The Flowing and POP Models
(supplementary material for Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA)

1. The Flowing Model in Detail
1.1 The Storage Subsystem/Thread Interface
The model is expressed in terms of read, write, and barrier requests. Read and write requests include the kind of the memory access (e.g. exclusive, release, acquire), the ID of the issuing thread and the memory access address. Write requests also include a value. Barrier requests include the kind of the barrier (e.g. sy, ld, st) and the issuing thread ID.

When we refer to a write or read request without mentioning the kind of request we mean the request can be of any kind.

The storage subsystem and a thread subsystem can exchange messages through synchronous transitions:

- a non-exclusive write request can be passed to the storage subsystem by a thread Commit store instruction transition coupled with a storage subsystem Accept request transition;
- if r is an exclusive write request can be passed to the storage subsystem by a thread Commit store-exclusive transition coupled with a storage subsystem Accept a successful write-exclusive (from segment or memory) transition;
- a (memory) barrier request can be passed to the storage subsystem by a thread Commit barrier instruction transition coupled with a storage subsystem Accept request transition;
- a read request can be passed to the storage subsystem by a thread Issue read request transition coupled with a storage subsystem Accept request transition; and
- a read response can be returned from the storage subsystem to a thread by a storage subsystem Satisfy read from segment or Satisfy read from memory transition coupled with a thread Satisfy memory read from storage response transition.

1.2 Storage Subsystem States
The Flowing storage subsystem state includes:

- thread_ids, the set of thread IDs that exist in the system;
- topology, a data structure describing how the segments are connected to each other;
- thread_to_segment, a map from thread IDs to segments, associating each thread with its leaf segment;
- buffers, a map from segments to list of requests associating each segment with the requests queued in that segment;
- reordered, a set of request pairs that have been reordered w.r.t. each other; and
- memory, a map from memory addresses to the most recent write request to that address to reach memory.

1.3 Storage Subsystem Transitions
Accept request A request r from thread r.tid can be accepted if:
1. r has not been accepted before (i.e. r is not in buffers); and
2. r.tid is in thread_ids.

Action: add r to the top of buffers(thread_to_segment(r.tid)).

Flow request A request r can flow from segment s1 to s2 if:
1. r is at the bottom of buffers(s1);
2. s1 is a child of s2 in topology;
3. [s1] if r is a write to address raddr, s1 is not blocked (§1.4) by any write-exclusive to raddr; and
4. [s1] if r.kind is dmb sy or dmb st, s1 is not blocked by any write-exclusive.

Action:
1. remove r from buffers(s1);
2. add r to the top of buffers(s2); and
3. remove from reordered any pair that contains r.

Reorder requests Two requests r_new, r_old that appear consecutively in buffers(s) (r_new nearer to the top) can be reordered if:
1. (r_new, r_old) does not appear in reordered (i.e. they have not been reordered (in segment s) with each other before (preventing live lock)); and
2. r_new and r_old satisfy the reorder condition (§1.4).

Action:
1. switch the positions of r_new and r_old in buffers(s); and
2. record the reordering of r_new and r_old (by adding the pair (r_new, r_old) to reordered).

Satisfy read from segment Two requests, r_read, r_write, can generate a read response to thread r_read.tid if:
1. r_read is a read request that has not been satisfied yet;
2. r_write is a write request;
3. r_read, r_write appear consecutively in buffers(s) for some segment s, with r_read closer to the top (newer);
4. r_read and r_write are to the same address; and
5. [s] if r_read.kind is read-acquire, r_write.kind is not write-release.

Action:
1. send a read response for request r_read to thread r_read.tid, containing r_write; and
2. [s] if r_read.kind is read-acquire, switch the positions of r_read and r_write in buffers(s) and mark r_read as satisfied; else
3. remove r_read.

Satisfy read from memory A read request r_read from thread r_read.tid can generate a read response to thread r_read.tid if r_read is at the bottom of buffers(s), where s is the root segment in topology, and r_read has not been satisfied yet.

Action:
1. send a read response for request r_read to thread r_read.tid, containing the write stored in memory for the address r_read.addr; and
2. remove r_read.
Flow satisfied read to memory A satisfied read acquire request $r_{read}$ can be discarded if: $r_{read}$ is at the bottom of $buffers(s)$, where $s$ is the root segment in topology. Action: remove $r_{read}$.

Flow write to memory The write request $r_{write}$ can be stored into memory if: $r_{write}$ is at the bottom of $buffers(s)$, where $s$ is the root segment in topology. Action:
1. update memory to map the address $r_{write}.addr$ to $r_{write}$; and
2. remove $r_{write}$ from $buffers(s)$ and reordered.

Flow barrier to memory A barrier request $r_{barr}$ can be discarded if: $r_{barr}$ is at the bottom of $buffers(s)$, where $s$ is the root segment in topology. Action: remove $r_{barr}$.

Accept a successful write-exclusive (from segment) A write-exclusive request $r_{write}$ from thread $r_{write}.tid$ with an accompanying write request $r_{write}'$ that was read by a po-previous load-exclusive can be accepted and succeed if:
1. all the conditions of Accept request are satisfied for $r_{write}$;
2. $r_{write}$ and $r_{write}'$ are to the same address;
3. exists $0 \leq i \leq n$ such that $r_{write}'$ is in $buffers(si)$;
4. for all $i < j \leq n$, segment $sj$ is not blocked by a write-exclusive to the address $r_{write}.addr$;
5. let $rs$ be the set of requests $\bigcup_{0 \leq j \leq n} buffers(sj)$ unioned with all the requests in $buffers(s)$ above $r_{write}'$;
6. there is no write request in $rs$ to the same address as $r_{write}$ from a different thread;
7. there is no dmb sy barrier request in $rs$; and
8. $[s0,s1,...,sn]$ there is no dmb st barrier request in $rs$.

where $s0,s1,...,sn$ is the path of segments in topology from the root segment $s0$ to the leaf segment $sn$ associated with thread $r_{write}.tid$. Action: the same as Accept request.

Accept a successful write-exclusive (from memory) A write-exclusive request $r_{write}$ from thread $r_{write}.tid$ with an accompanying write request $r_{write}'$ that was read by a po-previous load-exclusive can be accepted and succeed if:
1. all the conditions of Accept request are satisfied for $r_{write}$;
2. $r_{write}$ and $r_{write}'$ are to the same address;
3. $r_{write}'$ is in memory;
4. for all $0 < j \leq n$, segment $sj$ is not blocked by a write-exclusive to the address $r_{write}.addr$;
5. there is no write request in $rs$ to the same address as $r_{write}$ from a different thread;
6. there is no dmb sy barrier request in $rs$; and
7. $[s0,s1,...,sn]$ there is no dmb st barrier request in $rs$.

where $s0,s1,...,sn$ is the path of segments in topology from the root segment $s0$ to the leaf segment $sn$ associated with thread $r_{write}.tid$, and let $rs$ be the set of requests $\bigcup_{0 \leq j \leq n} buffers(sj)$. Action: the same as Accept request.

1.4 Auxiliary Definitions for Storage Subsystem

Immediate predecessor Intuitively, the immediate predecessor of a write-exclusive is the write associated the load-exclusive read from, which must be kept coherence-immediately-before the write-exclusive to the address $r_{write}.addr$.

Formally we say a write $w'$ in segment $s_{w'}$ (or memory) is the immediate predecessor of a write-exclusive in segment $s_{w}$ if $s_{w'}$ is a predecessor of $s_{w}$ (in topology), $w'$ and $w$ are to the same address, and all the writes queued between $w$ and $w'$ are to different addresses than that. Notice that a write-exclusive can have at most one immediate predecessor (in fact we will make sure it has exactly one immediate predecessor).

As write-exclusive must be the coherence-immediate-successor of its immediate predecessor, no other write to the same address can be allowed to flow in between them. We say a write-exclusive in segment $s_{w}$ is blocking segment $s$ if $s_{w}$ is a successor of $s$'s parent, but is not a successor of $s$ (or $s$ itself), and the immediate predecessor of $w$ is in a segment that is predecessor of $s$.

Reorder condition Two requests $r_{new}$ and $r_{old}$ are said to meet the reorder condition if:
1. $[s_{r_{new}}]$ if $r_{new}$ is a read-acquire and $r_{old}$ is a write-release, they are to a different address;
2. $[s_{r_{new}}]$ $r_{new}$ is not a write-release;
3. $[s_{r_{new}}]$ if $r_{old}$ is a read-acquire then $r_{new}$ and $r_{old}$ originated from different threads;
4. $[s_{r_{new}}]$ if $r_{new}$ is a read-acquire and $r_{old}$ is a write-release, $r_{new}$ and $r_{old}$ originated from different threads;
5. $[s_{r_{new}}]$ at least one of $r_{new}, r_{old}$ is not a memory barrier (of any kind);
6. $[s_{r_{new}}]$ if $r_{new}$ is dmb ld then $r_{old}$ is not a read from the same thread;
7. $[s_{r_{new}}]$ if $r_{old}$ is dmb ld then $r_{new}$ is not a memory access from the same thread;
8. $[s_{r_{new}}]$ if $r_{new}$ is dmb st then $r_{old}$ is not a write from the same thread;
9. $[s_{r_{new}}]$ if $r_{old}$ is dmb st then $r_{new}$ is not a write from the same thread;
10. neither one of $r_{new}, r_{old}$ is a dmb sy; and
11. if both $r_{new}$ and $r_{old}$ are memory access requests, and none of them is a satisfied read, they are to different addresses.

1.5 Thread States

The state of a simple hardware thread includes:

- $thread_id$, a unique identifier of the thread;
- $register_data$, general information about the available registers, including name, bit width, initial bit index and direction;
- $initial_register_state$, the initial values for each register;
- $initial_fetch_address$, the initial fetch address for this thread;
- $instruction_tree$, a data structure holding the instructions that have been fetched in program order; and
- $reading_order$, a list of read requests in the order they were issued to the storage subsystem.

1.6 Instruction State

Each instruction in the instruction_tree has a state including:

- $program_loc$, the memory address where the instruction’s opcode resides;
- $instruction_kind$, the kind of the instruction (e.g. load-acquire);
- $regs_in$, the input registers, for dependency calculation;
- $regs_out$, the output registers, for dependency calculation;
- $reg_reads$, accumulated register reads;
- $reg_writes$, accumulated register writes;
- $mem_read$, one of none, pending a, requested a, write_read_from w, where a is a memory address and w is a memory write;
- $mem_write$, one of none, potential_address a, pending w, committed w, where a is a memory address and w is a memory write;
- $committed$, set to true when the instruction can no longer affect the memory model (the Sail interpreter might still need to complete the execution of the ISA definition);
- $finished$, set to true only after committed is set to true and the Sail interpreter has finished executing the ISA definition; and
- $micro_op_state$, the meta-state of the instruction, one of plain interpreter_state (ready to make a Sail interpreter transition from interpreter_state), pending_mem_read_read_cont (ready to perform a read from memory) or potential_mem_write_write_cont (ready to perform a write to memory) where
read_cont is a Sail interpreter continuation that given the value read from memory returns the next interpreter state, and write_cont is a Sail interpreter continuation that given a boolean value (that indicates if the store was successful) returns the next interpreter state.

Instructions that have been fetched (i.e. in instruction_tree) and have a finished value of false are said to be in-flight. Instructions that have a committed value of true are said to be committed. Load instructions with the value requested for mem_read are said to have an outstanding read request, and if they have the value write_read_from they are said to be satisfied. Store instructions with mem_write = pending w are said to have a pending write. We say instruction i has fully determined address if all instructions that write to input registers of i are committed. We say i is fully determined if all instructions that write to input registers of i are committed.

1.7 Thread Transitions

Sail interpreter step An instruction i in meta-state plain interpreter_state can perform an interpreter step as follows:

- if interpreter_state indicates a memory-read event: set i.mem_read to pending interpreter_state.read_addr and update the instruction meta-state to pending mem_read interpreter_state.read_cont;
- if interpreter_state indicates a memory-write-address event: set mem_write to potential_address interpreter_state.write_addr and update the instruction meta-state to plain interpreter_state.next;
- if interpreter_state indicates a memory-write-value event: set mem_write to pending w (where w is a write request with the value interpreter_state.write_value and the address that was in mem_write before) and update the instruction meta-state to potential_mem_write interpreter_state.write_cont;
- if interpreter_state indicates a register-read event: look in instruction_tree for the most recent po-previous instruction, i’ that has interpreter_state.reg in i’.regs_out. If interpreter_state.reg is not in i’.reg_write the transition is disabled. Otherwise, add interpreter_state.reg to i.reg_reads and update the meta-state of i to plain interpreter_state.next;
- if interpreter_state indicates a register-write event: add interpreter_state.reg to i.reg_writes and update the meta-state of i to plain interpreter_state.next; and
- if interpreter_state indicates an internal step: update the meta-state of i to plain interpreter_state.next.

Fetch instruction An instruction i, from address loc, can be fetched, following its program-order predecessor i’ if:

1. i’ is in instruction_tree;
2. loc is a possible next fetch address for i’ according to the ISA model;
3. none of the successors of i’ in instruction_tree are from loc; and
4. i is the instruction of the program at loc.

The possible next fetch addresses address speculation past calculated jumps and conditional branches; they are defined as:

- for a non-branch/jump instruction, the successor instruction address;
- for a jump to constant address, that address;
- for a conditional branch, the possible addresses for a jump1 together with the successor; and

for a jump to an address which is not yet fully determined (i.e., where there is an uncommitted instruction with a dataflow path to the address), any address. This is (necessarily) approximated in our implementation, c.f. §4.5.

Action: construct an initialized instruction instance, including static information available from the ISA model such as instruction_kind, reg_in, reg_out, and add it to instruction_tree as a successor of i’.

This is an internal action of the thread, not involving the storage subsystem, as we assume a fixed program rather than modelling fetches with memory reads; we do not model self-modifying code.

Issue read request An in-flight instruction i in meta-state pending_mem_read_cont can issue a read request of address a to the storage subsystem if:

1. i mem_read has the value pending a, i.e., any other reads with dataflow path to the address have been satisfied, though not necessarily committed, and any arithmetic on such a path is completed;
2. all po-previous dmb sy and isb instructions are committed;
3. i is po-before all po-previous dmb ld instructions are committed;
4. i is not-load-acquire instruction, all po-previous store-release instructions are committed; and
5. all in-flight po-previous load-acquire instructions have outstanding read requests or were already satisfied.

Action:

1. send a read-request to the storage subsystem;
2. change i mem_read to requested a; and
3. update read_issuing_order to note that the read was issued last.

Satisfy memory read from storage response A read response with write w, for read request from instruction i in meta-state pending_mem_read_cont can always be received. Action:

1. if i mem_read does not have the value requested the response is ignored (this can happen when the read is satisfied by write forwarding while waiting for the response); else
2. if there exists a po-previous load instruction to the same address that was issued after i (i.e., issued out of order) and was satisfied by a write that is not w, set i mem_read to pending; else
3. for every in-flight instruction i’ that is po-after i and has read from a write to the same address as i that is not w and not po-successor of i, restart i’ and its data flow dependents;
4. change i mem_read to write_read_from w; and
5. update the meta-state of i to plain (read_cont w.value).

Satisfy memory read by forwarding an in-flight write A pending memory write w from an in-flight store instruction i’ can be forwarded directly to a load instruction i in meta-state pending_mem_read_cont if:

1. i is not load-acquire;
2. i is not-load-exclusive;
3. i mem_read has the value pending w.address or requested w.address; and
4. i’ is po-before i, there is no other store instruction to the same address in between them, and there is no other load instruction in between them that has read from a different store instruction to the same address.

Action:

1. for every in-flight instruction i” that is po-after i and has read from a write to the same address as i that is not w and not po-successor of i, restart i” and its data flow dependents;
2. change i mem_read to write_read_from w; and
3. update the meta-state of i to plain (read_cont w.value).

Commit store instruction A store (not exclusive) instruction i in meta-state potential_mem_write write_cont and with pending write...
w can be committed if:
1. i is fully determined;
2. all po-previous conditional branches are committed;
3. all po-previous dmb sy and isb instructions are committed;
4. [dmb st] all po-previous dmb ld instructions are committed;
5. [isb st] all po-previous load-acquire instructions are committed;
6. [dmb ld] all po-previous store-release instructions to the same address are committed;
7. [dmb ld] if i is a store-release, all po-previous memory access instructions are committed;
8. [init status] all po-previous store-exclusive to the same address are committed;
9. [dmb ld] all po-previous dmb st are committed;
10. all po-previous memory access instructions have a fully determined address; and
11. all po-previous instructions that read from the same address have either an outstanding read request or are satisfied, and cannot be restarted (see §1.8).

Action:
1. restart any in-flight loads (and their dataflow dependants) that:
   (a) are po-after i and have read from the same address, but from a different write and where the read could not have been by forwarding an in-flight write that is po-after i; or
   (b) have issued a read request that has not been satisfied yet.
2. if there is no committed po-following store to the same address, send a write request to the storage subsystem;
3. record the store as committed (i.e. set i.mem_write to committed and committed to true); and
4. update the meta-state of i to plain (write_cont true).

Commit failed store-exclusive instruction A store-exclusive instruction i in meta-state potential_mem_write write_cont can always be committed as “failed”. Action:
1. restart any in-flight loads (and their dataflow dependants) that were satisfied by a write forward from i;
2. record the store as committed (i.e. set i.mem_write to none and set committed to true); and
3. update the meta-state of i to plain (write_cont false).

Commit store-exclusive A store-exclusive instruction i in meta-state potential_mem_write write_cont with pending write w and an associated load-exclusive instruction i’ with i’.mem_read = write_read_from w’. can be committed if:
1. i’ is committed and po-before i;
2. there is no other store-exclusive or load-exclusive instructions po-between i’ and i; and
3. all the conditions of Commit store instruction are met.

If i’ is to the same address as i the following action is applied, otherwise the action of Commit store instruction is applied and the destination register is set to 0. Action:
1. send a write-exclusive request (with w as the requested write and w’ as the required coherence predecessor) to the storage subsystem and receive the corresponding fail/success response, setting the destination register accordingly (0 for success and 1 for fail);
2. if the write succeeded, perform the action of Commit store instruction; else
3. apply the action of Commit failed store-exclusive instruction.

Commit barrier instruction A barrier instruction i in meta-state plain interpreter_state can be committed if:
1. interpreter_state indicates a barrier event is pending;
2. all po-previous conditional branches are committed;
3. [dmb ld] if i is dmb ld, all po-previous load instructions are committed;
4. [dmb st] if i is dmb st, all po-previous store instructions are committed;
5. all po-previous barriers (of any kind) are committed;
6. if i is isb, all po-previous memory access instructions have a fully determined address; and
7. if i is dmb sy, all po-previous memory access instructions are committed.

Action:
1. record the barrier as committed (i.e. set committed to true); and
2. update the meta-state of i to plain interpreter_state.next.

Finish instruction An in-flight instruction i in meta-state plain interpreter_state can be finished if:
1. interpreter_state indicates the execution of the ISA definition has finished;
2. if i is a load instruction:
   (a) all po-previous dmb sy and isb instructions are committed;
   (b) [dmb st] all po-previous dmb ld instructions are committed;
   (c) [dmb ld] all po-previous load-acquire instructions are committed;
   (d) let i’ be the store instruction to the same address as i that appears last in program order before i:
      i. if i’ was forwarded to rand is not a store-exclusive, i’ is fully determined, otherwise i’ is committed;
      ii. all memory access instructions, po-between i’ and i, have a fully determined address; and
      iii. all load instructions to the same address as i, that are po-between i’ and i, are committed.
   (e) [mem write] if i is a load-acquire, all po-previous store-release instructions are committed;
3. i is fully determined; and
4. all po-previous conditional branches are committed.

Action:
1. if i is a branch instruction, abort any untaken path of execution, i.e., any in-flight instruction that are not reachable by the branch taken in instruction_tree; and
2. record the instruction as finished, i.e., set finished (and committed) to true.

1.8 Auxiliary Definitions for Thread Subsystem

Restart condition To determine if instruction i might be restarted we use the following recursive condition: i is an in-flight instruction and at least one of the following holds,

1. there exists an in-flight store instruction s such that applying the action of the Commit store instruction transition to s will result in the restart of i;
2. there exists an in-flight load instruction l such that applying the action of the Satisfy memory read from storage response transition to l will result in the restart of i (even if l is already satisfied);
3. i has an outstanding read request that has not been satisfied yet, and there exists a load instruction po-before i that has an outstanding read request to the same address (maybe already satisfied) after i; or
4. there exists an in-flight instruction i’ that might be restarted and an output register of i’ feeds an input register of i, or i’ is a load-acquire and i is a load and i’ is po-before i.
2. The POP Model

2.1 The Storage Subsystem/Thread Interface

The storage subsystem and a thread subsystem can exchange messages through synchronous transitions:

- a non-exclusive write request can be passed to the storage subsystem by a thread Commit store instruction transition coupled with a storage subsystem Accept request transition;
- an exclusive write request can be passed to the storage subsystem by a thread Commit store-exclusive transition coupled with a storage subsystem Accept a successful write-exclusive transition;
- (memory) barrier request can be passed to the storage subsystem by a thread Issue read request transition coupled with a storage subsystem Accept request transition;
- a read request can be passed to the storage subsystem by a thread Commit barrier instruction transition coupled with a storage subsystem Accept request transition;
- a read response can be passed from the storage subsystem to a thread by a storage subsystem Send read-response to thread transition coupled with a thread Satisfy memory read from response transition.

In addition to the above, when a load instruction is restarted in the thread subsystem, all its read-requests are removed from the storage subsystem.

2.2 Storage Subsystem State

The POP storage subsystem state includes:

- thread_ids, the set of thread IDs that exist in the system;
- requests_seen, the set of requests (memory read/write requests and barrier requests) that have been seen by the subsystem;
- order_constraints, the set of pairs of requests from requests_seen. The pair (r_old, r_new) indicates that r_old is before r_new (r_old and r_new might be to different addresses and might even be of different kinds); and
- requests_propagated_to, a map from thread IDs to subsets of requests_seen, associating with each thread the set of requests that has propagated (potentially visible) to it.

2.3 Storage Subsystem Transitions

Accept request A request r_new, that is not store-exclusive, from thread r_new.tid can be accepted if:

1. r_new has not been accepted before (i.e., r_new is not in requests_seen); and
2. r_new.tid is in thread_ids.

Action:

1. add r_new to requests_seen;
2. add r_new to requests_propagated_to(r_new.tid); and
3. update order_constraints to note that r_new is after every request r_old that has propagated to thread r_new.tid, and r_new and r_old do not meet the Flowing reorder condition (see Reorder condition).

Propagate request to another thread The storage subsystem can propagate request r (by thread tid) to another thread tid’ if:

1. r has been seen before (i.e., r is in requests_seen);
2. r has not yet been propagated to thread tid’;
3. every request that is before r in order_constraints has already been propagated to thread tid’;
4. if r is a write, all the write exclusive requests that have propagated to tid’ but not to tid, for which the immediate predecessor write to the same address is order_constraints-before r, are to a different address; and
5. if r is a dmb sy or dmb st, there are no write exclusive requests that have propagated to tid’ but not to tid for which the immediate predecessor write to the same address is order_constraints-before r.

Action:

1. add r to requests_propagated_to(tid’);
2. update order_constraints to note that r is before every request r_new that has propagated to thread tid’ but not to thread tid, where r_new and r do not meet the Flowing reorder condition (see Reorder condition).

Send read-response to thread The storage subsystem can send a read-response for read request r_read that has not been satisfied yet to thread r_read.tid containing the write request r_write if:

1. r_write and r_read are to the same address;
2. r_write and r_read have been propagated to (exactly) the same threads;
3. r_write is order_constraints-before r_read;
4. any request that is order_constraints-between r_write and r_read has been fully-propagated (§2.4) and is to a different address; and
5. if r_read is a read-acquire and r_write is a write-release then r_write must be fully-propagated.

Action:

1. send thread r_read.tid a read-response for r_read containing r_write;
2. if r_read.kind is read-acquire and there are no other requests between r_read and r_write in order_constraints,
3. switch the positions of r_read and r_write in order_constraints and mark r_read as satisfied; else
4. remove r_read.

Accept a successful write-exclusive A write-exclusive request r_write from thread r_write.tid with an accompanying write request r_write’ that was read by a po-previous load-exclusive can be accepted and succeed if:

1. r_write meets the condition of Accept request;
2. r_write’ is in requests_seen;
3. the write r_write is to the same address as r_write’;
4. r_write’ is not the immediate predecessor of a different write-exclusive;
5. every immediate predecessor write that is order_constraints-after r_write’ and has propagated to thread r_write.tid is to a different address than r_write; and
6. for every request r that has been propagated to thread r_write.tid and is order_constraints-after r_write’:
   (a) if rtid is different then r_write.tid then r is not a write to the same address as r_write; and
   (b) if r is a dmb sy or dmb st then r is fully-propagated.

Action: the same as Accept request.

2.4 Auxiliary Definitions for Storage Subsystem

Fully propagated Request r is said to be fully-propagated if it has been propagated to all threads and so has every request that is order_constraints-before it.

Removing read request When a read request is removed from the storage subsystem, due to restart of the instruction in the thread subsystem or satisfaction, first order_constraints is restricted to exclude the request; it is then further restricted by applying the reorder condition to each pair of ordered requests and removing pairs that can be reordered; finally the transitive closure of the result is calculated.
Store-exclusive immediate predecessor The following is an important invariant that guarantees the correct behaviour of load/store-exclusive. Every write-exclusive has exactly one immediate predecessor write to the same address in order_constraints; moreover, that immediate predecessor has exactly one immediate successor write to the same address in order_constraints (namely the write-exclusive).