A Formalization of a Simplified Subset of the Alpha Shared Memory Model

Mike Gordon	??? ??? ???
Computer Laboratory	??? ??? ???
New Museums Site	??? ??? ???
Pembroke Street	??? ??? ???
Cambridge CB2 3QG	??? ??? ???

Contents

1 Introduction

The Alpha Architecture Reference Manual [4] (AARM) defines the behaviour required of all Alpha implementations. This includes both existing processors, as well as future products that might be designed using instruction execution techniques that have not yet been invented.

The Alpha architects claim a "15-25 year design horizon" and they have tried to avoid features that unnecessarily constrain future implementations. One result of this long term view is that the Alpha architecture only partially specifies behaviour. In particular, the order in which reads and writes to shared memory are performed is not fully defined.

Programs intended to run on all implementations of Alpha must not assume that events are ordered in ways that are not specified by the architecture. Particular implementations may define the order more fully than AARM, and it is tempting to write programs that exploit this to gain enhanced performance. However, such programs may fail to work correctly on other implementations.

AARM is an informal specification, but there is a formalization of large parts of it by Tim Leonard. This paper is a tutorial introduction to a simplified subset of Leonard's formalization of the Alpha Shared Memory Model. The intended use of this subset is to support a software verification methodology for establishing properties that hold for programs running on *all* implementations of the Alpha architecture.

A companion paper uses the formal model to give a semantics to a simple instruction set [2]. In that paper, a number of results are presented that show that Leonard's formalization of AARM can be the basis for practical program verification tools. In particular, a bridge is built between the reference ordering models of AARM and standard state machine models such as those used in recent work on verifying machine code [5].

It is intended that what follows be accessible to:

(i) computer architects who are unfamiliar with formal specification methods;

(ii) theoretical computer scientists who are unfamiliar with the shared memory models used in modern RISC architectures.

The structure of this paper is:

Section 2 is a brief introduction to the ideas used in AARM to specify the Alpha shared memory model. These ideas are similar to those found in architectural specifications of other RISC processors, such as SPARC [1].

Section 3 discusses the methodology used in formalizing AARM.

Section 4 summarizes the logical notation used in the formalization.

Section 5 reproduces verbatim the parts of AARM that specify the Alpha shared memory model (including errors), interspersed with commentary (which is included in boxes to separate it from the AARM text). This commentary includes the simplified Leonard formalization.

Section 6 briefly discusses the mechanization of the model in a theorem prover.

2 Instructions and memory references

An Alpha implementation may have one or more processors, each having its own local memory consisting of 64 registers. Processors communicate by reading and writing to a shared memory.

As each processor executes a program, it issues a sequence of memory references (e.g. reads and writes) to the shared memory. These references are not necessarily handled in the order in which they are issued. This will be illustrated using the following simplified instructions:

Instruction	**Meaning**
LD x R	Load value of location x in shared memory into register R
ST x v	Store value v in location x of shared memory
MB	Impose order on memory references (explained later)

The execution of these instructions results in memory references being generated:

Instruction	**Reference generated**
LD x R	Generates a read from location x
ST x v	Generates a write of value v to location x
MB	Generates a memory barrier reference

The terminology used here is that loads and stores are kinds of instructions (a syntactic notion), whereas reads and writes are memory references (an abstract semantic notion). Individual instructions are said to *generate* references, whereas a processor running a program *issues* a sequence of references.

It is assumed that instructions are executed in the order specified by the program text. Thus instruction reordering is not considered here — only memory reference ordering is treated. In practice, programs are executed by fetching instructions from memory, and the order in which individual instructions are executed may be different from the order in which the instruction fetches are issued. Because the reading and writing of instructions may be optimized, there are some special rules in AARM concerning the ordering of references generated by instruction fetches. These are not considered (though they are formalized in Leonard's Alpha specification). However, the techiques presented here and in the companion paper [2] are intended to gracefully extend to handle instruction fetching, but the details have not yet been worked out.

The following two-processor example is based on a suggestion by Jim Saxe:

Program for Processor 1 (P1)	**Program for Processor 2 (P2)**
ST x v	ST flag 0
MB	LD flag R1
ST flag 1	MB
	LD x R2

The program running on P1 stores v in location `x` of the shared memory; it then executes an `MB` and then it stores `1` into location `flag`. The memory barrier instruction `MB` ensures that the write generated by `ST x` v is performed before the write generated by `ST flag 1`. Without it, the references generated by these two instructions could be reordered. In general, a memory barrier forces all memory references issued before it to be before those issued after it.

The program running on P2 first stores `0` into location `flag`, then loads the value of `flag` into its local register `R1`. (Note that the value read is not necessarily the `0` just written, because the write generated by `ST flag 1` on P1 might be executed after the write generated by `ST flag 0` on P2.) Next a memory barrier is executed on P2 and finally the value in location `x` of shared memory is loaded into P2's register `R2`. Because of the preceding `MB`, the read generated by this load is constrained by the architecture to happen after the references generated by the two instructions preceding the `MB`.

If P1 and P2 start up running the two programs in parallel, then the results are not completely predictable. However, the following must hold (assuming no other processors are accessing the shared memory):

> If the value in P2's register `R1` in the final state is `1`, then the value in P2's register `R2` in the final state must be v.

The intuitive argument that this is so is as follows:

I–1. AARM specifies that references to the same location on the same processor cannot be re-ordered, so the write generated by `ST flag 0` must be before the read generated by `LD flag R1`.

I–2. If the value in P2's register `R1` in the final state is `1`, then it must be so because the write generated by `ST flag 1` is after the write generated by `ST flag 0` and before the read generated by `LD flag R1`.

I–3. The `MB` in P1 ensures that the write generated by `ST x` v is before the write generated by `ST flag 1`.

I–4. The memory barrier in P2 ensures that that the read generated by `LD flag R1` is before the one generated by `LD x R2`.

I–5. By transitivity of "before", the write generated by `ST x` v is before the read generated by `LD x R2`.

I–6. Since there are no writes to `x` between the write generated by `ST x` v and the subsequent read generated by `LD x R2`, the value read into P2's register `R2` must be v.

To make this argument rigorous two things are required:

(i) The rules defining the permissible orderings on memory references must be formalized.

(ii) Instructions must be given a semantics that incorporates these rules.

This paper concentrates on (i), but a discussion of (ii) can be found in the companion paper [2].

It is instructive to consider variations of the example just presented.

Suppose the first instruction of P2's program (i.e. `ST flag 0`) is omitted. It may then be possible that the initial value of `flag` in the shared memory is `1`, thus one could not be sure that if the final value of `R1` in P2 is `1` then this is because of the write generated by the instruction `ST flag 1` in P1. Thus one could not be sure that the write from `ST x` v in P1 is performed before the read from `LD x R2` in P2.

Suppose the `MB` in P1's program is omitted. Then the write generated by `ST flag 1` might occur before the write generated by `ST x` v, and thus P2 reading a `1` from `flag` is no guarantee that v has been written into `x` by P1.

Finally, suppose the `MB` in P2's program is omitted. Then the read generated by `LD x R2` may occur before the read generated by `LD flag R1` and so the read from `LD flag R1` occurring after the write from `ST x` v does not ensure that that the read from `LD x R2` occurs after the write from `ST x` v.

In AARM, the rules governing the order in which references happen are illustrated with 'litmus tests'. These are examples of possible and impossible orderings. Here is a litmus test illustrating an impossible ordering. This is derived by imagining an execution of the example program in which P1 writes `2` into `x`, then P2 reads `1` into `R1`, and finally P2 reads `3` into `R2`. By the argument given above this is impossible: P2 must read `2` into `R2`. The litmus test is first stated using AARM notation and then explained.

Litmus Test A (Impossible Sequence)

P1	P2
[U1] P1:W(x,2)	[V1] P2:W(flag,0)
[U2] P1:MB	[V2] P2:R(flag,1)
[U3] P1:W(flag,1)	[V3] P2:MB
	[V4] P2:R(x,3)

The items in the litmus test are memory references:

Reference	**Meaning**
Pi:W(l,a)	Write by processor Pi to location l with value a
Pi:R(l,a)	Read by processor Pi from location l resulting in value a
Pi:MB	Memory barrier on processor Pi

The references in each column of the litmus test are listed in their *issue sequence*, i.e. the sequence in which they are issued by the source processor. The rules given in AARM allow one to deduce the following ordering relations, which are the formal counterparts of I–1, ... , I–5 above.

F–1. V1 is before V2.

F–2. U3 is after V1 and before V2.

F–3. U1 is before U3.

F–4. V2 is before V4.

AARM specifies that the 'before' relation is transitive, hence:

F–4. U1 is before V4.

Finally, AARM specifies that the value read from a location is the last value written to it (this is called 'storage'), hence:

F–5. The value read by V4 (viz. 3) must be equal to the value written by U1 (viz. 2).

Clearly F5 is false, so the litmus test is an impossible sequence.

If V4 is changed to P2:R(x,2) then the litmus test would represent a possible sequence (called an "Okay sequence" in AARM).

3 Methodology for formalization

The final authority on the Alpha architecture is the reference manual AARM. Any formalization must accurately represent it, but since AARM is an informal document there is a fundamental problem in ensuring this. It is impossible to prove rigorously that the formalization accurately represents the informal text, since such texts do not support formal proof. A further problem is that AARM contains a number of minor typographical errors (e.g. see the comment following **Table 5–2** on page 10). Clearly one does not want to incorporate these in the formalization.

The approach taken by Leonard is to try to make the formalization self-evidently consistent with AARM by expressing it directly in terms of the notions used in AARM. The result of this should be intelligible to the authorities defining the Alpha architecture, and hence (one hopes) they will be able to easily certify the formalization as an accurate representation of their design intentions.

Because Leonard's formalization has been written in terms of AARM notions, it does not use standard methods developed by theoretical computer scientists (process algebra, temporal logic etc.). This means that standard verification technologies are not directly applicable to it. The goal of this paper (and its companion [2]) is to demonstrate that Leonard's formalization is nevertheless suitable as a basis for formal verification. This is done in two ways: (i) by showing that one can reason directly about the specification, and (ii) by proving theorems that relate Leonard's model to conventional models.

Only a simplified subset of Leonard's full Alpha specification is given here. This will be presented by annotating sections of AARM. These annotations describe interpretations and corrections to the AARM text as well as giving the formal definitions. The notation used for the formalization is a simple combination of predicate calculus with set theory. The next section summarizes this. The section after that is the annotated definition from AARM of the Alpha shared memory model.

4 Summary of set theory and logic

The application of a function f to an argument x will be written either as $f(x)$, or without brackets as $f\ x$. If f is 'curried' then its application to n arguments $x_1, \ldots, x_n$ is written $f\ x_1\ \ldots\ x_n$. An application of the form $f(x_1, \ldots, x_n)$ is regarded as the application of f to a single tuple argument $(x_1, \ldots, x_n)$.

Predicates are regarded as functions that return the truth-values T and F. For example, the set membership predicate $\in$ is defined by:

$$\begin{array}{lcl} u \in S & = & \mathsf{T} \text{ if } u \text{ is a member of } S \\ u \in S & = & \mathsf{F} \text{ if } u \text{ is not a member of } S \end{array}$$

$\in$ is regarded as a function taking two arguments and returning a truth value (following convention, it is written infixed).

Functions and predicates can be applied to other functions and predicates and return them as results (i.e. higher order predicate calculus is used). For example, the function $\mathsf{Closure}$ (defined on page 12) takes two relations as arguments and returns the transitive closure of their union (disjunction).

The symbols used for the logical operators are: $\neg$ (negation – "not), $\wedge$ (conjunction – "and"), $\vee$ (disjunction – "or"), $\Rightarrow$ (implication – "implies"), $\forall$ (universal quantification – "for all") and $\exists$ (existential quantificatioEn – "there exists"). Quantifications may be restricted to a set S using the notations $\forall x \in S$ and $\exists x \in S$.

$\vdash \mathcal{F}$ means that $\mathcal{F}$ follows from the definitions that precede it.

5 Formalization of the Alpha Shared Memory Model

> This section reproduces verbatim (with minor typographical changes) part of Section 5.6 of AARM. The interpretation and formalization is included in the 'shadow boxes' that are interspersed with the AARM text.

5.6 Read/Write Ordering

This section does not apply to programs that run on a single processor and do not write to the instruction stream. On a single processor, all memory accesses appear to happen in the order specified by the programmer. This section deals entirely with predictable read/write ordering across multiple processors.

The order of reads and writes done in an Alpha implementation may differ from that specified by the programmer.

For any two memory references A and B, either A must occur before B in all Alpha implementations, B must occur before A, or they are UNORDERED. In the last case, software cannot depend upon one occurring first: the order may vary from implementation to implementation, and even from run to run or moment to moment on a single implementation.

If two references cannot be shown to be ordered by the rules given, they are UNORDERED and implementations are free to do them in any order that is convenient. Implementations may take advantage of this freedom to deliver substantially higher performance.

The discussion that follows first defines the architectural issue sequence of memory references on a single processor, then defines the (partial) ordering on this issue sequence that *all* Alpha implementations are required to maintain.

The individual issue sequences on multiple processors are merged into access sequences at each shared memory location. The discussion defines the (partial) ordering on the individual access sequences that *all* Alpha implementations are required to maintain.

The net result is that for any code that executes on multiple processors, one can determine which memory accesses are required to occur before others on *all* Alpha implementations and hence can write useful shared-variable software.

Software writers can force one reference to occur before another by inserting a memory barrier instruction (MB or IMB) between the references.

5.6.1 Alpha Shared Memory Model

An Alpha system consists of a collection of *processors* and shared coherent *memories* that are accessible by all processors. (There may also be unshared memories, but they are outside the scope of this section.)

A *processor* is an Alpha CPU or an I/O device (or anthing else that gets added).

A *shared memory* is the primary storage place for one or more locations.

A *location* is an aligned quadword, specified by its physical address. Multiple virtual addresses may map to the same physical address. Ordering considerations are based only on the physical address.

Locations are formalized as strings, representing their symbolic names.

The contents of locations are represented by natural numbers.

IMPLEMENTATION NOTE

An implementation may allow a location to have multiple physical addresses, but the rules for accesses via mixtures of addresses are implementation-specific and outside the scope of this section. Accesses via exactly one of the physical addresses follow the rules described next.

Each processor may generate *accesses* to shared memory locations. There are five types of accesses:

1. Instruction fetch by processor i to location x, returning value a, denoted Pi:I(x,a).
2. Data read by processor i to location x, returning value a, denoted Pi:R(x,a).
3. Data write by processor i to location x, storing value a, denoted Pi:W(x,a).
4. Memory barrier instruction issued by processor i, denoted Pi:MB.
5. I-stream memory barrier instruction issued by processor i, denoted Pi:IMB.

The first access type is also called an I-stream access or I-fetch. The next two are also called D-stream accesses. The first three types collectively are called read/write accesses, denoted Pi:*(x,a). The last two types collectively are called barriers.

During actual execution in an Alpha system, each processor has a time-ordered *issue sequence* of all the memory references presented by that processor (to all memory locations), and each memory location has a time-ordered *access sequence* of all the accesses presented to that location (from all processors).

Instruction fetches and I-stream memory barriers are not considered here (see Section 2).

The three kinds of memory reference that are considered are:

(i) Reads: $\mathsf{R}\ p\ n\ x\ a$;

(ii) Writes: $\mathsf{W}\ p\ n\ x\ a$;

(iii) Memory barriers: $\mathsf{MB}\ p\ n$

where R, W and MB are functions that construct references. They take as arguments a natural number p representing a processor, and a natural number n specifying the order in which the reference is issued. Reads and writes also take a location x in shared memory (a string) and a value read or written a (a natural number).

The components of references can be extracted with the functions Proc, Seq, Var and Val, which are defined by:

$$
\begin{array}{llllll}
\mathsf{Proc}(\mathsf{R}\ p\ n\ x\ a) = p, & \mathsf{Proc}(\mathsf{W}\ p\ n\ x\ a) = p, & \mathsf{Proc}(\mathsf{MB}\ p\ n) = p, \\
\mathsf{Seq}(\mathsf{R}\ p\ n\ x\ a) = n, & \mathsf{Seq}(\mathsf{W}\ p\ n\ x\ a) = n, & \mathsf{Seq}(\mathsf{MB}\ p\ n) = n, \\
\mathsf{Var}(\mathsf{R}\ p\ n\ x\ a) = x, & \mathsf{Var}(\mathsf{W}\ p\ n\ x\ a) = x, & \\
\mathsf{Val}(\mathsf{R}\ p\ n\ x\ a) = a, & \mathsf{Val}(\mathsf{W}\ p\ n\ x\ a) = a. &
\end{array}
$$

The predicates IsR, IsW and IsMB are defined by:

$$
\begin{array}{lll}
\mathsf{IsR}(\mathsf{R}\ p\ n\ x\ a) = \mathsf{T}, & \mathsf{IsR}(\mathsf{W}\ p\ n\ x\ a) = \mathsf{F}, & \mathsf{IsR}(\mathsf{MB}\ p\ n) = \mathsf{F}, \\
\mathsf{IsW}(\mathsf{R}\ p\ n\ x\ a) = \mathsf{F}, & \mathsf{IsW}(\mathsf{W}\ p\ n\ x\ a) = \mathsf{T}, & \mathsf{IsW}(\mathsf{MB}\ p\ n) = \mathsf{F}, \\
\mathsf{IsMB}(\mathsf{R}\ p\ n\ x\ a) = \mathsf{F}, & \mathsf{IsMB}(\mathsf{W}\ p\ n\ x\ a) = \mathsf{F}, & \mathsf{IsMB}(\mathsf{MB}\ p\ n) = \mathsf{T}.
\end{array}
$$

5.6.1.1 Architectural Definition of Processor Issue Sequence

The issue sequence for a processor is architecturally defined with respect to a hypothetical simple implementation that contains one processor and a single shared memory, with no caches or buffers. This is the instruction execution model:

1. I-fetch: An Alpha instruction is fetched from memory.
2. Read/Write: That instruction is executed and runs to completion, including a single data read from memory for a Load instruction or a single data write to memory for a Store instruction.
3. Update: The PC for the processor is updated.
4. LOOP: Repeat the above sequence indefinitely.

If the instruction fetch step gets a memory management fault, the I-fetch is not done and the PC is updated to a point to a PALcode fault handler. If the read/write step gets a memory management fault, the read/write is not done and the PC is updated to point to a PALcode fault handler.

All memory references are aligned quadwords. For the purpose of defining ordering, aligned longword references are modeled as quadword references to the containing aligned quadword.

Reference u precedes v in the issue sequence if they are issued by the same processor and the sequence number of u is less than the sequence number of v.

$$\mathsf{IssueSeq}\ u\ v \;=\; (\mathsf{Proc}\ u \;=\; \mathsf{Proc}\ v) \wedge (\mathsf{Seq}\ u < \mathsf{Seq}\ v)$$

5.6.1.2 Definition of Processor Issue Order

A partial ordering, called processor issue order, is imposed on the issue sequence defined in Section 5.6.1.1.

For two accesses u and v issued by processor Pi, u is said to PRECEDE v IN ISSUE ORDER ($<$) if u occurrs earlier than v in the issue sequence for Pi, and either of the following applies:

1. The access types are of the following issue order:

Table 5–1: Processor Issue Order

1st↓/2nd→	Pi:I(y,b)	Pi:R(y,b)	Pi:W(y,b)	Pi:MB	Pi:IMB
Pi:I(x,a)	< if x=y		< if x=y	<	<
Pi:R(x,a)		< if x=y	< if x=y	<	<
Pi:W(x,a)		< if x=y	< if x=y	<	<
Pi:MB		<	<	<	<
Pi:IMB	<	<	<	<	<

2. Or, u is a TB fill, for example, a PTE read in order to satisfy a TB miss, and v is an I- or D-stream access using that PTE (see Section 5.6.2).

Since I-fetches and IMB are not being considered here, **Table 5–1** can be formalised by defining:

$$\mathsf{Table\_5\_1}\ u\ v\ =\ \mathsf{IsMB}\ u \lor \mathsf{IsMB}\ v \lor (\mathsf{Var}\ u = \mathsf{Var}\ v)$$

Issue order is thus a partial order imposed on the architecturally specified issue sequence. Implementations are free to do memory accesses from a single processor in any sequence that is consistent with this partial order.

Note that accesses to different locations are ordered only with respect to barriers and TB fill. The table asymmetry for I-fetch allows writes to the I-stream to be incoherent until an IMB is executed.

Issue order is formalized with a predicate IssueOrder defined by:

$$\mathsf{IssueOrder}\ u\ v\ =\ \mathsf{IssueSeq}\ u\ v \land \mathsf{Table\_5\_1}\ u\ v$$

IssueOrder u v means that u precedes v in issue order ($u < v$ in AARM notation).

The restriction of IssueOrder to a set *refs* will be needed.

$$\mathsf{Restrict}\ \mathit{refs}\ \mathit{ord}\ u\ v\ =\ u \in \mathit{refs} \land v \in \mathit{refs} \land \mathit{ord}\ u\ v$$

The restricted order is thus Restrict IssueOrder *refs*.

5.6.1.3 Definition of Memory Access sequence

The access sequence for a location cannot be observed directly, not fully predicted before an actual execution, nor reproduced exactly from one execution to another. Nonetheless, some useful ordering properties must hold in all Alpha implementations.

Location access order is only partially specified. It is required to satisfy the properties described in **Section 5.6.1.4**, **Section 5.6.1.5** and **Section 5.6.1.6**. This is formalized by defining predicates on orderings as follows:

Section_5_6_1_4 *refs ord* iff *ord* is an ordering of *refs* satisfying **Section 5.6.1.4**
Section_5_6_1_5 *refs ord* iff *ord* is an ordering of *refs* satisfying **Section 5.6.1.5**
Section_5_6_1_6 *refs ord* iff *ord* is an ordering of *refs* satisfying **Section 5.6.1.6**

5.6.1.4 Definition of Location Access Order

A partial ordering, called location access order, is imposed on the memory access sequence defined above.

For two accesses *u* and *v* to location *x*, *u* is said to PRECEDE *v* IN ACCESS ORDER ($\ll$) if *u* occurs earlier than *v* in the access sequence for *x*, and at least one of them is a write:

Table 5–2: Location Access Order

1st↓/2nd→	Pi:I(x,b)	Pi:R(x,b)	Pi:W(x,b)
Pi:I(x,a)			$\ll$
Pi:R(x,a)			$\ll$
Pi:W(x,a)	$\ll$	$\ll$	$\ll$

Table 5–2 is probably wrong: either the column headings or the row headings should be Pi rather than Pj.

Access order is thus a partial order imposed on the actual access sequence for a given location. Each location has a separate access order. There is no direct ordering relationship between accesses to different locations.

Note that reads and I-fetches are ordered only with respect to writes.

Leonard interprets **Section 5.6.1.4** as just saying that any two accesses to the same location, one of which is a write, must be ordered.

$$\mathsf{AtLeastOneWrite}\ u\ v \ =$$
$$\neg(u = v) \wedge (\mathsf{Var}\ u = \mathsf{Var}\ v) \wedge ((\mathsf{IsR}\ u \wedge \mathsf{IsW}\ v) \vee (\mathsf{IsW}\ u \wedge \mathsf{IsR}\ v) \vee (\mathsf{IsW}\ u \wedge \mathsf{IsW}\ v))$$

$$\mathsf{Section\_5\_6\_1\_4}\ \mathit{refs}\ \mathit{ord} \ =$$
$$\forall u\ v \in \mathit{refs}.\ \mathsf{AtLeastOneWrite}\ u\ v \Rightarrow (\mathit{ord}\ u\ v \vee \mathit{ord}\ v\ u)$$

5.6.1.5 Definition of Storage

If *u* is Pi:W(x,a), and *v* is either Pj:I(x,b) or Pj:R(x,b), and $u \ll v$, and no *w* Pk:W(x,c) exists such that $u \ll w \ll v$, then the value *b* returned by *v* is exactly the value *a* written by *u*.

Conversely, if *u* is Pi:W(x,a), and *v* is either Pj:I(x,b) or Pj:R(x,b), and b=a (and a is distinguishable from values written by accesses other than *u*), then u$\ll$v and for any other *w* Pk:W(x,c) either w≪u or v≪w.

The only way to communicate information between different processors is for one to write a shared location and the other to read the shared location and receive the newly written value. (In this context, the sending of an interrupt from processor Pi to processor Pj is modelled as Pi writing to a location INTij, and Pj reading from INTij.)

Leonard interprets this section as specifying two things.

(i) If a read u to a location x follows a write v to x, and there is no other intervening write to x with a different value, then the value read in v must be the value written in u.

(ii) If a read u to a location x follows a write v to x and the value read in v is different from the value written in u, then there must exist an intervening write w to x with the value read by v.

The first of these, (i), is formalized with the following predicate:

$$\begin{array}{l} \mathsf{ReadGetsLastWrite}\ \mathit{refs}\ \mathit{ord} \quad = \\ \forall u\ v \in \mathit{refs}. \\ \quad \mathsf{IsW}\ u \wedge \mathsf{IsR}\ v \wedge (\mathsf{Var}\ u = \mathsf{Var}\ v) \wedge \mathit{ord}\ u\ v\ \wedge \\ \quad \neg(\exists w \in \mathit{refs}.\ \mathsf{IsW}\ w \wedge (\mathsf{Var}\ w = \mathsf{Var}\ u) \wedge \neg(\mathsf{Val}\ u = \mathsf{Val}\ w) \wedge \mathit{ord}\ u\ w \wedge \mathit{ord}\ w\ v) \\ \quad \Rightarrow \\ \quad (\mathsf{Val}\ u = \mathsf{Val}\ v) \end{array}$$

The second, (ii), is formalized by:

$$\begin{array}{l} \mathsf{ReadRequiresWrite}\ \mathit{refs}\ \mathit{ord} \quad = \\ \forall u\ v \in \mathit{refs}. \\ \quad \mathsf{IsW}\ u \wedge \mathsf{IsR}\ v \wedge (\mathsf{Var}\ u = \mathsf{Var}\ v) \wedge \mathit{ord}\ u\ v \wedge \neg(\mathsf{Val}\ u = \mathsf{Val}\ v) \\ \quad \Rightarrow \\ \quad \exists w \in \mathit{refs}.\ \mathsf{IsW}\ w \wedge (\mathsf{Var}\ w = \mathsf{Var}\ u) \wedge (\mathsf{Val}\ v = \mathsf{Val}\ w) \wedge \mathit{ord}\ u\ w \wedge \mathit{ord}\ w\ v \end{array}$$

Leonard's interpretation of **Section 5.6.1.5** is thus:

$$\begin{array}{l} \mathsf{Section\_5\_6\_1\_5}\ \mathit{refs}\ \mathit{ord} \quad = \\ \ \mathsf{ReadGetsLastWrite}\ \mathit{refs}\ \mathit{ord} \wedge \mathsf{ReadRequiresWrite}\ \mathit{refs}\ \mathit{ord} \end{array}$$

5.6.1.6 Relationship Between Issue Order and Access order

If u is Pi:*(x,a), and v is Pi:*(x,b), one of which is a write, and u<v in the issue order for processor Pi, then u≪v in the access order for location x.

In other words, if two accesses to the same location are ordered on a given processor, they are ordered in the same way at the location.

Leonard's interpretation of **Section 5.6.1.6** is:

$$\mathsf{Section\_5\_6\_1\_6}\ \mathit{refs}\ \mathit{ord} \ = \ \forall u\ v \in \mathit{refs}.\ \mathsf{AtLeastOneWrite}\ u\ v \wedge \mathsf{IssueOrder}\ u\ v \Rightarrow \mathit{ord}\ u\ v$$

The conditions for an order *ord* to be an access order on a set of references *refs* are:

$$\mathsf{IsAccessOrder}\ \mathit{refs}\ \mathit{ord} \ = \ \mathsf{Section\_5\_6\_1\_4}\ \mathit{refs}\ \mathit{ord} \wedge \mathsf{Section\_5\_6\_1\_5}\ \mathit{refs}\ \mathit{ord} \wedge \mathsf{Section\_5\_6\_1\_6}\ \mathit{refs}\ \mathit{ord}$$

5.6.1.7 Definition of Before

For two accesses *u* and *v*, *u* is said to be BEFORE v ($\Leftarrow$) if:

u $<$ v or
u $\ll$ v or
there exists an access w such that:
(u $<$ w and w $\Leftarrow$ v) or
(u $\ll$ w and w $\Leftarrow$ v).

In other words, "before" is the transitive closure over issue order and access order.

If R_1 and R_2 are binary relations, then "the transitive closure over R_1 and R_2" is taken to mean the least transitive relation including R_1 and R_2.

This is the relation $\mathsf{Closure}\ R_1\ R_2$ characterized by the four properties:

$$\forall u\ v.\ R_1\ u\ v \Rightarrow \mathsf{Closure}\ R_1\ R_2\ u\ v$$

$$\forall u\ v.\ R_2\ u\ v \Rightarrow \mathsf{Closure}\ R_1\ R_2\ u\ v$$

$$\forall u\ v.\ (\exists w.\ \mathsf{Closure}\ R_1\ R_2\ u\ w \wedge \mathsf{Closure}\ R_1\ R_2\ w\ v) \Rightarrow \mathsf{Closure}\ R_1\ R_2\ u\ v$$

$$\begin{aligned}\forall P.\ &(\forall u\ v.\ R_1\ u\ v \Rightarrow P\ u\ v) \wedge \\ &(\forall u\ v.\ R_2\ u\ v \Rightarrow P\ u\ v) \wedge \\ &(\forall u\ v.\ (\exists w.\ P\ u\ w \wedge P\ w\ v) \Rightarrow P\ u\ v) \\ &\Rightarrow \\ &(\forall u\ v.\ \mathsf{Closure}\ R_1\ R_2\ u\ v \Rightarrow P\ u\ v)\end{aligned}$$

5.6.1.8 Definition of After

If u $\Leftarrow$ v, then *v* is said to be AFTER *u*.

At most one of u $\Leftarrow$ v and v $\Leftarrow$ u is true.

Since the access order is only partially specified, the "before" relation can only be partially specified too.

A relation is defined to be a 'before relation' on *refs* if it is the closure over the issue order on *refs* and *some* access order on *refs*, and in addition there are no 'loops'.

$$\begin{aligned}
&\mathsf{IsBefore}\ \mathit{refs}\ \mathit{ord} \quad = \\
&\exists \mathit{access_order}.\ \mathsf{IsAccessOrder}\ \mathit{refs}\ \mathit{access_order}\ \wedge \\
&\qquad\qquad (\mathit{ord} = \mathsf{Closure}\ (\mathsf{Restrict}\ \mathit{IssueOrder}\ \mathit{refs})\ \mathit{access_order})\ \wedge \\
&\qquad\qquad \neg(\exists u\ v \in \mathit{refs}.\ \mathit{ord}\ u\ v \wedge \mathit{ord}\ v\ u)
\end{aligned}$$

5.6.1.9 Timeliness

Even in the absence of a barrier after the write, a write by one processor to a given location may not be delayed indefinitely in the access order for that location.

5.6.2 LitmusTests

Many issues about writing and reading shared data can be cast into questions about whether a write is before or after a read. These questions can be answered by rigorously applying the ordering rules described previously to demonstrate whether the accesses in question are ordered at all.

A *memory state* is a function from locations to values.

A set *refs* of references is *initialized* with respect to a memory state *init* and an ordering *ord* iff every read in *refs* can be 'explained' by *init* or by a preceding write in *refs*.

More precisely: for every read u in *refs*, if u reads from location x, then the value a that is read is either the value of x in *init* or the value written by some write to x in *refs* that precedes the read in the ordering *ord*.

$$\begin{aligned}
&\mathsf{Init}\ \mathit{init}\ \mathit{refs}\ \mathit{ord} \quad = \\
&\forall u \in \mathit{refs}.\ \mathsf{IsR}\ u \Rightarrow (\mathsf{Val}\ u = \mathit{init}(\mathsf{Var}\ u)) \\
&\qquad\qquad\qquad\quad \vee \\
&\qquad\qquad\qquad\quad \exists v \in \mathit{refs}.\ \mathsf{IsW}\ v \wedge (\mathsf{Var}\ v = \mathsf{Var}\ u) \wedge (\mathsf{Val}\ v = \mathsf{Val}\ u) \wedge \mathit{ord}\ v\ u
\end{aligned}$$

Assume, in the litmus tests below, that initially all memory locations contain 1.

This assumption is formalized by defining $\mathsf{InitialState}$ by

$$\forall x.\ \mathsf{InitialState}\ x \quad = \quad 1$$

and then assuming that $\mathsf{Init}\ \mathsf{InitialState}\ \mathit{refs}\ \mathit{ord}$.

A litmus test is represented by a set of references. The predicates OkaySequence and ImpossibleSequence define whether such a set represents a possible sequence or an impossible one, respectively.

$$\begin{array}{lcl} \mathsf{OkaySequence}\ refs & = & \exists ord.\ \mathsf{Init}\ \mathsf{InitialState}\ refs\ ord \wedge \mathsf{IsBefore}\ refs\ ord \\ \mathsf{ImpossibleSequence}\ refs & = & \neg(\mathsf{OkaySequence}\ refs) \end{array}$$

In the formalized litmus tests that follow, $\mathbf{i} = 0$, $\mathbf{j} = 1$ and $\mathbf{k} = 2$.

5.6.2.1 Litmus Test 1 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(x,2)
	[V2] Pj:R(x,1)

V1 reading 2 implies U1 $\ll$ V1, by the definition of storage
V2 reading 1 implies V2 $\ll$ U1, by the definition of storage
V1 $<$ V2, by the definition of issue order

The first two orderings imply that V2 $\Leftarrow$ V1, whereas the last implies that V1 $\Leftarrow$ V2.

Both implications cannot be true. Thus, once a processor reads a new value from a location, it must never see an old value—time must not go backward. V2 must read 2.

$$\vdash \mathsf{ImpossibleSequence}\ \ \{\mathsf{W}\ 0\ 0\ \mathrm{x}\ 2,\ \ \begin{array}{l}\mathsf{R}\ 1\ 0\ \mathrm{x}\ 2,\\ \mathsf{R}\ 1\ 1\ \mathrm{x}\ 1\}\end{array}$$

5.6.2.2 Litmus Test 2 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:W(x,3)
	[V2] Pj:R(x,2)
	[V2] Pj:R(x,3)

V2 reading 2 implies V1 $\Leftarrow$ U1
V3 reading 3 implies U1 $\Leftarrow$ V1

Both implications cannot be true. Thus, once a processor reads a new value written by U1, any other writes that must precede the read must also precede U1. V3 must read 2.

$$\vdash \mathsf{ImpossibleSequence}\ \ \{\mathsf{W}\ 0\ 0\ \mathrm{x}\ 2,\ \ \begin{array}{l}\mathsf{W}\ 1\ 0\ \mathrm{x}\ 3,\\ \mathsf{R}\ 1\ 1\ \mathrm{x}\ 2,\\ \mathsf{R}\ 1\ 2\ \mathrm{x}\ 3\}\end{array}$$

5.6.2.3 Litmus Test 3 (Impossible Sequence)

Pi	Pj	Pk
[U1] Pi:W(x,2)	[V1] Pj:W(x,3)	[W1] Pk:R(x,3)
[U2] Pi:R(x,3)		[W1] Pk:R(x,2)

U2 reading 3 implies U1 $\Leftarrow$ V1
W2 reading 2 implies V1 $\Leftarrow$ U1

Both implications cannot be true. Again, time cannot go backward. If U2 reads 3 then W2 must read 3. Alternatively, if W2 reads 2, then U2 must read 2.

$$\vdash \mathsf{ImpossibleSequence}\ \{\mathrm{W}\ 0\ 0\ x\ 2,\quad \mathrm{W}\ 1\ 0\ x\ 3,\quad \mathrm{R}\ 2\ 0\ x\ 3,\quad \mathrm{R}\ 0\ 1\ x\ 3,\quad \mathrm{R}\ 2\ 1\ x\ 2\}$$

5.6.2.4 Litmus Test 4 (Sequence Okay)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:W(y,2)	[V2] Pj:R(x,1)

There are no conflicts in this sequence. U2 $\Leftarrow$ V1 and V2 $\Leftarrow$ U1. U1 and U2 are not ordered with respect to each other. V1 and V2 are not ordered with respect to each other. There is no conflicting implication that U1 $\Leftarrow$ V2.

$$\vdash \mathsf{OkaySequence}\ \{\mathrm{W}\ 0\ 0\ x\ 2,\quad \mathrm{R}\ 1\ 0\ y\ 2,\quad \mathrm{W}\ 0\ 1\ y\ 2,\quad \mathrm{R}\ 1\ 1\ x\ 1\}$$

5.6.2.5 Litmus Test 5 (Sequence Okay)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
	[V2] Pj:MB
[U2] Pi:W(y,2)	[V3] Pj:R(x,1)

There are no conflicts in this sequence. U2 $\Leftarrow$ V1 $\Leftarrow$ V3 $\Leftarrow$ U1. There is no conflicting implication that U1 $\Leftarrow$ U2.

$$\vdash \mathsf{OkaySequence}\ \{\mathrm{W}\ 0\ 0\ x\ 2,\quad \mathrm{R}\ 1\ 0\ y\ 2,\quad \mathsf{MB}\ 1\ 1,\quad \mathrm{W}\ 0\ 1\ y\ 2,\quad \mathrm{R}\ 1\ 2\ x\ 1\}$$

5.6.2.6 Litmus Test 6 (Sequence Okay)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:MB	
[U3] Pi:W(y,2)	[V3] Pj:R(x,1)

There are no conflicts in this sequence. V2 $\Leftarrow$ U1 $\Leftarrow$ U3 $\Leftarrow$ V1. There is no conflicting implication that V1 $\Leftarrow$ V2.

In scenarios 4, 5, and 6, writes to two different locations x and y are observed (by another processor) to occur in the opposite order than that in which they were performed. An update to y propagates quickly to Pj, but the update to x is delayed, and Pi and Pj do not both have MBs.

$$\vdash \mathsf{OkaySequence}\ \begin{array}{l}\{\mathsf{W}\ \ 0\ 0\ x\ 2,\quad \mathsf{R}\ 1\ 0\ y\ 2,\\ \ \mathsf{MB}\ 0\ 1,\\ \ \mathsf{W}\ \ 0\ 2\ y\ 2,\quad \mathsf{R}\ 1\ 1\ x\ 1\}\end{array}$$

5.6.2.7 Litmus Test 7 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:R(y,2)
[U2] Pi:MB	[V2] Pj:MB
[U3] Pi:W(y,2)	[V3] Pj:R(x,1)

V1 reading 2 implies U3 $\Leftarrow$ V1
V3 reading 1 implies V3 $\Leftarrow$ U1
But, by transitivity, U1 $\Leftarrow$ U3 $\Leftarrow$ V1 $\Leftarrow$ V3

Both cannot be true, so if V1 reads 2, then V3 must also read 2.

$$\vdash \mathsf{ImpossibleSequence}\ \begin{array}{l}\{\mathsf{W}\ \ 0\ 0\ x\ 2,\quad \mathsf{R}\ \ \ 1\ 0\ y\ 2,\\ \ \mathsf{MB}\ 0\ 1,\qquad \mathsf{MB}\ 1\ 1,\\ \ \mathsf{W}\ \ 0\ 2\ y\ 2,\quad \mathsf{R}\ \ \ 1\ 2\ x\ 1\}\end{array}$$

5.6.2.8 Litmus Test 8 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:W(y,2)
[U2] Pi:MB	[V2] Pj:MB
[U3] Pi:R(y,1)	[V3] Pj:R(x,1)

U3 reading 1 implies U3 $\Leftarrow$ V1
V3 reading 1 implies V3 $\Leftarrow$ U1
But, by transitivity, U1 $\Leftarrow$ U3 $\Leftarrow$ V1 $\Leftarrow$ V3

Both cannot be true, so if U3 reads 1, then V3 must read 2, and vice versa.

$$\vdash \mathsf{ImpossibleSequence}\ \ \begin{array}{ll}\{\mathsf{W}\ \ 0\ 0\ \mathrm{x}\ 2, & \mathsf{W}\ \ 1\ 0\ \mathrm{y}\ 2, \\ \ \mathsf{MB}\ 0\ 1, & \mathsf{MB}\ 1\ 1, \\ \ \mathsf{R}\ \ 0\ 2\ \mathrm{y}\ 1, & \mathsf{R}\ \ 1\ 2\ \mathrm{x}\ 1\}\end{array}$$

5.6.2.9 Litmus Test 9 (Impossible Sequence)

Pi	Pj
[U1] Pi:W(x,2)	[V1] Pj:W(x,3)
[U2] Pi:R(x,2)	[V2] Pj:R(x,3)
[U3] Pi:R(x,3)	[V3] Pj:R(x,2)

V3 reading 2 implies U1 $\Leftarrow$ V3
V2 $\Leftarrow$ V3 and V2 reading 3 implies V2 $\Leftarrow$ U1
V1 $\Leftarrow$ V2 and V2 $\Leftarrow$ U1 implies V1 $\Leftarrow$ U1

U3 reading 3 implies V1 $\Leftarrow$ U3
U2 $\Leftarrow$ U3 and U2 reading 2 implies U2 $\Leftarrow$ V1
U1 $\Leftarrow$ U2 and U2 $\Leftarrow$ V1 implies U1 $\Leftarrow$ V1

Both V1 $\Leftarrow$ U1 and U1 $\Leftarrow$ V1 cannot be true. Time cannot go backwards. If V3 reads 2, then U3 must read 2. Alternatively, If U3 reads 3, then V3 must read 3.

$$\vdash \mathsf{ImpossibleSequence}\ \ \begin{array}{ll}\{\mathsf{W}\ 0\ 0\ \mathrm{x}\ 2, & \mathsf{W}\ 1\ 0\ \mathrm{x}\ 3, \\ \ \mathsf{R}\ \ 0\ 1\ \mathrm{x}\ 2, & \mathsf{R}\ \ 1\ 1\ \mathrm{x}\ 3, \\ \ \mathsf{R}\ \ 0\ 2\ \mathrm{x}\ 3, & \mathsf{R}\ \ 1\ 2\ \mathrm{x}\ 2\}\end{array}$$

6 Mechanization

The formalized Alpha shared memory model is suitable for use with mechanical theorem provers that support higher order logic. Leonard's full Alpha specification, as well as the simplified subset presented here, have been input to the HOL system [3]. All the litmus tests have been verified with HOL using a semi-automatic 'litmus tester' that has been implemented (this works by brute force enumeration of all orderings and is very inefficient – verifying all the litmus tests takes several hours).

A considerable number of properties of the model have been proved as part of the work on relating state based uniprocessor models to reference based multiprocessor models [2]. For example, HOL has been used to show that:

$$\begin{array}{lll}\vdash \forall \mathit{refs}\ \mathit{ord}.\ \mathsf{Finite}\ \mathit{refs} \Rightarrow (\mathsf{IsBefore}\ \mathit{refs}\ \mathit{ord} & = & \\ & & \mathsf{IsAccessOrder}\ \mathit{refs}\ \mathit{ord} \wedge \\ & & \mathsf{Trans}\ (\mathsf{Restrict}\ \mathit{ord}\ \mathit{refs}) \wedge \\ & & \neg(\exists u\ v \in \mathit{refs}.\ \mathit{ord}\ u\ v \wedge \mathit{ord}\ v\ u))\end{array}$$

where $\mathsf{Finite}\ S$ means S is finite and $\mathsf{Trans}\ \mathit{ord}$ means ord is transitive. Because it does not use $\mathsf{Closure}$, this characterization of $\mathsf{IsBefore}$ turns out be be easier to work with than the definition given on page 13.

7 Acknowledgements

The work reported here started during a four week visit I made to DEC's Systems Research Center in March/April, 1993. The main ideas were worked out then through a combination of conversations with Martin Abadi, Jim Horning, Jim Saxe and Yuan Yu and crucial `email` correspondence with Tim Leonard.

I am particularly grateful to Yuan Yu for many hours of help and discussions, to Martin Abadi for hosting my visit to SRC and to Digital for supporting it.

References

[1] *The SPARC Architecture Manual.* Prentice Hall, 1992.

[2] M. J. C. Gordon and ??? Memory access semantics of a multiprocessor instruction set. 1993.

[3] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[4] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992.

[5] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor.* PhD thesis, The University of Texas at Austin, yuanyu@com.dec.src, 1992.