ARM Memory Rules

February 2021

1 Syntax for programs (simpler version)

We use the following syntax for the program.

$$
\begin{aligned}
\mathit{Prog} ::= \ & \texttt{vars:} \ \ x^* \\
& \texttt{procs:} \ \ p^*
\end{aligned}
$$

$$
\begin{aligned}
p ::= \ & \texttt{regs:} \ \ \$r^* \\
& \texttt{instrs:} \ \ i^*
\end{aligned}
$$

$$
i ::= l : \texttt{s}
$$

$$
\begin{aligned}
\texttt{s} ::= \ & x \leftarrow \mathit{exp} \\
& |\$r \leftarrow x \\
& |\$r \leftarrow \mathit{exp} \\
& |\texttt{if } \mathit{exp} \texttt{ then } i^* \texttt{ else } i^* \\
& |\texttt{while } \mathit{exp} \texttt{ do } i^* \\
& |\texttt{assume } \mathit{exp} \\
& |\texttt{assert } \mathit{exp} \\
& |\texttt{term}
\end{aligned}
$$

A program *Prog* first declares a set $\mathcal{X}$ of (shared) variables followed by the code of a set $\mathcal{P}$ of processes. Each process in turn declares a set of registers $\$r$ and it's code in the form of a sequence of instructions denoted by i^*. Each such instruction consists of a label l and a statement s. A *write* instruction has a statement of the form $x \leftarrow \mathit{exp}$ where $x \in \mathcal{X}$ is a variable and *exp* is an *expression*. In all the above rules, *exp* is an expression containing only constants and registers, and no shared variables. A *read* instruction in a process $p \in \mathcal{P}$ is a statement of the form $\$r \leftarrow x$, where $\$r$ is a register in p and $x \in \mathcal{X}$ is a variable. An *assign* instruction in a process $p \in \mathcal{P}$ has the form $\$r \leftarrow \mathit{exp}$, where $\$r$ is a register and *exp* is an expression. Conditional, iterative, assume and assert instructions, collectively called aci instructions describe the rest of the instructions. The special instruction `term` is a syntactic sugar to mark the end of the code for the process.

2 Definitions for the rules

We first define a few helper functions:

- We refer to the statement corresponding to an instruction `i` by the function `stmt(i)`.
- For a write instruction i whose statement is of the form $x \leftarrow \mathit{exp}$, or a read instruction i whose statement is of the form $\$r \leftarrow x$, we define $\texttt{var}(i) := x$ to be the variable corresponding to the instruction. For all other types of instructions we define $\texttt{var}(i) = \perp$.
- For a instruction i which is of one of the following forms: a write instruction with $\texttt{stmt}(i) = \$r \leftarrow \mathit{exp}$, an assign instruction with $\$r \leftarrow \mathit{exp}$, an aci instruction with either $\texttt{stmt}(i) = \texttt{if } \mathit{exp} \texttt{ then } i^* \texttt{ else } i^*$ or $\texttt{stmt}(i) = \texttt{while } \mathit{exp} \texttt{ do } i^*$, or $\texttt{stmt}(i) = \texttt{assume } \mathit{exp}$, or $\texttt{stmt}(i) = \texttt{assert } \mathit{exp}$, we define the expression occuring in the statement as $\texttt{exp}(i) := \mathit{exp}$. For all other types of instructions we define $\texttt{exp}(i) := \perp$.
- For an instruction instance i belonging to the process p, we define $\texttt{proc}(i) := p$. Further, we denote the set of instructions of process p by $\mathcal{J}_p$. Also, we denote the statement at which process p begins execution (i.e. the first instruction of the process), by i_p^{init}.
- For an instruction instance $i \in \mathcal{J}_p$, we denote by $\texttt{next}(i)$ the set of instructions that could immediately follow i in program order. In particular, for an ACI instruction i, we denote the set of instructions that could follow i upon evaluation of $\texttt{exp}(i)$ to True, as $\texttt{Tnext}(i)$, and the set of those that could follow on its evaluation to False as $\texttt{Fnext}(i)$.
- We define the closest write $CW(\mathbb{c}, \mathbb{e}) := \mathbb{e}'$ where $\mathbb{e}'$ is the unique event such that (here and in the rest of the document, an *event* is a single execution instance of an instruction),
  1. $\mathbb{e}' \in \mathbb{E}_p^W$ ($\mathbb{E}_p^W$ denotes the set of write events as defined below),
  2. $\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}$ (The poloc program order $\prec_{\texttt{poloc}}$ will be defined below),

3. there is no event $\mathbb{e}''$ such that $\mathbb{e}'' \in \mathbb{E}_p^W$ and $\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}'' \prec_{\texttt{poloc}} \mathbb{e}$.

If no such events exists, we write $CW(\mathbb{e}) := \bot$.

- We define by $\mathcal{R}(i)$ the set of registers appearing in $\texttt{exp}(i)$. In the special case that $\texttt{exp}(i) = \bot$, we write $\mathcal{R}(i) := \emptyset$.

We define a *configuration* $\mathbb{c}$ as the tuple $\langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}, \texttt{rf}, \texttt{Prop} \rangle$. Here, $\mathbb{E} \in \mathcal{E}$ is the set of events that have been created up to that point in the program. For any process p, we denote by $\mathbb{E}_p$ the events of process p that have been created so far. By $\mathbb{E}^W$ we denote the subset of $\mathbb{E}$ containing precisely the write events. We similarly define $\mathbb{E}^R$, $\mathbb{E}^{ACI}$ and so on. The program order relation $\prec \subseteq \mathbb{E} \times \mathbb{E}$ is an irreflexive partial order that describes for each process $p \in \mathcal{P}$ the order in which events are fetched from the code of p. Note that $\mathbb{e}_1 \not\prec \mathbb{e}_2$ if $\texttt{proc}(\mathbb{e}_1) \neq \texttt{proc}(\mathbb{e}_2)$, i.e. if they belong to different processes, and $\prec$ is a total order on each of the $\mathbb{E}_p$ individually. The function $\texttt{status} : \mathbb{E} \mapsto \{\texttt{fetch}, \texttt{init}, \texttt{com}\}$ defines the current status of each event, which is one of fetched, initialized, and committed. The function $\texttt{Prop} : \mathcal{X} \mapsto \mathbb{E}^W \cup \mathcal{E}^{\texttt{init}}$ defines for each variable the latest value of that variable that has been propagated to the main memory. The function $\texttt{rf} : \mathbb{E}^R \mapsto \mathbb{E}^W \cup \mathcal{E}^{\texttt{init}}$ defines for each read event $\mathbb{e}$ the event $\texttt{rf}(\mathbb{e})$ from which $\mathbb{e}$ gets its value. We introduce a number of dependency relations to help us frame the rules. The *per-location program order* $\prec_{\texttt{poloc}} \subseteq \mathbb{E} \times \mathbb{E}$ is such that $\mathbb{e}_1 \prec_{\texttt{poloc}} \mathbb{e}_2$ if and only if $\mathbb{e}_1 \prec \mathbb{e}_2$ and $\texttt{var}(\mathbb{e}_1) = \texttt{var}(\mathbb{e}_2)$. Further, we define the data dependency order $\prec_{\texttt{data}}$ such that $\mathbb{e}_1 \prec_{\texttt{data}} \mathbb{e}_2$ if

- $\mathbb{e}_1 \in \mathbb{E}^R \cup \mathbb{E}^A$, i.e. $\mathbb{e}_1$ is a read or assign event.
- $\mathbb{e}_2 \in \mathbb{E}^W \cup \mathbb{E}^A \cup \mathbb{E}^{ACI}$, i.e. $\mathbb{e}_2$ is a write, assign or ACI event.
- $\mathbb{e}_1 \prec \mathbb{e}_2$
- $\texttt{stmt}(\texttt{ins}(\mathbb{e}_1))$ is of the form $\$r \leftarrow x$ or $\$r \leftarrow exp$,
- $\$r \in \mathcal{R}(\texttt{ins}(\mathbb{e}_2))$
- There is no event $\mathbb{e}_3 \in \mathbb{E}^R \cup \mathbb{E}^A$ such that $\mathbb{e}_1 \prec \mathbb{e}_3 \prec \mathbb{e}_2$ is of the form $\$r \leftarrow y$ or $\$r \leftarrow exp'$. That is, we want that $\mathbb{e}_1$ be responsible for "supplying" the data to $\mathbb{e}_2$.

Finally, we define the control dependency $\prec_{\texttt{ctrl}}$ as $\mathbb{e}_1 \prec_{\texttt{ctrl}} \mathbb{e}_2$ if $\mathbb{e}_1 \in \mathbb{E}^{ACI}$ and $\mathbb{e}_1 \prec \mathbb{e}_2$.
We also define the transition relation as a relation $\longrightarrow \subseteq \mathbb{C} \times \mathcal{P} \times \mathbb{C}$. For configurations $\mathbb{c}_1, \mathbb{c}_2 \in \mathbb{C}$ and a process $p \in \mathcal{P}$, we write $\mathbb{c}_1 \xrightarrow{p} \mathbb{c}_2$ to denote that $\langle \mathbb{c}_1, p, \mathbb{c}_2 \rangle \in \longrightarrow$.

3 Helper predicates for next section

Predicate	Definition	Meaning
$\mathbb{e} \in \mathbb{E}$: $\texttt{ComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $((\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{ctrl}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e})) \implies (\texttt{status}(\mathbb{e}') = \texttt{com})$	All events preceeding $\mathbb{e}$ in $\prec_{\texttt{data}}$, $\prec_{\texttt{ctrl}}$ or $\prec_{\texttt{poloc}}$ have already been commited.
$\mathbb{e} \in \mathbb{E}^W \cup \mathbb{E}^A$: $\texttt{InitCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^R \cup \mathbb{E}^A$: $(\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All instructions on which $\mathbb{e}$ is data-dependent on have already been initialized.
$\mathbb{e} \in \mathbb{E}^{\texttt{ACI}}$: $\texttt{ValidCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{ACI}}$: $((\mathbb{e} \prec \mathbb{e}') \wedge (\nexists \mathbb{e}'' \in \mathbb{E} : \mathbb{e} \prec \mathbb{e}'' \prec \mathbb{e}')) \implies (((\texttt{Val}(\mathbb{c}, \mathbb{e}) = true) \wedge (\texttt{ins}(\mathbb{e}') = \texttt{Tnext}(\texttt{ins}(\mathbb{e})))) \vee ((\texttt{Val}(\mathbb{c}, \mathbb{e}) = false) \wedge (\texttt{ins}(\mathbb{e}') = \texttt{Fnext}(\texttt{ins}(\mathbb{e})))))$	The instruction that was fetched right after the ACI instruction was consistent with its truth value.

4 Rules without synchronization instructions

In each of these rules, $\mathbb{c} \equiv \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}, \texttt{rf}, \texttt{Prop} \rangle$ denotes the "current" configuration, and the rule describes one possible way in which this configuration may evolve.

$$\frac{\mathbb{e} \notin \mathbb{E}, \quad \prec' = \prec \cup \{\langle \mathbb{e}', \mathbb{e} \rangle | \mathbb{e}' \in \mathbb{E}\}, \quad i \in \texttt{MaxI}(\mathbb{c}, p)}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E} \cup \{\mathbb{e}\}, \prec', \texttt{ins}[\mathbb{e} \leftarrow i], \texttt{status}[\mathbb{e} \leftarrow \texttt{fetch}], \texttt{rf}, \texttt{Prop} \rangle} \texttt{Fetch}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^R, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}', \quad \texttt{status}(\mathbb{e}') = \texttt{init}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \texttt{Prop} \rangle} \texttt{InitReadFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^R, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad (CW(\mathbb{c}, \mathbb{e}) = \bot) \vee (CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \texttt{status}(\mathbb{e}') = \texttt{com})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \texttt{Prop}(\texttt{var}(\mathbb{e}))], \texttt{Prop} \rangle} \texttt{InitReadFromProp}$$

$$\frac{\mathtt{e} \in \mathbb{E}_p^R, \quad \mathtt{status}(\mathtt{e}) = \mathtt{init}, \quad \mathtt{ComCnd}(\mathtt{c}, \mathtt{e})}{\mathtt{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathtt{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop} \rangle}\mathtt{ComRead}$$

$$\frac{\mathtt{e} \in \mathbb{E}_p^W, \quad \mathtt{status}(\mathtt{e}) = \mathtt{fetch}, \quad \mathtt{InitCnd}(\mathtt{c}, \mathtt{e})}{\mathtt{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathtt{e} \leftarrow \mathtt{init}], \mathtt{rf}, \mathtt{Prop} \rangle}\mathtt{InitWrite}$$

$$\frac{\mathtt{e} \in \mathbb{E}_p^W, \quad \mathtt{status}(\mathtt{e}) = \mathtt{init}, \quad \mathtt{ComCnd}(\mathtt{c}, \mathtt{e})}{\mathtt{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathtt{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}[\mathtt{var}(\mathtt{e}) \leftarrow \mathtt{e}] \rangle}\mathtt{ComWrite}$$

$$\frac{\mathtt{e} \in \mathbb{E}_p^A, \quad \mathtt{status}(\mathtt{e}) = \mathtt{fetch}, \quad \mathtt{InitCnd}(\mathtt{c}, \mathtt{e})}{\mathtt{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathtt{e} \leftarrow \mathtt{init}], \mathtt{rf}, \mathtt{Prop} \rangle}\mathtt{InitAssign}$$

$$\frac{\mathtt{e} \in \mathbb{E}_p^A, \quad \mathtt{status}(\mathtt{e}) = \mathtt{init}, \quad \mathtt{ComCnd}(\mathtt{c}, \mathtt{e})}{\mathtt{c} \xrightarrow{p} \langle \mathbb{E}, \mathtt{ins}, \mathtt{status}[\mathtt{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop} \rangle}\mathtt{ComAssign}$$

$$\frac{\mathtt{e} \in \mathbb{E}_p^{\mathtt{ACI}}, \quad \mathtt{status}(\mathtt{e}) = \mathtt{fetch}, \quad \mathtt{ComCnd}(\mathtt{c}, \mathtt{e}), \quad \mathtt{ValidCnd}(\mathtt{c}, \mathtt{e})}{\mathtt{c} \xrightarrow{p} \langle \mathbb{E}, \mathtt{ins}, \mathtt{status}[\mathtt{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop} \rangle}\mathtt{ComACI}$$

The rule `Fetch` chooses the next instruction to be executed from the code of a process $p \in \mathcal{P}$. To do this it should be able to select the next instruction i given the configuration $\mathtt{c}$. To do this, we define the function $\mathtt{MaxI}(\mathtt{c}, p)$ to be the set of instructions that can (intuitively) follow the current one. Formally,

- If $\mathbb{E}_p = \emptyset$, then we define $\mathtt{MaxI}(\mathtt{c}, p) := \{i_p^{init}\}$, i.e. the first instruction to be fetched by p is i_p^{init}.
- If $\mathbb{E}_p \neq \emptyset$ let $\mathtt{e}' \in \mathbb{E}_p$ be the maximal event in $\mathbb{E}_p$ w.r.t $\prec$ in the configuration $\mathtt{c}$. Define $\mathtt{MaxI}(\mathtt{c}, p) := \mathtt{next}(\mathtt{ins}(\mathtt{e}'))$

The remaining rules can be explained as follows:

- Once a read instruction is fetched, it can be satisfied by reading from either the latest `po`-previous write instruction (from the same process) that has not yet been propagated to memory, or can be satisfied by reading the value from memory if no such instruction exists. The rule `InitReadFromLocal` captures the former case, while the rule `InitReadFromProp` captures the latter. In both cases, we update the status of $\mathtt{e}$ to show that it has been 'initialized', and update `rf` to indicate the instruction that satisfied it.
- Once all `po`-previous instructions on which $\mathtt{e}$ depends in some way (via a data, control or poloc dependency) have been committed (i.e. been propagated to memory), a satisfied read can be propagated to memory and be committed. This is expressed in the form of the rule `ComRead`.
- A write instruction can be initialized if all events it is data-dependent on have been initialized. This rule is captured in the form of `InitWrite`. At this point events of the same thread can read from this write.
- Once all `po`-previous instructions on which $\mathtt{e}$ depends in some way (via a data, control or poloc dependency) have been committed (i.e. been propagated to memory), an initialized read can be propagated to memory and be committed. At this point events from all threads can read from this write. This forms `ComWrite`.
- The rules for an assign event are similar to that of a write event.
- An ACI event undergoes only one state transition after being fetched, i.e. being committed. However, such an event can be committed if and only if it is *valid*, i.e., e.g. the predicted branch turns out to be correct, or a loop is entered, etc. Of course, along with this it must satisfy the `ComCnd` which says that all events that $\mathtt{e}$ depends upon must have already been committed. This is captured as the rule `ComACI`.

5 Syntax for programs (complete version)

We introduce a number of instructions resembling those that are used in the AArch64. We explain them in the next section.

$$\begin{aligned}
Prog &::= \mathtt{procs:}\ \ p^* \\
p &::= \mathtt{regs:}\ \ \$r^* \\
&\quad\ \ \mathtt{instrs:}\ \ i^* \\
i &::= l : \mathtt{s} \\
\mathtt{s} &::= \ \ \$r \leftarrow exp \\
&\quad |\mathtt{LD}\ \ \$r' \leftarrow [\$r] \\
&\quad |\mathtt{ST}\ \ [\$r'] \leftarrow \$r \\
&\quad |\mathtt{LDA}\ \ \$r' \leftarrow [\$r] \\
&\quad |\mathtt{STL}\ \ [\$r'] \leftarrow \$r \\
&\quad |\mathtt{LDX}\ \ \$r' \leftarrow [\$r] \\
&\quad |\mathtt{STX}\ \ \$r'',\ [\$r'] \leftarrow \$r \\
&\quad |\mathtt{LDAX}\ \ \$r' \leftarrow [\$r] \\
&\quad |\mathtt{STLX}\ \ \$r'',\ [\$r'] \leftarrow \$r
\end{aligned}$$

| `if` exp `then` i^* `else` i^*
| `while` exp `do` i^*
| `assume` exp
| `assert` exp
| `dmb.ld`
| `dmb.st`
| `dmb.sy`
| `isb`
| `term`

We comment about a few things here: the first thing to notice is that we have removed variables, and instead introduced addressed memory: the latter is exactly the same as the former, with the added complexity that there can now be address dependencies between instructions, which did not exist for variables. We refer to all kinds of loads collectively as "reads", and all kinds of stores collectively as "writes". Second, the assign instruction is closer to a read than it is a write: the primary reason is that it writes to registers and not memory (and thus it's side effects affect the former). In fact, an assign may be represented as a store to a "private" variable followed by a load from it, hence it is in a sense a compound instruction. However, since it does not directly interface with the memory at all, any run is indistinguishable from another wherein an assign is initialized or committed at a different time, as long as the read-from relations to and from it are preserved. Hence, we do not include the assign statement in the statement of the predicates. Third, for simplicity we assume that the addresses of loads and stores are registers, and not expressions of registers. The reason for this is that one, exactly the same rules given in the next-to-next section apply if we use expressions instead, and this, two, we can simply assume that we assign the expression to a reserved register and use it as the address immediately after (this is the same under this model).

6 Definitions and Helper predicates for next section

Along with the usual load/store and read/write operations, we also introduce three "special" types of loads and stores. First, a load or a store can be exclusive. The significance of a load exclusive that loads from an address and a store exclusive that writes back to that address is that the store succeeds if and only if no other event (from any process) has written to that address since the load exclusive. Also, corresponding to whether the store exclusive succeeded or not, the success value is written to a specified register. Note that such a primitive may be used to implement, e.g. locks or `CAS` instructions. There are also load-acquire and store-release instructions. These are special in that it is forbidden for any instruction that appears `po`-after the load-acquire to be reordered with it. Similarly, it is fordbidden for any instruction that appears `po`-before the store-release to be reordered with it. In effect, the combination of a load-acquire and store-release serves like a `dmb.ld`. Finally there are `LDAX` and `STLX` instructions which are simply the combination of both the acquire/release and exclusive type of instructions, i.e. they provide the strictest guarantees. One point to observe here is that by nature of its definition the exclusive versions of stores are multi-copy atomic: every process (including the local one) sees the write at the same time; consequently we can consider it to have only a commit rule (since the `init` rules were meant to portray the period where effects were visible only to the local process).
Since we now have address dependencies, we note that at any point in an execution, for an event $\mathbb{e}$, $\texttt{addr}(\mathbb{e})$ may not be defined, which we write as $\texttt{addr}(\mathbb{e}) = \top$. Then, for two events $\mathbb{e}_1 \prec \mathbb{e}_2$, if $\texttt{addr}(\mathbb{e}_1) = \top$, it is possible that upon resolution, we get later that $\mathbb{e}_1 \prec_{\texttt{poloc}} \mathbb{e}_2$. Subsequently, we include in `poloc`-dependencies all event pairs $(\mathbb{e}_1, \mathbb{e}_2)$ such that $\texttt{addr}(\mathbb{e}_1) = \top$.
Coming to the barriers, there are four types: three are `dmb.ld`,`dmb.st` and `dmb.sy`. The `ld` variant stops the reordering of (local) load instructions (and variants therof, including "reads" from a variable) with it, while the `st` variant does that with stores and writes. The `sy` variant does both. The `isb` instruction is a bit different: all we need is that when it is committed, all `po`-preceeding events have their addresses (to which they write or from which they read) fully determined, i.e. any instruction that they are `addr`-dependent on must have been initialized (or committed).
We introduce events corresponding to these new instructions: we retain our nomenclature of their partition, with the corresponding $\mathbb{E}^{\texttt{name}}$ being the subset of $\mathbb{E}$ corresponding to `name` type of instructions. We should also revise our definition of CW. The events that can now write to a memory location are stores and store releases. Note however that Store Releases will write the success/failure indicator bit (0 is success) into a *register* (called $\$r$ above), hence we need to consider them for the definition of CW as well. Thus, our definition of CW is revised to be the following: for reads and loads $\mathbb{e}$, $CW(\mathbb{e})$ is defined to the `po`-latest store/write or variant therof that is `poloc`-before the store release. Note that this works even if that instruction is exclusive, because since a store exclusive has only a commit rule, that any read that is `poloc` after it has to wait for it to commit; by which time due to the rule, all events `po`-before the SRel are committed as well. Hence, the effects of the `SRel` are already well defined and visible to all processes. We also comment that we refer to both the old type of reads as well as the normal load instructions as "reads", and the old type of writes as well as plain stores as "writes".
Since we need to be able to determine if an intervening write overwrote the value written by an exclusive load, we introduce into our concept of a configuration a mapping function $\texttt{Mark} : \mathcal{X} \mapsto \mathcal{P} \cup \{\bot\}$. Here $\mathcal{X}$ includes addressable memory locations instead of variables as it did before: both are the same, except that variables do not have address dependencies. Accordingly, we also reuse the function `var` to return the address of a load/store event. The idea is that any exclusive load "marks" the memory location with its originating process, while all other store based instructions simply overwrite the marker with $\bot$ to record that the location is dirty.

We also define the function $\texttt{GetUpdate}(\mathbb{e}) : (\mathbb{E}^{\texttt{STX}} \cup \mathbb{E}^{\texttt{STLX}}) \mapsto \mathbb{E}$, which, depending on whether the exclusive event $\mathbb{e}_2$ succeeds, updates the memory. It is defined as follows: if $\texttt{Mark}(\texttt{var}(\mathbb{e})) = \texttt{proc}(\mathbb{e})$, then the function equals $\mathbb{e}_2$, and otherwise $\texttt{Prop}(\texttt{var}(\mathbb{e}))$. Note that this correctly captures the idea of the success-value of a store-exclusive: a store exclusive can be successful only if the last event propagated to memory is an exclusive load from the same process. We also define $\texttt{GetMark}(\mathbb{e}_2) : \times(\mathbb{E}^{\texttt{STX}} \cup \mathbb{E}^{\texttt{STLX}}) \mapsto \mathcal{P} \cup \{\bot\}$. This function determines whether to update the mark of a location based on whether an exclusive store is successful. Similarly to $\texttt{GetUpdate}$, if $\texttt{Mark}(\texttt{var}(\mathbb{e})) = \texttt{proc}(\mathbb{e})$, then the function equals $\bot$, and otherwise $\texttt{Mark}(\texttt{var}(\mathbb{e}))$. Similarly, when an exclusive load is committed, we must ensure that the event we read from is the one that is on "top" at the memory, since we may have had some other process' writes intervening in after we read the value. Hence we define, $\texttt{GetLoadMark}(\mathbb{e}) : (\mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}}) \mapsto \mathcal{P} \cup \{\bot\}$: if $\texttt{Prop}(\texttt{var}(\mathbb{e}_2)) = \texttt{rf}(\mathbb{e})$, then the function equals $\texttt{proc}(\mathbb{e})$, and otherwise $\texttt{Mark}(\texttt{var}(\mathbb{e}))$.
We next define a bunch of predicates which will prove to be useful in stating the transition rules.

Predicate	Definition	Meaning
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllDmbLds}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbLd}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.ld`'s have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllDmbSts}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbSt}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.st`'s have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllDmbSys}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbSy}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.sy`'s have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}} \cup \mathbb{E}^{\texttt{DmbSt}}$: $\texttt{AllWrites}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STX}} \cup \mathbb{E}^{\texttt{STLX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous write-variants have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}} \cup \mathbb{E}^{\texttt{DmbLd}}$: $\texttt{AllReads}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LD}} \cup \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous read-variants have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllSyncs}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbSy}} \cup \mathbb{E}^{\texttt{Isb}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.sy`'s and `isb`'s have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDAX}}$: $\texttt{AllSRels}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous Store-Releases have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{LD}} \cup \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}}$: $\texttt{AllLAcqInit}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDAX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com})))$	All po-previous Load-Acquires have been initialized.
$\mathbb{e} \in \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STX}}$: $\texttt{AllLAcqs}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDAX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous Load-Acquires have been committed.
$\mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}}$: $\texttt{AllMem}(\mathbb{c}, \mathbb{e})$	$\texttt{AllWrites}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllReads}(\mathbb{c}, \mathbb{e})$	All po-previous events that access memory have been committed.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllBarriers}(\mathbb{c}, \mathbb{e})$	$\texttt{AllDmbLds}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllDmbSts}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllSyncs}(\mathbb{c}, \mathbb{e})$	All po-previous barriers have been committed.
$\mathbb{e} \in \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{A}$: $\texttt{InitCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LD}} \cup \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}} \cup \mathbb{E}^{A}$: $(\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All events on which $\mathbb{e}$ is dependent on have already been initialized.
$\mathbb{e} \in \mathbb{E}^{\texttt{Isb}} \cup \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STX}} \cup \mathbb{E}^{\texttt{STLX}}$: $\texttt{AddrCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LD}} \cup \mathbb{E}^{A} \cup \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}}, \mathbb{e}'' \in \mathbb{E}$: $(\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}'') \wedge (\mathbb{e}'' \prec_{\texttt{po}} \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All events po-before $\mathbb{e}$ have fully defined memory footprints.
$\mathbb{e} \in \mathbb{E}$: $\texttt{ComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $(((\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{ctrl}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}))) \implies (\texttt{status}(\mathbb{e}') = \texttt{com})) \wedge ((\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}) \implies (\texttt{status}(\mathbb{e}) = \texttt{init}) \vee (\texttt{status}(\mathbb{e}) = \texttt{commit}))$	All events preceding $\mathbb{e}$ in $\prec_{\texttt{data}}$, $\prec_{\texttt{ctrl}}$ or $\prec_{\texttt{poloc}}$ have already been commited, and all events preceding $\mathbb{e}$ in $\prec_{\texttt{addr}}$ have already been initialized.

7 Rules with synchronization conditions

For each of the rules below, $\mathbb{c}$ is the configuration before the rule, and is of the form $\langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}, \texttt{rf}, \texttt{Prop}, \texttt{Mark} \rangle$. The rules describe one possible way in which this configuration may evolve. The initial configuration is $\langle \emptyset, \emptyset, \lambda i.\emptyset, \lambda i.\emptyset, \lambda i.\emptyset, \lambda i.0, \lambda i.\bot \rangle$.

$$\frac{\mathbb{e} \in \mathbb{E}_p, \quad \prec' = \prec \cup \{\langle \mathbb{e}', \mathbb{e} \rangle | \mathbb{e}' \in \mathbb{E}_p\}, \quad i \in \texttt{MaxI}(\mathbb{c}, p)}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E} \cup \mathbb{e}, \prec', \texttt{ins}[\mathbb{e} \leftarrow i], \texttt{status}[\mathbb{e} \leftarrow \texttt{fetch}], \texttt{rf}, \texttt{Prop}, \texttt{Mark} \rangle} \texttt{Fetch}$$

$$\mathbb{e} \in \mathbb{E}_p^{\texttt{LD}} \cup \mathbb{E}_p^{\texttt{LDX}}, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{AllSyncs}(\mathbb{c}, \mathbb{e}),$$

$$\frac{\texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllLacqInit}(\mathbb{c},\mathbb{e}),\quad \mathbb{e}' = CW(\mathbb{c},\mathbb{e}),\quad \texttt{status}(\mathbb{e}') = \texttt{init}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \texttt{Prop}, \texttt{Mark}\rangle}\texttt{InitLDOrLDXFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{LD}} \cup \mathbb{E}_p^{\texttt{LDX}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllLacqInit}(\mathbb{c},\mathbb{e}),\quad (CW(\mathbb{c},\mathbb{e}) = \bot) \vee (CW(\mathbb{c},\mathbb{e}) = \mathbb{e}' \wedge \texttt{status}(\mathbb{e}') = \texttt{com})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \texttt{Prop}(\texttt{Var}(\mathbb{c},\mathbb{e}))], \texttt{Prop}, \texttt{Mark}\rangle}\texttt{InitLDOrLDXFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{LDA}} \cup \mathbb{E}_p^{\texttt{LDAX}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSRels}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllLacqInit}(\mathbb{c},\mathbb{e}),\quad \mathbb{e}' = CW(\mathbb{c},\mathbb{e}),\quad \texttt{status}(\mathbb{e}') = \texttt{init}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \texttt{Prop}, \texttt{Mark}\rangle}\texttt{InitLDAOrLDAXFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{LDA}} \cup \mathbb{E}_p^{\texttt{LDAX}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllLacqInit}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSRels}(\mathbb{c},\mathbb{e}),\quad (CW(\mathbb{c},\mathbb{e}) = \bot) \vee (CW(\mathbb{c},\mathbb{e}) = \mathbb{e}' \wedge \texttt{status}(\mathbb{e}') = \texttt{com})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \texttt{Prop}(\texttt{Var}(\mathbb{c},\mathbb{e}))], \texttt{Prop}, \texttt{Mark}\rangle}\texttt{InitLDAOrLDAXFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{LD}} \cup \mathbb{E}_p^{\texttt{LDA}},\quad \texttt{status}(\mathbb{e}) = \texttt{init},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark}\rangle}\texttt{ComNonExclusiveRead}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{LDX}} \cup \mathbb{E}_p^{\texttt{LDAX}},\quad \texttt{status}(\mathbb{e}) = \texttt{init},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark}[\texttt{var}(\mathbb{e}) \leftarrow \texttt{GetLoadMark}(\mathbb{e})]\rangle}\texttt{ComExclusiveRead}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{ST}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{InitCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllLAcqs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSts}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}, \texttt{Prop}, \texttt{Mark}\rangle}\texttt{InitST}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{STL}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{InitCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllMem}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSts}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{Prop}, \texttt{Mark}\rangle}\texttt{InitSTL}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{ST}} \cup \mathbb{E}_p^{\texttt{STL}},\quad \texttt{status}(\mathbb{e}) = \texttt{init},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AddrCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}[\texttt{var}(\mathbb{e}) \leftarrow \mathbb{e}], \texttt{Mark}[\texttt{var}(\mathbb{e}) \leftarrow \bot]\rangle}\texttt{ComNonExclusiveWrite}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{STX}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AddrCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllLAcqs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSts}(\mathbb{c},\mathbb{e}),\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}, \texttt{Prop}[\texttt{var}(\mathbb{e}) \leftarrow \texttt{GetUpdate}(\mathbb{e})], \texttt{Mark}[\texttt{var}(\mathbb{e}) \leftarrow \texttt{GetMark}(\mathbb{e})]\rangle}\texttt{ComSTX}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{STLX}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{AddrCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbLds}(\mathbb{c},\mathbb{e}),\quad \texttt{AllMem}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSts}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}[\texttt{var}(\mathbb{e}) \leftarrow \texttt{GetUpdate}(\mathbb{e})], \texttt{Mark}[\texttt{var}(\mathbb{e}) \leftarrow \texttt{GetMark}(\mathbb{e})]\rangle}\texttt{ComSTLX}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{DmbSy}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllMem}(\mathbb{c},\mathbb{e}),\quad \texttt{AllBarriers}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}\rangle}\texttt{ComDmbSy}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{Isb}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AddrCnd}\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSys}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}\rangle}\texttt{ComIsb}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{DmbLd}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllReads}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSys}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark}\rangle}\texttt{ComDmbLd}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{DmbSt}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllWrites}(\mathbb{c},\mathbb{e}),\quad \texttt{AllDmbSys}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark}\rangle}\texttt{ComDmbSt}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{ACI}},\quad \texttt{status}(\mathbb{e}) = \texttt{fetch},\quad \texttt{ComCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{ValidCnd}(\mathbb{c},\mathbb{e}),\quad \texttt{AllSyncs}(\mathbb{c},\mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark}\rangle}\texttt{ComACI}$$

We explain the above rules as thus:

- The `Fetch` rule is the same as before.
- We need the following pre-conditions to hold for initializing (satisfying) a non-acquire read (exclusive as well as not) event $\mathbb{e}$ (from either local uncommitted events or memory):
  1. For all events $\mathbb{e}'$ that are poloc-predecessors of $\mathbb{e}$, $\mathbb{e}'$ must have been initialized.
  2. All `dmb.sy`, `isb` and `dmb.ld` instructions that are po-predecessors of $\mathbb{e}$ have been committed.
  3. All po-preceeding load acquire instructions have been initialized, i.e. entirely satisfied.
- The rule for initializing an acquire-read (exclusive as well as not) is similar to that above, but with the extra condition that all po-previous Store Releases have been committed.

- One condition required for committing for any kind of write needs that we know the "memory footprint", i.e. address of all po-previous memory access events is known.
- To initialize a (non-exclusive, non-release) write, along with an InitCnd that we had previously (but which also includes address dependencies now), we need the following condition to hold: all po-previous dmb.sy, isb, dmb.ld, dmb.st, and Load Acquire instructions have been committed.
- To initialize an acquire but non-exclusive write (STL), instead of just load-acquires, we need *all* po-preceeding memory access instructions to have been committed.
- To commit a store release, the required conditions are almost the same as for initializing writes, with the difference that instead of just load acquires, all po-previous memory access instructions have been committed. Another difference is that it directly goes to the commit stage, and swaps out memory if and only if no other instruction has modified the main memory at that location since the last load acquire from the same process.
- The exclusive versions of the two kinds of writes are to be directly committed: an STX can be committed only if it satisfies all the conditions mentioned above for simple write - initialization, along with the ComCnd. On the other hand, an STLX instruction can be committed only if all po-preceeding instructions excluding ACI events have been committed.
- The condition for committing a non-exclusive write is similar to before, but with the added condition that all po-previous synchronization instructions(dmb.sy and isb) are committed.
- All ACI and barrier instructions have only one transition: from fetch to commit. Across all of them, the ComCnd and ValidCnd are common conditions, as is the condition that all dmb.sy's have been committed (for ACI's and dmb.sy's we also need po-previous isb's to have been committed). Apart from this,

  1. For dmb.sy's we need all po-previous memory access instructions to have committed.
  2. For isb's we need all po-previous memory access instructions to have fully defined memory footprints.
  3. For dmb.ld's we need all po-previous load (read and load acquire) instructions to have been committed.
  4. For dmb.st's we need all po-previous stores (write and store release) instructions to have been committed.

Assign instructions. The Herd or the Flat model do not explicitly handle assign instructions. We can instead imagine it as writing the value of exp to a new variable z which is later read into the register $\$r$. Since this variable is never used by any other process, we can construct a rule for this assign statement by simply merging the rules for read and write. Since AllLacqInit is a weaker condition than AllLacqs, we can simply use the predicates for the write event. One must, however, keep in mind that this is a derived rule.

$$\frac{\mathfrak{e} \in \mathbb{E}^A_p, \quad \mathtt{status}(\mathfrak{e}) = \mathtt{fetch}, \quad \mathtt{InitCnd}(\mathbb{c}, \mathfrak{e}), \quad \mathtt{AllSyncs}(\mathbb{c}, \mathfrak{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathfrak{e}), \quad \mathtt{AllLAcqs}(\mathbb{c}, \mathfrak{e}), \quad \mathtt{AllDmbSts}(\mathbb{c}, \mathfrak{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathfrak{e} \leftarrow \mathtt{init}], \mathtt{rf}, \mathtt{Prop} \rangle}\,\mathtt{InitAssign}$$

$$\frac{\mathfrak{e} \in \mathbb{E}^A_p, \quad \mathtt{status}(\mathfrak{e}) = \mathtt{init}, \quad \mathtt{ComCnd}(\mathbb{c}, \mathfrak{e}), \quad \mathtt{AllSyncs}(\mathbb{c}, \mathfrak{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathfrak{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}[\mathtt{var}(\mathfrak{e}) \leftarrow \mathfrak{e}] \rangle}\,\mathtt{ComAssign}$$

8 Proof of equivalence to the Herd Model

We give here the proof of equivalence of the above model to the Herd model, as found in section B2.3 of the ARM developer manual. There are two main conditions for the equivalence to hold: the local ordering constraints and the external ordering constraints. We do not take into account tag reads or tag writes since our model does not include them. The local ordering constraints consist of the following:

- Constraints due to local-write successors. These correspond to write-write pairs related by $\prec_{\mathtt{poloc}}$. Consider any valid execution under our model, and consider two write events $\mathfrak{e}_1 \prec_{\mathtt{poloc}} \mathfrak{e}_2$. Note that $\mathfrak{e}_1$ is always committed before $\mathfrak{e}_2$, and because reads can read only from po-previous writes, an execution in which $\mathfrak{e}_1$ is initialized after $\mathfrak{e}_2$ is functionally equivalent to one where $\mathfrak{e}_1$ is initialized just before $\mathfrak{e}_2$ (because reads that occur po-between the two events will anyway read from $\mathfrak{e}_2$ in our model; and for those after $\mathfrak{e}_2$ this leads to a functionally same scenario). Hence, reachability is preserved by our model for this rule.

- Dependency ordered before relations are preserved. This consists of the rf,addr,data and ctrl dependencies. The first follows immediately from the definition of CW. In our formulation of the syntax of concurrent programs addr and data dependencies both are part of $\prec_{\mathtt{data}}$ (since we represent loads by $\$r \longleftarrow x$ and stores by $x \longleftarrow \$r$). Then the other three follow from InitCnd which appears in all initialization rules for writes, and the definition of CW for reads. The AllSyncs predicate which appears in all synchronization rules takes care of the fourth.
  We should also resolve one point here: suppose that a read instance $\mathfrak{e}$ has $CW(\mathfrak{e}) = \mathfrak{e}' \in \mathbb{E}^{\mathtt{SRel}}$. Then, $\mathfrak{e}$ should read from $\mathfrak{e}'$ only if the latter is successful, and otherwise from the po-latest event $\mathfrak{e}''$ po-before $\mathfrak{e}'$ (or from a write by another process). Note however, that before an SRel is committed, we need all po-preceeding memory instructions to have been committed; further we need $\mathfrak{e}'$ to have been committed by the time $\mathfrak{e}$ is initialized, according to our model. Hence, we

always have that the effects of $\mathbb{e}'$ (and preceeding instructions including $\mathbb{e}''$) if any, have already propagated to memory before $\mathbb{e}$ is initialized. Hence, our set of rules handles this case correctly.

- Atomic-ordered-before relations are preserved. These consist of the following two cases:
  - The relation between a successful load exclusive - store exclusive pair is preserved. This is true in our model, since we need the load exclusive to have been committed before the store exclusive: hence, if the store exclusive succeeded, there must have been no intervening writes and hence the store exclusive must have read from the load exclusive.
  - The ordering between a store exclusive and a `poloc`-later load acquire that is a local read successor of it is preserved: this follows since we need the local read predecessor of a load to have been initialized (at least) before it itself is; since a store exclusive directly commits, we have that the store exclusive must have been committed under our model by the time the load acquire is initialized.

- Barrier-ordered-before relations are preserved. These relations are further divided into the following:
  1. The ordering between two events separated by a dmb.full. The `AllSyncs` predicate takes care of this, along with the predicates of the `ComDmbSy` and `ComIsb` rules.
  2. The ordering between two events that form an release-acquire pair. The release-acquire pairs are said to be like two halves of a `dmb.full`. The `AllSRels` predicate of `LAcqInit` handles this.
  3. The ordering between a read-event pair where the read is po-before a `dmb.ld` which is po-before the event. The `AllReads` condition of `ComDmbLd` combined with `AllDmbLds` in all of the `init` rules ensures that this constraint holds.
  4. The ordering between a `LAcq` and po-later event is preserved. The `AllLacqInit` (and the `AllLacqs` in case of `InitWrite`) that appears in all the `init` rules takes care of this.
  5. The ordering between a write-write pair where the first write is po-before a `dmb.st` which is po-before the second write. First note that the `AllWrites` predicate in `ComDmbSt` ensures that the write is committed before the `dmb.st`. Further, the `AllDmbSts` predicate of `InitWrite` and `InitSRel` ensure that the `dmb.st` is committed before the second write is initialized.
  6. The ordering between a release and a *preceeding* event is preserved. This is ensured by the `AllMem` predicate in `ComSRel`.

- The transitive closure of the above must hold. Since we showed that each of the above orderings holds individually, the transitive closure follows easily.

Next we tackle the proof of the global ordering constraints. We choose the ordered-before formulation of the external ordering constraints as presented in ARM's manual. An event $\mathbb{e}_1$ is *ordered-before* another event $\mathbb{e}_2$ if and only if at least one of the following hold:

- $\mathbb{e}_1$ is *observed by* (i.e. is read by) $\mathbb{e}_2$, or
- $\mathbb{e}_1$ is *locally ordered* before $\mathbb{e}_2$, or
- There exists an event $\mathbb{e}_3$ such that $\mathbb{e}_1$ is ordered before $\mathbb{e}_3$ which in turn is ordered before $\mathbb{e}_2$ (transitive closure).

Then, the external ordering constraint requires that there are no cycles in the ordered-before relation, i.e. that all processors see a consistent view of the order. Note that, denoting the ordered-before relation by $\prec_{ob}$, if $\mathbb{e}_1 \prec_{ob} \mathbb{e}_2$ where $\mathbb{e}_1$ and $\mathbb{e}_2$ belong to different processes, then since in our model the memory's contents can be overwritten by newer events but not "uncovered", $\mathbb{e}_2$ "covers" $\mathbb{e}_1$ and hence the first event cannot be read after that by any events: this is because remote reads see a more recent version of the memory, while local reads either read from the main memory which has the same effect, or from a local read which is *more recent* than the main memory which is in turn more recent than $\mathbb{e}_1$. Thus, the only way cycles can enter the $\prec_{ob}$ ordering is if there is a cycle in the local ordering, which as established above is not possible.
Thus we conclude that every valid execution under our model (which we call ARM') corresponds to a valid execution under ARM.
Further, note that *every predicate that occurs as a part of our rules follows from one of the conditions of ARM*. This means that since any valid execution under ARM satisfies exactly these predicates, any valid execution under ARM satisfies each of these rules. Thus, every execution under ARM is valid under ARM'.
Using the above two results, we conclude that $\boxed{ARM \equiv ARM'}$, completing the proof of equivalence.

9 Context Bound Model Checking for ARM

In the spirit of Context Bound Model Checking, we propose a code-to-code translation from source code written under the ARM v8 model to one that runs under Sequential Consistency (SC), but with a bound on the maximum number of contexts of execution. Here, we define a context as a stretch of execution (across time) in which only one processor is active. Thus, inside a single context, the Herd model behaves exactly like SC to the executing process (and the other processes are not

executing). This context-bound reduction was first pioneered by Lal and Reps, and we adapt their approach (with the required changes) to fit the ARM model. In particular, our code guesses the global state (the state of the main memory) at all context switches. Then, our code simulates, for each process, the contexts in which it is active, starting from the respective guessed global states. In the end, we verify, that for each context k, the process active in context k *transforms* the global state from that guessed at the end of context $k-1$ to that guessed at the end of context k. The advantage of this approach over naive simulation is that it avoids having to deal with cross-products of local spaces, and hence avoids the exponential blowup associated with it. As an end result, we have code that when run under SC successfully simulates all possible runs under the ARM v8 memory model that incur at most k context switches. That is, Context Bound Model Checking is an efficient under-approximation to state reachability. Since it has been demonstrated before that most bugs are (as a general rule of thumb) reachable in a small number of context switches, our approach is an effective means of tacking the undecidability of the ARM v8 memory model.
We first demonstrate the above method on the simple model introduced at the beginning of the document, and then proceed to adapt it to the whole model.

10 Code-to-code translation for reduction to SC (for the simple model)

For the sake of performing context-bound model checking, we next define the scheme for the code-to-code translation.

10.1 Scheme for the first part of model (without synchronization instructions)

Our translation scheme translates a program $Prog$ into a program $Prog^{\clubsuit}$ using the map function $[\![\cdot]\!]_K$. Let $\mathcal{P}$ and $\mathcal{X}$ be the set of processes and shared variables in our program. Then, the map $[\![\cdot]\!]_K$ replaces the variables of $Prog$ by $|\mathcal{P}| \cdot K$ copies of the set $\mathcal{X}$, along with a finite set of finite data structures defined below. Below, the function `gen` takes in a finite set and returns a randomly chosen element of the set.

$$[\![Prog]\!]_K \stackrel{\text{def}}{=} \begin{array}{ll}\texttt{vars:} & x^* \ \langle \texttt{addvars} \rangle_K \\ \texttt{procs:} & ([\![p]\!]_K)^* \ \langle \texttt{initProc} \rangle_K \ \langle \texttt{verProc} \rangle_K\end{array}$$

$$[\![p]\!]_K \stackrel{\text{def}}{=} \begin{array}{ll}\texttt{regs:} & \$r^* \\ \texttt{instrs:} & ([\![i]\!]^p_K)^*\end{array}$$

$$[\![i]\!]^p_K \stackrel{\text{def}}{=} l: \ \langle \texttt{activeCnt} \rangle^p_K \ [\![\texttt{s}]\!]^p_K \ \langle \texttt{closeCnt} \rangle^p_K$$

$$[\![x \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![x \leftarrow exp]\!]^{\texttt{p,Write}}_K$$

$$[\![\$r \leftarrow x]\!]^p_K \stackrel{\text{def}}{=} [\![\$r \leftarrow x]\!]^{\texttt{p,Read}}_K$$

$$[\![\$r \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![\$r \leftarrow exp]\!]^{\texttt{p,Assign}}_K$$

$$[\![\texttt{if } exp \texttt{ then } i^* \texttt{ else } i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{if } exp \texttt{ then } ([\![i]\!]^p_K)^* \texttt{ else } ([\![i]\!]^p_K)^*; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{while } exp \texttt{ do } i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{while } exp \texttt{ do } ([\![i]\!]^p_K)^*; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assume } exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assume } exp; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assert } exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assert } exp; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{term}]\!]^p_K \stackrel{\text{def}}{=} \texttt{term}$$

$$\begin{array}{c}\langle \texttt{addvars} \rangle_K \stackrel{\text{def}}{=} \mu(|\mathcal{X}|, K), \mu^{init}(|\mathcal{X}|, K), \nu(|\mathcal{P}|, |\mathcal{X}|), \\ \texttt{iR}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cR}(|\mathcal{P}|, |\mathcal{X}|), \\ \texttt{iW}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cW}(|\mathcal{P}|, |\mathcal{X}|), \\ \texttt{iReg}(|\mathcal{P}|, |\mathcal{R}|), \texttt{cReg}(|\mathcal{P}|, |\mathcal{R}|), \\ \texttt{ctrl}(|\mathcal{P}|), \texttt{active}(K), cnt\end{array}$$

$$\langle \texttt{activeCnt} \rangle^p_K \stackrel{\text{def}}{=} \texttt{assume}(\texttt{active}(\texttt{cnt}) = p)$$

$$\begin{array}{c}\langle \texttt{closeCnt} \rangle^p_K \stackrel{\text{def}}{=} cnt \leftarrow cnt + \texttt{gen}([0, \cdots, K-1]); \\ \texttt{assume}(cnt \leq K)\end{array}$$

$$\begin{array}{c}\langle \texttt{control} \rangle^p_K \stackrel{\text{def}}{=} \texttt{ctrl}(p) \leftarrow \texttt{ctrl}(p) + \texttt{gen}(0, \cdots, K-1); \\ \texttt{assume}(\texttt{ctrl}(p) \leq K)\end{array}$$

10.2 Data Structures (for the model without synchronization)

We denote by $\mathcal{D}$ the domain of all possible values of expressions and variables. Our simulation maintains, as described above, a finite set of finite data structures. We explain each in turn. First, for each context k, we store the ID of the active process p in the context k, using the mapping $\texttt{active} : [1, \cdots, K] \mapsto \mathcal{P}$. The mapping $\mu^{init} : \mathcal{X} \times [1, \cdots, K] \mapsto \mathcal{D}$ maintains, for each variable x and context k the last value of the variable x that has been propagated to memory by any process *upto the beginning of context* k. We also define the mapping $\mu : \mathcal{X} \times [1, \cdots, K] \mapsto \mathcal{D}$ as the counterpart of the above mapping that actually changes (gets updated) through the course of context k, i.e. at any given point in time covered by context k, $\mu(x, k)$ gives the latest value of variable x propagated to memory until that point. Further, the mapping $\nu : \mathcal{P} \times \mathcal{X} \mapsto \mathcal{D}$ denotes for each process p and variable x the latest value that has been written to x by p. We also maintain the maps

$$\texttt{iW} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{cW} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{iR} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{cR} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

The first map indicates, for each process p and variable x, the latest context in which a write on x has been initialized by p. Similarly, the second one indicates for each such pair the latest context in which a write was committed by p; and the third indicates the latest context in which a read of x was initialized by p. Similarly, we define `cR` for committing reads. It must be noted that a read is not propagated to other processes. The only purpose of separately initializing and committing them is for a scenario like this, consider a dependency $w_1 \xrightarrow{\texttt{rf}} r \xrightarrow{\texttt{data}} w_2$. Suppose the `init` times of the three are $1, 2, 3$ respectively, with w_1 committed only at 5. We need to ensure that w_2 is not committed before 5, but we would like to consider only immediate dependencies. Thus it is more convenient to assign r a commit timestamp ≥ 5, so that we can impose the constraint saying "w_2 commits after whatever time r commits".
Since registers are shared among all processes, we need to define special mappings for them. Thus we define $\texttt{iReg} : \mathcal{P} \times \mathcal{R} \mapsto [1, \cdots, K]$ which captures for each register $\$r$ the initializing context of the latest read or assign event loading a value to $\$r$. Similarly $\texttt{cReg} : \mathcal{P} \times \mathcal{R} \mapsto [1, \cdots, K]$ captures for each register $\$r$ the committing context of the latest read or assign event loading a value to $\$r$. We also extend this to expressions, by defining $\texttt{iReg}(p, exp) = \max\{\texttt{iReg}(p, \$r) | \$r \in \mathcal{R}(exp)\}$. Similarly we extend the other two definitions to expressions.
Finally, the mapping $\texttt{ctrl} : \mathcal{P} \mapsto [1, \cdots, K]$ gives for each process p the committing context of the latest `aci` event in p. The variable *cnt* tracks the current context.
The function `gen` is assumed to return, given a set S, a random element of S.

10.3 The initializing process

We next present the algorithm $\langle \texttt{initProc} \rangle_K$.

Algorithm 1: Algorithm $\langle \texttt{initProc} \rangle_K$.

for $p \in \mathcal{P} \wedge x \in \mathcal{X}$ **do**
  $\texttt{iR}(p, x) \leftarrow 1$;
  $\texttt{iW}(p, x) \leftarrow 1$;
  $\texttt{cW}(p, x) \leftarrow 1$;
  $\texttt{cR}(p, x) \leftarrow 1$;
  $\nu(p, x) \leftarrow 0$;
  $\mu(p, x, 1) \leftarrow 0$;
  for $k \in [2, \cdots, K]$ **do**
    $\mu^{init}(x, k) \leftarrow \texttt{gen}(\mathcal{D})$;
    $\mu(x, k) \leftarrow \mu^{init}(p, x, k)$;
  end
end
for $p \in \mathcal{P}$ **do**
  $\texttt{ctrl}(p) \leftarrow 1$;
end
for $\$r \in \mathcal{R}$ **do**
  $\texttt{iReg}(p, \$r) \leftarrow 1$;
  $\texttt{cReg}(p, \$r) \leftarrow 1$;
end
for $k \in [1, \cdots, K]$ **do**
  $\texttt{active}(k) \leftarrow \texttt{gen}([1, \cdots, K])$;
end
$cnt \leftarrow 1$;

The initial process's job is to initialize all the data structures. We assume that in the beginning all variables are assigned the value 0. If needed we can replace this by a symbolic variable depicting all possible values. The structures which record contexts are all initialized to 1, since that is the earliest context possible. We also assign to each context a randomly chosen active process.

10.4 Write instructions

Algorithm 2: $[\![x \leftarrow exp]\!]_K^{p,\texttt{Write}}$

```
1  //Guess
2  iW(p, x) ← gen([1, ··· , K]);
3  old-cW ← cW(p, x);
4  cW(p, x) ← gen([1, ··· , K]);
5  //Check
6  assume(iW(p, x) ≥ cnt);
7  assume(active(iW(p, x)) = p);
8  assume(iW(p, x) ≥ iReg(p, exp));
9  assume(cW(p, x) ≥ iW(p, x));
10 assume(cW(p, x) ≥ max(old-cW, cReg(p, exp), cR(p, x), ctrl(p));
11 //Update
12 μ(x, cW(p, x)) ← exp;
13 ν(p, x) ← exp;
```

The above listing shows how write instructions are handled. First, we guess the contexts in which the write will be initialized and committed respectively. Next, we perform a set of sanity checks to ensure that the write conforms to the rule `InitWrite`: line 6 checks that the write is initialized after the current context (when it is fetched); line 7 ensures that p is the active process in that context. We then make sure that `InitCnd` is satisfied in line 8. Further, the write must be initialized before committed, as expressed in line 9. Finally, we must ensure that the write is committed not before any `poloc`-preceeding write or read, and also not before any `po`-previous aci instruction or before any instruction which supplies the data for this write. That is expressed in line 10. Then, we update the new values of the global values of x in the respective context, and the latest local value of x.

10.5 Read Instructions

Algorithm 3: $[\![\$r \leftarrow x]\!]_K^{p,\texttt{Read}}$

```
1  //Guess
2  iR(p, x) ← gen([1, ··· , K]);
3  old-cR ← cR(p, x);
4  cR(p, x) ← gen([1, ··· , K]);
5  iReg($r) ← iR(p, x);
6  cReg(p, $r) ← cR(p, x);
7  //Check
8  assume(iR(p, x) ≥ cnt);
9  assume(active(iR(p, x)) = p);
10 assume(iR(p, x) ≥ iW(p, x));
11 assume(cR(p, x) ≥ iR(p, x));
12 assume(active(cR(p, x)) = p);
13 assume(cR(p, x) ≥ max(old-cR, cW(p, x), ctrl(p));
14 //Update
15 if iR(p, x) < cW(p, x) then
16 |  $r ← ν(p, x);
17 else
18 |  $r ← μ(x, iR(p, x));
19 end
```

As for writes, we start by guessing the contexts in which the read is initialized and committed. We then perform some sanity checks in the lines 8-14, which we explain next. Line 8 checks that the read is initialized only after being fetched (after or in the current context). Lines 9 and 12 ensure that p is the active process during the initialization and commit of the read. Line 10 ensures that the read is initialized after the closest po-before write is; this is needed due to the model. Line 11 expresses that the read is committed no earlier than it is initialized. Further, we need that the read is committed no earlier than a read/write that is `poloc`-before it, or any `aci` instruction that is `po`-before it. This is expressed in line 13. If the read is initialized before the closest `po`-before write (say w) is committed, then we must follow the rule `InitReadFromLocal`, as dictated by line 16. Otherwise, the global memory is at least as up-to-date, and we must use the rule `InitReadFromProp` to

read from it. This is captured by the update of rule 18.

10.6 Assign Instructions

Algorithm 4: $[\![\$r \leftarrow exp]\!]_K^{p,\texttt{Assign}}$

```
1  //Guess
2  iReg(p, $r) ← gen([1, ··· , K]);
3  cReg(p, $r) ← gen([1, ··· , K]);
4  //Check
5  assume(iReg(p, $r) ≥ cnt);
6  assume(active(iReg(p, $r)) = p);
7  assume(iReg($r) ≥ iReg(p, exp));
8  assume(active(cReg(p, $r)) = p);
9  assume(cReg(p, $r) ≥ iReg(p, $r));
10 assume(cReg(p, $r) ≥ max(cReg(p, exp), ctrl(p)));
11 //Update
12 $r ← exp;
```

We first guess the initializing and commiting context of the assign statement. Then we run through a few sanity checks that we now describe. We first check at line 4 that the statement is initialized only after it is fetched in the current context *cnt*. We also want that the statement only be committed no earlier than it is initialized; this is captured by line 9. We further need the process p to be active in both the contexts responsible for initialization and commit: this is expressed in lines 6 and 8. Since an assign statement can be initialized only after the statements it is dependent on (via `data`), we have line 7 which is `InitCnd`. Finally, the statement cannot be committed any earlier than a statement it depends on, or any `po`-preceeding aci statement. This corresponds to line 10. The update step is simple: we just record the new value of the register $\$r$.

10.7 Verifying process

Algorithm 5: $\langle \texttt{verProc} \rangle_K$.

```
1 for p ∈ P ∧ x ∈ X ∧ k ∈ [1, ··· , K − 1] do
2 |  assume(μ(p, x, k) = μ^init(p, x, k + 1));
3 end
4 if l is reachable then
5 |  error;
6 end
```

The verifying process simply makes sure that the modifications to global state by a process in a context leaves it exactly in the state that the process of the next context finds it. If this is satisfied, then the execution is valid and the *error* state is reachable if and only if a bad state can be reached in this reduced SC program.

11 Code-to-code translation for reduction to SC (for the complete model)

11.1 Scheme for the complete model

Our translation scheme translates a program *Prog* into a program $Prog^{\clubsuit}$ using the map function $[\![\cdot]\!]_K$. Let $\mathcal{P}$ and $\mathcal{X}$ be the set of processes and shared memory locations in our program. Then, the map $[\![\cdot]\!]_K$ replaces the memory of *Prog* by $|\mathcal{P}| \cdot K$ copies of the set $\mathcal{X}$, along with a finite set of finite data structures defined below. Below, the function `gen` takes in a finite set S and returns a randomly chosen element of S.

$$[\![Prog]\!]_K \stackrel{\text{def}}{=} \langle \texttt{addvars} \rangle_K \; \texttt{procs:} \; ([\![p]\!]_K)^* \; \langle \texttt{initProc} \rangle_K \; \langle \texttt{verProc} \rangle_K$$

$$[\![p]\!]_K \stackrel{\text{def}}{=} \texttt{regs:} \; \$r^* \; \texttt{instrs:} \; ([\![i]\!]_K^p)^*$$

$$[\![i]\!]_K^p \stackrel{\text{def}}{=} l : \; \langle \texttt{activeCnt} \rangle_K^p \; [\![\texttt{s}]\!]_K^p \; \langle \texttt{closeCnt} \rangle_K^p$$

$$[\![\$r \leftarrow exp]\!]_K^p \stackrel{\text{def}}{=} [\![\$r \leftarrow exp]\!]_K^{\texttt{p,Assign}}$$

$$[\![\texttt{LD} \;\; \$r' \leftarrow [\$r]]\!]_K^p \stackrel{\text{def}}{=} [\![\texttt{LD} \;\; \$r' \leftarrow [\$r]]\!]_K^{\texttt{p,LD}}$$

$$[\![\texttt{ST} \;\; [\$r'] \leftarrow \$r]\!]_K^p \stackrel{\text{def}}{=} [\![\texttt{ST} \;\; [\$r'] \leftarrow \$r]\!]_K^{\texttt{p,ST}}$$

$$[\![\texttt{LDA } \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LDA } \$r' \leftarrow [\$r]]\!]^{\texttt{p,LDA}}_K$$

$$[\![\texttt{STL } [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{STL } [\$r'] \leftarrow \$r]\!]^{\texttt{p,STL}}_K$$

$$[\![\texttt{LDX } \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LDX } \$r' \leftarrow [\$r]]\!]^{\texttt{p,LDX}}_K$$

$$[\![\texttt{STX } \$r'', [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{STX } \$r'', [\$r'] \leftarrow \$r]\!]^{\texttt{p,STX}}_K$$

$$[\![\texttt{LDAX } \$r'', \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LDAX } \$r' \leftarrow [\$r]]\!]^{\texttt{p,LDAX}}_K$$

$$[\![\texttt{STLX } [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{STLX } \$r'', [\$r'] \leftarrow \$r]\!]^{\texttt{p,STLX}}_K$$

$$[\![\texttt{if } exp \texttt{ then } i^* \texttt{ else } i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{if } exp \texttt{ then } ([\![i]\!]^p_K)^* \texttt{ else } ([\![i]\!]^p_K)^*;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{while } exp \texttt{ do } i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{while } exp \texttt{ do } ([\![i]\!]^p_K)^*;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assume } exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assume } exp;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assert } exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assert } exp;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{acquire \$r} \leftarrow [exp]]\!] \stackrel{\text{def}}{=} [\![\$r \leftarrow [exp]]\!]^{\texttt{p,LAcq}}_K$$

$$[\![\texttt{release [exp']} \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![[exp'] \leftarrow exp]\!]^{\texttt{p,SRel}}_K$$

$$[\![\texttt{dmb.ld}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{dmb.ld}]\!]^{\texttt{p,DmbLd}}_K$$

$$[\![\texttt{dmb.st}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{dmb.st}]\!]^{\texttt{p,DmbSt}}_K$$

$$[\![\texttt{dmb.sy}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{dmb.sy}]\!]^{\texttt{p,DmbSy}}_K$$

$$[\![\texttt{isb}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{isb}]\!]^{\texttt{p,isb}}_K$$

$$[\![\texttt{term}]\!]^p_K \stackrel{\text{def}}{=} \texttt{term}$$

$$\begin{aligned}\langle \texttt{addvars} \rangle_K \stackrel{\text{def}}{=} \mu(|\mathcal{X}|, K), \mu^{init}(|\mathcal{X}|, K), \nu(|\mathcal{P}|, |\mathcal{X}|),\\ \delta(|\mathcal{X}|, K), \delta^{init}(|\mathcal{X}|, K),\\ \texttt{iR}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cR}(|\mathcal{P}|, |\mathcal{X}|),\\ \texttt{iW}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cW}(|\mathcal{P}|, |\mathcal{X}|),\\ \texttt{iReg}(|\mathcal{P}|, |\mathcal{R}|), \texttt{cReg}(|\mathcal{P}|, |\mathcal{R}|),\\ \texttt{iL}(|\mathcal{P}|, |\mathcal{R}|), \texttt{cL}(|\mathcal{P}|, |\mathcal{R}|),\\ \texttt{iS}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cS}(|\mathcal{P}|, |\mathcal{X}|),\\ \texttt{cDY}(|\mathcal{P}|), \texttt{cDL}(|\mathcal{P}|), \texttt{cDS}(|\mathcal{P}|), \texttt{cISB}(|\mathcal{P}|),\\ \texttt{ctrl}(|\mathcal{P}|), \texttt{iAddr}(|\mathcal{P}|), \texttt{active}(K), cnt\end{aligned}$$

$$\langle \texttt{activeCnt} \rangle^p_K \stackrel{\text{def}}{=} \texttt{assume}(\texttt{active}(\texttt{cnt}) = p)$$

$$\begin{aligned}\langle \texttt{closeCnt} \rangle^p_K \stackrel{\text{def}}{=} cnt \leftarrow cnt + \texttt{gen}([0, \cdots, K-1]);\\ \texttt{assume}(cnt \leq K)\end{aligned}$$

$$\begin{aligned}\langle \texttt{control} \rangle^p_K \stackrel{\text{def}}{=} \texttt{ctrl}(p) \leftarrow \texttt{ctrl}(p) + \texttt{gen}(0, \cdots, K-1);\\ \texttt{assume}(\texttt{ctrl}(p) \leq K)\end{aligned}$$

11.2 Additional data structures for the complete model

We use all of the data structures described in the section of the simple model, with a few minor changes. One, the set $\mathcal{X}$ now refers to the set of shared memory locations instead of shared variables. Further, functions such as `cR` are updated by all four types of reads, and similar conditions hold for the other three analogous functions. Since some predicates in this model are over just load-acquires or store-releases, we also define the predicates:

$$\texttt{iL} : \mathcal{P} \times \mathcal{R} \mapsto [1, \cdots, K]$$

$$\texttt{cL} : \mathcal{P} \times \mathcal{R} \mapsto [1, \cdots, K]$$

$$\texttt{iS} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{cS} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{cDY} : \mathcal{P} \mapsto [1, \cdots, K]$$

$$\texttt{cDS} : \mathcal{P} \mapsto [1, \cdots, K]$$

$$\texttt{cDL} : \mathcal{P} \mapsto [1, \cdots, K]$$

$$\texttt{cISB} : \mathcal{P} \mapsto [1, \cdots, K]$$

The first four describe, for each process-register or process-variable pair, the initializing and commiting contexts of the latest `STL` or instruction that loads to, and the latest `SRel` instruction that stores from that register/to that variable. There is however one change that we must keep in mind: now the set $\mathcal{X}$ consists of addressable memory regions (as opposed to variables). Also note that since exclusive stores directly commit, we use their commiting context to also update `iW` and `iL`, since later instructions need them to be "at least" initialized, and so we need these two functions to take into account their committing (it's as if they initialized and spontaneously committed). Also, noting that the predicates on `LAcq` or `SRel` that appear in our rules, such as `AllLAcqs`, are not about those to a particular variable but over all of them, it will be helpful to define for $p \in \mathcal{P}$:

$$\texttt{iS}(p) = \texttt{max}_{x \in \mathcal{X}} \texttt{iS}(p, x)$$

We similarly define $\texttt{iL}(p)$, $\texttt{cL}(p)$ and $\texttt{cS}(p)$. The last four are the analogues for the four types of barrier instructions respectively: `dmb.sy`, `dmb.st`, `dmb.ld` and `isb`.

We shall also need to track the tags of memory locations. To this end, we introduce two new data structures, $\delta^{init} : \mathcal{X} \times [2, \cdots, K] \mapsto \mathcal{P} \cup \{\perp\}$, and $\delta : \mathcal{X} \times [1, \cdots, K] \mapsto \mathcal{P} \cup \{\perp\}$. These give the value of the tag of a memory location in a particular context: the first at the beginning of the context, and the second at any given moment in time (i.e. it gets updated as more and more transitions are taken in the corresponding context). During implementation, we can use, e.g., 0 to represent $\perp$.

With these structures defined, we now elaborate upon the components of the code-to-code translation above. Below, when we refer to a register in the pseudocode, such as $\$r$, we are actually referring to the *value* stored in the register. Thus, for example, $\texttt{cW}(p, [\$r])$ refers to the context of latest write-commit that writes to the location whose address is given by the then-present-value in $\$r$.

Finally, we need to track, for the sake of `isb` instructions, the latest initializing context of any instruction that supplies `addr`-data to a read or write: this is because when an `isb` is committed we need all `po`-preceeding instructions to have well defined memory footprints. So, we define

$$\texttt{iAddr} : \mathcal{P} \mapsto [1, \cdots, K]$$

which gives for each process P, the maximum initializing context of an instruction that supplies the address to another instruction seen so far.

11.3 The initializing process

Algorithm 6: Algorithm $\langle \texttt{initProc} \rangle_K$.

```
1  for p ∈ 𝒫 do
2      for x ∈ 𝒳 do
3          iR(p, x) ← 1;
4          iW(p, x) ← 1;
5          cW(p, x) ← 1;
6          cR(p, x) ← 1;
7          ν(p, x) ← 0;
8          iS(p, x) ← 1;
9          cS(p, x) ← 1;
10     end
11     for $r ∈ ℛ do
12         iL(p, r) ← 1; cL(p, r) ← 1;
13         iS(p, r) ← 1; cS(p, r) ← 1;
14         iReg(p, $r) ← 1;
15         cReg(p, $r) ← 1;
16     end
17     ctrl(p) ← 1;
18     iAddr(p) ← 1;
19     cDY(p) ← 1;
20     cDS(p) ← 1;
21     cDL(p) ← 1;
22 end
23 for k ∈ [1, ··· , K] do
24     active(k) ← gen(𝒫);
25     for x ∈ 𝒳 do
26         if k ≠ 1 then
27             μ^init(x, k) ← ⊥;
28             δ^init(x, k) ← ⊥;
29         end
30         μ(x, k) ← ⊥;
31         δ(x, k) ← ⊥;
32     end
33 end
34 cnt ← 1;
```

The algorithm is largely the same as before. We add statements that initialize the values of the new structures defined above to 1.

Implementation detail. In case certain registers and/or memory contents are to be set to initial values, the statements for doing so are added at the end of `initProc` as well. Further, we can use -1 as a representative for $\perp$ in the implementation.

11.4 Write Statements

First we show the algorithm for ST statements.

Algorithm 7: $[\![\texttt{ST}\ \ [\$r'] \leftarrow \$r]\!]_K^{p,\texttt{ST}}$

```
1  //Guess
2  iW(p, [$r']) ← gen([1, ··· , K]);
3  old-cW ← cW(p, [$r']);
4  cW(p, [$r']) ← gen([1, ··· , K]);
5  //Check
6  assume(iW(p, [$r']) ≥ cnt);
7  assume(active(iW(p, [$r'])) = p);
8  assume(iW(p, [$r']) ≥ max(iReg(p, $r'), iReg(p, $r)));
9  assume(cW(p, [$r']) ≥ iW(p, [$r']));
10 assume(iW(p, [$r']) ≥ max(cDY(p), cISB(p)));
11 assume(iW(p, [$r']) ≥ max(cDS(p), cDL(p)));
12 for $r ∈ R do
13 |  assume(iW(p, [$r']) ≥ cL(p, $r));
14 end
15 assume(active(cW(p, [$r'])) = p);
16 assume(cW(p, [$r']) ≥ old-cW);
17 assume(cW(p, [$r']) ≥ max(cReg(p, $r), iReg(p, $r')));
18 assume(cW(p, [$r']) ≥ cR(p, [$r']));
19 assume(cW(p, [$r']) ≥ max(iAddr(p), ctrl(p)));
20 //Update
21 iAddr(p) ← max(iAddr(p), iReg(p, $r'));
22 μ([$r'], cW(p, [$r'])) ← $r;
23 ν(p, [$r']) ← $r;
24 δ([$r'], cW(p, [$r'])) ← ⊥;
```

We first guess the initializing and committing contexts of the ST in lines 1-3. Then, we run it through a bunch of sanity checks. Line 6 ensures that the write initializes only after being fetched (which happens at context *cnt*). At lines 7 and 15 we check that the process p is active during init and commit, respectively. Then we need that the two register values are ready, i.e. the events providing them have been initialized, at the moment when the event is initialized: this is expressed by line 8. Line 9 posits that the instruction is committed no sooner than it is initialized, and lines 10 and 11 ensure that all po-preceeding barriers are committed. We also need all po-previous load-acquires to have been initalized no later than this write is: that is checked in line 13. We then ensure the in-order commit of poloc-related instructions by comparing with old-cW and cR(...) in lines 16 and 18. To commit a write (or read) event, we need that all data suppliants are committed (i.e. already globally visible), but address suppliants only need be locally visible ("initialized"). This leads to line 17. Finally, line 19 checks that 1) any po-preceding ACI instruction has been committed, and 2) all po-previous memory-access instructions have fully determined addresses ("memory footprints"). Once all the checks pass, we update the respective data structures in lines 21 through 24.

We next present the other three types of writes. Since the changes are minor, we only highlight the differences in text.

Algorithm 8: $[\![\texttt{STL}\ \ [\$r'] \leftarrow \$r]\!]_K^{p,\texttt{STL}}$

1 //Guess
2 $\texttt{iW}(p, [\$r']) \leftarrow \texttt{gen}([1, \cdots, K])$;
3 $\texttt{old-cW} \leftarrow \texttt{cW}(p, [\$r'])$;
4 $\texttt{cW}(p, [\$r']) \leftarrow \texttt{gen}([1, \cdots, K])$;
5 //Check
6 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq cnt)$;
7 $\texttt{assume}(\texttt{active}(\texttt{iW}(p, [\$r'])) = p)$;
8 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{iReg}(p, \$r'), \texttt{iReg}(p, \$r)))$;
9 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{iW}(p, [\$r']))$;
10 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cDY}(p), \texttt{cISB}(p)))$;
11 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cDS}(p), \texttt{cDL}(p)))$;
12 **for** $x \in \mathcal{X}$ **do**
13 | $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cR}(p, x), \texttt{cW}(p, x)))$;
14 **end**
15 $\texttt{assume}(\texttt{active}(\texttt{cW}(p, [\$r'])) = p)$;
16 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{old-cW})$;
17 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{max}(\texttt{cReg}(p, \$r), \texttt{iReg}(p, \$r')))$;
18 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{cR}(p, [\$r']))$;
19 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{max}(\texttt{iAddr}(p), \texttt{ctrl}(p)))$;
20 //Update
21 $\texttt{iAddr}(p) \leftarrow \texttt{max}(\texttt{iAddr}(p), \texttt{iReg}(p, \$r'))$;
22 $\mu([\$r'], \texttt{cW}(p, [\$r'])) \leftarrow \$r$;
23 $\nu(p, [\$r']) \leftarrow \$r$;
24 $\delta([\$r'], \texttt{cW}(p, [\$r'])) \leftarrow \bot$;

The difference of this from the previous listing is only that instead of just load-acquires, we need all memory-access instructions that precede this one in program order to have been committed: this change results in line 13.

Algorithm 9: $[\![\texttt{STX}\ \ \$r'',\ [\$r'] \leftarrow \$r]\!]_K^{p,\texttt{STX}}$

1 //Guess
2 $\texttt{old-cW} \leftarrow \texttt{cW}(p, [\$r'])$;
3 $\texttt{new-cW} \leftarrow \texttt{gen}([1, \cdots, K])$;
4 //Check
5 $\texttt{assume}(\texttt{new-cW} \geq cnt)$;
6 $\texttt{assume}(\texttt{active}(\texttt{new-cW}) = p)$;
7 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cDY}(p), \texttt{cISB}(p)))$;
8 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cDS}(p), \texttt{cDL}(p)))$;
9 **for** $\$r \in \mathcal{R}$ **do**
10 | $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{cL}(p, \$r))$;
11 **end**
12 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{old-cW}, \texttt{cR}(p, [\$r'])))$;
13 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cReg}(p, \$r), \texttt{iReg}(p, \$r')))$;
14 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{iAddr}(p), \texttt{ctrl}(p)))$;
15 //Update
16 $\texttt{iAddr}(p) \leftarrow \texttt{max}(\texttt{iAddr}(p), \texttt{iReg}(p, \$r'))$;
17 **if** $\delta([\$r'], \texttt{new-cW}) = p$ **then**
18 | $\mu([\$r'], \texttt{new-cW}) \leftarrow \$r$;
19 | $\nu(p, [\$r']) \leftarrow \$r$;
20 | $\delta([\$r'], \texttt{new-cW}) \leftarrow \bot$;
21 | $\$r'' \leftarrow 0$;
22 | $\texttt{cW}(p, [\$r']) \leftarrow \texttt{new-cW}$;
23 | $\texttt{iW}(p, [\$r']) \leftarrow \texttt{cW}(p, [\$r'])$;
24 **else**
25 | $\$r'' \leftarrow 1$;
26 **end**
27 $\texttt{cReg}(p, \$r'') \leftarrow \texttt{new-cW}$;
28 $\texttt{iReg}(p, \$r'') \leftarrow \texttt{cReg}(p, \$r'')$;

This listing is still very similar to the first one: the difference is that all changes (except those to $\$r''$ and iAddr) are predicated on the "success" of the store-exclusive instruction. The same difference applied to the second listing leads to the one be-

Algorithm 10: $[\![\texttt{STLX}\ \$r'',\ [\$r']\leftarrow \$r]\!]_K^{p,\texttt{STLX}}$

```
1  //Guess
2  old-cW ← cW(p, [$r']);
3  new-cW ← gen([1, ··· , K]);
4  //Check
5  assume(new-cW ≥ cnt);
6  assume(active(new-cW) = p);
7  assume(new-cW ≥ max(cDY(p), cISB(p)));
8  assume(new-cW ≥ max(cDS(p), cDL(p)));
9  for $r ∈ R do
10 |  assume(iW(p, [$r']) ≥ max(cR(p, x), cW(p, x)));
11 end
12 assume(new-cW ≥ max(old-cW, cR(p, [$r']));
13 assume(new-cW ≥ max(cReg(p, $r), iReg(p, $r'));
14 assume(new-cW ≥ max(iAddr(p), ctrl(p));
15 //Update
16 iAddr(p) ← max(iAddr(p), iReg(p, $r'));
17 if δ([$r'], new-cW) = p then
18 |  μ([$r'], new-cW) ← $r;
19 |  ν(p, [$r']) ← $r;
20 |  δ([$r'], new-cW) ← ⊥;
21 |  $r'' ← 0;
22 |  cW(p, [$r']) ← new-cW;
23 |  iW(p, [$r']) ← cW(p, [$r']);
24 else
25 |  $r'' ← 1;
26 end
27 cReg(p, $r'') ← new-cW;
28 iReg(p, $r'') ← cReg(p, $r'');
```

low.

Implementation detail. For write (and read) instructions with offsets, we reserve a private register. We first `assign` the sum of the base and offset to this private register followed by one of the above listings, in order to replicate the effect of a write-to-offset style instruction. This preserves correctness since, for an assign instruction, the only condition is that it is not initialized(committed) before any of its operands are initialized(committed): hence, every execution trace valid without this hack still remains so.

11.5 Assign Instructions

Algorithm 11: $[\![\$r \leftarrow exp]\!]_K^{p,\texttt{Assign}}$

```
1  //Guess
2  iReg(p, $r) ← gen([1, ··· , K]);
3  cReg(p, $r) ← gen([1, ··· , K]);
4  //Check
5  assume(iReg(p, $r) ≥ cnt);
6  assume(active(iReg(p, $r)) = p);
7  for $r' ∈ R(exp) do
8  |  assume(iReg(p, $r) ≥ iReg(p, $r'));
9  end
10 assume(active(cReg(p, $r)) = p);
11 assume(cReg(p, $r) ≥ iReg(p, $r));
12 for $r' ∈ R(exp) do
13 |  assume(cReg(p, $r) ≥ cReg(p, $r'));
14 end
15 assume(cReg(p, $r) ≥ ctrl(p));
16 //Update
17 $r ← exp;
```

Assign instructions are simple: the only condition we have on them is that they cannot `init`/`commit` before every one of their operands (which are part of exp) have done so, as expressed in lines 8 and 13.

11.6 Read Instructions

We show, in order, the listings for LD, LDA, LDX and LDAX respectively.

Algorithm 12: $[\![\texttt{LD}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LD}}$

1 //Guess
2 $\texttt{iR}(p, [\$r]) \leftarrow \texttt{gen}([1, \cdots, K])$;
3 $\texttt{old-cR} \leftarrow \texttt{cR}(p, [\$r])$;
4 $\texttt{cR}(p, [\$r]) \leftarrow \texttt{gen}([1, \cdots, K])$;
5 $\texttt{iReg}(p, \$r') \leftarrow \texttt{iR}(p, [\$r])$;
6 $\texttt{cReg}(p, \$r') \leftarrow \texttt{cR}(p, [\$r])$;
7 //Check
8 $\texttt{assume}(\texttt{iR}(p, [\$r]) \geq cnt)$;
9 $\texttt{assume}(\texttt{active}(\texttt{iR}(p, [\$r])) = p)$;
10 $\texttt{assume}(\texttt{iR}(p, [\$r]) \geq \texttt{iW}(p, [\$r]))$;
11 $\texttt{assume}(\texttt{iR}(p, [\$r]) \geq \texttt{iReg}(p, \$r))$;
12 $\texttt{assume}(\texttt{iR}(p, [\$r]) \geq \texttt{max}(\texttt{cDY}(p), \texttt{cISB}(p)))$;
13 $\texttt{assume}(\texttt{iR}(p, [\$r]) \geq \texttt{cDL}(p))$;
14 **for** $\$r'' \in \mathcal{R}$ **do**
15 | $\texttt{assume}(\texttt{iR}(p, [\$r]) \geq \texttt{iL}(p, \$r''))$;
16 **end**
17 $\texttt{assume}(\texttt{cR}(p, [\$r]) \geq \texttt{iR}(p, [\$r]))$;
18 $\texttt{assume}(\texttt{active}(\texttt{cR}(p, [\$r])) = p)$;
19 $\texttt{assume}(\texttt{cR}(p, [\$r]) \geq \texttt{max}(\texttt{old-cR}, \texttt{cW}(p, [\$r])))$;
20 $\texttt{assume}(\texttt{cR}(p, [\$r]) \geq \texttt{max}(\texttt{iReg}(\$r), \texttt{ctrl}(p)))$;
21 //Update
22 $\texttt{iAddr}(p) \leftarrow \texttt{max}(\texttt{iAddr}(p), \texttt{iReg}(p, \$r))$;
23 **if** $iR(p, [\$r]) < cW(p, [\$r])$ **then**
24 | $\$r' \leftarrow \nu(p, [\$r])$;
25 **else**
26 | $\$r' \leftarrow \mu([\$r], \texttt{iR}(p, [\$r]))$;
27 **end**

The conditions for read instructions, apart from the obvious ones about being initialized after being fetched, committed only after initialized, and process p being active during these transitions, are: 1) we need all dmb.sy's, isb's and dmb.ld's that precede this instruction in program order, to have been committed before it is initialized, as lines 12-13 describe. We also need all events supplying the address to have been initialized before it, as in line 11. For committing, we need all poloc-preceeding memory-access events to have committed: this follows from line 19. We also need the address to be well known during commit, and that all po-previous ACI instructions are committed: this is handled by line 20. Once the sanity checks are done, we update the required data structures to indicate that the LD instruction has committed.
We next present the listings for the other three types of reads. The modifications as compared to the above listing are minor and obvious, so we do not elaborate upon them.

Algorithm 13: $[\![\texttt{LDA}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LDA}}$

```
//Guess
iR(p, [$r]) ← gen([1, ··· , K]);
old-cR ← cR(p, [$r]);
cR(p, [$r]) ← gen([1, ··· , K]);
iReg(p, $r′) ← iR(p, [$r]);
cReg(p, $r′) ← cR(p, [$r]);
iL(p, $r′) ← iR(p, [$r]);
cL(p, $r′) ← cR(p, [$r]);
//Check
assume(iR(p, [$r]) ≥ cnt);
assume(active(iR(p, [$r])) = p);
assume(iR(p, [$r]) ≥ iW(p, [$r]));
assume(iR(p, [$r]) ≥ iReg(p, $r));
assume(iR(p, [$r]) ≥ max(cDY(p), cISB(p));
assume(iR(p, [$r]) ≥ cDL(p));
for $r″ ∈ R do
    assume(iR(p, [$r]) ≥ iL(p, $r″));
end
for x ∈ X do
    assume(iR(p, [$r]) ≥ cS(p, x));
end
assume(cR(p, [$r]) ≥ iR(p, [$r]));
assume(active(cR(p, [$r])) = p);
assume(cR(p, [$r]) ≥ max(old-cR, cW(p, [$r]));
assume(cR(p, [$r]) ≥ max(iReg($r), ctrl(p));
//Update
iAddr(p) ← max(iAddr(p), iReg(p, $r));
if iR(p, [$r]) < cW(p, [$r]) then
    $r′ ← ν(p, [$r]);
else
    $r′ ← μ([$r], iR(p, [$r]));
end
```

Algorithm 14: $[\![\texttt{LDX}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LDX}}$

```
//Guess
iR(p, [$r]) ← gen([1, ··· , K]);
old-cR ← cR(p, [$r]);
cR(p, [$r]) ← gen([1, ··· , K]);
iReg(p, $r′) ← iR(p, [$r]);
cReg(p, $r′) ← cR(p, [$r]);
//Check
assume(iR(p, [$r]) ≥ cnt);
assume(active(iR(p, [$r])) = p);
assume(iR(p, [$r]) ≥ iW(p, [$r]));
assume(iR(p, [$r]) ≥ iReg(p, $r));
assume(iR(p, [$r]) ≥ max(cDY(p), cISB(p)));
assume(iR(p, [$r]) ≥ cDL(p));
for $r″ ∈ R do
    assume(iR(p, [$r]) ≥ iL(p, $r″));
end
assume(cR(p, [$r]) ≥ iR(p, [$r]));
assume(active(cR(p, [$r])) = p);
assume(cR(p, [$r]) ≥ max(old-cR, cW(p, [$r])));
assume(cR(p, [$r]) ≥ max(iReg($r), ctrl(p)));
//Update
iAddr(p) ← max(iAddr(p), iReg(p, $r));
if iR(p, [$r]) < cW(p, [$r]) then
    $r′ ← ν(p, [$r]);
else
    $r′ ← μ([$r], iR(p, [$r]));
end
δ([$r], cR(p, [$r])) ← p;
```

Algorithm 15: $[\![\texttt{LDAX}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LDAX}}$

```
1  //Guess
2  iR(p, [$r]) ← gen([1, ··· , K]);
3  old-cR ← cR(p, [$r]);
4  cR(p, [$r]) ← gen([1, ··· , K]);
5  iReg(p, $r') ← iR(p, [$r]);
6  cReg(p, $r') ← cR(p, [$r]);
7  iL(p, $r') ← iR(p, [$r]);
8  cL(p, $r') ← cR(p, [$r]);
9  //Check
10 assume(iR(p, [$r]) ≥ cnt);
11 assume(active(iR(p, [$r])) = p);
12 assume(iR(p, [$r]) ≥ iW(p, [$r]));
13 assume(iR(p, [$r]) ≥ iReg(p, $r));
14 assume(iR(p, [$r]) ≥ max(cDY(p), cISB(p)));
15 assume(iR(p, [$r]) ≥ cDL(p));
16 for $r'' ∈ R do
17 |  assume(iR(p, [$r]) ≥ iL(p, $r''));
18 end
19 for x ∈ X do
20 |  assume(iR(p, [$r]) ≥ cS(p, x));
21 end
22 assume(cR(p, [$r]) ≥ iR(p, [$r]));
23 assume(active(cR(p, [$r])) = p);
24 assume(cR(p, [$r]) ≥ max(old-cR, cW(p, [$r])));
25 assume(cR(p, [$r]) ≥ max(iReg($r), ctrl(p)));
26 //Update
27 iAddr(p) ← max(iAddr(p), iReg(p, $r));
28 if iR(p, [$r]) < cW(p, [$r]) then
29 |  $r' ← ν(p, [$r]);
30 else
31 |  $r' ← μ([$r], iR(p, [$r]));
32 end
33 δ([$r], cR(p, [$r])) ← p;
```

Implementation detail. The procedure for offset-ed reads is the same as that for offset-ed writes: see the corresponding subsection for details.

11.7 Barrier Instructions

Algorithm 16: $[\![\texttt{dmb.sy}]\!]_K^{p,\texttt{DmbSy}}$

```
1  //Guess
2  old-cDY ← cDY(p);
3  cDY(p) ← gen[1, ··· , K];
4  //Check
5  assume(cDY(p) ≥ cnt);
6  assume(cDY(p) ≥ max(old-cDY, cISB(p)));
7  assume(cDY(p) ≥ max(cDL(p), cDS(p)));
8  assume(cDY(p) ≥ ctrl(p));
9  for x ∈ X do
10 |  assume(cDY(p) ≥ cW(p, x));
11 |  assume(cDY(p) ≥ cR(p, x));
12 end
```

We first guess the context in which the `dmb.sy` will be committed, and then run it through some sanity checks. The `AllBarriers` predicate is spread across lines 6 and 7. The `AllMem` predicate appears in lines 6-8. Line 8 ensures that all po-previous ACI instructions have been committed.
Next up is the ISB instruction:

Algorithm 17: $[\![\texttt{isb}]\!]_K^{p,\texttt{Isb}}$

```
1 //Guess
2 cISB(p) ← gen[1, ··· , K];
3 //Check
4 assume(cISB(p) ≥ cnt);
5 assume(cISB(p) ≥ max(cDY(p), ctrl(p)));
6 assume(cISB(p) ≥ iAddr(p));
```

For an ISB instruction, at the time of commit we need that all po-preceding dmb.sy and ACI instructions are committed; and also that all po-preceding memory access instructions have fully defined addresses: these conditions are a part of lines 5,5 and 6 respectively.

Algorithm 18: $[\![\texttt{dmb.ld}]\!]_K^{p,\texttt{DmbLd}}$

```
1 //Guess
2 cDL(p) ← gen[1, ··· , K];
3 //Check
4 assume(cDL(p) ≥ cnt);
5 assume(cDL(p) ≥ max(cDY(p), ctrl(p));
6 for x ∈ 𝒳 do
7 |  assume(cDY(p) ≥ cR(p, x));
8 end
```

Again, for a dmb.ld we first guess the committing context for it, in line 2. We then check that it does not commit earlier than the current context. Then, we run it through the AllDmbSys and AllReads predicates. The first is in line 4, and the second is in line 7. The ctrl(p) comes from the ComCnd predicate.

Algorithm 19: $[\![\texttt{dmb.st}]\!]_K^{p,\texttt{DmbSt}}$

```
1 //Guess
2 cDS(p) ← gen[1, ··· , K];
3 //Check
4 assume(cDS(p) ≥ cnt);
5 assume(cDS(p) ≥ max(cDY(p), ctrl(p)));
6 for x ∈ 𝒳 do
7 |  assume(cDY(p) ≥ cW(p, x));
8 end
```

11.8 ACI statements

As shown in the translation schemes, the ACI statements themselves are retained as if-then-else C-statements. However, apart from the ⟨control⟩, we also need the condition that control(p) is greater than or equal to the initializing context of the register it depends on (not committing). Thus, for example, if we have a CBNZ instruction:

```
CBNZ $r, label
```

Then we need that after modification, ctrl$(p) \geq$ iReg$(p, \$r)$.
Implementation detail. For instructions such as BEQ which follow a CMP, we do the following: during the CMP we simply assign the two operand-registers to two fixed private registers of the process, and consider BEQ and other such instructions as ACI instructions that use these fixed private registers.

11.9 Verifying process

Algorithm 20: $\langle \texttt{verProc} \rangle_K$.

```
1 for ∧x ∈ 𝒳 ∧ k ∈ [1, ··· , K − 1] do
2 |  assume(μ(x, k) = μ^init(x, k + 1));
3 |  assume(δ(x, k) = δ^init(x, k + 1));
4 end
5 if l is reachable then
6 |  error;
7 end
```

The verifying process is the same as in section 10: it simply ensures that each context leaves the global state in a consistent state for the next one.
Implementation detail. The checking for the final conditions (for which we are to determine reachability) are the contents of line 5.