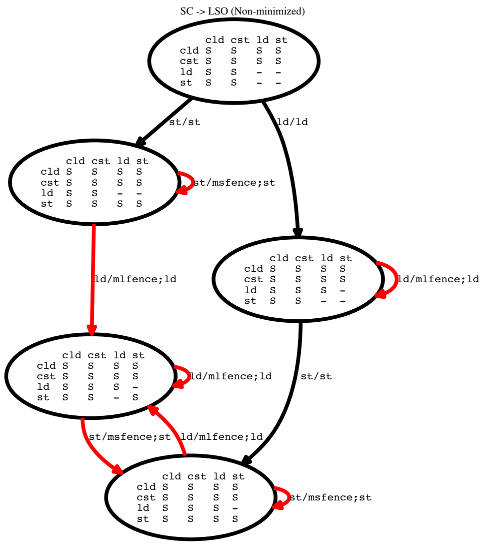


Figure A.7.c: FSM (Pre-minimization)

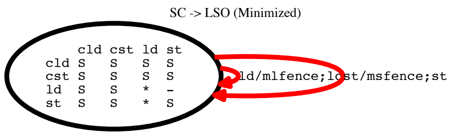


Figure A.7.d: FSM

A.8 SC Upstream, RMO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	S	-	S	-

Figure A.8.a: PPO of (Upstream SC - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS**`	ld	mlfence; ld	0
0	`SSSS SSSS SS** SS**`	st	msfence; st	0

Figure A.8.b: FSM Transition Table

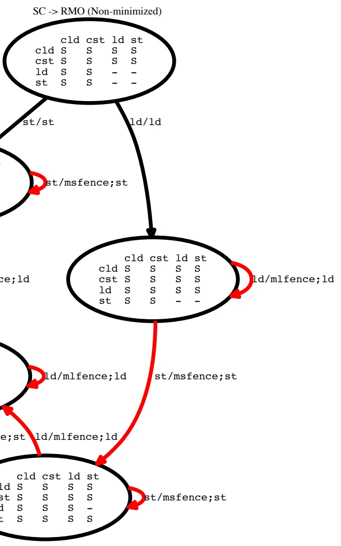


Figure A.8.c: FSM (Pre-minimization)

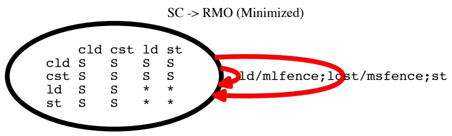


Figure A.8.d: FSM

A.9 SC Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	S	-	S	-

Figure A.9.a: PPO of (Upstream SC - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SSS- SSSS`	ld	fence_LL_SL; ld	1
0	`SSSS SSSS SSS- SSSS`	st	fence_SS; st	0
1	`SSSS SSSS SSSS SS-S`	ld	fence_LL; ld	1
1	`SSSS SSSS SSSS SS-S`	st	fence_LS_SS; st	0

Figure A.9.b: FSM Transition Table

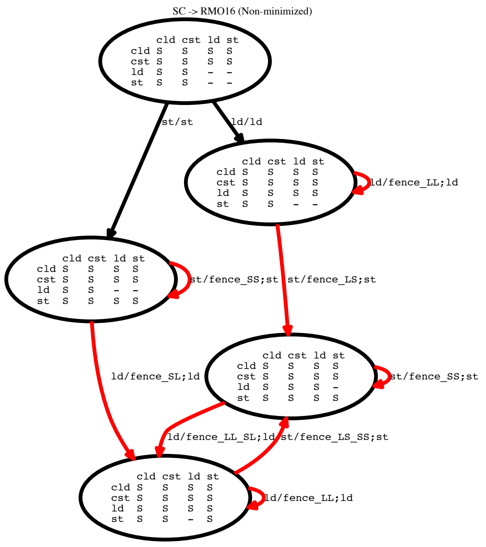

SC -> RMO16 (Non-minimized)
cld cst ld st / cld S S S S / cst S S S S / ld S S - - / st S S - -
st/st
ld/ld
cld cst ld st / cld S S S S / cst S S S S / ld S S S S / st S S - -
ld/fence_LL;ld
cld cst ld st / cld S S S S / cst S S S S / ld S S - - / st S S S S
st/fence_SS;st
st/fence_LS;st
cld cst ld st / cld S S S S / cst S S S S / ld S S S - / st S S S S
st/fence_SS;st
ld/fence_SL;ld
ld/fence_LL_SL;ld
st/fence_LS_SS;st
cld cst ld st / cld S S S S / cst S S S S / ld S S S S / st S S - S
ld/fence_LL;ld

Figure A.9.c: FSM (Pre-minimization)

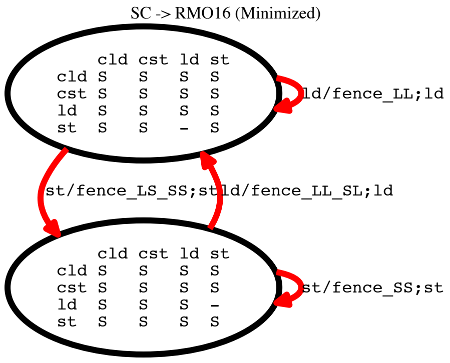

SC -> RMO16 (Minimized)
cld cst ld st / cld S S S S / cst S S S S / ld S S S S / st S S - S
ld/fence_LL;ld
st/fence_LS_SS;st
ld/fence_LL_SL;ld
cld cst ld st / cld S S S S / cst S S S S / ld S S S - / st S S S S
st/fence_SS;st

Figure A.9.d: FSM

A.10 SC Upstream, POWERA Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	S	-	S	-

Figure A.10.a: PPO of (Upstream SC - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SSSS`	ld	sync; ld	1
0	`SSSS SSSS SS-- SSSS`	st	sync; st	0
1	`SSSS SSSS SSSS SS--`	ld	lwsync; ld	1
1	`SSSS SSSS SSSS SS--`	st	lwsync; st	0

Figure A.10.b: FSM Transition Table

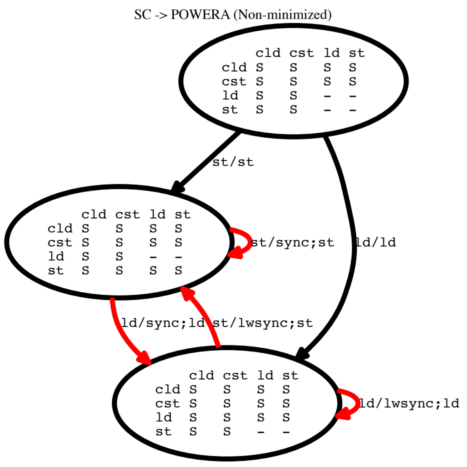


Figure A.10.c: FSM (Pre-minimization)

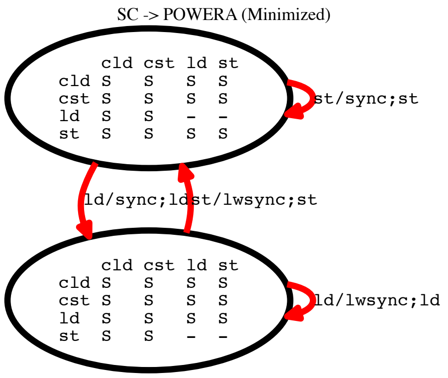


Figure A.10.d: FSM

A.11 SC Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.11.a: PPO of (Upstream SC - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS**`	ld	sync; ld	0
0	`SSSS SSSS SS** SS**`	st	sync; st	0

Figure A.11.b: FSM Transition Table

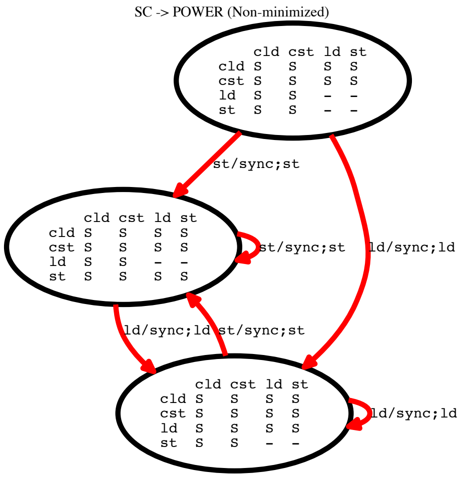


Figure A.11.c: FSM (Pre-minimization)

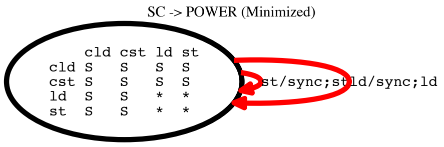


Figure A.11.d: FSM

A.12 SC Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	S	S	S	S

Figure A.12.a: PPO of (Upstream SC - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS**`	ld	dmb; ld	0
0	`SSSS SSSS SS** SS**`	st	dmb; st	0

Figure A.12.b: FSM Transition Table

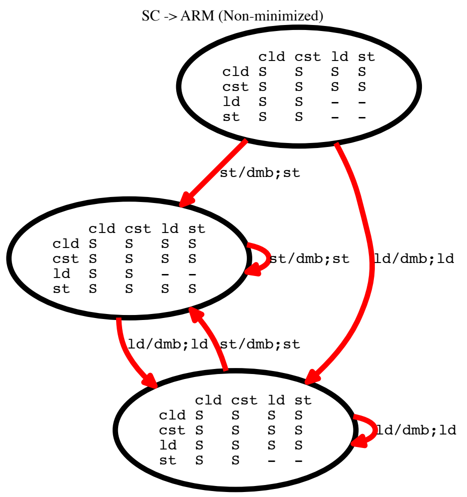


Figure A.12.c: FSM (Pre-minimization)

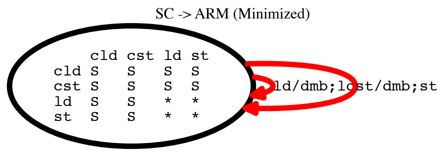


Figure A.12.d: FSM

A.13 TSO Upstream, PLO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	-	-
st	-	-	-	-

Figure A.13.a: PPO of (Upstream TSO - Downstream PLO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	1
0	`SSSS SSSS SS-- SS--`	mfence	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	0
1	`SSSS SSSS SSS- SS--`	ld	mlfence; ld	1
1	`SSSS SSSS SSS- SS--`	mfence	mfence	0
1	`SSSS SSSS SSS- SS--`	st	st	1

Figure A.13.b: FSM Transition Table

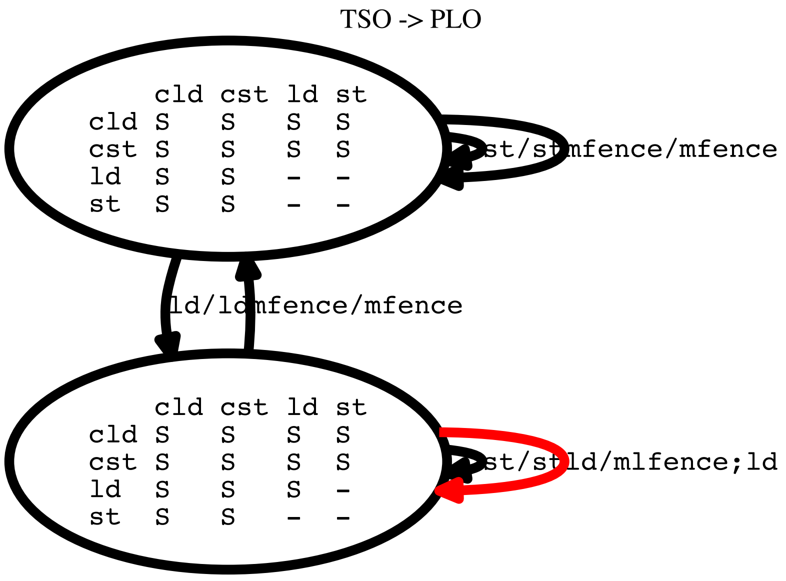


Figure A.13.c: FSM

A.14 TSO Upstream, PSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	M	-

Figure A.14.a: PPO of (Upstream TSO - Downstream PSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS-M`	ld	ld	0
0	`SSSS SSSS SS-- SS-M`	mfence	mfence	1
0	`SSSS SSSS SS-- SS-M`	st	msfence; st	0
1	`SSSS SSSS SS-- SS--`	ld	ld	1
1	`SSSS SSSS SS-- SS--`	mfence	mfence	1
1	`SSSS SSSS SS-- SS--`	st	st	0

Figure A.14.b: FSM Transition Table

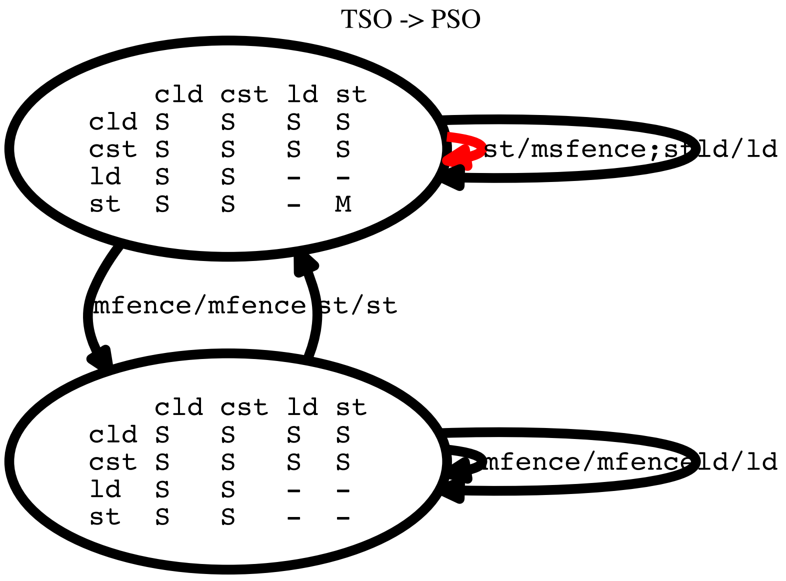


Figure A.14.c: FSM

A.15 TSO Upstream, LSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	-	-
st	-	-	M	-

Figure A.15.a: PPO of (Upstream TSO - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	3
0	`SSSS SSSS SS-- SS--`	mfence	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	1
1	`SSSS SSSS SS-- SS-M`	ld	ld	2
1	`SSSS SSSS SS-- SS-M`	mfence	mfence	0
1	`SSSS SSSS SS-- SS-M`	st	msfence; st	1
2	`SSSS SSSS SSS- SS-M`	ld	mlfence; ld	2
2	`SSSS SSSS SSS- SS-M`	mfence	mfence	0
2	`SSSS SSSS SSS- SS-M`	st	msfence; st	2
3	`SSSS SSSS SSS- SS--`	ld	mlfence; ld	3
3	`SSSS SSSS SSS- SS--`	mfence	mfence	0
3	`SSSS SSSS SSS- SS--`	st	st	2

Figure A.15.b: FSM Transition Table

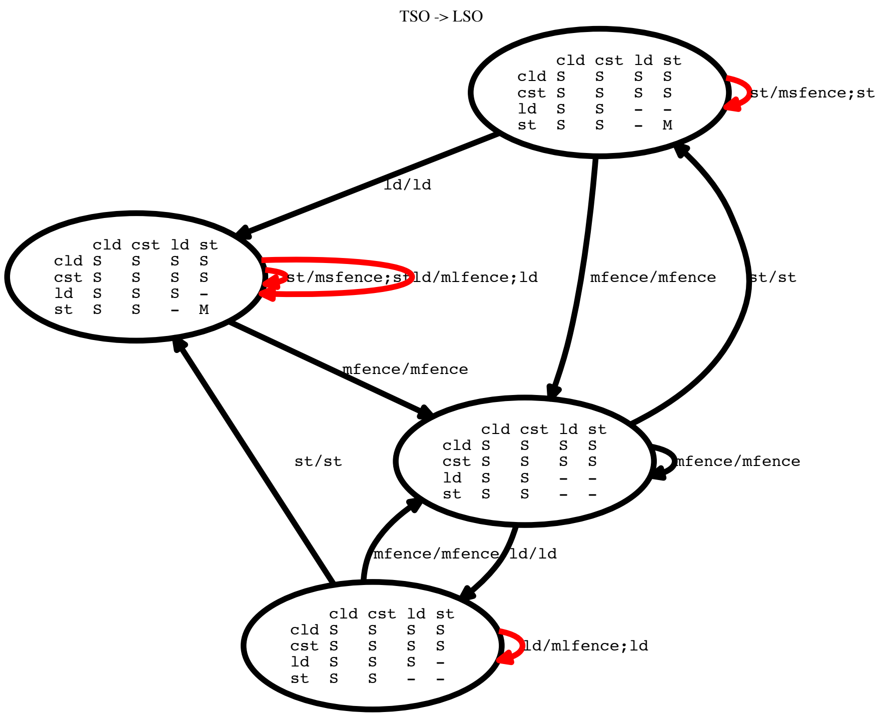


Figure A.15.c: FSM

A.16 TSO Upstream, RMO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	-	-	M	-

Figure A.16.a: PPO of (Upstream TSO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	2
0	`SSSS SSSS SS-- SS--`	mfence	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	1
1	`SSSS SSSS SS-- SS-M`	ld	ld	2
1	`SSSS SSSS SS-- SS-M`	mfence	mfence	0
1	`SSSS SSSS SS-- SS-M`	st	msfence; st	1
2	`SSSS SSSS SSS* SS-*`	ld	mlfence; ld	2
2	`SSSS SSSS SSS* SS-*`	mfence	mfence	0
2	`SSSS SSSS SSS* SS-*`	st	msfence; st	2

Figure A.16.b: FSM Transition Table

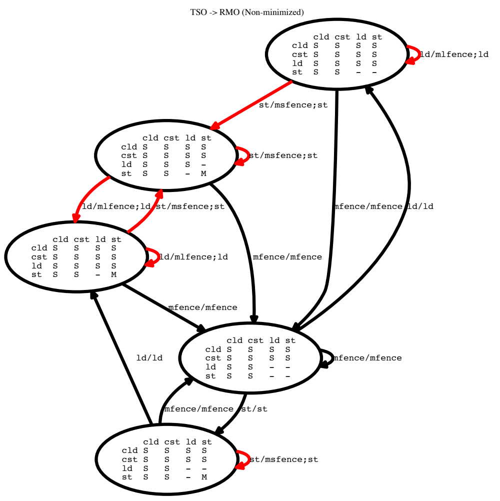


Figure A.16.c: FSM (Pre-minimization)

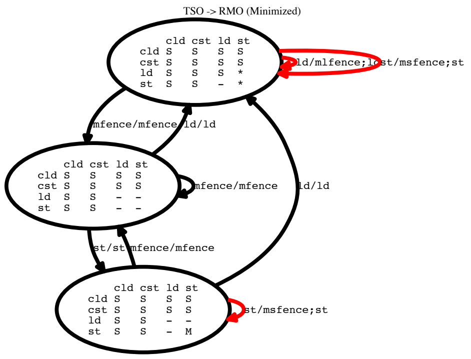


Figure A.16.d: FSM

A.17 TSO Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	-	-	M	-

Figure A.17.a: PPO of (Upstream TSO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SSSS SS--`	ld	fence_LL; ld	0
0	`SSSS SSSS SSSS SS--`	mfence	fence_LL_LS_SL_SS	2
0	`SSSS SSSS SSSS SS--`	st	fence_LS; st	4
1	`SSSS SSSS SSSS SS-M`	ld	fence_LL; ld	1
1	`SSSS SSSS SSSS SS-M`	mfence	fence_LL_LS_SL_SS	2
1	`SSSS SSSS SSSS SS-M`	st	fence_LS_SS; st	4
2	`SSSS SSSS SS-- SS--`	ld	ld	0
2	`SSSS SSSS SS-- SS--`	mfence	fence_LL_LS_SL_SS	2
2	`SSSS SSSS SS-- SS--`	st	st	3
3	`SSSS SSSS SS-- SS-M`	ld	ld	1
3	`SSSS SSSS SS-- SS-M`	mfence	fence_LL_LS_SL_SS	2
3	`SSSS SSSS SS-- SS-M`	st	fence_SS; st	3
4	`SSSS SSSS SSS- SS-M`	ld	fence_LL; ld	1
4	`SSSS SSSS SSS- SS-M`	mfence	fence_LL_LS_SL_SS	2
4	`SSSS SSSS SSS- SS-M`	st	fence_SS; st	4

Figure A.17.b: FSM Transition Table

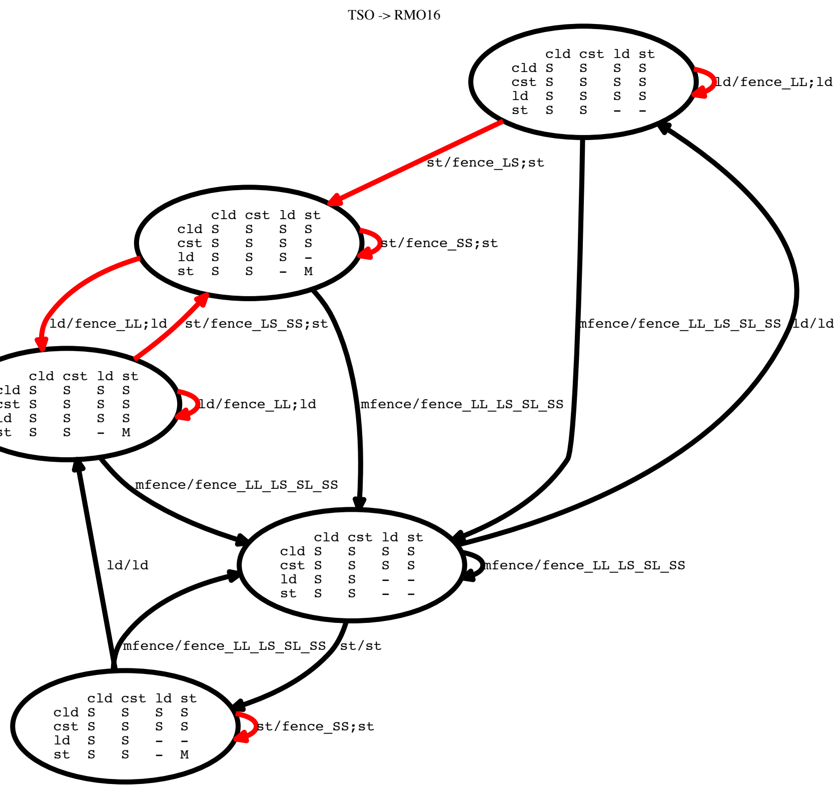


Figure A.17.c: FSM

A.18 TSO Upstream, POWERA Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	-	-	M	-

Figure A.18.a: PPO of (Upstream TSO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	3
0	SSSS SSSS SS-- SS--	mfence	sync	0
0	SSSS SSSS SS-- SS--	st	st	1
1	SSSS SSSS SS-- SS-M	ld	ld	2
1	SSSS SSSS SS-- SS-M	mfence	sync	0
1	SSSS SSSS SS-- SS-M	st	sync; st	1
2	SSSS SSSS SSSS SS-M	ld	lwsync; ld	2
2	SSSS SSSS SSSS SS-M	mfence	sync	0
2	SSSS SSSS SSSS SS-M	st	sync; st	1
3	SSSS SSSS SSSS SS--	ld	lwsync; ld	3
3	SSSS SSSS SSSS SS--	mfence	sync	0
3	SSSS SSSS SSSS SS--	st	lwsync; st	1

Figure A.18.b: FSM Transition Table

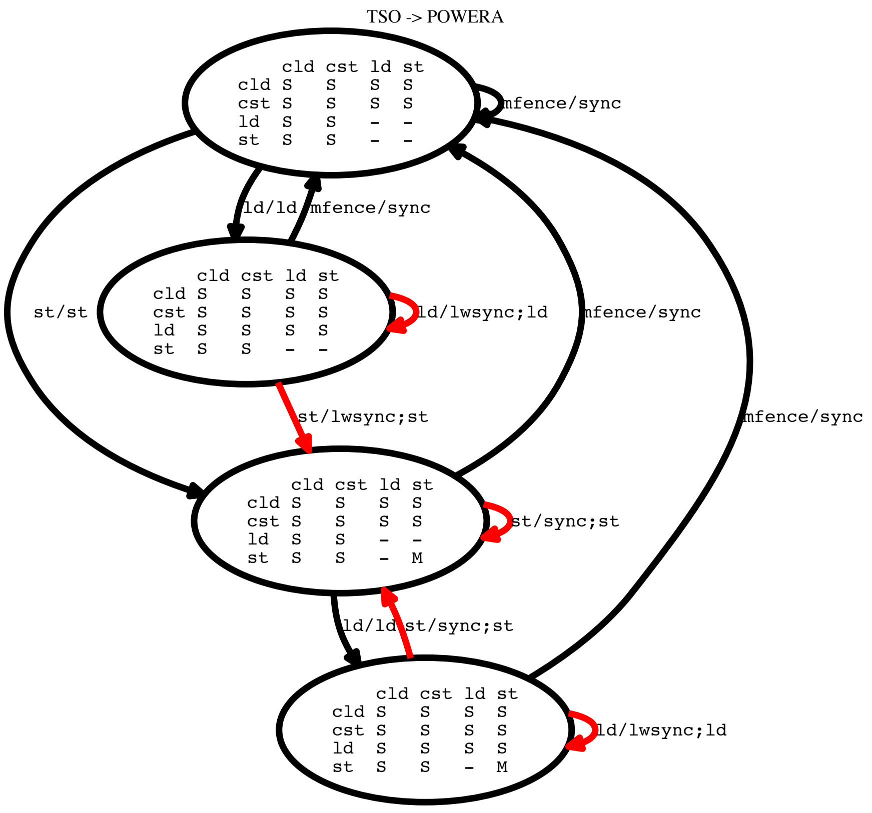


Figure A.18.c: FSM

A.19 TSO Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	M	S

Figure A.19.a: PPO of (Upstream TSO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS-*`	ld	sync; ld	0
0	`SSSS SSSS SS** SS-*`	mfence	sync	0
0	`SSSS SSSS SS** SS-*`	st	sync; st	0

Figure A.19.b: FSM Transition Table

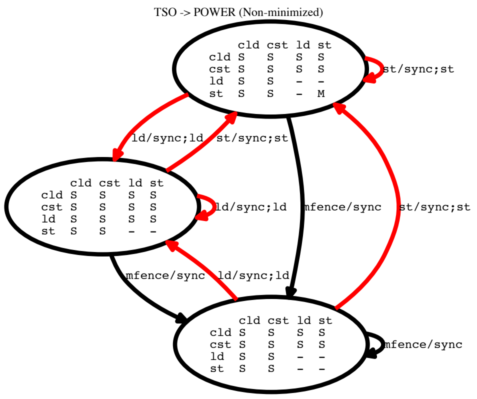


Figure A.19.c: FSM (Pre-minimization)

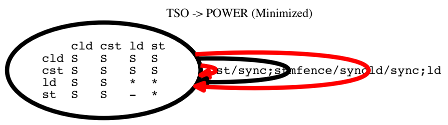


Figure A.19.d: FSM

A.20 TSO Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	M	S

Figure A.20.a: PPO of (Upstream TSO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS-*`	ld	dmb; ld	0
0	`SSSS SSSS SS** SS-*`	mfence	dmb	0
0	`SSSS SSSS SS** SS-*`	st	dmb; st	0

Figure A.20.b: FSM Transition Table

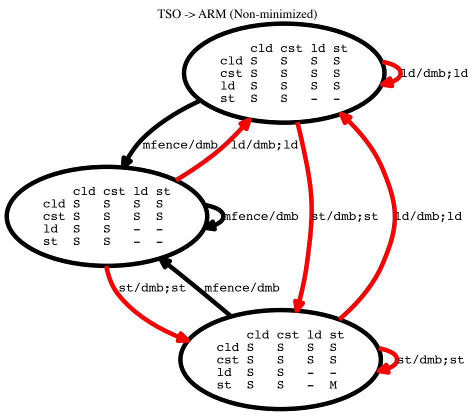


Figure A.20.c: FSM (Pre-minimization)

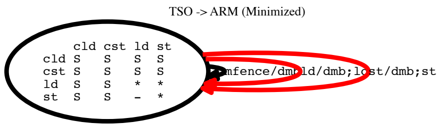


Figure A.20.d: FSM

A.21 PLO Upstream, PSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	M	-

Figure A.21.a: PPO of (Upstream PLO - Downstream PSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS-M`	ld	ld	0
0	`SSSS SSSS SS-- SS-M`	mfence	mfence	1
0	`SSSS SSSS SS-- SS-M`	st	msfence; st	0
1	`SSSS SSSS SS-- SS--`	ld	ld	1
1	`SSSS SSSS SS-- SS--`	mfence	mfence	1
1	`SSSS SSSS SS-- SS--`	st	st	0

Figure A.21.b: FSM Transition Table

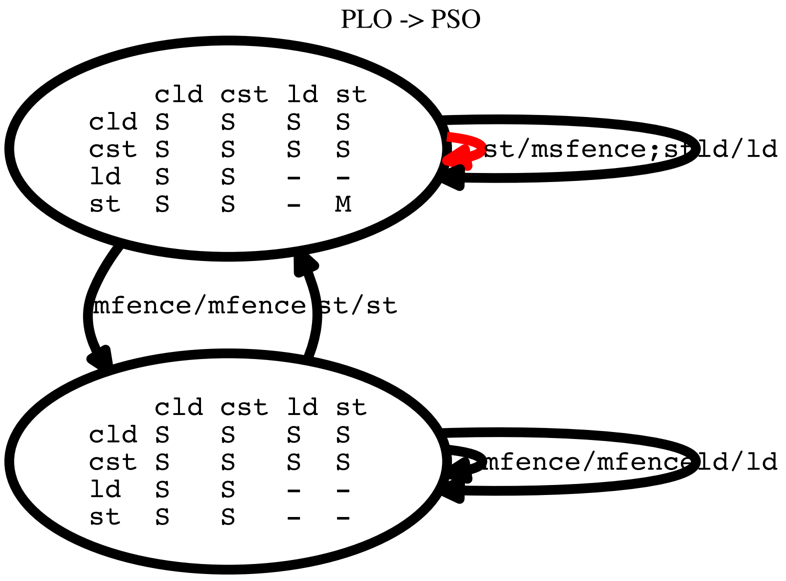


Figure A.21.c: FSM

A.22 PLO Upstream, LSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	M	-

Figure A.22.a: PPO of (Upstream PLO - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS-M`	ld	ld	0
0	`SSSS SSSS SS-- SS-M`	mfence	mfence	1
0	`SSSS SSSS SS-- SS-M`	st	msfence; st	0
1	`SSSS SSSS SS-- SS--`	ld	ld	1
1	`SSSS SSSS SS-- SS--`	mfence	mfence	1
1	`SSSS SSSS SS-- SS--`	st	st	0

Figure A.22.b: FSM Transition Table

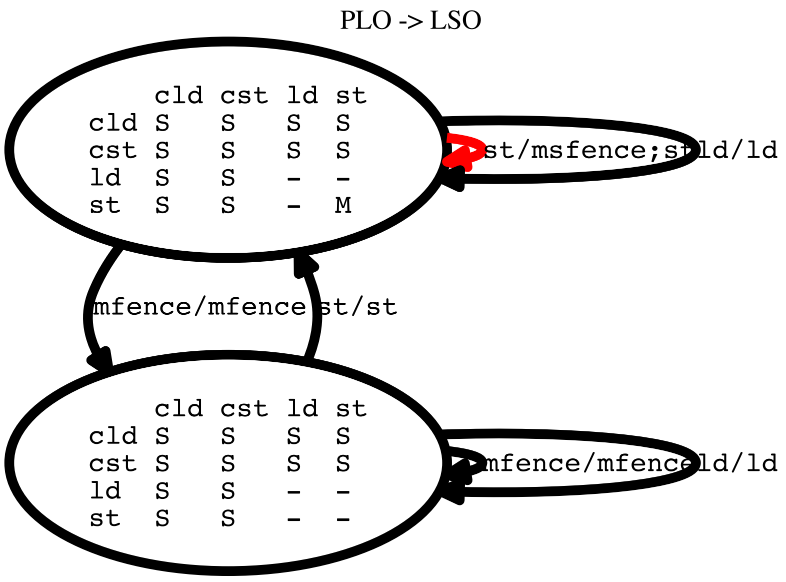


Figure A.22.c: FSM

A.23 PLO Upstream, RMO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	S	-
st	-	-	M	-

Figure A.23.a: PPO of (Upstream PLO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	1
0	`SSSS SSSS SS-- SS--`	mfence	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	1
1	`SSSS SSSS SS-* SS-*`	ld	ld	1
1	`SSSS SSSS SS-* SS-*`	mfence	mfence	0
1	`SSSS SSSS SS-* SS-*`	st	msfence; st	1

Figure A.23.b: FSM Transition Table

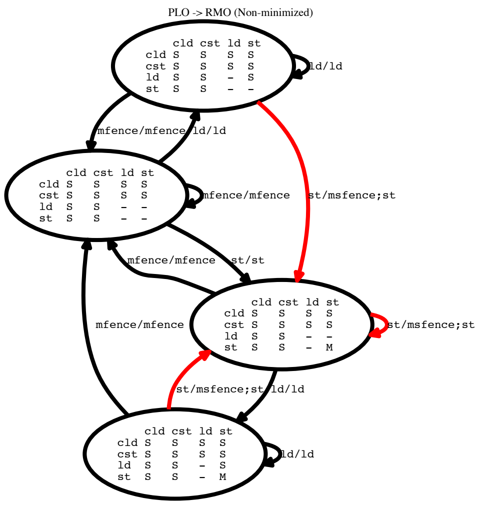


Figure A.23.c: FSM (Pre-minimization)

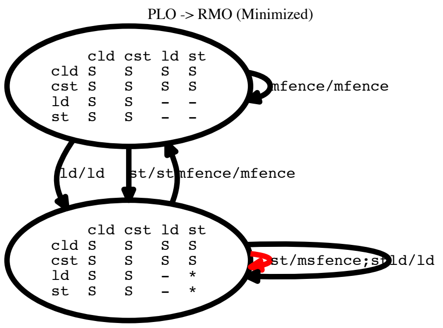


Figure A.23.d: FSM

A.24 PLO Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	S	-
st	-	-	M	-

Figure A.24.a: PPO of (Upstream PLO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS-M`	ld	ld	3
0	`SSSS SSSS SS-- SS-M`	mfence	fence_LL_LS_SL_SS	1
0	`SSSS SSSS SS-- SS-M`	st	fence_SS; st	0
1	`SSSS SSSS SS-- SS--`	ld	ld	2
1	`SSSS SSSS SS-- SS--`	mfence	fence_LL_LS_SL_SS	1
1	`SSSS SSSS SS-- SS--`	st	st	0
2	`SSSS SSSS SS-S SS--`	ld	ld	2
2	`SSSS SSSS SS-S SS--`	mfence	fence_LL_LS_SL_SS	1
2	`SSSS SSSS SS-S SS--`	st	fence_LS; st	0
3	`SSSS SSSS SS-S SS-M`	ld	ld	3
3	`SSSS SSSS SS-S SS-M`	mfence	fence_LL_LS_SL_SS	1
3	`SSSS SSSS SS-S SS-M`	st	fence_LS_SS; st	0

Figure A.24.b: FSM Transition Table

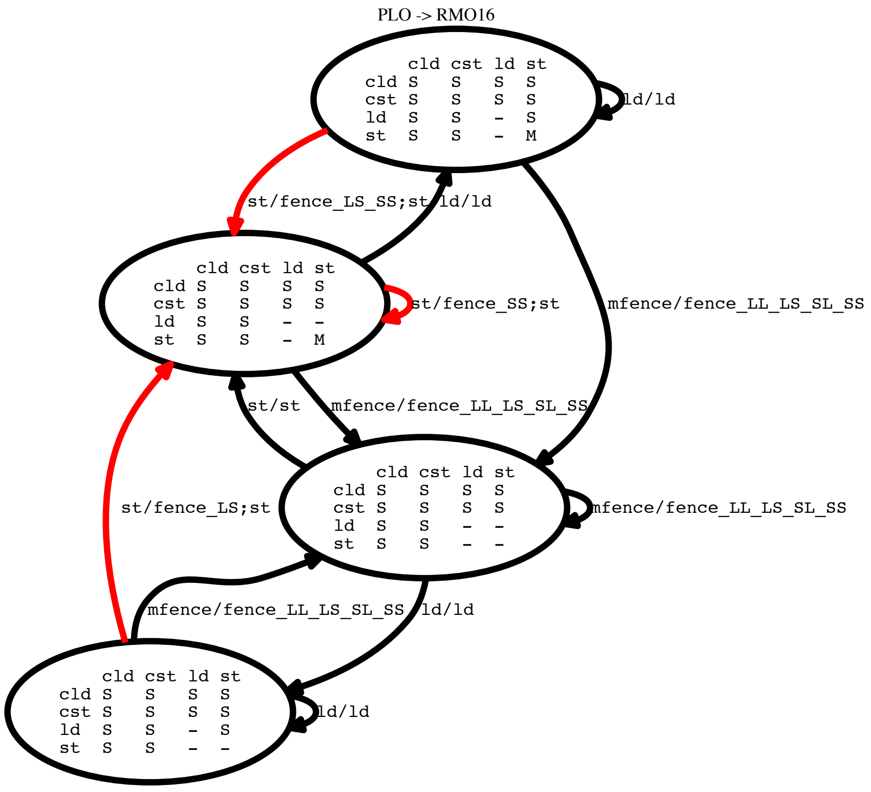

PLO -> RMO16
cld cst ld st / cld S S S S / cst S S S S / ld S S - S / st S S - M
ld/ld
st/fence_LS_SS;st
ld/ld
cld cst ld st / cld S S S S / cst S S S S / ld S S - - / st S S - M
st/fence_SS;st
mfence/fence_LL_LS_SL_SS
st/st
mfence/fence_LL_LS_SL_SS
cld cst ld st / cld S S S S / cst S S S S / ld S S - - / st S S - -
st/fence_LS;st
mfence/fence_LL_LS_SL_SS
mfence/fence_LL_LS_SL_SS
ld/ld
cld cst ld st / cld S S S S / cst S S S S / ld S S - S / st S S - -
ld/ld

Figure A.24.c: FSM

A.25 PLO Upstream, POWERA Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	S	-
st	-	-	M	-

Figure A.25.a: PPO of (Upstream PLO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	2
0	`SSSS SSSS SS-- SS--`	mfence	sync	0
0	`SSSS SSSS SS-- SS--`	st	st	1
1	`SSSS SSSS SS-* SS-M`	ld	ld	1
1	`SSSS SSSS SS-* SS-M`	mfence	sync	0
1	`SSSS SSSS SS-* SS-M`	st	sync; st	1
2	`SSSS SSSS SS-S SS--`	ld	ld	2
2	`SSSS SSSS SS-S SS--`	mfence	sync	0
2	`SSSS SSSS SS-S SS--`	st	lwsync; st	1

Figure A.25.b: FSM Transition Table

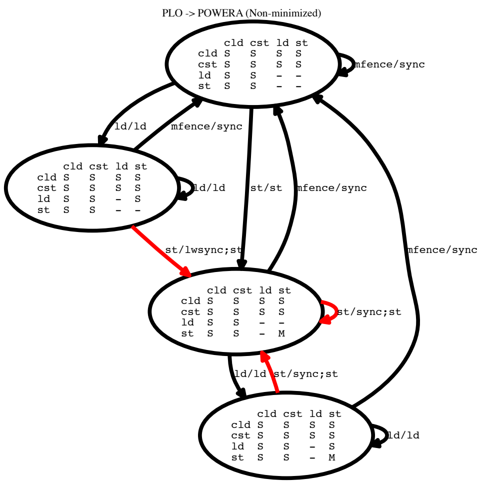


Figure A.25.c: FSM (Pre-minimization)

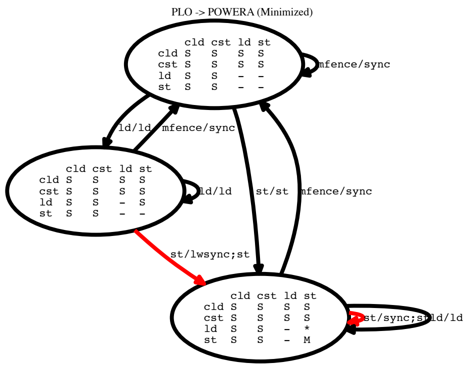


Figure A.25.d: FSM

A.26 PLO Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	M	S

Figure A.26.a: PPO of (Upstream PLO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-* SS-*`	ld	sync; ld	0
0	`SSSS SSSS SS-* SS-*`	mfence	sync	0
0	`SSSS SSSS SS-* SS-*`	st	sync; st	0

Figure A.26.b: FSM Transition Table

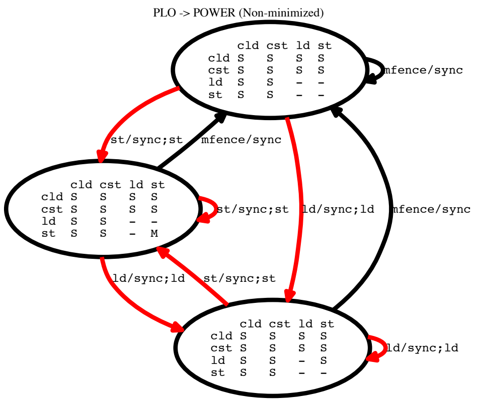


Figure A.26.c: FSM (Pre-minimization)

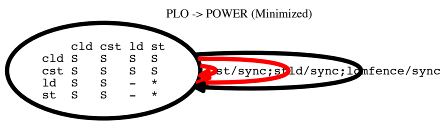


Figure A.26.d: FSM

A.27 PLO Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	M	S

Figure A.27.a: PPO of (Upstream PLO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-* SS-*`	ld	dmb; ld	0
0	`SSSS SSSS SS-* SS-*`	mfence	dmb	0
0	`SSSS SSSS SS-* SS-*`	st	dmb; st	0

Figure A.27.b: FSM Transition Table

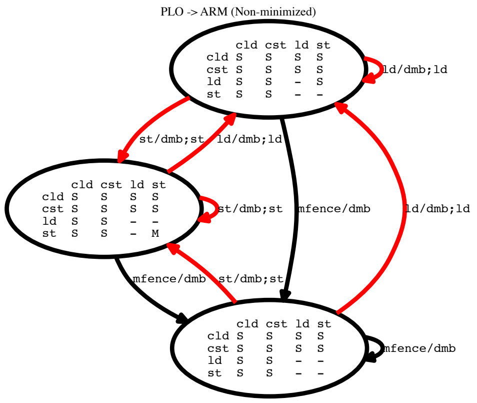


Figure A.27.c: FSM (Pre-minimization)

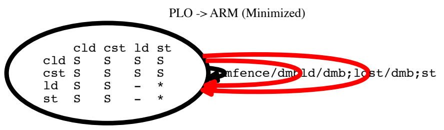


Figure A.27.d: FSM

A.28 PSO Upstream, PLO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	-	-
st	-	-	-	-

Figure A.28.a: PPO of (Upstream PSO - Downstream PLO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	1
0	`SSSS SSSS SS-- SS--`	mfence	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	0
1	`SSSS SSSS SSS- SS--`	ld	mlfence; ld	1
1	`SSSS SSSS SSS- SS--`	mfence	mfence	0
1	`SSSS SSSS SSS- SS--`	st	st	1

Figure A.28.b: FSM Transition Table

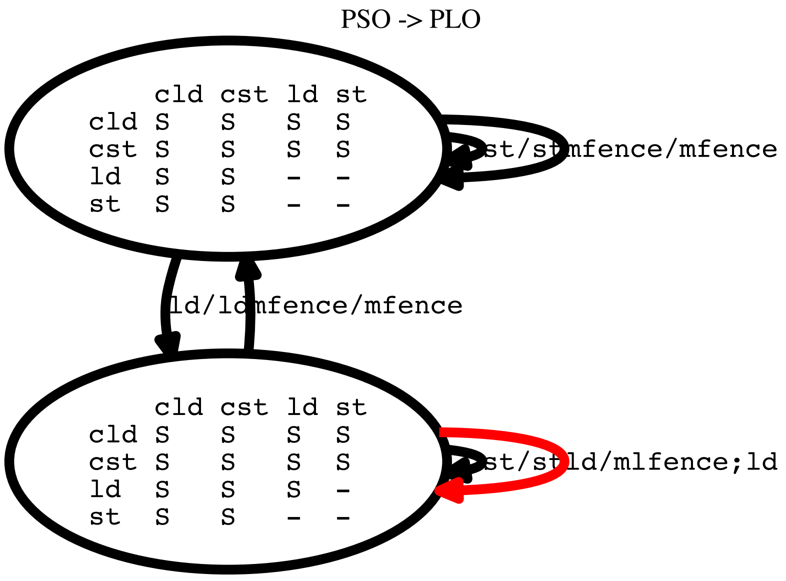


Figure A.28.c: FSM

A.29 PSO Upstream, LSO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	-	-
st	-	-	-	-

Figure A.29.a: PPO of (Upstream PSO - Downstream LSO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	1
0	`SSSS SSSS SS-- SS--`	mfence	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	0
1	`SSSS SSSS SSS- SS--`	ld	mlfence; ld	1
1	`SSSS SSSS SSS- SS--`	mfence	mfence	0
1	`SSSS SSSS SSS- SS--`	st	st	1

Figure A.29.b: FSM Transition Table

PSO -> LSO

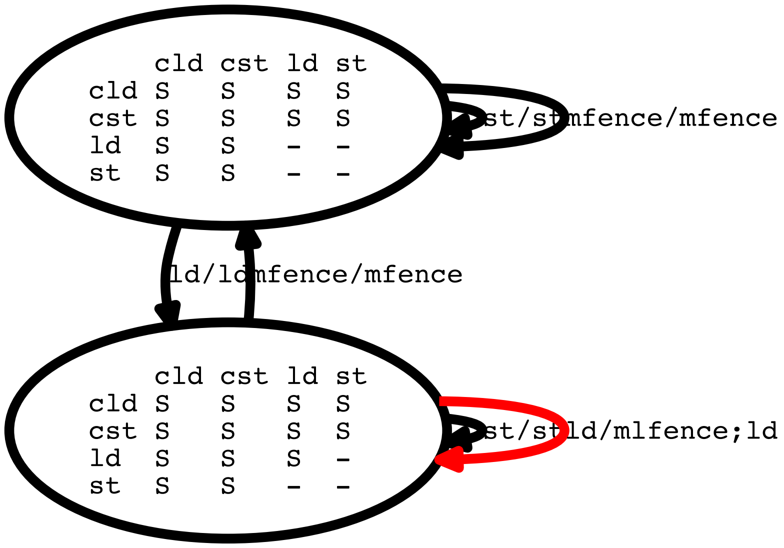


Figure A.29.c: FSM

A.30 PSO Upstream, RMO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	-	-	-	-

Figure A.30.a: PPO of (Upstream PSO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SSSS SS--	ld	mlfence; ld	0
0	SSSS SSSS SSSS SS--	mfence	mfence	1
0	SSSS SSSS SSSS SS--	st	msfence; st	2
1	SSSS SSSS SS-- SS--	ld	ld	0
1	SSSS SSSS SS-- SS--	mfence	mfence	1
1	SSSS SSSS SS-- SS--	st	st	1
2	SSSS SSSS SSS- SS--	ld	mlfence; ld	0
2	SSSS SSSS SSS- SS--	mfence	mfence	1
2	SSSS SSSS SSS- SS--	st	st	2

Figure A.30.b: FSM Transition Table

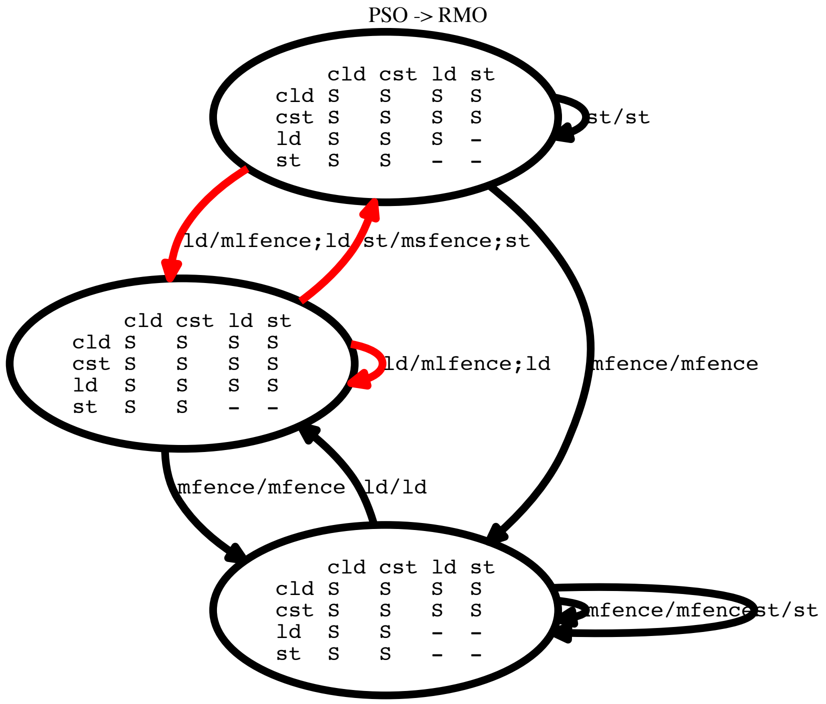


Figure A.30.c: FSM

A.31 PSO Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	-	-	-	-

Figure A.31.a: PPO of (Upstream PSO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SSSS SS--`	ld	fence_LL; ld	0
0	`SSSS SSSS SSSS SS--`	mfence	fence_LL_LS_SL_SS	1
0	`SSSS SSSS SSSS SS--`	st	fence_LS; st	2
1	`SSSS SSSS SS-- SS--`	ld	ld	0
1	`SSSS SSSS SS-- SS--`	mfence	fence_LL_LS_SL_SS	1
1	`SSSS SSSS SS-- SS--`	st	st	1
2	`SSSS SSSS SSS- SS--`	ld	fence_LL; ld	0
2	`SSSS SSSS SSS- SS--`	mfence	fence_LL_LS_SL_SS	1
2	`SSSS SSSS SSS- SS--`	st	st	2

Figure A.31.b: FSM Transition Table

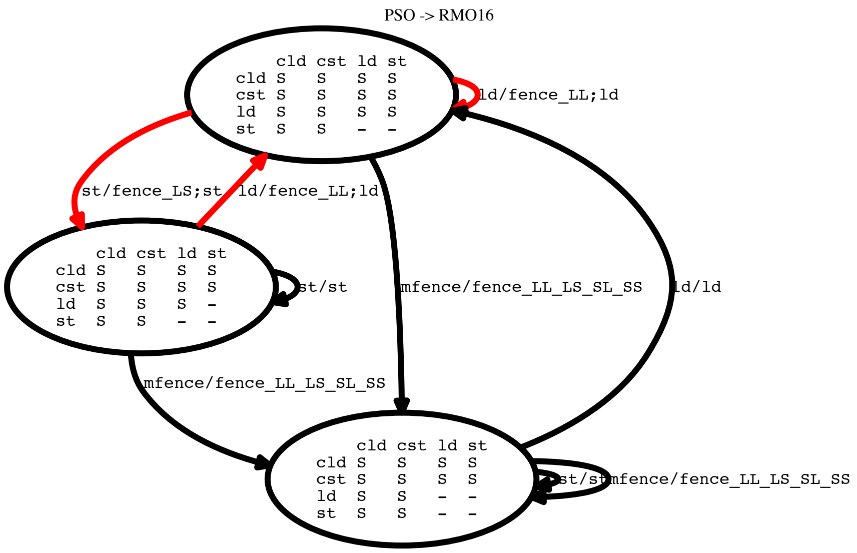


Figure A.31.c: FSM

A.32 PSO Upstream, POWERA Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	S	-	S	-
st	-	-	-	-

Figure A.32.a: PPO of (Upstream PSO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SSSS SS--`	ld	lwsync; ld	0
0	`SSSS SSSS SSSS SS--`	mfence	sync	1
0	`SSSS SSSS SSSS SS--`	st	lwsync; st	1
1	`SSSS SSSS SS-- SS--`	ld	ld	0
1	`SSSS SSSS SS-- SS--`	mfence	sync	1
1	`SSSS SSSS SS-- SS--`	st	st	1

Figure A.32.b: FSM Transition Table

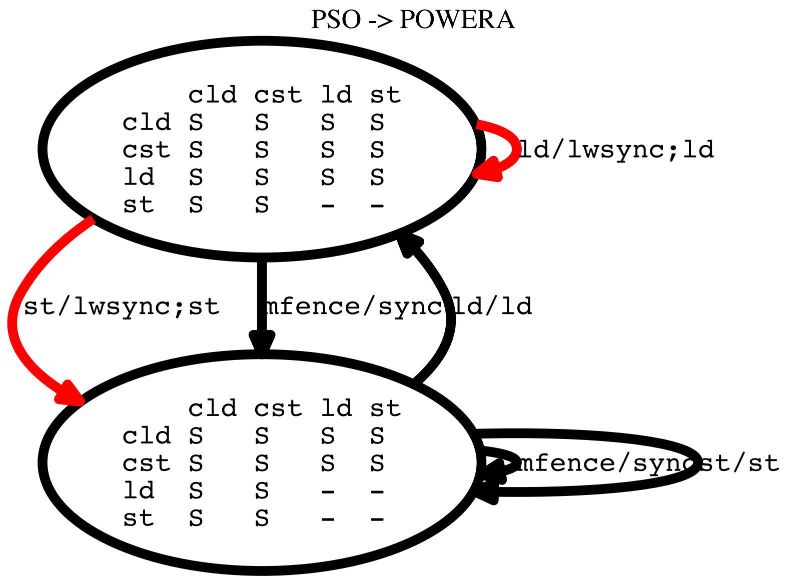


Figure A.32.c: FSM

A.33 PSO Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	-	S

Figure A.33.a: PPO of (Upstream PSO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS--`	ld	sync; ld	0
0	`SSSS SSSS SS** SS--`	mfence	sync	0
0	`SSSS SSSS SS** SS--`	st	sync; st	0

Figure A.33.b: FSM Transition Table

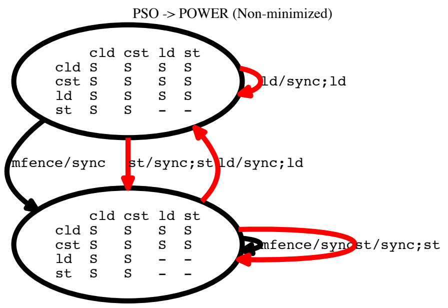


Figure A.33.c: FSM (Pre-minimization)

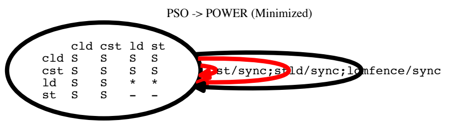


Figure A.33.d: FSM

A.34 PSO Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	S	S	S	S
st	-	S	-	S

Figure A.34.a: PPO of (Upstream PSO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS** SS--`	ld	dmb; ld	0
0	`SSSS SSSS SS** SS--`	mfence	dmb	0
0	`SSSS SSSS SS** SS--`	st	dmb; st	0

Figure A.34.b: FSM Transition Table

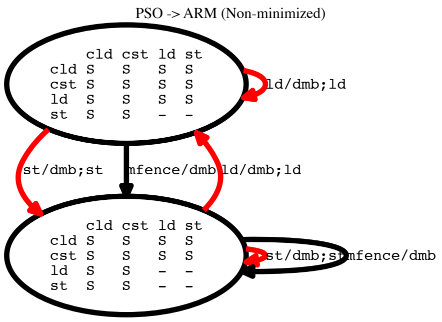


Figure A.34.c: FSM (Pre-minimization)

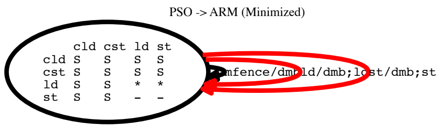


Figure A.34.d: FSM

A.35 LSO Upstream, RMO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	S	-
st	-	-	-	-

Figure A.35.a: PPO of (Upstream LSO - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	1
0	SSSS SSSS SS-- SS--	mfence	mfence	0
0	SSSS SSSS SS-- SS--	st	st	0
1	SSSS SSSS SS-S SS--	ld	ld	1
1	SSSS SSSS SS-S SS--	mfence	mfence	0
1	SSSS SSSS SS-S SS--	st	msfence; st	0

Figure A.35.b: FSM Transition Table

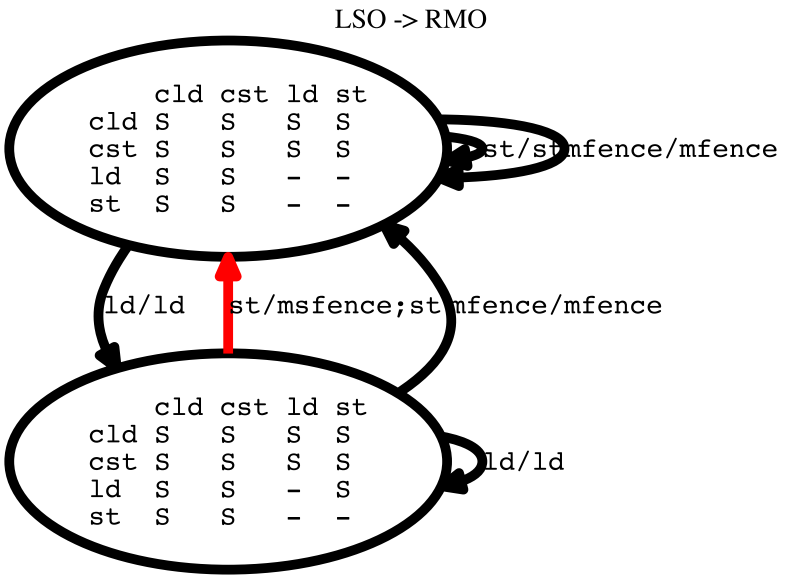


Figure A.35.c: FSM

A.36 LSO Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	S	-
st	-	-	-	-

Figure A.36.a: PPO of (Upstream LSO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	1
0	`SSSS SSSS SS-- SS--`	mfence	fence_LL_LS_SL_SS	0
0	`SSSS SSSS SS-- SS--`	st	st	0
1	`SSSS SSSS SS-S SS--`	ld	ld	1
1	`SSSS SSSS SS-S SS--`	mfence	fence_LL_LS_SL_SS	0
1	`SSSS SSSS SS-S SS--`	st	fence_LS; st	0

Figure A.36.b: FSM Transition Table

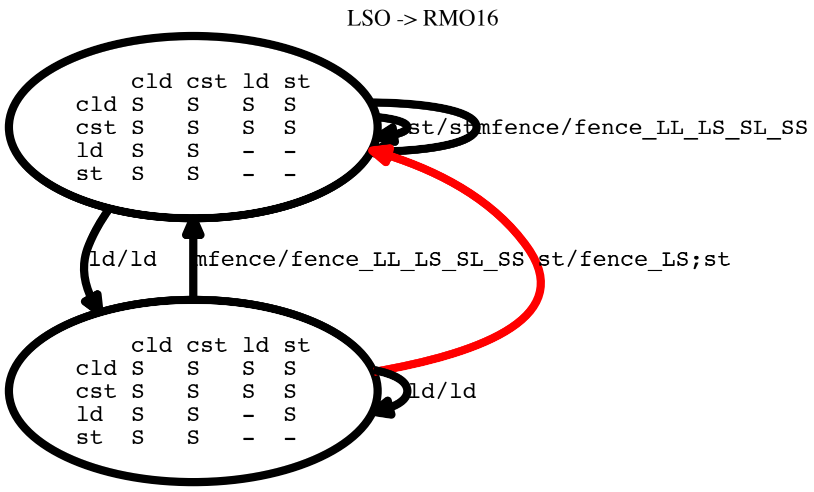


Figure A.36.c: FSM

A.37 LSO Upstream, POWERA Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	S	-
st	-	-	-	-

Figure A.37.a: PPO of (Upstream LSO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	1
0	`SSSS SSSS SS-- SS--`	mfence	sync	0
0	`SSSS SSSS SS-- SS--`	st	st	0
1	`SSSS SSSS SS-S SS--`	ld	ld	1
1	`SSSS SSSS SS-S SS--`	mfence	sync	0
1	`SSSS SSSS SS-S SS--`	st	lwsync; st	0

Figure A.37.b: FSM Transition Table

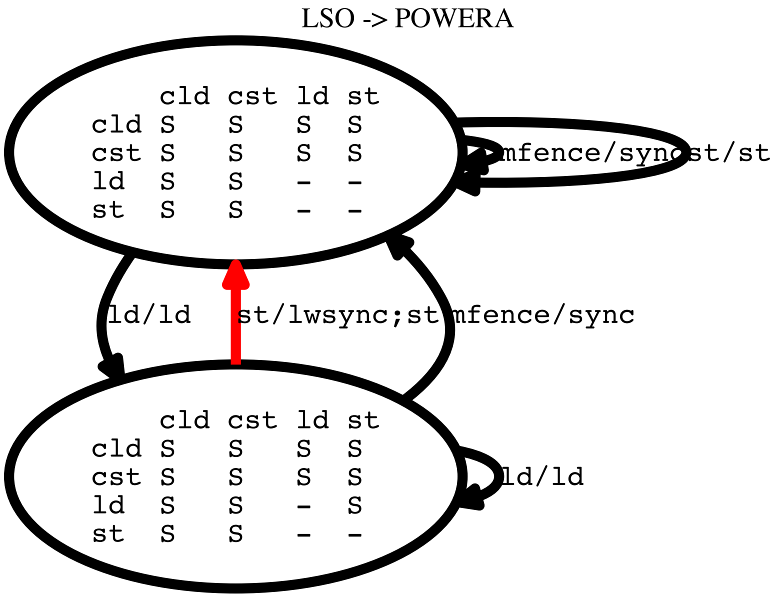


Figure A.37.c: FSM

A.38 LSO Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	-	S

Figure A.38.a: PPO of (Upstream LSO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-* SS--`	ld	sync; ld	0
0	`SSSS SSSS SS-* SS--`	mfence	sync	0
0	`SSSS SSSS SS-* SS--`	st	sync; st	0

Figure A.38.b: FSM Transition Table

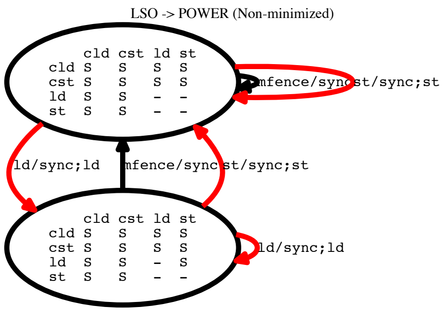


Figure A.38.c: FSM (Pre-minimization)

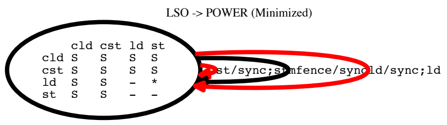


Figure A.38.d: FSM

A.39 LSO Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	S	S
st	-	S	-	S

Figure A.39.a: PPO of (Upstream LSO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-* SS--`	ld	dmb; ld	0
0	`SSSS SSSS SS-* SS--`	mfence	dmb	0
0	`SSSS SSSS SS-* SS--`	st	dmb; st	0

Figure A.39.b: FSM Transition Table

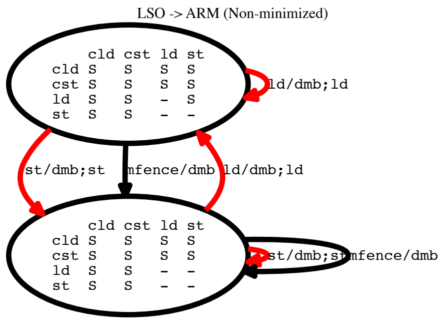


Figure A.39.c: FSM (Pre-minimization)

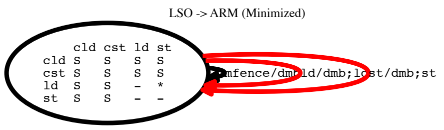


Figure A.39.d: FSM

A.40 RMO Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

Figure A.40.a: PPO of (Upstream RMO - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	SSSS SSSS SS-- SS--	ld	ld	0
0	SSSS SSSS SS-- SS--	mfence	fence_LL_LS_SL_SS	0
0	SSSS SSSS SS-- SS--	st	st	0

Figure A.40.b: FSM Transition Table

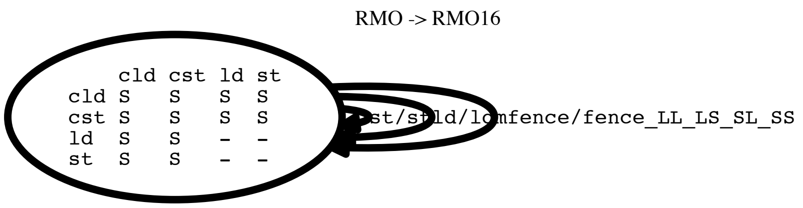


Figure A.40.c: FSM

A.41 RMO Upstream, POWERA Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

Figure A.41.a: PPO of (Upstream RMO - Downstream POWERA)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	0
0	`SSSS SSSS SS-- SS--`	mfence	sync	0
0	`SSSS SSSS SS-- SS--`	st	st	0

Figure A.41.b: FSM Transition Table

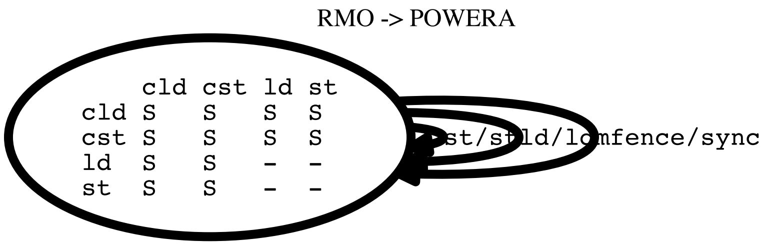


Figure A.41.c: FSM

A.42 RMO Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

Figure A.42.a: PPO of (Upstream RMO - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	sync; ld	0
0	`SSSS SSSS SS-- SS--`	mfence	sync	0
0	`SSSS SSSS SS-- SS--`	st	sync; st	0

Figure A.42.b: FSM Transition Table

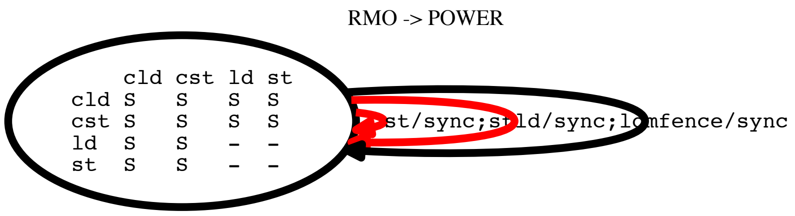


Figure A.42.c: FSM

A.43 RMO Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

Figure A.43.a: PPO of (Upstream RMO - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	dmb; ld	0
0	`SSSS SSSS SS-- SS--`	mfence	dmb	0
0	`SSSS SSSS SS-- SS--`	st	dmb; st	0

Figure A.43.b: FSM Transition Table

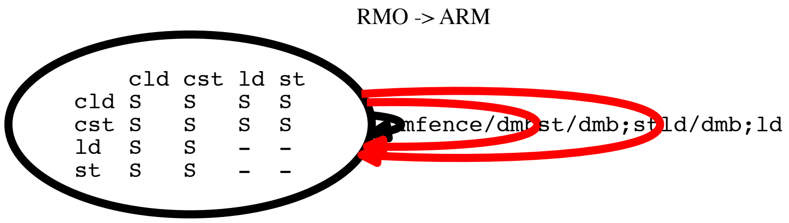


Figure A.43.c: FSM

A.44 POWERA Upstream, RMO Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

Figure A.44.a: PPO of (Upstream POWERA - Downstream RMO)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	0
0	`SSSS SSSS SS-- SS--`	lwsync	mfence	0
0	`SSSS SSSS SS-- SS--`	sync	mfence	0
0	`SSSS SSSS SS-- SS--`	st	st	0

Figure A.44.b: FSM Transition Table

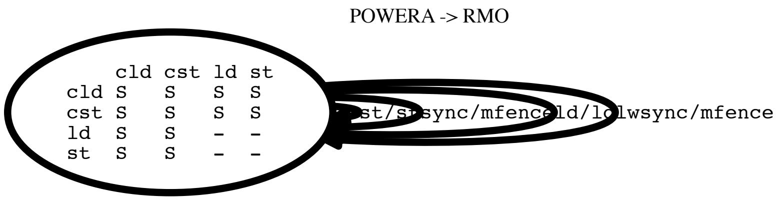


Figure A.44.c: FSM

A.45 POWERA Upstream, RMO16 Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

Figure A.45.a: PPO of (Upstream POWERA - Downstream RMO16)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	ld	0
0	`SSSS SSSS SS-- SS--`	lwsync	fence_LL_LS_SS	0
0	`SSSS SSSS SS-- SS--`	sync	fence_LL_LS_SL_SS	0
0	`SSSS SSSS SS-- SS--`	st	st	0

Figure A.45.b: FSM Transition Table

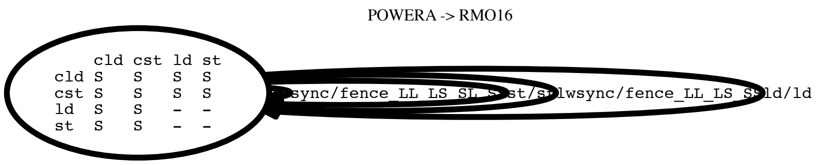


Figure A.45.c: FSM

A.46 POWERA Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

Figure A.46.a: PPO of (Upstream POWERA - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	sync; ld	0
0	`SSSS SSSS SS-- SS--`	lwsync	sync	0
0	`SSSS SSSS SS-- SS--`	sync	sync	0
0	`SSSS SSSS SS-- SS--`	st	sync; st	0

Figure A.46.b: FSM Transition Table

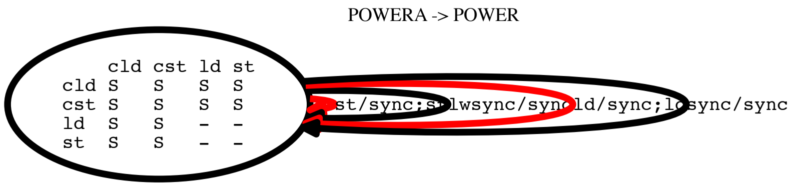


Figure A.46.c: FSM

A.47 POWERA Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	S	S	S	S
cst	S	S	S	S
ld	-	S	-	S
st	-	S	-	S

Figure A.47.a: PPO of (Upstream POWERA - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	`SSSS SSSS SS-- SS--`	ld	dmb; ld	0
0	`SSSS SSSS SS-- SS--`	lwsync	dmb	0
0	`SSSS SSSS SS-- SS--`	sync	dmb	0
0	`SSSS SSSS SS-- SS--`	st	dmb; st	0

Figure A.47.b: FSM Transition Table

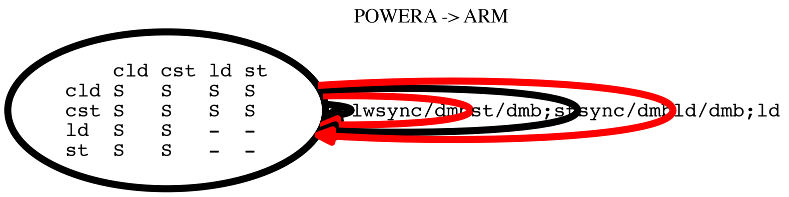


Figure A.47.c: FSM

A.48 POWER Upstream, ARM Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

Figure A.48.a: PPO of (Upstream POWER - Downstream ARM)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	---- ---- ---- ----	ld	ld	0
0	---- ---- ---- ----	lwsync	dmb	0
0	---- ---- ---- ----	sync	dmb	0
0	---- ---- ---- ----	st	st	0

Figure A.48.b: FSM Transition Table

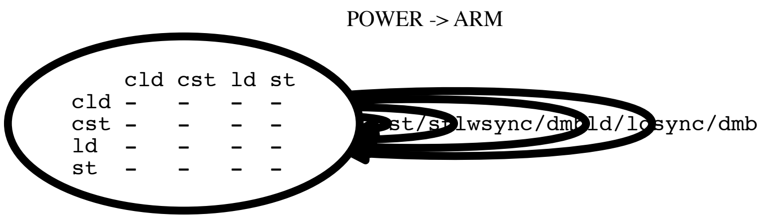


Figure A.48.c: FSM

A.49 ARM Upstream, POWER Downstream

PPO Diff.

	ld	cld	st	cst
cld	-	-	-	-
cst	-	-	-	-
ld	-	-	-	-
st	-	-	-	-

Figure A.49.a: PPO of (Upstream ARM - Downstream POWER)

Input			Output	
State	MOST	Op.	Op(s).	Next State
0	---- ---- ---- ----	dmb	sync	0
0	---- ---- ---- ----	ld	ld	0
0	---- ---- ---- ----	st	st	0

Figure A.49.b: FSM Transition Table

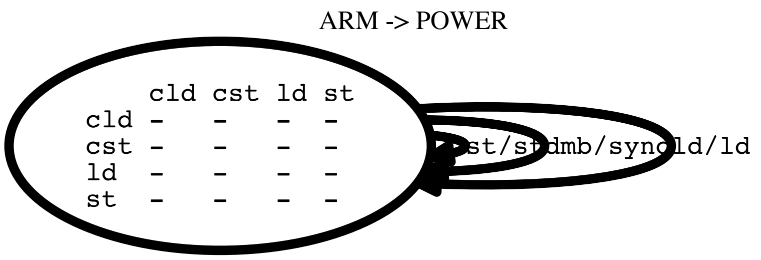


Figure A.49.c: FSM