Caveats:

- $\equiv$ is equality up to renaming of IDs (of events and instructions) and the ID state of the system, and the value of the `next_read_order` field of the thread state. The `next_read_order` value orders read events but as long as the relation between values is preserved the exact value does not matter.

- the proof assumes:
  - `apply_tree_context` does a correct update of the tree
  - the pending_read cleanup for `T_only` transitions will only be done for reads from forwarded writes
  - instructions returned from `list_old_instructions` can be removed from the instruction tree without affecting the possible transitions.

(‘Non-memory instruction’ means no memory read or write and no barrier.)

Define:

```plaintext
p1 = function
| T_only_trans _ _ _ (T_internal _) -> true
| T_only_trans _ _ _ (T_finish _) -> true
| T_only_trans _ _ _ (T_register_read _) -> true
| T_only_trans _ _ _ (T_potential_mem_write _) -> true
| T_only_trans _ _ _ (T_register_write _) -> true
| _ -> false

p1' = function
| TSS_fetch _ ioid _ _ _ _ -> (ioid is a branch register instruction)
| _ -> false

p2 = function
| TSS_fetch _ ioid _ _ fdo _ -> (ioid is not a branch register instruction
  ∧ fdo is of the form FDO_success)
| _ -> false
```

Theorem 1

Assume the model is POP.

Let $t, t'$ in `enumerate_transitions_of_system s_0` such that $t \neq t'$, the condition $p1 t$ holds and $p1' t'$ does not hold, and let `system_state_after_transition s_0 t = s` and `system_state_after_transition s_0 t' = s'`. Then:
Assume the model is POP.

Let \( t, t' \) in \( \text{enumerate_transitions_of_system} \, s_0 \) such that \( t \neq t' \) and \( p2 \, t \) holds, and let 
\[
\text{system_state_after_transition} \, s_0 \, t = s \quad \text{and} \quad \text{system_state_after_transition} \, s_0 \, t' = s'.
\]
Then:
\[
\begin{align*}
( t \in \text{enumerate_transitions_of_system} \, s' \wedge \\
\quad t' \in \text{enumerate_transitions_of_system} \, s \wedge \\
\quad \text{system_state_after_transition} \, s' \, t = \text{system_state_after_transition} \, s \, t' ) \\
\lor \\
( t' \in \text{enumerate_transitions_of_system} \, s \wedge \\
\quad \text{system_state_after_transition} \, s \, t' = s' )
\end{align*}
\]

**Proof of Theorem 1**

Let \( t = T\_\text{only_trans} \, \text{tid} \, \text{ioid} \, \text{ids} \, \text{tt} \) and \( \text{inst} \) the instruction instance with 
\( \text{inst.ioid} = \text{ioid} \).

**1. Case \( t' = SS\_\text{only_trans} \, \text{t} \, \text{st} \)**

Then by definition of \( \text{system_state_after_transition} \),
\[
s = s_0 \, \text{with} \, \text{thread_states} \, \text{tid} = \text{thread_state_after_transition} \, \text{tt}.
\]

Because of \( t' \) in \( \text{enumerate_transitions_of_system} \, s_0 \), by definition of \( \text{enumerate_transitions_of_system} : \)
\[
\begin{align*}
&\text{t'} = SS\_\text{only} \, \text{st} \\
&\quad \text{in} \, \text{enumerate_transitions_of_storage_subsystem} \, s_0.\text{storage_subsystem} = \\
&\quad \text{enumerate_transitions_of_storage_subsystem} \, s'.\text{storage_subsystem} \\
&\quad s' = s_0 \, \text{with} \, \text{storage_subsystem} = \text{st}.
\end{align*}
\]

Because of \( t \) in \( \text{enumerate_transitions_of_storage_subsystem} \, s_0 \), by definition of \( \text{enumerate_transitions_of_system} : \)
\[(\text{ioid, (T\_only, idstate'))}\]
\[\text{in enumerate\_transitions\_of\_thread (s\_0.thread\_states tid)}\]
\[= \text{enumerate\_transitions\_of\_thread (s'.thread\_states tid)}\]
\[\implies t \text{ in enumerate\_transitions\_of\_system s'}\]

\[
\text{system\_state\_after\_transition s' t = system\_state\_after\_transition (s\_0 with storage\_subsystem st) t = s\_0 (with storage\_subsystem = st)}
\]
\[
\text{thread\_states tid = thread\_state\_after\_transition tt) = system\_state\_after\_transition (s\_0 with thread\_states tid = thread\_state\_after\_transition tt) t = system\_state\_after\_transition s t'}
\]

2. Case \(t' = \text{SS\_lazy\_trans (SS\_POP\_read\_response read source st)}\)

then by definition of \(\text{enumerate\_transitions\_of\_system}:\)

\[
\text{SS\_interact\_lazy in pop\_ss\_enumerate\_transitions (s\_0.storage) = pop\_ss\_enumerate\_transitions (s.storage)}
\]
\[\implies t' \text{ in enumerate\_transitions\_of\_system s}\]

Let \(\text{inst'}\) be the instruction instance with \(\text{inst'\_ioid = read\_r\_ioid}\).

By definition of \(\text{pop\_satisfy\_read\_action t'}\), for any instruction \(\text{inst''}\): either \(\text{inst''}\) in \(s'\) is unchanged from \(\text{inst''}\) in \(s\_0\); or \(\text{inst''} = \text{inst'}\) and is updated according to \(\text{pop\_satisfy\_read\_action}\); or \(\text{inst''}\) is restarted by \(t'\).

Then \(\text{inst} \neq \text{inst'}\) has to hold: Two cases of \(\text{inst.micro\_op\_state}\) that can enable a \(\text{T\_only}\) transition:

1. \(\text{MOS\_plain}\)
2. \(\text{MOS\_pending\_mem\_read sr c}\)

Assume \(\text{inst} = \text{inst'}\)

1. \(\text{pop\_satisfy\_read\_action\_trans}\) requires \(\text{inst'\_micro\_op\_state}\) to be of the form \(\text{MOS\_pending\_read}\). Contradiction to \(\text{inst} = \text{inst'}\).

2. The only \(\text{T\_only}\) transition enabled in \(\text{inst.micro\_op\_state}\) is \(\text{actually\_satisfy\_transitions}\) and requires \(\text{sr\_sr\_not\_yet\_requested} = []\) and \(\text{sr\_sr\_requested} = []\). Because \(t'\) is enabled, as an invariant of the system state \(\text{sr\_sr\_not\_yet\_requested} \neq []\) or \(\text{sr\_sr\_requested} \neq []\) holds. Contradiction to \(\text{inst} = \text{inst'}\). Therefore \(\text{inst} \neq \text{inst'}\).

Two cases: Taking \(t'\) in state \(s\) restarts \(\text{inst}\) or not.

2.1. Case \(\text{inst}\) restarted by \(t'\)
Then tt cannot be of form T_commit_simple or T_finish because by definition of system_state_after_transition and enumerate_transitions_of_instruction for t in \{T_only tid ioid ids (T_commit_simple _), T_only tid ioid ids (T_finish _)} and inst.committed = true in s and committed instructions are not restarted.

By definition of system_state_after_transition, enumerate_transitions_of_thread, enumerate_transitions_of_instruction,

\[s = s_0 \text{ with } \text{(thread state tid).instruction_tree with inst updated}\]
\[\text{next_read_order updated in the case of } tt = T_{potential\_mem\_write} _\]

By definition of system_state_after_transition, pop_satisfy_read_action_trans, and pop_satisfy_read_action, and by assumption that inst is restarted:

\[s' = s_0 \text{ with } [ \text{(thread states tid).instruction_tree with inst' updated and dependent instructions restarted }\]
\[\text{storage_subsystem cleaned up from old read_requests}\]

\[\equiv (s_0 \text{ with } \text{(thread state tid).instruction_tree with inst updated}\]
\[\text{with next_read_order updated in the case of } tt = T_{potential\_mem\_write} _\]
\[\text{with } [ \text{(thread states tid).instruction_tree with inst' updated and}
\[\text{dependent instructions restarted }\]
\[\text{storage_subsystem cleaned up from old read_requests}\]

\[\equiv (s_0 \text{ with next_read_order updated in the case of } tt = T_{potential\_mem\_write} _\]
\[\text{with } [ \text{(thread states tid).instruction_tree with inst' updated and}
\[\text{dependent instructions restarted }\]
\[\text{storage_subsystem cleaned up from old read_requests}\]

\[= \text{system_state_after_transition s t' (up to next_read_order value)}\]

2.2. Case inst not restarted by t'

By definition of system_state_after_transition, pop_satisfy_read_action_trans, and pop_satisfy_read_action, by inst ≠ inst', and the assumption that inst is not restarted, inst is not changed by t'.

Check that t is enabled in s' by case analysis on inst state in s_0 that enabled t:

2.2.1. Case inst.micro_op_state = MOS_plain with interpreter outcome Read_mem

Then by definition of enumerate_transitions_of_instruction the condition pop_memory_read_request_cand holds in s_0. Then pop_memory_read_request_cand also holds in s': The conjuncts of pop_memory_read_request_cand are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:
forall (prev_inst in inst_context.active_prefix).

is_load_acquire prev_inst \implies
(prev_inst.committed \lor \neg (Set.null prev_inst.writes_read_from)) \lor
match prev_inst.micro_op_state with
| MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
| _ -> false

All requirements of the first form also hold in \( s' \) because \( t' \) cannot restart committed instructions.

The second condition concerns load-acquire instructions \( \text{prev_inst po-before inst} \). Assume condition 2 no longer holds. Then there is a load-acquire instruction \( \text{prev_inst in inst.active_prefix} \) that has been restarted by \( t' \). But then by definition of \( \text{pop_satisfy_read_action}, \text{restart_dependent_subtrees}, \text{dependent_suffix_to_restart} \), and \( \text{dependent_suffix_to_restart_helper} \) the clause \( \text{load_after_load_acquire_dependent} \) would have caused \( \text{inst} \) to be restarted as well, which contradicts the assumption.

Because \( \text{inst} \) is unchanged by \( t' \) and \( \text{pop_memory_read_request_cand} \) holds in \( s' \) transition \( t \) is enabled in \( s' \).

2.2.2. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_mem

No conditions to check: \( \text{inst} \) is unchanged, so \( t \) is still enabled.

2.2.3. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_ea

Doesn’t produce transitions \( t \) for which \( p1 \ t \) holds -- nothing to check.

2.2.4. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_memv

No conditions to check: \( \text{inst} \) is unchanged, so \( t \) is still enabled.

2.2.5. Case inst.micro_op_state = MOS_plain with interpreter outcome Barrier

Doesn’t produce transitions \( t \) for which \( p1 \ t \) holds -- nothing to check.

2.2.6 Case inst.micro_op_state = MOS_plain for interpreter outcome Read_reg r

Need to check that \( \text{find_reg_read} \ r \) in \( s' \) still finds the same value:

For \( \text{find_reg_read} \) to return a different value, by definition of \( \text{enumerate_transitions_of_instruction}, \text{find_reg_read}, \text{and reg_writes_to_this_register} \), the \( \text{reg_writes} \) field of instructions in \( \text{inst.active_prefix} \) \( ++ \) \( \text{inst.old_prefix} \) has to have been changed by \( t' \). The instructions
that have been changed by $t'$ are $\text{inst}'$ and the instructions $\text{insts}$ that have been restarted. By definition of $\text{pop_satisfy_read_action}$ $\text{inst}'.\text{regWrites}$ in $s'$ is the same as in $s_0$. Assume some instruction $\text{inst}'$ in $\text{insts}$ has an updated $\text{regWrites}$ field, so it cannot be $\text{inst}'$ and thus must have been restarted by $t'$. As in $s_0$ instruction $\text{inst}$ reads from $\text{inst}''$ (by assumption that $\text{findRegRead}$ yields a different result), there must be a register in $\text{inst}.\text{regsIn}$ that is contained in $\text{inst}'.\text{regsOut}$, but then by definition of $\text{dependentSuffixToRestartHelper}$ instruction $\text{inst}$ must have been restarted as well, which contradicts the assumption. Therefore $\text{findRegRead} \ r \ in \ s_0$ and $\text{findRegRead} \ r \ in \ s$ return the same value, and $t$ is also enabled in $s'$.

2.2.7. Case $\text{inst.microOpState} = \text{MOS_plain with interpreter outcome Write_reg}$

No conditions to check: $\text{inst}$ is unchanged, so $t$ is still enabled.

2.2.8. Case $\text{inst.microOpState} = \text{MOS_plain with interpreter outcome Internal}$

No conditions to check: $\text{inst}$ is unchanged, so $t$ is still enabled.

2.2.9. Case $\text{inst.microOpState} = \text{MOS_plain with interpreter outcome Footprint}$

 Doesn’t produce transitions $t$ for which $p1_t$ holds -- nothing to check.

2.2.10. Case $\text{inst.microOpState} = \text{MOS_plain with interpreter outcome Done}$

If $\text{inst}$ is committed, there is nothing to check and $t$ is enabled in $s'$. So assume $\text{inst}$ is not committed. Then by assumption $\text{inst}$ is not a memory instruction.

Need to check if $\text{popCommitCand}$ still holds in $s'$. For non-memory instructions $\text{popCommitCand}$ requires $\text{commitDataflow}$ and $\text{commitControlflow}$ to hold. $\text{commitDataflow}$ requires instructions directly feeding into $\text{inst}$'s registers to be committed, more formally: Let $\text{iprevs}$ be the set of po-previous instructions $\text{iprev}$ for which there exists a register $r$ in $\text{iprev.regOut}$ such that $r$ is in $\text{inst.regOut}$ and there is no po-between instruction $\text{ibetween}$ with $r$ in $\text{ibetween.regOut}$. Then all instructions in $\text{iprevs}$ have to be committed.

Let $\text{iprevs}$ be this set for $s$ and $\text{iprevs}'$ for $s'$.

Assume $\text{iprev}'$ is an instruction in $\text{iprevs}'$ which is not committed, and $\text{rs}'$ all registers in $\text{iprev}'.\text{regOut}$ and $\text{inst}\text{.regIn}$ that witness the membership of $\text{iprev}'$ in $\text{iprevs}'$. Since $t'$ does not change the $\text{regIn}$ or $\text{regOut}$ fields of $\text{r}\text{.inst}$ and only sets $\text{regIn}$ and $\text{regOut}$ of restarted instructions to the empty set, for any $r'$ in $\text{rs}'$ the instruction $\text{iprev}'$ must be po-before instructions $\text{iprev}$ from $\text{iprevs}$ that have $r'$ in $\text{iprev}\text{.regOut}$ but are restarted by $t'$ and thus have an empty $\text{regOut}$ field in $s'$ so that
iprev' becomes the "closest" instruction with regs_out determined to contain r'. But then, the restart of any such instruction iprev by definition of dependent_suffix_to_restart_helper would cause the restart of inst, which contradicts the assumption. Therefore iprevs = iprevs', which have all been committed in s_0 and thus also in s'. Thus commitDataflow still holds in s'.

commitControlFlow requires all previous branch instructions to be committed. Since this is true in s_0 it still holds in s'.

Therefore t is also enabled in s'.

2.2.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition t for which \( \mathbf{p1} t \) holds that is enabled in this state is described in actually_satisfy_transitions. Since inst is unchanged by t' the conditions in actually_satisfy_transitions still hold and t is still enabled in s'.

In all cases t is still enabled in s'.

Now show system_state_after_transition s t' ≅ system_state_after_transition s' t. since t and t' obviously update separate parts of the system state.

3. Case t' = TSS_Flowing_POP_commit_mem_write_exclusive_successful

Let tid' be the thread that enables t' and inst' the instruction with inst'.ioid = ioid'.

Then

\[
t' = \text{TSS\_Flowing\_POP\_commit\_mem\_write\_exclusive\_successful} \\
\text{writes prev\_bare\_write (thread\_cont true)} \\
\text{in enumerate\_transitions\_of\_system s_0}
\]

from which follows that

\[
tie = (ioid', (\text{T\_interact\_eager T\_POP\_commit\_mem\_write\_exclusive} \\
\text{writes prev\_writes thread\_cont, ist'}) ) \\
\text{in enumerate\_transitions\_of\_thread tid' in s_0}
\]

for ioid'.micro_op_state = MOS_potential_mem_write wk ws c; that wk is of the form
Write_exclusive, Write_exclusive_release or Write_conditional; that pop_commit_cand holds for tid'; and that pop_ss_accept_write_exclusive_success_cand holds for s_0'.storage_subsystem.

From the definition of enumerate_transitions_of_instruction follows inst ≠ inst' because for t to be enabled inst.micro_op_state cannot be MOS_potential_mem_write.

Show t' in enumerate_transitions_of_system s
If \( t \) is po-before \( t' \), then \( t \) cannot be of the form \( T_{\text{only}} T_{\text{commit simple}} \) for an uncommitted branch: Assume \( t \) commits an uncommitted branch. Then by definition of \commitControlFlow condition \( \text{pop\_commit\_cand} \) cannot hold, because \( \text{inst}' \)'s prefix contains an uncommitted branch. Thus when a branch that is inconsistent with the \( \text{inst}' \)'s NIA is discarded after taking transition \( t \) no instructions are removed from the instruction tree that are po-before \( \text{inst}' \). \( t \) only changes instruction \( \text{inst} \).

Therefore, since \( \text{inst} \neq \text{inst}' \) is the only instruction po-before \( \text{inst}' \) changed by \( t \) is \( \text{inst} \) itself, in case it is in \( \text{inst}' \)'s prefix.

As \( \text{inst}' \) is unchanged by \( t \), if \( \text{pop\_commit\_cand} \) also holds in \( s \), then \( t \) is in \enumerateTransitionsOfThread \( s \).

By definition of \enumerateTransitionsOfInstruction, the following kinds of conditions are checked by \( \text{pop\_commit\_cand} \) and hold in \( s_0 \):

1. \commitDataflow
2. certain kinds of instructions po-before \( \text{inst}' \) are committed
3. there is a load-exclusive po-before \( \text{inst}' \)
4. all previous memory access addresses have been fully determined and for po-earlier reads to overlapping addresses it is determined which writes they read from (and they cannot be restarted any more)

1 to 4 still hold in \( s \):

1. All instructions that directly feed into \( \text{inst}' \)'s input registers have been committed. Since the \( \text{regs\_in} \) and \( \text{regs\_out} \) fields of instruction po-before \( \text{inst}' \) are not changed by \( t \) and committed instructions remain committed this condition is preserved by \( t \).
2. As 2 holds in \( s_0 \) this also holds in \( s \) since committed instructions are not restarted (or "uncommitted").
3. Since \( t \) does not remove or restart instructions po-before \( \text{inst}' \) and it holds in \( s_0 \) 3 also holds in \( s \).
4. As 4 holds in \( s_0 \) and \( t \) does not restart or remove any po-before \( \text{inst}' \) instructions this also holds in \( s' \). Taking \( t \) only progresses the instruction state of \( \text{inst} \) and does not change the state of other instructions po-before \( \text{inst}' \).

As 1 to 4 still hold in \( s \) transition \( t \) is still in \enumerateTransitionsOfThread.

Since \( \text{pop\_ss\_accept\_write\_exclusive\_success\_cand} \) holds in \( s_0 \) and \( s.\text{storage\_subsystem} = s_0.\text{storage\_subsystem} \) the condition \( \text{pop\_ss\_accept\_write\_exclusive\_success\_cand} \) also holds in \( s \) so that \( t' \) in \enumerateTransitionsOfSystem \( s' \).

Show \( t \) in \enumerateTransitionsOfSystem \( s' \) or progress by \( t \) overwritten by \( t' \).
By definition of \texttt{pop-commit-mem-store-action},

\begin{verbatim}
  s'.thread_states tid' =
  s_0.thread_state tid' with
    (inst' updated to include writes in committed_mem_writes,
     with committed = true
     micro_op_state updated)
  dependent instructions restarted
\end{verbatim}

Two cases: inst restarted or inst not restarted.

3.1. Case inst restarted

Then \textit{system-state-after-transition} \( s \cdot t' \equiv s' \) (up to IDs and \textit{next-read-order}). The proof is the same as in 2.1.

3.2. Case inst not restarted.

By definition of \texttt{pop-commit-mem-store-action} any instruction changed by transition \( t' \) is either \( \text{inst}' \) or is restarted by \( t' \).

Then \( t \) in \texttt{enumerate-transitions-of-system} \( s' \) and
\textit{system-state-after-transition} \( s \cdot t' \equiv s' \cdot t \).

The proof is the same as in Case 2.2.

4. Case \( t' = TSS\_fetch \ tid' \ iid' \ ids' \ addr' \ fdo' \ tc' \)

Then by definition of \texttt{enumerate-transitions-of-system} \( s_0 \)

\begin{verbatim}
tfetch = (iid',(T_interact_eager T_fetch addr tc,ids'))
in enumerate_transitions_of_thread tid'
\end{verbatim}

and \( \text{fdo} = s_0.program\_memory addr \). Then by definition of
\texttt{enumerate-transitions-of-thread} \( tfetch \) is either in
\texttt{enumerate-fetch-transitions-of-instruction} \( iiic' \) for some \( iiic' \) with
\( iiic'.iidc_instance.instance_ioid = iid' \) or
\texttt{enumerate-initial-fetch-transitions-of-thread} \( tid' \).

Case \( tfetch \) in \texttt{enumerate-initial-fetch-transitions}

If \( tfetch \) is in \texttt{enumerate-initial-fetch-transitions-of-thread} \( tid' \), then from the definition of \texttt{enumerate-initial-fetch-transitions-of-thread} \( tid \neq tid' \), because the function requires an empty instruction tree. Thus \( t \neq t' \) cannot be enabled in \( tid' \) and \( tid \neq tid' \) follows. In Case \( tfetch \) is enabled by
enumerate_initial_fetch_transitions_of_thread. Thus by assumption
\[ t' \in \text{enumerate.transitions.of.system } s \] as well, because by \( \text{tid} \neq \text{tid}' \):
\[ s \text{.thread.states[tid']} = s_0 \text{.thread.states[tid']} \] and
\[ s \text{.storage.subsystem} = s_0 \text{.storage.subsystem}. \]

Now there are two cases: \( fdo \) of the form
\[ \text{FDO.success address opcode inst init.instruction.state} \] or otherwise.

In case \( fdo \) is of the form \( \text{FDO.success} \) state \( s' \) is \( s_0 \) with \( \text{thread.states tid'} \) updated to include \( \text{inst} \) in initial instruction state in its instruction tree. From the fact that thread \( \text{tid} \) is not changed by \( t' \) follows that \( t \in \text{enumerate.transitions.of.system } s' \) and because \( t \) and \( t' \) update separate parts of the system state
\[ \text{system.state.after.transition } s \ t' \equiv \text{system.state.after.transitions } s' \ t \]

In Case \( fdo \) has the form of a fetch decode outcome error, \( s' \) is an error state. Since this error state does not have any information that depends on the threads state of \( \text{tid} \),
\[ \text{system.state.after.transition } s \ t' \equiv s'. \]

**Case tfetch in enumerate_fetch_transitions_of_instruction**

Now assume \( \text{tfetch} \) is not in \( \text{enumerate.initial.fetch_transitions.of.thread } tid' \) but in
\( \text{enumerate_fetch_transitions_of_instruction } \text{iic'} \) for some \( \text{iic'} \). Let
\[ \text{inst'} = \text{iic'}.\text{iic.instance}. \]

Now there are two cases: \( fdo \) of the form
\[ \text{FDO.success address opcode inst init.instruction.state} \] or otherwise.

**4.1. fdo is fetch decode outcome error**

In Case \( fdo \) has the form of a fetch decode outcome error, \( s' \) is an error state. Since this error state does not have any information that depends on the threads state of \( \text{tid} \) it follows that
\[ \text{system.state.after.transition } s \ t' \equiv s'. \]

**4.2. fdo is of the form FDO_success.**

Show \( t \) in enumerate_transitions_of_system \( s' \)

By definition of \( \text{enumerate.fetch_transitions.of.instruction} \) and
\[ \text{system.state.after.transition } t' \text{ does not affect } \text{inst} \text{ or any instruction po-} \]
\[ \text{before } \text{inst}. \]

Check that \( t \) is enabled in \( s' \) by case analysis on \( \text{inst} \) state in \( s_0 \) that enabled \( t \):
Then by definition of \text{enumerate_transitions_of_instruction} the condition \text{pop_memory_read_request_cand} holds in \text{s}_0. Then \text{pop_memory_read_request_cand} also holds in \text{s}' : The conjuncts of \text{pop_memory_read_request_cand} are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

\[
\text{(forall (prev\_inst in inst\_context.active\_prefix).} \\
\text{is\_load\_acquire prev\_inst} \implies \text{prev\_inst.committed v not (Set.null prev\_inst.writes\_read\_from) v} \\
\text{match prev\_inst.macro\_op\_state with} \\
\text{| MOS\_pending\_mem\_read sr \_ -> sr.sr\_not\_yet\_requested = []} \\
\text{| \_ \_ -> false)}
\]

Both requirements still hold in \text{s}' as \text{t'} does not change any instruction po-before \text{inst}.

4.2.2. Case \text{inst.macro\_op\_state = MOS\_plain} with interpreter outcome \text{Write\_mem}

No conditions to check: \text{inst} is unchanged, so \text{t} is still enabled.

4.2.3. Case \text{inst.macro\_op\_state = MOS\_plain} with interpreter outcome \text{Write\_ea}

Doesn’t produce transitions \text{t} for which \text{p1 t} holds -- nothing to check.

4.2.4. Case \text{inst.macro\_op\_state = MOS\_plain} with interpreter outcome \text{Write\_memv}

No conditions to check: \text{inst} is unchanged, so \text{t} is still enabled.

4.2.5. Case \text{inst.macro\_op\_state = MOS\_plain} with interpreter outcome \text{Barrier}

Doesn’t produce transitions \text{t} for which \text{p1 t} holds -- nothing to check.

4.2.6. Case \text{inst.macro\_op\_state = MOS\_plain} with interpreter outcome \text{Read\_reg\ r}

Need to check that \text{find\_reg\_read} \text{r} in \text{s}' still finds the same value:

For \text{find\_reg\_read} to return a different value, by definition of \text{enumerate_transitions_of_instruction}, \text{find\_reg\_read}, \text{reg\_writes\_to\_this\_register}, the \text{reg\_writes} field of instructions in \text{inst.active\_prefix ++ inst.old\_prefix} has to have been changed by \text{t'}. \text{t'} does not change any instruction in \text{inst'}'s prefix, so this condition still holds in \text{s'}.
4.2.7. Case inst.micro_op_state = MOS_plain with interpreter outcome Write_reg

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

4.2.8. Case inst.micro_op_state = MOS_plain with interpreter outcome Internal

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

4.2.9. Case inst.micro_op_state = MOS_plain with interpreter outcome Footprint

Doesn’t produce transitions \texttt{t} for which \texttt{p1 t} holds -- nothing to check.

4.2.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If \texttt{inst} is committed, there is nothing to check and \texttt{t} is enabled in \texttt{s’}. So assume \texttt{inst} is not committed. Nothing to check either, since this case does not produce any transitions \texttt{t} for which \texttt{p1 t} holds.

4.2.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition \texttt{t} for which \texttt{p1 t} holds that is enabled in this state is described in \texttt{actually_satisfy_transitions}. Since \texttt{inst} is unchanged by \texttt{t}, the conditions in \texttt{actually_satisfy_transitions} still hold and \texttt{t} is still enabled in \texttt{s’}.

In all cases \texttt{t} is still enabled in \texttt{s’}.

Show \texttt{t’} in \texttt{enumerate_transitions_of_system s}

Need to show that \texttt{iiic’} is still in \texttt{unold_instructions}, and \texttt{tfetch} is still in \texttt{enumerate_fetch_transitions_of_instruction iiic’} in state \texttt{s}.

Assume \texttt{iiic’} is not in \texttt{unold_instructions} anymore. Then \texttt{t} moved \texttt{inst’} from the instruction tree to \texttt{old_instructions}, which by definition of \texttt{list_old_instruction} means that \texttt{inst’} must have been committed after taking transition \texttt{t} and its successor instruction must have been fetched. By definition of \texttt{p1} and \texttt{enumerate_transitions_of_instruction} transition \texttt{t} does not commit any instructions and does not fetch new instructions, so that \texttt{inst’}’s unique successor must already have been fetched in \texttt{s_0}, which by \texttt{enumerate_fetch_transitions_of_instruction} contradicts the assumption that \texttt{t’} was enabled in \texttt{s_0}. Assume therefore, that \texttt{iiic’} is still in \texttt{unold_instructions}.

Remains to show that \texttt{tfetch} is in \texttt{enumerate_fetch_transitions_of_instruction iiic’} in state \texttt{s}, which by definition reduces to showing that
1. The value of `potential_fetch_addresses` remains the same.
2. `already_fetched_addresses` in state `s` does not contain any elements that `already_fetched_addresses` in state `s_0` does not contain, and
3. `fetch_transition_of_address` returns the same values.

1. As `t'` is enabled in `s_0` the condition `is_stop_fetch_instruction` cannot hold for `iic'`.

   1. Case `iic'` is committed. By definition of `p1'` transition `t` does not commit any instruction. Therefore `iic'` must have been committed in `s_0` and cannot have written to the PC, so that `next_address_of_committed_instruction` returns the same value in `s'` as in `s_0`.
   2. Case `iic'` is not committed. As `inst'` is not the direct post-successor of a branch instruction the function returns the same successor_fetch_address value as in `s_0`.

2. This condition holds because `t` does not add new elements to the instruction tree or change any instruction's `program_loc` field.

3. This reduces to showing that `ioids_feeding_address` and therefore `starting_inst_instance` returns the same value in state `s` as in `s_0`. An instruction `inst''`'s `reg_writes` field might have been changed by `t` when doing a register write, but that does not change `inst''`'s `ioids_feeding_address` list, as the exhaustive interpreter already finds the register write already when initially analysing `inst''`.

Therefore `t'` in enumerate_transitions_of_system `s`.

Show system_state_after_transition `s' = system_state_after_transition `s` `t`

This simply follows from the fact that `t` and `t'` update separate parts of the system state.

5. Case `t' = T_lazy_trans tid' ioid' ist' tt'`

Then by definition of `enumerate_transitions_of_system`,

\[
\text{tlazy } = (\text{ioid}', (\text{T_interact_lazy } tt', \text{ids'})) \text{ in enumerate_transitions_of_thread } \text{tid}'
\]

in state `s_0`. Let `inst'` be the instruction instance with `inst'.instance_ioid = ioid'`.

5.1. Case `tt' = T_mem_read_request rr rr_slices t`

Then by definition of `enumerate_transitions_of_system`,

\[
\text{pop_ss_accept_event_cand } s_0.\text{storage_subsysem (FRead } \text{rr rr_slices [ ]}) \text{ holds and (ioid', (T_interact_lazy (T_mem_read_request } \text{rr rr_slices } t'), \text{ist'})) is in enumerate_transitions_of_instruction } \text{inst'} \text{ for micro_op_state = MOS_pending_mem_read sr c}.
\]
Because inst's micro_op_state is MOS_pending_mem_read it follows that inst ≠ inst' must hold, since the only transition t for which p1 t holds that is enabled for inst.micro_op_state = MOS_pending_mem_read requires sr.sr_not_yet_requested to be empty whereas for tt' to be enabled sr.sr_not_yet_requested must be non-empty. So assume inst ≠ inst'.

Show t' in enumerate_transitions_of_system s

Since pop_ss_accept_event_cand holds in s_0, it must also hold in s because s.storage_subsystem = s_0.storage_subsystem. Remains to check that tt' in enumerate_transitions_of_instruction in s.

By definition of enumerate_transitions_of_instruction and the fact inst ≠ inst' transition t does not change inst' so that tt' is still enabled in s. Therefore t' in enumerate_transitions_of_system.

Show t in enumerate_transitions_of_system s'

By definition of enumerate_transitions_of_system, pop_ss_accept_event_action, and enumerate_transitions_of_instruction transition t' only updates inst' and the storage subsystem state.

Check that t in enumerate_transitions_of_instruction in s' by case analysis on the micro_op_state that enabled t in s_0.

5.1.1. Case inst.micro_op_state = MOS_plain for interpreter outcome Read_mem

Then pop_memory_read_request_cand holds in s_0. pop_memory_read_request_cand requires

1. instruction of certain type po-before inst to be committed
2. that all load-require instructions inst'' are either committed; or if inst'' = MOS_pending_mem_read sr' _ then sr.sr_not_yet_requested = [] must hold; or inst''.writes_read_from not empty.

The only instruction updated by t' is inst'. As 1 and 2 hold in s_0, 1 is still true in s and 2 is still true for all instructions inst'' ≠ inst'. Check that 2 holds for inst'.

By definition of enumerate_transitions_of_instruction instruction inst' cannot be committed in s_0 and cannot have sr.sr_not_request = [], otherwise t' wouldn't be enabled in s_0. Therefore if condition 2 was true for inst' in s_0 the field inst'.writes_read_from must have been non-empty. Since t' does not change inst's writes_read_from field, pop_memory_read_request_cand must still hold in s'.

Therefore t in enumerate_transitions_of_instruction s'.
5.1.2. Case inst.micro_op_state = MOS_plain for interpreter outcome Write_mem

No condition to check. From the fact that inst is not changed by \( t' \) follows \( t \) in enumerate_transitions_of \( s' \).

5.1.3. Case inst.micro_op_state = MOS_plain for interpreter outcome Write_memv

No condition to check. Since inst is not changed by \( t' \) it follows \( t \) in enumerate_transitions_of \( s' \).

5.1.4. Case inst.micro_op_state = MOS_plain for interpreter outcome Barrier

Then \( \text{pop\_commit\_cand} \) holds in \( s_0 \). Check that \( \text{pop\_commit\_cand} \) still holds in \( s' \):

\( \text{pop\_commit\_cand} \) checks:
1. commitDataflow inst_context
2. commitControlflow inst_context
3. \( \text{pop\_commit\_barrier\_cand} \)

1. still holds, since \( t' \) does not change any instruction's \( \text{regs\_in} \) or \( \text{regs\_out} \) fields and does not "uncommit" instructions.
2. requires conditional branches po-before \( \text{inst} \) to be committed. \( \text{inst}' \) does not affect those instructions and 2 still holds in \( s' \).
3. requires instructions of certain type po-before \( \text{inst} \) to be committed and that the memory access of all po-before memory_access are determined. If this was true in \( s_0 \), this still holds in \( s' \) since \( t' \) only progresses \( \text{inst}' \), by which its memory accesses don't become undetermined.

Therefore \( \text{pop\_commit\_cand} \) still holds in \( s' \) and \( t \) in enumerate_transitions_of_instruction in state \( s' \).

5.1.5. Case inst.micro_op_state = MOS_plain for interpreter outcome Read_reg

Only need to check that \( \text{find\_reg\_read} \) returns the same value.

For \( \text{find\_reg\_read} \) to return a different value, by definition of enumerate_transitions_of_instruction, \( \text{find\_reg\_read} \), \( \text{reg\_writes\_to\_this\_register} \), the \( \text{reg\_writes} \) field of instructions in \( \text{inst}.\text{active\_prefix}++\text{inst}.\text{old\_prefix} \) has to have been changed by \( t' \). Since \( t' \) does not change any instruction's \( \text{reg\_writes} \) field, \( \text{find\_reg\_read} \) will return the same value and \( t \) is enabled in enumerate_transitions_of_instruction in \( s' \).

5.1.6. Case inst.micro_op_state = MOS_plain for interpreter outcome Write_reg
No condition to check. Since \(\text{inst}\) is not changed by \(t'\) it follows \(t\) in \(\text{enumerate_transitions_of } s'\).

5.1.7. Case \(\text{inst.micro_op_state} = \text{MOS\_plain}\) for interpreter outcome Internal

No condition to check. Since \(\text{inst}\) is not changed by \(t'\) it follows \(t\) in \(\text{enumerate_transitions_of } s'\).

5.1.8. Case \(\text{inst.micro_op_state} = \text{MOS\_plain}\) for interpreter outcome Done

No condition to check.

5.1.9. Case \(\text{inst.micro_op_state} = \text{MOS\_pending\_mem\_read sr c}\)

Only need to check that \(sr.\text{sr\_not\_yet\_request} = []\). Since this condition was true in \(s_0\) transition \(t'\) only changes \(\text{inst}'\), and \(\text{inst}' \neq \text{inst}\), this must still be true in \(s'\).

In all cases \(t\) in \(\text{enumerate_transitions_of_instruction}\) in state \(s'\) and therefore in \(\text{enumerate_transitions_of_system } s'\).

Show \(\text{system\_state\_after\_transition } s \; t' = \text{system\_state\_after\_transitions } s' \; t\)

Since \(\text{inst} \neq \text{inst}'\), by definition of \(\text{enumerate_transitions_of_instruction}\) and \(\text{enumerate_transitions_of_system}\) transitions \(t\) and \(t'\) update separate parts of the system state. Therefore,

\[
\text{system\_state\_after\_transition } s \; t' \equiv \text{system\_state\_after\_transition } s' \; t.
\]

5.2. Case \(tt' = T\_commit\_mem\_write ws t\)

Then by definition of \(\text{enumerate_transitions_of_system}\) the condition \(\text{pop\_ss\_accept\_event\_cand} (F\text{Write write})\) holds in \(s_0\),

\[
\text{tlazy} = (\text{oid}',(T\_interact\_lazy (T\_commit\_mem\_write ws t'), \text{ist}'))) \quad \text{in } \text{enumerate_transitions_of_instruction } \text{inst}'
\]

and in state \(s_0\) the conditions \(\text{inst}'.\text{micro\_op\_state} = \text{MOS\_potential\_mem\_write wk ws}\) and \(\text{pop\_commit\_cand}\) hold.

From the definition of \(\text{enumerate_transitions_of_instruction}\) follows \(\text{inst} \neq \text{inst}'\) since for \(t\) to be enabled \(\text{inst}\) cannot have \text{micro\_op\_state} of form \(\text{MOS\_potential\_mem\_write}\).

Show \(t'\) in \(\text{enumerate_transitions_of_system } s\)
Since \( s.storage_subsystem = s_0.storage_subsystem \) the condition \( \text{pop_ss_accept_event_cand} \) also holds in \( s \). Therefore still need to check that \( \text{tlazy} \) is in \( \text{enumerate_transitions_of_thread} \) in state \( s \).

Check that \( \text{find_committed_writes} \) returns the same value. This follows from the fact that by definition of \( \text{enumerate_transitions_of_instruction} \) transition \( t \) does not change the \( \text{committed_mem_writes} \) field of any instruction.

Check that \( \text{pop_commit_cand} \) still holds: check the following requirements:

1. \( \text{commitDataflow} \ inst_context \)
2. \( \text{commitControlflow} \ inst_context \)
3. \( \text{pop_commit_mem_access_cand} \)

1. Still holds in state \( s \) because \( t \) does not change the \( \text{regs_in} \) or \( \text{regs_out} \) fields of instructions and does not "uncommit" instructions.
2. Still holds because committed instructions are not uncommitted by \( t \).
3. checks that instructions of certain kinds po-before \( \text{inst'} \) are committed, that all previous memory accesses are fully determined, and the condition \( \text{arch64_write_commitPrevMightSameAddress_helper} \). Since \( t \) does not uncommit committed instructions the first condition is preserved by \( t \).

\( t \) only progresses \( \text{inst} \) so that \( \text{inst'} \)'s memory accesses remain determined and the second condition still holds.

Since \( t \) only in the case of the \( \text{actually_satisfy_transitions} \) transition changes the \( \text{micro_op_state} \) from \( \text{MOS_pending_mem_read} \) to a different \( \text{micro_op_state} \), only need to check if this transition preserves condition 3. The \( \text{actually_satisfy_transitions} \) transition is only possible for \( \text{micro_op_state} = \text{MOS_pending_mem_read sr c} \) with \( \text{sr.sr_not_yet_requested} = [] \) which in turn requires read-satisfy transition, but by definition of \( \text{pop_satisfy_read_action} \) any read-satisfy transition updates \( \text{writes_read_from} \) to include the source of the read.

Therefore when \( t \) changes \( \text{micro_op_state} \) from \( \text{MOS_pending_mem_read} \) to a different \( \text{micro_op_state} \) the \( \text{writes_read_from} \) field has to be non-empty and the third condition holds.

From \( \text{pop_commit_cand} \) also holds in \( s \) follows \( \text{tlazy} \) in \( \text{enumerate_transitions_of_thread} \) in \( s \).

Two cases: \( t' \) restarts \( \text{inst} \) or \( t' \) doesn't restart \( \text{inst} \).

5.2.1. Case \( t' \) restarts \( \text{inst} \)

Since \( s \equiv s_0 \) with \( \text{inst} \) updated (up to \( \text{old_instructions} \) and \( \text{next_read_order} \) )
system_state_after_transition s t' \equiv 
system_state_after_transition (s_0 with inst updated) t'

As by assumption and by definition of \textit{pop_commit_mem_store_action} transition \(t'\) overwrites inst's state with \textit{restart_inst_instance} it follows that 
\text{system_state_after_transition} s t' \equiv s'.

5.2.2. Case \(t'\) does not restart inst

By definition of \textit{enumerate_transitions_of_instruction} and \textit{pop_commit_mem_store_action} transition \(t'\) only changes instructions that are restarted and \textit{inst}'. As by assumption \textit{inst} is not restarted and because of \textit{inst} \(\neq\) \textit{inst'} instruction \textit{inst} is unchanged.

Show \(t\) in \textit{enumerate_transitions_of_system} \textit{s'}

This requires showing \(t\) in \textit{enumerate_transitions_of_instruction} in state \(s'\) by case analysis on the \textit{micro_op_state} that enabled \(t\) in \(s_0\).

# 5.2.2.1. Case \textit{micro_op_state} = \textit{MOS_plain} with interpreter outcome \textit{Read_mem}

Then by definition of \textit{enumerate_transitions_of_instruction} the condition \textit{pop_memory_read_request_cand} holds in \(s_0\). Then \textit{pop_memory_read_request_cand} also holds in \(s'\): The conditions of \textit{pop_memory_read_request_cand} are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

\[
\text{forall (prev_inst in inst_context.active_prefix).}
\text{is_load_acquire prev_inst \implies}
\text{(prev_inst.committed \vee not (Set.null prev_inst.writes_read_from) \vee}
\text{match prev_inst.micro_op_state with}
\text{| MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []}
\text{| _ -> false)}
\]

All requirements of the first form also hold in \(s'\) because \(t'\) cannot restart committed instructions.

The second condition concerns load-acquire instructions \textit{prev_inst} po-before \textit{inst}. Assume condition 2 no longer holds. Then there is a load-acquire instruction \textit{prev_inst} in \textit{inst.active_prefix} that has been restarted by \(t'\). (Committed instructions are not changed, and other than by restarting \(t'\) does not change the \textit{writes_read_from} or \textit{micro_op_state} fields.) But then by definition of \textit{pop_commit_mem_store_action}, \textit{restart_dependent_subtrees}, \textit{dependent_suffix_to_restart}, and \textit{dependent_suffix_to_restart_helper} the clause \textit{load_after_load_acquire_dependent} would have caused \textit{inst} to be restarted as well, which contradicts the assumption. Because \textit{inst} is unchanged by \(t'\) and \textit{pop_memory_read_request_cand} holds in \(s'\) transition \(t\) is enabled in \(s'\).
No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

Doesn’t produce transitions \texttt{t} for which \texttt{p1 t} holds -- nothing to check.

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

Doesn’t produce transitions \texttt{t} for which \texttt{p1 t} holds -- nothing to check.

Need to check that \texttt{find_reg_read} \texttt{r} in \texttt{s’} still finds the same value:

For \texttt{find_reg_read} to return a different value, by definition of
\texttt{enumerate_transitions_of_instruction}, \texttt{find_reg_read}, \texttt{regWrites_to_this_register}, the \texttt{regWrites} field of instructions in \texttt{inst.active_prefix ++ inst.old_prefix} has to have been changed by \texttt{t’}. The instructions that have been changed by \texttt{t’} are \texttt{inst’} and the instructions \texttt{insts} that have been restarted. By definition of \texttt{pop_commit_mem_store_action} the field \texttt{inst’.regWrites} in \texttt{s’} is the same as in \texttt{s_0}. Assume some instruction \texttt{inst’'} in \texttt{insts} has an updated \texttt{regWrites} field, so it cannot be \texttt{inst’} and thus must have been restarted by \texttt{t’}. As in \texttt{s_0} instruction \texttt{inst} reads from \texttt{inst’’} (by assumption that \texttt{find_reg_read} returns a different result), there must be a register in \texttt{inst.regs_in} that is contained in \texttt{inst’’}.\texttt{regs_out}, but then by definition of \texttt{dependent_suffix_to_restart_helper} instruction \texttt{inst} must have been restarted as well, which contradicts the assumption. Therefore \texttt{find_reg_read} \texttt{r} in \texttt{s’} and \texttt{find_reg_read} \texttt{r} in \texttt{s} return the same value, and \texttt{t} is also enabled in \texttt{s’}.

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.
Doesn’t produce transitions \( t \) for which \( p_1 t \) holds -- nothing to check.

If \( \text{inst} \) is committed, there is nothing to check and \( t \) is enabled in \( s' \). If \( \text{inst} \) is not committed, it doesn’t produce transitions \( t \) for which \( p_1 t \) holds -- nothing to check.

The only kind of transition \( t \) for which \( p_1 t \) holds that is enabled in this state is described in \( \text{actually_satisfy_transitions} \). Since \( \text{inst} \) is unchanged by \( t' \) the conditions in \( \text{actually_satisfy_transitions} \) still hold and \( t \) is still enabled in \( s' \).

In all cases \( t \) is still enabled in \( s' \).

Show \( \text{system_state_after_transition \ s \ t'} = \text{system_state_after_transition \ s' \ t} \)

This follows from the fact that \( t \) and \( t' \) update separate parts of the system state: by definition of \( \text{enumerate_transitions_of_instructions} \), \( \text{pop_commit_mem_store_action} \), and \( \text{system_state_after_transitions} \), \( t' \) changes the storage subsytem state, updates \( \text{inst}' \) and restarts dependent instructions.

By definition of \( \text{enumerate_transitions_of_instruction} \) transition \( t \) only updates \( \text{inst} \), \( \text{old_instructions} \) and \( \text{tid}'s \) next_read_order field. Because of \( \text{inst} \neq \text{inst}' \) and by assumption that \( \text{inst} \) is not restarted \( t \) and \( t' \) update separate parts of the system state, so that \( \text{system_state_after_transitions \ s \ t'} = \text{system_state_after_transitions \ s' \ t} \).

5.3. Case \( tt' = \text{T_commit_barrier \ b \ t} \)

Then by definition of \( \text{enumerate_transitions_of_system} \) and \( \text{enumerate_transitions_of_instruction} \) condition \( \text{pop_ss_accept_event_cand} \ (\text{FBarrier} \ b) \) holds in \( s_0 \).

\[
\text{tlazy} = \left( \text{ioid}', (\text{T_interact_lazy} (\text{T_commit_barrier \ b \ t}), \text{ist}')) \right)
\]

in \( s_0 \) and \( \text{inst}'s \) micro_op_state is MOS_plain with interpreter outcome Barrier bk is' with \( bk \neq \text{ISB} \), and \( \text{pop_commit_cand} \) holds in \( s_0 \).

Then \( \text{inst} \neq \text{inst}' \) as to enable \( t \) instruction \( \text{inst}'s \) micro_op_state cannot enable interpreter outcome Barrier bk is' for a non-ISB barrier.
As \( s \cdot \text{storage_subsystem} = s_0 \cdot \text{storage_subsystem} \) condition

\( \text{pop_ss_accept_event_cand} (\text{Fbarrier} \ b) \) also holds in \( s \). Remains to show

\( t_{\text{lazy}} \) in \( \text{enumerate_transitions_of_instruction} \) in state \( s \), which because of \( \text{inst} \neq \text{inst}' \)

reduces to showing that \( \text{pop_commit_cand} \) still holds in \( s \). The conditions required by

\( \text{pop_commit_cand} \) are
1. \( \text{commitDataflow} \)
2. \( \text{commitControlflow} \)
3. \( \text{pop_commit_barrier_cand} \)

The proof that 1 and 2 still hold is the same as in Case 5.2. Remains 3: for non-ISB barriers condition
3 only requires instructions of particular kinds be committed. Since \( t \) does not "uncommit" any
instructions this still holds in \( s \).

Therefore \( t_{\text{lazy}} \) in \( \text{enumerate_transitions_of_instruction} \) in state \( s \) and thus

\( t' \) in \( \text{enumerate_transitions_of_system} \) \( s \).

Show \( t' \) in \( \text{enumerate_transitions_of_system} \) \( s \)

This reduces to showing \( t \) in \( \text{enumerate_transitions_of_instruction} \) in state \( s' \).

By definition of \( \text{system_state_after_transition} \), \( \text{enumerate_transitions_of_instruction} \),

and \( \text{pop_commit_barrier_action} \)

\[
\begin{align*}
s' &= s_0 \text{ with storage_subsystem updated} \\
& \quad \text{inst' updated.}
\end{align*}
\]

Proof by case analysis on the \( \text{micro_op_state} \) in \( s_0 \) that enabled \( t \).

5.2.3.1. Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome \( \text{Read_mem} \)

Then by definition of \( \text{enumerate_transitions_of_instruction} \) the condition

\( \text{pop_memory_read_request_cand} \) holds in \( s_0 \). Then \( \text{pop_memory_read_request_cand} \) also holds

in \( s' \): The conditions of \( \text{pop_memory_read_request_cand} \) are of two forms:

1. requiring all po-before instructions of certain types to be committed
2. The following:

\[
\begin{align*}
\forall (\text{prev_inst} \in \text{inst_context.active_prefix}). \\
\text{is_load_acquire} \ \text{prev_inst} & \implies \text{prev_inst.committed} \\
(\text{Set.null} \ \text{prev_inst.writes_read_from}) & \lor \text{match prev_inst.micro_op_state with} \\
| \text{MOS_pending_mem_read} \text{sr} & \rightarrow \text{sr.sr_not_yet_requested} = [] \\
| _ & \rightarrow \text{false}
\end{align*}
\]

Show \( t' \) in enumerate_transitions_of_system s'}
Both requirements also hold in $s'$ because $t'$ cannot restart committed instructions, and because $t'$ does not "uncommit" any instructions, changes the \texttt{writes_read_from} or \texttt{sr.sr_not_yet_requested} fields or the form of the \texttt{micro_op_state}. Therefore $t$ is enabled in $s'$.

5.2.3.2. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome \texttt{Write\_mem}

No conditions to check: $\text{inst}$ is unchanged, so $t$ is still enabled.

5.2.3.3. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome \texttt{Write\_ea}

Doesn't produce transitions $t$ for which $p1 t$ holds -- nothing to check.

5.2.3.4. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome \texttt{Write\_memv}

No conditions to check: $\text{inst}$ is unchanged, so $t$ is still enabled.

5.2.3.5. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome $\text{Barrier}$

Doesn't produce transitions $t$ for which $p1 t$ holds -- nothing to check.

5.2.3.6 Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome \texttt{Read\_reg} $r$

Need to check that $\text{find_reg_read} r$ in $s'$ still finds the same value:

For $\text{find_reg_read}$ to return a different value, by definition of $\text{enumerate\_transitions\_of\_instruction}$, $\text{find\_reg\_read}$, $\text{reg\_writes\_to\_this\_register}$, the $\text{reg\_writes}$ field of instructions in $\text{inst.active\_prefix ++ inst.old\_prefix}$ has to have been changed by $t'$. As $t$ does not change any instruction's $\text{reg\_writes}$ fields, $t$ is still enabled in $s'$.

5.2.3.7. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome $\text{Write\_reg}$

No conditions to check: $\text{inst}$ is unchanged, so $t$ is still enabled.

5.2.3.8. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome $\text{Internal}$

No conditions to check: $\text{inst}$ is unchanged, so $t$ is still enabled.

5.2.3.9. Case $\text{inst.micro_op_state} = \text{MOS\_plain}$ with interpreter outcome $\text{Footprint}$
Doesn’t produce transitions $t$ for which $p_1 t$ holds -- nothing to check.

5.2.3.10. Case inst.micro_op_state = MOS_plain with interpreter outcome Done

If $\text{inst}$ is committed, there is nothing to check and $t$ is enabled in $s'$. If $\text{inst}$ is not committed, it doesn’t produce transitions $t$ for which $p_1 t$ holds -- nothing to check.

5.2.3.11. Case inst.micro_op_state = MOS_pending_mem_read

The only kind of transition $t$ for which $p_1 t$ holds that is enabled in this state is described in actually_satisfy_transitions. Since $\text{inst}$ is unchanged by $t'$, the conditions in actually_satisfy_transitions still hold and $t$ is still enabled in $s'$.

In all cases $t$ is still enabled in $s'$.

Show system_state_after_transition $s t' = \text{system_state_after_transition} s' t$

This follows from the fact that $t$ and $t'$ update separate parts of the system state: by definition of enumerate_transitions_of_instruction, pop_commit_barrier_action, and system_state_after_transitions transition $t'$ changes the storage subsystem state and updates $\text{inst}'$. By definition of enumerate_transitions_of_instruction transition $t$ only updates $\text{inst}$, old_instructions and $\text{tid}'s$ next_read_order field. Because of $\text{inst} \neq \text{inst}'$ and by assumption that $\text{inst}$ is not restarted $t$ and $t'$ update separate parts of the system state, so that system_state_after_transitions $s t' \cong \text{system_state_after_transitions} s' t$.

6. Case $t' = \text{T_only_trans tid' ioid' ids' tt'}$

then either (Case 6.1)

\[ t' = \text{T_only_trans tid' ioid' ist' tt'} \]
\[ \text{for } tt' = (\text{T_POP_commit_mem_write_exclusive_fail writes' (thread_cont' false)}) \]

and pop_ss_accept_write exclusive_success_cand' does not hold and

\[ \text{tie} = \text{T_interact_eager (T_POP_commit_mem_write_exclusive writes' prev_writes')} \]
\[ \text{in } \text{enumerate_transitions_of_thread} \]

and pop_commit_cand holds in $s_0$.

or

(Case 6.2) $(\text{ioid'},(\text{T_only tt'},\text{ids'}))$ in enumerate_transitions_of_thread
Since $s$.storage_subsystem = $s_0$.storage_subsystem the condition 
$\text{pop_ss_accept_write_exclusive_success_cand}$ still holds in $s$. From the definition of $p1$ follows $ioid \neq ioid'$ because for $t$ to be enabled inst's micro_op_state cannot be of the form $\text{MOS_potential_mem_write}$. Therefore inst' is unchanged by $t$ and it remains to show that $\text{pop_commit_cand}$ still holds.

Check that $\text{pop_commit_cand}$ still holds: check the following requirements:

1. $\text{commitDataflow inst_context}$
2. $\text{commitControlflow inst_context}$
3. $\text{pop_commit_mem_access_cand}$

1. Still holds in state $s$ because $t$ does not change the regs_in or regs_out fields of instructions and does not "uncommit" instructions.
2. Still holds because committed instructions are not uncommitted by $t$.
3. checks that instructions of certain kinds po-before inst' are committed, that all previous memory accesses are fully determined, and $\text{aarch64_write_commitPrevMightSameAddress_helper}$ holds. Since $t$ does not uncommit committed instructions the first condition is preserved by $t$.

$t$ only progresses inst so that inst's memory accesses remain determined, so the second condition still holds.

Since $t$ only in the case of the $\text{actually_satisfy_transitions}$ transition changes the micro_op_state from $\text{MOS_pending_mem_read}$ to a different micro_op_state, only need to check if this transition preserves condition 3. The $\text{actually_satisfy_transitions}$ transition is only possible for a micro_op_state of the form $\text{MOS_pending_mem_read}$ src with empty src.$\text{sr_not_yet_requested} = []$. This in turn requires a read-satisfy transition which by definition of $\text{pop_satisfy_read_action}$ updates writes_read_from to include the source of the read.

Therefore when $t$ changes the micro_op_state from $\text{MOS_pending_mem_read}$ to a different micro_op_state the writes_read_from field has to be non-empty. Therefore also the third condition holds.

Since $\text{pop_commit_cand}$ also holds in $s$ transition tie in $\text{enumerate_transitions_of_thread}$ in $s$ and therefore $t'$ in $\text{enumerate_transitions_of_system s}$.

Show $t$ in $\text{enumerate_transitions_of_system s'}$ or progress made by $t$ overwritten by $t'$
According to the definition of `pop_commit_mem_store_action` with parameter `maybe_successful` set to `(Just false)` transition `t'` updates `inst'` and restarts instructions that have read from the `writes'` set.

Now there are two cases: `inst` is restarted by `t'` or not. For the case where `inst` is restarted `t`'s progress is overwritten by `t'` and `system_state_after_transition s t' ≅ s'`. The proof is the same as for Case 2.1.

For the case where `inst` is not restarted, `t` in `enumerate_transitions_of_system s'` holds and `system_state_after_transition s t' ≅ system_state_after_transition s' t`. The proof is the same as for Case 2.2.

### 6.2.

Cases of `inst'.micro_op_state` and interpreter outcome that enable `t'` in `s_0`:

#### 6.2.1. Case of `inst'.micro_op_state` = MOS_plain and interpreter outcome Read_mem

Then `inst ≠ inst'`, since by definition of `enumerate_transitions_of_instruction` this `micro_op_state` enables only one transition.

Show `t'` in `enumerate_transitions_of_system s`

Since `inst ≠ inst'` instruction `inst'` is not changed by `t`, so the proof reduces to showing that `pop_memory_read_request_cand` still holds in `s`.

This requires

1. That certain kinds of instructions po-before `inst'` are committed and
2. The following:

```plaintext
forall (prev_inst in inst_context.active_prefix).
  is_load_acquire prev_inst ==
  (prev_inst.committed v not (Set.null prev_inst.writes_read_from) v
   match prev_inst.micro_op_state with
   | MOS_pending_mem_read sr _ -> sr.sr_not_yetRequested = []
   | _ -> false)
```

1. Still holds in `s` because `t` does not "uncommit" instructions.
2. `t` does not change any instruction's `committed` or `writes_read_from` field. Therefore, if either of the first clauses of the disjunction was true in `s_0` it is still true in `s`. If both were false, then the third one must have been true. But then `inst` cannot be po-before `inst'` since for `t` to be enabled the `micro_op_state` must be `MOS_plain`. Therefore `t` does not affect the third clause.
From the fact that `pop_memory_read_request_cand` holds in `s` follows `t'` in `enumerate_transitions_of_system`.

Show `t` in `enumerate_transitions_of_system s'`

Since `inst ≠ inst'` instruction `inst` is not changed by `t'`.

Check that `t` is enabled in `s'` by case analysis on `inst` state in `s_0` that enabled `t`:

6.2.1. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Read_mem`

The proof is symmetrical to what was just proved.

6.2.2. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Write_mem`

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.3. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Write_ea`

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.4. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Write_memv`

No conditions to check: `inst` is unchanged, so `t` is still enabled.

6.2.5. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Barrier`

Doesn't produce transitions `t` for which `p1 t` holds -- nothing to check.

6.2.6. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Read_reg r`

Need to check that `find_reg_read r` in `s'` still finds the same value:

For `find_reg_read` to return a different value, by definition of `enumerate_transitions_of_instruction`, `find_reg_read`, `reg_writes_to_this_register`, the `reg_writes` field of instructions in `inst.active_prefix ++ inst.old_prefix` has to have been changed by `t'`. Because `t'` does not change any instruction's `reg_writes` field, this `find_reg_read` returns the same value in `s'`.

6.2.7. Case `inst.micro_op_state` = MOS_plain with interpreter outcome `Write_reg`

No conditions to check: `inst` is unchanged, so `t` is still enabled.
No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

Doesn't produce transitions \texttt{t} for which \texttt{p1 t} holds -- nothing to check.

If \texttt{inst} is committed, there is nothing to check and \texttt{t} is enabled in \texttt{s'}. So assume \texttt{inst} is not committed. Then by assumption \texttt{inst} is not a memory instruction.

For \texttt{inst.committed = false} there is nothing to do, since this does not produce a transition \texttt{t} for which \texttt{p1 t} holds.

The only kind of transition \texttt{t} for which \texttt{p1 t} holds that is enabled in this state is described in \texttt{actually_satisfy_transitions}. Since \texttt{inst} is unchanged by \texttt{t'}, the conditions in \texttt{actually_satisfy_transitions} still hold and \texttt{t} is still enabled in \texttt{s'}.

In all cases \texttt{t} is still enabled in \texttt{s'}.

Now \texttt{system_state_after_transition s t' == system_state_after_transition s' t} follows from the fact that \texttt{t} and \texttt{t'} update separate parts of the system state

Now there are two cases:

\texttt{inst = inst'} or \texttt{inst \neq inst'}

Then by definition of \texttt{enumerate_transitions_of_instruction} and \texttt{p1},

\[
t = (\text{oid}, (\text{T_only (T_potential_mem_write ws t_potential)}, \text{ist}))
\]

\[
t' = (\text{oid'}, (\text{T_only (T_POP_commit_mem_write_exclusive_fail ws t\_fail)}, \text{ist'}))
\]

for

\[
t\_fail = \text{pop_commit_mem_store_action m t iic ws (c false) (Just false)}
\]
According to \texttt{enumerate_transitions_of_instruction} transition \( t \) updates \texttt{inst}’s \texttt{micro_op_state} to \texttt{MOS\_potential\_mem\_write wk ws c}.

Then, in \( s \) either \texttt{pop\_commit\_cand} holds for

\[
\text{(ioid, (T\_interact\_eager (T\_POP\_commit\_mem\_write\_exclusive ws
load.writes\_read\_from thread\_continuation), ist))}
\]

or not. If it does not hold, then

\[
\text{[ (T\_only (T\_POP\_commit\_mem\_write\_exclusive\_fail ws
\texttt{t\_fail}), ist))}
\]

and therefore \( t’ \) in \texttt{enumerate_transitions_of_instruction}.

If it does hold, then

\[
\text{T\_only\_trans tid ioid ist (T\_POP\_commit\_mem\_write\_exclusive\_fail ws}
\text{(thread\_continuation false)) =}
\text{T\_only\_trans tid ioid ist (T\_POP\_commit\_mem\_write\_exclusive\_fail ws t\_fail}
\text{for t\_fail = pop\_commit\_mem\_store\_action m t iic ws}
\text{(c false) (Just false) ]}
\]

and therefore \( t’ \) in \texttt{enumerate_transitions_of_instruction}.

From the definition of \texttt{enumerate_transitions_of_instruction} follows \( s = s_0 \) with

\texttt{inst.micro_op_state = MOS\_potential\_mem\_write wk ws c}.

By definition of \texttt{pop\_commit\_mem\_store\_action}

\[
s’ = s_0 \text{ with inst.micro_op_state of form MOS\_plain (c false)}
\text{inst.committed_mem\_writes = [] and}
\text{inst.committed = true}
\text{depedent instructions restarted}
\]

Therefore

\[
\text{system\_state\_after\_transition \( s \ t’ =
\text{system\_state\_after\_transition}
\text{(s_0 with inst.micro_op\_state = MOS\_potential\_mem\_write wk ws c.)}
\text{with inst.micro\_op\_state of form MOS\_plain (c false)}
\text{inst.committed\_mem\_writes = [] and}
\text{inst.committed = true}
\text{depedent instructions restarted ≡}
\text{system\_state\_after\_transition s_0}
\text{with inst.micro\_op\_state of form MOS\_plain (c false)}
\text{inst.committed\_mem\_writes = [] and}
\text{inst.committed = true}
\text{depedent instructions restarted}
\]

\[
\text{Show t’ in enumerate\_transitions\_of\_system s}
\]
Since by definition of `dependent_suffix_to_restart_helper` taking \( t' \) in \( s \) will restart the same instructions as in \( s_0 \) and \( t' \) overwrites the changes \( t \) made to \( \text{inst}' \)’s \text{micro_op_state} \text{system_state_after_transition} s t' \( \equiv s' \).

6.2.2.2. Case \( \text{inst} \neq \text{inst}' \)

### 6.2.2.2.1. Case \( t' = \text{T_only T_potential_mem_write} \)

Since \( \text{inst} \neq \text{inst}' \), transition \( t \) does not change \( \text{inst}' \), and from the fact that there are no other preconditions to enabling \( t' \) follows \( t' \) in `enumerate_transitions_of_system s`.

#### Show \( t' \) in `enumerate_transitions_of_system s'`

By case analysis on the state of \( \text{inst} \) in \( s_0 \) that enables \( t' \).

#### 6.2.2.2.1.1. Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Read_mem

This requires

1. That certain kinds of instructions 
   before \( \text{inst}' \) are committed and
2. The following:

   ```
   forall (prev_inst in inst_context.active_prefix).
   is_load_acquire prev_inst =>
   (prev_inst.committed \( \lor \) not (Set.null prev_inst.writes_read_from) \( \lor \)
   match prev_inst.micro_op_state with
   | MOS_pending_mem_read sr _ -> sr.sr_not_yet_requested = []
   | _ --> false)
   ```

   1. still holds in \( s' \) because \( t' \) does not "uncommit" instructions.
2. \( t' \) does not "uncommitted" any instruction, does not change any instruction’s
   \text{writes_read_from} or \text{sr_not_yet_requested} field, and does not change any instruction’s
   \text{micro_op_state} from \text{MOS_pending_mem_read} to a different one. Therefore 2 still holds in
   \( s' \).

#### 6.2.2.2.1.2. Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_mem

No conditions to check: \( \text{inst} \) is unchanged, so \( t \) is still enabled.

#### 6.2.2.2.1.3. Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_ea

Doesn’t produce transitions \( t \) for which \( p1_t \) holds -- nothing to check.

#### 6.2.2.2.1.4. Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_memv
No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

### 6.2.2.2.1.5. Case \texttt{inst.micro_op_state} = MOS\_plain with interpreter outcome Barrier

Doesn’t produce transitions \texttt{t} for which \( p_1 \) \texttt{t} holds -- nothing to check.

### 6.2.2.2.1.6. Case \texttt{inst.micro_op_state} = MOS\_plain with interpreter outcome Read\_reg \texttt{r}

Need to check that \texttt{find\_reg\_read} \texttt{r} in \( s' \) still finds the same value:

For \texttt{find\_reg\_read} to return a different value, by definition of \texttt{enumerate\_transitions\_of\_instruction}, \texttt{find\_reg\_read}, \texttt{reg\_writes\_to\_this\_register}, the \texttt{reg\_writes} field of instructions in \texttt{inst.active\_prefix ++ inst.old\_prefix} has to have been changed by \texttt{t}. Because \texttt{t} does not change any instruction’s \texttt{reg\_writes} field, \texttt{find\_reg\_read} returns the same value in \( s' \).

### 6.2.2.2.1.7. Case \texttt{inst.micro_op_state} = MOS\_plain with interpreter outcome Write\_reg

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

### 6.2.2.2.1.8. Case \texttt{inst.micro_op_state} = MOS\_plain with interpreter outcome Internal

No conditions to check: \texttt{inst} is unchanged, so \texttt{t} is still enabled.

### 6.2.2.2.1.9. Case \texttt{inst.micro_op_state} = MOS\_plain with interpreter outcome Footprint

Doesn’t produce transitions \texttt{t} for which \( p_1 \) \texttt{t} holds--nothing to check.

### 6.2.2.2.1.10. Case \texttt{inst.micro_op_state} = MOS\_plain with interpreter outcome Done

If \texttt{inst} is committed, there is nothing to check and \texttt{t} is enabled in \( s' \). So assume \texttt{inst} is not committed. Then by assumption \texttt{inst} is not a memory instruction.

For \texttt{inst.committed = false} there is nothing to do, since this does not produce a transition \texttt{t} for which \( p_1 \) \texttt{t} holds.

### 6.2.2.2.1.11. Case \texttt{inst.micro_op_state} = MOS\_pending\_mem\_read

The only kind of transition \texttt{t} for which \( p_1 \) \texttt{t} holds that is enabled in this state is described in \texttt{actually\_satisfy\_transitions}. Since \texttt{inst} is unchanged by \texttt{t'}, the conditions in \texttt{actually\_satisfy\_transitions} still hold and \texttt{t} is still enabled in \( s' \).
In all cases \( t \) is still enabled in \( s' \).

Now \( \text{system\_state\_after\_transition} \ s \ t' \equiv \text{system\_state\_after\_transition} \ s' \ t \) follows from the fact that \( t \) and \( t' \) update separate parts of the system state.

6.2.2.2.2. Case \( t' = \text{T\_only T\_POP\_commit\_mem\_write\_exclusive\_fail} \)

Two cases: Taking \( t' \) in state \( s \) restarts \( \text{inst} \) or not.

\# 6.2.2.2.2.1. Case \( \text{inst} \) restarted by \( t' \)

Then \( t' \) overwrites \( t' \)'s progress and \( \text{system\_state\_after\_transition} \ s \ t' \equiv \text{system\_state\_after\_transition} \ s' \ t \). The proof is the same as for Case 2.1.

\# 6.2.2.2.2. Case \( \text{inst} \) not restarted by \( t' \)

Then \( t \) in \( \text{enumerate\_transitions\_of\_system} \ s' \) and

\( \text{system\_state\_after\_transition} \ s \ t' \equiv \text{system\_state\_after\_transition} \ s' \ t \). The proof is the same as for Case 2.2.

6.2.3. Case of \( \text{inst'.micro\_op\_state} = \text{MOS\_plain} \) and interpreter outcome \( \text{Write\_ea} \)

From the definition of \( \text{enumerate\_transitions\_of\_instruction} \) follows \( \text{inst} \neq \text{inst'} \). Since \( t' \) does not have other preconditions and because \( \text{inst'} \) is unchanged by \( t \):

\( t' \) in \( \text{enumerate\_transitions\_of\_system} \ s \).

And:

\( t \) in \( \text{enumerate\_transitions\_of\_system} \ s' \). Proof by case analysis on the state of \( \text{inst} \) in \( s_0 \) that enables \( t \). The proof is the same as for case 6.2.2.2.1.

Because \( t \) and \( t' \) update separate parts of the system state:

\( \text{system\_state\_after\_transition} \ s \ t' \equiv \text{system\_state\_after\_transition} \ s' \ t \).

6.2.4. Case of \( \text{inst'.micro\_op\_state} = \text{MOS\_plain} \) and interpreter outcome \( \text{Write\_memv} \)

The proof here is the same as for Case 6.2.

6.2.5. Case of \( \text{inst'.micro\_op\_state} = \text{MOS\_plain} \) and interpreter outcome \( \text{Barrier} \)
By definition of \texttt{enumerate_transitions_of_system} and
\texttt{enumerate_transitions_of_instruction} transition \( t' \) has to be of the form
\( T_{\text{only\_trans\_\_}} (T_{\text{commit\_simple}} \text{Nothing Nothing tt'}) \) for
\( tt' = \text{pop\_commit\_barrier\_action} m t iic' b is' \) and \( \text{pop\_commit\_cand} m t iic' \) holds for
\( \text{inst}' \).

By definition of \texttt{enumerate_transitions_of_instruction} \( \text{inst} \neq \text{inst}' \).

Show \( t' \) in \texttt{enumerate_transitions_of_system} \( s \).

Since \( \text{inst} \neq \text{inst}' \) transition \( t \) does not change \( \text{inst}' \) and the proof reduces to showing
\( \text{pop\_commit\_cand} \) still holds in \( s' \).

\( \text{pop\_commit\_cand} \) checks:
1. \( \text{commitDataflow} \ \text{inst\_context} \)
2. \( \text{commitControlflow} \ \text{inst\_context} \)
3. \( \text{pop\_commit\_barrier\_cand} \)

1. Still holds, since \( t' \) does not change any instruction's \( \text{regs\_in} \) or \( \text{regs\_out} \) fields and does
not "uncommit" instructions, this is still true in \( s' \).
2. requires conditional branches po-before \( \text{inst} \) to be committed. \( \text{inst}' \) does not affect those
instructions and 2 still holds in \( s' \).
3. requires instruction of certain type po-before \( \text{inst} \) to be committed and that all po-before
memory accesses are be determined. If this was true in \( s_0 \), this still holds in \( s \) since \( t \) only
progresses \( \text{inst}' \), by which its memory accesses don't become undetermined.

Therefore \( t' \) in \texttt{enumerate_transitions_of_system} \( s \).

The proof of \( t \) in \texttt{enumerate_transitions_of_system} \( s' \) and
\( \text{system\_state\_after\_transition} \ s \ t' \equiv \text{system\_state\_after\_transition} \ s' \ t \) is the same as
for 6.2.2.2.1.

6.2.6) Case of \( \text{inst'.micro\_op\_state} = \text{MOS\_plain} \) and \text{interpreter outcome Read\_reg}

From the definition of \texttt{enumerate_transitions_of_instruction} follows \( \text{inst} \neq \text{inst}' \) and the
proof of \( t' \) in \texttt{enumerate_transitions_of_system} only requires showing that \( \text{find\_reg\_read} \)
returns the same value in \( s_0 \) and in \( s \).

Assume \( \text{find\_reg\_read} \) returns a different value. Then by definition of \( \text{find\_reg\_read} \),
\( \text{reg\_writes\_to\_this\_register} \) must return a different value for an instruction in \( \text{inst}' \)'s prefix,
which requires \( t \) to have changed the \( \text{reg\_writes} \) or \( \text{reg\_outs} \) field of an instruction. The only
transition \( t \) for which \( p1 \ t \) holds that changes either of these is of the form
\( T_{\text{only\_T\_register\_write}} \) and only changes \( \text{inst}' \)'s \( \text{reg\_writes} \) field. So assume \( t \) is of that
form and writes to the register \( r \) that \( t' \) reads from. Since all instructions only write to every register once \( \text{inst} \) cannot have written to \( r \) before, but \( \text{analyse_instruction} \) must have included \( r \) in \( \text{inst} \)'s \text{regs_out} field. Therefore \( \text{find_reg_read} \) must have returned \( \text{FRRO} \text{blocked} \) in \( s_0 \). But then by definition of \( \text{enumerate_transitions_of_instruction} \) transition \( t' \) would not have been enabled in \( s_0 \).

Therefore \( t' \) in \( \text{enumerate_transitions_of_system} \ s' \).

The proof of \( t \) in \( \text{enumerate_transitions_of_system} \ s' \) and \( \text{system_state_after_transition} \ s \ t' \equiv \text{system_state_after_transition} \ s' \ t \) is the same as for 6.2.2.2.1.

6.2.7. Case of \( \text{inst'.micro_op_state} = \text{MOS_plain} \) and interpreter outcome Write_reg

From the definition of \( \text{enumerate_transitions_of_instruction} \) follows \( \text{inst} \neq \text{inst'} \).

Since \( t' \) does not have any precondition and \( t \) does not change \( \text{inst'} \), \( t' \) in \( \text{enumerate_transitions_of_instruction} \).

Show \( t \) in \( \text{enumerate_transitions_of_instruction} \):

Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Read_mem See proof of 6.2.2.2.1.1.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_mem See 6.2.2.2.1.2.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_ea See 6.2.2.2.1.3.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_memv See 6.2.2.2.1.4.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Barrier See 6.2.2.2.1.5.

Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Read_reg \( r \)

Need to check that \( \text{find_reg_read} \ r \) in \( s' \) still finds the same value: The proof is symmetrical to the one in 6.2.6.

Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Write_reg See 6.2.2.2.1.7.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Internal See 6.2.2.2.1.8.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Footprint See 6.2.2.2.1.9.
Case \( \text{inst.micro_op_state} = \text{MOS_plain} \) with interpreter outcome Done See 6.2.2.2.1.10. Case \( \text{inst.micro_op_state} = \text{MOS_pending_mem_read} \) See 6.2.2.2.1.11.

Since \( t \) and \( t' \) update separate parts of the system state:

\( \text{system_state_after_transitions} \ s \ t' \equiv \text{system_state_after_transition} \ s' \ t \).

6.2.8. Case of \( \text{inst'.micro_op_state} = \text{MOS_plain} \) and interpreter outcome Internal
From the definition of enumerate_transitions_of_instruction follows inst ≠ inst'. Therefore t does not change inst' and since t' has no other precondition, t' in enumerate_transitions_of_systems s.

The proof of t in enumerate_transitions_of_system s' and system_state_after_transition s t' ≅ system_state_after_transition s' t is the same as for 6.2.2.2.1.

6.2.9. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Footprint

This does not produce a transition t' for which p1 t' holds.

6.2.10. Case of inst'.micro_op_state = MOS_plain and interpreter outcome Done

From enumerate_transitions_of_instruction's definition follows inst ≠ inst'. Therefore t does not change inst' and the proof of t' in enumerate_transitions_of_system s reduces to showing that pop_commit_cand still holds. The proof of this is the same as in 5.1.4.

Now there are two cases:

- t' commits a branch instruction and discards a subtree that contains inst or otherwise.

  6.2.10.1: t' discards inst

Since s ≅ s_0 with inst updated (up to old_instructions and next_read_order) and t' removes inst from the instruction tree, system_state_after_transition s t' ≅ s'.

  6.2.10.2: t' does not discard inst

The proof of t in enumerate_transitions_of_system s'

and system_state_after_transition s t' ≅ system_state_after_transition s' t is the same as for 6.2.2.2.1.

6.2.11. Case of inst'.micro_op_state = MOS_pending_mem_read

From the definition of enumerate_transitions_of_instruction follows inst ≠ inst': Assume inst = inst'. Since p1 t holds, t must be an actually_satisfy_transitions transition. There is only one transition of this type and it requires sr.sr_not_yet_requested to be empty whereas the write_forward_transitions requires sr.sr_not_yet_requested to be non-empty. Contradiction. Therefore assume inst ≠ inst'.
In the case that $t'$ is an `actually_satisfy_transitions` transition it only has the precondition that $sr.sr_not_yet_requested$ is empty. Since $inst \neq inst'$ transition $t$ does not change this and $t'$ is still enabled in $s$.

The proof of $t$ in `enumerate_transitions_of_system s'` and $\text{system_state_after_transition } s \ t' \equiv \text{system_state_after_transition } s' \ t$ is the same as for 6.2.2.2.1.

6.2.11.2.

If $t'$ is a `write_forward_transitions` transition, then the only precondition to $t'$ being enabled in $s$ is that $sr.sr_not_yet_requested$ in $s$ is the same as in $s'$. Because of $inst \neq inst'$ transition $t$ does not change $inst'$ and this is still true in $s$.

Now there are two cases: $inst$ is restarted by $t'$ or not.

In the case that $inst$ is restarted by $t'$ transition $t$'s changes are overwritten by $t'$ and $\text{system_state_after_transition } s' \ t \equiv \text{system_state_after_transition } s \ t'$. For the proof see Case 2.1.

Otherwise $t$ in `enumerate_transitions_of_system s'` and $\text{system_state_after_transition } s \ t' \equiv \text{system_state_after_transition } s' \ t$. For the proof see Case 2.2.

**Proof of Theorem 2**

Let $t = \text{TSS_fetch } tid \ iid \ ist' \ addr \ fdo \ tc$ such that $p2 \ t$, and let $fdo = \text{FDO_success } addr \ opcode \ inst' \ init_instruction_state$. By definition of `enumerate_transitions_of_system` this means

\[
t_{\text{fetch}} = (iid, (\text{T_interact_eager } T_{\text{fetch }} addr tc, ids')) \in \text{enumerate_transitions_of_thread } tid
\]

for some $tid$.

Then

$s = s_0$ with thread_states $tid$ updated with $inst''$ added as a leaf in the instruction tree.

1. Case $t' = \text{SS_only_trans } t \ st$

The proof is the same as for Case 1 in the proof of Theorem 1.
2. Case \( t' = \text{SS\_lazy\_trans (SS\_POP\_read\_response read source st)} \)

Show \( t' \) in \text{enumerate\_transitions\_of\_system s} \)

then by definition of \text{enumerate\_transitions\_of\_system}:

\[
\text{SS\_interact\_lazy in pop\_ss\_enumerate\_transitions (s_0.storage) = pop\_ss\_enumerate\_transitions (s.storage)}
\]

and \( t' \) in \text{enumerate\_transitions\_of\_system s} .

Show \( t \) in \text{enumerate\_transitions\_of\_system s'}

Let \( \text{inst'} \) be the instruction instance with \( \text{inst'}.ioid = \text{read.r_ioid} \).

By definition of \text{enumerate\_transitions\_of\_thread} transition \( t \text{fetch} \) is either in

\text{enumerate\_fetch\_transitions\_of\_instruction iic} for some \( iic \) with

\( \text{iic.iic\_instance.instance\_ioid = ioid} \) or

\text{enumerate\_initial\_fetch\_transitions\_of\_thread tid} .

Case \( t \text{fetch} \) in \text{enumerate\_initial\_fetch\_transitions}

If \( t \text{fetch} \) is in \text{enumerate\_initial\_fetch\_transitions\_of\_thread tid} , then from the definition
of \text{enumerate\_initial\_fetch\_transitions\_of\_thread} follows \( \text{tid} \neq \text{read.r\_thread} \), because
the function requires an empty instruction tree which cannot have enabled \( t' \). From the fact that
thread \( \text{tid} \) by definition of \text{pop\_satisfy\_read\_action\_trans} is not changed by \( t' \) follows
\( t \) in \text{enumerate\_transitions\_of\_system s'} and because \( t \) and \( t' \) update separate parts of
the system state

\[
\text{system\_state\_after\_transition s t' } \equiv \text{system\_state\_after\_transitions s' t} .
\]

Case \( t \text{fetch} \) in \text{enumerate\_fetch\_transitions\_of\_instruction}

Now assume \( t \text{fetch} \) is not in \text{enumerate\_initial\_fetch\_transitions\_of\_thread tid} but in
\text{enumerate\_fetch\_transitions\_of\_instruction iic} for some \( iic \). Let
\( \text{inst} = \text{iic.iic\_instance} \).

By definition of \text{pop\_satisfy\_read\_action\_trans} , for any instruction \( \text{inst'''} \): either \( \text{inst'''} \) in
\( s' \) is unchanged from \( \text{inst'''} \) in \( s_0 \), \( \text{inst'''} = \text{inst'} \) and is updated according to
\text{pop\_satisfy\_read\_action\_trans} , or \( \text{inst'''} \) is restarted by \( t' \).
Therefore, only need to show that and \( tfetch \) is still in
\[ \text{enumerate_fetch_transitions_of_instruction} \ iic \ \text{in state} \ s', \] which by definition reduces to showing that

1. the value of \( \text{potential_fetch_addresses} \) remains the same,
2. \( \text{already_fetched_addresses} \) in state \( s \) does not contain any elements that \( \text{already_fetched_addresses} \) in state \( s' \) does not contain.
3. \( \text{fetch_transition_of_address} \) returns the same value

1. As \( t \) is enabled in \( s_0 \) the condition \( \text{is_stop_fetch_instruction} \) cannot hold for \( iic \).
   1. Case \( iic \) is committed. Then it cannot write to the PC and
      \[ \text{next_address_of_committed_instruction} \] returns the same value in \( s' \) as in \( s_0 \).
   2. Case \( iic \) is not committed. As \( iic \) is no branch instruction, \( \text{successor_fetch_address} \) returns \( iic.\text{program_loc} + 4 \). Since \( t' \) does not change any instruction's \( \text{program_loc} \) field this value is the same in \( s \) and \( s_0 \).
2. This condition holds because \( t' \) does not add new elements to the instruction tree or change any instruction's \( \text{program_loc} \) field.
3. This reduces to showing that \( \text{iods_feeding_address} \) and therefore
   \[ \text{starting_inst_instance} \] returns the same value in state \( s' \) as in \( s_0 \). An instruction \( \text{inst'}'\)'s \( \text{reg_writes} \) field might have been changed by \( t \) when it is restarted, but that does not change the \( \text{iods_feeding_address} \) list, as the exhaustive interpreter already finds the register write already when initially analysing \( \text{inst'}' \).

Therefore \( t \) in \( \text{enumerate_transitions_of_system} \ s' \).

Now \( \text{system_state_after_transition} \ s \ t' \equiv \text{system_state_after_transition} \ s' \ t \) follows from the fact that \( t \) and \( t' \) update separate parts of the system state.

3. Case \( t' = \text{TSS_Flowing_POP_commit_mem_write_exclusive_successful} \)

Let \( \text{tid'} \) be the thread that enables \( t' \).

Then
\[ t' = \text{TSS_Flowing_POP_commit_mem_write_exclusive_successful} \] writes \( \text{prev_bare_write} \)
\[ \text{(thread_cont true)} \]
\[ \text{in enumerate_transitions_of_system} \ s_0 \from which follows that \]
\[ \text{tie} = (ioid', \text{(T_interact_eager T_POP_commit_mem_write_exclusive writes prev_writes thread_cont, inst')}) \]
\[ \text{in enumerate_transitions_of_thread} \ \text{tid'} \ \text{in} \ s_0 \]
\[ \text{for} \ ioid'.\text{micro_op_state} = \text{MOS_potential_mem_write wk ws c} \]
and that \( wk \) is of the form \( \text{Write}\_\text{exclusive}, \text{Write}\_\text{exclusive}\_\text{release} \) or \( \text{Write}\_\text{conditional} \); that \( \text{pop}\_\text{commit}\_\text{cand} \) holds for \( \text{tid}' \); and that 
\[ \text{pop}\_\text{ss}\_\text{accept}\_\text{write}\_\text{exclusive}\_\text{success}\_\text{cand} \ s.\text{model.ss storage write prev}\_\text{bare write} \]
holds for \( s_0'\text{.storage_subsystem} \).

**Show \( t \) in enumerate_transitions_of_system \( s \)**

\( t \) does not change the state of any instruction in \( s_0 ' \)'s instruction tree, only adds a leaf to the instruction tree for the instruction that was fetched. Therefore \( \text{inst}' \) and any instruction po-before \( \text{inst}' \) is not changed by \( t \).

As \( \text{inst}' \) is unchanged by \( t \), if \( \text{pop}\_\text{commit}\_\text{cand} \) also holds in \( s \), then \( \text{tie} \) is in enumerate_transitions_of_thread \( s \). Show \( \text{pop}\_\text{commit}\_\text{cand} \) still holds in \( s \).

By definition of enumerate_transitions_of_instruction, the following kinds of conditions are checked by \( \text{pop}\_\text{commit}\_\text{cand} \) and hold in \( s_0 \):

1. instructions directly feeding into \( \text{inst}' \)'s input registers are committed
2. certain kinds of po-before instructions of \( \text{inst}' \) are committed
3. there is a po-before \( \text{inst}' \) load-exclusive
4. all previous memory access addresses have been fully determined and for po-earlier reads to overlapping addresses it is determined which writes they read from (and they cannot be restarted any more)

Since all conditions refer to instruction po-before \( \text{inst}' \) which are unchanged by \( t \), condition \( \text{pop}\_\text{commit}\_\text{cand} \) still holds in \( s \).

Since \( \text{pop}\_\text{ss}\_\text{accept}\_\text{write}\_\text{exclusive}\_\text{success}\_\text{cand} \) holds in \( s_0 \) and 
\( s.\text{storage_subsystem} = s_0.\text{storage_subsystem} \),
\( \text{pop}\_\text{ss}\_\text{accept}\_\text{write}\_\text{exclusive}\_\text{success}\_\text{cand} \) also holds in \( s \) so that \( t' \) in enumerate_transitions_of_system \( s \).

**Show \( t \) in enumerate_transitions_of_system \( s' \)**

By definition of enumerate_transitions_of_thread transition \( \text{tfetch} \) is either in enumerate_fetch_transitions_of_instruction \( iic \) for some \( iic \) with
\( iic.\text{iic_instance.instance_ioid} = \text{ioid} \) or enumerate_initial_fetch_transitions_of_thread \( \text{tid} \).

**Case \( \text{tfetch} \) in enumerate_initial_fetch_transitions**
If \( \text{tfetch} \) is in \( \text{enumerate_initial_fetch_transitions_of_thread} \ tid \), then from the definition of \( \text{enumerate_initial_fetch_transitions_of_thread} \) follows \( \text{tid} \neq \text{read.r_thread} \), because the function requires an empty instruction tree which cannot have enabled \( t' \). From the fact that thread \( \text{tid} \) by definition of \( \text{pop_satisfy_read_action_trans} \) is not changed by \( t' \) follows \( t \) in \( \text{enumerate_transitions_of_system} s' \) and because \( t \) and \( t' \) update separate parts of the system state
\[
\text{system_state_after_transition} \ s \ t' \neq \text{system_state_after_transitions} \ s' \ t.
\]

Case \( \text{tfetch} \) in \( \text{enumerate_fetch_transitions_of_instruction} \ iic \)

Now assume \( \text{tfetch} \) is not in \( \text{enumerate_initial_fetch_transitions_of_thread} \ tid \) but in \( \text{enumerate_fetch_transitions_of_instruction} \ iic \) for some \( iic \). Let \( \text{inst} = iic.iic_instance \).

By definition of \( \text{pop_commit_mem_store_action} \), for any instruction \( \text{inst}'\) : either \( \text{inst}' \) in \( s' \) is unchanged from \( \text{inst}' \) in \( s_0 \), \( \text{inst}' = \text{inst} \) and \( \text{inst}' \) is updated according to \( \text{pop_commit_mem_store_action} \), or \( \text{inst}' \) is restarted by \( t' \).

Showing that \( \text{tfetch} \) is still in \( \text{enumerate_fetch_transitions_of_instruction} \ iic \) in state \( s' \), which by definition reduces to showing that

1. the value of \( \text{potential_fetch_addresses} \) remains the same,
2. \( \text{already_fetched_addresses} \) in state \( s \) does not contain any elements that \( \text{already_fetched_addresses} \) in state \( s' \) does not contain.
3. \( \text{fetch_transition_of_address} \) returns the same value

   1. As \( t \) is enabled in \( s_0 \) the condition \( \text{is_stop_fetch_instruction} \) cannot hold for \( iic \).

      1. Case \( iic \) is committed. Then it cannot write to the PC and
         \( \text{next_address_of_committed_instruction} \) returns the same value in \( s' \) as in \( s_0 \).
      2. Case \( iic \) is not committed. As \( iic \) is no branch instruction, \( \text{successor_fetch_address} \)
         returns \( iic.program_loc + 4 \). Since \( t' \) does not change any instruction's \( \text{program_loc} \) field this value is the same in \( s \) and \( s_0 \).

   2. This condition holds because \( t' \) does not add new elements to the instruction tree or change any instruction's \( \text{program_loc} \) field.

   3. This reduces to showing that \( \text{ioids_feeding_address} \) and therefore \( \text{start_inst_instance} \)
      returns the same value in state \( s' \) as in \( s_0 \). As \( t' \) does not change any instruction's
      \( \text{reg_out} \) or \( \text{reg_writes} \) fields this still holds in \( s' \).

Therefore \( t \) in \( \text{enumerate_transitions_of_system} s' \).

Now \( \text{system_state_after_transition} \ s \ t' \neq \text{system_state_after_transitions} \ s' \ t \) follows from the fact that \( t \) and \( t' \) update separate parts of the system state.
4. Case $t' = \text{TSS\_fetch\_tid'} \text{ ioid'} \text{ ist'} \text{ addr'} \text{ fdo'} \text{ tc'}$

Two cases: at least one of the transitions is enabled by
\(\text{enumerate\_initial\_fetch\_transitions\_of\_thread}\) or both are enabled by
\(\text{enumerate\_fetch\_transitions\_of\_instruction}\).

4.1. $t$ or $t'$ enabled by \(\text{enumerate\_initial\_fetch\_transitions\_of\_thread}\)

Then $\text{tid} \neq \text{tid'}$ as the \(\text{enumerate\_initial\_fetch\_transitions}\) requires an empty instruction tree, in which case there is only one transition enabled for the thread.

Then $s\_\text{thread\_state tid'} = s\_0\_\text{thread\_state tid'}$. Therefore $t'$ in
\(\text{enumerate\_transitions\_of\_system}\ s$.

Now there are two cases: \(\text{fdo}\) is an error fetch decode outcome or not.

If \(\text{fdo}\) is an error fetch decode outcome, then taking transition $t'$ leads to a whole-system error state and $\text{system\_state\_after\_transition} s \ t' \equiv s'$.

If \(\text{fdo}\) is a \(\text{FDO\_success}\) outcome, then $s\_\text{thread\_state tid} = s\_0\_\text{thread\_state tid}$ so that $t$ in $\text{enumerate\_transitions\_of\_system}\ s'$. 

As $t$ and $t'$ update separate parts of the system state:

$\text{system\_state\_after\_transition} s \ t' \equiv \text{system\_state\_after\_transition} s' \ t$.

4.2. $t$ and $t'$ enabled by \(\text{enumerate\_fetch\_transitions\_of\_instruction}\)

By assumption $t \neq t'$, and by definition of \(\text{enumerate\_fetch\_transitions\_of\_instruction}\) for $t$ and $t'$ to be different, \text{addr} \neq \text{addr'}$.(As \text{already\_fetched\_addresses}, \text{outstanding\_fetch\_addresses}, and \text{fetch\_transition\_of\_address} are functions only different addresses can produce different transitions).

Since $t$ does not change any instruction's state, \(\text{enumerate\_fetch\_transitions\_of\_instruction}\) in system state $s$ the function \text{potential\_fetch\_addresses} returns the same set, including \text{addr'} the field \text{already\_fetched\_addresses} contains the same elements as in state $s\_0$, but with \text{addr} added, which has been fetch and added to the instruction tree by $t$. Since \text{addr'} \neq \text{addr} the address \text{addr'} cannot be in this set and therefore must be contained in \text{outstanding\_fetch\_addresses}. Again, since $t$ does not change any instruction's state, \text{fetch\_transition\_of\_address} produces the same result for \text{addr'} as in $s\_0$, so that $t'$ in $\text{enumerate\_transitions\_of\_system}\ s$. 

The proof for \( t \) in `enumerate_transitions_of_system s'` is symmetrical. Since \( t \) and \( t' \) update separate parts of the system state:
\[
\text{system_state_after_transition s t'} \cong \text{system_state_state_after_transition s' t}.
\]

5. Case \( t' = \text{T_lazy_trans tid' ioid' ist' tt'} \)

Then by definition of `enumerate_transitions_of_system`,
\[
\text{tlazy} = (\text{ioid'},(\text{T_interact_lazy tt'},\text{ids'})) \in \\
\text{enumerate_transitions_of_thread tid'} \text{ in state s}_0
\]

Let \( \text{inst}' \) be the instruction instance with \( \text{inst'}.\text{instance_ioid} = \text{ioid}' \).

5.1. Case \( tt' \) is of form \( \text{T_mem_read_request rr rr_slices tt} \)

Then by definition of `enumerate_transitions_of_system`,
\[
\text{pop_ss_accept_event_cand s}_0.\text{storage_subsystem} (\text{FRead rr rr_slices []}) \text{ holds and} \\
(\text{ioid'}, (\text{T_interact_lazy (T_mem_read_request rr rr_slices t'}), \text{ist'})) \text{ is in} \\
\text{enumerate_transitions_of_instruction inst'} \text{ for} \\
\text{micro_op_state} = \text{MOS_pending_mem_read sr c}.
\]

Show \( t' \) in `enumerate_transitions_of_system s`

Since \( \text{pop_ss_accept_event_cand} \) holds in \( s_0 \), it must also hold in \( s \) because \( s.\text{storage_subsystem} = s_0.\text{storage_subsystem} \). Remains to check that \( tt' \) in \text{enumerate_transitions_of_instruction in s}.

By definition of `enumerate_fetch_transitions_of_instruction` and \text{enumerate_initial_fetch_transitions_of_thread transition t} does not change \( \text{inst'} \) so that \( tt' \) is still enabled in \( s \). Therefore \( t' \) in `enumerate_transitions_of_system s`.

Show \( t \) in `enumerate_transitions_of_system s'`

By definition of `enumerate_transitions_of_system`, `pop_ss_accept_event_action`, and \text{enumerate_transitions_of_instruction transition t'} only updates \( \text{inst'} \) and the storage subsystem state.

Then \( t \) in `enumerate_transitions_of_system s'`, since \( t' \) does fetch any instructions, remove instructions from the instruction tree or change any instruction’s `regs_out` or `reg_writes` fields.

By definition of `enumerate_transitions_of_instruction` and \text{enumerate_transitions_of_system t} and \( t' \) update separate parts of the system state so that \( \text{system_state_after_transition s t'} \cong \text{system_state_after_transition s' t} \).
5.2. Case \( t' \) is of form \( T_{\text{commit mem write}} \) \( ws \ t \)

Then by definition of \( \text{enumerate transitions of system} \) the condition
\[
\text{pop ss accept event cand (FWrite write)} \text{ holds in } s_0,
\]

\[
\text{tlazy} = \text{(iolid}',(T_{\text{interact lazy}} (T_{\text{commit mem write}} ws t'), \text{ist}'))
\]

in \( \text{enumerate transitions of instruction inst}' \)
in state \( s_0 \) and \( \text{inst}'.\text{micro op state} = \text{MOS potential mem write wk ws} \), and
\( \text{pop commit cand} \) holds.

**Show \( t' \) in \( \text{enumerate transitions of system} \ s \)**

Since \( s.\text{storage subsystem} = s_0.\text{storage subsystem} \) condition \( \text{pop ss accept event cand} \)
also holds in \( s \). Therefore only need to check that \( \text{tlazy} \) is in
\( \text{enumerate transitions of thread} \) in state \( s \).

Check that \( \text{find committed writes} \) returns the same value: follows from the fact that \( t \) does not
change the \( \text{committed mem writes} \) field of any instruction.

Check that \( \text{pop commit cand} \) still holds: check the following requirements:

1. \( \text{commitDataflow inst context} \)
2. \( \text{commitControlflow inst context} \)
3. \( \text{pop commit mem access cand} \)

1 to 3 all check conditions for instructions po-before \( \text{inst}' \). Since \( t \) does not change \( \text{inst}' \)'s
prefix, all of them still hold in \( s \). Since \( \text{pop commit cand} \) also holds in \( s \) transition
\( \text{tlazy} \) in \( \text{enumerate transitions of thread} \) in \( s \).

**Show \( t \) in \( \text{enumerate transitions of system} \ s' \)**

By definition of \( \text{enumerate transitions of instruction} \) and \( \text{pop commit mem store action} \)
transition \( t' \) only changes instructions that are restarted and \( \text{inst}' \). \( t' \) does fetch any
instructions, remove instructions from the instruction tree. The proof of
\( t \) in \( \text{enumerate transitions of system} \ s' \) is the same as in Case 2 of the proof of Theorem 2.

Therefore \( t \) in \( \text{enumerate transitions of system} \ s' \).

Because \( t \) and \( t' \) update separate parts of the system state:
\( \text{system state after transitions} \ s \ t' \ \cong \ \text{system state after transitions} \ s' \ t \).

5.3. Case \( tt' \) is of form \( T_{\text{commit barrier}} b t \)
Then by definition of enumerate_transitions_of_system and enumerate_transitions_of_instruction condition pop_ss_accept_event_cand (FBarrier b) holds in s_0.

\[
\text{tlazy} = (\text{ioid}', (T_{\text{interact_lazy}} (T_{\text{commit_barrier}} b t), \text{ist}'))
\]

in enumerate_transitions_of_instruction in s_0, inst's micro_op_state is MOS/plain with interpreter outcome Barrier bk is' and bk ≠ ISB, and pop_commit_cand holds in s_0.

\[
\text{Show } t' \text{ in enumerate_transitions_of_system } s
\]

As s.storage_subsystem = s_0.storage_subsystem condition pop_ss_accept_event_cand (FBarrier b) also holds in s. Remains to show tlazy in enumerate_transitions_of_instruction in state s, which because inst' is not changed by t reduces to showing that pop_commit_cand still holds in s. The conditions require pop_commit_cand are

1. commitDataflow
2. commitControlflow
3. pop_commit_barrier_cand

The proof that 1 and 2 still hold is the same as in Case 5.2. Remains 3: 3 for non-ISB barriers only requires instructions of particular kinds be committed. Since t does not "uncommit" any instructions this still holds in s.

Therefore tlazy in enumerate_transitions_of_instruction in state s and thus t' in enumerate_transitions_of_system s.

\[
\text{Show } t \text{ in enumerate_transitions_of_system } s'
\]

By definition of enumerate_transitions_of_instruction and pop_commit_barrier_action transition t' only updates inst' s micro_op_state, and changes the committed_barriers and committed fields, does not fetch new instructions or remove instructions from the instruction tree. The proof of t in enumerate_transitions_of_system s' now goes as in Case 2 of the proof of Theorem 2.

Because t and t' update separate parts of the system state:

\[
\text{system_state_after_transitions s t' } \equiv \text{system_state_after_transitions s' t'.}
\]

\[
6. \text{Case } t' = T\text{\_only\_trans tid' ioid' ids' tt'}
\]
For the cases for which $p_1 \ t'$ holds the proof is the same as that of Case 4.2, since we always proved $t$ in enumerate_transitions_of_system, $t'$ in enumerate_transitions_of_system and system_state_after_transition $s \ t' \equiv$ system_state_after_transition $s' \ t$.

Remains the proof for those cases of $t'$ for which $p_1 \ t'$ doesn't hold:

- $T_{\text{only_trans}} \ ioid' \ tid' \_ T_{\text{register_write}}$ for registers LR or CTR,
- $T_{\text{only_trans}} \ ioid' \ tid' \_ T_{\text{POP_commit_mem_write_exclusive_fail}}$
- $T_{\text{only_trans}} \ ioid' \ tid' \_ T_{\text{POP_mem_write_footprint}}$
- $T_{\text{only_trans}} \ ioid' \ tid' \_ T_{\text{commit_simple}}$

Two cases:

1. $t$ enabled by enumerate_initial_fetch_transitions_of_thread or
2. $t$ enabled by enumerate_fetch_transitions_of_instruction

If $t$ is enabled by enumerate_initial_fetch_transitions_of_thread, then by definition of enumerate_initial_fetch_transitions_of_thread: tid $\neq$ tid', since otherwise $t$ would not be enabled.

Then, from $s.\text{storage_subsystem} = s_0.\text{storage_subsystem}$ and $s.\text{thread_states} \ tid' = s_0.\text{thread_states} \ tid'$ follows $t'$ in enumerate_transitions_of_system $s$; from $s'.\text{program_memory} = s_0.\text{program_memory}$ and $s'.\text{thread_states} \ tid = s_0.\text{thread_states} \ tid$ follows $t$ in enumerate_transitions_of_system $s'$.

Because $t$ and $t'$ update separate parts of the system state:

system_state_after_transition $s \ t' \equiv$ system_state_after_transition $s' \ t$.

Now assume $t$ is enabled by enumerate_fetch_transitions_of_thread. Then $t$ in enumerate_fetch_transitions_of_instruction iic for some iic. Let $\text{inst} = \text{iic.iic_instance}$.

Proof by case analysis on $t'$:

### 6.2. T_{\text{POP_commit_mem_write_exclusive_fail}}

The proof of $t'$ in enumerate_transitions_of_system $s$ is the same as in Cases 6.1 and 6.2.2 of the proof of Theorem 1, the proof of $t$ in enumerate_transitions_of_system $s'$ and system_state_after_transitions $s \ t' \equiv$ system_state_after_transition $s' \ t$ is the same as in Case 2 of the proof of Theorem 2.

### 6.3. T_{\text{only T_POP_mem_write_footprint}}
The proof of \( t' \) in enumerate_transitions_of_system \( s \) is the same as in Case 6.2.3 of the Proof of Theorem 1, the proof of \( t \) in enumerate_transitions_of_system \( s' \) and of system_state_after_transition \( s \ t' \equiv \text{system_state_after_transition} \ s' \ t \) is the same as in Case 2 of the proof of Theorem 2.

### 6.4. \text{T\_only (T\_commit\_simple Nothing Nothing t'),ist')}

There are two cases:

1. \( t' \) is enabled by \( \text{inst'} \)'s micro_op_state Barrier or Done. In the case of \( \text{inst'}.\text{micro_op_state} = \text{Barrier} \) the proof for \( t' \) in enumerate_transitions_of_system \( s \) is the same as in Case 6.2.5 of the proof of Theorem 1, the proof of \( t \) in enumerate_transitions_of_system \( s' \) and system_state_after_transition \( s \ t' \equiv \text{system_state_after_transition} \ s' \ t \) is the same as in Case 2 of the proof of Theorem 2.

   Therefore, assume \( t' \) enabled by \( \text{inst'}.\text{micro_op_state} = \text{Done} \).

   **Show \( t' \) in enumerate_transitions_of_system \( s \)**

   Since \( t \) does not change \( \text{inst'} \), only need to show that pop_commit_cand still holds in \( s \). But since pop_commit_cand only refers to instructions in \( \text{inst'} \)'s prefix and \( t \) does not change \( t' \)'s prefix, \( t \) preserves pop_commit_cand so that \( t' \) in enumerate_transitions_of_system \( s \).

   Now there are two cases: potential_fetch_addresses in \( s' \) returns the same value or a different value than in \( s_0 \).

   **6.4.1. potential_fetch_addresses in \( s' \) returns the same value**

   Since \( t' \) does not fetch any instruction out_standing_fetch_addresses remains the same, and since \( t' \) does not change any instruction's regs_out or regWrites fields fetch_transitions_of_address returns the same value as well. Therefore \( t \) in enumerate_transitions_of_system \( s' \) and as \( t \) and \( t' \) update separate parts of the system state: system_state_after_transition \( s \ t' \equiv \text{system_state_after_transitions} \ s' \ t \).

   **6.4.2. potential_fetch_addresses in \( s' \) return a different value than in \( s \)**

   Because \( t' \) does not change the instruction kind of \( \text{inst'} \), does not change its regWrites or nias fields, is_stop_fetch_instruction the find_reg_read and inst.nias fields must have the same values in \( s' \) and in \( s_0 \). Therefore \( t' \) must have committed \( \text{inst'} \) and
potential_fetch_address returns \{ \text{next_address_of_committed_instruction } \text{iic} \}. If addr is the member of this set, the proof goes as in 6.4.1, so assume addr is not in this set.

Then \( t' \) is the commit of a branch instruction with a now-determined successor address that is different from addr: By definition of \text{commit_simple_action} transition \( t' \) deletes any subtrees with address different from \text{next_address_of_committed_instruction}. Therefore taking \( t' \) in state \( s \) removes inst from the instruction tree and undoes the progress made by transition \( t \), so that \( \text{system_state_after_transition } s \ t' \equiv s' \).