ARM Memory Rules

February 2021

1 Syntax for programs (without synchronization instructions)

GP: We should start defining macros for most of the notations.

We use the following syntax for the program.

$$
\begin{aligned}
\mathit{Prog} ::= \ & \texttt{vars:} \ \ x^* \\
& \texttt{procs:} \ \ p^* \\
p ::= \ & \texttt{regs:} \ \ \$r^* \\
& \texttt{instrs:} \ \ i^* \\
i ::= \ & l : \texttt{s} \\
\texttt{s} ::= \ & x \leftarrow \mathit{exp} \\
& \mid \$r \leftarrow x \\
& \mid \$r \leftarrow \mathit{exp} \\
& \mid \texttt{if } \mathit{exp} \texttt{ then } i^* \texttt{ else } i^* \\
& \mid \texttt{while } \mathit{exp} \texttt{ do } i^* \\
& \mid \texttt{assume } \mathit{exp} \\
& \mid \texttt{assert } \mathit{exp} \\
& \mid \texttt{term}
\end{aligned}
$$

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

GP: The configurations on the left-hand-side $\mathbb{c}$ and the right-hand-side are not bound to each other. I think there is an implicit hypothesis throughout that $\mathbb{c} =< E, ... >$. This has to be stated somewhere.
AB: I have now added 2 lines explaining $\mathbb{c}$.
GP: I understand the intuition, but I'm not sure I understand the formalization of ins. Is it a mapping from events to a (set of) immediately subsequent instruction(s) in PO?
AB: ins just maps events to the instructions they represent (since an event is just one instance of an instruction). Maybe it can be removed from the formalization though, as it doesn't seem to be used further anywhere...
In this set of rules, we do not include the dmb or load/acquire instructions. Note that in this set of rules, unlike the POWER paper, Prop is a map from variables (and not a process-variable pair) to the set of values. 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.

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

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

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

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

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

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

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

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

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

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 $\mathbb{c}$. To do this, we define the function $\texttt{MaxI}(\mathbb{c}, p)$ to be the set of instructions that can (intuitively) follow the current one. Formally,

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

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 $\mathbb{e}$ to show that it has been 'initialized', and update `rf` to indicate the instruction that satisfied it.
- Once all `po`-previous instructions on which $\mathbb{e}$ 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 $\mathbb{e}$ 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 $\mathbb{e}$ depends upon must have already been committed. This is captured as the rule `ComACI`.

5 Syntax for programs (with synchronization instructions)

We add load and store operations, along with synchronization primitives. Note that loads and stores are still referred to as writes and reads, since they follow the same rules, except that they are also bound by data dependencies.

$$
\begin{aligned}
Prog ::= \ & \texttt{vars:} \ \ x^* \\
& \texttt{procs:} \ \ p^* \\
p ::= \ & \texttt{regs:} \ \ \$r^* \\
& \texttt{instrs:} \ \ i^* \\
i ::= \ & l : \texttt{s} \\
\texttt{s} ::= \ & x \leftarrow exp \\
& \mid \$r \leftarrow x \\
& \mid \$r \leftarrow exp \\
& \mid [exp'] \leftarrow exp \\
& \mid \$r \leftarrow [exp] \\
& \mid \texttt{if } exp \texttt{ then } i^* \texttt{ else } i^* \\
& \mid \texttt{while } exp \texttt{ do } i^* \\
& \mid \texttt{assume } exp \\
& \mid \texttt{assert } exp \\
& \mid \texttt{acquire \$r} \leftarrow [exp] \\
& \mid \texttt{release [exp']} \leftarrow exp \\
& \mid \texttt{dmb.ld} \\
& \mid \texttt{dmb.st} \\
& \mid \texttt{dmb.sy} \\
& \mid \texttt{isb} \\
& \mid \texttt{term}
\end{aligned}
$$

6 Helper predicates for next section

Note that we introduce events corresponding to Dmb.Sy's, Dmb.St's, Dmb.Ld's, Isb's, Load Acquires and Store Releases with the corresponding $\mathbb{E}^{\texttt{name}}$ being the subset of $\mathbb{E}$ corresponding to `name` type of instructions. GP: Many of the predicates are identical but for the set of events. I would define a parametric predicate instead of repeating them many times.

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.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllWrites}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{W} \cup \mathbb{E}^{\texttt{SRel}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous writes (including Store Releases) have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllReads}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{R} \cup \mathbb{E}^{\texttt{LAcq}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous reads (including Load Acquires) 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.
$\mathbb{e} \in \mathbb{E}^{\texttt{LAcq}}$: $\texttt{AllSRels}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e} \in \mathbb{E}^{\texttt{SRel}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous Store Releases have been commited.
$\mathbb{e} \in \mathbb{E}^{R} \cup \mathbb{E}^{\texttt{LAcq}}$: $\texttt{AllLAcqInit}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LAcq}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com})))$	All po-previous Load Acquires have been initialized.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllLAcqs}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LAcq}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous Load Acquires have been committed.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllStoreInit}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{W} \cup \mathbb{E}^{\texttt{SRel}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com})))$	All po-previous write instructions (including Store Releases) have been initialized
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllLoadInit}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{R} \cup \mathbb{E}^{\texttt{LAcq}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com})))$	All po-previous read instructions (including Load Acquires) have been initialized
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllMemInit}(\mathbb{c}, \mathbb{e})$	$\texttt{AllLoadInit}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllStoreInit}(\mathbb{c}, \mathbb{e})$	All po-previous instructions that access memory have been initialized.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllMem}(\mathbb{c}, \mathbb{e})$	$\texttt{AllWrites}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllReads}(\mathbb{c}, \mathbb{e})$	All po-previous instructions 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.
$\mathbb{e} \in \mathbb{E}^{W} \cup \mathbb{E}^{A}$: $\texttt{InitCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{R} \cup \mathbb{E}^{A}$: $(\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All instructions on which $\mathbb{e}$ is dependent on have already been initialized.

7 Rules with synchronization conditions

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

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

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

$$\mathbb{e} \in \mathbb{E}^{R}, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{AllSyncs}(\mathbb{c}, \mathbb{e}),$$

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

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

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

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

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

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

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

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

AB: The `ComIsb` rule is somehow wrong. In the Flat model they say "all po-previous memory instructions have fully defined memory footprints. This corresponds just to `fetch` in our model [the rule lists `init` instead]. Or does it? what about a dependency of type $r \xrightarrow{addr;po} w \xrightarrow{po} \texttt{isb}$? Here for w's "memory footprint" to be defined do we need r to be `inited`? This could complicate stuff.

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

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

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

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 read 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 a Load Acquire is similar to that of a read, but with the extra condition that all po-previous Store Releases have been committed.
- To initialize a write, along with an `InitCnd` that we had previously, we need the following conditions to hold:
  1. All po-previous Dmb.Sy, Isb, Dmb.Ld, Dmb.St, and Load Acquire instructions have been committed.
  2. All po-previous store (writes and store releases) instructions have been initiated.
- To initialize a store release, the required conditions are almost the same as for writes, with the difference that instead of just load acquires, all po-previous memory access instructions have been committed.
- The condition for committing a write or store release, is similar to before, but with the added condition that all po-previous synchronization instructions(Dmb.Sy and Isb) are committed.
- All ACI and synchronization 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 been initialized.
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 ARM developer manual 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.
AB: There is one hairy point about `Assigns`: are they allowed to be reordered with `dmb.sy/isb`'s? The models say `dmb.sy` only stops memory-access instructions from being reordered with it; and technically an `assign` only goes from registers to registers, so it doesn't access memory...
AB: Also, it may be helpful to verify once that this rule is correct. I just assumed it to be a write to a private variable + read into register merged into a single instruction.

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

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

8 Proof of equivalence to the ARMv8 Model

We give here the proof of equivalence of the above model to the ARMv8 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 $\prec_{\texttt{poloc}}$. Consider any valid execution under our model, and consider two write events $\mathbb{e}_1 \prec_{\texttt{poloc}} \mathbb{e}_2$. Note that $\mathbb{e}_1$ is always committed before $\mathbb{e}_2$, and because reads can read only from po-previous writes, an execution in which $\mathbb{e}_1$ is initialized after $\mathbb{e}_2$ is functionally equivalent to one where $\mathbb{e}_1$ is initialized just before $\mathbb{e}_2$ (because reads that occur po-between the two events will anyway read from $\mathbb{e}_1$ in our model; and for those after $\mathbb{e}_2$ 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. The othersIn our formulation of the syntax of concurrent programs `addr` and `data` dependencies both are part of $\prec_{\texttt{data}}$ (since we represent loads by $\$r \longleftarrow x$ and stores by $x \longleftarrow \$r$). Then this follows 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.

- Atomic-ordered-before relations are preserved. These refer to the ordering between `SRel` and `LAcq` instructions: 1) the per-loc ordering between `LAcq-SRel` pairs must be maintained and 2) the local read successor relation between a `SRel` and a `LAcq` must be preserved. The `AllMem` predicate in `InitSRels` takes care of (1), and `AllSRel` in the `InitLacq`-rules takes care of (2).

- 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 `InitSRel`.

- 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 $\prec_{\mathtt{ob}}$, if $\mathbb{e}_1 \prec_{\mathtt{ob}} \mathbb{e}_2$ 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 $\prec_{\mathtt{ob}}$ 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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 seprately 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 $\texttt{iReg} : \mathcal{R} \mapsto [1, \cdots, K]$ which captures for each register $\$r$ the initializing context of the latest read or assign event loading a value to $\$r$. Similarly $\texttt{cReg} : \mathcal{R} \mapsto [1, \cdots, K]$ captures for each register $\$r$ the commiting context of the latest read or assign event loading a value to $\$r$. We also extend this to expressions, by defining $\texttt{iReg}(exp) = \max\{\texttt{iReg}(\$r) | \$r \in \mathcal{R}(exp)\}$. 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 $\langle \texttt{initProc} \rangle_K$.

```
1  for p ∈ 𝒫 ∧ x ∈ 𝒳 do
2  |  iR(p, x) ← 1;
3  |  iW(p, x) ← 1;
4  |  cW(p, x) ← 1;
5  |  cR(p, x) ← 1;
6  |  ν(p, x) ← 0;
7  |  μ(p, x, 1) ← 0;
8  |  for k ∈ [2, ··· , K] do
9  |  |  μ^init(p, x, k) ← gen(𝒟);
10 |  |  μ(p, x, k) ← μ^init(p, x, k);
11 |  end
12 end
13 for p ∈ 𝒫 do
14 |  ctrl(p) ← 1;
15 end
16 for $r ∈ ℛ do
17 |  iReg(p) ← 1;
18 |  cReg(p) ← 1;
19 end
20 for k ∈ [1, ··· , K] do
21 |  active(k) ← gen([1, ··· , K]);
22 end
23 cnt ← 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(exp));
9  assume(cW(p, x) ≥ iW(p, x));
10 assume(cW(p, x) ≥ max(old-cW, cReg(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($r) ← iR(p, x);
6  cReg($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($r) ← gen([1, ··· , K]);
3  cReg($r) ← gen([1, ··· , K]);
4  //Check
5  assume(iReg($r) ≥ cnt);
6  assume(active(iReg($r)) = p);
7  assume(iReg($r) ≥ iReg(exp));
8  assume(active(cReg($r)) = p);
9  assume(cReg($r) ≥ iReg($r));
10 assume(cReg($r) ≥ max(cReg(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: $\langle \texttt{verProc} \rangle_K$.

```
for p ∈ 𝒫 ∧ x ∈ 𝒳 ∧ k ∈ [1, ··· , K − 1] do
    assume(μ(p, x, k) = μ^init(p, x, k + 1));
end
if l is reachable then
    error;
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 (with synchronization included)

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\texttt{iL}(|\mathcal{P}|,|\mathcal{R}|),\texttt{cL}(|\mathcal{P}|,|\mathcal{R}|),$$
$$\texttt{iS}(|\mathcal{P}|,|\mathcal{X}|),\texttt{cS}(|\mathcal{P}|,|\mathcal{X}|),$$
$$\texttt{cDY}(|\mathcal{P}|),\texttt{cDL}(|\mathcal{P}|),\texttt{cDS}(|\mathcal{P}|),$$
$$\texttt{ctrl}(|\mathcal{P}|),\texttt{active}(K),cnt$$
$$\langle\texttt{activeCnt}\rangle_K^p \stackrel{\text{def}}{=} \texttt{assume}(\texttt{active}(\texttt{cnt}) = p)$$
$$\langle\texttt{closeCnt}\rangle_K^p \stackrel{\text{def}}{=} cnt \leftarrow cnt + \texttt{gen}([0,\cdots,K-1]);\ \texttt{assume}(cnt \leq K)$$
$$\langle\texttt{control}\rangle_K^p \stackrel{\text{def}}{=} \texttt{ctrl}(p) \leftarrow \texttt{ctrl}(p) + \texttt{gen}(0,\cdots,K-1);\ \texttt{assume}(\texttt{ctrl}(p) \leq K)$$

11.2 Additional data structures for the complete model

We use all of the data structures described in the section of the simple model.
AB: `isb` instuructions are not handled yet due to the doubt.
Now, similar to the structures in section 10, we define the mappings

$$\texttt{iL} : \mathcal{P}\times\mathcal{R}\mapsto[1,\cdots,K]$$
$$\texttt{cL} : \mathcal{P}\times\mathcal{R}\mapsto[1,\cdots,K]$$
$$\texttt{iS} : \mathcal{P}\times\mathcal{X}\mapsto[1,\cdots,K]$$
$$\texttt{cS} : \mathcal{P}\times\mathcal{X}\mapsto[1,\cdots,K]$$
$$\texttt{cDY} : \mathcal{P}\mapsto[1,\cdots,K]$$
$$\texttt{cDS} : \mathcal{P}\mapsto[1,\cdots,K]$$
$$\texttt{cDL} : \mathcal{P}\mapsto[1,\cdots,K]$$

The first four describe, for each process-register or process-variable pair, the initializing and commiting contexts of the latest `LAcq` 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 both of variables and addressable memory regions (as opposed to just variables). 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 hepful 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 $\texttt{cS}(p)$. With these structures defined, we now elaborate upon the components of the code-to-code translation above.

11.3 The initializing process

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

```
1  for p ∈ 𝒫 ∧ x ∈ 𝒳 do
2  |  iR(p, x) ← 1;
3  |  iW(p, x) ← 1;
4  |  cW(p, x) ← 1;
5  |  cR(p, x) ← 1;
6  |  ν(p, x) ← 0;
7  |  μ(p, x, 1) ← 0;
8  |  for k ∈ [2, ··· , K] do
9  |  |  μ^init(p, x, k) ← gen(𝒟);
10 |  |  μ(p, x, k) ← μ^init(p, x, k);
11 |  end
12 end
13 for p ∈ 𝒫 do
14 |  ctrl(p) ← 1;
15 |  cDY(p) ← 1; cDS(p) ← 1; cDL(p) ← 1;
16 |  for $r ∈ ℛ do
17 |  |  iL(p, r) ← 1; cL(p, r) ← 1;
18 |  |  iS(p, r) ← 1; cS(p, r) ← 1;
19 |  end
20 end
21 for $r ∈ ℛ do
22 |  iReg(p) ← 1;
23 |  cReg(p) ← 1;
24 end
25 for k ∈ [1, ··· , K] do
26 |  active(k) ← gen([1, ··· , K]);
27 end
28 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, at lines 14-18.

11.4 Write Statements

Algorithm 7: $[\![x \leftarrow exp]\!]_K^{p,\texttt{Write}}$

```
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(exp));
9  assume(cW(p, x) ≥ iW(p, x));
10 assume(iW(p, x) ≥ max(cDY(p), cDS(p), cDL(p)));
11 assume(iW(p, x) ≥ cL(p));
12 assume(cW(p, x) ≥ max(old-cW, cReg(exp), cR(p, x), ctrl(p));
13 //Update
14 μ(x, cW(p, x)) ← exp;
15 ν(p, x) ← exp;
```

We explain only the changes here compared to the listing in section 10. We check for the rules `AllSyncs`,`AllDmbLds` and `AllDmbSts` in line 10. In line 11 we verify that `AllLAcqs` holds.

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

1 //Guess
2 $\texttt{iW}(p, [exp']) \leftarrow \texttt{gen}([1, \cdots, K])$;
3 $\texttt{old-cW} \leftarrow \texttt{cW}(p, [exp'])$;
4 $\texttt{cW}(p, [exp']) \leftarrow \texttt{gen}([1, \cdots, K])$;
5 //Check
6 $\texttt{assume}(\texttt{iW}(p, [exp']) \geq cnt)$;
7 $\texttt{assume}(\texttt{active}(\texttt{iW}(p, [exp'])) = p)$;
8 $\texttt{assume}(\texttt{iW}(p, [exp']) \geq \texttt{iReg}(exp))$;
9 $\texttt{assume}(\texttt{iW}(p, [exp']) \geq \texttt{iReg}(exp'))$;
10 $\texttt{assume}(\texttt{cW}(p, [exp']) \geq \texttt{iW}(p, [exp']))$;
11 $\texttt{assume}(\texttt{iW}(p, [exp']) \geq \texttt{max}(\texttt{cDY}(p), \texttt{cDS}(p), \texttt{cDL}(p)))$;
12 $\texttt{assume}(\texttt{iW}(p, [exp']) \geq \texttt{cL}(p))$;
13 $\texttt{assume}(\texttt{cW}(p, x) \geq \texttt{max}(\texttt{old-cW}, \texttt{cReg}(exp), \texttt{cReg}(exp'), \texttt{cR}(p, x), \texttt{ctrl}(p))$;
14 //Update
15 $\mu(x, \texttt{cW}(p, [exp'])) \leftarrow exp$;
16 $\nu(p, [exp']) \leftarrow exp$;

The expansion of this component is similar to the last, with the difference that we need to take into account `addr` dependencies. Accordingly, the check inserted at line 9 and in line 13 make sure that the `addr` part of `InitCnd` and `CommitCnd` are satisfied, respectively. Other than that, this expansion is similar to the previous one.

11.5 Assign Instructions

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

1 //Guess
2 $\texttt{iReg}(\$r) \leftarrow \texttt{gen}([1, \cdots, K])$;
3 $\texttt{cReg}(\$r) \leftarrow \texttt{gen}([1, \cdots, K])$;
4 //Check
5 $\texttt{assume}(\texttt{iReg}(\$r) \geq cnt)$;
6 $\texttt{assume}(\texttt{active}(\texttt{iReg}(\$r)) = p)$;
7 $\texttt{assume}(\texttt{iReg}(\$r) \geq \texttt{iReg}(exp))$;
8 $\texttt{assume}(\texttt{iReg}(\$r) \geq \texttt{max}(\texttt{cDY}(p), \texttt{cDS}(p), \texttt{cDL}(p)))$;
9 $\texttt{assume}(\texttt{iReg}(\$r) \geq \texttt{cL}(p))$;
10 $\texttt{assume}(\texttt{active}(\texttt{cReg}(\$r)) = p)$;
11 $\texttt{assume}(\texttt{cReg}(\$r) \geq \texttt{iReg}(\$r))$;
12 $\texttt{assume}(\texttt{cReg}(\$r) \geq \texttt{max}(\texttt{cReg}(exp), \texttt{ctrl}(p))$;
13 //Update
14 $\$r \leftarrow exp$;

The algorithm is largely same as section 10. The changes are the ones we applied to write instructions as well: the rules `AllDmbLds`, `AllDmbSts` and `AllSyncs` constitute line 8, and line 9 expresses `AllLAcqs`.

11.6 Read Instructions

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

```
1  //Guess
2  iR(p, x) ← gen([1, ··· , K]);
3  old-cR ← cR(p, x);
4  cR(p, x) ← gen([1, ··· , K]);
5  iReg($r) ← iR(p, x);
6  cReg($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(iR(p, x) ≥ cDY(p));
12 assume(iR(p, x) ≥ cDL(p));
13 assume(iR(p, x) ≥ iL(p));
14 assume(cR(p, x) ≥ iR(p, x));
15 assume(active(cR(p, x)) = p);
16 assume(cR(p, x) ≥ max(old-cR, cW(p, x), ctrl(p)));
17 //Update
18 if iR(p, x) < cW(p, x) then
19 |  $r ← ν(p, x);
20 else
21 |  $r ← μ(x, iR(p, x));
22 end
```

There are two changes here as compared to section 10: the rules `AllDmbLds` and `AllLAcqInit` have been incorporated into the listing, in lines 11 and 12 respectively. Line 10 lists the `AllSyncs` predicate that appears in the `InitReadFrom...` rules.

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

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

This algorithm is similar to the previous one. The one difference is the incorporation of `data`-order related predicates: line 11 ensures that the read is initialized no earlier than an address-supplying event is. Similarly the modification to line 17 expresses the part of `ComCnd` which needs the instruction to be committed no earlier than any instruction which supplies the addresss to it.

11.7 Load Acquire instructions

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

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

This is similar to the listing for load instructions (which are in the above section of read instructions). The one difference is the `AllSRels` predicate during `init` which shows up on line 15.

11.8 Store Release Instructions

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

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

Store Release Instructions have one extra condition as compared to normal store conditions. It is the `AllMem` predicate which requires *all* po-preceeding instructions that access memory to have been committed. Hence we can do away with `AllLAcqInit` which is subsumed in this. This predicate is expressed in lines 13 through 16: line 13 enforces this for `LAcq`s and `SRel`s, while the loop enforces this for all non-release/acquire accesses to memory (which includes variables). Line 12 as

usual expresses the AllSyncs, AllDmbLds and AllDmbSts predicates.

11.8.1 Barrier Instructions

As mentioned before, we do not elaborate upon the expansion of isb as it is the same as that of dmb.sy; which is given below.

Algorithm 14: $[\![\texttt{dmb.sy}]\!]_K^{p,\texttt{DmbSy}}$

```
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, cDL(p), cDS(p), cL(p), cS(p), ctrl(p)));
7 for x ∈ X do
8 |   assume(cDY(p) ≥ max(cW(p, x), cR(p, x)));
9 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 part of line 5. In the same line we also have part of the AllMem predicate, with the rest of it being in lines 6-8. Line 5 also ensures that all po-previous aci instructions have been committed.

Algorithm 15: $[\![\texttt{dmb.ld}]\!]_K^{p,\texttt{DmbLd}}$

```
1 //Guess
2 cDL(p) ← gen[1, ··· , K];
3 //Check
4 assume(cDL(p) ≥ cnt);
5 assume(cDL(p) ≥ max(cDY(p), cL(p), ctrl(p));
6 for x ∈ X do
7 |   assume(cDL(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 split across line 4 and line 6. The ctrl(p) comes from the ComCnd predicate in ComDmbLd.

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

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

We first guess the commiting context of the dmb.st and run it through some sanity checks: one that it does not commit earlier than the current context. AllDmbSys and ComCnd make their appearance on line 4. The predicate AllWrites is spread across line 4 and line 6.

11.9 Verifying process

Algorithm 17: $\langle \texttt{verProc} \rangle_K$.

```
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 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.