Hexa ISA: A Multi-Valued Instruction Set Architecture for Confidence-Aware and Approximate Computation (2025)

Hye-Eun Yoon. Author, *Independent Researcher*

***Abstract*— Contemporary binary instruction set architectures (ISAs) encode operations under a precision-centric assumption, leaving approximation tolerance, confidence, and fallback behavior unrepresentable at the instruction level. As a result, uncertainty and execution validity are reconstructed post hoc in software, obscuring intent and preventing explicit coordination across the ISA–firmware–hardware boundary—an increasingly acute limitation for relational reasoning systems and complex multi-scale simulations.**

This paper introduces Hexa ISA, a hexadecimal opcode–based ISA framework that restores a missing semantic layer by decoupling logical intent from execution constraints. Hexa defines each instruction as a composite of an opcode-derived semantic domain and a meta-execution field specifying approximation allowance, confidence targets, and fallback preferences. By leveraging nibble-aligned radix-16 encoding, the framework enables orthogonal partitioning of semantic operator families (e.g., modal, relational/quantifier, set-theoretic, and equivalence domains) while remaining explicitly emulatable on existing binary hardware via firmware-level interpretation.

Hexa reframes ISA extension as a minimal representational upgrade: it introduces explicit contract-based execution semantics without requiring new hardware paradigms or claiming immediate performance gains. The proposed semantic map and contract model provide a stable interface for heterogeneous execution substrates and support formal reasoning at the opcode level independent of physical realization.

***Index Terms*— Approximate computing, Computer architecture, Hardware-software codesign, Instruction sets, Multivalued logic, Probabilistic computing, Semantics.**

I. INTRODUCTION

A. Background

CONTEMPORARY **binary instruction set architectures (ISAs)** are fundamentally designed under the assumption that computation is characterized solely by **numerical precision [1]**.

However, modern computational workloads—particularly in **artificial intelligence** and **large-scale scientific simulation**—routinely rely on operations whose semantics inherently involve **confidence**, **approximation tolerance**, and **fallback behavior** under failure [2], [3].

Current ISAs provide no mechanism to express these properties at the level of **operational semantics**. As a result, even when approximate or probabilistic computations are executed in hardware, their associated **uncertainty** and **error characteristics** are handled *post hoc* at higher software layers. This leads to the absence of an **explicit contract** across the **ISA–firmware–hardware boundary** [4].

This structural gap is acute in **relational reasoning systems** and **complex multi-scale simulations** *(Section II.C)*, where the absence of explicit semantic contracts causes **super-linear error accumulation** and forces optimization to be bounded by software-level orchestration rather than physical computation [5], [6], [7].

Against this backdrop, this paper proposes a **hexadecimal-based instruction set architecture framework** that extends the expressive capacity of operation semantics beyond binary precision.

B. Design Considerations

The choice of **hexadecimal (radix-16)** is not motivated by information density alone [8]. Rather, it represents the **minimal radix** that allows **orthogonal grouping** of binary-compatible operations, relational operators, modal semantics, and **meta-level execution contracts** within a fixed-width opcode space, while remaining **efficiently emulatable on existing binary hardware.** The proposed framework aims to provide a **structural foundation** for representing **relational operations** at their **smallest meaningful unit,** aligning computational expressivity with both human relational reasoning and hardware-level feasibility.

1) Semantic Atomicity and Minimal Expressive Units

Binary logic (0,1) provides an optimal substrate for representing physical states, but it is **insufficient for expressing relational or modal semantics as atomic operations.** Modal notions such as *necessity, possibility, contradiction, and indeterminacy* cannot be naturally encoded as **indivisible instruction-level primitives** under a purely binary encoding [8], [9].

Instead, they are decomposed into multiple binary operations, resulting in **semantic fragmentation.**

While a **2-bit encoding (radix-4)** is sufficient to represent basic modal distinctions, it lacks the expressive capacity to jointly encode both the **semantic family** of an operation and its

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. The author declares no conflicts of interest. The author takes sole responsibility for the conceptualization, theoretical development, and manuscript preparation. **(Corresponding author: H.-E. Yoon.)**

H.-E. Yoon is an Independent Researcher, South Korea (e-mail: sellyes@selly.org).

Digital Object Identifier <placeholder>

contextual specialization *(e.g., relational, set-theoretic, or equivalence-based semantics)*.

A **4-bit encoding** (**radix-16**) constitutes the **minimal information unit** that enables **semantic atomicity** at the instruction-level, allowing operation families and their variants to be represented as a single, indivisible opcode.

2) Nibble-Level Alignment with Existing Binary Architectures

Contemporary computing architectures are natively aligned to **8-bit (byte-level) boundaries.** A hexadecimal encoding maps exactly to a **4-bit nibble,** enabling efficient partitioning of instruction fields within existing binary hardware. This allows, for example, a single byte to encode both an **opcode and execution metadata,** or two independent hexadecimal instructions, without introducing additional bit masking or shifting overhead [10].

Higher-radix encodings *(e.g., radix-32 or radix-64)* would incur significant **emulation overhead** on binary hardware, due to increased bit manipulation and alignment costs. In contrast, **radix-16** represents a **minimal extension** that remains efficiently emulatable, while significantly expanding **semantic expressiveness** at the instruction-level.

3) Orthogonality and Operator Grouping

The radix-16 opcode space enables **orthogonal grouping** of operator families, such as *modal logic, relational operators, set-theoretic operations, and equivalence relations.* By allocating **contiguous opcode ranges** to each semantic family, the proposed framework enforces **structural separation** between distinct classes of operations.

This orthogonal structure is not only beneficial for extensibility and clarity, but also provides a foundation for **formal reasoning** at the ISA level. Because instruction classes are explicitly separated and **hierarchically encoded,** properties such as non-interference, compositionality, and semantic isolation become amenable to formal verification techniques [11], [12]**.**

4) Compatibility with Emerging Multi-State Devices

Many emerging hardware substrates, including **memristive, spintronic,** and **photonic devices**, naturally exhibit **multi-state or continuous behaviors** [13], [14]. A radix-16 abstraction provides a **structurally compatible interface** between such physical substrates and binary emulation layers, while maintaining a **power-of-two encoding** that preserves **computational stability**.

5) Scalability and Instruction Space Extension

To accommodate future expansion of execution metadata, the framework allows for **firmware-level escape or prefix opcodes**, through which extended semantic or physical execution information may be **negotiated**. This approach preserves **backward compatibility** at the binary execution level, while allowing the **semantic depth** of instructions to grow hierarchically over time.

C. Related Work and Positioning

1) Multivalued Logic and Non-Binary Computing

The conceptual exploration of multi-valued logic spans several decades.

Various **multi-valued logic systems**, including **ternary and quaternary logic**, have been proposed since the 1960s–1980s [9]. Theoretically, they have been discussed as having advantages such as increased **information density**, improved **computational efficiency**, and reduced **wiring complexity**. However, these approaches failed to scale to general-purpose computing architectures [1], [9] and mostly remained at the experimental stage or confined to specific application research.

The limitations of past multi-valued attempts can be summarized into three main points.

First, physical stability and process issues.

Past multi-valued logic primarily relied on subdividing analog voltage levels to represent multiple states, making it extremely vulnerable to **process variations**, **thermal noise**, and **reduced noise margins**. This made it difficult to ensure large-scale integration and reliable operation, ultimately failing to secure a clear manufacturing advantage over CMOS-based binary logic.

Second, the absence of a semantic hierarchy.

Existing multi-valued logic research primarily emphasized advantages centered on **expression density**—such as *"expressing more states per bit"*—but there were almost no attempts to structurally represent the **semantics of computation**—for example, *whether approximation is allowed, reliability, or alternative paths* in case of failure. In other words, multi-valued logic was treated merely as a simple replacement for binary logic and failed to evolve towards reconfiguring the computational model itself.

Third, there is a disconnect with the software-hardware ecosystem.

Multiply-precision logic has shown little consideration for compatibility with existing ISAs, compilers, operating systems, and programming languages, consequently failing to form a practical software stack. This has become a major factor severely limiting real-world adoption potential, separate from technical feasibility.

Fourth, the historical absence of a qualitative necessity for semantic expansion.

In previous decades, computational workloads were primarily centered on **deterministic, precision-oriented arithmetic and logic**. Within this paradigm, the standard binary operator set provided a sufficient substrate for nearly all practical applications, and the inherent **semantic limitations** of the ISA did not yet constitute a critical bottleneck. Consequently, prior attempts at multi-valued logic were viewed merely as a means of increasing information density rather than a **necessary evolution of operational semantics**. Lacking a clear mandate for new types of **relational or modal operators**, these attempts failed to gain traction against the rapid scaling of the binary CMOS ecosystem [1], [15].

2) Emerging Hardware Devices Beyond Binary Logic

Meanwhile, new hardware elements such as *memristors, spintronic devices, and photonic computing* do not inherently presuppose only discrete binary states [2], [4]; they possess physical characteristics that can naturally express **continuous or multiple states.**

Existing research encompasses various approaches to non-binary and approximate computation. **Stochastic computing (SC),** dating back to the 1960s, represents data as probabilistic bit streams to simplify arithmetic logic [16], [17]. Similarly, neuromorphic architectures like Intel's **Loihi** [18] and IBM's **TrueNorth** [19] leverage spiking neural networks to perform inherently approximate and event-driven processing.

Furthermore, **asynchronous logic** *(or micropipelines)* has long proposed breaking the rigid clock synchronization in favor of handshake-based execution contracts [20]. In the domain of deep learning accelerators, quality-configurable multipliers and approximate circuits have been extensively studied to trade precision for energy efficiency [21].

Nevertheless, most of these devices are currently used to **emulate or accelerate specific kernels** within the **binary computation paradigm.** Despite the hardware level enabling **multi-state or stochastic representation,** the ISA and compiler layers still expose only **binary computational semantics** [2], [4]. The rich expressive capabilities of SC or neuromorphic chips remain confined to the level of internal optimization or specialized APIs, rather than being exposed as general-purpose architectural primitives.

This is less a technical limitation and more a consequence of the **absence of a higher-level syntax** capable of encapsulating computational semantics.

Current ISAs are designed with computational precision as a given and cannot explicitly express concepts like **confidence**, **approximation range**, **error tolerance**, or **fallback paths** at the level of computational meaning. As a result, approximate calculations or probabilistic variations arising in hardware must be handled retrospectively at the software layer [22], leaving an **explicit contract between ISA–firmware–hardware absent**.

3) Quantum Computing as a Partial, Not Structural, Solution

Quantum computing is often proposed as an alternative paradigm to overcome the limitations of classical binary computation, particularly in domains characterized by **exponential state spaces.** However, despite its theoretical advantages, current quantum computing frameworks face fundamental bottlenecks that are not solely attributable to hardware constraints such as qubit count or decoherence.

A central limitation lies in the **representational and control layers:** quantum computing instruction sets primarily focus on precise unitary evolution and probabilistic measurement outcomes, while lacking mechanisms to explicitly encode **approximation tolerance, confidence bounds, or fallback semantics** at the instruction-level [23].

As a result, the burden of managing uncertainty, approximation, and error propagation remains largely externalized to **classical control software**, leading to significant overhead at the **classical–quantum interface**. This structural separation prevents quantum systems from serving as a general-purpose substrate for **relational reasoning** and **confidence-aware computation**.

In this sense, quantum computing alleviates certain computational bottlenecks but does not resolve the more general problem of **insufficient expressiveness in instruction-level semantics**—a problem that also manifests in classical binary ISAs.

Unlike quantum computing frameworks, which focus on overcoming computational complexity through unitary evolution but remain tethered to classical semantic control, **Hexa ISA** resolves the more fundamental problem of **instruction-level expressiveness**.

4) Positioning of This Work

This study does not inherit the goal of **"increasing expression density"** from existing multiradix logic.

Instead, it **redefines the problem** from the perspective of the **minimal radix required to expand the expressive range of operational meaning**.

The proposed **Hexadecimal Opcode-based ISA framework** is designed to:

- Maintain **emulation feasibility** on binary hardware;
- **Orthogonally separate** the semantic classes of operators;
- Explicitly express **meta-operation information** such as *approximations, reliability, and alternative paths* at the ISA level.

This approach differs from existing multiradix research in that it does not require new hardware, aims for **gradual integration** with existing compilers and software stacks, and provides a **common computational grammar** that can naturally accommodate future heterogeneous device-based hardware.

This work **does not claim immediate performance superiority** over existing binary ISAs, nor does it propose a complete replacement of current hardware architectures.

Instead, the goal is to introduce an **intermediate semantic layer** at the ISA level, through which approximation, confidence, and fallback behaviors can be **explicitly expressed and negotiated** across the ISA–firmware–hardware boundary.

Empirical performance evaluation and hardware-level validation are intentionally left for future work, as the **primary contribution** of this paper lies in the **formalization of a missing representational layer** in contemporary computing systems.

II. Problem Statement: The Missing Semantic Layer in ISAs

A. *Precision-Centric Assumptions in Contemporary ISAs*

Contemporary instruction set architectures are fundamentally designed around a **precision-centric view of computation**. [1], [24]

At the ISA level, operations are specified under the implicit assumption that execution semantics are fully characterized by

numerical correctness and **deterministic behavior**.
Instructions encode *what* operation to perform and *on which* operands, but remain agnostic to *how reliable*, *how approximate*, or *under what conditions* the result should be considered acceptable.
Within this paradigm, **correctness is treated as a binary property**: an operation either produces the correct result or it does not. Any notion of **uncertainty**, **approximation tolerance**, or **execution contingency** is assumed to be handled outside the ISA, typically at higher software layers. As a result, the instruction set itself provides **no formal means** to express whether an operation permits approximation, requires a confidence threshold, or mandates an alternative execution path in the event of failure or degradation.
This design assumption was historically well-aligned with workloads dominated by **deterministic arithmetic**, **control flow**, and **exact logical operations**. In such contexts, numerical precision served as a sufficient proxy for correctness, and the absence of richer semantic descriptors at the ISA level did not constitute a practical limitation.
However, **modern computational workloads increasingly violate this assumption**[15]. In artificial intelligence, probabilistic inference, and large-scale simulation, many operations are **semantically valid across a range of approximations**, confidence levels, or execution strategies. Yet contemporary ISAs lack the **expressive capacity** to distinguish between operations that require strict precision and those whose correctness is inherently **conditional or graded**. Consequently, current systems implicitly **conflate two fundamentally different concerns**:

- the **numerical execution** of an operation, and
- the **semantic validity** of its result under uncertainty or approximation.

By encoding only the former at the ISA level, existing architectures force the latter to be **reconstructed *post hoc* in software**, obscuring execution intent and preventing explicit coordination between the ISA, firmware, and hardware layers. This precision-centric assumption forms the root of a broader **structural gap**, which becomes explicit when approximation, confidence, and fallback behaviors are required as **first-class properties of computation**. The following sections formalize what is missing (*II.B*) and why it matters at scale (*II.C*).

B. Semantic Gaps: Approximation, Confidence, Fallback

Contemporary ISAs expose computation primarily through **deterministic, precision-oriented operations**.
While modern hardware increasingly supports **approximate**, **probabilistic**, or **non-deterministic execution modes**, these properties are **not reflected at the level of instruction semantics**.
In particular, three categories of semantic information remain **fundamentally unrepresentable** within current ISA designs.

1) Approximation

Whether an operation may be approximated, and under what tolerance, **cannot be expressed explicitly** at the instruction-level [2], [7]. Approximate execution is therefore implemented *implicitly*—either through specialized accelerators or through software-level heuristics [4]—without a **formal contract** specifying acceptable error bounds or approximation intent.

2) Confidence

The **reliability or confidence** associated with an execution result is **not a first-class concept** in existing ISAs. Even when hardware exhibits **probabilistic behavior** or **variable accuracy** *(e.g.*, *due to noise, reduced precision, or analog effects)* [22], the resulting uncertainty is **not communicated upward** through the execution stack [3]. As a result, confidence assessment must be reconstructed *post hoc* in software, if at all.

3) Fallback Semantics

Current ISAs provide **no mechanism** to specify **alternative execution paths** when a desired accuracy, confidence threshold, or execution condition cannot be satisfied. Failure handling is therefore **externalized to software control flow**, introducing additional overhead and obscuring the original semantic intent of the operation.
Crucially, these limitations are **not artifacts of insufficient hardware capability**. Rather, they arise from the **absence of an explicit semantic layer** at the ISA level through which approximation, confidence, and fallback behavior can be **declared and negotiated** across the ISA–firmware–hardware boundary.
As computational workloads increasingly rely on **non-exact**, **relational**, and **inference-driven operations**, the lack of such semantic expressiveness becomes a **structural bottleneck** rather than an implementation detail.

C. Consequences for AI and Simulation Workloads

The absence of an explicit semantic layer at the ISA level has **concrete and compounding consequences** for contemporary computational workloads, particularly in **artificial intelligence** and **large-scale scientific simulation**.
In modern AI systems—including machine learning, natural language processing, and reasoning models—core operations are inherently **relational**, **approximate**, and **probabilistic** [5], [25]**.** Model training and inference routinely rely on **soft decisions**, **confidence thresholds**, **approximate comparisons**, and **fallback strategies** under uncertainty. However, because current ISAs expose only precision-centric arithmetic and logic, these semantic properties must be **reconstructed at higher software layers** through control flow, numerical heuristics, or auxiliary metadata structures.
This leads to a **systematic inversion of responsibility**: operations that are **semantically primitive** at the algorithmic level are **decomposed into sequences of low-level binary instructions**, while their associated uncertainty and reliability constraints are managed externally by software frameworks. As model complexity grows, this decomposition introduces **cumulative overhead**, **brittle control logic**, and **limited opportunities for optimization** across the software–hardware boundary [6].
A similar pattern arises in **large-scale scientific simulations**. Domains such as *quantum many-body systems, lattice QCD, turbulence modeling, protein folding, and strongly correlated*

physical systems rely heavily on **approximate solvers**, **iterative convergence criteria**, and **adaptive precision strategies**. In these settings, the inability to express approximation tolerance, confidence bounds, or fallback conditions at the instruction-level forces such semantics to be handled through **global control structures** and **repeated synchronization points**.

As a result, **error accumulation may grow super-linearly** with system scale, and optimization becomes constrained not by the underlying physical computation but by the **limits of software-level orchestration**. This creates **hard thresholds** beyond which additional computational resources yield **diminishing returns**, regardless of hardware capability.

Crucially, these limitations do not arise from **insufficient computational power**, nor from the lack of specialized accelerators. Rather, they stem from a **structural mismatch** between the **semantic nature of modern workloads** and the **expressiveness of the instruction set architectures** on which they are executed.

From this perspective, the challenge is not to further accelerate existing binary operations, but to provide a **representational substrate** in which **relational**, **approximate**, and **confidence-aware operations** can be treated as **first-class computational entities**.

Taken together, this mismatch reflects not merely a limitation of performance optimization, but an **architectural deficiency** arising from the **absence of an explicit semantic layer** at the ISA level.

Addressing this deficiency at the ISA level enables a **principled redistribution of responsibility** across the ISA–firmware–hardware stack, reducing semantic reconstruction overhead and opening new avenues for optimization that remain inaccessible under precision-only instruction semantics.

Accordingly, this paper proposes a **hexadecimal-based instruction set architecture** designed to restore this missing layer of **semantic expressiveness**.

III. Hexa ISA Overview

A. *Design Principles*

The design of Hexa ISA is guided by three core principles: **Binary Emulatability**, **Semantic Explicitness**, and **Orthogonality**.

Together, these principles define the **minimal conditions** under which instruction-level semantic expressiveness can be expanded without disrupting existing binary execution models.

1) Binary Emulatability

Hexa ISA is designed to be **explicitly emulatable on existing binary hardware** without requiring modifications to physical execution units.

All hexadecimal opcodes and meta-fields are representable through **structured binary decoding** and **firmware-level interpretation**, ensuring **backward compatibility** with contemporary x86-64 and ARM-based systems [10]. This constraint is intentional. The goal of Hexa is not to introduce a new hardware paradigm, but to **extend the semantic expressiveness** of the instruction set while remaining deployable within current software–hardware ecosystems [1]. By preserving binary emulatability, Hexa enables **incremental adoption** and evaluation without fragmenting existing toolchains.

2) Semantic Explicitness

Conventional ISAs encode *what* operation to perform, but remain silent about *how* the result should be interpreted under uncertainty, approximation, or execution degradation [2]. Hexa ISA addresses this limitation by **explicitly separating operation semantics from execution semantics**.

Each instruction may carry **meta-level information** describing **approximation intent**, **confidence requirements**, or **fallback behavior**. These semantic attributes are treated as **first-class components** of the instruction definition, rather than implicit properties reconstructed in software [4]. By making **execution intent explicit** at the ISA level, Hexa establishes a **clear semantic contract** [26] that can be interpreted consistently across the ISA–firmware–hardware stack.

3) Orthogonality

Hexa ISA enforces **orthogonality** across instruction classes by structurally separating **semantic operator families**—such as *modal, relational, set-theoretic, and equivalence operators*—into distinct opcode regions. This separation ensures that the **meaning of an instruction is invariant** with respect to its physical execution strategy.

Crucially, this orthogonality enables **flexible role assignment** across **heterogeneous hardware substrates**. While instruction semantics remain fixed at the ISA level, their realization may vary across **CMOS logic**, **memristive arrays**, **photonic devices**, or other emerging multi-state technologies [13], [14]. The choice of physical substrate affects *how* an operation is executed, but not *what it means*.

This design **decouples semantic correctness from hardware implementation**, allowing compilers and firmware to negotiate execution strategies without altering the logical structure of programs. As a result, Hexa provides a **stable semantic interface** through which both software languages and hardware designs can evolve independently, ensuring that complex operations remain amenable to **formal verification** within their respective semantic domains [11].

B. Instruction Encoding Model

The Hexa ISA adopts a **structured instruction encoding model** that explicitly separates **operation semantics** from **execution semantics**.

This separation is realized through a **dual-field design** consisting of an **opcode field**, which defines the semantic meaning of an instruction, and a **meta-execution field** (hereafter, **meta-field**), which specifies how that meaning should be realized under practical execution constraints.

1) Opcode vs. Meta-Execution Field

n conventional ISAs, opcodes implicitly **conflate semantic intent with execution assumptions**.

Hexa ISA **decouples these concerns** by assigning distinct roles to each component of the instruction encoding:

- **Opcode field:** Encodes the **semantic class** and **logical meaning** of an operation—such as *modal, relational, set-theoretic, or equivalence semantics*. It defines *what* is being asserted or computed, maintaining **logical independence** from physical execution.
- **Meta-field:** Encodes **execution-level attributes**, including **approximation allowance**, **confidence targets**, and **fallback preferences**. These fields do not alter the logical meaning of the operation; instead, they specify the **constraints and expectations** for its realization.

This separation ensures that **semantic correctness** and **execution strategy** remain orthogonal, enabling consistent interpretation across software, firmware, and hardware layers.

2) Nibble-Based Layout and Binary Alignment

Hexa ISA utilizes a **4-bit nibble-based structure** to ensure natural alignment with byte-oriented binary architectures. By mapping each hexadecimal digit to a structured binary representation, the architecture enables:

- **Structural Orthogonality:** Semantic operator families are grouped into contiguous ranges without disrupting binary alignment.
- **Embeddable Metadata:** Meta-fields can be integrated into existing instruction widths or introduced via prefixed extensions.
- **Hardware Compatibility:** Decoding is handled by standard binary decode paths and firmware-level interpretation, requiring no additional bit-manipulation hardware [10].

This layout provides a **minimal yet powerful extension,** substantially expanding semantic expressiveness within existing binary ecosystems, similar to recent efforts in extending RISC-V for controlled approximation [27].

3) Fixed Semantic Meaning, Variable Execution Realization

The encoding model strictly **decouples logical intent from its physical realization**. The opcode defines the **immutable semantic meaning** (*what*), while the meta-field governs the **realization** (*how*) under specific constraints. This separation ensures that:

- **Heterogeneous Mapping:** Diverse hardware substrates *(e.g., CMOS, memristive)* can implement the same opcode based on their unique physical characteristics [22].
- **Dynamic Strategy Selection:** Compilers and firmware can negotiate execution strategies *(e.g., approximation levels)* without altering program semantics.
- **Formal Verifiability:** Correctness can be reasoned at the opcode level, independent of execution variability.

This design establishes a **stable semantic core,** allowing heterogeneous execution mechanisms to be coordinated without ambiguity.

C. Execution Model Overview

The Hexa ISA execution model introduces a **structured flow of semantic intent** across the ISA–firmware–hardware stack, formalizing how instruction-level meaning and execution expectations are negotiated without prescribing specific implementation strategies.

At the **ISA level**, each instruction defines an **immutable semantic intent** through its opcode, accompanied by **meta-execution attributes** that specify constraints such as approximation tolerance and confidence requirements. These attributes do not alter program logic; instead, they function as **declarative execution contracts** [26] that inform downstream decisions across system layers.

At the **firmware level**, these contracts are translated into **concrete execution strategies**. As a **semantic mediator**, the firmware resolves how an instruction's declared intent should be realized given available execution units or precision modes, without modifying the logical structure of the instruction stream.

At the **hardware level**, execution mechanisms optimize within the bounds of the declared contract. Different physical substrates—from conventional CMOS to emerging multi-state devices—may realize the same instruction through distinct paths, provided that the **semantic and confidence constraints are respected**. This **principled redistribution of responsibility** allows uncertainty and fallback behavior to be managed coherently as **first-class architectural concepts [15]**.

IV. Instruction Semantics and Opcode Structure

This section formalizes the Hexa ISA semantic map by **decoupling logical intent from execution constraints**. We define a hexadecimal instruction **I** as a composite of its semantic domain and execution attributes:

$$I = (S, M) \in O \times E$$

where $\boldsymbol{S}$ represents the **opcode-derived semantic domain** and $\boldsymbol{M}$ denotes the **meta-field constraints.** This formulation adopts a **contract-based architectural approach** [26], ensuring that the logical definition of an operation remains invariant while its physical realization allows for negotiated variability.

Table I presents a unified overview of this semantic space, partitioning the opcode domain into orthogonal families and assigning **canonical operators** within each. The table **defines semantic intent** rather than execution strategy; variability is governed independently through the meta-execution field. This structure ensures each semantic family $\boldsymbol{S_i}$ remains **invariant and isolated** under heterogeneous execution realizations.

Detailed opcode mappings and bit-level specifications are reserved for future technical standards and the official Hexa ISA reference manual. The complete specification is currently accessible via the repository linked in *Appendix A*.

TABLE I
Overview of Hexa Opcode Semantic Structure

Hex Range	*Family (Semantic Domain)*	*Canonical Operators (Examples)*	*Notes (Orthogonality/Isolation)*
0x00–0x7F	Legacy Binary	MOV, ADD, SUB, MUL, DIV, JMP, …	Backward-compatible region [10]
0x80–0x8F	Modal	□(Necessity), ◇(Possibility), ⊤, ⊥	Mutually exclusive modal flags; no dataflowside-effects
0x90–0x9F	Quantifier/ Relational	∀, ∃, ∃!, ∄	Scope encoded via MF; no aliasing with 0xB*
0xA0–0xAF	Comparison/ Order	≪, ≫, ⩽, ⩾, ⩽̸, ⩾̸	Total/partial order variants
0xB0–0xBF	Set-Theoretic	⊂, ⊃, ⊆, ⊇, ∩, ∪, \, △	Closed under family; MF selects tolerance
0xC0–0xCF	Equivalence/ Similarity	≡, ≅, ∼, ≢, ≇	Metric-bound via MF.conf
0xD0–0xDF	Quantum Control (abstract)	prep, entangle, measure (abstracted)	Classical contract only; no physical requirement [23]
0xE0–0xEF	Reserved (Future)	—	Escape/prefix expansions
0xF0–0xFF	Prefix/ Escape	MF width extension, vendor space	Negotiated by firmware

A. *Opcode Space Partitioning*

The Hexa ISA partitions the **8-bit opcode space** into **contiguous hexadecimal regions**, each corresponding to a **distinct semantic domain**.

The **lower half of the space (0x00–0x7F)** is reserved for **legacy binary instructions**, ensuring **full backward compatibility** with existing x86-64 and ARM-style execution semantics [10]. These opcodes retain their conventional meaning and are unaffected by Hexa-specific extensions.

The **upper half of the space (0x80–0xFF)** is allocated to **Hexa semantic regions**. Each region encodes a **single semantic family**—modal, relational, comparative, set-theoretic, equivalence-based, or abstract control semantics—**without overlap or aliasing**. This strict partitioning enforces **semantic isolation**: instructions from different families cannot be confused or conflated through decoding or execution shortcuts.

Reserved and prefix regions (0xE*–0xF*) provide **controlled extensibility**. Rather than expanding the core semantic space indiscriminately, Hexa delegates future growth to **explicit escape and prefix mechanisms**, preserving the integrity of existing opcode meanings while allowing **firmware-mediated negotiation** of extended semantics.

B. Semantic Operator Families

Within each semantic region, opcodes represent **canonical operators** drawn from **well-defined logical or relational domains**.

Modal operators encode **necessity**, **possibility**, and **logical extrema** as first-class instruction semantics, without introducing side effects on dataflow. **Quantifier and relational operators** express **existence**, **universality**, and **relational scope** directly at the instruction-level, eliminating the need to reconstruct such semantics through control-flow patterns.

Comparison and order operators distinguish **total and partial ordering semantics**, while **set-theoretic operators** encode **membership**, **inclusion**, and **set operations** as **closed families**. **Equivalence and similarity operators** represent **graded notions of sameness and difference**, explicitly separating **exact identity** from **metric- or tolerance-based similarity**.

Crucially, each family is **internally closed and externally isolated**. Operators within a family may vary by specialization or meta-execution parameters, but their **semantic domain does not overlap** with that of other families. This **orthogonality** enables formal reasoning about instruction behavior and prevents unintended semantic interactions across domains.

Abstract control regions, including **quantum-inspired operators**, are defined purely at the **semantic contract level**. Their inclusion does not imply physical quantum hardware requirements; rather, they establish a **unified semantic vocabulary** through which heterogeneous execution substrates may later be addressed, consistent with control abstractions for NISQ-era systems [23].

C. Meta-Execution Fields

While opcodes define *what* an instruction means, **meta-execution fields** specify *how* that meaning should be realized under practical constraints.

Meta-fields encode attributes such as ***approximation allowance***, ***confidence targets***, *and* ***fallback preferences***, without modifying the logical semantics of the opcode itself. This approach mirrors recent architectural trends in explicit approximation control, such as *Risk-5 [27],* but extends the concept to a broader range of semantic domains beyond arithmetic.

This separation ensures that **semantic correctness** can be reasoned about **independently of execution variability**. **Approximation flags** indicate whether relaxed execution is permissible; **confidence targets** specify acceptable reliability thresholds; **fallback semantics** define alternative execution expectations when constraints cannot be satisfied. None of these alter the asserted operation—they instead form a **declarative execution contract** [26]**.**

By externalizing execution variability into meta-fields, Hexa allows **compilers and firmware to select execution strategies dynamically** while preserving **semantic invariance**. This design enables heterogeneous hardware substrates to participate in execution without fragmenting program meaning, and it establishes a **stable foundation** for **confidence-aware** and **approximation-aware computation** at the ISA level.

V. ISA–Firmware Contract Model

A. Explicit Contracts vs Implicit Software Handling

Contemporary computing systems rely predominantly on **implicit software-level mechanisms** to manage approximation, uncertainty, and execution failure [2], [3]. In existing ISAs, instruction semantics specify only the operation to be performed, while **execution validity**—such as *acceptable error bounds, confidence thresholds, or fallback behavior*—is **reconstructed externally** through control flow, numerical heuristics, or runtime checks.

This design places the **burden of semantic interpretation entirely on higher software layers**. As a result, execution intent is **fragmented across the software stack**: compilers encode heuristics, runtime systems enforce thresholds, and hardware executes operations without awareness of their semantic acceptability. The absence of an **explicit contract** at the ISA level obscures execution intent and prevents **principled coordination** between software, firmware, and hardware components [26].

Hexa ISA replaces this implicit handling model with an **explicit contract-based architecture**. Each instruction declares not only its **semantic intent** through the opcode, but also its **execution expectations** through meta-execution attributes. These attributes specify whether approximation is permitted, what confidence level is required, and how execution should proceed if constraints cannot be satisfied. By elevating these properties to **first-class ISA constructs**, Hexa transforms execution variability from an implicit software concern into an **explicit architectural agreement**. The ISA no longer encodes only *what* computation is requested, but also *under what conditions* the result is considered **semantically valid**.

This explicit contract model enables a **principled redistribution of responsibility** across system layers. The compiler expresses semantic tolerance and intent **declaratively**, firmware resolves feasible execution strategies given available resources, and hardware optimizes execution within the bounds of the declared contract [27]. Crucially, **none of these layers are required to reconstruct or infer semantic intent *post hoc***.

In contrast to traditional designs—where correctness is binary and deviation is treated as failure—Hexa allows **graded correctness** and **conditional validity** to be represented natively. This shift establishes a **stable interface** for **confidence-aware**, **approximation-tolerant computation** without entangling semantic meaning with specific execution mechanisms.

B. Confidence-Aware Execution Semantics

Hexa allows **confidence requirements** to be declared explicitly at the ISA level, enabling execution outcomes to be evaluated against **semantic thresholds** rather than binary correctness alone. At runtime, the meta-execution field encodes a **target confidence level** that can be compared against hardware- or firmware-level reliability signals, such as ***precision modes, noise estimates, or self-diagnostic feedback [22], [25].*** This mechanism allows execution validity to be assessed within the declared semantic contract, without reconstructing uncertainty through software-level control flow.

C. Error Budget and Fallback Negotiation

Fallback behavior is treated as a **contractual property** of execution, allowing firmware-level negotiation when declared constraints cannot be satisfied. When a physical execution substrate signals that confidence or approximation requirements cannot be met within the available **error budget [2],** firmware selects a **predefined semantic fallback path**—such as *relaxed precision, alternative execution units, or deferred evaluation*—without altering program logic. By localizing fallback handling within the contract model, Hexa avoids software-level interrupt overhead while preserving semantic intent.

D. Formal Verifiability and Orthogonality Implications

By separating semantic intent from execution variability, Hexa enables **formal reasoning** at the opcode level **independent of physical realization**. Because semantic operator families are **orthogonally partitioned** and execution attributes are explicitly declared, **correctness properties can be verified** with respect to semantic domains rather than specific hardware implementations [11]. This separation establishes a **stable foundation** for **ISA-level verification**, even under heterogeneous or evolving execution substrates [12].

VI. Emulation Feasibility on Binary Hardware

This section argues that Hexa ISA can be realized on contemporary binary hardware **without requiring architectural modification**. The goal is not to demonstrate performance gains, but to establish **logical feasibility** and **compatibility** within existing execution models.

A. *Binary Emulation Strategy*

Hexa ISA is designed such that all instruction semantics can be emulated through **structured binary decoding** and **firmware-level interpretation**. Each hexadecimal instruction is decomposable into a **fixed-width binary representation**, where the opcode defines semantic intent and the meta-field encodes execution constraints. These components can be interpreted sequentially or in parallel by existing firmware mechanisms, without altering the underlying execution units. Crucially, Hexa **does not require new primitive operations** at the hardware level. All semantic operators are mapped onto existing **binary control paths**, **microcode routines**, or **software-managed execution flows**. The ISA extension therefore operates as a **semantic overlay** rather than a physical replacement, allowing deployment within current binary execution pipelines.

B. Opcode Decoding without Hardware Modification

Hexa opcode decoding **does not rely on new hardware decoding logic**. Instead, semantic interpretation is delegated to **firmware or microcode layers** that already mediate

instruction decoding, exception handling, and execution mode selection in modern processors. By reserving **contiguous opcode regions** and enforcing **strict partitioning** between semantic families, Hexa enables **deterministic decoding paths** that can be implemented through **lookup tables** or **microcode extensions**. Prefix and escape regions further allow semantic expansion without interfering with legacy instruction decoding. Because decoding remains aligned to existing **byte- and nibble-level boundaries**, no additional bit-level manipulation or hardware redesign is required. From the perspective of the physical execution units, Hexa instructions remain **binary-encoded control signals** whose interpretation is resolved upstream.

C. Why Radix-16 is the Minimal Viable Extension

Radix-16 represents the **minimal extension beyond binary** that simultaneously satisfies **semantic expressiveness**, **orthogonality**, and **emulation feasibility**. Lower-radix encodings fail to meet these requirements for fundamentally similar reasons.

Radix-4 and radix-8, while capable of representing limited modal or categorical distinctions, lack sufficient **representational capacity** to jointly encode **semantic family**, **operator variant**, and **execution attributes** within a single atomic instruction. As a result, relational or contextual semantics must be **decomposed across multiple instructions** or overloaded fields, reintroducing **semantic fragmentation** and undermining orthogonality.

Decimal (radix-10) encodings present a different limitation. Although they offer greater nominal expressiveness than radix-8, they lack **power-of-two alignment** and cannot be cleanly partitioned into fixed semantic and meta-execution fields. This makes it difficult to reserve stable opcode regions while simultaneously allocating space for meta-fields and future extensions. The resulting **irregular binary mapping** increases decoding complexity and prevents efficient integration with byte-oriented architectures.

Conversely, **higher-radix encodings** introduce unnecessary overhead.

Radix-32 and radix-64 provide ample representational space but impose significant **decoding**, **alignment**, and **control-path complexity** when emulated on binary hardware. Supporting such radices requires wider instruction formats, additional bit manipulation, or expanded lookup mechanisms, increasing hardware and firmware complexity without proportional semantic gain. Moreover, higher radices **dilute semantic atomicity** by encouraging excessive granularity within a single instruction space, complicating formal reasoning and verification at the ISA level.

A **4-bit nibble (radix-16)** occupies a **unique boundary point**. It aligns naturally with byte-oriented architectures, enables strict partitioning of semantic families, and preserves sufficient internal space for operator variants, meta-execution attributes, and controlled extension mechanisms.

From this perspective, **radix-16 is not chosen to maximize information density**, but to establish the **smallest representational unit** capable of supporting **instruction-level semantic contracts** under binary emulation constraints.

D. Summary of Section VI

Taken together, these properties demonstrate that Hexa ISA **does not depend on speculative hardware assumptions**. Its semantic extensions are realizable through existing **firmware-mediated execution paths**, preserving binary compatibility while introducing a **missing representational layer** at the ISA level.

VII. Discussion

A. *Scope and Limitations*

This work introduces a **semantic extension** at the ISA level and is intentionally scoped to **formalization** and **representational design** rather than implementation. Hexa does not prescribe specific hardware realizations, compiler optimizations, or execution policies, nor does it attempt to replace existing binary ISAs. Instead, it defines a **missing semantic layer** through which approximation, confidence, and fallback intent can be **expressed explicitly** and **reasoned about formally** at the instruction level. Empirical validation, performance characterization, and hardware-level specialization are **deliberately deferred**, not due to feasibility constraints, but to preserve the **conceptual clarity** of the proposed abstraction. These aspects are expected to be addressed through subsequent, domain-specific investigations.

B. Design Intent and Extension Space.

Hexa is proposed as a **prototype-level semantic ISA scaffold**, rather than a closed or finalized architecture. Its design intentionally leaves **extension fields** and **semantic degrees of freedom unspecified**, allowing future work to explore how different hardware substrates, compiler strategies, and execution environments may instantiate or specialize these fields.

Importantly, this openness is **not accidental but a core design choice**, reflecting the view that **semantic intent should remain stable** even as implementation strategies evolve. As such, Hexa serves as a **coordination layer** across system boundaries, enabling independent yet compatible innovation in hardware, firmware, and compiler design [26].

C. Why This Is Not a Performance Paper

This paper **does not claim immediate or direct performance improvements** over existing binary ISAs. Any performance gains are expected to emerge **indirectly**, through reduced semantic reconstruction overhead, improved cross-layer coordination, and clearer execution contracts between software and hardware.

The **primary contribution** therefore lies in **semantic expressiveness** and **architectural clarity**, not in cycle-level optimization or throughput benchmarking.

Accordingly, **no benchmarks or hardware measurements are reported**, as such evaluations would necessarily depend on specific instantiations of the proposed semantic layer rather than on the layer itself.

D. Implications for Compilers and Hardware

By exposing execution intent explicitly at the ISA level, Hexa enables compilers to express **semantic tolerance declaratively** rather than procedurally. This allows compiler passes to reason about approximation, confidence, and fallback behavior as **first-class properties** [27], rather than encoding them implicitly through heuristic transformations. At the same time, firmware assumes a principled role as a **mediator of execution strategies**, while hardware implementations remain free to optimize within **well-defined semantic contracts**. Unlike approximate arithmetic circuits which focus primarily on trading precision for energy in numerical operations [21], Hexa generalizes this contract to logical and relational domains. This separation permits software languages and hardware substrates to **evolve independently**, while preserving **formal verifiability** and **interpretability** at the instruction level.

VIII. Future Work

Future work may explore multiple, independent research directions building upon the proposed semantic layer.

At the **compiler level**, integration with **LLVM-based pipelines** could enable **confidence-aware lowering**, **semantic metadata propagation**, and **fallback-aware execution** within existing infrastructures. These efforts need not assume a specific hardware realization and may be pursued modularly.

In parallel, **hardware-oriented studies** may investigate how the Hexa semantic layer can be instantiated across diverse substrates, including **heterogeneous architectures** that extend beyond conventional CMOS-only designs, such as ***memristive, photonic, or hybrid device technologies [13], [14], [18], [19].***

Crucially, these directions are **intentionally left open**: the present work does not privilege any particular implementation path. The role of this paper is to provide a **stable, formal semantic prototype** upon which multiple lines of inquiry—spanning compilers, firmware, and hardware—can proceed independently while remaining **conceptually aligned**.

IX. Conclusion

This paper introduced **Hexa ISA**, a hexadecimal opcode–based instruction set architecture that restores a **missing semantic layer** in contemporary computing systems. By **decoupling logical intent from execution constraints**, Hexa enables approximation, confidence, and fallback behavior to be expressed explicitly at the instruction-level and negotiated coherently across the **ISA–firmware–hardware stack**.

Rather than proposing a new hardware paradigm or performance optimization, this work formalizes a **minimal and emulatable extension** that expands the **expressive capacity of instruction semantics** while preserving compatibility with existing binary architectures [1], [10].

In doing so, Hexa establishes a **principled foundation** for **confidence-aware** and **relational computation**, addressing a structural limitation of precision-only ISAs and opening a path toward **semantically aligned compiler and hardware evolution**.

Appendix

A. Full Opcode Semantic Map

Due to space constraints, the complete opcode mapping table, including all 256 hexadecimal instruction definitions and their corresponding semantic domains, is hosted in an external repository. The full specification, along with the machine-readable .md format tables, is available at:

- **Repository:** https://github.com/Selly-Yoon/Hexa-ISA

B. Meta-field Encoding Examples

To illustrate the binary emulation feasibility of Hexa ISA, we provide concrete encoding examples for two representative operations: **Confidence-Aware Comparison** and **Approximate Set Inclusion**.

1) Example 1: Approximate Equality with Confidence Threshold

- **Logical Intent:** Compare two operands ($A \approx B$) with a required confidence of $\geq 95\%$.
- **Hexa Instruction:** 0xC231 (Opcode 0xC2 + Meta 0x31)
  - **Opcode 0xC2:** EQ_SIM (Similarity check) from the Equivalence family.
  - **Meta-field 0x31:** Encodes "High Confidence" and "Enable Approximation".

[Bit-Level Breakdown]

$$Instruction\ (16-bit) = \left[\ 1100_{Opcode\ (0xC2)} \mid 0011\ 0001_{Meta\ (0x31)}\ \right]$$

- **Byte 1 (Opcode):** 0xC2 selects the specific Similarity Operator ($\sim$).
- **Byte 2 (Meta):** 0x31 maps to the execution contract:
  - High Nibble (3): Target Confidence $\geq 95\%$ (Level 3).
  - Low Nibble (1): Approx Mode = Enabled.

2) Example 2: Set Inclusion with Fallback Strategy

- **Logical Intent:** Check if ($A \subseteq B$). If uncertain, return False (safe fallback)
- **Hexa Instruction:** 0xB006 (Opcode 0xB0 + Meta 0x06)
  - **Opcode 0xB:** SET_SUB (Subset check $\subseteq$).
  - **Meta-field 0x06:** Encodes "Strict Mode" + "Fallback: Return False".

[Interpretation]

- **Byte 1 (Opcode):** 1011 0000 (0xB0) activates the Subset logic unit.
- **Byte 2 (Meta):** 0000 0110 (0x06):
  - High Nibble (0): Confidence Target = Default/Ignored.
  - Low Nibble (6): Fallback Policy = Return False (Safe Fail).

These examples demonstrate how semantic intent (Opcode)

and execution constraints (Meta-field) are packed into aligned byte sequences, facilitating efficient decoding on standard 16-bit or 32-bit datapaths.

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, Fifth edition. Amsterdam Heidelberg: Elsevier, Morgan Kaufmann, 2012.
[2] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 1–33, May 2016, doi: 10.1145/2893356.
[3] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," in *Proceedings of The 33rd International Conference on Machine Learning*, PMLR, June 2016, pp. 1050–1059. Accessed: Dec. 24, 2025. [Online]. Available: https://proceedings.mlr.press/v48/gal16.html
[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," *Commun. ACM*, vol. 58, no. 1, pp. 105–115, Jan. 2015, doi: 10.1145/2589750.
[5] A. Santoro *et al.*, "A simple neural network module for relational reasoning," 2017, *arXiv*. doi: 10.48550/ARXIV.1706.01427.
[6] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, in ISCA '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–12. doi: 10.1145/3079856.3080246.
[7] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," 2015, *arXiv*. doi: 10.48550/ARXIV.1510.00149.
[8] Ł. Jan, *On 3-valued logic*. Amsterdam, Netherlands: North-Holland, 1970.
[9] Smith, "The Prospects for Multivalued Logic: A Technology and Applications View," *IEEE Trans. Comput.*, vol. C–30, no. 9, pp. 619–634, Sept. 1981, doi: 10.1109/TC.1981.1675860.
[10] "Intel 64 and IA-32 Architectures Software Developer's Manual." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[11] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, July 2009, doi: 10.1145/1538788.1538814.
[12] G. Klein *et al.*, "seL4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Big Sky Montana USA: ACM, Oct. 2009, pp. 207–220. doi: 10.1145/1629575.1629596.
[13] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008, doi: 10.1038/nature06932.
[14] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotech*, vol. 8, no. 1, pp. 13–24, Jan. 2013, doi: 10.1038/nnano.2012.240.
[15] C. E. Leiserson *et al.*, "There's plenty of room at the Top: What will drive computer performance after Moore's law?," *Science*, vol. 368, no. 6495, p. eaam9744, June 2020, doi: 10.1126/science.aam9744.
[16] B. R. Gaines, "Stochastic Computing Systems," in *Advances in Information Systems Science: Volume 2*, J. T. Tou, Ed., Boston, MA: Springer US, 1969, pp. 37–172. doi: 10.1007/978-1-4899-5841-9_2.
[17] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, p. 92:1-92:19, May 2013, doi: 10.1145/2465787.2465794.
[18] M. Davies *et al.*, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan. 2018, doi: 10.1109/MM.2018.112130359.
[19] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014, doi: 10.1126/science.1254642.
[20] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, June 1989, doi: 10.1145/63526.63532.
[21] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, Feb. 2020, doi: 10.1109/JPROC.2020.3006451.
[22] K. V. Palem, "Energy aware computing through probabilistic switching: a study of limits," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1123–1137, Sept. 2005, doi: 10.1109/TC.2005.145.
[23] J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018, doi: 10.22331/q-2018-08-06-79.
[24] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991, doi: 10.1145/103162.103163.
[25] V. Kuleshov, N. Fenner, and S. Ermon, "Accurate Uncertainties for Deep Learning Using Calibrated Regression," in *Proceedings of the 35th International Conference on Machine Learning*, PMLR, July 2018, pp. 2796–2804. Accessed: Dec. 24, 2025. [Online]. Available: https://proceedings.mlr.press/v80/kuleshov18a.html
[26] A. Sampson, J. Bornholt, and L. Ceze, "Hardware-Software Co-Design: Not Just a Cliché," *LIPIcs, Volume 32, SNAPL 2015*, vol. 32, pp. 262–273, 2015, doi: 10.4230/LIPICS.SNAPL.2015.262.
[27] I. Felzmann, J. F. Filho, and L. Wanner, "Risk-5: Controlled Approximations for RISC-V," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4052–4063, Jan. 2020, doi: 10.1109/TCAD.2020.3012312.

Hye-Eun (Selly) Yoon. Author (Independent Researcher)

Hye-Eun Yoon was born in **Deagu**, South Korea. She received the B.S. degree in computer science from Korea National Open University (KNOU), Seoul, South Korea, in **2022**.

She is currently an Independent Researcher based in Bucheon-si, South Korea. Her research focuses on addressing semantic limitations in conventional binary architectures through instruction-level abstractions. She proposed the **Hexa ISA** to enable confidence-aware execution and formalize the coordination between software intent and hardware constraints.

Her research interests include **computer architecture, hardware-software co-design, and neuro-symbolic systems**, with a particular focus on the intersection of biological cognition and computational logic.