

Multitape Alphabet Reduction

Formal Proof in Isabelle/HOL

András Z. Salamon Michael Wehar

August 26, 2026

Abstract

Machine-checked formal proof in Isabelle/HOL that an arbitrary finite tape alphabet can be reduced to a fixed four-element alphabet on a nondeterministic multitape Turing machine while preserving the accepted language and tape count — and determinism, in the deterministic special case — at the cost of a single constant-factor slowdown that grows only logarithmically with the alphabet size.

Given a well-formed multitape machine M whose tape alphabet Γ_M has at least four symbols, the operation $M'' = \text{alphabet_reduce } M$ produces a machine over the fixed four-element alphabet $\text{sym4} = \{\text{BLANK4}, \text{LE4}, \text{BIT0}, \text{BIT1}\}$ (a blank $\square$, a left-end marker $\triangleright$, and the two bits 0, 1). It simulates M under a fixed per-symbol binary encoding, spelling each Γ_M -symbol as a block of $\lceil \log_2 |\Gamma_M| \rceil$ bits, and it preserves determinism. This session (`Multitape_Alphabet_Reduction`) establishes:

- *Language preservation:* for every input word w , $\text{encode_input_ar } \Gamma_M \text{ bl}_M w \in L(M'') \iff w \in L(M)$ (theorem `alphabet_reduce_language`; the forward inclusion is also available separately as `alphabet_reduce_language_forward`).
- *Time bound:* if M accepts w within $T(|w|)$ steps, then M'' accepts $\text{encode_input_ar } \Gamma_M \text{ bl}_M w$ within $(6k + 1) b T(|w|)$ steps, where k is the number of tapes of M and $b = \lceil \log_2 |\Gamma_M| \rceil$ is the per-symbol block width (theorem `alphabet_reduce_time_explicit`). The slowdown is thus a single constant factor $(6k + 1) b$ multiplying the running time, with no additive or lower-order term; it is fixed by the machine — the tape count (linearly) and the alphabet size (logarithmically, through the block width b above) — and does not depend on the input. The classical existential form $dbT(|w|) + eb|w| + f$ is the corollary `alphabet_reduce_time`.
- *Well-formedness, determinism, and tape count:* M'' is a well-formed substrate machine (`alphabet_reduce_wf`) that preserves determinism (`alphabet_reduce_det`) and the tape count (`alphabet_reduce_preserves_tape_count`); its tape alphabet

is the full four-element type `sym4`, of cardinality exactly four (`alphabet_reduce_produces_alphabet_size_4`).

This is an elementary per-symbol multi-cell encoding leading to a reduction in the number of tape symbols: the folklore alphabet homomorphism onto a fixed small alphabet, stated (but not costed) by Balcázar, Díaz, and Gabarró [2, p. 23] and set within the wider machine-model-simulation landscape [7, 6].

The nondeterministic multitape substrate lives in the sibling `Multitape_TM_Substrate` session; this session depends on `Multitape_TM_Substrate` and `HOL-Library`.

Acknowledgement. This formalisation was developed with proof engineering by Anthropic’s Claude Opus 4.5, 4.6, 4.7, and 4.8 as the work progressed, together with other Claude family models in supporting roles. Claude Code was used to structure the work and for access to Isabelle/HOL. We also thank Arthur Freitas Ramos who provided feedback and also contributed several tooling improvements.

Contents

1	Overview	4
1.1	High-level overview	4
1.2	Glossary	5
1.3	Scope and limitations	7
1.4	Details of theorem statements	8
2	Alphabet reduction	11
2.1	Per-symbol encoder helpers	11
2.2	Per-symbol encoder	12
2.3	Per-symbol encoder lemmas	13
2.4	Per-symbol decoder helpers	19
2.5	Per-symbol decoder	20
2.6	Stage type and validity	25
2.7	Per-tape enumeration helpers	28
2.8	Per-substep dispatch helpers	29
2.9	Substep transition relations	30
2.10	Determinism preservation	41
2.11	Simulation correspondence	46
2.12	Per-substep simulation steps	50
2.13	Read phase	83
2.14	Write phase composition	127
2.15	Advance phase	170
2.16	Per- M -step simulation phase and the chunked engine	193
2.17	Top-level theorems	217

3	Alphabet reduction: reverse language inclusion	234
3.1	Terminal accept configuration	234
3.2	Source substep-tags partition the substep relations	235
3.3	Destination substep-tags carve the cycle's substep order	236
3.4	Compute-step inversion	237
3.5	Next-step inversion (cycle closure / halt dispatch)	239
3.6	Determinism of the read substep	241
3.7	Source disambiguation for the remaining substeps	242
3.8	Substep step semantics — <i>mttm_step</i> -level lands-at	243
3.9	Walker invariants — per-substep stage predicates	246
3.10	Step lifts — <i>mttm_step</i> to specific substep	248
3.11	Chain pinning in <i>mttm_step</i> (<i>ar_delta_read M</i>)	251
3.12	Read-phase leaves, sub-relation chain variants	260
3.13	Read-phase combiners, sub-relation chain variants	264
3.14	Write-phase leaves, sub-relation chain variants	270
3.15	Advance-phase leaves, sub-relation chain variants	273
3.16	Write-phase combiners, sub-relation chain variants	275
3.17	Walker preservation — per-substep M'-step lemmas	286
3.18	Cycle-close bridging lemma	297
3.19	Chunked reverse engine	310

1 Overview

1.1 High-level overview

In one sentence: this session proves that *any* multitape Turing machine can be rebuilt as an equivalent machine whose tape alphabet has only four symbols, at the cost of a constant-factor slowdown. The rebuilt machine keeps every property a later construction relies on: the accepted language, the number of tapes, structural well-formedness, and determinism.

The entire development concerns a single function, `alphabet_reduce`, which maps a machine M over an arbitrary finite tape alphabet to a machine $M'' = \text{alphabet_reduce } M$ over the fixed four-element alphabet `sym4`. Each headline theorem is a guarantee about this one function:

- the output really uses four symbols (`alphabet_reduce_produces_alphabet_size_4`);
- it accepts the same language, as an “if and only if” modulo a fixed input re-encoding (`alphabet_reduce_language`);
- it runs within a linear-time slowdown (`alphabet_reduce_time`);
- it is still a legal machine (`alphabet_reduce_wf`), with the same number of tapes (`alphabet_reduce_preserves_tape_count`) and the same determinism status (`alphabet_reduce_det`).

Two points help a first-time reader. First, a machine here may be non-deterministic by default: its transition is a *relation*, so acceptance means “some run reaches the accept state”, and these theorems are statements about nondeterministic machines. The deterministic case is singled out by a separate condition, `det_mttm`. Second, because the alphabet shrinks, the reduced machine cannot literally accept the same strings; instead each source symbol is re-spelled as a fixed block of `sym4` symbols by the encoding `encode_input_ar`, and “same language” is meant up to that translation.

Every theorem fixes an arbitrary M with no constraint on its behaviour, so each is a genuine “for all M ” result: read `alphabet_reduce M` as “apply this to *any* machine”.

The sibling entry `Multitape_Alphabet_Enlargement` goes the other way: it *enlarges* the alphabet to buy speed (the classical linear-speedup theorem).

Terminology and prior work. *Alphabet reduction* is our name for this construction, chosen to pair with the sibling *alphabet enlargement*; the classical literature does not usually present the two together. The reduction itself is a folklore normalisation: a multitape machine over an arbitrary finite tape alphabet Γ is simulated over a fixed small alphabet at a per-machine

constant-factor cost of $\lceil \log_2 |\Gamma| \rceil$ cells per symbol. Classic texts assume it rather than cost it. Balcázar, Díaz, and Gabarró [2, p. 23] state it directly — a machine, being a word over its alphabet, “can be translated into any other alphabet having at least two symbols via a homomorphism” (their example encodes into $\{0, 1\}$, en route to gödelizing Turing machines) — but do not analyse its time cost. Hopcroft and Ullman [5, §7.2] likewise *assume* each machine’s tape symbols are already binary-encoded (enough to justify a universal machine over a fixed alphabet) and leave the construction as a reader exercise with no time analysis. The restricted-machine survey of the era [4] catalogues the surrounding landscape without isolating it. This factor is a *fixed constant* once Γ is fixed, *not* a $\log T$ -style slowdown; that slowdown arises instead from the separate tape-count reduction, and only appears once the two are composed. Neither van Emde Boas’s simulation survey [7] — which frames such alphabet re-encodings under the *Invariance Thesis* — nor Papadimitriou’s textbook [6] isolates the reduction as a costed construction; both treat it as part of the general machine-model-simulation landscape.

Related AFP developments. The closest formalisation prior is inside Balbach’s `Cook_Levin` [1]: its `Two_Four_Symbols` theory re-encodes between a four-symbol content alphabet and two binary symbols by fixed two-cell pairs, an encoding utility whose time cost stays implicit in the surrounding framework. This entry generalises that picture — an arbitrary l -symbol alphabet reduced to a fixed four, with the constant-factor slowdown stated as an explicit top-level theorem — and, unlike Balbach’s deterministic development, proves the reduction for nondeterministic machines while separately preserving determinism. Xu, Zhang, Urban, and Joosten’s `Universal_Turing_Machine` [8] assumes a fixed binary alphabet outright. The substrate descends from Dalvit and Thiemann’s `Multitape_To_Singletape_TM` [3] but makes the tape count a value rather than a type parameter, so this reduction and its time bound hold, in a single theorem, for machines of any number of tapes.

1.2 Glossary

The machine type is `mttm`, the ten-component descriptor `MTTM  $Q \Sigma \Gamma bl le \delta s t r k$` : state set, input alphabet, tape alphabet, blank symbol, left endmarker, transition relation, start/accept/reject states, and tape count. Throughout, M is the source machine.

Projections. Positional selectors that read one component out of M :

- `k_tm` M — the tape count k_M (a natural number);
- `Sigma_tm` M — the input alphabet Σ_M ;

- $\Gamma_{\text{tm}} M$ — the tape alphabet Γ_M ;
- $\text{bl_tm } M$ — the blank symbol;
- $\text{delta_tm } M$ — the transition relation δ_M ;
- $\text{t_tm } M$ — the accept state.

Shared substrate predicates. Each entry opens with the gist, then the precise conditions.

valid_mttm M “ M is a legal machine”. The structural invariant: Q and Γ finite, $\Sigma \subseteq \Gamma$, the three distinguished states lie in Q , blank and endmarker are tape-but-not-input symbols, $\text{accept} \neq \text{reject}$, $k > 0$, the transition relation is correctly typed, the *left-endmarker read discipline* holds (reading the endmarker forces re-writing it and never moving off the left edge), and every tape beyond index k is frozen blank.

well_formed_mttm M a legal machine that also starts cleanly and keeps its endmarker. An abbreviation: **valid_mttm** M strengthened with *non-degeneracy* ($\text{start} \neq \text{accept}$, $\text{start} \neq \text{reject}$, $\text{endmarker} \neq \text{blank}$) and the endmarker *write discipline* **le_unique** M . Required wherever a run must genuinely start, the endmarker must be detectable, and it must never be forged.

le_unique M the left endmarker is never freshly planted. A machine property in the same family as **det_mttm** — an opt-in structural guarantee a downstream construction may rely on. Precisely: no transition writes the endmarker where it was not already reading it ($a'j = le \Rightarrow aj = le$ over δ). Folded into **well_formed_mttm**; the alphabet transformations use it to pin the endmarker, and the origin-floating speed-up wrapper is the one construction that deliberately forgoes it (so it is **valid_mttm** but not **well_formed_mttm**).

det_mttm M M is deterministic. The transition relation is a partial function: each (state, read-vector) pair has at most one successor triple — the functionality predicate layered on the otherwise relational δ .

$L(M) = \text{Lang_mttm } M$ the language M accepts. All input words w (with set $w \subseteq \Sigma_M$) from which some run reaches the accept state; acceptance is *existential* over runs — nondeterministic.

accepts_in_time_mttm M w t M accepts w within t steps. Weak time-bounded acceptance: some run of length at most t reaches the accept state.

The input lives on a work tape. The substrate has *no* separate read-only input tape: the input word is placed on tape 0, which is an ordinary work tape. This is why `valid_mttm` requires $\Sigma \subseteq \Gamma$ — every input symbol occupies a tape cell, so it must also be a tape symbol — and it is load-bearing for the alphabet transformations, which re-encode the input *in place* on tape 0 rather than copying it to a fresh tape.

Alphabet-reduction vocabulary.

$b = \max(1, \lceil \log_2 |\Gamma| \rceil)$ the *block width*: the number of sym4 cells that spell one source symbol, and the alphabet-dependent factor in the slowdown $(6k + 1)b$ (with $k = \mathbf{k_tm} M$ the tape count). For $|\Gamma| \geq 4$ the max is inactive, so $b = \lceil \log_2 |\Gamma| \rceil$; it is the Isabelle constant `block_width` Γ .

`encode_input_ar` Γ bl w the input encoding
`concat (map (encode_symbol  $\Gamma$   $bl$ )  $w$ )`: the flat concatenation of the per-symbol encodings. Each source symbol is spelled as a fixed-width block of b cells over $\{\mathbf{BIT0}, \mathbf{BIT1}\}$, the binary numeral of the symbol’s index under a *blank-anchored* enumeration of Γ (a bijection $\Gamma \rightarrow \{0, \dots, |\Gamma| - 1\}$ that sends the blank bl to 0). The markers `BLANK4` and `LE4` never occur inside an encoded symbol; blank-anchoring is what lets an all-`BLANK4` block decode back to bl .

`sym4` the four-element output alphabet $\{\mathbf{BLANK4}, \mathbf{LE4}, \mathbf{BIT0}, \mathbf{BIT1}\}$: the blank $\square$, the left endmarker $\triangleright$, and the two bits 0, 1.

`ar_stage` the per-state bookkeeping carried by M'' (substep tag, current tape, bit counter, symbol buffer, direction vector, position kind); this is why M'' has state type $'q \times 'a$ `ar_stage`.

1.3 Scope and limitations

A reader deciding whether this construction fits their needs should note four caveats.

- *Minimal alphabet.* The source must carry at least four tape symbols ($|\Gamma_M| \geq 4$). The block-encoding is injective for *any* alphabet (the width always fits $|\Gamma_M|$); the real requirement is `block_width` ≥ 2 (the walker’s block logic assumes at least two cells per symbol), for which $|\Gamma_M| \geq 4$ is the (loose) sufficient condition the theorems carry.
- *Same language, up to a fixed re-encoding.* Because the alphabet shrinks, M'' cannot literally accept M ’s strings; “same language” means M'' accepts `encode_input_ar` Γ_M bl_M w exactly when M accepts w , stated modulo that translation.

- *Constant-factor slowdown, not $\log T$.* The overhead is *fixed* once Γ_M is fixed: the exact factor is $(6k + 1)b$ for a k -tape machine, with $b = \lceil \log_2 |\Gamma_M| \rceil$ cells per symbol (so it grows only logarithmically with the alphabet size). This is *not* the $\log T$ -style slowdown of a tape-count reduction, which arises separately and only once the two are composed.
- *The slowdown jumps at powers of two.* As a function of the alphabet size the factor $\lceil \log_2 |\Gamma_M| \rceil$ is a *step function*, not a smooth one: it is constant on each band $2^{b-1} < |\Gamma_M| \leq 2^b$ and rises by one exactly as $|\Gamma_M|$ crosses a power of two. A machine over five symbols pays the same width ($b = 3$) as one over eight, but strictly more than one over four ($b = 2$); a symbol added within a band is free, one that crosses a power of two costs a whole extra bit. Both edges of the band are proved ($2^{b-1} < |\Gamma_M| \leq 2^b$), so b is exactly the width to index the whole alphabet, though one cell above the coding minimum $\lceil \log_2 (|\Gamma_M| - 1) \rceil$ for the non-blank symbols (index 0 is reserved for the blank, buying a branch-free decoder), the two agreeing except just past a power of two.

Each theorem’s exact hypotheses appear in the next subsection.

1.4 Details of theorem statements

For each headline theorem we give a cleaned statement and the role of every predicate it mentions. The hypotheses form a *ladder*: purely structural facts need none, validity needs the minimal-alphabet bound $|\Gamma_M| \geq 4$, behavioural equivalence needs the stronger `well_formed_mttm`, and determinism preservation needs `valid_mttm` together with `det_mttm`.

`alphabet_reduce_produces_alphabet_size_4.`

$$|\text{sym4}| = 4.$$

No machine and no hypotheses. This is the operational meaning of “reduction to four”: the reduction hard-codes the output tape alphabet as the whole type `sym4`, and this theorem certifies that type has exactly four inhabitants. It is a fact about the target *type*, deliberately independent of any particular M .

`alphabet_reduce_preserves_tape_count.`

$$k_{M''} = k_M.$$

Mentions only the tape-count projection `k_tm` and the reduction, with no hypotheses. The reduction acts on the alphabet dimension alone; the tape count is copied verbatim, so the equation holds for every term of machine type, valid or not.

alphabet_reduce_wf.

$$\text{valid_mttm } M \wedge |\Gamma_M| \geq 4 \implies \text{valid_mttm } M''.$$

Closure of the structural invariant; **valid_mttm** appears as both hypothesis and conclusion. The side condition $|\Gamma_M| \geq 4$ is the *minimal-alphabet* requirement: the source must carry at least four tape symbols for the binary block-encoding to be well-defined and injective. Discharging it amounts to re-checking the endmarker discipline and frozen-tape clauses in the encoded world.

alphabet_reduce_language.

$$\begin{aligned} &\text{well_formed_mttm } M \wedge |\Gamma_M| \geq 4 \implies \\ &\forall w. \text{ set } w \subseteq \Sigma_M \longrightarrow \\ &\quad (\text{encode_input_ar } \Gamma_M \text{ bl}_M w \in L(M'')) \\ &\quad = (w \in L(M)). \end{aligned}$$

The correctness core, stated as an equivalence between two membership facts. Note the upgrade from **valid_mttm** to **well_formed_mttm**: the simulation needs the start state to be a genuine pre-halt state and the endmarker to be distinguishable from the blank. The guard $\text{set } w \subseteq \Sigma_M$ restricts attention to genuine input words, and **encode_input_ar** is the translation under which the two languages coincide. The forward inclusion ($\Rightarrow$) is available separately as **alphabet_reduce_language_forward**; this theorem strengthens it to the equivalence.

alphabet_reduce_time_explicit and alphabet_reduce_time. Write k for the number of tapes of M and $b = \text{block_width}(\Gamma_M) = \lceil \log_2 |\Gamma_M| \rceil$ for the per-symbol block width. Under $\text{well_formed_mttm } M \wedge |\Gamma_M| \geq 4$,

$$\begin{aligned} &\forall w. \text{ set } w \subseteq \Sigma_M \longrightarrow \\ &\quad \text{accepts_in_time_mttm } M w (T(|w|)) \longrightarrow \\ &\quad \text{accepts_in_time_mttm } M'' \\ &\quad (\text{encode_input_ar } \Gamma_M \text{ bl}_M w) ((6k + 1) b T(|w|)). \end{aligned}$$

Linear-time slowdown with an *explicit* constant. The factor is the single number $(6k + 1) b$ multiplying the running time, with no additive and no linear-in- $|w|$ term. It comes from an exact per-step super-step count of $k(5b + 2) + 2$, folded up to $(6k + 1) b$ using $b \geq 2$ (which holds because $|\Gamma_M| \geq 4$) and tight at $b = 2$, i.e. $|\Gamma_M| = 4$. The crux is that this constant is bound *once, outside the* $\forall w$: it depends only on M (its tape count, and logarithmically its alphabet size), so the slowdown is uniform across all inputs rather than tuned per input. The classical existential form $\exists d, e, f, b \geq$

1. $dbT(|w|) + eb|w| + f$ is the obtains-corollary `alphabet_reduce_time`, hiding the explicit constants behind existentials instantiated at $d = 6k + 1$ and $e = f = 0$.

`alphabet_reduce_det.`

$$\text{valid_mttm } M \wedge \text{det_mttm } M \implies \text{det_mttm } M''.$$

Determinism preservation. It needs *both* `valid_mttm` and `det_mttm`: functionality of the reduced transition relation follows from functionality of the source only because validity pins the transition typing on which the per-substep transitions of M'' are built. The proof factors through one functionality lemma per substep (read, compute, write, advance, next).

```

theory AlphabetReduction_Codec
  imports Multitape_TM_Substrate.Multitape_Substrate
begin

```

2 Alphabet reduction

This theory opens the alphabet-reduction development. The *alphabet_reduce* combinator takes a well-formed substrate machine over an arbitrary finite alphabet *a* (with tape-alphabet cardinality ≥ 4) and produces a well-formed substrate machine over the fixed four-element alphabet *sym4*, preserving the accepted language, determinism, and tape count.

This is the elementary determinism-preserving per-symbol multi-cell encoding.

The development is layered: this first theory fixes the output alphabet *sym4* and the per-symbol binary codec; the simulator, its delta, and the headline theorems (*alphabet_reduce_wf*, *alphabet_reduce_language*, *alphabet_reduce_time*, *alphabet_reduce_det*) are built across the theories that follow. Throughout, *b* abbreviates *block_width* Γ (the per-symbol cell width).

The fixed four-element output alphabet: *BLANK4* (blank), *LE4* (left endmarker), *BIT0* (binary 0), *BIT1* (binary 1).

```

datatype sym4 = BLANK4 | LE4 | BIT0 | BIT1

```

```

lemma sym4_UNIV: (UNIV :: sym4 set) = {BLANK4, LE4, BIT0, BIT1}
  using sym4.exhaust by blast

```

```

lemma sym4_card: card (UNIV :: sym4 set) = 4
proof –
  have (UNIV :: sym4 set) = {BLANK4, LE4, BIT0, BIT1}
    by (rule sym4_UNIV)
  moreover have card {BLANK4, LE4, BIT0, BIT1} = 4
    by simp
  ultimately show ?thesis by simp
qed

```

2.1 Per-symbol encoder helpers

The per-source-symbol block length: the number of *sym4* cells that encode one Γ -symbol, and the slowdown factor of the reduction. At least 1 (to avoid degenerate empty encodings even for trivial Γ), and otherwise $\lceil \log_2 (\text{card } \Gamma) \rceil$, the fewest binary digits that index all of Γ . Defined combinatorially via *LEAST* on the predicate $\text{card } \Gamma \leq 2 \wedge n$; existence of such an *n* follows from $\text{card } \Gamma \leq 2 \wedge \text{card } \Gamma$ (HOL.Power.less_exp).

What is counted. The index space $\{.. < \text{card } \Gamma\}$ covers *every* symbol of Γ : the blank at index 0 (the anchor *gamma_enum* relies on — see below), and the left endmarker *le* as an ordinary coded symbol, since *valid_mttm*

permits a machine to write le at any tape position (its only special handling is the single $LE4$ cell at position 0). So $block_width$ is genuinely the width of this uniform code, not a loose bound on a smaller one — but it is not the *coding* minimum. Reserving index 0 for the blank (so an all- $BLANK4$ block reads back through the same accumulator) puts it one digit above $\lceil \log_2 (card\ \Gamma - 1) \rceil$, the width for the $card\ \Gamma - 1$ non-blank symbols; the two agree except just past a power of two. Detecting the all- $BLANK4$ block directly would recover that digit, and additionally giving le its own marker block would reach $\lceil \log_2 (card\ \Gamma - 2) \rceil$ — but each costs a marker-detection branch in the read and write phases, so that constant-factor gain is deliberately not taken.

definition $block_width :: 'a\ set \Rightarrow nat$ **where**
 $block_width\ \Gamma = max\ 1\ (LEAST\ n.\ card\ \Gamma \leq 2^{\wedge} n)$

A *blank-anchored* enumeration of Γ as a bijection to $\{..< card\ \Gamma\}$ that additionally sends the blank bl to 0 . The anchor is load-bearing for the read phase: a blank source cell is encoded as a b -cell all- $BLANK4$ block ($bit_value = 0$), so the read fold is a fixpoint at $gamma_unenum\ \Gamma\ bl\ 0$, which must equal bl — and the decode round-trips force that to $gamma_enum\ \Gamma\ bl\ bl = 0$. Existence of such a bijection is unconditional on *finite* Γ for *any* bl (*ex_bij_betw_anchor* below: swap 0 with $g\ bl$ when $bl \in \Gamma$, otherwise free off-domain), so the carried bl needs no $bl \in \Gamma$ side condition. Downstream proofs reason about *bij_betw*- and anchor-derived facts only; the Hilbert choice picks one specific anchored bijection.

definition $gamma_enum :: 'a\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow nat$ **where**
 $gamma_enum\ \Gamma\ bl =$
 $(SOME\ f.\ bij_betw\ f\ \Gamma\ \{..< card\ \Gamma\} \wedge f\ bl = 0)$

Render a natural number as a fixed-length *sym4* bit list ($BIT0$ / $BIT1$ only). The list has shape $[bit_{b-1}, \dots, bit_1, bit_0]$: most-significant bit at the head, least-significant bit at the tail. For $n \geq 2^{\wedge} b$ the high bits are dropped (only the low b bits are rendered).

fun $nat_to_bits :: nat \Rightarrow nat \Rightarrow sym4\ list$ **where**
 $nat_to_bits\ 0\ _ = []$
 $| nat_to_bits\ (Suc\ k)\ n =$
 $nat_to_bits\ k\ (n\ div\ 2)$
 $@ [if\ n\ mod\ 2 = 0\ then\ BIT0\ else\ BIT1]$

2.2 Per-symbol encoder

Encoding a single $'a$ -symbol as a fixed-length list of *sym4*-symbols, parameterised over the source alphabet Γ (a finite set, passed explicitly as a value rather than through a type-class constraint, so the construction is uniform over all source alphabets). The length b is the slowdown factor; the encoding is injective on Γ . The two bit symbols $BIT0$, $BIT1$ carry the payload;

LE_4 and $BLANK_4$ do not appear in any encoder image. For $x \notin \Gamma$ the encoder produces a value that's structurally well-shaped (a b -cell bit list) but semantically arbitrary — downstream proofs always operate on $x \in \Gamma$.

definition $encode_symbol ::$

$'a \text{ set} \Rightarrow 'a \Rightarrow 'a \Rightarrow sym_4 \text{ list}$ **where**
 $encode_symbol \Gamma \text{ bl } x = nat_to_bits (block_width \Gamma) (gamma_enum \Gamma \text{ bl } x)$

Encoding an input word as a flat list of sym_4 -symbols (concatenation of per-symbol encodings under the given Γ).

definition $encode_input_ar ::$

$'a \text{ set} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow sym_4 \text{ list}$ **where**
 $encode_input_ar \Gamma \text{ bl } w = concat (map (encode_symbol \Gamma \text{ bl}) w)$

2.3 Per-symbol encoder lemmas

The encoded list has length b , by induction on the block width. Total — no $x \in \Gamma$ precondition is needed.

lemma $length_nat_to_bits$ [simp]:

$length (nat_to_bits k n) = k$
by (induct k arbitrary: n) auto

lemma $length_encode_symbol$ [simp]:

$length (encode_symbol \Gamma \text{ bl } x) = block_width \Gamma$
unfolding $encode_symbol_def$ **by** simp

Every cell of an encoded symbol is in the bit alphabet $\{BIT0, BIT1\}$. In particular, the substrate-reserved markers LE_4 and $BLANK_4$ never appear in an encoder image.

lemma $set_nat_to_bits$:

$set (nat_to_bits k n) \subseteq \{BIT0, BIT1\}$
by (induct k arbitrary: n) auto

lemma $encode_symbol_cell_domain$:

$set (encode_symbol \Gamma \text{ bl } x) \subseteq \{BIT0, BIT1\}$
unfolding $encode_symbol_def$
by (rule set_nat_to_bits)

The enumeration $gamma_enum \Gamma \text{ bl}$ is a genuine bijection from Γ to $\{.. $card \Gamma\}$ whenever Γ is finite. Existence is library ($ex_bij_betw_finite_nat$); the Hilbert-choice operator picks one such bijection.$

Existence of a blank-anchored bijection $\Gamma \rightarrow \{.. $card \Gamma\}$ sending bl to 0, unconditional on $finite \Gamma$ for any bl . Take any bijection g (library $ex_bij_betw_finite_nat$); if $bl \in \Gamma$, post-compose with the nat-level transposition of 0 and $g \text{ bl}$ (both $< card \Gamma$), which is a bij_betw of $\{.. $card \Gamma\}$ onto itself by $endo_inj_surj$; if $bl \notin \Gamma$, just override g at bl (off-domain, so bij_betw is unchanged by bij_betw_cong).$$

```

lemma ex_bij_betw_anchor:
  fixes  $\Gamma :: 'a \text{ set}$  and  $bl :: 'a$ 
  assumes finG: finite  $\Gamma$ 
  shows  $\exists f. \text{bij\_betw } f \ \Gamma \ \{..< \text{card } \Gamma\} \wedge f \ bl = 0$ 
proof -
  obtain g where  $g: \text{bij\_betw } g \ \Gamma \ \{..< \text{card } \Gamma\}$ 
    using ex_bij_betw_finite_nat[OF finG] by (auto simp: atLeast0LessThan)
  show ?thesis
proof (cases  $bl \in \Gamma$ )
  case False
  have agree:  $\bigwedge x. x \in \Gamma \implies (g(bl := 0)) \ x = g \ x$ 
    using False by auto
  have  $\text{bij\_betw } (g(bl := 0)) \ \Gamma \ \{..< \text{card } \Gamma\}$ 
    using g agree by (metis bij_betw_cong)
  moreover have  $(g(bl := 0)) \ bl = 0$  by simp
  ultimately show ?thesis by blast
next
  case True
  have npos:  $0 < \text{card } \Gamma$  using True finG by (auto simp: card_gt_0_iff)
  have gbl:  $g \ bl < \text{card } \Gamma$  using g True by (auto simp: bij_betw_def)
  let ?s =  $\lambda i. \text{if } i = 0 \text{ then } g \ bl \text{ else if } i = g \ bl \text{ then } 0 \text{ else } i$ 
  have into: ?s '  $\{..< \text{card } \Gamma\} \subseteq \{..< \text{card } \Gamma\}$ 
    using gbl npos by auto
  have inj: inj_on ?s  $\{..< \text{card } \Gamma\}$ 
    by (auto simp: inj_on_def split: if_splits)
  have  $\text{bij\_betw } ?s \ \{..< \text{card } \Gamma\} \ \{..< \text{card } \Gamma\}$ 
    unfolding bij_betw_def
    using inj endo_inj_surj[OF finite_lessThan into inj] by blast
  from bij_betw_trans[OF g this]
  have  $\text{bij\_betw } (?s \circ g) \ \Gamma \ \{..< \text{card } \Gamma\}$  .
  moreover have  $(?s \circ g) \ bl = 0$  using gbl by (auto simp: comp_def)
  ultimately show ?thesis by blast
qed
qed

lemma gamma_enum_anchor:
  assumes finite  $\Gamma$ 
  shows  $\text{bij\_betw } (\text{gamma\_enum } \Gamma \ bl) \ \Gamma \ \{..< \text{card } \Gamma\}$ 
     $\wedge \text{gamma\_enum } \Gamma \ bl = 0$ 
  unfolding gamma_enum_def
  using ex_bij_betw_anchor[OF assms] by (rule someI_ex)

lemma gamma_enum_bij:
  assumes finite  $\Gamma$ 
  shows  $\text{bij\_betw } (\text{gamma\_enum } \Gamma \ bl) \ \Gamma \ \{..< \text{card } \Gamma\}$ 
  using gamma_enum_anchor[OF assms] by simp

```

The anchor: the blank-anchored enumeration sends the blank to 0. This is what makes an all-*BLANK*₄ block decode to *bl* (*decode_blank*).

```

lemma gamma_enum_blank:
  assumes finite  $\Gamma$ 
  shows gamma_enum  $\Gamma$  bl bl = 0
  using gamma_enum_anchor[OF assms] by simp

```

```

lemma gamma_enum_lt_card:
  assumes finite  $\Gamma$  and  $x \in \Gamma$ 
  shows gamma_enum  $\Gamma$  bl x < card  $\Gamma$ 
proof –
  from gamma_enum_bij[OF  $\langle$ finite  $\Gamma$  $\rangle$ ]  $\langle x \in \Gamma \rangle$ 
  have gamma_enum  $\Gamma$  bl x  $\in \{..card \Gamma\}$ 
    by (auto simp: bij_betw_def)
  thus ?thesis by simp
qed

```

The cardinality of Γ fits in b bits: $\text{card } \Gamma \leq 2^b$. This is the structural bound that underwrites injectivity: every enumeration value *gamma_enum* Γ *bl x* < *card* Γ for $x \in \Gamma$ is in the range where *nat_to_bits* is injective.

```

lemma card_le_two_pow_block_width:  $\text{card } \Gamma \leq 2^{\text{block\_width } \Gamma}$ 
proof –
  have ex:  $\exists n. \text{card } \Gamma \leq 2^n$ 
    using less_exp[of card  $\Gamma$ ] by (intro exI[of _ card  $\Gamma$ ]) simp
  let ?m = LEAST  $n. \text{card } \Gamma \leq 2^n$ 
  have m_bound:  $\text{card } \Gamma \leq 2^{?m}$ 
    using LeastI_ex[OF ex] .
  have  $?m \leq \text{block\_width } \Gamma$ 
    unfolding block_width_def by simp
  hence  $(2::\text{nat})^{?m} \leq 2^{\text{block\_width } \Gamma}$ 
    by (rule power_increasing) simp
  with m_bound show ?thesis by linarith
qed

```

Minimality (the lower edge): b is the *smallest* width that indexes all of Γ , so once $\text{card } \Gamma \geq 2$ one bit fewer does not suffice — $2^{(b-1)} < \text{card } \Gamma$. With *card_le_two_pow_block_width* this pins b to $\lceil \log_2 (\text{card } \Gamma) \rceil$ exactly: b is a *step function* of *card* Γ , constant on each band $2^{(b-1)} < \text{card } \Gamma \leq 2^b$ and jumping by one as *card* Γ crosses a power of two ($4 \rightarrow 5$, $8 \rightarrow 9$, $16 \rightarrow 17$). The slowdown factor therefore rises in unit steps at the powers of two, not smoothly with alphabet size. This is minimality for the *uniform* code that indexes every symbol of Γ with the blank at index 0; the *coding* minimum (the non-blank symbols alone) sits one digit lower just past each power of two — see *block_width*.

```

lemma two_pow_block_width_pred_less_card:
  assumes card2:  $2 \leq \text{card } \Gamma$ 
  shows  $2^{(\text{block\_width } \Gamma - 1)} < \text{card } \Gamma$ 
proof –
  have ex:  $\exists n. \text{card } \Gamma \leq 2^n$ 
    using less_exp[of card  $\Gamma$ ] by (intro exI[of _ card  $\Gamma$ ]) simp

```

```

let ?m = LEAST n. card  $\Gamma \leq 2 \wedge n$ 
have notP0:  $\neg \text{card } \Gamma \leq 2 \wedge (0::\text{nat})$  using card2 by simp
have m_pos:  $0 < ?m$ 
proof (rule ccontr)
  assume  $\neg 0 < ?m$ 
  then have ?m = 0 by simp
  then have card  $\Gamma \leq 2 \wedge (0::\text{nat})$  using LeastI_ex[OF ex] by simp
  with notP0 show False by simp
qed
hence bw: block_width  $\Gamma = ?m$  unfolding block_width_def by simp
have block_width  $\Gamma - 1 < ?m$  using bw m_pos by simp
hence  $\neg \text{card } \Gamma \leq 2 \wedge (\text{block\_width } \Gamma - 1)$  by (rule not_less_Least)
thus ?thesis by simp
qed

```

The bit-renderer is injective on inputs bounded by $2 \wedge b$: two values with the same b -bit representation are equal. Proven by induction on b : the head of the bit list determines the high bit $n \text{ div } 2$ (recursive case), and the tail single element determines the low bit $n \text{ mod } 2$; combining via $n = 2 \cdot (n \text{ div } 2) + n \text{ mod } 2$ gives equality.

```

lemma nat_to_bits_inj_bounded:
  assumes n1 <  $2 \wedge k$ 
  and n2 <  $2 \wedge k$ 
  and nat_to_bits k n1 = nat_to_bits k n2
  shows n1 = n2
  using assms
proof (induct k arbitrary: n1 n2)
  case 0
  thus ?case by simp
next
  case (Suc k)
  from  $\langle \text{nat\_to\_bits } (\text{Suc } k) \text{ n1} = \text{nat\_to\_bits } (\text{Suc } k) \text{ n2} \rangle$ 
  have eq:
    nat_to_bits k (n1 div 2)
      @ [if n1 mod 2 = 0 then BIT0 else BIT1]
    = nat_to_bits k (n2 div 2)
      @ [if n2 mod 2 = 0 then BIT0 else BIT1]
  by simp
  hence pref_eq:
    nat_to_bits k (n1 div 2) = nat_to_bits k (n2 div 2)
  and suf_eq:
    (if n1 mod 2 = 0 then BIT0 else BIT1)
      = (if n2 mod 2 = 0 then BIT0 else BIT1)
  by simp_all
  from suf_eq have mod_zero_iff:
    n1 mod 2 = 0  $\longleftrightarrow$  n2 mod 2 = 0
  by (auto split: if_split_asm)
  have n1 mod 2 < 2 and n2 mod 2 < 2 by simp_all
  with mod_zero_iff have mod_eq: n1 mod 2 = n2 mod 2

```



```

  by (cases n1 mod 2 = 0; cases n2 mod 2 = 0) auto
from <n1 < 2 ^ Suc k> have lt1: n1 div 2 < 2 ^ k by simp
from <n2 < 2 ^ Suc k> have lt2: n2 div 2 < 2 ^ k by simp
from Suc.hyps[OF lt1 lt2 pref_eq]
have div_eq: n1 div 2 = n2 div 2 .
have n1 = 2 * (n1 div 2) + n1 mod 2 by simp
also from div_eq mod_eq have ... = 2 * (n2 div 2) + n2 mod 2
  by simp
also have ... = n2 by simp
finally show ?case .
qed

```

Injectivity-on- Γ : distinct source symbols in Γ produce distinct encoder images. This is the load-bearing property for language preservation: no two source symbols collide under the encoder.

```

lemma encode_symbol_inj_on_Gamma:
  assumes finite  $\Gamma$ 
  and  $x \in \Gamma$  and  $y \in \Gamma$ 
  and encode_symbol  $\Gamma$  bl  $x$  = encode_symbol  $\Gamma$  bl  $y$ 
  shows  $x = y$ 
proof -
  from <encode_symbol  $\Gamma$  bl  $x$  = encode_symbol  $\Gamma$  bl  $y$ >
  have bits_eq:
    nat_to_bits (block_width  $\Gamma$ ) (gamma_enum  $\Gamma$  bl  $x$ )
    = nat_to_bits (block_width  $\Gamma$ ) (gamma_enum  $\Gamma$  bl  $y$ )
  by (simp add: encode_symbol_def)
  have lt_x: gamma_enum  $\Gamma$  bl  $x$  < 2 ^ block_width  $\Gamma$ 
  using gamma_enum_lt_card[OF <finite  $\Gamma$ > < $x \in \Gamma$ >, of bl]
    card_le_two_pow_block_width[of  $\Gamma$ ]
  by linarith
  have lt_y: gamma_enum  $\Gamma$  bl  $y$  < 2 ^ block_width  $\Gamma$ 
  using gamma_enum_lt_card[OF <finite  $\Gamma$ > < $y \in \Gamma$ >, of bl]
    card_le_two_pow_block_width[of  $\Gamma$ ]
  by linarith
  from nat_to_bits_inj_bounded[OF lt_x lt_y bits_eq]
  have enum_eq: gamma_enum  $\Gamma$  bl  $x$  = gamma_enum  $\Gamma$  bl  $y$  .
  have inj_on (gamma_enum  $\Gamma$  bl)  $\Gamma$ 
  using gamma_enum_bij[OF <finite  $\Gamma$ >]
  by (simp add: bij_betw_def)
  from this enum_eq < $x \in \Gamma$ > < $y \in \Gamma$ > show ?thesis
  by (rule inj_onD)
qed

```

Structural properties of the input-word encoder. These follow directly from the *concat* (*map* (encode_symbol Γ bl) ...) definition and are useful for the validation-phase reach lemmas in later slices.

```

lemma encode_input_ar_Nil [simp]:
  encode_input_ar  $\Gamma$  bl [] = []
  unfolding encode_input_ar_def by simp

```

```

lemma encode_input_ar_append:
  encode_input_ar  $\Gamma$  bl (w1 @ w2)
    = encode_input_ar  $\Gamma$  bl w1 @ encode_input_ar  $\Gamma$  bl w2
unfolding encode_input_ar_def by simp

```

```

lemma length_encode_input_ar:
  length (encode_input_ar  $\Gamma$  bl w) = block_width  $\Gamma$  * length w
unfolding encode_input_ar_def
by (induct w) auto

```

Uniform-width block indexing: when every block $f\ x$ has the same length K , the $(i \cdot K + j)$ -th cell of $\text{concat } (\text{map } f\ xs)$ is the j -th cell of the i -th block. The list-level fact underlying the initial-tape correspondence: *encode_input_ar* is exactly such a uniform concatenation, every block b wide by *length_encode_symbol*.

```

lemma nth_concat_map_uniform:
  assumes  $K$ :  $\bigwedge x. \text{length } (f\ x) = K$ 
    and  $i$ :  $i < \text{length } xs$ 
    and  $j$ :  $j < K$ 
  shows  $\text{concat } (\text{map } f\ xs) ! (i * K + j) = f\ (xs ! i) ! j$ 
  using  $i$ 
proof (induct xs arbitrary:  $i$ )
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases  $i$ )
  case 0
  have  $\text{concat } (\text{map } f\ (x \# xs)) ! (i * K + j)$ 
    =  $(f\ x @ \text{concat } (\text{map } f\ xs)) ! j$ 
  using 0 by simp
  also have  $\dots = f\ x ! j$ 
  using  $j\ K[\text{of } x]$  by (simp add: nth_append)
  finally show ?thesis using 0 by simp
next
  case (Suc  $i'$ )
  have  $i'\text{len}$ :  $i' < \text{length } xs$  using Cons.prems Suc by simp
  have  $\text{idx}$ :  $i * K + j = \text{length } (f\ x) + (i' * K + j)$ 
  using Suc  $K[\text{of } x]$  by simp
  have  $\text{concat } (\text{map } f\ (x \# xs)) ! (i * K + j)$ 
    =  $\text{concat } (\text{map } f\ xs) ! (i' * K + j)$ 
  using  $\text{idx}$  by (simp add: nth_append)
  also have  $\dots = f\ (xs ! i') ! j$ 
  using Cons.hyps[OF  $i'\text{len}$ ] .
  finally show ?thesis using Suc by simp
qed
qed

```

```

lemma nth_encode_input_ar:
  assumes  $i < \text{length } w$  and  $j < \text{block\_width } \Gamma$ 
  shows  $\text{encode\_input\_ar } \Gamma \text{ bl } w ! (i * \text{block\_width } \Gamma + j)$ 
     $= \text{encode\_symbol } \Gamma \text{ bl } (w ! i) ! j$ 
  unfolding encode_input_ar_def
  by (rule nth_concat_map_uniform[OF length_encode_symbol assms])

```

2.4 Per-symbol decoder helpers

Bit value of a *sym4* cell. *BIT1* contributes 1, every other symbol contributes 0; the non-bit symbols (*BIT0*, *LE4*, *BLANK4*) are conflated to zero — junk-in, junk-out — so the decoder produces a well-defined nat even on malformed input. Validity checking is performed separately by *decode_symbol*'s cell-domain test.

```

fun bit_value :: sym4  $\Rightarrow$  nat where
  bit_value BIT0 = 0
| bit_value BIT1 = 1
| bit_value LE4 = 0
| bit_value BLANK4 = 0

```

Read a *sym4* list as a binary number, MSB at the head (matches *nat_to_bits*'s output shape). Each position contributes *bit_value* times the appropriate power of two.

```

fun bits_to_nat :: sym4 list  $\Rightarrow$  nat where
  bits_to_nat [] = 0
| bits_to_nat (b # bs) = bit_value b * 2 ^ length bs + bits_to_nat bs

```

Snoc-form of *bits_to_nat*: appending a low bit shifts the existing value left and adds the new bit. This is the inductive workhorse for the round-trip lemma — it matches *nat_to_bits*'s recursive shape (which appends the LSB at the tail).

```

lemma bits_to_nat_snoc:
  bits_to_nat (xs @ [b]) = 2 * bits_to_nat xs + bit_value b
by (induct xs) auto

```

Round-trip on bounded naturals: rendering n as a b -bit list and reading it back recovers n , provided $n < 2^b$. Proof by induction on b : the snoc-form of *bits_to_nat* peels off the trailing bit, the IH handles the prefix $n \text{ div } 2$, and $n = 2 \cdot (n \text{ div } 2) + n \text{ mod } 2$ reassembles.

```

lemma bit_value_low_bit:
  bit_value (if  $n \text{ mod } 2 = 0$  then BIT0 else BIT1) =  $n \text{ mod } 2$ 
using mod_less_divisor[of 2  $n$ ] by (auto split: if_split)

```

```

lemma bits_to_nat_nat_to_bits:
  assumes  $n < 2^k$ 
  shows bits_to_nat (nat_to_bits k  $n$ ) =  $n$ 

```

```

using assms
proof (induct k arbitrary: n)
  case 0
  thus ?case by simp
next
  case (Suc k)
  have lt: n div 2 < 2 ^ k
    using <n < 2 ^ Suc k> by simp
  have bits_to_nat (nat_to_bits (Suc k) n)
    = bits_to_nat
      (nat_to_bits k (n div 2)
       @ [if n mod 2 = 0 then BIT0 else BIT1])
    by simp
  also have ...
    = 2 * bits_to_nat (nat_to_bits k (n div 2))
      + bit_value (if n mod 2 = 0 then BIT0 else BIT1)
    by (rule bits_to_nat_snoc)
  also have ... = 2 * (n div 2) + (n mod 2)
    using Suc.hyps[OF lt] bit_value_low_bit[of n] by simp
  also have ... = n by presburger
  finally show ?case .
qed

```

2.5 Per-symbol decoder

Partial inverse of *encode_symbol*. Returns *Some x* when the input list:

1. has length *b* (correct cell count);
2. contains only *BIT0* / *BIT1* cells (no substrate-reserved markers); and
3. has binary interpretation strictly below *card* Γ (in the enumeration range).

Otherwise returns *None*. The validation phase in later slices implements this check as a sequence of substep transitions; the abstract *decode_symbol* partial function is the specification target.

definition *decode_symbol* ::
'a set $\Rightarrow$ *'a* $\Rightarrow$ *sym4 list* $\Rightarrow$ *'a option* **where**
decode_symbol Γ *bl ys* =
 (*if length ys = block_width* Γ
 $\wedge$ *set ys* $\subseteq \{BIT0, BIT1\}$
 $\wedge$ *bits_to_nat ys* $<$ *card* Γ
 then Some (inv_into Γ (*gamma_enum* Γ *bl*) (*bits_to_nat ys*))

Round-trip: decoding an encoded symbol from Γ recovers the source value. All three validity preconditions of *decode_symbol* are satisfied by the encoder image; *inv_into* resolves to *x* via *gamma_enum*'s injectivity-on- Γ .

```

lemma decode_symbol_encode_symbol:
  assumes finite  $\Gamma$  and  $x \in \Gamma$ 
  shows decode_symbol  $\Gamma$  bl (encode_symbol  $\Gamma$  bl  $x$ ) = Some  $x$ 
proof -
  let ?ys = encode_symbol  $\Gamma$  bl  $x$ 
  have len: length ?ys = block_width  $\Gamma$  by simp
  have cd: set ?ys  $\subseteq \{BIT0, BIT1\}$ 
    by (rule encode_symbol_cell_domain)
  have lt_x: gamma_enum  $\Gamma$  bl  $x < 2^{\text{block\_width } \Gamma}$ 
    using gamma_enum_lt_card[OF <finite  $\Gamma$ > < $x \in \Gamma$ >, of bl]
    card_le_two_pow_block_width[of  $\Gamma$ ]
    by linarith
  have bn: bits_to_nat ?ys = gamma_enum  $\Gamma$  bl  $x$ 
    unfolding encode_symbol_def
    by (rule bits_to_nat_nat_to_bits[OF lt_x])
  have lt_card: bits_to_nat ?ys < card  $\Gamma$ 
    using bn gamma_enum_lt_card[OF <finite  $\Gamma$ > < $x \in \Gamma$ >] by simp
  have inj: inj_on (gamma_enum  $\Gamma$  bl)  $\Gamma$ 
    using gamma_enum_bij[OF <finite  $\Gamma$ >]
    by (simp add: bij_betw_def)
  have inv_eq: inv_into  $\Gamma$  (gamma_enum  $\Gamma$  bl) (bits_to_nat ?ys) =  $x$ 
    using bn inv_into_f_f[OF inj < $x \in \Gamma$ >] by simp
  show ?thesis
    unfolding decode_symbol_def
    using len cd lt_card inv_eq by simp
qed

```

A total form of the symbol decoder, indexed by a natural number rather than a bit list. For $n < \text{card } \Gamma$, returns the unique $x \in \Gamma$ with $\text{gamma_enum } \Gamma \text{ bl } x = n$ (via inv_into); for $n \geq \text{card } \Gamma$, returns the blank bl as a defensive fallback. Used by ar_delta_read 's per-bit accumulator to maintain buf as a partial-decoded 'a value: at every per-bit substep before the boundary, the partial nat is strictly less than $2^{(b-1)} < \text{card } \Gamma$ (since b is the *least* n with $\text{card } \Gamma \leq 2^n$), so the fallback is never substantively reached for valid inputs; only the boundary substep can land out of range under non-encoder-image inputs, where the language theorem doesn't care.

```

definition gamma_unenum :: 'a set  $\Rightarrow$  'a  $\Rightarrow$  nat  $\Rightarrow$  'a where
  gamma_unenum  $\Gamma$  bl  $n =$ 
    (if  $n < \text{card } \Gamma$ 
     then inv_into  $\Gamma$  (gamma_enum  $\Gamma$  bl)  $n$ 
     else bl)

```

Enumeration round-trips between gamma_enum and its total inverse gamma_unenum , the arithmetic core the read phase's incremental decode rests on. $\text{gamma_enum} \circ \text{gamma_unenum}$ is the identity on the in-range index set $\{0..<\text{card } \Gamma\}$ (where gamma_unenum resolves to inv_into and gamma_enum is surjective onto that set), and $\text{gamma_unenum} \circ \text{gamma_enum}$ is the identity on Γ (where gamma_enum is injective and

lands below $\text{card } \Gamma$).

```

lemma gamma_enum_gamma_unenum:
  assumes finite  $\Gamma$  and  $n < \text{card } \Gamma$ 
  shows  $\text{gamma\_enum } \Gamma \text{ bl } (\text{gamma\_unenum } \Gamma \text{ bl } n) = n$ 
proof -
  have img:  $\text{gamma\_enum } \Gamma \text{ bl } ' \Gamma = \{..<\text{card } \Gamma\}$ 
    using  $\text{gamma\_enum\_bij}[OF \text{ assms}(1)]$  by (simp add: bij_betw_def)
  have  $n \in \text{gamma\_enum } \Gamma \text{ bl } ' \Gamma$  using  $\text{assms}(2)$  img by simp
  hence  $\text{gamma\_enum } \Gamma \text{ bl } (\text{inv\_into } \Gamma (\text{gamma\_enum } \Gamma \text{ bl } n)) = n$ 
    by (rule f_inv_into_f)
  thus ?thesis using  $\text{assms}(2)$  by (simp add: gamma_unenum_def)
qed

```

```

lemma gamma_unenum_gamma_enum:
  assumes finite  $\Gamma$  and  $x \in \Gamma$ 
  shows  $\text{gamma\_unenum } \Gamma \text{ bl } (\text{gamma\_enum } \Gamma \text{ bl } x) = x$ 
proof -
  have lt:  $\text{gamma\_enum } \Gamma \text{ bl } x < \text{card } \Gamma$ 
    by (rule gamma_enum_lt_card[OF assms])
  have inj: inj_on ( $\text{gamma\_enum } \Gamma \text{ bl}$ )  $\Gamma$ 
    using  $\text{gamma\_enum\_bij}[OF \text{ assms}(1)]$  by (simp add: bij_betw_def)
  have  $\text{inv\_into } \Gamma (\text{gamma\_enum } \Gamma \text{ bl}) (\text{gamma\_enum } \Gamma \text{ bl } x) = x$ 
    by (rule inv_into_f_f[OF inj assms(2)])
  thus ?thesis using lt by (simp add: gamma_unenum_def)
qed

```

Reading an encoded symbol back via *bits_to_nat* recovers its enumeration index, the named form of the local *bn* fact inside *decode_symbol_encode_symbol*. Needed by the read-phase accumulator below.

```

lemma bits_to_nat_encode_symbol:
  assumes finite  $\Gamma$  and  $x \in \Gamma$ 
  shows  $\text{bits\_to\_nat } (\text{encode\_symbol } \Gamma \text{ bl } x) = \text{gamma\_enum } \Gamma \text{ bl } x$ 
proof -
  have lt_x:  $\text{gamma\_enum } \Gamma \text{ bl } x < 2^{\text{block\_width } \Gamma}$ 
    using  $\text{gamma\_enum\_lt\_card}[OF \text{ assms, of bl}]$ 
    card_le_two_pow_block_width[of  $\Gamma$ ]
    by linarith
  show ?thesis
    unfolding encode_symbol_def by (rule bits_to_nat_nat_to_bits[OF lt_x])
qed

```

The read-phase accumulator: one bit-cell folded into the running symbol. Mirrors the per-bit update the read substeps perform — $\text{buf} := \text{gamma_unenum } (2 \cdot \text{gamma_enum } \text{buf} + \text{bit_value } c)$ — reading the *b*-cell block most-significant bit first.

```

definition ar_acc :: 'a set  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  sym4  $\Rightarrow$  'a where
  ar_acc  $\Gamma \text{ bl } b \text{ c} = \text{gamma\_unenum } \Gamma \text{ bl } (2 * \text{gamma\_enum } \Gamma \text{ bl } b + \text{bit\_value } c)$ 

```

Folding the accumulator from the seed $\text{gamma_unenum } 0$ over a bit list whose value is in enumeration range yields exactly $\text{gamma_unenum } (\text{bits_to_nat } ys)$. Proof by *rev_induct*: appending a low bit shifts the running value left and adds it (bits_to_nat_snoc), the prefix value stays in range ($\text{bits_to_nat } zs \leq \text{bits_to_nat } (zs @ [b])$, no separate *take-bound* lemma needed), and the $\text{gamma_enum} \circ \text{gamma_unenum}$ round-trip cancels at each step.

```

lemma foldl_ar_acc_eq:
  assumes finite  $\Gamma$ 
  shows  $\text{set } ys \subseteq \{BIT0, BIT1\} \implies \text{bits\_to\_nat } ys < \text{card } \Gamma$ 
     $\implies \text{foldl } (\text{ar\_acc } \Gamma \text{ bl}) (\text{gamma\_unenum } \Gamma \text{ bl } 0) \text{ ys}$ 
       $= \text{gamma\_unenum } \Gamma \text{ bl } (\text{bits\_to\_nat } ys)$ 
proof (induct ys rule: rev_induct)
  case Nil
  show ?case by simp
next
  case (snoc b zs)
  have  $\text{set\_zs: set } zs \subseteq \{BIT0, BIT1\}$  using snoc.prem1 by simp
  have  $\text{snoc\_eq: bits\_to\_nat } (zs @ [b]) = 2 * \text{bits\_to\_nat } zs + \text{bit\_value } b$ 
    by (rule bits_to_nat_snoc)
  have  $\text{bound\_zs: bits\_to\_nat } zs < \text{card } \Gamma$ 
    using snoc.prem2 snoc_eq by linarith
  have  $\text{IH: foldl } (\text{ar\_acc } \Gamma \text{ bl}) (\text{gamma\_unenum } \Gamma \text{ bl } 0) \text{ zs}$ 
     $= \text{gamma\_unenum } \Gamma \text{ bl } (\text{bits\_to\_nat } zs)$ 
    using snoc.hyps set_zs bound_zs by blast
  have  $\text{ge\_zs: gamma\_enum } \Gamma \text{ bl}$ 
     $(\text{foldl } (\text{ar\_acc } \Gamma \text{ bl}) (\text{gamma\_unenum } \Gamma \text{ bl } 0) \text{ zs})$ 
     $= \text{bits\_to\_nat } zs$ 
    using IH gamma_enum_gamma_unenum[OF assms bound_zs] by simp
  have  $\text{foldl } (\text{ar\_acc } \Gamma \text{ bl}) (\text{gamma\_unenum } \Gamma \text{ bl } 0) (zs @ [b])$ 
     $= \text{ar\_acc } \Gamma \text{ bl}$ 
     $(\text{foldl } (\text{ar\_acc } \Gamma \text{ bl}) (\text{gamma\_unenum } \Gamma \text{ bl } 0) \text{ zs}) \text{ b}$ 
    by simp
  also have  $\dots = \text{gamma\_unenum } \Gamma \text{ bl } (2 * \text{bits\_to\_nat } zs + \text{bit\_value } b)$ 
    by (simp add: ar_acc_def ge_zs)
  also have  $\dots = \text{gamma\_unenum } \Gamma \text{ bl } (\text{bits\_to\_nat } (zs @ [b]))$ 
    using snoc_eq by simp
  finally show ?case .
qed

```

Read-phase decode correctness: folding the accumulator over a symbol's encoding recovers the symbol. This is the arithmetic content the read phase delivers — every tape's *buf* field, after walking its *b*-cell block, holds the source cell's value.

```

lemma foldl_ar_acc_encode_symbol:
  assumes finite  $\Gamma$  and  $x \in \Gamma$ 
  shows  $\text{foldl } (\text{ar\_acc } \Gamma \text{ bl}) (\text{gamma\_unenum } \Gamma \text{ bl } 0) (\text{encode\_symbol } \Gamma \text{ bl } x) =$ 
     $x$ 

```

```

proof –
  have  $cd$ :  $set\ (encode\_symbol\ \Gamma\ bl\ x) \subseteq \{BIT0, BIT1\}$ 
    by (rule  $encode\_symbol\_cell\_domain$ )
  have  $bn$ :  $bits\_to\_nat\ (encode\_symbol\ \Gamma\ bl\ x) = gamma\_enum\ \Gamma\ bl\ x$ 
    by (rule  $bits\_to\_nat\_encode\_symbol[OF\ assms]$ )
  have  $lt$ :  $bits\_to\_nat\ (encode\_symbol\ \Gamma\ bl\ x) < card\ \Gamma$ 
    using  $bn\ gamma\_enum\_lt\_card[OF\ assms]$  by  $simp$ 
  have  $foldl\ (ar\_acc\ \Gamma\ bl)\ (gamma\_unenum\ \Gamma\ bl\ 0)\ (encode\_symbol\ \Gamma\ bl\ x)$ 
     $= gamma\_unenum\ \Gamma\ bl\ (bits\_to\_nat\ (encode\_symbol\ \Gamma\ bl\ x))$ 
    by (rule  $foldl\_ar\_acc\_eq[OF\ assms(1)\ cd\ lt]$ )
  also have  $\dots = gamma\_unenum\ \Gamma\ bl\ (gamma\_enum\ \Gamma\ bl\ x)$  using  $bn$  by  $simp$ 
  also have  $\dots = x$  by (rule  $gamma\_unenum\_gamma\_enum[OF\ assms]$ )
  finally show  $?thesis$  .
qed

```

The decoder's neutral element is the blank: $gamma_unenum\ \Gamma\ bl\ 0 = bl$, since the anchor puts bl at index 0 and the enumeration is injective.

```

lemma  $gamma\_unenum\_zero$ :
  assumes  $finite\ \Gamma$  and  $bl \in \Gamma$ 
  shows  $gamma\_unenum\ \Gamma\ bl\ 0 = bl$ 
proof –
  have  $npos$ :  $0 < card\ \Gamma$  using  $assms$  by (auto  $simp$ :  $card\_gt\_0\_iff$ )
  have  $inj$ :  $inj\_on\ (gamma\_enum\ \Gamma\ bl)\ \Gamma$ 
    using  $gamma\_enum\_bij[OF\ assms(1)]$  by (simp  $add$ :  $bij\_betw\_def$ )
  have  $gamma\_unenum\ \Gamma\ bl\ 0 = inv\_into\ \Gamma\ (gamma\_enum\ \Gamma\ bl)\ 0$ 
    using  $npos$  by (simp  $add$ :  $gamma\_unenum\_def$ )
  also have  $\dots$ 
     $= inv\_into\ \Gamma\ (gamma\_enum\ \Gamma\ bl)\ (gamma\_enum\ \Gamma\ bl\ bl)$ 
    using  $gamma\_enum\_blank[OF\ assms(1)]$  by  $simp$ 
  also have  $\dots = bl$  by (rule  $inv\_into\_f\_f[OF\ inj\ assms(2)]$ )
  finally show  $?thesis$  .
qed

```

An all- $BLANK_4$ block decodes to the blank: bl is a fixpoint of the accumulator on a $BLANK_4$ cell ($2 \cdot gamma_enum\ bl + 0 = 0$, decoding back to bl), so folding any number of $BLANK_4$ cells from bl stays bl . This is the read-phase decode correctness for the blank cell, the $BLANK_4$ counterpart of $foldl_ar_acc_encode_symbol$.

```

lemma  $foldl\_ar\_acc\_blank$ :
  assumes  $finite\ \Gamma$  and  $bl \in \Gamma$ 
  shows  $foldl\ (ar\_acc\ \Gamma\ bl)\ bl\ (replicate\ k\ BLANK_4) = bl$ 
proof (induct  $k$ )
  case  $0$ 
  show  $?case$  by  $simp$ 
next
  case ( $Suc\ k$ )
  have  $step$ :  $ar\_acc\ \Gamma\ bl\ bl\ BLANK_4 = bl$ 
    unfolding  $ar\_acc\_def$ 

```



```

    using gamma_enum_blank[OF assms(1)] gamma_unenum_zero[OF assms]
  by simp
  have foldl (ar_acc  $\Gamma$  bl) bl (replicate (Suc k) BLANK4)
    = foldl (ar_acc  $\Gamma$  bl) (ar_acc  $\Gamma$  bl bl BLANK4)
      (replicate k BLANK4)
  by (simp add: replicate_Suc)
  also have ... = foldl (ar_acc  $\Gamma$  bl) bl (replicate k BLANK4)
    using step by simp
  also have ... = bl by (rule Suc.hyps)
  finally show ?case .
qed

end
theory AlphabetReduction_Stage
  imports AlphabetReduction_Codec
begin

```

2.6 Stage type and validity

Substep counter / phase indicator for the output machine's state set, mirroring AE's *substep_idx*. The five simulation sub-tags *AR_SimRead* / *AR_SimCompute* / *AR_SimWrite* / *AR_SimAdvance* / *AR_SimNext* mark the five phases of one M-step simulation; *AR_HaltAccept* and *AR_HaltReject* are the AR-specific terminal states that map to the substrate's *t_tm M''* and *r_tm M''*. No validation sub-tags: AR's language theorem is quantified over encoder-image inputs only, and AR's per-tape *buf* field is populated on-the-fly during read substeps (no buffer-initialisation prelude to motivate a validation phase). Throughout, *b* abbreviates *block_width* Γ (the per-symbol cell width).

```

datatype ar_substep_idx =
  AR_SimRead | AR_SimCompute | AR_SimWrite | AR_SimAdvance
  | AR_SimNext
  | AR_HaltAccept | AR_HaltReject

instance ar_substep_idx :: finite
proof (intro_classes)
  have (UNIV :: ar_substep_idx set)  $\subseteq$ 
    {AR_SimRead, AR_SimCompute, AR_SimWrite, AR_SimAdvance,
     AR_SimNext,
     AR_HaltAccept, AR_HaltReject}
  using ar_substep_idx.exhaust by blast
  thus finite (UNIV :: ar_substep_idx set)
    using finite_subset by auto
qed

```

Per-tape position-kind tag, the AR analogue of AE's per-tape regime dispatch (*le0* / *le1* / *steady*), generalised to three values because of the LE-1-cell / proper-b-cell layout: *AR_AtLE* for the head at M-position 0 (the

substrate-mandated single $LE4$ cell), $AR_AtFirstProper$ for the head at M-position 1 (immediately past the LE cell, where an L -move lands on the LE cell), and $AR_AtFurtherProper$ for the head at M-position ≥ 2 (where an L -move stays in proper territory). Read by $AR_SimRead$ / $AR_SimWrite$ to dispatch between LE-mode and proper-mode handling, by $AR_SimAdvance$ to compute the per-direction displacement, and updated by $AR_SimAdvance$ per the direction.

```

datatype  $ar\_pos\_kind$  =
   $AR\_AtLE$ 
  |  $AR\_AtFirstProper$ 
  |  $AR\_AtFurtherProper$ 

instance  $ar\_pos\_kind :: finite$ 
proof ( $intro\_classes$ )
  have ( $UNIV :: ar\_pos\_kind\ set$ )  $\subseteq$ 
    { $AR\_AtLE, AR\_AtFirstProper, AR\_AtFurtherProper$ }
  using  $ar\_pos\_kind.exhaust$  by  $blast$ 
  thus  $finite$  ( $UNIV :: ar\_pos\_kind\ set$ )
  using  $finite\_subset$  by  $auto$ 
qed

```

Stage bookkeeping carried in the output machine's state set: substep tag, current-tape index (nat), per-bit counter (nat), per-tape symbol buf ($nat \Rightarrow 'a$: holds the decoded value after read, the symbol-to-encode during write), direction-vector ($nat \Rightarrow dir$: emitted by $AR_SimCompute$, consumed by $AR_SimAdvance$), and per-tape position-kind ($nat \Rightarrow ar_pos_kind$: read by $SimRead/SimWrite$ for LE/proper dispatch, updated by $SimAdvance$ per the direction). The nat counter is loose-typed; ar_valid_stage below imposes the bound $i < 2 * b$ (accommodating the $2b$ -cell L-walks of $AR_SimAdvance$) to recover finiteness of the state set.

```

type_synonym  $'a\ ar\_stage =$ 
   $ar\_substep\_idx$ 
   $\times nat$ 
   $\times nat$ 
   $\times (nat \Rightarrow 'a)$ 
   $\times (nat \Rightarrow dir)$ 
   $\times (nat \Rightarrow ar\_pos\_kind)$ 

```

State-level stage validity: the $nat \Rightarrow 'a$ buf field stores per-tape symbols in $\Gamma \cup \{bl\}$ (so $finite\ \Gamma$ bounds it), and the nat bit-counter is bounded by $2 * b$ (the factor 2 accommodates $AR_SimAdvance$'s L-walks of up to $2b$ cells; AR's δ' keeps the counter's substantively-reachable range within this bound, so the bound is non-restrictive on runtime states but load-bearing for finiteness of the state set). The position-kind field is unconstrained at this level — its type is already finite. Spec-side counterpart used as the stage-component restriction in $alphabet_reduce$'s output state set Q' , making Q' finite from the set-level premise $finite\ \Gamma$ plus the value-level tape-count bound

(*ar_stage_bounded*, next block) — without requiring $UNIV(nat \Rightarrow 'a)$ to be a finite type. Direct AR analogue of *ae_valid_stage*.

definition *ar_valid_stage* ::
 $'a \text{ set} \Rightarrow 'a \Rightarrow 'a \text{ ar_stage} \Rightarrow \text{bool}$ **where**
 $\text{ar_valid_stage } \Gamma \text{ bl stg} \longleftrightarrow$
 $(\text{case stg of } (_, _, i, \text{buf}, _, _) \Rightarrow$
 $i < 2 * \text{block_width } \Gamma \wedge (\forall k. \text{buf } k \in \Gamma \cup \{\text{bl}\}))$

Tape count bound: the value-level isolation of the tape-count fact. The per-tape function fields freeze to their initial values beyond the active tape count K ($\text{buf} \rightarrow \text{bl}$, $\text{dvec} \rightarrow \text{dir}.N$, $\text{posk} \rightarrow \text{AR_AtLE}$), and the current-tape scalar tk stays $\leq K$. Kept *separate* from *ar_valid_stage* (the per-step content invariant threaded through the simulation): the bound is needed only by *alphabet_reduce*'s Q' , *finite_ar_valid_stages*, the three phase combiners, and a single substep-union preservation step — not at the 120 content sites — so the tape count fact is localised rather than smeared. (AE's *ae_valid_stage* folds the bound in directly; that ripple is slated for de-rippling in AE's pre-AFP defensive review, converging on this isolated shape.)

definition *ar_stage_bounded* ::
 $'a \Rightarrow nat \Rightarrow 'a \text{ ar_stage} \Rightarrow \text{bool}$ **where**
 $\text{ar_stage_bounded } \text{bl } K \text{ stg} \longleftrightarrow$
 $(\text{case stg of } (_, tk, _, \text{buf}, \text{dvec}, \text{posk}) \Rightarrow$
 $tk < K$
 $\wedge (\forall j \geq K. \text{buf } j = \text{bl})$
 $\wedge (\forall j \geq K. \text{dvec } j = \text{dir}.N)$
 $\wedge (\forall j \geq K. \text{posk } j = \text{AR_AtLE}))$

lemma *finite_ar_valid_stages*:
fixes $\Gamma :: 'a \text{ set}$ **and** $\text{bl} :: 'a$ **and** $K :: nat$
assumes *finG*: *finite* Γ
shows *finite* $\{\text{stg} :: 'a \text{ ar_stage}.$
 $\text{ar_valid_stage } \Gamma \text{ bl stg} \wedge \text{ar_stage_bounded } \text{bl } K \text{ stg}\}$

proof —

let $?B = \Gamma \cup \{\text{bl}\}$
have *fB*: *finite* $?B$ **using** *finG* **by** *simp*
— Each $nat \Rightarrow X$ field is finite by *finite_tail_const_funcs*: finite codomain, constant beyond K .
let $?bufs = \{\text{buf} :: nat \Rightarrow 'a.$
 $(\forall j. \text{buf } j \in ?B) \wedge (\forall j \geq K. \text{buf } j = \text{bl})\}$
let $?dvecs = \{\text{dvec} :: nat \Rightarrow \text{dir}.$
 $(\forall j. \text{dvec } j \in (UNIV :: \text{dir set}))$
 $\wedge (\forall j \geq K. \text{dvec } j = \text{dir}.N)\}$
let $?posks = \{\text{posk} :: nat \Rightarrow \text{ar_pos_kind}.$
 $(\forall j. \text{posk } j \in (UNIV :: \text{ar_pos_kind set}))$
 $\wedge (\forall j \geq K. \text{posk } j = \text{AR_AtLE})\}$
have *fbufs*: *finite* $?bufs$
using *finite_tail_const_funcs*[*OF fB*, *of K bl*] .
have *fdvecs*: *finite* $?dvecs$

```

using finite_tail_const_funcs[OF finite_UNIV_dir, of K dir.N] .
have fposks: finite ?posks
using finite_tail_const_funcs[OF finite_UNIV, of K AR_AtLE] .
let ?ENV = (UNIV :: ar_substep_idx set) × {..K} × {.. $2 * \text{block\_width } \Gamma$ }
          × ?bufs × ?dvecs × ?posks
have {stg :: 'a ar_stage.
      ar_valid_stage  $\Gamma$  bl stg  $\wedge$  ar_stage_bounded bl K stg}  $\subseteq$  ?ENV
unfolding ar_valid_stage_def ar_stage_bounded_def
by (auto simp: mem_Times_iff split: prod.splits)
moreover have finite ?ENV
using fbufs fdvecs fposks by (intro finite_cartesian_product) simp_all
ultimately show ?thesis by (rule finite_subset)
qed

```

2.7 Per-tape enumeration helpers

Per-tape walk helpers. Under value-level tape count a tape index is a natural number directly (tape kk sits at position kk , the active tapes being $\{.. k_tm M \}), so the abstract enumeration of the old ' k ' tape-index type collapses: k_idx and k_unidx are the identity and k_succ is Suc . They are retained as thin wrappers through the tape count migration (inlined in the cleanup sweep) so the per-phase walk lemmas keep their shape. The per-tape substep relations (ar_delta_read , ar_delta_write , $ar_delta_advance$) walk tapes in order, processing tape kk , advancing to $k_succ\ kk$, and leaving the phase when $is_last_k\ M\ kk$ ($Suc\ kk = k_tm\ M$) fires.$

definition $k_idx :: nat \Rightarrow nat$ **where**
 $k_idx\ kk = kk$

definition $k_unidx :: nat \Rightarrow nat$ **where**
 $k_unidx\ j = j$

definition $k_succ :: nat \Rightarrow nat$ **where**
 $k_succ\ kk = Suc\ kk$

definition $is_last_k :: ('q, 'a) mttm \Rightarrow nat \Rightarrow bool$ **where**
 $is_last_k\ M\ kk \longleftrightarrow Suc\ kk = k_tm\ M$

Navigation facts the per-phase tape walks consume. Under the identity collapse these are immediate: k_idx and k_unidx fix 0 and k_idx is injective, the successor of $k_unidx\ j$ is $k_unidx\ (Suc\ j)$, and $is_last_k\ M\ (k_unidx\ j)$ reduces to $Suc\ j = k_tm\ M$. The bounded hypotheses are retained so existing call sites keep their shape across the migration.

lemma $k_idx_zero: k_idx\ 0 = 0$
by (simp add: k_idx_def)

lemma $k_unidx_zero: k_unidx\ 0 = 0$
by (simp add: k_unidx_def)

lemma k_idx_inj : $inj_on\ k_idx\ UNIV$
by ($simp\ add:\ k_idx_def\ inj_on_def$)

lemma k_idx_unidx :
assumes $j < k_tm\ M$
shows $k_idx\ (k_unidx\ j) = j$
by ($simp\ add:\ k_idx_def\ k_unidx_def$)

lemma k_succ_unidx :
assumes $Suc\ j < k_tm\ M$
shows $k_succ\ (k_unidx\ j) = k_unidx\ (Suc\ j)$
by ($simp\ add:\ k_succ_def\ k_unidx_def$)

lemma $is_last_k_unidx$:
assumes $j < k_tm\ M$
shows $is_last_k\ M\ (k_unidx\ j) \longleftrightarrow Suc\ j = k_tm\ M$
by ($simp\ add:\ is_last_k_def\ k_unidx_def$)

2.8 Per-substep dispatch helpers

Displacement of an $AR_SimAdvance$ walk on one tape, dispatched on the tape's current direction (from $dvec$) and position-kind. Encodes the displacement table: zero for R (head already at $sim_pos\ (p+1)$ after the per-tape write phase), one for N from AR_AtLE (single L -move back to position 0), b for N from proper (back to $sim_pos\ p$), $Suc\ b$ for L from $AR_AtFirstProper$ (back to position 0), and $2 \cdot b$ for L from $AR_AtFurtherProper$ (back to $sim_pos\ (p-1)$). The L -from- AR_AtLE case is forbidden by the substrate's δLE invariant; setting its displacement to 0 here makes the advance relation exclude that case naturally (no stepping arm is satisfied, no boundary condition fires).

fun $ar_disp :: nat \Rightarrow dir \Rightarrow ar_pos_kind \Rightarrow nat$ **where**
 $ar_disp\ _ \ dir.R\ _ = 0$
 $| ar_disp\ _ \ dir.N\ AR_AtLE = 1$
 $| ar_disp\ k\ dir.N\ AR_AtFirstProper = k$
 $| ar_disp\ k\ dir.N\ AR_AtFurtherProper = k$
 $| ar_disp\ _ \ dir.L\ AR_AtLE = 0$
 $| ar_disp\ k\ dir.L\ AR_AtFirstProper = Suc\ k$
 $| ar_disp\ k\ dir.L\ AR_AtFurtherProper = 2 * k$

Per-tape position-kind transition under $AR_SimAdvance$, dispatched on (direction, pre-advance position-kind). Encodes the transition table: R -advance bumps position-kind up the ladder ($AR_AtLE \rightarrow AR_AtFirstProper \rightarrow AR_AtFurtherProper$; the last is a fixed point); N -advance preserves position-kind; L -advance from $AR_AtFirstProper$ drops to AR_AtLE ; L -advance from $AR_AtFurtherProper$ remains $AR_AtFurtherProper$ as a defensive default — the actual post-advance posi-

tion can be either $AR_AtFirstProper$ (if $p - 1 = 1$) or $AR_AtFurtherProper$ (otherwise), and a one-cell look-back substep at the next $AR_SimRead$ refines this distinction. The L -from- AR_AtLE case is forbidden by δLE and is never substantively reached; the dummy mapping to AR_AtLE below keeps ar_newpos total.

```
fun  $ar\_newpos :: dir \Rightarrow ar\_pos\_kind \Rightarrow ar\_pos\_kind$  where
   $ar\_newpos\ dir.R\ AR\_AtLE = AR\_AtFirstProper$ 
|  $ar\_newpos\ dir.R\ AR\_AtFirstProper = AR\_AtFurtherProper$ 
|  $ar\_newpos\ dir.R\ AR\_AtFurtherProper = AR\_AtFurtherProper$ 
|  $ar\_newpos\ dir.N\ AR\_AtLE = AR\_AtLE$ 
|  $ar\_newpos\ dir.N\ AR\_AtFirstProper = AR\_AtFirstProper$ 
|  $ar\_newpos\ dir.N\ AR\_AtFurtherProper = AR\_AtFurtherProper$ 
|  $ar\_newpos\ dir.L\ AR\_AtLE = AR\_AtLE$ 
|  $ar\_newpos\ dir.L\ AR\_AtFirstProper = AR\_AtLE$ 
|  $ar\_newpos\ dir.L\ AR\_AtFurtherProper = AR\_AtFurtherProper$ 
```

The cell-level write image for a per-tape symbol in the proper region. Returns the j -th sym4 cell of the symbol's image: $BLANK_4$ for every position if the symbol is the blank bl (the blank's cell-repr is b consecutive blanks); the j -th bit of $encode_symbol\ \Gamma\ bl\ x$ otherwise. Used by ar_delta_write 's proper-arm forward-write phase to emit one cell per substep. The $le_tm\ M$ case is excluded — ar_delta_write 's LE-arm short-circuits writes for that symbol, so the head never enters the forward-write phase with $buf\ tk = le_tm\ M$.

```
definition  $write\_bit ::$ 
   $'a\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow nat \Rightarrow sym4$  where
   $write\_bit\ \Gamma\ bl\ x\ j =$ 
     $(if\ x = bl\ then\ BLANK_4\ else\ encode\_symbol\ \Gamma\ bl\ x\ !\ j)$ 
```

```
end
theory  $AlphabetReduction\_Delta$ 
  imports  $AlphabetReduction\_Stage$ 
begin
```

2.9 Substep transition relations

The output machine's δ' is defined as a union of five per-substep transition relations, mirroring AE's $alphabet_enlarge_delta$ shape. Each substep relation is a set of substrate-shape 5-tuples $((q, stg), a, (q', stg'), a', d)$ where q ranges over $Q_tm\ M$, stg over $'a\ ar_stage$, a / a' over $nat \Rightarrow sym4$, and d over $nat \Rightarrow dir$. The five substep relations are defined below, one per simulation phase.

No validation cluster: AR's language theorem is quantified over encoder-image inputs only, so non-canonical sym4 inputs are outside the theorem's scope and M's behaviour on them is unconstrained.

The union is intersected with two global restrictions: LE-preservation

(no transition forges LE_4 out of a non- LE_4 cell, matching the substrate's δLE invariant) and ar_valid_stage -membership on both source and target stages (matching the non-product Q' shape). Throughout, b abbreviates $block_width \Gamma$ (the per-symbol cell width).

definition $ar_delta_read ::$

$(q, a) mttm$
 $\Rightarrow ((q \times a \text{ ar_stage})$
 $\times (nat \Rightarrow sym_4)$
 $\times (q \times a \text{ ar_stage})$
 $\times (nat \Rightarrow sym_4)$
 $\times (nat \Rightarrow dir)) \text{ set where}$
 $ar_delta_read M =$
 — LE-arm, non-last tape: head at position 0 reads LE_4 , sets $buf\ tk := le_tm\ M$ directly (no accumulator needed for the single LE-cell), advances R on tk to position 1, and transitions to the next tape's read. $posk\ tk$ stays AR_AtLE throughout this substep — $SimAdvance$ will update it later when the head leaves the LE region.
 $\{((q, AR_SimRead, tk, 0, buf, dvec, posk), a,$
 $(q, AR_SimRead, k_succ\ tk, 0,$
 $buf(tk := le_tm\ M), dvec, posk), a, d) \mid$
 $q\ tk\ buf\ dvec\ posk\ a\ d.$
 $q \in Q_tm\ M$
 $\wedge \neg is_last_k\ M\ tk$
 $\wedge posk\ tk = AR_AtLE$
 $\wedge a\ tk = LE_4$
 $\wedge d = (\lambda kk. \text{ if } kk = tk \text{ then } dir.R \text{ else } dir.N)\}$
 $\cup$
 — LE-arm, last tape: same single-substep LE-read, transitions to $AR_SimCompute$ instead of the next tape.
 $\{((q, AR_SimRead, tk, 0, buf, dvec, posk), a,$
 $(q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf(tk := le_tm\ M), dvec, posk), a, d) \mid$
 $q\ tk\ buf\ dvec\ posk\ a\ d.$
 $q \in Q_tm\ M$
 $\wedge is_last_k\ M\ tk$
 $\wedge posk\ tk = AR_AtLE$
 $\wedge a\ tk = LE_4$
 $\wedge d = (\lambda kk. \text{ if } kk = tk \text{ then } dir.R \text{ else } dir.N)\}$
 $\cup$
 — Proper-arm look-back step 1 ($i = 0$): head at $sim_pos(p)$ moves L on tk to $sim_pos(p) - 1$. The cell at $sim_pos(p)$ is some $BIT0$ / $BIT1$ / $BLANK_4$ (not LE_4 ; enforced as a δLE precondition). Buf , $posk$, $dvec$ unchanged; substep transitions to $i = 1$.
 $\{((q, AR_SimRead, tk, 0, buf, dvec, posk), a,$
 $(q, AR_SimRead, tk, Suc\ 0, buf, dvec, posk), a, d) \mid$
 $q\ tk\ buf\ dvec\ posk\ a\ d.$
 $q \in Q_tm\ M$
 $\wedge posk\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
 $\wedge a\ tk \neq LE_4$
 $\wedge d = (\lambda kk. \text{ if } kk = tk \text{ then } dir.L \text{ else } dir.N)\}$

$\cup$
 — Proper-arm look-back step 2 ($i = 1$): head at $\text{sim_pos}(p) - 1$ reads the cell there; if $LE4$, the head was at $\text{sim_pos } 1 = 1$ in the previous step so refine $\text{posk } tk := AR_AtFirstProper$; otherwise refine to $AR_AtFurtherProper$. Reset $\text{buf } tk := \text{gamma_unenum } \Gamma \text{ bl } 0$ (the partial-decoded “0 bits read so far” symbol) to prepare for the per-bit accumulator below. R -move back to $\text{sim_pos}(p)$; δLE with $a \text{ tk} = LE4$ is fine because the direction is R . Transitions to $i = 2$.

$\{((q, AR_SimRead, tk, Suc \ 0, buf, dvec, posk), a,$
 $(q, AR_SimRead, tk, Suc (Suc \ 0),$
 $buf(tk := \text{gamma_unenum } (\Gamma_tm \ M) (bl_tm \ M) \ 0),$
 $dvec,$
 $posk(tk := \text{if } a \text{ tk} = LE4 \text{ then } AR_AtFirstProper$
 $\text{else } AR_AtFurtherProper)), a, d) \mid$
 $q \text{ tk buf dvec posk } a \ d.$
 $q \in Q_tm \ M$
 $\wedge posk \text{ tk} \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
 $\wedge d = (\lambda kk. \text{if } kk = tk \text{ then } dir.R \text{ else } dir.N)\}$

$\cup$
 — Proper-arm per-bit stepping ($2 \leq i \leq b$): head at $\text{sim_pos}(p) + (i - 2)$, read cell, accumulate bit_value into the partial-decoded $\text{buf } tk$ via the $\text{gamma_enum} / \text{gamma_unenum}$ roundtrip $\text{partial}' = 2 \cdot \text{gamma_enum } (buf \ tk) + \text{bit_value } (a \ tk)$, $\text{buf}' \ tk := \text{gamma_unenum } \text{partial}'$. R -move on tk . Transitions to $i + 1$ within $AR_SimRead$; the next per-bit substep continues the accumulator chain.

$\{((q, AR_SimRead, tk, i, buf, dvec, posk), a,$
 $(q, AR_SimRead, tk, Suc \ i,$
 $buf(tk := \text{gamma_unenum } (\Gamma_tm \ M) (bl_tm \ M)$
 $(2 * \text{gamma_enum } (\Gamma_tm \ M) (bl_tm \ M) (buf \ tk)$
 $+ \text{bit_value } (a \ tk))),$
 $dvec, posk), a, d) \mid$
 $q \text{ tk } i \text{ buf dvec posk } a \ d.$
 $q \in Q_tm \ M$
 $\wedge posk \text{ tk} \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
 $\wedge 2 \leq i$
 $\wedge Suc \ i \leq Suc (\text{block_width } (\Gamma_tm \ M))$
 $\wedge d = (\lambda kk. \text{if } kk = tk \text{ then } dir.R \text{ else } dir.N)\}$

$\cup$
 — Proper-arm per-bit boundary ($i = Suc \ (b)$), non-last tape: same accumulator step as above on the last bit ($j = b - 1$), $\text{buf}' \ tk$ holds the fully-decoded M -symbol ($\in \Gamma$ for valid encoder images; falls back to $bl_tm \ M$ for non-encoder inputs), then transitions to the same phase on $k_succ \ tk$ with bit-counter reset to 0.

$\{((q, AR_SimRead, tk, i, buf, dvec, posk), a,$
 $(q, AR_SimRead, k_succ \ tk, 0,$
 $buf(tk := \text{gamma_unenum } (\Gamma_tm \ M) (bl_tm \ M)$
 $(2 * \text{gamma_enum } (\Gamma_tm \ M) (bl_tm \ M) (buf \ tk)$
 $+ \text{bit_value } (a \ tk))),$
 $dvec, posk), a, d) \mid$
 $q \text{ tk } i \text{ buf dvec posk } a \ d.$
 $q \in Q_tm \ M$
 $\wedge \neg is_last_k \ M \ tk$

$\wedge \text{posk } tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
 $\wedge i = \text{Suc } (\text{block_width } (\Gamma_tm M))$
 $\wedge d = (\lambda kk. \text{if } kk = tk \text{ then } dir.R \text{ else } dir.N)\}$
 $\cup$
 — Proper-arm per-bit boundary, last tape: same last-bit accumulator step, transitions to $AR_SimCompute$ with current-tape reset to $k_unidx 0$.
 $\{((q, AR_SimRead, tk, i, buf, dvec, posk), a,$
 $(q, AR_SimCompute, k_unidx 0, 0,$
 $buf(tk := \text{gamma_unenum } (\Gamma_tm M) (bl_tm M)$
 $(2 * \text{gamma_enum } (\Gamma_tm M) (bl_tm M) (buf tk)$
 $+ \text{bit_value } (a tk))),$
 $dvec, posk), a, d) \mid$
 $q \text{ tk } i \text{ buf } dvec \text{ posk } a \text{ d.}$
 $q \in Q_tm M$
 $\wedge is_last_k M tk$
 $\wedge \text{posk } tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
 $\wedge i = \text{Suc } (\text{block_width } (\Gamma_tm M))$
 $\wedge d = (\lambda kk. \text{if } kk = tk \text{ then } dir.R \text{ else } dir.N)\}$

definition $ar_delta_compute ::$

$(q, a) \text{ mttm}$
 $\Rightarrow ((q \times a \text{ ar_stage})$
 $\times (\text{nat} \Rightarrow \text{sym4})$
 $\times (q \times a \text{ ar_stage})$
 $\times (\text{nat} \Rightarrow \text{sym4})$
 $\times (\text{nat} \Rightarrow \text{dir})) \text{ set where}$
 $ar_delta_compute M =$
 $\{((q, AR_SimCompute, tk, i, buf, dvec, posk), a,$
 $(q', AR_SimWrite, 0, 0, m_a', m_d, posk), a, d) \mid$
 $q \text{ buf } q' m_a' m_d \text{ tk } i \text{ dvec } posk \text{ a d.}$
 $(q, buf, q', m_a', m_d) \in \text{delta_tm } M$
 $\wedge d = (\lambda_ . \text{dir}.N)\}$

— One δ' -tuple per M 's δ -tuple. Matches the source-stage buf field (the decoded per-tape symbol vector, produced by $AR_SimRead$) against the M-side read symbol vector of the M - δ -tuple (q, buf, q', m_a', m_d) , and threads the M-side output: new M-state q' , write-back vector m_a' (next phase's per-tape symbols to encode), and direction-vector m_d (consumed by $AR_SimAdvance$). The substrate-side read a and write $a' = \bar{a}$ are unconstrained at this substep (no head moves, no writes); direction is constant N . The per-tape $posk$ carries through unchanged (head positions don't change during compute). No halt-state shortcut: M-side halt detection is centralised at ar_delta_next 's three-arm dispatch.

definition $ar_delta_write ::$

$(q, a) \text{ mttm}$
 $\Rightarrow ((q \times a \text{ ar_stage})$
 $\times (\text{nat} \Rightarrow \text{sym4})$
 $\times (q \times a \text{ ar_stage})$
 $\times (\text{nat} \Rightarrow \text{sym4})$
 $\times (\text{nat} \Rightarrow \text{dir})) \text{ set where}$

$ar_delta_write\ M =$
 — LE-arm, non-last tape: $posk\ tk = AR_AtLE$ means tape tk 's head sits on the reduced boundary ($sim_pos\ 0 = 0$), whose $LE4$ cell need not be rewritten (δLE requires writing le back, which is exactly the present cell). Keyed on the position-kind flag, not on $buf\ tk = le_tm\ M$, so a source machine that writes le off the boundary (a tape cut) is handled by the proper arm below, as ordinary data. Skip the per-tape write phase entirely: single substep with all directions N , transitioning to the same phase on the next tape with bit-counter still at 0 .
 $\{((q, AR_SimWrite, tk, 0, buf, dvec, posk), a,$
 $(q, AR_SimWrite, k_succ\ tk, 0, buf, dvec, posk), a, d) \mid$
 $q\ tk\ buf\ dvec\ posk\ a\ d.$
 $q \in Q_tm\ M$
 $\wedge\ posk\ tk = AR_AtLE$
 $\wedge\ \neg\ is_last_k\ M\ tk$
 $\wedge\ d = (\lambda_.\ dir.N)\}$
 $\cup$
 — LE-arm, last tape: transitions to $AR_SimAdvance$ with the current-tape index reset to $k_unidx\ 0$.
 $\{((q, AR_SimWrite, tk, 0, buf, dvec, posk), a,$
 $(q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk), a, d) \mid$
 $q\ tk\ buf\ dvec\ posk\ a\ d.$
 $q \in Q_tm\ M$
 $\wedge\ posk\ tk = AR_AtLE$
 $\wedge\ is_last_k\ M\ tk$
 $\wedge\ d = (\lambda_.\ dir.N)\}$
 $\cup$
 — Proper-arm, back-walk phase ($0 \leq i < b$): head moves L on tk , N elsewhere; no writes. At $i = b - 1$ the substep transitions to $i = b$, starting the forward-write phase below. Both phases share the $AR_SimWrite$ substep tag; the bit-counter distinguishes them. $a\ tk \neq LE4$ enforced for δLE .
 $\{((q, AR_SimWrite, tk, i, buf, dvec, posk), a,$
 $(q, AR_SimWrite, tk, Suc\ i, buf, dvec, posk), a, d) \mid$
 $q\ tk\ i\ buf\ dvec\ posk\ a\ d.$
 $q \in Q_tm\ M$
 $\wedge\ posk\ tk \neq AR_AtLE$
 $\wedge\ a\ tk \neq LE4$
 $\wedge\ Suc\ i \leq block_width\ (\Gamma_tm\ M)$
 $\wedge\ d = (\lambda kk.\ if\ kk = tk\ then\ dir.L\ else\ dir.N)\}$
 $\cup$
 — Proper-arm, forward-write stepping phase ($b \leq i < 2b - 1$): write $write_bit\ \Gamma\ bl\ (buf\ tk)\ (i - b)$ at tk , R on tk , N elsewhere. All other tapes have their cells preserved. The bit-counter advances; the next substep continues forward-write.
 $\{((q, AR_SimWrite, tk, i, buf, dvec, posk), a,$
 $(q, AR_SimWrite, tk, Suc\ i, buf, dvec, posk), a', d) \mid$
 $q\ tk\ i\ buf\ dvec\ posk\ a\ a'\ d.$
 $q \in Q_tm\ M$
 $\wedge\ posk\ tk \neq AR_AtLE$
 $\wedge\ block_width\ (\Gamma_tm\ M) \leq i$
 $\wedge\ Suc\ i < 2 * block_width\ (\Gamma_tm\ M)$

$$\begin{aligned}
& \wedge a' = (\lambda kk. \text{ if } kk = tk \\
& \quad \text{ then write_bit } (\Gamma_tm\ M) \ (bl_tm\ M) \\
& \quad \quad (buf\ tk) \\
& \quad \quad (i - block_width\ (\Gamma_tm\ M)) \\
& \quad \text{ else } a\ kk) \\
& \wedge d = (\lambda kk. \text{ if } kk = tk \text{ then } dir.R \text{ else } dir.N) \} \\
& \cup \\
& \text{--- Proper-arm, forward-write boundary } (Suc\ i = 2b), \text{ non-last tape: last cell of } \\
& \text{cell_repr } (buf\ tk) \text{ is written, head moves } R \text{ on } tk, \text{ transitions to the same phase on } \\
& k_succ\ tk \text{ with bit-counter reset. Cells on other tapes preserved.} \\
& \{((q, AR_SimWrite, tk, i, buf, dvec, posk), a, \\
& \quad (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec, posk), a', d) \mid \\
& \quad q\ tk\ i\ buf\ dvec\ posk\ a\ a'\ d. \\
& \quad q \in Q_tm\ M \\
& \quad \wedge posk\ tk \neq AR_AtLE \\
& \quad \wedge \neg is_last_k\ M\ tk \\
& \quad \wedge Suc\ i = 2 * block_width\ (\Gamma_tm\ M) \\
& \quad \wedge a' = (\lambda kk. \text{ if } kk = tk \\
& \quad \quad \text{ then write_bit } (\Gamma_tm\ M) \ (bl_tm\ M) \\
& \quad \quad \quad (buf\ tk) \\
& \quad \quad \quad (i - block_width\ (\Gamma_tm\ M)) \\
& \quad \quad \text{ else } a\ kk) \\
& \quad \wedge d = (\lambda kk. \text{ if } kk = tk \text{ then } dir.R \text{ else } dir.N) \} \\
& \cup \\
& \text{--- Proper-arm, forward-write boundary, last tape: same last-cell write as above,} \\
& \text{transitions to } AR_SimAdvance \text{ with the current-tape index reset to } k_unidx\ 0. \\
& \{((q, AR_SimWrite, tk, i, buf, dvec, posk), a, \\
& \quad (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk), a', d) \mid \\
& \quad q\ tk\ i\ buf\ dvec\ posk\ a\ a'\ d. \\
& \quad q \in Q_tm\ M \\
& \quad \wedge posk\ tk \neq AR_AtLE \\
& \quad \wedge is_last_k\ M\ tk \\
& \quad \wedge Suc\ i = 2 * block_width\ (\Gamma_tm\ M) \\
& \quad \wedge a' = (\lambda kk. \text{ if } kk = tk \\
& \quad \quad \text{ then write_bit } (\Gamma_tm\ M) \ (bl_tm\ M) \\
& \quad \quad \quad (buf\ tk) \\
& \quad \quad \quad (i - block_width\ (\Gamma_tm\ M)) \\
& \quad \quad \text{ else } a\ kk) \\
& \quad \wedge d = (\lambda kk. \text{ if } kk = tk \text{ then } dir.R \text{ else } dir.N) \}
\end{aligned}$$

definition *ar_delta_advance* ::

$$\begin{aligned}
& ('q, 'a) mttm \\
& \Rightarrow (('q \times 'a\ ar_stage) \\
& \quad \times (nat \Rightarrow sym4) \\
& \quad \times ('q \times 'a\ ar_stage) \\
& \quad \times (nat \Rightarrow sym4) \\
& \quad \times (nat \Rightarrow dir)) \text{ set } \mathbf{where}
\end{aligned}$$

ar_delta_advance *M* =

--- Stepping arm: head moves *L* on tape *tk*, *N* elsewhere; stays in *AR_SimAdvance*

with the bit-counter advanced. Active when $Suc\ i$ is strictly below the per-tape displacement $ar_disp\ b\ (dvec\ tk)\ (posk\ tk)$; the R - and L -from- AR_AtLE cases have displacement 0 , so no stepping tuple fires; for N -from- AR_AtLE the displacement is 1 , so stepping never fires and the single substep goes through one of the boundary arms. The read symbol on tk must not be $LE4$ (δLE : an L -move from a tape reading $LE4$ is forbidden).

$$\begin{aligned} &\{((q, AR_SimAdvance, tk, i, buf, dvec, posk), a, \\ &\quad (q, AR_SimAdvance, tk, Suc\ i, buf, dvec, posk), a, d) \mid \\ &\quad q\ tk\ i\ buf\ dvec\ posk\ a\ d. \\ &\quad q \in Q_tm\ M \\ &\quad \wedge a\ tk \neq LE4 \\ &\quad \wedge Suc\ i < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk) \\ &\quad \wedge d = (\lambda kk. \text{if } kk = tk \text{ then } dir.L \text{ else } dir.N)\} \end{aligned}$$

$\cup$

— Boundary arm, non-last tape. Two firing sub-cases: the R -direction case (zero displacement, no L -move; $i = 0$ and the entire direction vector is N), and the non- R case at $Suc\ i$ equal to the displacement (one last L -move on tk ; $a\ tk \neq LE4$ enforced for δLE). Both sub-cases transition to the same phase on the next tape $k_succ\ tk$ (bit-counter reset to 0) with $posk\ tk$ updated by ar_newpos ; all other tapes' $posk$ entries are preserved.

$$\begin{aligned} &\{((q, AR_SimAdvance, tk, i, buf, dvec, posk), a, \\ &\quad (q, AR_SimAdvance, k_succ\ tk, 0, buf, dvec, \\ &\quad posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk))), a, d) \mid \\ &\quad q\ tk\ i\ buf\ dvec\ posk\ a\ d. \\ &\quad q \in Q_tm\ M \\ &\quad \wedge \neg is_last_k\ M\ tk \\ &\quad \wedge ((dvec\ tk = dir.R \wedge i = 0) \\ &\quad \vee (dvec\ tk \neq dir.R \\ &\quad \wedge a\ tk \neq LE4 \\ &\quad \wedge Suc\ i = ar_disp\ (block_width\ (\Gamma_tm\ M)) \\ &\quad \quad (dvec\ tk)\ (posk\ tk))) \\ &\quad \wedge d = (\lambda kk. \text{if } kk = tk \wedge dvec\ tk \neq dir.R \\ &\quad \quad \text{then } dir.L \text{ else } dir.N)\} \end{aligned}$$

$\cup$

— Boundary arm, last tape. Same firing sub-cases as the non-last-tape arm; transitions to $AR_SimNext$ instead of advancing tk . The current-tape field is reset to $k_unidx\ 0$ (the first tape in the enumeration; matches $AR_SimNext$'s starting convention and the next $AR_SimRead$'s initial tape).

$$\begin{aligned} &\{((q, AR_SimAdvance, tk, i, buf, dvec, posk), a, \\ &\quad (q, AR_SimNext, k_unidx\ 0, 0, buf, dvec, \\ &\quad posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk))), a, d) \mid \\ &\quad q\ tk\ i\ buf\ dvec\ posk\ a\ d. \\ &\quad q \in Q_tm\ M \\ &\quad \wedge is_last_k\ M\ tk \\ &\quad \wedge ((dvec\ tk = dir.R \wedge i = 0) \\ &\quad \vee (dvec\ tk \neq dir.R \\ &\quad \wedge a\ tk \neq LE4 \\ &\quad \wedge Suc\ i = ar_disp\ (block_width\ (\Gamma_tm\ M)) \\ &\quad \quad (dvec\ tk)\ (posk\ tk))) \end{aligned}$$

$$\wedge d = (\lambda kk. \text{ if } kk = tk \wedge dvec\ tk \neq dir.R \\ \text{ then } dir.L \text{ else } dir.N)$$

definition *ar_delta_next* ::

$$\begin{aligned} &('q, 'a) mttm \\ &\Rightarrow (('q \times 'a\ ar_stage) \\ &\quad \times (nat \Rightarrow sym4) \\ &\quad \times ('q \times 'a\ ar_stage) \\ &\quad \times (nat \Rightarrow sym4) \\ &\quad \times (nat \Rightarrow dir)) \text{ set where} \\ ar_delta_next\ M = & \\ &\{((q, AR_SimNext, tk, i, buf, dvec, posk), a, \\ &\quad (q, AR_SimRead, 0, 0, buf, dvec, posk), a, d) \mid \\ &\quad q\ tk\ i\ buf\ dvec\ posk\ a\ d. \\ &\quad q \in Q_tm\ M \wedge q \neq t_tm\ M \wedge q \neq r_tm\ M \\ &\quad \wedge d = (\lambda_ . dir.N)\} \\ \cup & \\ &\{((q, AR_SimNext, tk, i, buf, dvec, posk), a, \\ &\quad (q, AR_HaltAccept, 0, 0, \\ &\quad (\lambda_ . bl_tm\ M), (\lambda_ . dir.N), (\lambda_ . AR_AtLE)), a, d) \mid \\ &\quad q\ tk\ i\ buf\ dvec\ posk\ a\ d. \\ &\quad q = t_tm\ M \\ &\quad \wedge d = (\lambda_ . dir.N)\} \\ \cup & \\ &\{((q, AR_SimNext, tk, i, buf, dvec, posk), a, \\ &\quad (q, AR_HaltReject, 0, 0, \\ &\quad (\lambda_ . bl_tm\ M), (\lambda_ . dir.N), (\lambda_ . AR_AtLE)), a, d) \mid \\ &\quad q\ tk\ i\ buf\ dvec\ posk\ a\ d. \\ &\quad q = r_tm\ M \\ &\quad \wedge d = (\lambda_ . dir.N)\} \end{aligned}$$

— End-of-M-step handshake. Three arms dispatch on the M-state q : non-terminal routes to $AR_SimRead$ for the next M-step (with bit-counter and current-tape both reset to 0); $q = t_tm\ M$ routes to the canonical accept stage; $q = r_tm\ M$ routes to the canonical reject stage. The non-terminal arm preserves buf , the direction-vector field $dvec$ (residual from the just-completed M-step; no longer consulted), and the per-tape position-kind $posk$ (carries the head-region across M-steps so the next $AR_SimRead$'s LE-vs-proper dispatch sees the correct value). The two terminal arms instead reset buf / $dvec$ / $posk$ to the canonical halt values $(\lambda_ . bl_tm\ M$ / $\lambda_ . dir.N$ / $\lambda_ . AR_AtLE)$, so the target stage is exactly ar_accept_stage ($bl_tm\ M$) / ar_reject_stage ($bl_tm\ M$) and the handshake lands on M 's halt state $t_tm\ M'$ / $r_tm\ M'$. $Lang_mttm$ matches the full mt_state (tape and heads free), so this canonical landing is what makes M' acceptance / rejection detectable; the reset is semantically free (the residual fields are never read after halting). The direction vector is all N (no head moves at the handshake); cell vectors $a' = a$ (no writes), so LE-preservation passes trivially.

The output machine's full transition relation: union of the five per-substep relations, intersected with three global restrictions — δLE -backward (no transition forges $LE4$ out of a non- $LE4$ cell), δLE -forward (every transi-

tion reading $LE4$ on tape k rewrites $LE4$ back on the same tape and moves N or R), and ar_valid_stage -membership on both source and target stages. The two δLE filters together discharge the substrate's bidirectional δLE invariant uniformly — without per-arm $a\ tk \neq LE4$ preconditions on every arm that performs a write distinct from the read cell (proper-arm forward-write in particular). The per-arm $a\ tk \neq LE4$ constraints that do appear (in look-back step 1, back-walk, and advance L-step) reflect the simulation invariant rather than the bare δLE requirement: those arms move L on the current tape, which the substrate forbids from an $LE4$ cell regardless of write content.

definition $alphabet_reduce_delta ::$

$$\begin{aligned}
& ('q, 'a) mttm \\
& \Rightarrow (('q \times 'a\ ar_stage) \\
& \quad \times (nat \Rightarrow sym4) \\
& \quad \times ('q \times 'a\ ar_stage) \\
& \quad \times (nat \Rightarrow sym4) \\
& \quad \times (nat \Rightarrow dir))\ set\ \mathbf{where} \\
alphabet_reduce_delta\ M = & \\
& (ar_delta_read\ M \cup ar_delta_compute\ M \\
& \quad \cup ar_delta_write\ M \cup ar_delta_advance\ M \\
& \quad \cup ar_delta_next\ M) \\
& \cap \{(s, a, s', a', d). \\
& \quad \forall k. a' k = LE4 \longrightarrow a k = LE4\} \\
& \cap \{(s, a, s', a', d). \\
& \quad \forall k. a k = LE4 \longrightarrow a' k = LE4 \wedge d k \in \{dir.N, dir.R\}\} \\
& \cap \{(s, a, s', a', d). \\
& \quad ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ s) \\
& \quad \wedge ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ s')\} \\
& \cap \{(s, a, s', a', d). \\
& \quad \forall j \geq k_tm\ M. a j = BLANK4 \wedge a' j = BLANK4 \wedge d j = dir.N\} \\
& \cap \{(s, a, s', a', d). \\
& \quad ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)\ (snd\ s) \\
& \quad \wedge ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)\ (snd\ s')\}
\end{aligned}$$

Initial / halt stages used to populate the output machine's $s' / t' / r'$ components. All three share the same shape: substep-counter $i = 0$, current-tape 0 placeholder (per-tape phase fields are inactive in $AR_SimRead$'s initial entry and inactive in halt states), per-tape buf initialised to bl (any value in $\Gamma \cup \{bl\}$ would satisfy ar_valid_stage ; bl is the canonical placeholder), per-tape direction-vector $dir.N$ (irrelevant outside $AR_SimAdvance$), per-tape position-kind AR_AtLE (the initial head position is 0 ; for halt states the field is vestigial). The three stages differ only in their $ar_substep_idx$ tag.

definition $ar_init_stage ::$

$$\begin{aligned}
& 'a \Rightarrow 'a\ ar_stage\ \mathbf{where} \\
ar_init_stage\ bl = & \\
& (AR_SimRead, 0, 0, (\lambda_. bl), (\lambda_. dir.N), (\lambda_. AR_AtLE))
\end{aligned}$$

definition *ar_accept_stage* ::
 $'a \Rightarrow 'a \text{ ar_stage}$ **where**
 $\text{ar_accept_stage } bl =$
 $(AR_HaltAccept, 0, 0, (\lambda_ . bl), (\lambda_ . dir.N), (\lambda_ . AR_AtLE))$

definition *ar_reject_stage* ::
 $'a \Rightarrow 'a \text{ ar_stage}$ **where**
 $\text{ar_reject_stage } bl =$
 $(AR_HaltReject, 0, 0, (\lambda_ . bl), (\lambda_ . dir.N), (\lambda_ . AR_AtLE))$

Positivity of the block width: b is bounded below by 1 via the *max* on the right-hand side. Used below to discharge the $i < 2 \cdot b$ conjunct of *ar_valid_stage* at $i = 0$, and downstream by the per-substep step-count lemmas.

lemma *block_width_pos*: $1 \leq \text{block_width } \Gamma$
unfolding *block_width_def* **by** *simp*

The three stage constants used to populate s' , t' , and r' all satisfy *ar_valid_stage* Γ bl for any Γ and bl : the bit-counter is $0 < 2 \cdot b$ (since $b \geq 1$) and the buf field is constantly $bl \in \Gamma \cup \{\!\!| bl |\!\!\}$.

lemma *ar_valid_stage_init*: *ar_valid_stage* Γ bl (*ar_init_stage* bl)

proof –

have $(0 :: nat) < 2 * \text{block_width } \Gamma$ **using** *block_width_pos*[of Γ] **by** *linarith*
thus *?thesis*

unfolding *ar_valid_stage_def* *ar_init_stage_def* **by** *simp*

qed

lemma *ar_valid_stage_accept*: *ar_valid_stage* Γ bl (*ar_accept_stage* bl)

proof –

have $(0 :: nat) < 2 * \text{block_width } \Gamma$ **using** *block_width_pos*[of Γ] **by** *linarith*
thus *?thesis*

unfolding *ar_valid_stage_def* *ar_accept_stage_def* **by** *simp*

qed

lemma *ar_valid_stage_reject*: *ar_valid_stage* Γ bl (*ar_reject_stage* bl)

proof –

have $(0 :: nat) < 2 * \text{block_width } \Gamma$ **using** *block_width_pos*[of Γ] **by** *linarith*
thus *?thesis*

unfolding *ar_valid_stage_def* *ar_reject_stage_def* **by** *simp*

qed

The alphabet-reduction combinator. Input: *mttm* over $'a$, with tape alphabet Γ_tm M a finite subset of $'a$ (via *valid_mttm*) of cardinality ≥ 4 . Output: *mttm* over *sym4*, with state set $'q \times 'a \text{ ar_stage}$. Tape count k_tm M is preserved.

Output components:

- $Q' = Q_M \times \{\!\!| stg. \text{ ar_valid_stage } \Gamma_M bl_M stg |\!\!\}$ (the non-product Q-shape);

- $\Sigma' = \{\text{BIT0}, \text{BIT1}\}$ (the encoded input alphabet);
- $\Gamma' = \text{UNIV} :: \text{sym4 set}$ (all four cell shapes — BIT0, BIT1, BLANK4, LE4);
- $\text{bl}' = \text{BLANK4}, \text{le}' = \text{LE4}$;
- $\delta' = \text{alphabet_reduce_delta } M$ (the five-substep union under the global LE-preservation and ar_valid_stage filters);
- $s' = (s_M, \text{ar_init_stage } \text{bl_}M), t' = (t_M, \text{ar_accept_stage } \text{bl_}M), r' = (r_M, \text{ar_reject_stage } \text{bl_}M)$ (M's start / accept / reject states paired with the matching ar_substep_idx tag).

definition $\text{alphabet_reduce} ::$

$(q, a) \text{ mttm} \Rightarrow (q \times a \text{ ar_stage}, \text{sym4}) \text{ mttm}$

where

$\text{alphabet_reduce } M =$
 $(\text{case } M \text{ of } \text{MTTM } Q_M _ \Gamma_M \text{ bl_}M _ _ s_M t_M r_M k_M \Rightarrow$
 $\text{MTTM } (Q_M \times \{\text{stg. ar_valid_stage } \Gamma_M \text{ bl_}M \text{ stg}$
 $\wedge \text{ar_stage_bounded } \text{bl_}M k_M \text{ stg}\})$
 $\{\text{BIT0}, \text{BIT1}\}$
 $(\text{UNIV} :: \text{sym4 set})$
 BLANK4
 LE4
 $(\text{alphabet_reduce_delta } M)$
 $(s_M, \text{ar_init_stage } \text{bl_}M)$
 $(t_M, \text{ar_accept_stage } \text{bl_}M)$
 $(r_M, \text{ar_reject_stage } \text{bl_}M)$
 $k_M)$

Projection-simp lemmas for the reduced machine: each structural accessor reads straight off the MTTM the combinator builds. Marked $[\text{simp}]$ so the language and time proofs never re-derive them via $\text{cases } M$; the accept-state projection in particular is the bridge that makes the simulation engine's terminal state syntactically equal to $t_tm (\text{alphabet_reduce } M)$.

lemma $\text{alphabet_reduce_Sigma } [\text{simp}]$:

$\text{Sigma_tm } (\text{alphabet_reduce } M) = \{\text{BIT0}, \text{BIT1}\}$

by $(\text{cases } M) (\text{simp add: alphabet_reduce_def})$

lemma $\text{alphabet_reduce_Gamma } [\text{simp}]$:

$\Gamma_tm (\text{alphabet_reduce } M) = (\text{UNIV} :: \text{sym4 set})$

by $(\text{cases } M) (\text{simp add: alphabet_reduce_def})$

lemma $\text{alphabet_reduce_bl } [\text{simp}]$:

$\text{bl_tm } (\text{alphabet_reduce } M) = \text{BLANK4}$

by $(\text{cases } M) (\text{simp add: alphabet_reduce_def})$


```

lemma alphabet_reduce_le [simp]:
  le_tm (alphabet_reduce M) = LE4
  by (cases M) (simp add: alphabet_reduce_def)

lemma alphabet_reduce_start [simp]:
  s_tm (alphabet_reduce M) = (s_tm M, ar_init_stage (bl_tm M))
  by (cases M) (simp add: alphabet_reduce_def)

lemma alphabet_reduce_accept [simp]:
  t_tm (alphabet_reduce M) = (t_tm M, ar_accept_stage (bl_tm M))
  by (cases M) (simp add: alphabet_reduce_def)

lemma alphabet_reduce_reject [simp]:
  r_tm (alphabet_reduce M) = (r_tm M, ar_reject_stage (bl_tm M))
  by (cases M) (simp add: alphabet_reduce_def)

lemma alphabet_reduce_delta [simp]:
  delta_tm (alphabet_reduce M) = alphabet_reduce_delta M
  by (cases M) (simp add: alphabet_reduce_def)

end
theory AlphabetReduction_Determinism
  imports AlphabetReduction_Delta
begin

```

2.10 Determinism preservation

The alphabet-reduction combinator preserves determinism: if M is deterministic then so is $\text{alphabet_reduce } M$. The language, time, and well-formedness theorems assume only well-formedness, so they already cover nondeterministic M ; the result below extends the reduction *up* to deterministic machines, turning the determinism-agnostic construction into a determinism-preserving one.

The argument is structural. $\text{alphabet_reduce_delta } M$ is the union of the five phase builders ar_delta_read , ar_delta_compute , ar_delta_write , ar_delta_advance , ar_delta_next , intersected with side conditions. Each builder is single-valued: among the five, only ar_delta_compute consults M 's own transition relation, so it is the sole place M -determinism enters; the other four are single-valued by pure case analysis on the bit-counter, the tape index, and the read cell (ar_delta_next additionally needs $t \neq r$, which valid_mttm supplies). The five builders are keyed to disjoint source substep tags, so two transitions sharing a source land in the same builder; and intersection with the side conditions only shrinks the relation, so single-valuedness of the union transfers to $\text{alphabet_reduce_delta } M$ and hence to the reduced machine.

Each per-builder lemma below splits the *source* tuple into its arms with

elim UnE and closes every arm against the full target builder: with the source fixed the arm discriminants are pinned, so the target collapses to one matching arm. The *block_width_pos* fact ($1 \leq b$) rules out the degenerate $b = 0$ aliasing where the per-bit arm boundaries would coincide. Throughout, b abbreviates *block_width* Γ (the per-symbol cell width).

lemma *ar_delta_read_functional*:
assumes $h1: (s, a, s1, a1, d1) \in \text{ar_delta_read } M$
and $h2: (s, a, s2, a2, d2) \in \text{ar_delta_read } M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using $h1$ **unfolding** *ar_delta_read_def*
by (*elim UnE*;
 use h2 block_width_pos[of Γ tm M] **in** $\langle \text{auto simp: ar_delta_read_def} \rangle$)

lemma *ar_delta_write_functional*:
assumes $h1: (s, a, s1, a1, d1) \in \text{ar_delta_write } M$
and $h2: (s, a, s2, a2, d2) \in \text{ar_delta_write } M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using $h1$ **unfolding** *ar_delta_write_def*
by (*elim UnE*;
 use h2 block_width_pos[of Γ tm M] **in** $\langle \text{auto simp: ar_delta_write_def} \rangle$)

lemma *ar_delta_advance_functional*:
assumes $h1: (s, a, s1, a1, d1) \in \text{ar_delta_advance } M$
and $h2: (s, a, s2, a2, d2) \in \text{ar_delta_advance } M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using $h1$ **unfolding** *ar_delta_advance_def*
by (*elim UnE*;
 use h2 block_width_pos[of Γ tm M] **in** $\langle \text{auto simp: ar_delta_advance_def} \rangle$)

lemma *ar_delta_next_functional*:
assumes $tr: t_tm\ M \neq r_tm\ M$
and $h1: (s, a, s1, a1, d1) \in \text{ar_delta_next } M$
and $h2: (s, a, s2, a2, d2) \in \text{ar_delta_next } M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using $h1$ **unfolding** *ar_delta_next_def*
by (*elim UnE*; *use h2 tr* **in** $\langle \text{auto simp: ar_delta_next_def} \rangle$)

The compute substep is the one place M 's transition relation enters *alphabet_reduce_delta*: a single δ' -tuple per M - δ -tuple, matching the source-stage decoded symbol vector *buf*. Single-valuedness here is exactly M -determinism transported through that match.

lemma *ar_delta_compute_functional*:
fixes $M :: ('q, 'a)\ \text{mttm}$
assumes $det: \text{det_mttm } M$
and $h1: (s, a, s1, a1, d1) \in \text{ar_delta_compute } M$
and $h2: (s, a, s2, a2, d2) \in \text{ar_delta_compute } M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
proof –

```

from h1 obtain qa buf qa' m_aa' m_da tka ia dvecb poska where
  sa: s = (qa, AR_SimCompute, tka, ia, buf, dvecb, poska)
and s1a: s1 = (qa', AR_SimWrite, 0, 0, m_aa', m_da, poska)
and a1a: a1 = a
and d1a: d1 = (λ_. dir.N)
and ma: (qa, buf, qa', m_aa', m_da) ∈ delta_tm M
unfolding ar_delta_compute_def by auto
from h2 obtain qb bufb qb' m_ab' m_db tkb ib dvecb poskb where
  sb: s = (qb, AR_SimCompute, tkb, ib, bufb, dvecb, poskb)
and s2b: s2 = (qb', AR_SimWrite, 0, 0, m_ab', m_db, poskb)
and a2b: a2 = a
and d2b: d2 = (λ_. dir.N)
and mb: (qb, bufb, qb', m_ab', m_db) ∈ delta_tm M
unfolding ar_delta_compute_def by auto
from sa sb have qeq: qb = qa and bufeq: bufb = buf
and poskeq: poskb = poska
by simp_all
have mb': (qa, buf, qb', m_ab', m_db) ∈ delta_tm M
using mb qeq bufeq by simp
have (qa', m_aa', m_da) = (qb', m_ab', m_db)
using ma mb' det unfolding det_mttm_def by blast
hence qe: qa' = qb' and mae: m_aa' = m_ab' and mde: m_da = m_db
by simp_all
show ?thesis
using s1a s2b a1a a2b d1a d2b poskeq qe mae mde by simp
qed

```

Dispatch on the source substep tag: the five builders have pairwise-disjoint source tags, so two transitions out of a common source s are governed by the same builder, and per-builder single-valuedness applies. The two halt tags carry no transition.

```

lemma alphabet_reduce_delta_functional:
  fixes M :: ('q, 'a) mttm
  assumes valM: valid_mttm M
  and detM: det_mttm M
  and h1: (s, a, s1, a1, d1) ∈ alphabet_reduce_delta M
  and h2: (s, a, s2, a2, d2) ∈ alphabet_reduce_delta M
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
proof –
  have tr: t_tm M ≠ r_tm M by (rule valid_mttm_t_neq_r[OF valM])
from h1 have u1: (s, a, s1, a1, d1) ∈
    ar_delta_read M ∪ ar_delta_compute M ∪ ar_delta_write M
    ∪ ar_delta_advance M ∪ ar_delta_next M
unfolding alphabet_reduce_delta_def by blast
from h2 have u2: (s, a, s2, a2, d2) ∈
    ar_delta_read M ∪ ar_delta_compute M ∪ ar_delta_write M
    ∪ ar_delta_advance M ∪ ar_delta_next M
unfolding alphabet_reduce_delta_def by blast
show ?thesis

```

```

proof (cases fst (snd s))
  case AR_SimRead
  have e1: (s, a, s1, a1, d1) ∈ ar_delta_read M
    using u1 AR_SimRead
    by (auto simp: ar_delta_compute_def ar_delta_write_def
      ar_delta_advance_def ar_delta_next_def)
  have e2: (s, a, s2, a2, d2) ∈ ar_delta_read M
    using u2 AR_SimRead
    by (auto simp: ar_delta_compute_def ar_delta_write_def
      ar_delta_advance_def ar_delta_next_def)
  show ?thesis by (rule ar_delta_read_functional[OF e1 e2])
next
  case AR_SimCompute
  have e1: (s, a, s1, a1, d1) ∈ ar_delta_compute M
    using u1 AR_SimCompute
    by (auto simp: ar_delta_read_def ar_delta_write_def
      ar_delta_advance_def ar_delta_next_def)
  have e2: (s, a, s2, a2, d2) ∈ ar_delta_compute M
    using u2 AR_SimCompute
    by (auto simp: ar_delta_read_def ar_delta_write_def
      ar_delta_advance_def ar_delta_next_def)
  show ?thesis by (rule ar_delta_compute_functional[OF detM e1 e2])
next
  case AR_SimWrite
  have e1: (s, a, s1, a1, d1) ∈ ar_delta_write M
    using u1 AR_SimWrite
    by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_advance_def ar_delta_next_def)
  have e2: (s, a, s2, a2, d2) ∈ ar_delta_write M
    using u2 AR_SimWrite
    by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_advance_def ar_delta_next_def)
  show ?thesis by (rule ar_delta_write_functional[OF e1 e2])
next
  case AR_SimAdvance
  have e1: (s, a, s1, a1, d1) ∈ ar_delta_advance M
    using u1 AR_SimAdvance
    by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_write_def ar_delta_next_def)
  have e2: (s, a, s2, a2, d2) ∈ ar_delta_advance M
    using u2 AR_SimAdvance
    by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_write_def ar_delta_next_def)
  show ?thesis by (rule ar_delta_advance_functional[OF e1 e2])
next
  case AR_SimNext
  have e1: (s, a, s1, a1, d1) ∈ ar_delta_next M
    using u1 AR_SimNext
    by (auto simp: ar_delta_read_def ar_delta_compute_def

```

```

      ar_delta_write_def ar_delta_advance_def)
have e2: (s, a, s2, a2, d2) ∈ ar_delta_next M
  using u2 AR_SimNext
  by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_write_def ar_delta_advance_def)
show ?thesis by (rule ar_delta_next_functional[OF tr e1 e2])
next
case AR_HaltAccept
with u1 show ?thesis
  by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_write_def
      ar_delta_advance_def ar_delta_next_def)
next
case AR_HaltReject
with u1 show ?thesis
  by (auto simp: ar_delta_read_def ar_delta_compute_def
      ar_delta_write_def
      ar_delta_advance_def ar_delta_next_def)
qed
qed

```

Headline result: the alphabet-reduction combinator carries determinism of M over to $\text{alphabet_reduce } M$. With the determinism-agnostic language / time / output theorems this completes the picture for deterministic machines.

```

theorem alphabet_reduce_det:
  fixes M :: ('q, 'a) mttm
  assumes valM: valid_mttm M
    and detM: det_mttm M
  shows det_mttm (alphabet_reduce M :: ('q × 'a ar_stage, sym4) mttm)
  unfolding det_mttm_def
proof (intro allI impI)
  fix q a p1 b1 dd1 p2 b2 dd2
  assume (q, a, p1, b1, dd1) ∈ delta_tm (alphabet_reduce M)
    and (q, a, p2, b2, dd2) ∈ delta_tm (alphabet_reduce M)
  hence m1: (q, a, p1, b1, dd1) ∈ alphabet_reduce_delta M
    and m2: (q, a, p2, b2, dd2) ∈ alphabet_reduce_delta M
    by simp_all
  from alphabet_reduce_delta_functional[OF valM detM m1 m2]
  show (p1, b1, dd1) = (p2, b2, dd2) by simp
qed

end
theory AlphabetReduction_Simulation
  imports AlphabetReduction_Determinism
begin

```

2.11 Simulation correspondence

Position correspondence $\text{sim_pos } b$, the AR analogue of AE's ae_decode_pos read forwards (M-position to M'-position). M -position 0 (the mandatory LE cell) maps to M' -position 0 ; M -position $p \geq 1$ maps to the start of its b -cell block at $(p - 1) \cdot b + 1$. The 1 -cell LE / b -cell proper layout is intrinsic to the substrate's mandatory single LE cell at position 0 . Throughout, b abbreviates $\text{block_width } \Gamma$ (the per-symbol cell width).

definition $\text{sim_pos} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**

$\text{sim_pos } k \ p = (\text{if } p = 0 \text{ then } 0 \text{ else } (p - 1) * k + 1)$

The proper-region tiling fact: the encoded cells are b -wide, aligned, and disjoint, so a position $\text{sim_pos } b \ p + j$ ($j < b$) lands in another cell's block $[\text{sim_pos } b \ q, \text{sim_pos } b \ q + b)$ exactly when $p = q$. This is the bridge the forward-step tape re-establishment (tcorr_1) and the advance back-walk (notLE) both turn on: a write at M -position q touches the p -block iff $p = q$. Proved by the alignment argument — a block index off by one shifts the position by a full b , past the $j < b$ offset.

lemma $\text{sim_pos_in_block_iff}$:

fixes $K \ p \ q \ j :: \text{nat}$

assumes $K1: 1 \leq K$ **and** $p1: 1 \leq p$ **and** $q1: 1 \leq q$ **and** $jK: j < K$

shows $(\text{sim_pos } K \ q \leq \text{sim_pos } K \ p + j \wedge \text{sim_pos } K \ p + j < \text{sim_pos } K \ q + K)$

$\longleftrightarrow p = q$

proof

assume $p = q$

thus $\text{sim_pos } K \ q \leq \text{sim_pos } K \ p + j \wedge \text{sim_pos } K \ p + j < \text{sim_pos } K \ q + K$
using jK **by** ($\text{simp add: sim_pos_def}$)

next

assume $L: \text{sim_pos } K \ q \leq \text{sim_pos } K \ p + j$
 $\wedge \text{sim_pos } K \ p + j < \text{sim_pos } K \ q + K$

have $lo: (q - 1) * K \leq (p - 1) * K + j$

and $hi: (p - 1) * K + j < (q - 1) * K + K$

using $L \ p1 \ q1$ **by** ($\text{auto simp: sim_pos_def}$)

have $p - 1 = q - 1$

proof (rule ccontr)

assume $p - 1 \neq q - 1$

then consider $p - 1 < q - 1 \mid q - 1 < p - 1$ **by** linarith

thus False

proof cases

case 1

hence $(p - 1) + 1 \leq q - 1$ **by** simp

hence $((p - 1) + 1) * K \leq (q - 1) * K$ **by** ($\text{rule mult_le_mono1}$)

hence $(p - 1) * K + K \leq (q - 1) * K$ **by** ($\text{simp add: algebra_sims}$)

thus False **using** $lo \ jK$ **by** linarith

next

case 2

hence $(q - 1) + 1 \leq p - 1$ **by** simp

```

    hence  $((q - 1) + 1) * K \leq (p - 1) * K$  by (rule mult_le_mono1)
    hence  $(q - 1) * K + K \leq (p - 1) * K$  by (simp add: algebra_simps)
    thus False using hi by linarith
  qed
qed
thus  $p = q$  using p1 q1 by linarith
qed

```

Per-cell encoding image $cell_repr \ \Gamma \ bl \ x$: the b -cell $sym4$ block that a single source cell of value x occupies on M' 's tape (at proper positions $p \geq 1$). The blank bl maps to b consecutive $BLANK4$ cells; every other source symbol maps to its $encode_symbol$ bit-block. Both branches have length b , so the block is uniformly b -wide. The endmarker le is not a case here: le occurs only at position 0 (a 1-cell $LE4$), handled directly in $ar_tape_correspondence$.

definition $cell_repr :: 'a \text{ set} \Rightarrow 'a \Rightarrow 'a \Rightarrow sym4 \text{ list}$ **where**

```

  cell_repr  $\Gamma \ bl \ x =$ 
    (if  $x = bl$  then replicate (block_width  $\Gamma$ )  $BLANK4$ 
     else encode_symbol  $\Gamma \ bl \ x$ )

```

Read-phase decode correctness for a whole cell block: folding the accumulator from $gamma_unenum \ 0$ over $cell_repr \ \Gamma \ bl \ x$ recovers x , uniformly across the blank and proper branches. The blank branch is $foldl_ar_acc_blank$ (seed rewritten to bl via $gamma_unenum_zero$); the proper branch is $foldl_ar_acc_encode_symbol$. This is the fact the single-tape read composition discharges its $buf \ tk$ -correctness against.

lemma $foldl_ar_acc_cell_repr$:

```

  assumes finite  $\Gamma$  and  $bl \in \Gamma$  and  $x \in \Gamma$ 
  shows foldl (ar_acc  $\Gamma \ bl$ ) (gamma_unenum  $\Gamma \ bl \ 0$ )
    (cell_repr  $\Gamma \ bl \ x$ ) =  $x$ 

```

proof (cases $x = bl$)

case True

```

  have foldl (ar_acc  $\Gamma \ bl$ ) (gamma_unenum  $\Gamma \ bl \ 0$ ) (cell_repr  $\Gamma \ bl \ x$ )
    = foldl (ar_acc  $\Gamma \ bl$ )  $bl$  (replicate (block_width  $\Gamma$ )  $BLANK4$ )
    using True gamma_unenum_zero[OF assms(1,2)] by (simp add:

```

$cell_repr_def$)

```

  also have ... =  $bl$  by (rule foldl_ar_acc_blank[OF assms(1,2)])

```

```

  finally show ?thesis using True by simp

```

next

case False

```

  have foldl (ar_acc  $\Gamma \ bl$ ) (gamma_unenum  $\Gamma \ bl \ 0$ ) (cell_repr  $\Gamma \ bl \ x$ )
    = foldl (ar_acc  $\Gamma \ bl$ ) (gamma_unenum  $\Gamma \ bl \ 0$ )
      (encode_symbol  $\Gamma \ bl \ x$ )

```

```

  using False by (simp add: cell_repr_def)

```

```

  also have ... =  $x$  by (rule foldl_ar_acc_encode_symbol[OF assms(1,3)])

```

```

  finally show ?thesis .

```

qed

Tape-content correspondence under the encoding, the AR analogue of AE's $ae_tape_correspondence$. M 's tape cell 0 holds le and M' 's cell 0

holds LE_4 ; for every proper position $p \geq 1$, the b -cell block on M' starting at $sim_pos\ b\ p$ spells out $cell_repr\ \Gamma\ bl\ (tM\ p)$. Since M' 's alphabet is the whole of sym_4 , no separate gamma-block invariant is carried (AE's $ae_tape_in_gamma_block$ would be vacuous here): the correspondence pins every M' -cell.

definition $ar_tape_correspondence ::$

$'a\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow (nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow sym_4) \Rightarrow bool$ **where**
 $ar_tape_correspondence\ \Gamma\ le\ bl\ tM\ tM' \longleftrightarrow$
 $tM\ 0 = le \wedge tM'\ 0 = LE_4 \wedge$
 $(\forall p. 1 \leq p \longrightarrow$
 $(\forall j. j < block_width\ \Gamma \longrightarrow$
 $tM'\ (sim_pos\ (block_width\ \Gamma)\ p + j) = cell_repr\ \Gamma\ bl\ (tM\ p) ! j))$

The simulation relation, stage-granular, the AR analogue of AE's $ae_simulates$. Holds at $AR_SimRead$ boundaries (M-step start, M-state not halted) or at the two halt configurations between an M -configuration cM and an M' -configuration cM' :

- the $ar_substep_idx$ tag is $AR_SimRead$ with the embedded M-state not yet in $\llbracket t, r \rrbracket$, or the M-state is t / r and the stage is the matching $ar_accept_stage / ar_reject_stage$ (AR's dedicated halt stages, unlike AE's parked $init_stage$);
- the embedded M-state matches M 's state;
- every tape corresponds cell-for-cell via $ar_tape_correspondence$;
- at an $AR_SimRead$ boundary, every head sits at $sim_pos\ b$ of M 's head.

definition $ar_simulates ::$

$('q, 'a)\ mttm$
 $\Rightarrow ('a, 'q)\ mt_config$
 $\Rightarrow (sym_4, 'q \times 'a\ ar_stage)\ mt_config$
 $\Rightarrow bool$ **where**
 $ar_simulates\ M\ cM\ cM' \longleftrightarrow$
 $(let\ qM = mt_state\ cM; tsM = mt_tape\ cM; nM = mt_pos\ cM;$
 $full = mt_state\ cM';$
 $tsM' = mt_tape\ cM'; nM' = mt_pos\ cM'\ in$
 $(case\ full\ of\ (qM',\ stg) \Rightarrow$
 $(case\ stg\ of\ (idx, _, _, _, _, _) \Rightarrow$
 $((idx = AR_SimRead \wedge qM' \notin \{t_tm\ M, r_tm\ M\})$
 $\vee (qM' = t_tm\ M \wedge stg = ar_accept_stage\ (bl_tm\ M))$
 $\vee (qM' = r_tm\ M \wedge stg = ar_reject_stage\ (bl_tm\ M)))$
 $\wedge qM = qM'$
 $\wedge (\forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)$
 $(bl_tm\ M)$
 $(tsM\ k)\ (tsM'\ k)))$

$$\wedge (idx = AR_SimRead \rightarrow (\forall k. nM' k = sim_pos (block_width (\Gamma_tm M)) (nM k))))$$

Companion invariant carried alongside *ar_simulates* (the AR analogue of AE's *ae_buffer_in_gamma_block*): at an *AR_SimRead* boundary the per-tape position-kind flag *posk* agrees with *M*'s head being on the left-end marker, $posk\ k = AR_AtLE \longleftrightarrow nM\ k = 0$. This is the bit the read's LE-vs-proper dispatch consumes; the *AR_AtFirstProper*/*AR_AtFurtherProper* split is deliberately not pinned (the read re-derives it via look-back). Kept out of *ar_simulates* proper so the reverse-direction language proof, which shares *ar_simulates*, carries no *posk* reasoning. Vacuous off the *AR_SimRead* boundary (terminal accept/reject configs), where the flag is never read again. Sound by the look-back re-sync argument: established at the initial config (all heads at 0, all flags *AR_AtLE*) and preserved each *M*-step by *ar_simulates_forward_step*.

definition *ar_posk_consistent* ::

$$\begin{aligned} &('q, 'a) mttm \\ &\Rightarrow ('a, 'q) mt_config \\ &\Rightarrow (sym4, 'q \times 'a\ ar_stage) mt_config \Rightarrow bool \text{ where} \\ &ar_posk_consistent\ M\ cM\ cM' \longleftrightarrow \\ &\quad (case\ mt_state\ cM'\ of\ (qM',\ stg) \Rightarrow \\ &\quad\ (case\ stg\ of\ (idx, _, _, _, _,\ posk) \Rightarrow \\ &\quad\ idx = AR_SimRead \\ &\quad\ \longrightarrow (\forall k. posk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0))) \end{aligned}$$

Second companion invariant, the boundary-shape companion (the buffer-in- Γ companion, plus the canonical-position pin). *ar_simulates*'s *AR_SimRead* arm pins only $idx = AR_SimRead$ — not the current-tape / bit-counter fields, nor that *buf* is a valid alphabet vector — but the read phase (*ar_read_phase*) starts a fresh per-*M*-step scan at tape 0, counter 0, and *ar_valid_stage* requires *buf* in $\Gamma \cup \{bl\}$. So this companion asserts, at an *AR_SimRead* boundary, the canonical entry shape $tk = 0, i = 0, \forall k. buf\ k \in \Gamma \cup \{bl\}$. Vacuous off the boundary (the halt configs), where the read phase never runs. Under value-level tape count it carries two further invariants the reverse walker's read-phase lift needs: the boundary stage is *ar_stage_bounded* (its *buf* / *dvec* / *posk* tails freeze beyond $k_tm\ M$), and the whole config is blank-tailed ($\forall j \geq k_tm\ M. mt_tape\ cM'\ j\ (mt_pos\ cM'\ j) = BLANK4$) — the *src0_b* / *pad0_b* facts the substrate's *alphabet_reduce_delta* support filter forces on every produced step. Established at the initial config (*ar_init_stage*: tape 0, counter 0, *buf* = $\lambda_. bl$) and preserved each *M*-step by the next handshake's non-terminal arm (resets tape / counter to 0; *buf* carries the just-written symbols, all in Γ). Kept separate from *ar_simulates* for the same reason as *ar_posk_consistent* — the reverse-direction language proof shares *ar_simulates* and should carry no forward-only boundary bookkeeping.

definition *ar_at_read_boundary* ::

```

('q, 'a) mttm
⇒ (sym4, 'q × 'a ar_stage) mt_config ⇒ bool where
ar_at_read_boundary M cM' ⇔
(case mt_state cM' of (qM', stg) ⇒
(case stg of (idx, tk, i, buf, dvec, posk) ⇒
idx = AR_SimRead
→ (tk = 0 ∧ i = 0
  ∧ (∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M})
  ∧ ar_stage_bounded (bl_tm M) (k_tm M)
  (idx, tk, i, buf, dvec, posk))))
  ∧ (∀ j ≥ k_tm M. mt_tape cM' j (mt_pos cM' j) = BLANK4)
end
theory AlphabetReduction_ForwardSubsteps
imports AlphabetReduction_Simulation
begin

```

2.12 Per-substep simulation steps

The compute substep: one M' -step from an $AR_SimCompute$ stage whose buf field matches an M - δ -tuple's read vector fires that tuple, landing at the $AR_SimWrite$ stage with M 's post-step state q' , write vector m_a' , and direction vector m_d threaded into the stage. No tape cell changes and no head moves (the substrate write is the read symbol back, direction N); the per-tape $posk$ carries through. The two global δLE filters are discharged reflexively (write equals read, move N); the target-stage validity rests on $valid_mttm$'s δ -range typing ($m_a' k \in \Gamma_tm M$). Throughout, b abbreviates $block_width \Gamma$ (the per-symbol cell width).

```

lemma ar_compute_step:
fixes M :: ('q, 'a) mttm
and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
and stg: mt_state c' = (q, AR_SimCompute, tk, i, buf, dvec, posk)
and mdelta: (q, buf, q', m_a', m_d) ∈ delta_tm M
and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimCompute, tk, i, buf, dvec, posk)
and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimCompute, tk, i, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
  ∧ mt_state c'' = (q', AR_SimWrite, 0, 0, m_a', m_d, posk)
  ∧ mt_tape c'' = mt_tape c'
  ∧ mt_pos c'' = mt_pos c'
proof –
obtain ts n where c'_eq:
  c' = Config_M (q, AR_SimCompute, tk, i, buf, dvec, posk) ts n
using stg by (cases c') auto
let ?a = λk. ts k (n k)

```

```

let ?s = (q, AR_SimCompute, tk, i, buf, dvec, posk)
let ?s' = (q', AR_SimWrite, 0, 0, m_a', m_d, posk)
have rel_in: (?s, ?a, ?s', ?a, (λ_. dir.N)) ∈ ar_delta_compute M
  unfolding ar_delta_compute_def using mdelta by auto
have ma'_gamma: ∀k. m_a' k ∈ Γ_tm M
  using valid_mttm_delta(4)[OF vM mdelta] by simp
have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
  using ma'_gamma block_width_pos[of Γ_tm M]
  by (auto simp: ar_valid_stage_def)
have pad_a: ∀j ≥ k_tm M. ?a j = BLANK4 using pad_blank c'_eq by simp
have dsupp: ∀j ≥ k_tm M. m_a' j = bl_tm M ∧ m_d j = dir.N
  using valid_mttm_delta_support[OF vM mdelta] by blast
have src_posk: ∀j ≥ k_tm M. posk j = AR_AtLE
  using src_bounded by (simp add: ar_stage_bounded_def)
have kpos_tm: 0 < k_tm M using vM by (cases M) auto
have pad_read: ∀j ≥ k_tm M. ?a j = BLANK4 ∧ ?a j = BLANK4
  ∧ (λ_. dir.N) j = dir.N
  using pad_a by simp
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  using dsupp src_posk kpos_tm by (simp add: ar_stage_bounded_def)
have ard_in: (?s, ?a, ?s', ?a, (λ_. dir.N))
  ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd by auto
have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
have pos_unchanged: (λk. go_dir ((λ_. dir.N) k) (n k)) = n
  by (rule ext) auto
let ?c'' = Config_M ?s' ts n
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a k))
  (λk. go_dir ((λ_. dir.N) k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, (λ_. dir.N))
    ∈ alphabet_reduce_delta M
  by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_unchanged by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

Displacement bound: an *AR_SimAdvance* walk on one tape is at most $2b$ cells (the *L*-from-*AR_AtFurtherProper* worst case). The $0 < b$ hypothesis is needed: the *N*-from-*AR_AtLE* displacement is the constant 1, which exceeds $2b = 0$. Used to discharge target-stage validity (*Suc* $i < 2 * b$) for the walk and boundary arms, whose reached bit-counter is bounded by the displacement.

lemma $ar_disp_le_2k$:
assumes $0 < k$
shows $ar_disp\ k\ d\ pk \leq 2 * k$
using *assms* **by** (*cases d*; *cases pk*) *auto*

The advance walk substep: from an $AR_SimAdvance$ stage with the bit-counter strictly below the per-tape displacement, one M' -step moves the active tape tk 's head one cell L (N elsewhere) and stays in $AR_SimAdvance$ at $Suc\ i$; the tape contents and all other heads are unchanged. The $a\ tk \neq LE4$ hypothesis is load-bearing: it both selects this arm (δ 's stepping arm forbids an L -walk off $LE4$) and discharges the backward- δLE filter, which would otherwise reject the L -move on a tape reading $LE4$. Target-stage validity rests on the displacement bound $ar_disp_le_2k$.

lemma $ar_advance_walk_step$:
fixes $M :: ('q, 'a)\ mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes vM : $valid_mttm\ M$
and stg : $mt_state\ c' = (q, AR_SimAdvance, tk, i, buf, dvec, posk)$
and qQ : $q \in Q_tm\ M$
and $notLE$: $mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) \neq LE4$
and $step_lt$: $Suc\ i < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
and $vsrc$: $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk)$
and pad_blank : $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src_bounded$: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step\ (alphabet_reduce_delta\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimAdvance, tk, Suc\ i, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c') (tk := mt_pos\ c'\ tk - 1)$

proof –

obtain $ts\ n$ **where** c'_eq :
 $c' = Config_M\ (q, AR_SimAdvance, tk, i, buf, dvec, posk)\ ts\ n$
using stg **by** (*cases c'*) *auto*
let $?a = \lambda k. ts\ k\ (n\ k)$
let $?d = \lambda kk. if\ kk = tk\ then\ dir.L\ else\ dir.N$
let $?s = (q, AR_SimAdvance, tk, i, buf, dvec, posk)$
let $?s' = (q, AR_SimAdvance, tk, Suc\ i, buf, dvec, posk)$
have aTk : $?a\ tk \neq LE4$ **using** $notLE\ c'_eq$ **by** *simp*
have rel_in : $(?s, ?a, ?s', ?a, ?d) \in ar_delta_advance\ M$
unfolding $ar_delta_advance_def$
using $qQ\ aTk\ step_lt$ **by** (*intro UnI1*) *blast*
have buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using $vsrc$ **by** (*simp add: ar_valid_stage_def*)
have $kpos$: $0 < block_width\ (\Gamma_tm\ M)$ **using** $block_width_pos[of\ \Gamma_tm\ M]$ **by** *simp*
have suc_i_lt : $Suc\ i < 2 * block_width\ (\Gamma_tm\ M)$
using $step_lt\ ar_disp_le_2k[OF\ kpos, of\ dvec\ tk\ posk\ tk]$ **by** *linarith*
have dst_valid : $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ ?s')$

```

    using suc_i_lt buf_valid by (simp add: ar_valid_stage_def)
    have pad_a:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4$  using pad_blank c'_eq by simp
    have tk_lt:  $tk < k\_tm\ M$  using src_bounded by (simp add:
ar_stage_bounded_def)
    have pad_read:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j =$ 
dir.N
  proof (intro allI impI)
    fix j assume jge:  $k\_tm\ M \leq j$ 
    have jne:  $j \neq tk$  using tk_lt jge by linarith
    show  $?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j = dir.N$ 
      using pad_a jge jne by simp
  qed
  have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
    using src_bounded by (simp add: ar_stage_bounded_def)
  have ard_in:  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$ 
    unfolding alphabet_reduce_delta_def
    using rel_in vsrc dst_valid aTk pad_read src_bounded dst_bd
    by (auto split: if_splits)
  have ts_unchanged:  $(\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$  by (rule ext) auto
  have pos_new:  $(\lambda k. go\_dir\ (?d\ k)\ (n\ k)) = n(tk := n\ tk - 1)$ 
    by (rule ext) simp
  let ?c'' = Config_M ?s' ts (n(tk := n tk - 1))
  have (Config_M ?s ts n,
    Config_M ?s'  $(\lambda k. (ts\ k)(n\ k := ?a\ k))$ 
     $(\lambda k. go\_dir\ (?d\ k)\ (n\ k))$ )
     $\in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
  proof (rule mttm_step.intros)
    show  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$  by (rule ard_in)
  qed
  hence step:  $(c', ?c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
    using c'_eq ts_unchanged pos_new by simp
  show ?thesis
    using step by (intro exI[where  $x = ?c''$ ]) (simp add: c'_eq)
  qed

```

The advance boundary substep, non-last tape: the final M' -step of tape tk 's walk, handing off to the next tape $k_succ\ tk$ (bit-counter reset to 0, position-kind updated by ar_newpos , other tapes' $posk$ preserved). A single hypothesis covers both firing sub-cases via the arm's own disjunction: the R -sub-case ($dvec\ tk = R$, zero displacement, no head move) and the non- R sub-case ($dvec\ tk \neq R$, the read not LE_4 , counter at $Suc\ i = displacement$, one last L -move). The head conclusion is therefore conditional on $dvec\ tk$.

lemma $ar_advance_boundary_step$:

```

fixes M :: ('q, 'a) mttm
and c' :: (sym4, 'q  $\times$  'a ar_stage) mt_config
assumes vM: valid_mttm M
and stg: mt_state c' = (q, AR_SimAdvance, tk, i, buf, dvec, posk)
and qQ:  $q \in Q\_tm\ M$ 
and notlast:  $\neg is\_last\_k\ M\ tk$ 

```

and fire: $(dvec\ tk = dir.R \wedge i = 0)$
 $\vee (dvec\ tk \neq dir.R$
 $\quad \wedge mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) \neq LE4$
 $\quad \wedge Suc\ i = ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk))$
and vsrc: $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk)$
and pad_blank: $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and src_bounded: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step\ (alphabet_reduce_delta\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimAdvance, k_succ\ tk, 0, buf, dvec,$
 $\quad posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (if\ dvec\ tk = dir.R\ then\ mt_pos\ c'$
 $\quad else\ (mt_pos\ c')(tk := mt_pos\ c'\ tk - 1))$
proof –
obtain $ts\ n$ **where** c'_eq :
 $c' = Config_M\ (q, AR_SimAdvance, tk, i, buf, dvec, posk)\ ts\ n$
using stg **by** $(cases\ c')\ auto$
let $?a = \lambda k. ts\ k\ (n\ k)$
let $?d = \lambda kk. if\ kk = tk \wedge dvec\ tk \neq dir.R\ then\ dir.L\ else\ dir.N$
let $?s = (q, AR_SimAdvance, tk, i, buf, dvec, posk)$
let $?s' = (q, AR_SimAdvance, k_succ\ tk, 0, buf, dvec,$
 $\quad posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
have $fire'$: $(dvec\ tk = dir.R \wedge i = 0)$
 $\vee (dvec\ tk \neq dir.R \wedge ?a\ tk \neq LE4$
 $\quad \wedge Suc\ i = ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk))$
using $fire\ c'_eq$ **by** $simp$
have $rel\ in$: $(?s, ?a, ?s', ?a, ?d) \in ar_delta_advance\ M$
unfolding $ar_delta_advance_def$
by $(rule\ UnI1, rule\ UnI2)\ (use\ qQ\ notlast\ fire'\ in\ blast)$
have buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using $vsrc$ **by** $(simp\ add: ar_valid_stage_def)$
have $kpos$: $0 < block_width\ (\Gamma_tm\ M)$ **using** $block_width_pos[of\ \Gamma_tm\ M]$ **by**
 $simp$
have dst_valid : $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ ?s')$
using $kpos\ buf_valid$ **by** $(simp\ add: ar_valid_stage_def)$
have pad_a : $\forall j \geq k_tm\ M. ?a\ j = BLANK4$ **using** $pad_blank\ c'_eq$ **by** $simp$
have tk_lt : $tk < k_tm\ M$ **using** $src_bounded$ **by** $(simp\ add:$
 $ar_stage_bounded_def)$
have $ksucc_lt$: $k_succ\ tk < k_tm\ M$
proof –
have ne : $Suc\ tk \neq k_tm\ M$ **using** $notlast$ **by** $(simp\ add: is_last_k_def)$
have le : $Suc\ tk \leq k_tm\ M$ **using** tk_lt **by** $simp$
show $?thesis$ **unfolding** k_succ_def **using** $le_neq_implies_less[OF\ le\ ne]$.
qed
have pad_read : $\forall j \geq k_tm\ M. ?a\ j = BLANK4 \wedge ?a\ j = BLANK4 \wedge ?d\ j =$
 $dir.N$
proof $(intro\ allI\ impI)$

```

    fix j assume jge: k_tm M ≤ j
    have jne: j ≠ tk using tk_lt jge by linarith
    show ?a j = BLANK₄ ∧ ?a j = BLANK₄ ∧ ?d j = dir.N
      using pad_a jge jne by simp
  qed
  have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  proof -
    have pk: ∀ j ≥ k_tm M. (posk(tk := ar_newpos (dvec tk) (posk tk))) j =
AR_AtLE
    proof (intro allI impI)
      fix j assume jge: k_tm M ≤ j
      have j ≠ tk using tk_lt jge by linarith
      thus (posk(tk := ar_newpos (dvec tk) (posk tk))) j = AR_AtLE
        using src_bounded jge by (simp add: ar_stage_bounded_def)
    qed
    show ?thesis using src_bounded ksucc_lt pk by (simp add:
ar_stage_bounded_def)
  qed
  have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
    unfolding alphabet_reduce_delta_def
    using rel_in vsrc dst_valid fire' pad_read src_bounded dst_bd
    by (auto split: if_splits)
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
  have pos_new: (λk. go_dir (?d k) (n k))
    = (if dvec tk = dir.R then n else n(tk := n tk - 1))
    by (rule ext) (auto split: if_splits)
  let ?c'' = Config_M ?s' ts (if dvec tk = dir.R then n else n(tk := n tk - 1))
  have (Config_M ?s ts n,
    Config_M ?s' (λk. (ts k)(n k := ?a k))
    (λk. go_dir (?d k) (n k)))
    ∈ mttm_step (alphabet_reduce_delta M)
  proof (rule mttm_step.intros)
    show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
  qed
  hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
    using c'_eq ts_unchanged pos_new by simp
  show ?thesis
    using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
  qed

```

The advance boundary substep, last tape: as *ar_advance_boundary_step* but *tk* is the last tape in the enumeration, so the hand-off goes to *AR_SimNext* (with the current-tape field reset to *k_unidx 0*) instead of advancing to *k_succ tk*. Same two firing sub-cases and the same conditional head conclusion; the arm-selection move is the single rule *UnI2* (the third, rightmost arm of *ar_delta_advance*).

lemma *ar_advance_finish_step*:
 fixes *M* :: ('q, 'a) mttm
 and *c'* :: (sym₄, 'q × 'a ar_stage) mt_config

```

assumes  $vM$ :  $\text{valid\_mttm } M$ 
and  $\text{stg}$ :  $\text{mt\_state } c' = (q, \text{AR\_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
and  $qQ$ :  $q \in Q\_tm\ M$ 
and  $\text{last}$ :  $\text{is\_last\_k } M\ tk$ 
and  $\text{fire}$ :  $(\text{dvec } tk = \text{dir}.R \wedge i = 0)$ 
 $\vee (\text{dvec } tk \neq \text{dir}.R$ 
 $\wedge \text{mt\_tape } c' tk (\text{mt\_pos } c' tk) \neq \text{LE4}$ 
 $\wedge \text{Suc } i = \text{ar\_disp } (\text{block\_width } (\Gamma\_tm\ M)) (\text{dvec } tk) (\text{posk } tk))$ 
and  $\text{vsrc}$ :  $\text{ar\_valid\_stage } (\Gamma\_tm\ M) (\text{bl\_tm } M)$ 
 $(\text{AR\_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
and  $\text{pad\_blank}$ :  $\forall j \geq k\_tm\ M. \text{mt\_tape } c' j (\text{mt\_pos } c' j) = \text{BLANK4}$ 
and  $\text{src\_bounded}$ :  $\text{ar\_stage\_bounded } (\text{bl\_tm } M) (k\_tm\ M)$ 
 $(\text{AR\_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
shows  $\exists c''. (c', c'') \in \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 
 $\wedge \text{mt\_state } c'' = (q, \text{AR\_SimNext}, k\_unidx\ 0, 0, \text{buf}, \text{dvec},$ 
 $\text{posk}(tk := \text{ar\_newpos } (\text{dvec } tk) (\text{posk } tk)))$ 
 $\wedge \text{mt\_tape } c'' = \text{mt\_tape } c'$ 
 $\wedge \text{mt\_pos } c'' = (\text{if } \text{dvec } tk = \text{dir}.R \text{ then } \text{mt\_pos } c'$ 
 $\text{else } (\text{mt\_pos } c')(tk := \text{mt\_pos } c' tk - 1))$ 

proof –
obtain  $ts\ n$  where  $c'_eq$ :
 $c' = \text{Config}_M (q, \text{AR\_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})\ ts\ n$ 
using  $\text{stg}$  by  $(\text{cases } c')\ \text{auto}$ 
let  $?a = \lambda k. ts\ k\ (n\ k)$ 
let  $?d = \lambda kk. \text{if } kk = tk \wedge \text{dvec } tk \neq \text{dir}.R \text{ then } \text{dir}.L \text{ else } \text{dir}.N$ 
let  $?s = (q, \text{AR\_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
let  $?s' = (q, \text{AR\_SimNext}, k\_unidx\ 0, 0, \text{buf}, \text{dvec},$ 
 $\text{posk}(tk := \text{ar\_newpos } (\text{dvec } tk) (\text{posk } tk)))$ 
have  $\text{fire}'$ :  $(\text{dvec } tk = \text{dir}.R \wedge i = 0)$ 
 $\vee (\text{dvec } tk \neq \text{dir}.R \wedge ?a\ tk \neq \text{LE4}$ 
 $\wedge \text{Suc } i = \text{ar\_disp } (\text{block\_width } (\Gamma\_tm\ M)) (\text{dvec } tk) (\text{posk } tk))$ 
using  $\text{fire } c'_eq$  by  $\text{simp}$ 
have  $\text{rel\_in}$ :  $(?s, ?a, ?s', ?a, ?d) \in \text{ar\_delta\_advance } M$ 
unfolding  $\text{ar\_delta\_advance\_def}$ 
by  $(\text{rule } \text{UnI2})\ (\text{use } qQ\ \text{last } \text{fire}'\ \text{in } \text{blast})$ 
have  $\text{buf\_valid}$ :  $\forall k. \text{buf } k \in \Gamma\_tm\ M \cup \{\text{bl\_tm } M\}$ 
using  $\text{vsrc}$  by  $(\text{simp add: ar\_valid\_stage\_def})$ 
have  $kpos$ :  $0 < \text{block\_width } (\Gamma\_tm\ M)$  using  $\text{block\_width\_pos}[of\ \Gamma\_tm\ M]$  by
 $\text{simp}$ 
have  $\text{dst\_valid}$ :  $\text{ar\_valid\_stage } (\Gamma\_tm\ M) (\text{bl\_tm } M) (\text{snd } ?s')$ 
using  $kpos\ \text{buf\_valid}$  by  $(\text{simp add: ar\_valid\_stage\_def})$ 
have  $\text{pad\_a}$ :  $\forall j \geq k\_tm\ M. ?a\ j = \text{BLANK4}$  using  $\text{pad\_blank } c'_eq$  by  $\text{simp}$ 
have  $tk\_lt$ :  $tk < k\_tm\ M$  using  $\text{src\_bounded}$  by  $(\text{simp add:}$ 
 $\text{ar\_stage\_bounded\_def})$ 
have  $kpos\_tm$ :  $0 < k\_tm\ M$  using  $vM$  by  $(\text{cases } M)\ \text{auto}$ 
have  $\text{pad\_read}$ :  $\forall j \geq k\_tm\ M. ?a\ j = \text{BLANK4} \wedge ?a\ j = \text{BLANK4} \wedge ?d\ j =$ 
 $\text{dir}.N$ 
proof  $(\text{intro allI impI})$ 
fix  $j$  assume  $jge$ :  $k\_tm\ M \leq j$ 

```



```

    have jne:  $j \neq tk$  using tk_lt jge by linarith
    show  $?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j = dir.N$ 
      using pad_a jge jne by simp
  qed
  have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  proof -
    have pk:  $\forall j \geq k\_tm\ M. (posk(tk := ar\_newpos\ (dvec\ tk)\ (posk\ tk)))\ j =$ 
    AR_AtLE
    proof (intro allI impI)
      fix j assume jge:  $k\_tm\ M \leq j$ 
      have  $j \neq tk$  using tk_lt jge by linarith
      thus  $(posk(tk := ar\_newpos\ (dvec\ tk)\ (posk\ tk)))\ j = AR\_AtLE$ 
        using src_bounded jge by (simp add: ar_stage_bounded_def)
    qed
    show ?thesis using src_bounded kpos_tm pk
      by (simp add: ar_stage_bounded_def k_unidx_def)
  qed
  have ard_in:  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$ 
    unfolding alphabet_reduce_delta_def
    using rel_in vsrc dst_valid fire' pad_read src_bounded dst_bd
    by (auto split: if_splits)
  have ts_unchanged:  $(\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$  by (rule ext) auto
  have pos_new:  $(\lambda k. go\_dir\ (?d\ k)\ (n\ k))$ 
    =  $(if\ dvec\ tk = dir.R\ then\ n\ else\ n(tk := n\ tk - 1))$ 
    by (rule ext) (auto split: if_splits)
  let ?c'' = Config_M ?s' ts (if dvec tk = dir.R then n else n(tk := n tk - 1))
  have (Config_M ?s ts n,
    Config_M ?s'  $(\lambda k. (ts\ k)(n\ k := ?a\ k))$ 
     $(\lambda k. go\_dir\ (?d\ k)\ (n\ k))$ )
    ∈ mttm_step (alphabet_reduce_delta M)
  proof (rule mttm_step.intros)
    show  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$  by (rule ard_in)
  qed
  hence step:  $(c', ?c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
    using c'_eq ts_unchanged pos_new by simp
  show ?thesis
    using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
  qed

```

The write LE-skip substep, non-last tape: when $buf\ tk$ is the left-end marker, the cell at $sim_pos\ 0 = 0$ is already LE_4 and need not be rewritten, so the per-tape write phase is skipped — a single all- N substep handing off to the next tape $k_succ\ tk$ with the bit-counter still 0 . Tape and heads unchanged (the compute-step shape).

```

lemma ar_write_le_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)

```

```

and qQ: q ∈ Q_tm M
and poskLE: posk tk = AR_AtLE
and notlast: ¬ is_last_k M tk
and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
            (AR_SimWrite, tk, 0, buf, dvec, posk)
and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
            (AR_SimWrite, tk, 0, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
            ∧ mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
            ∧ mt_tape c'' = mt_tape c'
            ∧ mt_pos c'' = mt_pos c'
proof -
  obtain ts n where c'_eq:
    c' = Config_M (q, AR_SimWrite, tk, 0, buf, dvec, posk) ts n
  using sty by (cases c') auto
  let ?a = λk. ts k (n k)
  let ?d = λ_. dir.N
  let ?s = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
  let ?s' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
  have rel_in: (?s, ?a, ?s', ?a, ?d) ∈ ar_delta_write M
    unfolding ar_delta_write_def
    using qQ poskLE notlast by (intro UnI1) blast
  have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
    using vsrc by (simp add: ar_valid_stage_def)
  have kpos: 0 < block_width (Γ_tm M) using block_width_pos[of Γ_tm M] by
simp
  have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
    using kpos buf_valid by (simp add: ar_valid_stage_def)
  have pad_a: ∀ j ≥ k_tm M. ?a j = BLANK4 using pad_blank c'_eq by simp
  have tk_lt: tk < k_tm M using src_bounded by (simp add:
ar_stage_bounded_def)
  have ksucc_lt: k_succ tk < k_tm M
  proof -
    have ne: Suc tk ≠ k_tm M using notlast by (simp add: is_last_k_def)
    have le: Suc tk ≤ k_tm M using tk_lt by simp
    show ?thesis unfolding k_succ_def using le_neq_implies_less[OF le ne] .
  qed
  have pad_read: ∀ j ≥ k_tm M. ?a j = BLANK4 ∧ ?a j = BLANK4 ∧ ?d j =
dir.N
  using pad_a by simp
  have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
    using src_bounded ksucc_lt by (simp add: ar_stage_bounded_def)
  have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
    unfolding alphabet_reduce_delta_def
    using rel_in vsrc dst_valid pad_read src_bounded dst_bd by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
  have pos_unchanged: (λk. go_dir (?d k) (n k)) = n by (rule ext) auto
  let ?c'' = Config_M ?s' ts n

```

```

have (ConfigM ?s ts n,
      ConfigM ?s' (λk. (ts k)(n k := ?a k))
      (λk. go_dir (?d k) (n k)))
      ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_unchanged by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The write LE-skip substep, last tape: as *ar_write_le_step* but *tk* is the last tape, so the hand-off goes to *AR_SimAdvance* (current-tape field reset to *k_unidx 0*). Arm 2 of the six-arm *ar_delta_write* union.

```

lemma ar_write_le_finish_step:
fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
  and qQ: q ∈ Q_tm M
  and poskLE: posk tk = AR_AtLE
  and last: is_last_k M tk
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimWrite, tk, 0, buf, dvec, posk)
  and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, tk, 0, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
  ∧ mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
  ∧ mt_tape c'' = mt_tape c'
  ∧ mt_pos c'' = mt_pos c'
proof –
obtain ts n where c'_eq:
  c' = ConfigM (q, AR_SimWrite, tk, 0, buf, dvec, posk) ts n
  using stg by (cases c') auto
let ?a = λk. ts k (n k)
let ?d = λ_. dir.N
let ?s = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
let ?s' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
have rel_in: (?s, ?a, ?s', ?a, ?d) ∈ ar_delta_write M
  unfolding ar_delta_write_def
  by (rule UnI1, rule UnI1, rule UnI1, rule UnI1, rule UnI2)
  (use qQ poskLE last in blast)
have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  using vsrc by (simp add: ar_valid_stage_def)
have kpos: 0 < block_width (Γ_tm M) using block_width_pos[of Γ_tm M] by
simp

```

```

have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
  using kpos buf_valid by (simp add: ar_valid_stage_def)
have pad_a: ∀ j ≥ k_tm M. ?a j = BLANK₄ using pad_blank c'_eq by simp
have kpos_tm: 0 < k_tm M using vM by (cases M) auto
have pad_read: ∀ j ≥ k_tm M. ?a j = BLANK₄ ∧ ?a j = BLANK₄ ∧ ?d j =
dir.N
  using pad_a by simp
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  using src_bounded kpos_tm by (simp add: ar_stage_bounded_def
k_unidx_def)
have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd by auto
have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
have pos_unchanged: (λk. go_dir (?d k) (n k)) = n by (rule ext) auto
let ?c'' = Config_M ?s' ts n
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a k))
  (λk. go_dir (?d k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_unchanged by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The write back-walk substep (proper region, $Suc\ i \leq b$): with $buf\ tk$ a proper symbol, the head first walks L back across the b -cell block before the forward-write phase; one L -move on tk (N elsewhere), no writes, staying in $AR_SimWrite$ at $Suc\ i$. Structurally the $ar_advance_walk_step$ shape; arm 3 of the union. As there, a $tk \neq LE_4$ selects the arm and discharges the backward- δLE filter.

```

lemma ar_write_walk_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym₄, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimWrite, tk, i, buf, dvec, posk)
  and qQ: q ∈ Q_tm M
  and poskproper: posk tk ≠ AR_AtLE
  and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE₄
  and step_le: Suc i ≤ block_width (Γ_tm M)
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimWrite, tk, i, buf, dvec, posk)
  and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK₄
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, tk, i, buf, dvec, posk)

```

shows $\exists c''. (c', c'') \in \text{mttm_step } (\text{alphabet_reduce_delta } M)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimWrite}, tk, \text{Suc } i, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c')(tk := \text{mt_pos } c' tk - 1)$

proof –

obtain $ts\ n$ **where** c'_eq :
 $c' = \text{Config}_M (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})\ ts\ n$
using stg **by** $(\text{cases } c')\ \text{auto}$
let $?a = \lambda k. ts\ k\ (n\ k)$
let $?d = \lambda kk. \text{if } kk = tk \text{ then } dir.L \text{ else } dir.N$
let $?s = (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
let $?s' = (q, \text{AR_SimWrite}, tk, \text{Suc } i, \text{buf}, \text{dvec}, \text{posk})$
have $aTk: ?a\ tk \neq LE_4$ **using** $\text{notLE } c'_eq$ **by** simp
have $\text{rel_in}: (?s, ?a, ?s', ?a, ?d) \in \text{ar_delta_write } M$
unfolding $\text{ar_delta_write_def}$
by $(\text{rule } UnI1, \text{rule } UnI1, \text{rule } UnI1, \text{rule } UnI2)$
 $(\text{use } qQ\ \text{poskproper } aTk\ \text{step_le in blast})$
have $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using $\text{vsrce by (simp add: ar_valid_stage_def)}$
have $kpos: 0 < \text{block_width } (\Gamma_tm\ M)$ **using** $\text{block_width_pos[of } \Gamma_tm\ M]$ **by**
 simp
have $\text{suc_i_lt}: \text{Suc } i < 2 * \text{block_width } (\Gamma_tm\ M)$
using $\text{step_le } kpos$ **by** linarith
have $\text{dst_valid}: \text{ar_valid_stage } (\Gamma_tm\ M)\ (bl_tm\ M)\ (\text{snd } ?s')$
using $\text{suc_i_lt } \text{buf_valid by (simp add: ar_valid_stage_def)}$
have $\text{pad_a}: \forall j \geq k_tm\ M. ?a\ j = BLANK_4$ **using** $\text{pad_blank } c'_eq$ **by** simp
have $tk\ lt: tk < k_tm\ M$ **using** $\text{src_bounded by (simp add: ar_stage_bounded_def)}$
have $\text{pad_read}: \forall j \geq k_tm\ M. ?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j = dir.N$

proof (intro allI impI)

fix j **assume** $jge: k_tm\ M \leq j$
have $jne: j \neq tk$ **using** $tk_lt\ jge$ **by** linarith
show $?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j = dir.N$
using $\text{pad_a } jge\ jne$ **by** simp

qed

have $\text{dst_bd}: \text{ar_stage_bounded } (bl_tm\ M)\ (k_tm\ M)\ (\text{snd } ?s')$
using $\text{src_bounded by (simp add: ar_stage_bounded_def)}$
have $\text{ard_in}: (?s, ?a, ?s', ?a, ?d) \in \text{alphabet_reduce_delta } M$
unfolding $\text{alphabet_reduce_delta_def}$
using $\text{rel_in vsrce dst_valid aTk pad_read src_bounded dst_bd}$
by $(\text{auto split: if_splits})$
have $\text{ts_unchanged}: (\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$ **by** $(\text{rule ext})\ \text{auto}$
have $\text{pos_new}: (\lambda k. \text{go_dir } (?d\ k)\ (n\ k)) = n(tk := n\ tk - 1)$
by $(\text{rule ext})\ \text{simp}$
let $?c'' = \text{Config}_M\ ?s'\ ts\ (n(tk := n\ tk - 1))$
have $(\text{Config}_M\ ?s\ ts\ n,$
 $\text{Config}_M\ ?s'\ (\lambda k. (ts\ k)(n\ k := ?a\ k))$
 $(\lambda k. \text{go_dir } (?d\ k)\ (n\ k)))$

```

      ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_new by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The forward-write image is never the left-end marker: it is *BLANK*₄ (blank symbol) or a bit cell of *encode_symbol* (*BIT0*/*BIT1* only, by *encode_symbol_cell_domain*), in range $j < b$. This is what makes the forward-write arms legal under both δLE filters.

```

lemma write_bit_not_LE4:
  assumes j < block_width Γ
  shows write_bit Γ bl x j ≠ LE4
proof (cases x = bl)
  case True
  thus ?thesis by (simp add: write_bit_def)
next
  case False
  have jl: j < length (encode_symbol Γ bl x)
    using assms by (simp add: encode_symbol_def)
  have encode_symbol Γ bl x ! j ∈ {BIT0, BIT1}
    using nth_mem[OF jl] encode_symbol_cell_domain[of Γ bl x] by blast
  thus ?thesis using False by (auto simp: write_bit_def)
qed

```

The j -th cell of a block's *cell_repr* is the j -th *write_bit* image ($j < b$), uniformly across the blank and proper branches. This is the bridge from the tape-correspondence cell value (*cell_repr* ... ! j) to the *write_bit* image that the write phase produces, and to the $\neq LE4$ fact the back-walk / bit-write guards need.

```

lemma cell_repr_nth_write_bit:
  assumes j < block_width Γ
  shows cell_repr Γ bl x ! j = write_bit Γ bl x j
  using assms by (cases x = bl)
    (simp_all add: cell_repr_def write_bit_def length_encode_symbol)

```

```

lemma cell_repr_nth_not_LE4:
  assumes j < block_width Γ
  shows cell_repr Γ bl x ! j ≠ LE4
  using cell_repr_nth_write_bit[OF assms] write_bit_not_LE4[OF assms] by
  simp

```

Under the encoding, the only *LE4* cell of the simulated tape is at position 0: every position ≥ 1 lies in some proper block [*sim_pos* b p , *sim_pos* b $p + b$) (the *div/mod* decomposition of *pos* - 1) and so carries a *cell_repr* cell,

which is never $LE4$. The fact the advance back-walk's $notLE$ guard rests on: a head walking through proper cells never reads the left-end marker.

lemma $ar_tape_correspondence_not_LE4$:
assumes $corr$: $ar_tape_correspondence \ \Gamma \ le \ bl \ tM \ tM'$
and $pos1$: $1 \leq pos$
shows $tM' \ pos \neq LE4$
proof –
let $?K = block_width \ \Gamma$
have $K1$: $1 \leq ?K$ **using** $block_width_pos$.
have jK : $(pos - 1) \bmod ?K < ?K$ **using** $K1$ **by** $simp$
have $p1$: $1 \leq (pos - 1) \div ?K + 1$ **by** $simp$
have pe : $pos = sim_pos \ ?K \ ((pos - 1) \div ?K + 1) + (pos - 1) \bmod ?K$
proof –
have $sim_pos \ ?K \ ((pos - 1) \div ?K + 1) + (pos - 1) \bmod ?K$
 $= (pos - 1) \div ?K * ?K + (pos - 1) \bmod ?K + 1$
by $(simp \ add: \ sim_pos_def)$
also have $\dots = (pos - 1) + 1$ **by** $(simp \ add: \ div_mult_mod_eq)$
also have $\dots = pos$ **using** $pos1$ **by** $simp$
finally show $?thesis$ **by** $simp$
qed
have $corrprop$: $tM' \ (sim_pos \ ?K \ p + j) = cell_repr \ \Gamma \ bl \ (tM \ p) \ ! \ j$
if $1 \leq p$ **and** $j < ?K$ **for** $p \ j$
using $corr$ **that** **unfolding** $ar_tape_correspondence_def$ **by** $blast$
have $tM' \ pos$
 $= cell_repr \ \Gamma \ bl \ (tM \ ((pos - 1) \div ?K + 1)) \ ! \ ((pos - 1) \bmod ?K)$
using $corrprop[OF \ p1 \ jK] \ pe$ **by** $simp$
thus $?thesis$ **using** $cell_repr_nth_not_LE4[OF \ jK]$ **by** $simp$
qed

The forward-write stepping substep (proper region, $b \leq i$, $Suc \ i < 2b$): the $(i-b)$ -th cell of $cell_repr \ (buf \ tk)$ is written at tk , the head moves R on tk (N elsewhere), staying in $AR_SimWrite$ at $Suc \ i$. First per-substep lemma that mutates the tape: the conclusion's mt_tape is a nested fun_upd writing $write_bit \ \Gamma \ bl \ (buf \ tk) \ (i-b)$ at tk 's head cell. The read precondition $a \ tk \neq LE4$ (proper region is past $LE4$) discharges the backward- δLE filter; $write_bit_not_LE4$ discharges the forward filter.

lemma $ar_write_bit_step$:
fixes $M :: ('q, 'a) \ mttm$
and $c' :: (sym4, 'q \times 'a \ ar_stage) \ mt_config$
assumes vM : $valid_mttm \ M$
and stg : $mt_state \ c' = (q, AR_SimWrite, tk, i, buf, dvec, posk)$
and qQ : $q \in Q_tm \ M$
and $poskproper$: $posk \ tk \neq AR_AtLE$
and $notLE$: $mt_tape \ c' \ tk \ (mt_pos \ c' \ tk) \neq LE4$
and ilo : $block_width \ (\Gamma_tm \ M) \leq i$
and ihi : $Suc \ i < 2 * block_width \ (\Gamma_tm \ M)$
and $vsrc$: $ar_valid_stage \ (\Gamma_tm \ M) \ (bl_tm \ M)$
 $(AR_SimWrite, tk, i, buf, dvec, posk)$

```

and pad_blank:  $\forall j \geq k\_tm\ M. mt\_tape\ c'\ j\ (mt\_pos\ c'\ j) = BLANK_4$ 
and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, tk, i, buf, dvec, posk)
shows  $\exists c''. (c', c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
   $\wedge mt\_state\ c'' = (q, AR\_SimWrite, tk, Suc\ i, buf, dvec, posk)$ 
   $\wedge mt\_tape\ c'' = (mt\_tape\ c')(tk :=$ 
     $(mt\_tape\ c'\ tk)(mt\_pos\ c'\ tk :=$ 
       $write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (buf\ tk)\ (i - block\_width\ (\Gamma\_tm\ M))))$ 
   $\wedge mt\_pos\ c'' = (mt\_pos\ c')(tk := Suc\ (mt\_pos\ c'\ tk))$ 
proof -
  obtain ts n where c'_eq:
    c' = Config_M (q, AR_SimWrite, tk, i, buf, dvec, posk) ts n
  using stg by (cases c') auto
  let ?wb = write_bit (Γ_tm M) (bl_tm M) (buf tk) (i - block_width (Γ_tm
M))
  let ?a = λk. ts k (n k)
  let ?a' = λkk. if kk = tk then ?wb else ?a kk
  let ?d = λkk. if kk = tk then dir.R else dir.N
  let ?s = (q, AR_SimWrite, tk, i, buf, dvec, posk)
  let ?s' = (q, AR_SimWrite, tk, Suc i, buf, dvec, posk)
  have aTk: ?a tk ≠ LE4 using notLE c'_eq by simp
  have jlt: i - block_width (Γ_tm M) < block_width (Γ_tm M) using ilo ihi by
linarith
  have wb_not_LE: ?wb ≠ LE4 using write_bit_not_LE4[OF jlt] .
  have rel_in: (?s, ?a, ?s', ?a', ?d) ∈ ar_delta_write M
    unfolding ar_delta_write_def
    by (rule UnI1, rule UnI1, rule UnI2)
    (use qQ poskproper ilo ihi in blast)
  have buf_valid:  $\forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
    using vsrc by (simp add: ar_valid_stage_def)
  have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
    using ihi buf_valid by (simp add: ar_valid_stage_def)
  have pad_a:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4$  using pad_blank c'_eq by simp
    have tk_lt: tk < k_tm M using src_bounded by (simp add:
ar_stage_bounded_def)
  have pad_read:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4 \wedge ?a'\ j = BLANK_4 \wedge ?d\ j =$ 
dir.N
  proof (intro allI impI)
    fix j assume jge: k_tm M ≤ j
    have jne: j ≠ tk using tk_lt jge by linarith
    show ?a j = BLANK4 ∧ ?a' j = BLANK4 ∧ ?d j = dir.N
      using pad_a jge jne by simp
  qed
  have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
    using src_bounded by (simp add: ar_stage_bounded_def)
  have ard_in: (?s, ?a, ?s', ?a', ?d) ∈ alphabet_reduce_delta M
    unfolding alphabet_reduce_delta_def
    using rel_in vsrc dst_valid aTk wb_not_LE pad_read src_bounded dst_bd

```



```

  by (auto split: if_splits)
have tape_new: (λk. (ts k)(n k := ?a' k))
  = ts(tk := (ts tk)(n tk := ?wb))
  by (rule ext) (auto simp: fun_upd_triv)
have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := Suc (n tk))
  by (rule ext) simp
let ?c'' = Config_M ?s' (ts(tk := (ts tk)(n tk := ?wb))) (n(tk := Suc (n tk)))
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a' k))
  (λk. go_dir (?d k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a', ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq tape_new pos_new by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The forward-write boundary substep, non-last tape ($Suc\ i = 2b$): the last cell of $cell_repr\ (buf\ tk)$ is written ($j = i - b = b - 1$), the head moves R , and the phase hands off to the next tape $k_succ\ tk$ with the bit-counter reset to 0. The $ar_write_bit_step$ tape-change shape; arm 5 of the union.

lemma $ar_write_bit_boundary_step$:

```

fixes M :: ('q, 'a) mttm
and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimWrite, tk, i, buf, dvec, posk)
  and qQ: q ∈ Q_tm M
  and poskproper: posk tk ≠ AR_AtLE
  and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
  and notlast: ¬ is_last_k M tk
  and ihi: Suc i = 2 * block_width (Γ_tm M)
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimWrite, tk, i, buf, dvec, posk)
  and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, tk, i, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
  ∧ mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
  ∧ mt_tape c'' = (mt_tape c')(tk :=
    (mt_tape c' tk)(mt_pos c' tk :=
      write_bit (Γ_tm M) (bl_tm M) (buf tk)
        (i - block_width (Γ_tm M))))
  ∧ mt_pos c'' = (mt_pos c')(tk := Suc (mt_pos c' tk))

```

proof –

```

obtain ts n where c'_eq:
  c' = Config_M (q, AR_SimWrite, tk, i, buf, dvec, posk) ts n

```

```

    using stg by (cases c') auto
    let ?wb = write_bit (Γ_tm M) (bl_tm M) (buf tk) (i - block_width (Γ_tm
M))
    let ?a = λk. ts k (n k)
    let ?a' = λkk. if kk = tk then ?wb else ?a kk
    let ?d = λkk. if kk = tk then dir.R else dir.N
    let ?s = (q, AR_SimWrite, tk, i, buf, dvec, posk)
    let ?s' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
    have aTk: ?a tk ≠ LE4 using notLE c'_eq by simp
    have kpos: 0 < block_width (Γ_tm M) using block_width_pos[of Γ_tm M] by
simp
    have jlt: i - block_width (Γ_tm M) < block_width (Γ_tm M) using ihi kpos
by linarith
    have wb_not_LE: ?wb ≠ LE4 using write_bit_not_LE4[OF jlt] .
    have rel_in: (?s, ?a, ?s', ?a', ?d) ∈ ar_delta_write M
      unfolding ar_delta_write_def
      by (rule UnI1, rule UnI2) (use qQ poskproper notlast ihi in blast)
    have buf_valid: ∀k. buf k ∈ Γ_tm M ∪ {bl_tm M}
      using vsrc by (simp add: ar_valid_stage_def)
    have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
      using kpos buf_valid by (simp add: ar_valid_stage_def)
    have pad_a: ∀j ≥ k_tm M. ?a j = BLANK4 using pad_blank c'_eq by simp
    have tk_lt: tk < k_tm M using src_bounded by (simp add:
ar_stage_bounded_def)
    have ksucc_lt: k_succ tk < k_tm M
    proof -
      have ne: Suc tk ≠ k_tm M using notlast by (simp add: is_last_k_def)
      have le: Suc tk ≤ k_tm M using tk_lt by simp
      show ?thesis unfolding k_succ_def using le_neq_implies_less[OF le ne] .
    qed
    have pad_read: ∀j ≥ k_tm M. ?a j = BLANK4 ∧ ?a' j = BLANK4 ∧ ?d j =
dir.N
    proof (intro allI impI)
      fix j assume jge: k_tm M ≤ j
      have jne: j ≠ tk using tk_lt jge by linarith
      show ?a j = BLANK4 ∧ ?a' j = BLANK4 ∧ ?d j = dir.N
        using pad_a jge jne by simp
    qed
    have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
      using src_bounded ksucc_lt by (simp add: ar_stage_bounded_def)
    have ard_in: (?s, ?a, ?s', ?a', ?d) ∈ alphabet_reduce_delta M
      unfolding alphabet_reduce_delta_def
      using rel_in vsrc dst_valid aTk wb_not_LE pad_read src_bounded dst_bd
      by (auto split: if_splits)
    have tape_new: (λk. (ts k)(n k := ?a' k))
      = ts(tk := (ts tk)(n tk := ?wb))
      by (rule ext) (auto simp: fun_upd_triv)
    have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := Suc (n tk))
      by (rule ext) simp

```

```

let ?c'' = ConfigM ?s' (ts(tk := (ts tk)(n tk := ?wb))) (n(tk := Suc (n tk)))
have (ConfigM ?s ts n,
      ConfigM ?s' (λk. (ts k)(n k := ?a' k))
      (λk. go_dir (?d k) (n k)))
      ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a', ?d) ∈ alphabet_reduce_delta M by (rule ar_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq tape_new pos_new by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The forward-write boundary substep, last tape: as *ar_write_bit_boundary_step* but *tk* is the last tape, so after the last-cell write the phase transitions to *AR_SimAdvance* (current-tape field reset to *k_unidx 0*). Arm 6 (rightmost) of the union.

lemma *ar_write_bit_finish_step*:

```

fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimWrite, tk, i, buf, dvec, posk)
  and qQ: q ∈ Q_tm M
  and poskproper: posk tk ≠ AR_AtLE
  and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
  and last: is_last_k M tk
  and ihi: Suc i = 2 * block_width (Γ_tm M)
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimWrite, tk, i, buf, dvec, posk)
  and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, tk, i, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
  ∧ mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
  ∧ mt_tape c'' = (mt_tape c')(tk :=
    (mt_tape c' tk)(mt_pos c' tk :=
      write_bit (Γ_tm M) (bl_tm M) (buf tk)
      (i - block_width (Γ_tm M))))
  ∧ mt_pos c'' = (mt_pos c')(tk := Suc (mt_pos c' tk))
proof –
  obtain ts n where c'_eq:
    c' = ConfigM (q, AR_SimWrite, tk, i, buf, dvec, posk) ts n
  using stg by (cases c') auto
  let ?wb = write_bit (Γ_tm M) (bl_tm M) (buf tk) (i - block_width (Γ_tm M))
  let ?a = λk. ts k (n k)
  let ?a' = λkk. if kk = tk then ?wb else ?a kk
  let ?d = λkk. if kk = tk then dir.R else dir.N

```

```

let ?s = (q, AR_SimWrite, tk, i, buf, dvec, posk)
let ?s' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
have aTk: ?a tk ≠ LE4 using notLE c'_eq by simp
have kpos: 0 < block_width (Γ_tm M) using block_width_pos[of Γ_tm M] by
simp
have jlt: i - block_width (Γ_tm M) < block_width (Γ_tm M) using ihi kpos
by linarith
have wb_not_LE: ?wb ≠ LE4 using write_bit_not_LE4[OF jlt] .
have rel_in: (?s, ?a, ?s', ?a', ?d) ∈ ar_delta_write M
  unfolding ar_delta_write_def
  by (rule UnI2) (use qQ poskproper last ihi in blast)
have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  using vsrc by (simp add: ar_valid_stage_def)
have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
  using kpos buf_valid by (simp add: ar_valid_stage_def)
have pad_a: ∀ j ≥ k_tm M. ?a j = BLANK4 using pad_blank c'_eq by simp
have tk_lt: tk < k_tm M using src_bounded by (simp add:
ar_stage_bounded_def)
have kpos_tm: 0 < k_tm M using vM by (cases M) auto
have pad_read: ∀ j ≥ k_tm M. ?a j = BLANK4 ∧ ?a' j = BLANK4 ∧ ?d j =
dir.N
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using tk_lt jge by linarith
  show ?a j = BLANK4 ∧ ?a' j = BLANK4 ∧ ?d j = dir.N
    using pad_a jge jne by simp
qed
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  using src_bounded kpos_tm by (simp add: ar_stage_bounded_def
k_unidx_def)
have ard_in: (?s, ?a, ?s', ?a', ?d) ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid aTk wb_not_LE pad_read src_bounded dst_bd
  by (auto split: if_splits)
have tape_new: (λk. (ts k)(n k := ?a' k))
  = ts(tk := (ts tk)(n tk := ?wb))
  by (rule ext) (auto simp: fun_upd_triv)
have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := Suc (n tk))
  by (rule ext) simp
let ?c'' = Config_M ?s' (ts(tk := (ts tk)(n tk := ?wb))) (n(tk := Suc (n tk)))
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a' k))
  (λk. go_dir (?d k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a', ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq tape_new pos_new by simp

```

```

show ?thesis
  using step by (intro exI[where  $x = ?c'$ ]) (simp add:  $c'_eq$ )
qed

```

The read LE substep, non-last tape: at position 0 the head reads LE_4 , sets $buf\ tk := le_tm\ M$ directly (the single LE-cell needs no accumulator), moves R on tk to position 1, and hands off to the next tape's read. Tape unchanged. Target-stage validity uses $valid_mttm_LE_in_Gamma$ for the $buf\ tk := le_tm\ M$ entry.

```

lemma ar_read_le_step:
  fixes  $M :: ('q, 'a)\ mttm$ 
  and  $c' :: (sym_4, 'q \times 'a\ ar\_stage)\ mt\_config$ 
  assumes  $vM: valid\_mttm\ M$ 
  and  $stg: mt\_state\ c' = (q, AR\_SimRead, tk, 0, buf, dvec, posk)$ 
  and  $qQ: q \in Q\_tm\ M$ 
  and  $notlast: \neg is\_last\_k\ M\ tk$ 
  and  $posk\_le: posk\ tk = AR\_AtLE$ 
  and  $aLE: mt\_tape\ c'\ tk\ (mt\_pos\ c'\ tk) = LE_4$ 
  and  $vsrc: ar\_valid\_stage\ (\Gamma\_tm\ M)\ (bl\_tm\ M)$ 
     $(AR\_SimRead, tk, 0, buf, dvec, posk)$ 
  and  $pad\_blank: \forall j \geq k\_tm\ M. mt\_tape\ c'\ j\ (mt\_pos\ c'\ j) = BLANK_4$ 
  and  $src\_bounded: ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
     $(AR\_SimRead, tk, 0, buf, dvec, posk)$ 
  shows  $\exists c''. (c', c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
     $\wedge mt\_state\ c'' = (q, AR\_SimRead, k\_succ\ tk, 0,$ 
       $buf(tk := le\_tm\ M), dvec, posk)$ 
     $\wedge mt\_tape\ c'' = mt\_tape\ c'$ 
     $\wedge mt\_pos\ c'' = (mt\_pos\ c')(tk := Suc\ (mt\_pos\ c'\ tk))$ 
proof –
  obtain  $ts\ n$  where  $c'_eq$ :
     $c' = Config_M\ (q, AR\_SimRead, tk, 0, buf, dvec, posk)\ ts\ n$ 
  using stg by (cases  $c'$ ) auto
  let  $?a = \lambda k. ts\ k\ (n\ k)$ 
  let  $?d = \lambda kk. if\ kk = tk\ then\ dir.R\ else\ dir.N$ 
  let  $?s = (q, AR\_SimRead, tk, 0, buf, dvec, posk)$ 
  let  $?s' = (q, AR\_SimRead, k\_succ\ tk, 0, buf(tk := le\_tm\ M), dvec, posk)$ 
  have  $aTk: ?a\ tk = LE_4$  using  $aLE\ c'_eq$  by simp
  have  $rel\_in: (?s, ?a, ?s', ?a, ?d) \in ar\_delta\_read\ M$ 
  unfolding ar_delta_read_def
  using  $qQ\ notlast\ posk\_le\ aTk$  by (intro UnI1) blast
  have  $buf\_valid: \forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  using vsrc by (simp add: ar_valid_stage_def)
  have  $kpos: 0 < block\_width\ (\Gamma\_tm\ M)$  using  $block\_width\_pos[of\ \Gamma\_tm\ M]$  by
simp
  have  $dst\_valid: ar\_valid\_stage\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (snd\ ?s')$ 
  using  $buf\_valid\ valid\_mttm\ LE\_in\_Gamma[OF\ vM]\ kpos$ 
  by (auto simp: ar_valid_stage_def)
  have  $pad\_a: \forall j \geq k\_tm\ M. ?a\ j = BLANK_4$  using  $pad\_blank\ c'_eq$  by simp
  have  $tk\_lt: tk < k\_tm\ M$  using src_bounded by (simp add:

```

```

ar_stage_bounded_def)
have ksucc_lt: k_succ tk < k_tm M
proof -
  have ne: Suc tk ≠ k_tm M using notlast by (simp add: is_last_k_def)
  have le: Suc tk ≤ k_tm M using tk_lt by simp
  show ?thesis unfolding k_succ_def using le_neq_implies_less[OF le ne] .
qed
have pad_read: ∀ j ≥ k_tm M. ?a j = BLANK4 ∧ ?a j = BLANK4 ∧ ?d j =
dir.N
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using tk_lt jge by linarith
  show ?a j = BLANK4 ∧ ?a j = BLANK4 ∧ ?d j = dir.N
    using pad_a jge jne by simp
qed
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
proof -
  have bt: ∀ j ≥ k_tm M. (buf(tk := le_tm M)) j = bl_tm M
  proof (intro allI impI)
    fix j assume jge: k_tm M ≤ j
    have j ≠ tk using tk_lt jge by linarith
    thus (buf(tk := le_tm M)) j = bl_tm M
      using src_bounded jge by (simp add: ar_stage_bounded_def)
  qed
  show ?thesis using src_bounded ksucc_lt bt by (simp add:
ar_stage_bounded_def)
qed
have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd
  by (auto split: if_splits)
have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := Suc (n tk))
  by (rule ext) simp
let ?c'' = Config_M ?s' ts (n(tk := Suc (n tk)))
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a k))
  (λk. go_dir (?d k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_new by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The read LE substep, last tape: as *ar_read_le_step* but *tk* is the last tape, so the hand-off goes to *AR_SimCompute* (current-tape field reset

to $k_unidx\ 0$), beginning the compute substep. Arm 2 of the seven-arm ar_delta_read union.

lemma $ar_read_le_finish_step$:

fixes $M :: ('q, 'a) mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage) mt_config$
assumes vM : $valid_mttm\ M$
and stg : $mt_state\ c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and qQ : $q \in Q_tm\ M$
and $last$: $is_last_k\ M\ tk$
and $posk_le$: $posk\ tk = AR_AtLE$
and aLE : $mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) = LE4$
and $vsrc$: $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
and pad_blank : $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src_bounded$: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step\ (alphabet_reduce_delta\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf(tk := le_tm\ M), dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := Suc\ (mt_pos\ c'\ tk))$

proof –

obtain $ts\ n$ **where** c'_eq :
 $c' = Config_M\ (q, AR_SimRead, tk, 0, buf, dvec, posk)\ ts\ n$
using stg **by** $(cases\ c')\ auto$
let $?a = \lambda k. ts\ k\ (n\ k)$
let $?d = \lambda kk. if\ kk = tk\ then\ dir.R\ else\ dir.N$
let $?s = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
let $?s' = (q, AR_SimCompute, k_unidx\ 0, 0, buf(tk := le_tm\ M), dvec, posk)$
have aTk : $?a\ tk = LE4$ **using** $aLE\ c'_eq$ **by** $simp$
have rel_in : $(?s, ?a, ?s', ?a, ?d) \in ar_delta_read\ M$
unfolding $ar_delta_read_def$
by $(rule\ UnI1, rule\ UnI1, rule\ UnI1, rule\ UnI1, rule\ UnI1, rule\ UnI2)$
 $(use\ qQ\ last\ posk_le\ aTk\ in\ blast)$
have buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using $vsrc$ **by** $(simp\ add: ar_valid_stage_def)$
have $kpos$: $0 < block_width\ (\Gamma_tm\ M)$ **using** $block_width_pos[of\ \Gamma_tm\ M]$ **by**
 $simp$
have dst_valid : $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ ?s')$
using $buf_valid\ valid_mttm\ LE_in_Gamma[OF\ vM]\ kpos$
by $(auto\ simp: ar_valid_stage_def)$
have pad_a : $\forall j \geq k_tm\ M. ?a\ j = BLANK4$ **using** $pad_blank\ c'_eq$ **by** $simp$
have $tk\ lt$: $tk < k_tm\ M$ **using** $src_bounded$ **by** $(simp\ add:$
 $ar_stage_bounded_def)$
have $kpos_tm$: $0 < k_tm\ M$ **using** vM **by** $(cases\ M)\ auto$
have pad_read : $\forall j \geq k_tm\ M. ?a\ j = BLANK4 \wedge ?a\ j = BLANK4 \wedge ?d\ j =$
 $dir.N$
proof $(intro\ allI\ impI)$
fix j **assume** jge : $k_tm\ M \leq j$

```

have jne:  $j \neq tk$  using  $tk\_lt\ jge$  by linarith
show  $?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j = dir.N$ 
  using  $pad\_a\ jge\ jne$  by simp
qed
have  $dst\_bd: ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)\ (snd\ ?s')$ 
proof -
  have  $bt: \forall j \geq k\_tm\ M. (buf(tk := le\_tm\ M))\ j = bl\_tm\ M$ 
  proof (intro allI impI)
    fix  $j$  assume  $jge: k\_tm\ M \leq j$ 
    have  $j \neq tk$  using  $tk\_lt\ jge$  by linarith
    thus  $(buf(tk := le\_tm\ M))\ j = bl\_tm\ M$ 
      using  $src\_bounded\ jge$  by (simp add: ar\_stage\_bounded\_def)
  qed
show ?thesis using  $src\_bounded\ kpos\_tm\ bt$ 
  by (simp add: ar\_stage\_bounded\_def k\_unidx\_def)
qed
have  $ard\_in: (?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$ 
  unfolding  $alphabet\_reduce\_delta\_def$ 
  using  $rel\_in\ vsrc\ dst\_valid\ pad\_read\ src\_bounded\ dst\_bd$ 
  by (auto split: if_splits)
have  $ts\_unchanged: (\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$  by (rule ext) auto
have  $pos\_new: (\lambda k. go\_dir\ (?d\ k)\ (n\ k)) = n(tk := Suc\ (n\ tk))$ 
  by (rule ext) simp
let  $?c'' = Config_M\ ?s'\ ts\ (n(tk := Suc\ (n\ tk)))$ 
have ( $Config_M\ ?s\ ts\ n,$ 
   $Config_M\ ?s'\ (\lambda k. (ts\ k)(n\ k := ?a\ k))$ 
   $(\lambda k. go\_dir\ (?d\ k)\ (n\ k))$ )
   $\in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
proof (rule mttm\_step.intros)
  show  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$  by (rule ard\_in)
qed
hence  $step: (c', ?c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
  using  $c'\_eq\ ts\_unchanged\ pos\_new$  by simp
show ?thesis
  using step by (intro exI[where  $x = ?c''$ ]) (simp add: c'\_eq)
qed

```

The decoder image always lies in $\Gamma \cup \{bl\}$: for $n < card\ \Gamma$ it is *inv_into* $\Gamma\ (gamma_enum\ \Gamma\ bl)\ n$, which is in Γ since *gamma_enum* is onto $\{.. < card\ \Gamma\}$ (bijection); the out-of-range fallback is *bl*. Totality here means the read arms' *buf*-update validity holds for any accumulator value without per-arm range reasoning.

```

lemma gamma_unenum_mem:
  assumes finite  $\Gamma$ 
  shows  $gamma\_unenum\ \Gamma\ bl\ n \in \Gamma \cup \{bl\}$ 
proof (cases  $n < card\ \Gamma$ )
  case True
  have  $n \in gamma\_enum\ \Gamma\ bl\ ' \Gamma$ 
    using True gamma_enum_bij[OF assms] by (auto simp: bij_betw_def)

```


hence $inv_into \Gamma (gamma_enum \Gamma bl) n \in \Gamma$ **by** (rule inv_into_into)
 thus $?thesis$ **using** $True$ **by** (simp add: $gamma_unenum_def$)
next
 case $False$
 thus $?thesis$ **by** (simp add: $gamma_unenum_def$)
qed

The read proper-arm look-back step 1 ($i = 0$): with the position-kind already in the proper region, the head moves L from $sim_pos(p)$ to $sim_pos(p) - 1$ (the cell to inspect for refining the position-kind), no buf change, transitioning to $i = 1$. Tape unchanged; a $tk \neq LE4$ selects the arm and discharges the backward- δLE filter. Arm 3 of the seven-arm union.

lemma $ar_read_lookback1_step$:

fixes $M :: ('q, 'a) mttm$
and $c' :: (sym4, 'q \times 'a ar_stage) mt_config$
assumes vM : $valid_mttm M$
and stg : $mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and qQ : $q \in Q_tm M$
and $posk_proper$: $posk tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and $notLE$: $mt_tape c' tk (mt_pos c' tk) \neq LE4$
and $vsrc$: $ar_valid_stage (\Gamma_tm M) (bl_tm M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
and pad_blank : $\forall j \geq k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4$
and $src_bounded$: $ar_stage_bounded (bl_tm M) (k_tm M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step (alphabet_reduce_delta M)$
 $\wedge mt_state c'' = (q, AR_SimRead, tk, Suc 0, buf, dvec, posk)$
 $\wedge mt_tape c'' = mt_tape c'$
 $\wedge mt_pos c'' = (mt_pos c')(tk := mt_pos c' tk - 1)$

proof –

obtain $ts\ n$ **where** c'_eq :
 $c' = Config_M (q, AR_SimRead, tk, 0, buf, dvec, posk) ts\ n$
using stg **by** (cases c') *auto*
let $?a = \lambda k. ts\ k\ (n\ k)$
let $?d = \lambda kk. if\ kk = tk\ then\ dir.L\ else\ dir.N$
let $?s = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
let $?s' = (q, AR_SimRead, tk, Suc\ 0, buf, dvec, posk)$
have aTk : $?a\ tk \neq LE4$ **using** $notLE\ c'_eq$ **by** *simp*
have rel_in : $(?s, ?a, ?s', ?a, ?d) \in ar_delta_read\ M$
unfolding $ar_delta_read_def$
by (rule $UnI1$, rule $UnI1$, rule $UnI1$, rule $UnI1$, rule $UnI2$)
 $(use\ qQ\ posk_proper\ aTk\ in\ blast)$
have buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using $vsrc$ **by** (simp add: $ar_valid_stage_def$)
have $kpos$: $0 < block_width (\Gamma_tm\ M)$ **using** $block_width_pos[of\ \Gamma_tm\ M]$ **by** *simp*
have dst_valid : $ar_valid_stage (\Gamma_tm\ M) (bl_tm\ M) (snd\ ?s')$
using $kpos\ buf_valid$ **by** (simp add: $ar_valid_stage_def$)
have pad_a : $\forall j \geq k_tm\ M. ?a\ j = BLANK4$ **using** $pad_blank\ c'_eq$ **by** *simp*

```

    have tk_lt: tk < k_tm M using src_bounded by (simp add:
ar_stage_bounded_def)
    have pad_read:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j =$ 
dir.N
    proof (intro allI impI)
      fix j assume jge: k_tm M ≤ j
      have jne: j ≠ tk using tk_lt jge by linarith
      show ?a j = BLANK4 ∧ ?a j = BLANK4 ∧ ?d j = dir.N
        using pad_a jge jne by simp
    qed
    have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
      using src_bounded by (simp add: ar_stage_bounded_def)
    have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
      unfolding alphabet_reduce_delta_def
      using rel_in vsrc dst_valid aTk pad_read src_bounded dst_bd
      by (auto split: if_splits)
    have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
    have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := n tk - 1)
      by (rule ext) simp
    let ?c'' = ConfigM ?s' ts (n(tk := n tk - 1))
    have (ConfigM ?s ts n,
      ConfigM ?s' (λk. (ts k)(n k := ?a k))
      (λk. go_dir (?d k) (n k)))
      ∈ mttm_step (alphabet_reduce_delta M)
    proof (rule mttm_step.intros)
      show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
    qed
    hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
      using c'_eq ts_unchanged pos_new by simp
    show ?thesis
      using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
  qed

```

The read proper-arm look-back step 2 ($i = 1$): at $sim_pos(p) - 1$ the head reads the cell, refines the position-kind ($AR_AtFirstProper$ if that cell is LE_4 , i.e. the head was at $sim_pos\ 1$, else $AR_AtFurtherProper$), resets $buf\ tk$ to the zero-bits partial decode, and moves R back to $sim_pos(p)$, transitioning to $i = 2$. The R -move makes δLE trivial even when reading LE_4 . Needs $2 \leq b$ for target validity ($i = 2$). Arm 4 of the union.

lemma $ar_read_lookback2_step$:

```

fixes M :: ('q, 'a) mttm
and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
and stg: mt_state c' = (q, AR_SimRead, tk, Suc 0, buf, dvec, posk)
and qQ: q ∈ Q_tm M
and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
and kge2: 2 ≤ block_width (Γ_tm M)
and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
      (AR_SimRead, tk, Suc 0, buf, dvec, posk)

```

```

    and pad_blank:  $\forall j \geq k\_tm\ M. mt\_tape\ c'\ j\ (mt\_pos\ c'\ j) = BLANK4$ 
    and src_bounded:  $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
       $(AR\_SimRead, tk, Suc\ 0, buf, dvec, posk)$ 
shows  $\exists c''. (c', c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
   $\wedge mt\_state\ c'' = (q, AR\_SimRead, tk, Suc\ (Suc\ 0),$ 
     $buf(tk := gamma\_unenum\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ 0), dvec,$ 
     $posk(tk := if\ mt\_tape\ c'\ tk\ (mt\_pos\ c'\ tk) = LE4$ 
       $then\ AR\_AtFirstProper\ else\ AR\_AtFurtherProper))$ 
   $\wedge mt\_tape\ c'' = mt\_tape\ c'$ 
   $\wedge mt\_pos\ c'' = (mt\_pos\ c')(tk := Suc\ (mt\_pos\ c'\ tk))$ 
proof -
  obtain  $ts\ n$  where  $c'\ eq$ :
     $c' = Config_M\ (q, AR\_SimRead, tk, Suc\ 0, buf, dvec, posk)\ ts\ n$ 
  using  $stg$  by  $(cases\ c')\ auto$ 
  let  $?a = \lambda k. ts\ k\ (n\ k)$ 
  let  $?d = \lambda kk. if\ kk = tk\ then\ dir.R\ else\ dir.N$ 
  let  $?s = (q, AR\_SimRead, tk, Suc\ 0, buf, dvec, posk)$ 
  let  $?s' = (q, AR\_SimRead, tk, Suc\ (Suc\ 0),$ 
     $buf(tk := gamma\_unenum\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ 0), dvec,$ 
     $posk(tk := if\ ?a\ tk = LE4$ 
       $then\ AR\_AtFirstProper\ else\ AR\_AtFurtherProper))$ 
  have  $rel\ in: (?s, ?a, ?s', ?a, ?d) \in ar\_delta\_read\ M$ 
  unfolding  $ar\_delta\_read\_def$ 
  by  $(rule\ UnI1, rule\ UnI1, rule\ UnI1, rule\ UnI2)$ 
   $(use\ qQ\ posk\_proper\ in\ blast)$ 
  have  $buf\_valid: \forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  using  $vsr$  by  $(simp\ add: ar\_valid\_stage\_def)$ 
  have  $gu\_mem: gamma\_unenum\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ 0 \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  using  $gamma\_unenum\_mem[OF\ valid\_mttm\_finite\_Gamma[OF\ vM]]$  .
  have  $dst\_valid: ar\_valid\_stage\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (snd\ ?s')$ 
  using  $kge2\ buf\_valid\ gu\_mem$  by  $(auto\ simp: ar\_valid\_stage\_def)$ 
  have  $pad\_a: \forall j \geq k\_tm\ M. ?a\ j = BLANK4$  using  $pad\_blank\ c'\ eq$  by  $simp$ 
  have  $tk\_lt: tk < k\_tm\ M$  using  $src\_bounded$  by  $(simp\ add: ar\_stage\_bounded\_def)$ 
  have  $pad\_read: \forall j \geq k\_tm\ M. ?a\ j = BLANK4 \wedge ?a\ j = BLANK4 \wedge ?d\ j = dir.N$ 
  proof  $(intro\ allI\ impI)$ 
    fix  $j$  assume  $jge: k\_tm\ M \leq j$ 
    have  $jne: j \neq tk$  using  $tk\_lt\ jge$  by  $linarith$ 
    show  $?a\ j = BLANK4 \wedge ?a\ j = BLANK4 \wedge ?d\ j = dir.N$ 
      using  $pad\_a\ jge\ jne$  by  $simp$ 
  qed
  have  $dst\_bd: ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)\ (snd\ ?s')$ 
  proof -
    have  $bt: \forall j \geq k\_tm\ M.$ 
       $(buf(tk := gamma\_unenum\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ 0))\ j = bl\_tm\ M$ 
    proof  $(intro\ allI\ impI)$ 
      fix  $j$  assume  $jge: k\_tm\ M \leq j$ 

```

```

    have  $j \neq tk$  using  $tk\_lt\ jge$  by linarith
    thus ( $buf(tk := gamma\_unenum (\Gamma\_tm\ M) (bl\_tm\ M)\ 0)$ )  $j = bl\_tm\ M$ 
      using  $src\_bounded\ jge$  by (simp add: ar\_stage\_bounded\_def)
  qed
  have  $pt: \forall j \geq k\_tm\ M. (posk(tk := if\ ?a\ tk = LE4$ 
    then  $AR\_AtFirstProper$  else  $AR\_AtFurtherProper$ ))  $j = AR\_AtLE$ 
  proof (intro allI impI)
    fix  $j$  assume  $jge: k\_tm\ M \leq j$ 
    have  $j \neq tk$  using  $tk\_lt\ jge$  by linarith
    thus ( $posk(tk := if\ ?a\ tk = LE4$ 
      then  $AR\_AtFirstProper$  else  $AR\_AtFurtherProper$ ))  $j = AR\_AtLE$ 
      using  $src\_bounded\ jge$  by (simp add: ar\_stage\_bounded\_def)
  qed
  show ?thesis using  $src\_bounded\ tk\_lt\ bt\ pt$ 
    by (simp add: ar\_stage\_bounded\_def)
  qed
  have  $ard\_in: (?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$ 
    unfolding  $alphabet\_reduce\_delta\_def$ 
    using  $rel\_in\ vsrc\ dst\_valid\ pad\_read\ src\_bounded\ dst\_bd$ 
    by (auto split: if\_splits)
  have  $ts\_unchanged: (\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$  by (rule ext) auto
  have  $pos\_new: (\lambda k. go\_dir\ (?d\ k)\ (n\ k)) = n(tk := Suc\ (n\ tk))$ 
    by (rule ext) simp
  let  $?c'' = Config_M\ ?s'\ ts\ (n(tk := Suc\ (n\ tk)))$ 
  have ( $Config_M\ ?s\ ts\ n,$ 
     $Config_M\ ?s'\ (\lambda k. (ts\ k)(n\ k := ?a\ k))$ 
     $(\lambda k. go\_dir\ (?d\ k)\ (n\ k))$ 
     $\in mttm\_step\ (alphabet\_reduce\_delta\ M)$ )
  proof (rule mttm\_step.intros)
    show  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$  by (rule ard\_in)
  qed
  hence  $step: (c', ?c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
    using  $c'\_eq\ ts\_unchanged\ pos\_new$  by simp
  show ?thesis
    using  $step$  by (intro exI[where  $x = ?c''$ ]) (simp add: c'\_eq)
  qed

```

The read proper-arm per-bit stepping substep ($2 \leq i, Suc\ i \leq Suc\ b$): at $sim_pos(p) + (i-2)$ the head reads a bit cell and folds it into the partial decode via the $gamma_enum/gamma_unenum$ roundtrip $partial' = 2 \cdot gamma_enum\ (buf\ tk) + bit_value\ (a\ tk)$, moves R , and continues to $i + 1$. Tape unchanged. Needs $2 \leq b$ for target validity at the upper end ($Suc\ i = Suc\ b$). Arm 5 of the union.

```

lemma ar_read_bit_step:
  fixes  $M :: ('q, 'a)\ mttm$ 
  and  $c' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config$ 
  assumes  $vM: valid\_mttm\ M$ 
    and  $stg: mt\_state\ c' = (q, AR\_SimRead, tk, i, buf, dvec, posk)$ 
    and  $qQ: q \in Q\_tm\ M$ 

```

```

and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
and ilo:  $2 \leq i$ 
and ihi:  $\text{Suc } i \leq \text{Suc } (\text{block\_width } (\Gamma\_tm \ M))$ 
and kge2:  $2 \leq \text{block\_width } (\Gamma\_tm \ M)$ 
and vsr: ar_valid_stage ( $\Gamma\_tm \ M$ ) (bl_tm M)
      (AR_SimRead, tk, i, buf, dvec, posk)
and pad_blank:  $\forall j \geq k\_tm \ M. \text{mt\_tape } c' \ j \ (\text{mt\_pos } c' \ j) = \text{BLANK4}$ 
and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
      (AR_SimRead, tk, i, buf, dvec, posk)
shows  $\exists c''. (c', c'') \in \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 
       $\wedge \text{mt\_state } c'' = (q, \text{AR\_SimRead}, tk, \text{Suc } i,$ 
       $\text{buf}(tk := \text{gamma\_unenum } (\Gamma\_tm \ M) (\text{bl\_tm } M)$ 
       $(2 * \text{gamma\_enum } (\Gamma\_tm \ M) (\text{bl\_tm } M) (\text{buf } tk)$ 
       $+ \text{bit\_value } (\text{mt\_tape } c' \ tk \ (\text{mt\_pos } c' \ tk))))),$ 
      dvec, posk)
       $\wedge \text{mt\_tape } c'' = \text{mt\_tape } c'$ 
       $\wedge \text{mt\_pos } c'' = (\text{mt\_pos } c')(tk := \text{Suc } (\text{mt\_pos } c' \ tk))$ 
proof –
  obtain ts n where c'_eq:
    c' = ConfigM (q, AR_SimRead, tk, i, buf, dvec, posk) ts n
    using stg by (cases c') auto
  let ?a =  $\lambda k. \text{ts } k \ (n \ k)$ 
  let ?d =  $\lambda kk. \text{if } kk = tk \text{ then } dir.R \text{ else } dir.N$ 
  let ?s = (q, AR_SimRead, tk, i, buf, dvec, posk)
  let ?s' = (q, AR_SimRead, tk, Suc i,
    buf(tk := gamma_unenum ( $\Gamma\_tm \ M$ ) (bl_tm M)
      ( $2 * \text{gamma\_enum } (\Gamma\_tm \ M) (\text{bl\_tm } M) (\text{buf } tk) + \text{bit\_value } ($ 
       $(?a \ tk)))$ ),
    dvec, posk)
  have rel_in: (?s, ?a, ?s', ?a, ?d) ∈ ar_delta_read M
  unfolding ar_delta_read_def
  by (rule UnI1, rule UnI1, rule UnI2)
    (use qQ posk_proper ilo ihi in blast)
  have buf_valid:  $\forall k. \text{buf } k \in \Gamma\_tm \ M \cup \{\text{bl\_tm } M\}$ 
  using vsr by (simp add: ar_valid_stage_def)
  have gu_mem: gamma_unenum ( $\Gamma\_tm \ M$ ) (bl_tm M)
    ( $2 * \text{gamma\_enum } (\Gamma\_tm \ M) (\text{bl\_tm } M) (\text{buf } tk) + \text{bit\_value } (?a$ 
    tk))
    ∈  $\Gamma\_tm \ M \cup \{\text{bl\_tm } M\}$ 
  using gamma_unenum_mem[OF valid_mttm_finite_Gamma[OF vM]] .
  have suc_i_lt:  $\text{Suc } i < 2 * \text{block\_width } (\Gamma\_tm \ M)$ 
  using ihi kge2 by linarith
  have dst_valid: ar_valid_stage ( $\Gamma\_tm \ M$ ) (bl_tm M) (snd ?s')
  using suc_i_lt buf_valid gu_mem by (auto simp: ar_valid_stage_def)
  have pad_a:  $\forall j \geq k\_tm \ M. ?a \ j = \text{BLANK4}$  using pad_blank c'_eq by simp
  have tk_lt: tk < k_tm M using src_bounded by (simp add: ar_stage_bounded_def)
  have pad_read:  $\forall j \geq k\_tm \ M. ?a \ j = \text{BLANK4} \wedge ?a \ j = \text{BLANK4} \wedge ?d \ j =$ 
dir.N

```

```

proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using tk_lt jge by linarith
  show ?a j = BLANK₄ ∧ ?a j = BLANK₄ ∧ ?d j = dir.N
    using pad_a jge jne by simp
qed
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
proof -
  have bt: ∀ j ≥ k_tm M.
    (buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)
      (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)
        + bit_value (?a tk)))) j = bl_tm M
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have j ≠ tk using tk_lt jge by linarith
  thus (buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)
    (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)
      + bit_value (?a tk)))) j = bl_tm M
    using src_bounded jge by (simp add: ar_stage_bounded_def)
qed
  show ?thesis using src_bounded tk_lt bt by (simp add:
ar_stage_bounded_def)
qed
have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd
  by (auto split: if_splits)
have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := Suc (n tk))
  by (rule ext) simp
let ?c'' = Config_M ?s' ts (n(tk := Suc (n tk)))
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a k))
    (λk. go_dir (?d k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_new by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The read proper-arm per-bit boundary substep ($i = \text{Suc } b$), non-last tape: the last-bit accumulator step (as ar_read_bit_step) leaving $\text{buf } tk$ the fully-decoded M -symbol, then R -move and hand-off to the next tape k_{succ} tk with the bit-counter reset. Arm 6 of the union.

lemma $\text{ar_read_bit_boundary_step}$:

```

fixes  $M :: ('q, 'a) \text{mttm}$ 
and  $c' :: (\text{sym}_4, 'q \times 'a \text{ar\_stage}) \text{mt\_config}$ 
assumes  $vM: \text{valid\_mttm } M$ 
and  $\text{stg}: \text{mt\_state } c' = (q, \text{AR\_SimRead}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
and  $qQ: q \in Q\_tm \ M$ 
and  $\text{notlast}: \neg \text{is\_last\_k } M \ tk$ 
and  $\text{posk\_proper}: \text{posk } tk \in \{\text{AR\_AtFirstProper}, \text{AR\_AtFurtherProper}\}$ 
and  $\text{ieq}: i = \text{Suc } (\text{block\_width } (\Gamma\_tm \ M))$ 
and  $\text{vsrc}: \text{ar\_valid\_stage } (\Gamma\_tm \ M) \ (\text{bl\_tm } M)$ 
 $(\text{AR\_SimRead}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
and  $\text{pad\_blank}: \forall j \geq k\_tm \ M. \text{mt\_tape } c' \ j \ (\text{mt\_pos } c' \ j) = \text{BLANK}_4$ 
and  $\text{src\_bounded}: \text{ar\_stage\_bounded } (\text{bl\_tm } M) \ (k\_tm \ M)$ 
 $(\text{AR\_SimRead}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
shows  $\exists c''. (c', c'') \in \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 
 $\wedge \text{mt\_state } c'' = (q, \text{AR\_SimRead}, k\_succ \ tk, 0,$ 
 $\text{buf}(tk := \text{gamma\_unenum } (\Gamma\_tm \ M) \ (\text{bl\_tm } M)$ 
 $(2 * \text{gamma\_enum } (\Gamma\_tm \ M) \ (\text{bl\_tm } M) \ (\text{buf } tk)$ 
 $+ \text{bit\_value } (\text{mt\_tape } c' \ tk \ (\text{mt\_pos } c' \ tk))))),$ 
 $\text{dvec}, \text{posk})$ 
 $\wedge \text{mt\_tape } c'' = \text{mt\_tape } c'$ 
 $\wedge \text{mt\_pos } c'' = (\text{mt\_pos } c') (tk := \text{Suc } (\text{mt\_pos } c' \ tk))$ 

proof –
obtain  $ts \ n$  where  $c' \text{\_eq}$ :
 $c' = \text{Config}_M \ (q, \text{AR\_SimRead}, tk, i, \text{buf}, \text{dvec}, \text{posk}) \ ts \ n$ 
using  $\text{stg}$  by  $(\text{cases } c') \ \text{auto}$ 
let  $?a = \lambda k. ts \ k \ (n \ k)$ 
let  $?d = \lambda kk. \text{if } kk = tk \ \text{then } \text{dir}.R \ \text{else } \text{dir}.N$ 
let  $?s = (q, \text{AR\_SimRead}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
let  $?s' = (q, \text{AR\_SimRead}, k\_succ \ tk, 0,$ 
 $\text{buf}(tk := \text{gamma\_unenum } (\Gamma\_tm \ M) \ (\text{bl\_tm } M)$ 
 $(2 * \text{gamma\_enum } (\Gamma\_tm \ M) \ (\text{bl\_tm } M) \ (\text{buf } tk) + \text{bit\_value}$ 
 $(?a \ tk))))),$ 
 $\text{dvec}, \text{posk})$ 
have  $\text{rel\_in}: (?s, ?a, ?s', ?a, ?d) \in \text{ar\_delta\_read } M$ 
unfolding  $\text{ar\_delta\_read\_def}$ 
by  $(\text{rule } \text{UnI1}, \text{rule } \text{UnI2}) \ (\text{use } qQ \ \text{notlast } \text{posk\_proper } \text{ieq} \ \text{in } \text{blast})$ 
have  $\text{buf\_valid}: \forall k. \text{buf } k \in \Gamma\_tm \ M \cup \{\text{bl\_tm } M\}$ 
using  $\text{vsrc}$  by  $(\text{simp } \text{add}: \text{ar\_valid\_stage\_def})$ 
have  $\text{gu\_mem}: \text{gamma\_unenum } (\Gamma\_tm \ M) \ (\text{bl\_tm } M)$ 
 $(2 * \text{gamma\_enum } (\Gamma\_tm \ M) \ (\text{bl\_tm } M) \ (\text{buf } tk) + \text{bit\_value } (?a$ 
 $tk))$ 
 $\in \Gamma\_tm \ M \cup \{\text{bl\_tm } M\}$ 
using  $\text{gamma\_unenum\_mem}[\text{OF } \text{valid\_mttm\_finite\_Gamma}[\text{OF } vM]]$  .
have  $kpos: 0 < \text{block\_width } (\Gamma\_tm \ M)$  using  $\text{block\_width\_pos}[\text{of } \Gamma\_tm \ M]$  by
 $\text{simp}$ 
have  $\text{dst\_valid}: \text{ar\_valid\_stage } (\Gamma\_tm \ M) \ (\text{bl\_tm } M) \ (\text{snd } ?s')$ 
using  $kpos \ \text{buf\_valid } \text{gu\_mem}$  by  $(\text{auto } \text{simp}: \text{ar\_valid\_stage\_def})$ 
have  $\text{pad\_a}: \forall j \geq k\_tm \ M. ?a \ j = \text{BLANK}_4$  using  $\text{pad\_blank } c' \text{\_eq}$  by  $\text{simp}$ 
have  $tk\_lt: tk < k\_tm \ M$  using  $\text{src\_bounded}$  by  $(\text{simp } \text{add}:$ 

```

```

ar_stage_bounded_def)
have ksucc_lt: k_succ tk < k_tm M
proof -
  have ne: Suc tk ≠ k_tm M using notlast by (simp add: is_last_k_def)
  have le: Suc tk ≤ k_tm M using tk_lt by simp
  show ?thesis unfolding k_succ_def using le_neq_implies_less[OF le ne] .
qed
have pad_read: ∀ j ≥ k_tm M. ?a j = BLANK4 ∧ ?a j = BLANK4 ∧ ?d j =
dir.N
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using tk_lt jge by linarith
  show ?a j = BLANK4 ∧ ?a j = BLANK4 ∧ ?d j = dir.N
    using pad_a jge jne by simp
qed
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
proof -
  have bt: ∀ j ≥ k_tm M.
    (buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)
      (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)
        + bit_value (?a tk)))) j = bl_tm M
  proof (intro allI impI)
    fix j assume jge: k_tm M ≤ j
    have j ≠ tk using tk_lt jge by linarith
    thus (buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)
      (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)
        + bit_value (?a tk)))) j = bl_tm M
      using src_bounded jge by (simp add: ar_stage_bounded_def)
  qed
  show ?thesis using src_bounded ksucc_lt bt by (simp add:
ar_stage_bounded_def)
qed
have ard_in: (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd
  by (auto split: if_splits)
have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
have pos_new: (λk. go_dir (?d k) (n k)) = n(tk := Suc (n tk))
  by (rule ext) simp
let ?c'' = Config_M ?s' ts (n(tk := Suc (n tk)))
have (Config_M ?s ts n,
  Config_M ?s' (λk. (ts k)(n k := ?a k))
  (λk. go_dir (?d k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_new by simp

```



```

show ?thesis
  using step by (intro exI[where x = ?c']) (simp add: c'_eq)
qed

```

The read proper-arm per-bit boundary substep, last tape: same last-bit accumulator step, transitioning to $AR_SimCompute$ with the current-tape field reset to $k_unidx\ 0$ (all tapes decoded, begin the compute substep). Arm 7 (rightmost) of the union.

```

lemma ar_read_bit_finish_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimRead, tk, i, buf, dvec, posk)
  and qQ: q ∈ Q_tm M
  and last: is_last_k M tk
  and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
  and ieq: i = Suc (block_width (Γ_tm M))
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, i, buf, dvec, posk)
  and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, i, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    ∧ mt_state c'' = (q, AR_SimCompute, k_unidx 0, 0,
      buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)
        (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)
          + bit_value (mt_tape c' tk (mt_pos c' tk)))),
      dvec, posk)
    ∧ mt_tape c'' = mt_tape c'
    ∧ mt_pos c'' = (mt_pos c')(tk := Suc (mt_pos c' tk))
proof -
  obtain ts n where c'_eq:
    c' = Config_M (q, AR_SimRead, tk, i, buf, dvec, posk) ts n
  using stg by (cases c') auto
  let ?a = λk. ts k (n k)
  let ?d = λkk. if kk = tk then dir.R else dir.N
  let ?s = (q, AR_SimRead, tk, i, buf, dvec, posk)
  let ?s' = (q, AR_SimCompute, k_unidx 0, 0,
    buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)
      (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)
        + bit_value (?a tk))),
    dvec, posk)
  have rel_in: (?s, ?a, ?s', ?a, ?d) ∈ ar_delta_read M
  unfolding ar_delta_read_def
  by (rule UnI2) (use qQ last posk_proper ieq in blast)
  have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  using vsrc by (simp add: ar_valid_stage_def)
  have gu_mem: gamma_unenum (Γ_tm M) (bl_tm M)
    (2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk) + bit_value (?a

```

```

tk))
      ∈  $\Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
    using gamma_unenum_mem[OF valid_mttm_finite_Gamma[OF vM]] .
    have kpos:  $0 < block\_width\ (\Gamma\_tm\ M)$  using block_width_pos[of  $\Gamma\_tm\ M$ ] by
simp
    have dst_valid: ar_valid_stage ( $\Gamma\_tm\ M$ ) ( $bl\_tm\ M$ ) (snd ?s')
    using kpos buf_valid gu_mem by (auto simp: ar_valid_stage_def)
    have pad_a:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4$  using pad_blank c'_eq by simp
    have tk_lt:  $tk < k\_tm\ M$  using src_bounded by (simp add:
ar_stage_bounded_def)
    have kpos_tm:  $0 < k\_tm\ M$  using vM by (cases M) auto
    have pad_read:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j =$ 
dir.N
  proof (intro allI impI)
    fix j assume jge:  $k\_tm\ M \leq j$ 
    have jne:  $j \neq tk$  using tk_lt jge by linarith
    show  $?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4 \wedge ?d\ j = dir.N$ 
      using pad_a jge jne by simp
  qed
  have dst_bd: ar_stage_bounded ( $bl\_tm\ M$ ) ( $k\_tm\ M$ ) (snd ?s')
  proof -
    have bt:  $\forall j \geq k\_tm\ M.$ 
      (buf(tk := gamma_unenum ( $\Gamma\_tm\ M$ ) ( $bl\_tm\ M$ )
        (2 * gamma_enum ( $\Gamma\_tm\ M$ ) ( $bl\_tm\ M$ ) (buf tk)
          + bit_value (?a tk))))  $j = bl\_tm\ M$ 
    proof (intro allI impI)
      fix j assume jge:  $k\_tm\ M \leq j$ 
      have jne:  $j \neq tk$  using tk_lt jge by linarith
      thus (buf(tk := gamma_unenum ( $\Gamma\_tm\ M$ ) ( $bl\_tm\ M$ )
        (2 * gamma_enum ( $\Gamma\_tm\ M$ ) ( $bl\_tm\ M$ ) (buf tk)
          + bit_value (?a tk))))  $j = bl\_tm\ M$ 
        using src_bounded jge by (simp add: ar_stage_bounded_def)
    qed
    show ?thesis using src_bounded kpos_tm bt
      by (simp add: ar_stage_bounded_def k_unidx_def)
  qed
  have ard_in:  $(?s, ?a, ?s', ?a, ?d) \in alphabet\_reduce\_delta\ M$ 
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd
  by (auto split: if_splits)
  have ts_unchanged:  $(\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$  by (rule ext) auto
  have pos_new:  $(\lambda k. go\_dir\ (?d\ k)\ (n\ k)) = n(tk := Suc\ (n\ tk))$ 
  by (rule ext) simp
  let ?c'' = Config_M ?s' ts (n(tk := Suc (n tk)))
  have (Config_M ?s ts n,
    Config_M ?s' ( $\lambda k. (ts\ k)(n\ k := ?a\ k)$ )
      ( $\lambda k. go\_dir\ (?d\ k)\ (n\ k)$ ))
    ∈ mttm_step (alphabet_reduce_delta M)
  proof (rule mttm_step.intros)

```

```

    show (?s, ?a, ?s', ?a, ?d) ∈ alphabet_reduce_delta M by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
    using c'_eq ts_unchanged pos_new by simp
show ?thesis
    using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

end
theory AlphabetReduction_ForwardRead
    imports AlphabetReduction_ForwardSubsteps
begin

```

2.13 Read phase

The accumulator fold stays inside $\Gamma \cup \{bl\}$: on a non-empty bit list the outermost ar_acc is a $gamma_unenum$ image (always in range by $gamma_unenum_mem$); on the empty list it is the seed. This is the read-loop's ar_valid_stage obligation discharged once for the running buf value, independent of how many bits have been folded so far. Throughout, b abbreviates $block_width \Gamma$ (the per-symbol cell width).

```

lemma foldl_ar_acc_mem:
  assumes finite  $\Gamma$  and  $a \in \Gamma \cup \{bl\}$ 
  shows foldl (ar_acc  $\Gamma$  bl) a xs ∈  $\Gamma \cup \{bl\}$ 
proof (induct xs rule: rev_induct)
  case Nil
  show ?case using assms(2) by simp
next
  case (snoc x ys)
  have foldl (ar_acc  $\Gamma$  bl) a (ys @ [x])
    = ar_acc  $\Gamma$  bl (foldl (ar_acc  $\Gamma$  bl) a ys) x by simp
  also have ... ∈  $\Gamma \cup \{bl\}$ 
    unfolding ar_acc_def by (rule gamma_unenum_mem[OF assms(1)])
  finally show ?case .
qed

```

The inner read loop, single tape, proper cell: starting at bit-counter 2 with the partial decode $buf0$ and the head at $base$ (the first bit cell, $sim_pos p$), the $ar_read_bit_step$ arm fires $b - 1$ times, walking R across the first $b - 1$ bit cells and folding each into $buf\ tk$ via ar_acc . After the loop the counter is $Suc\ b$ (ready for the boundary substep that reads the b -th, last bit), the head is at $base + (b - 1)$, the tape is untouched, and every other tape's head is where it was. Instantiates $relpow_invariant_chain$ with the loop-index-parameterised invariant whose $buf\ tk$ field is the $foldl\ (ar_acc)\ buf0_tk$ of the cells read so far — the same $foldl$ the read-decode arithmetic ($foldl_ar_acc_encode_symbol$) is stated against, so the per-step buf update is one $foldl_append$ rewrite.

Relation-generic read bit-accumulation loop scaffold. The bit-reading leaf (*ar_read_bit_step*) is the *leaf* hypothesis, quantified over config, counter, and buffer (the accumulator mutates each step); union and sub versions are thin instantiations.

lemma *ar_read_bit_loop_gen*:

fixes $M :: ('q, 'a) \text{ mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config rel}$
assumes *leaf*:
 $\bigwedge cc \ i \ \text{buf}' . \llbracket \text{valid_mttm } M;$
 $\text{mt_state } cc = (q, \text{AR_SimRead}, tk, i, \text{buf}', \text{dvec}, \text{posk});$
 $q \in Q_tm \ M;$
 $\text{posk } tk \in \{\text{AR_AtFirstProper}, \text{AR_AtFurtherProper}\};$
 $2 \leq i;$
 $\text{Suc } i \leq \text{Suc } (\text{block_width } (\Gamma_tm \ M));$
 $2 \leq \text{block_width } (\Gamma_tm \ M);$
 $\text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$
 $(\text{AR_SimRead}, tk, i, \text{buf}', \text{dvec}, \text{posk});$
 $\forall j \geq k_tm \ M. \text{mt_tape } cc \ j \ (\text{mt_pos } cc \ j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$
 $(\text{AR_SimRead}, tk, i, \text{buf}', \text{dvec}, \text{posk}) \rrbracket$
 $\implies \exists c'' . (cc, c'') \in R'$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimRead}, tk, \text{Suc } i,$
 $\text{buf}'(tk := \text{gamma_unenum } (\Gamma_tm \ M) \ (\text{bl_tm } M)$
 $(2 * \text{gamma_enum } (\Gamma_tm \ M) \ (\text{bl_tm } M) \ (\text{buf}'$
 $tk)$
 $+ \text{bit_value } (\text{mt_tape } cc \ tk \ (\text{mt_pos } cc \ tk))))),$
 $\text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } cc$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } cc)(tk := \text{Suc } (\text{mt_pos } cc \ tk))$
and $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $\text{posk_proper}: \text{posk } tk \in \{\text{AR_AtFirstProper}, \text{AR_AtFurtherProper}\}$
and $\text{stg}: \text{mt_state } c' = (q, \text{AR_SimRead}, tk, 2, \text{buf0}, \text{dvec}, \text{posk})$
and $\text{buf0_valid}: \forall k. \text{buf0 } k \in \Gamma_tm \ M \cup \{\text{bl_tm } M\}$
and $\text{pos_base}: \text{mt_pos } c' \ tk = \text{base}$
and $\text{pad0}: \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$
 $(\text{AR_SimRead}, tk, 2, \text{buf0}, \text{dvec}, \text{posk})$
shows $\exists c'' . (c', c'') \in R' \wedge \wedge (\text{block_width } (\Gamma_tm \ M) - 1)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimRead}, tk, \text{Suc } (\text{block_width } (\Gamma_tm \ M)),$
 $\text{buf0}(tk := \text{foldl } (\text{ar_acc } (\Gamma_tm \ M) \ (\text{bl_tm } M)) \ (\text{buf0 } tk)$
 $(\text{map } (\lambda m. \text{mt_tape } c' \ tk \ (\text{base} + m))$
 $[0..<\text{block_width } (\Gamma_tm \ M) - 1])),$
 $\text{dvec}, \text{posk})$
 $\wedge \text{mt_pos } c'' \ tk = \text{base} + (\text{block_width } (\Gamma_tm \ M) - 1)$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c'' \ k' = \text{mt_pos } c' \ k')$

proof –
have *finG*: *finite* ($\Gamma_tm\ M$) **by** (*rule valid_mttm_finite_Gamma*[*OF vM*])
let *?fold* = $\lambda j. foldl\ (ar_acc\ (\Gamma_tm\ M)\ (bl_tm\ M))\ (buf0\ tk)$
 $(map\ (\lambda m. mt_tape\ c'\ tk\ (base + m))\ [0..<j])$
let *?P* = $\lambda j\ c.$
 $mt_state\ c = (q, AR_SimRead, tk, j + 2, buf0(tk := ?fold\ j), dvec, posk)$
 $\wedge\ mt_pos\ c\ tk = base + j$
 $\wedge\ mt_tape\ c = mt_tape\ c'$
 $\wedge\ (\forall k'. k' \neq tk \longrightarrow mt_pos\ c\ k' = mt_pos\ c'\ k')$
have *mybase*: *?P* 0 *c'* **using** *stg pos_base* **by** *simp*
have *mystep*:
 $\wedge j\ cc. \llbracket j < block_width\ (\Gamma_tm\ M) - 1; ?P\ j\ cc \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge\ ?P\ (Suc\ j)\ c''$
proof –
fix *j cc*
assume *jlt*: $j < block_width\ (\Gamma_tm\ M) - 1$ **and** *Pj*: *?P j cc*
from *Pj* **have** *st_cc*:
 $mt_state\ cc = (q, AR_SimRead, tk, j + 2, buf0(tk := ?fold\ j), dvec, posk)$
and *pos_cc*: $mt_pos\ cc\ tk = base + j$
and *tape_cc*: $mt_tape\ cc = mt_tape\ c'$
and *other_cc*: $\forall k'. k' \neq tk \longrightarrow mt_pos\ cc\ k' = mt_pos\ c'\ k'$
by *simp_all*
have *tk_lt*: $tk < k_tm\ M$ **using** *src0* **by** (*simp add: ar_stage_bounded_def*)
have *pad_cc*: $\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4$
proof (*intro allI impI*)
fix *j' assume jge*: $k_tm\ M \leq j'$
have *jne*: $j' \neq tk$ **using** *jge tk_lt* **by** *simp*
have $mt_tape\ cc\ j'\ (mt_pos\ cc\ j') = mt_tape\ c'\ j'\ (mt_pos\ c'\ j')$
using *tape_cc other_cc jne* **by** *simp*
thus $mt_tape\ cc\ j'\ (mt_pos\ cc\ j') = BLANK4$ **using** *pad0 jge* **by** *simp*
qed
have *src_cc*: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, j + 2, buf0(tk := ?fold\ j), dvec, posk)$
using *src0 tk_lt* **by** (*auto simp: ar_stage_bounded_def*)
have *jhi*: $j + 2 \leq block_width\ (\Gamma_tm\ M)$ **using** *jlt kge2* **by** *linarith*
have *foldj_mem*: $?fold\ j \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using *buf0_valid* **by** (*intro foldl_ar_acc_mem*[*OF finG*]) *simp*
have *bufj_valid*: $\forall k. (buf0(tk := ?fold\ j))\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using *buf0_valid foldj_mem* **by** *simp*
have *vsrcc_cc*:
 $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, j + 2, buf0(tk := ?fold\ j), dvec, posk)$
using *jhi kge2 bufj_valid* **by** (*auto simp: ar_valid_stage_def*)
obtain *ccc* **where**
 $bstep: (cc, ccc) \in R'$
and *bst_state*: $mt_state\ ccc = (q, AR_SimRead, tk, Suc\ (j + 2),$
 $(buf0(tk := ?fold\ j))$
 $(tk := gamma_unenum\ (\Gamma_tm\ M)\ (bl_tm\ M))$

```

      (2 * gamma_enum (Γ_tm M) (bl_tm M) ((buf0(tk := ?fold
j)) tk)
      + bit_value (mt_tape cc tk (mt_pos cc tk)))),
      dvec, posk)
  and bst_tape: mt_tape ccc = mt_tape cc
  and bst_pos: mt_pos ccc = (mt_pos cc)(tk := Suc (mt_pos cc tk))
  using leaf[OF vM st_cc qQ posk_proper _ _ kge2 vsrc_cc pad_cc src_cc]
  jhi by force
  have read_cell: mt_tape cc tk (mt_pos cc tk) = mt_tape c' tk (base + j)
  using tape_cc pos_cc by simp
  have fold_step:
    gamma_unenum (Γ_tm M) (bl_tm M)
    (2 * gamma_enum (Γ_tm M) (bl_tm M) (?fold j)
    + bit_value (mt_tape c' tk (base + j)))
    = ?fold (Suc j)
  proof -
    have ?fold (Suc j)
      = foldl (ar_acc (Γ_tm M) (bl_tm M)) (buf0 tk)
        (map (λm. mt_tape c' tk (base + m)) [0..<j]
        @ [mt_tape c' tk (base + j)])
    by (simp add: upt_Suc_append)
    also have ... = ar_acc (Γ_tm M) (bl_tm M) (?fold j)
      (mt_tape c' tk (base + j))
    by simp
    finally show ?thesis by (simp add: ar_acc_def)
  qed
  have state_eq:
    mt_state ccc = (q, AR_SimRead, tk, Suc j + 2,
      buf0(tk := ?fold (Suc j)), dvec, posk)
  using bst_state by (simp add: read_cell fold_step)
  have pos_eq: mt_pos ccc tk = base + Suc j
  using bst_pos pos_cc by simp
  have other_eq: ∀ k'. k' ≠ tk → mt_pos ccc k' = mt_pos c' k'
  using bst_pos other_cc by simp
  have tape_eq: mt_tape ccc = mt_tape c' using bst_tape tape_cc by simp
  show ∃ c''. (cc, c'') ∈ R' ∧ ?P (Suc j) c''
  using bstep state_eq pos_eq tape_eq other_eq by blast
  qed
  have loop:
    ∃ c''. (c', c'') ∈ R' ^^ (block_width (Γ_tm M) - 1)
    ∧ ?P (block_width (Γ_tm M) - 1) c''
  by (rule relpow_invariant_chain
    [where P = ?P and n = block_width (Γ_tm M) - 1, OF mystep mybase])
  obtain c'' where
    chain: (c', c'') ∈ R' ^^ (block_width (Γ_tm M) - 1)
    and Pfin: ?P (block_width (Γ_tm M) - 1) c''
  using loop by blast
  have ctr: (block_width (Γ_tm M) - 1) + 2 = Suc (block_width (Γ_tm M))
  using kge2 by simp

```

show *?thesis* **using** *chain Pfin ctr* **by** (*intro exI*[**where** $x = c''$]) *simp*
qed

lemma *ar_read_bit_loop*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $\text{posk_proper}: \text{posk } tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and $\text{stg}: \text{mt_state } c' = (q, AR_SimRead, tk, 2, \text{buf0}, \text{dvec}, \text{posk})$
and $\text{buf0_valid}: \forall k. \text{buf0 } k \in \Gamma_tm \ M \cup \{bl_tm \ M\}$
and $\text{pos_base}: \text{mt_pos } c' \ tk = \text{base}$
and $\text{pad0}: \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (bl_tm \ M) \ (k_tm \ M)$
 $(AR_SimRead, tk, 2, \text{buf0}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in (\text{mttm_step } (\text{alphabet_reduce_delta } M))$
 $\wedge (\text{block_width } (\Gamma_tm \ M) - 1)$
 $\wedge \text{mt_state } c'' = (q, AR_SimRead, tk, \text{Suc } (\text{block_width } (\Gamma_tm \ M)),$
 $\text{buf0}(tk := \text{foldl } (\text{ar_acc } (\Gamma_tm \ M) \ (bl_tm \ M)) \ (\text{buf0 } tk)$
 $(\text{map } (\lambda m. \text{mt_tape } c' \ tk \ (\text{base} + m))$
 $[0..<\text{block_width } (\Gamma_tm \ M) - 1])),$
 $\text{dvec}, \text{posk})$
 $\wedge \text{mt_pos } c'' \ tk = \text{base} + (\text{block_width } (\Gamma_tm \ M) - 1)$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c'' \ k' = \text{mt_pos } c' \ k')$
by (*rule ar_read_bit_loop_gen*
 $[OF \ \text{ar_read_bit_step } vM \ qQ \ kge2 \ \text{posk_proper} \ \text{stg} \ \text{buf0_valid} \ \text{pos_base}$
 $\text{pad0 } \text{src0}]$)

The proper-cell single-tape read, non-last tape. From an *AR_SimRead* stage at bit-counter 0 with the head over a proper block $\text{base} = \text{sim_pos } p$ ($p \geq 1$, so $0 < \text{base}$) whose current cell is not *LE4* and $\text{posk } tk$ proper, the read fires the two look-back steps, the inner per-bit loop (*ar_read_bit_loop*), and the bit boundary, decoding the b -cell block at base into $\text{buf } tk$ and landing at the next tape's *AR_SimRead* ($k_succ \ tk$, counter 0). The decoded $\text{buf } tk$ is the abstract fold $\text{foldl } (\text{ar_acc } \Gamma \ bl) \ (\text{gamma_unenum } \Gamma \ bl \ 0)$ over the block's b cells – the form the caller collapses to the source symbol via $\text{foldl_ar_acc_cell_repr}$ once tape-correspondence pins those cells to cell_repr . The chain has length $b + 2$ (look-back 1, look-back 2, loop $b - 1$, boundary). The refined $\text{posk } tk$ records whether the look-back cell was *LE4* (the head sat at $\text{sim_pos } 1$).

The snoc step of a left fold over a $\text{map } f \ [0..<b]$: for $0 < b$ the fold equals one ar_acc applied to the fold over the first $b - 1$ cells and the last cell $f \ (b - 1)$. This is the boundary/finish step's last-bit accumulator collapse, shared by both proper-read consumers.

lemma *ar_acc_foldl_upt_last*:

assumes $0 < k$
shows $\text{foldl } (\text{ar_acc } \Gamma \text{ bl}) \ a \ (\text{map } f \ [0..<k])$
 $= \text{ar_acc } \Gamma \text{ bl}$
 $(\text{foldl } (\text{ar_acc } \Gamma \text{ bl}) \ a \ (\text{map } f \ [0..<k-1])) \ (f \ (k-1))$
proof –
obtain kk **where** $k = \text{Suc } kk$ **using** *assms* **by** (*cases* k) *auto*
thus *?thesis* **by** (*simp add: upt_Suc_append*)
qed

The proper-cell read prefix: the part of a proper-cell single-tape read shared by the non-last (*boundary*) and last (*finish*) variants. From an *AR_SimRead* stage at bit-counter 0 with the head over a proper block $\text{base} = \text{sim_pos } p$ ($0 < \text{base}$) whose cell is not *LE4* and $\text{posk } tk$ proper, it fires look-back 1, look-back 2, and the inner per-bit loop (*ar_read_bit_loop*), reaching bit-counter $\text{Suc } b$ with the head at $\text{base} + (b-1)$ and $\text{buf } tk$ the fold over the first $b-1$ cells. The remaining last-bit step (boundary or finish) is added by the consumer. Chain length $\text{Suc } b \ (1 + 1 + (b-1))$.

Relation-generic read proper-prefix scaffold. The two lookback steps and the bit loop are the *leaf_lb1*, *leaf_lb2*, *leaf_loop* hypotheses; union and sub versions are thin instantiations.

lemma *ar_read_proper_prefix_gen*:

fixes $M :: ('q, 'a) \text{mttm}$

and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config}$

and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config rel}$

assumes *leaf_lb1*:

$\bigwedge cc. \llbracket \text{valid_mttm } M;$

$\text{mt_state } cc = (q, \text{AR_SimRead}, tk, 0, \text{buf}, \text{dvec}, \text{posk});$

$q \in Q_tm \ M;$

$\text{posk } tk \in \{\text{AR_AtFirstProper}, \text{AR_AtFurtherProper}\};$

$\text{mt_tape } cc \ tk \ (\text{mt_pos } cc \ tk) \neq \text{LE4};$

$\text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$

$(\text{AR_SimRead}, tk, 0, \text{buf}, \text{dvec}, \text{posk});$

$\forall j \geq k_tm \ M. \text{mt_tape } cc \ j \ (\text{mt_pos } cc \ j) = \text{BLANK4};$

$\text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$

$(\text{AR_SimRead}, tk, 0, \text{buf}, \text{dvec}, \text{posk}) \rrbracket$

$\implies \exists c1. (cc, c1) \in R'$

$\wedge \text{mt_state } c1 = (q, \text{AR_SimRead}, tk, \text{Suc } 0, \text{buf}, \text{dvec}, \text{posk})$

$\wedge \text{mt_tape } c1 = \text{mt_tape } cc$

$\wedge \text{mt_pos } c1 = (\text{mt_pos } cc)(tk := \text{mt_pos } cc \ tk - 1)$

and *leaf_lb2*:

$\bigwedge cc. \llbracket \text{valid_mttm } M;$

$\text{mt_state } cc = (q, \text{AR_SimRead}, tk, \text{Suc } 0, \text{buf}, \text{dvec}, \text{posk});$

$q \in Q_tm \ M;$

$\text{posk } tk \in \{\text{AR_AtFirstProper}, \text{AR_AtFurtherProper}\};$

$2 \leq \text{block_width } (\Gamma_tm \ M);$

$\text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$

$(\text{AR_SimRead}, tk, \text{Suc } 0, \text{buf}, \text{dvec}, \text{posk});$

$\forall j \geq k_tm \ M. \text{mt_tape } cc \ j \ (\text{mt_pos } cc \ j) = \text{BLANK4};$

$$\begin{aligned}
& ar_stage_bounded (bl_tm M) (k_tm M) \\
& (AR_SimRead, tk, Suc 0, buf, dvec, posk) \mathbb{I} \\
\Rightarrow & \exists c2. (cc, c2) \in R' \\
& \wedge mt_state c2 = (q, AR_SimRead, tk, Suc (Suc 0), \\
& \quad buf(tk := gamma_unenum (\Gamma_tm M) (bl_tm M) 0), dvec, \\
& \quad posk(tk := if mt_tape cc tk (mt_pos cc tk) = LE4 \\
& \quad \quad then AR_AtFirstProper else AR_AtFurtherProper)) \\
& \wedge mt_tape c2 = mt_tape cc \\
& \wedge mt_pos c2 = (mt_pos cc)(tk := Suc (mt_pos cc tk)) \\
\text{and leaf_loop:} \\
& \wedge cc buf0' posk' base'. \mathbb{I} valid_mttm M; q \in Q_tm M; 2 \leq block_width \\
& (\Gamma_tm M); \\
& posk' tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}; \\
& mt_state cc = (q, AR_SimRead, tk, 2, buf0', dvec, posk'); \\
& \forall k. buf0' k \in \Gamma_tm M \cup \{bl_tm M\}; \\
& mt_pos cc tk = base'; \\
& \forall j \geq k_tm M. mt_tape cc j (mt_pos cc j) = BLANK4; \\
& ar_stage_bounded (bl_tm M) (k_tm M) \\
& (AR_SimRead, tk, 2, buf0', dvec, posk') \mathbb{I} \\
\Rightarrow & \exists c''. (cc, c'') \in R' \wedge (block_width (\Gamma_tm M) - 1) \\
& \wedge mt_state c'' = (q, AR_SimRead, tk, Suc (block_width (\Gamma_tm \\
& M)), \\
& \quad buf0'(tk := foldl (ar_acc (\Gamma_tm M) (bl_tm M)) (buf0' tk) \\
& \quad \quad (map (\lambda m. mt_tape cc tk (base' + m)) \\
& \quad \quad [0..<block_width (\Gamma_tm M) - 1])), \\
& \quad dvec, posk') \\
& \wedge mt_pos c'' tk = base' + (block_width (\Gamma_tm M) - 1) \\
& \wedge mt_tape c'' = mt_tape cc \\
& \wedge (\forall k'. k' \neq tk \longrightarrow mt_pos c'' k' = mt_pos cc k') \\
\text{and } vM: & valid_mttm M \\
\text{and } qQ: & q \in Q_tm M \\
\text{and } kge2: & 2 \leq block_width (\Gamma_tm M) \\
\text{and } posk_proper: & posk tk \in \{AR_AtFirstProper, AR_AtFurtherProper\} \\
\text{and } stg: & mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk) \\
\text{and } notLE: & mt_tape c' tk (mt_pos c' tk) \neq LE4 \\
\text{and } base_pos: & 0 < base \\
\text{and } pos_base: & mt_pos c' tk = base \\
\text{and } vsrc: & ar_valid_stage (\Gamma_tm M) (bl_tm M) \\
& (AR_SimRead, tk, 0, buf, dvec, posk) \\
\text{and } pad0: & \forall j \geq k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4 \\
\text{and } src0: & ar_stage_bounded (bl_tm M) (k_tm M) \\
& (AR_SimRead, tk, 0, buf, dvec, posk) \\
\text{shows } \exists c'': & (c', c'') \in R' \wedge Suc (block_width (\Gamma_tm M)) \\
& \wedge mt_state c'' = (q, AR_SimRead, tk, Suc (block_width (\Gamma_tm M)), \\
& \quad buf(tk := foldl (ar_acc (\Gamma_tm M) (bl_tm M)) \\
& \quad \quad (gamma_unenum (\Gamma_tm M) (bl_tm M) 0) \\
& \quad \quad (map (\lambda m. mt_tape c' tk (base + m)) \\
& \quad \quad [0..<block_width (\Gamma_tm M) - 1])), \\
& \quad dvec,
\end{aligned}$$

```

      posk(tk := if mt_tape c' tk (base - 1) = LE4
            then AR_AtFirstProper else AR_AtFurtherProper))
    ∧ mt_tape c'' = mt_tape c'
    ∧ mt_pos c'' = (mt_pos c')(tk := base + (block_width (Γ_tm M) -
1))
proof -
  let ?g0 = gamma_unenum (Γ_tm M) (bl_tm M) 0
  let ?cells1 = map (λm. mt_tape c' tk (base + m)) [0..<block_width (Γ_tm M)
- 1]
  let ?posk' = posk(tk := if mt_tape c' tk (base - 1) = LE4
                    then AR_AtFirstProper else AR_AtFurtherProper)
  have finG: finite (Γ_tm M) by (rule valid_mttm_finite_Gamma[OF vM])
  have ksuc: Suc (block_width (Γ_tm M) - 1) = block_width (Γ_tm M)
    using kge2 by (cases block_width (Γ_tm M)) auto
  have bsuc: Suc (base - 1) = base using base_pos by (cases base) auto
  have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
  have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
    using vsrc by (simp add: ar_valid_stage_def)
  have g0_mem: ?g0 ∈ Γ_tm M ∪ {bl_tm M} by (rule
gamma_unenum_mem[OF finG])
  obtain c1 where
    s1: (c', c1) ∈ R'
  and st1: mt_state c1 = (q, AR_SimRead, tk, Suc 0, buf, dvec, posk)
  and tp1: mt_tape c1 = mt_tape c'
  and pp1: mt_pos c1 = (mt_pos c')(tk := mt_pos c' tk - 1)
    using leaf_lb1[OF vM stg qQ posk_proper notLE vsrc pad0 src0]
    by blast
  have pos1_tk: mt_pos c1 tk = base - 1 using pp1 pos_base by simp
  have pad_c1: ∀ j ≥ k_tm M. mt_tape c1 j (mt_pos c1 j) = BLANK4
proof (intro allI impI)
    fix j assume jge: k_tm M ≤ j
    have jne: j ≠ tk using jge tk_lt by simp
    have mt_tape c1 j (mt_pos c1 j) = mt_tape c' j (mt_pos c' j)
      using tp1 pp1 jne by simp
    thus mt_tape c1 j (mt_pos c1 j) = BLANK4 using pad0 jge by simp
  qed
  have src_c1: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, Suc 0, buf, dvec, posk)
    using src0 by (simp add: ar_stage_bounded_def)
  have i1lt: Suc 0 < 2 * block_width (Γ_tm M) using kge2 by linarith
  have vsrc1: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, Suc 0, buf, dvec, posk)
    using i1lt buf_valid by (simp add: ar_valid_stage_def)
  obtain c2 where
    s2: (c1, c2) ∈ R'
  and st2: mt_state c2 = (q, AR_SimRead, tk, Suc (Suc 0),
    buf(tk := ?g0), dvec,
    posk(tk := if mt_tape c1 tk (mt_pos c1 tk) = LE4
              then AR_AtFirstProper else AR_AtFurtherProper))

```

```

and tp2: mt_tape c2 = mt_tape c1
and pp2: mt_pos c2 = (mt_pos c1)(tk := Suc (mt_pos c1 tk))
  using leaf_lb2[OF vM st1 qQ posk_proper kge2 vsrc1 pad_c1 src_c1]
  by blast
have posk2_eq: posk(tk := if mt_tape c1 tk (mt_pos c1 tk) = LE4
  then AR_AtFirstProper else AR_AtFurtherProper) = ?posk'
  by (simp add: tp1 pos1_tk)
have st2': mt_state c2 = (q, AR_SimRead, tk, 2, buf(tk := ?g0), dvec, ?posk')
  using st2 posk2_eq by (simp add: numeral_2_eq_2)
have tp2': mt_tape c2 = mt_tape c' using tp2 tp1 by simp
have pos2_tk: mt_pos c2 tk = base using pp2 pos1_tk bsuc by simp
have pad_c2:  $\forall j \geq k\_tm\ M. mt\_tape\ c2\ j\ (mt\_pos\ c2\ j) = BLANK4$ 
proof (intro allI impI)
  fix j assume jge:  $k\_tm\ M \leq j$ 
  have jne:  $j \neq tk$  using jge tk_lt by simp
  have mt_tape c2 j (mt_pos c2 j) = mt_tape c1 j (mt_pos c1 j)
    using tp2 pp2 jne by simp
  thus mt_tape c2 j (mt_pos c2 j) = BLANK4 using pad_c1 jge by simp
qed
have src_c2: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, tk, 2, buf(tk := ?g0), dvec, ?posk')
  using src0 tk_lt by (auto simp: ar_stage_bounded_def)
have posk'_proper: ?posk' tk  $\in \{AR\_AtFirstProper, AR\_AtFurtherProper\}$  by
simp
have buf0_valid:  $\forall k. (buf(tk := ?g0))\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  using buf_valid g0_mem by simp
obtain c3 where
  s3:  $(c2, c3) \in R' \wedge (block\_width\ (\Gamma\_tm\ M) - 1)$ 
and st3: mt_state c3 = (q, AR_SimRead, tk, Suc (block_width ( $\Gamma\_tm\ M$ )),
  (buf(tk := ?g0))
    (tk := foldl (ar_acc ( $\Gamma\_tm\ M$ ) (bl_tm M)) ((buf(tk := ?g0)) tk)
      (map ( $\lambda m. mt\_tape\ c2\ tk\ (base + m)$ )
        [0.. $block\_width\ (\Gamma\_tm\ M) - 1$ ])),
    dvec, ?posk')
and pp3: mt_pos c3 tk = base + (block_width ( $\Gamma\_tm\ M$ ) - 1)
and tp3: mt_tape c3 = mt_tape c2
and pp3off:  $\forall k'. k' \neq tk \longrightarrow mt\_pos\ c3\ k' = mt\_pos\ c2\ k'$ 
  using leaf_loop[OF vM qQ kge2 posk'_proper st2' buf0_valid pos2_tk
    pad_c2 src_c2]
  by blast
have st3': mt_state c3 = (q, AR_SimRead, tk, Suc (block_width ( $\Gamma\_tm\ M$ )),
  buf(tk := foldl (ar_acc ( $\Gamma\_tm\ M$ ) (bl_tm M)) ?g0 ?cells1),
  dvec, ?posk')
  using st3 by (simp add: tp2')
have tp3': mt_tape c3 = mt_tape c' using tp3 tp2' by simp
have pos3: mt_pos c3 = (mt_pos c')(tk := base + (block_width ( $\Gamma\_tm\ M$ ) -
1))
proof (rule ext)
  fix k'

```

```

  show mt_pos c3 k' = ((mt_pos c')(tk := base + (block_width (Γ_tm M) -
1))) k'
  proof (cases k' = tk)
    case True thus ?thesis using pp3 by simp
  next
    case False
    have mt_pos c3 k' = mt_pos c2 k' using pp3off False by simp
    also have ... = mt_pos c1 k' using pp2 False by simp
    also have ... = mt_pos c' k' using pp1 False by simp
    finally show ?thesis using False by simp
  qed
qed
have ch13: (c1, c3) ∈ R' ^^ Suc (block_width (Γ_tm M) - 1)
  by (rule relpow_Suc_I2[OF s2 s3])
have ch03: (c', c3) ∈ R' ^^ Suc (Suc (block_width (Γ_tm M) - 1))
  by (rule relpow_Suc_I2[OF s1 ch13])
have eqn: Suc (Suc (block_width (Γ_tm M) - 1)) = Suc (block_width (Γ_tm
M))
  using ksuc by simp
show ?thesis
proof (intro exI[where x = c3] conjI)
  show (c', c3) ∈ R' ^^ Suc (block_width (Γ_tm M))
    using ch03 by (simp only: eqn[symmetric])
  show mt_state c3 = (q, AR_SimRead, tk, Suc (block_width (Γ_tm M)),
    buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),
    dvec, ?posk')
    by (rule st3')
  show mt_tape c3 = mt_tape c' by (rule tp3')
  show mt_pos c3 = (mt_pos c')(tk := base + (block_width (Γ_tm M) - 1))
    by (rule pos3)
qed
qed

lemma ar_read_proper_prefix:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
  and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
  and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
  and base_pos: 0 < base
  and pos_base: mt_pos c' tk = base
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)

```

shows $\exists c''. (c', c'') \in (\text{mttm_step } (\text{alphabet_reduce_delta } M))$
 $\wedge \text{Suc } (\text{block_width } (\Gamma_tm M))$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimRead}, tk, \text{Suc } (\text{block_width } (\Gamma_tm M)),$
 $\text{buf}(tk := \text{foldl } (\text{ar_acc } (\Gamma_tm M) (\text{bl_tm } M))$
 $(\text{gamma_unenum } (\Gamma_tm M) (\text{bl_tm } M) 0)$
 $(\text{map } (\lambda m. \text{mt_tape } c' tk (\text{base} + m))$
 $[0..<\text{block_width } (\Gamma_tm M) - 1])),$
 $dvec,$
 $\text{posk}(tk := \text{if } \text{mt_tape } c' tk (\text{base} - 1) = \text{LE4}$
 $\text{then } \text{AR_AtFirstProper} \text{ else } \text{AR_AtFurtherProper}))$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c')(tk := \text{base} + (\text{block_width } (\Gamma_tm M) -$
 $1))$
by (*rule ar_read_proper_prefix_gen*
 $[OF \text{ ar_read_lookback1_step ar_read_lookback2_step ar_read_bit_loop}$
 $vM \text{ } qQ \text{ } kge2 \text{ } \text{posk_proper } stg \text{ notLE } \text{base_pos } \text{pos_base } \text{vsr } \text{pad0 } \text{src0}])$

The proper-cell single-tape read, non-last tape: the prefix followed by the bit-boundary step (*ar_read_bit_boundary_step*), decoding the *b*-cell block at *base* into *buf tk* (the abstract fold *foldl* (*ar_acc* Γ *bl*) (*gamma_unenum* Γ *bl* 0), collapsed to the source symbol by *foldl_ar_acc_cell_repr*) and landing at the next tape's *AR_SimRead* (*k_succ tk*, counter 0). Chain length *b* + 2.

Relation-generic read proper-step scaffold. The prefix walk and the closing bit-boundary step are the *leaf_prefix* and *leaf_boundary* hypotheses; union and sub versions are thin instantiations.

lemma *ar_read_proper_step_gen*:

fixes $M :: ('q, 'a) \text{mttm}$

and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config}$

and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config rel}$

assumes *leaf_prefix*:

$\wedge cc. \llbracket \text{valid_mttm } M; q \in Q_tm M; 2 \leq \text{block_width } (\Gamma_tm M);$

$\text{posk } tk \in \{\text{AR_AtFirstProper}, \text{AR_AtFurtherProper}\};$

$\text{mt_state } cc = (q, \text{AR_SimRead}, tk, 0, \text{buf}, dvec, \text{posk});$

$\text{mt_tape } cc tk (\text{mt_pos } cc tk) \neq \text{LE4};$

$0 < \text{base};$

$\text{mt_pos } cc tk = \text{base};$

$\text{ar_valid_stage } (\Gamma_tm M) (\text{bl_tm } M)$

$(\text{AR_SimRead}, tk, 0, \text{buf}, dvec, \text{posk});$

$\forall j \geq k_tm M. \text{mt_tape } cc j (\text{mt_pos } cc j) = \text{BLANK4};$

$\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm M)$

$(\text{AR_SimRead}, tk, 0, \text{buf}, dvec, \text{posk}) \rrbracket$

$\implies \exists c''. (cc, c'') \in R' \wedge \wedge \text{Suc } (\text{block_width } (\Gamma_tm M))$

$\wedge \text{mt_state } c'' = (q, \text{AR_SimRead}, tk, \text{Suc } (\text{block_width } (\Gamma_tm$

$M)),$

$\text{buf}(tk := \text{foldl } (\text{ar_acc } (\Gamma_tm M) (\text{bl_tm } M))$

$(\text{gamma_unenum } (\Gamma_tm M) (\text{bl_tm } M) 0)$

$(\text{map } (\lambda m. \text{mt_tape } cc tk (\text{base} + m))$

$[0..<block_width (\Gamma_tm M) - 1]]),$
 $dvec,$
 $posk(tk := if\ mt_tape\ cc\ tk\ (base - 1) = LE4$
 $\quad then\ AR_AtFirstProper\ else\ AR_AtFurtherProper))$
 $\wedge\ mt_tape\ c'' = mt_tape\ cc$
 $\wedge\ mt_pos\ c'' = (mt_pos\ cc)(tk := base + (block_width (\Gamma_tm$
 $M) - 1))$
and $leaf_boundary:$
 $\wedge_{cc\ i\ buf'\ posk'}. \llbracket valid_mttm\ M;$
 $\quad mt_state\ cc = (q, AR_SimRead, tk, i, buf', dvec, posk');$
 $\quad q \in Q_tm\ M;$
 $\quad \neg is_last_k\ M\ tk;$
 $\quad posk'\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\};$
 $\quad i = Suc\ (block_width (\Gamma_tm\ M));$
 $\quad ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $\quad (AR_SimRead, tk, i, buf', dvec, posk');$
 $\quad \forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $\quad ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $\quad (AR_SimRead, tk, i, buf', dvec, posk') \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\quad \wedge\ mt_state\ c'' = (q, AR_SimRead, k_succ\ tk, 0,$
 $\quad\quad buf'(tk := gamma_unenum (\Gamma_tm\ M)\ (bl_tm\ M)$
 $\quad\quad (2 * gamma_enum (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf'\ tk)$
 $\quad\quad + bit_value\ (mt_tape\ cc\ tk\ (mt_pos\ cc\ tk))))),$
 $\quad dvec, posk')$
 $\quad \wedge\ mt_tape\ c'' = mt_tape\ cc$
 $\quad \wedge\ mt_pos\ c'' = (mt_pos\ cc)(tk := Suc\ (mt_pos\ cc\ tk))$
and $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width (\Gamma_tm\ M)$
and $notlast: \neg is_last_k\ M\ tk$
and $posk_proper: posk\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and $stg: mt_state\ c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and $notLE: mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) \neq LE4$
and $base_pos: 0 < base$
and $pos_base: mt_pos\ c'\ tk = base$
and $vsrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $\quad (AR_SimRead, tk, 0, buf, dvec, posk)$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $\quad (AR_SimRead, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in R' \wedge (block_width (\Gamma_tm\ M) + 2)$
 $\quad \wedge\ mt_state\ c'' = (q, AR_SimRead, k_succ\ tk, 0,$
 $\quad\quad buf(tk := foldl\ (ar_acc\ (\Gamma_tm\ M)\ (bl_tm\ M))$
 $\quad\quad (gamma_unenum (\Gamma_tm\ M)\ (bl_tm\ M)\ 0)$
 $\quad\quad (map\ (\lambda m. mt_tape\ c'\ tk\ (base + m))$
 $\quad\quad [0..<block_width (\Gamma_tm\ M)])),$
 $\quad dvec,$
 $\quad posk(tk := if\ mt_tape\ c'\ tk\ (base - 1) = LE4$

```

      then AR_AtFirstProper else AR_AtFurtherProper))
    ∧ mt_tape c'' = mt_tape c'
    ∧ mt_pos c'' = (mt_pos c')(tk := base + block_width (Γ_tm M))
proof -
  let ?g0 = gamma_unenum (Γ_tm M) (bl_tm M) 0
  let ?cells1 = map (λm. mt_tape c' tk (base + m)) [0..<block_width (Γ_tm M)
- 1]
  let ?posk' = posk(tk := if mt_tape c' tk (base - 1) = LE4
    then AR_AtFirstProper else AR_AtFurtherProper)
  have finG: finite (Γ_tm M) by (rule valid_mttm_finite_Gamma[OF vM])
  have kpos: 0 < block_width (Γ_tm M) using kge2 by simp
  have ksuc: Suc (block_width (Γ_tm M) - 1) = block_width (Γ_tm M)
    using kge2 by (cases block_width (Γ_tm M)) auto
  have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
    using vsrc by (simp add: ar_valid_stage_def)
    have g0_mem: ?g0 ∈ Γ_tm M ∪ {bl_tm M} by (rule
gamma_unenum_mem[OF finG])
  obtain c3 where
    pre_chain: (c', c3) ∈ R' ^^ Suc (block_width (Γ_tm M))
  and pre_st: mt_state c3 = (q, AR_SimRead, tk, Suc (block_width (Γ_tm M)),
    buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),
    dvec, ?posk')
  and pre_tp: mt_tape c3 = mt_tape c'
  and pre_pp: mt_pos c3 = (mt_pos c')(tk := base + (block_width (Γ_tm M)
- 1))
    using leaf_prefix[OF vM qQ kge2 posk_proper stg_notLE
      base_pos pos_base vsrc pad0 src0] by blast
  have posk'_proper: ?posk' tk ∈ {AR_AtFirstProper, AR_AtFurtherProper} by
simp
  have fold3_mem:
    foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1 ∈ Γ_tm M ∪ {bl_tm M}
    using g0_mem by (rule foldl_ar_acc_mem[OF finG])
  have buf3_valid:
    ∀ k. (buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1)) k
      ∈ Γ_tm M ∪ {bl_tm M}
    using buf_valid fold3_mem by simp
  have ilt: Suc (block_width (Γ_tm M)) < 2 * block_width (Γ_tm M) using
kge2 by linarith
  have vsrc3: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, Suc (block_width (Γ_tm M)),
    buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),
    dvec, ?posk')
    using ilt buf3_valid by (simp add: ar_valid_stage_def)
  have pos3_tk: mt_pos c3 tk = base + (block_width (Γ_tm M) - 1) using
pre_pp by simp
  have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
  have pad_c3: ∀ j ≥ k_tm M. mt_tape c3 j (mt_pos c3 j) = BLANK4
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j

```

```

have jne: j ≠ tk using jge tk lt by simp
have mt_tape c3 j (mt_pos c3 j) = mt_tape c' j (mt_pos c' j)
  using pre_tp pre_pp jne by simp
thus mt_tape c3 j (mt_pos c3 j) = BLANK4 using pad0 jge by simp
qed
have src_c3: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, tk, Suc (block_width (Γ_tm M)),
   buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),
   dvec, ?posk')
  using src0 tk lt by (auto simp: ar_stage_bounded_def)
obtain c4 where
  s4: (c3, c4) ∈ R'
and st4: mt_state c4 = (q, AR_SimRead, k_succ tk, 0,
  (buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1))
  (tk := gamma_unenum (Γ_tm M) (bl_tm M)
   (2 * gamma_enum (Γ_tm M) (bl_tm M)
    ((buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0
?cells1)) tk)
   + bit_value (mt_tape c3 tk (mt_pos c3 tk))))),
  dvec, ?posk')
and tp4: mt_tape c4 = mt_tape c3
and pp4: mt_pos c4 = (mt_pos c3)(tk := Suc (mt_pos c3 tk))
  using leaf_boundary[OF vM pre_st qQ notlast posk'_proper
    refl vsrc3 pad_c3 src_c3] by blast
have read_cell4: mt_tape c3 tk (mt_pos c3 tk)
  = mt_tape c' tk (base + (block_width (Γ_tm M) - 1))
  using pre_tp pos3_tk by simp
have st4': mt_state c4 = (q, AR_SimRead, k_succ tk, 0,
  buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0
  (map (λm. mt_tape c' tk (base + m))
    [0..<block_width (Γ_tm M)])),
  dvec, ?posk')
  using st4 by (simp add: read_cell4 ar_acc_foldl_upt_last[OF kpos]
ar_acc_def)
have tape4: mt_tape c4 = mt_tape c' using tp4 pre_tp by simp
have pos4: mt_pos c4 = (mt_pos c')(tk := base + block_width (Γ_tm M))
proof (rule ext)
  fix k'
  show mt_pos c4 k' = ((mt_pos c')(tk := base + block_width (Γ_tm M))) k'
  proof (cases k' = tk)
    case True
    have mt_pos c4 tk = Suc (mt_pos c3 tk) using pp4 by simp
    also have ... = Suc (base + (block_width (Γ_tm M) - 1)) using pos3_tk
by simp
    also have ... = base + block_width (Γ_tm M) using ksuc by simp
    finally show ?thesis using True by simp
  next
    case False
    have mt_pos c4 k' = mt_pos c3 k' using pp4 False by simp

```



```

    also have ... = mt_pos c' k' using pre_pp False by simp
    finally show ?thesis using False by simp
  qed
qed
have ch04: (c', c4) ∈ R' ^ Suc (Suc (block_width (Γ_tm M)))
  by (rule relpow_Suc_I[OF pre_chain s4])
have eqn: Suc (Suc (block_width (Γ_tm M))) = block_width (Γ_tm M) + 2
by simp
show ?thesis
proof (intro exI[where x = c4] conjI)
  show (c', c4) ∈ R' ^ (block_width (Γ_tm M) + 2)
    using ch04 by (simp only: eqn[symmetric])
  show mt_state c4 = (q, AR_SimRead, k_succ tk, 0,
    buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0
      (map (λm. mt_tape c' tk (base + m))
        [0..

```

```

lemma ar_read_proper_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and notlast: ¬ is_last_k M tk
  and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
  and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
  and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
  and base_pos: 0 < base
  and pos_base: mt_pos c' tk = base
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^ (block_width (Γ_tm M) + 2)
    ∧ mt_state c'' = (q, AR_SimRead, k_succ tk, 0,
      buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M))
        (gamma_unenum (Γ_tm M) (bl_tm M) 0)
        (map (λm. mt_tape c' tk (base + m))
          [0..

```

$$\begin{aligned}
& \text{posk}(tk := \text{if } mt_tape\ c'\ tk\ (base - 1) = LE4 \\
& \quad \text{then } AR_AtFirstProper\ \text{else } AR_AtFurtherProper)) \\
& \wedge mt_tape\ c'' = mt_tape\ c' \\
& \wedge mt_pos\ c'' = (mt_pos\ c')(tk := base + block_width\ (\Gamma_tm\ M)) \\
\text{by } & (\text{rule } ar_read_proper_step_gen \\
& [OF\ ar_read_proper_prefix\ ar_read_bit_boundary_step \\
& \quad vM\ qQ\ kge2\ \text{notlast}\ posk_proper\ stg\ \text{notLE}\ base_pos\ pos_base\ vsrc \\
& \quad pad0\ src0])
\end{aligned}$$

The proper-cell single-tape read, last tape: the prefix followed by the bit-finish step ($ar_read_bit_finish_step$), decoding the b -cell block at $base$ into $buf\ tk$ exactly as the non-last variant, but transitioning to $AR_SimCompute$ with the current-tape field reset to $k_unidx\ 0$ (all tapes decoded). Chain length $b + 2$.

Relation-generic read proper-finish-step scaffold. As $ar_read_proper_step_gen$ but the closing leaf is the last-tape bit-finish (transition to $AR_SimCompute$); the prefix and finish helpers are the $leaf_prefix$ and $leaf_finish$ hypotheses.

lemma $ar_read_proper_finish_step_gen$:

fixes $M :: ('q, 'a)\ mttm$

and $c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$

and $R' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config\ rel$

assumes $leaf_prefix$:

$\wedge cc. \llbracket \text{valid_mttm}\ M; q \in Q_tm\ M; 2 \leq block_width\ (\Gamma_tm\ M);$

$\text{posk}\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\};$

$mt_state\ cc = (q, AR_SimRead, tk, 0, buf, dvec, posk);$

$mt_tape\ cc\ tk\ (mt_pos\ cc\ tk) \neq LE4;$

$0 < base;$

$mt_pos\ cc\ tk = base;$

$ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$

$(AR_SimRead, tk, 0, buf, dvec, posk);$

$\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$

$ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$

$(AR_SimRead, tk, 0, buf, dvec, posk) \rrbracket$

$\implies \exists c''. (cc, c'') \in R' \wedge \wedge Suc\ (block_width\ (\Gamma_tm\ M))$

$\wedge mt_state\ c'' = (q, AR_SimRead, tk, Suc\ (block_width\ (\Gamma_tm\ M)),$

$M)),$

$buf(tk := foldl\ (ar_acc\ (\Gamma_tm\ M)\ (bl_tm\ M))$

$(gamma_unenum\ (\Gamma_tm\ M)\ (bl_tm\ M)\ 0)$

$(map\ (\lambda m. mt_tape\ cc\ tk\ (base + m))$

$[0..<block_width\ (\Gamma_tm\ M) - 1])),$

$dvec,$

$posk(tk := \text{if } mt_tape\ cc\ tk\ (base - 1) = LE4$

$\text{then } AR_AtFirstProper\ \text{else } AR_AtFurtherProper))$

$\wedge mt_tape\ c'' = mt_tape\ cc$

$\wedge mt_pos\ c'' = (mt_pos\ cc)(tk := base + (block_width\ (\Gamma_tm\ M) - 1))$

$M) - 1))$

and $leaf_finish$:

$\wedge cc\ i\ buf'\ posk'. \llbracket valid_mttm\ M;$
 $mt_state\ cc = (q, AR_SimRead, tk, i, buf', dvec, posk');$
 $q \in Q_tm\ M;$
 $is_last_k\ M\ tk;$
 $posk'\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\};$
 $i = Suc\ (block_width\ (\Gamma_tm\ M));$
 $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, i, buf', dvec, posk');$
 $\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, i, buf', dvec, posk') \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge mt_state\ c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf'(tk := gamma_unenum\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(2 * gamma_enum\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf'\ tk)$
 $+ bit_value\ (mt_tape\ cc\ tk\ (mt_pos\ cc\ tk))),$
 $dvec, posk')$
 $\wedge mt_tape\ c'' = mt_tape\ cc$
 $\wedge mt_pos\ c'' = (mt_pos\ cc)(tk := Suc\ (mt_pos\ cc\ tk))$
and $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $last: is_last_k\ M\ tk$
and $posk_proper: posk\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and $stg: mt_state\ c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and $notLE: mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) \neq LE4$
and $base_pos: 0 < base$
and $pos_base: mt_pos\ c'\ tk = base$
and $usrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in R' \wedge (block_width\ (\Gamma_tm\ M) + 2)$
 $\wedge mt_state\ c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf(tk := foldl\ (ar_acc\ (\Gamma_tm\ M)\ (bl_tm\ M))$
 $(gamma_unenum\ (\Gamma_tm\ M)\ (bl_tm\ M)\ 0)$
 $(map\ (\lambda m. mt_tape\ c'\ tk\ (base + m))$
 $[0..<block_width\ (\Gamma_tm\ M)])),$
 $dvec,$
 $posk(tk := if\ mt_tape\ c'\ tk\ (base - 1) = LE4$
 $then\ AR_AtFirstProper\ else\ AR_AtFurtherProper))$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := base + block_width\ (\Gamma_tm\ M))$
proof –
let $?g0 = gamma_unenum\ (\Gamma_tm\ M)\ (bl_tm\ M)\ 0$
let $?cells1 = map\ (\lambda m. mt_tape\ c'\ tk\ (base + m))\ [0..<block_width\ (\Gamma_tm\ M)$
 $- 1]$
let $?posk' = posk(tk := if\ mt_tape\ c'\ tk\ (base - 1) = LE4$

```

      then AR_AtFirstProper else AR_AtFurtherProper)
have finG: finite (Γ_tm M) by (rule valid_mttm_finite_Gamma[OF vM])
have kpos: 0 < block_width (Γ_tm M) using kge2 by simp
have ksuc: Suc (block_width (Γ_tm M) - 1) = block_width (Γ_tm M)
  using kge2 by (cases block_width (Γ_tm M)) auto
have buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  using vsrc by (simp add: ar_valid_stage_def)
  have g0_mem: ?g0 ∈ Γ_tm M ∪ {bl_tm M} by (rule
gamma_unenum_mem[OF finG])
obtain c3 where
  pre_chain: (c', c3) ∈ R' ^^ Suc (block_width (Γ_tm M))
and pre_st: mt_state c3 = (q, AR_SimRead, tk, Suc (block_width (Γ_tm M)),
  buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),
  dvec, ?posk')
and pre_tp: mt_tape c3 = mt_tape c'
and pre_pp: mt_pos c3 = (mt_pos c')(tk := base + (block_width (Γ_tm M)
- 1))
  using leaf_prefix[OF vM qQ kge2 posk_proper stg notLE
    base_pos pos_base vsrc pad0 src0] by blast
have posk'_proper: ?posk' tk ∈ {AR_AtFirstProper, AR_AtFurtherProper} by
simp
have fold3_mem:
  foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1 ∈ Γ_tm M ∪ {bl_tm M}
  using g0_mem by (rule foldl_ar_acc_mem[OF finG])
have buf3_valid:
  ∀ k. (buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1)) k
    ∈ Γ_tm M ∪ {bl_tm M}
  using buf_valid fold3_mem by simp
have ilt: Suc (block_width (Γ_tm M)) < 2 * block_width (Γ_tm M) using
kge2 by linarith
have vsrc3: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimRead, tk, Suc (block_width (Γ_tm M)),
  buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),
  dvec, ?posk')
  using ilt buf3_valid by (simp add: ar_valid_stage_def)
have pos3_tk: mt_pos c3 tk = base + (block_width (Γ_tm M) - 1) using
pre_pp by simp
have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
have pad_c3: ∀ j ≥ k_tm M. mt_tape c3 j (mt_pos c3 j) = BLANK4
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using jge tk_lt by simp
  have mt_tape_c3_j (mt_pos c3 j) = mt_tape c' j (mt_pos c' j)
    using pre_tp pre_pp jne by simp
  thus mt_tape c3 j (mt_pos c3 j) = BLANK4 using pad0 jge by simp
qed
have src_c3: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, tk, Suc (block_width (Γ_tm M)),
  buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1),

```

```

      dvec, ?posk')
  using src0 tk_lt by (auto simp: ar_stage_bounded_def)
obtain c4 where
  s4: (c3, c4) ∈ R'
and st4: mt_state c4 = (q, AR_SimCompute, k_unidx 0, 0,
  (buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0 ?cells1))
  (tk := gamma_unenum (Γ_tm M) (bl_tm M)
    (2 * gamma_enum (Γ_tm M) (bl_tm M)
      ((buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0
?cells1)) tk)
      + bit_value (mt_tape c3 tk (mt_pos c3 tk))))),
  dvec, ?posk')
and tp4: mt_tape c4 = mt_tape c3
and pp4: mt_pos c4 = (mt_pos c3)(tk := Suc (mt_pos c3 tk))
  using leaf_finish[OF vM pre_st qQ last posk'_proper
    refl vsrc3 pad_c3 src_c3] by blast
have read_cell4: mt_tape c3 tk (mt_pos c3 tk)
  = mt_tape c' tk (base + (block_width (Γ_tm M) - 1))
  using pre_tp pos3_tk by simp
have st4': mt_state c4 = (q, AR_SimCompute, k_unidx 0, 0,
  buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0
    (map (λm. mt_tape c' tk (base + m))
      [0..

```

```

show ?thesis
proof (intro exI[where x = c4] conjI)
  show (c', c4) ∈ R' ^^ (block_width (Γ_tm M) + 2)
    using ch04 by (simp only: eqn[symmetric])
  show mt_state c4 = (q, AR_SimCompute, k_unidx 0, 0,
    buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M)) ?g0
      (map (λm. mt_tape c' tk (base + m))
        [0..

```

```

lemma ar_read_proper_finish_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and last: is_last_k M tk
  and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
  and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
  and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
  and base_pos: 0 < base
  and pos_base: mt_pos c' tk = base
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  and pad0: ∀j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^^ (block_width (Γ_tm M) + 2)
    ∧ mt_state c'' = (q, AR_SimCompute, k_unidx 0, 0,
      buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M))
        (gamma_unenum (Γ_tm M) (bl_tm M) 0)
        (map (λm. mt_tape c' tk (base + m))
          [0..

```

Unified single-tape read, non-last tape: dispatches the per-tape read on whether M 's head sits on the left-end marker ($p = 0$, the LE arm) or in the proper region ($p \geq 1$, the proper arm) from the per-tape correspondence facts rather than the raw cell pattern. Both arms land back at $AR_SimRead$ on the successor tape with $buf\ tk$ holding the decoded M -symbol $tM\ p$ (via $foldl_ar_acc_cell_repr$) and $posk\ tk$ re-synced exact to the head's position kind (AR_AtLE at 0, $AR_AtFirstProper$ at 1, $AR_AtFurtherProper$ beyond — the marker test $tM' (base - 1) = LE4$ coincides with $p = 1$ by the left-end discipline). Cost 1 (LE) or $b + 2$ (proper). This is the uniform per-tape step the read-phase tape walk iterates.

Relation-generic read single-tape step scaffold (non-last). The two cases ($p = 0$ le-step, $p \neq 0$ proper-step) are the $leaf_le$ and $leaf_proper$ hypotheses; union and sub versions are thin instantiations.

lemma $ar_read_tape_step_gen$:
fixes $M :: ('q, 'a) mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage) mt_config$
and $tM :: nat \Rightarrow 'a$
and $R' :: (sym4, 'q \times 'a\ ar_stage) mt_config\ rel$
assumes $leaf_le$:
 $\bigwedge cc. \llbracket valid_mttm\ M;$
 $mt_state\ cc = (q, AR_SimRead, tk, 0, buf, dvec, posk);$
 $q \in Q_tm\ M;$
 $\neg is_last_k\ M\ tk;$
 $posk\ tk = AR_AtLE;$
 $mt_tape\ cc\ tk\ (mt_pos\ cc\ tk) = LE4;$
 $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk);$
 $\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk) \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge mt_state\ c'' = (q, AR_SimRead, k_succ\ tk, 0,$
 $buf(tk := le_tm\ M), dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ cc$
 $\wedge mt_pos\ c'' = (mt_pos\ cc)(tk := Suc\ (mt_pos\ cc\ tk))$
and $leaf_proper$:
 $\bigwedge cc\ base'. \llbracket valid_mttm\ M; q \in Q_tm\ M; 2 \leq block_width\ (\Gamma_tm\ M);$
 $\neg is_last_k\ M\ tk;$
 $posk\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\};$
 $mt_state\ cc = (q, AR_SimRead, tk, 0, buf, dvec, posk);$
 $mt_tape\ cc\ tk\ (mt_pos\ cc\ tk) \neq LE4;$
 $0 < base';$
 $mt_pos\ cc\ tk = base';$
 $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk);$
 $\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$

$$\begin{aligned}
& (AR_SimRead, tk, 0, buf, dvec, posk) \parallel \\
\implies & \exists c''. (cc, c'') \in R' \wedge (block_width (\Gamma_tm M) + 2) \\
& \wedge mt_state c'' = (q, AR_SimRead, k_succ tk, 0, \\
& \quad buf(tk := foldl (ar_acc (\Gamma_tm M) (bl_tm M)) \\
& \quad \quad (gamma_unenum (\Gamma_tm M) (bl_tm M) 0) \\
& \quad \quad (map (\lambda m. mt_tape cc tk (base' + m)) \\
& \quad \quad \quad [0..<block_width (\Gamma_tm M)])), \\
& \quad dvec, \\
& \quad posk(tk := if mt_tape cc tk (base' - 1) = LE4 \\
& \quad \quad then AR_AtFirstProper else AR_AtFurtherProper)) \\
& \wedge mt_tape c'' = mt_tape cc \\
& \wedge mt_pos c'' = (mt_pos cc)(tk := base' + block_width \\
& (\Gamma_tm M)) \\
& \text{and } vM: \text{valid_mttm } M \\
& \text{and } qQ: q \in Q_tm M \\
& \text{and } kge2: 2 \leq block_width (\Gamma_tm M) \\
& \text{and } notlast: \neg is_last_k M tk \\
& \text{and } stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk) \\
& \text{and } tcorr: ar_tape_correspondence (\Gamma_tm M) (le_tm M) (bl_tm M) \\
& \quad tM (mt_tape c' tk) \\
& \text{and } ppos: mt_pos c' tk = sim_pos (block_width (\Gamma_tm M)) p \\
& \text{and } pkok: posk tk = AR_AtLE \longleftrightarrow p = 0 \\
& \text{and } proper_mem: 1 \leq p \implies tM p \in \Gamma_tm M \\
& \text{and } vsrc: ar_valid_stage (\Gamma_tm M) (bl_tm M) \\
& \quad (AR_SimRead, tk, 0, buf, dvec, posk) \\
& \text{and } pad0: \forall j \geq k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4 \\
& \text{and } src0: ar_stage_bounded (bl_tm M) (k_tm M) \\
& \quad (AR_SimRead, tk, 0, buf, dvec, posk) \\
\text{shows } & \exists c''. (c', c'') \in R' \wedge (if p = 0 then 1 else block_width (\Gamma_tm M) + 2) \\
& \wedge mt_state c'' = (q, AR_SimRead, k_succ tk, 0, \\
& \quad buf(tk := tM p), dvec, \\
& \quad posk(tk := if p = 0 then AR_AtLE \\
& \quad \quad else if p = 1 then AR_AtFirstProper \\
& \quad \quad else AR_AtFurtherProper)) \\
& \wedge mt_tape c'' = mt_tape c' \\
& \wedge mt_pos c'' = (mt_pos c')(tk := \\
& \quad if p = 0 then Suc 0 \\
& \quad else sim_pos (block_width (\Gamma_tm M)) p + block_width (\Gamma_tm \\
& M)) \\
\text{proof } & - \\
& \text{have } kpos: 0 < block_width (\Gamma_tm M) \text{ using } kge2 \text{ by simp} \\
& \text{have } finG: finite (\Gamma_tm M) \text{ by (rule valid_mttm_finite_Gamma[OF vM])} \\
& \text{have } blG: bl_tm M \in \Gamma_tm M \text{ by (rule valid_mttm_blank_in_Gamma[OF} \\
& vM])} \\
& \text{have } tape0: mt_tape c' tk 0 = LE4 \\
& \quad \text{using } tcorr \text{ by (simp add: ar_tape_correspondence_def)} \\
& \text{have } tM0: tM 0 = le_tm M \\
& \quad \text{using } tcorr \text{ by (simp add: ar_tape_correspondence_def)} \\
& \text{have } cr_wb: \bigwedge x j. j < block_width (\Gamma_tm M) \implies
\end{aligned}$$


```

      cell_repr (Γ_tm M) (bl_tm M) x ! j
      = write_bit (Γ_tm M) (bl_tm M) x j
proof -
  fix x :: 'a and j
  assume jk: j < block_width (Γ_tm M)
  show cell_repr (Γ_tm M) (bl_tm M) x ! j
    = write_bit (Γ_tm M) (bl_tm M) x j
    using jk by (cases x = bl_tm M)
    (simp_all add: cell_repr_def write_bit_def length_encode_symbol)
qed
show ?thesis
proof (cases p = 0)
  case True
  have posk_le: posk tk = AR_AtLE using pkok True by simp
  have pos0: mt_pos c' tk = 0 using ppos True by (simp add: sim_pos_def)
  have aLE: mt_tape c' tk (mt_pos c' tk) = LE4 using tape0 pos0 by simp
  obtain c'' where
    step: (c', c'') ∈ R'
    and st: mt_state c'' = (q, AR_SimRead, k_succ tk, 0,
      buf(tk := le_tm M), dvec, posk)
    and tp: mt_tape c'' = mt_tape c'
    and ps: mt_pos c'' = (mt_pos c')(tk := Suc (mt_pos c' tk))
    using leaf_le[OF vM stg qQ notlast posk_le aLE vsrc pad0 src0] by blast
  have poskid: posk(tk := AR_AtLE) = posk
    using posk_le by (simp add: fun_upd_idem)
  show ?thesis
  proof (intro exI[where x = c''] conjI)
    show (c', c'') ∈ R'
      ^ (if p = 0 then 1 else block_width (Γ_tm M) + 2)
    using step True by simp
    show mt_state c'' = (q, AR_SimRead, k_succ tk, 0, buf(tk := tM p), dvec,
      posk(tk := if p = 0 then AR_AtLE
        else if p = 1 then AR_AtFirstProper
        else AR_AtFurtherProper))
      using st tM0 True poskid by simp
    show mt_tape c'' = mt_tape c' by (rule tp)
    show mt_pos c'' = (mt_pos c')(tk :=
      if p = 0 then Suc 0
      else sim_pos (block_width (Γ_tm M)) p + block_width (Γ_tm M))
      using ps pos0 True by simp
  qed
next
  case False
  hence pge1: 1 ≤ p by simp
  have bpos: 0 < sim_pos (block_width (Γ_tm M)) p
    using pge1 by (simp add: sim_pos_def)
  have posk_ne: posk tk ≠ AR_AtLE using pkok False by simp
  have posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
    using posk_ne by (cases posk tk) auto

```

```

have corr:  $\forall j < \text{block\_width } (\Gamma\_tm\ M).$ 
   $\text{mt\_tape } c' \text{ tk } (\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p + j)$ 
   $= \text{cell\_repr } (\Gamma\_tm\ M) (\text{bl\_tm } M) (tM\ p) ! j$ 
  using tcorr pge1 unfolding ar_tape_correspondence_def by blast
have cell0:  $\text{mt\_tape } c' \text{ tk } (\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p)$ 
   $= \text{cell\_repr } (\Gamma\_tm\ M) (\text{bl\_tm } M) (tM\ p) ! 0$ 
  using corr[rule_format, OF kpos] by simp
have notLE:  $\text{mt\_tape } c' \text{ tk } (\text{mt\_pos } c' \text{ tk}) \neq LE4$ 
proof -
  have  $\text{mt\_tape } c' \text{ tk } (\text{mt\_pos } c' \text{ tk})$ 
     $= \text{write\_bit } (\Gamma\_tm\ M) (\text{bl\_tm } M) (tM\ p)\ 0$ 
    using ppos cell0 cr_wb[OF kpos] by simp
  thus ?thesis using write_bit_not_LE4[OF kpos] by simp
qed
obtain  $c''$  where
  step:  $(c', c'') \in R' \wedge (\text{block\_width } (\Gamma\_tm\ M) + 2)$ 
  and st:  $\text{mt\_state } c'' = (q, AR\_SimRead, k\_succ\ tk, 0,$ 
     $\text{buf}(tk := \text{foldl } (\text{ar\_acc } (\Gamma\_tm\ M) (\text{bl\_tm } M))$ 
     $(\text{gamma\_unenum } (\Gamma\_tm\ M) (\text{bl\_tm } M)\ 0)$ 
     $(\text{map } (\lambda m. \text{mt\_tape } c' \text{ tk}$ 
     $(\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p + m))$ 
     $[0..<\text{block\_width } (\Gamma\_tm\ M)]))$ ,
    dvec,
     $\text{posk}(tk := \text{if } \text{mt\_tape } c' \text{ tk}$ 
     $(\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p - 1) = LE4$ 
     $\text{then } AR\_AtFirstProper \text{ else } AR\_AtFurtherProper))$ 
  and tp:  $\text{mt\_tape } c'' = \text{mt\_tape } c'$ 
  and ps:  $\text{mt\_pos } c'' = (\text{mt\_pos } c')(tk :=$ 
     $\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p + \text{block\_width } (\Gamma\_tm\ M))$ 
  using leaf_proper[OF vM qQ kge2 notlast posk_proper stg
    notLE bpos ppos vsrc pad0 src0] by blast
have len_cr:  $\text{length } (\text{cell\_repr } (\Gamma\_tm\ M) (\text{bl\_tm } M) (tM\ p))$ 
   $= \text{block\_width } (\Gamma\_tm\ M)$ 
  by (simp add: cell_repr_def length_encode_symbol)
have cells_eq:  $\text{map } (\lambda m. \text{mt\_tape } c' \text{ tk}$ 
   $(\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p + m))$ 
   $[0..<\text{block\_width } (\Gamma\_tm\ M)]$ 
   $= \text{cell\_repr } (\Gamma\_tm\ M) (\text{bl\_tm } M) (tM\ p)$ 
proof (rule nth_equalityI)
  show  $\text{length } (\text{map } (\lambda m. \text{mt\_tape } c' \text{ tk}$ 
     $(\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p + m))$ 
     $[0..<\text{block\_width } (\Gamma\_tm\ M)])$ 
     $= \text{length } (\text{cell\_repr } (\Gamma\_tm\ M) (\text{bl\_tm } M) (tM\ p))$ 
  using len_cr by simp
next
fix j
assume  $j < \text{length } (\text{map } (\lambda m. \text{mt\_tape } c' \text{ tk}$ 
   $(\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p + m))$ 
   $[0..<\text{block\_width } (\Gamma\_tm\ M)])$ 

```

```

hence  $jk$ :  $j < \text{block\_width } (\Gamma\_tm\ M)$  by simp
show  $\text{map } (\lambda m. \text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ p + m))$ 
 $[0..<\text{block\_width } (\Gamma\_tm\ M)] ! j$ 
 $= \text{cell\_repr } (\Gamma\_tm\ M) (bl\_tm\ M) (tM\ p) ! j$ 
using corr[rule\_format, OF jk]  $jk$  by simp
qed
have decoded:  $\text{foldl } (ar\_acc\ (\Gamma\_tm\ M)\ (bl\_tm\ M))$ 
 $(\text{gamma\_unenum } (\Gamma\_tm\ M)\ (bl\_tm\ M)\ 0)$ 
 $(\text{map } (\lambda m. \text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ p + m))$ 
 $[0..<\text{block\_width } (\Gamma\_tm\ M)]])$ 
 $= tM\ p$ 
using cells_eq foldl ar_acc cell_repr[OF finG blG proper_mem[OF pge1]]
by simp
have resync:  $(\text{if } \text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ p - 1) =$ 
 $LE4$ 
 $\text{then } AR\_AtFirstProper \text{ else } AR\_AtFurtherProper)$ 
 $= (\text{if } p = 1 \text{ then } AR\_AtFirstProper \text{ else } AR\_AtFurtherProper)$ 
proof (cases  $p = 1$ )
  case True
    have  $\text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ p - 1) = LE4$ 
    using True tape0 by (simp add: sim_pos_def)
    thus ?thesis using True by simp
  next
    case False
      with pge1 have pge2:  $2 \leq p$  by simp
      then obtain  $p2$  where  $pp$ :  $p = \text{Suc } (\text{Suc } p2)$ 
      using Suc_le_D by (metis Suc_1 le_Suc_ex add_2_eq_Suc)
      obtain  $k2$  where  $kk$ :  $\text{block\_width } (\Gamma\_tm\ M) = \text{Suc } k2$ 
      using kpos by (cases block_width (\Gamma_tm M)) auto
      have  $idx\_eq$ :  $\text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ p - 1$ 
 $= \text{sim\_pos } (\text{block\_width } (\Gamma\_tm\ M))\ (p - 1) + (\text{block\_width}$ 
 $(\Gamma\_tm\ M) - 1)$ 
      by (simp add: sim_pos_def pp kk algebra_simps)
      have  $pm1$ :  $1 \leq p - 1$  using pge2 by simp
      have  $corr2$ :  $\forall j < \text{block\_width } (\Gamma\_tm\ M).$ 
 $\text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ (p - 1) + j)$ 
 $= \text{cell\_repr } (\Gamma\_tm\ M) (bl\_tm\ M) (tM\ (p - 1)) ! j$ 
      using tcrr pm1 unfolding ar_tape_correspondence_def by blast
      have  $km1$ :  $\text{block\_width } (\Gamma\_tm\ M) - 1 < \text{block\_width } (\Gamma\_tm\ M)$  using kpos
by simp
      have  $\text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ p - 1)$ 
 $= \text{write\_bit } (\Gamma\_tm\ M) (bl\_tm\ M) (tM\ (p - 1))$ 
 $(\text{block\_width } (\Gamma\_tm\ M) - 1)$ 
      using corr2[rule\_format, OF km1]  $idx\_eq\ cr\_wb$ [OF km1] by simp
      hence  $\text{mt\_tape } c' \text{ tk } (sim\_pos (\text{block\_width } (\Gamma\_tm\ M))\ p - 1) \neq LE4$ 
      using write_bit_not_LE4[OF km1] by simp
      thus ?thesis using False by simp

```

```

qed
show ?thesis
proof (intro exI[where x = c'] conjI)
  show (c', c'') ∈ R'
    ^^ (if p = 0 then 1 else block_width (Γ_tm M) + 2)
    using step False by simp
  show mt_state c'' = (q, AR_SimRead, k_succ tk, 0, buf(tk := tM p), dvec,
    posk(tk := if p = 0 then AR_AtLE
      else if p = 1 then AR_AtFirstProper
      else AR_AtFurtherProper))
  using st decoded resync False by simp
  show mt_tape c'' = mt_tape c' by (rule tp)
  show mt_pos c'' = (mt_pos c')(tk :=
    if p = 0 then Suc 0
    else sim_pos (block_width (Γ_tm M)) p + block_width (Γ_tm M))
  using ps False by simp
qed
qed
qed

```

```

lemma ar_read_tape_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  and tM :: nat ⇒ 'a
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and notlast: ¬ is_last_k M tk
  and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
  and tcorr: ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
    tM (mt_tape c' tk)
  and ppos: mt_pos c' tk = sim_pos (block_width (Γ_tm M)) p
  and pkok: posk tk = AR_AtLE ↔ p = 0
  and proper_mem: 1 ≤ p ⇒ tM p ∈ Γ_tm M
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^^ (if p = 0 then 1 else block_width (Γ_tm M) + 2)
    ∧ mt_state c'' = (q, AR_SimRead, k_succ tk, 0,
      buf(tk := tM p), dvec,
      posk(tk := if p = 0 then AR_AtLE
        else if p = 1 then AR_AtFirstProper
        else AR_AtFurtherProper))
    ∧ mt_tape c'' = mt_tape c'
    ∧ mt_pos c'' = (mt_pos c')(tk :=
      if p = 0 then Suc 0

```

$$\text{else sim_pos (block_width } (\Gamma_tm\ M))\ p + \text{block_width } (\Gamma_tm\ M))$$
by (*rule* *ar_read_tape_step_gen*

$$[OF\ ar_read_le_step\ ar_read_proper_step$$

$$vM\ qQ\ kge2\ notlast\ stg\ tcorr\ ppos\ pkok\ proper_mem\ vsrc$$

$$pad0\ src0])$$

Unified single-tape read, last tape: as *ar_read_tape_step* but *tk* is the last tape, so both arms transition to *AR_SimCompute* with the current-tape field reset to *k_unidx 0* (all tapes decoded, the read phase complete). The *buf tk* decode and *posk tk* re-sync are identical to the non-last variant; only the hand-off target differs.

Relation-generic read single-tape finish-step scaffold (last tape). As *ar_read_tape_step_gen* but both cases use the last-tape finish leaves (transition to *AR_SimCompute*); the *le*-finish and *proper*-finish helpers are the *leaf_le* and *leaf_proper* hypotheses.

lemma *ar_read_tape_finish_step_gen*:

fixes *M* :: ('q, 'a) *mttm*

and *c'* :: (*sym4*, 'q × 'a *ar_stage*) *mt_config*

and *tM* :: *nat* ⇒ 'a

and *R'* :: (*sym4*, 'q × 'a *ar_stage*) *mt_config rel*

assumes *leaf_le*:

$$\wedge cc. \llbracket \text{valid_mttm } M;$$

$$\text{mt_state } cc = (q, AR_SimRead, tk, 0, buf, dvec, posk);$$

$$q \in Q_tm\ M;$$

$$is_last_k\ M\ tk;$$

$$posk\ tk = AR_AtLE;$$

$$\text{mt_tape } cc\ tk\ (\text{mt_pos } cc\ tk) = LE4;$$

$$ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$$

$$(AR_SimRead, tk, 0, buf, dvec, posk);$$

$$\forall j \geq k_tm\ M. \text{mt_tape } cc\ j\ (\text{mt_pos } cc\ j) = BLANK4;$$

$$ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$$

$$(AR_SimRead, tk, 0, buf, dvec, posk) \rrbracket$$

$$\implies \exists c''. (cc, c'') \in R'$$

$$\wedge \text{mt_state } c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$$

$$buf(tk := le_tm\ M), dvec, posk)$$

$$\wedge \text{mt_tape } c'' = \text{mt_tape } cc$$

$$\wedge \text{mt_pos } c'' = (\text{mt_pos } cc)(tk := Suc\ (\text{mt_pos } cc\ tk))$$

and *leaf_proper*:

$$\wedge cc\ base'. \llbracket \text{valid_mttm } M; q \in Q_tm\ M; 2 \leq \text{block_width } (\Gamma_tm\ M);$$

$$is_last_k\ M\ tk;$$

$$posk\ tk \in \{AR_AtFirstProper, AR_AtFurtherProper\};$$

$$\text{mt_state } cc = (q, AR_SimRead, tk, 0, buf, dvec, posk);$$

$$\text{mt_tape } cc\ tk\ (\text{mt_pos } cc\ tk) \neq LE4;$$

$$0 < base';$$

$$\text{mt_pos } cc\ tk = base';$$

$$ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$$

$$(AR_SimRead, tk, 0, buf, dvec, posk);$$

$\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk) \parallel$
 $\implies \exists c''. (cc, c'') \in R' \wedge (block_width\ (\Gamma_tm\ M) + 2)$
 $\wedge mt_state\ c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf(tk := foldl\ (ar_acc\ (\Gamma_tm\ M)\ (bl_tm\ M))$
 $(gamma_unenum\ (\Gamma_tm\ M)\ (bl_tm\ M)\ 0)$
 $(map\ (\lambda m. mt_tape\ cc\ tk\ (base' + m))$
 $[0..<block_width\ (\Gamma_tm\ M)])),$
 $dvec,$
 $posk(tk := if\ mt_tape\ cc\ tk\ (base' - 1) = LE4$
 $then\ AR_AtFirstProper\ else\ AR_AtFurtherProper))$
 $\wedge mt_tape\ c'' = mt_tape\ cc$
 $\wedge mt_pos\ c'' = (mt_pos\ cc)(tk := base' + block_width$
 $(\Gamma_tm\ M))$
and vM : $valid_mttm\ M$
and qQ : $q \in Q_tm\ M$
and $kge2$: $2 \leq block_width\ (\Gamma_tm\ M)$
and $last$: $is_last_k\ M\ tk$
and stg : $mt_state\ c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and $tcrr$: $ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$
 $tM\ (mt_tape\ c'\ tk)$
and $ppos$: $mt_pos\ c'\ tk = sim_pos\ (block_width\ (\Gamma_tm\ M))\ p$
and $pkok$: $posk\ tk = AR_AtLE \iff p = 0$
and $proper_mem$: $1 \leq p \implies tM\ p \in \Gamma_tm\ M$
and $usrc$: $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
and $pad0$: $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0$: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in R' \wedge (if\ p = 0\ then\ 1\ else\ block_width\ (\Gamma_tm\ M) + 2)$
 $\wedge mt_state\ c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf(tk := tM\ p), dvec,$
 $posk(tk := if\ p = 0\ then\ AR_AtLE$
 $else\ if\ p = 1\ then\ AR_AtFirstProper$
 $else\ AR_AtFurtherProper))$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk :=$
 $if\ p = 0\ then\ Suc\ 0$
 $else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ p + block_width\ (\Gamma_tm$
 $M))$
proof –
have $kpos$: $0 < block_width\ (\Gamma_tm\ M)$ **using** $kge2$ **by** $simp$
have $finG$: $finite\ (\Gamma_tm\ M)$ **by** $(rule\ valid_mttm_finite_Gamma[OF\ vM])$
have blG : $bl_tm\ M \in \Gamma_tm\ M$ **by** $(rule\ valid_mttm_blank_in_Gamma[OF\ vM])$
have $tape0$: $mt_tape\ c'\ tk\ 0 = LE4$
using $tcrr$ **by** $(simp\ add: ar_tape_correspondence_def)$
have $tM0$: $tM\ 0 = le_tm\ M$

```

    using tcorr by (simp add: ar_tape_correspondence_def)
  have cr_wb:  $\bigwedge x j. j < \text{block\_width } (\Gamma\_tm M) \implies$ 
    cell_repr  $(\Gamma\_tm M)$   $(bl\_tm M)$   $x ! j$ 
    = write_bit  $(\Gamma\_tm M)$   $(bl\_tm M)$   $x j$ 
  proof -
    fix x :: 'a and j
    assume jk:  $j < \text{block\_width } (\Gamma\_tm M)$ 
    show cell_repr  $(\Gamma\_tm M)$   $(bl\_tm M)$   $x ! j$ 
      = write_bit  $(\Gamma\_tm M)$   $(bl\_tm M)$   $x j$ 
    using jk by (cases x = bl_tm M)
      (simp_all add: cell_repr_def write_bit_def length_encode_symbol)
  qed
  show ?thesis
  proof (cases p = 0)
    case True
    have posk_le:  $\text{posk } tk = AR\_AtLE$  using pkok True by simp
    have pos0:  $\text{mt\_pos } c' tk = 0$  using ppos True by (simp add: sim_pos_def)
    have aLE:  $\text{mt\_tape } c' tk (\text{mt\_pos } c' tk) = LE4$  using tape0 pos0 by simp
    obtain c'' where
      step:  $(c', c'') \in R'$ 
      and st:  $\text{mt\_state } c'' = (q, AR\_SimCompute, k\_unidx 0, 0,$ 
        buf( $tk := le\_tm M$ ), dvec, posk)
      and tp:  $\text{mt\_tape } c'' = \text{mt\_tape } c'$ 
      and ps:  $\text{mt\_pos } c'' = (\text{mt\_pos } c')(tk := Suc (\text{mt\_pos } c' tk))$ 
      using leaf_le[OF vM stg qQ last posk_le aLE vsrc pad0 src0]
      by blast
    have poskid:  $\text{posk}(tk := AR\_AtLE) = \text{posk}$ 
      using posk_le by (simp add: fun_upd_idem)
    show ?thesis
    proof (intro exI[where x = c''] conjI)
      show  $(c', c'') \in R'$ 
        ^ (if p = 0 then 1 else block_width  $(\Gamma\_tm M) + 2$ )
      using step True by simp
      show  $\text{mt\_state } c'' = (q, AR\_SimCompute, k\_unidx 0, 0, \text{buf}(tk := tM p),$ 
        dvec,
        posk( $tk := \text{if } p = 0 \text{ then } AR\_AtLE$ 
          else if  $p = 1$  then  $AR\_AtFirstProper$ 
          else  $AR\_AtFurtherProper$ )))
      using st tM0 True poskid by simp
      show  $\text{mt\_tape } c'' = \text{mt\_tape } c'$  by (rule tp)
      show  $\text{mt\_pos } c'' = (\text{mt\_pos } c')(tk :=$ 
        if p = 0 then  $Suc 0$ 
        else  $\text{sim\_pos } (\text{block\_width } (\Gamma\_tm M)) p + \text{block\_width } (\Gamma\_tm M))$ 
      using ps pos0 True by simp
    qed
  next
    case False
    hence pge1:  $1 \leq p$  by simp
    have bpos:  $0 < \text{sim\_pos } (\text{block\_width } (\Gamma\_tm M)) p$ 

```

```

    using pge1 by (simp add: sim_pos_def)
  have posk_ne: posk tk ≠ AR_AtLE using pkok False by simp
  have posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
    using posk_ne by (cases posk tk) auto
  have corr: ∀ j < block_width (Γ_tm M).
    mt_tape c' tk (sim_pos (block_width (Γ_tm M)) p + j)
      = cell_repr (Γ_tm M) (bl_tm M) (tM p) ! j
    using tcorr pge1 unfolding ar_tape_correspondence_def by blast
  have cell0: mt_tape c' tk (sim_pos (block_width (Γ_tm M)) p)
    = cell_repr (Γ_tm M) (bl_tm M) (tM p) ! 0
    using corr[rule_format, OF kpos] by simp
  have notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
  proof -
    have mt_tape c' tk (mt_pos c' tk)
      = write_bit (Γ_tm M) (bl_tm M) (tM p) 0
      using ppos cell0 cr_wb[OF kpos] by simp
    thus ?thesis using write_bit_not_LE4[OF kpos] by simp
  qed
  obtain c'' where
    step: (c', c'') ∈ R' ^^ (block_width (Γ_tm M) + 2)
  and st: mt_state c'' = (q, AR_SimCompute, k_unidx 0, 0,
    buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M))
      (gamma_unenum (Γ_tm M) (bl_tm M) 0)
      (map (λm. mt_tape c' tk
        (sim_pos (block_width (Γ_tm M)) p + m))
        [0..

```



```

fix j
assume j < length (map (λm. mt_tape c' tk
  (sim_pos (block_width (Γ_tm M)) p + m))
  [0..<block_width (Γ_tm M)])
hence jk: j < block_width (Γ_tm M) by simp
show map (λm. mt_tape c' tk
  (sim_pos (block_width (Γ_tm M)) p + m))
  [0..<block_width (Γ_tm M)] ! j
  = cell_repr (Γ_tm M) (bl_tm M) (tM p) ! j
  using corr[rule_format, OF jk] jk by simp
qed
have decoded: foldl (ar_acc (Γ_tm M) (bl_tm M))
  (gamma_unenum (Γ_tm M) (bl_tm M) 0)
  (map (λm. mt_tape c' tk
    (sim_pos (block_width (Γ_tm M)) p + m))
    [0..<block_width (Γ_tm M)])
  = tM p
  using cells_eq foldl_ar_acc_cell_repr[OF finG blG proper_mem[OF pge1]]
  by simp
have resync: (if mt_tape c' tk (sim_pos (block_width (Γ_tm M)) p - 1) =
  LE4
    then AR_AtFirstProper else AR_AtFurtherProper)
  = (if p = 1 then AR_AtFirstProper else AR_AtFurtherProper)
proof (cases p = 1)
case True
  have mt_tape c' tk (sim_pos (block_width (Γ_tm M)) p - 1) = LE4
    using True tape0 by (simp add: sim_pos_def)
  thus ?thesis using True by simp
next
case False
  with pge1 have pge2: 2 ≤ p by simp
  then obtain p2 where pp: p = Suc (Suc p2)
    using Suc_le_D by (metis Suc_1 le_Suc_ex add_2_eq_Suc)
  obtain k2 where kk: block_width (Γ_tm M) = Suc k2
    using kpos by (cases block_width (Γ_tm M)) auto
  have idx_eq: sim_pos (block_width (Γ_tm M)) p - 1
    = sim_pos (block_width (Γ_tm M)) (p - 1) + (block_width
  (Γ_tm M) - 1)
    by (simp add: sim_pos_def pp kk algebra_simps)
  have pm1: 1 ≤ p - 1 using pge2 by simp
  have corr2: ∀ j < block_width (Γ_tm M).
    mt_tape c' tk (sim_pos (block_width (Γ_tm M)) (p - 1) + j)
    = cell_repr (Γ_tm M) (bl_tm M) (tM (p - 1)) ! j
    using tcorr pm1 unfolding ar_tape_correspondence_def by blast
  have km1: block_width (Γ_tm M) - 1 < block_width (Γ_tm M) using kpos
  by simp
  have mt_tape c' tk (sim_pos (block_width (Γ_tm M)) p - 1)
    = write_bit (Γ_tm M) (bl_tm M) (tM (p - 1))
      (block_width (Γ_tm M) - 1)

```

```

    using corr2[rule_format, OF km1] idx_eq cr_wb[OF km1] by simp
  hence mt_tape c' tk (sim_pos (block_width (Γ_tm M)) p - 1) ≠ LE4
    using write_bit_not_LE4[OF km1] by simp
  thus ?thesis using False by simp
qed
show ?thesis
proof (intro exI[where x = c''] conjI)
  show (c', c'') ∈ R'
    ^ (if p = 0 then 1 else block_width (Γ_tm M) + 2)
    using step False by simp
  show mt_state c'' = (q, AR_SimCompute, k_unidx 0, 0, buf(tk := tM p),
dvec,
    posk(tk := if p = 0 then AR_AtLE
      else if p = 1 then AR_AtFirstProper
      else AR_AtFurtherProper))
    using st_decoded_resync False by simp
  show mt_tape c'' = mt_tape c' by (rule tp)
  show mt_pos c'' = (mt_pos c')(tk :=
    if p = 0 then Suc 0
    else sim_pos (block_width (Γ_tm M)) p + block_width (Γ_tm M))
    using ps False by simp
qed
qed
qed
lemma ar_read_tape_finish_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  and tM :: nat ⇒ 'a
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and last: is_last_k M tk
  and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
  and tcorr: ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
    tM (mt_tape c' tk)
  and ppos: mt_pos c' tk = sim_pos (block_width (Γ_tm M)) p
  and pkok: posk tk = AR_AtLE ⟷ p = 0
  and proper_mem: 1 ≤ p ⟹ tM p ∈ Γ_tm M
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, tk, 0, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^ (if p = 0 then 1 else block_width (Γ_tm M) + 2)
    ∧ mt_state c'' = (q, AR_SimCompute, k_unidx 0, 0,
      buf(tk := tM p), dvec,
      posk(tk := if p = 0 then AR_AtLE

```

```

      else if  $p = 1$  then  $AR\_AtFirstProper$ 
      else  $AR\_AtFurtherProper$ )
 $\wedge mt\_tape\ c'' = mt\_tape\ c'$ 
 $\wedge mt\_pos\ c'' = (mt\_pos\ c')(tk :=$ 
  if  $p = 0$  then  $Suc\ 0$ 
  else  $sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ p + block\_width\ (\Gamma\_tm$ 
 $M))$ 
by ( $rule\ ar\_read\_tape\_finish\_step\_gen$ 
  [ $OF\ ar\_read\_le\_finish\_step\ ar\_read\_proper\_finish\_step$ 
   $vM\ qQ\ kge2\ last\ stg\ tcorr\ ppos\ pkok\ proper\_mem\ vsrc$ 
   $pad0\ src0]$ )

```

The read-phase prefix walk: starting from the read boundary (current-tape field $k_unidx\ 0$, bit-counter 0), iterate the unified non-last per-tape read $ar_read_tape_step$ over the first j tapes of the enumeration ($j \leq k_tm\ M - 1$, so every tape touched is non-last), landing back at $AR_SimRead$ on tape $k_unidx\ j$. The walk descriptor splits on $k_idx\ k < j$: tapes already visited carry the decoded M -symbol in buf , the exact re-synced $posk$, and the block-end head position; the rest retain their boundary values. Variable per-tape cost (1 or $b + 2$), aggregate bounded by $j \cdot (b + 2)$. Custom induction on j ; the inductive step's function-update bookkeeping rests on k_idx injectivity ($k \neq k_unidx\ j \implies k_idx\ k \neq j$).

Relation-generic read prefix walk scaffold. The single per-tape helper ($ar_read_tape_step$) is the *leaf* hypothesis, quantified over the per-tape config, tape index, buffer, position-kind, tape-content function, and position; union and sub versions are thin instantiations.

lemma $ar_read_prefix_gen$:

fixes $M :: ('q, 'a)\ mttm$

and $cM :: ('a, 'q)\ mt_config$

and $c0 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$

and $R' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config\ rel$

assumes $leaf$:

$\bigwedge cc\ tk'\ buf'\ posk'\ tM'\ p'.$

$\llbracket valid_mttm\ M; q \in Q_tm\ M; 2 \leq block_width\ (\Gamma_tm\ M);$

$\neg is_last_k\ M\ tk';$

$mt_state\ cc = (q, AR_SimRead, tk', 0, buf', dvec, posk');$

$ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$

$tM'\ (mt_tape\ cc\ tk');$

$mt_pos\ cc\ tk' = sim_pos\ (block_width\ (\Gamma_tm\ M))\ p';$

$posk'\ tk' = AR_AtLE \longleftrightarrow p' = 0;$

$1 \leq p' \implies tM'\ p' \in \Gamma_tm\ M;$

$ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$

$(AR_SimRead, tk', 0, buf', dvec, posk');$

$\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$

$ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$

$(AR_SimRead, tk', 0, buf', dvec, posk') \rrbracket$

$\implies \exists c''. (cc, c'') \in R' \wedge \wedge (if\ p' = 0\ then\ 1\ else\ block_width\ (\Gamma_tm\ M))$

+ 2)

$$\begin{aligned}
& \wedge \text{mt_state } c'' = (q, \text{AR_SimRead}, k_succ \text{ tk}', 0, \\
& \quad \text{buf}'(\text{tk}' := tM' p'), \text{dvec}, \\
& \quad \text{posk}'(\text{tk}' := \text{if } p' = 0 \text{ then AR_AtLE} \\
& \quad \quad \text{else if } p' = 1 \text{ then AR_AtFirstProper} \\
& \quad \quad \text{else AR_AtFurtherProper})) \\
& \wedge \text{mt_tape } c'' = \text{mt_tape } cc \\
& \wedge \text{mt_pos } c'' = (\text{mt_pos } cc)(\text{tk}' := \\
& \quad \text{if } p' = 0 \text{ then Suc } 0 \\
& \quad \text{else sim_pos (block_width } (\Gamma_tm \text{ } M)) \text{ } p' + \text{block_width} \\
& (\Gamma_tm \text{ } M)) \\
& \text{and } vM: \text{valid_mttm } M \\
& \text{and } qQ: q \in \bar{Q}_tm \text{ } M \\
& \text{and } kge2: 2 \leq \text{block_width } (\Gamma_tm \text{ } M) \\
& \text{and } stg0: \text{mt_state } c0 = (q, \text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0}) \\
& \text{and } tcorr: \forall k < k_tm \text{ } M. \text{ar_tape_correspondence } (\Gamma_tm \text{ } M) (\text{le_tm } M) \\
& (\text{bl_tm } M) \\
& \quad (\text{mt_tape } cM \text{ } k) (\text{mt_tape } c0 \text{ } k) \\
& \text{and } ppos: \bigwedge k. \text{mt_pos } c0 \text{ } k = \text{sim_pos (block_width } (\Gamma_tm \text{ } M)) (\text{mt_pos} \\
& cM \text{ } k) \\
& \text{and } pkok: \bigwedge k. \text{posk0 } k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM \text{ } k = 0 \\
& \text{and } tapeG: \bigwedge k. \text{mt_tape } cM \text{ } k (\text{mt_pos } cM \text{ } k) \in \Gamma_tm \text{ } M \\
& \text{and } bufG: \bigwedge k. \text{buf0 } k \in \Gamma_tm \text{ } M \cup \{\text{bl_tm } M\} \\
& \text{and } pad0: \forall j \geq k_tm \text{ } M. \text{mt_tape } c0 \text{ } j (\text{mt_pos } c0 \text{ } j) = \text{BLANK4} \\
& \text{and } src0: \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \text{ } M) \\
& \quad (\text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0}) \\
& \text{shows } j \leq k_tm \text{ } M - 1 \implies \\
& (\exists c \text{ m. } (c0, c) \in R' \wedge \wedge m \\
& \quad \wedge m \leq j * (\text{block_width } (\Gamma_tm \text{ } M) + 2) \\
& \quad \wedge \text{mt_state } c = (q, \text{AR_SimRead}, k_unidx \text{ } j, 0, \\
& \quad \quad (\lambda k. \text{if } k_idx \text{ } k < j \text{ then } \text{mt_tape } cM \text{ } k (\text{mt_pos } cM \text{ } k) \text{ else } \text{buf0 } k), \\
& \quad \quad \text{dvec}, \\
& \quad \quad (\lambda k. \text{if } k_idx \text{ } k < j \\
& \quad \quad \quad \text{then (if } \text{mt_pos } cM \text{ } k = 0 \text{ then AR_AtLE} \\
& \quad \quad \quad \quad \text{else if } \text{mt_pos } cM \text{ } k = 1 \text{ then AR_AtFirstProper} \\
& \quad \quad \quad \quad \text{else AR_AtFurtherProper}) \\
& \quad \quad \quad \text{else posk0 } k)) \\
& \quad \wedge \text{mt_tape } c = \text{mt_tape } c0 \\
& \quad \wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx \text{ } k < j \\
& \quad \quad \text{then (if } \text{mt_pos } cM \text{ } k = 0 \text{ then Suc } 0 \\
& \quad \quad \quad \text{else sim_pos (block_width } (\Gamma_tm \text{ } M)) (\text{mt_pos } cM \text{ } k) \\
& \quad \quad \quad \quad + \text{block_width } (\Gamma_tm \text{ } M)) \\
& \quad \quad \text{else mt_pos } c0 \text{ } k)) \\
& \text{proof (induction } j) \\
& \text{case } 0 \\
& \text{have } (c0, c0) \in R' \wedge \wedge 0 \text{ by simp} \\
& \text{moreover have } \text{mt_state } c0 = (q, \text{AR_SimRead}, k_unidx \text{ } 0, 0, \\
& \quad (\lambda k. \text{if } k_idx \text{ } k < 0 \text{ then } \text{mt_tape } cM \text{ } k (\text{mt_pos } cM \text{ } k) \text{ else } \text{buf0 } k), \text{dvec}, \\
& \quad (\lambda k. \text{if } k_idx \text{ } k < 0
\end{aligned}$$

```

      then (if mt_pos cM k = 0 then AR_AtLE
            else if mt_pos cM k = 1 then AR_AtFirstProper
            else AR_AtFurtherProper)
      else posk0 k))
    using stg0 by (simp add: k_unidx_zero)
  ultimately show ?case
    by (intro exI[where x = c0] exI[where x = 0]) (simp add: ppos)
next
case (Suc j)
have sucle: Suc j ≤ k_tm M - 1 by (rule Suc.prem)
have jcard: j < k_tm M using sucle by simp
have sjcard: Suc j < k_tm M using sucle by simp
have jle: j ≤ k_tm M - 1 using sucle by simp
let ?k = block_width (Γ_tm M)
let ?tk = k_unidx j
let ?p = mt_pos cM ?tk
let ?aM = λk. mt_tape cM k (mt_pos cM k)
let ?bf = λi. (λk. if k_idx k < i then ?aM k else buf0 k)
let ?pk = λi. (λk. if k_idx k < i
  then (if mt_pos cM k = 0 then AR_AtLE
        else if mt_pos cM k = 1 then AR_AtFirstProper
        else AR_AtFurtherProper)
  else posk0 k)
let ?ps = λi. (λk. if k_idx k < i
  then (if mt_pos cM k = 0 then Suc 0
        else sim_pos ?k (mt_pos cM k) + ?k)
  else mt_pos c0 k)
have tkidx: k_idx ?tk = j by (rule k_idx_unidx[OF jcard])
have notlast: ¬ is_last k M ?tk
  using is_last_k_unidx[OF jcard] sjcard by simp
have ksucc: k_succ ?tk = k_unidx (Suc j) by (rule k_succ_unidx[OF sjcard])
have inj: inj_on k_idx UNIV by (rule k_idx_inj)
have kne: ∧k. k_idx k = j ⇒ k = ?tk
proof -
  fix k assume k_idx k = j
  hence k_idx k = k_idx ?tk using tkidx by simp
  thus k = ?tk using inj_on_eq_iff[OF inj UNIV_I UNIV_I] by simp
qed
obtain c m where
  cm_rel: (c0, c) ∈ R' ^^ m
  and m_le: m ≤ j * (?k + 2)
  and c_st: mt_state c = (q, AR_SimRead, ?tk, 0, ?bf j, dvec, ?pk j)
  and c_tp: mt_tape c = mt_tape c0
  and c_ps: mt_pos c = ?ps j
  using Suc.IH[OF jle] by blast
have tk_active: ?tk < k_tm M using jcard by (simp add: k_unidx_def)
have tk_corr: ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
  (mt_tape cM ?tk) (mt_tape c ?tk)
  using tcorr[rule_format, OF tk_active] c_tp by simp

```

```

have tk_pos: mt_pos c ?tk = sim_pos ?k ?p
  using c_ps ppos tkidx by simp
have tk_pkok: ?pk j ?tk = AR_AtLE  $\longleftrightarrow$  ?p = 0
  using pkok tkidx by simp
have bf_in:  $\bigwedge k. ?bf j k \in \Gamma\_tm M \cup \{bl\_tm M\}$ 
  using tapeG bufG by auto
have tk_vsrc: ar_valid_stage ( $\Gamma\_tm M$ ) (bl_tm M)
  (AR_SimRead, ?tk, 0, ?bf j, dvec, ?pk j)
  using kge2 bf_in by (auto simp: ar_valid_stage_def)
have pad_c:  $\forall j' \geq k\_tm M. mt\_tape c j' (mt\_pos c j') = BLANK4$ 
proof (intro allI impI)
  fix j' assume jge:  $k\_tm M \leq j'$ 
  have jnj:  $\neg k\_idx j' < j$  using jge jcard by (simp add: k_idx_def)
  have mt_tape c j' (mt_pos c j') = mt_tape c0 j' (mt_pos c0 j')
    using c_tp c_ps jnj by simp
  thus mt_tape c j' (mt_pos c j') = BLANK4 using pad0 jge by simp
qed
have src_c: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, ?tk, 0, ?bf j, dvec, ?pk j)
  using src0 tk_active jcard by (auto simp: ar_stage_bounded_def k_idx_def)
obtain c' where
  step:  $(c, c') \in R' \wedge (if ?p = 0 then 1 else ?k + 2)$ 
  and c'_st:  $mt\_state c' = (q, AR\_SimRead, k\_succ ?tk, 0,$ 
     $(?bf j)(?tk := ?aM ?tk), dvec,$ 
     $(?pk j)(?tk := if ?p = 0 then AR\_AtLE$ 
       $else if ?p = 1 then AR\_AtFirstProper$ 
       $else AR\_AtFurtherProper))$ 
  and c'_tp:  $mt\_tape c' = mt\_tape c$ 
  and c'_ps:  $mt\_pos c' = (mt\_pos c)(?tk :=$ 
     $if ?p = 0 then Suc 0 else sim\_pos ?k ?p + ?k)$ 
  using leaf[OF vM qQ kge2 notlast c_st tk_corr tk_pos
    tk_pkok tk_vsrc pad_c src_c] tapeG by blast
have bf_eq:  $(?bf j)(?tk := ?aM ?tk) = ?bf (Suc j)$ 
proof (rule ext)
  fix k
  show  $((?bf j)(?tk := ?aM ?tk)) k = ?bf (Suc j) k$ 
  proof (cases k = ?tk)
    case True thus ?thesis using tkidx by simp
  next
    case False
    hence k_idx k  $\neq j$  using kne by blast
    thus ?thesis using False by (simp add: less_Suc_eq)
  qed
qed
have pk_eq:  $(?pk j)(?tk := if ?p = 0 then AR\_AtLE$ 
   $else if ?p = 1 then AR\_AtFirstProper$ 
   $else AR\_AtFurtherProper) = ?pk (Suc j)$ 
proof (rule ext)
  fix k

```

```

show ((?pk j)(?tk := if ?p = 0 then AR_AtLE
                    else if ?p = 1 then AR_AtFirstProper
                    else AR_AtFurtherProper)) k = ?pk (Suc j) k
proof (cases k = ?tk)
  case True thus ?thesis using tkidx by simp
next
  case False
  hence k_idx k ≠ j using kne by blast
  thus ?thesis using False by (simp add: less_Suc_eq)
qed
qed
have ps_eq: (?ps j)(?tk := if ?p = 0 then Suc 0 else sim_pos ?k ?p + ?k)
      = ?ps (Suc j)
proof (rule ext)
  fix k
  show ((?ps j)(?tk := if ?p = 0 then Suc 0
                    else sim_pos ?k ?p + ?k)) k = ?ps (Suc j) k
proof (cases k = ?tk)
  case True thus ?thesis using tkidx by simp
next
  case False
  hence k_idx k ≠ j using kne by blast
  thus ?thesis using False by (simp add: less_Suc_eq)
qed
qed
have chain: (c0, c') ∈ R' ^^ (m + (if ?p = 0 then 1 else ?k + 2))
proof -
  have (c0, c') ∈ R' ^^ m
      O R' ^^ (if ?p = 0 then 1 else ?k + 2)
  using cm_rel step by (rule relcompI)
  thus ?thesis by (simp add: relpow_add)
qed
have bound: m + (if ?p = 0 then 1 else ?k + 2) ≤ Suc j * (?k + 2)
  using m_le by (cases ?p = 0) auto
show ?case
proof (intro exI[where x = c'] exI[where x = m + (if ?p = 0 then 1 else ?k
+ 2)])
  conjI)
  show (c0, c') ∈ R' ^^ (m + (if ?p = 0 then 1 else ?k + 2)) by (rule chain)
  show m + (if ?p = 0 then 1 else ?k + 2) ≤ Suc j * (?k + 2)
  by (rule bound)
  show mt_state c' = (q, AR_SimRead, k_unidx (Suc j), 0, ?bf (Suc j), dvec,
    ?pk (Suc j))
  using c'_st ksucc bf_eq pk_eq by simp
  show mt_tape c' = mt_tape c0 using c'_tp c_tp by simp
  show mt_pos c' = ?ps (Suc j) using c'_ps c_ps ps_eq by simp
qed
qed

```

lemma *ar_read_prefix*:

fixes $M :: ('q, 'a) \text{ mttm}$
and $cM :: ('a, 'q) \text{ mt_config}$
and $c0 :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $stg0: \text{mt_state } c0 = (q, \text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$
and $tcorr: \forall k < k_tm \ M. \text{ar_tape_correspondence } (\Gamma_tm \ M) (\text{le_tm } M)$
 $(\text{bl_tm } M)$
 $(\text{mt_tape } cM \ k) (\text{mt_tape } c0 \ k)$
and $ppos: \bigwedge k. \text{mt_pos } c0 \ k = \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) (\text{mt_pos } cM \ k)$
and $pkok: \bigwedge k. \text{posk0 } k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM \ k = 0$
and $\text{tapeG}: \bigwedge k. \text{mt_tape } cM \ k (\text{mt_pos } cM \ k) \in \Gamma_tm \ M$
and $\text{bufG}: \bigwedge k. \text{buf0 } k \in \Gamma_tm \ M \cup \{\text{bl_tm } M\}$
and $\text{pad0}: \forall j \geq k_tm \ M. \text{mt_tape } c0 \ j (\text{mt_pos } c0 \ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \ M)$
 $(\text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$
shows $j \leq k_tm \ M - 1 \implies$
 $(\exists c \ m. (c0, c) \in (\text{mttm_step } (\text{alphabet_reduce_delta } M)) \ \wedge \wedge \ m$
 $\wedge m \leq j * (\text{block_width } (\Gamma_tm \ M) + 2)$
 $\wedge \text{mt_state } c = (q, \text{AR_SimRead}, k_unidx \ j, 0,$
 $(\lambda k. \text{if } k_idx \ k < j \text{ then } \text{mt_tape } cM \ k (\text{mt_pos } cM \ k) \text{ else } \text{buf0 } k),$
 $\text{dvec},$
 $(\lambda k. \text{if } k_idx \ k < j$
 $\text{ then } (\text{if } \text{mt_pos } cM \ k = 0 \text{ then } \text{AR_AtLE}$
 $\text{ else if } \text{mt_pos } cM \ k = 1 \text{ then } \text{AR_AtFirstProper}$
 $\text{ else } \text{AR_AtFurtherProper}$
 $\text{ else } \text{posk0 } k))$
 $\wedge \text{mt_tape } c = \text{mt_tape } c0$
 $\wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx \ k < j$
 $\text{ then } (\text{if } \text{mt_pos } cM \ k = 0 \text{ then } \text{Suc } 0$
 $\text{ else } \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) (\text{mt_pos } cM \ k)$
 $\text{ + block_width } (\Gamma_tm \ M))$
 $\text{ else } \text{mt_pos } c0 \ k))$
by $(\text{rule } \text{ar_read_prefix_gen}$
 $[\text{OF } vM \ qQ \ kge2 \ stg0 \ tcorr \ ppos \ pkok \ \text{tapeG } \text{bufG } \text{pad0 } \text{src0}])$
 $(\text{rule } \text{ar_read_tape_step}; \text{assumption})$

The full read phase: the prefix walk over the first $k_tm \ M - 1$ tapes followed by the unified last-tape read *ar_read_tape_finish_step*, landing at *AR_SimCompute* (current-tape field $k_unidx \ 0$) with every tape's M -read symbol decoded into *buf*, every *posk* re-synced exact, and every head left at its block-end (or position 1 for an *LE* tape). Aggregate cost $\leq k_tm \ M \cdot (b + 2)$. The final *buf*/*posk*/position vectors collapse from the $k_idx \ k < j$ split because, on the last tape, every other tape has index $< k_tm \ M - 1$ (k_idx bijective into $\{..< k_tm \ M\}$).

Relation-generic full read phase scaffold. The two helpers (prefix walk, last-tape finish) are the *leaf_prefix* and *leaf_finish* hypotheses; union and sub versions are thin instantiations. The finish leaf is function-heavy, so it is discharged via (*rule* ...; *assumption*) rather than positional *OF*.

lemma *ar_read_phase_gen*:

fixes $M :: ('q, 'a) \text{mttm}$
and $cM :: ('a, 'q) \text{mt_config}$
and $c0 :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config}$
and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config rel}$
assumes *leaf_prefix*:
 $\bigwedge jj. \llbracket \text{valid_mttm } M; q \in Q_tm \ M; 2 \leq \text{block_width } (\Gamma_tm \ M);$
 $\text{mt_state } c0 = (q, \text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0});$
 $\forall k < k_tm \ M. \text{ar_tape_correspondence } (\Gamma_tm \ M) (le_tm \ M)$
 $(bl_tm \ M)$
 $(\text{mt_tape } cM \ k) (\text{mt_tape } c0 \ k);$
 $\bigwedge k. \text{mt_pos } c0 \ k = \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) (\text{mt_pos } cM$
 $k);$
 $\bigwedge k. \text{posk0 } k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM \ k = 0;$
 $\bigwedge k. \text{mt_tape } cM \ k (\text{mt_pos } cM \ k) \in \Gamma_tm \ M;$
 $\bigwedge k. \text{buf0 } k \in \Gamma_tm \ M \cup \{bl_tm \ M\};$
 $\forall j \geq k_tm \ M. \text{mt_tape } c0 \ j (\text{mt_pos } c0 \ j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (bl_tm \ M) (k_tm \ M)$
 $(\text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0});$
 $jj \leq k_tm \ M - 1 \rrbracket$
 $\implies (\exists c \ m. (c0, c) \in R' \wedge m$
 $\wedge m \leq jj * (\text{block_width } (\Gamma_tm \ M) + 2)$
 $\wedge \text{mt_state } c = (q, \text{AR_SimRead}, k_unidx \ jj, 0,$
 $(\lambda k. \text{if } k_idx \ k < jj \text{ then } \text{mt_tape } cM \ k (\text{mt_pos } cM \ k) \text{ else}$
 $\text{buf0 } k),$
 $\text{dvec},$
 $(\lambda k. \text{if } k_idx \ k < jj$
 $\text{then } (\text{if } \text{mt_pos } cM \ k = 0 \text{ then } \text{AR_AtLE}$
 $\text{else if } \text{mt_pos } cM \ k = 1 \text{ then } \text{AR_AtFirstProper}$
 $\text{else } \text{AR_AtFurtherProper})$
 $\text{else } \text{posk0 } k))$
 $\wedge \text{mt_tape } c = \text{mt_tape } c0$
 $\wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx \ k < jj$
 $\text{then } (\text{if } \text{mt_pos } cM \ k = 0 \text{ then } \text{Suc } 0$
 $\text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) (\text{mt_pos } cM \ k)$
 $+ \text{block_width } (\Gamma_tm \ M))$
 $\text{else } \text{mt_pos } c0 \ k))$
and *leaf_finish*:
 $\bigwedge cc \ tk' \ \text{buf}' \ \text{posk}' \ tM' \ p'.$
 $\llbracket \text{valid_mttm } M; q \in Q_tm \ M; 2 \leq \text{block_width } (\Gamma_tm \ M);$
 $\text{is_last_k } M \ tk';$
 $\text{mt_state } cc = (q, \text{AR_SimRead}, tk', 0, \text{buf}', \text{dvec}, \text{posk}');$
 $\text{ar_tape_correspondence } (\Gamma_tm \ M) (le_tm \ M) (bl_tm \ M)$
 $tM' (\text{mt_tape } cc \ tk');$
 $\text{mt_pos } cc \ tk' = \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) p';$

$$\begin{aligned}
& \text{posk}' \text{ tk}' = \text{AR_AtLE} \longleftrightarrow p' = 0; \\
& 1 \leq p' \implies tM' p' \in \Gamma_tm M; \\
& \text{ar_valid_stage } (\Gamma_tm M) (bl_tm M) \\
& \quad (\text{AR_SimRead}, \text{tk}', 0, \text{buf}', \text{dvec}, \text{posk}'); \\
& \forall j \geq k_tm M. \text{mt_tape } cc j (\text{mt_pos } cc j) = \text{BLANK4}; \\
& \text{ar_stage_bounded } (bl_tm M) (k_tm M) \\
& \quad (\text{AR_SimRead}, \text{tk}', 0, \text{buf}', \text{dvec}, \text{posk}') \parallel \\
& \implies \exists c''. (cc, c'') \in R' \wedge (if p' = 0 then 1 else \text{block_width } (\Gamma_tm M) \\
& + 2) \\
& \quad \wedge \text{mt_state } c'' = (q, \text{AR_SimCompute}, k_unidx 0, 0, \\
& \quad \text{buf}'(\text{tk}' := tM' p'), \text{dvec}, \\
& \quad \text{posk}'(\text{tk}' := if p' = 0 then \text{AR_AtLE} \\
& \quad \quad \text{else if } p' = 1 \text{ then } \text{AR_AtFirstProper} \\
& \quad \quad \text{else } \text{AR_AtFurtherProper})) \\
& \quad \wedge \text{mt_tape } c'' = \text{mt_tape } cc \\
& \quad \wedge \text{mt_pos } c'' = (\text{mt_pos } cc)(\text{tk}' := \\
& \quad \quad if p' = 0 then \text{Suc } 0 \\
& \quad \quad \text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm M)) p' + \text{block_width} \\
& (\Gamma_tm M)) \\
& \quad \text{and } vM: \text{valid_mttm } M \\
& \quad \text{and } qQ: q \in Q_tm M \\
& \quad \text{and } kge2: 2 \leq \text{block_width } (\Gamma_tm M) \\
& \quad \text{and } stg0: \text{mt_state } c0 = (q, \text{AR_SimRead}, 0, 0, \text{buf}0, \text{dvec}, \text{posk}0) \\
& \quad \text{and } tcorr: \forall k < k_tm M. \text{ar_tape_correspondence } (\Gamma_tm M) (le_tm M) \\
& (bl_tm M) \\
& \quad (\text{mt_tape } cM k) (\text{mt_tape } c0 k) \\
& \quad \text{and } ppos: \bigwedge k. \text{mt_pos } c0 k = \text{sim_pos } (\text{block_width } (\Gamma_tm M)) (\text{mt_pos } \\
& cM k) \\
& \quad \text{and } pkok: \bigwedge k. \text{posk}0 k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM k = 0 \\
& \quad \text{and } tapeG: \bigwedge k. \text{mt_tape } cM k (\text{mt_pos } cM k) \in \Gamma_tm M \\
& \quad \text{and } bufG: \bigwedge k. \text{buf}0 k \in \Gamma_tm M \cup \{bl_tm M\} \\
& \quad \text{and } pad0: \forall j \geq k_tm M. \text{mt_tape } c0 j (\text{mt_pos } c0 j) = \text{BLANK4} \\
& \quad \text{and } src0: \text{ar_stage_bounded } (bl_tm M) (k_tm M) \\
& \quad \quad (\text{AR_SimRead}, 0, 0, \text{buf}0, \text{dvec}, \text{posk}0) \\
& \text{shows } \exists c m. (c0, c) \in R' \wedge m \\
& \quad \wedge m \leq k_tm M * (\text{block_width } (\Gamma_tm M) + 2) \\
& \quad \wedge \text{mt_state } c = (q, \text{AR_SimCompute}, k_unidx 0, 0, \\
& \quad \quad (\lambda k. if k < k_tm M then \text{mt_tape } cM k (\text{mt_pos } cM k) \text{ else } \text{buf}0 k), \\
& \quad \quad \text{dvec}, \\
& \quad \quad (\lambda k. if k < k_tm M \\
& \quad \quad \quad \text{then } (if \text{mt_pos } cM k = 0 \text{ then } \text{AR_AtLE} \\
& \quad \quad \quad \text{else if } \text{mt_pos } cM k = 1 \text{ then } \text{AR_AtFirstProper} \\
& \quad \quad \quad \text{else } \text{AR_AtFurtherProper}) \\
& \quad \quad \quad \text{else } \text{posk}0 k)) \\
& \quad \wedge \text{mt_tape } c = \text{mt_tape } c0 \\
& \quad \wedge \text{mt_pos } c = (\lambda k. if k < k_tm M \\
& \quad \quad \text{then } (if \text{mt_pos } cM k = 0 \text{ then } \text{Suc } 0 \\
& \quad \quad \text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm M)) (\text{mt_pos } cM k) + \text{block_width} \\
& (\Gamma_tm M))
\end{aligned}$$

else mt_pos c0 k)

proof –

let $?k = \text{block_width } (\Gamma_tm\ M)$
let $?N = k_tm\ M$
let $?m1 = ?N - 1$
let $?tl = k_unidx\ ?m1$
let $?p = mt_pos\ cM\ ?tl$
let $?aM = \lambda k. mt_tape\ cM\ k\ (mt_pos\ cM\ k)$
have $card1: 0 < ?N$ **using** vM **by** $(cases\ M)$ **auto**
have $m1suc: ?N = Suc\ ?m1$ **using** $card1$ **by** $simp$
have $m1card: ?m1 < ?N$ **using** $card1$ **by** $simp$
have $m1lt: ?m1 < k_tm\ M$ **using** $card1$ **by** $simp$
have $tl_eq: ?tl = ?m1$ **by** $(simp\ add: k_unidx_def)$
have $tlidx: k_idx\ ?tl = ?m1$ **by** $(rule\ k_idx_unidx[OF\ m1card])$
have $tl_active: ?tl < k_tm\ M$ **using** $m1lt\ tl_eq$ **by** $simp$
have $islast: is_last_k\ M\ ?tl$ **using** $is_last_k_unidx[OF\ m1card]\ m1suc$ **by** $simp$
— A tape index other than the last active one lies strictly below $?m1$ exactly when it is active; this replaces the deleted k_idx -surjectivity step (false on padding).
have $cond: (k < ?m1) = (k < k_tm\ M)$ **if** $k \neq ?m1$ **for** k
using $that\ m1suc$ **by** $(auto\ simp: less_Suc_eq)$
let $?bf = \lambda k. \text{if } k_idx\ k < ?m1 \text{ then } ?aM\ k \text{ else } buf0\ k$
let $?pk = \lambda k. \text{if } k_idx\ k < ?m1$
then (if mt_pos cM k = 0 then AR_AtLE
else if mt_pos cM k = 1 then AR_AtFirstProper
else AR_AtFurtherProper)
else posk0 k
let $?ps = \lambda k. \text{if } k_idx\ k < ?m1$
then (if mt_pos cM k = 0 then Suc 0
else sim_pos ?k (mt_pos cM k) + ?k)
else mt_pos c0 k
obtain $c1\ m1$ **where**
 $c1_rel: (c0, c1) \in R' \wedge m1$
 $m1_le: m1 \leq ?m1 * (?k + 2)$
 $c1_st: mt_state\ c1 = (q, AR_SimRead, ?tl, 0, ?bf, dvec, ?pk)$
 $c1_tp: mt_tape\ c1 = mt_tape\ c0$
 $c1_ps: mt_pos\ c1 = ?ps$
using $leaf_prefix[OF\ vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ pkok\ tapeG\ bufG\ pad0\ src0,$
 $of\ ?m1]$
by $auto$
have $tl_corr: ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$
 $(mt_tape\ cM\ ?tl)\ (mt_tape\ c1\ ?tl)$
using $tcorr[rule_format, OF\ tl_active]\ c1_tp$ **by** $simp$
have $tl_pos: mt_pos\ c1\ ?tl = sim_pos\ ?k\ ?p$
using $c1_ps\ ppos\ tlidx$ **by** $simp$
have $tl_pkok: ?pk\ ?tl = AR_AtLE \longleftrightarrow ?p = 0$
using $pkok\ tlidx$ **by** $simp$
have $bf_in: \bigwedge k. ?bf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using $tapeG\ bufG$ **by** $auto$
have $tl_vsr: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$

```

      (AR_SimRead, ?tl, 0, ?bf, dvec, ?pk)
    using kge2 bf_in by (auto simp: ar_valid_stage_def)
  have tail_idx:  $\bigwedge j. k\_tm\ M \leq j \implies \neg k\_idx\ j < ?m1$ 
    using m1suc unfolding k_idx_def by linarith
  have pad_c1:  $\forall j \geq k\_tm\ M. mt\_tape\ c1\ j\ (mt\_pos\ c1\ j) = BLANK4$ 
  proof (intro allI impI)
    fix j assume jge:  $k\_tm\ M \leq j$ 
    have mt_tape_c1_j (mt_pos c1 j) = mt_tape c0 j (mt_pos c0 j)
      using c1_tp c1_ps tail_idx[OF jge] by simp
    thus mt_tape c1 j (mt_pos c1 j) = BLANK4 using pad0 jge by simp
  qed
  have src_c1: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimRead, ?tl, 0, ?bf, dvec, ?pk)
    using src0 tl_active tail_idx by (auto simp: ar_stage_bounded_def)
  obtain c2 where
    step:  $(c1, c2) \in R' \wedge (if\ ?p = 0\ then\ 1\ else\ ?k + 2)$ 
    and c2_st:  $mt\_state\ c2 = (q, AR\_SimCompute, k\_unidx\ 0, 0,$ 
       $?bf(?tl := ?aM\ ?tl), dvec,$ 
       $?pk(?tl := if\ ?p = 0\ then\ AR\_AtLE$ 
         $else\ if\ ?p = 1\ then\ AR\_AtFirstProper$ 
         $else\ AR\_AtFurtherProper))$ 
    and c2_tp:  $mt\_tape\ c2 = mt\_tape\ c1$ 
    and c2_ps:  $mt\_pos\ c2 = (mt\_pos\ c1)(?tl :=$ 
       $if\ ?p = 0\ then\ Suc\ 0\ else\ sim\_pos\ ?k\ ?p + ?k)$ 
    using leaf_finish[OF vM qQ kge2 islast c1_st tl_corr
      tl_pos tl_pkok _ tl_vsrc pad_c1 src_c1] tapeG
    by blast
  — The last-tape fun_upd collapses to the active-guarded vector: active tapes ( $k < k\_tm\ M$ ) take the walked value, padding tapes stay at their entry value.
  have bf_full:  $?bf(?tl := ?aM\ ?tl)$ 
    =  $(\lambda k. if\ k < k\_tm\ M\ then\ ?aM\ k\ else\ buf0\ k)$ 
  proof (rule ext)
    fix k
    show  $(?bf(?tl := ?aM\ ?tl))\ k = (if\ k < k\_tm\ M\ then\ ?aM\ k\ else\ buf0\ k)$ 
    proof (cases k = ?tl)
      case True thus ?thesis using tl_eq m1lt by simp
    next
      case False
      hence kne:  $k \neq ?m1$  using tl_eq by simp
      show ?thesis using False kne cond[OF kne] by (simp add: k_idx_def)
    qed
  qed
  have pk_full:  $?pk(?tl := if\ ?p = 0\ then\ AR\_AtLE$ 
     $else\ if\ ?p = 1\ then\ AR\_AtFirstProper$ 
     $else\ AR\_AtFurtherProper)$ 
    =  $(\lambda k. if\ k < k\_tm\ M$ 
       $then\ (if\ mt\_pos\ cM\ k = 0\ then\ AR\_AtLE$ 
         $else\ if\ mt\_pos\ cM\ k = 1\ then\ AR\_AtFirstProper$ 
         $else\ AR\_AtFurtherProper)$ 

```

```

      else posk0 k)
proof (rule ext)
  fix k
  show (?pk(?tl := if ?p = 0 then AR_AtLE
    else if ?p = 1 then AR_AtFirstProper
    else AR_AtFurtherProper)) k
    = (if k < k_tm M
      then (if mt_pos cM k = 0 then AR_AtLE
        else if mt_pos cM k = 1 then AR_AtFirstProper
        else AR_AtFurtherProper)
      else posk0 k)
proof (cases k = ?tl)
  case True thus ?thesis using tl_eq m1lt by simp
next
  case False
  hence kne: k ≠ ?m1 using tl_eq by simp
  show ?thesis using False kne cond[OF kne] by (simp add: k_idx_def)
qed
qed
have ps_full: (mt_pos c1)(?tl := if ?p = 0 then Suc 0
  else sim_pos ?k ?p + ?k)
  = (λk. if k < k_tm M
    then (if mt_pos cM k = 0 then Suc 0
      else sim_pos ?k (mt_pos cM k) + ?k)
    else mt_pos c0 k)
proof (rule ext)
  fix k
  show ((mt_pos c1)(?tl := if ?p = 0 then Suc 0
    else sim_pos ?k ?p + ?k)) k
    = (if k < k_tm M
      then (if mt_pos cM k = 0 then Suc 0
        else sim_pos ?k (mt_pos cM k) + ?k)
      else mt_pos c0 k)
proof (cases k = ?tl)
  case True thus ?thesis using tl_eq m1lt by simp
next
  case False
  hence kne: k ≠ ?m1 using tl_eq by simp
  show ?thesis
    using False kne cond[OF kne] c1_ps by (simp add: k_idx_def)
qed
qed
have chain: (c0, c2) ∈ R' ^^ (m1 + (if ?p = 0 then 1 else ?k + 2))
proof -
  have (c0, c2) ∈ R' ^^ m1
    O R' ^^ (if ?p = 0 then 1 else ?k + 2)
  using c1_rel step by (rule relcompI)
  thus ?thesis by (simp add: relpow_add)
qed

```

```

have bound:  $m1 + (\text{if } ?p = 0 \text{ then } 1 \text{ else } ?k + 2) \leq ?N * (?k + 2)$ 
proof -
  obtain Nm where Nm:  $?N = \text{Suc } Nm$  using card1 by (cases ?N) auto
  have e1:  $?N * (?k + 2) = (?k + 2) + Nm * (?k + 2)$  by (simp add: Nm)
  have e2:  $m1 \leq Nm * (?k + 2)$  using m1_le Nm by simp
  have e3:  $(\text{if } ?p = 0 \text{ then } 1 \text{ else } ?k + 2) \leq ?k + 2$  by simp
  show ?thesis using e1 e2 e3 by linarith
qed
show ?thesis
proof (intro exI[where x = c2]
  exI[where x =  $m1 + (\text{if } ?p = 0 \text{ then } 1 \text{ else } ?k + 2)$ ] conjI)
  show  $(c0, c2) \in R' \wedge (m1 + (\text{if } ?p = 0 \text{ then } 1 \text{ else } ?k + 2))$  by (rule chain)
  show  $m1 + (\text{if } ?p = 0 \text{ then } 1 \text{ else } ?k + 2) \leq ?N * (?k + 2)$  by (rule bound)
  show  $mt\_state\ c2 = (q, AR\_SimCompute, k\_unidx\ 0, 0,$ 
     $(\lambda k. \text{if } k < k\_tm\ M \text{ then } ?aM\ k \text{ else } buf0\ k), dvec,$ 
     $(\lambda k. \text{if } k < k\_tm\ M$ 
      then  $(\text{if } mt\_pos\ cM\ k = 0 \text{ then } AR\_AtLE$ 
        else  $\text{if } mt\_pos\ cM\ k = 1 \text{ then } AR\_AtFirstProper$ 
        else  $AR\_AtFurtherProper)$ 
      else  $posk0\ k))$ 
    using c2_st bf_full pk_full by simp
  show  $mt\_tape\ c2 = mt\_tape\ c0$  using c2_tp c1_tp by simp
  show  $mt\_pos\ c2 = (\lambda k. \text{if } k < k\_tm\ M$ 
    then  $(\text{if } mt\_pos\ cM\ k = 0 \text{ then } Suc\ 0$ 
      else  $sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k)$ 
    else  $mt\_pos\ c0\ k)$ 
    using c2_ps ps_full by simp
qed
qed

lemma ar_read_phase:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c0 :: (sym4, 'q  $\times$  'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ:  $q \in Q\_tm\ M$ 
  and kge2:  $2 \leq block\_width\ (\Gamma\_tm\ M)$ 
  and stg0:  $mt\_state\ c0 = (q, AR\_SimRead, 0, 0, buf0, dvec, posk0)$ 
  and tcorr:  $\forall k < k\_tm\ M. ar\_tape\_correspondence\ (\Gamma\_tm\ M)\ (le\_tm\ M)$ 
  (bl_tm M)
  (mt_tape cM k) (mt_tape c0 k)
  and ppos:  $\bigwedge k. mt\_pos\ c0\ k = sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ (mt\_pos\ cM\ k)$ 
  and pkok:  $\bigwedge k. posk0\ k = AR\_AtLE \longleftrightarrow mt\_pos\ cM\ k = 0$ 
  and tapeG:  $\bigwedge k. mt\_tape\ cM\ k\ (mt\_pos\ cM\ k) \in \Gamma\_tm\ M$ 
  and bufG:  $\bigwedge k. buf0\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  and pad0:  $\forall j \geq k\_tm\ M. mt\_tape\ c0\ j\ (mt\_pos\ c0\ j) = BLANK4$ 
  and src0:  $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
  (AR_SimRead, 0, 0, buf0, dvec, posk0)

```

```

shows  $\exists c\ m. (c0, c) \in (mttm\_step\ (alphabet\_reduce\_delta\ M)) \wedge^m$ 
 $\wedge m \leq k\_tm\ M * (block\_width\ (\Gamma\_tm\ M) + 2)$ 
 $\wedge mt\_state\ c = (q, AR\_SimCompute, k\_unidx\ 0, 0,$ 
 $(\lambda k. \text{if } k < k\_tm\ M \text{ then } mt\_tape\ cM\ k\ (mt\_pos\ cM\ k) \text{ else } buf0\ k),$ 
 $dvec,$ 
 $(\lambda k. \text{if } k < k\_tm\ M$ 
 $\text{ then } (\text{if } mt\_pos\ cM\ k = 0 \text{ then } AR\_AtLE$ 
 $\text{ else if } mt\_pos\ cM\ k = 1 \text{ then } AR\_AtFirstProper$ 
 $\text{ else } AR\_AtFurtherProper)$ 
 $\text{ else } posk0\ k))$ 
 $\wedge mt\_tape\ c = mt\_tape\ c0$ 
 $\wedge mt\_pos\ c = (\lambda k. \text{if } k < k\_tm\ M$ 
 $\text{ then } (\text{if } mt\_pos\ cM\ k = 0 \text{ then } Suc\ 0$ 
 $\text{ else } sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ (mt\_pos\ cM\ k) + block\_width$ 
 $(\Gamma\_tm\ M))$ 
 $\text{ else } mt\_pos\ c0\ k)$ 
by (rule ar_read_phase_gen
 $[OF\_ \_ vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ pkok\ tapeG\ bufG\ pad0\ src0];$ 
 $(rule\ ar\_read\_prefix\ ar\_read\_tape\_finish\_step; assumption))$ 

end
theory AlphabetReduction_ForwardWrite
imports AlphabetReduction_ForwardRead
begin

```

2.14 Write phase composition

Relation-generic write back-walk loop scaffold. The walk leaf (*ar_write_walk_step*) is the *leaf* hypothesis; the union- and sub-relation back-loops are thin instantiations. Throughout, *b* abbreviates *block_width* Γ (the per-symbol cell width).

```

lemma ar_write_back_loop_gen:
fixes  $M :: ('q, 'a)\ mttm$ 
and  $c0 :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config$ 
and  $R' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config\ rel$ 
assumes leaf:
 $\bigwedge cc\ i. \llbracket valid\_mttm\ M;$ 
 $mt\_state\ cc = (q, AR\_SimWrite, tk, i, buf, dvec, posk);$ 
 $q \in Q\_tm\ M;$ 
 $posk\ tk \neq AR\_AtLE;$ 
 $mt\_tape\ cc\ tk\ (mt\_pos\ cc\ tk) \neq LE4;$ 
 $Suc\ i \leq block\_width\ (\Gamma\_tm\ M);$ 
 $ar\_valid\_stage\ (\Gamma\_tm\ M)\ (bl\_tm\ M)$ 
 $(AR\_SimWrite, tk, i, buf, dvec, posk);$ 
 $\forall j \geq k\_tm\ M. mt\_tape\ cc\ j\ (mt\_pos\ cc\ j) = BLANK4;$ 
 $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
 $(AR\_SimWrite, tk, i, buf, dvec, posk) \rrbracket$ 
 $\implies \exists c''. (cc, c'') \in R'$ 

```

$\wedge mt_state\ c'' = (q, AR_SimWrite, tk, Suc\ i, buf, dvec,$
 $posk)$
 $\wedge mt_tape\ c'' = mt_tape\ cc$
 $\wedge mt_pos\ c'' = (mt_pos\ cc)(tk := mt_pos\ cc\ tk - 1)$
and vM : $valid_mttm\ M$
and qQ : $q \in \bar{Q_tm}\ M$
and $kge2$: $2 \leq block_width\ (\Gamma_tm\ M)$
and $poskproper$: $posk\ tk \neq AR_AtLE$
and stg : $mt_state\ c0 = (q, AR_SimWrite, tk, 0, buf, dvec, posk)$
and buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and pos_base : $mt_pos\ c0\ tk = base + block_width\ (\Gamma_tm\ M)$
and $notLE$: $\bigwedge m. \llbracket 1 \leq m; m \leq block_width\ (\Gamma_tm\ M) \rrbracket$
 $\implies mt_tape\ c0\ tk\ (base + m) \neq LE4$
and $pad0$: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0$: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in R' \wedge block_width\ (\Gamma_tm\ M)$
 $\wedge mt_state\ c' = (q, AR_SimWrite, tk, block_width\ (\Gamma_tm\ M),$
 $buf, dvec, posk)$
 $\wedge mt_pos\ c'\ tk = base$
 $\wedge mt_tape\ c' = mt_tape\ c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c'\ k' = mt_pos\ c0\ k')$
proof –
let $?k = block_width\ (\Gamma_tm\ M)$
let $?P = \lambda j\ c.$
 $mt_state\ c = (q, AR_SimWrite, tk, j, buf, dvec, posk)$
 $\wedge mt_pos\ c\ tk = base + (?k - j)$
 $\wedge mt_tape\ c = mt_tape\ c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c\ k' = mt_pos\ c0\ k')$
have $mybase$: $?P\ 0\ c0$ **using** $stg\ pos_base$ **by** $simp$
have $mystep$:
 $\bigwedge j\ cc. \llbracket j < ?k; ?P\ j\ cc \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge ?P\ (Suc\ j)\ c''$
proof –
fix $j\ cc$
assume jlt : $j < ?k$ **and** Pj : $?P\ j\ cc$
from Pj **have** st_cc :
 $mt_state\ cc = (q, AR_SimWrite, tk, j, buf, dvec, posk)$
and pos_cc : $mt_pos\ cc\ tk = base + (?k - j)$
and $tape_cc$: $mt_tape\ cc = mt_tape\ c0$
and $other_cc$: $\forall k'. k' \neq tk \longrightarrow mt_pos\ cc\ k' = mt_pos\ c0\ k'$
by $simp_all$
have tk_lt : $tk < k_tm\ M$ **using** $src0$ **by** $(simp\ add: ar_stage_bounded_def)$
have pad_cc : $\forall j' \geq k_tm\ M. mt_tape\ cc\ j'\ (mt_pos\ cc\ j') = BLANK4$
proof $(intro\ allI\ impI)$
fix j' **assume** jge : $k_tm\ M \leq j'$
have jne : $j' \neq tk$ **using** $jge\ tk_lt$ **by** $simp$
have $mt_tape\ cc\ j'\ (mt_pos\ cc\ j') = mt_tape\ c0\ j'\ (mt_pos\ c0\ j')$


```

    using tape_cc other_cc jne by simp
    thus mt_tape cc j' (mt_pos cc j') = BLANK4 using pad0 jge by simp
qed
have src_cc: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, tk, j, buf, dvec, posk)
  using src0 by (simp add: ar_stage_bounded_def)
have mge1: 1 ≤ ?k - j using jlt by simp
have mlek: ?k - j ≤ ?k by simp
have suc_le: Suc j ≤ ?k using jlt by simp
have jlt2: j < 2 * ?k using jlt by simp
have ksub: ?k - j = Suc (?k - Suc j) using jlt by simp
have vsrc_cc: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimWrite, tk, j, buf, dvec, posk)
  using jlt2 buf_valid by (simp add: ar_valid_stage_def)
have notLE_cc: mt_tape cc tk (mt_pos cc tk) ≠ LE4
  using tape_cc pos_cc notLE[OF mge1 mlek] by simp
obtain c'' where
  bstep: (cc, c'') ∈ R'
  and bst_state: mt_state c'' = (q, AR_SimWrite, tk, Suc j, buf, dvec, posk)
  and bst_tape: mt_tape c'' = mt_tape cc
  and bst_pos: mt_pos c'' = (mt_pos cc)(tk := mt_pos cc tk - 1)
  using leaf
    [OF vM st_cc qQ poskproper notLE_cc suc_le vsrc_cc pad_cc src_cc]
  by blast
have pos_eq: mt_pos c'' tk = base + (?k - Suc j)
  using bst_pos pos_cc ksub by simp
have tape_eq: mt_tape c'' = mt_tape c0 using bst_tape tape_cc by simp
have other_eq: ∀ k'. k' ≠ tk → mt_pos c'' k' = mt_pos c0 k'
  using bst_pos other_cc by simp
show ∃ c''. (cc, c'') ∈ R'
  ∧ ?P (Suc j) c''
  using bstep bst_state pos_eq tape_eq other_eq by blast
qed
have loop:
  ∃ c'. (c0, c') ∈ R' ^^ ?k
  ∧ ?P ?k c'
  by (rule relpow_invariant_chain
    [where P = ?P and n = ?k, OF mystep mybase])
obtain c' where
  chain: (c0, c') ∈ R' ^^ ?k
  and Pfin: ?P ?k c'
  using loop by blast
show ?thesis
proof (intro exI[where x = c'] conjI)
  show (c0, c') ∈ R' ^^ ?k by (rule chain)
  show mt_state c' = (q, AR_SimWrite, tk, ?k, buf, dvec, posk)
    using Pfin by simp
  show mt_pos c' tk = base using Pfin by simp
  show mt_tape c' = mt_tape c0 using Pfin by simp

```

show $\forall k'. k' \neq tk \longrightarrow mt_pos\ c'\ k' = mt_pos\ c0\ k'$ **using** *Pfin* **by** *simp*
qed
qed

The write back-walk loop, proper tape. From an *AR_SimWrite* stage at bit-counter *0* with the head at the block end *base + b* (where the read phase left it) and *buf tk* a proper symbol, the head walks *L* across the *b*-cell block back to *base = sim_pos p*, reaching bit-counter *b*. Tape unchanged (the back-walk only moves); this is the simpler of the two write loops, the analogue of *ar_read_bit_loop* with a constant-tape invariant. Each *ar_write_walk_step* needs its current cell $\neq LE4$; the visited cells are *base + 1 ... base + b* (all proper positions), supplied by *notLE*. Chain length *b*.

lemma *ar_write_back_loop*:
fixes *M* :: ('q, 'a) *mttm*
and *c0* :: (*sym4*, 'q $\times$ 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm M*
and *qQ*: *q* $\in$ *Q_tm M*
and *kge2*: $2 \leq block_width\ (\Gamma_tm\ M)$
and *poskproper*: *posk tk* $\neq AR_AtLE$
and *stg*: *mt_state c0* = (*q*, *AR_SimWrite*, *tk*, *0*, *buf*, *dvec*, *posk*)
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *pos_base*: *mt_pos c0 tk* = *base + block_width* ($\Gamma_tm\ M$)
and *notLE*: $\bigwedge m. \llbracket 1 \leq m; m \leq block_width\ (\Gamma_tm\ M) \rrbracket$
 $\implies mt_tape\ c0\ tk\ (base + m) \neq LE4$
and *pad0*: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and *src0*: *ar_stage_bounded* (*bl_tm M*) (*k_tm M*)
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in (mttm_step\ (alphabet_reduce_delta\ M))$
 $\wedge \wedge block_width\ (\Gamma_tm\ M)$
 $\wedge mt_state\ c' = (q, AR_SimWrite, tk, block_width\ (\Gamma_tm\ M),$
 $buf, dvec, posk)$
 $\wedge mt_pos\ c'\ tk = base$
 $\wedge mt_tape\ c' = mt_tape\ c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c'\ k' = mt_pos\ c0\ k')$
by (*rule ar_write_back_loop_gen*
 $[OF\ ar_write_walk_step\ vM\ qQ\ kge2\ poskproper\ stg\ buf_valid\ pos_base$
notLE
 $pad0\ src0]$)

The write forward bit-write loop, proper tape. From an *AR_SimWrite* stage at bit-counter *b* with the head back at *base* (where *ar_write_back_loop* left it), the head walks *R* across the block writing each cell, reaching bit-counter *b + (b - 1)* with the head at *base + (b - 1)*. Unlike the read loops this loop *mutates* the tape: its invariant carries a partially-overwritten tape (cells *base ... base + j - 1* already hold the new *write_bit* image, the rest hold old content), and each *ar_write_bit_step*'s $\neq LE4$ guard reads the *not-yet-written* current cell *base + j*, which still holds old content (supplied

$\neq LE4$ by *notLE*). Writes the first $b - 1$ cells; the last is the boundary step's job. Chain length $b - 1$.

Relation-generic write forward-walk loop scaffold. The bit-writing leaf (*ar_write_bit_step*) is the *leaf* hypothesis; the union- and sub-relation forward-loops are thin instantiations.

lemma *ar_write_fwd_loop_gen*:

fixes $M :: ('q, 'a) \text{ mttm}$

and $c0 :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$

and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config rel}$

assumes *leaf*:

$\bigwedge_{cc} i. \llbracket \text{valid_mttm } M;$

$\text{mt_state } cc = (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk});$

$q \in Q_tm \ M;$

$\text{posk } tk \neq \text{AR_AtLE};$

$\text{mt_tape } cc \ tk \ (\text{mt_pos } cc \ tk) \neq LE4;$

$\text{block_width } (\Gamma_tm \ M) \leq i;$

$\text{Suc } i < 2 * \text{block_width } (\Gamma_tm \ M);$

$\text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk});$

$\forall j \geq k_tm \ M. \text{mt_tape } cc \ j \ (\text{mt_pos } cc \ j) = \text{BLANK4};$

$\text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$

$(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk}) \rrbracket$

$\implies \exists c''. (cc, c'') \in R'$

$\wedge \text{mt_state } c'' = (q, \text{AR_SimWrite}, tk, \text{Suc } i, \text{buf}, \text{dvec},$

$\text{posk})$

$\wedge \text{mt_tape } c'' = (\text{mt_tape } cc)(tk :=$

$(\text{mt_tape } cc \ tk)(\text{mt_pos } cc \ tk$

$:= \text{write_bit } (\Gamma_tm \ M) \ (\text{bl_tm } M) \ (\text{buf } tk)$

$(i - \text{block_width } (\Gamma_tm \ M))))$

$\wedge \text{mt_pos } c'' = (\text{mt_pos } cc)(tk := \text{Suc } (\text{mt_pos } cc \ tk))$

and $vM: \text{valid_mttm } M$

and $qQ: q \in Q_tm \ M$

and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$

and $\text{poskproper}: \text{posk } tk \neq \text{AR_AtLE}$

and $\text{stg}: \text{mt_state } c0 = (q, \text{AR_SimWrite}, tk, \text{block_width } (\Gamma_tm \ M),$
 $\text{buf}, \text{dvec}, \text{posk})$

and $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm \ M \cup \{\text{bl_tm } M\}$

and $\text{pos_base}: \text{mt_pos } c0 \ tk = \text{base}$

and $\text{notLE}: \bigwedge m. m < \text{block_width } (\Gamma_tm \ M)$

$\implies \text{mt_tape } c0 \ tk \ (\text{base} + m) \neq LE4$

and $\text{pad0}: \forall j \geq k_tm \ M. \text{mt_tape } c0 \ j \ (\text{mt_pos } c0 \ j) = \text{BLANK4}$

and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$

$(\text{AR_SimWrite}, tk, \text{block_width } (\Gamma_tm \ M), \text{buf}, \text{dvec}, \text{posk})$

shows $\exists c'. (c0, c') \in R' \wedge (\text{block_width } (\Gamma_tm \ M) - 1)$

$\wedge \text{mt_state } c' = (q, \text{AR_SimWrite}, tk,$

$\text{block_width } (\Gamma_tm \ M) + (\text{block_width } (\Gamma_tm \ M) - 1),$

$\text{buf}, \text{dvec}, \text{posk})$

$\wedge \text{mt_pos } c' \ tk = \text{base} + (\text{block_width } (\Gamma_tm \ M) - 1)$

```

    ∧ mt_tape c' tk = (λpos.
      if base ≤ pos ∧ pos < base + (block_width (Γ_tm M) - 1)
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
      else mt_tape c0 tk pos)
    ∧ (∀ k'. k' ≠ tk → mt_tape c' k' = mt_tape c0 k')
    ∧ (∀ k'. k' ≠ tk → mt_pos c' k' = mt_pos c0 k')
  proof -
    let ?k = block_width (Γ_tm M)
    let ?G = Γ_tm M and ?bl = bl_tm M
    let ?wb = λm. write_bit ?G ?bl (buf tk) m
    let ?tape = λj pos. if base ≤ pos ∧ pos < base + j
      then ?wb (pos - base) else mt_tape c0 tk pos
    let ?P = λj c.
      mt_state c = (q, AR_SimWrite, tk, ?k + j, buf, dvec, posk)
      ∧ mt_pos c tk = base + j
      ∧ mt_tape c tk = ?tape j
      ∧ (∀ k'. k' ≠ tk → mt_tape c k' = mt_tape c0 k')
      ∧ (∀ k'. k' ≠ tk → mt_pos c k' = mt_pos c0 k')
    have mybase: ?P 0 c0
    proof -
      have ?tape 0 = mt_tape c0 tk by (rule ext) auto
      thus ?thesis using stg_pos_base by simp
    qed
    have mystep:
      ∧j cc. [ j < ?k - 1; ?P j cc ]
      ⇒ ∃ c''. (cc, c'') ∈ R'
      ∧ ?P (Suc j) c''
    proof -
      fix j cc
      assume jlt: j < ?k - 1 and Pj: ?P j cc
      from Pj have st_cc:
        mt_state cc = (q, AR_SimWrite, tk, ?k + j, buf, dvec, posk)
        and pos_cc: mt_pos cc tk = base + j
        and tape_cc: mt_tape cc tk = ?tape j
        and otape_cc: ∀ k'. k' ≠ tk → mt_tape cc k' = mt_tape c0 k'
        and opos_cc: ∀ k'. k' ≠ tk → mt_pos cc k' = mt_pos c0 k'
        by simp_all
      have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
      have pad_cc: ∀ j' ≥ k_tm M. mt_tape cc j' (mt_pos cc j') = BLANK4
      proof (intro allI impI)
        fix j' assume jge: k_tm M ≤ j'
        have jne: j' ≠ tk using jge tk_lt by simp
        have mt_tape_cc_j' (mt_pos cc j') = mt_tape c0 j' (mt_pos c0 j')
          using otape_cc opos_cc jne by simp
        thus mt_tape cc j' (mt_pos cc j') = BLANK4 using pad0 jge by simp
      qed
      have src_cc: ar_stage_bounded (bl_tm M) (k_tm M)
        (AR_SimWrite, tk, ?k + j, buf, dvec, posk)
        using src0 by (simp add: ar_stage_bounded_def)

```

```

have jltk:  $j < ?k$  using jlt by simp
have ilo:  $?k \leq ?k + j$  by simp
have ihi:  $Suc (?k + j) < 2 * ?k$  using jlt by simp
have idx:  $?k + j - ?k = j$  by simp
have vsrc_cc: ar_valid_stage ?G ?bl
  (AR_SimWrite, tk, ?k + j, buf, dvec, posk)
  using ihi buf_valid by (simp add: ar_valid_stage_def)
have cell_cur: mt_tape cc tk (mt_pos cc tk) = mt_tape c0 tk (base + j)
  using tape_cc pos_cc by simp
have notLE_cc: mt_tape cc tk (mt_pos cc tk)  $\neq LE4$ 
  using cell_cur notLE[OF jltk] by simp
obtain c'' where
  bstep:  $(cc, c'') \in R'$ 
  and bst_state: mt_state c'' = (q, AR_SimWrite, tk, Suc (?k + j),
    buf, dvec, posk)
  and bst_tape: mt_tape c'' = (mt_tape cc)(tk :=
    (mt_tape cc tk)(mt_pos cc tk := ?wb (?k + j - ?k)))
  and bst_pos: mt_pos c'' = (mt_pos cc)(tk := Suc (mt_pos cc tk))
  using leaf
    [OF vM st_cc qQ poskproper notLE_cc ilo ihi vsrc_cc pad_cc src_cc]
  by blast
have state_eq: mt_state c'' = (q, AR_SimWrite, tk, ?k + Suc j, buf, dvec,
posk)
  using bst_state by simp
have pos_eq: mt_pos c'' tk = base + Suc j
  using bst_pos pos_cc by simp
have lhs: mt_tape c'' tk = (?tape j)(base + j := ?wb j)
  using bst_tape tape_cc pos_cc idx by simp
have tape_eq: mt_tape c'' tk = ?tape (Suc j)
  using pos_eq by simp
proof (rule ext)
  fix pos
  show mt_tape c'' tk pos = ?tape (Suc j) pos
  proof (cases pos = base + j)
    case True
    thus ?thesis using lhs by simp
  next
    case False
    have mt_tape c'' tk pos = ?tape j pos using lhs False by simp
    thus ?thesis using False by (auto simp: less_Suc_eq)
  qed
qed
have otape_eq:  $\forall k'. k' \neq tk \longrightarrow mt\_tape\ c''\ k' = mt\_tape\ c0\ k'$ 
  using bst_tape otape_cc by simp
have opos_eq:  $\forall k'. k' \neq tk \longrightarrow mt\_pos\ c''\ k' = mt\_pos\ c0\ k'$ 
  using bst_pos opos_cc by simp
show  $\exists c''. (cc, c'') \in R'$ 
   $\wedge ?P (Suc\ j)\ c''$ 
  using bstep state_eq pos_eq tape_eq otape_eq opos_eq by blast
qed

```

```

have loop:
   $\exists c'. (c0, c') \in R' \wedge (\text{?}k - 1)$ 
   $\wedge \text{?}P (\text{?}k - 1) c'$ 
  by (rule relpow_invariant_chain
    [where  $P = \text{?}P$  and  $n = \text{?}k - 1$ , OF mystep mybase])
obtain c' where
  chain:  $(c0, c') \in R' \wedge (\text{?}k - 1)$ 
  and Pfin:  $\text{?}P (\text{?}k - 1) c'$ 
  using loop by blast
show ?thesis
proof (intro exI[where  $x = c'$ ] conjI)
  show  $(c0, c') \in R' \wedge (\text{?}k - 1)$ 
  by (rule chain)
  show  $mt\_state\ c' = (q, AR\_SimWrite, tk, \text{?}k + (\text{?}k - 1), buf, dvec, posk)$ 
  using Pfin by simp
  show  $mt\_pos\ c'\ tk = base + (\text{?}k - 1)$  using Pfin by simp
  show  $mt\_tape\ c'\ tk = (\lambda pos.$ 
    if  $base \leq pos \wedge pos < base + (\text{?}k - 1)$ 
    then  $write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (buf\ tk)\ (pos - base)$ 
    else  $mt\_tape\ c0\ tk\ pos$ )
    using Pfin by simp
  show  $\forall k'. k' \neq tk \longrightarrow mt\_tape\ c'\ k' = mt\_tape\ c0\ k'$  using Pfin by simp
  show  $\forall k'. k' \neq tk \longrightarrow mt\_pos\ c'\ k' = mt\_pos\ c0\ k'$  using Pfin by simp
qed
qed

lemma ar_write_fwd_loop:
  fixes M :: ('q, 'a) mttm
  and c0 :: (sym4, 'q  $\times$  'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ:  $q \in Q\_tm\ M$ 
  and kge2:  $2 \leq block\_width\ (\Gamma\_tm\ M)$ 
  and poskproper:  $posk\ tk \neq AR\_AtLE$ 
  and stg:  $mt\_state\ c0 = (q, AR\_SimWrite, tk, block\_width\ (\Gamma\_tm\ M),$ 
    buf, dvec, posk)
  and buf_valid:  $\forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  and pos_base:  $mt\_pos\ c0\ tk = base$ 
  and notLE:  $\bigwedge m. m < block\_width\ (\Gamma\_tm\ M)$ 
     $\implies mt\_tape\ c0\ tk\ (base + m) \neq LE4$ 
  and pad0:  $\forall j \geq k\_tm\ M. mt\_tape\ c0\ j\ (mt\_pos\ c0\ j) = BLANK4$ 
  and src0:  $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
     $(AR\_SimWrite, tk, block\_width\ (\Gamma\_tm\ M), buf, dvec, posk)$ 
  shows  $\exists c'. (c0, c') \in (mttm\_step\ (alphabet\_reduce\_delta\ M))$ 
     $\wedge (block\_width\ (\Gamma\_tm\ M) - 1)$ 
     $\wedge mt\_state\ c' = (q, AR\_SimWrite, tk,$ 
     $block\_width\ (\Gamma\_tm\ M) + (block\_width\ (\Gamma\_tm\ M) - 1),$ 
    buf, dvec, posk)
     $\wedge mt\_pos\ c'\ tk = base + (block\_width\ (\Gamma\_tm\ M) - 1)$ 
     $\wedge mt\_tape\ c'\ tk = (\lambda pos.$ 

```

$$\begin{aligned}
& \text{if } \text{base} \leq \text{pos} \wedge \text{pos} < \text{base} + (\text{block_width } (\Gamma_tm M) - 1) \\
& \text{then write_bit } (\Gamma_tm M) (\text{bl_tm } M) (\text{buf } tk) (\text{pos} - \text{base}) \\
& \text{else mt_tape } c0 \text{ } tk \text{ } pos) \\
& \wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_tape } c' k' = \text{mt_tape } c0 k') \\
& \wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c' k' = \text{mt_pos } c0 k') \\
\text{by } & (\text{rule ar_write_fwd_loop_gen} \\
& [\text{OF ar_write_bit_step vM qQ kge2 poskproper stg buf_valid pos_base notLE} \\
& \text{pad0 src0}])
\end{aligned}$$

The proper-cell single-tape write prefix: the part of a proper-cell single-tape write shared by the non-last (*boundary*) and last (*finish*) variants. From an *AR_SimWrite* stage at bit-counter 0 with the head at the block end $\text{base} + b$ and *buf tk* a proper symbol, it fires the back-walk loop (*ar_write_back_loop*, head to *base*) and the forward bit-write loop (*ar_write_fwd_loop*, writing the first $b - 1$ cells), reaching bit-counter $b + (b - 1)$ with the head at $\text{base} + (b - 1)$. The remaining last-cell step (boundary or finish) is added by the consumer. The two segment costs compose by *relpow_add* on *relcompI* (chain length $b + (b - 1)$). The *notLE* hypothesis (every block cell and the next-block lead cell $\neq LE4$) feeds both loops.

Relation-generic write proper-prefix scaffold. The two loop helpers (back, forward) are the *leaf_back* and *leaf_fwd* hypotheses; the union- and sub-relation prefixes are thin instantiations.

lemma *ar_write_proper_prefix_gen*:

fixes $M :: ('q, 'a) \text{ mttm}$

and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$

and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config rel}$

assumes *leaf_back*:

$$\begin{aligned}
& \wedge cc. \llbracket \text{valid_mttm } M; q \in Q_tm M; 2 \leq \text{block_width } (\Gamma_tm M); \\
& \text{posk } tk \neq \text{AR_AtLE}; \\
& \text{mt_state } cc = (q, \text{AR_SimWrite}, tk, 0, \text{buf}, \text{dvec}, \text{posk}); \\
& \forall k. \text{buf } k \in \Gamma_tm M \cup \{\text{bl_tm } M\}; \\
& \text{mt_pos } cc \text{ } tk = \text{base} + \text{block_width } (\Gamma_tm M); \\
& \wedge m. \llbracket 1 \leq m; m \leq \text{block_width } (\Gamma_tm M) \rrbracket \\
& \implies \text{mt_tape } cc \text{ } tk \text{ } (\text{base} + m) \neq \text{LE4}; \\
& \forall j \geq k_tm M. \text{mt_tape } cc \text{ } j \text{ } (\text{mt_pos } cc \text{ } j) = \text{BLANK4}; \\
& \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm M) \\
& (\text{AR_SimWrite}, tk, 0, \text{buf}, \text{dvec}, \text{posk}) \rrbracket \\
& \implies \exists c1. (cc, c1) \in R' \wedge \wedge \text{block_width } (\Gamma_tm M) \\
& \wedge \text{mt_state } c1 = (q, \text{AR_SimWrite}, tk, \text{block_width } (\Gamma_tm M), \\
& \text{buf}, \text{dvec}, \text{posk}) \\
& \wedge \text{mt_pos } c1 \text{ } tk = \text{base} \\
& \wedge \text{mt_tape } c1 = \text{mt_tape } cc \\
& \wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c1 \text{ } k' = \text{mt_pos } cc \text{ } k')
\end{aligned}$$

and *leaf_fwd*:

$$\begin{aligned}
& \wedge cc. \llbracket \text{valid_mttm } M; q \in Q_tm M; 2 \leq \text{block_width } (\Gamma_tm M); \\
& \text{posk } tk \neq \text{AR_AtLE};
\end{aligned}$$

$mt_state\ cc = (q, AR_SimWrite, tk, block_width\ (\Gamma_tm\ M), buf,$
 $dvec, posk);$
 $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\};$
 $mt_pos\ cc\ tk = base;$
 $\bigwedge m. m < block_width\ (\Gamma_tm\ M) \implies mt_tape\ cc\ tk\ (base + m) \neq$
 $LE4;$
 $\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, block_width\ (\Gamma_tm\ M), buf, dvec, posk) \parallel$
 $\implies \exists c2. (cc, c2) \in R' \wedge (block_width\ (\Gamma_tm\ M) - 1)$
 $\wedge mt_state\ c2 = (q, AR_SimWrite, tk,$
 $block_width\ (\Gamma_tm\ M) + (block_width\ (\Gamma_tm\ M) - 1),$
 $buf, dvec, posk)$
 $\wedge mt_pos\ c2\ tk = base + (block_width\ (\Gamma_tm\ M) - 1)$
 $\wedge mt_tape\ c2\ tk = (\lambda pos.$
 $\quad if\ base \leq pos \wedge pos < base + (block_width\ (\Gamma_tm\ M) - 1)$
 $\quad then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ tk)\ (pos - base)$
 $\quad else\ mt_tape\ cc\ tk\ pos)$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_tape\ c2\ k' = mt_tape\ cc\ k')$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c2\ k' = mt_pos\ cc\ k')$
and vM : $valid_mttm\ M$
and qQ : $q \in Q_tm\ M$
and $kge2$: $2 \leq block_width\ (\Gamma_tm\ M)$
and $poskproper$: $posk\ tk \neq AR_AtLE$
and stg : $mt_state\ c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)$
and buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and pos_base : $mt_pos\ c'\ tk = base + block_width\ (\Gamma_tm\ M)$
and $notLE$: $\bigwedge m. m \leq block_width\ (\Gamma_tm\ M)$
 $\implies mt_tape\ c'\ tk\ (base + m) \neq LE4$
and $pad0$: $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0$: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c2. (c', c2) \in R' \wedge (block_width\ (\Gamma_tm\ M) + (block_width\ (\Gamma_tm\ M) - 1))$
 $\wedge mt_state\ c2 = (q, AR_SimWrite, tk,$
 $block_width\ (\Gamma_tm\ M) + (block_width\ (\Gamma_tm\ M) - 1),$
 $buf, dvec, posk)$
 $\wedge mt_tape\ c2 = (mt_tape\ c')(tk := (\lambda pos.$
 $\quad if\ base \leq pos \wedge pos < base + (block_width\ (\Gamma_tm\ M) - 1)$
 $\quad then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ tk)\ (pos - base)$
 $\quad else\ mt_tape\ c'\ tk\ pos))$
 $\wedge mt_pos\ c2 = (mt_pos\ c')(tk := base + (block_width\ (\Gamma_tm\ M) -$
 $1))$
proof –
let $?k = block_width\ (\Gamma_tm\ M)$
let $?R = R'$
have $notLE_back$: $\bigwedge m. \parallel 1 \leq m; m \leq ?k \parallel$
 $\implies mt_tape\ c'\ tk\ (base + m) \neq LE4$
using $notLE$ **by** $blast$


```

obtain  $c1$  where
   $ch1: (c', c1) \in ?R \wedge ?k$ 
  and  $st1: mt\_state\ c1 = (q, AR\_SimWrite, tk, ?k, buf, dvec, posk)$ 
  and  $pos1: mt\_pos\ c1\ tk = base$ 
  and  $tape1: mt\_tape\ c1 = mt\_tape\ c'$ 
  and  $opos1: \forall k'. k' \neq tk \longrightarrow mt\_pos\ c1\ k' = mt\_pos\ c'\ k'$ 
  using  $leaf\_back[OF\ vM\ qQ\ kge2\ poskproper\ stg\ buf\_valid\ pos\_base\ notLE\_back$ 
     $pad0\ src0]$ 
  by blast
have  $notLE\_fwd: \bigwedge m. m < ?k \implies mt\_tape\ c1\ tk\ (base + m) \neq LE4$ 
proof -
  fix  $m :: nat$  assume  $mlt: m < ?k$ 
  have  $mle: m \leq ?k$  using  $mlt$  by simp
  show  $mt\_tape\ c1\ tk\ (base + m) \neq LE4$ 
    using  $notLE[OF\ mle]\ tape1$  by simp
qed
have  $tk\_lt: tk < k\_tm\ M$  using  $src0$  by (simp add: ar\_stage\_bounded\_def)
have  $pad\_c1: \forall j \geq k\_tm\ M. mt\_tape\ c1\ j\ (mt\_pos\ c1\ j) = BLANK4$ 
proof (intro allI impI)
  fix  $j$  assume  $jge: k\_tm\ M \leq j$ 
  have  $jne: j \neq tk$  using  $jge\ tk\_lt$  by simp
  have  $mt\_tape\ c1\ j\ (mt\_pos\ c1\ j) = mt\_tape\ c'\ j\ (mt\_pos\ c'\ j)$ 
    using  $tape1\ opos1\ jne$  by simp
  thus  $mt\_tape\ c1\ j\ (mt\_pos\ c1\ j) = BLANK4$  using  $pad0\ jge$  by simp
qed
have  $src\_c1: ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
   $(AR\_SimWrite, tk, block\_width\ (\Gamma\_tm\ M), buf, dvec, posk)$ 
  using  $src0$  by (simp add: ar\_stage\_bounded\_def)
obtain  $c2$  where
   $ch2: (c1, c2) \in ?R \wedge (?k - 1)$ 
  and  $st2: mt\_state\ c2 = (q, AR\_SimWrite, tk, ?k + (?k - 1), buf, dvec, posk)$ 
  and  $pos2: mt\_pos\ c2\ tk = base + (?k - 1)$ 
  and  $tape2: mt\_tape\ c2\ tk = (\lambda pos.$ 
     $if\ base \leq pos \wedge pos < base + (?k - 1)$ 
     $then\ write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (buf\ tk)\ (pos - base)$ 
     $else\ mt\_tape\ c1\ tk\ pos)$ 
  and  $otape2: \forall k'. k' \neq tk \longrightarrow mt\_tape\ c2\ k' = mt\_tape\ c1\ k'$ 
  and  $opos2: \forall k'. k' \neq tk \longrightarrow mt\_pos\ c2\ k' = mt\_pos\ c1\ k'$ 
  using  $leaf\_fwd[OF\ vM\ qQ\ kge2\ poskproper\ st1\ buf\_valid\ pos1\ notLE\_fwd$ 
     $pad\_c1\ src\_c1]$ 
  by blast
have  $ch12: (c', c2) \in ?R \wedge (?k + (?k - 1))$ 
proof -
  have  $(c', c2) \in ?R \wedge ?k\ O\ ?R \wedge (?k - 1)$  using  $ch1\ ch2$  by (rule relcompI)
  thus  $?thesis$  by (simp add: relpow\_add)
qed
have  $tape\_final: mt\_tape\ c2 = (mt\_tape\ c')(tk := (\lambda pos.$ 
   $if\ base \leq pos \wedge pos < base + (?k - 1)$ 
   $then\ write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (buf\ tk)\ (pos - base)$ 

```

```

      else mt_tape c' tk pos))
proof (rule ext)
  fix k'
  show mt_tape c2 k' = ((mt_tape c')(tk := (λpos.
    if base ≤ pos ∧ pos < base + (?k - 1)
    then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
    else mt_tape c' tk pos))) k'
proof (cases k' = tk)
  case True
  have mt_tape c2 tk = (λpos.
    if base ≤ pos ∧ pos < base + (?k - 1)
    then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
    else mt_tape c' tk pos)
  proof (rule ext)
    fix pos
    show mt_tape c2 tk pos = (if base ≤ pos ∧ pos < base + (?k - 1)
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
      else mt_tape c' tk pos)
    using tape2 tape1 by simp
  qed
  thus ?thesis using True by simp
next
  case False
  have mt_tape c2 k' = mt_tape c1 k' using otape2 False by simp
  also have ... = mt_tape c' k' using tape1 by simp
  finally show ?thesis using False by simp
qed
qed
have pos_final: mt_pos c2 = (mt_pos c')(tk := base + (?k - 1))
proof (rule ext)
  fix k'
  show mt_pos c2 k' = ((mt_pos c')(tk := base + (?k - 1))) k'
proof (cases k' = tk)
  case True thus ?thesis using pos2 by simp
next
  case False
  have mt_pos c2 k' = mt_pos c1 k' using opos2 False by simp
  also have ... = mt_pos c' k' using opos1 False by simp
  finally show ?thesis using False by simp
qed
qed
show ?thesis
proof (intro exI[where x = c2] conjI)
  show (c', c2) ∈ ?R ^^ (?k + (?k - 1)) by (rule ch12)
  show mt_state c2 = (q, AR_SimWrite, tk, ?k + (?k - 1), buf, dvec, posk)
    by (rule st2)
  show mt_tape c2 = (mt_tape c')(tk := (λpos.
    if base ≤ pos ∧ pos < base + (?k - 1)
    then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)

```

```

      else mt_tape c' tk pos))
    by (rule tape_final)
  show mt_pos c2 = (mt_pos c')(tk := base + (?k - 1)) by (rule pos_final)
qed
qed

lemma ar_write_proper_prefix:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and poskproper: posk tk ≠ AR_AtLE
  and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and pos_base: mt_pos c' tk = base + block_width (Γ_tm M)
  and notLE: ∧ m. m ≤ block_width (Γ_tm M)
    ⇒ mt_tape c' tk (base + m) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, tk, 0, buf, dvec, posk)
  shows ∃ c2. (c', c2) ∈ (mttm_step (alphabet_reduce_delta M))
    ^ (block_width (Γ_tm M) + (block_width (Γ_tm M) -
1))
    ∧ mt_state c2 = (q, AR_SimWrite, tk,
      block_width (Γ_tm M) + (block_width (Γ_tm M) - 1),
      buf, dvec, posk)
    ∧ mt_tape c2 = (mt_tape c')(tk := (λ pos.
      if base ≤ pos ∧ pos < base + (block_width (Γ_tm M) - 1)
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
      else mt_tape c' tk pos))
    ∧ mt_pos c2 = (mt_pos c')(tk := base + (block_width (Γ_tm M) -
1))
  by (rule ar_write_proper_prefix_gen
    [OF ar_write_back_loop ar_write_fwd_loop
      vM qQ kge2 poskproper stg buf_valid pos_base notLE pad0 src0])

```

Extend a written block prefix by its last cell. Both proper-write finishers reach a tape whose first $b - 1$ block cells already hold the new *write_bit* image; the boundary / finish step writes the last cell $base + (b - 1)$. This *fun_upd* closes the gap: updating the $b - 1$ -prefix function at $base + (b - 1)$ with $f (b - 1)$ yields the full b -cell write. Pure list/function arithmetic, shared by *ar_write_proper_step* and *ar_write_proper_finish_step*.

```

lemma write_block_extend:
  fixes k base :: nat and f g :: nat ⇒ sym4
  assumes 0 < k
  shows ((λ pos. if base ≤ pos ∧ pos < base + (k - 1)
    then f (pos - base) else g pos)
    (base + (k - 1) := f (k - 1)))

```

```

      = ( $\lambda pos.$  if  $base \leq pos \wedge pos < base + k$ 
          then  $f (pos - base)$  else  $g pos$ )
proof -
  obtain  $kk$  where  $k: k = Suc\ kk$  using  $assms$  by ( $cases\ k$ )  $auto$ 
  show  $?thesis$ 
  proof ( $rule\ ext$ )
    fix  $pos$ 
    show ( $(\lambda pos.$  if  $base \leq pos \wedge pos < base + (k - 1)$ 
              then  $f (pos - base)$  else  $g pos$ )
            ( $base + (k - 1) := f (k - 1)$ ))  $pos$ 
      = ( $if\ base \leq pos \wedge pos < base + k$  then  $f (pos - base)$  else  $g pos$ )
  proof ( $cases\ pos = base + (k - 1)$ )
    case  $True$ 
      thus  $?thesis$  using  $k$  by  $simp$ 
    next
      case  $False$ 
      thus  $?thesis$  using  $k$  by ( $auto\ simp: less\_Suc\_eq$ )
  qed
qed
qed

```

The proper-cell single-tape write, non-last tape: the prefix followed by the bit-boundary step ($ar_write_bit_boundary_step$), writing the last block cell $base + (b - 1)$ and landing at the next tape's $AR_SimWrite (k_succ\ tk, counter\ 0)$. After the $2b$ -step walk the b -cell block at $base$ spells out $write_bit (buf\ tk)$ (the encoded new symbol), every other cell unchanged, and the head returns to the block end $base + b$. Chain length $2b$.

Relation-generic write proper-step scaffold. The prefix walk and the closing bit-boundary step are the $leaf_prefix$ and $leaf_boundary$ hypotheses; union and sub versions are thin instantiations.

```

lemma  $ar\_write\_proper\_step\_gen$ :
  fixes  $M :: ('q, 'a)\ mttm$ 
    and  $c' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config$ 
    and  $R' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config\ rel$ 
  assumes  $leaf\_prefix$ :
     $\bigwedge cc. \llbracket valid\_mttm\ M; q \in Q\_tm\ M; 2 \leq block\_width\ (\Gamma\_tm\ M);$ 
       $posk\ tk \neq AR\_AtLE;$ 
       $mt\_state\ cc = (q, AR\_SimWrite, tk, 0, buf, dvec, posk);$ 
       $\forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\};$ 
       $mt\_pos\ cc\ tk = base + block\_width\ (\Gamma\_tm\ M);$ 
       $\bigwedge m. m \leq block\_width\ (\Gamma\_tm\ M)$ 
         $\implies mt\_tape\ cc\ tk\ (base + m) \neq LE4;$ 
       $\forall j \geq k\_tm\ M. mt\_tape\ cc\ j\ (mt\_pos\ cc\ j) = BLANK4;$ 
       $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
         $(AR\_SimWrite, tk, 0, buf, dvec, posk) \rrbracket$ 
     $\implies \exists c2. (cc, c2) \in R' \wedge (block\_width\ (\Gamma\_tm\ M) + (block\_width$ 
       $(\Gamma\_tm\ M) - 1))$ 
       $\wedge mt\_state\ c2 = (q, AR\_SimWrite, tk,$ 

```

$block_width (\Gamma_tm M) + (block_width (\Gamma_tm M) - 1),$
 $buf, dvec, posk)$
 $\wedge mt_tape\ c2 = (mt_tape\ cc)(tk := (\lambda pos.$
 $\quad if\ base \leq pos \wedge pos < base + (block_width (\Gamma_tm M) - 1)$
 $\quad then\ write_bit (\Gamma_tm M) (bl_tm M) (buf\ tk) (pos - base)$
 $\quad else\ mt_tape\ cc\ tk\ pos))$
 $\wedge mt_pos\ c2 = (mt_pos\ cc)(tk := base + (block_width (\Gamma_tm$
 $M) - 1))$
and $leaf_boundary:$
 $\wedge cc\ i. \llbracket valid_mttm\ M;$
 $\quad mt_state\ cc = (q, AR_SimWrite, tk, i, buf, dvec, posk);$
 $\quad q \in Q_tm\ M;$
 $\quad posk\ tk \neq AR_AtLE;$
 $\quad mt_tape\ cc\ tk\ (mt_pos\ cc\ tk) \neq LE4;$
 $\quad \neg is_last_k\ M\ tk;$
 $\quad Suc\ i = 2 * block_width (\Gamma_tm M);$
 $\quad ar_valid_stage (\Gamma_tm M) (bl_tm M)$
 $\quad (AR_SimWrite, tk, i, buf, dvec, posk);$
 $\quad \forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $\quad ar_stage_bounded (bl_tm M) (k_tm M)$
 $\quad (AR_SimWrite, tk, i, buf, dvec, posk) \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\quad \wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec,$
 $posk)$
 $\quad \wedge mt_tape\ c'' = (mt_tape\ cc)(tk :=$
 $\quad (mt_tape\ cc\ tk)(mt_pos\ cc\ tk :=$
 $\quad \quad write_bit (\Gamma_tm M) (bl_tm M) (buf\ tk)$
 $\quad \quad (i - block_width (\Gamma_tm M))))$
 $\quad \wedge mt_pos\ c'' = (mt_pos\ cc)(tk := Suc\ (mt_pos\ cc\ tk))$
and $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width (\Gamma_tm M)$
and $notlast: \neg is_last_k\ M\ tk$
and $poskproper: posk\ tk \neq AR_AtLE$
and $stg: mt_state\ c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)$
and $buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $pos_base: mt_pos\ c'\ tk = base + block_width (\Gamma_tm M)$
and $notLE: \bigwedge m. m \leq block_width (\Gamma_tm M)$
 $\implies mt_tape\ c'\ tk\ (base + m) \neq LE4$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0: ar_stage_bounded (bl_tm M) (k_tm M)$
 $\quad (AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in R' \wedge (2 * block_width (\Gamma_tm M))$
 $\quad \wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec, posk)$
 $\quad \wedge mt_tape\ c'' = (mt_tape\ c')(tk := (\lambda pos.$
 $\quad \quad if\ base \leq pos \wedge pos < base + block_width (\Gamma_tm M)$
 $\quad \quad then\ write_bit (\Gamma_tm M) (bl_tm M) (buf\ tk) (pos - base)$
 $\quad \quad else\ mt_tape\ c'\ tk\ pos))$
 $\quad \wedge mt_pos\ c'' = (mt_pos\ c')(tk := base + block_width (\Gamma_tm M))$

```

proof –
  let ?k = block_width (Γ_tm M)
  let ?R = R'
  let ?wb = λm. write_bit (Γ_tm M) (bl_tm M) (buf tk) m
  let ?part = λpos. if base ≤ pos ∧ pos < base + (?k - 1)
    then ?wb (pos - base) else mt_tape c' tk pos
  let ?full = λpos. if base ≤ pos ∧ pos < base + ?k
    then ?wb (pos - base) else mt_tape c' tk pos
  have kpos: 0 < ?k using kge2 by simp
  obtain kk where kdef: ?k = Suc kk using kge2 by (cases ?k) auto
  obtain c2 where
    ch_pre: (c', c2) ∈ ?R ^ ( ?k + (?k - 1) )
    and st2: mt_state c2 = (q, AR_SimWrite, tk, ?k + (?k - 1), buf, dvec, posk)
    and tape2: mt_tape c2 = (mt_tape c')(tk := ?part)
    and pos2: mt_pos c2 = (mt_pos c')(tk := base + (?k - 1))
    using leaf_prefix[OF vM qQ kge2 poskproper stg buf_valid pos_base notLE
      pad0 src0]
    by blast
  have c2tk: mt_tape c2 tk = ?part using tape2 by simp
  have pos2_tk: mt_pos c2 tk = base + (?k - 1) using pos2 by simp
  have ihi: Suc (?k + (?k - 1)) = 2 * ?k using kdef by simp
  have lt2: ?k + (?k - 1) < 2 * ?k using kdef by simp
  have vsrc2: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimWrite, tk, ?k + (?k - 1), buf, dvec, posk)
    using lt2 buf_valid by (simp add: ar_valid_stage_def)
  have cell_cur: mt_tape c2 tk (mt_pos c2 tk) = mt_tape c' tk (base + (?k -
1))
proof –
  have mt_tape c2 tk (base + (?k - 1)) = mt_tape c' tk (base + (?k - 1))
    using c2tk by simp
  thus ?thesis using pos2_tk by simp
qed
have km1le: ?k - 1 ≤ ?k by simp
have notLE2: mt_tape c2 tk (mt_pos c2 tk) ≠ LE4
  using cell_cur notLE[OF km1le] by simp
have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
have pad_c2: ∀ j ≥ k_tm M. mt_tape c2 j (mt_pos c2 j) = BLANK4
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using jge tk_lt by simp
  have mt_tape c2 j (mt_pos c2 j) = mt_tape c' j (mt_pos c' j)
    using tape2 pos2 jne by simp
  thus mt_tape c2 j (mt_pos c2 j) = BLANK4 using pad0 jge by simp
qed
have src_c2: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, tk, block_width (Γ_tm M) + (block_width (Γ_tm
M) - 1),
    buf, dvec, posk)
  using src0 by (simp add: ar_stage_bounded_def)

```

```

obtain  $c3$  where
   $s3: (c2, c3) \in ?R$ 
  and  $st3: mt\_state\ c3 = (q, AR\_SimWrite, k\_succ\ tk, 0, buf, dvec, posk)$ 
  and  $tape3: mt\_tape\ c3 = (mt\_tape\ c2)(tk :=$ 
     $(mt\_tape\ c2\ tk)(mt\_pos\ c2\ tk := ?wb\ (?k + (?k - 1) - ?k)))$ 
  and  $pos3: mt\_pos\ c3 = (mt\_pos\ c2)(tk := Suc\ (mt\_pos\ c2\ tk))$ 
  using  $leaf\_boundary[OF\ vM\ st2\ qQ\ poskproper\ notLE2\ notlast\ ihi\ vsrc2$ 
     $pad\_c2\ src\_c2]$ 
  by blast
  — chain: prefix  $b + (b - 1)$  then one boundary step =  $2b$ 
  have  $ch\_full: (c', c3) \in ?R \wedge (2 * ?k)$ 
  proof —
    have  $h: (c', c3) \in ?R \wedge Suc\ (?k + (?k - 1))$ 
    by  $(rule\ relpow\_Suc\_I[OF\ ch\_pre\ s3])$ 
    from  $h$  show  $?thesis$  by  $(simp\ only: ihi)$ 
  qed
  — tape: the last block cell  $base + (b - 1)$  completes the write
  have  $tape3\_tk: mt\_tape\ c3\ tk = (?part(base + (?k - 1) := ?wb\ (?k - 1)))$ 
  using  $tape3\ pos2\_tk\ c2tk$  by simp
  have  $tape3\_tk\_full: mt\_tape\ c3\ tk = ?full$ 
  using  $tape3\_tk\ write\_block\_extend[OF\ kpos, of\ base\ ?wb\ mt\_tape\ c'\ tk]$  by
simp
  have  $tape\_final: mt\_tape\ c3 = (mt\_tape\ c')(tk := ?full)$ 
  proof  $(rule\ ext)$ 
    fix  $k'$ 
    show  $mt\_tape\ c3\ k' = ((mt\_tape\ c')(tk := ?full))\ k'$ 
    proof  $(cases\ k' = tk)$ 
      case True thus  $?thesis$  using  $tape3\_tk\_full$  by simp
    next
      case False
      have  $mt\_tape\ c3\ k' = mt\_tape\ c2\ k'$  using  $tape3\ False$  by simp
      also have  $\dots = mt\_tape\ c'\ k'$  using  $tape2\ False$  by simp
      finally show  $?thesis$  using False by simp
    qed
  qed
  have  $pos\_final: mt\_pos\ c3 = (mt\_pos\ c')(tk := base + ?k)$ 
  proof —
    have  $mt\_pos\ c3 = (mt\_pos\ c2)(tk := Suc\ (mt\_pos\ c2\ tk))$  by  $(rule\ pos3)$ 
    also have  $\dots = (mt\_pos\ c2)(tk := base + ?k)$  using  $pos2\_tk\ kdef$  by simp
    also have  $\dots = (mt\_pos\ c')(tk := base + ?k)$  using  $pos2$  by simp
    finally show  $?thesis$  .
  qed
  show  $?thesis$ 
  proof  $(intro\ exI[where\ x = c3]\ conjI)$ 
    show  $(c', c3) \in R' \wedge (2 * ?k)$ 
    by  $(rule\ ch\_full)$ 
    show  $mt\_state\ c3 = (q, AR\_SimWrite, k\_succ\ tk, 0, buf, dvec, posk)$ 
    by  $(rule\ st3)$ 
    show  $mt\_tape\ c3 = (mt\_tape\ c')(tk := (\lambda pos.$ 

```

```

      if base ≤ pos ∧ pos < base + ?k
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
      else mt_tape c' tk pos))
    by (rule tape_final)
  show mt_pos c3 = (mt_pos c')(tk := base + ?k) by (rule pos_final)
qed
qed

```

```

lemma ar_write_proper_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and notlast: ¬ is_last_k M tk
  and poskproper: posk tk ≠ AR_AtLE
  and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and pos_base: mt_pos c' tk = base + block_width (Γ_tm M)
  and notLE: ∧ m. m ≤ block_width (Γ_tm M)
    ⇒ mt_tape c' tk (base + m) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, tk, 0, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^ (2 * block_width (Γ_tm M))
    ∧ mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
    ∧ mt_tape c'' = (mt_tape c')(tk := (λ pos.
      if base ≤ pos ∧ pos < base + block_width (Γ_tm M)
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
      else mt_tape c' tk pos))
    ∧ mt_pos c'' = (mt_pos c')(tk := base + block_width (Γ_tm M))
  by (rule ar_write_proper_step_gen
    [OF ar_write_proper_prefix ar_write_bit_boundary_step
      vM qQ kge2 notlast poskproper stg buf_valid pos_base notLE
      pad0 src0])

```

The proper-cell single-tape write, last tape: as *ar_write_proper_step* but *tk* is the last tape, so after the last-cell write (*ar_write_bit_finish_step*) the phase transitions to *AR_SimAdvance* with the current-tape field reset to *k_unidx 0* (all tapes written). Same *b*-cell block write and head return to *base + b*. Chain length *2b*.

Relation-generic write proper-finish-step scaffold. As *ar_write_proper_step_gen* but the closing leaf is the last-tape finish (transition to *AR_SimAdvance*); the prefix and finish helpers are the *leaf_prefix* and *leaf_finish* hypotheses.

```

lemma ar_write_proper_finish_step_gen:
  fixes M :: ('q, 'a) mttm

```


and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config rel}$
assumes leaf_prefix :
 $\bigwedge cc. \llbracket \text{valid_mttm } M; q \in Q_tm \ M; 2 \leq \text{block_width } (\Gamma_tm \ M);$
 $\text{posk } tk \neq \text{AR_AtLE};$
 $\text{mt_state } cc = (q, \text{AR_SimWrite}, tk, 0, \text{buf}, \text{dvec}, \text{posk});$
 $\forall k. \text{buf } k \in \Gamma_tm \ M \cup \{\text{bl_tm } M\};$
 $\text{mt_pos } cc \ tk = \text{base} + \text{block_width } (\Gamma_tm \ M);$
 $\bigwedge m. m \leq \text{block_width } (\Gamma_tm \ M)$
 $\implies \text{mt_tape } cc \ tk \ (\text{base} + m) \neq \text{LE4};$
 $\forall j \geq k_tm \ M. \text{mt_tape } cc \ j \ (\text{mt_pos } cc \ j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \ M)$
 $(\text{AR_SimWrite}, tk, 0, \text{buf}, \text{dvec}, \text{posk}) \rrbracket$
 $\implies \exists c2. (cc, c2) \in R' \wedge (block_width (\Gamma_tm \ M) + (block_width$
 $(\Gamma_tm \ M) - 1))$
 $\wedge \text{mt_state } c2 = (q, \text{AR_SimWrite}, tk,$
 $\text{block_width } (\Gamma_tm \ M) + (block_width (\Gamma_tm \ M) - 1),$
 $\text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c2 = (\text{mt_tape } cc)(tk := (\lambda pos.$
 $\text{if } \text{base} \leq pos \wedge pos < \text{base} + (block_width (\Gamma_tm \ M) - 1)$
 $\text{then write_bit } (\Gamma_tm \ M) (\text{bl_tm } M) (\text{buf } tk) (pos - \text{base})$
 $\text{else mt_tape } cc \ tk \ pos))$
 $\wedge \text{mt_pos } c2 = (\text{mt_pos } cc)(tk := \text{base} + (block_width (\Gamma_tm$
 $M) - 1))$
and leaf_finish :
 $\bigwedge cc \ i. \llbracket \text{valid_mttm } M;$
 $\text{mt_state } cc = (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk});$
 $q \in Q_tm \ M;$
 $\text{posk } tk \neq \text{AR_AtLE};$
 $\text{mt_tape } cc \ tk \ (\text{mt_pos } cc \ tk) \neq \text{LE4};$
 $\text{is_last_k } M \ tk;$
 $\text{Suc } i = 2 * \text{block_width } (\Gamma_tm \ M);$
 $\text{ar_valid_stage } (\Gamma_tm \ M) (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk});$
 $\forall j \geq k_tm \ M. \text{mt_tape } cc \ j \ (\text{mt_pos } cc \ j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \ M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk}) \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimAdvance}, k_unidx \ 0, 0, \text{buf},$
 $\text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (\text{mt_tape } cc)(tk :=$
 $(\text{mt_tape } cc \ tk)(\text{mt_pos } cc \ tk :=$
 $\text{write_bit } (\Gamma_tm \ M) (\text{bl_tm } M) (\text{buf } tk)$
 $(i - \text{block_width } (\Gamma_tm \ M))))$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } cc)(tk := \text{Suc } (\text{mt_pos } cc \ tk))$
and $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $\text{last}: \text{is_last_k } M \ tk$

```

and poskproper: posk tk ≠ AR_AtLE
and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
and pos_base: mt_pos c' tk = base + block_width (Γ_tm M)
and notLE: ∧ m. m ≤ block_width (Γ_tm M)
              ⇒ mt_tape c' tk (base + m) ≠ LE4
and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
and src0: ar_stage_bounded (bl_tm M) (k_tm M)
              (AR_SimWrite, tk, 0, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ R' ^ (2 * block_width (Γ_tm M))
              ∧ mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
              ∧ mt_tape c'' = (mt_tape c')(tk := (λ pos.
                  if base ≤ pos ∧ pos < base + block_width (Γ_tm M)
                  then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
                  else mt_tape c' tk pos))
              ∧ mt_pos c'' = (mt_pos c')(tk := base + block_width (Γ_tm M))
proof -
let ?k = block_width (Γ_tm M)
let ?R = R'
let ?wb = λ m. write_bit (Γ_tm M) (bl_tm M) (buf tk) m
let ?part = λ pos. if base ≤ pos ∧ pos < base + (?k - 1)
                  then ?wb (pos - base) else mt_tape c' tk pos
let ?full = λ pos. if base ≤ pos ∧ pos < base + ?k
                  then ?wb (pos - base) else mt_tape c' tk pos
have kpos: 0 < ?k using kge2 by simp
obtain kk where kdef: ?k = Suc kk using kge2 by (cases ?k) auto
obtain c2 where
  ch_pre: (c', c2) ∈ ?R ^ (?k + (?k - 1))
  and st2: mt_state c2 = (q, AR_SimWrite, tk, ?k + (?k - 1), buf, dvec, posk)
  and tape2: mt_tape c2 = (mt_tape c')(tk := ?part)
  and pos2: mt_pos c2 = (mt_pos c')(tk := base + (?k - 1))
  using leaf_prefix[OF vM qQ kge2 poskproper stg buf_valid pos_base notLE
    pad0 src0]
  by blast
have c2tk: mt_tape c2 tk = ?part using tape2 by simp
have pos2_tk: mt_pos c2 tk = base + (?k - 1) using pos2 by simp
have ihi: Suc (?k + (?k - 1)) = 2 * ?k using kdef by simp
have lt2: ?k + (?k - 1) < 2 * ?k using kdef by simp
have vsrc2: ar_valid_stage (Γ_tm M) (bl_tm M)
              (AR_SimWrite, tk, ?k + (?k - 1), buf, dvec, posk)
  using lt2 buf_valid by (simp add: ar_valid_stage_def)
have cell_cur: mt_tape c2 tk (mt_pos c2 tk) = mt_tape c' tk (base + (?k -
1))
proof -
  have mt_tape c2 tk (base + (?k - 1)) = mt_tape c' tk (base + (?k - 1))
    using c2tk by simp
  thus ?thesis using pos2_tk by simp
qed
have km1le: ?k - 1 ≤ ?k by simp

```

```

have notLE2: mt_tape c2 tk (mt_pos c2 tk) ≠ LE4
  using cell_cur notLE[OF km1le] by simp
have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
have pad_c2: ∀ j ≥ k_tm M. mt_tape c2 j (mt_pos c2 j) = BLANK4
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j ≠ tk using jge tk_lt by simp
  have mt_tape c2 j (mt_pos c2 j) = mt_tape c' j (mt_pos c' j)
    using tape2 pos2 jne by simp
  thus mt_tape c2 j (mt_pos c2 j) = BLANK4 using pad0 jge by simp
qed
have src_c2: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, tk, block_width (Γ_tm M) + (block_width (Γ_tm
M) - 1),
    buf, dvec, posk)
  using src0 by (simp add: ar_stage_bounded_def)
obtain c3 where
  s3: (c2, c3) ∈ ?R
  and st3: mt_state c3 = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
  and tape3: mt_tape c3 = (mt_tape c2)(tk :=
    (mt_tape c2 tk)(mt_pos c2 tk := ?wb (?k + (?k - 1) - ?k)))
  and pos3: mt_pos c3 = (mt_pos c2)(tk := Suc (mt_pos c2 tk))
  using leaf_finish[OF vM st2 qQ poskproper notLE2 last ihi vsrc2
    pad_c2 src_c2]
  by blast
have ch_full: (c', c3) ∈ ?R ^^ (2 * ?k)
proof -
  have h: (c', c3) ∈ ?R ^^ Suc (?k + (?k - 1))
    by (rule relpow_Suc_I[OF ch_pre s3])
  from h show ?thesis by (simp only: ihi)
qed
have tape3_tk: mt_tape c3 tk = (?part(base + (?k - 1) := ?wb (?k - 1)))
  using tape3 pos2_tk c2tk by simp
have tape3_tk_full: mt_tape c3 tk = ?full
  using tape3_tk write_block_extend[OF kpos, of base ?wb mt_tape c' tk] by
simp
have tape_final: mt_tape c3 = (mt_tape c')(tk := ?full)
proof (rule ext)
  fix k'
  show mt_tape c3 k' = ((mt_tape c')(tk := ?full)) k'
  proof (cases k' = tk)
    case True thus ?thesis using tape3_tk_full by simp
  next
    case False
    have mt_tape c3 k' = mt_tape c2 k' using tape3 False by simp
    also have ... = mt_tape c' k' using tape2 False by simp
    finally show ?thesis using False by simp
  qed
qed

```

```

have pos_final: mt_pos c3 = (mt_pos c')(tk := base + ?k)
proof -
  have mt_pos c3 = (mt_pos c2)(tk := Suc (mt_pos c2 tk)) by (rule pos3)
  also have ... = (mt_pos c2)(tk := base + ?k) using pos2 tk kdef by simp
  also have ... = (mt_pos c')(tk := base + ?k) using pos2 by simp
  finally show ?thesis .
qed
show ?thesis
proof (intro exI[where x = c3] conjI)
  show (c', c3) ∈ R' ^^ (2 * ?k)
    by (rule ch_full)
  show mt_state c3 = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
    by (rule st3)
  show mt_tape c3 = (mt_tape c')(tk := (λpos.
    if base ≤ pos ∧ pos < base + ?k
    then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
    else mt_tape c' tk pos))
    by (rule tape_final)
  show mt_pos c3 = (mt_pos c')(tk := base + ?k) by (rule pos_final)
qed
qed

lemma ar_write_proper_finish_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and last: is_last_k M tk
  and poskproper: posk tk ≠ AR_AtLE
  and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and pos_base: mt_pos c' tk = base + block_width (Γ_tm M)
  and notLE: ∧ m. m ≤ block_width (Γ_tm M)
    ⇒ mt_tape c' tk (base + m) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, tk, 0, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^^ (2 * block_width (Γ_tm M))
    ∧ mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
    ∧ mt_tape c'' = (mt_tape c')(tk := (λpos.
      if base ≤ pos ∧ pos < base + block_width (Γ_tm M)
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - base)
      else mt_tape c' tk pos))
    ∧ mt_pos c'' = (mt_pos c')(tk := base + block_width (Γ_tm M))
  by (rule ar_write_proper_finish_step_gen
    [OF ar_write_proper_prefix ar_write_bit_finish_step
      vM qQ kge2 last poskproper stg buf_valid pos_base notLE

```

pad0 src0])

The unified single-tape write, non-last tape: the LE/proper dispatch, the AR analogue of *ar_read_tape_step*. From an *AR_SimWrite* stage at bit-counter *0*, dispatch on *posk tk = AR_AtLE* (which the caller pins to $p = 0$, *M*'s head on the marker): the LE arm (*ar_write_le_step*, 1 step) leaves the tape and head untouched and hands off; the proper arm (*ar_write_proper_step*, 2*b* steps) overwrites the *b*-cell block at *sim_pos p* with *write_bit (buf tk)*. In both arms every head is unchanged (the LE arm is all-*N*; the proper arm back-walks then returns to the block end), so the conclusion is $mt_pos\ c'' = mt_pos\ c'$ uniformly. The proper-arm $\neq LE4$ facts come from the input correspondence *tcorr* (block-*p* cells and the block- $(p+1)$ lead cell, all *cell_repr* cells via *cell_repr_nth_not_LE4*).

Relation-generic write single-tape step scaffold (non-last). The two cases ($p = 0$ le-step, $p \neq 0$ proper-step) are the *leaf_le* and *leaf_proper* hypotheses; union and sub versions are thin instantiations.

lemma *ar_write_tape_step_gen*:

fixes *M* :: ('q, 'a) mttm

and *c'* :: (sym4, 'q $\times$ 'a ar_stage) mt_config

and *tM* :: nat $\Rightarrow$ 'a

and *R'* :: (sym4, 'q $\times$ 'a ar_stage) mt_config rel

assumes *leaf_le*:

$\bigwedge cc. \llbracket \text{valid_mttm } M;$

$mt_state\ cc = (q, AR_SimWrite, tk, 0, buf, dvec, posk);$

$q \in Q_tm\ M;$

$posk\ tk = AR_AtLE;$

$\neg is_last_k\ M\ tk;$

$ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$

$(AR_SimWrite, tk, 0, buf, dvec, posk);$

$\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$

$ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$

$(AR_SimWrite, tk, 0, buf, dvec, posk) \rrbracket$

$\implies \exists c''. (cc, c'') \in R'$

$\wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec,$

posk)

$\wedge mt_tape\ c'' = mt_tape\ cc$

$\wedge mt_pos\ c'' = mt_pos\ cc$

and *leaf_proper*:

$\bigwedge cc\ base'. \llbracket \text{valid_mttm } M; q \in Q_tm\ M; 2 \leq block_width\ (\Gamma_tm\ M);$

$\neg is_last_k\ M\ tk;$

$posk\ tk \neq AR_AtLE;$

$mt_state\ cc = (q, AR_SimWrite, tk, 0, buf, dvec, posk);$

$\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\};$

$mt_pos\ cc\ tk = base' + block_width\ (\Gamma_tm\ M);$

$\bigwedge m. m \leq block_width\ (\Gamma_tm\ M)$

$\implies mt_tape\ cc\ tk\ (base' + m) \neq LE4;$

$\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$

$$\begin{aligned} & \text{ar_stage_bounded } (bl_tm\ M) (k_tm\ M) \\ & (AR_SimWrite, tk, 0, buf, dvec, posk) \mathbb{I} \\ \implies & \exists c''. (cc, c'') \in R' \wedge (2 * block_width\ (\Gamma_tm\ M)) \\ & \wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, \\ & dvec, posk) \\ & \wedge mt_tape\ c'' = (mt_tape\ cc)(tk := (\lambda pos. \\ & \text{if } base' \leq pos \wedge pos < base' + block_width\ (\Gamma_tm\ M) \\ & \text{then write_bit } (\Gamma_tm\ M) (bl_tm\ M) (buf\ tk) (pos - \\ & base') \\ & \text{else } mt_tape\ cc\ tk\ pos)) \\ & \wedge mt_pos\ c'' = (mt_pos\ cc)(tk := base' + block_width \\ & (\Gamma_tm\ M)) \\ & \text{and } vM: \text{valid_mttm } M \\ & \text{and } qQ: q \in \bar{Q}_tm\ M \\ & \text{and } kge2: 2 \leq block_width\ (\Gamma_tm\ M) \\ & \text{and } notlast: \neg is_last_k\ M\ tk \\ & \text{and } stg: mt_state\ c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk) \\ & \text{and } tcorr: ar_tape_correspondence\ (\Gamma_tm\ M) (le_tm\ M) (bl_tm\ M) \\ & \quad tM\ (mt_tape\ c'\ tk) \\ & \text{and } ppos: mt_pos\ c'\ tk = (\text{if } p = 0 \text{ then } Suc\ 0 \\ & \quad \text{else } sim_pos\ (block_width\ (\Gamma_tm\ M))\ p + block_width\ (\Gamma_tm \\ & M)) \\ & \text{and } poskle: posk\ tk = AR_AtLE \longleftrightarrow p = 0 \\ & \text{and } buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\} \\ & \text{and } vsrc: ar_valid_stage\ (\Gamma_tm\ M) (bl_tm\ M) \\ & \quad (AR_SimWrite, tk, 0, buf, dvec, posk) \\ & \text{and } pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4 \\ & \text{and } src0: ar_stage_bounded\ (bl_tm\ M) (k_tm\ M) \\ & \quad (AR_SimWrite, tk, 0, buf, dvec, posk) \\ & \text{shows } \exists c''. (c', c'') \in R' \wedge (\text{if } p = 0 \text{ then } 1 \text{ else } 2 * block_width\ (\Gamma_tm\ M)) \\ & \quad \wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec, posk) \\ & \quad \wedge mt_tape\ c'' = (\text{if } p = 0 \text{ then } mt_tape\ c' \\ & \quad \text{else } (mt_tape\ c')(tk := (\lambda pos. \\ & \quad \text{if } sim_pos\ (block_width\ (\Gamma_tm\ M))\ p \leq pos \\ & \quad \wedge pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ p + block_width \\ & (\Gamma_tm\ M) \\ & \quad \text{then write_bit } (\Gamma_tm\ M) (bl_tm\ M) (buf\ tk) \\ & \quad (pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ p) \\ & \quad \text{else } mt_tape\ c'\ tk\ pos))) \\ & \quad \wedge mt_pos\ c'' = mt_pos\ c' \\ & \text{proof -} \\ & \text{let } ?k = block_width\ (\Gamma_tm\ M) \\ & \text{let } ?R = R' \\ & \text{have } kpos: 0 < ?k \text{ using } kge2 \text{ by simp} \\ & \text{show } ?thesis \\ & \text{proof (cases } p = 0) \\ & \text{case True} \\ & \text{have } poskle: posk\ tk = AR_AtLE \text{ using } poskle\ True \text{ by simp} \\ & \text{obtain } c'' \text{ where}
\end{aligned}$$

```

    step: (c', c'') ∈ ?R
  and st: mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
  and tp: mt_tape c'' = mt_tape c'
  and ps: mt_pos c'' = mt_pos c'
  using leaf_le[OF vM stg qQ poskLE notlast vsrc pad0 src0] by blast
show ?thesis
proof (intro exI[where x = c''] conjI)
  show (c', c'') ∈ ?R ^^ (if p = 0 then 1 else 2 * ?k)
    using step True by simp
  show mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
    by (rule st)
  show mt_tape c'' = (if p = 0 then mt_tape c'
    else (mt_tape c')(tk := (λpos.
      if sim_pos ?k p ≤ pos ∧ pos < sim_pos ?k p + ?k
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - sim_pos ?k p)
      else mt_tape c' tk pos)))
    using tp True by simp
  show mt_pos c'' = mt_pos c' by (rule ps)
qed
next
case False
hence pge1: 1 ≤ p by simp
have poskproper: posk tk ≠ AR_AtLE using poskle False by simp
let ?base = sim_pos ?k p
have pos_base: mt_pos c' tk = ?base + ?k using ppos False by simp
have notLE: ∧m. m ≤ ?k ⇒ mt_tape c' tk (?base + m) ≠ LE4
proof -
  fix m assume mle: m ≤ ?k
  show mt_tape c' tk (?base + m) ≠ LE4
  proof (cases m < ?k)
    case True
    have mt_tape c' tk (?base + m)
      = cell_repr (Γ_tm M) (bl_tm M) (tm p) ! m
      using tcorr pge1 True unfolding ar_tape_correspondence_def by blast
    thus ?thesis using cell_repr_nth_not_LE4[OF True] by simp
  next
    case False
    hence meq: m = ?k using mle by simp
    obtain pp where pp: p = Suc pp using pge1 by (cases p) auto
    have psuc1: 1 ≤ Suc p by simp
    have base_succ: ?base + ?k = sim_pos ?k (Suc p)
      using pp by (simp add: sim_pos_def)
    have corr_succ: ∀j < ?k. mt_tape c' tk (sim_pos ?k (Suc p) + j)
      = cell_repr (Γ_tm M) (bl_tm M) (tm (Suc p)) ! j
      using tcorr psuc1 unfolding ar_tape_correspondence_def by blast
    have mt_tape c' tk (?base + m)
      = cell_repr (Γ_tm M) (bl_tm M) (tm (Suc p)) ! 0
      using meq base_succ corr_succ[rule_format, OF kpos] by simp
    thus ?thesis using cell_repr_nth_not_LE4[OF kpos] by simp
  end
end

```

```

qed
qed
obtain c'' where
  step: (c', c'') ∈ ?R ^^ (2 * ?k)
  and st: mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
  and tp: mt_tape c'' = (mt_tape c')(tk := (λpos.
    if ?base ≤ pos ∧ pos < ?base + ?k
    then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - ?base)
    else mt_tape c' tk pos))
  and ps: mt_pos c'' = (mt_pos c')(tk := ?base + ?k)
  using leaf_proper[OF vM qQ kge2 notlast poskproper stg buf_valid
    pos_base notLE pad0 src0]

  by blast
have ps': mt_pos c'' = mt_pos c'
proof -
  have (mt_pos c')(tk := ?base + ?k) = mt_pos c'
    using pos_base by (rule fun_upd_idem)
  thus ?thesis using ps by simp
qed
show ?thesis
proof (intro exI[where x = c''] conjI)
  show (c', c'') ∈ ?R ^^ (if p = 0 then 1 else 2 * ?k)
    using step False by simp
  show mt_state c'' = (q, AR_SimWrite, k_succ tk, 0, buf, dvec, posk)
    by (rule st)
  show mt_tape c'' = (if p = 0 then mt_tape c'
    else (mt_tape c')(tk := (λpos.
      if sim_pos ?k p ≤ pos ∧ pos < sim_pos ?k p + ?k
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - sim_pos ?k p)
      else mt_tape c' tk pos)))
    using tp False by simp
  show mt_pos c'' = mt_pos c' by (rule ps')
qed
qed
qed
lemma ar_write_tape_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  and tM :: nat ⇒ 'a
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and notlast: ¬ is_last_k M tk
  and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
  and tcorr: ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
    tM (mt_tape c' tk)
  and ppos: mt_pos c' tk = (if p = 0 then Suc 0
    else sim_pos (block_width (Γ_tm M)) p + block_width (Γ_tm

```


$M))$
and *poskle*: $\text{posk } tk = AR_AtLE \longleftrightarrow p = 0$
and *buf_valid*: $\forall k. \text{buf } k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *vsr*: $\text{ar_valid_stage } (\Gamma_tm\ M) (bl_tm\ M)$
 $(AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
and *pad0*: $\forall j \geq k_tm\ M. \text{mt_tape } c' j (\text{mt_pos } c' j) = BLANK4$
and *src0*: $\text{ar_stage_bounded } (bl_tm\ M) (k_tm\ M)$
 $(AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in (\text{mttm_step } (\text{alphabet_reduce_delta } M))$
 $\wedge (if\ p = 0\ then\ 1\ else\ 2 * \text{block_width } (\Gamma_tm\ M))$
 $\wedge \text{mt_state } c'' = (q, AR_SimWrite, k_succ\ tk, 0, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (if\ p = 0\ then\ \text{mt_tape } c'$
 $\quad else\ (\text{mt_tape } c') (tk := (\lambda pos.$
 $\quad \quad if\ \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ p \leq pos$
 $\quad \quad \wedge\ pos < \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ p + \text{block_width}$
 $(\Gamma_tm\ M)$
 $\quad \quad then\ \text{write_bit } (\Gamma_tm\ M) (bl_tm\ M) (\text{buf } tk)$
 $\quad \quad (\text{pos} - \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ p)$
 $\quad \quad else\ \text{mt_tape } c' tk\ pos)))$
 $\wedge \text{mt_pos } c'' = \text{mt_pos } c'$
by (*rule ar_write_tape_step_gen*
 $[OF\ \text{ar_write_le_step } \text{ar_write_proper_step}$
 $vM\ qQ\ kge2\ \text{notlast } stg\ tcorr\ ppos\ \text{poskle } \text{buf_valid } vsr$
 $\text{pad0 } src0])$

The unified single-tape write, last tape: as *ar_write_tape_step* but *tk* is the last tape, so both arms transition to *AR_SimAdvance* (current-tape field reset to *k_unidx 0*, all tapes written): the LE arm via *ar_write_le_finish_step*, the proper arm via *ar_write_proper_finish_step*. Same tape edit and head invariance.

Relation-generic write single-tape finish-step scaffold (last tape). As *ar_write_tape_step_gen* but both cases use the last-tape finish leaves (transition to *AR_SimAdvance*); the le-finish and proper-finish helpers are the *leaf_le* and *leaf_proper* hypotheses.

lemma *ar_write_tape_finish_step_gen*:

fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a\ \text{ar_stage})\ \text{mt_config}$
and $tM :: \text{nat} \Rightarrow 'a$
and $R' :: (\text{sym4}, 'q \times 'a\ \text{ar_stage})\ \text{mt_config}\ rel$
assumes *leaf_le*:
 $\wedge cc. \llbracket \text{valid_mttm } M;$
 $\text{mt_state } cc = (q, AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk});$
 $q \in Q_tm\ M;$
 $\text{posk } tk = AR_AtLE;$
 $is_last_k\ M\ tk;$
 $\text{ar_valid_stage } (\Gamma_tm\ M) (bl_tm\ M)$
 $(AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk});$
 $\forall j \geq k_tm\ M. \text{mt_tape } cc\ j (\text{mt_pos } cc\ j) = BLANK4;$

$$\begin{aligned}
& ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M) \\
& (AR_SimWrite, tk, 0, buf, dvec, posk) \] \\
\implies & \exists c''. (cc, c'') \in R' \\
& \wedge mt_state\ c'' = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, \\
posk) \\
& \wedge mt_tape\ c'' = mt_tape\ cc \\
& \wedge mt_pos\ c'' = mt_pos\ cc \\
\text{and leaf_proper:} \\
& \wedge cc\ base'. \ [\ valid_mttm\ M; q \in Q_tm\ M; 2 \leq block_width\ (\Gamma_tm\ M); \\
& \quad is_last_k\ M\ tk; \\
& \quad posk\ tk \neq AR_AtLE; \\
& \quad mt_state\ cc = (q, AR_SimWrite, tk, 0, buf, dvec, posk); \\
& \quad \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}; \\
& \quad mt_pos\ cc\ tk = base' + block_width\ (\Gamma_tm\ M); \\
& \quad \wedge m. m \leq block_width\ (\Gamma_tm\ M) \\
& \quad \implies mt_tape\ cc\ tk\ (base' + m) \neq LE4; \\
& \quad \forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4; \\
& \quad ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M) \\
& \quad (AR_SimWrite, tk, 0, buf, dvec, posk) \] \\
\implies & \exists c''. (cc, c'') \in R' \wedge (2 * block_width\ (\Gamma_tm\ M)) \\
& \wedge mt_state\ c'' = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, \\
dvec, posk) \\
& \wedge mt_tape\ c'' = (mt_tape\ cc)(tk := (\lambda pos. \\
& \quad if\ base' \leq pos \wedge pos < base' + block_width\ (\Gamma_tm\ M) \\
& \quad then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ tk)\ (pos - \\
base') \\
& \quad else\ mt_tape\ cc\ tk\ pos)) \\
& \wedge mt_pos\ c'' = (mt_pos\ cc)(tk := base' + block_width \\
(\Gamma_tm\ M)) \\
& \text{and } vM: valid_mttm\ M \\
& \text{and } qQ: q \in Q_tm\ M \\
& \text{and } kge2: 2 \leq block_width\ (\Gamma_tm\ M) \\
& \text{and } last: is_last_k\ M\ tk \\
& \text{and } stg: mt_state\ c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk) \\
& \text{and } tcorr: ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M) \\
& \quad tM\ (mt_tape\ c'\ tk) \\
& \text{and } ppos: mt_pos\ c'\ tk = (if\ p = 0\ then\ Suc\ 0 \\
& \quad else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ p + block_width\ (\Gamma_tm \\
M)) \\
& \text{and } poskle: posk\ tk = AR_AtLE \longleftrightarrow p = 0 \\
& \text{and } buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\} \\
& \text{and } vsrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M) \\
& \quad (AR_SimWrite, tk, 0, buf, dvec, posk) \\
& \text{and } pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4 \\
& \text{and } src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M) \\
& \quad (AR_SimWrite, tk, 0, buf, dvec, posk) \\
\text{shows } & \exists c''. (c', c'') \in R' \wedge (if\ p = 0\ then\ 1\ else\ 2 * block_width\ (\Gamma_tm\ M)) \\
& \wedge mt_state\ c'' = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk) \\
& \wedge mt_tape\ c'' = (if\ p = 0\ then\ mt_tape\ c'
\end{aligned}$$

```

      else (mt_tape c')(tk := (λpos.
        if sim_pos (block_width (Γ_tm M)) p ≤ pos
          ∧ pos < sim_pos (block_width (Γ_tm M)) p + block_width
(Γ_tm M)
          then write_bit (Γ_tm M) (bl_tm M) (buf tk)
            (pos - sim_pos (block_width (Γ_tm M)) p)
          else mt_tape c' tk pos)))
      ∧ mt_pos c'' = mt_pos c'
proof -
  let ?k = block_width (Γ_tm M)
  let ?R = R'
  have kpos: 0 < ?k using kge2 by simp
  show ?thesis
  proof (cases p = 0)
    case True
      have poskLE: posk tk = AR_AtLE using poskle True by simp
      obtain c'' where
        step: (c', c'') ∈ ?R
        and st: mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
        and tp: mt_tape c'' = mt_tape c'
        and ps: mt_pos c'' = mt_pos c'
        using leaf_le[OF vM stg qQ poskLE last vsrc pad0 src0] by blast
      show ?thesis
      proof (intro exI[where x = c''] conjI)
        show (c', c'') ∈ ?R ^^ (if p = 0 then 1 else 2 * ?k)
          using step True by simp
        show mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
          by (rule st)
        show mt_tape c'' = (if p = 0 then mt_tape c'
          else (mt_tape c')(tk := (λpos.
            if sim_pos ?k p ≤ pos ∧ pos < sim_pos ?k p + ?k
              then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - sim_pos ?k p)
              else mt_tape c' tk pos)))
          using tp True by simp
        show mt_pos c'' = mt_pos c' by (rule ps)
      qed
    next
      case False
        hence pge1: 1 ≤ p by simp
        have poskproper: posk tk ≠ AR_AtLE using poskle False by simp
        let ?base = sim_pos ?k p
        have pos_base: mt_pos c' tk = ?base + ?k using ppos False by simp
        have notLE: ∧m. m ≤ ?k ⇒ mt_tape c' tk (?base + m) ≠ LE4
        proof -
          fix m assume mle: m ≤ ?k
          show mt_tape c' tk (?base + m) ≠ LE4
          proof (cases m < ?k)
            case True
              have mt_tape c' tk (?base + m)

```

```

      = cell_repr (Γ_tm M) (bl_tm M) (tM p) ! m
    using tcorr pge1 True unfolding ar_tape_correspondence_def by blast
  thus ?thesis using cell_repr_nth_not_LE4[OF True] by simp
next
case False
hence meq: m = ?k using mle by simp
obtain pp where pp: p = Suc pp using pge1 by (cases p) auto
have psuc1: 1 ≤ Suc p by simp
have base_succ: ?base + ?k = sim_pos ?k (Suc p)
  using pp by (simp add: sim_pos_def)
have corr_succ: ∀ j < ?k. mt_tape c' tk (sim_pos ?k (Suc p) + j)
  = cell_repr (Γ_tm M) (bl_tm M) (tM (Suc p)) ! j
  using tcorr psuc1 unfolding ar_tape_correspondence_def by blast
have mt_tape c' tk (?base + m)
  = cell_repr (Γ_tm M) (bl_tm M) (tM (Suc p)) ! 0
  using meq base_succ corr_succ[rule_format, OF kpos] by simp
thus ?thesis using cell_repr_nth_not_LE4[OF kpos] by simp
qed
qed
obtain c'' where
  step: (c', c'') ∈ ?R ^^ (2 * ?k)
  and st: mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
  and tp: mt_tape c'' = (mt_tape c')(tk := (λpos.
    if ?base ≤ pos ∧ pos < ?base + ?k
    then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - ?base)
    else mt_tape c' tk pos))
  and ps: mt_pos c'' = (mt_pos c')(tk := ?base + ?k)
  using leaf_proper[OF vM qQ kge2 last poskproper stg buf_valid
    pos_base notLE pad0 src0]
  by blast
have ps': mt_pos c'' = mt_pos c'
proof -
  have (mt_pos c')(tk := ?base + ?k) = mt_pos c'
    using pos_base by (rule fun_upd_idem)
  thus ?thesis using ps by simp
qed
show ?thesis
proof (intro exI[where x = c''] conjI)
  show (c', c'') ∈ ?R ^^ (if p = 0 then 1 else 2 * ?k)
    using step False by simp
  show mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
    by (rule st)
  show mt_tape c'' = (if p = 0 then mt_tape c'
    else (mt_tape c')(tk := (λpos.
      if sim_pos ?k p ≤ pos ∧ pos < sim_pos ?k p + ?k
      then write_bit (Γ_tm M) (bl_tm M) (buf tk) (pos - sim_pos ?k p)
      else mt_tape c' tk pos)))
    using tp False by simp
  show mt_pos c'' = mt_pos c' by (rule ps')

```

qed
qed
qed

lemma *ar_write_tape_finish_step*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym}_4, 'q \times 'a \text{ar_stage}) \text{mt_config}$
and $tM :: \text{nat} \Rightarrow 'a$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $\text{last}: \text{is_last_k } M \ tk$
and $\text{stg}: \text{mt_state } c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)$
and $\text{tcrr}: \text{ar_tape_correspondence } (\Gamma_tm \ M) (\text{le_tm } M) (\text{bl_tm } M)$
 $tM \ (\text{mt_tape } c' \ tk)$
and $\text{ppos}: \text{mt_pos } c' \ tk = (\text{if } p = 0 \text{ then } \text{Suc } 0$
 $\text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p + \text{block_width } (\Gamma_tm$
 $M))$
and $\text{poskle}: \text{posk } tk = AR_AtLE \longleftrightarrow p = 0$
and $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm \ M \cup \{\text{bl_tm } M\}$
and $\text{vsrr}: \text{ar_valid_stage } (\Gamma_tm \ M) (\text{bl_tm } M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
and $\text{pad0}: \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = \text{BLANK}_4$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in (\text{mttm_step } (\text{alphabet_reduce_delta } M))$
 $\wedge \wedge (\text{if } p = 0 \text{ then } 1 \text{ else } 2 * \text{block_width } (\Gamma_tm \ M))$
 $\wedge \text{mt_state } c'' = (q, AR_SimAdvance, k_unidx \ 0, 0, buf, dvec, posk)$
 $\wedge \text{mt_tape } c'' = (\text{if } p = 0 \text{ then } \text{mt_tape } c'$
 $\text{else } (\text{mt_tape } c') (tk := (\lambda \text{pos.}$
 $\text{if } \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p \leq \text{pos}$
 $\wedge \text{pos} < \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p + \text{block_width}$
 $(\Gamma_tm \ M)$
 $\text{then } \text{write_bit } (\Gamma_tm \ M) (\text{bl_tm } M) (\text{buf } tk)$
 $(\text{pos} - \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p)$
 $\text{else } \text{mt_tape } c' \ tk \ \text{pos}))$
 $\wedge \text{mt_pos } c'' = \text{mt_pos } c'$
by (*rule ar_write_tape_finish_step_gen*
 $[OF \ \text{ar_write_le_finish_step } \text{ar_write_proper_finish_step}$
 $vM \ qQ \ kge2 \ \text{last} \ \text{stg} \ \text{tcrr} \ \text{ppos} \ \text{poskle} \ \text{buf_valid} \ \text{vsrr}$
 $\text{pad0 } \text{src0}]$)

The write-phase prefix walk: from the write boundary (current-tape field $k_unidx \ 0$, bit-counter 0 , the head positions the read phase left), iterate the unified non-last per-tape write *ar_write_tape_step* over the first j tapes ($j \leq k_tm \ M - 1$, all non-last), landing back at *AR_SimWrite* on tape $k_unidx \ j$. Dual to *ar_read_prefix*: the read kept the tape constant and varied *buf/posk*; the write keeps *buf/posk/positions* constant and varies the *tape*. The descriptor splits on $k_idx \ k < j$: visited tapes carry their over-

written block (*?out*, the *write_bit* image for proper tapes, or the unchanged tape for *LE* tapes); the rest are unchanged. Each per-tape step's *tcorr* / position entry facts hold because the current tape is not yet visited (*mt_tape c k_unidx j = mt_tape c0 (k_unidx j)*). Variable per-tape cost (*1* or *2b*), aggregate bounded by $j \cdot 2b$.

Relation-generic write prefix walk scaffold. The single per-tape helper (*ar_write_tape_step*) is the *leaf* hypothesis, quantified over the per-tape config, tape index, tape-content function, and position; union and sub versions are thin instantiations.

lemma *ar_write_prefix_gen*:

fixes $M :: ('q, 'a) \text{mttm}$
and $cM :: ('a, 'q) \text{mt_config}$
and $c0 :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config}$
and $R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config rel}$
assumes *leaf*:
 $\bigwedge cc \text{ tk}' tM' p'.$
 $\llbracket \text{valid_mttm } M; q \in Q_tm \ M; 2 \leq \text{block_width } (\Gamma_tm \ M);$
 $\neg \text{is_last_k } M \ \text{tk}';$
 $\text{mt_state } cc = (q, \text{AR_SimWrite}, \text{tk}', 0, \text{buf}, \text{dvec}, \text{posk});$
 $\text{ar_tape_correspondence } (\Gamma_tm \ M) (\text{le_tm } M) (\text{bl_tm } M)$
 $tM' (\text{mt_tape } cc \ \text{tk}');$
 $\text{mt_pos } cc \ \text{tk}' = (\text{if } p' = 0 \text{ then } \text{Suc } 0$
 $\text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p' + \text{block_width } (\Gamma_tm \ M));$
 $\text{posk } \text{tk}' = \text{AR_AtLE} \longleftrightarrow p' = 0;$
 $\forall k. \text{buf } k \in \Gamma_tm \ M \cup \{\text{bl_tm } M\};$
 $\text{ar_valid_stage } (\Gamma_tm \ M) (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, \text{tk}', 0, \text{buf}, \text{dvec}, \text{posk});$
 $\forall j \geq k_tm \ M. \text{mt_tape } cc \ j (\text{mt_pos } cc \ j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \ M)$
 $(\text{AR_SimWrite}, \text{tk}', 0, \text{buf}, \text{dvec}, \text{posk}) \rrbracket$
 $\implies \exists c''. (cc, c'') \in R' \wedge (\text{if } p' = 0 \text{ then } 1 \text{ else } 2 * \text{block_width } (\Gamma_tm$
 $M))$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimWrite}, k_succ \ \text{tk}', 0, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (\text{if } p' = 0 \text{ then } \text{mt_tape } cc$
 $\text{else } (\text{mt_tape } cc)(\text{tk}' := (\lambda pos.$
 $\text{if } \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p' \leq pos$
 $\wedge pos < \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p' + \text{block_width}$
 $(\Gamma_tm \ M)$
 $\text{then } \text{write_bit } (\Gamma_tm \ M) (\text{bl_tm } M) (\text{buf } \text{tk}')$
 $(pos - \text{sim_pos } (\text{block_width } (\Gamma_tm \ M)) \ p')$
 $\text{else } \text{mt_tape } cc \ \text{tk}' \ pos)))$
 $\wedge \text{mt_pos } c'' = \text{mt_pos } cc$
and $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $stg0: \text{mt_state } c0 = (q, \text{AR_SimWrite}, 0, 0, \text{buf}, \text{dvec}, \text{posk})$
and $tcorr: \forall k < k_tm \ M. \text{ar_tape_correspondence } (\Gamma_tm \ M) (\text{le_tm } M)$
 $(\text{bl_tm } M)$

```

      (mt_tape cM k) (mt_tape c0 k)
    and ppos:  $\forall k < k\_tm\ M. mt\_pos\ c0\ k = (if\ mt\_pos\ cM\ k = 0\ then\ Suc\ 0$ 
      else  $sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ (mt\_pos\ cM\ k)$ 
      +  $block\_width\ (\Gamma\_tm\ M))$ 
    and poskle:  $\forall k < k\_tm\ M. posk\ k = AR\_AtLE \longleftrightarrow mt\_pos\ cM\ k = 0$ 
    and buf_valid:  $\forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
    and pad0:  $\forall j \geq k\_tm\ M. mt\_tape\ c0\ j\ (mt\_pos\ c0\ j) = BLANK4$ 
    and src0:  $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
      ( $AR\_SimWrite, 0, 0, buf, dvec, posk$ )
  shows  $j \leq k\_tm\ M - 1 \implies$ 
    ( $\exists c\ m. (c0, c) \in R' \wedge m$ 
       $\wedge m \leq j * (2 * block\_width\ (\Gamma\_tm\ M))$ 
       $\wedge mt\_state\ c = (q, AR\_SimWrite, k\_unidx\ j, 0, buf, dvec, posk)$ 
       $\wedge mt\_tape\ c = (\lambda k. if\ k\_idx\ k < j$ 
        then  $(if\ mt\_pos\ cM\ k = 0\ then\ mt\_tape\ c0\ k$ 
        else  $(\lambda pos. if\ sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ (mt\_pos\ cM\ k)$ 
           $\leq pos$ 
             $\wedge pos < sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ (mt\_pos$ 
               $cM\ k)$ 
                +  $block\_width\ (\Gamma\_tm\ M)$ 
                then  $write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (buf\ k)$ 
                ( $pos - sim\_pos\ (block\_width\ (\Gamma\_tm\ M))\ (mt\_pos$ 
                   $cM\ k))$ 
                  else  $mt\_tape\ c0\ k\ pos))$ 
                else  $mt\_tape\ c0\ k)$ 
                 $\wedge mt\_pos\ c = mt\_pos\ c0)$ 
      proof (induction j)
      case 0
      let ?k =  $block\_width\ (\Gamma\_tm\ M)$ 
      have  $(c0, c0) \in R' \wedge 0$  by simp
      moreover have  $mt\_tape\ c0 = (\lambda k. if\ k\_idx\ k < 0$ 
        then  $(if\ mt\_pos\ cM\ k = 0\ then\ mt\_tape\ c0\ k$ 
        else  $(\lambda pos. if\ sim\_pos\ ?k\ (mt\_pos\ cM\ k) \leq pos$ 
           $\wedge pos < sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k$ 
          then  $write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (buf\ k)$ 
          ( $pos - sim\_pos\ ?k\ (mt\_pos\ cM\ k))$ 
          else  $mt\_tape\ c0\ k\ pos))$ 
        else  $mt\_tape\ c0\ k)$ 
        by simp
      ultimately show ?case
      using stg0 by (intro exI[where x = c0] exI[where x = 0]) (simp add:
        k_unidx_zero)
    next
    case (Suc j)
    have sucle:  $Suc\ j \leq k\_tm\ M - 1$  by (rule Suc.prem)
    have jcard:  $j < k\_tm\ M$  using sucle by simp
    have sjcard:  $Suc\ j < k\_tm\ M$  using sucle by simp
    have jle:  $j \leq k\_tm\ M - 1$  using sucle by simp
    let ?k =  $block\_width\ (\Gamma\_tm\ M)$ 

```

```

let ?tk = k_idx j
let ?p = mt_pos cM ?tk
let ?out =  $\lambda k.$  if mt_pos cM k = 0 then mt_tape c0 k
    else ( $\lambda pos.$  if sim_pos ?k (mt_pos cM k)  $\leq$  pos
         $\wedge$  pos < sim_pos ?k (mt_pos cM k) + ?k
        then write_bit ( $\Gamma_{tm}$  M) (bl_tm M) (buf k)
            (pos - sim_pos ?k (mt_pos cM k))
        else mt_tape c0 k pos)
let ?tape =  $\lambda i.$  ( $\lambda k.$  if k_idx k < i then ?out k else mt_tape c0 k)
have tkidx: k_idx ?tk = j by (rule k_idx_unidx[OF jcard])
have notlast:  $\neg$  is_last k M ?tk
    using is_last_k_unidx[OF jcard] sjcard by simp
have ksucc: k_succ ?tk = k_unidx (Suc j) by (rule k_succ_unidx[OF sjcard])
have inj: inj_on k_idx UNIV by (rule k_idx_inj)
have kne:  $\bigwedge k.$  k_idx k = j  $\implies$  k = ?tk
proof -
    fix k assume k_idx k = j
    hence k_idx k = k_idx ?tk using tkidx by simp
    thus k = ?tk using inj_on_eq_iff[OF inj UNIV_I UNIV_I] by simp
qed
obtain c m where
    cm_rel: (c0, c)  $\in R' \wedge m$ 
    and m_le: m  $\leq j * (2 * ?k)$ 
    and c_st: mt_state c = (q, AR_SimWrite, ?tk, 0, buf, dvec, posk)
    and c_tp: mt_tape c = ?tape j
    and c_ps: mt_pos c = mt_pos c0
    using Suc.IH[OF jle] by blast
— tape ?tk is not yet visited, so still matches c0
have tc_tk: mt_tape c ?tk = mt_tape c0 ?tk using c_tp tkidx by simp
have tk_active: ?tk < k_tm M using jcard by (simp add: k_unidx_def)
have tk_corr: ar_tape_correspondence ( $\Gamma_{tm}$  M) (le_tm M) (bl_tm M)
    (mt_tape cM ?tk) (mt_tape c ?tk)
    using tcorr[rule_format, OF tk_active] tc_tk by simp
have tk_ppos: mt_pos c ?tk
    = (if ?p = 0 then Suc 0 else sim_pos ?k ?p + ?k)
    using c_ps ppos[rule_format, OF tk_active] by simp
have tk_poskle: posk ?tk = AR_AtLE  $\longleftrightarrow$  ?p = 0
    by (rule poskle[rule_format, OF tk_active])
have tk_vsrc: ar_valid_stage ( $\Gamma_{tm}$  M) (bl_tm M)
    (AR_SimWrite, ?tk, 0, buf, dvec, posk)
    using kge2 buf_valid by (auto simp: ar_valid_stage_def)
have tail_idx:  $\bigwedge j'. k_{tm} M \leq j' \implies \neg k_{idx} j' < j$ 
    using jcard unfolding k_idx_def by linarith
have pad_c:  $\forall j' \geq k_{tm} M. mt\_tape\ c\ j' (mt\_pos\ c\ j') = BLANK4$ 
proof (intro allI impI)
    fix j' assume jge: k_tm M  $\leq j'$ 
    have mt_tape c j' (mt_pos c j') = mt_tape c0 j' (mt_pos c0 j')
    using c_tp c_ps tail_idx[OF jge] by simp
    thus mt_tape c j' (mt_pos c j') = BLANK4 using pad0 jge by simp

```



```

qed
have src_c: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, ?tk, 0, buf, dvec, posk)
  using src0 tk_active by (auto simp: ar_stage_bounded_def)
obtain c' where
  step: (c, c') ∈ R' ^ (if ?p = 0 then 1 else 2 * ?k)
  and c'_st: mt_state c' = (q, AR_SimWrite, k_succ ?tk, 0, buf, dvec, posk)
  and c'_tp: mt_tape c' = (if ?p = 0 then mt_tape c
    else (mt_tape c)(?tk := (λpos.
      if sim_pos ?k ?p ≤ pos ∧ pos < sim_pos ?k ?p + ?k
      then write_bit (Γ_tm M) (bl_tm M) (buf ?tk)
        (pos - sim_pos ?k ?p)
      else mt_tape c ?tk pos)))
  and c'_ps: mt_pos c' = mt_pos c
  using leaf[OF vM qQ kge2 notlast c_st tk_corr tk_ppos tk_poskle
    buf_valid tk_vsrc pad_c src_c]
  by blast
— the post-state tape descriptor coincides with the Suc j one
have tape_eq: mt_tape c' = ?tape (Suc j)
proof (rule ext)
  fix k
  show mt_tape c' k = ?tape (Suc j) k
  proof (cases k = ?tk)
    case True
    show ?thesis
    proof (cases ?p = 0)
      case True
      have mt_tape c' k = mt_tape c0 ?tk
        using c'_tp <k = ?tk> <?p = 0> tc_tk by simp
      moreover have ?tape (Suc j) k = mt_tape c0 ?tk
        using <k = ?tk> tkidx <?p = 0> by simp
      ultimately show ?thesis by simp
    next
    case False
    show ?thesis
    proof (rule ext)
      fix pos
      have lhs: mt_tape c' k pos
        = (if sim_pos ?k ?p ≤ pos ∧ pos < sim_pos ?k ?p + ?k
          then write_bit (Γ_tm M) (bl_tm M) (buf ?tk) (pos - sim_pos
?k ?p)
          else mt_tape c ?tk pos)
        using c'_tp <k = ?tk> <?p ≠ 0> by simp
      have rhs: ?tape (Suc j) k pos
        = (if sim_pos ?k ?p ≤ pos ∧ pos < sim_pos ?k ?p + ?k
          then write_bit (Γ_tm M) (bl_tm M) (buf ?tk) (pos - sim_pos
?k ?p)
          else mt_tape c0 ?tk pos)
        using <k = ?tk> tkidx <?p ≠ 0> by simp
    qed
  qed

```

```

    show mt_tape c' k pos = ?tape (Suc j) k pos
    using lhs rhs tc_tk by simp
  qed
qed
next
case False
hence kidx_ne: k_idx k ≠ j using kne by blast
have mt_tape c' k = mt_tape c k
  using c'_tp False by (cases ?p = 0) simp_all
also have ... = ?tape j k using c_tp by simp
also have ... = ?tape (Suc j) k using kidx_ne by (simp add: less_Suc_eq)
finally show ?thesis .
qed
qed
have chain: (c0, c') ∈ R' ^^ (m + (if ?p = 0 then 1 else 2 * ?k))
proof -
  have (c0, c') ∈ R' ^^ m
    O R' ^^ (if ?p = 0 then 1 else 2 * ?k)
  using cm_rel step by (rule relcompI)
  thus ?thesis by (simp add: relpow_add)
qed
have bound: m + (if ?p = 0 then 1 else 2 * ?k) ≤ Suc j * (2 * ?k)
  using m_le kge2 by (cases ?p = 0) auto
show ?case
proof (intro exI[where x = c'])
  exI[where x = m + (if ?p = 0 then 1 else 2 * ?k)] conjI
  show (c0, c') ∈ R' ^^ (m + (if ?p = 0 then 1 else 2 * ?k)) by (rule chain)
  show m + (if ?p = 0 then 1 else 2 * ?k) ≤ Suc j * (2 * ?k)
    by (rule bound)
  show mt_state c' = (q, AR_SimWrite, k_unidx (Suc j), 0, buf, dvec, posk)
    using c'_st ksucc by simp
  show mt_tape c' = (λk. if k_idx k < Suc j
    then (if mt_pos cM k = 0 then mt_tape c0 k
      else (λpos. if sim_pos ?k (mt_pos cM k) ≤ pos
        ∧ pos < sim_pos ?k (mt_pos cM k) + ?k
        then write_bit (Γ_tm M) (bl_tm M) (buf k)
          (pos - sim_pos ?k (mt_pos cM k))
        else mt_tape c0 k pos))
    else mt_tape c0 k)
    using tape_eq by simp
  show mt_pos c' = mt_pos c0 using c'_ps c_ps by simp
qed
qed

lemma ar_write_prefix:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M

```

and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $stg0: mt_state\ c0 = (q, AR_SimWrite, 0, 0, buf, dvec, posk)$
and $tcrr: \forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)$
 $(bl_tm\ M)$
 $(mt_tape\ cM\ k)\ (mt_tape\ c0\ k)$
and $ppos: \forall k < k_tm\ M. mt_pos\ c0\ k = (if\ mt_pos\ cM\ k = 0\ then\ Suc\ 0$
 $else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $+ block_width\ (\Gamma_tm\ M))$
and $poskle: \forall k < k_tm\ M. posk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0$
and $buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, 0, 0, buf, dvec, posk)$
shows $j \leq k_tm\ M - 1 \implies$
 $(\exists c\ m. (c0, c) \in (mttm_step\ (alphabet_reduce_delta\ M)) \wedge m$
 $\wedge m \leq j * (2 * block_width\ (\Gamma_tm\ M))$
 $\wedge mt_state\ c = (q, AR_SimWrite, k_unidx\ j, 0, buf, dvec, posk)$
 $\wedge mt_tape\ c = (\lambda k. if\ k_idx\ k < j$
 $then\ (if\ mt_pos\ cM\ k = 0\ then\ mt_tape\ c0\ k$
 $else\ (\lambda pos. if\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $\leq pos$
 $cM\ k)$
 $\wedge pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $+ block_width\ (\Gamma_tm\ M)$
 $then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ k)$
 $(pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k))$
 $else\ mt_tape\ c0\ k\ pos))$
 $else\ mt_tape\ c0\ k)$
 $\wedge mt_pos\ c = mt_pos\ c0)$
by $(rule\ ar_write_prefix_gen$
 $[OF\ ar_write_tape_step\ vM\ qQ\ kge2\ stg0\ tcrr\ ppos\ poskle\ buf_valid$
 $pad0\ src0])$

The full write phase: the prefix walk over the first $k_tm\ M - 1$ tapes followed by the unified last-tape write $ar_write_tape_finish_step$, landing at $AR_SimAdvance$ (current-tape field $k_unidx\ 0$) with every proper tape's b -cell block overwritten by $write_bit\ (buf\ k)$ (the encoded M -write symbol) and every LE tape and head position unchanged. Aggregate cost $\leq k_tm\ M \cdot 2b$. The split tape descriptor collapses to the full per-tape overwrite on the last tape (every other tape has index $< k_tm\ M - 1$).

Relation-generic full write phase scaffold. The two helpers (prefix walk, last-tape finish) are the $leaf_prefix$ and $leaf_finish$ hypotheses; union and sub versions are thin instantiations.

lemma $ar_write_phase_gen:$
fixes $M :: ('q, 'a)\ mttm$
and $cM :: ('a, 'q)\ mt_config$

and $c0 :: (sym4, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
and $R' :: (sym4, 'q \times 'a \text{ ar_stage}) \text{ mt_config rel}$
assumes leaf_prefix :
 $\bigwedge jj. \llbracket \text{valid_mttm } M; q \in Q_tm \text{ } M; 2 \leq \text{block_width } (\Gamma_tm \text{ } M);$
 $\text{mt_state } c0 = (q, \text{AR_SimWrite}, 0, 0, \text{buf}, \text{dvec}, \text{posk});$
 $\forall k < k_tm \text{ } M. \text{ar_tape_correspondence } (\Gamma_tm \text{ } M) (\text{le_tm } M)$
 $(\text{bl_tm } M)$
 $(\text{mt_tape } cM \text{ } k) (\text{mt_tape } c0 \text{ } k);$
 $\forall k < k_tm \text{ } M. \text{mt_pos } c0 \text{ } k = (\text{if } \text{mt_pos } cM \text{ } k = 0 \text{ then } \text{Suc } 0$
 $\text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{mt_pos } cM \text{ } k)$
 $+ \text{block_width } (\Gamma_tm \text{ } M));$
 $\forall k < k_tm \text{ } M. \text{posk } k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM \text{ } k = 0;$
 $\forall k. \text{buf } k \in \Gamma_tm \text{ } M \cup \{\text{bl_tm } M\};$
 $\forall j \geq k_tm \text{ } M. \text{mt_tape } c0 \text{ } j (\text{mt_pos } c0 \text{ } j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \text{ } M)$
 $(\text{AR_SimWrite}, 0, 0, \text{buf}, \text{dvec}, \text{posk});$
 $jj \leq k_tm \text{ } M - 1 \rrbracket$
 $\implies (\exists c \text{ } m. (c0, c) \in R' \wedge m$
 $\wedge m \leq jj * (2 * \text{block_width } (\Gamma_tm \text{ } M))$
 $\wedge \text{mt_state } c = (q, \text{AR_SimWrite}, k_unidx \text{ } jj, 0, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c = (\lambda k. \text{if } k_idx \text{ } k < jj$
 $\text{then } (\text{if } \text{mt_pos } cM \text{ } k = 0 \text{ then } \text{mt_tape } c0 \text{ } k$
 $\text{else } (\lambda \text{pos. if } \text{sim_pos } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{mt_pos}$
 $cM \text{ } k) \leq \text{pos}$
 $\wedge \text{pos} < \text{sim_pos } (\text{block_width } (\Gamma_tm \text{ } M))$
 $(\text{mt_pos } cM \text{ } k)$
 $+ \text{block_width } (\Gamma_tm \text{ } M)$
 $\text{then } \text{write_bit } (\Gamma_tm \text{ } M) (\text{bl_tm } M) (\text{buf } k)$
 $(\text{pos} - \text{sim_pos } (\text{block_width } (\Gamma_tm \text{ } M))$
 $(\text{mt_pos } cM \text{ } k))$
 $\text{else } \text{mt_tape } c0 \text{ } k \text{ pos}))$
 $\text{else } \text{mt_tape } c0 \text{ } k)$
 $\wedge \text{mt_pos } c = \text{mt_pos } c0)$
and leaf_finish :
 $\bigwedge cc \text{ } tk' \text{ } tM' \text{ } p'.$
 $\llbracket \text{valid_mttm } M; q \in Q_tm \text{ } M; 2 \leq \text{block_width } (\Gamma_tm \text{ } M);$
 $\text{is_last_k } M \text{ } tk';$
 $\text{mt_state } cc = (q, \text{AR_SimWrite}, tk', 0, \text{buf}, \text{dvec}, \text{posk});$
 $\text{ar_tape_correspondence } (\Gamma_tm \text{ } M) (\text{le_tm } M) (\text{bl_tm } M)$
 $tM' (\text{mt_tape } cc \text{ } tk');$
 $\text{mt_pos } cc \text{ } tk' = (\text{if } p' = 0 \text{ then } \text{Suc } 0$
 $\text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm \text{ } M)) p' + \text{block_width } (\Gamma_tm \text{ } M));$
 $\text{posk } tk' = \text{AR_AtLE} \longleftrightarrow p' = 0;$
 $\forall k. \text{buf } k \in \Gamma_tm \text{ } M \cup \{\text{bl_tm } M\};$
 $\text{ar_valid_stage } (\Gamma_tm \text{ } M) (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, tk', 0, \text{buf}, \text{dvec}, \text{posk});$
 $\forall j \geq k_tm \text{ } M. \text{mt_tape } cc \text{ } j (\text{mt_pos } cc \text{ } j) = \text{BLANK4};$
 $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \text{ } M)$
 $(\text{AR_SimWrite}, tk', 0, \text{buf}, \text{dvec}, \text{posk}) \rrbracket$

$\implies \exists c''. (cc, c'') \in R' \wedge (if\ p' = 0\ then\ 1\ else\ 2 * block_width\ (\Gamma_tm\ M))$
 $\wedge\ mt_state\ c'' = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec,$
 $posk)$
 $\wedge\ mt_tape\ c'' = (if\ p' = 0\ then\ mt_tape\ cc$
 $\quad else\ (mt_tape\ cc)(tk' := (\lambda pos.$
 $\quad\quad if\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ p' \leq pos$
 $\quad\quad \wedge\ pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ p' + block_width$
 $(\Gamma_tm\ M)$
 $\quad\quad then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ tk')$
 $\quad\quad\quad (pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ p')$
 $\quad\quad\quad else\ mt_tape\ cc\ tk'\ pos)))$
 $\wedge\ mt_pos\ c'' = mt_pos\ cc$
and vM : $valid_mttm\ M$
and qQ : $q \in \bar{Q}_tm\ M$
and $kge2$: $2 \leq block_width\ (\Gamma_tm\ M)$
and $stg0$: $mt_state\ c0 = (q, AR_SimWrite, 0, 0, buf, dvec, posk)$
and $tcrr$: $\forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)$
 $(bl_tm\ M)$
 $(mt_tape\ cM\ k)\ (mt_tape\ c0\ k)$
and $ppos$: $\forall k < k_tm\ M. mt_pos\ c0\ k = (if\ mt_pos\ cM\ k = 0\ then\ Suc\ 0$
 $\quad else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $\quad +\ block_width\ (\Gamma_tm\ M))$
and $poskle$: $\forall k < k_tm\ M. posk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0$
and buf_valid : $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $pad0$: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0$: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, 0, 0, buf, dvec, posk)$
shows $\exists c\ m. (c0, c) \in R' \wedge m$
 $\wedge\ m \leq k_tm\ M * (2 * block_width\ (\Gamma_tm\ M))$
 $\wedge\ mt_state\ c = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk)$
 $\wedge\ mt_tape\ c = (\lambda k. if\ k < k_tm\ M$
 $\quad then\ (if\ mt_pos\ cM\ k = 0\ then\ mt_tape\ c0\ k$
 $\quad else\ (\lambda pos. if\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $\leq pos$
 $\quad \wedge\ pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k)$
 $\quad +\ block_width\ (\Gamma_tm\ M)$
 $\quad then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ k)$
 $\quad\quad (pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k))$
 $\quad else\ mt_tape\ c0\ k\ pos))$
 $\quad else\ mt_tape\ c0\ k)$
 $\wedge\ mt_pos\ c = mt_pos\ c0$
proof –
let $?k = block_width\ (\Gamma_tm\ M)$
let $?N = k_tm\ M$
let $?m1 = ?N - 1$
let $?tl = k_unidx\ ?m1$

```

let ?p = mt_pos cM ?tl
let ?out =  $\lambda k.$  if mt_pos cM k = 0 then mt_tape c0 k
    else ( $\lambda pos.$  if sim_pos ?k (mt_pos cM k)  $\leq$  pos
         $\wedge$  pos < sim_pos ?k (mt_pos cM k) + ?k
        then write_bit ( $\Gamma_{tm}$  M) (bl_tm M) (buf k)
            (pos - sim_pos ?k (mt_pos cM k))
        else mt_tape c0 k pos)
let ?tape =  $\lambda i.$  ( $\lambda k.$  if k_idx k < i then ?out k else mt_tape c0 k)
have card1: 0 < ?N using vM by (cases M) auto
have m1suc: ?N = Suc ?m1 using card1 by simp
have m1card: ?m1 < ?N using card1 by simp
have m1lt: ?m1 < k_tm M using card1 by simp
have tl_eq: ?tl = ?m1 by (simp add: k_unidx_def)
have tlidx: k_idx ?tl = ?m1 by (rule k_idx_unidx[OF m1card])
have tl_active: ?tl < k_tm M using m1lt tl_eq by simp
have islast: is_last_k M ?tl using is_last_k_unidx[OF m1card] m1suc by simp
have cond: (k < ?m1) = (k < k_tm M) if k  $\neq$  ?m1 for k
    using that m1suc by (auto simp: less_Suc_eq)
obtain c1 m1 where
    c1_rel: (c0, c1)  $\in$  R'  $\wedge$  m1
    and m1_le: m1  $\leq$  ?m1 * (2 * ?k)
    and c1_st: mt_state c1 = (q, AR_SimWrite, ?tl, 0, buf, dvec, posk)
    and c1_tp: mt_tape c1 = ?tape ?m1
    and c1_ps: mt_pos c1 = mt_pos c0
    using leaf_prefix[OF vM qQ kge2 stg0 tcorr ppos poskle buf_valid pad0 src0,
        of ?m1]
    by auto
have tc_tl: mt_tape c1 ?tl = mt_tape c0 ?tl using c1_tp tlidx by simp
have tl_corr: ar_tape_correspondence ( $\Gamma_{tm}$  M) (le_tm M) (bl_tm M)
    (mt_tape cM ?tl) (mt_tape c1 ?tl)
    using tcorr[rule_format, OF tl_active] tc_tl by simp
have tl_ppos: mt_pos c1 ?tl
    = (if ?p = 0 then Suc 0 else sim_pos ?k ?p + ?k)
    using c1_ps ppos[rule_format, OF tl_active] by simp
have tl_poskle: posk ?tl = AR_AtLE  $\longleftrightarrow$  ?p = 0
    by (rule poskle[rule_format, OF tl_active])
have tl_vsrc: ar_valid_stage ( $\Gamma_{tm}$  M) (bl_tm M)
    (AR_SimWrite, ?tl, 0, buf, dvec, posk)
    using kge2 buf_valid by (auto simp: ar_valid_stage_def)
have tail_idx:  $\bigwedge j.$  k_tm M  $\leq$  j  $\implies$   $\neg$  k_idx j < ?m1
    using m1suc unfolding k_idx_def by linarith
have pad_c1:  $\forall j \geq k_{tm} M.$  mt_tape c1 j (mt_pos c1 j) = BLANK4
proof (intro allI impI)
    fix j assume jge: k_tm M  $\leq$  j
    have mt_tape c1 j (mt_pos c1 j) = mt_tape c0 j (mt_pos c0 j)
    using c1_tp c1_ps tail_idx[OF jge] by simp
    thus mt_tape c1 j (mt_pos c1 j) = BLANK4 using pad0 jge by simp
qed
have src_c1: ar_stage_bounded (bl_tm M) (k_tm M)

```

```

      (AR_SimWrite, ?tl, 0, buf, dvec, posk)
    using src0 tl_active by (auto simp: ar_stage_bounded_def)
  obtain c2 where
    step: (c1, c2) ∈ R' ^^ (if ?p = 0 then 1 else 2 * ?k)
  and c2_st: mt_state c2 = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
  and c2_tp: mt_tape c2 = (if ?p = 0 then mt_tape c1
    else (mt_tape c1)(?tl := (λpos.
      if sim_pos ?k ?p ≤ pos ∧ pos < sim_pos ?k ?p + ?k
      then write_bit (Γ_tm M) (bl_tm M) (buf ?tl)
        (pos - sim_pos ?k ?p)
      else mt_tape c1 ?tl pos)))
  and c2_ps: mt_pos c2 = mt_pos c1
  using leaf_finish[OF vM qQ kge2 islast c1_st tl_corr tl_ppos
    tl_poskle buf_valid tl_vsrc pad_c1 src_c1]
  by blast
  — The last-tape fun_upd collapses to the active-guarded per-tape overwrite:
  active tapes get ?out, padding tapes stay at c0's tape (the walk never visits them).
  have tape_full: mt_tape c2 = (λk. if k < k_tm M then ?out k else mt_tape c0
    k)
  proof (rule ext)
    fix k
    show mt_tape c2 k = (if k < k_tm M then ?out k else mt_tape c0 k)
  proof (cases k = ?tl)
    case True
    have out_tl: mt_tape c2 k = ?out k
    proof (cases ?p = 0)
      case True
      have mt_tape c2 k = mt_tape c0 ?tl
      using c2_tp ⟨k = ?tl⟩ ⟨?p = 0⟩ tc_tl by simp
      moreover have ?out k = mt_tape c0 ?tl
      using ⟨k = ?tl⟩ ⟨?p = 0⟩ by simp
      ultimately show ?thesis by simp
    next
    case False
    show ?thesis
    proof (rule ext)
      fix pos
      have lhs: mt_tape c2 k pos
      = (if sim_pos ?k ?p ≤ pos ∧ pos < sim_pos ?k ?p + ?k
        then write_bit (Γ_tm M) (bl_tm M) (buf ?tl) (pos - sim_pos
    ?k ?p)
        else mt_tape c1 ?tl pos)
      using c2_tp ⟨k = ?tl⟩ ⟨?p ≠ 0⟩ by simp
      have rhs: ?out k pos
      = (if sim_pos ?k ?p ≤ pos ∧ pos < sim_pos ?k ?p + ?k
        then write_bit (Γ_tm M) (bl_tm M) (buf ?tl) (pos - sim_pos
    ?k ?p)
        else mt_tape c0 ?tl pos)
      using ⟨k = ?tl⟩ ⟨?p ≠ 0⟩ by simp
    end
  end
end

```

```

    show  $mt\_tape\ c2\ k\ pos = ?out\ k\ pos$ 
      using  $lhs\ rhs\ tc\_tl$  by simp
  qed
qed
show  $?thesis$  using  $out\_tl\ \langle k = ?tl \rangle\ tl\_eq\ m1lt$  by simp
next
case False
hence  $kne: k \neq ?m1$  using  $tl\_eq$  by simp
show  $?thesis$ 
proof (cases  $k < ?m1$ )
  case True
  have  $klt: k < k\_tm\ M$  using True  $m1lt$  by simp
  have  $mt\_tape\ c2\ k = mt\_tape\ c1\ k$ 
    using  $c2\_tp\ \langle k \neq ?tl \rangle$  by (cases  $?p = 0$ ) simp_all
  also have  $\dots = ?out\ k$  using  $c1\_tp\ True$  by (simp add:  $k\_idx\_def$ )
  finally show  $?thesis$  using  $klt$  by simp
next
case False
have  $kge: \neg k < k\_tm\ M$  using False  $kne\ cond[OF\ kne]$  by simp
have  $mt\_tape\ c2\ k = mt\_tape\ c1\ k$ 
  using  $c2\_tp\ \langle k \neq ?tl \rangle$  by (cases  $?p = 0$ ) simp_all
also have  $\dots = mt\_tape\ c0\ k$  using  $c1\_tp\ False$  by (simp add:  $k\_idx\_def$ )
  finally show  $?thesis$  using  $kge$  by simp
qed
qed
qed
have  $chain: (c0, c2) \in R' \wedge (m1 + (if\ ?p = 0\ then\ 1\ else\ 2 * ?k))$ 
proof -
  have  $(c0, c2) \in R' \wedge m1$ 
     $O\ R' \wedge (if\ ?p = 0\ then\ 1\ else\ 2 * ?k)$ 
  using  $c1\_rel\ step$  by (rule  $relcompI$ )
  thus  $?thesis$  by (simp add:  $relpow\_add$ )
qed
have  $bound: m1 + (if\ ?p = 0\ then\ 1\ else\ 2 * ?k) \leq ?N * (2 * ?k)$ 
proof -
  obtain  $Nm$  where  $Nm: ?N = Suc\ Nm$  using  $card1$  by (cases  $?N$ ) auto
  have  $e1: ?N * (2 * ?k) = (2 * ?k) + Nm * (2 * ?k)$  by (simp add:  $Nm$ )
  have  $e2: m1 \leq Nm * (2 * ?k)$  using  $m1\_le\ Nm$  by simp
  have  $e3: (if\ ?p = 0\ then\ 1\ else\ 2 * ?k) \leq 2 * ?k$  using  $kge2$  by simp
  show  $?thesis$  using  $e1\ e2\ e3$  by linarith
qed
show  $?thesis$ 
proof (intro  $exI$  [where  $x = c2$ ])
   $exI$  [where  $x = m1 + (if\ ?p = 0\ then\ 1\ else\ 2 * ?k)$ ] conjI
  show  $(c0, c2) \in R' \wedge (m1 + (if\ ?p = 0\ then\ 1\ else\ 2 * ?k))$  by (rule  $chain$ )
  show  $m1 + (if\ ?p = 0\ then\ 1\ else\ 2 * ?k) \leq ?N * (2 * ?k)$  by (rule  $bound$ )
  show  $mt\_state\ c2 = (q, AR\_SimAdvance, k\_unidx\ 0, 0, buf, dvec, posk)$ 
    by (rule  $c2\_st$ )
  show  $mt\_tape\ c2 = (\lambda k. if\ k < k\_tm\ M$ 

```



```

      then (if mt_pos cM k = 0 then mt_tape c0 k
            else (λpos. if sim_pos ?k (mt_pos cM k) ≤ pos
                        ∧ pos < sim_pos ?k (mt_pos cM k) + ?k
                        then write_bit (Γ_tm M) (bl_tm M) (buf k)
                        (pos - sim_pos ?k (mt_pos cM k))
                        else mt_tape c0 k pos))
            else mt_tape c0 k)
    using tape_full by simp
  show mt_pos c2 = mt_pos c0 using c2_ps c1_ps by simp
qed
qed

lemma ar_write_phase:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and stg0: mt_state c0 = (q, AR_SimWrite, 0, 0, buf, dvec, posk)
  and tcorr: ∀ k < k_tm M. ar_tape_correspondence (Γ_tm M) (le_tm M)
    (bl_tm M)
    (mt_tape cM k) (mt_tape c0 k)
  and ppos: ∀ k < k_tm M. mt_pos c0 k = (if mt_pos cM k = 0 then Suc 0
    else sim_pos (block_width (Γ_tm M)) (mt_pos cM k)
    + block_width (Γ_tm M))
  and poskle: ∀ k < k_tm M. posk k = AR_AtLE ⟷ mt_pos cM k = 0
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and pad0: ∀ j ≥ k_tm M. mt_tape c0 j (mt_pos c0 j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, 0, 0, buf, dvec, posk)
  shows ∃ c m. (c0, c) ∈ (mttm_step (alphabet_reduce_delta M)) ^~ m
    ∧ m ≤ k_tm M * (2 * block_width (Γ_tm M))
    ∧ mt_state c = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
    ∧ mt_tape c = (λk. if k < k_tm M
      then (if mt_pos cM k = 0 then mt_tape c0 k
            else (λpos. if sim_pos (block_width (Γ_tm M)) (mt_pos
      cM k)
      ≤ pos
      ∧ pos < sim_pos (block_width (Γ_tm M)) (mt_pos
      cM k)
      + block_width (Γ_tm M)
      then write_bit (Γ_tm M) (bl_tm M) (buf k)
      (pos - sim_pos (block_width (Γ_tm M)) (mt_pos
      cM k))
      else mt_tape c0 k pos))
      else mt_tape c0 k)
    ∧ mt_pos c = mt_pos c0
  by (rule ar_write_phase_gen
    [OF ar_write_prefix ar_write_tape_finish_step

```

```

vM qQ kge2 stg0 tcorr ppos poskle buf_valid pad0 src0])

end
theory AlphabetReduction_ForwardAdvance
  imports AlphabetReduction_ForwardWrite
begin

```

2.15 Advance phase

Relation-generic advance left-walk loop scaffold. The chain over R' is built natively by *relpow_invariant_chain* from the per-step leaf contract supplied as the *leaf* hypothesis; the union-relation *ar_advance_walk_loop* and the sub-relation *ar_advance_walk_loop_in_sub* are both thin instantiations, discharging *leaf* by *ar_advance_walk_step* and *ar_advance_walk_step_in_sub* respectively. No *stays* side-condition arises: the chain is native to R' rather than lifted from the union relation, so the leaf-parametric scaffold applies where a union-to-sub chain lift would not. Throughout, b abbreviates *block_width* Γ (the per-symbol cell width).

lemma *ar_advance_walk_loop_gen*:

```

fixes M :: ('q, 'a) mttm
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  and R' :: (sym4, 'q × 'a ar_stage) mt_config rel
assumes leaf:
  ∧cc i. [ valid_mttm M;
    mt_state cc = (q, AR_SimAdvance, tk, i, buf, dvec, posk);
    q ∈ Q_tm M;
    mt_tape cc tk (mt_pos cc tk) ≠ LE4;
    Suc i < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk);
    ar_valid_stage (Γ_tm M) (bl_tm M)
      (AR_SimAdvance, tk, i, buf, dvec, posk);
    ∀ j ≥ k_tm M. mt_tape cc j (mt_pos cc j) = BLANK4;
    ar_stage_bounded (bl_tm M) (k_tm M)
      (AR_SimAdvance, tk, i, buf, dvec, posk) ]
  ⇒ ∃ c''. (cc, c'') ∈ R'
    ∧ mt_state c'' = (q, AR_SimAdvance, tk, Suc i, buf, dvec,
posk)
    ∧ mt_tape c'' = mt_tape cc
    ∧ mt_pos c'' = (mt_pos cc)(tk := mt_pos cc tk - 1)
  and vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and stg: mt_state c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and ndisp: n < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
  and notLE: ∧j. j < n ⇒ mt_tape c0 tk (mt_pos c0 tk - j) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c0 j (mt_pos c0 j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimAdvance, tk, 0, buf, dvec, posk)
shows ∃ c'. (c0, c') ∈ R' ^ n

```

$$\begin{aligned}
& \wedge \text{mt_state } c' = (q, \text{AR_SimAdvance}, tk, n, \text{buf}, \text{dvec}, \text{posk}) \\
& \wedge \text{mt_pos } c' \text{ tk} = \text{mt_pos } c0 \text{ tk} - n \\
& \wedge \text{mt_tape } c' = \text{mt_tape } c0 \\
& \wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c' k' = \text{mt_pos } c0 k')
\end{aligned}$$

proof –

let $?k = \text{block_width } (\Gamma_tm \ M)$
let $?D = \text{ar_disp } ?k \ (\text{dvec } tk) \ (\text{posk } tk)$
let $?start = \text{mt_pos } c0 \ tk$
have $kpos: 0 < ?k$ **using** $\text{block_width_pos}[of \ \Gamma_tm \ M]$ **by** simp
let $?P = \lambda j \ c.$
 $\text{mt_state } c = (q, \text{AR_SimAdvance}, tk, j, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_pos } c \ tk = ?start - j$
 $\wedge \text{mt_tape } c = \text{mt_tape } c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c \ k' = \text{mt_pos } c0 \ k')$
have $\text{mybase}: ?P \ 0 \ c0$ **using** stg **by** simp
have $\text{mystep}:$
 $\wedge j \ cc. \llbracket j < n; ?P \ j \ cc \rrbracket$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge ?P \ (\text{Suc } j) \ c''$

proof –

fix $j \ cc$
assume $jlt: j < n$ **and** $Pj: ?P \ j \ cc$
from Pj **have** $st_cc:$
 $\text{mt_state } cc = (q, \text{AR_SimAdvance}, tk, j, \text{buf}, \text{dvec}, \text{posk})$
and $\text{pos_cc}: \text{mt_pos } cc \ tk = ?start - j$
and $\text{tape_cc}: \text{mt_tape } cc = \text{mt_tape } c0$
and $\text{other_cc}: \forall k'. k' \neq tk \longrightarrow \text{mt_pos } cc \ k' = \text{mt_pos } c0 \ k'$
by simp_all
have $tk_lt: tk < k_tm \ M$ **using** src0 **by** $(\text{simp add: ar_stage_bounded_def})$
have $\text{pad_cc}: \forall j' \geq k_tm \ M. \text{mt_tape } cc \ j' (\text{mt_pos } cc \ j') = \text{BLANK4}$
proof (intro allI impI)
fix j' **assume** $jge: k_tm \ M \leq j'$
have $jne: j' \neq tk$ **using** $jge \ tk_lt$ **by** simp
have $\text{mt_tape } cc \ j' (\text{mt_pos } cc \ j') = \text{mt_tape } c0 \ j' (\text{mt_pos } c0 \ j')$
using $\text{tape_cc other_cc jne}$ **by** simp
thus $\text{mt_tape } cc \ j' (\text{mt_pos } cc \ j') = \text{BLANK4}$ **using** pad0 jge **by** simp
qed
have $\text{src_cc}: \text{ar_stage_bounded } (bl_tm \ M) \ (k_tm \ M)$
 $(\text{AR_SimAdvance}, tk, j, \text{buf}, \text{dvec}, \text{posk})$
using src0 **by** $(\text{simp add: ar_stage_bounded_def})$
have $\text{sucj_lt}: \text{Suc } j < ?D$ **using** $jlt \ \text{ndisp}$ **by** linarith
have $j_lt2k: j < 2 * ?k$
using $\text{sucj_lt ar_disp_le_2k}[OF \ kpos, of \ \text{dvec } tk \ \text{posk } tk]$ **by** linarith
have $\text{vsrc_cc}: \text{ar_valid_stage } (\Gamma_tm \ M) \ (bl_tm \ M)$
 $(\text{AR_SimAdvance}, tk, j, \text{buf}, \text{dvec}, \text{posk})$
using $j_lt2k \ \text{buf_valid}$ **by** $(\text{simp add: ar_valid_stage_def})$
have $\text{notLE_cc}: \text{mt_tape } cc \ tk (\text{mt_pos } cc \ tk) \neq \text{LE4}$
using $\text{tape_cc pos_cc notLE}[OF \ jlt]$ **by** simp
obtain c'' **where**

```

      bstep: (cc, c'') ∈ R'
    and bst_state: mt_state c'' = (q, AR_SimAdvance, tk, Suc j, buf, dvec,
posk)
    and bst_tape: mt_tape c'' = mt_tape cc
    and bst_pos: mt_pos c'' = (mt_pos cc)(tk := mt_pos cc tk - 1)
    using leaf[OF vM st_cc qQ notLE_cc sucj_lt vsrc_cc pad_cc src_cc] by
blast
    have pos_eq: mt_pos c'' tk = ?start - Suc j
    using bst_pos pos_cc by simp
    have tape_eq: mt_tape c'' = mt_tape c0 using bst_tape tape_cc by simp
    have other_eq: ∀ k'. k' ≠ tk → mt_pos c'' k' = mt_pos c0 k'
    using bst_pos other_cc by simp
    show ∃ c''. (cc, c'') ∈ R'
      ∧ ?P (Suc j) c''
    using bstep bst_state pos_eq tape_eq other_eq by blast
  qed
  have loop:
    ∃ c'. (c0, c') ∈ R' ^^ n
      ∧ ?P n c'
  by (rule relpow_invariant_chain
    [where P = ?P and n = n, OF mystep mybase])
  obtain c' where
    chain: (c0, c') ∈ R' ^^ n
    and Pfn: ?P n c'
    using loop by blast
  show ?thesis
  proof (intro exI[where x = c'] conjI)
    show (c0, c') ∈ R' ^^ n by (rule chain)
    show mt_state c' = (q, AR_SimAdvance, tk, n, buf, dvec, posk)
      using Pfn by simp
    show mt_pos c' tk = ?start - n using Pfn by simp
    show mt_tape c' = mt_tape c0 using Pfn by simp
    show ∀ k'. k' ≠ tk → mt_pos c' k' = mt_pos c0 k' using Pfn by simp
  qed
qed

```

The advance left-walk loop, proper tape. From an *AR_SimAdvance* stage at bit-counter 0 with the head at *start*, *n* *M*-steps walk the head *n* cells *L* (counter 0 → *n*), the tape and all other heads unchanged. Each *ar_advance_walk_step*'s $\neq LE4$ guard reads the current cell *start* - *j*, supplied $\neq LE4$ by *notLE*; the per-step displacement bound *Suc j* < *D* follows from *n* < *D* by *linarith*. Unlike the write loops this loop does not mutate the tape (advance only moves heads), so the invariant keeps *mt_tape c* = *mt_tape c0* constant. Chain length *n*.

```

lemma ar_advance_walk_loop:
  fixes M :: ('q, 'a) mttm
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M

```

and *stg*: $mt_state\ c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)$
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *ndisp*: $n < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
and *notLE*: $\bigwedge j. j < n \implies mt_tape\ c0\ tk\ (mt_pos\ c0\ tk - j) \neq LE4$
and *pad0*: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and *src0*: $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in (mttm_step\ (alphabet_reduce_delta\ M)) \wedge^n$
 $\wedge mt_state\ c' = (q, AR_SimAdvance, tk, n, buf, dvec, posk)$
 $\wedge mt_pos\ c'\ tk = mt_pos\ c0\ tk - n$
 $\wedge mt_tape\ c' = mt_tape\ c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c'\ k' = mt_pos\ c0\ k')$
by (*rule* *ar_advance_walk_loop_gen*
 $[OF\ ar_advance_walk_step\ vM\ qQ\ stg\ buf_valid\ ndisp\ notLE$
 $pad0\ src0]$)

Relation-generic unified single-tape advance scaffold (non-last tape).
 Both helper references of the dispatch — the walk loop and the bound-
 ary step — are abstracted as the *leaf_loop* and *leaf_boundary* hypothe-
 ses, so the union-relation *ar_advance_tape_step* and the sub-relation
ar_advance_tape_step_in_sub are both thin instantiations (discharging
 the two leaves by the union- resp. sub-relation helper lemmas). Multi-leaf
 instance of the leaf-parametric scaffold; no *stays* side-condition since both
 leaves' chains are native to R' .

lemma *ar_advance_tape_step_gen*:

fixes $M :: ('q, 'a)\ mttm$

and $c0 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$

and $R' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config\ rel$

assumes *leaf_loop*:

$\bigwedge c0' n. \llbracket valid_mttm\ M;$

$q \in Q_tm\ M;$

$mt_state\ c0' = (q, AR_SimAdvance, tk, 0, buf, dvec, posk);$

$\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\};$

$n < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk);$

$\bigwedge j. j < n \implies mt_tape\ c0'\ tk\ (mt_pos\ c0'\ tk - j) \neq LE4;$

$\forall j \geq k_tm\ M. mt_tape\ c0'\ j\ (mt_pos\ c0'\ j) = BLANK4;$

$ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$

$(AR_SimAdvance, tk, 0, buf, dvec, posk) \rrbracket$

$\implies \exists c'. (c0', c') \in R' \wedge^n$

$\wedge mt_state\ c' = (q, AR_SimAdvance, tk, n, buf, dvec, posk)$

$\wedge mt_pos\ c'\ tk = mt_pos\ c0'\ tk - n$

$\wedge mt_tape\ c' = mt_tape\ c0'$

$\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c'\ k' = mt_pos\ c0'\ k')$

and *leaf_boundary*:

$\bigwedge cc\ i. \llbracket valid_mttm\ M;$

$mt_state\ cc = (q, AR_SimAdvance, tk, i, buf, dvec, posk);$

$q \in Q_tm\ M;$

$\neg is_last_k\ M\ tk;$

$(dvec\ tk = dir.R \wedge i = 0)$

$\vee (dvec\ tk \neq dir.R$
 $\wedge mt_tape\ cc\ tk\ (mt_pos\ cc\ tk) \neq LE4$
 $\wedge Suc\ i = ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk$
 $tk));$
 $ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk);$
 $\forall j \geq k_tm\ M. mt_tape\ cc\ j\ (mt_pos\ cc\ j) = BLANK4;$
 $ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk) \parallel$
 $\implies \exists c''. (cc, c'') \in R'$
 $\wedge mt_state\ c'' = (q, AR_SimAdvance, k_succ\ tk, 0, buf,$
 $dvec,$
 $posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
 $\wedge mt_tape\ c'' = mt_tape\ cc$
 $\wedge mt_pos\ c'' = (if\ dvec\ tk = dir.R\ then\ mt_pos\ cc$
 $else\ (mt_pos\ cc)(tk := mt_pos\ cc\ tk - 1))$
and $vM: valid_mttm\ M$
and $qQ: q \in \bar{Q}_tm\ M$
and $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $notlast: \neg is_last_k\ M\ tk$
and $stg: mt_state\ c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)$
and $buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $dge1: dvec\ tk \neq dir.R$
 $\implies 0 < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
and $notLE: \bigwedge m. m < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
 $\implies mt_tape\ c0\ tk\ (mt_pos\ c0\ tk - m) \neq LE4$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in R'$
 $\wedge (if\ dvec\ tk = dir.R\ then\ 1$
 $else\ ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk))$
 $\wedge mt_state\ c' = (q, AR_SimAdvance, k_succ\ tk, 0, buf, dvec,$
 $posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
 $\wedge mt_tape\ c' = mt_tape\ c0$
 $\wedge mt_pos\ c' = (if\ dvec\ tk = dir.R\ then\ mt_pos\ c0$
 $else\ (mt_pos\ c0)(tk := mt_pos\ c0\ tk$
 $- ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk$
 $tk)))$
proof –
let $?k = block_width\ (\Gamma_tm\ M)$
let $?R = R'$
let $?D = ar_disp\ ?k\ (dvec\ tk)\ (posk\ tk)$
let $?start = mt_pos\ c0\ tk$
have $kpos: 0 < ?k$ **using** $kge2$ **by** $simp$
show $?thesis$
proof $(cases\ dvec\ tk = dir.R)$
case $True$
have $vsr0: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$

```

      (AR_SimAdvance, tk, 0, buf, dvec, posk)
    using kpos buf_valid by (simp add: ar_valid_stage_def)
  have fire: (dvec tk = dir.R ∧ (0::nat) = 0)
    ∨ (dvec tk ≠ dir.R ∧ mt_tape c0 tk (mt_pos c0 tk) ≠ LE4
      ∧ Suc 0 = ?D)
    using True by simp
  obtain c' where
    step: (c0, c') ∈ ?R
    and st: mt_state c' = (q, AR_SimAdvance, k_succ tk, 0, buf, dvec,
      posk(tk := ar_newpos (dvec tk) (posk tk)))
    and tp: mt_tape c' = mt_tape c0
    and ps: mt_pos c' = (if dvec tk = dir.R then mt_pos c0
      else (mt_pos c0)(tk := mt_pos c0 tk - 1))
    using leaf_boundary[OF vM stg qQ notlast fire vsrc0 pad0 src0] by blast
  show ?thesis
  proof (intro exI[where x = c'] conjI)
    show (c0, c') ∈ ?R ^^ (if dvec tk = dir.R then 1 else ?D)
      using step True by simp
    show mt_state c' = (q, AR_SimAdvance, k_succ tk, 0, buf, dvec,
      posk(tk := ar_newpos (dvec tk) (posk tk))) by (rule st)
    show mt_tape c' = mt_tape c0 by (rule tp)
    show mt_pos c' = (if dvec tk = dir.R then mt_pos c0
      else (mt_pos c0)(tk := mt_pos c0 tk - ?D))
      using ps True by simp
  qed
next
case False
have Dpos: 0 < ?D using dge1[OF False] by simp
obtain DD where DD: ?D = Suc DD using Dpos by (cases ?D) auto
have ndisp: DD < ?D using DD by simp
have notLE_loop: ∧j. j < DD ⇒ mt_tape c0 tk (?start - j) ≠ LE4
  proof -
    fix j assume j < DD
    hence j < ?D using DD by simp
    thus mt_tape c0 tk (?start - j) ≠ LE4 by (rule notLE)
  qed
obtain cmid where
  lstep: (c0, cmid) ∈ ?R ^^ DD
  and lst: mt_state cmid = (q, AR_SimAdvance, tk, DD, buf, dvec, posk)
  and lpos: mt_pos cmid tk = ?start - DD
  and ltape: mt_tape cmid = mt_tape c0
  and lother: ∀ k'. k' ≠ tk ⇒ mt_pos cmid k' = mt_pos c0 k'
  using leaf_loop[OF vM qQ stg buf_valid ndisp notLE_loop pad0 src0] by
blast
have DD_lt2k: DD < 2 * ?k
  using DD ar_disp_le_2k[OF kpos, of dvec tk posk tk] by linarith
have vsrcmid: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimAdvance, tk, DD, buf, dvec, posk)
  using DD_lt2k buf_valid by (simp add: ar_valid_stage_def)

```

```

have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
have pad_cmid:  $\forall j \geq k\_tm\ M. mt\_tape\ cmid\ j\ (mt\_pos\ cmid\ j) = BLANK4$ 
proof (intro allI impI)
  fix j assume jge: k_tm M  $\leq j$ 
  have jne: j  $\neq tk$  using jge tk_lt by simp
  have mt_tape_cmid_j (mt_pos cmid j) = mt_tape c0 j (mt_pos c0 j)
    using ltape lother jne by simp
  thus mt_tape cmid j (mt_pos cmid j) = BLANK4 using pad0 jge by simp
qed
have src_cmid: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimAdvance, tk, DD, buf, dvec, posk)
  using src0 by (simp add: ar_stage_bounded_def)
have notLE_b: mt_tape cmid tk (mt_pos cmid tk)  $\neq LE4$ 
  using ltape lpos notLE[OF ndisp] by simp
have fire: (dvec tk = dir.R  $\wedge$  DD = 0)
   $\vee$  (dvec tk  $\neq$  dir.R
     $\wedge$  mt_tape cmid tk (mt_pos cmid tk)  $\neq LE4$ 
     $\wedge$  Suc DD = ?D)
  using False notLE_b DD by simp
obtain c' where
  bstep: (cmid, c')  $\in$  ?R
  and bst: mt_state c' = (q, AR_SimAdvance, k_succ tk, 0, buf, dvec,
    posk(tk := ar_newpos (dvec tk) (posk tk)))
  and btp: mt_tape c' = mt_tape cmid
  and bps: mt_pos c' = (if dvec tk = dir.R then mt_pos cmid
    else (mt_pos cmid)(tk := mt_pos cmid tk - 1))
  using leaf_boundary[OF vM lst qQ notlast fire vsrcmid pad_cmid src_cmid]
by blast
have bps_ne: mt_pos c' = (mt_pos cmid)(tk := mt_pos cmid tk - 1)
  using bps False by simp
have chainD: (c0, c')  $\in$  ?R  $\wedge$  ?D
proof -
  have (c0, c')  $\in$  ?R  $\wedge$  Suc DD by (rule relpow_Suc_I[OF lstep bstep])
  thus ?thesis using DD by simp
qed
have ps_final: mt_pos c' = (mt_pos c0)(tk := ?start - ?D)
proof (rule ext)
  fix k'
  show mt_pos c' k' = ((mt_pos c0)(tk := ?start - ?D)) k'
proof (cases k' = tk)
  case True
  have mt_pos_c'_tk = mt_pos cmid tk - 1 using bps_ne by simp
  also have ... = (?start - DD) - 1 using lpos by simp
  also have ... = ?start - ?D using DD by simp
  finally show ?thesis using True by simp
next
  case False
  have mt_pos_c'_k' = mt_pos cmid k' using bps_ne False by simp
  also have ... = mt_pos c0 k' using lother False by simp

```



```

    finally show ?thesis using False by simp
  qed
qed
have tp_final: mt_tape c' = mt_tape c0 using btp ltape by simp
show ?thesis
proof (intro exI[where x = c'] conjI)
  show (c0, c') ∈ ?R ^^ (if dvec tk = dir.R then 1 else ?D)
    using chainD False by simp
  show mt_state c' = (q, AR_SimAdvance, k_succ tk, 0, buf, dvec,
    posk(tk := ar_newpos (dvec tk) (posk tk))) by (rule bst)
  show mt_tape c' = mt_tape c0 by (rule tp_final)
  show mt_pos c' = (if dvec tk = dir.R then mt_pos c0
    else (mt_pos c0)(tk := mt_pos c0 tk - ?D))
    using ps_final False by simp
  qed
qed
qed

```

The unified single-tape advance, non-last tape: the R /non- R dispatch, the advance analogue of $ar_write_tape_step$. From an $AR_SimAdvance$ stage at bit-counter 0 , tape tk 's head is repositioned to M 's new cell and the hand-off goes to the next tape $k_succ\ tk$ (counter 0 , $posk\ tk$ updated by ar_newpos). An R -move needs no head motion (the head already sits at $sim_pos\ (p + 1)$): a single boundary step, cost 1 . A non- R -move walks the head $D = ar_disp$ cells L : the $D - 1$ -step $ar_advance_walk_loop$ followed by the final boundary L -move, cost D , landing the head at $start - D$. The $dge1$ hypothesis rules out the forbidden L -from- AR_AtLE (zero displacement) so that $D \geq 1$; $notLE$ covers every cell the walk reads ($start - m$, $m < D$). The tape is unchanged; only the head on tk moves.

```

lemma ar_advance_tape_step:
  fixes M :: ('q, 'a) mttm
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and notlast: ¬ is_last_k M tk
  and stg: mt_state c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and dge1: dvec tk ≠ dir.R
  ⇒ 0 < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
  and notLE: ∧ m. m < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
    ⇒ mt_tape c0 tk (mt_pos c0 tk - m) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c0 j (mt_pos c0 j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimAdvance, tk, 0, buf, dvec, posk)
  shows ∃ c'. (c0, c') ∈ (mttm_step (alphabet_reduce_delta M))
    ^^ (if dvec tk = dir.R then 1
      else ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk))

```

$$\begin{aligned}
& \wedge \text{mt_state } c' = (q, \text{AR_SimAdvance}, k_succ \text{ tk}, 0, \text{buf}, \text{dvec}, \\
& \quad \text{posk}(tk := \text{ar_newpos}(\text{dvec tk})(\text{posk tk}))) \\
& \wedge \text{mt_tape } c' = \text{mt_tape } c0 \\
& \wedge \text{mt_pos } c' = (\text{if } \text{dvec tk} = \text{dir.R} \text{ then } \text{mt_pos } c0 \\
& \quad \text{else } (\text{mt_pos } c0)(tk := \text{mt_pos } c0 \text{ tk} \\
& \quad \quad - \text{ar_disp}(\text{block_width}(\Gamma_tm M))(\text{dvec tk})(\text{posk} \\
& \quad \text{tk}))) \\
& \text{by } (\text{rule } \text{ar_advance_tape_step_gen} \\
& \quad [\text{OF } \text{ar_advance_walk_loop } \text{ar_advance_boundary_step} \\
& \quad \quad vM \text{ } qQ \text{ } kge2 \text{ notlast stg } \text{buf_valid } dge1 \text{ notLE} \\
& \quad \quad \text{pad0 src0}])
\end{aligned}$$

Relation-generic unified single-tape advance scaffold (last tape). As *ar_advance_tape_step_gen* but the boundary leaf transitions to *AR_SimNext*; both helper references (walk loop, finish step) are the *leaf_loop* and *leaf_finish* hypotheses, so the union- and sub-relation versions are thin instantiations.

lemma *ar_advance_tape_finish_step_gen*:

$$\begin{aligned}
& \text{fixes } M :: ('q, 'a) \text{ mttm} \\
& \quad \text{and } c0 :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \\
& \quad \text{and } R' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config rel} \\
& \text{assumes } \text{leaf_loop}: \\
& \quad \wedge c0' n. \llbracket \text{valid_mttm } M; \\
& \quad \quad q \in Q_tm M; \\
& \quad \quad \text{mt_state } c0' = (q, \text{AR_SimAdvance}, tk, 0, \text{buf}, \text{dvec}, \text{posk}); \\
& \quad \quad \forall k. \text{buf } k \in \Gamma_tm M \cup \{bl_tm M\}; \\
& \quad \quad n < \text{ar_disp}(\text{block_width}(\Gamma_tm M))(\text{dvec tk})(\text{posk tk}); \\
& \quad \quad \bigwedge j. j < n \implies \text{mt_tape } c0' tk (\text{mt_pos } c0' tk - j) \neq LE4; \\
& \quad \quad \forall j \geq k_tm M. \text{mt_tape } c0' j (\text{mt_pos } c0' j) = BLANK4; \\
& \quad \quad \text{ar_stage_bounded}(bl_tm M)(k_tm M) \\
& \quad \quad (\text{AR_SimAdvance}, tk, 0, \text{buf}, \text{dvec}, \text{posk}) \rrbracket \\
& \implies \exists c'. (c0', c') \in R' \wedge \wedge n \\
& \quad \wedge \text{mt_state } c' = (q, \text{AR_SimAdvance}, tk, n, \text{buf}, \text{dvec}, \text{posk}) \\
& \quad \wedge \text{mt_pos } c' tk = \text{mt_pos } c0' tk - n \\
& \quad \wedge \text{mt_tape } c' = \text{mt_tape } c0' \\
& \quad \wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c' k' = \text{mt_pos } c0' k') \\
& \text{and } \text{leaf_finish}: \\
& \quad \wedge cc i. \llbracket \text{valid_mttm } M; \\
& \quad \quad \text{mt_state } cc = (q, \text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk}); \\
& \quad \quad q \in Q_tm M; \\
& \quad \quad \text{is_last_k } M tk; \\
& \quad \quad (\text{dvec tk} = \text{dir.R} \wedge i = 0) \\
& \quad \quad \vee (\text{dvec tk} \neq \text{dir.R} \\
& \quad \quad \quad \wedge \text{mt_tape } cc tk (\text{mt_pos } cc tk) \neq LE4 \\
& \quad \quad \quad \wedge \text{Suc } i = \text{ar_disp}(\text{block_width}(\Gamma_tm M))(\text{dvec tk})(\text{posk} \\
& \quad \text{tk})); \\
& \quad \text{ar_valid_stage}(\Gamma_tm M)(bl_tm M) \\
& \quad (\text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk}); \\
& \quad \forall j \geq k_tm M. \text{mt_tape } cc j (\text{mt_pos } cc j) = BLANK4;
\end{aligned}$$

```

      ar_stage_bounded (bl_tm M) (k_tm M)
      (AR_SimAdvance, tk, i, buf, dvec, posk) ]
    ==> ∃ c''. (cc, c'') ∈ R'
      ∧ mt_state c'' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,
        posk(tk := ar_newpos (dvec tk) (posk tk)))
      ∧ mt_tape c'' = mt_tape cc
      ∧ mt_pos c'' = (if dvec tk = dir.R then mt_pos cc
        else (mt_pos cc)(tk := mt_pos cc tk - 1))

  and vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and last: is_last_k M tk
  and stg: mt_state c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and dge1: dvec tk ≠ dir.R
    ==> 0 < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
  and notLE: ∧ m. m < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
    ==> mt_tape c0 tk (mt_pos c0 tk - m) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c0 j (mt_pos c0 j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimAdvance, tk, 0, buf, dvec, posk)
shows ∃ c'. (c0, c') ∈ R'
  ^^ (if dvec tk = dir.R then 1
    else ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk))
  ∧ mt_state c' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,
    posk(tk := ar_newpos (dvec tk) (posk tk)))
  ∧ mt_tape c' = mt_tape c0
  ∧ mt_pos c' = (if dvec tk = dir.R then mt_pos c0
    else (mt_pos c0)(tk := mt_pos c0 tk
      - ar_disp (block_width (Γ_tm M)) (dvec tk) (posk
        tk)))
proof -
  let ?k = block_width (Γ_tm M)
  let ?R = R'
  let ?D = ar_disp ?k (dvec tk) (posk tk)
  let ?start = mt_pos c0 tk
  have kpos: 0 < ?k using kge2 by simp
  show ?thesis
  proof (cases dvec tk = dir.R)
    case True
    have vsrc0: ar_valid_stage (Γ_tm M) (bl_tm M)
      (AR_SimAdvance, tk, 0, buf, dvec, posk)
    using kpos buf_valid by (simp add: ar_valid_stage_def)
    have fire: (dvec tk = dir.R ∧ (0::nat) = 0)
      ∨ (dvec tk ≠ dir.R ∧ mt_tape c0 tk (mt_pos c0 tk) ≠ LE4
        ∧ Suc 0 = ?D)
    using True by simp
    obtain c' where
      step: (c0, c') ∈ ?R

```

```

and st: mt_state c' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,
  posk(tk := ar_newpos (dvec tk) (posk tk)))
and tp: mt_tape c' = mt_tape c0
and ps: mt_pos c' = (if dvec tk = dir.R then mt_pos c0
  else (mt_pos c0)(tk := mt_pos c0 tk - 1))
using leaf_finish[OF vM stg qQ last fire vsrc0 pad0 src0] by blast
show ?thesis
proof (intro exI[where x = c'] conjI)
  show (c0, c') ∈ ?R ^^ (if dvec tk = dir.R then 1 else ?D)
    using step True by simp
  show mt_state c' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,
    posk(tk := ar_newpos (dvec tk) (posk tk))) by (rule st)
  show mt_tape c' = mt_tape c0 by (rule tp)
  show mt_pos c' = (if dvec tk = dir.R then mt_pos c0
    else (mt_pos c0)(tk := mt_pos c0 tk - ?D))
    using ps True by simp
qed
next
case False
have Dpos: 0 < ?D using dge1[OF False] by simp
obtain DD where DD: ?D = Suc DD using Dpos by (cases ?D) auto
have ndisp: DD < ?D using DD by simp
have notLE_loop:  $\bigwedge j. j < DD \implies \text{mt\_tape } c0 \text{ tk } (?start - j) \neq LE4$ 
proof -
  fix j assume j < DD
  hence j < ?D using DD by simp
  thus mt_tape c0 tk (?start - j)  $\neq$  LE4 by (rule notLE)
qed
obtain cmid where
  lstep: (c0, cmid) ∈ ?R ^^ DD
  and lst: mt_state cmid = (q, AR_SimAdvance, tk, DD, buf, dvec, posk)
  and lpos: mt_pos cmid tk = ?start - DD
  and ltape: mt_tape cmid = mt_tape c0
  and lother:  $\forall k'. k' \neq tk \longrightarrow \text{mt\_pos cmid } k' = \text{mt\_pos c0 } k'$ 
  using leaf_loop[OF vM qQ stg buf_valid ndisp notLE_loop pad0 src0] by
blast
have DD_lt2k: DD < 2 * ?k
  using DD ar_disp_le_2k[OF kpos, of dvec tk posk tk] by linarith
have vsrcmid: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimAdvance, tk, DD, buf, dvec, posk)
  using DD_lt2k buf_valid by (simp add: ar_valid_stage_def)
have tk_lt: tk < k_tm M using src0 by (simp add: ar_stage_bounded_def)
have pad_cmid:  $\forall j \geq k\_tm \ M. \text{mt\_tape cmid } j (\text{mt\_pos cmid } j) = \text{BLANK4}$ 
proof (intro allI impI)
  fix j assume jge: k_tm M ≤ j
  have jne: j  $\neq$  tk using jge tk_lt by simp
  have mt_tape cmid j (mt_pos cmid j) = mt_tape c0 j (mt_pos c0 j)
    using ltape lother jne by simp
  thus mt_tape cmid j (mt_pos cmid j) = BLANK4 using pad0 jge by simp

```

```

qed
have src_cmidx: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimAdvance, tk, DD, buf, dvec, posk)
  using src0 by (simp add: ar_stage_bounded_def)
have notLE_b: mt_tape cmidx tk (mt_pos cmidx tk)  $\neq$  LE4
  using ltape lpos notLE[OF ndisp] by simp
have fire: (dvec tk = dir.R  $\wedge$  DD = 0)
   $\vee$  (dvec tk  $\neq$  dir.R
     $\wedge$  mt_tape cmidx tk (mt_pos cmidx tk)  $\neq$  LE4
     $\wedge$  Suc DD = ?D)
  using False notLE_b DD by simp
obtain c' where
  bstep: (cmidx, c')  $\in$  ?R
  and bst: mt_state c' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,
    posk(tk := ar_newpos (dvec tk) (posk tk)))
  and btp: mt_tape c' = mt_tape cmidx
  and bps: mt_pos c' = (if dvec tk = dir.R then mt_pos cmidx
    else (mt_pos cmidx)(tk := mt_pos cmidx tk - 1))
  using leaf_finish[OF vM lst qQ last fire vsrcmid pad_cmidx src_cmidx] by blast
have bps_ne: mt_pos c' = (mt_pos cmidx)(tk := mt_pos cmidx tk - 1)
  using bps False by simp
have chainD: (c0, c')  $\in$  ?R  $\wedge$  ?D
proof -
  have (c0, c')  $\in$  ?R  $\wedge$  Suc DD by (rule relpow_Suc_I[OF lstep bstep])
  thus ?thesis using DD by simp
qed
have ps_final: mt_pos c' = (mt_pos c0)(tk := ?start - ?D)
proof (rule ext)
  fix k'
  show mt_pos c' k' = ((mt_pos c0)(tk := ?start - ?D)) k'
  proof (cases k' = tk)
    case True
    have mt_pos c' tk = mt_pos cmidx tk - 1 using bps_ne by simp
    also have ... = (?start - DD) - 1 using lpos by simp
    also have ... = ?start - ?D using DD by simp
    finally show ?thesis using True by simp
  next
    case False
    have mt_pos c' k' = mt_pos cmidx k' using bps_ne False by simp
    also have ... = mt_pos c0 k' using lother False by simp
    finally show ?thesis using False by simp
  qed
qed
have tp_final: mt_tape c' = mt_tape c0 using btp ltape by simp
show ?thesis
proof (intro exI[where x = c'] conjI)
  show (c0, c')  $\in$  ?R  $\wedge$  (if dvec tk = dir.R then 1 else ?D)
    using chainD False by simp
  show mt_state c' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,

```

```

      posk(tk := ar_newpos (dvec tk) (posk tk))) by (rule bst)
    show mt_tape c' = mt_tape c0 by (rule tp_final)
    show mt_pos c' = (if dvec tk = dir.R then mt_pos c0
      else (mt_pos c0)(tk := mt_pos c0 tk - ?D))
    using ps_final False by simp
  qed
qed
qed

```

The unified single-tape advance, last tape: as *ar_advance_tape_step* but *tk* is the last tape, so the boundary step (*ar_advance_finish_step*) transitions to *AR_SimNext* (current-tape field reset to *k_unidx 0*) rather than advancing to *k_succ tk*. Same *R*/non-*R* dispatch, the same walk-then-boundary decomposition, and the same conditional head conclusion.

```

lemma ar_advance_tape_finish_step:
  fixes M :: ('q, 'a) mttm
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and last: is_last_k M tk
  and stg: mt_state c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)
  and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
  and dge1: dvec tk ≠ dir.R
  ⇒ 0 < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
  and notLE: ∧ m. m < ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk)
    ⇒ mt_tape c0 tk (mt_pos c0 tk - m) ≠ LE4
  and pad0: ∀ j ≥ k_tm M. mt_tape c0 j (mt_pos c0 j) = BLANK4
  and src0: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimAdvance, tk, 0, buf, dvec, posk)
  shows ∃ c'. (c0, c') ∈ (mttm_step (alphabet_reduce_delta M))
    ^ (if dvec tk = dir.R then 1
      else ar_disp (block_width (Γ_tm M)) (dvec tk) (posk tk))
    ∧ mt_state c' = (q, AR_SimNext, k_unidx 0, 0, buf, dvec,
      posk(tk := ar_newpos (dvec tk) (posk tk)))
    ∧ mt_tape c' = mt_tape c0
    ∧ mt_pos c' = (if dvec tk = dir.R then mt_pos c0
      else (mt_pos c0)(tk := mt_pos c0 tk
        - ar_disp (block_width (Γ_tm M)) (dvec tk) (posk
          tk)))
  by (rule ar_advance_tape_finish_step_gen
    [OF ar_advance_walk_loop ar_advance_finish_step
      vM qQ kge2 last stg buf_valid dge1 notLE
      pad0 src0])

```

Relation-generic advance prefix walk scaffold. The single per-tape helper (*ar_advance_tape_step*) is the *leaf_tape* hypothesis, quantified over the per-tape config, tape index, and position-kind function; the union- and sub-relation prefixes are thin instantiations.

lemma *ar_advance_prefix_gen*:

fixes $M :: ('q, 'a) \text{mttm}$

and $c0 :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$

and $R' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config rel}$

assumes *leaf_tape*:

$\bigwedge cc \text{ tk}' \text{ pk}'.$

$\llbracket \text{valid_mttm } M; q \in Q_tm \text{ } M; 2 \leq \text{block_width } (\Gamma_tm \text{ } M);$

$\neg \text{is_last_k } M \text{ tk}';$

$\text{mt_state } cc = (q, \text{AR_SimAdvance}, \text{tk}', 0, \text{buf0}, \text{dvec}, \text{pk}');$

$\forall k. \text{buf0 } k \in \Gamma_tm \text{ } M \cup \{\text{bl_tm } M\};$

$\text{dvec } \text{tk}' \neq \text{dir}.R$

$\implies 0 < \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } \text{tk}') (\text{pk}' \text{ tk}');$

$\bigwedge m. m < \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } \text{tk}') (\text{pk}' \text{ tk}')$

$\implies \text{mt_tape } cc \text{ tk}' (\text{mt_pos } cc \text{ tk}' - m) \neq \text{LE4};$

$\forall j \geq k_tm \text{ } M. \text{mt_tape } cc \text{ } j (\text{mt_pos } cc \text{ } j) = \text{BLANK4};$

$\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \text{ } M)$

$(\text{AR_SimAdvance}, \text{tk}', 0, \text{buf0}, \text{dvec}, \text{pk}') \rrbracket$

$\implies \exists c'. (cc, c') \in R'$

$\wedge \wedge (\text{if } \text{dvec } \text{tk}' = \text{dir}.R \text{ then } 1$

$\text{else } \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } \text{tk}') (\text{pk}' \text{ tk}'))$

$\wedge \text{mt_state } c' = (q, \text{AR_SimAdvance}, k_succ \text{ tk}', 0, \text{buf0}, \text{dvec},$

$\text{pk}'(\text{tk}' := \text{ar_newpos } (\text{dvec } \text{tk}') (\text{pk}' \text{ tk}'))$

$\wedge \text{mt_tape } c' = \text{mt_tape } cc$

$\wedge \text{mt_pos } c' = (\text{if } \text{dvec } \text{tk}' = \text{dir}.R \text{ then } \text{mt_pos } cc$

$\text{else } (\text{mt_pos } cc)(\text{tk}' := \text{mt_pos } cc \text{ tk}'$

$- \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } \text{tk}') (\text{pk}' \text{ tk}'))$

and $vM: \text{valid_mttm } M$

and $qQ: q \in Q_tm \text{ } M$

and $kge2: 2 \leq \text{block_width } (\Gamma_tm \text{ } M)$

and $stg0: \text{mt_state } c0 = (q, \text{AR_SimAdvance}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$

and $\text{buf_valid}: \forall k. \text{buf0 } k \in \Gamma_tm \text{ } M \cup \{\text{bl_tm } M\}$

and $\text{dge1}: \forall k < k_tm \text{ } M. \text{dvec } k \neq \text{dir}.R$

$\implies 0 < \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } k) (\text{posk0 } k)$

and $\text{notLE}: \forall k < k_tm \text{ } M. \forall m. m < \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M))$

$(\text{dvec } k) (\text{posk0 } k)$

$\implies \text{mt_tape } c0 \text{ } k (\text{mt_pos } c0 \text{ } k - m) \neq \text{LE4}$

and $\text{pad0}: \forall j \geq k_tm \text{ } M. \text{mt_tape } c0 \text{ } j (\text{mt_pos } c0 \text{ } j) = \text{BLANK4}$

and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm \text{ } M)$

$(\text{AR_SimAdvance}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$

shows $j \leq k_tm \text{ } M - 1 \implies$

$(\exists c \text{ } m. (c0, c) \in R' \wedge \wedge m$

$\wedge m \leq j * (2 * \text{block_width } (\Gamma_tm \text{ } M))$

$\wedge \text{mt_state } c = (q, \text{AR_SimAdvance}, k_unidx \text{ } j, 0, \text{buf0}, \text{dvec},$

$(\lambda k. \text{if } k_idx \text{ } k < j \text{ then } \text{ar_newpos } (\text{dvec } k) (\text{posk0 } k)$

$\text{else } \text{posk0 } k))$

$\wedge \text{mt_tape } c = \text{mt_tape } c0$

$\wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx \text{ } k < j$

$\text{then } (\text{if } \text{dvec } k = \text{dir}.R \text{ then } \text{mt_pos } c0 \text{ } k$

$\text{else } \text{mt_pos } c0 \text{ } k$

```

      - ar_disp (block_width (Γ_tm M)) (dvec k) (posk0 k))
    else mt_pos c0 k))
proof (induction j)
  case 0
  let ?pk0 = λk. if k_idx k < (0::nat) then ar_newpos (dvec k) (posk0 k)
    else posk0 k
  have (c0, c0) ∈ R' ^ 0 by simp
  moreover have mt_state c0 = (q, AR_SimAdvance, k_unidx 0, 0, buf0, dvec,
    ?pk0)
    using stg0 by (simp add: k_unidx_zero)
  ultimately show ?case
    by (intro exI[where x = c0] exI[where x = 0]) simp
next
  case (Suc j)
  have sucle: Suc j ≤ k_tm M - 1 by (rule Suc.prem)
  have jcard: j < k_tm M using sucle by simp
  have sjcard: Suc j < k_tm M using sucle by simp
  have jle: j ≤ k_tm M - 1 using sucle by simp
  let ?k = block_width (Γ_tm M)
  let ?R = R'
  let ?tk = k_unidx j
  let ?pk = λi. (λk. if k_idx k < i then ar_newpos (dvec k) (posk0 k)
    else posk0 k)
  let ?ps = λi. (λk. if k_idx k < i
    then (if dvec k = dir.R then mt_pos c0 k
      else mt_pos c0 k - ar_disp ?k (dvec k) (posk0 k))
    else mt_pos c0 k)
  have kpos: 0 < ?k using kge2 by simp
  have tkidx: k_idx ?tk = j by (rule k_idx_unidx[OF jcard])
  have notlast: ¬ is_last_k M ?tk
    using is_last_k_unidx[OF jcard] sjcard by simp
  have ksucc: k_succ ?tk = k_unidx (Suc j) by (rule k_succ_unidx[OF sjcard])
  have inj: inj_on k_idx UNIV by (rule k_idx_inj)
  have kne: ∧k. k_idx k = j ⇒ k = ?tk
  proof -
    fix k assume k_idx k = j
    hence k_idx k = k_idx ?tk using tkidx by simp
    thus k = ?tk using inj_on_eq_iff[OF inj UNIV_I UNIV_I] by simp
  qed
  obtain c m where
    cm_rel: (c0, c) ∈ ?R ^ m
    and m_le: m ≤ j * (2 * ?k)
    and c_st: mt_state c = (q, AR_SimAdvance, ?tk, 0, buf0, dvec, ?pk j)
    and c_tp: mt_tape c = mt_tape c0
    and c_ps: mt_pos c = ?ps j
    using Suc.IH[OF jle] by blast
  — per-tape entry facts for tape ?tk at config c
  have pkj_tk: ?pk j ?tk = posk0 ?tk using tkidx by simp
  have psj_tk: ?ps j ?tk = mt_pos c0 ?tk using tkidx by simp

```



```

have c_pos_tk: mt_pos c ?tk = mt_pos c0 ?tk using c_ps psj_tk by simp
have tk_active: ?tk < k_tm M using jcard by (simp add: k_unidx_def)
have tk_dge1: dvec ?tk ≠ dir.R ⇒ 0 < ar_disp ?k (dvec ?tk) (?pk j ?tk)
  using dge1[rule_format, OF tk_active] pkj_tk by simp
have tk_notLE: ∧m. m < ar_disp ?k (dvec ?tk) (?pk j ?tk)
  ⇒ mt_tape c ?tk (mt_pos c ?tk - m) ≠ LE4

proof -
  fix m assume m < ar_disp ?k (dvec ?tk) (?pk j ?tk)
  hence mlt: m < ar_disp ?k (dvec ?tk) (posk0 ?tk) using pkj_tk by simp
  have mt_tape c ?tk (mt_pos c ?tk - m) = mt_tape c0 ?tk (mt_pos c0 ?tk -
m)
    using c_tp c_pos_tk by simp
  thus mt_tape c ?tk (mt_pos c ?tk - m) ≠ LE4
    using notLE[rule_format, OF tk_active mlt] by simp
qed
have tail_idx: ∧j'. k_tm M ≤ j' ⇒ ¬ k_idx j' < j
  using jcard unfolding k_idx_def by linarith
have pad_c: ∀j' ≥ k_tm M. mt_tape c j' (mt_pos c j') = BLANK4
proof (intro allI impI)
  fix j' assume jge: k_tm M ≤ j'
  have mt_tape c j' (mt_pos c j') = mt_tape c0 j' (mt_pos c0 j')
    using c_tp c_ps tail_idx[OF jge] by simp
  thus mt_tape c j' (mt_pos c j') = BLANK4 using pad0 jge by simp
qed
have src_c: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimAdvance, ?tk, 0, buf0, dvec, ?pk j)
  using src0 tk_active tail_idx by (auto simp: ar_stage_bounded_def)
obtain c' where
  step: (c, c') ∈ ?R ^^ (if dvec ?tk = dir.R then 1
    else ar_disp ?k (dvec ?tk) (?pk j ?tk))
  and c'_st: mt_state c' = (q, AR_SimAdvance, k_succ ?tk, 0, buf0, dvec,
    (?pk j)(?tk := ar_newpos (dvec ?tk) (?pk j ?tk)))
  and c'_tp: mt_tape c' = mt_tape c
  and c'_ps: mt_pos c' = (if dvec ?tk = dir.R then mt_pos c
    else (mt_pos c)(?tk := mt_pos c ?tk
      - ar_disp ?k (dvec ?tk) (?pk j ?tk)))
  using leaf_tape[OF vM qQ kge2 notlast c_st buf_valid tk_dge1 tk_notLE
    pad_c src_c]
  by blast
— the post-state descriptors coincide with the Suc j ones
have pk_eq: (?pk j)(?tk := ar_newpos (dvec ?tk) (?pk j ?tk)) = ?pk (Suc j)
proof (rule ext)
  fix k
  show ((?pk j)(?tk := ar_newpos (dvec ?tk) (?pk j ?tk))) k = ?pk (Suc j) k
  proof (cases k = ?tk)
    case True thus ?thesis using tkidx by simp
  next
    case False
    hence k_idx k ≠ j using kne by blast

```

```

      thus ?thesis using False by (simp add: less_Suc_eq)
    qed
  qed
  have ps_eq: (if dvec ?tk = dir.R then mt_pos c
    else (mt_pos c)(?tk := mt_pos c ?tk
      - ar_disp ?k (dvec ?tk) (?pk j ?tk))) = ?ps (Suc j)
  proof (rule ext)
    fix k
    show (if dvec ?tk = dir.R then mt_pos c
      else (mt_pos c)(?tk := mt_pos c ?tk
        - ar_disp ?k (dvec ?tk) (?pk j ?tk))) k = ?ps (Suc j) k
    proof (cases k = ?tk)
      case True thus ?thesis using c_ps tkidx by simp
    next
      case False
      hence k_idx k ≠ j using kne by blast
      thus ?thesis using False c_ps by (simp add: less_Suc_eq)
    qed
  qed
  have chain: (c0, c') ∈ ?R ^^ (m + (if dvec ?tk = dir.R then 1
    else ar_disp ?k (dvec ?tk) (?pk j ?tk)))
  proof -
    have (c0, c') ∈ ?R ^^ m
      O ?R ^^ (if dvec ?tk = dir.R then 1
        else ar_disp ?k (dvec ?tk) (?pk j ?tk))
    using cm_rel step by (rule relcompI)
    thus ?thesis by (simp add: relpow_add)
  qed
  have bound: m + (if dvec ?tk = dir.R then 1
    else ar_disp ?k (dvec ?tk) (?pk j ?tk)) ≤ Suc j * (2 * ?k)
  using m_le kpos ar_disp_le_2k[OF kpos, of dvec ?tk ?pk j ?tk]
  by (cases dvec ?tk = dir.R) auto
  show ?case
  proof (intro exI[where x = c'])
    exI[where x = m + (if dvec ?tk = dir.R then 1
      else ar_disp ?k (dvec ?tk) (?pk j ?tk))] conjI)
    show (c0, c') ∈ ?R ^^ (m + (if dvec ?tk = dir.R then 1
      else ar_disp ?k (dvec ?tk) (?pk j ?tk))) by (rule chain)
    show m + (if dvec ?tk = dir.R then 1
      else ar_disp ?k (dvec ?tk) (?pk j ?tk)) ≤ Suc j * (2 * ?k)
    by (rule bound)
    show mt_state c' = (q, AR_SimAdvance, k_unidx (Suc j), 0, buf0, dvec,
      ?pk (Suc j))
    using c'_st ksucc pk_eq by simp
    show mt_tape c' = mt_tape c0 using c'_tp c_tp by simp
    show mt_pos c' = ?ps (Suc j) using c'_ps ps_eq by simp
  qed
qed

```

The advance prefix walk: a custom induction on the tape index j

($j \leq k_tm \ M - 1$, so every tape it touches is non-last), iterating $ar_advance_tape_step$ from the boundary tape $k_unidx \ 0$. Like the read prefix, the tape is constant and the carried state is a $k_idx \ k < j$ split: tapes already advanced ($k_idx \ k < j$) hold their ar_newpos -updated position-kind and their repositioned head, the rest hold boundary values. The per-tape $notLE/dge1$ entry facts hold of the current tape $?tk$ because it is not yet visited ($?pk \ j \ ?tk = posk0 \ ?tk$, $mt_pos \ c \ ?tk = mt_pos \ c0 \ ?tk$). Aggregate cost $\leq j \cdot 2b$ (each per-tape move is at most $2b$ cells, $ar_disp_le_2k$).

lemma $ar_advance_prefix$:

fixes $M :: ('q, 'a) \ mttm$
and $c0 :: (sym4, 'q \times 'a \ ar_stage) \ mt_config$
assumes $vM: valid_mttm \ M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq block_width \ (\Gamma_tm \ M)$
and $stg0: mt_state \ c0 = (q, AR_SimAdvance, 0, 0, buf0, dvec, posk0)$
and $buf_valid: \forall k. buf0 \ k \in \Gamma_tm \ M \cup \{bl_tm \ M\}$
and $dge1: \forall k < k_tm \ M. dvec \ k \neq dir.R$
 $\longrightarrow 0 < ar_disp \ (block_width \ (\Gamma_tm \ M)) \ (dvec \ k) \ (posk0 \ k)$
and $notLE: \forall k < k_tm \ M. \forall m. m < ar_disp \ (block_width \ (\Gamma_tm \ M))$
 $(dvec \ k) \ (posk0 \ k)$
 $\longrightarrow mt_tape \ c0 \ k \ (mt_pos \ c0 \ k - m) \neq LE4$
and $pad0: \forall j \geq k_tm \ M. mt_tape \ c0 \ j \ (mt_pos \ c0 \ j) = BLANK4$
and $src0: ar_stage_bounded \ (bl_tm \ M) \ (k_tm \ M)$
 $(AR_SimAdvance, 0, 0, buf0, dvec, posk0)$
shows $j \leq k_tm \ M - 1 \implies$
 $(\exists c \ m. (c0, c) \in (mttm_step \ (alphabet_reduce_delta \ M)) \ \wedge^{\wedge} m$
 $\wedge m \leq j * (2 * block_width \ (\Gamma_tm \ M))$
 $\wedge mt_state \ c = (q, AR_SimAdvance, k_unidx \ j, 0, buf0, dvec,$
 $(\lambda k. \text{if } k_idx \ k < j \text{ then } ar_newpos \ (dvec \ k) \ (posk0 \ k)$
 $\text{else } posk0 \ k))$
 $\wedge mt_tape \ c = mt_tape \ c0$
 $\wedge mt_pos \ c = (\lambda k. \text{if } k_idx \ k < j$
 $\text{then } (\text{if } dvec \ k = dir.R \text{ then } mt_pos \ c0 \ k$
 $\text{else } mt_pos \ c0 \ k$
 $\text{--- } ar_disp \ (block_width \ (\Gamma_tm \ M)) \ (dvec \ k) \ (posk0 \ k))$
 $\text{else } mt_pos \ c0 \ k))$
by $(rule \ ar_advance_prefix_gen$
 $[OF \ ar_advance_tape_step \ vM \ qQ \ kge2 \ stg0 \ buf_valid \ dge1 \ notLE$
 $pad0 \ src0])$

Relation-generic full advance phase scaffold. The two helpers (prefix walk, last-tape finish) are the $leaf_prefix$ and $leaf_finish$ hypotheses; the union- and sub-relation phases are thin instantiations.

lemma $ar_advance_phase_gen$:

fixes $M :: ('q, 'a) \ mttm$
and $c0 :: (sym4, 'q \times 'a \ ar_stage) \ mt_config$
and $R' :: (sym4, 'q \times 'a \ ar_stage) \ mt_config \ rel$
assumes $leaf_prefix$:

$\wedge jj. \llbracket \text{valid_mttm } M; q \in Q_tm\ M; 2 \leq \text{block_width } (\Gamma_tm\ M);$
 $\text{mt_state } c0 = (q, AR_SimAdvance, 0, 0, \text{buf0}, \text{dvec}, \text{posk0});$
 $\forall k. \text{buf0 } k \in \Gamma_tm\ M \cup \{bl_tm\ M\};$
 $\forall k < k_tm\ M. \text{dvec } k \neq \text{dir}.R$
 $\longrightarrow 0 < \text{ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } k) (\text{posk0 } k);$
 $\forall k < k_tm\ M. \forall m. m < \text{ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } k) (\text{posk0 } k)$
 $\longrightarrow \text{mt_tape } c0\ k (\text{mt_pos } c0\ k - m) \neq LE4;$
 $\forall j \geq k_tm\ M. \text{mt_tape } c0\ j (\text{mt_pos } c0\ j) = BLANK4;$
 $\text{ar_stage_bounded } (bl_tm\ M) (k_tm\ M)$
 $(AR_SimAdvance, 0, 0, \text{buf0}, \text{dvec}, \text{posk0});$
 $jj \leq k_tm\ M - 1 \rrbracket$
 $\implies (\exists c\ m. (c0, c) \in R' \wedge m$
 $\wedge m \leq jj * (2 * \text{block_width } (\Gamma_tm\ M))$
 $\wedge \text{mt_state } c = (q, AR_SimAdvance, k_unidx\ jj, 0, \text{buf0}, \text{dvec},$
 $(\lambda k. \text{if } k_idx\ k < jj \text{ then } \text{ar_newpos } (\text{dvec } k) (\text{posk0 } k)$
 $\text{else } \text{posk0 } k))$
 $\wedge \text{mt_tape } c = \text{mt_tape } c0$
 $\wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx\ k < jj$
 $\text{then } (\text{if } \text{dvec } k = \text{dir}.R \text{ then } \text{mt_pos } c0\ k$
 $\text{else } \text{mt_pos } c0\ k$
 $\text{-- ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } k) (\text{posk0 } k))$
 $\text{else } \text{mt_pos } c0\ k))$
and $\text{leaf_finish}:$
 $\wedge cc\ tk'\ pk'.$
 $\llbracket \text{valid_mttm } M; q \in Q_tm\ M; 2 \leq \text{block_width } (\Gamma_tm\ M);$
 $\text{is_last_k } M\ tk';$
 $\text{mt_state } cc = (q, AR_SimAdvance, tk', 0, \text{buf0}, \text{dvec}, \text{pk}');$
 $\forall k. \text{buf0 } k \in \Gamma_tm\ M \cup \{bl_tm\ M\};$
 $\text{dvec } tk' \neq \text{dir}.R$
 $\implies 0 < \text{ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } tk') (\text{pk}'\ tk');$
 $\wedge m. m < \text{ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } tk') (\text{pk}'\ tk')$
 $\implies \text{mt_tape } cc\ tk' (\text{mt_pos } cc\ tk' - m) \neq LE4;$
 $\forall j \geq k_tm\ M. \text{mt_tape } cc\ j (\text{mt_pos } cc\ j) = BLANK4;$
 $\text{ar_stage_bounded } (bl_tm\ M) (k_tm\ M)$
 $(AR_SimAdvance, tk', 0, \text{buf0}, \text{dvec}, \text{pk}') \rrbracket$
 $\implies \exists c'. (cc, c') \in R' \wedge (\text{if } \text{dvec } tk' = \text{dir}.R \text{ then } 1$
 $\text{else } \text{ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } tk') (\text{pk}'\ tk'))$
 $\wedge \text{mt_state } c' = (q, AR_SimNext, k_unidx\ 0, 0, \text{buf0}, \text{dvec},$
 $\text{pk}'(tk' := \text{ar_newpos } (\text{dvec } tk') (\text{pk}'\ tk'))$
 $\wedge \text{mt_tape } c' = \text{mt_tape } cc$
 $\wedge \text{mt_pos } c' = (\text{if } \text{dvec } tk' = \text{dir}.R \text{ then } \text{mt_pos } cc$
 $\text{else } (\text{mt_pos } cc)(tk' := \text{mt_pos } cc\ tk'$
 $\text{-- ar_disp } (\text{block_width } (\Gamma_tm\ M)) (\text{dvec } tk') (\text{pk}'\ tk'))$
and $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm\ M)$
and $stg0: \text{mt_state } c0 = (q, AR_SimAdvance, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$

```

and buf_valid:  $\forall k. \text{buf0 } k \in \Gamma\_tm \ M \cup \{\text{bl\_tm } M\}$ 
and dge1:  $\forall k < k\_tm \ M. \text{dvec } k \neq \text{dir}.R$ 
       $\longrightarrow 0 < \text{ar\_disp } (\text{block\_width } (\Gamma\_tm \ M)) (\text{dvec } k) (\text{posk0 } k)$ 
and notLE:  $\forall k < k\_tm \ M. \forall m. m < \text{ar\_disp } (\text{block\_width } (\Gamma\_tm \ M))$ 
   $(\text{dvec } k) (\text{posk0 } k)$ 
       $\longrightarrow \text{mt\_tape } c0 \ k (\text{mt\_pos } c0 \ k - m) \neq LE4$ 
and pad0:  $\forall j \geq k\_tm \ M. \text{mt\_tape } c0 \ j (\text{mt\_pos } c0 \ j) = \text{BLANK4}$ 
and src0:  $\text{ar\_stage\_bounded } (\text{bl\_tm } M) (k\_tm \ M)$ 
       $(AR\_SimAdvance, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$ 
shows  $\exists c \ m. (c0, c) \in R' \wedge \wedge m$ 
       $\wedge m \leq k\_tm \ M * (2 * \text{block\_width } (\Gamma\_tm \ M))$ 
       $\wedge \text{mt\_state } c = (q, AR\_SimNext, k\_unidx \ 0, 0, \text{buf0}, \text{dvec},$ 
         $(\lambda k. \text{if } k < k\_tm \ M \text{ then } \text{ar\_newpos } (\text{dvec } k) (\text{posk0 } k)$ 
           $\text{else } \text{posk0 } k))$ 
       $\wedge \text{mt\_tape } c = \text{mt\_tape } c0$ 
       $\wedge \text{mt\_pos } c = (\lambda k. \text{if } k < k\_tm \ M$ 
         $\text{then } (\text{if } \text{dvec } k = \text{dir}.R \text{ then } \text{mt\_pos } c0 \ k$ 
           $\text{else } \text{mt\_pos } c0 \ k - \text{ar\_disp } (\text{block\_width } (\Gamma\_tm \ M)) (\text{dvec } k)$ 
             $(\text{posk0 } k))$ 
           $\text{else } \text{mt\_pos } c0 \ k)$ 
proof –
  let ?k =  $\text{block\_width } (\Gamma\_tm \ M)$ 
  let ?N =  $k\_tm \ M$ 
  let ?m1 =  $?N - 1$ 
  let ?tl =  $k\_unidx \ ?m1$ 
  have card1:  $0 < ?N$  using vM by (cases M) auto
  have m1suc:  $?N = \text{Suc } ?m1$  using card1 by simp
  have m1card:  $?m1 < ?N$  using card1 by simp
  have m1lt:  $?m1 < k\_tm \ M$  using card1 by simp
  have kpos:  $0 < ?k$  using kge2 by simp
  have tl_eq:  $?tl = ?m1$  by (simp add: k_unidx_def)
  have tlidx:  $k\_idx \ ?tl = ?m1$  by (rule k_idx_unidx[OF m1card])
  have tl_active:  $?tl < k\_tm \ M$  using m1lt tl_eq by simp
  have islast:  $\text{is\_last\_k } M \ ?tl$  using is_last_k_unidx[OF m1card] m1suc by simp
  have cond:  $(k < ?m1) = (k < k\_tm \ M)$  if  $k \neq ?m1$  for k
    using that m1suc by (auto simp: less_Suc_eq)
  let ?pk =  $\lambda k. \text{if } k\_idx \ k < ?m1 \text{ then } \text{ar\_newpos } (\text{dvec } k) (\text{posk0 } k)$ 
     $\text{else } \text{posk0 } k$ 
  let ?ps =  $\lambda k. \text{if } k\_idx \ k < ?m1$ 
     $\text{then } (\text{if } \text{dvec } k = \text{dir}.R \text{ then } \text{mt\_pos } c0 \ k$ 
       $\text{else } \text{mt\_pos } c0 \ k - \text{ar\_disp } ?k (\text{dvec } k) (\text{posk0 } k))$ 
     $\text{else } \text{mt\_pos } c0 \ k$ 
  have m1le:  $?m1 \leq k\_tm \ M - 1$  by simp
  obtain c1 m1 where
     $c1\_rel: (c0, c1) \in R' \wedge \wedge m1$ 
    and m1_le:  $m1 \leq ?m1 * (2 * ?k)$ 
    and c1_st:  $\text{mt\_state } c1 = (q, AR\_SimAdvance, ?tl, 0, \text{buf0}, \text{dvec}, ?pk)$ 
    and c1_tp:  $\text{mt\_tape } c1 = \text{mt\_tape } c0$ 
    and c1_ps:  $\text{mt\_pos } c1 = ?ps$ 

```

```

    using leaf_prefix[OF vM qQ kge2 stg0 buf_valid dge1 notLE pad0 src0 m1le]
    by blast
  have pk_tl: ?pk ?tl = posk0 ?tl using tlidx by simp
  have c1_pos_tl: mt_pos c1 ?tl = mt_pos c0 ?tl using c1_ps tlidx by simp
  have tl_dge1: dvec ?tl  $\neq$  dir.R  $\implies$  0 < ar_disp ?k (dvec ?tl) (?pk ?tl)
    using dge1[rule_format, OF tl_active] pk_tl by simp
  have tl_notLE:  $\bigwedge m. m < ar\_disp \text{ ?k } (dvec \text{ ?tl}) \text{ (?pk ?tl)}$ 
     $\implies mt\_tape \text{ c1 ?tl } (mt\_pos \text{ c1 ?tl} - m) \neq LE4$ 
  proof -
    fix m assume m < ar_disp ?k (dvec ?tl) (?pk ?tl)
    hence mlt: m < ar_disp ?k (dvec ?tl) (posk0 ?tl) using pk_tl by simp
    have mt_tape_c1 ?tl (mt_pos c1 ?tl - m) = mt_tape c0 ?tl (mt_pos c0 ?tl -
m)
      using c1_tp c1_pos_tl by simp
    thus mt_tape c1 ?tl (mt_pos c1 ?tl - m)  $\neq$  LE4
      using notLE[rule_format, OF tl_active mlt] by simp
  qed
  have tail_idx:  $\bigwedge j. k\_tm \text{ M} \leq j \implies \neg k\_idx \text{ j} < ?m1$ 
    using m1suc unfolding k_idx_def by linarith
  have pad_c1:  $\forall j \geq k\_tm \text{ M}. mt\_tape \text{ c1 j } (mt\_pos \text{ c1 j}) = BLANK4$ 
  proof (intro allI impI)
    fix j assume jge: k_tm M  $\leq$  j
    have mt_tape_c1 j (mt_pos c1 j) = mt_tape c0 j (mt_pos c0 j)
      using c1_tp c1_ps tail_idx[OF jge] by simp
    thus mt_tape c1 j (mt_pos c1 j) = BLANK4 using pad0 jge by simp
  qed
  have src_c1: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimAdvance, ?tl, 0, buf0, dvec, ?pk)
    using src0 tl_active tail_idx by (auto simp: ar_stage_bounded_def)
  obtain c2 where
    step: (c1, c2)  $\in R' \wedge$  (if dvec ?tl = dir.R then 1
      else ar_disp ?k (dvec ?tl) (?pk ?tl))
    and c2_st: mt_state c2 = (q, AR_SimNext, k_unidx 0, 0, buf0, dvec,
      ?pk(?tl := ar_newpos (dvec ?tl) (?pk ?tl)))
    and c2_tp: mt_tape c2 = mt_tape c1
    and c2_ps: mt_pos c2 = (if dvec ?tl = dir.R then mt_pos c1
      else (mt_pos c1)(?tl := mt_pos c1 ?tl
        - ar_disp ?k (dvec ?tl) (?pk ?tl)))
    using leaf_finish[OF vM qQ kge2 islast c1_st buf_valid
      tl_dge1 tl_notLE pad_c1 src_c1]
    by blast
  — The last-tape fun_upds collapse to active-guarded vectors: active tapes take
  the ar_newpos / displaced value, padding tapes stay frozen at the entry posk0 /
  position.
  have pk_full: ?pk(?tl := ar_newpos (dvec ?tl) (?pk ?tl))
    = ( $\lambda k. \text{ if } k < k\_tm \text{ M then ar\_newpos (dvec k) (posk0 k)}$ 
      else posk0 k)
  proof (rule ext)
    fix k

```

```

show (?pk(?tl := ar_newpos (dvec ?tl) (?pk ?tl))) k
      = (if k < k_tm M then ar_newpos (dvec k) (posk0 k) else posk0 k)
proof (cases k = ?tl)
  case True thus ?thesis using pk_tl tl_active by simp
next
  case False
  hence kne: k ≠ ?m1 using tl_eq by simp
  show ?thesis using False kne cond[OF kne] by (simp add: k_idx_def)
qed
qed
have ps_full: (if dvec ?tl = dir.R then mt_pos c1
                else (mt_pos c1)(?tl := mt_pos c1 ?tl
                                - ar_disp ?k (dvec ?tl) (?pk ?tl)))
              = (λk. if k < k_tm M
                    then (if dvec k = dir.R then mt_pos c0 k
                        else mt_pos c0 k - ar_disp ?k (dvec k) (posk0 k))
                    else mt_pos c0 k)
proof (rule ext)
  fix k
  show (if dvec ?tl = dir.R then mt_pos c1
        else (mt_pos c1)(?tl := mt_pos c1 ?tl
                        - ar_disp ?k (dvec ?tl) (?pk ?tl))) k
      = (if k < k_tm M
        then (if dvec k = dir.R then mt_pos c0 k
            else mt_pos c0 k - ar_disp ?k (dvec k) (posk0 k))
        else mt_pos c0 k)
proof (cases k = ?tl)
  case True thus ?thesis using c1_pos_tl pk_tl tl_active by simp
next
  case False
  hence kne: k ≠ ?m1 using tl_eq by simp
  show ?thesis using False kne cond[OF kne] c1_ps by (simp add: k_idx_def)
qed
qed
have chain: (c0, c2) ∈ R' ^ (m1 + (if dvec ?tl = dir.R then 1
                                     else ar_disp ?k (dvec ?tl) (?pk ?tl)))
proof -
  have (c0, c2) ∈ R' ^ m1
    O R' ^ (if dvec ?tl = dir.R then 1
            else ar_disp ?k (dvec ?tl) (?pk ?tl))
    using c1_rel step by (rule relcompI)
  thus ?thesis by (simp add: relpow_add)
qed
have bound: m1 + (if dvec ?tl = dir.R then 1
                    else ar_disp ?k (dvec ?tl) (?pk ?tl)) ≤ ?N * (2 * ?k)
proof -
  obtain Nm where Nm: ?N = Suc Nm using card1 by (cases ?N) auto
  have e1: ?N * (2 * ?k) = (2 * ?k) + Nm * (2 * ?k) by (simp add: Nm)
  have e2: m1 ≤ Nm * (2 * ?k) using m1_le Nm by simp

```

```

have e3: (if dvec ?tl = dir.R then 1 else ar_disp ?k (dvec ?tl) (?pk ?tl))
  ≤ 2 * ?k
  using ar_disp_le_2k[OF kpos, of dvec ?tl ?pk ?tl] kpos
  by (cases dvec ?tl = dir.R) auto
show ?thesis using e1 e2 e3 by linarith
qed
show ?thesis
proof (intro exI[where x = c2]
  exI[where x = m1 + (if dvec ?tl = dir.R then 1
    else ar_disp ?k (dvec ?tl) (?pk ?tl))] conjI)
show (c0, c2) ∈ R' ^^ (m1 + (if dvec ?tl = dir.R then 1
  else ar_disp ?k (dvec ?tl) (?pk ?tl))) by (rule chain)
show m1 + (if dvec ?tl = dir.R then 1
  else ar_disp ?k (dvec ?tl) (?pk ?tl)) ≤ ?N * (2 * ?k)
  by (rule bound)
show mt_state c2 = (q, AR_SimNext, k_unidx 0, 0, buf0, dvec,
  (λk. if k < k_tm M then ar_newpos (dvec k) (posk0 k) else posk0 k))
  using c2_st pk_full by simp
show mt_tape c2 = mt_tape c0 using c2_tp c1_tp by simp
show mt_pos c2 = (λk. if k < k_tm M
  then (if dvec k = dir.R then mt_pos c0 k
    else mt_pos c0 k - ar_disp ?k (dvec k) (posk0 k))
  else mt_pos c0 k)
  using c2_ps ps_full by simp
qed
qed

```

The full advance phase: the prefix walk over the first $k_tm\ M - 1$ tapes followed by the unified last-tape advance *ar_advance_tape_finish_step*, landing at *AR_SimNext* (current-tape field $k_unidx\ 0$) with every tape's head moved to M 's new position (R : unchanged; otherwise $- ar_disp$) and every *posk* updated by *ar_newpos*. The tape is unchanged throughout (advance only moves heads). Aggregate cost $\leq k_tm\ M \cdot 2b$. The final *posk*/position vectors collapse from the $k_idx\ k < ?m1$ split because, on the last tape, every other tape has index $< k_tm\ M - 1$ (k_idx bijective into $\{..< k_tm\ M\}$).

```

lemma ar_advance_phase:
  fixes M :: ('q, 'a) mttm
  and c0 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and qQ: q ∈ Q_tm M
  and kge2: 2 ≤ block_width (Γ_tm M)
  and stg0: mt_state c0 = (q, AR_SimAdvance, 0, 0, buf0, dvec, posk0)
  and buf_valid: ∀ k. buf0 k ∈ Γ_tm M ∪ {bl_tm M}
  and dge1: ∀ k < k_tm M. dvec k ≠ dir.R
    → 0 < ar_disp (block_width (Γ_tm M)) (dvec k) (posk0 k)
  and notLE: ∀ k < k_tm M. ∀ m. m < ar_disp (block_width (Γ_tm M))
    (dvec k) (posk0 k)

```



```

       $\longrightarrow$   $mt\_tape\ c0\ k\ (mt\_pos\ c0\ k - m) \neq LE4$ 
and  $pad0$ :  $\forall j \geq k\_tm\ M. mt\_tape\ c0\ j\ (mt\_pos\ c0\ j) = BLANK4$ 
and  $src0$ :  $ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
      ( $AR\_SimAdvance, 0, 0, buf0, dvec, posk0$ )
shows  $\exists c\ m. (c0, c) \in (mttm\_step\ (alphabet\_reduce\_delta\ M)) \wedge m$ 
       $\wedge m \leq k\_tm\ M * (2 * block\_width\ (\Gamma\_tm\ M))$ 
       $\wedge mt\_state\ c = (q, AR\_SimNext, k\_unidx\ 0, 0, buf0, dvec,$ 
       $(\lambda k. \text{if } k < k\_tm\ M \text{ then } ar\_newpos\ (dvec\ k)\ (posk0\ k)$ 
       $\text{else } posk0\ k))$ 
       $\wedge mt\_tape\ c = mt\_tape\ c0$ 
       $\wedge mt\_pos\ c = (\lambda k. \text{if } k < k\_tm\ M$ 
       $\text{then } (\text{if } dvec\ k = dir.R \text{ then } mt\_pos\ c0\ k$ 
       $\text{else } mt\_pos\ c0\ k - ar\_disp\ (block\_width\ (\Gamma\_tm\ M))\ (dvec\ k)$ 
       $(posk0\ k))$ 
       $\text{else } mt\_pos\ c0\ k)$ 
by ( $rule\ ar\_advance\_phase\_gen$ 
      [ $OF\ ar\_advance\_prefix\ ar\_advance\_tape\_finish\_step$ 
       $vM\ qQ\ kge2\ stg0\ buf\_valid\ dge1\ notLE\ pad0\ src0$ ])
end
theory AlphabetReduction_Forward
imports AlphabetReduction_ForwardAdvance
begin

```

2.16 Per- M -step simulation phase and the chunked engine

Acceptance read-out, the AR analogue of AE's *ae_simulates_accept_iff*. Under *ar_simulates*, the source M -config is in the accept state iff the paired M' -config sits in the canonical accept config ($t_tm\ M, ar_accept_stage\ bl_M$). Forward: $q_M = t$ forces the second *ar_simulates* disjunct (the first needs $q_M' \notin \{t, r\}$; the third needs $t = r$, ruled out by *valid_mttm_t_neq_r*). Reverse is immediate from the state-equality conjunct. Throughout, b abbreviates *block_width* Γ (the per-symbol cell width).

```

lemma ar_simulates_accept_iff:
  fixes  $M :: ('q, 'a)\ mttm$ 
    and  $cM :: ('a, 'q)\ mt\_config$ 
    and  $c' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config$ 
assumes  $vM$ : valid_mttm  $M$ 
    and  $sim$ : ar_simulates  $M\ cM\ c'$ 
shows ( $mt\_state\ cM = t\_tm\ M$ )
       $\longleftrightarrow (mt\_state\ c' = (t\_tm\ M, ar\_accept\_stage\ (bl\_tm\ M)))$ 
proof -
  obtain  $qM'\ stg$  where  $st$ :  $mt\_state\ c' = (qM', stg)$ 
    by (cases  $mt\_state\ c'$ ) auto
  obtain  $idx\ tk\ i\ buf\ dvec\ posk$  where
     $sg$ :  $stg = (idx, tk, i, buf, dvec, posk)$ 
    by (cases  $stg$ ) auto
  have  $sim\_body$ :

```

```

    ((idx = AR_SimRead ∧ qM' ∉ {t_tm M, r_tm M})
      ∨ (qM' = t_tm M
        ∧ (idx, tk, i, buf, dvec, posk) = ar_accept_stage (bl_tm M))
      ∨ (qM' = r_tm M
        ∧ (idx, tk, i, buf, dvec, posk) = ar_reject_stage (bl_tm M)))
    ∧ mt_state cM = qM'
    ∧ (∀ k < k_tm M. ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm
M)
      (mt_tape cM k) (mt_tape c' k))
    ∧ (idx = AR_SimRead
      → (∀ k. mt_pos c' k = sim_pos (block_width (Γ_tm M)) (mt_pos cM
k)))
  using sim_unfolding ar_simulates_def by (simp add: st_sg Let_def)
  have disj:
    (idx = AR_SimRead ∧ qM' ∉ {t_tm M, r_tm M})
      ∨ (qM' = t_tm M
        ∧ (idx, tk, i, buf, dvec, posk) = ar_accept_stage (bl_tm M))
      ∨ (qM' = r_tm M
        ∧ (idx, tk, i, buf, dvec, posk) = ar_reject_stage (bl_tm M))
  using sim_body by simp
  have qM_eq: mt_state cM = qM' using sim_body by simp
  show ?thesis
  proof
    assume mt_state cM = t_tm M
    hence qt: qM' = t_tm M using qM_eq by simp
    have (idx, tk, i, buf, dvec, posk) = ar_accept_stage (bl_tm M)
      using disj qt valid_mttm_t_neq_r[OF vM] by auto
    thus mt_state c' = (t_tm M, ar_accept_stage (bl_tm M))
      using st_sg qt by simp
  next
    assume mt_state c' = (t_tm M, ar_accept_stage (bl_tm M))
    hence qM' = t_tm M using st by simp
    thus mt_state cM = t_tm M using qM_eq by simp
  qed
qed

```

The next substep, non-terminal hand-off: a single M' -step from an $AR_SimNext$ stage whose M -state q is neither accepting nor rejecting routes to the next M -step's $AR_SimRead$ boundary, resetting both the current-tape and bit-counter fields to 0 while preserving buf , $dvec$, and the per-tape $posk$. No tape cell changes and no head moves (all directions N , the write vector is the read vector). The cleanest of the five phases — the $ar_compute_step$ shape with arm 1 of the three-arm ar_delta_next union; the two δLE filters discharge reflexively (write equals read, move N) and target-stage validity rests only on $0 < b$. The terminal arms ($q = t_tm M$ / $q = r_tm M$) route to the halt stages instead and are handled at assembly, not here.

lemma ar_next_step :

```

fixes  $M :: ('q, 'a) \text{mttm}$ 
and  $c' :: (\text{sym}_4, 'q \times 'a \text{ar\_stage}) \text{mt\_config}$ 
assumes  $vM: \text{valid\_mttm } M$ 
and  $\text{stg}: \text{mt\_state } c' = (q, \text{AR\_SimNext}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
and  $qQ: q \in Q\_tm \ M$ 
and  $\text{notT}: q \neq t\_tm \ M$ 
and  $\text{notR}: q \neq r\_tm \ M$ 
and  $\text{vsrc}: \text{ar\_valid\_stage } (\Gamma\_tm \ M) \ (\text{bl\_tm } M)$ 
 $(\text{AR\_SimNext}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
and  $\text{pad\_blank}: \forall j \geq k\_tm \ M. \text{mt\_tape } c' \ j \ (\text{mt\_pos } c' \ j) = \text{BLANK}_4$ 
and  $\text{src\_bounded}: \text{ar\_stage\_bounded } (\text{bl\_tm } M) \ (k\_tm \ M)$ 
 $(\text{AR\_SimNext}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
shows  $\exists c''. (c', c'') \in \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 
 $\wedge \text{mt\_state } c'' = (q, \text{AR\_SimRead}, 0, 0, \text{buf}, \text{dvec}, \text{posk})$ 
 $\wedge \text{mt\_tape } c'' = \text{mt\_tape } c'$ 
 $\wedge \text{mt\_pos } c'' = \text{mt\_pos } c'$ 

proof –
obtain  $ts \ n$  where  $c' \text{\_eq}$ :
 $c' = \text{Config}_M \ (q, \text{AR\_SimNext}, tk, i, \text{buf}, \text{dvec}, \text{posk}) \ ts \ n$ 
using  $\text{stg}$  by  $(\text{cases } c') \ \text{auto}$ 
let  $?a = \lambda k. ts \ k \ (n \ k)$ 
let  $?s = (q, \text{AR\_SimNext}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
let  $?s' = (q, \text{AR\_SimRead}, 0, 0, \text{buf}, \text{dvec}, \text{posk})$ 
have  $\text{rel\_in}: (?s, ?a, ?s', ?a, (\lambda_. \text{dir}.N)) \in \text{ar\_delta\_next } M$ 
unfolding  $\text{ar\_delta\_next\_def}$  using  $qQ \ \text{notT} \ \text{notR}$  by  $(\text{intro } \text{UnI1}) \ \text{blast}$ 
have  $\text{buf\_valid}: \forall k. \text{buf } k \in \Gamma\_tm \ M \cup \{\text{bl\_tm } M\}$ 
using  $\text{vsrc}$  by  $(\text{simp add: ar\_valid\_stage\_def})$ 
have  $kpos: 0 < \text{block\_width } (\Gamma\_tm \ M)$  using  $\text{block\_width\_pos[of } \Gamma\_tm \ M]$  by
 $\text{simp}$ 
have  $\text{dst\_valid}: \text{ar\_valid\_stage } (\Gamma\_tm \ M) \ (\text{bl\_tm } M) \ (\text{snd } ?s')$ 
using  $kpos \ \text{buf\_valid}$  by  $(\text{simp add: ar\_valid\_stage\_def})$ 
have  $\text{pad\_a}: \forall j \geq k\_tm \ M. ?a \ j = \text{BLANK}_4$  using  $\text{pad\_blank } c' \text{\_eq}$  by  $\text{simp}$ 
have  $\text{pad\_read}: \forall j \geq k\_tm \ M. ?a \ j = \text{BLANK}_4 \wedge ?a \ j = \text{BLANK}_4$ 
 $\wedge (\lambda_. \text{dir}.N) \ j = \text{dir}.N$ 
using  $\text{pad\_a}$  by  $\text{simp}$ 
have  $kpos\_tm: 0 < k\_tm \ M$  using  $vM$  by  $(\text{cases } M) \ \text{auto}$ 
have  $\text{dst\_bd}: \text{ar\_stage\_bounded } (\text{bl\_tm } M) \ (k\_tm \ M) \ (\text{snd } ?s')$ 
using  $\text{src\_bounded } kpos\_tm$  by  $(\text{simp add: ar\_stage\_bounded\_def})$ 
have  $\text{ard\_in}: (?s, ?a, ?s', ?a, (\lambda_. \text{dir}.N)) \in \text{alphabet\_reduce\_delta } M$ 
unfolding  $\text{alphabet\_reduce\_delta\_def}$ 
using  $\text{rel\_in } \text{vsrc } \text{dst\_valid } \text{pad\_read } \text{src\_bounded } \text{dst\_bd}$  by  $\text{auto}$ 
have  $\text{ts\_unchanged}: (\lambda k. (ts \ k)(n \ k := ?a \ k)) = ts$  by  $(\text{rule ext}) \ \text{auto}$ 
have  $\text{pos\_unchanged}: (\lambda k. \text{go\_dir } ((\lambda_. \text{dir}.N) \ k) \ (n \ k)) = n$ 
by  $(\text{rule ext}) \ \text{auto}$ 
let  $?c'' = \text{Config}_M \ ?s' \ ts \ n$ 
have  $(\text{Config}_M \ ?s \ ts \ n,$ 
 $\text{Config}_M \ ?s' \ (\lambda k. (ts \ k)(n \ k := ?a \ k))$ 
 $(\lambda k. \text{go\_dir } ((\lambda_. \text{dir}.N) \ k) \ (n \ k)))$ 
 $\in \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 

```

```

proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, (λ_. dir.N))
    ∈ alphabet_reduce_delta M
  by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_unchanged by simp
show ?thesis
  using step by (intro exI[where x = ?c'']) (simp add: c'_eq)
qed

```

The two terminal hand-offs, the accept and reject arms of ar_delta_next : a single M' -step from an $AR_SimNext$ stage whose M -state q is the accepting (resp. rejecting) state lands on the canonical halt stage ar_accept_stage ($bl_tm\ M$) (resp. ar_reject_stage), so the full target state equals t_tm ($alphabet_reduce\ M$) (resp. r_tm): full-state acceptance forces the canonicalisation. Tape and heads are unchanged (the move vector is all- N , no writes). Modelled on ar_next_step ; the membership lets *blast* select the halt comprehension and instantiate its witnesses.

```

lemma ar_accept_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimNext, tk, i, buf, dvec, posk)
  and qT: q = t_tm M
  and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
    (AR_SimNext, tk, i, buf, dvec, posk)
  and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimNext, tk, i, buf, dvec, posk)
  shows ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    ∧ mt_state c'' = (t_tm M, ar_accept_stage (bl_tm M))
    ∧ mt_tape c'' = mt_tape c'
    ∧ mt_pos c'' = mt_pos c'

```

proof –

```

obtain ts n where c'_eq:
  c' = Config_M (q, AR_SimNext, tk, i, buf, dvec, posk) ts n
  using stg by (cases c') auto
let ?a = λk. ts k (n k)
let ?s = (q, AR_SimNext, tk, i, buf, dvec, posk)
let ?s' = (q, AR_HaltAccept, 0, 0,
  (λ_. bl_tm M), (λ_. dir.N), (λ_. AR_AtLE))
have rel_in: (?s, ?a, ?s', ?a, (λ_. dir.N)) ∈ ar_delta_next M
  unfolding ar_delta_next_def using qT by blast
have kpos: 0 < block_width (Γ_tm M) using block_width_pos[of Γ_tm M] by
simp
have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
  using kpos by (simp add: ar_valid_stage_def)
have pad_a: ∀ j ≥ k_tm M. ?a j = BLANK4 using pad_blank c'_eq by simp

```

```

have pad_read:  $\forall j \geq k\_tm\ M. ?a\ j = BLANK_4 \wedge ?a\ j = BLANK_4$ 
                $\wedge (\lambda\_ . dir.N)\ j = dir.N$ 
  using pad_a by simp
have kpos_tm:  $0 < k\_tm\ M$  using vM by (cases M) auto
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  using kpos_tm by (simp add: ar_stage_bounded_def)
have ard_in:  $(?s, ?a, ?s', ?a, (\lambda\_ . dir.N)) \in alphabet\_reduce\_delta\ M$ 
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd by auto
have ts_unchanged:  $(\lambda k. (ts\ k)(n\ k := ?a\ k)) = ts$  by (rule ext) auto
have pos_unchanged:  $(\lambda k. go\_dir\ ((\lambda\_ . dir.N)\ k)\ (n\ k)) = n$ 
  by (rule ext) auto
let ?c'' = Config_M ?s ts n
have (Config_M ?s ts n,
      Config_M ?s'  $(\lambda k. (ts\ k)(n\ k := ?a\ k))$ 
       $(\lambda k. go\_dir\ ((\lambda\_ . dir.N)\ k)\ (n\ k))$ )
   $\in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
proof (rule mttm_step.intros)
  show  $(?s, ?a, ?s', ?a, (\lambda\_ . dir.N))$ 
     $\in alphabet\_reduce\_delta\ M$ 
  by (rule ard_in)
qed
hence step:  $(c', ?c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
  using c'_eq ts_unchanged pos_unchanged by simp
show ?thesis
  using step by (intro exI[where x = ?c''])
  (simp add: c'_eq qT ar_accept_stage_def)
qed

```

```

lemma ar_reject_step:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q  $\times$  'a ar_stage) mt_config
  assumes vM: valid_mttm M
  and stg: mt_state c' = (q, AR_SimNext, tk, i, buf, dvec, posk)
  and qR: q = r_tm M
  and vsrc: ar_valid_stage ( $\Gamma\_tm\ M$ ) (bl_tm M)
             (AR_SimNext, tk, i, buf, dvec, posk)
  and pad_blank:  $\forall j \geq k\_tm\ M. mt\_tape\ c'\ j\ (mt\_pos\ c'\ j) = BLANK_4$ 
  and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)
             (AR_SimNext, tk, i, buf, dvec, posk)
  shows  $\exists c''. (c', c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
            $\wedge mt\_state\ c'' = (r\_tm\ M, ar\_reject\_stage\ (bl\_tm\ M))$ 
            $\wedge mt\_tape\ c'' = mt\_tape\ c'$ 
            $\wedge mt\_pos\ c'' = mt\_pos\ c'$ 
proof -
  obtain ts n where c'_eq:
    c' = Config_M (q, AR_SimNext, tk, i, buf, dvec, posk) ts n
  using stg by (cases c') auto
  let ?a =  $\lambda k. ts\ k\ (n\ k)$ 

```

```

let ?s = (q, AR_SimNext, tk, i, buf, dvec, posk)
let ?s' = (q, AR_HaltReject, 0, 0,
          (λ_. bl_tm M), (λ_. dir.N), (λ_. AR_AtLE))
have rel_in: (?s, ?a, ?s', ?a, (λ_. dir.N)) ∈ ar_delta_next M
  unfolding ar_delta_next_def using qR by blast
have kpos: 0 < block_width (Γ_tm M) using block_width_pos[of Γ_tm M] by
simp
have dst_valid: ar_valid_stage (Γ_tm M) (bl_tm M) (snd ?s')
  using kpos by (simp add: ar_valid_stage_def)
have pad_a: ∀j ≥ k_tm M. ?a j = BLANK₄ using pad_blank c'_eq by simp
have pad_read: ∀j ≥ k_tm M. ?a j = BLANK₄ ∧ ?a j = BLANK₄
  ∧ (λ_. dir.N) j = dir.N
  using pad_a by simp
have kpos_tm: 0 < k_tm M using vM by (cases M) auto
have dst_bd: ar_stage_bounded (bl_tm M) (k_tm M) (snd ?s')
  using kpos_tm by (simp add: ar_stage_bounded_def)
have ard_in: (?s, ?a, ?s', ?a, (λ_. dir.N)) ∈ alphabet_reduce_delta M
  unfolding alphabet_reduce_delta_def
  using rel_in vsrc dst_valid pad_read src_bounded dst_bd by auto
have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts by (rule ext) auto
have pos_unchanged: (λk. go_dir ((λ_. dir.N) k) (n k)) = n
  by (rule ext) auto
let ?c'' = Config_M ?s' ts n
have (Config_M ?s ts n,
      Config_M ?s' (λk. (ts k)(n k := ?a k))
      (λk. go_dir ((λ_. dir.N) k) (n k)))
  ∈ mttm_step (alphabet_reduce_delta M)
proof (rule mttm_step.intros)
  show (?s, ?a, ?s', ?a, (λ_. dir.N))
    ∈ alphabet_reduce_delta M
  by (rule ard_in)
qed
hence step: (c', ?c'') ∈ mttm_step (alphabet_reduce_delta M)
  using c'_eq ts_unchanged pos_unchanged by simp
show ?thesis
  using step by (intro exI[where x = ?c''])
  (simp add: c'_eq qR ar_reject_stage_def)
qed

```

Two position-bookkeeping facts for the advance phase, decided once in a clean context (no *let*-bound *?*-vars to tangle the case split). Given *M*'s head at position *p* moving *dr* (with *dr* ≠ *L* when *p* = 0, the *δLE* discipline), the read-phase position-kind *if p=0 then AtLE else if p=1 then FirstProper else FurtherProper* advances via *ar_newpos* to *AtLE* exactly when *M*'s new position is 0, and the read-phase landing position less the *ar_disp* displacement is exactly *sim_pos* of *M*'s new position. Both reduce to a *p* ∈ {0, 1, ≥ 2} cross *dr* ∈ {N, R, L} grid.

lemma *ar_newpos_atLE_iff*:
 fixes *p* :: nat and *dr* :: dir

```

assumes  $nL0: p = 0 \implies dr \neq dir.L$ 
shows  $(ar\_newpos\ dr\ (if\ p = 0\ then\ AR\_AtLE$ 
       $\quad\quad\quad else\ if\ p = 1\ then\ AR\_AtFirstProper$ 
       $\quad\quad\quad else\ AR\_AtFurtherProper) = AR\_AtLE)$ 
       $\longleftrightarrow go\_dir\ dr\ p = 0$ 
proof (cases  $p$ )
  case 0
    hence  $dr = dir.N \vee dr = dir.R$  using  $nL0$  by (cases  $dr$ ) auto
    thus ?thesis using 0 by auto
  next
    case (Suc  $nat$ )
    show ?thesis
    proof (cases  $nat$ )
      case 0
        thus ?thesis using Suc by (cases  $dr$ ) auto
      next
        case (Suc  $m$ )
        thus ?thesis using  $\langle p = Suc\ nat \rangle$  by (cases  $dr$ ) auto
    qed
  qed

lemma  $ar\_advance\_newsimps:$ 
  fixes  $K\ p :: nat$  and  $dr :: dir$ 
  assumes  $Kge2: 2 \leq K$ 
  and  $nL0: p = 0 \implies dr \neq dir.L$ 
  shows  $(if\ dr = dir.R$ 
     $\quad\quad\quad then\ (if\ p = 0\ then\ Suc\ 0\ else\ sim\_pos\ K\ p + K)$ 
     $\quad\quad\quad else\ (if\ p = 0\ then\ Suc\ 0\ else\ sim\_pos\ K\ p + K)$ 
     $\quad\quad\quad -\ ar\_disp\ K\ dr\ (if\ p = 0\ then\ AR\_AtLE$ 
     $\quad\quad\quad\quad\quad\quad else\ if\ p = 1\ then\ AR\_AtFirstProper$ 
     $\quad\quad\quad\quad\quad\quad else\ AR\_AtFurtherProper))$ 
     $= sim\_pos\ K\ (go\_dir\ dr\ p)$ 
proof (cases  $p$ )
  case 0
    hence  $dr = dir.N \vee dr = dir.R$  using  $nL0$  by (cases  $dr$ ) auto
    thus ?thesis using 0 by (auto simp:  $sim\_pos\_def$ )
  next
    case (Suc  $nat$ )
    show ?thesis
    proof (cases  $nat$ )
      case 0
        thus ?thesis using Suc  $Kge2$  by (cases  $dr$ ) (auto simp:  $sim\_pos\_def$ )
      next
        case (Suc  $m$ )
        thus ?thesis using  $\langle p = Suc\ nat \rangle\ Kge2$ 
        by (cases  $dr$ ) (auto simp:  $sim\_pos\_def\ algebra\_simps$ )
    qed
  qed

```

Per- M -step forward simulation: one M -step $cM \rightarrow cM_1$ from an

$AR_SimRead$ boundary is matched by a bounded M' -phase (read $\rightarrow$ compute $\rightarrow$ write $\rightarrow$ advance $\rightarrow$ next) of at most $k_tm \cdot M \cdot (5b + 2) + 2$ M' -steps (the per-phase $k_tm \cdot M$ -scaled sum: read $\leq k_tm \cdot M \cdot (b+2)$, write and advance each $\leq k_tm \cdot M \cdot 2b$, compute and next one step apiece) that re-establishes $ar_simulates$ at the next boundary. This is the composition of the per-substep phase lemmas above into one M -step; the chunked engine below iterates it. The reachability hypothesis $reach_M$ with w_sub supplies the substrate LE -only-at-position-0 fact the read look-back needs, and $card_ge$ gives $b \geq 2$ for the proper-region read arms.

The post-write tape correspondence, factored out of $ar_simulates_forward_step$ so the reverse cycle-close reuses it. Given the read-boundary correspondence between M 's tape and the output tape rt , and that cM_1 is M after writing a' under each head, the write phase's per-tape overwrite wt — which stamps a' 's cell block at the head's block and leaves the rest of rt untouched — is again a correspondence, now for cM_1 . Both arms supply their own wt and discharge wt_def from their write-phase output contract.

lemma $ar_write_tape_correspondence$:
fixes $M :: ('q, 'a) mttm$
and $cM\ cM_1 :: ('a, 'q) mt_config$
and $rt\ wt :: nat \Rightarrow nat \Rightarrow sym4$
and $a' :: nat \Rightarrow 'a$
assumes $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $kN: k < k_tm\ M$
and $tcorr: \forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$
 $(mt_tape\ cM\ k)\ (rt\ k)$
and $cM1tape: \bigwedge k. mt_tape\ cM_1\ k = (mt_tape\ cM\ k)(mt_pos\ cM\ k := a'$
 $k)$
and $a'le0: \forall k < k_tm\ M. mt_pos\ cM\ k = 0 \longrightarrow a'\ k = le_tm\ M$
and $wt_def: \bigwedge k\ pos. wt\ k\ pos$
 $= (if\ mt_pos\ cM\ k = 0\ then\ rt\ k\ pos$
 $else\ if\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k) \leq pos$
 $\wedge\ pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $+ block_width\ (\Gamma_tm\ M)$
 $then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (a'\ k)$
 $(pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k))$
 $else\ rt\ k\ pos)$
shows $ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$
 $(mt_tape\ cM_1\ k)\ (wt\ k)$
proof –
let $?k = block_width\ (\Gamma_tm\ M)$
have $k1: 1 \leq ?k$ **using** $kge2$ **by** $simp$
have $tcorr_k: ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$
 $(mt_tape\ cM\ k)\ (rt\ k)$
using $tcorr\ kN$ **by** $blast$


```

have a'le0_k: mt_pos cM k = 0 → a' k = le_tm M
  using a'le0 kN by blast
have cM0: mt_tape cM k 0 = le_tm M
  using tcorr_k unfolding ar_tape_correspondence_def by blast
have rt0: rt k 0 = LE4
  using tcorr_k unfolding ar_tape_correspondence_def by blast
have rtprop: rt k (sim_pos ?k p + j)
  = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM k p) ! j
  if 1 ≤ p and j < ?k for p j
  using tcorr_k that unfolding ar_tape_correspondence_def by blast
— conjunct (a): M's updated tape still reads le at 0
have a: mt_tape cM_1 k 0 = le_tm M
proof (cases mt_pos cM k = 0)
  case True
  hence a' k = le_tm M using a'le0_k by simp
  thus ?thesis using cM1tape True by simp
next
  case False
  thus ?thesis using cM1tape cM0 by simp
qed
— conjunct (b): wt carries LE4 at 0
have b: wt k 0 = LE4
proof (cases mt_pos cM k = 0)
  case True
  thus ?thesis using rt0 by (simp add: wt_def)
next
  case False
  thus ?thesis using rt0 by (auto simp: wt_def sim_pos_def)
qed
— conjunct (c): every proper block of wt encodes the matching updated M-cell
have c: wt k (sim_pos ?k p + j)
  = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM_1 k p) ! j
  if p1: 1 ≤ p and jk: j < ?k for p j
proof (cases mt_pos cM k = 0)
  case True
  have pne: p ≠ mt_pos cM k using True p1 by simp
  have cM1p: mt_tape cM_1 k p = mt_tape cM k p using cM1tape pne by
simp
  have wt k (sim_pos ?k p + j) = rt k (sim_pos ?k p + j)
    using True by (simp add: wt_def)
  also have ... = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM k p) ! j
    using rtprop[OF p1 jk] by simp
  also have ... = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM_1 k p) ! j
    using cM1p by simp
  finally show ?thesis .
next
  case False
  have pw1: 1 ≤ mt_pos cM k using False by simp
  show ?thesis

```

```

proof (cases p = mt_pos cM k)
  case True
    have inblk: sim_pos ?k (mt_pos cM k) ≤ sim_pos ?k p + j
      ∧ sim_pos ?k p + j < sim_pos ?k (mt_pos cM k) + ?k
      using True jk by simp
    have off: sim_pos ?k p + j - sim_pos ?k (mt_pos cM k) = j
      using True by simp
    have wt k (sim_pos ?k p + j) = write_bit (Γ_tm M) (bl_tm M) (a' k) j
      using False inblk off by (simp add: wt_def)
    also have ... = cell_repr (Γ_tm M) (bl_tm M) (a' k) ! j
      using cell_repr_nth_write_bit[OF jk] by simp
    also have ... = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM_1 k p) ! j
      using cM1tape True by simp
    finally show ?thesis .
  next
    case False
      have notblk: ¬ (sim_pos ?k (mt_pos cM k) ≤ sim_pos ?k p + j
        ∧ sim_pos ?k p + j < sim_pos ?k (mt_pos cM k) + ?k)
        using sim_pos_in_block_iff[OF k1 p1 pw1 jk] False by simp
      have pne: p ≠ mt_pos cM k using False by simp
      have cM1p: mt_tape cM_1 k p = mt_tape cM k p using cM1tape pne by
simp
      have wt k (sim_pos ?k p + j) = rt k (sim_pos ?k p + j)
        using <mt_pos cM k ≠ 0> by (simp add: wt_def if_not_P[OF notblk])
      also have ... = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM k p) ! j
        using rtprop[OF p1 jk] by simp
      also have ... = cell_repr (Γ_tm M) (bl_tm M) (mt_tape cM_1 k p) ! j
        using cM1p by simp
      finally show ?thesis .
    qed
  qed
show ?thesis
  unfolding ar_tape_correspondence_def using a b c by blast
qed

lemma ar_simulates_forward_step:
  fixes M :: ('q, 'a) mttm
  and w :: 'a list
  and cM cM_1 :: ('a, 'q) mt_config
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
  and w_sub: set w ⊆ Sigma_tm M
  and card_ge: card (Γ_tm M) ≥ 4
  and reach_M: (init_config_mttm M w, cM) ∈ (mttm_step (delta_tm M))*
  and step: (cM, cM_1) ∈ mttm_step (delta_tm M)
  and sim: ar_simulates M cM c'
  and posk_ok: ar_posk_consistent M cM c'
  and rbnd: ar_at_read_boundary M c'
  and lebl: le_tm M ≠ bl_tm M

```

obtains $m\ c''$ **where**
 $m \leq k_tm\ M * (5 * block_width\ (\Gamma_tm\ M) + 2) + 2$
and $(c', c'') \in mttm_step\ (alphabet_reduce_delta\ M) \wedge m$
and $ar_simulates\ M\ cM_1\ c''$
and $ar_posk_consistent\ M\ cM_1\ c''$
and $ar_at_read_boundary\ M\ c''$
proof –
let $?k = block_width\ (\Gamma_tm\ M)$
let $?N = k_tm\ M$
– encoding length ≥ 2 from $card\ \Gamma_M \geq 4$
have $kge2: 2 \leq ?k$
proof –
have $(2::nat) \wedge 2 \leq 2 \wedge ?k$
using $card_ge\ card_le_two_pow_block_width[of\ \Gamma_tm\ M]$ **by** *simp*
thus $2 \leq ?k$ **using** $power_le_imp_le_exp[of\ 2::nat\ 2\ ?k]$ **by** *simp*
qed
have $kpos_tm: 0 < k_tm\ M$ **using** vM **by** $(cases\ M)\ auto$
– extract M 's firing δ -tuple and the successor config from the one M -step
from *step* **obtain** $qC\ tsM\ nM\ q'\ a'\ d$ **where**
 $cM_eq: cM = Config_M\ qC\ tsM\ nM$
and $cM1_eq: cM_1 = Config_M\ q'\ (\lambda k. (tsM\ k)(nM\ k := a'\ k))$
 $(\lambda k. go_dir\ (d\ k)\ (nM\ k))$
and $mdelta: (qC, \lambda k. tsM\ k\ (nM\ k), q', a', d) \in delta_tm\ M$
by $(auto\ elim: mttm_step.cases)$
have $qC_eq: qC = mt_state\ cM$ **using** cM_eq **by** *simp*
have $q'_eq: q' = mt_state\ cM_1$ **using** $cM1_eq$ **by** *simp*
– the source M -state is a genuine, non-halting state
have $qQ: mt_state\ cM \in Q_tm\ M$ **using** $mttm_step_src(1)[OF\ vM\ step]$.
have $qnt: mt_state\ cM \neq t_tm\ M\ mt_state\ cM \neq r_tm\ M$
using $mttm_step_src(2)[OF\ vM\ step]\ mttm_step_src(3)[OF\ vM\ step]$ **by** *blast+*
– a reachable M -config is valid: every head reads a Γ symbol, and beyond the
tape count it reads the blank.
have $valcM: valid_config_mttm\ M\ cM$
using $valid_reach_mttm[OF\ vM\ w_sub\ reach_M]$.
have $tapeG: mt_tape\ cM\ k\ (mt_pos\ cM\ k) \in \Gamma_tm\ M$ **for** k
proof –
have $range\ (tsM\ k) \subseteq \Gamma_tm\ M$ **using** $valcM\ cM_eq$ **by** $(cases\ M)\ auto$
thus $mt_tape\ cM\ k\ (mt_pos\ cM\ k) \in \Gamma_tm\ M$ **using** cM_eq **by** *auto*
qed
have $aM_tail: \forall j \geq k_tm\ M. mt_tape\ cM\ j\ (mt_pos\ cM\ j) = bl_tm\ M$
using $valid_config_mttm_blank_tail[OF\ valcM]$ **by** *blast*
– destructure the simulation invariants at the $AR_SimRead$ boundary; the dis-
junction collapses to the $AR_SimRead$ arm because the source state is non-halting.
obtain $qM'\ stg$ **where** $st: mt_state\ c' = (qM', stg)$
by $(cases\ mt_state\ c')\ auto$
obtain $idx\ tk\ ii\ buf0\ dvec\ posk0$ **where**
 $sg: stg = (idx, tk, ii, buf0, dvec, posk0)$
by $(cases\ stg)\ auto$

```

have sim_body:
  ((idx = AR_SimRead  $\wedge$   $qM' \notin \{t\_tm\ M, r\_tm\ M\}$ )
     $\vee$  ( $qM' = t\_tm\ M$ 
       $\wedge$  (idx, tk, ii, buf0, dvec, posk0) = ar_accept_stage (bl_tm M))
     $\vee$  ( $qM' = r\_tm\ M$ 
       $\wedge$  (idx, tk, ii, buf0, dvec, posk0) = ar_reject_stage (bl_tm M)))
   $\wedge$  mt_state cM =  $qM'$ 
   $\wedge$  ( $\forall k < k\_tm\ M. ar\_tape\_correspondence\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (bl\_tm\ M)$ )
  (mt_tape cM k) (mt_tape c' k)
   $\wedge$  (idx = AR_SimRead
     $\rightarrow$  ( $\forall k. mt\_pos\ c'\ k = sim\_pos\ ?k\ (mt\_pos\ cM\ k)$ ))
  using sim_unfolding ar_simulates_def by (simp add: st_sg Let_def)
have qM'_eq:  $qM' = mt\_state\ cM$  using sim_body by simp
have idx_eq: idx = AR_SimRead
  using sim_body qnt qM'_eq
  by (auto simp: ar_accept_stage_def ar_reject_stage_def)
have tcorr:  $\forall k < k\_tm\ M. ar\_tape\_correspondence\ (\Gamma\_tm\ M)\ (le\_tm\ M)$ 
(bl_tm M)
  (mt_tape cM k) (mt_tape c' k)
  using sim_body by simp
have ppos0:  $mt\_pos\ c'\ k = sim\_pos\ ?k\ (mt\_pos\ cM\ k)$  for k
  using sim_body idx_eq by simp
  — The strengthened read boundary: canonical entry shape, the bounded-stage
  fact src_c', and the config padding-blank pad_c'.
have rbnd_body:
  idx = AR_SimRead
   $\rightarrow$  ( $tk = 0 \wedge ii = 0 \wedge (\forall k. buf0\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\})$ 
     $\wedge$  ar_stage_bounded (bl_tm M) (k_tm M)
    (idx, tk, ii, buf0, dvec, posk0))
  using rbnd st_sg unfolding ar_at_read_boundary_def
  by (auto split: prod.splits)
have tk0:  $tk = 0$  using rbnd_body idx_eq by simp
have ii0:  $ii = 0$  using rbnd_body idx_eq by simp
have bufG:  $buf0\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$  for k
  using rbnd_body idx_eq by simp
have src_c': ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, 0, 0, buf0, dvec, posk0)
  using rbnd_body idx_eq tk0 ii0 by simp
have pad_c':  $\forall j \geq k\_tm\ M. mt\_tape\ c'\ j\ (mt\_pos\ c'\ j) = BLANK_4$ 
  using rbnd unfolding ar_at_read_boundary_def by (auto split: prod.splits)
have posk_body:
  idx = AR_SimRead
   $\rightarrow$  ( $\forall k. posk0\ k = AR\_AtLE \longleftrightarrow mt\_pos\ cM\ k = 0$ )
  using posk_ok unfolding ar_posk_consistent_def by (simp add: st_sg)
have pkok:  $posk0\ k = AR\_AtLE \longleftrightarrow mt\_pos\ cM\ k = 0$  for k
  using posk_body idx_eq by simp
have c'_state:  $mt\_state\ c' = (mt\_state\ cM, AR\_SimRead, 0, 0, buf0, dvec,$ 
posk0)

```

using *st sg qM' eq idx eq tk0 ii0* **by** *simp*
 — The frozen tails beyond the tape count, read off *src_c'*.
have *buf0_tail*: $\forall j \geq k_tm\ M. \text{buf0 } j = \text{bl_tm } M$
using *src_c'* **by** (*simp add: ar_stage_bounded_def*)
have *dvec_tail*: $\forall j \geq k_tm\ M. \text{dvec } j = \text{dir}.N$
using *src_c'* **by** (*simp add: ar_stage_bounded_def*)
have *posk0_tail*: $\forall j \geq k_tm\ M. \text{posk0 } j = \text{AR_AtLE}$
using *src_c'* **by** (*simp add: ar_stage_bounded_def*)
 — Phase 1 — read: scan every active tape's *M*-cell block into *buf*, landing at *AR_SimCompute*. Beyond the tape count *buf* / *posk* / position freeze to the entry config.
let *?aM* = $\lambda k. \text{mt_tape } cM\ k\ (\text{mt_pos } cM\ k)$
let *?rbuf* = $\lambda k. \text{if } k < k_tm\ M \text{ then } ?aM\ k \text{ else } \text{buf0 } k$
let *?rposk* = $\lambda k. \text{if } k < k_tm\ M$
 then (*if* *mt_pos cM k* = 0 *then* *AR_AtLE*
 else if *mt_pos cM k* = 1 *then* *AR_AtFirstProper*
 else *AR_AtFurtherProper*)
 else *posk0 k*
let *?rpos* = $\lambda k. \text{if } k < k_tm\ M$
 then (*if* *mt_pos cM k* = 0 *then* *Suc 0*
 else *sim_pos ?k (mt_pos cM k) + ?k*)
 else *mt_pos c' k*
obtain *c_r m_r* **where**
 r_chain: $(c', c_r) \in (\text{mttm_step } (\text{alphabet_reduce_delta } M)) \wedge m_r$
and *r_bound*: $m_r \leq ?N * (?k + 2)$
and *r_state*: *mt_state c_r* = (*mt_state cM*, *AR_SimCompute*, *k_unidx 0*,
 0,
 ?rbuf, *dvec*, *?rposk*)
and *r_tape*: *mt_tape c_r* = *mt_tape c'*
and *r_pos*: *mt_pos c_r* = *?rpos*
using *ar_read_phase*[*OF vM qQ kge2 c'_state tcorr ppos0 pkok tapeG bufG*
 pad_c' src_c']
by *blast*
 — Read-exit config padding-blank and bounded-stage facts: the tape and (beyond the tape count) position are unchanged, and the buffer / direction / posk tails freeze to the entry config.
have *pad_cr*: $\forall j \geq k_tm\ M. \text{mt_tape } c_r\ j\ (\text{mt_pos } c_r\ j) = \text{BLANK4}$
proof (*intro allI impI*)
 fix *j* **assume** *jk*: $k_tm\ M \leq j$
 have *mt_pos c_r j* = *mt_pos c' j* **using** *r_pos jk* **by** *simp*
 moreover **have** *mt_tape c_r j* = *mt_tape c' j* **using** *r_tape* **by** *simp*
 ultimately show *mt_tape c_r j (mt_pos c_r j)* = *BLANK4*
 using *pad_c' jk* **by** *simp*
qed
have *src_cr*: *ar_stage_bounded* (*bl_tm M*) (*k_tm M*)
 (*AR_SimCompute*, *k_unidx 0*, 0, *?rbuf*, *dvec*, *?rposk*)
using *kpos_tm buf0_tail dvec_tail posk0_tail*
by (*simp add: ar_stage_bounded_def k_unidx_zero*)
 — Phase 2 — compute: *M*'s read vector now sits in *buf* (the guarded read buffer

collapses to $?aM$ because both tails are the blank), so the fired δ -tuple steps the compute substep to $AR_SimWrite$.

```

have  $aM\_eq$ :  $(\lambda k. tsM\ k\ (nM\ k)) = ?aM$  by ( $simp\ add: cM\_eq$ )
have  $rbuf\_eq$ :  $?rbuf = ?aM$ 
proof ( $rule\ ext$ )
  fix  $k$  show  $?rbuf\ k = ?aM\ k$ 
  proof ( $cases\ k < k\_tm\ M$ )
    case  $True$  thus  $?thesis$  by  $simp$ 
  next
    case  $False$ 
    have  $?rbuf\ k = bl\_tm\ M$  using  $buf0\_tail\ False$  by  $simp$ 
    moreover have  $?aM\ k = bl\_tm\ M$  using  $aM\_tail\ False$  by  $simp$ 
    ultimately show  $?thesis$  by  $simp$ 
  qed
qed
have  $mdelta'$ :  $(mt\_state\ cM, ?aM, q', a', d) \in delta\_tm\ M$ 
  using  $mdelta$  by ( $simp\ add: qC\_eq\ aM\_eq$ )
have  $mdelta''$ :  $(mt\_state\ cM, ?rbuf, q', a', d) \in delta\_tm\ M$ 
  using  $mdelta'\ rbuf\_eq$  by  $simp$ 
have  $vsrc\_c$ :  $ar\_valid\_stage\ (\Gamma\_tm\ M)\ (bl\_tm\ M)$ 
  ( $AR\_SimCompute, k\_unidx\ 0, 0, ?rbuf, dvec, ?rposk$ )
  using  $kge2\ tapeG\ bufG$  by ( $auto\ simp: ar\_valid\_stage\_def$ )
obtain  $c\_c$  where
   $c\_step$ :  $(c\_r, c\_c) \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
  and  $c\_state$ :  $mt\_state\ c\_c = (q', AR\_SimWrite, 0, 0, a', d, ?rposk)$ 
  and  $c\_tape$ :  $mt\_tape\ c\_c = mt\_tape\ c\_r$ 
  and  $c\_pos$ :  $mt\_pos\ c\_c = mt\_pos\ c\_r$ 
  using  $ar\_compute\_step[OF\ vM\ r\_state\ mdelta''\ vsrc\_c\ pad\_cr\ src\_cr]$  by
 $blast$ 
— Phase 3 setup — write:  $M$ 's write vector  $a'$  is the new  $buf$ . Range / post-state
from  $valid\_mttm$ 's  $\delta$ -typing; the  $bufle$  biconditional from  $M$ 's  $\delta LE$  discipline and
 $le$ -only-at-0.
have  $q'Q$ :  $q' \in Q\_tm\ M$  using  $valid\_mttm\_delta(3)[OF\ vM\ mdelta]$  .
have  $a'G$ :  $a'\ k \in \Gamma\_tm\ M$  for  $k$  using  $valid\_mttm\_delta(4)[OF\ vM\ mdelta]$  .
have  $a'val$ :  $\forall k. a'\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$  using  $a'G$  by  $blast$ 
have  $le\_neq\_bl$ :  $bl\_tm\ M \neq le\_tm\ M$  using  $lebl$  by  $simp$ 
have  $dsupp$ :  $\forall j \geq k\_tm\ M. a'\ j = bl\_tm\ M \wedge d\ j = dir.N$ 
  using  $valid\_mttm\_delta\_support[OF\ vM\ mdelta]$  by  $blast$ 
— Cut-tolerant write dispatch. The write phase keys the boundary on the position-
kind flag  $posk$  (the read-phase  $?rposk$ , fixed by  $mt\_pos\ cM$ ), not on  $a' = le$  which
is false for a cut machine (it writes  $le$  off the boundary).  $poskle$  is that dispatch
fact, read off  $?rposk$  with no  $le\_unique$ . The correspondence rebuild needs only
the  $mt\_pos\ cM\ k = 0 \Rightarrow a'\ k = le$  direction ( $a'le0$ , by clause-1  $\delta LE$ , again no
 $le\_unique$ ).
have  $a'le0$ :  $\forall k < k\_tm\ M. mt\_pos\ cM\ k = 0 \longrightarrow a'\ k = le\_tm\ M$ 
proof ( $intro\ allI\ impI$ )
  fix  $k$  assume  $kN$ :  $k < k\_tm\ M$  and  $p0$ :  $mt\_pos\ cM\ k = 0$ 
  have  $tsk0$ :  $tsM\ k\ 0 = le\_tm\ M$  using  $valcM\ cM\_eq\ kN$  by ( $cases\ M$ )  $auto$ 
  have  $(\lambda j. tsM\ j\ (nM\ j))\ k = le\_tm\ M$  using  $tsk0\ p0\ cM\_eq$  by  $simp$ 

```

thus $a' k = le_tm\ M$ **using** $valid_mttm_deltaLE[OF\ vM\ mdelta]$ **by** $simp$
qed
have $poskle: \forall k < k_tm\ M. \ ?rposk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0$
by $auto$
have $tcorr_c: \forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)$
 $(bl_tm\ M)$
 $(mt_tape\ cM\ k)\ (mt_tape\ c_c\ k)$
using $tcorr\ c_tape\ r_tape$ **by** $simp$
have $ppos_c: \forall k < k_tm\ M. mt_pos\ c_c\ k = (if\ mt_pos\ cM\ k = 0\ then\ Suc\ 0$
 $else\ sim_pos\ ?k\ (mt_pos\ cM\ k) + ?k)$
using $c_pos\ r_pos$ **by** $simp$
have $pad_cc: \forall j \geq k_tm\ M. mt_tape\ c_c\ j\ (mt_pos\ c_c\ j) = BLANK4$
using $pad_cr\ c_tape\ c_pos$ **by** $simp$
have $src_cc: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite,\ 0,\ 0,\ a',\ d,\ ?rposk)$
using $kpos_tm\ dsupp\ posk0_tail$ **by** $(simp\ add: ar_stage_bounded_def)$
 — Phase 3 — write: stamp a 's cell blocks onto the active tapes, landing
 at $AR_SimAdvance$; heads unchanged, the tape becomes the per-tape overwrite
 $?wtape$.
let $?wtape = \lambda k. if\ k < k_tm\ M$
 $then\ (if\ mt_pos\ cM\ k = 0\ then\ mt_tape\ c_c\ k$
 $else\ (\lambda pos. if\ sim_pos\ ?k\ (mt_pos\ cM\ k) \leq pos$
 $\wedge pos < sim_pos\ ?k\ (mt_pos\ cM\ k) + ?k$
 $then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (a'\ k)$
 $(pos - sim_pos\ ?k\ (mt_pos\ cM\ k))$
 $else\ mt_tape\ c_c\ k\ pos))$
 $else\ mt_tape\ c_c\ k$
obtain $c_w\ m_w$ **where**
 $w_chain: (c_c,\ c_w) \in (mttm_step\ (alphabet_reduce_delta\ M))\ \wedge\wedge\ m_w$
and $w_bound: m_w \leq ?N * (2 * ?k)$
and $w_state: mt_state\ c_w = (q', AR_SimAdvance,\ k_unidx\ 0,\ 0,\ a',\ d,$
 $?rposk)$
and $w_tape: mt_tape\ c_w = ?wtape$
and $w_pos: mt_pos\ c_w = mt_pos\ c_c$
using $ar_write_phase[OF\ vM\ q'Q\ kge2\ c_state\ tcorr_c\ ppos_c\ poskle\ a'val$
 $pad_cc\ src_cc]$
by $blast$
 — Phase 4 setup — advance: move each active head to M 's new cell, landing at
 $AR_SimNext$. $dge1$ (no L -from- AR_AtLE): a head at M -position 0 reads le , so
 δLE -forward forbids an L -move. $notLE$: every walked cell of the write-output tape
 is $\neq LE4$.
have $w_state': mt_state\ c_w = (q', AR_SimAdvance,\ 0,\ 0,\ a',\ d,\ ?rposk)$
using w_state **by** $(simp\ add: k_unidx_zero)$
have $pad_cw: \forall j \geq k_tm\ M. mt_tape\ c_w\ j\ (mt_pos\ c_w\ j) = BLANK4$
proof $(intro\ allI\ impI)$
fix j **assume** $jk: k_tm\ M \leq j$
have $mt_tape\ c_w\ j = mt_tape\ c_c\ j$ **using** $w_tape\ jk$ **by** $simp$
moreover **have** $mt_pos\ c_w\ j = mt_pos\ c_c\ j$ **using** w_pos **by** $simp$
ultimately **show** $mt_tape\ c_w\ j\ (mt_pos\ c_w\ j) = BLANK4$

```

    using pad_cc jk by simp
qed
have src_cw: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimAdvance, 0, 0, a', d, ?rposk)
  using kpos_tm dsupp posk0_tail by (simp add: ar_stage_bounded_def)
have dge1:  $\forall k < k\_tm\ M. d\ k \neq dir.R \longrightarrow 0 < ar\_disp\ ?k\ (d\ k)\ (?rposk\ k)$ 
proof (intro allI impI)
  fix k assume kN:  $k < k\_tm\ M$  and dkR:  $d\ k \neq dir.R$ 
  show  $0 < ar\_disp\ ?k\ (d\ k)\ (?rposk\ k)$ 
  proof (cases mt_pos cM k = 0)
    case True
    have nM0:  $nM\ k = 0$  using True cM_eq by simp
    have tsM k 0 = le_tm M using valcM cM_eq kN by (cases M) auto
    hence  $(\lambda j. tsM\ j\ (nM\ j))\ k = le\_tm\ M$  using nM0 by simp
    hence  $d\ k \in \{dir.N, dir.R\}$  using valid_mttm_deltaLE[OF vM mdelta] by
simp
    hence  $d\ k = dir.N$  using dkR by auto
    moreover have ?rposk k = AR_AtLE using True kN by simp
    ultimately show  $0 < ar\_disp\ ?k\ (d\ k)\ (?rposk\ k)$  by simp
  next
    case False
    have dN_or_L:  $d\ k = dir.N \vee d\ k = dir.L$  using dkR by (cases d k) auto
    have rk: ?rposk k = AR_AtFirstProper  $\vee$  ?rposk k = AR_AtFurtherProper
      using False kN by auto
    from dN_or_L rk kge2 show  $0 < ar\_disp\ ?k\ (d\ k)\ (?rposk\ k)$  by auto
  qed
qed
have notLE:  $\forall k < k\_tm\ M. \forall m. m < ar\_disp\ ?k\ (d\ k)\ (?rposk\ k)$ 
 $\longrightarrow mt\_tape\ c\_w\ k\ (mt\_pos\ c\_w\ k - m) \neq LE4$ 
proof (intro allI impI)
  fix k m assume kN:  $k < k\_tm\ M$ 
  and mlt:  $m < ar\_disp\ ?k\ (d\ k)\ (?rposk\ k)$ 
  have tcorr_ck: ar_tape_correspondence ( $\Gamma\_tm\ M$ ) (le_tm M) (bl_tm M)
    (mt_tape cM k) (mt_tape c_c k)
  using tcorr_c kN by blast
  have wcpo:  $mt\_pos\ c\_w\ k = (if\ mt\_pos\ cM\ k = 0\ then\ Suc\ 0$ 
     $else\ sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k)$ 
  using w_pos ppos_c kN by simp
  have wtape_notLE:  $mt\_tape\ c\_w\ k\ pos \neq LE4$  if pos1:  $1 \leq pos$  for pos
  proof (cases mt_pos cM k = 0)
    case True
    have  $mt\_tape\ c\_w\ k\ pos = mt\_tape\ c\_c\ k\ pos$ 
    using w_tape True kN by simp
    thus ?thesis
    using ar_tape_correspondence_not_LE4[OF tcorr_ck pos1] by simp
  next
    case False
    have wexp:  $mt\_tape\ c\_w\ k\ pos$ 
       $= (if\ sim\_pos\ ?k\ (mt\_pos\ cM\ k) \leq pos$ 

```



```

       $\wedge pos < sim\_pos \ ?k \ (mt\_pos \ cM \ k) + \ ?k$ 
    then write_bit ( $\Gamma\_tm \ M$ ) ( $bl\_tm \ M$ ) ( $a' \ k$ )
      ( $pos - sim\_pos \ ?k \ (mt\_pos \ cM \ k)$ )
    else mt_tape c_c k pos)
  using w_tape False kN by simp
show ?thesis
proof (cases sim_pos ?k (mt_pos cM k)  $\leq pos$ 
       $\wedge pos < sim\_pos \ ?k \ (mt\_pos \ cM \ k) + \ ?k$ )
  case True
  have b:  $pos - sim\_pos \ ?k \ (mt\_pos \ cM \ k) < \ ?k$  using True by linarith
  have mt_tape c_w k pos
    = write_bit ( $\Gamma\_tm \ M$ ) ( $bl\_tm \ M$ ) ( $a' \ k$ )
      ( $pos - sim\_pos \ ?k \ (mt\_pos \ cM \ k)$ )
    using wexp True by simp
  thus ?thesis using write_bit_not_LE4[OF b] by simp
next
  case False
  have nreg:  $\neg (sim\_pos \ ?k \ (mt\_pos \ cM \ k) \leq pos$ 
       $\wedge pos < sim\_pos \ ?k \ (mt\_pos \ cM \ k) + \ ?k)$ 
    using False by simp
  have mt_tape c_w k pos = mt_tape c_c k pos
    using wexp by (simp add: if_not_P[OF nreg])
  thus ?thesis
    using ar_tape_correspondence_not_LE4[OF tcorr_ck pos1] by simp
qed
qed
have rge:  $ar\_disp \ ?k \ (d \ k) \ (\?rposk \ k) \leq mt\_pos \ c\_w \ k$ 
proof (cases mt_pos cM k = 0)
  case True
  have ar_disp ?k (d k) (?rposk k)  $\leq Suc \ 0$ 
    using True kN by (cases d k) auto
  thus ?thesis using wcpes True by simp
next
  case nz: False
  show ?thesis
  proof (cases mt_pos cM k = 1)
    case True
    have ar_disp ?k (d k) (?rposk k)  $\leq Suc \ ?k$ 
      using True kN by (cases d k) auto
    thus ?thesis using wcpes True by (simp add: sim_pos_def)
  next
    case False
    have p2:  $2 \leq mt\_pos \ cM \ k$  using nz False by simp
    have d2:  $ar\_disp \ ?k \ (d \ k) \ (\?rposk \ k) \leq 2 * \ ?k$ 
      using nz False kN by (cases d k) auto
    have cw_eq:  $mt\_pos \ c\_w \ k = (mt\_pos \ cM \ k - 1) * \ ?k + 1 + \ ?k$ 
      using wcpes nz by (simp add: sim_pos_def)
    have (1::nat)  $\leq mt\_pos \ cM \ k - 1$  using p2 by simp
    hence  $1 * \ ?k \leq (mt\_pos \ cM \ k - 1) * \ ?k$  by (rule mult_le_mono1)

```

hence $kk: ?k \leq (mt_pos\ cM\ k - 1) * ?k$ **by** (*simp only: mult_1_left*)
 have $2 * ?k \leq mt_pos\ c_w\ k$ **using** $kk\ cw_eq$ **by** *linarith*
 thus $?thesis$ **using** $d2$ **by** *linarith*
 qed
 qed
 have $m < mt_pos\ c_w\ k$ **using** $mlt\ rge$ **by** *simp*
 hence $1 \leq mt_pos\ c_w\ k - m$ **by** *simp*
 thus $mt_tape\ c_w\ k\ (mt_pos\ c_w\ k - m) \neq LE4$ **by** (*rule wtape_notLE*)
 qed
 let $?apok = \lambda k. \text{if } k < k_tm\ M \text{ then } ar_newpos\ (d\ k)\ (?rposk\ k) \text{ else } ?rposk\ k$
 let $?apos = \lambda k. \text{if } k < k_tm\ M$
 then (*if* $d\ k = dir.R$ then $mt_pos\ c_w\ k$
 else $mt_pos\ c_w\ k - ar_disp\ ?k\ (d\ k)\ (?rposk\ k)$)
 else $mt_pos\ c_w\ k$
 obtain $c_adv\ m_a$ **where**
 $a_chain: (c_w, c_adv) \in (mttm_step\ (alphabet_reduce_delta\ M)) \wedge m_a$
 and $a_bound: m_a \leq ?N * (2 * ?k)$
 and $a_state: mt_state\ c_adv = (q', AR_SimNext, k_unidx\ 0, 0, a', d, ?apok)$
 and $a_tape: mt_tape\ c_adv = mt_tape\ c_w$
 and $a_pos: mt_pos\ c_adv = ?apos$
 using $ar_advance_phase[OF\ vM\ q'Q\ kge2\ w_state'\ a'val\ dge1\ notLE\ pad_cw\ src_cw]$
 by *blast*
 — Phase 5 setup — next hand-off: destination-stage validity, the folded read $\rightarrow$ advance prefix chain with its bound, and the advance-exit padding-blank / bounded-stage facts.
 have $st_cM1: mt_state\ cM_1 = q'$ **using** q'_eq **by** *simp*
 have $vsr_n: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 ($AR_SimNext, k_unidx\ 0, 0, a', d, ?apok$)
 using $kge2\ a'val$ **by** (*auto simp: ar_valid_stage_def*)
 have $rc: (c', c_c) \in (mttm_step\ (alphabet_reduce_delta\ M)) \wedge Suc\ m_r$
 using $r_chain\ c_step$ **by** (*rule relpow_Suc_I*)
 have $rcw: (c', c_w)$
 $\in (mttm_step\ (alphabet_reduce_delta\ M)) \wedge (Suc\ m_r + m_w)$
 by (*subst relpow_add*) (*rule relcompI[OF rc w_chain]*)
 have $rcwa: (c', c_adv)$
 $\in (mttm_step\ (alphabet_reduce_delta\ M))$
 $\wedge (Suc\ m_r + m_w + m_a)$
 by (*subst relpow_add*) (*rule relcompI[OF rcw a_chain]*)
 have $dist: ?N * (5 * ?k + 2)$
 $= ?N * (?k + 2) + ?N * (2 * ?k) + ?N * (2 * ?k)$
 by (*simp add: algebra_simps*)
 have $pre_bound: Suc\ m_r + m_w + m_a \leq ?N * (5 * ?k + 2) + 1$
 using $r_bound\ w_bound\ a_bound\ dist$ **by** *linarith*
 — Padding heads sit at position 0: $posk0$'s tail is AR_AtLE , so $pkok$ reads off $mt_pos\ cM\ j = 0$.
 have $padpos0: \forall j \geq k_tm\ M. mt_pos\ cM\ j = 0$
 using $pkok\ posk0_tail$ **by** *blast*
 have $pad_cadv: \forall j \geq k_tm\ M. mt_tape\ c_adv\ j\ (mt_pos\ c_adv\ j) = BLANK4$

```

proof (intro allI impI)
  fix j assume jk:  $k\_tm\ M \leq j$ 
  have mt_pos c_adv j = mt_pos c_w j using a_pos jk by simp
  moreover have mt_tape c_adv j = mt_tape c_w j using a_tape by simp
  ultimately show mt_tape c_adv j (mt_pos c_adv j) = BLANK4
    using pad_cw jk by simp
qed
have src_cadv: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimNext, k_unidx 0, 0, a', d, ?apok)
  using kpos_tm dsupp posk0_tail
  by (simp add: ar_stage_bounded_def k_unidx_zero)
have src_read: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, 0, 0, a', d, ?apok)
  using kpos_tm dsupp posk0_tail by (simp add: ar_stage_bounded_def)
— M's post-step head position, and the no-L-from-0 discipline (active tapes only).
have posM1: mt_pos cM_1 k = go_dir (d k) (mt_pos cM k) for k
  using cM1_eq cM_eq by simp
have dposL:  $d\ k \neq dir.L$  if p0: mt_pos cM k = 0 and kN:  $k < k\_tm\ M$  for k
proof —
  have nM0:  $nM\ k = 0$  using p0 cM_eq by simp
  have tsM k 0 = le_tm M using valcM cM_eq kN by (cases M) auto
  hence ( $\lambda j. tsM\ j\ (nM\ j)$ ) k = le_tm M using nM0 by simp
  hence  $d\ k \in \{dir.N, dir.R\}$  using valid_mttm_deltaLE[OF vM mdelta] by
simp
  thus  $d\ k \neq dir.L$  by auto
qed
— The tape correspondence re-established at cM_1 (forward simulation core):
on active tapes via ar_write_tape_correspondence.
have cM1tape: mt_tape cM_1 k = (mt_tape cM k)(mt_pos cM k := a' k) for
k
  using cM1_eq cM_eq by simp
have wt_def: mt_tape c_adv k pos
  = (if mt_pos cM k = 0 then mt_tape c' k pos
    else if sim_pos ?k (mt_pos cM k) ≤ pos
      ∧ pos < sim_pos ?k (mt_pos cM k) + ?k
      then write_bit (Γ_tm M) (bl_tm M) (a' k)
      (pos - sim_pos ?k (mt_pos cM k))
    else mt_tape c' k pos) for k pos
proof (cases k < k_tm M)
  case True
    thus ?thesis using a_tape w_tape c_tape r_tape by simp
  next
    case False
      hence kge:  $k\_tm\ M \leq k$  by simp
      have p0: mt_pos cM k = 0 using padpos0 kge by blast
      have mt_tape c_adv k = mt_tape c' k
      using a_tape w_tape c_tape r_tape kge by simp
      thus ?thesis using p0 by simp
qed

```

```

have tcorr_1:  $\forall k < k\_tm\ M. ar\_tape\_correspondence\ (\Gamma\_tm\ M)\ (le\_tm\ M)$ 
(bl_tm M)
      (mt_tape cM_1 k) (mt_tape c_adv k)
proof (intro allI impI)
  fix k assume kN:  $k < k\_tm\ M$ 
  show ar_tape_correspondence  $(\Gamma\_tm\ M)\ (le\_tm\ M)\ (bl\_tm\ M)$ 
      (mt_tape cM_1 k) (mt_tape c_adv k)
  by (rule ar_write_tape_correspondence
      [OF kge2 kN tcorr cM1tape a'le0 wt_def])
qed
— Head position and posk consistency at cM_1: on active tapes via the advance
helpers; on padding ( $d\ k = N$ , head at 0) both reduce to the entry config.
have apos_1:  $mt\_pos\ c\_adv\ k = sim\_pos\ ?k\ (mt\_pos\ cM\_1\ k)$  for k
proof (cases  $k < k\_tm\ M$ )
  case kN: True
  have key: (if  $d\ k = dir.R$ 
    then (if  $mt\_pos\ cM\ k = 0$  then Suc 0
      else  $sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k$ )
    else (if  $mt\_pos\ cM\ k = 0$  then Suc 0
      else  $sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k$ )
      - ar_disp ?k (d k)
      (if  $mt\_pos\ cM\ k = 0$  then AR_AtLE
        else if  $mt\_pos\ cM\ k = 1$  then AR_AtFirstProper
        else AR_AtFurtherProper))
    =  $sim\_pos\ ?k\ (go\_dir\ (d\ k)\ (mt\_pos\ cM\ k))$ )
  proof (rule ar_advance_newsimpos)
  show  $2 \leq ?k$  by (rule kge2)
  show  $mt\_pos\ cM\ k = 0 \implies d\ k \neq dir.L$  using dposL kN by blast
qed
have c_adv_pos:  $mt\_pos\ c\_adv\ k = (if\ d\ k = dir.R$ 
  then (if  $mt\_pos\ cM\ k = 0$  then Suc 0
    else  $sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k$ )
  else (if  $mt\_pos\ cM\ k = 0$  then Suc 0
    else  $sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k$ )
    - ar_disp ?k (d k)
    (if  $mt\_pos\ cM\ k = 0$  then AR_AtLE
      else if  $mt\_pos\ cM\ k = 1$  then AR_AtFirstProper
      else AR_AtFurtherProper))
  using a_pos w_pos c_pos r_pos kN by simp
show ?thesis by (simp add: c_adv_pos key posM1)
next
case False
hence kge:  $k\_tm\ M \leq k$  by simp
have dN:  $d\ k = dir.N$  using dsupp kge by blast
have  $mt\_pos\ c\_adv\ k = mt\_pos\ c'\ k$ 
  using a_pos w_pos c_pos r_pos kge by simp
also have ... =  $sim\_pos\ ?k\ (mt\_pos\ cM\ k)$  using ppos0 by simp
also have ... =  $sim\_pos\ ?k\ (mt\_pos\ cM\_1\ k)$  using posM1 dN by simp
finally show ?thesis .

```

```

qed
have aposk_1: ?aposk k = AR_AtLE  $\longleftrightarrow$  mt_pos cM_1 k = 0 for k
proof (cases k < k_tm M)
  case kN: True
  have key: (ar_newpos (d k) (if mt_pos cM k = 0 then AR_AtLE
    else if mt_pos cM k = 1 then AR_AtFirstProper
    else AR_AtFurtherProper) = AR_AtLE)
     $\longleftrightarrow$  go_dir (d k) (mt_pos cM k) = 0
  proof (rule ar_newpos_atLE_iff)
    show mt_pos cM k = 0  $\implies$  d k  $\neq$  dir.L using dposL kN by blast
  qed
  have aposk_k: ?aposk k = ar_newpos (d k)
    (if mt_pos cM k = 0 then AR_AtLE
    else if mt_pos cM k = 1 then AR_AtFirstProper
    else AR_AtFurtherProper)
    using kN by simp
  show ?thesis by (simp add: aposk_k key posM1)
next
case False
hence kge: k_tm M  $\leq$  k by simp
have dN: d k = dir.N using dsupp kge by blast
have p0: mt_pos cM k = 0 using padpos0 kge by blast
have mt_pos cM_1 k = 0 using posM1 dN p0 by simp
moreover have pk0: posk0 k = AR_AtLE using posk0_tail kge by blast
moreover have ?aposk k = posk0 k using kge by simp
ultimately show ?thesis by simp
qed
— Three-way dispatch on the post-step  $M$ -state  $q'$ : accept / reject (terminal
hand-off) or continue (next  $AR\_SimRead$  boundary). Each appends one  $M'$ -step
within bound and re-establishes the three invariants.
consider (acc)  $q' = t\_tm\ M$  | (rej)  $q' = r\_tm\ M$ 
  | (cont)  $q' \neq t\_tm\ M$   $q' \neq r\_tm\ M$ 
  by blast
then show thesis
proof cases
  case acc
  obtain c'' where
    acc_step: (c_adv, c'')  $\in$  mttm_step (alphabet_reduce_delta M)
    and acc_state: mt_state c'' = (t_tm M, ar_accept_stage (bl_tm M))
    and acc_tape: mt_tape c'' = mt_tape c_adv
    and acc_pos: mt_pos c'' = mt_pos c_adv
    using ar_accept_step[OF vM a_state acc vsrc_n pad_cadv src_cadv] by
blast
  show thesis
proof (rule that[of Suc (Suc m_r + m_w + m_a) c''])
  show Suc (Suc m_r + m_w + m_a)  $\leq$  k_tm M * (5 * ?k + 2) + 2
    using pre_bound by linarith
  show (c', c'')  $\in$  (mttm_step (alphabet_reduce_delta M))
    ^^ Suc (Suc m_r + m_w + m_a)

```

```

    using rcwa acc_step by (rule relpow_Suc_I)
  show ar_simulates M cM_1 c''
    by (simp add: ar_simulates_def Let_def acc_state ar_accept_stage_def
      st_cM1 acc_tcorr_1 acc_tape)
  show ar_posk_consistent M cM_1 c''
    by (simp add: ar_posk_consistent_def acc_state ar_accept_stage_def)
  show ar_at_read_boundary M c''
    by (simp add: ar_at_read_boundary_def acc_state ar_accept_stage_def
      acc_tape acc_pos pad_cadv)
qed
next
case rej
obtain c'' where
  rej_step: (c_adv, c'') ∈ mttm_step (alphabet_reduce_delta M)
  and rej_state: mt_state c'' = (r_tm M, ar_reject_stage (bl_tm M))
  and rej_tape: mt_tape c'' = mt_tape c_adv
  and rej_pos: mt_pos c'' = mt_pos c_adv
  using ar_reject_step[OF vM a_state rej_vsrc_n pad_cadv src_cadv] by blast
show thesis
proof (rule that[of Suc (Suc m_r + m_w + m_a) c''])
  show Suc (Suc m_r + m_w + m_a) ≤ k_tm M * (5 * ?k + 2) + 2
    using pre_bound by linarith
  show (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^ Suc (Suc m_r + m_w + m_a)
    using rcwa rej_step by (rule relpow_Suc_I)
  show ar_simulates M cM_1 c''
    by (simp add: ar_simulates_def Let_def rej_state ar_reject_stage_def
      st_cM1 rej_tcorr_1 rej_tape)
  show ar_posk_consistent M cM_1 c''
    by (simp add: ar_posk_consistent_def rej_state ar_reject_stage_def)
  show ar_at_read_boundary M c''
    by (simp add: ar_at_read_boundary_def rej_state ar_reject_stage_def
      rej_tape rej_pos pad_cadv)
qed
next
case cont
obtain c'' where
  nxt_step: (c_adv, c'') ∈ mttm_step (alphabet_reduce_delta M)
  and nxt_state: mt_state c'' = (q', AR_SimRead, 0, 0, a', d, ?apok)
  and nxt_tape: mt_tape c'' = mt_tape c_adv
  and nxt_pos: mt_pos c'' = mt_pos c_adv
  using ar_next_step[OF vM a_state q'Q cont(1) cont(2) vsrc_n
    pad_cadv src_cadv] by blast
show thesis
proof (rule that[of Suc (Suc m_r + m_w + m_a) c''])
  show Suc (Suc m_r + m_w + m_a) ≤ k_tm M * (5 * ?k + 2) + 2
    using pre_bound by linarith
  show (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))
    ^ Suc (Suc m_r + m_w + m_a)

```

```

    using rcwa next_step by (rule relpow_Suc_I)
  show ar_simulates M cM_1 c''
    by (simp add: ar_simulates_def Let_def next_state next_tape next_pos
        st_cM1 cont tcorr_1 apos_1)
  show ar_posk_consistent M cM_1 c''
    by (simp add: ar_posk_consistent_def next_state next_pos aposk_1)
  show ar_at_read_boundary M c''
    by (simp add: ar_at_read_boundary_def next_state next_tape next_pos
        a'G pad_cadv src_read)
qed
qed
qed

```

Chunked-induction engine for the simulation phase, the AR analogue of AE's *ae_simulation_phase_chunked*. Given an accepting *M*-path of length *n* from a reachable *cM* with a paired *AR_SimRead* *M'*-config *c'* satisfying *ar_simulates*, exhibit a corresponding accepting *M'*-path of length at most $(5b + 2) \cdot n$ ending in the canonical accept config. Unlike AE — which groups *c* source steps per stage and needs the strong *less_induct* with *min* *n* *c* chunk arithmetic — AR simulates one *M*-step per phase, so ordinary *induction* *n* with the fixed per-step bound suffices: each step contributes $\leq 5b + 2$, summing to $(5b + 2) \cdot n$ by a single *linarith*. All substrate reasoning is delegated to *ar_simulates_forward_step*; the engine only splits the trace, recurses on the tail, and composes the two *M'*-paths via *relpow_add*.

```

lemma ar_simulation_phase_chunked:
  fixes M :: ('q, 'a) mttm
    and w :: 'a list
    and cM cM_final :: ('a, 'q) mt_config
    and c' :: (sym4, 'q × 'a ar_stage) mt_config
    and n :: nat
  assumes vM:      valid_mttm M
    and w_sub:    set w ⊆ Sigma_tm M
    and card_ge:  card (Γ_tm M) ≥ 4
    and reach_M:  (init_config_mttm M w, cM) ∈ (mttm_step (delta_tm M))^*
    and trace:    (cM, cM_final) ∈ (mttm_step (delta_tm M))^n
    and accept:   mt_state cM_final = t_tm M
    and sim:      ar_simulates M cM c'
    and posk_ok:  ar_posk_consistent M cM c'
    and rbnd:     ar_at_read_boundary M c'
    and lebl:     le_tm M ≠ bl_tm M
  shows ∃ m c''. m ≤ (k_tm M
    * (5 * block_width (Γ_tm M) + 2) + 2) * n
    ∧ (c', c'') ∈ (mttm_step (alphabet_reduce_delta M))^m
    ∧ mt_state c'' = (t_tm M, ar_accept_stage (bl_tm M))
  using reach_M trace sim posk_ok rbnd
proof (induction n arbitrary: cM c')
  case 0
    — Base case: a length-0 trace forces cM = cM_final, so cM accepts;

```

$ar_simulates_accept_iff$ pins c' to the canonical accept config. Witness $m = 0$, $c'' = c'$.

have cMf : $cM = cM_final$ **using** $0.premis(2)$ **by** *simp*
have mt_state $cM = t_tm\ M$ **using** $accept\ cMf$ **by** *simp*
hence $state_c'$: $mt_state\ c' = (t_tm\ M, ar_accept_stage\ (bl_tm\ M))$
using $ar_simulates_accept_iff[OF\ vM\ 0.premis(3)]$ **by** *simp*
have $run0$: $(c', c') \in (mttm_step\ (alphabet_reduce_delta\ M))^{\wedge 0}$ **by** *simp*
have $bound0$: $(0::nat) \leq (5 * block_width\ (\Gamma_tm\ M) + 2) * 0$ **by** *simp*
show $?case$ **using** $state_c'\ run0\ bound0$ **by** *blast*

next

case $(Suc\ n')$

— Inductive case: split off the first M -step $cM \rightarrow cM_1$, run $ar_simulates_forward_step$ for its M' -phase $c' \rightarrow p\ c8$ ($p \leq 5b + 2$), extend reachability to cM_1 , recurse on the length- n' tail, and compose.

obtain cM_1 **where**

$step1$: $(cM, cM_1) \in mttm_step\ (delta_tm\ M)$
and $rest$: $(cM_1, cM_final) \in mttm_step\ (delta_tm\ M)^{\wedge n'}$
using $relpow_Suc_D2[OF\ Suc.premis(2)]$ **by** *blast*

obtain $p\ c8$ **where**

$pbound$: $p \leq k_tm\ M * (5 * block_width\ (\Gamma_tm\ M) + 2) + 2$
and $chainp$: $(c', c8) \in mttm_step\ (alphabet_reduce_delta\ M)^{\wedge p}$
and $sim8$: $ar_simulates\ M\ cM_1\ c8$
and $posk8$: $ar_posk_consistent\ M\ cM_1\ c8$
and $bnd8$: $ar_at_read_boundary\ M\ c8$
by $(rule\ ar_simulates_forward_step$

$[OF\ vM\ w_sub\ card_ge\ Suc.premis(1)\ step1$
 $Suc.premis(3)\ Suc.premis(4)\ Suc.premis(5)\ lebl])$

have $reach1$: $(init_config_mttm\ M\ w, cM_1) \in (mttm_step\ (delta_tm\ M))^*$
using $Suc.premis(1)\ step1$ **by** $(rule\ rtrancl_into_rtrancl)$

obtain $m_rec\ c''$ **where**

$mbound$: $m_rec \leq (k_tm\ M$
 $* (5 * block_width\ (\Gamma_tm\ M) + 2) + 2) * n'$
and $chainrec$: $(c8, c'') \in (mttm_step\ (alphabet_reduce_delta\ M))^{\wedge m_rec}$
and $statec''$: $mt_state\ c'' = (t_tm\ M, ar_accept_stage\ (bl_tm\ M))$
using $Suc.IH[OF\ reach1\ rest\ sim8\ posk8\ bnd8]$ **by** *blast*

have $chain$: $(c', c'') \in (mttm_step\ (alphabet_reduce_delta\ M))^{\wedge (p + m_rec)}$
using $chainp\ chainrec$ **by** $(auto\ simp: relpow_add)$

have $bound$: $p + m_rec \leq (k_tm\ M$
 $* (5 * block_width\ (\Gamma_tm\ M) + 2) + 2) * Suc\ n'$

proof —

have $p + m_rec \leq (k_tm\ M * (5 * block_width\ (\Gamma_tm\ M) + 2) + 2)$
 $+ (k_tm\ M$
 $* (5 * block_width\ (\Gamma_tm\ M) + 2) + 2) * n'$

using $pbound\ mbound$ **by** *linarith*

also have $\dots = (k_tm\ M$
 $* (5 * block_width\ (\Gamma_tm\ M) + 2) + 2) * Suc\ n'$ **by** *simp*

finally show $?thesis$.

qed

show $?case$ **using** $bound\ chain\ statec''$ **by** *blast*


```

qed

end
theory AlphabetReduction_Theorems
  imports AlphabetReduction_Forward
begin

```

2.17 Top-level theorems

Per-substep typing facts: for each substep relation in the five-substep union, the source and target states' q components lie in $Q_tm\ M$, and the source's $ar_substep_idx$ tag is the relation-specific value (never $AR_HaltAccept$ or $AR_HaltReject$). Used by $alphabet_reduce_wf$'s δ' -typing conjunct to show δ' -tuples land in $(Q' - \{t', r'\}) \times \dots \times Q' \times \dots$.

```

lemma ar_delta_read_typing:
  assumes  $(s, a, s', a', d) \in ar\_delta\_read\ M$ 
  shows  $fst\ s \in Q\_tm\ M \wedge fst\ (snd\ s) = AR\_SimRead$ 
         $\wedge fst\ s' \in Q\_tm\ M$ 
  using assms
  unfolding ar_delta_read_def
  by (cases M) auto

```

```

lemma ar_delta_compute_typing:
  assumes valM: valid_mttm M
  and tr:  $(s, a, s', a', d) \in ar\_delta\_compute\ M$ 
  shows  $fst\ s \in Q\_tm\ M \wedge fst\ (snd\ s) = AR\_SimCompute$ 
         $\wedge fst\ s' \in Q\_tm\ M$ 

```

```

proof -
  from tr obtain q buf q' m_a' m_d tk i dvec posk a'' d''
  where decomp:  $s = (q, AR\_SimCompute, tk, i, buf, dvec, posk)$ 
  and decomp':  $s' = (q', AR\_SimWrite, 0, 0, m\_a', m\_d, posk)$ 
  and m_tr:  $(q, buf, q', m\_a', m\_d) \in delta\_tm\ M$ 
  unfolding ar_delta_compute_def by auto
  from valid_mttm_delta_set[OF valM] m_tr
  have  $q \in Q\_tm\ M$  and  $q' \in Q\_tm\ M$  by auto
  thus ?thesis using decomp decomp' by simp
qed

```

```

lemma ar_delta_write_typing:
  assumes  $(s, a, s', a', d) \in ar\_delta\_write\ M$ 
  shows  $fst\ s \in Q\_tm\ M \wedge fst\ (snd\ s) = AR\_SimWrite$ 
         $\wedge fst\ s' \in Q\_tm\ M$ 
  using assms
  unfolding ar_delta_write_def
  by (cases M) auto

```

```

lemma ar_delta_advance_typing:
  assumes  $(s, a, s', a', d) \in ar\_delta\_advance\ M$ 

```

```

shows  $\text{fst } s \in Q\_tm\ M \wedge \text{fst } (\text{snd } s) = AR\_SimAdvance$ 
          $\wedge \text{fst } s' \in Q\_tm\ M$ 
using assms
unfolding ar_delta_advance_def
by (cases M) auto

lemma ar_delta_next_typing:
assumes valM: valid_mttm M
and tr:  $(s, a, s', a', d) \in ar\_delta\_next\ M$ 
shows  $\text{fst } s \in Q\_tm\ M \wedge \text{fst } (\text{snd } s) = AR\_SimNext$ 
          $\wedge \text{fst } s' \in Q\_tm\ M$ 
proof –
from valid_mttm_t_in_Q[OF valM] valid_mttm_r_in_Q[OF valM]
have  $t\_tm\ M \in Q\_tm\ M$  and  $r\_tm\ M \in Q\_tm\ M$  by auto
thus ?thesis
using tr
unfolding ar_delta_next_def
by (cases M) auto
qed

```

State-shape extraction for *alphabet_reduce_delta*: every transition's source and target carry an *M*-state in *Q*, and the source stage's index is a *simulation* index — never a halt index — so the source state is distinct from both canonical halt states. Case-split across the five substep relations (the delta is their union intersected with global guards, so membership lands in the union), each branch discharged by its typing lemma; the halt-index distinctness is datatype-level.

```

lemma alphabet_reduce_delta_state_shape:
assumes valM: valid_mttm M
and mem:  $(s, a, s', a', d) \in alphabet\_reduce\_delta\ M$ 
shows  $\text{fst } s \in Q\_tm\ M \wedge \text{fst } s' \in Q\_tm\ M$ 
          $\wedge \text{fst } (\text{snd } s) \neq AR\_HaltAccept$ 
          $\wedge \text{fst } (\text{snd } s) \neq AR\_HaltReject$ 
proof –
have memU:  $(s, a, s', a', d)$ 
          $\in ar\_delta\_read\ M \cup ar\_delta\_compute\ M$ 
          $\cup ar\_delta\_write\ M \cup ar\_delta\_advance\ M$ 
          $\cup ar\_delta\_next\ M$ 
using mem unfolding alphabet_reduce_delta_def by blast
then show ?thesis
proof (elim UnE)
assume  $(s, a, s', a', d) \in ar\_delta\_read\ M$ 
from ar_delta_read_typing[OF this] show ?thesis by auto
next
assume  $(s, a, s', a', d) \in ar\_delta\_compute\ M$ 
from ar_delta_compute_typing[OF valM this] show ?thesis by auto
next
assume  $(s, a, s', a', d) \in ar\_delta\_write\ M$ 
from ar_delta_write_typing[OF this] show ?thesis by auto

```

```

next
  assume  $(s, a, s', a', d) \in ar\_delta\_advance\ M$ 
  from  $ar\_delta\_advance\_typing[OF\ this]$  show ?thesis by auto
next
  assume  $(s, a, s', a', d) \in ar\_delta\_next\ M$ 
  from  $ar\_delta\_next\_typing[OF\ valM\ this]$  show ?thesis by auto
qed
qed

```

Output well-formedness: *alphabet_reduce* preserves the substrate's wf predicate when the input tape alphabet has at least 4 symbols. The reduced machine's tape alphabet is the whole finite type *sym4*, so the read / write codomain conditions are vacuous (*UNIV*), and the δLE -preservation, δLE -no-write, and δ -support-past-tape-count conjuncts are read straight off *alphabet_reduce_delta*'s three intersection guards — no substep-builder unfold. The only structural work is finite Q' (product of finite Q with the bounded valid stages, *finite_ar_valid_stages*), the three distinguished states landing in Q' (their stages are valid and bounded), and the δ -shape's source / target state membership (*alphabet_reduce_delta_state_shape* plus the stage guards).

```

theorem alphabet_reduce_wf:
  fixes  $M :: ('q, 'a) mttm$ 
  assumes  $valM: valid\_mttm\ M$ 
  and  $cardG: card\ (\Gamma\_tm\ M) \geq 4$ 
  shows  $valid\_mttm$ 
     $(alphabet\_reduce\ M$ 
       $:: ('q \times 'a\ ar\_stage, sym4) mttm)$ 
proof -
  obtain  $Q_M\ \Sigma_M\ \Gamma_M\ bl_M\ le_M\ \delta_M\ s_M\ t_M\ r_M\ k_M$  where
     $M\_eq: M = MTTM\ Q_M\ \Sigma_M\ \Gamma_M\ bl_M\ le_M\ \delta_M\ s_M\ t_M\ r_M\ k_M$ 
  using mttm.exhaust by metis

```

```

have  $fin\_Gamma: finite\ \Gamma_M$ 
  using  $valid\_mttm\_finite\_Gamma[OF\ valM]\ M\_eq$  by simp
have  $kpos: 0 < k_M$ 
  using  $valid\_mttm\_k\_pos[OF\ valM]\ M\_eq$  by simp
have  $kfor1: 1 \leq block\_width\ \Gamma_M$  by (rule block_width_pos)

```

— The reduced machine, expanded under M_eq .

```

have  $ar\_eq: (alphabet\_reduce\ M :: ('q \times 'a\ ar\_stage, sym4) mttm) =$ 
   $MTTM\ (Q_M \times \{stg.\ ar\_valid\_stage\ \Gamma_M\ bl_M\ stg$ 
     $\wedge\ ar\_stage\_bounded\ bl_M\ k_M\ stg\})$ 
   $\{BIT0, BIT1\}$ 
   $(UNIV :: sym4\ set)$ 
   $BLANK4\ LE4$ 
   $(alphabet\_reduce\_delta\ M)$ 
   $(s_M, ar\_init\_stage\ bl_M)$ 
   $(t_M, ar\_accept\_stage\ bl_M)$ 

```

$(r_M, ar_reject_stage\ bl_M)$
 k_M

unfolding $M_eq\ alphabet_reduce_def$ **by** *simp*

— The three distinguished stages are valid and bounded: counter $0 < 2 \cdot b$, current-tape $0 < k$, frozen tails.

have $init_P$: $ar_valid_stage\ \Gamma_M\ bl_M\ (ar_init_stage\ bl_M)$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ (ar_init_stage\ bl_M)$
using *kfor1 kpos*
by (*simp add: ar_valid_stage_def ar_stage_bounded_def ar_init_stage_def*)
have acc_P : $ar_valid_stage\ \Gamma_M\ bl_M\ (ar_accept_stage\ bl_M)$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ (ar_accept_stage\ bl_M)$
using *kfor1 kpos*
by (*simp add: ar_valid_stage_def ar_stage_bounded_def*
 $ar_accept_stage_def$)
have rej_P : $ar_valid_stage\ \Gamma_M\ bl_M\ (ar_reject_stage\ bl_M)$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ (ar_reject_stage\ bl_M)$
using *kfor1 kpos*
by (*simp add: ar_valid_stage_def ar_stage_bounded_def*
 $ar_reject_stage_def$)

show *valid_mttm*
 $(alphabet_reduce\ M :: ('q \times 'a\ ar_stage, sym4)\ mttm)$
unfolding $ar_eq\ valid_mttm.simps$
proof (*intro conjI*)

— (1) finite Q' : product of finite Q and the bounded valid stages.

show *finite* $(Q_M \times$
 $\{stg.\ ar_valid_stage\ \Gamma_M\ bl_M\ stg$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ stg\}$
 $:: 'a\ ar_stage\ set)$
using *valid_mttm_finite_Q[OF valM] M_eq*
finite_ar_valid_stages[OF fin_Gamma]
by (*auto intro: finite_cartesian_product*)

— (2) finite Γ' : the whole finite type *sym4*.

show *finite* $(UNIV :: sym4\ set)$ **by** (*simp add: sym4_UNIV*)

— (3) $\Sigma' \subseteq \Gamma'$

show $\{BIT0, BIT1\} \subseteq (UNIV :: sym4\ set)$ **by** *simp*

— (4)-(6) the three distinguished states land in Q' .

show $(s_M, ar_init_stage\ bl_M)$
 $\in Q_M \times \{stg.\ ar_valid_stage\ \Gamma_M\ bl_M\ stg$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ stg\}$
using *valid_mttm_s_in_Q[OF valM] M_eq init_P* **by** *simp*
show $(t_M, ar_accept_stage\ bl_M)$
 $\in Q_M \times \{stg.\ ar_valid_stage\ \Gamma_M\ bl_M\ stg$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ stg\}$
using *valid_mttm_t_in_Q[OF valM] M_eq acc_P* **by** *simp*

show $(r_M, ar_reject_stage\ bl_M)$
 $\in Q_M \times \{stg.\ ar_valid_stage\ \Gamma_M\ bl_M\ stg$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ stg\}$
using $valid_mttm_r_in_Q[OF\ valM]\ M_eq\ rej_P$ **by** *simp*

— (7a)-(8b) blank / LE in Γ' , not in Σ' .

show $BLANK_4 \in (UNIV :: sym_4\ set)$ **by** *simp*

show $BLANK_4 \notin \{BIT0, BIT1\}$ **by** *simp*

show $LE_4 \in (UNIV :: sym_4\ set)$ **by** *simp*

show $LE_4 \notin \{BIT0, BIT1\}$ **by** *simp*

— (9) accept $\neq$ reject (states differ in M -component).

show $(t_M, ar_accept_stage\ bl_M) \neq (r_M, ar_reject_stage\ bl_M)$
using $valid_mttm_t_neq_r[OF\ valM]\ M_eq$ **by** *simp*

— (0) $0 < k'$: reduction keeps M 's tape count.

show $0 < k_M$ **using** *kpos* .

— (10) δ -shape. Read / write codomains are $UNIV$ (vacuous); source / target state membership from $alphabet_reduce_delta_state_shape$ (discrete M -state) plus the stage-validity / boundedness guards (off the intersection, no builder unfold).

show $alphabet_reduce_delta\ M$
 $\subseteq ((Q_M \times \{stg.\ ar_valid_stage\ \Gamma_M\ bl_M\ stg$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ stg\})$
 $- \{(t_M, ar_accept_stage\ bl_M),$
 $(r_M, ar_reject_stage\ bl_M)\})$
 $\times (UNIV \rightarrow (UNIV :: sym_4\ set))$
 $\times (Q_M \times \{stg.\ ar_valid_stage\ \Gamma_M\ bl_M\ stg$
 $\wedge ar_stage_bounded\ bl_M\ k_M\ stg\})$
 $\times (UNIV \rightarrow (UNIV :: sym_4\ set))$
 $\times (UNIV \rightarrow UNIV)$

proof (*rule subsetI*)

fix x **assume** $xin: x \in alphabet_reduce_delta\ M$

obtain $s\ a\ s'\ a'\ d$ **where** $x_eq: x = (s, a, s', a', d)$

by (*cases x*) *auto*

from xin **have** $mem_t: (s, a, s', a', d) \in alphabet_reduce_delta\ M$

unfolding x_eq **by** *simp*

— Stage validity / boundedness off the intersection guards.

from mem_t **have** *inter*:

$ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ s)$
 $\wedge ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (snd\ s')$
 $\wedge ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)\ (snd\ s)$
 $\wedge ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)\ (snd\ s')$

unfolding $alphabet_reduce_delta_def$ **by** *simp*

— Discrete M -state shape from the substep builders.

have $shape: fst\ s \in Q_tm\ M \wedge fst\ s' \in Q_tm\ M$

$\wedge fst\ (snd\ s) \neq AR_HaltAccept$

$\wedge fst\ (snd\ s) \neq AR_HaltReject$

by (*rule alphabet_reduce_delta_state_shape[OF\ valM\ mem_t]*)

```

have  $s\_in$ :  $s \in (Q_M \times \{stg. ar\_valid\_stage \Gamma_M bl_M stg$ 
   $\wedge ar\_stage\_bounded bl_M k_M stg\})$ 
   $- \{(t_M, ar\_accept\_stage bl_M),$ 
     $(r_M, ar\_reject\_stage bl_M)\}$ 
using shape inter  $M\_eq$ 
by (cases  $s$ ) (auto simp:  $ar\_accept\_stage\_def ar\_reject\_stage\_def$ )
have  $s'\_in$ :  $s' \in Q_M \times \{stg. ar\_valid\_stage \Gamma_M bl_M stg$ 
   $\wedge ar\_stage\_bounded bl_M k_M stg\}$ 
using shape inter  $M\_eq$  by (cases  $s'$ ) auto
show  $x \in ((Q_M \times \{stg. ar\_valid\_stage \Gamma_M bl_M stg$ 
   $\wedge ar\_stage\_bounded bl_M k_M stg\})$ 
   $- \{(t_M, ar\_accept\_stage bl_M),$ 
     $(r_M, ar\_reject\_stage bl_M)\})$ 
   $\times (UNIV \rightarrow (UNIV :: sym4 set))$ 
   $\times (Q_M \times \{stg. ar\_valid\_stage \Gamma_M bl_M stg$ 
     $\wedge ar\_stage\_bounded bl_M k_M stg\})$ 
   $\times (UNIV \rightarrow (UNIV :: sym4 set))$ 
   $\times (UNIV \rightarrow UNIV)$ 
unfolding  $x\_eq$  using  $s\_in s'\_in$ 
by (auto simp:  $mem\_Times\_iff Pi\_iff$ )
qed

```

— (11) δLE preservation — exactly the second intersection guard.

```

show  $\forall q a q' a' d j.$ 
   $(q, a, q', a', d) \in alphabet\_reduce\_delta M \longrightarrow$ 
   $a j = LE4 \longrightarrow a' j = LE4 \wedge d j \in \{dir.N, dir.R\}$ 
unfolding  $alphabet\_reduce\_delta\_def$  by auto

```

— (12) δ -support past tape count — exactly the padding-read intersection guard.

```

show  $\forall q a q' a' d.$ 
   $(q, a, q', a', d) \in alphabet\_reduce\_delta M \longrightarrow$ 
   $(\forall j \geq k_M. a j = BLANK4 \wedge a' j = BLANK4 \wedge d j = dir.N)$ 
unfolding  $alphabet\_reduce\_delta\_def$  using  $M\_eq$  by auto
qed

```

qed

The reduced machine is *le-unique* unconditionally: its δ' is intersected with the support filter $\{(s, a, s', a', d). \forall k. a' k = LE4 \longrightarrow a k = LE4\}$ (the *le_unique* clause, with *le_tm* ($alphabet_reduce M$) = $LE4$), so every produced transition writes $LE4$ only where it read $LE4$ — no hypothesis on M at all. This is the output-side guarantee that makes *alphabet_reduce* a *cut-absorbing* normaliser: even a source machine that has planted a fresh *le* (cut its tape) reduces to a machine whose only $LE4$ cell is the mandatory boundary.

lemma *alphabet_reduce_le_unique*:

le_unique (*alphabet_reduce* M)

proof (*unfold le_unique_def, intro allI impI*)

fix $q a q' a' d j$

assume *mem*: $(q, a, q', a', d) \in delta_tm (alphabet_reduce M)$

and $a'le: a'j = le_tm (alphabet_reduce\ M)$
have (q, a, q', a', d)
 $\in \{(s, a, s', a', d). \forall k. a'k = LE4 \longrightarrow a k = LE4\}$
using *mem unfolding* $alphabet_reduce_delta\ alphabet_reduce_delta_def$ **by**
blast
hence $allk: \forall k. a'k = LE4 \longrightarrow a k = LE4$ **by** *simp*
from $a'le$ **have** $a'j = LE4$ **by** *simp*
with $allk$ **have** $a j = LE4$ **by** *blast*
thus $a j = le_tm (alphabet_reduce\ M)$ **by** *simp*
qed

Output well-formedness, the strong form: *alphabet_reduce* maps any valid M (with a ≥ 4 -symbol tape alphabet) to a *well-formed* machine — *valid_mttm* plus the three non-degeneracy conditions plus *le_unique*. The start-vs-halt-state distinctness is structural (the *AR_SimRead* init stage differs from the *AR_HaltAccept* / *AR_HaltReject* halt stages in its substep tag), so no $s \neq t$ hypothesis on M is needed. This is the precondition the alphabet-enlargement combinator requires of its input, so it is the bridge that lets *alphabet_reduce*'s output feed *alphabet_enlarge* in a composition.

theorem *alphabet_reduce_well_formed*:

fixes $M :: ('q, 'a)\ mttm$
assumes $valM: valid_mttm\ M$
and $cardG: card\ (\bar{\Gamma}_tm\ M) \geq 4$
shows *well_formed_mttm*
 $(alphabet_reduce\ M :: ('q \times 'a\ ar_stage, sym4)\ mttm)$

proof —

have $v: valid_mttm (alphabet_reduce\ M)$ **by** $(rule\ alphabet_reduce_wf[OF\ valM\ cardG])$
have $lu: le_unique (alphabet_reduce\ M)$ **by** $(rule\ alphabet_reduce_le_unique)$
have $st: s_tm (alphabet_reduce\ M) \neq t_tm (alphabet_reduce\ M)$
by $(simp\ add: ar_init_stage_def\ ar_accept_stage_def)$
have $sr: s_tm (alphabet_reduce\ M) \neq r_tm (alphabet_reduce\ M)$
by $(simp\ add: ar_init_stage_def\ ar_reject_stage_def)$
have $lebl: le_tm (alphabet_reduce\ M) \neq bl_tm (alphabet_reduce\ M)$
by *simp*
from $v\ st\ sr\ lebl\ lu$ **show** *?thesis* **by** *blast*
qed

Initial-configuration accessors, in terms of the machine's structural projections. Generic (any M); they let the init-correspondence proofs read the start tape / state / heads without an inline *cases* M .

lemma *mt_tape_init_config*:

$mt_tape (init_config_mttm\ M\ w)$
 $= (\lambda k\ n. if\ k < k_tm\ M$
 $\quad then\ (if\ n = 0\ then\ le_tm\ M$
 $\quad \quad else\ if\ k = 0 \wedge n \leq length\ w\ then\ w\ !\ (n - 1)$
 $\quad \quad else\ bl_tm\ M)$
 $\quad else\ bl_tm\ M)$

```

proof (cases M)
  case (MTTM Q Sg Gm bl le dl s tt rr kk)
  then have lhs: mt_tape (init_config_mttm M w)
    = ( $\lambda k\ n.$  if  $k < kk$ 
      then (if  $n = 0$  then le
            else if  $k = 0 \wedge n \leq \text{length } w$  then  $w ! (n - 1)$ 
            else bl)
      else bl)
    and le': le_tm M = le and bl': bl_tm M = bl and kk': k_tm M = kk
  by simp_all
  show ?thesis by (simp only: lhs le' bl' kk')
qed

```

```

lemma mt_state_init_config:
  mt_state (init_config_mttm M w) = s_tm M
proof (cases M)
  case (MTTM Q Sg Gm bl le dl s tt rr kk)
  then have lhs: mt_state (init_config_mttm M w) = s and s': s_tm M = s
  by simp_all
  show ?thesis by (simp only: lhs s')
qed

```

```

lemma mt_pos_init_config:
  mt_pos (init_config_mttm M w) = ( $\lambda\_.$  0)
by (cases M) simp

```

Specialised initial tape of the reduced machine: LE_4 at position 0, the (already-encoded) input w' on tape 0, and $BLANK_4$ everywhere else. Stated with w' free and an M -free right-hand side, so it closes by the same *cases M/alphabet_reduce_def* route as the structural projections — sidestepping selector-reduction on $\Gamma_tm\ M$ etc.

```

lemma mt_tape_init_config_ar:
  mt_tape (init_config_mttm (alphabet_reduce M) w')
    = ( $\lambda k\ n.$  if  $k < k\_tm\ M$ 
      then (if  $n = 0$  then  $LE_4$ 
            else if  $k = 0 \wedge n \leq \text{length } w'$  then  $w' ! (n - 1)$ 
            else  $BLANK_4$ )
      else  $BLANK_4$ )
  by (cases M) (simp add: alphabet_reduce_def)

```

Initial-tape correspondence: at the start configuration the encoded input *encode_input_ar* $\Gamma\ bl\ w$ lays out, block by block, exactly as *cell_repr* demands of M 's initial tape. The proper region (M -position $1 \leq p \leq |w|$) reads off *nth_encode_input_ar*; the blank tail ($p > |w|$) and the non-input tapes ($k \neq 0$) both reduce to the all- $BLANK_4$ block *cell_repr* $\Gamma\ bl\ bl$.

```

lemma ar_tape_correspondence_init:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M

```



```

    and wS: set w ⊆ Sigma_tm M
    and kK: k < k_tm M
  shows ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
    (mt_tape (init_config_mttm M w) k)
    (mt_tape (init_config_mttm (alphabet_reduce M)
      (encode_input_ar (Γ_tm M) (bl_tm M) w)) k)
  (is ar_tape_correspondence ?G ?le ?bl ?tM ?tM')
proof -
  let ?K = block_width ?G
  let ?w' = encode_input_ar ?G ?bl w
  have len_w': length ?w' = ?K * length w by (rule length_encode_input_ar)
  have bl_notin: ?bl ∉ Sigma_tm M using vM by (cases M) auto
  have tM_eq: ?tM = (λn. if n = 0 then ?le
    else if k = 0 ∧ n ≤ length w then w ! (n - 1)
    else ?bl)
  by (simp add: mt_tape_init_config kK)
  have tM'_eq: ?tM' = (λn. if n = 0 then LE4
    else if k = 0 ∧ n ≤ length ?w' then ?w' ! (n - 1)
    else BLANK4)
  by (simp add: mt_tape_init_config_ar kK)
  have blank_repr: cell_repr ?G ?bl ! j = BLANK4 if j < ?K for j
  using that by (simp add: cell_repr_def)
  show ?thesis
  unfolding ar_tape_correspondence_def
  proof (intro conjI)
    show ?tM 0 = ?le by (simp add: tM_eq)
  next
    show ?tM' 0 = LE4 by (simp add: tM'_eq)
  next
    show ∀ p. 1 ≤ p ⟶ (∀ j. j < ?K ⟶
      ?tM' (sim_pos ?K p + j) = cell_repr ?G ?bl (?tM p) ! j)
  proof (intro allI impI)
    fix p j :: nat
    assume p1: 1 ≤ p and jK: j < ?K
    obtain p0 where p_eq: p = Suc p0 using p1 by (cases p) auto
    have sp: sim_pos ?K p = p0 * ?K + 1
    by (simp add: p_eq sim_pos_def)
    have q_ge1: 1 ≤ sim_pos ?K p + j using sp by simp
    have sppos: 0 < sim_pos ?K p using sp by simp
    show ?tM' (sim_pos ?K p + j) = cell_repr ?G ?bl (?tM p) ! j
    proof (cases k = 0)
      case False
      have ?tM p = ?bl using False p1 by (simp add: tM_eq)
      moreover have ?tM' (sim_pos ?K p + j) = BLANK4
      using False sppos by (simp add: tM'_eq)
      ultimately show ?thesis using blank_repr[OF jK] by simp
    next
      case True
      note k0 = True

```

```

show ?thesis
proof (cases  $p \leq \text{length } w$ )
  case True
    note  $pw = True$ 
    have  $p0w$ :  $p0 < \text{length } w$  using  $pw \ p\_eq$  by simp
    have  $neg$ :  $w \ ! \ p0 \neq ?bl$ 
    proof -
      have  $w \ ! \ p0 \in \text{set } w$  using  $p0w$  by simp
      hence  $w \ ! \ p0 \in \text{Sigma\_tm } M$  using  $wS$  by blast
      thus ?thesis using  $bl\_notin$  by auto
    qed
    have  $gle$ :  $\text{sim\_pos } ?K \ p + j \leq \text{length } ?w'$ 
    proof -
      have  $\text{sim\_pos } ?K \ p + j = p0 * ?K + 1 + j$  using  $sp$  by simp
      also have  $\dots \leq p0 * ?K + ?K$  using  $jK$  by linarith
      also have  $\dots = \text{Suc } p0 * ?K$  by (simp add: mult_Suc)
      also have  $\dots \leq \text{length } w * ?K$ 
        using  $\text{Suc\_leI}[OF \ p0w]$  by (rule mult_le_mono1)
      also have  $\dots = \text{length } ?w'$  using  $\text{len\_w'}$  by (simp add: mult.commute)
      finally show ?thesis .
    qed
    have  $lhs$ :  $?tM' (\text{sim\_pos } ?K \ p + j) = \text{encode\_symbol } ?G \ ?bl \ (w \ ! \ p0) \ ! \ j$ 
    proof -
      have  $idx$ :  $\text{sim\_pos } ?K \ p + j - 1 = p0 * ?K + j$  using  $sp$  by simp
      have  $?tM' (\text{sim\_pos } ?K \ p + j) = ?w' \ ! \ (\text{sim\_pos } ?K \ p + j - 1)$ 
        unfolding  $tM'\_eq$  using  $k0 \ sposs \ gle$  by simp
      also have  $\dots = ?w' \ ! \ (p0 * ?K + j)$  using  $idx$  by simp
      also have  $\dots = \text{encode\_symbol } ?G \ ?bl \ (w \ ! \ p0) \ ! \ j$ 
        using  $p0w \ jK$  by (simp add: nth_encode_input_ar)
      finally show ?thesis .
    qed
    have  $rhs$ :  $\text{cell\_repr } ?G \ ?bl \ (?tM \ p) \ ! \ j = \text{encode\_symbol } ?G \ ?bl \ (w \ ! \ p0)$ 
    ! j
    proof -
      have  $?tM \ p = w \ ! \ p0$  unfolding  $tM\_eq$  using  $k0 \ pw \ p\_eq$  by simp
      thus ?thesis using  $neg$  by (simp add: cell_repr_def)
    qed
    show ?thesis using  $lhs \ rhs$  by simp
  next
  case False
    note  $pw = False$ 
    have  $lwle$ :  $\text{length } w \leq p0$  using  $pw \ p\_eq$  by simp
    have  $?tM \ p = ?bl$  unfolding  $tM\_eq$  using  $k0 \ pw \ p\_eq$  by simp
    hence  $rhs$ :  $\text{cell\_repr } ?G \ ?bl \ (?tM \ p) \ ! \ j = \text{BLANK}_4$ 
      using  $\text{blank\_repr}[OF \ jK]$  by simp
    have  $qgt$ :  $\text{length } ?w' < \text{sim\_pos } ?K \ p + j$ 
    proof -
      have  $a$ :  $\text{length } w * ?K \leq p0 * ?K$  using  $lwle$  by (rule mult_le_mono1)
      have  $b$ :  $\text{sim\_pos } ?K \ p + j = p0 * ?K + 1 + j$  using  $sp$  by simp

```

```

      have c: length ?w' = length w * ?K using len_w' by (simp add:
mult.commute)
      show ?thesis using a b c by linarith
    qed
    have ?tM' (sim_pos ?K p + j) = BLANK4
      unfolding tM'_eq using k0 sposs qgt by simp
    thus ?thesis using rhs by simp
  qed
qed
qed
qed
qed

```

The three simulation invariants at the start configuration — the entry premises the chunked engine *ar_simulation_phase_chunked* consumes. *ar_simulates_init* carries the weight (its *AR_SimRead* arm needs $s \neq t$, $s \neq r$ from *valid_mttm*, and its tape clause is *ar_tape_correspondence_init*); the other two read straight off *ar_init_stage*.

```

lemma ar_simulates_init:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
    and wS: set w  $\subseteq$  Sigma_tm M
    and snt: s_tm M  $\neq$  t_tm M
    and snr: s_tm M  $\neq$  r_tm M
  shows ar_simulates M (init_config_mttm M w)
    (init_config_mttm (alphabet_reduce M)
      (encode_input_ar (Γ_tm M) (bl_tm M) w))
proof -
  let ?cM = init_config_mttm M w
  let ?c' = init_config_mttm (alphabet_reduce M)
    (encode_input_ar (Γ_tm M) (bl_tm M) w)
  have st: mt_state ?c' = (s_tm M, ar_init_stage (bl_tm M))
    by (simp add: mt_state_init_config)
  have stM: mt_state ?cM = s_tm M by (simp add: mt_state_init_config)
  have tc:  $\forall k < k\_tm M. ar\_tape\_correspondence (\Gamma\_tm M) (l\_tm M) (bl\_tm M)$ 
    (mt_tape ?cM k) (mt_tape ?c' k)
    using ar_tape_correspondence_init[OF vM wS] by blast
  have posM: mt_pos ?cM = ( $\lambda\_ . 0$ ) and pos': mt_pos ?c' = ( $\lambda\_ . 0$ )
    by (simp_all add: mt_pos_init_config)
  show ?thesis
    unfolding ar_simulates_def Let_def
    by (simp add: st stM tc posM pos' ar_init_stage_def sim_pos_def snt snr)
qed

```

```

lemma ar_posk_consistent_init:
  shows ar_posk_consistent M (init_config_mttm M w)
    (init_config_mttm (alphabet_reduce M)
      (encode_input_ar (Γ_tm M) (bl_tm M) w))

```

```

unfolding ar_posk_consistent_def
by (simp add: mt_state_init_config mt_pos_init_config ar_init_stage_def)

lemma ar_at_read_boundary_init:
  assumes vM: valid_mttm M
  shows ar_at_read_boundary M
    (init_config_mttm (alphabet_reduce M)
      (encode_input_ar (Γ_tm M) (bl_tm M) w))
proof -
  let ?M' = alphabet_reduce M
  let ?c' = init_config_mttm ?M' (encode_input_ar (Γ_tm M) (bl_tm M) w)
  have kpos: 0 < k_tm M using vM by (cases M) auto
  — The padding tapes  $j \geq k\_tm\ M$  of the reduced machine's initial config read
  its blank BLANK4 (else-branch of init_config_mttm;  $k\_tm\ ?M' = k\_tm\ M$ ).
  have pad:  $\forall j \geq k\_tm\ M. mt\_tape\ ?c'\ j\ (mt\_pos\ ?c'\ j) = BLANK_4$ 
    by (cases M) (auto simp: alphabet_reduce_def)
  show ?thesis
    unfolding ar_at_read_boundary_def
    using pad kpos
    by (simp add: mt_state_init_config ar_init_stage_def
      ar_stage_bounded_def)
qed

```

The encoded input is a *BIT0*/*BIT1* string, unconditionally: every block is *encode_symbol*'s image, whose cells lie in $\{\text{BIT0}, \text{BIT1}\}$ by *encode_symbol_cell_domain* — so this needs no $set\ w \subseteq \Sigma$ hypothesis. It discharges the set-containment side of *Lang_mttm* (*alphabet_reduce M*) membership.

```

lemma set_encode_input_ar:
  set (encode_input_ar Γ bl w) ⊆ {BIT0, BIT1}
proof (induct w)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  have encode_input_ar Γ bl (x # xs)
    = encode_symbol Γ bl x @ encode_input_ar Γ bl xs
    by (simp add: encode_input_ar_def)
  thus ?case using Cons.hypos encode_symbol_cell_domain[of Γ bl x] by auto
qed

```

Forward language preservation: an accepting *M*-run on *w* lifts to an accepting *M'*-run on the encoded input, via the chunked engine off the initial correspondence. This is the forward half of *alphabet_reduce_language*; it mirrors *alphabet_enlarge_language_forward*. The reverse half, and the full biconditional *alphabet_reduce_language*, live in *AlphabetReduction_Reverse.thy* (which imports this theory), supplied by the chunked-reverse engine *ar_simulation_phase_chunked_reverse*.

```

theorem alphabet_reduce_language_forward:

```

```

fixes  $M :: ('q, 'a) \text{ mttm}$ 
assumes  $vM: \text{valid\_mttm } M$ 
  and  $s\_neq\_t: s\_tm \ M \neq t\_tm \ M$ 
  and  $s\_neq\_r: s\_tm \ M \neq r\_tm \ M$ 
  and  $le\_neq\_bl: le\_tm \ M \neq bl\_tm \ M$ 
  and  $card\_ge: card \ (\Gamma\_tm \ M) \geq 4$ 
shows  $\forall w. \text{set } w \subseteq \text{Sigma\_tm } M \longrightarrow w \in \text{Lang\_mttm } M$ 
   $\longrightarrow \text{encode\_input\_ar } (\Gamma\_tm \ M) \ (bl\_tm \ M) \ w$ 
   $\in \text{Lang\_mttm } (\text{alphabet\_reduce } M$ 
     $:: ('q \times 'a \text{ ar\_stage}, \text{sym4}) \text{ mttm})$ 
proof –
  let  $?M' = \text{alphabet\_reduce } M :: ('q \times 'a \text{ ar\_stage}, \text{sym4}) \text{ mttm}$ 
  show  $?thesis$ 
  proof (intro allI impI)
    fix  $w :: 'a \text{ list}$ 
    assume  $w\_sub: \text{set } w \subseteq \text{Sigma\_tm } M$ 
    assume  $w\_in: w \in \text{Lang\_mttm } M$ 
    from  $w\_in$  obtain  $w' \ nw$  where
       $\text{run\_star}: (\text{init\_config\_mttm } M \ w, \text{Config}_M \ (t\_tm \ M) \ w' \ nw)$ 
       $\in (\text{mttm\_step } (\text{delta\_tm } M))^*$ 
    unfolding  $\text{Lang\_mttm\_def}$  by blast
    then obtain  $nn$  where
       $\text{trace}: (\text{init\_config\_mttm } M \ w, \text{Config}_M \ (t\_tm \ M) \ w' \ nw)$ 
       $\in (\text{mttm\_step } (\text{delta\_tm } M)) \ \wedge \wedge \ nn$ 
    using rtrancl_imp_relpow by blast
    let  $?c' = \text{init\_config\_mttm } ?M' \ (\text{encode\_input\_ar } (\Gamma\_tm \ M) \ (bl\_tm \ M) \ w)$ 
    have  $\text{reach0}: (\text{init\_config\_mttm } M \ w, \text{init\_config\_mttm } M \ w)$ 
       $\in (\text{mttm\_step } (\text{delta\_tm } M))^* \text{ by } \text{blast}$ 
    have  $\text{accf}: \text{mt\_state } (\text{Config}_M \ (t\_tm \ M) \ w' \ nw) = t\_tm \ M \text{ by } \text{simp}$ 
    have  $\text{sim}: \text{ar\_simulates } M \ (\text{init\_config\_mttm } M \ w) \ ?c'$ 
      by (rule ar_simulates_init [OF  $vM \ w\_sub \ s\_neq\_t \ s\_neq\_r$ ])
    have  $\text{posk}: \text{ar\_posk\_consistent } M \ (\text{init\_config\_mttm } M \ w) \ ?c'$ 
      by (rule ar_posk_consistent_init)
    have  $\text{rbnd}: \text{ar\_at\_read\_boundary } M \ ?c'$ 
      by (rule ar_at_read_boundary_init [OF  $vM$ ])
    obtain  $m \ c''$  where
       $\text{run}: (?c', c'') \in (\text{mttm\_step } (\text{alphabet\_reduce\_delta } M)) \ \wedge \wedge \ m$ 
      and  $\text{st}'': \text{mt\_state } c'' = (t\_tm \ M, \text{ar\_accept\_stage } (bl\_tm \ M))$ 
    using ar_simulation_phase_chunked
      [OF  $vM \ w\_sub \ card\_ge \ \text{reach0} \ \text{trace} \ \text{accf} \ \text{sim} \ \text{posk} \ \text{rbnd} \ le\_neq\_bl$ ]
    by blast
    have  $\text{st2}: \text{mt\_state } c'' = t\_tm \ ?M' \text{ using } \text{st}'' \text{ by } \text{simp}$ 
    obtain  $wM' \ nM'$  where  $c''\_eq: c'' = \text{Config}_M \ (t\_tm \ ?M') \ wM' \ nM'$ 
    using  $\text{st2}$  by (cases  $c''$ ) simp
    have  $\text{run}' : (?c', c'') \in (\text{mttm\_step } (\text{delta\_tm } ?M')) \ \wedge \wedge \ m$ 
      using  $\text{run}$  by simp
    have  $\text{run\_star}' : (?c', \text{Config}_M \ (t\_tm \ ?M') \ wM' \ nM')$ 
       $\in (\text{mttm\_step } (\text{delta\_tm } ?M'))^*$ 
    using  $\text{run}' \ c''\_eq \ \text{relpow\_imp\_rtrancl}$  by metis

```

```

have enc_sub: set (encode_input_ar ( $\Gamma_{tm} M$ ) (bl_tm M) w)
   $\subseteq$  Sigma_tm ?M'
using set_encode_input_ar by simp
show encode_input_ar ( $\Gamma_{tm} M$ ) (bl_tm M) w  $\in$  Lang_mttm ?M'
unfolding Lang_mttm_def using enc_sub run_star' by blast
qed
qed

```

The per- M -step super-step count $C \cdot (5 \cdot k + 2) + 2$ ($C = k_{tm} M$, $k = b = \text{block_width}$) folds into one b -factor $(6 \cdot C + 1) \cdot k$. The slack is $(C + 1) \cdot (k - 2) \geq 0$, tight at $k = 2$ — so $6 \cdot C + 1$ is the least factor that absorbs the count for every $k \geq 2$. The reduction always encodes into blocks of width ≥ 2 (from $\text{card } \Gamma_{tm} M \geq 4$), so the $b \geq 2$ precondition is always met; the looser $b \geq 1$ fold to $7 \cdot C + 2$ is unnecessary.

```

lemma ar_time_bound_arith_sharp:
  fixes C k :: nat
  assumes  $2 \leq k$ 
  shows  $C * (5 * k + 2) + 2 \leq (6 * C + 1) * k$ 
proof —
  obtain k0 where k:  $k = 2 + k0$  using le_Suc_ex[OF assms] by blast
  have  $(6 * C + 1) * k = C * (5 * k + 2) + 2 + (C + 1) * k0$ 
    by (simp add: k algebra_simps)
  thus ?thesis by linarith
qed

```

Time bound under weak time-bounded acceptance, with explicit constants. If M accepts w within $T(|w|)$ steps, the output machine accepts the encoded input within $(6 \cdot k_{tm} M + 1) \cdot b \cdot T(|w|)$ steps, where $b = \text{block_width } (\Gamma_{tm} M)$ is the per-symbol block length (in *sym4* cells) and $k_{tm} M$ is the tape count. The exact per- M -step super-step count is $k_{tm} M \cdot (5 \cdot b + 2) + 2$, folded up to $(6 \cdot k_{tm} M + 1) \cdot b$ using $b \geq 2$ (*ar_time_bound_arith_sharp*; $b \geq 2$ holds because $\text{card } (\Gamma_{tm} M) \geq 4$, and the factor is tight at $b = 2$, i.e. $\text{card } (\Gamma_{tm} M) = 4$); the linear-in- $|w|$ coefficient and the additive constant are both 0. b is logarithmic in the source-alphabet cardinality $\text{card } (\Gamma_{tm} M)$ (the binary-encoding design point) — a *ceiling* log, so the slowdown factor is a step function of $\text{card } (\Gamma_{tm} M)$ that jumps by one at each power of two (the band $2^{b-1} < \text{card } (\Gamma_{tm} M) \leq 2^b$ is pinned by *card_le_two_pow_block_width* and *two_pow_block_width_pred_less_card*). The count is the whole alphabet, blank included at index 0 (the endmarker *le* is a genuinely coded symbol, so it too is counted); so b is the uniform-code width, one cell above the coding minimum $\lceil \log_2 (\text{card } (\Gamma_{tm} M) - 1) \rceil$ just past a power of two — see *block_width*. Weak acceptance, not the strong all-paths *upperb_time_mttm*, is the target: the strong bound is unprovable for the no-validation construction, and weak acceptance serves both the DTM and NDTM applications. Mirrors *alphabet_enlarge_time_explicit*. The classical existential form is the *obtains*-corollary *alphabet_reduce_time* below.

```

theorem alphabet_reduce_time_explicit:
  fixes  $M :: ('q, 'a) \text{ mttm}$ 
    and  $T :: \text{nat} \Rightarrow \text{nat}$ 
  assumes  $vM: \text{valid\_mttm } M$ 
    and  $s\_neq\_t: s\_tm\ M \neq t\_tm\ M$ 
    and  $s\_neq\_r: s\_tm\ M \neq r\_tm\ M$ 
    and  $le\_neq\_bl: le\_tm\ M \neq bl\_tm\ M$ 
    and  $card\_ge: card\ (\Gamma\_tm\ M) \geq 4$ 
  shows  $\forall w. \text{set } w \subseteq \text{Sigma\_tm } M$ 
     $\longrightarrow \text{accepts\_in\_time\_mttm } M\ w\ (T\ (\text{length } w))$ 
     $\longrightarrow \text{accepts\_in\_time\_mttm}$ 
       $(\text{alphabet\_reduce } M$ 
         $:: ('q \times 'a\ \text{ar\_stage}, \text{sym4}) \text{ mttm})$ 
         $(\text{encode\_input\_ar } (\Gamma\_tm\ M)\ (bl\_tm\ M)\ w)$ 
         $((6 * k\_tm\ M + 1) * \text{block\_width } (\Gamma\_tm\ M) * T\ (\text{length } w))$ 
  proof -
    let  $?C = k\_tm\ M$ 
    let  $?k = \text{block\_width } (\Gamma\_tm\ M)$ 
    let  $?M' = \text{alphabet\_reduce } M :: ('q \times 'a\ \text{ar\_stage}, \text{sym4}) \text{ mttm}$ 
    let  $?d = 6 * ?C + 1$ 
    — encoding length  $\geq 2$  from  $card\ \Gamma\_M \geq 4$ , as in the walker
    have  $kge2: 2 \leq ?k$ 
    proof -
      have  $(2::\text{nat}) \wedge 2 \leq 2 \wedge ?k$ 
      using  $card\_ge\ card\_le\_two\_pow\_block\_width[of\ \Gamma\_tm\ M]$  by simp
      thus  $2 \leq ?k$  using  $power\_le\_imp\_le\_exp[of\ 2::\text{nat } 2\ ?k]$  by simp
    qed
    have  $bnd: ?C * (5 * ?k + 2) + 2 \leq ?d * ?k$ 
    by  $(\text{rule } ar\_time\_bound\_arith\_sharp[OF\ kge2])$ 
    show  $\forall w. \text{set } w \subseteq \text{Sigma\_tm } M$ 
       $\longrightarrow \text{accepts\_in\_time\_mttm } M\ w\ (T\ (\text{length } w))$ 
       $\longrightarrow \text{accepts\_in\_time\_mttm } ?M'$ 
         $(\text{encode\_input\_ar } (\Gamma\_tm\ M)\ (bl\_tm\ M)\ w)$ 
         $(?d * ?k * T\ (\text{length } w))$ 
    proof  $(\text{intro allI impI})$ 
      fix  $w :: 'a\ \text{list}$ 
      assume  $w\_sub: \text{set } w \subseteq \text{Sigma\_tm } M$ 
      assume  $m\_acc: \text{accepts\_in\_time\_mttm } M\ w\ (T\ (\text{length } w))$ 
      from  $m\_acc$  obtain  $n\_T\ cMf$  where
         $nT\_le: n\_T \leq T\ (\text{length } w)$ 
        and  $\text{trace}: (\text{init\_config\_mttm } M\ w, cMf)$ 
           $\in (\text{mttm\_step } (\text{delta\_tm } M))^{n\_T}$ 
        and  $\text{accf}: \text{mt\_state } cMf = t\_tm\ M$ 
        unfolding  $\text{accepts\_in\_time\_mttm\_def}$  by blast
      let  $?c' = \text{init\_config\_mttm } ?M'\ (\text{encode\_input\_ar } (\Gamma\_tm\ M)\ (bl\_tm\ M)\ w)$ 
      have  $\text{reach0}: (\text{init\_config\_mttm } M\ w, \text{init\_config\_mttm } M\ w)$ 
         $\in (\text{mttm\_step } (\text{delta\_tm } M))^*$  by blast
      have  $\text{sim}: ar\_simulates\ M\ (\text{init\_config\_mttm } M\ w)\ ?c'$ 

```

```

    by (rule ar_simulates_init[OF vM w_sub s_neq_t s_neq_r])
  have posk: ar_posk_consistent M (init_config_mttm M w) ?c'
    by (rule ar_posk_consistent_init)
  have rbnd: ar_at_read_boundary M ?c'
    by (rule ar_at_read_boundary_init[OF vM])
  obtain m c'' where
    m_le: m ≤ (?C * (5 * ?k + 2) + 2) * n_T
    and run: (?c', c'') ∈ (mttm_step (alphabet_reduce_delta M)) ^^ m
    and st'': mt_state c'' = (t_tm M, ar_accept_stage (bl_tm M))
    using ar_simulation_phase_chunked
    [OF vM w_sub card_ge reach0 trace accf sim posk rbnd le_neq_bl]
    by blast
  have m_bound: m ≤ ?d * ?k * T (length w)
  proof -
    have m ≤ (?C * (5 * ?k + 2) + 2) * n_T by (rule m_le)
    also have ... ≤ (?d * ?k) * n_T using bnd by (rule mult_le_mono1)
    also have ... ≤ (?d * ?k) * T (length w)
      using nT_le by (rule mult_le_mono2)
    finally show ?thesis by (simp add: mult.assoc)
  qed
  have run': (?c', c'') ∈ (mttm_step (delta_tm ?M')) ^^ m
    using run by simp
  have st''': mt_state c'' = t_tm ?M' using st'' by simp
  show accepts_in_time_mttm ?M'
    (encode_input_ar (Γ_tm M) (bl_tm M) w)
    (?d * ?k * T (length w))
    unfolding accepts_in_time_mttm_def
    using m_bound run' st''' by blast
qed
qed

```

Linear-time slowdown, classical existential form: reducing the tape alphabet to the fixed type *sym4* multiplies the running time by only a constant times the per-symbol block width. The *obtains*-corollary of *alphabet_reduce_time_explicit* above, hiding its four explicit constants behind existentials instantiated at $d = 6 \cdot k_{tm} M + 1$, $e = f = 0$, and $b = block_width (\Gamma_{tm} M)$.

```

theorem alphabet_reduce_time:
  fixes M :: ('q, 'a) mttm
  and T :: nat ⇒ nat
  assumes vM: valid_mttm M
  and s_neq_t: s_tm M ≠ t_tm M
  and s_neq_r: s_tm M ≠ r_tm M
  and le_neq_bl: le_tm M ≠ bl_tm M
  and card_ge: card (Γ_tm M) ≥ 4
  obtains d e f b :: nat
  where b ≥ 1
  and ∀ w. set w ⊆ Sigma_tm M
    → accepts_in_time_mttm M w (T (length w))

```



```

      → accepts_in_time_mttm
        (alphabet_reduce M
          :: ('q × 'a ar_stage, sym4) mttm)
        (encode_input_ar (Γ_tm M) (bl_tm M) w)
        (d * b * T (length w) + e * b * length w + f)
proof –
  have P: ∀ w. set w ⊆ Sigma_tm M
    → accepts_in_time_mttm M w (T (length w))
    → accepts_in_time_mttm
      (alphabet_reduce M
        :: ('q × 'a ar_stage, sym4) mttm)
      (encode_input_ar (Γ_tm M) (bl_tm M) w)
      ((6 * k_tm M + 1) * block_width (Γ_tm M) * T (length w)
        + 0 * block_width (Γ_tm M) * length w + 0)
    using alphabet_reduce_time_explicit[OF vM s_neq_t s_neq_r le_neq_bl
card_ge]
    by simp
  show ?thesis
proof (rule that)
    show (block_width (Γ_tm M) :: nat) ≥ 1 by (rule block_width_pos)
  next
    show ∀ w. set w ⊆ Sigma_tm M
      → accepts_in_time_mttm M w (T (length w))
      → accepts_in_time_mttm
        (alphabet_reduce M
          :: ('q × 'a ar_stage, sym4) mttm)
        (encode_input_ar (Γ_tm M) (bl_tm M) w)
        ((6 * k_tm M + 1) * block_width (Γ_tm M) * T (length w)
          + 0 * block_width (Γ_tm M) * length w + 0)
      by (rule P)
  qed
qed

```

Output alphabet: the reduced machine’s tape alphabet is the whole finite type *sym4*, of cardinality exactly four.

theorem *alphabet_reduce_produces_alphabet_size_4*:
shows *card (UNIV :: sym4 set) = 4*
by (rule *sym4_card*)

Tape-count preservation: the reduction re-encodes the tape alphabet symbol by symbol without changing the number of tapes, so the reduced machine has exactly *M*’s tape count. This is the formal counterpart of the headline “tape count preserved”, and the point of contrast with a *tape-count* reduction such as Book–Greibach–Wegbreit.

theorem *alphabet_reduce_preserves_tape_count*:
fixes *M* :: ('q, 'a) mttm
shows *k_tm (alphabet_reduce M :: ('q × 'a ar_stage, sym4) mttm)*
 = *k_tm M*
by (cases *M*) (simp add: *alphabet_reduce_def*)

```

end
theory AlphabetReduction_Reverse
  imports AlphabetReduction_Theorems
begin

```

3 Alphabet reduction: reverse language inclusion

The reverse leg of *alphabet_reduce_language: encode_input_ar* ($\Gamma_tm\ M$) ($bl_tm\ M$) $w \in Lang_mttm\ (alphabet_reduce\ M) \implies w \in Lang_mttm\ M$, the converse of the proven *alphabet_reduce_language_forward*.

Unlike AE's reverse arm, which restricts to *det_mttm* M and closes via chain uniqueness (Path C), AR proves this unconditionally — deterministic and nondeterministic M alike. The counterexample probe is negative: AR's nondeterminism gateway *ar_delta_compute* is a direct single-step embedding of *delta_tm* M (one substrate tuple per M -tuple, no AE-style *m_steps_buffered* slack), so every accepting M' -path decodes branch-by-branch to a genuine accepting M -path.

Strategy (Strategy B, direct backward inversion): a per-cycle backward step lemma reads the M -transition straight off the compute tuple on the given M' -path and reuses the forward arm's invariants *ar_simulates*, *ar_posk_consistent*, *ar_at_read_boundary* read backward; a backward chunked engine aggregates the per-cycle steps into an M -run. Throughout, b abbreviates *block_width* Γ (the per-symbol cell width).

3.1 Terminal accept configuration

Base case of the backward engine: the accept state *t_tm* (*alphabet_reduce* M) is terminal. A valid machine never steps from its accept state (substrate *mttm_step_src_neq_t*), and *alphabet_reduce* M is valid by *alphabet_reduce_wf*; its accept state is (*t_tm* M , *ar_accept_stage* (*bl_tm* M)) by *alphabet_reduce_accept*.

Itself currently uncalled: the backward engine kills reject boundaries mid-trace via *ar_reject_terminal*, not accept ones; retained as the documented half of the accept/reject terminal pair.

```

lemma ar_accept_terminal:
  fixes M :: ('q, 'a) mttm
  and c c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes vM:      valid_mttm M
  and card_ge: card (Γ_tm M) ≥ 4
  and step:      (c, c') ∈ mttm_step (alphabet_reduce_delta M)
  shows mt_state c ≠ (t_tm M, ar_accept_stage (bl_tm M))
proof -
  let ?M' = alphabet_reduce M :: ('q × 'a ar_stage, sym4) mttm
  have valM': valid_mttm ?M' by (rule alphabet_reduce_wf[OF vM card_ge])

```

have $step'$: $(c, c') \in mttm_step\ (delta_tm\ ?M')$ **using** $step$ **by** $simp$
have $mt_state\ c \neq t_tm\ ?M'$ **by** $(rule\ mttm_step_src_neq_t[OF\ valM'\ step])$
thus $?thesis$ **by** $simp$
qed

Reject companion of $ar_accept_terminal$: the reject state r_tm ($alphabet_reduce\ M$) = ($r_tm\ M$, $ar_reject_stage\ (bl_tm\ M)$) (by $alphabet_reduce_reject$) is terminal too (substrate $mttm_step_src_neq_r$). The backward engine uses it to kill a reject boundary reached mid-trace: the trace runs to the accept config, so a reject config can carry no outgoing step.

lemma $ar_reject_terminal$:

fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes vM : $valid_mttm\ M$
and $card_ge$: $card\ (\Gamma_tm\ M) \geq 4$
and $step$: $(c, c') \in mttm_step\ (alphabet_reduce_delta\ M)$
shows $mt_state\ c \neq (r_tm\ M, ar_reject_stage\ (bl_tm\ M))$

proof –

let $?M' = alphabet_reduce\ M :: ('q \times 'a\ ar_stage, sym4)\ mttm$
have $valM'$: $valid_mttm\ ?M'$ **by** $(rule\ alphabet_reduce_wf[OF\ vM\ card_ge])$
have $step'$: $(c, c') \in mttm_step\ (delta_tm\ ?M')$ **using** $step$ **by** $simp$
have $mt_state\ c \neq r_tm\ ?M'$ **by** $(rule\ mttm_step_src_neq_r[OF\ valM'\ step])$
thus $?thesis$ **by** $simp$

qed

3.2 Source substep-tags partition the substep relations

Each of the five substep relations carries a uniform source substep-tag ($fst\ (snd\ s)$ for a source state $s = (q, stg)$), and the five tags are pairwise distinct datatype constructors. This is the structural fact every step-inversion lemma rests on: a step whose source is at substep X can only come from the relation whose sources carry tag X .

lemma $ar_delta_read_src$:

$(s, a, s', a', d) \in ar_delta_read\ M \implies fst\ (snd\ s) = AR_SimRead$
by $(auto\ simp: ar_delta_read_def)$

lemma $ar_delta_compute_src$:

$(s, a, s', a', d) \in ar_delta_compute\ M \implies fst\ (snd\ s) = AR_SimCompute$
by $(auto\ simp: ar_delta_compute_def)$

lemma $ar_delta_write_src$:

$(s, a, s', a', d) \in ar_delta_write\ M \implies fst\ (snd\ s) = AR_SimWrite$
by $(auto\ simp: ar_delta_write_def)$

lemma $ar_delta_advance_src$:

$(s, a, s', a', d) \in ar_delta_advance\ M \implies fst\ (snd\ s) = AR_SimAdvance$
by $(auto\ simp: ar_delta_advance_def)$

lemma *ar_delta_next_src*:
 $(s, a, s', a', d) \in ar_delta_next\ M \implies fst\ (snd\ s) = AR_SimNext$
by (*auto simp: ar_delta_next_def*)

3.3 Destination substep-tags carve the cycle's substep order

Each substep relation's destination tag is confined to a small set: read loops to itself or transitions to compute; compute is the unique non-deterministic step and lands at write; write loops or advances; advance loops or hands off to next; next closes the cycle (back to read) or dispatches to a halt-coerced configuration. These five facts encode the substep transition graph and underpin the chain-shape arguments used in the reverse arm's pinning lemmas.

lemma *ar_delta_read_dest*:
 $(s, a, s', a', d) \in ar_delta_read\ M \implies$
 $fst\ (snd\ s') = AR_SimRead \vee fst\ (snd\ s') = AR_SimCompute$
by (*auto simp: ar_delta_read_def*)

lemma *ar_delta_compute_dest*:
 $(s, a, s', a', d) \in ar_delta_compute\ M \implies fst\ (snd\ s') = AR_SimWrite$
by (*auto simp: ar_delta_compute_def*)

lemma *ar_delta_write_dest*:
 $(s, a, s', a', d) \in ar_delta_write\ M \implies$
 $fst\ (snd\ s') = AR_SimWrite \vee fst\ (snd\ s') = AR_SimAdvance$
by (*auto simp: ar_delta_write_def*)

lemma *ar_delta_advance_dest*:
 $(s, a, s', a', d) \in ar_delta_advance\ M \implies$
 $fst\ (snd\ s') = AR_SimAdvance \vee fst\ (snd\ s') = AR_SimNext$
by (*auto simp: ar_delta_advance_def*)

lemma *ar_delta_next_dest*:
 $(s, a, s', a', d) \in ar_delta_next\ M \implies$
 $fst\ (snd\ s') = AR_SimRead \vee fst\ (snd\ s') = AR_HaltAccept$
 $\vee fst\ (snd\ s') = AR_HaltReject$
by (*auto simp: ar_delta_next_def ar_accept_stage_def ar_reject_stage_def*)

Union-disambiguation: a tuple in *alphabet_reduce_delta* *M* whose source is at *AR_SimCompute* must lie in the compute relation *ar_delta_compute* *M*. The intersection filters of *alphabet_reduce_delta* (δLE and the valid-stage guards) only shrink the union, so membership of the union is all we need; the other four source-tag lemmas rule out the other disjuncts by constructor-distinctness.

lemma *ar_delta_compute_from_src*:
assumes *mem*: $(s, a, s', a', d) \in alphabet_reduce_delta\ M$
and *src*: $fst\ (snd\ s) = AR_SimCompute$
shows $(s, a, s', a', d) \in ar_delta_compute\ M$

```

proof –
  from mem
  have u:  $(s, a, s', a', d) \in \text{ar\_delta\_read } M \cup \text{ar\_delta\_compute } M$ 
     $\cup \text{ar\_delta\_write } M \cup \text{ar\_delta\_advance } M \cup \text{ar\_delta\_next } M$ 
    unfolding alphabet_reduce_delta_def by blast
  have nr:  $(s, a, s', a', d) \notin \text{ar\_delta\_read } M$ 
  proof
    assume  $(s, a, s', a', d) \in \text{ar\_delta\_read } M$ 
    from ar_delta_read_src[OF this] src show False by simp
  qed
  have nw:  $(s, a, s', a', d) \notin \text{ar\_delta\_write } M$ 
  proof
    assume  $(s, a, s', a', d) \in \text{ar\_delta\_write } M$ 
    from ar_delta_write_src[OF this] src show False by simp
  qed
  have nad:  $(s, a, s', a', d) \notin \text{ar\_delta\_advance } M$ 
  proof
    assume  $(s, a, s', a', d) \in \text{ar\_delta\_advance } M$ 
    from ar_delta_advance_src[OF this] src show False by simp
  qed
  have nx:  $(s, a, s', a', d) \notin \text{ar\_delta\_next } M$ 
  proof
    assume  $(s, a, s', a', d) \in \text{ar\_delta\_next } M$ 
    from ar_delta_next_src[OF this] src show False by simp
  qed
  from u nr nw nad nx show ?thesis by blast
qed

```

3.4 Compute-step inversion

The keystone of the reverse arm: a single M' -step out of an *AR_SimCompute* configuration reads the simulated M -transition straight off the compute tuple. Because *ar_delta_compute* is a direct single-step embedding of *delta_tm* M , the inversion yields a genuine $(q, \text{buf}, q', m_{a'}, m_d) \in \text{delta_tm } M$ with no chain-uniqueness or determinism assumption — this is where AR's ND-generality is earned. Compute neither writes nor moves: the tape and head positions are unchanged (read symbol equals write symbol, direction N).

```

lemma ar_compute_step_inv_sub:
  fixes M ::  $('q, 'a) \text{mttm}$ 
    and c' c'' ::  $(\text{sym4}, 'q \times 'a \text{ar\_stage}) \text{mt\_config}$ 
  assumes step:  $(c', c'') \in \text{mttm\_step } (\text{ar\_delta\_compute } M)$ 
    and stg:  $\text{mt\_state } c' = (q, \text{AR\_SimCompute}, tk, i, \text{buf}, \text{dvec}, \text{posk})$ 
  obtains  $q' m_{a'} m_d$  where
     $(q, \text{buf}, q', m_{a'}, m_d) \in \text{delta\_tm } M$ 
    and  $\text{mt\_state } c'' = (q', \text{AR\_SimWrite}, 0, 0, m_{a'}, m_d, \text{posk})$ 
    and  $\text{mt\_tape } c'' = \text{mt\_tape } c'$ 
    and  $\text{mt\_pos } c'' = \text{mt\_pos } c'$ 

```

```

proof –
  from step obtain  $S \text{ ts } n \ S'' \text{ aw } \text{dir}$  where
     $c'_eq: c' = \text{Config}_M \ S \text{ ts } n$ 
    and  $c''_eq: c'' = \text{Config}_M \ S'' \ (\lambda k. (ts \ k)(n \ k := aw \ k))$ 
       $(\lambda k. go\_dir \ (\text{dir } k) \ (n \ k))$ 
    and  $rel: (S, (\lambda k. ts \ k \ (n \ k)), S'', aw, \text{dir})$ 
       $\in ar\_delta\_compute \ M$ 
    by (auto elim: mttm\_step.cases)
  have  $S\_eq: S = (q, AR\_SimCompute, tk, i, buf, dvec, posk)$ 
    using stg c'_eq by simp
  have  $crel: ((q, AR\_SimCompute, tk, i, buf, dvec, posk),$ 
     $(\lambda k. ts \ k \ (n \ k)), S'', aw, \text{dir}) \in ar\_delta\_compute \ M$ 
    using rel S_eq by simp
  from  $crel$  obtain  $q' \ m\_a' \ m\_d$  where
     $mdelta: (q, buf, q', m\_a', m\_d) \in delta\_tm \ M$ 
    and  $S''\_eq: S'' = (q', AR\_SimWrite, 0, 0, m\_a', m\_d, posk)$ 
    and  $aw\_eq: aw = (\lambda k. ts \ k \ (n \ k))$ 
    and  $dir\_eq: dir = (\lambda \_. dir.N)$ 
    unfolding ar\_delta\_compute\_def by auto
  have  $state: mt\_state \ c'' = (q', AR\_SimWrite, 0, 0, m\_a', m\_d, posk)$ 
    using  $c''\_eq \ S''\_eq$  by simp
  have  $tape: mt\_tape \ c'' = mt\_tape \ c'$ 
proof –
    have  $mt\_tape \ c'' = (\lambda k. (ts \ k)(n \ k := aw \ k))$  using  $c''\_eq$  by simp
    also have  $\dots = ts$  by (simp add: aw_eq fun_upd_triv)
    finally show ?thesis using  $c'_eq$  by simp
  qed
  have  $pos: mt\_pos \ c'' = mt\_pos \ c'$ 
proof –
    have  $mt\_pos \ c'' = (\lambda k. go\_dir \ (\text{dir } k) \ (n \ k))$  using  $c''\_eq$  by simp
    also have  $\dots = n$  by (simp add: dir_eq)
    finally show ?thesis using  $c'_eq$  by simp
  qed
  show ?thesis by (rule that[OF mdelta state tape pos])
qed

```

The union-step face of the inversion: lift the union step into the compute sub-relation (*ar_step_compute_lift*) and invert there. Used by the forward walker preservation *ar_walker_step_from_at_compute*; the reverse cycle-close inverts the walker's own sub-relation compute step directly via *ar_compute_step_inv_sub*.

```

lemma ar_compute_step_inv:
  fixes  $M :: ('q, 'a) \text{mttm}$ 
    and  $c' \ c'' :: (\text{sym4}, 'q \times 'a \text{ ar\_stage}) \text{mt\_config}$ 
  assumes step:  $(c', c'') \in \text{mttm\_step} \ (\text{alphabet\_reduce\_delta } M)$ 
    and stg:  $mt\_state \ c' = (q, AR\_SimCompute, tk, i, buf, dvec, posk)$ 
  obtains  $q' \ m\_a' \ m\_d$  where
     $(q, buf, q', m\_a', m\_d) \in delta\_tm \ M$ 
    and  $mt\_state \ c'' = (q', AR\_SimWrite, 0, 0, m\_a', m\_d, posk)$ 

```

```

    and mt_tape c'' = mt_tape c'
    and mt_pos c'' = mt_pos c'
  proof -
    from step obtain S ts n S'' aw dir where
      c'_eq: c' = Config_M S ts n
      and c''_eq: c'' = Config_M S'' (λk. (ts k)(n k := aw k))
        (λk. go_dir (dir k) (n k))
      and rel: (S, (λk. ts k (n k)), S'', aw, dir)
        ∈ alphabet_reduce_delta M
      by (auto elim: mttm_step.cases)
    have src: fst (snd S) = AR_SimCompute using stg c'_eq by simp
    have crel: (S, (λk. ts k (n k)), S'', aw, dir) ∈ ar_delta_compute M
      by (rule ar_delta_compute_from_src[OF rel src])
    have step_explicit:
      (Config_M S ts n, Config_M S'' (λk. (ts k)(n k := aw k))
        (λk. go_dir (dir k) (n k))) ∈ mttm_step (ar_delta_compute M)
      using crel by (rule mttm_step.step)
    have sub: (c', c'') ∈ mttm_step (ar_delta_compute M)
      using step_explicit c'_eq c''_eq by simp
    show thesis
  proof (rule ar_compute_step_inv_sub[OF sub stg])
    fix q' m_a' m_d
    assume (q, buf, q', m_a', m_d) ∈ delta_tm M
    and mt_state c'' = (q', AR_SimWrite, 0, 0, m_a', m_d, posk)
    and mt_tape c'' = mt_tape c'
    and mt_pos c'' = mt_pos c'
    thus thesis by (rule that)
  qed
qed

```

3.5 Next-step inversion (cycle closure / halt dispatch)

Source disambiguation for $AR_SimNext$, mirroring $ar_delta_compute_from_src$.

```

lemma ar_delta_next_from_src:
  assumes mem: (s, a, s', a', d) ∈ alphabet_reduce_delta M
  and src: fst (snd s) = AR_SimNext
  shows (s, a, s', a', d) ∈ ar_delta_next M
proof -
  from mem
  have u: (s, a, s', a', d) ∈ ar_delta_read M ∪ ar_delta_compute M
    ∪ ar_delta_write M ∪ ar_delta_advance M ∪ ar_delta_next M
    unfolding alphabet_reduce_delta_def by blast
  have nr: (s, a, s', a', d) ∉ ar_delta_read M
  proof
    assume (s, a, s', a', d) ∈ ar_delta_read M
    from ar_delta_read_src[OF this] src show False by simp
  qed
  have nc: (s, a, s', a', d) ∉ ar_delta_compute M

```

```

proof
  assume  $(s, a, s', a', d) \in ar\_delta\_compute\ M$ 
  from  $ar\_delta\_compute\_src[OF\ this]\ src$  show False by simp
qed
have  $nw: (s, a, s', a', d) \notin ar\_delta\_write\ M$ 
proof
  assume  $(s, a, s', a', d) \in ar\_delta\_write\ M$ 
  from  $ar\_delta\_write\_src[OF\ this]\ src$  show False by simp
qed
have  $nad: (s, a, s', a', d) \notin ar\_delta\_advance\ M$ 
proof
  assume  $(s, a, s', a', d) \in ar\_delta\_advance\ M$ 
  from  $ar\_delta\_advance\_src[OF\ this]\ src$  show False by simp
qed
from  $u\ nr\ nc\ nw\ nad$  show ?thesis by blast
qed

```

A single M' -step out of an $AR_SimNext$ configuration neither writes nor moves, and dispatches on the simulated M -state q : continue to the next $AR_SimRead$ boundary when q is non-halting, or land in the accept/reject halt stage when q is M 's accept/reject state. The accept landing is exactly t_tm ($alphabet_reduce\ M$).

```

lemma  $ar\_next\_step\_inv$ :
  fixes  $M :: ('q, 'a)\ mttm$ 
  and  $c'\ c'' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config$ 
  assumes  $step: (c', c'') \in mttm\_step\ (alphabet\_reduce\_delta\ M)$ 
  and  $stg: mt\_state\ c' = (q, AR\_SimNext, tk, i, buf, dvec, posk)$ 
  shows  $mt\_tape\ c'' = mt\_tape\ c' \wedge mt\_pos\ c'' = mt\_pos\ c'$ 
   $\wedge ((q \notin \{t\_tm\ M, r\_tm\ M\}$ 
     $\wedge mt\_state\ c'' = (q, AR\_SimRead, 0, 0, buf, dvec, posk))$ 
     $\vee (q = t\_tm\ M$ 
     $\wedge mt\_state\ c'' = (t\_tm\ M, ar\_accept\_stage\ (bl\_tm\ M)))$ 
     $\vee (q = r\_tm\ M$ 
     $\wedge mt\_state\ c'' = (r\_tm\ M, ar\_reject\_stage\ (bl\_tm\ M))))$ 
proof –
  from  $step$  obtain  $S\ ts\ n\ S''\ aw\ dir$  where
     $c'\_eq: c' = Config_M\ S\ ts\ n$ 
    and  $c''\_eq: c'' = Config_M\ S''\ (\lambda k. (ts\ k)(n\ k := aw\ k))$ 
     $(\lambda k. go\_dir\ (dir\ k)\ (n\ k))$ 
    and  $rel: (S, (\lambda k. ts\ k\ (n\ k)), S'', aw, dir)$ 
     $\in alphabet\_reduce\_delta\ M$ 
    by  $(auto\ elim: mttm\_step.cases)$ 
  have  $S\_eq: S = (q, AR\_SimNext, tk, i, buf, dvec, posk)$ 
    using  $stg\ c'\_eq$  by simp
  have  $src: fst\ (snd\ S) = AR\_SimNext$  using  $S\_eq$  by simp
  have  $nrel: ((q, AR\_SimNext, tk, i, buf, dvec, posk),$ 
     $(\lambda k. ts\ k\ (n\ k)), S'', aw, dir) \in ar\_delta\_next\ M$ 
    using  $ar\_delta\_next\_from\_src[OF\ rel\ src]\ S\_eq$  by simp
  have  $core: aw = (\lambda k. ts\ k\ (n\ k)) \wedge dir = (\lambda\_ . dir.N)$ 

```



```

    ∧ ((q ∉ {t_tm M, r_tm M}
      ∧ S'' = (q, AR_SimRead, 0, 0, buf, dvec, posk))
      ∨ (q = t_tm M ∧ S'' = (t_tm M, ar_accept_stage (bl_tm M)))
      ∨ (q = r_tm M ∧ S'' = (r_tm M, ar_reject_stage (bl_tm M))))
  using nrel
  unfolding ar_delta_next_def ar_accept_stage_def ar_reject_stage_def
  by auto
  have tape: mt_tape c'' = mt_tape c'
  proof -
    have mt_tape c'' = (λk. (ts k)(n k := aw k)) using c''_eq by simp
    also have ... = ts using core by (simp add: fun_upd_triv)
    finally show ?thesis using c'_eq by simp
  qed
  have pos: mt_pos c'' = mt_pos c'
  proof -
    have mt_pos c'' = (λk. go_dir (dir k) (n k)) using c''_eq by simp
    also have ... = n using core by simp
    finally show ?thesis using c'_eq by simp
  qed
  have st: mt_state c'' = S'' using c''_eq by simp
  show ?thesis using tape pos core st by simp
qed

```

3.6 Determinism of the read substep

The read relation is a *function* of (source state, read symbol): for a fixed source s and read vector a the target state, written vector, and direction are uniquely determined. The seven arms partition by the bit-counter i , then within an i -class by $posk\ tk / a\ tk / is_last_k\ M\ tk$. The one non-obvious exclusion is arm 4 ($i = Suc\ 0$) versus arms 6/7 ($i = Suc\ (b)$): these collide only if $b = 0$, ruled out by $block_width_pos$ (which is therefore load-bearing here, not decorative).

```

lemma ar_delta_read_func:
  fixes M :: ('q, 'a) mttm
  assumes (s, a, s1, a1, d1) ∈ ar_delta_read M
    and (s, a, s2, a2, d2) ∈ ar_delta_read M
  shows (s1, a1, d1) = (s2, a2, d2)
  using assms block_width_pos[of Γ_tm M]
  by (auto simp: ar_delta_read_def)

```

Write is a function of (source, read vector): the LE arms ($buf\ tk = le$) split from the proper arms by the buf cell, and the proper back-walk / forward-write / boundary arms partition by the bit-counter ranges $[0, b-1]$, $[b, 2b-2]$, $\{2b-1\}$, separated arithmetically; is_last_k splits the last-tape arms.

```

lemma ar_delta_write_func:
  fixes M :: ('q, 'a) mttm
  assumes (s, a, s1, a1, d1) ∈ ar_delta_write M

```

```

and  $(s, a, s_2, a_2, d_2) \in \text{ar\_delta\_write } M$ 
shows  $(s_1, a_1, d_1) = (s_2, a_2, d_2)$ 
using assms
by (auto simp: ar_delta_write_def)

```

Advance is a function of (source, read vector). The stepping arm (*Suc i* < *ar_disp b (dvec tk) (posk tk)*) is excluded from the *R*-direction boundary sub-case by *ar_disp_dir.R_ = 0* (the first *ar_disp* equation), and from the non-*R* boundary by < versus = on the displacement; *is_last_k* splits the two boundary arms.

```

lemma ar_delta_advance_func:
  fixes  $M :: ('q, 'a) \text{mttm}$ 
  assumes  $(s, a, s_1, a_1, d_1) \in \text{ar\_delta\_advance } M$ 
    and  $(s, a, s_2, a_2, d_2) \in \text{ar\_delta\_advance } M$ 
  shows  $(s_1, a_1, d_1) = (s_2, a_2, d_2)$ 
  using assms
  by (auto simp: ar_delta_advance_def)

```

Next is a function of the source: the continue / accept / reject arms partition on the simulated *M*-state *q* by $q \notin \{t, r\} / q = t / q = r$, mutually exclusive precisely because *valid_mttm M* supplies $t_tm M \neq r_tm M$.

Currently uncalled: the *next* substep is the cycle closer, handled by bespoke inversion (*ar_next_step_inv*) rather than functional pinning, so this member of the per-substep determinism family goes unused; kept to keep that family complete.

```

lemma ar_delta_next_func:
  fixes  $M :: ('q, 'a) \text{mttm}$ 
  assumes  $vM: \text{valid\_mttm } M$ 
    and  $(s, a, s_1, a_1, d_1) \in \text{ar\_delta\_next } M$ 
    and  $(s, a, s_2, a_2, d_2) \in \text{ar\_delta\_next } M$ 
  shows  $(s_1, a_1, d_1) = (s_2, a_2, d_2)$ 
proof –
  have  $tr: t\_tm M \neq r\_tm M$  using vM by (cases M) auto
  show ?thesis using assms(2,3) tr by (auto simp: ar_delta_next_def)
qed

```

3.7 Source disambiguation for the remaining substeps

Source disambiguation for the remaining three substeps, completing the *from_src* family alongside *ar_delta_compute_from_src* / *ar_delta_next_from_src*.

```

lemma ar_delta_read_from_src:
  assumes  $(s, a, s', a', d) \in \text{alphabet\_reduce\_delta } M$ 
    and  $\text{fst } (\text{snd } s) = \text{AR\_SimRead}$ 
  shows  $(s, a, s', a', d) \in \text{ar\_delta\_read } M$ 
  using assms unfolding alphabet_reduce_delta_def
  by (auto dest: ar_delta_compute_src ar_delta_write_src)

```

$ar_delta_advance_src\ ar_delta_next_src)$

lemma $ar_delta_write_from_src$:
assumes $(s, a, s', a', d) \in alphabet_reduce_delta\ M$
and $fst\ (snd\ s) = AR_SimWrite$
shows $(s, a, s', a', d) \in ar_delta_write\ M$
using *assms* **unfolding** $alphabet_reduce_delta_def$
by (*auto dest*: $ar_delta_read_src\ ar_delta_compute_src$
 $ar_delta_advance_src\ ar_delta_next_src)$

lemma $ar_delta_advance_from_src$:
assumes $(s, a, s', a', d) \in alphabet_reduce_delta\ M$
and $fst\ (snd\ s) = AR_SimAdvance$
shows $(s, a, s', a', d) \in ar_delta_advance\ M$
using *assms* **unfolding** $alphabet_reduce_delta_def$
by (*auto dest*: $ar_delta_read_src\ ar_delta_compute_src$
 $ar_delta_write_src\ ar_delta_next_src)$

Every $alphabet_reduce_delta$ tuple has its source at one of the five substep tags (the halt tags never appear as sources).

lemma $alphabet_reduce_delta_src_tag$:
assumes $(s, a, s', a', d) \in alphabet_reduce_delta\ M$
shows $fst\ (snd\ s) \in \{AR_SimRead, AR_SimCompute, AR_SimWrite,$
 $AR_SimAdvance, AR_SimNext\}$
using *assms* **unfolding** $alphabet_reduce_delta_def$
by (*auto dest*: $ar_delta_read_src\ ar_delta_compute_src\ ar_delta_write_src$
 $ar_delta_advance_src\ ar_delta_next_src)$

3.8 Substep step semantics — $mttm_step$ -level lands-at

Lift the per-relation destination-tag lemmas ($ar_delta_X_dest$) up through $mttm_step$: an M' -step out of a configuration whose source idx is AR_SimX lands at a configuration whose idx is in X 's dest set. The five lemmas compose $mttm_step.cases$ (extract the firing tuple), the $from_src$ union-disambiguation helpers, and $ar_delta_X_dest$. Together they encode the cycle's substep transition graph at the level the substep-walker engine consumes: SimRead-loop-or-compute, compute-to-write, write-loop-or-advance, advance-loop-or-next, next-to-read-or-halt.

lemma $ar_step_from_SimRead_lands$:
assumes $step: (c, c') \in mttm_step\ (alphabet_reduce_delta\ M)$
and $src: fst\ (snd\ (mt_state\ c)) = AR_SimRead$
shows $fst\ (snd\ (mt_state\ c')) = AR_SimRead$
 $\vee\ fst\ (snd\ (mt_state\ c')) = AR_SimCompute$

proof —

from $step$ **obtain** $S\ ts\ n\ S'\ aw\ dir$ **where**

$c_eq: c = Config_M\ S\ ts\ n$

and $c'_eq: c' = Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k))$

and *rel*: $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir)$
 $\in alphabet_reduce_delta\ M$
by (*auto elim: mttm_step.cases*)
have *src_S*: $fst\ (snd\ \bar{S}) = AR_SimRead$ **using** *src_c_eq* **by** *simp*
have $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_read\ M$
using *ar_delta_read_from_src*[*OF rel src_S*] .
hence $fst\ (snd\ S') = AR_SimRead \vee fst\ (snd\ S') = AR_SimCompute$
by (*rule ar_delta_read_dest*)
thus *?thesis* **using** *c'_eq* **by** *simp*
qed

The *compute* member of the five-lemma lands-at family above is currently uncalled: the compute substep is the nondeterministic branch point, handled by bespoke reconstruct-and-reuse reasoning rather than the generic lands-at lift. Retained to keep the substep transition graph complete.

lemma *ar_step_from_SimCompute_lands*:
assumes *step*: $(c, c') \in mttm_step\ (alphabet_reduce_delta\ M)$
and *src*: $fst\ (snd\ (mt_state\ c)) = AR_SimCompute$
shows $fst\ (snd\ (mt_state\ c')) = AR_SimWrite$
proof –
from *step* **obtain** $S\ ts\ n\ S'\ aw\ dir$ **where**
 c_eq : $c = Config_M\ S\ ts\ n$
and c'_eq : $c' = Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k))$
and *rel*: $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir)$
 $\in alphabet_reduce_delta\ M$
by (*auto elim: mttm_step.cases*)
have *src_S*: $fst\ (snd\ \bar{S}) = AR_SimCompute$ **using** *src_c_eq* **by** *simp*
have $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_compute\ M$
using *ar_delta_compute_from_src*[*OF rel src_S*] .
hence $fst\ (snd\ S') = AR_SimWrite$ **by** (*rule ar_delta_compute_dest*)
thus *?thesis* **using** *c'_eq* **by** *simp*
qed

lemma *ar_step_from_SimWrite_lands*:
assumes *step*: $(c, c') \in mttm_step\ (alphabet_reduce_delta\ M)$
and *src*: $fst\ (snd\ (mt_state\ c)) = AR_SimWrite$
shows $fst\ (snd\ (mt_state\ c')) = AR_SimWrite$
 $\vee fst\ (snd\ (mt_state\ c')) = AR_SimAdvance$
proof –
from *step* **obtain** $S\ ts\ n\ S'\ aw\ dir$ **where**
 c_eq : $c = Config_M\ S\ ts\ n$
and c'_eq : $c' = Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k))$
and *rel*: $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir)$
 $\in alphabet_reduce_delta\ M$
by (*auto elim: mttm_step.cases*)
have *src_S*: $fst\ (snd\ \bar{S}) = AR_SimWrite$ **using** *src_c_eq* **by** *simp*
have $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_write\ M$

using $ar_delta_write_from_src[OF\ rel\ src_S]$.
 hence $fst\ (snd\ S') = AR_SimWrite \vee fst\ (snd\ S') = AR_SimAdvance$
 by (rule $ar_delta_write_dest$)
 thus ?thesis using c'_eq by simp
 qed

lemma $ar_step_from_SimAdvance_lands$:
 assumes $step: (c, c') \in mttm_step\ (alphabet_reduce_delta\ M)$
 and $src: fst\ (snd\ (mt_state\ c)) = AR_SimAdvance$
 shows $fst\ (snd\ (mt_state\ c')) = AR_SimAdvance$
 $\vee fst\ (snd\ (mt_state\ c')) = AR_SimNext$
proof –
 from $step$ obtain $S\ ts\ n\ S'\ aw\ dir$ where
 $c_eq: c = Config_M\ S\ ts\ n$
 and $c'_eq: c' = Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k))$
 and $rel: (S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir)$
 $\in alphabet_reduce_delta\ M$
 by (auto elim: $mttm_step.cases$)
 have $src_S: fst\ (snd\ S) = AR_SimAdvance$ using $src\ c_eq$ by simp
 have $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_advance\ M$
 using $ar_delta_advance_from_src[OF\ rel\ src_S]$.
 hence $fst\ (snd\ S') = AR_SimAdvance \vee fst\ (snd\ S') = AR_SimNext$
 by (rule $ar_delta_advance_dest$)
 thus ?thesis using c'_eq by simp
 qed

lemma $ar_step_from_SimNext_lands$:
 assumes $step: (c, c') \in mttm_step\ (alphabet_reduce_delta\ M)$
 and $src: fst\ (snd\ (mt_state\ c)) = AR_SimNext$
 shows $fst\ (snd\ (mt_state\ c')) = AR_SimRead$
 $\vee fst\ (snd\ (mt_state\ c')) = AR_HaltAccept$
 $\vee fst\ (snd\ (mt_state\ c')) = AR_HaltReject$
proof –
 from $step$ obtain $S\ ts\ n\ S'\ aw\ dir$ where
 $c_eq: c = Config_M\ S\ ts\ n$
 and $c'_eq: c' = Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k))$
 and $rel: (S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir)$
 $\in alphabet_reduce_delta\ M$
 by (auto elim: $mttm_step.cases$)
 have $src_S: fst\ (snd\ S) = AR_SimNext$ using $src\ c_eq$ by simp
 have $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_next\ M$
 using $ar_delta_next_from_src[OF\ rel\ src_S]$.
 hence $fst\ (snd\ S') = AR_SimRead \vee fst\ (snd\ S') = AR_HaltAccept$
 $\vee fst\ (snd\ S') = AR_HaltReject$
 by (rule $ar_delta_next_dest$)
 thus ?thesis using c'_eq by simp
 qed

3.9 Walker invariants — per-substep stage predicates

The substep-walker is a per-step induction over the M' -path that tracks where in the cycle we are by reading the substep idx off each visited configuration's state. Six stage predicates carry the per-substep relationship between the cycle's source M -config cM and the current M' -configuration c' :

- $ar_walker_at_boundary\ M\ cM\ c'$: fresh cycle start — c' at $AR_SimRead$ boundary, the three forward invariants hold for cM .
- $ar_walker_in_read\ M\ cM\ c'$: mid-read-phase — reachable from a boundary by ar_delta_read -only steps, still at $AR_SimRead$.
- $ar_walker_at_compute\ M\ cM\ c'$: read complete — reachable from a boundary by ar_delta_read -only steps, now at $AR_SimCompute$. The next M' -step on the path extracts the M -tuple via $ar_compute_step_inv$.
- $ar_walker_in_write\ M\ cM\ c'$: M -tuple extracted, mid-write-phase — at $AR_SimWrite$.
- $ar_walker_in_advance\ M\ cM\ c'$: write done, mid-advance-phase — at $AR_SimAdvance$.
- $ar_walker_at_next\ M\ cM\ c'$: at $AR_SimNext$, about to dispatch to next boundary or halt via $ar_next_step_inv$.

The witness-chain formulation (rather than concrete per-state conditions) makes preservation lemmas mechanical: at a config with substep tag T , an M' -step fires the unique substep relation with src tag T (by $ar_delta_T_src + from_src$); extending the witness chain by one step preserves the invariant. Chain shape (no cycle-wrap before completing this cycle) follows from the witness chain living in the **specific** substep relation $mttm_step\ (ar_delta_T\ M)$, which by $ar_delta_T_src$ can only fire from sources at T — ruling out the wrap.

definition $ar_walker_at_boundary ::$

```

('q, 'a) mttm
  ⇒ ('a, 'q) mt_config
  ⇒ (sym4, 'q × 'a ar_stage) mt_config ⇒ bool where
   $ar\_walker\_at\_boundary\ M\ cM\ c' \longleftrightarrow$ 
     $ar\_simulates\ M\ cM\ c'$ 
   $\wedge\ ar\_posk\_consistent\ M\ cM\ c'$ 
   $\wedge\ ar\_at\_read\_boundary\ M\ c'$ 

```

definition $ar_walker_in_read ::$

```

('q, 'a) mttm

```

$$\begin{aligned}
&\Rightarrow ('a, 'q) \text{ mt_config} \\
&\Rightarrow (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \Rightarrow \text{bool} \textbf{ where} \\
&\text{ar_walker_in_read } M \text{ cM } c' \longleftrightarrow \\
&\quad \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimRead} \\
&\wedge (\exists c_b \text{ m. ar_walker_at_boundary } M \text{ cM } c_b \\
&\quad \wedge (c_b, c') \in (\text{mttm_step} (\text{ar_delta_read } M)) \wedge m)
\end{aligned}$$

definition *ar_walker_at_compute* ::

$$\begin{aligned}
&('q, 'a) \text{ mttm} \\
&\Rightarrow ('a, 'q) \text{ mt_config} \\
&\Rightarrow (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \Rightarrow \text{bool} \textbf{ where} \\
&\text{ar_walker_at_compute } M \text{ cM } c' \longleftrightarrow \\
&\quad \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimCompute} \\
&\wedge (\exists c_b \text{ m. ar_walker_at_boundary } M \text{ cM } c_b \\
&\quad \wedge (c_b, c') \in (\text{mttm_step} (\text{ar_delta_read } M)) \wedge m)
\end{aligned}$$

definition *ar_walker_in_write* ::

$$\begin{aligned}
&('q, 'a) \text{ mttm} \\
&\Rightarrow ('a, 'q) \text{ mt_config} \\
&\Rightarrow (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \Rightarrow \text{bool} \textbf{ where} \\
&\text{ar_walker_in_write } M \text{ cM } c' \longleftrightarrow \\
&\quad \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimWrite} \\
&\wedge (\exists c_b \text{ c_w } m_r \text{ m_w.} \\
&\quad \text{ar_walker_at_boundary } M \text{ cM } c_b \\
&\quad \wedge (c_b, c_w) \in (\text{mttm_step} (\text{ar_delta_read } M)) \wedge m_r \\
&\quad \wedge \text{fst} (\text{snd} (\text{mt_state } c_w)) = \text{AR_SimCompute} \\
&\quad \wedge (\exists c_w_post. (c_w, c_w_post) \in \text{mttm_step} (\text{ar_delta_compute } M) \\
&\quad \wedge (c_w_post, c') \in (\text{mttm_step} (\text{ar_delta_write } M)) \wedge m_w))
\end{aligned}$$

definition *ar_walker_in_advance* ::

$$\begin{aligned}
&('q, 'a) \text{ mttm} \\
&\Rightarrow ('a, 'q) \text{ mt_config} \\
&\Rightarrow (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \Rightarrow \text{bool} \textbf{ where} \\
&\text{ar_walker_in_advance } M \text{ cM } c' \longleftrightarrow \\
&\quad \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimAdvance} \\
&\wedge (\exists c_b \text{ c_w } c_a \text{ m_r } m_w \text{ m_a.} \\
&\quad \text{ar_walker_at_boundary } M \text{ cM } c_b \\
&\quad \wedge (c_b, c_w) \in (\text{mttm_step} (\text{ar_delta_read } M)) \wedge m_r \\
&\quad \wedge \text{fst} (\text{snd} (\text{mt_state } c_w)) = \text{AR_SimCompute} \\
&\quad \wedge \text{fst} (\text{snd} (\text{mt_state } c_a)) = \text{AR_SimAdvance} \\
&\quad \wedge (\exists c_w_post. (c_w, c_w_post) \in \text{mttm_step} (\text{ar_delta_compute } M) \\
&\quad \wedge (c_w_post, c_a) \in (\text{mttm_step} (\text{ar_delta_write } M)) \wedge m_w \\
&\quad \wedge (c_a, c') \in (\text{mttm_step} (\text{ar_delta_advance } M)) \wedge m_a))
\end{aligned}$$

definition *ar_walker_at_next* ::

$$\begin{aligned}
&('q, 'a) \text{ mttm} \\
&\Rightarrow ('a, 'q) \text{ mt_config} \\
&\Rightarrow (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \Rightarrow \text{bool} \textbf{ where}
\end{aligned}$$

$$\begin{aligned}
& ar_walker_at_next\ M\ cM\ c' \longleftrightarrow \\
& \quad fst\ (snd\ (mt_state\ c')) = AR_SimNext \\
& \quad \wedge\ (\exists\ c_b\ c_w\ c_a\ c_n\ m_r\ m_w\ m_a. \\
& \quad \quad ar_walker_at_boundary\ M\ cM\ c_b \\
& \quad \quad \wedge\ (c_b, c_w) \in (mttm_step\ (ar_delta_read\ M)) \wedge m_r \\
& \quad \quad \wedge\ fst\ (snd\ (mt_state\ c_w)) = AR_SimCompute \\
& \quad \quad \wedge\ fst\ (snd\ (mt_state\ c_a)) = AR_SimAdvance \\
& \quad \quad \wedge\ (\exists\ c_w_post. (c_w, c_w_post) \in mttm_step\ (ar_delta_compute\ M) \\
& \quad \quad \quad \wedge\ (c_w_post, c_a) \in (mttm_step\ (ar_delta_write\ M)) \wedge \\
m_w & \quad \quad \quad \wedge\ (c_a, c_n) \in (mttm_step\ (ar_delta_advance\ M)) \wedge m_a \\
& \quad \quad \quad \wedge\ c_n = c'))
\end{aligned}$$

3.10 Step lifts — *mttm_step* to specific substep

Five *mttm_step*-to-specific-substep lifts: an M' -step in *mttm_step* (*alphabet_reduce_delta* M) whose source carries substep tag T is in fact in the smaller *mttm_step* (*ar_delta_T* M). Each composes *mttm_step.cases* (destructure the step), the matching *ar_delta_T_from_src* helper (narrow the firing tuple by src-tag uniqueness), and *mttm_step.step* with *where* $ts = ts$ and $n = n$ instantiation (break the higher-order unification ambiguity inherent in *mttm_step.step*'s pattern when matched against concrete tuples). The walker preservation lemmas chain these lifts with the dest-tag dispatch (lands-at lemmas) to advance the witness chain by one step.

lemma *ar_step_read_lift*:

fixes $M :: ('q, 'a)\ mttm$
and $c'\ c'' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes *src*: $fst\ (snd\ (mt_state\ c')) = AR_SimRead$
and *step*: $(c', c'') \in mttm_step\ (alphabet_reduce_delta\ M)$
shows $(c', c'') \in mttm_step\ (ar_delta_read\ M)$
proof –
from *step* **obtain** $S\ ts\ n\ S'\ aw\ dir$ **where**
 $c'_eq: c' = Config_M\ S\ ts\ n$
and $c''_eq: c'' = Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k))$
and *rel*: $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir)$
 $\in alphabet_reduce_delta\ M$
by (*auto elim: mttm_step.cases*)
have *src_S*: $fst\ (snd\ S) = AR_SimRead$ **using** *src* c'_eq **by** *simp*
have *rel_T*: $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_read\ M$
using *ar_delta_read_from_src*[*OF rel src_S*] .
have *step_aux*: $(Config_M\ S\ ts\ n,$
 $Config_M\ S'\ (\lambda k. (ts\ k)(n\ k := aw\ k))$
 $(\lambda k. go_dir\ (dir\ k)\ (n\ k)))$
 $\in mttm_step\ (ar_delta_read\ M)$
by (*rule mttm_step.step[where* $ts = ts$ **and** $n = n,$ *OF rel_T***])**
show *?thesis* **using** *step_aux* $c'_eq\ c''_eq$ **by** *simp*
qed

lemma *ar_step_compute_lift*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' c'' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $\text{src}: \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimCompute}$
and $\text{step}: (c', c'') \in \text{mttm_step} (\text{alphabet_reduce_delta } M)$
shows $(c', c'') \in \text{mttm_step} (\text{ar_delta_compute } M)$
proof –
from *step* **obtain** $S \text{ ts } n S' \text{ aw } \text{dir}$ **where**
 $c'_\text{eq}: c' = \text{Config}_M S \text{ ts } n$
and $c''_\text{eq}: c'' = \text{Config}_M S' (\lambda k. (\text{ts } k)(n \text{ k} := \text{aw } k))$
 $(\lambda k. \text{go_dir } (\text{dir } k) (n \text{ k}))$
and $\text{rel}: (S, (\lambda k. \text{ts } k (n \text{ k})), S', \text{aw}, \text{dir})$
 $\in \text{alphabet_reduce_delta } M$
by (*auto elim: mttm_step.cases*)
have $\text{src_S}: \text{fst} (\text{snd } S) = \text{AR_SimCompute}$ **using** $\text{src } c'_\text{eq}$ **by** *simp*
have $\text{rel_T}: (S, (\lambda k. \text{ts } k (n \text{ k})), S', \text{aw}, \text{dir}) \in \text{ar_delta_compute } M$
using $\text{ar_delta_compute_from_src}[\text{OF rel src_S}]$.
have $\text{step_aux}: (\text{Config}_M S \text{ ts } n,$
 $\text{Config}_M S' (\lambda k. (\text{ts } k)(n \text{ k} := \text{aw } k))$
 $(\lambda k. \text{go_dir } (\text{dir } k) (n \text{ k})))$
 $\in \text{mttm_step} (\text{ar_delta_compute } M)$
by (*rule mttm_step.step[where ts = ts and n = n, OF rel_T]*)
show *?thesis* **using** $\text{step_aux } c'_\text{eq } c''_\text{eq}$ **by** *simp*
qed

lemma *ar_step_write_lift*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' c'' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $\text{src}: \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimWrite}$
and $\text{step}: (c', c'') \in \text{mttm_step} (\text{alphabet_reduce_delta } M)$
shows $(c', c'') \in \text{mttm_step} (\text{ar_delta_write } M)$
proof –
from *step* **obtain** $S \text{ ts } n S' \text{ aw } \text{dir}$ **where**
 $c'_\text{eq}: c' = \text{Config}_M S \text{ ts } n$
and $c''_\text{eq}: c'' = \text{Config}_M S' (\lambda k. (\text{ts } k)(n \text{ k} := \text{aw } k))$
 $(\lambda k. \text{go_dir } (\text{dir } k) (n \text{ k}))$
and $\text{rel}: (S, (\lambda k. \text{ts } k (n \text{ k})), S', \text{aw}, \text{dir})$
 $\in \text{alphabet_reduce_delta } M$
by (*auto elim: mttm_step.cases*)
have $\text{src_S}: \text{fst} (\text{snd } S) = \text{AR_SimWrite}$ **using** $\text{src } c'_\text{eq}$ **by** *simp*
have $\text{rel_T}: (S, (\lambda k. \text{ts } k (n \text{ k})), S', \text{aw}, \text{dir}) \in \text{ar_delta_write } M$
using $\text{ar_delta_write_from_src}[\text{OF rel src_S}]$.
have $\text{step_aux}: (\text{Config}_M S \text{ ts } n,$
 $\text{Config}_M S' (\lambda k. (\text{ts } k)(n \text{ k} := \text{aw } k))$
 $(\lambda k. \text{go_dir } (\text{dir } k) (n \text{ k})))$
 $\in \text{mttm_step} (\text{ar_delta_write } M)$
by (*rule mttm_step.step[where ts = ts and n = n, OF rel_T]*)
show *?thesis* **using** $\text{step_aux } c'_\text{eq } c''_\text{eq}$ **by** *simp*

qed

lemma *ar_step_advance_lift*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' \ c'' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
assumes *src*: $\text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimAdvance}$
and *step*: $(c', c'') \in \text{mttm_step} (\text{alphabet_reduce_delta } M)$
shows $(c', c'') \in \text{mttm_step} (\text{ar_delta_advance } M)$
proof –
from *step* **obtain** $S \ ts \ n \ S' \ aw \ dir$ **where**
 $c' \text{_eq}: c' = \text{Config}_M \ S \ ts \ n$
and $c'' \text{_eq}: c'' = \text{Config}_M \ S' (\lambda k. (ts \ k)(n \ k := aw \ k))$
 $(\lambda k. go_dir \ (dir \ k) \ (n \ k))$
and *rel*: $(S, (\lambda k. ts \ k \ (n \ k)), S', aw, dir)$
 $\in \text{alphabet_reduce_delta } M$
by (*auto elim: mttm_step.cases*)
have *src* S : $\text{fst} (\text{snd } S) = \text{AR_SimAdvance}$ **using** *src* $c' \text{_eq}$ **by** *simp*
have *rel* T : $(S, (\lambda k. ts \ k \ (n \ k)), S', aw, dir) \in \text{ar_delta_advance } M$
using *ar_delta_advance_from_src* [*OF rel src* S] .
have *step_aux*: $(\text{Config}_M \ S \ ts \ n,$
 $\text{Config}_M \ S' (\lambda k. (ts \ k)(n \ k := aw \ k))$
 $(\lambda k. go_dir \ (dir \ k) \ (n \ k)))$
 $\in \text{mttm_step} (\text{ar_delta_advance } M)$
by (*rule mttm_step.step[where ts = ts and n = n, OF rel T]*)
show *?thesis* **using** *step_aux c' _eq c'' _eq* **by** *simp*
qed

Existence-lift wrappers around the *ar_step_X_lift* lemmas for the three generic substeps (read, write, advance): convert an existential conclusion $\exists c''. (c', c'') \in \text{mttm_step} (\text{alphabet_reduce_delta } M) \wedge P \ c''$ into the same existential with the step in $\text{mttm_step} (\text{ar_delta_X } M)$, given the source tag of c' . These collapse the leaf-in-sub boilerplate (the *obtain ... using ar_step_X_lift[OF src ...] ... show ?thesis using ... by blast* scaffold) into a single application. The other two substeps carry no wrapper: compute is the nondeterministic branch and next closes the cycle, so both are handled by the cycle-close's reconstruct-and-reuse machinery rather than a generic lift.

lemma *ar_exists_step_in_sub_read*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
and $P :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config} \Rightarrow \text{bool}$
assumes *orig*: $\exists c''. (c', c'') \in \text{mttm_step} (\text{alphabet_reduce_delta } M)$
 $\wedge P \ c''$
and *src*: $\text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimRead}$
shows $\exists c''. (c', c'') \in \text{mttm_step} (\text{ar_delta_read } M) \wedge P \ c''$
proof –
obtain c'' **where** *step*: $(c', c'') \in \text{mttm_step} (\text{alphabet_reduce_delta } M)$
and *pc*: $P \ c''$

```

    using orig by blast
    have step_sub: (c', c'') ∈ mttm_step (ar_delta_read M)
    using ar_step_read_lift[OF src step] .
    show ?thesis using step_sub pc by blast
qed

lemma ar_exists_step_in_sub_write:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  and P :: (sym4, 'q × 'a ar_stage) mt_config ⇒ bool
  assumes orig: ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    ∧ P c''
    and src: fst (snd (mt_state c')) = AR_SimWrite
  shows ∃ c''. (c', c'') ∈ mttm_step (ar_delta_write M) ∧ P c''
proof -
  obtain c'' where step: (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    and pc: P c''
  using orig by blast
  have step_sub: (c', c'') ∈ mttm_step (ar_delta_write M)
  using ar_step_write_lift[OF src step] .
  show ?thesis using step_sub pc by blast
qed

lemma ar_exists_step_in_sub_advance:
  fixes M :: ('q, 'a) mttm
  and c' :: (sym4, 'q × 'a ar_stage) mt_config
  and P :: (sym4, 'q × 'a ar_stage) mt_config ⇒ bool
  assumes orig: ∃ c''. (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    ∧ P c''
    and src: fst (snd (mt_state c')) = AR_SimAdvance
  shows ∃ c''. (c', c'') ∈ mttm_step (ar_delta_advance M) ∧ P c''
proof -
  obtain c'' where step: (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    and pc: P c''
  using orig by blast
  have step_sub: (c', c'') ∈ mttm_step (ar_delta_advance M)
  using ar_step_advance_lift[OF src step] .
  show ?thesis using step_sub pc by blast
qed

```

3.11 Chain pinning in $mttm_step\ (ar_delta_read\ M)$

Two structural facts about chains in the read sub-relation: the sub-relation is **functional** (lifted from $ar_delta_read_func$ via $mttm_step$'s shape), so its m -step extension of any seed is unique; and it cannot fire from a source whose substep tag is $AR_SimCompute$ (by $ar_delta_read_src$). Together these pin a chain ending at $AR_SimCompute$ uniquely on both its length and its endpoint: if two chains in the sub-relation start at the same seed and both end at an $AR_SimCompute$ -tagged config, they coincide. This

is what bridges the walker's by-construction R_read witness chain to the existence chain produced by the (re-mirrored) $ar_read_phase_in_sub$: the witness chain inherits the latter's stated endpoint state shape, including the load-bearing $buf = \lambda k. mt_tape\ cM\ k\ (mt_pos\ cM\ k)$.

lemma $mttm_step_ar_delta_read_func$:

fixes $M :: ('q, 'a)\ mttm$
and $c\ c_1\ c_2 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes $h_1: (c, c_1) \in mttm_step\ (ar_delta_read\ M)$
and $h_2: (c, c_2) \in mttm_step\ (ar_delta_read\ M)$
shows $c_1 = c_2$

proof –

from h_1 **obtain** $S\ ts\ n\ S_1\ aw_1\ dir_1$ **where**
 $ceq_1: c = Config_M\ S\ ts\ n$
and $c_1eq: c_1 = Config_M\ S_1\ (\lambda k. (ts\ k)(n\ k := aw_1\ k))$
 $(\lambda k. go_dir\ (dir_1\ k)\ (n\ k))$
and $rel_1: (S, (\lambda k. ts\ k\ (n\ k)), S_1, aw_1, dir_1) \in ar_delta_read\ M$
by $(auto\ elim: mttm_step.cases)$
from h_2 **obtain** $S'\ ts'\ n'\ S_2\ aw_2\ dir_2$ **where**
 $ceq_2: c = Config_M\ S'\ ts'\ n'$
and $c_2eq: c_2 = Config_M\ S_2\ (\lambda k. (ts'\ k)(n'\ k := aw_2\ k))$
 $(\lambda k. go_dir\ (dir_2\ k)\ (n'\ k))$
and $rel_2: (S', (\lambda k. ts'\ k\ (n'\ k)), S_2, aw_2, dir_2) \in ar_delta_read\ M$
by $(auto\ elim: mttm_step.cases)$
have $eq: S = S' \wedge ts = ts' \wedge n = n'$ **using** $ceq_1\ ceq_2$ **by** $simp$
have $rel_2': (S, (\lambda k. ts\ k\ (n\ k)), S_2, aw_2, dir_2) \in ar_delta_read\ M$
using $rel_2\ eq$ **by** $simp$
have $(S_1, aw_1, dir_1) = (S_2, aw_2, dir_2)$
using $ar_delta_read_func[OF\ rel_1\ rel_2']$.
hence $S_1 = S_2 \wedge aw_1 = aw_2 \wedge dir_1 = dir_2$ **by** $simp$
thus $?thesis$ **using** $c_1eq\ c_2eq\ eq$ **by** $simp$

qed

lemma $chain_ar_delta_read_func$:

fixes $M :: ('q, 'a)\ mttm$
shows $(c, c_1) \in (mttm_step\ (ar_delta_read\ M))\ \wedge\wedge\ m$
 $\implies (c, c_2) \in (mttm_step\ (ar_delta_read\ M))\ \wedge\wedge\ m$
 $\implies c_1 = c_2$

proof $(induction\ m\ arbitrary: c_1\ c_2)$

case 0

thus $?case$ **by** $auto$

next

case $(Suc\ m)$

obtain c_1' **where**

$a: (c, c_1') \in (mttm_step\ (ar_delta_read\ M))\ \wedge\wedge\ m$
and $b: (c_1', c_1) \in mttm_step\ (ar_delta_read\ M)$
using $Suc(2)$ **by** $(auto\ elim: relpow_Suc_E)$

obtain c_2' **where**

$c: (c, c_2') \in (mttm_step\ (ar_delta_read\ M))\ \wedge\wedge\ m$
and $d: (c_2', c_2) \in mttm_step\ (ar_delta_read\ M)$

using $Suc(3)$ by (auto elim: relpow_Suc_E)
 have $c_1' = c_2'$ using $Suc(1)[OF\ a\ c]$.
 thus ?case using $b\ d\ mttm_step_ar_delta_read_func$ by simp
 qed

lemma $ar_delta_read_no_step_from_SimCompute$:
 fixes $M :: ('q, 'a)\ mttm$
 and $c\ c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
 assumes $fst\ (snd\ (mt_state\ c)) = AR_SimCompute$
 shows $(c, c') \notin mttm_step\ (ar_delta_read\ M)$
proof
 assume $h: (c, c') \in mttm_step\ (ar_delta_read\ M)$
 from h obtain $S\ ts\ n\ S'\ aw\ dir$ where
 $ceq: c = Config_M\ S\ ts\ n$
 and $rel: (S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_read\ M$
 by (auto elim: mttm_step.cases)
 have $fst\ (snd\ S) = AR_SimRead$ using $ar_delta_read_src[OF\ rel]$.
 hence $fst\ (snd\ (mt_state\ c)) = AR_SimRead$ using ceq by simp
 with $assms$ show $False$ by simp
 qed

lemma $chain_ar_delta_read_to_SimCompute_uniq$:
 fixes $M :: ('q, 'a)\ mttm$
 and $c\ c_1\ c_2 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
 assumes $h_1: (c, c_1) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ m}$
 and $h_2: (c, c_2) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ n}$
 and $c_1\ cpu: fst\ (snd\ (mt_state\ c_1)) = AR_SimCompute$
 and $c_2\ cpu: fst\ (snd\ (mt_state\ c_2)) = AR_SimCompute$
 shows $m = n \wedge c_1 = c_2$
proof –
 have aux :
 $\bigwedge m\ n\ c_1\ c_2. (c, c_1) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ m}$
 $\implies (c, c_2) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ n}$
 $\implies fst\ (snd\ (mt_state\ c_1)) = AR_SimCompute$
 $\implies m \leq n$
 $\implies m = n$
proof –
 fix $m\ n\ c_1\ c_2$
 assume $a_1: (c, c_1) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ m}$
 and $a_2: (c, c_2) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ n}$
 and $acpu: fst\ (snd\ (mt_state\ c_1)) = AR_SimCompute$
 and $ale: m \leq n$
 obtain dm where $ndecomp: n = m + dm$ using $ale\ le_Suc_ex$ by blast
 have $(c, c_2) \in ((mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ m})$
 $O\ ((mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ dm})$
 using $a_2\ ndecomp$ by (simp add: relpow_add)
 then obtain c_m where
 $am: (c, c_m) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ m}$
 and $adm: (c_m, c_2) \in (mttm_step\ (ar_delta_read\ M))^{\wedge\wedge\ dm}$

```

    by auto
  have cm_eq:  $c_m = c_1$  using chain_ar_delta_read_func[OF am a1] .
  show  $m = n$ 
  proof (rule ccontr)
    assume  $m \neq n$ 
    hence dm_pos:  $0 < dm$  using ndecomp by simp
    then obtain dm' where dm_eq:  $dm = \text{Suc } dm'$  using gr0_implies_Suc by
blast
    have hsuc:  $(c_1, c_2) \in (\text{mttm\_step } (\text{ar\_delta\_read } M)) \wedge (\text{Suc } dm')$ 
      using adm cm_eq dm_eq by simp
    obtain c_next where
      first:  $(c_1, c_{\text{next}}) \in \text{mttm\_step } (\text{ar\_delta\_read } M)$ 
      using relpow_Suc_D2[OF hsuc] by blast
    have  $(c_1, c_{\text{next}}) \notin \text{mttm\_step } (\text{ar\_delta\_read } M)$ 
      using acpu by (rule ar_delta_read_no_step_from_SimCompute)
    thus False using first by simp
  qed
qed
  have mn_eq:  $m = n$ 
  proof (cases  $m \leq n$ )
    case True
      show ?thesis using aux[OF h1 h2 c1cpu True] .
    next
      case False
        hence nle:  $n \leq m$  by simp
        show ?thesis using aux[OF h2 h1 c2cpu nle] by simp
  qed
  have  $c_1 = c_2$ 
    using chain_ar_delta_read_func[OF h1 h2[unfolded mn_eq[symmetric]]] .
  thus ?thesis using mn_eq by simp
qed

```

Two parallel chain-pinning suites for the write and advance sub-relations, mirroring the read suite verbatim with *ar_delta_write* / *ar_delta_advance* in place of *ar_delta_read* and *AR_SimAdvance* / *AR_SimNext* in place of *AR_SimCompute*. Functional projections (*ar_delta_write_func*, *ar_delta_advance_func*) and src uniqueness (*ar_delta_write_src*, *ar_delta_advance_src*) feed the same scaffold. These suites are used by the cycle-close bridging lemma to pin walker write/advance chains against the forward *ar_write_phase* / *ar_advance_phase* constructions.

```

lemma mttm_step_ar_delta_write_func:
  fixes M :: ('q, 'a) mttm
  and c c1 c2 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes h1:  $(c, c_1) \in \text{mttm\_step } (\text{ar\_delta\_write } M)$ 
    and h2:  $(c, c_2) \in \text{mttm\_step } (\text{ar\_delta\_write } M)$ 
  shows  $c_1 = c_2$ 
proof -

```

from h_1 **obtain** $S \text{ ts } n \ S_1 \ aw_1 \ dir_1$ **where**
 $ceq_1: c = \text{Config}_M \ S \ ts \ n$
and $c_1eq: c_1 = \text{Config}_M \ S_1 \ (\lambda k. (ts \ k)(n \ k := aw_1 \ k))$
 $(\lambda k. go_dir \ (dir_1 \ k) \ (n \ k))$
and $rel_1: (S, (\lambda k. ts \ k \ (n \ k)), S_1, aw_1, dir_1) \in ar_delta_write \ M$
by $(auto \ elim: mttm_step.cases)$
from h_2 **obtain** $S' \ ts' \ n' \ S_2 \ aw_2 \ dir_2$ **where**
 $ceq_2: c = \text{Config}_M \ S' \ ts' \ n'$
and $c_2eq: c_2 = \text{Config}_M \ S_2 \ (\lambda k. (ts' \ k)(n' \ k := aw_2 \ k))$
 $(\lambda k. go_dir \ (dir_2 \ k) \ (n' \ k))$
and $rel_2: (S', (\lambda k. ts' \ k \ (n' \ k)), S_2, aw_2, dir_2) \in ar_delta_write \ M$
by $(auto \ elim: mttm_step.cases)$
have $eq: S = S' \wedge ts = ts' \wedge n = n'$ **using** $ceq_1 \ ceq_2$ **by** $simp$
have $rel_2': (S, (\lambda k. ts \ k \ (n \ k)), S_2, aw_2, dir_2) \in ar_delta_write \ M$
using $rel_2 \ eq$ **by** $simp$
have $(S_1, aw_1, dir_1) = (S_2, aw_2, dir_2)$
using $ar_delta_write_func[OF \ rel_1 \ rel_2']$.
hence $S_1 = S_2 \wedge aw_1 = aw_2 \wedge dir_1 = dir_2$ **by** $simp$
thus $?thesis$ **using** $c_1eq \ c_2eq \ eq$ **by** $simp$
qed

lemma $chain_ar_delta_write_func$:
fixes $M :: ('q, 'a) \ mttm$
shows $(c, c_1) \in (mttm_step \ (ar_delta_write \ M)) \ \wedge \wedge \ m$
 $\implies (c, c_2) \in (mttm_step \ (ar_delta_write \ M)) \ \wedge \wedge \ m$
 $\implies c_1 = c_2$
proof $(induction \ m \ arbitrary: c_1 \ c_2)$
case 0
thus $?case$ **by** $auto$
next
case $(Suc \ m)$
obtain c_1' **where**
 $a: (c, c_1') \in (mttm_step \ (ar_delta_write \ M)) \ \wedge \wedge \ m$
and $b: (c_1', c_1) \in mttm_step \ (ar_delta_write \ M)$
using $Suc(2)$ **by** $(auto \ elim: relpow_Suc_E)$
obtain c_2' **where**
 $c: (c, c_2') \in (mttm_step \ (ar_delta_write \ M)) \ \wedge \wedge \ m$
and $d: (c_2', c_2) \in mttm_step \ (ar_delta_write \ M)$
using $Suc(3)$ **by** $(auto \ elim: relpow_Suc_E)$
have $c_1' = c_2'$ **using** $Suc(1)[OF \ a \ c]$.
thus $?case$ **using** $b \ d \ mttm_step_ar_delta_write_func$ **by** $simp$
qed

lemma $ar_delta_write_no_step_from_SimAdvance$:
fixes $M :: ('q, 'a) \ mttm$
and $c \ c' :: (sym4, 'q \times 'a \ ar_stage) \ mt_config$
assumes $fst \ (snd \ (mt_state \ c)) = AR_SimAdvance$
shows $(c, c') \notin mttm_step \ (ar_delta_write \ M)$
proof

assume $h: (c, c') \in \text{mttm_step } (\text{ar_delta_write } M)$
from h **obtain** $S \text{ ts } n \text{ S' aw dir}$ **where**
 $\text{ceq}: c = \text{Config}_M \text{ S ts } n$
and $\text{rel}: (S, (\lambda k. \text{ts } k (n \text{ k})), S', \text{aw}, \text{dir}) \in \text{ar_delta_write } M$
by $(\text{auto elim: mttm_step.cases})$
have $\text{fst } (\text{snd } S) = \text{AR_SimWrite}$ **using** $\text{ar_delta_write_src}[OF \text{ rel}]$.
hence $\text{fst } (\text{snd } (\text{mt_state } c)) = \text{AR_SimWrite}$ **using** ceq **by** simp
with assms **show** False **by** simp
qed

lemma $\text{chain_ar_delta_write_to_SimAdvance_uniq}$:

fixes $M :: ('q, 'a) \text{ mttm}$
and $c \ c_1 \ c_2 :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{ mt_config}$
assumes $h_1: (c, c_1) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^m$
and $h_2: (c, c_2) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^n$
and $c_1 \text{adv}: \text{fst } (\text{snd } (\text{mt_state } c_1)) = \text{AR_SimAdvance}$
and $c_2 \text{adv}: \text{fst } (\text{snd } (\text{mt_state } c_2)) = \text{AR_SimAdvance}$
shows $m = n \wedge c_1 = c_2$
proof –
have aux :
 $\bigwedge^m n \ c_1 \ c_2. (c, c_1) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^m$
 $\implies (c, c_2) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^n$
 $\implies \text{fst } (\text{snd } (\text{mt_state } c_1)) = \text{AR_SimAdvance}$
 $\implies m \leq n$
 $\implies m = n$

proof –

fix $m \ n \ c_1 \ c_2$
assume $a_1: (c, c_1) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^m$
and $a_2: (c, c_2) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^n$
and $a \text{adv}: \text{fst } (\text{snd } (\text{mt_state } c_1)) = \text{AR_SimAdvance}$
and $\text{ale}: m \leq n$
obtain dm **where** $\text{ndecomp}: n = m + dm$ **using** ale le_Suc_ex **by** blast
have $(c, c_2) \in ((\text{mttm_step } (\text{ar_delta_write } M)) \wedge^m)$
 $\quad O((\text{mttm_step } (\text{ar_delta_write } M)) \wedge^{dm})$
using $a_2 \text{ ndecomp}$ **by** $(\text{simp add: relpow_add})$
then obtain c_m **where**
 $\text{am}: (c, c_m) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^m$
and $\text{adm}: (c_m, c_2) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^{dm}$
by auto
have $\text{cm_eq}: c_m = c_1$ **using** $\text{chain_ar_delta_write_func}[OF \text{ am } a_1]$.
show $m = n$
proof (rule ccontr)
assume $m \neq n$
hence $dm_pos: 0 < dm$ **using** ndecomp **by** simp
then obtain dm' **where** $dm_eq: dm = \text{Suc } dm'$ **using** gr0_implies_Suc **by**
 blast
have $\text{hsuc}: (c_1, c_2) \in (\text{mttm_step } (\text{ar_delta_write } M)) \wedge^{(\text{Suc } dm')}$
using adm cm_eq dm_eq **by** simp
obtain c_next **where**


```

      first: (c1, cnext) ∈ mttm_step (ar_delta_write M)
    using relpow_Suc_D2[OF hsuc] by blast
  have (c1, cnext) ∉ mttm_step (ar_delta_write M)
    using aadv by (rule ar_delta_write_no_step_from_SimAdvance)
  thus False using first by simp
qed
qed
have mn_eq: m = n
proof (cases m ≤ n)
  case True
  show ?thesis using aux[OF h1 h2 c1adv True] .
next
  case False
  hence nle: n ≤ m by simp
  show ?thesis using aux[OF h2 h1 c2adv nle] by simp
qed
have c1 = c2
  using chain_ar_delta_write_func[OF h1 h2[unfolded mn_eq[symmetric]]] .
  thus ?thesis using mn_eq by simp
qed

lemma mttm_step_ar_delta_advance_func:
  fixes M :: ('q, 'a) mttm
  and c c1 c2 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes h1: (c, c1) ∈ mttm_step (ar_delta_advance M)
    and h2: (c, c2) ∈ mttm_step (ar_delta_advance M)
  shows c1 = c2
proof -
  from h1 obtain S ts n S1 aw1 dir1 where
    ceq1: c = ConfigM S ts n
    and c1eq: c1 = ConfigM S1 (λk. (ts k)(n k := aw1 k))
      (λk. go_dir (dir1 k) (n k))
    and rel1: (S, (λk. ts k (n k)), S1, aw1, dir1) ∈ ar_delta_advance M
    by (auto elim: mttm_step.cases)
  from h2 obtain S' ts' n' S2 aw2 dir2 where
    ceq2: c = ConfigM S' ts' n'
    and c2eq: c2 = ConfigM S2 (λk. (ts' k)(n' k := aw2 k))
      (λk. go_dir (dir2 k) (n' k))
    and rel2: (S', (λk. ts' k (n' k)), S2, aw2, dir2) ∈ ar_delta_advance M
    by (auto elim: mttm_step.cases)
  have eq: S = S' ∧ ts = ts' ∧ n = n' using ceq1 ceq2 by simp
  have rel2': (S, (λk. ts k (n k)), S2, aw2, dir2) ∈ ar_delta_advance M
    using rel2 eq by simp
  have (S1, aw1, dir1) = (S2, aw2, dir2)
    using ar_delta_advance_func[OF rel1 rel2'] .
  hence S1 = S2 ∧ aw1 = aw2 ∧ dir1 = dir2 by simp
  thus ?thesis using c1eq c2eq eq by simp
qed

```

```

lemma chain_ar_delta_advance_func:
  fixes M :: ('q, 'a) mttm
  shows (c, c1) ∈ (mttm_step (ar_delta_advance M)) ^m
    ⇒ (c, c2) ∈ (mttm_step (ar_delta_advance M)) ^m
    ⇒ c1 = c2
proof (induction m arbitrary: c1 c2)
  case 0
  thus ?case by auto
next
  case (Suc m)
  obtain c1' where
    a: (c, c1') ∈ (mttm_step (ar_delta_advance M)) ^m
    and b: (c1', c1) ∈ mttm_step (ar_delta_advance M)
    using Suc(2) by (auto elim: relpow_Suc_E)
  obtain c2' where
    c: (c, c2') ∈ (mttm_step (ar_delta_advance M)) ^m
    and d: (c2', c2) ∈ mttm_step (ar_delta_advance M)
    using Suc(3) by (auto elim: relpow_Suc_E)
  have c1' = c2' using Suc(1)[OF a c] .
  thus ?case using b d mttm_step_ar_delta_advance_func by simp
qed

```

```

lemma ar_delta_advance_no_step_from_SimNext:
  fixes M :: ('q, 'a) mttm
  and c c' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes fst (snd (mt_state c)) = AR_SimNext
  shows (c, c') ∉ mttm_step (ar_delta_advance M)
proof
  assume h: (c, c') ∈ mttm_step (ar_delta_advance M)
  from h obtain S ts n S' aw dir where
    ceq: c = ConfigM S ts n
    and rel: (S, (λk. ts k (n k)), S', aw, dir) ∈ ar_delta_advance M
    by (auto elim: mttm_step.cases)
  have fst (snd S) = AR_SimAdvance using ar_delta_advance_src[OF rel] .
  hence fst (snd (mt_state c)) = AR_SimAdvance using ceq by simp
  with assms show False by simp
qed

```

```

lemma chain_ar_delta_advance_to_SimNext_uniq:
  fixes M :: ('q, 'a) mttm
  and c c1 c2 :: (sym4, 'q × 'a ar_stage) mt_config
  assumes h1: (c, c1) ∈ (mttm_step (ar_delta_advance M)) ^m
    and h2: (c, c2) ∈ (mttm_step (ar_delta_advance M)) ^n
    and c1next: fst (snd (mt_state c1)) = AR_SimNext
    and c2next: fst (snd (mt_state c2)) = AR_SimNext
  shows m = n ∧ c1 = c2
proof –
  have aux:
    ∧ m n c1 c2. (c, c1) ∈ (mttm_step (ar_delta_advance M)) ^m

```

$$\begin{aligned} &\Rightarrow (c, c_2) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge n \\ &\Rightarrow \text{fst } (\text{snd } (\text{mt_state } c_1)) = \text{AR_SimNext} \\ &\Rightarrow m \leq n \\ &\Rightarrow m = n \end{aligned}$$

proof –

fix $m\ n\ c_1\ c_2$

assume $a_1: (c, c_1) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge m$

and $a_2: (c, c_2) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge n$

and $\text{anxt}: \text{fst } (\text{snd } (\text{mt_state } c_1)) = \text{AR_SimNext}$

and $\text{ale}: m \leq n$

obtain dm **where** $\text{ndecomp}: n = m + dm$ **using** $\text{ale}\ \text{le_Suc_ex}$ **by** *blast*

have $(c, c_2) \in ((\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge m)$

$O((\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge dm)$

using $a_2\ \text{ndecomp}$ **by** $(\text{simp add: relpow_add})$

then obtain c_m **where**

$\text{am}: (c, c_m) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge m$

and $\text{adm}: (c_m, c_2) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge dm$

by *auto*

have $\text{cm_eq}: c_m = c_1$ **using** $\text{chain_ar_delta_advance_func}[OF\ \text{am}\ a_1]$.

show $m = n$

proof (*rule ccontr*)

assume $m \neq n$

hence $dm_pos: 0 < dm$ **using** ndecomp **by** *simp*

then obtain dm' **where** $dm_eq: dm = \text{Suc } dm'$ **using** $gr0_implies_Suc$ **by** *blast*

have $hsuc: (c_1, c_2) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge (\text{Suc } dm')$

using $\text{adm}\ \text{cm_eq}\ dm_eq$ **by** *simp*

obtain c_next **where**

$\text{first}: (c_1, c_next) \in \text{mttm_step } (\text{ar_delta_advance } M)$

using $\text{relpow_Suc_D2}[OF\ hsuc]$ **by** *blast*

have $(c_1, c_next) \notin \text{mttm_step } (\text{ar_delta_advance } M)$

using anxt **by** (*rule ar_delta_advance_no_step_from_SimNext*)

thus *False* **using** *first* **by** *simp*

qed

qed

have $mn_eq: m = n$

proof (*cases* $m \leq n$)

case *True*

show *?thesis* **using** $\text{aux}[OF\ h_1\ h_2\ c_1\text{next}\ \text{True}]$.

next

case *False*

hence $nle: n \leq m$ **by** *simp*

show *?thesis* **using** $\text{aux}[OF\ h_2\ h_1\ c_2\text{next}\ nle]$ **by** *simp*

qed

have $c_1 = c_2$

using $\text{chain_ar_delta_advance_func}[OF\ h_1\ h_2[\text{unfolded } mn_eq[\text{symmetric}]]]$

.

thus *?thesis* **using** mn_eq **by** *simp*

qed

3.12 Read-phase leaves, sub-relation chain variants

For each single-step read-phase leaf $(ar_read_le_step, ar_read_le_finish_step, ar_read_lookback1_step, ar_read_lookback2_step, ar_read_bit_step, ar_read_bit_boundary_step, ar_read_bit_finish_step)$, a companion lemma producing the step in $mttm_step (ar_delta_read M)$ instead of $mttm_step (alphabet_reduce_delta M)$. Each variant uses the existing lemma to obtain the step, then lifts via $ar_step_read_lift$ (the source tag is $AR_SimRead$ by the leaf's stg hypothesis). No re-derivation of the step's effect — the existing leaf's stated post-state, post-tape, post-pos conclusions flow through verbatim.

lemma $ar_read_le_step_in_sub$:
fixes $M :: ('q, 'a) mttm$
and $c' :: (sym4, 'q \times 'a ar_stage) mt_config$
assumes $vM: valid_mttm M$
and $stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and $qQ: q \in Q_tm M$
and $notlast: \neg is_last_k M tk$
and $posk_le: posk tk = AR_AtLE$
and $aLE: mt_tape c' tk (mt_pos c' tk) = LE4$
and $vsrc: ar_valid_stage (\Gamma_tm M) (bl_tm M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
and $pad_blank: \forall j \geq k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4$
and $src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)$
 $(AR_SimRead, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step (ar_delta_read M)$
 $\wedge mt_state c'' = (q, AR_SimRead, k_succ tk, 0,$
 $buf(tk := le_tm M), dvec, posk)$
 $\wedge mt_tape c'' = mt_tape c'$
 $\wedge mt_pos c'' = (mt_pos c')(tk := Suc (mt_pos c' tk))$
proof –
have $src: fst (snd (mt_state c')) = AR_SimRead$ **using** stg **by** $simp$
show $?thesis$
using $ar_exists_step_in_sub_read$
 $[OF ar_read_le_step[OF vM stg qQ notlast posk_le aLE vsrc pad_blank$
 $src_bounded] src]$.
qed

lemma $ar_read_le_finish_step_in_sub$:
fixes $M :: ('q, 'a) mttm$
and $c' :: (sym4, 'q \times 'a ar_stage) mt_config$
assumes $vM: valid_mttm M$
and $stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)$
and $qQ: q \in Q_tm M$
and $last: is_last_k M tk$
and $posk_le: posk tk = AR_AtLE$
and $aLE: mt_tape c' tk (mt_pos c' tk) = LE4$

and *vsr*: *ar_valid_stage* ($\Gamma_{tm} M$) (*bl_tm* *M*)
 (*AR_SimRead*, *tk*, 0, *buf*, *dvec*, *posk*)
and *pad_blank*: $\forall j \geq k_{tm} M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and *src_bounded*: *ar_stage_bounded* (*bl_tm* *M*) (*k_tm* *M*)
 (*AR_SimRead*, *tk*, 0, *buf*, *dvec*, *posk*)
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_read\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimCompute, k_unidx\ 0, 0,$
 $buf(tk := le_tm\ M), dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := Suc\ (mt_pos\ c'\ tk))$
proof –
have *src*: *fst* (*snd* (*mt_state* *c'*)) = *AR_SimRead* **using** *stg* **by** *simp*
show ?thesis
using *ar_exists_step_in_sub_read*
 [*OF* *ar_read_le_finish_step*[*OF* *vM* *stg* *qQ* *last* *posk_le* *aLE* *vsr*
pad_blank *src_bounded*] *src*] .
qed

lemma *ar_read_lookback1_step_in_sub*:
fixes *M* :: ('*q*, '*a*) *mttm*
and *c'* :: (*sym4*, '*q* $\times$ '*a* *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *stg*: *mt_state* *c'* = (*q*, *AR_SimRead*, *tk*, 0, *buf*, *dvec*, *posk*)
and *qQ*: *q* $\in Q_{tm}\ M$
and *posk_proper*: *posk* *tk* $\in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and *notLE*: *mt_tape* *c'* *tk* (*mt_pos* *c'* *tk*) $\neq LE4$
and *vsr*: *ar_valid_stage* ($\Gamma_{tm} M$) (*bl_tm* *M*)
 (*AR_SimRead*, *tk*, 0, *buf*, *dvec*, *posk*)
and *pad_blank*: $\forall j \geq k_{tm} M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and *src_bounded*: *ar_stage_bounded* (*bl_tm* *M*) (*k_tm* *M*)
 (*AR_SimRead*, *tk*, 0, *buf*, *dvec*, *posk*)
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_read\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimRead, tk, Suc\ 0, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := mt_pos\ c'\ tk - 1)$

proof –
have *src*: *fst* (*snd* (*mt_state* *c'*)) = *AR_SimRead* **using** *stg* **by** *simp*
show ?thesis
using *ar_exists_step_in_sub_read*
 [*OF* *ar_read_lookback1_step*[*OF* *vM* *stg* *qQ* *posk_proper* *notLE* *vsr*
pad_blank *src_bounded*] *src*] .
qed

lemma *ar_read_lookback2_step_in_sub*:
fixes *M* :: ('*q*, '*a*) *mttm*
and *c'* :: (*sym4*, '*q* $\times$ '*a* *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *stg*: *mt_state* *c'* = (*q*, *AR_SimRead*, *tk*, *Suc* 0, *buf*, *dvec*, *posk*)
and *qQ*: *q* $\in Q_{tm}\ M$

and *posk_proper*: *posk tk* $\in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and *kge2*: $2 \leq block_width (\Gamma_tm M)$
and *vsr*: *ar_valid_stage* $(\Gamma_tm M) (bl_tm M)$
 (*AR_SimRead*, *tk*, *Suc 0*, *buf*, *dvec*, *posk*)
and *pad_blank*: $\forall j \geq k_tm M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and *src_bounded*: *ar_stage_bounded* $(bl_tm M) (k_tm M)$
 (*AR_SimRead*, *tk*, *Suc 0*, *buf*, *dvec*, *posk*)
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_read\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimRead, tk, Suc\ (Suc\ 0),$
 $buf(tk := gamma_unenum\ (\Gamma_tm M) (bl_tm M)\ 0), dvec,$
 $posk(tk := if\ mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) = LE4$
 $then\ AR_AtFirstProper\ else\ AR_AtFurtherProper))$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := Suc\ (mt_pos\ c'\ tk))$
proof –
have *src*: *fst* (*snd* (*mt_state c'*)) = *AR_SimRead* **using** *stg* **by** *simp*
show *?thesis*
using *ar_exists_step_in_sub_read*
 $[OF\ ar_read_lookback2_step[OF\ vM\ stg\ qQ\ posk_proper\ kge2\ vsr$
pad_blank\ src_bounded] *src*].
qed

lemma *ar_read_bit_step_in_sub*:
fixes *M* :: $(q, 'a)\ mttm$
and *c'* :: $(sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes *vM*: *valid_mttm M*
and *stg*: *mt_state c'* = $(q, AR_SimRead, tk, i, buf, dvec, posk)$
and *qQ*: *q* $\in Q_tm\ M$
and *posk_proper*: *posk tk* $\in \{AR_AtFirstProper, AR_AtFurtherProper\}$
and *ilo*: $2 \leq i$
and *ihi*: *Suc i* $\leq Suc\ (block_width\ (\Gamma_tm M))$
and *kge2*: $2 \leq block_width\ (\Gamma_tm M)$
and *vsr*: *ar_valid_stage* $(\Gamma_tm M) (bl_tm M)$
 (*AR_SimRead*, *tk*, *i*, *buf*, *dvec*, *posk*)
and *pad_blank*: $\forall j \geq k_tm M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and *src_bounded*: *ar_stage_bounded* $(bl_tm M) (k_tm M)$
 (*AR_SimRead*, *tk*, *i*, *buf*, *dvec*, *posk*)
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_read\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimRead, tk, Suc\ i,$
 $buf(tk := gamma_unenum\ (\Gamma_tm M) (bl_tm M)$
 $(2 * gamma_enum\ (\Gamma_tm M) (bl_tm M) (buf\ tk)$
 $+ bit_value\ (mt_tape\ c'\ tk\ (mt_pos\ c'\ tk))))),$
 $dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := Suc\ (mt_pos\ c'\ tk))$
proof –
have *src*: *fst* (*snd* (*mt_state c'*)) = *AR_SimRead* **using** *stg* **by** *simp*
show *?thesis*
using *ar_exists_step_in_sub_read*

[OF ar_read_bit_step[OF vM stg qQ posk_proper ilo ihi kge2 vsrc
pad_blank src_bounded] src] .

qed

lemma ar_read_bit_boundary_step_in_sub:

fixes M :: ('q, 'a) mttm

and c' :: (sym4, 'q × 'a ar_stage) mt_config

assumes vM: valid_mttm M

and stg: mt_state c' = (q, AR_SimRead, tk, i, buf, dvec, posk)

and qQ: q ∈ Q_tm M

and notlast: ¬ is_last_k M tk

and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}

and ieq: i = Suc (block_width (Γ_tm M))

and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)

(AR_SimRead, tk, i, buf, dvec, posk)

and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4

and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)

(AR_SimRead, tk, i, buf, dvec, posk)

shows ∃ c''. (c', c'') ∈ mttm_step (ar_delta_read M)

∧ mt_state c'' = (q, AR_SimRead, k_succ tk, 0,

buf(tk := gamma_unenum (Γ_tm M) (bl_tm M)

(2 * gamma_enum (Γ_tm M) (bl_tm M) (buf tk)

+ bit_value (mt_tape c' tk (mt_pos c' tk))))),

dvec, posk)

∧ mt_tape c'' = mt_tape c'

∧ mt_pos c'' = (mt_pos c')(tk := Suc (mt_pos c' tk))

proof –

have src: fst (snd (mt_state c')) = AR_SimRead **using** stg **by** simp

show ?thesis

using ar_exists_step_in_sub_read

[OF ar_read_bit_boundary_step[OF vM stg qQ notlast posk_proper ieq

vsrc pad_blank src_bounded]

src] .

qed

lemma ar_read_bit_finish_step_in_sub:

fixes M :: ('q, 'a) mttm

and c' :: (sym4, 'q × 'a ar_stage) mt_config

assumes vM: valid_mttm M

and stg: mt_state c' = (q, AR_SimRead, tk, i, buf, dvec, posk)

and qQ: q ∈ Q_tm M

and last: is_last_k M tk

and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}

and ieq: i = Suc (block_width (Γ_tm M))

and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)

(AR_SimRead, tk, i, buf, dvec, posk)

and pad_blank: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4

and src_bounded: ar_stage_bounded (bl_tm M) (k_tm M)

(AR_SimRead, tk, i, buf, dvec, posk)

```

shows  $\exists c''. (c', c'') \in \text{mttm\_step } (ar\_delta\_read\ M)$ 
 $\wedge \text{mt\_state } c'' = (q, AR\_SimCompute, k\_unidx\ 0, 0,$ 
 $\text{buf}(tk := \text{gamma\_unenum } (\Gamma\_tm\ M) (bl\_tm\ M)$ 
 $(2 * \text{gamma\_enum } (\Gamma\_tm\ M) (bl\_tm\ M) (\text{buf } tk)$ 
 $+ \text{bit\_value } (\text{mt\_tape } c' tk (\text{mt\_pos } c' tk))),$ 
 $\text{dvec}, \text{posk})$ 
 $\wedge \text{mt\_tape } c'' = \text{mt\_tape } c'$ 
 $\wedge \text{mt\_pos } c'' = (\text{mt\_pos } c') (tk := \text{Suc } (\text{mt\_pos } c' tk))$ 
proof –
  have  $\text{src: fst } (\text{snd } (\text{mt\_state } c')) = AR\_SimRead$  using  $\text{stg by simp}$ 
  show  $?thesis$ 
  using  $ar\_exists\_step\_in\_sub\_read$ 
 $[OF\ ar\_read\_bit\_finish\_step[OF\ vM\ stg\ qQ\ last\ posk\_proper\ ieq\ vsrc$ 
 $pad\_blank\ src\_bounded]$ 
 $src]$  .
qed

```

3.13 Read-phase combinators, sub-relation chain variants

Multi-step combiner $_in_sub$ variants follow the same proof structure as the originals, but obtain their sub-chains from the leaf $_in_sub$ companions and compose via the generic $\text{relpow_Suc_I2}/\text{relpow_add}$ combinators (which work over any relation, in particular $\text{mttm_step } (ar_delta_read\ M)$). All bookkeeping for tape, position, state shape transfers verbatim from the originals.

```

lemma  $ar\_read\_bit\_loop\_in\_sub$ :
  fixes  $M :: ('q, 'a) \text{mttm}$ 
  and  $c' :: (\text{sym4}, 'q \times 'a\ ar\_stage) \text{mt\_config}$ 
  assumes  $vM: \text{valid\_mttm } M$ 
  and  $qQ: q \in Q\_tm\ M$ 
  and  $kge2: 2 \leq \text{block\_width } (\Gamma\_tm\ M)$ 
  and  $\text{posk\_proper: } \text{posk } tk \in \{AR\_AtFirstProper, AR\_AtFurtherProper\}$ 
  and  $\text{stg: } \text{mt\_state } c' = (q, AR\_SimRead, tk, 2, \text{buf0}, \text{dvec}, \text{posk})$ 
  and  $\text{buf0\_valid: } \forall k. \text{buf0 } k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  and  $\text{pos\_base: } \text{mt\_pos } c' tk = \text{base}$ 
  and  $\text{pad0: } \forall j \geq k\_tm\ M. \text{mt\_tape } c' j (\text{mt\_pos } c' j) = \text{BLANK4}$ 
  and  $\text{src0: } ar\_stage\_bounded (bl\_tm\ M) (k\_tm\ M)$ 
 $(AR\_SimRead, tk, 2, \text{buf0}, \text{dvec}, \text{posk})$ 
shows  $\exists c''. (c', c'') \in (\text{mttm\_step } (ar\_delta\_read\ M))$ 
 $\wedge \wedge (\text{block\_width } (\Gamma\_tm\ M) - 1)$ 
 $\wedge \text{mt\_state } c'' = (q, AR\_SimRead, tk, \text{Suc } (\text{block\_width } (\Gamma\_tm\ M)),$ 
 $\text{buf0}(tk := \text{foldl } (ar\_acc\ (\Gamma\_tm\ M) (bl\_tm\ M)) (\text{buf0 } tk)$ 
 $(\text{map } (\lambda m. \text{mt\_tape } c' tk (\text{base} + m))$ 
 $[0..<\text{block\_width } (\Gamma\_tm\ M) - 1])),$ 
 $\text{dvec}, \text{posk})$ 
 $\wedge \text{mt\_pos } c'' tk = \text{base} + (\text{block\_width } (\Gamma\_tm\ M) - 1)$ 
 $\wedge \text{mt\_tape } c'' = \text{mt\_tape } c'$ 
 $\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt\_pos } c'' k' = \text{mt\_pos } c' k')$ 

```


by (rule ar_read_bit_loop_gen
 [OF ar_read_bit_step_in_sub vM qQ kge2 posk_proper stg buf0_valid
 pos_base pad0 src0])

lemma ar_read_proper_prefix_in_sub:
 fixes M :: ('q, 'a) mttm
 and c' :: (sym4, 'q × 'a ar_stage) mt_config
 assumes vM: valid_mttm M
 and qQ: q ∈ Q_tm M
 and kge2: 2 ≤ block_width (Γ_tm M)
 and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
 and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
 and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
 and base_pos: 0 < base
 and pos_base: mt_pos c' tk = base
 and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
 (AR_SimRead, tk, 0, buf, dvec, posk)
 and pad0: ∀j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
 and src0: ar_stage_bounded (bl_tm M) (k_tm M)
 (AR_SimRead, tk, 0, buf, dvec, posk)
 shows ∃ c''. (c', c'') ∈ (mttm_step (ar_delta_read M))
 ^ ^ Suc (block_width (Γ_tm M))
 ∧ mt_state c'' = (q, AR_SimRead, tk, Suc (block_width (Γ_tm M)),
 buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M))
 (gamma_unenum (Γ_tm M) (bl_tm M) 0)
 (map (λm. mt_tape c' tk (base + m))
 [0.. $\text{block_width } (\Gamma_tm M) - 1$])),
 dvec,
 posk(tk := if mt_tape c' tk (base - 1) = LE4
 then AR_AtFirstProper else AR_AtFurtherProper))
 ∧ mt_tape c'' = mt_tape c'
 ∧ mt_pos c'' = (mt_pos c')(tk := base + (block_width (Γ_tm M) -
 1))
 by (rule ar_read_proper_prefix_gen
 [OF ar_read_lookback1_step_in_sub ar_read_lookback2_step_in_sub
 ar_read_bit_loop_in_sub
 vM qQ kge2 posk_proper stg notLE base_pos pos_base vsrc pad0 src0])

lemma ar_read_proper_step_in_sub:
 fixes M :: ('q, 'a) mttm
 and c' :: (sym4, 'q × 'a ar_stage) mt_config
 assumes vM: valid_mttm M
 and qQ: q ∈ Q_tm M
 and kge2: 2 ≤ block_width (Γ_tm M)
 and notlast: ¬ is_last_k M tk
 and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
 and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
 and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
 and base_pos: 0 < base

```

and pos_base: mt_pos c' tk = base
and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
          (AR_SimRead, tk, 0, buf, dvec, posk)
and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
and src0: ar_stage_bounded (bl_tm M) (k_tm M)
          (AR_SimRead, tk, 0, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ (mttm_step (ar_delta_read M))
              ^^ (block_width (Γ_tm M) + 2)
    ∧ mt_state c'' = (q, AR_SimRead, k_succ tk, 0,
      buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M))
        (gamma_unenum (Γ_tm M) (bl_tm M) 0)
        (map (λm. mt_tape c' tk (base + m))
          [0..by (rule ar_read_proper_step_gen
    [OF ar_read_proper_prefix_in_sub ar_read_bit_boundary_step_in_sub
      vM qQ kge2 notlast posk_proper stg notLE base_pos pos_base vsrc pad0
      src0])

```

lemma ar_read_proper_finish_step_in_sub:

```

fixes M :: ('q, 'a) mttm
and c' :: (sym4, 'q × 'a ar_stage) mt_config
assumes vM: valid_mttm M
and qQ: q ∈ Q_tm M
and kge2: 2 ≤ block_width (Γ_tm M)
and last: is_last_k M tk
and posk_proper: posk tk ∈ {AR_AtFirstProper, AR_AtFurtherProper}
and stg: mt_state c' = (q, AR_SimRead, tk, 0, buf, dvec, posk)
and notLE: mt_tape c' tk (mt_pos c' tk) ≠ LE4
and base_pos: 0 < base
and pos_base: mt_pos c' tk = base
and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
          (AR_SimRead, tk, 0, buf, dvec, posk)
and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
and src0: ar_stage_bounded (bl_tm M) (k_tm M)
          (AR_SimRead, tk, 0, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ (mttm_step (ar_delta_read M))
              ^^ (block_width (Γ_tm M) + 2)
    ∧ mt_state c'' = (q, AR_SimCompute, k_unidx 0, 0,
      buf(tk := foldl (ar_acc (Γ_tm M) (bl_tm M))
        (gamma_unenum (Γ_tm M) (bl_tm M) 0)
        (map (λm. mt_tape c' tk (base + m))
          [0..

```

$$\begin{aligned} & \text{then } AR_AtFirstProper \text{ else } AR_AtFurtherProper)) \\ & \wedge mt_tape\ c'' = mt_tape\ c' \\ & \wedge mt_pos\ c'' = (mt_pos\ c')(tk := base + block_width\ (\Gamma_tm\ M)) \\ \text{by } & (rule\ ar_read_proper_finish_step_gen \\ & [OF\ ar_read_proper_prefix_in_sub\ ar_read_bit_finish_step_in_sub \\ & \quad vM\ qQ\ kge2\ last\ posk_proper\ stg\ notLE\ base_pos\ pos_base\ vsrc\ pad0 \\ & \quad src0]) \end{aligned}$$

lemma *ar_read_tape_step_in_sub*:

$$\begin{aligned} \text{fixes } & M :: ('q, 'a)\ mttm \\ \text{and } & c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config \\ \text{and } & tM :: nat \Rightarrow 'a \\ \text{assumes } & vM: valid_mttm\ M \\ \text{and } & qQ: q \in Q_tm\ M \\ \text{and } & kge2: 2 \leq block_width\ (\Gamma_tm\ M) \\ \text{and } & notlast: \neg is_last_k\ M\ tk \\ \text{and } & stg: mt_state\ c' = (q, AR_SimRead, tk, 0, buf, dvec, posk) \\ \text{and } & tcorr: ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M) \\ & \quad tM\ (mt_tape\ c'\ tk) \\ \text{and } & ppos: mt_pos\ c'\ tk = sim_pos\ (block_width\ (\Gamma_tm\ M))\ p \\ \text{and } & pkok: posk\ tk = AR_AtLE \longleftrightarrow p = 0 \\ \text{and } & proper_mem: 1 \leq p \implies tM\ p \in \Gamma_tm\ M \\ \text{and } & vsrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M) \\ & \quad (AR_SimRead, tk, 0, buf, dvec, posk) \\ \text{and } & pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4 \\ \text{and } & src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M) \\ & \quad (AR_SimRead, tk, 0, buf, dvec, posk) \\ \text{shows } & \exists c''. (c', c'') \in (mttm_step\ (ar_delta_read\ M)) \\ & \quad \wedge (if\ p = 0\ then\ 1\ else\ block_width\ (\Gamma_tm\ M) + 2) \\ & \quad \wedge mt_state\ c'' = (q, AR_SimRead, k_succ\ tk, 0, \\ & \quad \quad buf(tk := tM\ p), dvec, \\ & \quad \quad posk(tk := if\ p = 0\ then\ AR_AtLE \\ & \quad \quad \quad else\ if\ p = 1\ then\ AR_AtFirstProper \\ & \quad \quad \quad else\ AR_AtFurtherProper)) \\ & \quad \wedge mt_tape\ c'' = mt_tape\ c' \\ & \quad \wedge mt_pos\ c'' = (mt_pos\ c')(tk := \\ & \quad \quad if\ p = 0\ then\ Suc\ 0 \\ & \quad \quad \quad else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ p + block_width\ (\Gamma_tm \\ & \quad \quad M)) \\ \text{by } & (rule\ ar_read_tape_step_gen \\ & [OF\ ar_read_le_step_in_sub\ ar_read_proper_step_in_sub \\ & \quad vM\ qQ\ kge2\ notlast\ stg\ tcorr\ ppos\ pkok\ proper_mem\ vsrc\ pad0\ src0]) \end{aligned}$$

lemma *ar_read_tape_finish_step_in_sub*:

$$\begin{aligned} \text{fixes } & M :: ('q, 'a)\ mttm \\ \text{and } & c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config \\ \text{and } & tM :: nat \Rightarrow 'a \\ \text{assumes } & vM: valid_mttm\ M \\ \text{and } & qQ: q \in Q_tm\ M \end{aligned}$$

and $kge2: 2 \leq \text{block_width } (\Gamma_tm\ M)$
and $\text{last}: \text{is_last_k } M\ tk$
and $\text{stg}: \text{mt_state } c' = (q, \text{AR_SimRead}, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
and $\text{tcorr}: \text{ar_tape_correspondence } (\Gamma_tm\ M) (\text{le_tm } M) (\text{bl_tm } M)$
 $\quad tM\ (\text{mt_tape } c'\ tk)$
and $\text{ppos}: \text{mt_pos } c'\ tk = \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ p$
and $\text{pkok}: \text{posk } tk = \text{AR_AtLE} \longleftrightarrow p = 0$
and $\text{proper_mem}: 1 \leq p \implies tM\ p \in \Gamma_tm\ M$
and $\text{vsrc}: \text{ar_valid_stage } (\Gamma_tm\ M) (\text{bl_tm } M)$
 $\quad (\text{AR_SimRead}, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
and $\text{pad0}: \forall j \geq k_tm\ M. \text{mt_tape } c'\ j\ (\text{mt_pos } c'\ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm\ M)$
 $\quad (\text{AR_SimRead}, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in (\text{mttm_step } (\text{ar_delta_read } M))$
 $\quad \wedge (if\ p = 0\ \text{then } 1\ \text{else } \text{block_width } (\Gamma_tm\ M) + 2)$
 $\quad \wedge \text{mt_state } c'' = (q, \text{AR_SimCompute}, k_unidx\ 0, 0,$
 $\quad \text{buf}(tk := tM\ p), \text{dvec},$
 $\quad \text{posk}(tk := if\ p = 0\ \text{then } \text{AR_AtLE}$
 $\quad \text{else if } p = 1\ \text{then } \text{AR_AtFirstProper}$
 $\quad \text{else } \text{AR_AtFurtherProper}))$
 $\quad \wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\quad \wedge \text{mt_pos } c'' = (\text{mt_pos } c')(tk :=$
 $\quad if\ p = 0\ \text{then } \text{Suc } 0$
 $\quad \text{else } \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ p + \text{block_width } (\Gamma_tm$
 $\quad M))$
by $(\text{rule } \text{ar_read_tape_finish_step_gen}$
 $\quad [OF\ \text{ar_read_le_finish_step_in_sub } \text{ar_read_proper_finish_step_in_sub}$
 $\quad vM\ qQ\ kge2\ \text{last}\ \text{stg}\ \text{tcorr}\ \text{ppos}\ \text{pkok}\ \text{proper_mem}\ \text{vsrc}\ \text{pad0}\ \text{src0}])$

lemma $\text{ar_read_prefix_in_sub}$:
fixes $M :: ('q, 'a)\ \text{mttm}$
and $cM :: ('a, 'q)\ \text{mt_config}$
and $c0 :: (\text{sym4}, 'q \times 'a\ \text{ar_stage})\ \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm\ M)$
and $\text{stg0}: \text{mt_state } c0 = (q, \text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$
and $\text{tcorr}: \forall k < k_tm\ M. \text{ar_tape_correspondence } (\Gamma_tm\ M) (\text{le_tm } M)$
 $\quad (\text{bl_tm } M)$
 $\quad (\text{mt_tape } cM\ k)\ (\text{mt_tape } c0\ k)$
and $\text{ppos}: \bigwedge k. \text{mt_pos } c0\ k = \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ (\text{mt_pos } cM\ k)$
and $\text{pkok}: \bigwedge k. \text{posk0 } k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM\ k = 0$
and $\text{tapeG}: \bigwedge k. \text{mt_tape } cM\ k\ (\text{mt_pos } cM\ k) \in \Gamma_tm\ M$
and $\text{bufG}: \bigwedge k. \text{buf0 } k \in \Gamma_tm\ M \cup \{\text{bl_tm } M\}$
and $\text{pad0}: \forall j \geq k_tm\ M. \text{mt_tape } c0\ j\ (\text{mt_pos } c0\ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M) (k_tm\ M)$
 $\quad (\text{AR_SimRead}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$
shows $j \leq k_tm\ M - 1 \implies$

$(\exists c\ m. (c0, c) \in (\text{mttm_step } (\text{ar_delta_read } M)) \wedge m$
 $\wedge m \leq j * (\text{block_width } (\Gamma_tm\ M) + 2)$
 $\wedge \text{mt_state } c = (q, \text{AR_SimRead}, k_unidx\ j, 0,$
 $(\lambda k. \text{if } k_idx\ k < j \text{ then } \text{mt_tape } cM\ k\ (\text{mt_pos } cM\ k) \text{ else } \text{buf0 } k),$
 $dvec,$
 $(\lambda k. \text{if } k_idx\ k < j$
 $\text{ then } (\text{if } \text{mt_pos } cM\ k = 0 \text{ then } \text{AR_AtLE}$
 $\text{ else if } \text{mt_pos } cM\ k = 1 \text{ then } \text{AR_AtFirstProper}$
 $\text{ else } \text{AR_AtFurtherProper})$
 $\text{ else } \text{posk0 } k))$
 $\wedge \text{mt_tape } c = \text{mt_tape } c0$
 $\wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx\ k < j$
 $\text{ then } (\text{if } \text{mt_pos } cM\ k = 0 \text{ then } \text{Suc } 0$
 $\text{ else } \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ (\text{mt_pos } cM\ k)$
 $\text{ + block_width } (\Gamma_tm\ M))$
 $\text{ else } \text{mt_pos } c0\ k))$
by (rule ar_read_prefix_gen
 $[OF_vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ pkok\ \text{tapeG}\ \text{bufG}\ \text{pad0}\ \text{src0}]$)
(rule ar_read_tape_step_in_sub; assumption)

lemma ar_read_phase_in_sub:

fixes $M :: ('q, 'a)\ \text{mttm}$
and $cM :: ('a, 'q)\ \text{mt_config}$
and $c0 :: (\text{sym4}, 'q \times 'a\ \text{ar_stage})\ \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm\ M)$
and $stg0: \text{mt_state } c0 = (q, \text{AR_SimRead}, 0, 0, \text{buf0}, dvec, \text{posk0})$
and $tcorr: \forall k < k_tm\ M. \text{ar_tape_correspondence } (\Gamma_tm\ M)\ (\text{le_tm } M)$
 $(\text{bl_tm } M)$
 $(\text{mt_tape } cM\ k)\ (\text{mt_tape } c0\ k)$
and $ppos: \bigwedge k. \text{mt_pos } c0\ k = \text{sim_pos } (\text{block_width } (\Gamma_tm\ M))\ (\text{mt_pos } cM\ k)$
and $pkok: \bigwedge k. \text{posk0 } k = \text{AR_AtLE} \longleftrightarrow \text{mt_pos } cM\ k = 0$
and $\text{tapeG}: \bigwedge k. \text{mt_tape } cM\ k\ (\text{mt_pos } cM\ k) \in \Gamma_tm\ M$
and $\text{bufG}: \bigwedge k. \text{buf0 } k \in \Gamma_tm\ M \cup \{\text{bl_tm } M\}$
and $\text{pad0}: \forall j \geq k_tm\ M. \text{mt_tape } c0\ j\ (\text{mt_pos } c0\ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M)\ (k_tm\ M)$
 $(\text{AR_SimRead}, 0, 0, \text{buf0}, dvec, \text{posk0})$
shows $\exists c\ m. (c0, c) \in (\text{mttm_step } (\text{ar_delta_read } M)) \wedge m$
 $\wedge m \leq k_tm\ M * (\text{block_width } (\Gamma_tm\ M) + 2)$
 $\wedge \text{mt_state } c = (q, \text{AR_SimCompute}, k_unidx\ 0, 0,$
 $(\lambda k. \text{if } k < k_tm\ M \text{ then } \text{mt_tape } cM\ k\ (\text{mt_pos } cM\ k) \text{ else } \text{buf0 } k),$
 $dvec,$
 $(\lambda k. \text{if } k < k_tm\ M$
 $\text{ then } (\text{if } \text{mt_pos } cM\ k = 0 \text{ then } \text{AR_AtLE}$
 $\text{ else if } \text{mt_pos } cM\ k = 1 \text{ then } \text{AR_AtFirstProper}$
 $\text{ else } \text{AR_AtFurtherProper})$
 $\text{ else } \text{posk0 } k))$

$\wedge mt_tape\ c = mt_tape\ c0$
 $\wedge mt_pos\ c = (\lambda k. \text{if } k < k_tm\ M$
 $\quad \text{then } (if\ mt_pos\ cM\ k = 0\ \text{then } Suc\ 0$
 $\quad \text{else } sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k) + block_width$
 $(\Gamma_tm\ M))$
 $\quad \text{else } mt_pos\ c0\ k)$
by (*rule* *ar_read_phase_gen*
 $[OF_ _ vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ pkok\ tapeG\ bufG\ pad0\ src0];$
 $(rule\ ar_read_prefix_in_sub\ ar_read_tape_finish_step_in_sub; \text{assumption})$)

3.14 Write-phase leaves, sub-relation chain variants

Six write-phase leaves mirror to the sub-relation *mttm_step* (*ar_delta_write* *M*) via *ar_step_write_lift*, parallel to the seven read leaves at the earlier subsection. Each variant obtains the step from the original leaf, derives the *AR_SimWrite* source tag from the *stg* hypothesis, and lifts via *ar_step_write_lift*.

lemma *ar_write_le_step_in_sub*:

fixes *M* :: ('q, 'a) *mttm*

and *c'* :: (*sym4*, 'q $\times$ 'a *ar_stage*) *mt_config*

assumes *vM*: *valid_mttm M*

and *stg*: *mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)*

and *qQ*: *q* $\in Q_tm\ M$

and *poskLE*: *posk tk = AR_AtLE*

and *notlast*: $\neg is_last_k\ M\ tk$

and *vsr*: *ar_valid_stage* ($\Gamma_tm\ M$) (*bl_tm M*)
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$

and *pad_blank*: $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$

and *src_bounded*: *ar_stage_bounded* (*bl_tm M*) (*k_tm M*)
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$

shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_write\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = mt_pos\ c'$

proof –

have *src*: *fst (snd (mt_state c')) = AR_SimWrite* **using** *stg* **by** *simp*

show *?thesis*

using *ar_exists_step_in_sub_write*
 $[OF\ ar_write_le_step[OF\ vM\ stg\ qQ\ poskLE\ notlast\ vsr\ pad_blank$
 $src_bounded]\ src]$.

qed

lemma *ar_write_le_finish_step_in_sub*:

fixes *M* :: ('q, 'a) *mttm*

and *c'* :: (*sym4*, 'q $\times$ 'a *ar_stage*) *mt_config*

assumes *vM*: *valid_mttm M*

and *stg*: *mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)*

and $qQ: q \in Q_tm\ M$
and $poskLE: posk\ tk = AR_AtLE$
and $last: is_last_k\ M\ tk$
and $vsrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
and $pad_blank: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src_bounded: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_write\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = mt_pos\ c'$
proof –
have $src: fst\ (snd\ (mt_state\ c')) = AR_SimWrite$ **using** stg **by** $simp$
show $?thesis$
using $ar_exists_step_in_sub_write$
 $[OF\ ar_write_le_finish_step[OF\ vM\ stg\ qQ\ poskLE\ last\ vsrc\ pad_blank$
 $src_bounded]\ src]$.
qed

lemma $ar_write_walk_step_in_sub$:
fixes $M :: ('q, 'a)\ mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes $vM: valid_mttm\ M$
and $stg: mt_state\ c' = (q, AR_SimWrite, tk, i, buf, dvec, posk)$
and $qQ: q \in Q_tm\ M$
and $poskproper: posk\ tk \neq AR_AtLE$
and $notLE: mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) \neq LE4$
and $step_le: Suc\ i \leq block_width\ (\Gamma_tm\ M)$
and $vsrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimWrite, tk, i, buf, dvec, posk)$
and $pad_blank: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src_bounded: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, i, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_write\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimWrite, tk, Suc\ i, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := mt_pos\ c'\ tk - 1)$
proof –
have $src: fst\ (snd\ (mt_state\ c')) = AR_SimWrite$ **using** stg **by** $simp$
show $?thesis$
using $ar_exists_step_in_sub_write$
 $[OF\ ar_write_walk_step[OF\ vM\ stg\ qQ\ poskproper\ notLE\ step_le\ vsrc$
 $pad_blank\ src_bounded]\ src]$.
qed

lemma $ar_write_bit_step_in_sub$:
fixes $M :: ('q, 'a)\ mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$

assumes vM : $\text{valid_mttm } M$
and stg : $\text{mt_state } c' = (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and qQ : $q \in Q_tm M$
and poskproper : $\text{posk } tk \neq \text{AR_AtLE}$
and notLE : $\text{mt_tape } c' tk (\text{mt_pos } c' tk) \neq \text{LE4}$
and ilo : $\text{block_width } (\Gamma_tm M) \leq i$
and ihi : $\text{Suc } i < 2 * \text{block_width } (\Gamma_tm M)$
and vsrce : $\text{ar_valid_stage } (\Gamma_tm M) (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and pad_blank : $\forall j \geq k_tm M. \text{mt_tape } c' j (\text{mt_pos } c' j) = \text{BLANK4}$
and src_bounded : $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in \text{mttm_step } (\text{ar_delta_write } M)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimWrite}, tk, \text{Suc } i, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (\text{mt_tape } c') (tk :=$
 $(\text{mt_tape } c' tk) (\text{mt_pos } c' tk :=$
 $\text{write_bit } (\Gamma_tm M) (\text{bl_tm } M) (\text{buf } tk)$
 $(i - \text{block_width } (\Gamma_tm M))))$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c') (tk := \text{Suc } (\text{mt_pos } c' tk))$
proof –
have src : $\text{fst } (\text{snd } (\text{mt_state } c')) = \text{AR_SimWrite}$ **using** stg **by** simp
show $?thesis$
using $\text{ar_exists_step_in_sub_write}$
 $[\text{OF } \text{ar_write_bit_step} [\text{OF } vM \text{ stg } qQ \text{ poskproper notLE ilo ihi vsrce}$
 $\text{pad_blank src_bounded}] \text{src}]$.
qed

lemma $\text{ar_write_bit_boundary_step_in_sub}$:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config}$
assumes vM : $\text{valid_mttm } M$
and stg : $\text{mt_state } c' = (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and qQ : $q \in Q_tm M$
and poskproper : $\text{posk } tk \neq \text{AR_AtLE}$
and notLE : $\text{mt_tape } c' tk (\text{mt_pos } c' tk) \neq \text{LE4}$
and notlast : $\neg \text{is_last_k } M tk$
and ihi : $\text{Suc } i = 2 * \text{block_width } (\Gamma_tm M)$
and vsrce : $\text{ar_valid_stage } (\Gamma_tm M) (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and pad_blank : $\forall j \geq k_tm M. \text{mt_tape } c' j (\text{mt_pos } c' j) = \text{BLANK4}$
and src_bounded : $\text{ar_stage_bounded } (\text{bl_tm } M) (k_tm M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in \text{mttm_step } (\text{ar_delta_write } M)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimWrite}, k_succ tk, 0, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (\text{mt_tape } c') (tk :=$
 $(\text{mt_tape } c' tk) (\text{mt_pos } c' tk :=$
 $\text{write_bit } (\Gamma_tm M) (\text{bl_tm } M) (\text{buf } tk)$
 $(i - \text{block_width } (\Gamma_tm M))))$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c') (tk := \text{Suc } (\text{mt_pos } c' tk))$

proof –
have $\text{src} : \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimWrite}$ **using** stg **by** simp
show $?thesis$
using $\text{ar_exists_step_in_sub_write}$
 $[\text{OF } \text{ar_write_bit_boundary_step}$
 $[\text{OF } vM \text{ stg } qQ \text{ poskproper notLE notlast } ihi \text{ vsrc } \text{pad_blank}$
 $\text{src_bounded}] \text{ src}]$.
qed

lemma $\text{ar_write_bit_finish_step_in_sub}$:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ ar_stage}) \text{mt_config}$
assumes $vM : \text{valid_mttm } M$
and $\text{stg} : \text{mt_state } c' = (q, \text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and $qQ : q \in Q_tm \ M$
and $\text{poskproper} : \text{posk } tk \neq \text{AR_AtLE}$
and $\text{notLE} : \text{mt_tape } c' \ tk \ (\text{mt_pos } c' \ tk) \neq \text{LE4}$
and $\text{last} : \text{is_last_k } M \ tk$
and $ihi : \text{Suc } i = 2 * \text{block_width } (\Gamma_tm \ M)$
and $\text{vsrc} : \text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and $\text{pad_blank} : \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = \text{BLANK4}$
and $\text{src_bounded} : \text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$
 $(\text{AR_SimWrite}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in \text{mttm_step } (\text{ar_delta_write } M)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimAdvance}, k_unidx \ 0, 0, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (\text{mt_tape } c') (tk :=$
 $(\text{mt_tape } c' \ tk) (\text{mt_pos } c' \ tk :=$
 $\text{write_bit } (\Gamma_tm \ M) \ (\text{bl_tm } M) \ (\text{buf } tk)$
 $(i - \text{block_width } (\Gamma_tm \ M))))$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c') (tk := \text{Suc } (\text{mt_pos } c' \ tk))$

proof –
have $\text{src} : \text{fst} (\text{snd} (\text{mt_state } c')) = \text{AR_SimWrite}$ **using** stg **by** simp
show $?thesis$
using $\text{ar_exists_step_in_sub_write}$
 $[\text{OF } \text{ar_write_bit_finish_step}$
 $[\text{OF } vM \text{ stg } qQ \text{ poskproper notLE last } ihi \text{ vsrc } \text{pad_blank } \text{src_bounded}]$
 $\text{src}]$.
qed

3.15 Advance-phase leaves, sub-relation chain variants

Three advance-phase leaves $(\text{ar_advance_walk_step},$
 $\text{ar_advance_boundary_step}, \text{ar_advance_finish_step})$ mirror
to the sub-relation $\text{mttm_step } (\text{ar_delta_advance } M)$ via
 $\text{ar_exists_step_in_sub_advance}$, parallel to the write and read leaves
above. Each variant derives the AR_SimAdvance source tag from stg , then
composes the original leaf with the existence lifter.

lemma *ar_advance_walk_step_in_sub*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $\text{stg}: \text{mt_state } c' = (q, \text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and $qQ: q \in Q_tm \ M$
and $\text{notLE}: \text{mt_tape } c' \ tk \ (\text{mt_pos } c' \ tk) \neq \text{LE4}$
and $\text{step_lt}: \text{Suc } i < \text{ar_disp } (\text{block_width } (\Gamma_tm \ M)) \ (\text{dvec } tk) \ (\text{posk } tk)$
and $\text{vsrc}: \text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$
 $(\text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and $\text{pad_blank}: \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = \text{BLANK4}$
and $\text{src_bounded}: \text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$
 $(\text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in \text{mttm_step } (\text{ar_delta_advance } M)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimAdvance}, tk, \text{Suc } i, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c') (tk := \text{mt_pos } c' \ tk - 1)$
proof –
have $\text{src}: \text{fst } (\text{snd } (\text{mt_state } c')) = \text{AR_SimAdvance}$ **using** stg **by** simp
show $?thesis$
using $\text{ar_exists_step_in_sub_advance}$
 $[\text{OF } \text{ar_advance_walk_step} [\text{OF } vM \ \text{stg } qQ \ \text{notLE } \text{step_lt } \text{vsrc } \text{pad_blank}$
 $\text{src_bounded}] \ \text{src}]$.
qed

lemma *ar_advance_boundary_step_in_sub*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $\text{stg}: \text{mt_state } c' = (q, \text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and $qQ: q \in Q_tm \ M$
and $\text{notlast}: \neg \text{is_last_k } M \ tk$
and $\text{fire}: (\text{dvec } tk = \text{dir}.R \wedge i = 0)$
 $\vee (\text{dvec } tk \neq \text{dir}.R$
 $\wedge \text{mt_tape } c' \ tk \ (\text{mt_pos } c' \ tk) \neq \text{LE4}$
 $\wedge \text{Suc } i = \text{ar_disp } (\text{block_width } (\Gamma_tm \ M)) \ (\text{dvec } tk) \ (\text{posk } tk))$
and $\text{vsrc}: \text{ar_valid_stage } (\Gamma_tm \ M) \ (\text{bl_tm } M)$
 $(\text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
and $\text{pad_blank}: \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = \text{BLANK4}$
and $\text{src_bounded}: \text{ar_stage_bounded } (\text{bl_tm } M) \ (k_tm \ M)$
 $(\text{AR_SimAdvance}, tk, i, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in \text{mttm_step } (\text{ar_delta_advance } M)$
 $\wedge \text{mt_state } c'' = (q, \text{AR_SimAdvance}, k_succ \ tk, 0, \text{buf}, \text{dvec},$
 $\text{posk}(tk := \text{ar_newpos } (\text{dvec } tk) \ (\text{posk } tk)))$
 $\wedge \text{mt_tape } c'' = \text{mt_tape } c'$
 $\wedge \text{mt_pos } c'' = (\text{if } \text{dvec } tk = \text{dir}.R \text{ then } \text{mt_pos } c'$
 $\text{else } (\text{mt_pos } c') (tk := \text{mt_pos } c' \ tk - 1))$
proof –
have $\text{src}: \text{fst } (\text{snd } (\text{mt_state } c')) = \text{AR_SimAdvance}$ **using** stg **by** simp

show ?thesis
using *ar_exists_step_in_sub_advance*
 $[OF\ ar_advance_boundary_step[OF\ vM\ stg\ qQ\ notlast\ fire\ vsrc\ pad_blank\ src_bounded]\ src]$.
qed

lemma *ar_advance_finish_step_in_sub*:
fixes $M :: ('q, 'a)\ mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes $vM: valid_mttm\ M$
and $stg: mt_state\ c' = (q, AR_SimAdvance, tk, i, buf, dvec, posk)$
and $qQ: q \in Q_tm\ M$
and $last: is_last_k\ M\ tk$
and $fire: (dvec\ tk = dir.R \wedge i = 0)$
 $\vee (dvec\ tk \neq dir.R$
 $\wedge mt_tape\ c'\ tk\ (mt_pos\ c'\ tk) \neq LE4$
 $\wedge Suc\ i = ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk))$
and $vsrc: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk)$
and $pad_blank: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src_bounded: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, i, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in mttm_step\ (ar_delta_advance\ M)$
 $\wedge mt_state\ c'' = (q, AR_SimNext, k_unidx\ 0, 0, buf, dvec,$
 $posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
 $\wedge mt_tape\ c'' = mt_tape\ c'$
 $\wedge mt_pos\ c'' = (if\ dvec\ tk = dir.R\ then\ mt_pos\ c'$
 $else\ (mt_pos\ c')(tk := mt_pos\ c'\ tk - 1))$

proof –

have $src: fst\ (snd\ (mt_state\ c')) = AR_SimAdvance$ **using** stg **by** *simp*
show ?thesis
using *ar_exists_step_in_sub_advance*
 $[OF\ ar_advance_finish_step[OF\ vM\ stg\ qQ\ last\ fire\ vsrc\ pad_blank\ src_bounded]\ src]$.
qed

3.16 Write-phase combinators, sub-relation chain variants

Write-phase multi-step combiner *_in_sub* variants follow the same proof structure as the originals, obtaining sub-chains from the write-leaf *_in_sub* companions and composing via the generic *relpow_Suc_I2* / *relpow_invariant_chain* combinators (which work over any relation, in particular *mttm_step* (*ar_delta_write* M)).

lemma *ar_write_back_loop_in_sub*:
fixes $M :: ('q, 'a)\ mttm$
and $c0 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$

and $kge2: 2 \leq \text{block_width } (\Gamma_tm\ M)$
and $\text{poskproper}: \text{posk } tk \neq \text{AR_AtLE}$
and $\text{stg}: \text{mt_state } c0 = (q, \text{AR_SimWrite}, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
and $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm\ M \cup \{\text{bl_tm } M\}$
and $\text{pos_base}: \text{mt_pos } c0\ tk = \text{base} + \text{block_width } (\Gamma_tm\ M)$
and $\text{notLE}: \bigwedge m. \llbracket 1 \leq m; m \leq \text{block_width } (\Gamma_tm\ M) \rrbracket$
 $\implies \text{mt_tape } c0\ tk\ (\text{base} + m) \neq \text{LE4}$
and $\text{pad0}: \forall j \geq k_tm\ M. \text{mt_tape } c0\ j\ (\text{mt_pos } c0\ j) = \text{BLANK4}$
and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M)\ (k_tm\ M)$
 $(\text{AR_SimWrite}, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c'. (c0, c') \in (\text{mttm_step } (\text{ar_delta_write } M))$
 $\wedge \wedge \text{block_width } (\Gamma_tm\ M)$
 $\wedge \text{mt_state } c' = (q, \text{AR_SimWrite}, tk, \text{block_width } (\Gamma_tm\ M),$
 $\text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_pos } c'\ tk = \text{base}$
 $\wedge \text{mt_tape } c' = \text{mt_tape } c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c'\ k' = \text{mt_pos } c0\ k')$
by $(\text{rule ar_write_back_loop_gen}$
 $[\text{OF ar_write_walk_step_in_sub } vM\ qQ\ kge2\ \text{poskproper}\ \text{stg}\ \text{buf_valid}$
 $\text{pos_base notLE pad0 src0}])$

lemma $\text{ar_write_fwd_loop_in_sub}$:

fixes $M :: ('q, 'a)\ \text{mttm}$

and $c0 :: (\text{sym4}, 'q \times 'a\ \text{ar_stage})\ \text{mt_config}$

assumes $vM: \text{valid_mttm } M$

and $qQ: q \in Q_tm\ M$

and $kge2: 2 \leq \text{block_width } (\Gamma_tm\ M)$

and $\text{poskproper}: \text{posk } tk \neq \text{AR_AtLE}$

and $\text{stg}: \text{mt_state } c0 = (q, \text{AR_SimWrite}, tk, \text{block_width } (\Gamma_tm\ M),$
 $\text{buf}, \text{dvec}, \text{posk})$

and $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm\ M \cup \{\text{bl_tm } M\}$

and $\text{pos_base}: \text{mt_pos } c0\ tk = \text{base}$

and $\text{notLE}: \bigwedge m. m < \text{block_width } (\Gamma_tm\ M)$

$\implies \text{mt_tape } c0\ tk\ (\text{base} + m) \neq \text{LE4}$

and $\text{pad0}: \forall j \geq k_tm\ M. \text{mt_tape } c0\ j\ (\text{mt_pos } c0\ j) = \text{BLANK4}$

and $\text{src0}: \text{ar_stage_bounded } (\text{bl_tm } M)\ (k_tm\ M)$

$(\text{AR_SimWrite}, tk, \text{block_width } (\Gamma_tm\ M), \text{buf}, \text{dvec}, \text{posk})$

shows $\exists c'. (c0, c') \in (\text{mttm_step } (\text{ar_delta_write } M))$

$\wedge \wedge (\text{block_width } (\Gamma_tm\ M) - 1)$

$\wedge \text{mt_state } c' = (q, \text{AR_SimWrite}, tk,$
 $\text{block_width } (\Gamma_tm\ M) + (\text{block_width } (\Gamma_tm\ M) - 1),$
 $\text{buf}, \text{dvec}, \text{posk})$

$\wedge \text{mt_pos } c'\ tk = \text{base} + (\text{block_width } (\Gamma_tm\ M) - 1)$

$\wedge \text{mt_tape } c'\ tk = (\lambda \text{pos.}$

$\text{if } \text{base} \leq \text{pos} \wedge \text{pos} < \text{base} + (\text{block_width } (\Gamma_tm\ M) - 1)$

$\text{then write_bit } (\Gamma_tm\ M)\ (\text{bl_tm } M)\ (\text{buf } tk)\ (\text{pos} - \text{base})$

$\text{else mt_tape } c0\ tk\ \text{pos})$

$\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_tape } c'\ k' = \text{mt_tape } c0\ k')$

$\wedge (\forall k'. k' \neq tk \longrightarrow \text{mt_pos } c'\ k' = \text{mt_pos } c0\ k')$

by (*rule* *ar_write_fwd_loop_gen*
 $[OF\ ar_write_bit_step_in_sub\ vM\ qQ\ kge2\ poskproper\ stg\ buf_valid$
 $pos_base\ notLE\ pad0\ src0])$

The proper-cell single-tape write prefix, re-mirrored into *mttm_step* (*ar_delta_write* *M*). Composes the back-walk loop (*ar_write_back_loop_in_sub*) and the forward bit-write loop (*ar_write_fwd_loop_in_sub*) by *relcompI* + *relpow_add*; since both *_in_sub* sub-combiners carry field-for-field the same output contract as the originals, the composition and the two *ext* reassemblies (tape, pos) transfer verbatim with only the relation and the two helper calls changed.

lemma *ar_write_proper_prefix_in_sub*:

fixes *M* :: ('q, 'a) *mttm*
and *c'* :: (*sym4*, 'q × 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *qQ*: *q* ∈ *Q_tm* *M*
and *kge2*: $2 \leq block_width\ (\Gamma_tm\ M)$
and *poskproper*: *posk* *tk* ≠ *AR_AtLE*
and *stg*: *mt_state* *c'* = (*q*, *AR_SimWrite*, *tk*, 0, *buf*, *dvec*, *posk*)
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *pos_base*: *mt_pos* *c'* *tk* = *base* + *block_width* (*Γ_tm* *M*)
and *notLE*: $\bigwedge m. m \leq block_width\ (\Gamma_tm\ M)$
 $\implies mt_tape\ c'\ tk\ (base + m) \neq LE4$
and *pad0*: $\forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and *src0*: *ar_stage_bounded* (*bl_tm* *M*) (*k_tm* *M*)
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c2. (c', c2) \in (mttm_step\ (ar_delta_write\ M))$
 $\wedge\wedge (block_width\ (\Gamma_tm\ M) + (block_width\ (\Gamma_tm\ M) -$
 $1))$
 $\wedge\ mt_state\ c2 = (q, AR_SimWrite, tk,$
 $block_width\ (\Gamma_tm\ M) + (block_width\ (\Gamma_tm\ M) - 1),$
 $buf, dvec, posk)$
 $\wedge\ mt_tape\ c2 = (mt_tape\ c')(tk := (\lambda pos.$
 $if\ base \leq pos \wedge pos < base + (block_width\ (\Gamma_tm\ M) - 1)$
 $then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ tk)\ (pos - base)$
 $else\ mt_tape\ c'\ tk\ pos))$
 $\wedge\ mt_pos\ c2 = (mt_pos\ c')(tk := base + (block_width\ (\Gamma_tm\ M) -$
 $1))$

by (*rule* *ar_write_proper_prefix_gen*
 $[OF\ ar_write_back_loop_in_sub\ ar_write_fwd_loop_in_sub$
 $vM\ qQ\ kge2\ poskproper\ stg\ buf_valid\ pos_base\ notLE\ pad0\ src0])$

The proper-cell single-tape write, non-last tape, re-mirrored into *mttm_step* (*ar_delta_write* *M*). As *ar_write_proper_prefix_in_sub* followed by one bit-boundary step (*ar_write_bit_boundary_step_in_sub*), composing by *relpow_Suc_I*. The shared *write_block_extend* helper is relation-agnostic (pure *fun_upd* arithmetic) and reused verbatim; the body transfers from the original with only the relation and the two helper refer-

ences changed.

lemma *ar_write_proper_step_in_sub*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $\text{notlast}: \neg \text{is_last_k } M \ tk$
and $\text{poskproper}: \text{posk } tk \neq AR_AtLE$
and $\text{stg}: \text{mt_state } c' = (q, AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
and $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm \ M \cup \{bl_tm \ M\}$
and $\text{pos_base}: \text{mt_pos } c' \ tk = \text{base} + \text{block_width } (\Gamma_tm \ M)$
and $\text{notLE}: \bigwedge m. m \leq \text{block_width } (\Gamma_tm \ M)$
 $\implies \text{mt_tape } c' \ tk \ (\text{base} + m) \neq LE4$
and $\text{pad0}: \forall j \geq k_tm \ M. \text{mt_tape } c' \ j \ (\text{mt_pos } c' \ j) = BLANK4$
and $\text{src0}: \text{ar_stage_bounded } (bl_tm \ M) \ (k_tm \ M)$
 $(AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
shows $\exists c''. (c', c'') \in (\text{mttm_step } (\text{ar_delta_write } M))$
 $\wedge (2 * \text{block_width } (\Gamma_tm \ M))$
 $\wedge \text{mt_state } c'' = (q, AR_SimWrite, k_succ \ tk, 0, \text{buf}, \text{dvec}, \text{posk})$
 $\wedge \text{mt_tape } c'' = (\text{mt_tape } c') (tk := (\lambda pos.$
 $\text{if } \text{base} \leq pos \wedge pos < \text{base} + \text{block_width } (\Gamma_tm \ M)$
 $\text{then } \text{write_bit } (\Gamma_tm \ M) \ (bl_tm \ M) \ (\text{buf } tk) \ (pos - \text{base})$
 $\text{else } \text{mt_tape } c' \ tk \ pos))$
 $\wedge \text{mt_pos } c'' = (\text{mt_pos } c') (tk := \text{base} + \text{block_width } (\Gamma_tm \ M))$
by (*rule ar_write_proper_step_gen*
 $[OF \ \text{ar_write_proper_prefix_in_sub} \ \text{ar_write_bit_boundary_step_in_sub}$
 $vM \ qQ \ kge2 \ \text{notlast} \ \text{poskproper} \ \text{stg} \ \text{buf_valid} \ \text{pos_base} \ \text{notLE} \ \text{pad0} \ \text{src0}])$

The proper-cell single-tape write, last tape, re-mirrored into *mttm_step* (*ar_delta_write* M). As *ar_write_proper_step_in_sub* but tk is the last tape, so the closing step is *ar_write_bit_finish_step_in_sub*: after the last-cell write the phase transitions to *AR_SimAdvance* with the current-tape field reset to $k_unidx \ 0$. Same $2b$ -step block write and head return to $\text{base} + b$; body transfers verbatim with only the relation and the two helper references changed.

lemma *ar_write_proper_finish_step_in_sub*:
fixes $M :: ('q, 'a) \text{mttm}$
and $c' :: (\text{sym4}, 'q \times 'a \text{ar_stage}) \text{mt_config}$
assumes $vM: \text{valid_mttm } M$
and $qQ: q \in Q_tm \ M$
and $kge2: 2 \leq \text{block_width } (\Gamma_tm \ M)$
and $\text{last}: \text{is_last_k } M \ tk$
and $\text{poskproper}: \text{posk } tk \neq AR_AtLE$
and $\text{stg}: \text{mt_state } c' = (q, AR_SimWrite, tk, 0, \text{buf}, \text{dvec}, \text{posk})$
and $\text{buf_valid}: \forall k. \text{buf } k \in \Gamma_tm \ M \cup \{bl_tm \ M\}$
and $\text{pos_base}: \text{mt_pos } c' \ tk = \text{base} + \text{block_width } (\Gamma_tm \ M)$
and $\text{notLE}: \bigwedge m. m \leq \text{block_width } (\Gamma_tm \ M)$

$\implies mt_tape\ c'\ tk\ (base + m) \neq LE4$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in (mttm_step\ (ar_delta_write\ M))$
 $\wedge (2 * block_width\ (\Gamma_tm\ M))$
 $\wedge mt_state\ c'' = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = (mt_tape\ c')(tk := (\lambda pos.$
 $\quad if\ base \leq pos \wedge pos < base + block_width\ (\Gamma_tm\ M)$
 $\quad then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ tk)\ (pos - base)$
 $\quad else\ mt_tape\ c'\ tk\ pos))$
 $\wedge mt_pos\ c'' = (mt_pos\ c')(tk := base + block_width\ (\Gamma_tm\ M))$
by $(rule\ ar_write_proper_finish_step_gen$
 $\quad [OF\ ar_write_proper_prefix_in_sub\ ar_write_bit_finish_step_in_sub$
 $\quad vM\ qQ\ kge2\ last\ poskproper\ stg\ buf_valid\ pos_base\ notLE\ pad0\ src0])$

The unified single-tape write, non-last tape, re-mirrored into $mttm_step\ (ar_delta_write\ M)$: the LE/proper dispatch on $posk\ tk = AR_AtLE$ (pinned by $poskle$ to $p = 0$). The LE arm ($ar_write_le_step_in_sub, 1$ step) hands off untouched; the proper arm ($ar_write_proper_step_in_sub, 2b$ steps) overwrites the b -cell block. Head invariant in both arms. The $\neq LE4$ facts come from the input correspondence $tcrr$; that derivation is relation-agnostic and transfers verbatim along with the relation and two helper references changing.

lemma $ar_write_tape_step_in_sub$:

fixes $M :: ('q, 'a) mttm$
and $c' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
and $tM :: nat \Rightarrow 'a$
assumes $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $notlast: \neg is_last_k\ M\ tk$
and $stg: mt_state\ c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)$
and $tcrr: ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)\ (bl_tm\ M)$
 $tM\ (mt_tape\ c'\ tk)$
and $ppos: mt_pos\ c'\ tk = (if\ p = 0\ then\ Suc\ 0$
 $\quad else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ p + block_width\ (\Gamma_tm$
 $M))$
and $poskle: posk\ tk = AR_AtLE \longleftrightarrow p = 0$
and $buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $vsr: ar_valid_stage\ (\Gamma_tm\ M)\ (bl_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c'\ j\ (mt_pos\ c'\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimWrite, tk, 0, buf, dvec, posk)$
shows $\exists c''. (c', c'') \in (mttm_step\ (ar_delta_write\ M))$
 $\wedge (if\ p = 0\ then\ 1\ else\ 2 * block_width\ (\Gamma_tm\ M))$
 $\wedge mt_state\ c'' = (q, AR_SimWrite, k_succ\ tk, 0, buf, dvec, posk)$
 $\wedge mt_tape\ c'' = (if\ p = 0\ then\ mt_tape\ c'$

```

else (mt_tape c')(tk := (λpos.
  if sim_pos (block_width (Γ_tm M)) p ≤ pos
    ∧ pos < sim_pos (block_width (Γ_tm M)) p + block_width
(Γ_tm M)
    then write_bit (Γ_tm M) (bl_tm M) (buf tk)
      (pos - sim_pos (block_width (Γ_tm M)) p)
    else mt_tape c' tk pos)))
  ∧ mt_pos c'' = mt_pos c'
by (rule ar_write_tape_step_gen
  [OF ar_write_le_step_in_sub ar_write_proper_step_in_sub
  vM qQ kge2 notlast stg tcorr ppos poskle buf_valid vsrc pad0 src0])

```

The unified single-tape write, last tape, re-mirrored into *mttm_step* (*ar_delta_write* *M*): as *ar_write_tape_step_in_sub* but *tk* is the last tape, so both arms transition to *AR_SimAdvance* (current-tape field reset to *k_unidx 0*): the LE arm via *ar_write_le_finish_step_in_sub*, the proper arm via *ar_write_proper_finish_step_in_sub*. Same tape edit and head invariance; body transfers verbatim with only the relation and the two helper references changed.

lemma *ar_write_tape_finish_step_in_sub*:

```

fixes M :: ('q, 'a) mttm
and c' :: (sym4, 'q × 'a ar_stage) mt_config
and tM :: nat ⇒ 'a
assumes vM: valid_mttm M
and qQ: q ∈ Q_tm M
and kge2: 2 ≤ block_width (Γ_tm M)
and last: is_last_k M tk
and stg: mt_state c' = (q, AR_SimWrite, tk, 0, buf, dvec, posk)
and tcorr: ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
  tM (mt_tape c' tk)
and ppos: mt_pos c' tk = (if p = 0 then Suc 0
  else sim_pos (block_width (Γ_tm M)) p + block_width (Γ_tm
M))
and poskle: posk tk = AR_AtLE ⟷ p = 0
and buf_valid: ∀ k. buf k ∈ Γ_tm M ∪ {bl_tm M}
and vsrc: ar_valid_stage (Γ_tm M) (bl_tm M)
  (AR_SimWrite, tk, 0, buf, dvec, posk)
and pad0: ∀ j ≥ k_tm M. mt_tape c' j (mt_pos c' j) = BLANK4
and src0: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimWrite, tk, 0, buf, dvec, posk)
shows ∃ c''. (c', c'') ∈ (mttm_step (ar_delta_write M))
  ^ (if p = 0 then 1 else 2 * block_width (Γ_tm M))
  ∧ mt_state c'' = (q, AR_SimAdvance, k_unidx 0, 0, buf, dvec, posk)
  ∧ mt_tape c'' = (if p = 0 then mt_tape c'
  else (mt_tape c')(tk := (λpos.
    if sim_pos (block_width (Γ_tm M)) p ≤ pos
      ∧ pos < sim_pos (block_width (Γ_tm M)) p + block_width
(Γ_tm M)
      then write_bit (Γ_tm M) (bl_tm M) (buf tk)

```


$(pos - sim_pos (block_width (\Gamma_tm M)) p)$
 $else\ mt_tape\ c'\ tk\ pos)))$
 $\wedge\ mt_pos\ c'' = mt_pos\ c'$

by (*rule* *ar_write_tape_finish_step_gen*
 $[OF\ ar_write_le_finish_step_in_sub\ ar_write_proper_finish_step_in_sub$
 $vM\ qQ\ kge2\ last\ stg\ tcorr\ ppos\ poskle\ buf_valid\ vsrc\ pad0\ src0])$)

The write-phase prefix walk, re-mirrored into *mttm_step* (*ar_delta_write M*): from the write boundary, iterate the unified non-last per-tape write *ar_write_tape_step_in_sub* over the first *j* tapes (all non-last), landing back at *AR_SimWrite* on tape *k_unidx j*. The induction on *j*, the split tape descriptor, and the per-tape *tcorr*/position bookkeeping transfer verbatim from the original; only the relation and the one helper reference change. The internal IH is already over *ar_delta_write M*.

lemma *ar_write_prefix_in_sub*:

fixes *M* :: ('q, 'a) *mttm*
and *cM* :: ('a, 'q) *mt_config*
and *c0* :: (sym4, 'q × 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm M*
and *qQ*: *q* ∈ *Q_tm M*
and *kge2*: $2 \leq block_width (\Gamma_tm M)$
and *stg0*: *mt_state c0* = (*q*, *AR_SimWrite*, 0, 0, *buf*, *dvec*, *posk*)
and *tcorr*: $\forall k < k_tm\ M. ar_tape_correspondence (\Gamma_tm M) (le_tm M)$
 $(bl_tm M)$
 $(mt_tape\ cM\ k)\ (mt_tape\ c0\ k)$
and *ppos*: $\forall k < k_tm\ M. mt_pos\ c0\ k = (if\ mt_pos\ cM\ k = 0\ then\ Suc\ 0$
 $else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $+ block_width\ (\Gamma_tm\ M))$
and *poskle*: $\forall k < k_tm\ M. posk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0$
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *pad0*: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and *src0*: *ar_stage_bounded* (*bl_tm M*) (*k_tm M*)
 $(AR_SimWrite, 0, 0, buf, dvec, posk)$
shows $j \leq k_tm\ M - 1 \implies$
 $(\exists\ c\ m. (c0, c) \in (mttm_step\ (ar_delta_write\ M))\ \wedge\ m$
 $\wedge\ m \leq j * (2 * block_width\ (\Gamma_tm\ M))$
 $\wedge\ mt_state\ c = (q, AR_SimWrite, k_unidx\ j, 0, buf, dvec, posk)$
 $\wedge\ mt_tape\ c = (\lambda k. if\ k_idx\ k < j$
 $then\ (if\ mt_pos\ cM\ k = 0\ then\ mt_tape\ c0\ k$
 $else\ (\lambda pos. if\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $\leq\ pos$
 $\wedge\ pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k)$
 $+ block_width\ (\Gamma_tm\ M)$
 $then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ k)$
 $(pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k))$
 $else\ mt_tape\ c0\ k\ pos))$
 $else\ mt_tape\ c0\ k)$

$\wedge mt_pos\ c = mt_pos\ c0)$
by (*rule* *ar_write_prefix_gen*
 $[OF\ ar_write_tape_step_in_sub\ vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ poskle\ buf_valid$
 $pad0\ src0])$

The full write phase, re-mirrored into *mttm_step* (*ar_delta_write* *M*): the prefix walk (*ar_write_prefix_in_sub*) over the first $k_tm\ M - 1$ tapes followed by the unified last-tape write (*ar_write_tape_finish_step_in_sub*), landing at *AR_SimAdvance* with every proper tape's block overwritten by *write_bit* (*buf* *k*) and every *LE* tape / head unchanged. Aggregate cost $\leq k_tm\ M \cdot 2b$. The split-to-full descriptor collapse, the cardinality bookkeeping, and the cost bound are relation-agnostic and transfer verbatim; only the relation and the two helper references change.

lemma *ar_write_phase_in_sub*:
fixes *M* :: ('q, 'a) *mttm*
and *cM* :: ('a, 'q) *mt_config*
and *c0* :: (sym4, 'q $\times$ 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *qQ*: $q \in Q_tm\ M$
and *kge2*: $2 \leq block_width\ (\Gamma_tm\ M)$
and *stg0*: *mt_state* *c0* = (*q*, *AR_SimWrite*, 0, 0, *buf*, *dvec*, *posk*)
and *tcorr*: $\forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)$
 $(bl_tm\ M)$
 $(mt_tape\ cM\ k)\ (mt_tape\ c0\ k)$
and *ppos*: $\forall k < k_tm\ M. mt_pos\ c0\ k = (if\ mt_pos\ cM\ k = 0\ then\ Suc\ 0$
 $else\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $+ block_width\ (\Gamma_tm\ M))$
and *poskle*: $\forall k < k_tm\ M. posk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0$
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *pad0*: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and *src0*: *ar_stage_bounded* (*bl_tm* *M*) (*k_tm* *M*)
 $(AR_SimWrite, 0, 0, buf, dvec, posk)$
shows $\exists c\ m. (c0, c) \in (mttm_step\ (ar_delta_write\ M))\ \wedge\wedge\ m$
 $\wedge m \leq k_tm\ M * (2 * block_width\ (\Gamma_tm\ M))$
 $\wedge mt_state\ c = (q, AR_SimAdvance, k_unidx\ 0, 0, buf, dvec, posk)$
 $\wedge mt_tape\ c = (\lambda k. if\ k < k_tm\ M$
 $then\ (if\ mt_pos\ cM\ k = 0\ then\ mt_tape\ c0\ k$
 $else\ (\lambda pos. if\ sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos\ cM\ k)$
 $\leq pos$
 $\wedge pos < sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k)$
 $+ block_width\ (\Gamma_tm\ M)$
 $then\ write_bit\ (\Gamma_tm\ M)\ (bl_tm\ M)\ (buf\ k)$
 $(pos - sim_pos\ (block_width\ (\Gamma_tm\ M))\ (mt_pos$
 $cM\ k))$
 $else\ mt_tape\ c0\ k\ pos))$
 $else\ mt_tape\ c0\ k)$

$\wedge mt_pos\ c = mt_pos\ c0$
by (*rule* *ar_write_phase_gen*
 $[OF\ ar_write_prefix_in_sub\ ar_write_tape_finish_step_in_sub$
 $vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ poskle\ buf_valid\ pad0\ src0])$

The advance left-walk loop, re-mirrored into *mttm_step* (*ar_delta_advance* *M*). From an *AR_SimAdvance* stage at bit-counter 0, *n* *M*'-steps walk the head *n* cells *L* (counter $0 \rightarrow n$), tape and other heads unchanged. The *relpow_invariant_chain* loop, the per-step $\neq LE4$ guard, and the displacement bound (*ar_disp_le_2k*) are relation-agnostic and transfer verbatim; only the relation and the one helper reference change.

lemma *ar_advance_walk_loop_in_sub*:
fixes *M* :: ('q, 'a) *mttm*
and *c0* :: (*sym4*, 'q $\times$ 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *qQ*: *q* $\in Q_tm\ M$
and *stg*: *mt_state* *c0* = (*q*, *AR_SimAdvance*, *tk*, 0, *buf*, *dvec*, *posk*)
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and *ndisp*: $n < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
and *notLE*: $\bigwedge j. j < n \implies mt_tape\ c0\ tk\ (mt_pos\ c0\ tk - j) \neq LE4$
and *pad0*: $\forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and *src0*: *ar_stage_bounded* (*bl_tm* *M*) (*k_tm* *M*)
 $(AR_SimAdvance, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in (mttm_step\ (ar_delta_advance\ M)) \wedge \wedge n$
 $\wedge mt_state\ c' = (q, AR_SimAdvance, tk, n, buf, dvec, posk)$
 $\wedge mt_pos\ c'\ tk = mt_pos\ c0\ tk - n$
 $\wedge mt_tape\ c' = mt_tape\ c0$
 $\wedge (\forall k'. k' \neq tk \longrightarrow mt_pos\ c'\ k' = mt_pos\ c0\ k')$
by (*rule* *ar_advance_walk_loop_gen*
 $[OF\ ar_advance_walk_step_in_sub\ vM\ qQ\ stg\ buf_valid\ ndisp\ notLE\ pad0$
 $src0])$

The unified single-tape advance, non-last tape, re-mirrored into *mttm_step* (*ar_delta_advance* *M*): the *R*/non-*R* dispatch. An *R*-move needs no head motion (single boundary step, cost 1); a non-*R*-move walks the head *D* = *ar_disp* cells *L* (the *D* − 1-step *ar_advance_walk_loop_in_sub* then the final boundary *L*-move), landing at *start* − *D*. The walk-then-boundary decomposition and the conditional head conclusion transfer verbatim; only the relation and the two helper references change.

lemma *ar_advance_tape_step_in_sub*:
fixes *M* :: ('q, 'a) *mttm*
and *c0* :: (*sym4*, 'q $\times$ 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *qQ*: *q* $\in Q_tm\ M$
and *kge2*: $2 \leq block_width\ (\Gamma_tm\ M)$
and *notlast*: $\neg is_last_k\ M\ tk$
and *stg*: *mt_state* *c0* = (*q*, *AR_SimAdvance*, *tk*, 0, *buf*, *dvec*, *posk*)
and *buf_valid*: $\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$

and $dge1: dvec\ tk \neq dir.R$
 $\implies 0 < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
and $notLE: \bigwedge m. m < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
 $\implies mt_tape\ c0\ tk\ (mt_pos\ c0\ tk - m) \neq LE4$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in (mttm_step\ (ar_delta_advance\ M))$
 $\wedge (if\ dvec\ tk = dir.R\ then\ 1$
 $\quad else\ ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk))$
 $\wedge mt_state\ c' = (q, AR_SimAdvance, k_succ\ tk, 0, buf, dvec,$
 $\quad posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
 $\wedge mt_tape\ c' = mt_tape\ c0$
 $\wedge mt_pos\ c' = (if\ dvec\ tk = dir.R\ then\ mt_pos\ c0$
 $\quad else\ (mt_pos\ c0)(tk := mt_pos\ c0\ tk$
 $\quad - ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk$
 $\quad tk)))$
by $(rule\ ar_advance_tape_step_gen$
 $\quad [OF\ ar_advance_walk_loop_in_sub\ ar_advance_boundary_step_in_sub$
 $\quad vM\ qQ\ kge2\ notlast\ stg\ buf_valid\ dge1\ notLE\ pad0\ src0])$

The unified single-tape advance, last tape, re-mirrored into $mttm_step\ (ar_delta_advance\ M)$: as $ar_advance_tape_step_in_sub$ but tk is the last tape, so the boundary step ($ar_advance_finish_step_in_sub$) transitions to $AR_SimNext$ (current-tape field reset to $k_unidx\ 0$) rather than advancing to $k_succ\ tk$. Same R/non-R dispatch and walk-then-boundary decomposition; only the relation and the two helper references change.

lemma $ar_advance_tape_finish_step_in_sub$:
fixes $M :: ('q, 'a)\ mttm$
and $c0 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $last: is_last_k\ M\ tk$
and $stg: mt_state\ c0 = (q, AR_SimAdvance, tk, 0, buf, dvec, posk)$
and $buf_valid: \forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $dge1: dvec\ tk \neq dir.R$
 $\implies 0 < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
and $notLE: \bigwedge m. m < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk)$
 $\implies mt_tape\ c0\ tk\ (mt_pos\ c0\ tk - m) \neq LE4$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, tk, 0, buf, dvec, posk)$
shows $\exists c'. (c0, c') \in (mttm_step\ (ar_delta_advance\ M))$
 $\wedge (if\ dvec\ tk = dir.R\ then\ 1$
 $\quad else\ ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ tk)\ (posk\ tk))$
 $\wedge mt_state\ c' = (q, AR_SimNext, k_unidx\ 0, 0, buf, dvec,$
 $\quad posk(tk := ar_newpos\ (dvec\ tk)\ (posk\ tk)))$
 $\wedge mt_tape\ c' = mt_tape\ c0$

$\wedge \text{mt_pos } c' = (\text{if } \text{dvec } tk = \text{dir}.R \text{ then } \text{mt_pos } c0$
 $\quad \text{else } (\text{mt_pos } c0)(tk := \text{mt_pos } c0 \text{ } tk$
 $\quad \quad - \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } tk) (\text{posk}$
 $tk)))$
by (*rule* *ar_advance_tape_finish_step_gen*
 $[\text{OF } \text{ar_advance_walk_loop_in_sub } \text{ar_advance_finish_step_in_sub}$
 $\text{vM } qQ \text{ kge2 last stg buf_valid dge1 notLE pad0 src0}])$

The advance prefix walk, re-mirrored into *mttm_step* (*ar_delta_advance* *M*): a custom induction on the tape index *j* ($j \leq k_tm \text{ } M - 1$, every tape it touches non-last), iterating *ar_advance_tape_step_in_sub* from the boundary tape *k_unidx* 0. The tape is constant; the carried state is a *k_idx* *k* < *j* split over the *posk* and position vectors. The induction, the descriptor collapse, and the per-tape *notLE*/*dge1* entry facts are relation-agnostic and transfer verbatim; only the relation and the one helper reference change.

lemma *ar_advance_prefix_in_sub*:
fixes *M* :: ('q, 'a) *mttm*
and *c0* :: (*sym4*, 'q $\times$ 'a *ar_stage*) *mt_config*
assumes *vM*: *valid_mttm* *M*
and *qQ*: *q* $\in Q_tm \text{ } M$
and *kge2*: $2 \leq \text{block_width } (\Gamma_tm \text{ } M)$
and *stg0*: *mt_state* *c0* = (*q*, *AR_SimAdvance*, 0, 0, *buf0*, *dvec*, *posk0*)
and *buf_valid*: $\forall k. \text{buf0 } k \in \Gamma_tm \text{ } M \cup \{\text{bl_tm } M\}$
and *dge1*: $\forall k < k_tm \text{ } M. \text{dvec } k \neq \text{dir}.R$
 $\quad \rightarrow 0 < \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } k) (\text{posk0 } k)$
and *notLE*: $\forall k < k_tm \text{ } M. \forall m. m < \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M))$
 $(\text{dvec } k) (\text{posk0 } k)$
 $\quad \rightarrow \text{mt_tape } c0 \text{ } k (\text{mt_pos } c0 \text{ } k - m) \neq \text{LE4}$
and *pad0*: $\forall j \geq k_tm \text{ } M. \text{mt_tape } c0 \text{ } j (\text{mt_pos } c0 \text{ } j) = \text{BLANK4}$
and *src0*: *ar_stage_bounded* (*bl_tm* *M*) (*k_tm* *M*)
 $(\text{AR_SimAdvance}, 0, 0, \text{buf0}, \text{dvec}, \text{posk0})$
shows $j \leq k_tm \text{ } M - 1 \implies$
 $(\exists c \text{ } m. (c0, c) \in (\text{mttm_step } (\text{ar_delta_advance } M)) \wedge \wedge m$
 $\wedge m \leq j * (2 * \text{block_width } (\Gamma_tm \text{ } M))$
 $\wedge \text{mt_state } c = (q, \text{AR_SimAdvance}, k_unidx \text{ } j, 0, \text{buf0}, \text{dvec},$
 $(\lambda k. \text{if } k_idx \text{ } k < j \text{ then } \text{ar_newpos } (\text{dvec } k) (\text{posk0 } k)$
 $\quad \text{else } \text{posk0 } k))$
 $\wedge \text{mt_tape } c = \text{mt_tape } c0$
 $\wedge \text{mt_pos } c = (\lambda k. \text{if } k_idx \text{ } k < j$
 $\quad \text{then } (\text{if } \text{dvec } k = \text{dir}.R \text{ then } \text{mt_pos } c0 \text{ } k$
 $\quad \quad \text{else } \text{mt_pos } c0 \text{ } k$
 $\quad \quad - \text{ar_disp } (\text{block_width } (\Gamma_tm \text{ } M)) (\text{dvec } k) (\text{posk0 } k))$
 $\quad \text{else } \text{mt_pos } c0 \text{ } k))$
by (*rule* *ar_advance_prefix_gen*
 $[\text{OF } \text{ar_advance_tape_step_in_sub } \text{vM } qQ \text{ kge2 stg0 buf_valid dge1 notLE}$
 $\text{pad0 src0}])$

The full advance phase, re-mirrored into *mttm_step* (*ar_delta_advance* *M*): the prefix walk (*ar_advance_prefix_in_sub*) over the first

$k_tm\ M - 1$ tapes followed by the unified last-tape advance ($ar_advance_tape_finish_step_in_sub$), landing at $AR_SimNext$ (current-tape field $k_unidx\ 0$) with every head moved to M 's new position and every $posk$ updated by ar_newpos . The tape is unchanged. Aggregate cost $\leq k_tm\ M \cdot 2b$. The split-to-full descriptor collapse and cost bound are relation-agnostic and transfer verbatim; only the relation and the two helper references change.

lemma $ar_advance_phase_in_sub$:
fixes $M :: ('q, 'a)\ mttm$
and $c0 :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
assumes $vM: valid_mttm\ M$
and $qQ: q \in Q_tm\ M$
and $kge2: 2 \leq block_width\ (\Gamma_tm\ M)$
and $stg0: mt_state\ c0 = (q, AR_SimAdvance, 0, 0, buf0, dvec, posk0)$
and $buf_valid: \forall k. buf0\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
and $dge1: \forall k < k_tm\ M. dvec\ k \neq dir.R$
 $\longrightarrow 0 < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ k)\ (posk0\ k)$
and $notLE: \forall k < k_tm\ M. \forall m. m < ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ k)\ (posk0\ k)$
 $\longrightarrow mt_tape\ c0\ k\ (mt_pos\ c0\ k - m) \neq LE4$
and $pad0: \forall j \geq k_tm\ M. mt_tape\ c0\ j\ (mt_pos\ c0\ j) = BLANK4$
and $src0: ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(AR_SimAdvance, 0, 0, buf0, dvec, posk0)$
shows $\exists c\ m. (c0, c) \in (mttm_step\ (ar_delta_advance\ M))\ \wedge\ m$
 $\wedge\ m \leq k_tm\ M * (2 * block_width\ (\Gamma_tm\ M))$
 $\wedge\ mt_state\ c = (q, AR_SimNext, k_unidx\ 0, 0, buf0, dvec,$
 $(\lambda k. if\ k < k_tm\ M\ then\ ar_newpos\ (dvec\ k)\ (posk0\ k)$
 $else\ posk0\ k))$
 $\wedge\ mt_tape\ c = mt_tape\ c0$
 $\wedge\ mt_pos\ c = (\lambda k. if\ k < k_tm\ M$
 $then\ (if\ dvec\ k = dir.R\ then\ mt_pos\ c0\ k$
 $else\ mt_pos\ c0\ k - ar_disp\ (block_width\ (\Gamma_tm\ M))\ (dvec\ k)$
 $(posk0\ k))$
 $else\ mt_pos\ c0\ k)$
by ($rule\ ar_advance_phase_gen$
 $[OF\ ar_advance_prefix_in_sub\ ar_advance_tape_finish_step_in_sub$
 $vM\ qQ\ kge2\ stg0\ buf_valid\ dge1\ notLE\ pad0\ src0])$

3.17 Walker preservation — per-substep M' -step lemmas

One preservation lemma per substep predicate: an M' -step out of a config satisfying the current invariant lands at a config satisfying the next invariant in the cycle (or splits the disjunction when the substep's relation has multiple destination arms). Together with the chunked engine, these characterise the walker's per-step evolution: the substep idx carried in the M' -config is the dispatch discriminator at each step, no chain pinning needed.

lemma $ar_walker_step_from_boundary$:

```

fixes  $M :: ('q, 'a) \text{ mttm}$ 
and  $cM :: ('a, 'q) \text{ mt\_config}$ 
and  $c' \ c'' :: (\text{sym4}, 'q \times 'a \text{ ar\_stage}) \text{ mt\_config}$ 
assumes  $\text{inv}: \text{ar\_walker\_at\_boundary } M \ cM \ c'$ 
and  $\text{step}: (c', c'') \in \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 
shows  $\text{ar\_walker\_in\_read } M \ cM \ c'' \vee \text{ar\_walker\_at\_compute } M \ cM \ c''$ 
proof –
  have  $\text{rbnd}: \text{ar\_at\_read\_boundary } M \ c'$ 
  using  $\text{inv}$  unfolding  $\text{ar\_walker\_at\_boundary\_def}$  by  $\text{simp}$ 
  obtain  $qM' \ \text{stg}$  where  $\text{st}: \text{mt\_state } c' = (qM', \text{stg})$ 
  by  $(\text{cases } \text{mt\_state } c') \text{ auto}$ 
  obtain  $\text{idx } \text{tk } i \ \text{buf } \text{dvec } \text{posk}$  where
     $\text{sg}: \text{stg} = (\text{idx}, \text{tk}, i, \text{buf}, \text{dvec}, \text{posk})$  by  $(\text{cases } \text{stg}) \text{ auto}$ 
    —  $\text{ar\_at\_read\_boundary}$ 's condition is a non-trivial implication ( $\text{idx} = \text{AR\_SimRead} \longrightarrow \dots$ ) — but the only way  $\text{ar\_walker\_at\_boundary}$  can hold with  $\text{idx} \neq \text{AR\_SimRead}$  would require  $\text{ar\_simulates}$ 's halt disjunct to fire, which the  $\text{ar\_at\_read\_boundary}$  hypothesis combined with stage shape rules out at the boundary of a non-halting cycle. Discharged via  $\text{ar\_simulates}$ 's state-shape branches.
  have  $\text{src}: \text{fst } (\text{snd } (\text{mt\_state } c')) = \text{AR\_SimRead}$ 
  proof –
    have  $\text{sim}: \text{ar\_simulates } M \ cM \ c'$ 
    using  $\text{inv}$  unfolding  $\text{ar\_walker\_at\_boundary\_def}$  by  $\text{simp}$ 
    have  $\text{sim\_body}$ :
       $((\text{idx} = \text{AR\_SimRead} \wedge qM' \notin \{t\_tm \ M, r\_tm \ M\})$ 
         $\vee (qM' = t\_tm \ M$ 
           $\wedge (\text{idx}, \text{tk}, i, \text{buf}, \text{dvec}, \text{posk}) = \text{ar\_accept\_stage } (\text{bl\_tm } M))$ 
           $\vee (qM' = r\_tm \ M$ 
             $\wedge (\text{idx}, \text{tk}, i, \text{buf}, \text{dvec}, \text{posk}) = \text{ar\_reject\_stage } (\text{bl\_tm } M)))$ 
             $\wedge \text{mt\_state } cM = qM'$ 
    using  $\text{sim}$  unfolding  $\text{ar\_simulates\_def}$  by  $(\text{simp add: st sg Let\_def})$ 
    consider  $(R) \ \text{idx} = \text{AR\_SimRead}$ 
      |  $(A) \ qM' = t\_tm \ M \wedge (\text{idx}, \text{tk}, i, \text{buf}, \text{dvec}, \text{posk}) = \text{ar\_accept\_stage } (\text{bl\_tm } M)$ 
      |  $(J) \ qM' = r\_tm \ M \wedge (\text{idx}, \text{tk}, i, \text{buf}, \text{dvec}, \text{posk}) = \text{ar\_reject\_stage } (\text{bl\_tm } M)$ 
    using  $\text{sim\_body}$  by  $\text{blast}$ 
    thus  $?thesis$ 
  proof  $\text{cases}$ 
    case  $R$  thus  $?thesis$  using  $\text{st sg}$  by  $\text{simp}$ 
  next
    case  $A$ 
    — halt-accept stage has  $\text{idx} = \text{AR\_HaltAccept}$ , which together with  $\text{ar\_at\_read\_boundary}$ 's  $\text{idx} = \text{AR\_SimRead} \longrightarrow \dots$  is consistent only if we are NOT at  $\text{AR\_SimRead}$ . But the step out of a halt-coerced config is impossible —  $S$  in  $\text{mttm\_step.cases}$  with  $\text{fst } (\text{snd } S) = \text{AR\_HaltAccept}$  cannot fire any of the five substep relations (constructor-distinctness from  $\text{alphabet\_reduce\_delta\_src\_tag}$ ).
    have  $\text{idx\_acc}: \text{idx} = \text{AR\_HaltAccept}$ 
    using  $A$  by  $(\text{simp add: ar\_accept\_stage\_def})$ 

```

```

from step obtain  $S \text{ ts } n \ S' \text{ aw } \text{dir}$  where
   $c'_{\text{eq}}: c' = \text{Config}_M \ S \text{ ts } n$ 
  and  $\text{rel}: (S, (\lambda k. \text{ts } k \ (n \ k)), S', \text{aw}, \text{dir})$ 
     $\in \text{alphabet\_reduce\_delta } M$ 
  by (auto elim: mttm_step.cases)
have  $\text{src\_S}: \text{fst} (\text{snd } S) = \text{AR\_HaltAccept}$ 
  using  $c'_{\text{eq}} \text{ st sg idx\_acc}$  by simp
from  $\text{alphabet\_reduce\_delta\_src\_tag}[OF \text{ rel}]$ 
have  $\text{fst} (\text{snd } S) \in \{\text{AR\_SimRead}, \text{AR\_SimCompute}, \text{AR\_SimWrite},$ 
     $\text{AR\_SimAdvance}, \text{AR\_SimNext}\} .$ 
hence False using  $\text{src\_S}$  by auto
thus ?thesis ..
next
case J
have  $\text{idx\_rej}: \text{idx} = \text{AR\_HaltReject}$ 
  using J by (simp add: ar_reject_stage_def)
from step obtain  $S \text{ ts } n \ S' \text{ aw } \text{dir}$  where
   $c'_{\text{eq}}: c' = \text{Config}_M \ S \text{ ts } n$ 
  and  $\text{rel}: (S, (\lambda k. \text{ts } k \ (n \ k)), S', \text{aw}, \text{dir})$ 
     $\in \text{alphabet\_reduce\_delta } M$ 
  by (auto elim: mttm_step.cases)
have  $\text{src\_S}: \text{fst} (\text{snd } S) = \text{AR\_HaltReject}$ 
  using  $c'_{\text{eq}} \text{ st sg idx\_rej}$  by simp
from  $\text{alphabet\_reduce\_delta\_src\_tag}[OF \text{ rel}]$ 
have  $\text{fst} (\text{snd } S) \in \{\text{AR\_SimRead}, \text{AR\_SimCompute}, \text{AR\_SimWrite},$ 
     $\text{AR\_SimAdvance}, \text{AR\_SimNext}\} .$ 
hence False using  $\text{src\_S}$  by auto
thus ?thesis ..
qed
qed
— The  $M'$ -step fires  $\text{ar\_delta\_read}$  by  $\text{src\_tag}$  uniqueness; the lift extends the
witness chain by one step.
have  $\text{step\_read}: (c', c'') \in \text{mttm\_step} (\text{ar\_delta\_read } M)$ 
  using  $\text{ar\_step\_read\_lift}[OF \text{ src\_step}] .$ 
hence  $\text{step\_read1}: (c', c'') \in (\text{mttm\_step} (\text{ar\_delta\_read } M)) \wedge \wedge \text{Suc } 0$ 
  by simp
— Dispatch on the dest tag
from  $\text{ar\_step\_from\_SimRead\_lands}[OF \text{ step\_src}]$ 
have  $\text{dest}: \text{fst} (\text{snd} (\text{mt\_state } c'')) = \text{AR\_SimRead}$ 
   $\vee \text{fst} (\text{snd} (\text{mt\_state } c'')) = \text{AR\_SimCompute} .$ 
thus ?thesis
proof
  assume  $\text{dr}: \text{fst} (\text{snd} (\text{mt\_state } c'')) = \text{AR\_SimRead}$ 
  have  $\text{ar\_walker\_in\_read } M \text{ cM } c''$ 
    unfolding  $\text{ar\_walker\_in\_read\_def}$ 
    using  $\text{dr inv step\_read1}$  by blast
  thus ?thesis ..
next
  assume  $\text{dc}: \text{fst} (\text{snd} (\text{mt\_state } c'')) = \text{AR\_SimCompute}$ 

```



```

    have ar_walker_at_compute M cM c''
      unfolding ar_walker_at_compute_def
      using dc inv step_read1 by blast
    thus ?thesis ..
  qed
qed

lemma ar_walker_step_from_in_read:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c' c'' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes inv: ar_walker_in_read M cM c'
    and step: (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
  shows ar_walker_in_read M cM c'' ∨ ar_walker_at_compute M cM c''
proof -
  have src: fst (snd (mt_state c')) = AR_SimRead
    using inv unfolding ar_walker_in_read_def by simp
  obtain c_b m where
    wb: ar_walker_at_boundary M cM c_b
    and chain: (c_b, c') ∈ (mttm_step (ar_delta_read M)) ^^ m
    using inv unfolding ar_walker_in_read_def by blast
  have step_read: (c', c'') ∈ mttm_step (ar_delta_read M)
    using ar_step_read lift[OF src step] .
  have chain_ext: (c_b, c'') ∈ (mttm_step (ar_delta_read M)) ^^ Suc m
    using chain step_read by (rule relpow_Suc_I)
  from ar_step_from_SimRead_lands[OF step src]
  have dest: fst (snd (mt_state c'')) = AR_SimRead
    ∨ fst (snd (mt_state c'')) = AR_SimCompute .
  thus ?thesis
proof
  assume dr: fst (snd (mt_state c'')) = AR_SimRead
  have ar_walker_in_read M cM c''
    unfolding ar_walker_in_read_def
    using dr wb chain_ext by blast
  thus ?thesis ..
next
  assume dc: fst (snd (mt_state c'')) = AR_SimCompute
  have ar_walker_at_compute M cM c''
    unfolding ar_walker_at_compute_def
    using dc wb chain_ext by blast
  thus ?thesis ..
qed
qed

lemma ar_walker_step_from_at_compute:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c' c'' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes inv: ar_walker_at_compute M cM c'

```

```

    and step: (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
    and vM: valid_mttm M
    and qQ: mt_state cM ∈ Q_tm M
    and kge2: 2 ≤ block_width (Γ_tm M)
    and tapeG: ∧k. mt_tape cM k (mt_pos cM k) ∈ Γ_tm M
  shows ar_walker_in_write M cM c''
proof -
  let ?k = block_width (Γ_tm M)
  let ?aM = λk. mt_tape cM k (mt_pos cM k)
  have c'_compute: fst (snd (mt_state c')) = AR_SimCompute
    using inv unfolding ar_walker_at_compute_def by simp
  obtain c_b m_w where
    wb: ar_walker_at_boundary M cM c_b
    and chain_w: (c_b, c') ∈ (mttm_step (ar_delta_read M)) ^^ m_w
    using inv unfolding ar_walker_at_compute_def by blast
  have sim: ar_simulates M cM c_b
    using wb unfolding ar_walker_at_boundary_def by simp
  have pcons: ar_posk_consistent M cM c_b
    using wb unfolding ar_walker_at_boundary_def by simp
  have rbnd: ar_at_read_boundary M c_b
    using wb unfolding ar_walker_at_boundary_def by simp
  — c_b's tag is AR_SimRead: from a chain step (when m_w > 0) by
ar_delta_read_src, or by ruling out the halt branches of ar_simulates (when m_w
= 0).
  have cb_idx: fst (snd (mt_state c_b)) = AR_SimRead
  proof (cases m_w)
    case 0
    have cb_eq: c_b = c' using chain_w 0 by simp
    have cb_compute: fst (snd (mt_state c_b)) = AR_SimCompute
      using c'_compute cb_eq by simp
    obtain qM' stg where st_b: mt_state c_b = (qM', stg)
      by (cases mt_state c_b) auto
    obtain idx tk i buf dvec posk where
      sg_b: stg = (idx, tk, i, buf, dvec, posk) by (cases stg) auto
    have idx_cpu: idx = AR_SimCompute using cb_compute st_b sg_b by simp
    have (idx = AR_SimRead ∧ qM' ∉ {t_tm M, r_tm M})
      ∨ (qM' = t_tm M ∧ stg = ar_accept_stage (bl_tm M))
      ∨ (qM' = r_tm M ∧ stg = ar_reject_stage (bl_tm M))
    using sim st_b sg_b unfolding ar_simulates_def by (auto split: prod.splits)
    hence False
      using idx_cpu sg_b
      by (auto simp: ar_accept_stage_def ar_reject_stage_def)
    thus ?thesis ..
  next
  case (Suc m')
  have hsuc: (c_b, c') ∈ (mttm_step (ar_delta_read M)) ^^ Suc m'
    using chain_w Suc by simp
  obtain c_1 where
    first: (c_b, c_1) ∈ mttm_step (ar_delta_read M)

```

using *relpow_Suc_D2*[*OF hsuc*] **by** *blast*
from first obtain *S ts n S' aw dir* **where**
ceq: $c_b = \text{Config}_M S ts n$
and *rel*: $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in ar_delta_read\ M$
by (*auto elim*: *mttm_step.cases*)
have *fst* (*snd S*) = *AR_SimRead* **using** *ar_delta_read_src*[*OF rel*] .
thus *?thesis* **using** *ceq* **by** *simp*
qed
— Extract *c_b*'s full state shape and the boundary's correspondences.
obtain *qM' stg* **where** *st_b*: *mt_state c_b* = (*qM'*, *stg*)
by (*cases mt_state c_b*) *auto*
obtain *idx tk i buf dvec posk* **where**
sg_b: *stg* = (*idx, tk, i, buf, dvec, posk*) **by** (*cases stg*) *auto*
have *idx_read*: *idx* = *AR_SimRead* **using** *cb_idx st_b sg_b* **by** *simp*
have *sim_unfold*: (*idx* = *AR_SimRead* $\wedge$ *qM'* $\notin \{t_tm\ M, r_tm\ M\}$)
 $\vee$ (*qM'* = *t_tm M* $\wedge$ *stg* = *ar_accept_stage* (*bl_tm M*))
 $\vee$ (*qM'* = *r_tm M* $\wedge$ *stg* = *ar_reject_stage* (*bl_tm M*))
using *sim st_b sg_b unfolding ar_simulates_def* **by** (*auto split*: *prod.splits*)
have *qM'_eq*: *qM'* = *mt_state cM*
using *sim st_b sg_b unfolding ar_simulates_def* **by** (*auto split*: *prod.splits*)
have *tcorr*: $\forall k < k_tm\ M. ar_tape_correspondence\ (\Gamma_tm\ M)\ (le_tm\ M)$
(*bl_tm M*)
(*mt_tape cM k*) (*mt_tape c_b k*)
using *sim st_b sg_b unfolding ar_simulates_def* **by** (*auto split*: *prod.splits*)
have *ppos*: $\bigwedge k. mt_pos\ c_b\ k = sim_pos\ ?k\ (mt_pos\ cM\ k)$
using *sim st_b sg_b idx_read*
unfolding *ar_simulates_def* **by** (*auto split*: *prod.splits*)
have *pkok*: $\bigwedge k. posk\ k = AR_AtLE \longleftrightarrow mt_pos\ cM\ k = 0$
using *pcons st_b sg_b idx_read*
unfolding *ar_posk_consistent_def* **by** (*auto split*: *prod.splits*)
have *bnd_unfold*: *idx* = *AR_SimRead*
 $\longrightarrow (tk = 0 \wedge i = 0$
 $\wedge (\forall k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\})$
 $\wedge ar_stage_bounded\ (bl_tm\ M)\ (k_tm\ M)$
 $(idx, tk, i, buf, dvec, posk))$
using *rbnd st_b sg_b*
unfolding *ar_at_read_boundary_def* **by** (*auto split*: *prod.splits*)
have *tk0*: *tk* = 0 **and** *i0*: *i* = 0
and *bufG*: $\bigwedge k. buf\ k \in \Gamma_tm\ M \cup \{bl_tm\ M\}$
using *bnd_unfold idx_read* **by** *simp_all*
have *stg0*: *mt_state c_b* = (*mt_state cM*, *AR_SimRead*, 0, 0, *buf*, *dvec*, *posk*)
using *st_b sg_b qM'_eq idx_read tk0 i0* **by** *simp*
— Boundary blank-tail and bounded-stage facts, now produced by the strengthened *ar_at_read_boundary*: *src0_b* from its bounded-stage conjunct (specialised by *idx* = *AR_SimRead*, *tk* = 0, *i* = 0); *pad0_b* from its config padding-blank conjunct.
have *pad0_b*: $\forall j \geq k_tm\ M. mt_tape\ c_b\ j\ (mt_pos\ c_b\ j) = BLANK4$
using *rbnd unfolding ar_at_read_boundary_def* **by** (*auto split*: *prod.splits*)
have *src0_b*: *ar_stage_bounded* (*bl_tm M*) (*k_tm M*)

$(AR_SimRead, 0, 0, buf, dvec, posk)$
using $bnd_unfold\ idx_read\ tk0\ i0$ **by** $simp$
 — Apply $ar_read_phase_in_sub$ at the boundary.
obtain $c_r\ m_phase$ **where**
 $r_chain: (c_b, c_r) \in (mttm_step\ (ar_delta_read\ M)) \wedge m_phase$
and $r_state: mt_state\ c_r = (mt_state\ cM, AR_SimCompute, k_unidx\ 0,$
 $0,$
 $(\lambda k. \text{if } k < k_tm\ M \text{ then } ?aM\ k \text{ else } buf\ k), dvec,$
 $(\lambda k. \text{if } k < k_tm\ M$
 $\text{then } (if\ mt_pos\ cM\ k = 0 \text{ then } AR_AtLE$
 $\text{else if } mt_pos\ cM\ k = 1 \text{ then } AR_AtFirstProper$
 $\text{else } AR_AtFurtherProper)$
 $\text{else } posk\ k))$
using $ar_read_phase_in_sub[OF\ vM\ qQ\ kge2\ stg0\ tcorr\ ppos\ pkok\ tapeG\ bufG$
 $pad0_b\ src0_b]$
by $blast$
have $r_cpu: fst\ (snd\ (mt_state\ c_r)) = AR_SimCompute$
using r_state **by** $simp$
 — Pin the walker's witness chain against $ar_read_phase_in_sub$'s.
have $c'_eq_c_r: c' = c_r$
using $chain_ar_delta_read_to_SimCompute_uniq[OF\ chain_w\ r_chain$
 $c'_compute\ r_cpu]$
by $simp$
have $c'_state: mt_state\ c' = (mt_state\ cM, AR_SimCompute, k_unidx\ 0, 0,$
 $(\lambda k. \text{if } k < k_tm\ M \text{ then } ?aM\ k \text{ else } buf\ k), dvec,$
 $(\lambda k. \text{if } k < k_tm\ M$
 $\text{then } (if\ mt_pos\ cM\ k = 0 \text{ then } AR_AtLE$
 $\text{else if } mt_pos\ cM\ k = 1 \text{ then } AR_AtFirstProper$
 $\text{else } AR_AtFurtherProper)$
 $\text{else } posk\ k))$
using $c'_eq_c_r\ r_state$ **by** $simp$
 — Lift the compute step to the sub-relation and invert.
have $step_compute: (c', c'') \in mttm_step\ (ar_delta_compute\ M)$
using $ar_step_compute_lift[OF\ c'_compute\ step]$.
obtain $q'\ m_a'\ m_d$ **where**
 $mdelta: (mt_state\ cM, (\lambda k. \text{if } k < k_tm\ M \text{ then } ?aM\ k \text{ else } buf\ k),$
 $q', m_a', m_d) \in delta_tm\ M$
and $c''_state: mt_state\ c'' = (q', AR_SimWrite, 0, 0, m_a', m_d,$
 $(\lambda k. \text{if } k < k_tm\ M$
 $\text{then } (if\ mt_pos\ cM\ k = 0 \text{ then } AR_AtLE$
 $\text{else if } mt_pos\ cM\ k = 1 \text{ then } AR_AtFirstProper$
 $\text{else } AR_AtFurtherProper)$
 $\text{else } posk\ k))$
and $c''_tape: mt_tape\ c'' = mt_tape\ c'$
and $c''_pos: mt_pos\ c'' = mt_pos\ c'$
using $ar_compute_step_inv[OF\ step\ c'_state]$ **by** $blast$
 — Bundle the witness chain into $ar_walker_in_write$.
have $c''_write: fst\ (snd\ (mt_state\ c'')) = AR_SimWrite$
using c''_state **by** $simp$

```

have empty_w: (c'', c'') ∈ (mttm_step (ar_delta_write M)) ^^ 0 by simp
show ar_walker_in_write M cM c''
  unfolding ar_walker_in_write_def
proof (intro conjI)
  show fst (snd (mt_state c'')) = AR_SimWrite by (rule c''_write)
  show ∃ c_b c_w m_r m_w.
    ar_walker_at_boundary M cM c_b
    ∧ (c_b, c_w) ∈ (mttm_step (ar_delta_read M)) ^^ m_r
    ∧ fst (snd (mt_state c_w)) = AR_SimCompute
    ∧ (∃ c_w_post. (c_w, c_w_post) ∈ mttm_step (ar_delta_compute M)
      ∧ (c_w_post, c'') ∈ (mttm_step (ar_delta_write M)) ^^
m_w)
  by (intro exI[where x = c_b] exI[where x = c''] exI[where x = m_w]
    exI[where x = 0] conjI wb chain_w c'_compute
    exI[where x = c''] step_compute empty_w)
qed
qed

lemma ar_walker_step_from_in_write:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c' c'' :: (sym4, 'q × 'a ar_stage) mt_config
  assumes inv: ar_walker_in_write M cM c'
  and step: (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
  shows ar_walker_in_write M cM c'' ∨ ar_walker_in_advance M cM c''
proof -
  have src: fst (snd (mt_state c')) = AR_SimWrite
  using inv unfolding ar_walker_in_write_def by simp
  obtain c_b c_w m_r m_w where
    wb: ar_walker_at_boundary M cM c_b
    and chr: (c_b, c_w) ∈ (mttm_step (ar_delta_read M)) ^^ m_r
    and cwC: fst (snd (mt_state c_w)) = AR_SimCompute
    and rest: ∃ c_w_post. (c_w, c_w_post) ∈ mttm_step (ar_delta_compute M)
      ∧ (c_w_post, c') ∈ (mttm_step (ar_delta_write M)) ^^
m_w
  using inv unfolding ar_walker_in_write_def by blast
  obtain c_w_post where
    ccs: (c_w, c_w_post) ∈ mttm_step (ar_delta_compute M)
    and chw: (c_w_post, c') ∈ (mttm_step (ar_delta_write M)) ^^ m_w
  using rest by blast
  have step_write: (c', c'') ∈ mttm_step (ar_delta_write M)
  using ar_step_write_lift[OF src step] .
  have chw_ext: (c_w_post, c'') ∈ (mttm_step (ar_delta_write M)) ^^ Suc m_w
  using chw step_write by (rule relpow_Suc_I)
  from ar_step_from_SimWrite_lands[OF step src]
  have dest: fst (snd (mt_state c'')) = AR_SimWrite
    ∨ fst (snd (mt_state c'')) = AR_SimAdvance .
  thus ?thesis
proof

```

```

assume  $dw: fst (snd (mt\_state\ c'')) = AR\_SimWrite$ 
have  $ar\_walker\_in\_write\ M\ cM\ c''$ 
  unfolding  $ar\_walker\_in\_write\_def$ 
proof (intro conjI)
  show  $fst (snd (mt\_state\ c'')) = AR\_SimWrite$  by (rule dw)
  show  $\exists\ c\_b\ c\_w\ m\_r\ m\_w.$ 
     $ar\_walker\_at\_boundary\ M\ cM\ c\_b$ 
     $\wedge (c\_b, c\_w) \in (mttm\_step\ (ar\_delta\_read\ M)) \wedge m\_r$ 
     $\wedge fst (snd (mt\_state\ c\_w)) = AR\_SimCompute$ 
     $\wedge (\exists\ c\_w\_post.$ 
       $(c\_w, c\_w\_post) \in mttm\_step\ (ar\_delta\_compute\ M)$ 
       $\wedge (c\_w\_post, c'') \in (mttm\_step\ (ar\_delta\_write\ M)) \wedge m\_w)$ 
  by (intro exI[where x = c_b] exI[where x = c_w] exI[where x = m_r]
    exI[where x = Suc m_w] conjI wb chr cwC
    exI[where x = c_w_post] ccs chw_ext)

qed
thus ?thesis ..
next
  assume  $da: fst (snd (mt\_state\ c'')) = AR\_SimAdvance$ 
  — Cycle transition write to advance:  $c''$  opens the advance phase with zero
  advance-steps elapsed.
  have  $empty\_a: (c'', c'') \in (mttm\_step\ (ar\_delta\_advance\ M)) \wedge 0$ 
  by simp
  have  $ar\_walker\_in\_advance\ M\ cM\ c''$ 
  unfolding  $ar\_walker\_in\_advance\_def$ 
proof (intro conjI)
  show  $fst (snd (mt\_state\ c'')) = AR\_SimAdvance$  by (rule da)
  show  $\exists\ c\_b\ c\_w\ c\_a\ m\_r\ m\_w\ m\_a.$ 
     $ar\_walker\_at\_boundary\ M\ cM\ c\_b$ 
     $\wedge (c\_b, c\_w) \in (mttm\_step\ (ar\_delta\_read\ M)) \wedge m\_r$ 
     $\wedge fst (snd (mt\_state\ c\_w)) = AR\_SimCompute$ 
     $\wedge fst (snd (mt\_state\ c\_a)) = AR\_SimAdvance$ 
     $\wedge (\exists\ c\_w\_post.$ 
       $(c\_w, c\_w\_post) \in mttm\_step\ (ar\_delta\_compute\ M)$ 
       $\wedge (c\_w\_post, c\_a) \in (mttm\_step\ (ar\_delta\_write\ M)) \wedge m\_w$ 
       $\wedge (c\_a, c'') \in (mttm\_step\ (ar\_delta\_advance\ M)) \wedge m\_a)$ 
  by (intro exI[where x = c_b] exI[where x = c_w] exI[where x = c'']
    exI[where x = m_r] exI[where x = Suc m_w] exI[where x = 0]
    conjI wb chr cwC da
    exI[where x = c_w_post] ccs chw_ext empty_a)

qed
thus ?thesis ..
qed
qed

lemma  $ar\_walker\_step\_from\_in\_advance:$ 
fixes  $M :: ('q, 'a)\ mttm$ 
and  $cM :: ('a, 'q)\ mt\_config$ 
and  $c'\ c'' :: (sym4, 'q \times 'a\ ar\_stage)\ mt\_config$ 

```

```

assumes inv: ar_walker_in_advance M cM c'
and step: (c', c'') ∈ mttm_step (alphabet_reduce_delta M)
shows ar_walker_in_advance M cM c'' ∨ ar_walker_at_next M cM c''
proof –
have src: fst (snd (mt_state c')) = AR_SimAdvance
using inv unfolding ar_walker_in_advance_def by simp
obtain c_b c_w c_a m_r m_w m_a where
  wb: ar_walker_at_boundary M cM c_b
and chr: (c_b, c_w) ∈ (mttm_step (ar_delta_read M)) ^^ m_r
and cwC: fst (snd (mt_state c_w)) = AR_SimCompute
and caC: fst (snd (mt_state c_a)) = AR_SimAdvance
and rest: ∃ c_w_post.
  (c_w, c_w_post) ∈ mttm_step (ar_delta_compute M)
  ∧ (c_w_post, c_a) ∈ (mttm_step (ar_delta_write M)) ^^ m_w
  ∧ (c_a, c') ∈ (mttm_step (ar_delta_advance M)) ^^ m_a
using inv unfolding ar_walker_in_advance_def by blast
obtain c_w_post where
  ccs: (c_w, c_w_post) ∈ mttm_step (ar_delta_compute M)
and chw: (c_w_post, c_a) ∈ (mttm_step (ar_delta_write M)) ^^ m_w
and cha: (c_a, c') ∈ (mttm_step (ar_delta_advance M)) ^^ m_a
using rest by blast
have step_advance: (c', c'') ∈ mttm_step (ar_delta_advance M)
using ar_step_advance_lift[OF src step] .
have cha_ext: (c_a, c'') ∈ (mttm_step (ar_delta_advance M)) ^^ Suc m_a
using cha step_advance by (rule relpow_Suc_I)
from ar_step_from_SimAdvance_lands[OF step src]
have dest: fst (snd (mt_state c'')) = AR_SimAdvance
  ∨ fst (snd (mt_state c'')) = AR_SimNext .
thus ?thesis
proof
assume da: fst (snd (mt_state c'')) = AR_SimAdvance
have ar_walker_in_advance M cM c''
unfolding ar_walker_in_advance_def
proof (intro conjI)
show fst (snd (mt_state c'')) = AR_SimAdvance by (rule da)
show ∃ c_b c_w c_a m_r m_w m_a.
  ar_walker_at_boundary M cM c_b
  ∧ (c_b, c_w) ∈ (mttm_step (ar_delta_read M)) ^^ m_r
  ∧ fst (snd (mt_state c_w)) = AR_SimCompute
  ∧ fst (snd (mt_state c_a)) = AR_SimAdvance
  ∧ (∃ c_w_post.
    (c_w, c_w_post) ∈ mttm_step (ar_delta_compute M)
    ∧ (c_w_post, c_a) ∈ (mttm_step (ar_delta_write M)) ^^ m_w
    ∧ (c_a, c'') ∈ (mttm_step (ar_delta_advance M)) ^^ m_a)
by (intro exI[where x = c_b] exI[where x = c_w] exI[where x = c_a]
  exI[where x = m_r] exI[where x = m_w] exI[where x = Suc
m_a]
  conjI wb chr cwC caC
  exI[where x = c_w_post] ccs chw cha_ext)

```

```

qed
thus ?thesis ..
next
  assume dn: fst (snd (mt_state c'')) = AR_SimNext
  — Cycle transition advance to next: extended advance chain lands at c'', the
  redundant c_n = c' conjunct instantiates with c_n = c''.
  have ar_walker_at_next M cM c''
    unfolding ar_walker_at_next_def
  proof (intro conjI)
    show fst (snd (mt_state c'')) = AR_SimNext by (rule dn)
    show  $\exists c\_b\ c\_w\ c\_a\ c\_n\ m\_r\ m\_w\ m\_a.$ 
      ar_walker_at_boundary M cM c_b
       $\wedge (c\_b, c\_w) \in (\text{mttm\_step } (\text{ar\_delta\_read } M)) \hat{\wedge} m\_r$ 
       $\wedge \text{fst } (\text{snd } (\text{mt\_state } c\_w)) = \text{AR\_SimCompute}$ 
       $\wedge \text{fst } (\text{snd } (\text{mt\_state } c\_a)) = \text{AR\_SimAdvance}$ 
       $\wedge (\exists c\_w\_post.$ 
         $(c\_w, c\_w\_post) \in \text{mttm\_step } (\text{ar\_delta\_compute } M)$ 
         $\wedge (c\_w\_post, c\_a) \in (\text{mttm\_step } (\text{ar\_delta\_write } M)) \hat{\wedge} m\_w$ 
         $\wedge (c\_a, c\_n) \in (\text{mttm\_step } (\text{ar\_delta\_advance } M)) \hat{\wedge} m\_a$ 
         $\wedge c\_n = c'')$ 
      by (intro exI[where x = c_b] exI[where x = c_w] exI[where x = c_a]
        exI[where x = c''] exI[where x = m_r] exI[where x = m_w]
        exI[where x = Suc m_a] conjI wb chr cwC caC
        exI[where x = c_w_post] ccs chw cha_ext refl)
  qed
thus ?thesis ..
qed
qed

```

The sixth and final substep preservation closes the walker suite mechanically: an M' -step out of AR_SimNext lands at AR_SimRead (cycle close), AR_HaltAccept , or AR_HaltReject by $\text{ar_step_from_SimNext_lands}$. This is a pure dispatch lemma — establishing the boundary for the reconstructed M -successor at a cycle close is a separate cycle-level concern handled by $\text{ar_walker_cycle_close}$ and the chunked reverse engine. Keeping the at_next walker preservation thin keeps the suite uniform — all six are one-substep mechanical lemmas. Itself currently uncalled — the cycle close runs through $\text{ar_walker_cycle_close}$ directly — kept to complete the six-member walker-preservation suite.

```

lemma ar_walker_step_from_at_next:
  fixes M :: ('q, 'a) mttm
  and cM :: ('a, 'q) mt_config
  and c' c'' :: (sym4, 'q  $\times$  'a ar_stage) mt_config
  assumes inv: ar_walker_at_next M cM c'
  and step: (c', c'')  $\in$  mttm_step (alphabet_reduce_delta M)
  shows fst (snd (mt_state c'')) = AR_SimRead
     $\vee \text{fst } (\text{snd } (\text{mt\_state } c'')) = \text{AR\_HaltAccept}$ 
     $\vee \text{fst } (\text{snd } (\text{mt\_state } c'')) = \text{AR\_HaltReject}$ 

```


proof –

have $src: fst (snd (mt_state\ c')) = AR_SimNext$
 using inv **unfolding** $ar_walker_at_next_def$ **by** $simp$
 show $?thesis$ **using** $ar_step_from_SimNext_lands[OF\ step\ src]$.
qed

3.18 Cycle-close bridging lemma

An $AR_SimNext$ step closes a simulation cycle: the walker is re-established at a boundary for the next M -configuration. The successor is *reconstructed* from the pinned compute branch (the $\exists cM'$ in the conclusion): under non-deterministic $\delta_{tm}\ M$ the source cM may have several successors, so the fired branch is recovered from the walker's own witness chain rather than assumed. The forward arm is rebuilt entirely over the substep sub-relations and each walker witness chain is pinned by a $chain_ar_delta_X_to_Y_uniq$ suite.

The closing next step is inverted by $ar_next_step_inv$, which dispatches on the reconstructed M -state q' into three outcomes: continue ($q' \notin \{t, r\}$, landing at $AR_SimRead$), accept ($q' = t_{tm}\ M$) or reject ($q' = r_{tm}\ M$). All three re-establish $ar_walker_at_boundary\ M\ cMn\ c''$: the boundary invariant carries the halt cases too — $ar_simulates$'s state disjunction has dedicated accept/reject arms, and both $ar_posk_consistent$ and $ar_at_read_boundary$ are $AR_SimRead$ -guarded, hence vacuous off the read boundary. The three predicates are discharged from the explicit endpoint by the shared semantic lemmas ($ar_write_tape_correspondence$, $ar_advance_newsimpos$, $ar_newpos_atLE_iff$). No union chain is pinned and $ar_simulates_forward_step$ is not invoked.

lemma $ar_walker_cycle_close$:

fixes $M :: ('q, 'a)\ mttm$
 and $w :: 'a\ list$
 and $cM :: ('a, 'q)\ mt_config$
 and $c'\ c'' :: (sym4, 'q \times 'a\ ar_stage)\ mt_config$
 assumes $inv: ar_walker_at_next\ M\ cM\ c'$
 and $step: (c', c'') \in mttm_step\ (alphabet_reduce_delta\ M)$
 and $vM: valid_mttm\ M$
 and $qQ: mt_state\ cM \in Q_{tm}\ M$
 and $kge2: 2 \leq block_width\ (\Gamma_{tm}\ M)$
 and $tapeG: \bigwedge k. mt_tape\ cM\ k\ (mt_pos\ cM\ k) \in \Gamma_{tm}\ M$
 and $w_sub: set\ w \subseteq Sigma_{tm}\ M$
 and $reach_M: (init_config_mttm\ M\ w, cM) \in (mttm_step\ (\delta_{tm}\ M))^*$
 and $lebl: le_{tm}\ M \neq bl_{tm}\ M$
 shows $\exists cM'. (cM, cM') \in mttm_step\ (\delta_{tm}\ M)$
 and $ar_walker_at_boundary\ M\ cM'\ c''$

proof –

let $?k = block_width\ (\Gamma_{tm}\ M)$
 let $?aM = \lambda k. mt_tape\ cM\ k\ (mt_pos\ cM\ k)$
 have $kpos_tm: 0 < k_{tm}\ M$ **using** vM **by** $(cases\ M)\ auto$

— Unpack the `ar_walker_at_next` witness.

```

obtain c_b c_w c_a c_n m_r m_w m_a c_w_post where
  wb: ar_walker_at_boundary M cM c_b
  and chain_r:  $(c_b, c_w) \in (\text{mttm\_step } (\text{ar\_delta\_read } M)) \wedge \wedge m_r$ 
  and cw_cpu: fst (snd (mt_state c_w)) = AR_SimCompute
  and ca_adv: fst (snd (mt_state c_a)) = AR_SimAdvance
  and cstep:  $(c_w, c_w\_post) \in \text{mttm\_step } (\text{ar\_delta\_compute } M)$ 
  and wchain:  $(c_w\_post, c_a) \in (\text{mttm\_step } (\text{ar\_delta\_write } M)) \wedge \wedge m_w$ 
  and achain0:  $(c_a, c_n) \in (\text{mttm\_step } (\text{ar\_delta\_advance } M)) \wedge \wedge m_a$ 
  and cn_eq: c_n = c'
  using inv unfolding ar_walker_at_next_def by blast
have achain:  $(c_a, c') \in (\text{mttm\_step } (\text{ar\_delta\_advance } M)) \wedge \wedge m_a$ 
  using achain0 cn_eq by simp
have c'_next: fst (snd (mt_state c')) = AR_SimNext
  using inv unfolding ar_walker_at_next_def by simp
have sim: ar_simulates M cM c_b
  using wb unfolding ar_walker_at_boundary_def by simp
have pcons: ar_posk_consistent M cM c_b
  using wb unfolding ar_walker_at_boundary_def by simp
have rbnd: ar_at_read_boundary M c_b
  using wb unfolding ar_walker_at_boundary_def by simp
have cb_idx: fst (snd (mt_state c_b)) = AR_SimRead
proof (cases m_r)
  case 0
    have cb_eq: c_b = c_w using chain_r 0 by simp
    have cb_compute: fst (snd (mt_state c_b)) = AR_SimCompute
      using cw_cpu cb_eq by simp
    obtain qM' stg where st_b: mt_state c_b = (qM', stg)
      by (cases mt_state c_b) auto
    obtain idx tk i buf dvec posk where
      sg_b: stg = (idx, tk, i, buf, dvec, posk) by (cases stg) auto
    have idx_cpu: idx = AR_SimCompute using cb_compute st_b sg_b by simp
    have  $(\text{idx} = \text{AR\_SimRead} \wedge qM' \notin \{t\_tm\ M, r\_tm\ M\})$ 
       $\vee (qM' = t\_tm\ M \wedge stg = \text{ar\_accept\_stage } (bl\_tm\ M))$ 
       $\vee (qM' = r\_tm\ M \wedge stg = \text{ar\_reject\_stage } (bl\_tm\ M))$ 
      using sim st_b sg_b unfolding ar_simulates_def by (auto split: prod.splits)
    hence False using idx_cpu sg_b
      by (auto simp: ar_accept_stage_def ar_reject_stage_def)
    thus ?thesis ..
  next
    case (Suc m')
    have hsuc:  $(c_b, c_w) \in (\text{mttm\_step } (\text{ar\_delta\_read } M)) \wedge \wedge \text{Suc } m'$ 
      using chain_r Suc by simp
    obtain c_1 where first:  $(c_b, c_1) \in \text{mttm\_step } (\text{ar\_delta\_read } M)$ 
      using relpow_Suc_D2[OF hsuc] by blast
    from first obtain S ts n S' aw dir where
      ceq: c_b = Config_M S ts n
      and rel:  $(S, (\lambda k. ts\ k\ (n\ k)), S', aw, dir) \in \text{ar\_delta\_read } M$ 
      by (auto elim: mttm_step.cases)

```

```

    have fst (snd S) = AR_SimRead using ar_delta_read_src[OF rel] .
    thus ?thesis using ceq by simp
qed
obtain qM' stg where st_b: mt_state c_b = (qM', stg)
  by (cases mt_state c_b) auto
obtain idx tk i buf dvec posk where
  sg_b: stg = (idx, tk, i, buf, dvec, posk) by (cases stg) auto
have idx_read: idx = AR_SimRead using cb_idx st_b sg_b by simp
have qM'_eq: qM' = mt_state cM
  using sim st_b sg_b unfolding ar_simulates_def by (auto split: prod.splits)
have tcorr:  $\forall k < k\_tm\ M. ar\_tape\_correspondence\ (\Gamma\_tm\ M)\ (le\_tm\ M)$ 
  (bl_tm M)
  (mt_tape cM k) (mt_tape c_b k)
  using sim st_b sg_b unfolding ar_simulates_def by (auto split: prod.splits)
have ppos:  $\bigwedge k. mt\_pos\ c_b\ k = sim\_pos\ ?k\ (mt\_pos\ cM\ k)$ 
  using sim st_b sg_b idx_read
  unfolding ar_simulates_def by (auto split: prod.splits)
have pkok:  $\bigwedge k. posk\ k = AR\_AtLE \longleftrightarrow mt\_pos\ cM\ k = 0$ 
  using pcons st_b sg_b idx_read
  unfolding ar_posk_consistent_def by (auto split: prod.splits)
have bnd_unfold: idx = AR_SimRead
   $\longrightarrow (tk = 0 \wedge i = 0$ 
     $\wedge (\forall k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\})$ 
     $\wedge ar\_stage\_bounded\ (bl\_tm\ M)\ (k\_tm\ M)$ 
     $(idx, tk, i, buf, dvec, posk))$ 
  using rbnd st_b sg_b
  unfolding ar_at_read_boundary_def by (auto split: prod.splits)
have tk0: tk = 0 and i0: i = 0
  and bufG:  $\bigwedge k. buf\ k \in \Gamma\_tm\ M \cup \{bl\_tm\ M\}$ 
  using bnd_unfold idx_read by simp_all
have src_b: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, 0, 0, buf, dvec, posk)
  using bnd_unfold idx_read tk0 i0 by simp
have pad_b:  $\forall j \geq k\_tm\ M. mt\_tape\ c_b\ j\ (mt\_pos\ c_b\ j) = BLANK4$ 
  using rbnd unfolding ar_at_read_boundary_def by (auto split: prod.splits)
have stg0: mt_state c_b = (mt_state cM, AR_SimRead, 0, 0, buf, dvec, posk)
  using st_b sg_b qM'_eq idx_read tk0 i0 by simp
have buf_tail:  $\forall j \geq k\_tm\ M. buf\ j = bl\_tm\ M$ 
  using src_b by (simp add: ar_stage_bounded_def)
have dvec_tail:  $\forall j \geq k\_tm\ M. dvec\ j = dir.N$ 
  using src_b by (simp add: ar_stage_bounded_def)
have posk_tail:  $\forall j \geq k\_tm\ M. posk\ j = AR\_AtLE$ 
  using src_b by (simp add: ar_stage_bounded_def)
have valcM: valid_config_mttm M cM
  using valid_reach_mttm[OF vM w_sub reach_M] .
have aM_tail:  $\forall j \geq k\_tm\ M. mt\_tape\ cM\ j\ (mt\_pos\ cM\ j) = bl\_tm\ M$ 
  using valid_config_mttm_blank_tail[OF valcM] by blast
— Read-pin: reconstruct the canonical compute config (guarded read buffer /
posk / position), pin the walker's read chain to it.

```

```

let ?rbuf =  $\lambda k.$  if  $k < k\_tm\ M$  then ?aM  $k$  else buf  $k$ 
let ?rposk =  $\lambda k.$  if  $k < k\_tm\ M$ 
    then (if mt_pos cM  $k = 0$  then AR_AtLE
        else if mt_pos cM  $k = 1$  then AR_AtFirstProper
        else AR_AtFurtherProper)
    else posk  $k$ 
let ?rpos =  $\lambda k.$  if  $k < k\_tm\ M$ 
    then (if mt_pos cM  $k = 0$  then Suc 0
        else sim_pos ?k (mt_pos cM  $k$ ) + ?k)
    else mt_pos c_b  $k$ 
obtain c_r m_phase where
    r_chain: (c_b, c_r)  $\in$  (mttm_step (ar_delta_read M))  $\wedge$  m_phase
    and r_state: mt_state c_r = (mt_state cM, AR_SimCompute, k_unidx 0,
0,
    ?rbuf, dvec, ?rposk)
    and r_tape: mt_tape c_r = mt_tape c_b
    and r_pos: mt_pos c_r = ?rpos
    using ar_read_phase_in_sub[OF vM qQ kge2 stg0 tcorr ppos pkok tapeG bufG
    pad_b src_b]
    by blast
    have r_cpu: fst (snd (mt_state c_r)) = AR_SimCompute using r_state by
simp
    have cw_eq_cr: c_w = c_r
    using chain_ar_delta_read_to_SimCompute_uniq[OF chain_r r_chain
cw_cpu r_cpu]
    by simp
    have cw_state: mt_state c_w = (mt_state cM, AR_SimCompute, k_unidx 0,
0,
    ?rbuf, dvec, ?rposk)
    using cw_eq_cr r_state by simp
    — Invert the compute step: the explicit post-state and the fired delta_tm M
branch (on the guarded read buffer).
    obtain q' m_a' m_d where
    mdelta: (mt_state cM, ?rbuf, q', m_a', m_d)  $\in$  delta_tm M
    and cupost_state: mt_state c_w_post = (q', AR_SimWrite, 0, 0, m_a',
m_d, ?rposk)
    and cupost_tape: mt_tape c_w_post = mt_tape c_w
    and cupost_pos: mt_pos c_w_post = mt_pos c_w
    using ar_compute_step_inv_sub[OF cstep cw_state] by blast
    obtain qM tsM nM where cM_eq: cM = Config_M qM tsM nM by (cases cM)
auto
    have qM_eq: qM = mt_state cM using cM_eq by simp
    have aM_eq: ( $\lambda k.$  tsM  $k$  (nM  $k$ )) = ?aM using cM_eq by simp
    have rbuf_eq: ?rbuf = ?aM
    proof (rule ext)
    fix k show ?rbuf  $k = ?aM\ k$ 
    proof (cases  $k < k\_tm\ M$ )
    case True thus ?thesis by simp
    next

```

```

    case False
    have ?rbuf k = bl_tm M using buf_tail False by simp
    moreover have ?aM k = bl_tm M using aM_tail False by simp
    ultimately show ?thesis by simp
  qed
  qed
  have mdelta_aM: (mt_state cM, ?aM, q', m_a', m_d) ∈ delta_tm M
    using mdelta_rbuf_eq by simp
  define cMn where cMn_def: cMn = Config_M q' (λk. (tsM k)(nM k := m_a'
k))
    (λk. go_dir (m_d k) (nM k))
  have mdelta': (qM, (λk. tsM k (nM k)), q', m_a', m_d) ∈ delta_tm M
    using mdelta_aM qM_eq aM_eq by simp
  have m_step: (cM, cMn) ∈ mttm_step (delta_tm M)
    using mttm_step.step[where ts = tsM and n = nM, OF mdelta'] cM_eq
cMn_def by simp
  have q'Q: q' ∈ Q_tm M using valid_mttm_delta(3)[OF vM mdelta_aM] .
  have a'G: m_a' k ∈ Γ_tm M for k using valid_mttm_delta(4)[OF vM
mdelta_aM] .
  have a'val: ∀ k. m_a' k ∈ Γ_tm M ∪ {bl_tm M} using a'G by blast
  have dsupp: ∀ j ≥ k_tm M. m_a' j = bl_tm M ∧ m_d j = dir.N
    using valid_mttm_delta_support[OF vM mdelta_aM] by blast
  — Write phase: post-write tape / position setup and pad / src for the compute-exit
config, then pin the walker's write chain.
  have pad_cwpost: ∀ j ≥ k_tm M. mt_tape c_w_post j (mt_pos c_w_post j)
= BLANK4
  proof (intro allI impI)
    fix j assume jk: k_tm M ≤ j
    have mt_pos c_w_post j = mt_pos c_b j
      using cwpost_pos cw_eq_cr r_pos jk by simp
    moreover have mt_tape c_w_post j = mt_tape c_b j
      using cwpost_tape cw_eq_cr r_tape by simp
    ultimately show mt_tape c_w_post j (mt_pos c_w_post j) = BLANK4
      using pad_b jk by simp
  qed
  have src_cwpost: ar_stage_bounded (bl_tm M) (k_tm M)
    (AR_SimWrite, 0, 0, m_a', m_d, ?rposk)
    using kpos_tm dsupp posk_tail by (simp add: ar_stage_bounded_def)
  have tcorr_w: ∀ k < k_tm M. ar_tape_correspondence (Γ_tm M) (le_tm M)
(bl_tm M)
    (mt_tape cM k) (mt_tape c_w_post k)
  proof (intro allI impI)
    fix k assume kN: k < k_tm M
    have mt_tape c_w_post k = mt_tape c_b k
      using cwpost_tape cw_eq_cr r_tape by simp
    thus ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
      (mt_tape cM k) (mt_tape c_w_post k)
      using tcorr kN by simp
  qed
  qed

```

```

have ppos_w:  $\forall k < k\_tm\ M. mt\_pos\ c\_w\_post\ k = (if\ mt\_pos\ cM\ k = 0\ then$ 
Suc 0
       $else\ sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k)$ 
using cwpost_pos cw_eq_cr r_pos by simp
have a'le0_w:  $\forall k < k\_tm\ M. mt\_pos\ cM\ k = 0 \longrightarrow m\_a'\ k = le\_tm\ M$ 
proof (intro allI impI)
  fix k assume kN:  $k < k\_tm\ M$  and p0:  $mt\_pos\ cM\ k = 0$ 
  have tsk0:  $tsM\ k\ 0 = le\_tm\ M$  using valcM cM_eq kN by (cases M) auto
  have ( $\lambda j. tsM\ j\ (nM\ j)$ )  $k = le\_tm\ M$  using tsk0 p0 cM_eq by simp
  thus  $m\_a'\ k = le\_tm\ M$  using valid_mttm_deltaLE[OF vM mdelta'] by simp
qed
have poskle_w:  $\forall k < k\_tm\ M. ?rposk\ k = AR\_AtLE \longleftrightarrow mt\_pos\ cM\ k = 0$ 
by auto
obtain c_write mw where
  w_chain:  $(c\_w\_post, c\_write) \in (mttm\_step\ (ar\_delta\_write\ M)) \wedge mw$ 
  and w_state:  $mt\_state\ c\_write = (q', AR\_SimAdvance, k\_unidx\ 0, 0, m\_a',$ 
m_d,
      ?rposk)
  and w_tape:  $mt\_tape\ c\_write = (\lambda k. if\ k < k\_tm\ M$ 
     $then\ (if\ mt\_pos\ cM\ k = 0\ then\ mt\_tape\ c\_w\_post\ k$ 
     $else\ (\lambda pos. if\ sim\_pos\ ?k\ (mt\_pos\ cM\ k) \leq pos$ 
     $\wedge pos < sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k$ 
     $then\ write\_bit\ (\Gamma\_tm\ M)\ (bl\_tm\ M)\ (m\_a'\ k)$ 
     $(pos - sim\_pos\ ?k\ (mt\_pos\ cM\ k))$ 
     $else\ mt\_tape\ c\_w\_post\ k\ pos))$ 
     $else\ mt\_tape\ c\_w\_post\ k)$ 
  and w_pos:  $mt\_pos\ c\_write = mt\_pos\ c\_w\_post$ 
using ar_write_phase_in_sub[OF vM q'Q kge2 cwpost_state tcorr_w ppos_w
  poskle_w a'val pad_cwpost src_cwpost]
by blast
have c_write_adv:  $fst\ (snd\ (mt\_state\ c\_write)) = AR\_SimAdvance$ 
using w_state by simp
have ca_eq:  $c\_a = c\_write$ 
using chain_ar_delta_write_to_SimAdvance_uniq[OF wchain w_chain
ca_adv
      c_write_adv]
by simp
— Advance phase: bridge the tk-field, displacement facts (active tapes), pad / src
for the write-exit config, then pin.
have w_state':  $mt\_state\ c\_write = (q', AR\_SimAdvance, 0, 0, m\_a', m\_d,$ 
?rposk)
using w_state by (simp add: k_unidx_zero)
have pad_cwrite:  $\forall j \geq k\_tm\ M. mt\_tape\ c\_write\ j\ (mt\_pos\ c\_write\ j) =$ 
BLANK4
proof (intro allI impI)
  fix j assume jk:  $k\_tm\ M \leq j$ 
  have  $mt\_tape\ c\_write\ j = mt\_tape\ c\_w\_post\ j$  using w_tape jk by simp
  moreover have  $mt\_pos\ c\_write\ j = mt\_pos\ c\_w\_post\ j$  using w_pos by
simp

```

```

ultimately show mt_tape c_write j (mt_pos c_write j) = BLANK4
using pad_cwpost jk by simp
qed
have src_cwrite: ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimAdvance, 0, 0, m_a', m_d, ?rposk)
using kpos_tm dsupp posk_tail by (simp add: ar_stage_bounded_def)
have dge1:  $\forall k < k\_tm\ M. m\_d\ k \neq dir.R \longrightarrow 0 < ar\_disp\ ?k\ (m\_d\ k)\ (?rposk\ k)$ 
proof (intro allI impI)
  fix k assume kN:  $k < k\_tm\ M$  and dkR:  $m\_d\ k \neq dir.R$ 
  show  $0 < ar\_disp\ ?k\ (m\_d\ k)\ (?rposk\ k)$ 
  proof (cases mt_pos cM k = 0)
    case True
      have nM0:  $nM\ k = 0$  using True cM_eq by simp
      have tsM k 0 = le_tm M using valcM cM_eq kN by (cases M) auto
      hence  $(\lambda j. tsM\ j\ (nM\ j))\ k = le\_tm\ M$  using nM0 by simp
      hence  $m\_d\ k \in \{dir.N, dir.R\}$  using valid_mttm_deltaLE[OF vM mdelta']
    by simp
      hence  $m\_d\ k = dir.N$  using dkR by auto
      moreover have ?rposk k = AR_AtLE using True kN by simp
      ultimately show  $0 < ar\_disp\ ?k\ (m\_d\ k)\ (?rposk\ k)$  by simp
  next
    case False
      have dN_or_L:  $m\_d\ k = dir.N \vee m\_d\ k = dir.L$ 
      using dkR by (cases m_d k) auto
      have rk: ?rposk k = AR_AtFirstProper  $\vee$  ?rposk k = AR_AtFurtherProper
      using False kN by auto
      from dN_or_L rk kge2 show  $0 < ar\_disp\ ?k\ (m\_d\ k)\ (?rposk\ k)$  by auto
  qed
qed
have notLE:  $\forall k < k\_tm\ M. \forall m. m < ar\_disp\ ?k\ (m\_d\ k)\ (?rposk\ k) \longrightarrow mt\_tape\ c\_write\ k\ (mt\_pos\ c\_write\ k - m) \neq LE4$ 
proof (intro allI impI)
  fix k m assume kN:  $k < k\_tm\ M$ 
  and mlt:  $m < ar\_disp\ ?k\ (m\_d\ k)\ (?rposk\ k)$ 
  have tcorr_wk: ar_tape_correspondence ( $\Gamma\_tm\ M$ ) ( $le\_tm\ M$ ) ( $bl\_tm\ M$ )
    ( $mt\_tape\ cM\ k$ ) ( $mt\_tape\ c\_w\_post\ k$ )
  using tcorr_w kN by blast
  have wcpes:  $mt\_pos\ c\_write\ k = (if\ mt\_pos\ cM\ k = 0\ then\ Suc\ 0\ else\ sim\_pos\ ?k\ (mt\_pos\ cM\ k) + ?k)$ 
  using w_pos ppos_w kN by simp
  have wtape_notLE:  $mt\_tape\ c\_write\ k\ pos \neq LE4$  if pos1:  $1 \leq pos$  for pos
  proof (cases mt_pos cM k = 0)
    case True
      have  $mt\_tape\ c\_write\ k\ pos = mt\_tape\ c\_w\_post\ k\ pos$ 
      using w_tape True kN by simp
      thus ?thesis
      using ar_tape_correspondence_not_LE4[OF tcorr_wk pos1] by simp
    next

```

```

case False
have wexp: mt_tape c_write k pos
  = (if sim_pos ?k (mt_pos cM k) ≤ pos
      ∧ pos < sim_pos ?k (mt_pos cM k) + ?k
      then write_bit (Γ_tm M) (bl_tm M) (m_a' k)
          (pos - sim_pos ?k (mt_pos cM k))
      else mt_tape c_w_post k pos)
  using w_tape False kN by simp
show ?thesis
proof (cases sim_pos ?k (mt_pos cM k) ≤ pos
        ∧ pos < sim_pos ?k (mt_pos cM k) + ?k)
  case True
  have b: pos - sim_pos ?k (mt_pos cM k) < ?k using True by linarith
  have mt_tape c_write k pos
    = write_bit (Γ_tm M) (bl_tm M) (m_a' k)
      (pos - sim_pos ?k (mt_pos cM k))
    using wexp True by simp
  thus ?thesis using write_bit_not_LE4[OF b] by simp
next
  case False
  have nreg: ¬ (sim_pos ?k (mt_pos cM k) ≤ pos
    ∧ pos < sim_pos ?k (mt_pos cM k) + ?k)
    using False by simp
  have mt_tape c_write k pos = mt_tape c_w_post k pos
    using wexp by (simp add: if_not_P[OF nreg])
  thus ?thesis
    using ar_tape_correspondence_not_LE4[OF tcorr_wk pos1] by simp
qed
qed
have rge: ar_disp ?k (m_d k) (?rposk k) ≤ mt_pos c_write k
proof (cases mt_pos cM k = 0)
  case True
  have ar_disp ?k (m_d k) (?rposk k) ≤ Suc 0
    using True kN by (cases m_d k) auto
  thus ?thesis using wcpes True by simp
next
  case nz: False
  show ?thesis
  proof (cases mt_pos cM k = 1)
    case True
    have ar_disp ?k (m_d k) (?rposk k) ≤ Suc ?k
      using True kN by (cases m_d k) auto
    thus ?thesis using wcpes True by (simp add: sim_pos_def)
  next
    case False
    have p2: 2 ≤ mt_pos cM k using nz False by simp
    have d2: ar_disp ?k (m_d k) (?rposk k) ≤ 2 * ?k
      using nz False kN by (cases m_d k) auto
    have cw_eq2: mt_pos c_write k = (mt_pos cM k - 1) * ?k + 1 + ?k

```



```

      using wcpow nz by (simp add: sim_pos_def)
      have (1::nat) ≤ mt_pos cM k - 1 using p2 by simp
      hence 1 * ?k ≤ (mt_pos cM k - 1) * ?k by (rule mult_le_mono1)
      hence kk: ?k ≤ (mt_pos cM k - 1) * ?k by (simp only: mult_1_left)
      have 2 * ?k ≤ mt_pos c_write k using kk cw_eq2 by linarith
      thus ?thesis using d2 by linarith
    qed
  qed
  have m < mt_pos c_write k using mlt_rge by simp
  hence 1 ≤ mt_pos c_write k - m by simp
  thus mt_tape c_write k (mt_pos c_write k - m) ≠ LE4 by (rule
wtape_notLE)
  qed
  let ?apok = λk. if k < k_tm M then ar_newpos (m_d k) (?rposk k) else ?rposk
k
  let ?apos = λk. if k < k_tm M
    then (if m_d k = dir.R then mt_pos c_write k
      else mt_pos c_write k - ar_disp ?k (m_d k) (?rposk k))
    else mt_pos c_write k
  obtain c_adv ma where
    a_chain: (c_write, c_adv) ∈ (mttm_step (ar_delta_advance M)) ^^ ma
    and a_state: mt_state c_adv = (q', AR_SimNext, k_unidx 0, 0, m_a', m_d,
?apok)
    and a_tape: mt_tape c_adv = mt_tape c_write
    and a_pos: mt_pos c_adv = ?apos
    using ar_advance_phase_in_sub[OF vM q'Q kge2 w_state' a'val dge1 notLE
pad_cwrite src_cwrite]
  by blast
  have c_adv_nxt: fst (snd (mt_state c_adv)) = AR_SimNext using a_state
by simp
  have achain': (c_write, c') ∈ (mttm_step (ar_delta_advance M)) ^^ m_a
    using achain ca_eq by simp
  have c'_eq: c' = c_adv
    using chain_ar_delta_advance_to_SimNext_uniq[OF achain' a_chain
c'_next
c_adv_nxt]
  by simp
  have c'_state: mt_state c' = (q', AR_SimNext, k_unidx 0, 0, m_a', m_d,
?apok)
  using c'_eq a_state by simp
  — Closing next step: invert it; dispatch on the reconstructed M-state q'.
  have nxt: mt_tape c'' = mt_tape c' ∧ mt_pos c'' = mt_pos c'
    ∧ ((q' ∉ {t_tm M, r_tm M}
      ∧ mt_state c'' = (q', AR_SimRead, 0, 0, m_a', m_d, ?apok))
      ∨ (q' = t_tm M
      ∧ mt_state c'' = (t_tm M, ar_accept_stage (bl_tm M)))
      ∨ (q' = r_tm M
      ∧ mt_state c'' = (r_tm M, ar_reject_stage (bl_tm M))))
    using ar_next_step_inv[OF step c'_state] .

```

```

have next_tape: mt_tape c'' = mt_tape c' using next by simp
have next_pos: mt_pos c'' = mt_pos c' using next by simp
have next_disj:
  (q' ∉ {t_tm M, r_tm M}
   ∧ mt_state c'' = (q', AR_SimRead, 0, 0, m_a', m_d, ?apok))
  ∨ (q' = t_tm M
     ∧ mt_state c'' = (t_tm M, ar_accept_stage (bl_tm M)))
  ∨ (q' = r_tm M
     ∧ mt_state c'' = (r_tm M, ar_reject_stage (bl_tm M)))
using next by simp
— Re-establish the boundary predicates for cMn at c''.
have st_cMn: mt_state cMn = q' using cMn_def by simp
have padpos0: ∀ j ≥ k_tm M. mt_pos cM j = 0 using pkok posk_tail by blast
have dposL: m_d k ≠ dir.L if p0: mt_pos cM k = 0 and kN: k < k_tm M for
k
proof —
  have nM0: nM k = 0 using p0 cM_eq by simp
  have tsM k 0 = le_tm M using valcM cM_eq kN by (cases M) auto
  hence (λj. tsM j (nM j)) k = le_tm M using nM0 by simp
  hence m_d k ∈ {dir.N, dir.R} using valid_mttm_deltaLE[OF vM mdelta]
by simp
  thus m_d k ≠ dir.L by auto
qed
have cwpost_b: mt_tape c_w_post = mt_tape c_b
  using cwpost_tape cw_eq cr_r_tape by simp
have cM1tape: mt_tape cMn k = (mt_tape cM k)(mt_pos cM k := m_a' k)
for k
  using cMn_def cM_eq by simp
have wt_def: mt_tape c'' k pos
  = (if mt_pos cM k = 0 then mt_tape c_b k pos
     else if sim_pos ?k (mt_pos cM k) ≤ pos
       ∧ pos < sim_pos ?k (mt_pos cM k) + ?k
       then write_bit (Γ_tm M) (bl_tm M) (m_a' k)
         (pos - sim_pos ?k (mt_pos cM k))
       else mt_tape c_b k pos) for k pos
proof (cases k < k_tm M)
case True
  thus ?thesis using next_tape c'_eq a_tape w_tape cwpost_b by simp
next
case False
  hence kge: k_tm M ≤ k by simp
  have p0: mt_pos cM k = 0 using padpos0 kge by blast
  have mt_tape c'' k = mt_tape c_b k
    using next_tape c'_eq a_tape w_tape cwpost_b kge by simp
  thus ?thesis using p0 by simp
qed
have tcorr_1: ∀ k < k_tm M. ar_tape_correspondence (Γ_tm M) (le_tm M)
(bl_tm M)
  (mt_tape cMn k) (mt_tape c'' k)

```

```

proof (intro allI impI)
  fix k assume kN: k < k_tm M
  show ar_tape_correspondence (Γ_tm M) (le_tm M) (bl_tm M)
    (mt_tape cMn k) (mt_tape c'' k)
  by (rule ar_write_tape_correspondence
    [OF kge2 kN tcorr cM1tape a'le0_w wt_def])
qed
have apos_1: mt_pos c'' k = sim_pos ?k (mt_pos cMn k) for k
proof (cases k < k_tm M)
  case kN: True
  have posM1: mt_pos cMn k = go_dir (m_d k) (mt_pos cM k)
    using cMn_def cM_eq by simp
  have key: (if m_d k = dir.R
    then (if mt_pos cM k = 0 then Suc 0
      else sim_pos ?k (mt_pos cM k) + ?k)
    else (if mt_pos cM k = 0 then Suc 0
      else sim_pos ?k (mt_pos cM k) + ?k
      - ar_disp ?k (m_d k)
      (if mt_pos cM k = 0 then AR_AtLE
        else if mt_pos cM k = 1 then AR_AtFirstProper
        else AR_AtFurtherProper)))
    = sim_pos ?k (go_dir (m_d k) (mt_pos cM k))
  proof (rule ar_advance_newsimpos)
  show 2 ≤ ?k by (rule kge2)
  show mt_pos cM k = 0 ⇒ m_d k ≠ dir.L using dposL kN by blast
qed
have cadv_pos: mt_pos c'' k = (if m_d k = dir.R
  then (if mt_pos cM k = 0 then Suc 0
    else sim_pos ?k (mt_pos cM k) + ?k)
  else (if mt_pos cM k = 0 then Suc 0
    else sim_pos ?k (mt_pos cM k) + ?k
    - ar_disp ?k (m_d k)
    (if mt_pos cM k = 0 then AR_AtLE
      else if mt_pos cM k = 1 then AR_AtFirstProper
      else AR_AtFurtherProper)))
  using nxt_pos c'_eq a_pos w_pos ppos_w kN by simp
show ?thesis by (simp add: cadv_pos key posM1)
next
case False
hence kge: k_tm M ≤ k by simp
have dN: m_d k = dir.N using dsupp kge by blast
have posM1: mt_pos cMn k = go_dir (m_d k) (mt_pos cM k)
  using cMn_def cM_eq by simp
have mt_pos c'' k = mt_pos c_b k
  using nxt_pos c'_eq a_pos w_pos cwpost_pos cw_eq_cr r_pos kge by simp
also have ... = sim_pos ?k (mt_pos cM k) using ppos by simp
also have ... = sim_pos ?k (mt_pos cMn k) using posM1 dN by simp
finally show ?thesis .
qed

```

```

have aposk_1: ?aposk k = AR_AtLE  $\longleftrightarrow$  mt_pos cMn k = 0 for k
proof (cases k < k_tm M)
  case kN: True
  have posM1: mt_pos cMn k = go_dir (m_d k) (mt_pos cM k)
    using cMn_def cM_eq by simp
  have key: (ar_newpos (m_d k)
    (if mt_pos cM k = 0 then AR_AtLE
     else if mt_pos cM k = 1 then AR_AtFirstProper
     else AR_AtFurtherProper) = AR_AtLE)
     $\longleftrightarrow$  go_dir (m_d k) (mt_pos cM k) = 0
  proof (rule ar_newpos_atLE_iff)
    show mt_pos cM k = 0  $\implies$  m_d k  $\neq$  dir.L using dposL kN by blast
  qed
  have aposk_k: ?aposk k = ar_newpos (m_d k)
    (if mt_pos cM k = 0 then AR_AtLE
     else if mt_pos cM k = 1 then AR_AtFirstProper
     else AR_AtFurtherProper)
    using kN by simp
  show ?thesis by (simp add: aposk_k key posM1)
next
  case False
  hence kge: k_tm M  $\leq$  k by simp
  have dN: m_d k = dir.N using dsupp kge by blast
  have p0: mt_pos cM k = 0 using padpos0 kge by blast
  have posM1: mt_pos cMn k = go_dir (m_d k) (mt_pos cM k)
    using cMn_def cM_eq by simp
  have mt_pos cMn k = 0 using posM1 dN p0 by simp
  moreover have posk k = AR_AtLE using posk_tail kge by blast
  moreover have ?aposk k = posk k using kge by simp
  ultimately show ?thesis by simp
qed
have pad_c'':  $\forall j \geq k\_tm\ M. mt\_tape\ c''\ j\ (mt\_pos\ c''\ j) = BLANK4$ 
proof (intro allI impI)
  fix j assume jk: k_tm M  $\leq$  j
  have mt_pos c'' j = mt_pos c_write j
    using nxt_pos c'_eq a_pos jk by simp
  moreover have mt_tape c'' j = mt_tape c_write j
    using nxt_tape c'_eq a_tape by simp
  ultimately show mt_tape c'' j (mt_pos c'' j) = BLANK4
    using pad_cwrite jk by simp
qed
have src_read'': ar_stage_bounded (bl_tm M) (k_tm M)
  (AR_SimRead, 0, 0, m_a', m_d, ?aposk)
  using kpos_tm dsupp posk_tail by (simp add: ar_stage_bounded_def)
show ?thesis
proof (intro exI[where x = cMn] conjI)
  show (cM, cMn)  $\in$  mttm_step (delta_tm M) by (rule m_step)
  show ar_walker_at_boundary M cMn c''
proof -

```

```

consider
  (cont)  $q' \notin \{t\_tm\ M, r\_tm\ M\}$ 
           $mt\_state\ c'' = (q', AR\_SimRead, 0, 0, m\_a', m\_d, ?apok)$ 
| (acc)  $q' = t\_tm\ M$ 
           $mt\_state\ c'' = (t\_tm\ M, ar\_accept\_stage\ (bl\_tm\ M))$ 
| (rej)  $q' = r\_tm\ M$ 
           $mt\_state\ c'' = (r\_tm\ M, ar\_reject\_stage\ (bl\_tm\ M))$ 
using next_disj by blast
thus ?thesis
proof cases
  case cont
  have notT:  $q' \neq t\_tm\ M$  using cont(1) by simp
  have notR:  $q' \neq r\_tm\ M$  using cont(1) by simp
  show ?thesis
    unfolding ar_walker_at_boundary_def
  proof (intro conjI)
    show ar_simulates M cMn c''
      by (simp add: ar_simulates_def Let_def cont(2) st_cMn notT notR
          tcrr_1 apos_1)
    show ar_posk_consistent M cMn c''
      by (simp add: ar_posk_consistent_def cont(2) apok_1)
    show ar_at_read_boundary M c''
      by (simp add: ar_at_read_boundary_def cont(2) a'G pad_c'' src_read'')
  qed
next
  case acc
  show ?thesis
    unfolding ar_walker_at_boundary_def
  proof (intro conjI)
    show ar_simulates M cMn c''
      by (simp add: ar_simulates_def Let_def acc(2) st_cMn acc(1)
          tcrr_1 ar_accept_stage_def)
    show ar_posk_consistent M cMn c''
      by (simp add: ar_posk_consistent_def acc(2) ar_accept_stage_def)
    show ar_at_read_boundary M c''
      by (simp add: ar_at_read_boundary_def acc(2) ar_accept_stage_def
          pad_c'')
  qed
next
  case rej
  show ?thesis
    unfolding ar_walker_at_boundary_def
  proof (intro conjI)
    show ar_simulates M cMn c''
      by (simp add: ar_simulates_def Let_def rej(2) st_cMn rej(1)
          tcrr_1 ar_reject_stage_def)
    show ar_posk_consistent M cMn c''
      by (simp add: ar_posk_consistent_def rej(2) ar_reject_stage_def)
    show ar_at_read_boundary M c''

```

```

      by (simp add: ar_at_read_boundary_def rej(2) ar_reject_stage_def
              pad_c'')
qed
qed
qed
qed
qed

```

3.19 Chunked reverse engine

The reverse-arm loop invariant: the substep-walker is in *some* stage of the cycle. Six disjuncts, one per walker predicate. This is the AR analogue of carrying *ae_simulates* across AE's reverse induction, but stage-granular: where AE peels a whole fixed-length cycle per *ae_backward_stage*, AR peels one M' -substep and dispatches on the current substep tag (the cycle length is data-dependent here, which is why the pivot to the substep walker was needed in the first place).

definition *ar_walker* ::

```

('q, 'a) mttm
⇒ ('a, 'q) mt_config
⇒ (sym4, 'q × 'a ar_stage) mt_config ⇒ bool where
ar_walker M cM c' ⇔
  ar_walker_at_boundary M cM c'
  ∨ ar_walker_in_read M cM c'
  ∨ ar_walker_at_compute M cM c'
  ∨ ar_walker_in_write M cM c'
  ∨ ar_walker_in_advance M cM c'
  ∨ ar_walker_at_next M cM c'

```

Chunked-induction engine for the reverse arm, the AR analogue of AE's *ae_simulation_phase_chunked_reverse*. Given a finite M' -trace (c', c_acc) of length m ending in the canonical accept config $(t_tm\ M, ar_accept_stage\ (bl_tm\ M))$ and the walker invariant at c' relative to a reachable cM , deliver an accepting M -path from cM . Strong induction on m ; at each level dispatch on the walker stage and peel one M' -step:

- the five active stages (*in_read*, *at_compute*, *in_write*, *in_advance*, *at_next*, and an active read boundary) all have $c' \neq c_acc$, so $m = Suc\ m'$; peel the step via the matching preservation lemma (*at_compute* emits one M -step into the *in_write* witness; *at_next* emits one via *ar_walker_cycle_close* and advances cM), then recurse on m' ;
- a boundary at the accept config closes the induction: *ar_simulates_accept_iff* forces $mt_state\ cM = t_tm\ M$;
- a boundary at the reject config is impossible — the trace runs to the accept config, but reject is terminal (*ar_reject_terminal*) and the reject config is not the accept config.

The conclusion is in *rtrancl* form for the downstream *Lang_mttm*-
repacking in *alphabet_reduce_language*.

```

lemma ar_simulation_phase_chunked_reverse:
  fixes  $M :: ('q, 'a) \text{ mttm}$ 
    and  $w :: 'a \text{ list}$ 
    and  $cM :: ('a, 'q) \text{ mt\_config}$ 
    and  $c' \ c\_acc :: (\text{sym4}, 'q \times 'a \text{ ar\_stage}) \text{ mt\_config}$ 
    and  $m :: \text{nat}$ 
  assumes  $vM: \text{ valid\_mttm } M$ 
    and  $w\_sub: \text{ set } w \subseteq \text{ Sigma\_tm } M$ 
    and  $\text{card\_ge}: \text{ card } (\Gamma\_tm \ M) \geq 4$ 
    and  $\text{lebl}: \text{ le\_tm } M \neq \text{ bl\_tm } M$ 
    and  $\text{reach\_M}: (\text{init\_config\_mttm } M \ w, cM) \in (\text{mttm\_step } (\text{delta\_tm } M))^*$ 
    and  $\text{trace}: (c', c\_acc) \in (\text{mttm\_step } (\text{alphabet\_reduce\_delta } M))^{\wedge m}$ 
    and  $\text{accept}: \text{ mt\_state } c\_acc = (\text{t\_tm } M, \text{ ar\_accept\_stage } (\text{bl\_tm } M))$ 
    and  $\text{walk}: \text{ ar\_walker } M \ cM \ c'$ 
  shows  $\exists cM\_final. (cM, cM\_final) \in (\text{mttm\_step } (\text{delta\_tm } M))^*$ 
     $\wedge \text{ mt\_state } cM\_final = \text{t\_tm } M$ 
  using  $\text{reach\_M trace walk}$ 
proof (induction m arbitrary: cM c' rule: less_induct)
  case (less m)
    let  $?R = \text{mttm\_step } (\text{alphabet\_reduce\_delta } M)$ 
    — Encoding width  $\geq 2$  from  $\text{card } \Gamma\_M \geq 4$ , and the reachable-config invariants
     $cM$  needs for the substep lemmas.
    have  $\text{kge2}: 2 \leq \text{block\_width } (\Gamma\_tm \ M)$ 
    proof —
      have  $(2::\text{nat}) \wedge 2 \leq 2 \wedge \text{block\_width } (\Gamma\_tm \ M)$ 
      using  $\text{card\_ge card\_le\_two\_pow\_block\_width}[of \ \Gamma\_tm \ M]$  by simp
      thus  $2 \leq \text{block\_width } (\Gamma\_tm \ M)$ 
      using  $\text{power\_le\_imp\_le\_exp}[of \ 2::\text{nat} \ 2 \ \text{block\_width } (\Gamma\_tm \ M)]$  by simp
    qed
    have  $\text{valcM}: \text{ valid\_config\_mttm } M \ cM$ 
    by (rule valid_reach_mttm[OF vM w_sub less.prems(1)])
    have  $qQ: \text{ mt\_state } cM \in Q\_tm \ M$  using  $\text{valcM}$  by (cases M; cases cM) auto
    have  $\text{tapeG}: \text{ mt\_tape } cM \ k \ (\text{mt\_pos } cM \ k) \in \Gamma\_tm \ M$  for  $k$ 
    proof —
      have  $\text{range } (\text{mt\_tape } cM \ k) \subseteq \Gamma\_tm \ M$ 
      using  $\text{valcM}$  by (cases M; cases cM) auto
      thus ?thesis by blast
    qed
    have  $\text{acc\_idx}: \text{fst } (\text{snd } (\text{mt\_state } c\_acc)) = \text{AR\_HaltAccept}$ 
    using accept by (simp add: ar_accept_stage_def)
    — An active config (substep tag  $\neq \text{AR\_HaltAccept}$ ) cannot be  $c\_acc$ , so the trace
    has a first step to peel.
    have peel:  $\exists m' \ c'_1. m = \text{Suc } m'$ 
       $\wedge (c', c'_1) \in ?R \wedge (c'_1, c\_acc) \in ?R^{\wedge m'}$ 
    if notHA:  $\text{fst } (\text{snd } (\text{mt\_state } c')) \neq \text{AR\_HaltAccept}$ 
    proof —

```

```

have m ≠ 0
proof
  assume m = 0
  hence c' = c_acc using less.premis(2) by simp
  thus False using notHA acc_idx by simp
qed
then obtain m' where mSuc: m = Suc m' using not0_implies_Suc by blast
have (c', c_acc) ∈ ?R ^^ Suc m' using less.premis(2) mSuc by simp
from relpow_Suc_D2[OF this] obtain c'_1 where
  (c', c'_1) ∈ ?R and (c'_1, c_acc) ∈ ?R ^^ m' by blast
thus ?thesis using mSuc by blast
qed
from less.premis(3)
consider (bnd) ar_walker_at_boundary M cM c'
  | (rd) ar_walker_in_read M cM c'
  | (cmp) ar_walker_at_compute M cM c'
  | (wr) ar_walker_in_write M cM c'
  | (adv) ar_walker_in_advance M cM c'
  | (nx) ar_walker_at_next M cM c'
  unfolding ar_walker_def by blast
then show ?case
proof cases
case bnd
  have sim: ar_simulates M cM c'
  using bnd unfolding ar_walker_at_boundary_def by simp
  obtain qM0 stg where st: mt_state c' = (qM0, stg)
  by (cases mt_state c') auto
  obtain idx tk i buf dvec posk where
    sg: stg = (idx, tk, i, buf, dvec, posk) by (cases stg) auto
  have state_disj:
    (idx = AR_SimRead ∧ qM0 ∉ {t_tm M, r_tm M})
    ∨ (qM0 = t_tm M ∧ stg = ar_accept_stage (bl_tm M))
    ∨ (qM0 = r_tm M ∧ stg = ar_reject_stage (bl_tm M))
  using sim st sg unfolding ar_simulates_def by (auto split: prod.splits)
  consider (active) idx = AR_SimRead
    | (acc) stg = ar_accept_stage (bl_tm M) qM0 = t_tm M
    | (rej) stg = ar_reject_stage (bl_tm M) qM0 = r_tm M
  using state_disj by blast
  thus ?thesis
proof cases
case active
  have neq: fst (snd (mt_state c')) ≠ AR_HaltAccept
  using st sg active by simp
  obtain m' c'_1 where mSuc: m = Suc m'
  and fst_step: (c', c'_1) ∈ ?R
  and rest: (c'_1, c_acc) ∈ ?R ^^ m'
  using peel[OF neq] by blast
  have m'_lt: m' < m using mSuc by simp
  have ar_walker_in_read M cM c'_1 ∨ ar_walker_at_compute M cM c'_1

```



```

    using ar_walker_step_from_boundary[OF bnd fst_step] .
  hence walk_1: ar_walker M cM c'_1 unfolding ar_walker_def by blast
  show ?thesis using less.IH[OF m'_lt less.prem(1) rest walk_1] .
next
case acc
have c'_acc: mt_state c' = (t_tm M, ar_accept_stage (bl_tm M))
  using st_sg acc by simp
have cM_t: mt_state cM = t_tm M
  using ar_simulates_accept_iff[OF vM sim] c'_acc by simp
have (cM, cM) ∈ (mttm_step (delta_tm M))* by simp
thus ?thesis using cM_t by blast
next
case rej
have c'_rej: mt_state c' = (r_tm M, ar_reject_stage (bl_tm M))
  using st_sg rej by simp
show ?thesis
proof (cases m)
case 0
have ceq: c' = c_acc using less.prem(2) 0 by simp
have fst (snd (mt_state c')) = AR_HaltReject
  using c'_rej by (simp add: ar_reject_stage_def)
moreover have fst (snd (mt_state c')) = AR_HaltAccept
  using ceq accept by (simp add: ar_accept_stage_def)
ultimately show ?thesis by simp
next
case (Suc m')
have (c', c_acc) ∈ ?R ^^ Suc m' using less.prem(2) Suc by simp
from relpow_Suc_D2[OF this] obtain c'_1 where
  fst_step: (c', c'_1) ∈ ?R by blast
have mt_state c' ≠ (r_tm M, ar_reject_stage (bl_tm M))
  by (rule ar_reject_terminal[OF vM card_ge fst_step])
thus ?thesis using c'_rej by simp
qed
qed
next
case rd
have neq: fst (snd (mt_state c')) ≠ AR_HaltAccept
  using rd unfolding ar_walker_in_read_def by simp
obtain m' c'_1 where mSuc: m = Suc m'
  and fst_step: (c', c'_1) ∈ ?R
  and rest: (c'_1, c_acc) ∈ ?R ^^ m'
  using peel[OF neq] by blast
have m'_lt: m' < m using mSuc by simp
have ar_walker_in_read M cM c'_1 ∨ ar_walker_at_compute M cM c'_1
  using ar_walker_step_from_in_read[OF rd fst_step] .
hence walk_1: ar_walker M cM c'_1 unfolding ar_walker_def by blast
show ?thesis using less.IH[OF m'_lt less.prem(1) rest walk_1] .
next
case cmp

```

```

have neq: fst (snd (mt_state c')) ≠ AR_HaltAccept
  using cmp unfolding ar_walker_at_compute_def by simp
obtain m' c'_1 where mSuc: m = Suc m'
  and fst_step: (c', c'_1) ∈ ?R
  and rest: (c'_1, c_acc) ∈ ?R ^^ m'
  using peel[OF neq] by blast
have m'_lt: m' < m using mSuc by simp
have w_1: ar_walker_in_write M cM c'_1
  using ar_walker_step_from_at_compute[OF cmp fst_step vM qQ kge2
tapeG] .
  have walk_1: ar_walker M cM c'_1 using w_1 unfolding ar_walker_def
by blast
  show ?thesis using less.IH[OF m'_lt less.prem1 rest walk_1] .
next
case wr
have neq: fst (snd (mt_state c')) ≠ AR_HaltAccept
  using wr unfolding ar_walker_in_write_def by simp
obtain m' c'_1 where mSuc: m = Suc m'
  and fst_step: (c', c'_1) ∈ ?R
  and rest: (c'_1, c_acc) ∈ ?R ^^ m'
  using peel[OF neq] by blast
have m'_lt: m' < m using mSuc by simp
have ar_walker_in_write M cM c'_1 ∨ ar_walker_in_advance M cM c'_1
  using ar_walker_step_from_in_write[OF wr fst_step] .
hence walk_1: ar_walker M cM c'_1 unfolding ar_walker_def by blast
show ?thesis using less.IH[OF m'_lt less.prem1 rest walk_1] .
next
case adv
have neq: fst (snd (mt_state c')) ≠ AR_HaltAccept
  using adv unfolding ar_walker_in_advance_def by simp
obtain m' c'_1 where mSuc: m = Suc m'
  and fst_step: (c', c'_1) ∈ ?R
  and rest: (c'_1, c_acc) ∈ ?R ^^ m'
  using peel[OF neq] by blast
have m'_lt: m' < m using mSuc by simp
have ar_walker_in_advance M cM c'_1 ∨ ar_walker_at_next M cM c'_1
  using ar_walker_step_from_in_advance[OF adv fst_step] .
hence walk_1: ar_walker M cM c'_1 unfolding ar_walker_def by blast
show ?thesis using less.IH[OF m'_lt less.prem1 rest walk_1] .
next
case nx
have neq: fst (snd (mt_state c')) ≠ AR_HaltAccept
  using nx unfolding ar_walker_at_next_def by simp
obtain m' c'_1 where mSuc: m = Suc m'
  and fst_step: (c', c'_1) ∈ ?R
  and rest: (c'_1, c_acc) ∈ ?R ^^ m'
  using peel[OF neq] by blast
have m'_lt: m' < m using mSuc by simp
obtain cMn where step_cMn: (cM, cMn) ∈ mttm_step (delta_tm M)

```

```

and bnd_1: ar_walker_at_boundary M cMn c'_1
using ar_walker_cycle_close[OF nx fst_step vM qQ kge2 tapeG
                                w_sub less.premis(1) lebl]

by blast
have reach_Mn: (init_config_mttm M w, cMn) ∈ (mttm_step (delta_tm M))*
using less.premis(1) step_cMn by (rule rtrancl.rtrancl_into_rtrancl)
have walk_1: ar_walker M cMn c'_1
using bnd_1 unfolding ar_walker_def by blast
obtain cM_final where
  run_final: (cMn, cM_final) ∈ (mttm_step (delta_tm M))*
  and acc_final: mt_state cM_final = t_tm M
using less.IH[OF m'_lt reach_Mn rest walk_1] by blast
have (cM, cM_final) ∈ (mttm_step (delta_tm M))*
using step_cMn run_final by (meson r_into_rtrancl rtrancl_trans)
thus ?thesis using acc_final by blast
qed
qed

```

Classical language-equivalence (biconditional) for alphabet reduction: an $'a$ -word w over the input alphabet is in M 's language iff its *sym4*-encoding is in the language of the reduced machine M' . Pairs the forward inclusion *alphabet_reduce_language_forward* (in *AlphabetReduction_Theorems.thy*) with the reverse leg supplied by *ar_simulation_phase_chunked_reverse*.

Unlike AE, the alphabet-reduction construction has no input validation prefix (the no-validation design point), so the reverse direction needs neither a validation-peel nor a determinism / chain-uniqueness argument: the encoded input's accepting M' -run is handed to the engine directly off the initial read-boundary correspondence (*ar_walker_at_boundary*, the conjunction of the three *_init* invariants). No *det_mttm* hypothesis is required.

The *set w ⊆ Sigma_tm M* guard is required (and matches AE's statement): *Lang_mttm* bakes in the input-alphabet restriction, so $w ∈ \text{Lang_mttm } M$ supplies the guard for free in the forward direction, but membership of the encoded word in *Lang_mttm M'* says nothing about w 's alphabet. For w outside the input alphabet the encoded word can still be accepted by M' while $w ∉ \text{Lang_mttm } M$.

```

theorem alphabet_reduce_language:
  fixes M :: ('q, 'a) mttm
  assumes vM:      valid_mttm M
    and s_neq_t:   s_tm M ≠ t_tm M
    and s_neq_r:   s_tm M ≠ r_tm M
    and le_neq_bl: le_tm M ≠ bl_tm M
    and card_ge:   card (Γ_tm M) ≥ 4
  shows ∀ w. set w ⊆ Sigma_tm M
    → (encode_input_ar (Γ_tm M) (bl_tm M) w ∈ Lang_mttm
        (alphabet_reduce M
          :: ('q × 'a ar_stage, sym4) mttm))
    = (w ∈ Lang_mttm M)

```

proof –

let $?M' = \text{alphabet_reduce } M :: ('q \times 'a \text{ ar_stage}, \text{sym4}) \text{ mttm}$

have $\text{fwd}: \forall w. \text{set } w \subseteq \text{Sigma_tm } M$

$\longrightarrow w \in \text{Lang_mttm } M$

$\longrightarrow \text{encode_input_ar } (\Gamma_tm M) (bl_tm M) w \in \text{Lang_mttm } ?M'$

by (rule alphabet_reduce_language_forward[OF vM s_neq_t s_neq_r le_neq_bl card_ge])

show $?thesis$

proof (intro allI impI)

fix $w :: 'a \text{ list}$

assume $w_sub: \text{set } w \subseteq \text{Sigma_tm } M$

let $?enc = \text{encode_input_ar } (\Gamma_tm M) (bl_tm M) w$

let $?c' = \text{init_config_mttm } ?M' ?enc$

show $(?enc \in \text{Lang_mttm } ?M') = (w \in \text{Lang_mttm } M)$

proof

assume $w_in_M: w \in \text{Lang_mttm } M$

from fwd w_sub w_in_M **show** $?enc \in \text{Lang_mttm } ?M'$ **by** blast

next

— Reverse direction: an accepting M' -run on the encoded input yields an accepting M -run on w . No validation prefix to peel, so the accepting trace feeds the chunked-reverse engine directly off the initial read-boundary correspondence.

assume $\text{enc_in_}M': ?enc \in \text{Lang_mttm } ?M'$

— Step 1: unpack the M' -acceptance into a relpow trace.

from $\text{enc_in_}M'$ **obtain** $wM' nM'$ **where**

$\text{acc_path_}M':$

$(?c', \text{Config}_M (t_tm ?M') wM' nM')$

$\in (\text{mttm_step } (\text{delta_tm } ?M'))^*$

unfolding Lang_mttm_def **by** blast

have $\text{acc_path}:$

$(?c', \text{Config}_M (t_tm ?M') wM' nM')$

$\in (\text{mttm_step } (\text{alphabet_reduce_delta } M))^*$

using $\text{acc_path_}M'$ **by** simp

obtain m **where** $\text{trace}:$

$(?c', \text{Config}_M (t_tm ?M') wM' nM')$

$\in (\text{mttm_step } (\text{alphabet_reduce_delta } M)) \wedge^m m$

using acc_path rtranc_imp_relpow **by** metis

— Step 2: the accept state in the engine's canonical form.

have $\text{accept}:$

$\text{mt_state } (\text{Config}_M (t_tm ?M') wM' nM')$

$= (t_tm M, \text{ar_accept_stage } (bl_tm M))$

by simp

— Step 3: the initial read-boundary correspondence is the walker's boundary disjunct.

have $\text{sim}: \text{ar_simulates } M (\text{init_config_mttm } M w) ?c'$

by (rule ar_simulates_init[OF vM w_sub s_neq_t s_neq_r])

have $\text{posk}: \text{ar_posk_consistent } M (\text{init_config_mttm } M w) ?c'$

by (rule ar_posk_consistent_init)

have $\text{rbnd}: \text{ar_at_read_boundary } M ?c'$

by (rule ar_at_read_boundary_init[OF vM])

```

have walk: ar_walker M (init_config_mttm M w) ?c'
  unfolding ar_walker_def ar_walker_at_boundary_def
  using sim_posk_rbnd by blast
— Step 4: seed with the reflexive reach and run the engine.
have reach_refl: (init_config_mttm M w, init_config_mttm M w)
  ∈ (mttm_step (delta_tm M))* by simp
obtain cM_final where
  M_path: (init_config_mttm M w, cM_final)
    ∈ (mttm_step (delta_tm M))*
  and M_acc: mt_state cM_final = t_tm M
  using ar_simulation_phase_chunked_reverse
    [OF vM w_sub card_ge le_neq_bl reach_refl trace_accept walk]
  by blast
— Step 5: repack as  $w \in \text{Lang\_mttm } M$ .
obtain wM_acc nM_acc where
  cM_final_eq: cM_final = Config_M (t_tm M) wM_acc nM_acc
  using M_acc by (cases cM_final) simp
show w ∈ Lang_mttm M
  unfolding Lang_mttm_def
  using w_sub M_path cM_final_eq by blast
qed
qed
qed
end

```

References

- [1] F. J. Balbach. The Cook-Levin theorem. *Archive of Formal Proofs*, January 2023. https://isa-afp.org/entries/Cook_Levin.html, Formal proof development.
- [2] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, volume 11 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
- [3] C. Dalvit and R. Thiemann. A verified translation of multitape Turing machines into singletape Turing machines. *Archive of Formal Proofs*, November 2022. https://isa-afp.org/entries/Multitape_To_Singletape_TM.html, Formal proof development.
- [4] P. C. Fischer. Multi-tape and infinite-state automata—a survey. *Commun. ACM*, 8(12):799–805, Dec. 1965.
- [5] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.
- [6] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

- [7] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier, 1990.
- [8] J. Xu, X. Zhang, C. Urban, S. J. C. Joosten, and F. Regensburger. Universal Turing machine. *Archive of Formal Proofs*, February 2019. https://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.