ARMv8-A system semantics: instruction fetch in relaxed architectures

Ben Simner[1], Shaked Flur[1], Christopher Pulte[1], Alasdair Armstrong[1], Jean Pichon-Pharabod[1], Luc Maranget[2], and Peter Sewell[1]

[1] University of Cambridge, UK
[2] INRIA Paris, France

Abstract. This document gives a prose description of the Flat operational model, as formally defined in its Lem definition. This is part of the supplementary material for "ARMv8-A system semantics: instruction fetch in relaxed architectures8".

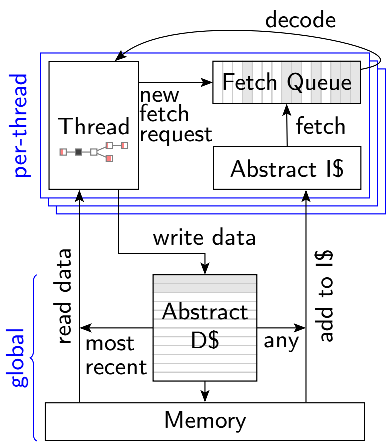
1 An operational model for MCA ARMv8

To help reading this document we have colour-coded some text as follows:

– [release/acquire] Release/Acquire instructions
– [exclusive] Exclusive instructions
– [dmb ld/dmb st] `dmb ld` and `dmb st` instructions
– [ifetch] Instruction fetch and cache maintenance instructions

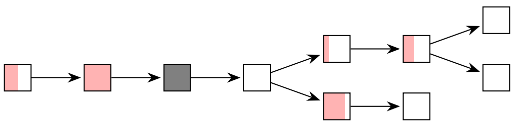
The operational model is expressed as a state machine, with states that are an abstract representation of hardware machine states. We first introduce the model states and transitions informally.

Model states A model state consists just of a shared memory and a tuple of thread model states:



The shared memory state effectively just records the most recent write to each location. To handle load/store-exclusives, the memory is extended with a map (the exclusives map) from read requests to sets of write slices, that associates a read request of a load-exclusive with the write slices it read from (excluding writes that have been forwarded to the read and have not reached memory yet). To handle instruction fetching, the shared memory is extended with a data cache buffer of all the writes still visible to instruction fetches. Each thread is extended with an instruction cache that can be fetched from and fetch queue of buffered pre-fetched instructions.

Each thread model state consists principally of a list or tree of instruction instances, some of which have been finished, and some of which have not. Below we show an example for a thread that is executing 10 instruction instances. Some (grey) are finished; others (pink) have run some but perhaps not all of their instruction semantics; instructions are not necessarily atomic. Those with multiple children are branch instructions with multiple potential speculative successors being explored simultaneously.



Non-finished instruction instances can be subject to restart, e.g. if they depend on an out-of-order or speculative read that turns out to be unsound. The finished instances are not necessarily contiguous: in the example, i_3 and i_4 are finished even though i_2 is not, which can only happen if they are sufficiently independent. Instruction instances i_5 and i_9 are conditional branches for which the thread has fetched multiple possible successors. When a conditional branch is finished, any un-taken alternative paths are discarded, and instruction instances that follow (in program order) a non-finished conditional branch cannot be finished until that conditional branch is. One can choose whether or not to allow simultaneous exploration of multiple successors of a conditional branch (as shown above); this does not affect the set of allowed outcomes.

The intra-instruction behaviour of a single instruction can largely be treated as sequential (but not atomic) execution of its ASL/Sail pseudocode. Each instruction instance state includes a pseudocode execution state, which one can think of as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information, detailed below, about the instruction instance's memory and register footprints, its register and memory reads and writes, whether it is finished, etc.

Model transitions For any state, the model defines the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Each transition arises from the next step of a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its thread state and/or the shared memory state. Instructions cannot be treated as atomic units: complete execution of a single instruction instance may involve many transitions, which can be interleaved with those of other instances in the same or other threads, and some of this is programmer-visible. The transitions are introduced below and defined in §1.4, with a precondition and a construction of the post-transition model state for each. The transitions labelled ∘ can always be taken eagerly, as soon as they are enabled, without excluding other behaviour; the • cannot.

Transitions for all instructions:

– [ifetch] Fetch request: This transition speculates the next address as a po-successor of a previously speculated instruction.
– Fetch instruction: Satisfy the fetch from instruction memory.
– [ifetch] Fetch instruction: Decode the instruction.
– Decode instruction: This transition represents a fetch and decode of a new instruction instance, as a program-order successor of a previously fetched instruction instance, or at the initial fetch address for a thread.
∘ Register read: This is a read of a register value from the most recent program-order predecessor instruction instance that writes to that register.
∘ Register write
∘ Pseudocode internal step: this covers ASL/Sail internal computation, function calls, etc.
∘ Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted or discarded, and all memory effects have taken place. For a conditional branch, any non-taken po-successor branches are discarded.

Load instructions:

∘ Initiate memory reads of load instruction: At this point the memory footprint of the load is provisionally known and its individual reads can start being satisfied.
– Satisfy memory read by forwarding from writes: This partially or entirely satisfies a single read by forwarding from its po-previous writes.
– Satisfy memory read from memory: This entirely satisfies the outstanding slices of a single read, from memory.
∘ Complete load instruction (when all its reads are entirely satisfied): At this point all the reads of the load have been entirely satisfied and the instruction pseudocode can continue execution.

A load instruction can be subject to being restarted until the Finish instruction transition. In some cases it is possible to tell that a load instruction will not be restarted or discarded before that, e.g. when all the instructions po-before the load instruction are finished. The Restart condition over-approximates the set of instructions that might be restarted.

Store instructions:

- ∘ Initiate memory writes of store instruction, with their footprints: At this point the memory footprint of the store is provisionally known.
- ∘ Instantiate memory write values of store instruction: At this point the writes have their values and program-order-subsequent reads can be satisfied by forwarding from them.
- ∘ Commit store instruction: At this point the store is guaranteed to happen (it cannot be restarted or discarded), and the writes can start being propagated to memory.
- – Propagate memory write: This propagates a single write to memory.
- ∘ Complete store instruction (when its writes are all propagated): At this point all writes have been propagated to memory, and the instruction pseudocode can continue execution.

Store-exclusive instructions:

- – Guarantee the success of store-exclusive: This guarantees the success of the store-exclusive.
- – Make a store-exclusive fail: This makes the store-exclusive fail.

Barrier instructions:

- ∘ Commit barrier

Cache maintenance instructions:

- – [ifetch] Begin IC: Perform instruction cache maintenance.
- – [ifetch] Propagate IC to thread: Wait for instruction cache maintenance to complete.
- – [ifetch] Perform DC: Perform data cache maintenance.

Instruction cache updates:

- – [ifetch] Add to instruction cache for thread: Update instruction cache for thread with write.

1.1 Intra-instruction Pseudocode Execution

To link the model transitions introduced above to the execution of the instructions an interface is needed between Sail and the rest of the concurrency model. For each instruction instance this intra-instruction semantics is expressed as a state machine, essentially running the instruction pseudocode, where each pseudocode execution state is a request of one of the following forms:

READ_MEM(*read_kind*, *address*, *size*, *read_continuation*)	Read request
EXCL_RES(*res_continuation*)	Store-exclusive result
PERFORM_IC(*address, res_continuation*)	Propagate an `ic ivau`
WAIT_IC(*address, res_continuation*)	Wait for an `ic ivau` to complete
PERFORM_DC(*address, res_continuation*)	Propagate a `dc cvau`
WRITE_EA(*write_kind*, *address*, *size*, *next_state*)	Write effective address
WRITE_MEMV(*memory_value*, *write_continuation*)	Write value
BARRIER(*barrier_kind*, *next_state*)	Barrier
READ_REG(*reg_name*, *read_continuation*)	Register read request
WRITE_REG(*reg_name*, *register_value*, *next_state*)	Write register
INTERNAL(*next_state*)	Pseudocode internal step
DONE	End of pseudocode

Each of these states is a suspended computation with a request for an action or input from the concurrency model and, except in the case of DONE, a continuation for the remaining execution.

Here memory values are lists of bytes, addresses are 64-bit numbers, read and write kinds identify whether they are regular, exclusive, and/or release/acquire operations, register names identify a register and slice thereof (start and end bit indices), and the continuations describe how the instruction instance will continue for any value that might be provided by the surrounding memory model. This largely follows [?]§2.2]micro2015, except that memory writes are split into two steps, WRITE_EA and WRITE_MEMV. We ensure these are paired in the pseudocode, but there may be other steps between them: it is observable that the WRITE_EA can occur before the value to be written is determined, because the potential memory footprint of the instruction becomes provisionally known then.

We ensure that each instruction has at most one memory read, memory write, or barrier step, by rewriting the pseudocode to coalesce multiple reads or writes, which are then split apart into the architecturally atomic units by the thread semantics; this gives a single commit point for all memory writes of an instruction.

Each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit, or from the thread's initial register state if there is no such. That instance may not have executed its register write yet, in which case the register read should block. The semantics therefore has to know the register write footprint of each instruction instance, which it calculates when the instruction instance is created. We ensure in the pseudocode that each instruction does exactly one register write to each bit of its register footprint, and also that instructions do not do register reads from their own register writes. In some cases, but not in the fragment of ARM that we cover at present, register write footprints need to be dynamically recalculated, when the actual footprint only becomes known during pseudocode execution.

Data-flow dependencies in the model emerge from the fact that a register read has to wait for the appropriate register write to be executed (as described above). This has to be carefully handled in order not to create unintentional strength. First, for some instructions we need to ensure that the pseudocode is in the maximally liberal order, e.g. to allow early computed-address register writebacks before the corresponding memory write. Leaving load-pair aside (which we do not cover), and the treatment of the multiple reads or writes that can be associated with a single load or store instruction (which we do), we have not so far needed other intra-instruction concurrency. Second, the model has to be able to know when a register read value can no longer change (i.e. due to instruction restart). We approximate that by recording, for each register write, the set of register and memory reads the instruction instance has performed at the point of executing the write. This information is then used as follows to determine whether a register read value is final: if the instruction instance that performed the register write from which the register reads from is finished, the value is final; otherwise check that the recorded reads for the register write do not include memory reads, and continue recursively with the recorded register reads. For the instructions we cover this approximation is exact.

We express the pseudocode execution semantics in two ways: a definitional interpreter for Sail [1], with an exhaustive symbolic mode to (re)calculate an instruction's memory and register footprints, and as a shallow embedding, translating Sail into directly executable code, with separate hand-written definitions of the footprint functions. The two are essentially equivalent: the first lets one small-step through the pseudocode interactively, while the second is more efficient and should be more convenient for proof.

1.2 Instruction Instance States

Each instruction instance i has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, or barrier instruction, each with the associated kind; or a conditional branch; or a 'simple' instruction.
- *regs_in*, the set of input *reg_name*s, as statically determined;
- *regs_out*, the output *reg_name*s, as statically determined;
- *pseudocode_state* (or sometimes just 'state' for short), one of
  - PLAIN *next_state*, ready to make a pseudocode transition;
  - PENDING_MEM_READS *read_cont*, performing the read(s) from memory of a load; or
  - PENDING_MEM_WRITES *write_cont*, performing the write(s) to memory of a store;

– *reg_reads*, the accumulated register reads, including their sources and values, of this instance's execution so far;
– *reg_writes*, the accumulated register writes, including dependency information to identify the register reads and memory reads (by this instruction) that might have affected each;
– *mem_reads*, a set of memory read requests. Each request includes a memory footprint (an address and size) and, if the request has already been satisfied, the set of write slices (each consisting of a write and a set of its byte indices) that satisfied it.
– *mem_writes*, a set of memory write requests. Each request includes a memory footprint and, when available, the memory value to be written. In addition, each write has a flag that indicates whether the write has been propagated (passed to the memory) or not.
– [exclusive] *successful_exclusive*, for store-exclusives, indicates whether it was previously guaranteed to succeed or made to fail.
– information recording whether the instance is committed, finished, etc.

Read requests include their read kind and their memory footprint (their address and size), the as-yet-unsatisfied slices (the byte indices that have not been satisfied), and, for the satisfied slices, information about the write(s) that they were satisfied from. Write requests include their write kind, their memory footprint, and their value. When we refer to a write or read request without mentioning the kind of request we mean the request can be of any kind. A load instruction which has initiated (so its read request list *mem_reads* is not empty) and for which all its read requests are satisfied (i.e. there are no unsatisfied slices) is said to be *entirely satisfied*. A load-exclusive is called *successful* if the first po-following store-exclusive has been guaranteed to succeed (as opposed to does not exist or has not been guaranteed to succeed or made to fail). The successful load-exclusive and the successful store-exclusive are said to be *paired*. If a successful load-exclusive has a read request that is mapped, in the exclusives map, to a write slice *ws*, we say the load-exclusive has an outstanding lock on *ws*.

1.3 Thread States

The model state of a single hardware thread includes:

– *thread_id*, a unique identifier of the thread;
– *register_data*, the name, bit width, and start bit index for each register;
– *initial_register_state*, the initial register value for each register;
– *initial_fetch_address*, the initial fetch address for this thread;
– *instruction_tree*, a tree or list of the instruction instances that have been fetched (and not discarded), in program order.

1.4 Model Transitions

Fetch request For some instruction *i*, any possible next fetch address *loc* can be requested, adding it to the fetch queue, if:

1. it has not already been requested, i.e., none of the immediate successors of *i* in the thread's *instruction_tree* are from *loc*; and
2. either *i* is not decoded, or, if it has been, *loc* is a possible next fetch address for *i*:
   (a) for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);
   (b) for a conditional branch, either the successor address or the branch target address[3]; or
   (c) for a jump to an address which is not yet determined, any address (this is approximated in our tool implementation, necessarily).

Note that this allows speculation past conditional branches and calculated jumps. Action: add an unfetched entry for *loc* to the fetch queue for *i*'s thread.

Fetch instruction For any fetch-queue entry in the UNFETCHED state, its fetch can be satisfied from the memory and abstract data cache, from write-slices *ws*, if:

[3] In AArch64, all the conditional branch instructions have statically determined addresses.

1. the write-slices (parts of writes) *ws* have the 4-byte footprint of the entry and can be constructed by composing some combination of the flat memory and a set of writes from the abstract data cache.

   Action: change the fetch-queue entry's state to Fetched*(ws)*.

Fetch instruction (unpredictable) For any fetch-queue entry in the Unfetched state, its fetch can be satisfied from the memory and abstract data cache in a constrained-unpredictable way, if:

1. there exists a set of sets of write-slices, each of which can be constructed in the same way as above;
2. that set contains multiple values, and at least one of those values corresponds to an instruction that is not `B.cond` or one of {`B`, `BL`, `BRK`, `HVC`, `SMC`, `SVC`, `ISB`, `NOP`}, and they are not all `B.cond` instructions.

Action: record the fetch-queue entry as Constrained_unpredictable. When this has reached decode and the corresponding point in the instruction tree becomes non-speculative, the entire thread state will become Constrained_unpredictable.

Fetch instruction (B.cond) For any fetch-queue entry in the Unfetched state, its fetch can be satisfied from the memory and abstract data cache, from write-slices ws and ws', with value ws'', if:

1. there exists write-slices ws and ws', each of which can be constructed in the same way as above;
2. ws and ws' correspond to the encoding of two conditional branch instructions b and b';
3. the write-slices ws'' can be constructed as the combination of ws and ws' such that ws'' is the encoding of the branch instruction with b's condition and b''s target.

   Action: record the fetch-queue entry as Fetched*(ws'')*.

Decode instruction If the last entry in the fetch queue is in Fetched*(ws)* state, it can be removed from the queue, decoded, and begin execution, if all po-previous `ISB` instructions in the instruction tree have finished. Action:

1. Construct a new instruction instance i with the correct instruction kind and state, for i's program location, and add it to the instruction tree.
2. Discard all speculative entries in the instruction tree that are successors of i that are now known to be incorrect speculations.

Initiate memory reads of load instruction An instruction instance i with next state Read_mem*(read_kind, address, size, read_cont)* can initiate the corresponding memory reads. Action:

1. Construct the appropriate read requests *rrs*:
   - if *address* is aligned to *size* then *rrs* is a single read request of *size* bytes from *address*;
   - otherwise, *rrs* is a set of *size* read requests, each of one byte, from the addresses *address...address+size-1*.
2. set *i.mem_reads* to *rrs*; and
3. update the state of i to Pending_mem_reads *read_cont*.

Satisfy memory read by forwarding from writes For a load instruction instance i in state Pending_mem_reads *read_cont*, and a read request, r in *i.mem_reads* that has unsatisfied slices, the read request can be partially or entirely satisfied by forwarding from unpropagated writes by store instruction instances that are po-before i, if the *read-request-condition* predicate holds. This is if:

1. [ifetch] all po-previous `dsb sy` instructions are finished;
2. all po-previous `dmb sy` and `isb` instructions are finished;
3. [`dmb ld`/ `dmb st`] all po-previous `dmb ld` instructions are finished;
4. [release/ acquire] if i is a load-acquire, all po-previous store-releases are finished; and
5. [release/ acquire] all non-finished po-previous load-acquire instructions are entirely satisfied.

Let *wss* be the maximal set of unpropagated write slices from store instruction instances po-before i (if i is a load-acquire, exclude store-exclusive writes), that overlap with the unsatisfied slices of r, and which are not superseded by intervening stores that are either propagated or read from by this thread. That last condition requires, for each write slice *ws* in *wss* from instruction i':

– that there is no store instruction po-between i and i' with a write overlapping *ws*, and
– that there is no load instruction po-between i and i' that was satisfied from an overlapping write slice from a different thread.

Action:

1. update r to indicate that it was satisfied by *wss*; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a po-successor of i, and every read request r' of i' that was satisfied from *wss*′, if there exists a write slice *ws*′ in *wss*′, and an overlapping write slice from a different write in *wss*, and *ws*′ is not from an instruction that is a po-successor of i, or if i' was a data-cache maintenance by virtual address to a cache line that overlaps with any of the write slices in *wss*′, restart i' and its data-flow dependents (including po-successors of load-acquire instructions).

Note that store-release writes cannot be forwarded to load-acquires: a load-acquire instruction cannot be satisfied before all po-previous store-release instructions are finished, and *wss* does not include writes from finished stores (as those must be propagated).

Satisfy memory read from memory For a load instruction instance i in state PENDING_MEM_READS *read_cont*, and a read request r in *i.mem_reads*, that has unsatisfied slices, the read request can be satisfied from memory if i is not a successful load-exclusive or no other successful load-exclusive from a different thread has an outstanding lock on the writes r is trying to read from.

If: the read-request-condition holds (see previous transition).

Action: let *wss* be the write slices from memory or the data cache network, whichever is newer, covering the unsatisfied slices of r, and apply the action of Satisfy memory read by forwarding from writes. In addition, if i is a successful load-exclusive, union *wss* with the set of write slices r is mapped to in the exclusives map.

Note that Satisfy memory read by forwarding from writes might leave some slices of the read request unsatisfied. Satisfy memory read from memory, on the other hand, will always satisfy all the unsatisfied slices of the read request.

Complete load instruction (when all its reads are entirely satisfied) A load instruction instance i in state PENDING_MEM_READS *read_cont* can be completed (not to be confused with finished) if all the read requests *i.mem_reads* are entirely satisfied (i.e., there are no unsatisfied slices).

Action: update the state of i to PLAIN *(read_cont (memory_value))*, where *memory_value* is assembled from all the write slices that satisfied *i.mem_reads*.

Guarantee the success of store-exclusive A store-exclusive instruction instance i with next state EXCL_RES*(res_cont)* can be guaranteed to succeed if:

1. the store-exclusive has not been made to fail (as recorded in *i.successful_exclusive*);
2. assuming i is successful, it can be paired with a load-exclusive i' (see §1.2); and
3. if i' has already been satisfied (not necessarily entirely), let *wss* be the set of propagated write slices i' has read from, then, no slice in *wss* has been overwritten (in memory) by a write from another thread, and no other successful load-exclusive from a different thread has an outstanding lock on a write slice from *wss*.

Action:

1. record in *i.successful_exclusive* that the store-exclusive will be successful;
2. if i' has already been satisfied, union *wss* with the set of write slices the read request of i' is mapped to in the exclusives map, where *wss* is as above; and
3. update the state of i to PLAIN *(res_cont (true))*.

Make a store-exclusive fail A store-exclusive instruction instance i with next state Excl_res*(res_continuation)* can be made to fail if the store-exclusive has not been guaranteed to succeed (as recorded in *i.successful_exclusive*) Action:

1. record in *i.successful_exclusive* that the store-exclusive was made to fail; and
2. update the state of i to Plain *(res_cont (false))*.

Note the promise-success transition is enabled before the store-exclusive commits, and we do not require it to have a fully-determined address or to be non-restartable. As a result, a store-exclusive that has already promised its success might be restarted. Since other instructions may rely on its promise, the restart will not affect the value of *i.successful_exclusive*. Instead, when the store-exclusive is restarted it will take the same promise/failure transition as before its restart — based on the value of *i.successful_exclusive*.

Initiate memory writes of store instruction, with their footprints An instruction instance i with next state Write_ea*(write_kind, address, size, next_state′)* can announce its pending write footprint. Action:

1. construct the appropriate write requests:
   - if *address* is aligned to *size* then *ws* is a single write request of *size* bytes to *address*;
   - otherwise *ws* is a set of *size* write requests, each of one byte size, to the addresses *address...address+size-1*.
2. set *i.mem_writes* to *ws*; and
3. update the state of i to Plain *next_state′*.

Note that at this point the write requests do not yet have their values. This state allows non-overlapping po-following writes to propagate.

Instantiate memory write values of store instruction An instruction instance i with next state Write_memv*(memory_value, write_cont)* can initiate the corresponding memory writes. Action:

1. split *memory_value* between the write requests *i.mem_writes*; and
2. update the state of i to Pending_mem_writes *write_cont*.

Commit store instruction For an uncommitted store instruction i in state Pending_mem_writes *write_cont*, i can commit if:

1. i has fully determined data (i.e., the register reads cannot change, see §1.5);
2. all po-previous conditional branch instructions are finished;
3. all po-previous `dmb sy` and `isb` instructions are finished;
4. [ifetch] all po-previous `dsb sy` instructions are finished;
5. [dmb ld/dmb st] all po-previous `dmb ld` instructions are finished;
6. [release/acquire] all po-previous load-acquire instructions are finished;
7. all po-previous store instructions, except for store-exclusives that failed, have initiated and so have non-empty *mem_writes*;
8. [release/acquire] if i is a store-release, all po-previous memory access instructions are finished;
9. [dmb ld/dmb st] all po-previous `dmb st` instructions are finished;
10. all po-previous memory access instructions have a fully determined memory footprint; and
11. all po-previous load instructions have initiated and so have non-empty *mem_reads*.

Action: record i as committed.

Propagate memory write For an instruction i in state Pending_mem_writes *write_cont*, and an unpropagated write, w in *i.mem_writes*, the write can be propagated if:

1. all memory writes of po-previous store instructions that overlap w have already propagated
2. all read requests of po-previous load instructions that overlap with w have already been satisfied, and the load instruction is non-restartable (see §1.5);
3. all read requests satisfied by forwarding w are entirely satisfied; and
4. [exclusive] no successful load-exclusive from a different thread has an outstanding lock on a write slice that overlaps with w.

Action:

1. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' po-after i and every read request r' of i' that was satisfied from *wss'* , if there exists a write slice *ws'* in *wss'* that overlaps with w and is not from w, and *ws'* is not from a po-successor of i, or if i' is a data-cache maintenace instruction to a cache line whose footprint overlaps with w, restart i' and its data-flow dependents;
2. record w as propagated;
3. add w as a complete slice to the data cache network.

Complete store instruction (when its writes are all propagated) A store instruction i in state PENDING_MEM_WRITES *write_cont*, for which all the memory writes in *i.mem_writes* have been propagated, can be completed. Action: update the state of i to PLAIN*(write_cont(true))*.

Commit barrier A barrier instruction i in state PLAIN *next_state* where *next_state* is BARRIER*(barrier_kind, next_state')* can be committed if:

1. all po-previous conditional branch instructions are finished;
2. [`dmb ld`/ `dmb st`] if i is a `dmb ld` instruction, all po-previous load instructions are finished;
3. [`dmb ld`/ `dmb st`] if i is a `dmb st` instruction, all po-previous store instructions are finished;
4. all po-previous `dmb sy` barriers are finished;
5. [ifetch] all po-previous `dsb sy` barriers are finished;
6. if i is an `isb` instruction, all po-previous memory access instructions have fully determined memory footprints; and
7. if i is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished;
8. [ifetch] if i is a `dsb sy` instruction, all po-previous memory access instructions, barriers and cache maintenance instructions have finished.

Note that this differs from the previous Flowing and POP models: there, barriers committed in program-order and potentially re-ordered in the storage subsystem. Here the thread subsystem is weakened to subsume the re-ordering of Flowing's (and POP's) storage subsystem.

Action:

1. update the state of i to PLAIN *next_state'*;
2. [ifetch] if i is an `isb` instruction, for all threads instruction tree's, for any instruction instance i in the FETCHED state, set it to the UNFETCHED state.

Begin IC An instruction i (with unique instruction instance ID *iiid*) in state PERFORM_IC*(address, state_cont)* can begin performing the `IC` behaviour if all po-previous `DSB ISH` instructions have finished. Action:

1. For each thread *tid'* (including this one), add *(iiid, address)* to that thread's *ic_writes*;
2. Set the state of i to PROPAGATE_IC*(address, state_cont)*.

Propagate IC to thread An instruction i (with ID *iiid*) in state WAIT_IC*(address, state_cont)* can do the relevant invalidate for any thread *tid'*, modifying that thread's instruction cache and fetch queue, if there exists a pending entry *(iiid, address)* in that thread's *ic_writes*. Action:

1. for any entry in the fetch queue for thread *tid*, whose *program_loc* is in the same minimum-size instruction cache line as *address*, and is in FETCHED(_) state, set it to the UNFETCHED state;
2. for the instruction cache of thread *tid*, remove any write-slices which are in the same instruction cache line of minimum size as *address*.

Complete IC An instruction i (with ID *iiid*) in the state WAIT_IC*(address, state_cont)* can complete if there exists no entry for *iiid* in any thread's *ic_writes*. Action: set the state of i to PLAIN*(state_cont)*.

Perform DC An instruction i in the state PERFORM_DC*(address, state_cont)* can complete if all po-previous `DMB ISH` and `DSB ISH` instructions have finished. Action:

1. For the most recent write slices *wss* which are in the same data cache line of minimum size in the abstract data cache as *address*, update the memory with *wss*;
2. Remove all those writes from the abstract data cache.
3. Set the state of i to Plain*(state_cont)*.

Add to instruction cache for thread A thread *tid*'s instruction cache can become spontaneously updated with a write w from the storage subsystem, if this write (as a complete slice) does not already exist in the instruction cache. Action: Add this write (as a complete slice) to thread *tid*'s instruction cache.

Register read An instruction instance i with next state Read_reg*(reg_name, read_cont)* can do a register read if every instruction instance that it needs to read from has already performed the expected register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *register_value* be the assembled value from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *register_value*; and
2. update the state of i to Plain *(read_cont(register_value))*.

Register write An instruction instance i with next state Write_reg*(reg_name, register_value, next_state')* can do the register write. Action:

1. add *reg_name* to *i.reg_writes* with *write_deps* and *register_value*; and
2. update the state of i to Plain *next_state'*.

where *write_deps* is the set of all *read_sources* from *i.reg_reads* and a flag that is set to true if i is a load instruction that has already been entirely satisfied.

Pseudocode internal step An instruction instance i with next state Internal*(next_state')* can do that pseudocode-internal step. Action: update the state of i to Plain *next_state'*.

Finish instruction A non-finished instruction i with next state Done can be finished if:

1. if i is a load instruction:
   (a) all po-previous `dmb sy` and `isb` instructions are finished;
   (b) [dmb ld/dmb st] all po-previous `dmb ld` instructions are finished;
   (c) [release/acquire] all po-previous load-acquire instructions are finished;
   (d) it is guaranteed that the values read by the read requests of i will not cause coherence violations, i.e., for any po-previous instruction instance i', let *cfp* be the combined footprint of propagated writes from store instructions po-between i and i' and fixed writes that were forwarded to i from store instructions po-between i and i' including i', and let *cfp'* be the complement of *cfp* in the memory footprint of i. If *cfp'* is not empty:
      i. i' has a fully determined memory footprint;
      ii. i' has no unpropagated memory write that overlaps with *cfp'*; and
      iii. If i' is a load with a memory footprint that overlaps with *cfp'*, then all the read requests of i' that overlap with *cfp'* are satisfied and i' can not be restarted (see §1.5).
      Here a memory write is called fixed if it is the write of a store instruction that has fully determined data.
   (e) [release/acquire] if i is a load-acquire, all po-previous store-release instructions are finished;
2. i has fully determined data; and
3. all po-previous conditional branches are finished.

Action:

1. if i is a branch instruction, discard any untaken path of execution, i.e., remove any (non-finished) instructions that are not reachable by the branch taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

1.5 Auxiliary Definitions

Fully determined An instruction is said to have fully determined footprint if the memory reads feeding into its footprint are finished: A register write w, of instruction i, with the associated *write_deps* from *i.reg_writes* is said to be *fully determined* if one of the following conditions hold:

1. i is finished; or
2. the load flag in *write_deps* is *false* and every register write in *write_deps* is fully determined.

An instruction i is said to have *fully determined data* if all the register writes of *read_sources* in *i.reg_reads* are fully determined. An instruction i is said to have a *fully determined memory footprint* if all the register writes of *read_sources* in *i.reg_reads* that are associated with registers that feed into i's memory access footprint are fully determined.

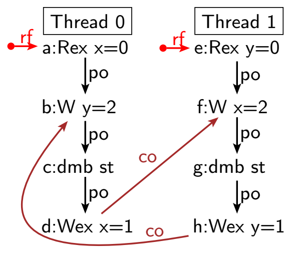
Restart condition To determine if instruction i might be restarted we use the following recursive condition: i is a non-finished instruction and at least one of the following holds,

1. there exists an unpropagated write w such that applying the action of the Propagate memory write transition to s will result in the restart of i;
2. there exists a non-finished load instruction l such that applying the action of the Satisfy memory read from memory transition to l will result in the restart of i (even if l is already entirely satisfied); or
3. there exists a non-finished instruction i' that might be restarted and i is in its data-flow dependents (including po-successors of load-acquire instructions).

Cache Line of Minimum Size Cache maintenance operations work over entire cache lines, not individual addresses. Each address is associated with at least one cache line for the data (and unified) caches, and one for the instruction caches. The cache line of minimum size is the smallest possible cache line for each of these. The `CTR_EL0.{DMinLine, IMinLine}` values describe the cache lines of minimum size for the data and instruction caches as $\log_2$ of the number of words in the cache line.

1.6 Remarks about load/store exclusive instructions

The MCA ARMv8 architecture intends that the success bit of store exclusives does not introduce dependencies, to allow (e.g.) hardware optimisations that dynamically replace load/store exclusive pairs by atomic read-modify-write operations that can execute in the memory subsystem and therefore be guaranteed to succeed. The ARMv8-axiomatic definition assumes all address/data/control dependencies to be from reads, not writes. In the operational model, matching this weakness has proved to be difficult: it means the operational model must be able to promise the success or failure of a store-exclusive instruction even before any of its registers reads/writes have been done, so before the store-exclusive's address and data are available. The early success promises are the



source of deadlocks in the operational model. To illustrate this consider, for example, the following litmus test and a state where both a and e are satisfied and finished, and where b and f are not propagated. Then d can promise its success, locking memory location x, and h can promise its success, locking location y. But now there is a deadlock:

- For d to propagate c has to be committed and hence b propagated.
  But b cannot propagate since y is locked.
- For h to propagate g has to be committed and hence f propagated.
  But f cannot propagate since x is locked.

Similar situations arise from cases where there are other barriers or release/acquire instructions in-between the load and the store exclusive, or if the store exclusive has additional dependencies that the load exclusive does not have. These are cases that are not really intended to be supported by the architecture.

The model can also currently deadlock if a load and a store-exclusive are paired successfully but later turn out to have different addresses: if the store-exclusive promises its success before its address is known it locks the matched load-exclusive's memory location; when they later turns out to be to a different addresses it never unlocks it. This issue can be fixed, but it is currently still being clarified what exactly the architecturally allowed behaviour should be.

References

1. Gray, K.E., Kerneis, G., Mulligan, D., Pulte, C., Sarkar, S., Sewell, P.: An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture (Dec 2015)