Policy-Centric Processor: Architecture Specification

Alexey A. Nekludoff

AstraVerge Research

E-mail: an@astraverge.org
ORCID: 0009-0002-7724-5762

Version 0.1 04 January 2026 Draft

1 Scope

This document specifies the *Policy-Centric Processor* (PCP) architecture: a processor model in which *memory access is governed by explicit policy* rather than treated as an unqualified consequence of addressing. The specification covers:

- the architectural primitives required to make *policy* a first-class constraint on memory access (identities, rules, enforcement points, and observability constraints);
- a non-speculative execution model in which control-flow prediction is permitted only insofar as it does not trigger memory-side effects;
- a massively multithreaded core model (fine-grain multithreading) as the primary mechanism for latency tolerance instead of speculative memory access;
- the minimum interfaces between the compute core and the policy enforcement subsystem (the *Memory Protector*) necessary for deterministic, enforceable access governance;
- the compatibility boundary between processor architecture and system software: what must be guaranteed by the architecture versus what is delegated to the operating system/runtime.

The PCP specification is intentionally written as an *architecture specification* rather than a performance paper. Where performance is discussed, it is treated as a derived consequence of the model (e.g., predictability, variance control, and throughput under concurrency) rather than as the primary objective.

2 Goals

The PCP architecture is designed to establish a *new ontological basis for computer systems* by replacing several implicit assumptions of classical computer architecture with explicit, enforceable primitives.

2.1 Ontological goal: policy as an architectural primitive

Promote *policy-governed access* to the same status as arithmetic and control flow. Concretely:

1. Memory access must be representable as an object of reasoning ("who accesses what under which constraints"), not merely as an address computation.
2. Enforcement must be *pre-access* (preventive), not only *post-access* (fault-driven).
3. The enforcement mechanism must avoid producing memory-visible side effects for denied or non-authorized access attempts.

2.2 Architectural goal: eliminate speculative memory side effects

Remove speculative memory access as an architectural technique. The processor may employ internal optimizations that do not create memory-observable effects; however, memory traffic must be initiated only by architecturally committed execution.

2.3 Execution goal: latency tolerance via massive multithreading

Adopt fine-grain multithreading as a default execution strategy:

1. Hardware provides a large number of independent architectural contexts per core (target: 64).
2. Stalls (branches, cache misses, memory latency) are tolerated by switching to runnable contexts rather than by speculative execution.

2.4 Systems goal: shift complexity from hidden heuristics to explicit governance

Reduce reliance on hidden heuristics whose correctness cannot be stated (and therefore cannot be verified). Replace them with explicit mechanisms that admit formal reasoning about:

- isolation and revocation,
- compositional safety under concurrency,
- predictability (bounded variance) under load.

2.5 Pragmatic goal: enable implementable reference designs

Define the smallest set of mechanisms sufficient for a working PCP implementation (silicon, FPGA, or emulator), so that the architecture can be tested, measured, and iterated without requiring an entirely new software ecosystem as a precondition.

3 Architectural Axioms

This section states the foundational axioms of the Policy-Centric Processor (PCP) architecture. These axioms are *normative*: all compliant implementations MUST satisfy them. They are not design preferences, performance heuristics, or implementation guidelines. They define the ontological commitments of the architecture.

3.1 Axiom 1: Computation is local; memory is global

Computation is defined as the transformation of values within a bounded, local context (registers, pipeline state, and architecturally local execution units). Memory is defined as a shared, globally observable resource whose effects extend beyond any single execution context.

Consequently:

- Local computational actions may be freely optimized, reordered, or discarded, provided they do not produce externally observable effects.
- Memory actions are globally significant and MUST be treated as boundary-crossing events.

3.2 Axiom 2: Memory access is a governed action

A memory access is not an unconditional consequence of address computation. It is a *governed action* that is subject to explicit authorization.

Formally:

- An address alone is insufficient to justify a memory access.

- Authorization depends on execution context, identity, and active policy.
- Absence of authorization MUST result in non-access, not merely in post-facto fault handling.

3.3 Axiom 3: Policy precedes access

All memory accesses MUST be validated against policy *before* any memory-side effect occurs.

The architecture MUST ensure that:

1. No unauthorized access produces observable changes in memory state.
2. No denied access produces timing-, cache-, or coherence-visible artifacts.
3. Enforcement is preventive rather than reactive.

This axiom explicitly rejects architectures in which correctness relies on detecting and repairing violations after memory interaction has already taken place.

3.4 Axiom 4: No speculative effects beyond the compute domain

Speculation is permitted only within the compute domain. Speculative actions MUST NOT cross into the memory domain.

Concretely:

- Control-flow speculation MAY occur if its effects are fully discardable and local.
- Memory reads, writes, or cache state changes MUST NOT be initiated speculatively.
- Any execution that reaches the memory boundary MUST be architecturally committed.

3.5 Axiom 5: Latency is not an architectural objective

The architecture does not treat latency minimization as a primary design goal. Instead, it prioritizes:

- bounded variance,
- predictable progress,
- compositional reasoning under concurrency.

Latency tolerance is achieved through parallelism and concurrency, not through speculative prediction of future behavior.

3.6 Axiom 6: Parallelism supersedes prediction

The architecture assumes that progress under delay is achieved by switching among independent execution contexts rather than by predicting future control or memory behavior.

Therefore:

- Fine-grain multithreading is a first-class architectural mechanism.
- Stalled execution contexts MUST NOT impede the progress of runnable contexts.
- Pipeline disruption due to branches or memory latency MUST be mitigated by context switching, not by speculative execution.

3.7 Axiom 7: Explicit mechanisms over implicit heuristics

The architecture rejects hidden heuristics whose correctness cannot be stated in architectural terms.

All mechanisms that affect:

- isolation,
- access ordering,
- visibility of state,

MUST be explicitly representable in the architecture and subject to reasoning, verification, and inspection.

3.8 Axiom 8: Architecture defines responsibility boundaries

The processor architecture is responsible for:

- enforcing access policy at the memory boundary,
- preserving non-observability of denied or invalid accesses,
- providing execution contexts with well-defined guarantees.

System software is responsible for:

- defining policy,
- assigning identities and contexts,
- structuring execution to exploit available parallelism.

This separation is intentional and normative: correctness MUST NOT depend on undefined cooperation between layers.

3.9 Summary

These axioms collectively define a processor architecture in which:

- access is governed rather than assumed,
- speculation is confined rather than global,
- progress arises from concurrency rather than prediction.

All subsequent sections refine, operationalize, and implement these axioms. Any design that violates them constitutes a different architecture.

4 Abandoned Assumptions

This section enumerates architectural assumptions deliberately rejected by the Policy-Centric Processor (PCP). These assumptions are historically common and often treated as axiomatic in classical processor design. Their rejection is not a matter of preference but a necessary consequence of the axioms stated in Section 1.

Each abandoned assumption is listed together with a brief explanation of why it is incompatible with a policy-centric architecture.

4.1 Assumption 1: Address implies authority

Classical architectures assume that possession of an address is sufficient to justify access.

This assumption is rejected.

An address is a locator, not a capability. Equating addressability with authority collapses identification, intent, and permission into a single arithmetic operation, making access control implicit, brittle, and non-compositional.

In the PCP architecture:

- An address identifies a location.
- Authority to access that location is determined separately by policy.

4.2 Assumption 2: Page-based memory is a fundamental abstraction

Page-based memory systems treat fixed-size pages as a natural unit of protection, translation, and accounting.

This assumption is rejected.

Fixed-size pages are an implementation artifact driven by historical hardware constraints, not an ontological property of memory. In practice, page size is rigid, globally imposed, and poorly aligned with application-level access patterns.

The PCP architecture does not assume:

- a globally fixed protection granularity,
- page-aligned authority boundaries,
- page faults as a correctness mechanism.

4.3 Assumption 3: Faults are an acceptable enforcement mechanism

Traditional systems rely on faults and exceptions to detect unauthorized or invalid memory accesses.

This assumption is rejected.

Fault-based enforcement is inherently reactive: the system learns that an access was invalid only *after* the access attempt has occurred. Even when the architectural state is rolled back, microarchitectural and timing effects may remain observable.

In the PCP architecture:

- Unauthorized access MUST result in non-access.
- Correctness MUST NOT depend on post-access recovery.

4.4 Assumption 4: Speculative memory access is benign

Modern processors assume that speculative memory access is safe provided that architectural state is eventually corrected.

This assumption is rejected.

Speculative memory access creates observable side effects (cache state, coherence traffic, timing variation) even when results are discarded. Such effects violate the requirement that denied or invalid accesses be non-observable.

The PCP architecture therefore prohibits:

- speculative loads,
- speculative stores,
- speculative cache fills.

4.5 Assumption 5: Latency minimization is the primary objective

Classical design treats latency reduction as a first-order architectural goal, often justifying complex heuristics and speculation.

This assumption is rejected.

Latency is a contingent property of memory technology and system load. Optimizing for minimal latency encourages hidden mechanisms whose behavior cannot be stated or verified.

The PCP architecture prioritizes:

- bounded variance,
- forward progress,
- predictable behavior under contention.

4.6 Assumption 6: Single-thread performance is a universal metric

Many architectures implicitly assume that improving single-thread performance benefits all workloads.

This assumption is rejected.

Single-thread performance reflects a narrow execution model that does not generalize to highly concurrent systems. Optimizing for it encourages speculation and serialization that conflict with policy-governed access.

The PCP architecture treats massive concurrency as the default execution mode.

4.7 Assumption 7: Mutexes and blocking primitives are fundamental

Classical systems assume mutual exclusion and blocking synchronization as necessary tools for correctness.

This assumption is rejected.

Mutexes enforce serialization by construction and prevent scalable exploitation of available parallelism. They encode contention rather than resolving it.

The PCP architecture assumes synchronization models based on:

- ownership,
- message passing,
- epoch- and phase-based coordination.

4.8 Assumption 8: Compatibility outweighs architectural clarity

It is often assumed that backward compatibility with existing software models is an overriding constraint on architecture.

This assumption is rejected.

Compatibility is a deployment concern, not an ontological one. An architecture that preserves flawed assumptions in order to maintain compatibility perpetuates those flaws.

The PCP architecture defines correctness and clarity first; compatibility is addressed, where necessary, by system software.

4.9 Summary

By abandoning these assumptions, the PCP architecture deliberately reduces implicit behavior and replaces it with explicit, enforceable structure. What is removed is not functionality but ambiguity.

Subsequent sections describe the concrete execution and memory models that replace these assumptions.

5 Memory Sharding and Independent Service

This section defines the normative memory service model of the Policy-Centric Processor (PCP). The model is based on symmetric memory sharding with independent service engines. All compliant implementations MUST satisfy the requirements stated below.

5.1 Memory Shards

A *Memory Shard* is defined as an independently serviced memory bank with a dedicated memory service engine (DMA).

- Each Memory Shard MUST provide its own request queue.
- Each Memory Shard MUST be serviced independently of all other shards.
- No Memory Shard MUST depend on a global arbitration queue for request acceptance.

The architecture MUST support one or more Memory Shards. The exact number of Memory Shards is not fixed architecturally and MUST be determined by the physical configuration of the system.

5.2 Symmetry Requirements

All Memory Shards MUST be architecturally symmetric.

Specifically:

- All shards MUST expose equivalent access semantics.
- All shards MUST belong to the same architectural latency class.
- No shard MAY be designated as local, remote, primary, or secondary.

The architecture MUST NOT expose NUMA-style locality distinctions to software.

5.3 Address Space and Shard Mapping

The architecture MUST present a single, unified address space.

Address-to-shard mapping MUST satisfy the following properties:

- Mapping MUST be deterministic.
- Mapping MUST distribute requests across shards in a balanced manner.
- Mapping MUST NOT depend on dynamic runtime heuristics.

Acceptable mapping strategies include, but are not limited to:

- round-robin distribution,
- hash-based distribution,
- fixed interleaving by address bits.

The specific mapping function MAY be implementation-defined but MUST be architecturally stable and observable.

5.4 Independent Service Engines

Each Memory Shard MUST be associated with an independent memory service engine.

- Each service engine MUST accept requests without coordination with other engines.
- Each service engine MUST maintain its own in-flight request state.
- Backpressure in one shard MUST NOT block request acceptance in other shards.

The architecture MUST NOT rely on a single centralized memory controller as the sole point of arbitration.

5.5 Latency Mitigation by Parallel Service

The architecture MUST NOT rely on speculative execution, caching heuristics, or prefetch prediction to mitigate memory latency.

Instead:

- Memory latency MUST be mitigated through parallel servicing across shards.
- Effective service time MUST scale with the number of available shards.
- Contention MUST be localized to individual shards.

From the architectural perspective, memory latency MUST be treated as a service property rather than a scheduling problem.

5.6 Bandwidth Provisioning

The architecture MUST define a per-core memory bandwidth budget.

- Each core MUST be able to issue concurrent requests to multiple shards.
- Aggregate memory bandwidth available to a core MUST scale with shard count.
- No single shard MUST be required to saturate core demand.

The architecture MUST NOT assume a fixed bus width between compute and memory. Bandwidth MUST emerge from shard parallelism rather than monolithic bus widening.

5.7 Interaction with Policy Enforcement

All memory requests MUST pass through policy enforcement before being routed to any Memory Shard.

- Policy enforcement MUST be completed prior to shard selection.
- Shard routing MUST NOT bypass policy checks.
- Denied requests MUST NOT reach any shard service engine.

Policy enforcement MUST be independent of shard count and shard topology.

5.8 Summary

The PCP memory architecture is defined by:

- symmetric memory sharding,
- independent service engines,
- deterministic request distribution,
- latency mitigation through parallel service.

Any implementation that relies on centralized arbitration, NUMA-style asymmetry, or speculative memory access violates this specification.

6 Execution Model

This section defines the normative execution model of the Policy-Centric Processor (PCP). All compliant implementations MUST adhere to the requirements stated below.

6.1 Execution Units

An *Execution Unit* is defined as a hardware core capable of executing a single instruction stream.

- Each Execution Unit MUST execute exactly one architectural thread.
- An Execution Unit MUST NOT multiplex multiple architectural threads.
- An architectural thread MUST NOT migrate between Execution Units during execution.

The architecture explicitly rejects simultaneous multithreading (SMT) and any form of implicit hardware thread interleaving.

6.2 Single-Thread Semantics

Each Execution Unit MUST provide strict single-thread execution semantics.

- Instructions MUST be executed in program order.
- No instruction MUST be reordered across architecturally visible boundaries.
- No instruction MUST be executed speculatively beyond its commit point.

The architectural state of a thread MUST evolve monotonically and deterministically.

6.3 Blocking Behavior

The execution model MUST permit blocking on memory operations.

- A memory access MAY stall the issuing Execution Unit.
- Stalling an Execution Unit MUST NOT affect the execution of other units.

- Blocking MUST be local to the issuing Execution Unit.

The architecture MUST NOT introduce speculative execution, helper threads, or implicit instruction replay to avoid blocking.

6.4 Latency Tolerance

The architecture MUST tolerate memory latency without relying on speculation or thread multiplexing within an Execution Unit.

- Latency tolerance MUST be achieved through independent execution units.
- Latency tolerance MUST be achieved through memory sharding and parallel service.
- No Execution Unit MUST depend on another unit for forward progress.

Latency tolerance is a system-level property and MUST NOT be implemented through microarchitectural prediction.

6.5 Progress Guarantees

Each Execution Unit MUST make forward progress whenever:

- instructions are available,
- required memory accesses are authorized,
- memory service engines accept requests.

The architecture MUST NOT permit starvation caused by:

- centralized scheduling,
- shared pipeline arbitration,
- speculative rollback mechanisms.

6.6 Instruction Simplicity

The instruction set architecture MUST be designed for simplicity of execution.

- Instructions MUST complete in a bounded number of cycles.
- Instructions MUST NOT encode implicit memory prefetch or speculation.
- Instructions MUST NOT trigger hidden microarchitectural state changes.

Complexity MUST be expressed explicitly in software, not inferred by hardware.

6.7 Concurrency Model

Concurrency MUST be expressed by parallel execution across Execution Units.

- Parallelism MUST be explicit.
- The architecture MUST NOT infer concurrency from instruction streams.
- Synchronization MUST occur through explicit mechanisms defined by software.

The architecture MUST NOT provide implicit lock elision, transactional memory, or speculative synchronization.

6.8 Interaction with Policy Enforcement

Execution Units MUST be subject to policy enforcement at all memory boundaries.

- An Execution Unit MUST NOT bypass policy checks.
- Denied memory accesses MUST stall or terminate execution deterministically.
- Policy enforcement MUST NOT modify execution order.

Execution correctness MUST NOT depend on speculative rollback or fault recovery.

6.9 Summary

The PCP execution model is defined by:

- one Execution Unit per architectural thread,
- in-order, non-speculative execution,
- explicit concurrency across units,
- latency tolerance through memory sharding.

Any implementation that relies on SMT, speculative execution, or implicit concurrency violates this specification.

7 Addressing Model

This section defines the addressing model of the Policy-Centric Processor (PCP). The addressing model is intentionally minimal and non-interpretive. All compliant implementations MUST satisfy the requirements stated below.

7.1 Address Semantics

An address MUST be treated as a pure numerical locator.

- An address MUST identify a location in the unified address space.
- An address MUST NOT encode authority, ownership, or permissions.
- An address MUST NOT imply validity or accessibility.

The architecture explicitly rejects any semantic overloading of addresses.

7.2 Unified Address Space

The architecture MUST expose a single, unified address space.

- All memory shards MUST participate in the same address space.
- Addressing MUST NOT expose shard boundaries to software.
- Software MUST NOT be required to distinguish memory banks by address.

Address space uniformity MUST be preserved independently of physical layout.

7.3 Absence of Address Translation

The architecture MUST NOT perform implicit address translation.

- No page tables MUST exist.
- No address remapping MUST occur during execution.
- No translation lookaside buffers (TLBs) MUST exist.

Address values issued by software MUST be used directly for memory access, subject only to policy enforcement and shard routing.

7.4 Absence of Address-Based Protection

The architecture MUST NOT derive protection semantics from address ranges.

- Access control MUST NOT be implemented via address bounds.
- Segmentation MUST NOT be used for protection.
- Page-based protection MUST NOT exist.

Protection MUST be enforced exclusively by the policy enforcement architecture.

7.5 Deterministic Address Routing

Address routing to Memory Shards MUST be deterministic.

- The same address MUST always map to the same shard under a fixed configuration.
- Routing MUST NOT depend on execution history or dynamic state.
- Routing MUST NOT alter address semantics.

Routing MUST be orthogonal to policy enforcement.

7.6 Address Granularity

The architecture MUST support byte-granular addressing.

- All valid addresses MUST address individual bytes.
- No minimum allocation unit MUST be imposed by the architecture.
- No fixed access granularity MUST be enforced beyond instruction semantics.

The architecture MUST NOT impose page-sized or block-sized access units.

7.7 Address Space Extensibility

The address width MUST be sufficient to represent the maximum supported memory.

- Address width MUST be an architectural parameter.
- Increasing physical memory MUST NOT require changes to address semantics.
- Address exhaustion MUST NOT affect correctness.

The architecture MUST NOT encode versioning or mode information into addresses.

7.8 Software Responsibility

Software MUST treat addresses as opaque numerical values.

- Software MUST NOT infer protection from address values.
- Software MUST NOT rely on address locality for correctness.
- Software MUST coordinate access through policy and explicit structure.

Correctness MUST NOT depend on accidental address properties.

7.9 Interaction with Policy Enforcement

Address computation MUST be independent of policy enforcement.

- Address calculation MUST occur prior to policy evaluation.
- Policy enforcement MUST evaluate access intent, not address form.
- Address routing MUST occur only after policy approval.

Addressing MUST NOT bypass or weaken policy enforcement.

7.10 Summary

The PCP addressing model is defined by:

- addresses as pure locators,
- a unified, untranslated address space,
- absence of address-based protection,

- deterministic routing independent of policy.

Any architecture that embeds protection, translation, or semantic meaning into addresses violates this specification.

8 Concurrency and Synchronization

This section defines the normative concurrency and synchronization model of the Policy-Centric Processor (PCP). Concurrency is an explicit architectural property. Synchronization is an explicit software responsibility. All compliant implementations MUST satisfy the requirements stated below.

8.1 Explicit Concurrency

Concurrency MUST be expressed explicitly by parallel execution across Execution Units.

- The architecture MUST NOT infer concurrency from instruction streams.
- The architecture MUST NOT create implicit parallelism.
- Parallel execution MUST occur only through multiple Execution Units.

A single Execution Unit MUST execute exactly one architectural thread.

8.2 Absence of Implicit Synchronization

The architecture MUST NOT provide implicit synchronization mechanisms.

- The architecture MUST NOT perform implicit locking.
- The architecture MUST NOT perform implicit memory ordering beyond instruction order.
- The architecture MUST NOT infer critical sections.

Synchronization MUST be explicit and visible in software.

8.3 Memory Visibility

Memory visibility MUST be well-defined and deterministic.

- A memory write MUST become visible only after it is completed.
- Visibility MUST NOT depend on speculative execution or rollback.
- Visibility MUST be consistent across all Memory Shards.

The architecture MUST NOT expose transient or speculative memory states.

8.4 Ordering Model

The architecture MUST provide a simple and strict ordering model.

- Memory operations MUST be observed in program order within a single Execution Unit.
- Cross-unit ordering MUST be established explicitly by software.
- The architecture MUST NOT introduce hidden reordering.

No memory operation MUST be reordered across an explicit synchronization point.

8.5 Synchronization Primitives

Synchronization primitives MUST be minimal and explicit.

- The architecture MUST support explicit atomic read-modify-write operations.
- Atomic operations MUST be linearizable.

- Atomic operations MUST be defined independently of cache coherence.

The architecture MUST NOT provide transactional memory, lock elision, or speculative synchronization mechanisms.

8.6 Prohibition of Mutex-Centric Semantics

The architecture MUST NOT assume mutex-based synchronization as a fundamental model.

- Blocking mutexes MUST NOT be required for correctness.
- The architecture MUST NOT optimize for lock-heavy workloads.
- Progress MUST NOT depend on global mutual exclusion.

Serialization MUST be an explicit design choice made by software.

8.7 Non-Blocking Progress

The architecture MUST support non-blocking progress.

- An Execution Unit MUST NOT be forced to wait on unrelated units.
- Failed synchronization MUST NOT stall independent execution.
- Contention MUST be localized to the minimal necessary scope.

The architecture MUST NOT enforce global barriers implicitly.

8.8 Interaction with Policy Enforcement

Synchronization MUST NOT bypass policy enforcement.

- All synchronization-related memory accesses MUST be policy-checked.
- Policy denial MUST override synchronization intent.
- Synchronization MUST NOT grant additional authority.

Correctness MUST NOT depend on synchronization side effects.

8.9 Software Responsibility

Software MUST define concurrency structure explicitly.

- Ownership of shared data MUST be explicit.
- Coordination MUST be expressed through explicit protocols.
- Correctness MUST NOT rely on timing assumptions.

The architecture MUST provide mechanisms, not policies.

8.10 Summary

The PCP concurrency and synchronization model is defined by:

- explicit parallel execution,
- explicit synchronization,
- strict visibility and ordering,
- absence of implicit locking and speculation.

Any architecture that relies on implicit synchronization, speculative ordering, or mutex-centric semantics violates this specification.

9 Memory Consistency Model

This section defines the mandatory memory consistency model of the Policy-Centric Processor (PCP). The model is intentionally simple and deterministic. All compliant implementations MUST satisfy the requirements stated below.

9.1 Single-Unit Consistency

Within a single Execution Unit, memory operations MUST be observed in program order.

- Loads and stores MUST appear in the order issued.
- No reordering MUST occur within an Execution Unit.
- Instruction execution MUST define the sole ordering source.

The architecture MUST NOT introduce intra-thread memory reordering.

9.2 Cross-Unit Visibility

Between Execution Units, memory visibility MUST be explicit.

- A write by one Execution Unit MUST become visible to others only after completion.
- Visibility MUST NOT depend on cache state or speculation.
- No implicit propagation guarantees MUST exist.

Software MUST establish visibility explicitly through synchronization.

9.3 Atomicity Guarantees

Atomic operations MUST be globally linearizable.

- Each atomic operation MUST appear to execute instantaneously.
- Atomicity MUST be preserved across all Memory Shards.
- Atomic operations MUST define total order with respect to each other.

Atomicity MUST NOT depend on cache coherence protocols.

9.4 Absence of Weak Memory Models

The architecture MUST NOT implement weak or relaxed memory models.

- No hidden reordering MUST be permitted.
- No speculative visibility MUST be permitted.
- No architecture-specific fences MUST be required for correctness.

Correctness MUST be derivable from program order and explicit synchronization alone.

9.5 Summary

The PCP memory consistency model is defined by:

- strict per-thread ordering,
- explicit cross-thread visibility,
- global atomic linearizability,
- absence of weak or speculative semantics.

Any architecture that relies on relaxed ordering or speculative visibility violates this specification.

10 System Software Responsibilities

This section defines the mandatory responsibilities of system software operating on a Policy-Centric Processor (PCP). The separation of concerns between hardware and software is normative. All compliant systems MUST satisfy the requirements stated below.

10.1 Policy Definition and Management

System software MUST define and manage all access policies.

- Policy creation MUST be performed by trusted software.
- Policy updates MUST be explicit and synchronized.
- Policy lifetime MUST be well-defined.

The processor MUST NOT infer or modify policy autonomously.

10.2 Identity Assignment

System software MUST assign execution identities.

- Each Execution Unit MUST execute with an explicit identity.
- Identity MUST be bound prior to execution.
- Identity MUST remain stable during execution.

Identity semantics MUST be defined entirely by software.

10.3 Concurrency Structure

System software MUST define the concurrency model.

- Thread creation MUST be explicit.
- Work distribution MUST be explicit.
- Synchronization protocols MUST be explicit.

The processor MUST NOT infer or optimize concurrency structure.

10.4 Memory Allocation

System software MUST manage memory allocation.

- Allocation MUST NOT rely on page boundaries.
- Allocation MUST respect policy constraints.
- Allocation strategy MUST be independent of Memory Shard topology.

The processor MUST NOT impose allocation granularity.

10.5 Scheduling

System software MUST schedule Execution Units.

- Scheduling MUST be explicit.
- Context switching MUST be explicit.
- No hidden preemption MUST occur.

The processor MUST NOT perform implicit scheduling decisions.

10.6 Failure Handling

System software MUST define responses to denied access and execution termination.

- Denial handling MUST be deterministic.
- Recovery MUST be explicit.
- No architectural state MUST be corrupted.

The processor MUST provide mechanisms, not recovery policies.

10.7 Summary

System software is responsible for:

- defining policy,
- assigning identity,
- structuring concurrency,
- managing memory and scheduling.

Any system that shifts these responsibilities into hardware violates this specification.

11 Failure, Denial, and Deterministic Termination

This section defines the mandatory behavior of the Policy-Centric Processor (PCP) under failure and access denial. All compliant implementations MUST satisfy the requirements stated below.

11.1 Denied Memory Access

A denied memory access MUST result in deterministic execution behavior.

- Denial MUST occur prior to memory access.
- Denial MUST NOT modify memory or microarchitectural state.
- Denial MUST be architecturally observable only through defined outcomes.

Denied access MUST NOT produce side effects.

11.2 Execution Response to Denial

Upon denied access, an Execution Unit MUST enter a well-defined state.

- Execution MUST stall or terminate deterministically.
- The response MUST NOT depend on timing or contention.
- The response MUST be externally observable by system software.

The architecture MUST NOT permit undefined continuation.

11.3 Failure Isolation

Failures MUST be isolated to the originating Execution Unit.

- One unit's failure MUST NOT affect others.
- Memory Shards MUST remain consistent.
- Policy state MUST remain intact.

Failure MUST NOT propagate implicitly.

11.4 Absence of Speculative Recovery

The architecture MUST NOT rely on speculative rollback for correctness.

- No speculative state MUST require cleanup.
- No rollback mechanism MUST exist.
- No hidden recovery MUST occur.

Correctness MUST be preserved by construction.

11.5 Deterministic Termination

Execution termination MUST be deterministic.

- Termination MUST leave memory in a consistent state.
- Termination MUST NOT depend on in-flight speculation.
- Termination MUST be observable and attributable.

Termination semantics MUST be simple and explicit.

11.6 Summary

The PCP failure model is defined by:

- preventive denial,
- deterministic execution response,
- strict isolation,
- absence of speculative recovery.

Any architecture that depends on speculative rollback, fault-driven enforcement, or undefined failure behavior violates this specification.

12 Implementation Profiles

This section defines normative implementation profiles of the Policy-Centric Processor (PCP). Implementation profiles specify permitted configurations of the architecture for different deployment classes. All compliant implementations MUST conform to at least one profile defined in this section.

Profiles differ only in quantitative parameters. All qualitative architectural requirements remain unchanged.

12.1 Profile Definition Rules

Each implementation profile MUST define the following parameters explicitly:

- number of Execution Units,
- number of Memory Shards,
- per-shard service capacity,
- address width,
- policy storage capacity.

Profiles MUST NOT modify:

- execution semantics,
- policy enforcement rules,
- addressing model,
- consistency guarantees.

Profiles MUST be mutually compatible at the software level.

12.2 Server Profile

The Server Profile targets high-throughput, multi-tenant systems.

- The implementation MUST support a large number of Execution Units.
- The implementation MUST support multiple Memory Shards.
- The number of Memory Shards MUST scale with physical memory sockets.

Additional requirements:

- Policy storage MUST support a large number of concurrent identities.
- Memory bandwidth MUST scale approximately linearly with shard count.
- Failure isolation MUST be preserved under full system load.

The Server Profile MUST prioritize throughput, isolation, and predictability.

12.3 Desktop Profile

The Desktop Profile targets general-purpose interactive systems.

- The implementation MUST support multiple Execution Units.
- The implementation MUST support multiple Memory Shards.
- The number of Memory Shards MAY be smaller than in the Server Profile.

Additional requirements:

- Interactive latency MUST remain bounded under load.
- Policy enforcement MUST incur deterministic overhead.
- Memory sharding MUST remain transparent to application software.

The Desktop Profile MUST balance responsiveness and parallel throughput.

12.4 Embedded Profile

The Embedded Profile targets resource-constrained systems.

- The implementation MUST support at least one Execution Unit.
- The implementation MUST support one or more Memory Shards.
- The number of Memory Shards MAY be minimal.

Additional requirements:

- Policy enforcement MUST be fully hardware-enforced.
- No dynamic address translation MUST exist.
- Deterministic execution MUST be preserved under all conditions.

The Embedded Profile MUST prioritize simplicity and verifiability.

12.5 Scalability Requirements

All profiles MUST satisfy the following scalability properties:

- Adding Execution Units MUST NOT require architectural changes.
- Adding Memory Shards MUST NOT affect software correctness.
- Policy enforcement semantics MUST remain invariant under scaling.

Scaling MUST NOT introduce centralized bottlenecks.

12.6 Interoperability Requirements

Different profiles MUST remain software-compatible.

- Executables MUST NOT encode profile-specific assumptions.
- Policy definitions MUST be portable across profiles.
- Address semantics MUST remain identical.

Profile selection MUST be a deployment decision, not a programming concern.

12.7 Summary

Implementation Profiles define:

- quantitative scaling parameters,
- deployment-oriented constraints,
- invariant architectural semantics.

Any implementation that alters architectural behavior based on profile selection violates this specification.

13 Formal Properties and Verifiability

This section defines the formal properties of the Policy-Centric Processor (PCP) architecture and the mandatory requirements for its verification. Verifiability is a primary architectural objective. All compliant implementations MUST satisfy the requirements stated below.

13.1 Architectural Determinism

The PCP architecture MUST be deterministic at the architectural level.

- Given identical initial architectural state and inputs, execution MUST produce identical architectural outcomes.
- Architectural behavior MUST NOT depend on timing, contention, or microarchitectural history.
- No hidden state MUST influence architectural correctness.

Determinism MUST hold independently of Memory Shard count and system scale.

13.2 Absence of Speculative State

The architecture MUST contain no speculative architectural state.

- No instruction MUST execute beyond its architectural commit point.
- No speculative memory access MUST occur.
- No rollback or recovery logic MUST be required for correctness.

All architectural state transitions MUST be final when performed.

13.3 Finite-State Characterization

The architectural state of the processor MUST be representable as a finite state machine.

- All architecturally visible state MUST be enumerable.
- No unbounded microarchitectural buffers MUST affect correctness.
- All state transitions MUST be explicitly defined.

Unbounded queues or heuristics MUST NOT influence architectural behavior.

13.4 Policy Enforcement Verifiability

Policy enforcement MUST be formally verifiable.

- All policy-relevant inputs MUST be explicit.
- Policy decisions MUST be traceable to architectural state.
- Policy evaluation MUST be side-effect free.

It MUST be possible to prove that no unauthorized memory access can occur under any execution.

13.5 Non-Interference Guarantees

The architecture MUST satisfy non-interference properties.

- Denied accesses MUST NOT influence observable system behavior.
- Execution timing MUST NOT encode policy decisions.
- Memory and execution isolation MUST be provable.

No side channel MUST arise from policy enforcement mechanisms.

13.6 Compositional Verification

Architectural components MUST be verifiable compositionally.

- Execution Units MUST be verifiable independently.
- Memory Shards MUST be verifiable independently.
- Policy enforcement MUST be verifiable independently of execution logic.

System-level correctness MUST follow from component-level proofs.

13.7 Scalability of Proofs

Verification complexity MUST scale linearly with system size.

- Adding Execution Units MUST NOT invalidate existing proofs.
- Adding Memory Shards MUST NOT require re-verification of execution semantics.
- Policy enforcement proofs MUST remain invariant under scaling.

Verification MUST NOT require global re-analysis.

13.8 Tool Independence

Formal verification MUST NOT depend on proprietary or opaque tools.

- Architectural properties MUST be expressible in standard formal models.
- Verification MUST be possible using multiple independent toolchains.
- Correctness MUST NOT rely on tool-specific assumptions.

The architecture MUST be verifiable using open formal methods.

13.9 Runtime Observability

The architecture MUST support observability for verification and auditing.

- Architectural state MUST be inspectable.
- Policy decisions MUST be auditable.
- Execution outcomes MUST be attributable to specific causes.

Observability MUST NOT weaken security or correctness.

13.10 Summary

The PCP architecture is formally verifiable due to:

- architectural determinism,
- absence of speculative state,
- explicit policy enforcement,
- finite-state characterization,
- compositional scalability of proofs.

Any architecture that relies on speculation, hidden heuristics, or unverifiable microarchitectural behavior violates this specification.

A Comparative Non-Goals

This appendix enumerates explicit non-goals of the Policy-Centric Processor (PCP) architecture. These non-goals clarify scope boundaries and prevent misinterpretation. All items listed below are intentionally excluded from the architecture.

A.1 Backward Compatibility

The PCP architecture does not aim to preserve backward compatibility with existing processor architectures.

- Compatibility with legacy instruction sets is not a goal.
- Compatibility with legacy memory models is not a goal.
- Compatibility with speculative execution semantics is not a goal.

Backward compatibility MAY be addressed by software layers, but MUST NOT constrain the architecture.

A.2 Peak Single-Thread Performance

Maximizing peak single-thread performance is not a goal.

- Instruction-per-cycle maximization is not a primary objective.
- Benchmark-driven optimization is not a design criterion.
- Performance derived from speculation is explicitly excluded.

The architecture prioritizes predictability and scalability over peak metrics.

A.3 Cache-Centric Optimization

Cache-centric performance optimization is not a goal.

- Complex cache hierarchies are not required.
- Cache coherence protocols are not assumed.
- Cache-based speculation is excluded.

Memory performance is addressed through sharding and parallel service, not cache hierarchy depth.

A.4 Implicit Concurrency

Implicit or inferred concurrency is not a goal.

- Automatic parallelization by hardware is excluded.
- Instruction-level parallelism inference is excluded.

- Hidden helper threads are excluded.

All concurrency MUST be explicit and software-directed.

A.5 Heuristic-Based Optimization

Heuristic-based optimization is not a goal.

- Performance heuristics are excluded from correctness.
- Adaptive prediction mechanisms are excluded.
- Learning-based microarchitectural behavior is excluded.

Correctness MUST NOT depend on probabilistic behavior.

A.6 Page-Oriented Memory Management

Page-oriented memory management is not a goal.

- Page tables are excluded.
- Address translation mechanisms are excluded.
- Page-sized protection granularity is excluded.

Memory protection is governed exclusively by explicit policy enforcement.

A.7 Fault-Driven Correctness

Fault-driven correctness is not a goal.

- Page faults are excluded as a protection mechanism.
- Speculative faults are excluded.
- Recovery-based enforcement is excluded.

Correctness MUST be preserved by construction.

A.8 Transparent Hardware Magic

Transparency through hidden hardware mechanisms is not a goal.

- Implicit prefetching is excluded.
- Implicit reordering is excluded.
- Undocumented microarchitectural state is excluded.

All behavior MUST be explicit and architecturally visible.

A.9 Universal Optimality

Universal optimality across all workloads is not a goal.

- No single architecture can optimize all use cases.
- Trade-offs are acknowledged and intentional.
- Profile-specific optimization is permitted only within defined profiles.

The architecture defines correctness and clarity, not universal supremacy.

A.10 Summary

The PCP architecture explicitly does not pursue:

- backward compatibility,
- speculative performance gains,
- cache- and page-centric models,
- heuristic-driven correctness.

These exclusions are intentional and foundational. They define the boundaries within which the architecture is coherent, verifiable, and policy-centric.

B Conformance

This section defines the criteria for conformance to the Policy-Centric Processor (PCP) architecture. Conformance is binary: an implementation either conforms to this specification or it does not.

B.1 Architectural Conformance

An implementation MUST be considered conformant if and only if it satisfies all normative requirements stated using the terms *MUST* and *MUST NOT* in this specification.

- All Architectural Axioms MUST be satisfied.
- All explicitly prohibited mechanisms MUST be absent.
- All required architectural properties MUST be implemented as specified.

Partial or selective conformance is not permitted.

B.2 Profile Conformance

A conformant implementation MUST declare exactly one or more supported Implementation Profiles as defined in Section *Implementation Profiles*.

- Each declared profile MUST be fully implemented.
- Profile selection MUST NOT alter architectural semantics.
- Profile parameters MUST remain within architecturally permitted bounds.

An implementation MUST NOT claim conformance by redefining or extending profiles.

B.3 Policy Enforcement Conformance

A conformant implementation MUST enforce all memory access policies as specified.

- All memory accesses MUST be subject to pre-access policy validation.
- Denied accesses MUST be non-observable.
- Policy enforcement MUST be deterministic and verifiable.

Any implementation that permits unauthorized memory access under any condition is non-conformant.

B.4 Execution Model Conformance

A conformant implementation MUST implement the execution model exactly as specified.

- One Execution Unit MUST execute exactly one architectural thread.
- No speculative execution MUST occur.
- No simultaneous multithreading MUST be implemented.

Any deviation from the defined execution semantics constitutes non-conformance.

B.5 Memory Architecture Conformance

A conformant implementation MUST implement symmetric memory sharding with independent service engines.

- Memory Shards MUST be architecturally symmetric.
- Shard service MUST be independent and non-centralized.
- Address-to-shard routing MUST be deterministic.

Architectures relying on centralized arbitration or NUMA-style asymmetry are non-conformant.

B.6 Addressing and Consistency Conformance

A conformant implementation MUST implement the addressing and memory consistency models as specified.

- Addresses MUST be pure numerical locators.
- No address translation MUST occur.
- Memory ordering MUST be strict and deterministic.

Relaxed or speculative memory models are not permitted.

B.7 Verification Conformance

A conformant implementation MUST be formally verifiable at the architectural level.

- Architectural state MUST be finite and explicit.
- Policy enforcement MUST be provably correct.
- Non-interference properties MUST be demonstrable.

Implementations whose correctness depends on undocumented or unverifiable behavior are non-conformant.

B.8 Documentation Requirements

A conformant implementation MUST provide sufficient documentation to demonstrate compliance.

- All architecturally visible mechanisms MUST be documented.
- All deviations from conventional architectures MUST be explicit.
- Conformance claims MUST be auditable.

Lack of documentation constitutes non-conformance.

B.9 Summary

An implementation conforms to the PCP architecture if and only if it:

- satisfies all mandatory requirements,
- declares and fully implements valid profiles,
- enforces policy deterministically and preventively,
- adheres to the defined execution, memory, and addressing models,
- is formally verifiable and fully documented.

Any implementation that violates or weakens any normative requirement defined in this specification is non-conformant.

C Glossary

This glossary defines the normative terminology used throughout the Policy-Centric Processor (PCP) architecture specification. All terms defined in this section have precise architectural meaning. Interpretations inconsistent with these definitions are non-conformant.

Address

A pure numerical locator identifying a byte location in the unified address space. An Address carries no authority, ownership, protection, or validity semantics.

Architectural State

All processor state that is architecturally visible and relevant to correctness, including registers, program counter, policy-relevant identifiers, and explicitly defined control state.

Architectural Determinism

The property that identical initial architectural state and inputs produce identical architectural outcomes, independent of timing or contention.

Execution Unit

A hardware core capable of executing exactly one architectural thread. An Execution Unit does not multiplex threads and executes instructions in program order without speculation.

Architectural Thread

A single, sequential instruction stream executed by one Execution Unit. Architectural threads do not migrate between Execution Units.

Identity

An explicit, immutable identifier associated with an executing architectural thread. Identity is used for policy evaluation and is assigned by system software.

Policy

An explicit set of rules governing whether a memory access is permitted. Policy is evaluated deterministically prior to memory access and produces an allow or deny decision.

Policy Enforcement

The mandatory architectural mechanism that validates all memory accesses against active policy prior to execution.

Denied Access

A memory access that fails policy validation. Denied accesses are non-observable and produce no memory, timing, or microarchitectural side effects.

Memory Shard

An independently serviced memory bank with a dedicated memory service engine (DMA). Memory Shards are architecturally symmetric and participate in a unified address space.

Memory Service Engine

The hardware component responsible for accepting, queuing, and servicing memory requests for a single Memory Shard.

Symmetric Memory Sharding

A memory architecture in which multiple Memory Shards are equal in latency class, semantics, and access rights, and are serviced independently.

Unified Address Space

A single, flat address space encompassing all Memory Shards. Shard boundaries are not exposed to software.

Shard Mapping

A deterministic function that maps addresses to Memory Shards. Shard Mapping does not encode policy or protection semantics.

Blocking

The condition in which an Execution Unit stalls awaiting completion of a memory operation. Blocking is local to the Execution Unit and does not affect others.

Synchronization

Explicit software-defined coordination between architectural threads. Synchronization is not inferred or performed implicitly by hardware.

Atomic Operation

An indivisible memory operation that is globally linearizable and appears to execute instantaneously with respect to other atomic operations.

Memory Consistency Model

The architectural rules governing the ordering and visibility of memory operations across Execution Units.

Non-Interference

The property that denied accesses and policy decisions do not influence observable system behavior through timing, ordering, or side channels.

Speculative Execution

Execution of instructions or memory accesses prior to architectural commitment. Speculative execution is explicitly excluded from the PCP architecture.

Fault-Based Enforcement

Any mechanism that detects unauthorized access after an access attempt. Fault-based enforcement is explicitly excluded from the PCP architecture.

Implementation Profile

A parameterized configuration of the PCP architecture defining quantitative characteristics such as number of Execution Units and Memory Shards, without altering architectural semantics.

Conformance

The property of an implementation that fully satisfies all mandatory requirements defined by this specification, without exception or weakening.

System Software

Software responsible for defining policy, assigning identity, structuring concurrency, managing memory allocation, and handling execution lifecycle.

Formal Verifiability

The property that architectural correctness, policy enforcement, and non-interference guarantees can be proven using formal methods.

NUMA (Non-Uniform Memory Access)

A memory architecture exposing locality-based latency distinctions. NUMA semantics are explicitly excluded from the PCP architecture.

D Reference Implementation Sketch

This section provides a non-exhaustive reference implementation sketch of a Policy-Centric Processor (PCP)–compliant system. The purpose of this section is illustrative: it demonstrates one concrete way to realize the architecture while preserving all normative requirements. Any deviation from this sketch is permitted provided conformance is maintained.

D.1 Top-Level Structure

A reference PCP implementation consists of the following top-level components:

- one or more Execution Units,
- a Policy Enforcement Unit,
- a Memory Routing Fabric,
- multiple Memory Service Engines,
- multiple Memory Shards.

All components operate under a single unified address space and are connected through explicit, deterministic interfaces.

D.2 Execution Units

Each Execution Unit is implemented as a simple, in-order core.

- The core executes a single architectural thread.
- The core contains a fixed register file, program counter, and control logic.
- No speculative execution logic is present.

The Execution Unit issues memory requests synchronously and stalls locally on blocked accesses.

D.3 Policy Enforcement Unit

The Policy Enforcement Unit (PEU) is a dedicated hardware block responsible for validating all memory accesses.

- The PEU receives memory access requests from Execution Units.
- The PEU evaluates access requests against active policy state.
- The PEU outputs a binary allow/deny decision.

The PEU operates strictly before any memory routing or shard selection.

D.4 Memory Routing Fabric

The Memory Routing Fabric is a simple, deterministic interconnect.

- It receives approved memory requests from the PEU.
- It applies a deterministic shard-mapping function.
- It forwards requests to the selected Memory Service Engine.

The routing fabric contains no centralized queues and no speculative buffering.

D.5 Memory Service Engines

Each Memory Service Engine (MSE) services exactly one Memory Shard.

- Each MSE maintains its own request queue.
- Each MSE operates independently of other MSEs.
- Backpressure from one MSE does not affect others.

MSEs handle request scheduling, ordering, and completion for their shard.

D.6 Memory Shards

Each Memory Shard is a physically distinct memory bank. For a system with N execution units and K memory shards, the maximum number of in-flight memory requests is $O(N \cdot K)$ without centralized arbitration.

- Memory Shards are architecturally symmetric.
- Each shard connects only to its associated MSE.
- Shards expose no policy or protection semantics.

Shards may be implemented using DRAM, HBM, SRAM, or other RAM technologies.

D.7 Control and Policy Storage

Policy state and execution identity information are stored in explicit, architecturally visible registers or tables.

- Policy storage is writable only by system software.
- Policy updates are synchronized explicitly.
- Policy state is independent of memory shard topology.

No policy information is cached or inferred implicitly.

D.8 System Software Interface

System software interacts with the processor through explicit control interfaces.

- Interfaces exist to load policy state.

- Interfaces exist to assign execution identities.
- Interfaces exist to start and stop Execution Units.

No implicit mode switching or hidden privilege transitions occur.

D.9 Scalability Characteristics

The reference implementation scales by replication.

- Execution Units scale horizontally.
- Memory Shards and MSEs scale horizontally.
- The routing fabric scales as a simple switch.

Scaling does not introduce centralized bottlenecks or semantic changes.

D.10 Minimal Configuration Example

A minimal conformant implementation may consist of:

- one Execution Unit,
- one Policy Enforcement Unit,
- one Memory Service Engine,
- one Memory Shard.

A maximal implementation may replicate these components arbitrarily, subject only to physical constraints.

D.11 Summary

This reference implementation sketch demonstrates that the PCP architecture can be realized using simple, modular components:

- no speculative execution,
- no centralized memory arbitration,
- explicit policy enforcement,
- symmetric memory sharding.

Correctness and scalability arise from structure, not from heuristic complexity.

E Formal Model Appendix: FSM and Transition Rules

This appendix defines a formal finite-state model of the Policy-Centric Processor (PCP) architecture. The model is normative: all conformant implementations MUST be representable as an instance of this model. All rules below use deterministic transition semantics.

E.1 State Space

The global architectural state S MUST be defined as:

$$S = \langle \mathcal{E}, \mathcal{P}, \mathcal{M}, \mathcal{R}, \mathcal{Q} \rangle$$

where:

- $\mathcal{E}$ is the set of Execution Unit states,
- $\mathcal{P}$ is the Policy state,
- $\mathcal{M}$ is the unified Memory state (partitioned into shards),
- $\mathcal{R}$ is the Shard Mapping state (deterministic and static),

- $\mathcal{Q}$ is the set of per-shard request/response queues.

No additional hidden architectural state MUST exist.

E.2 Execution Unit State

For each Execution Unit u, its architectural state E_u MUST be:

$$E_u \;=\; \langle PC_u, RF_u, ID_u, ST_u, OUT_u \rangle$$

where:

- PC_u is the program counter,
- RF_u is the architectural register file,
- ID_u is the immutable execution identity,
- $ST_u \in \{\textsc{Run}, \textsc{Stall}, \textsc{Term}\}$ is the unit status,
- OUT_u is the outcome/termination code (defined by software convention).

Each Execution Unit MUST have exactly one ID_u during execution. ID_u MUST NOT change while $ST_u = \textsc{Run}$.

E.3 Policy State

Policy state $\mathcal{P}$ MUST be a finite structure:

$$\mathcal{P} \;=\; \langle \Pi, \Pi_{\text{ver}} \rangle$$

where:

- Π is a finite set of policy rules,
- Π_{ver} is a monotonically increasing policy version.

Policy rules MUST be evaluated deterministically. Policy evaluation MUST be side-effect free.

E.4 Memory and Shards

Unified memory $\mathcal{M}$ MUST be partitioned into $k \geq 1$ shards:

$$\mathcal{M} \;=\; \{M_0, M_1, \ldots, M_{k-1}\}$$

Each shard M_i is a total mapping from addresses to bytes within its assigned address subset.

Shard topology MUST NOT change architectural semantics.

E.5 Shard Mapping Function

Shard routing MUST be defined by a deterministic mapping function:

$$\mathcal{R} : \mathit{Addr} \to \{0, 1, \ldots, k-1\}$$

$\mathcal{R}$ MUST be static under a fixed configuration. $\mathcal{R}$ MUST NOT depend on time, contention, or history.

E.6 Requests and Responses

A memory request req MUST be represented as:

$$req \;=\; \langle u, op, a, w, n, ver \rangle$$

where:

- u is the issuing Execution Unit,
- $op \in \{\textsc{Load}, \textsc{Store}, \textsc{Atomic}\}$,
- a is the address,
- w is write data (empty for loads),
- n is access size in bytes,
- ver is the policy version observed at issue time.

A response $resp$ MUST be represented as:

$$resp \;=\; \langle u, status, data \rangle$$

where:

- $status \in \{\textsc{Ok}, \textsc{Denied}, \textsc{Fault}\}$,
- $data$ is read data (empty for stores/denials).

Denied accesses MUST yield $status = \textsc{Denied}$ and MUST have no side effects.

E.7 Transition System

The PCP MUST be representable as a deterministic transition system:

$$S' \;=\; \delta(S, I)$$

where I is the set of external inputs (interrupts, control commands, and explicit software events). All transitions MUST be deterministic.

E.8 Instruction Step Rule

For any Execution Unit u with $ST_u = \textsc{Run}$, one instruction step MUST be:

$$\langle PC_u, RF_u \rangle \xrightarrow{instr} \langle PC'_u, RF'_u, act \rangle$$

where act is either:

- $\varnothing$ (no memory action),
- a memory action $mem(op, a, w, n)$.

Instruction decode and execution MUST NOT perform speculative actions.

E.9 Policy Evaluation Rule

For any memory action $mem(op, a, w, n)$ issued by unit u, policy evaluation MUST be performed before routing:

$$Allow \;=\; Eval(\Pi, ID_u, op, a, n)$$

$Eval$ MUST be deterministic and side-effect free.

- If $Allow = \textsc{false}$, then the action MUST NOT be routed to memory.
- If $Allow = \textsc{true}$, then the action MUST be routed to exactly one shard.

Denied evaluation MUST produce no architectural side effects beyond the defined unit response.

E.10 Denied Access Transition

If $Allow = \textsc{false}$ for a memory action issued by u, then the next state MUST satisfy:

- $ST'_u \in \{\textsc{Stall}, \textsc{Term}\}$ deterministically,
- OUT'_u MUST be set deterministically if $ST'_u = \textsc{Term}$,

- $\mathcal{M}$ MUST remain unchanged,
- $\mathcal{Q}$ MUST remain unchanged.

No cache, buffer, or microarchitectural state MUST be modified by denial.

E.11 Routing Transition

If $Allow = \textsc{true}$, shard routing MUST be:

$$i \;=\; \mathcal{R}(a)$$

and a request MUST be enqueued into shard queue Q_i:

$$Q_i' \;=\; Enq(Q_i, req)$$

All other shard queues MUST remain unchanged.

Routing MUST be deterministic.

E.12 Shard Service Transition

Each shard i MUST service requests independently.

For each shard i, if Q_i is non-empty, the service engine MUST select a request deterministically:

$$req \;=\; Select(Q_i)$$

and produce exactly one corresponding response $resp$.

- $Select$ MUST NOT depend on global queues.
- $Select$ MUST NOT depend on other shards' state.

Service MAY be modeled as one request per transition or batched transitions, but MUST preserve determinism.

E.13 Load Semantics

For $req.op = \textsc{Load}$, if serviced, the response MUST be:

$$data \;=\; Read(M_i, a, n)$$

$$resp \;=\; \langle u, \textsc{Ok}, data \rangle$$

and memory M_i MUST remain unchanged.

E.14 Store Semantics

For $req.op = \textsc{Store}$, if serviced, memory MUST be updated:

$$M_i' \;=\; Write(M_i, a, w, n)$$

$$resp \;=\; \langle u, \textsc{Ok}, \epsilon \rangle$$

E.15 Atomic Semantics

For $req.op = \textsc{Atomic}$, the operation MUST be linearizable.

There MUST exist a total order over all atomic requests such that each atomic transition appears instantaneous with respect to all other atomics.

For each atomic request:

$$(M_i, RF_u) \xrightarrow{\textsc{Atomic}} (M_i', RF_u')$$

and:

$$resp \;=\; \langle u, \textsc{Ok}, data \rangle$$

Atomic semantics MUST be independent of cache coherence.

E.16 Response Delivery

When a response $resp = \langle u, status, data \rangle$ is produced, it MUST be delivered deterministically to the issuing Execution Unit.

- If $status = \textsc{Ok}$ and $op = \textsc{Load}$, RF_u MUST be updated.
- If $status = \textsc{Ok}$ and $op = \textsc{Store}$, no registers MUST be updated.
- If $status = \textsc{Denied}$, the unit MUST already be in the denial response state.

Response delivery MUST NOT reorder instructions within an Execution Unit.

E.17 Consistency Properties

The following properties MUST hold:

- **Program Order:** within an Execution Unit, memory actions are issued in program order.
- **No Speculation:** no memory request exists without a committed instruction cause.
- **Preventive Enforcement:** denied requests never enter shard queues.
- **Shard Independence:** shard service is independent across shards.
- **Atomic Linearizability:** all atomic operations are globally linearizable.

E.18 Non-Interference Property

Denied accesses MUST be non-interfering.

Formally, for any two executions differing only in a denied access attempt, all externally observable outputs MUST be identical except for the defined denial outcome of the issuing Execution Unit.

No timing, ordering, or memory side effects MUST be observable due to denied access.

E.19 Conformance to the Formal Model

An implementation MUST be considered conformant to this appendix if:

- its architectural state can be mapped onto S,
- its behavior can be described by a deterministic transition function δ,
- policy enforcement satisfies the preventive denial transitions,
- memory and atomic semantics satisfy the rules above.

Any implementation that requires hidden speculative state, fault-driven enforcement, or non-deterministic transitions violates this formal model.

F Minimal Invariants: Safety, Liveness, and Non-Interference

This section defines the minimal invariant set that MUST hold for any Policy-Centric Processor (PCP) implementation. All invariants below are normative and MUST be satisfied under all executions.

F.1 Safety Invariants

S1: Preventive Authorization

No unauthorized memory access MUST occur.

Formally, for any transition that updates memory state:

$$\delta(S, I) = S' \wedge \mathcal{M}' \neq \mathcal{M} \;\Rightarrow\; \exists req :\; Eval(\Pi, ID_{req.u}, req.op, req.a, req.n) = \text{TRUE}$$

All memory-modifying transitions MUST be preceded by an allow decision under the active policy.

S2: Denial Non-Effect

Denied accesses MUST be non-observable and side-effect free.

If a request is denied, then:

$$\mathcal{M}' = \mathcal{M} \;\wedge\; \mathcal{Q}' = \mathcal{Q}$$

and no architectural state other than the issuing unit outcome MAY change.

S3: No Speculative Memory Actions

No memory request MUST exist without a committed instruction cause.

Formally, all enqueued requests MUST correspond to an executed instruction step of the issuing Execution Unit in program order.

S4: Single-Thread Execution Unit Integrity

Each Execution Unit MUST preserve single-thread semantics.

- PC_u MUST advance deterministically.
- RF_u MUST update only according to committed instruction semantics.
- No other thread state MUST be interleaved into E_u.

S5: Shard Symmetry Preservation

All Memory Shards MUST remain architecturally symmetric.

Formally, for any two shards i and j, access semantics MUST be equivalent:

$$Sem(M_i) \equiv Sem(M_j)$$

No shard-specific permission, priority, or locality semantics MUST exist.

S6: Deterministic Shard Mapping

Shard mapping MUST be deterministic and stable.

For any fixed configuration:

$$\forall a :\; \mathcal{R}(a) = \mathcal{R}(a)$$

and $\mathcal{R}$ MUST NOT depend on time, contention, or history.

S7: Atomic Linearizability

All atomic operations MUST be globally linearizable.

There MUST exist a total order over all atomic operations consistent with their observed effects such that each atomic appears instantaneous.

F.2 Liveness Invariants

L1: Local Progress Under Service

An Execution Unit in RUN state MUST make progress if its required memory requests are authorized and serviced.

Formally, if:

- $ST_u = \text{RUN}$,
- all issued memory actions satisfy $Eval = \text{TRUE}$,
- selected shards continue to service requests,

then PC_u MUST advance without indefinite stalling.

L2: Shard Service Non-Starvation

Each Memory Shard service engine MUST provide non-starvation under load.

Formally, if a request req remains enqueued in Q_i and the system continues to transition, then req MUST eventually be selected and completed.

L3: Denial Termination Determinism

Denied accesses MUST lead to a deterministic, finite outcome.

If a denial occurs for Execution Unit u, then within a finite number of transitions, u MUST enter either STALL or TERM in a deterministic manner defined by the architecture-software contract.

F.3 Non-Interference Invariants

N1: Policy Decision Non-Interference

Policy decisions MUST not influence observable behavior except through the defined allow/deny outcome.

For any two executions that differ only in a denied access attempt:

- all memory states of other identities MUST remain identical,
- all responses to other Execution Units MUST remain identical,
- no timing or ordering artifacts MUST differ architecturally.

N2: Denial Timing Non-Disclosure

Denial MUST not be distinguishable via architectural timing.

Formally, denial handling MUST not introduce variable-latency architectural effects observable by unauthorized identities.

N3: No Speculative Side Channels

No speculative state MUST exist that could leak information.

Since speculative execution is excluded, no speculative cache, buffer, or predictor state MUST influence architectural observables.

F.4 Compositionality Invariants

C1: Unit Independence

Execution Units MUST be compositional.

Adding an Execution Unit MUST NOT change the correctness of existing units, except through explicit, policy-authorized shared memory interactions.

C2: Shard Independence

Memory Shards MUST be compositional.

Adding a Memory Shard MUST NOT change access semantics or policy enforcement, except by increasing available service capacity.

C3: Policy Scaling Invariance

Policy enforcement MUST scale invariantly.

Changing the number of Execution Units or Memory Shards MUST NOT alter policy evaluation semantics.

F.5 Summary

A conformant PCP implementation MUST satisfy:

- Safety: preventive authorization, denial non-effect, no speculation, deterministic mapping, shard symmetry, atomic linearizability.
- Liveness: local progress under service, non-starvation, deterministic denial outcome.
- Non-Interference: policy decisions do not leak through timing or side effects.
- Compositionality: scaling preserves semantics and proofs.