HaackLang Language Specification

This is a the full HaackLang specification, made by Zoadrazorro using ChatGPT-5.1

Chapter 1 — The Need for HaackLang: Why Classical Logic Is Not Enough

Computing, since its inception, has been built upon the rigid skeleton of classical Boolean logic. True/false. Zero/one. A world measured in discrete, binary judgments. This simplicity enabled the early digital revolution, but it now limits the frontiers of cognitive modeling, AGI design, emergent reasoning, and complex adaptive systems.

Human cognition is not binary. Machine learning is not binary. Real reasoning unfolds across multiple tempos, levels, and logics simultaneously. HaackLang challenges the assumption that a single logical system can govern all reasoning, drawing heavily from Susan Haack's *Philosophy of Logics* to propose a pluralistic, polyrhythmic logical paradigm.

1.1 The Problem of Monologic Systems

Most programming languages assume a single, global logic—classical Boolean operations evaluated in a universal time-step. This works well when modeling:

- deterministic systems
- static propositions
- crisp truth assignments
- discrete decision boundaries

But it collapses under systems involving:

- uncertainty
- contradiction
- delayed reasoning
- emotional cognition
- reflective thought

- subconscious processes
- oscillatory or emergent behaviors

Classical logic allows only one truth rhythm—one update tempo. Minds, biological systems, and AGI-like architectures require many.

1.2 Susan Haack's Logical Pluralism

Susan Haack argued that logic should not be treated as a monolithic authoritarian structure ("there is one true logic") nor as an anything-goes relativism. Instead, logic should be treated as a *toolbox*, where different kinds of reasoning require different logical instruments.

HaackLang extends this idea into programming:

- Classical logic → crisp decisions
- Fuzzy logic → graded truth
- Paraconsistent logic → contradiction without collapse
- Temporal logics → reasoning over evolving states
- Meta-logic → reasoning about one's own reasoning

Rather than choosing between them, HaackLang allows **all to coexist**, each mapped to its own temporal rhythm.

1.3 Why Time Matters: Truth as a Signal

Traditional languages evaluate truth instantaneously and synchronously. But cognition is fundamentally asynchronous.

- Perception updates rapidly
- Beliefs update slowly
- Emotions update nonlinearly
- Meta-cognition updates sporadically
- Intuition updates in bursts

HaackLang's central innovation is the introduction of **Tracks**, each representing a separate logical timeline with its own update tempo. Truth-values become **dynamic signals**, not static assignments.

1.4 The Need for Polyrhythmic Logic

A single-track logic system cannot model systems where:

- fast instincts override slow beliefs
- slow reflection modifies fast impulses
- contradictory thoughts coexist
- truth oscillates or drifts over time
- emergent behaviors arise through temporal interference patterns

HaackLang's **BoolRhythm** type solves this, assigning a vector of truth-values to each proposition, with distinct values across tracks.

For example:

```
lamp_on = {
    main:   1.0,
    slow:   0.8,
    syncop: 0.2
}
```

This single proposition can be:

- objectively true on fast perception (`main`)
- tentatively believed on slow cognition (`slow`)
- emotionally ambiguous on intuition (`syncop`)

A mind-like system emerges from these asynchronous truth dynamics.

1.5 Why Contradiction Is Not an Error

Classical logic collapses under contradiction (explosion principle). But humans often maintain contradictory beliefs for years without mental collapse. HaackLang incorporates **paraconsistent tracks**, enabling safe handling of local contradictions.

Contradictions become *places of tension*, not catastrophic failures.

1.6 HaackLang as a Cognitive Modeling Language

HaackLang is not merely a new programming language. It is a framework for representing:

- multi-track reasoning
- distributed cognition
- AGI decision pipelines
- emotional/subconscious influences
- emergent truth-oscillation patterns
- context-sensitive logic
- meta-cognitive oversight

It is a language built for **minds**, not machines.

Future chapters will expand the full specification. Let me know when you're ready for **Chapter 2**.

Chapter 2 — Susan Haack and the Philosophy of Logics

Susan Haack's *Philosophy of Logics* forms the backbone of HaackLang's conceptual foundation. While classical logic aims for a single set of rules governing all reasoning, Haack proposes a radically different view: logic is plural, context-dependent, and deeply pragmatic. HaackLang translates these philosophical principles directly into computational constructs.

2.1 Classical Logic and Its Limits

Classical logic assumes:

- bivalence (true or false)
- non-contradiction
- explosion (anything follows from contradiction)
- timeless evaluation
- universal applicability

While powerful for mathematics and early computing, these assumptions fail to describe how humans and complex systems actually reason.

Humans:

- maintain ambiguous beliefs
- accept contradictions without collapse
- revise beliefs unevenly over time
- use different logics in different contexts
- shift reasoning modes under emotion, stress, or insight

Haack recognized these failures not as bugs of human cognition but as evidence that classical logic is insufficient as a universal model.

2.2 Logical Authoritarianism vs. Logical Anarchy

Haack positioned her critique between two extremes:

Authoritarianism:

"There is ONE true logic; all other logics are inferior or incorrect."

This view fails because different kinds of reasoning demand different tools. Deduction, induction, abduction, probabilistic inference, and analogical reasoning operate under distinct logics.

Anarchism:

"All logics are equally valid; choose whichever you like."

This leads to incoherence and destroys any notion of justification or rational structure.

Haack's Middle Path: Logical Pragmatism

Her insight was simple and profound:

> **Logic is a toolbox, not a religion.**

The correct logic is the one appropriate to the task.

HaackLang adopts this principle computationally by allowing:

- multiple logics inside a single program
- dynamic switching between logics
- parallel evaluation of logics via Tracks
- meta-logic to manage logical conflicts

2.3 Foundherentism and Hybrid Justification Models

Haack's theory of knowledge, foundherentism, rejects the dichotomy between:

- foundationalism (beliefs justified by base axioms)
- coherentism (beliefs justified by mutual coherence)

Instead, Haack proposes a hybrid system where justification:

- comes from both foundational elements **and** coherence relationships
- is dynamic rather than static
- reflects real inferential structures

In HaackLang, this maps to:

- **main track** → foundational updates (perception, crisp truth)
- **slow track** → coherence updates (reflection, belief revision)
- **syncop track** → contextual/emotional updates (intuition)

Belief justification becomes a *temporal, multi-track phenomenon*.

2.4 Logical Pluralism in HaackLang

Haack identifies several families of logic:

- classical
- fuzzy
- modal
- paraconsistent
- intuitionistic
- probabilistic
- relevance logics

HaackLang does not force a choice; it provides:

- **logic blocks** (local logic declarations)
- **track-bound logics** (different logics per temporal rhythm)
- **meta-logic** (for evaluating and reconciling cross-logic conflicts)

A HaackLang program may assign:

- classical logic to fast perceptual tracks
- fuzzy logic to slow belief-forming tracks
- paraconsistent logic to intuition or subconscious tracks

This mirrors how humans reason.

2.5 Contextualism and the Logic of Applicability

One of Haack's strongest insights is that logics have domains of applicability:

- Classical logic works well for arithmetic.
- Fuzzy logic works well for gradations.
- Paraconsistent logic works well for contradictions.
- Modal logic works well for possibilities.

HaackLang encodes this principle into **contexts**, so each context selects a logic (or a set of tracks with distinct logics) appropriate for its cognitive function.

Example:

```
context perception {
    using logic classical
}

context decision {
    using logic fuzzy
}

context intuition {
    using logic paraconsistent
}
```

2.6 The Foundation of HaackLang's Design

Haack's philosophy leads to three computational principles:

1. Polylogical Execution

Different logics are used in parallel.

2. Temporal Pluralism

Truth evolves at different tempos across tracks.

3. Meta-Logical Coherence

The system evaluates and resolves logical differences via high-level reasoning.

This produces a computational system that:

- tolerates contradictions
- adapts to context
- evolves reasoning over time
- uses multiple logics for different cognitive processes

2.7 From Philosophy to Computation

Haack’s theories were not originally intended for programming language design. But they translate naturally into it.

In HaackLang:

- Tracks = temporal logical lanes
- Contexts = domains of applicability
- BoolRhythm = vector of truth-values across logics
- Meta-logic = Haack’s pragmatic reconciliation layer

Her philosophy becomes a literal computational architecture.

Ready for **Chapter 3 — Logical Pluralism and Polylogical Pragmatism**?

Chapter 3 — Logical Pluralism and Polylogical Pragmatism

Classical programming languages assume a single logic that governs the entire system. HaackLang rejects this. Instead, it implements a computational form of **logical pluralism** — the idea that there are many legitimate logics, each suited to different circumstances.

This chapter formalizes the pluralistic philosophy underlying HaackLang and shows how it becomes executable code.

3.1 What Is Logical Pluralism?

Logical pluralism is the position that:

- No single logic is universally correct.
- Multiple logics can be equally valid.
- Different problems require different inferential tools.

Haack argues that logic is not an absolute system but a **family of systems** designed to model different aspects of reasoning.

Applied to computation:

- Calculations → classical logic
- Perception → fuzzy logic
- Contradictory inference → paraconsistent logic
- Probability estimation → probabilistic logic
- Planning and imagination → modal logics

HaackLang's Tracks make this pluralism operational.

3.2 Polylogical Pragmatism

Haack advocates a pragmatic approach: use the logic that works. HaackLang encodes this directly into its syntax.

Example:

```
track main   period 1 using classical
track slow   period 4 using fuzzy
track syncop period 7 using paraconsistent
```

Each track represents a separate logical world.

main — fast, crisp evaluation (perception & immediate decisions)
slow — gradual reasoning (belief revision & reflection)
syncop — chaotic and contradictory reasoning (intuition & subconscious)

A single proposition evaluated across these tracks forms a **polylogical truth vector**.

3.3 Why a Single Logic Cannot Model Minds

Human reasoning shows features incompatible with any one logic:

- Contradictions without collapse → requires paraconsistency
- Degrees of belief → requires fuzzy or probabilistic logic
- Hypotheticals and counterfactuals → requires modal logic
- Time-evolving thought → requires temporal logic
- Self-reference → requires meta-logic

HaackLang borrows each logic where appropriate.

A single-track logic would either:

- crash on contradictions, or
- fail to represent ambiguity, or
- freeze reflective thought into instantaneous updates, or
- flatten the temporal dynamics of belief.

Minds are not monologic. Computation shouldn't be either.

3.4 Pluralistic Execution: Running Multiple Logics at Once

The most radical element of HaackLang is that **all tracks run simultaneously**.

A truth declaration:

```
truthvalue fear = 0.6
```

automatically expands into:

```
fear.main   = 0.6 (classical or fuzzy)
fear.slow   = 0.6 (reflective logic)
fear.syncop = 0.6 (paraconsistent)
```

Each evolves independently.

This enables:

- emotional lag
- subconscious contradiction
- rational override
- cognitive dissonance
- reflective revision
- emergent oscillations
- multi-scale reasoning

This is not simulation — it is computation modeled after reasoning.

3.5 Logic Assignment in Code

In HaackLang, logics are first-class citizens.

Assign a logic to a track:

```
track slow period 4 using fuzzy
```

Assign a logic to a block:

```
logic paraconsistent {
    evaluate_paradox()
}
```

Assign a logic to an entire context:

```
context introspection {
    using logic fuzzy
    using track slow
}
```

Switch logics dynamically:

```
if @meta(confusion) > 0.7 {
    use logic paraconsistent
}
```

These constructs implement Haack's claim that logic should adapt to purpose.

3.6 Pluralistic Evaluation: The Polylogical Equation

Given proposition (p), and set of tracks:

[p = { p_{t_1}, p_{t_2}, ..., p_{t_n} }]

Each track has its own:

- logic function (L_t)
- update period (P_t)
- temporal history (H_t)

The overall system's truth-value is not a number — it is a **vector**.

Cross-track evaluation rules determine how combined expressions behave:

- Classical tracks → crisp boolean operations
- Fuzzy tracks → infimum/supremum or t-norms
- Paraconsistent tracks → contradiction-preserving conjunctions

This gives HaackLang unprecedented flexibility.

3.7 Pragmatic Resolution of Conflicts

Conflicts between tracks are expected.

HaackLang handles them via:

- meta-logic evaluation
- track priority rules
- temporal rhythm dominance
- context precedence
- resolution strategies (fuzzy blend, paraconsistent preserve, classical snap)

This mirrors how humans reconcile competing reasoning systems.

3.8 Logical Pluralism as a Computational Advantage

Pluralism isn't a weakness or compromise — it is a computational superpower.

It enables HaackLang to model:

- continuous reasoning
- reflective reasoning
- contradictory reasoning
- instinctive reasoning
- social reasoning
- probabilistic reasoning
- ethical reasoning
- creative leaps

All within the same language.

The polylogical architecture creates emergent properties that no single-logic language can replicate.

Ready for **Chapter 4 — Beyond True and False: The Case for Rhythmic Truth**?

Chapter 4 — Beyond True and False: The Case for Rhythmic Truth

Traditional programming languages treat truth as a single, static value — typically a Boolean. But cognition, perception, belief, intuition, emotion, and reflection all demonstrate that truth is not singular, static, or instantaneous.

Truth in human reasoning is **dynamic**, **multi-scale**, **contextual**, and often **asynchronous**. HaackLang formalizes this insight into a computational construct: **Rhythmic Truth**.

In this chapter, we define why truth must be rhythmic, how truth behaves across time and contexts, and how HaackLang implements truth as a structured temporal waveform rather than a binary state.

4.1 Truth as a Signal, Not a State

In classical logic, truth is a state:

```
true
false
```

But in real cognition, truth is a **signal evolving over time**. Consider how humans update beliefs:

- Perception updates instantly
- Emotions update slowly
- Habits update very slowly
- Intuition spikes unpredictably
- Beliefs update reluctantly
- Conviction solidifies over repetition
- Doubt oscillates depending on context

These processes unfold in **independent temporal rhythms**.

Truth is not a single value — it is a *vector changing at different speeds*.

HaackLang encodes this through its multi-track system, where each track updates at its own tempo.

4.2 The Failure of Static Truth in Modeling Minds

Static truth cannot model phenomena like:

- hesitation
- conflicted beliefs
- uncertainty
- doubt cycles
- partial trust
- emotional override
- delayed acceptance
- intuition spikes
- cognitive dissonance
- subconscious influence

These are not programming errors — they are features of thinking.

A static truth model cannot:

- oscillate
- drift
- spike
- fall
- syncopate
- desynchronize
- converge or diverge dynamically

HaackLang introduces **rhythmic truth** to bring computation closer to cognition.

4.3 Enter Rhythmic Truth

Rhythmic Truth means:

1. **Truth is multi-valued** (one value per track)
2. **Truth updates at different tempos**
3. **Truth may contradict itself across tracks**
4. **Truth evolves as a waveform, not a static bit**

A Boolean in HaackLang is actually a **BoolRhythm**:

```
t = {
    main:   0.9,   # fast track
    slow:   0.7,   # reflective track
    syncop: 0.3    # paraconsistent track
}
```

This single truth-value has:

- Crisp confidence on the main track
- Gradual belief on the slow track
- Intuitive hesitation on the syncopated track

A mind-like pattern emerges.

4.4 Why Truth Must Oscillate

Human truth oscillates for many reasons:

- second thoughts
- emotional instability
- repeated exposure
- cognitive dissonance
- reevaluation
- changing context

HaackLang models these oscillations using:

- periodic tracks
- syncopated tracks
- cross-track interference

Truth becomes a **temporal function** rather than a constant.

4.5 The Mathematics of Rhythmic Truth

Let (p) be a proposition.

Its truth in HaackLang is: [p(au) = [p_{t_1}(au), p_{t_2}(au), ..., p_{t_n}(au)]] Where:

- (au) is the global tick
- (t_i) are tracks with independent update rules

When track (t_i) does not fire at tick (au): [p_{t_i}(au+1) = p_{t_i}(au)] When it fires: [p_{t_i}(au+1) = L_{t_i}(expression)] Where (L_{t_i}) is the logic attached to that track.

This creates a **polyrhythmic update pattern**.

4.6 Cross-Rhythm Interference

When multiple tracks update at different rates, they create interference patterns.

Example:

- `main` (period 1)
- `slow` (period 4)
- `syncop` (period 7)

The combined update patterns produce:

- cognitive drift
- emergent oscillations
- sudden jumps in belief
- periodic reinforcement
- delayed convergence
- chaotic but meaningful intuition patterns

This is a computational analog of:

- rumination
- intuitive insight
- mounting fear
- gradual trust-building
- emotional turbulence

These emergent patterns are crucial for AGI-like behavior.

4.7 Rhythmic Truth in Practice

Example:

```
truthvalue danger = 0.5

if danger.main > 0.8 {
    run_away()
}

if danger.slow > 0.8 {
    reassess_environment()
}

if danger.syncop > 0.8 {
    panic()
}
```

The same proposition can:

- trigger instinct
- trigger reflection
- trigger panic

at different times.

This flexibility reflects how humans respond to danger.

4.8 Interpreting Rhythmic Truth as a Cognitive Field

Rhythmic truth allows each proposition to behave like a node in a temporal field.

This field:

- flows
- pulses
- resonates
- stabilizes or destabilizes
- creates emergent patterns
- supports multiple modes of reasoning

Truth becomes a *dynamic entity*.

This is why HaackLang is capable of modeling:

- emotional reasoning
- contradictory intuitions
- slow belief revision
- subconscious influences
- reflective self-correction

In a way neither traditional logic nor traditional programming languages can.

Ready for **Chapter 5 — Fuzzy, Paraconsistent, and Classical Logics as Coexisting Modes of Thought**?

Chapter 5 — Fuzzy, Paraconsistent, and Classical Logics as Coexisting Modes of Thought

Human cognition does not operate under a single logic. We reason with crisp judgments, vague impressions, contradictory intuitions, and probabilistic

expectations — often simultaneously. Susan Haack's pluralistic insights provide the conceptual grounding for HaackLang's logic model, where classical, fuzzy, and paraconsistent logics coexist in a unified computational framework.

This chapter explains why these logics must coexist, what each contributes, and how HaackLang integrates them seamlessly through Tracks and Contexts.

5.1 Classical Logic: The Logic of Crisp Decisions

Classical logic is the logic of mathematics, digital circuits, and unambiguous propositions. It assumes:

- **bivalence**: every statement is true or false
- **law of non-contradiction**: nothing can be both true and false
- **law of excluded middle**: everything is either true or false

What classical logic models well:

- immediate perception ("Is the light on?")
- discrete conditions ("Is health < 0?")
- deterministic computations
- exact rule-based inference

What it fails to model:

- partial truth
- uncertainty
- tension between beliefs
- contradictory conclusions
- emotional/intuitive reasoning

In HaackLang, classical logic is typically attached to **fast tracks** (e.g., `main`), representing crisp, immediate reasoning.

Example:

```
track main period 1 using classical
```

This gives fast perception and decision-making classical crispness.

5.2 Fuzzy Logic: The Logic of Gradients and Degrees

Fuzzy logic allows truth to take any value in the continuous interval:

```
0.0 ≤ truth ≤ 1.0
```

This models:

- partial beliefs ("probably true")
- emotional intensity ("very angry")
- uncertainty
- likelihood estimation
- vague categories

Fuzzy logic is essential for:

- belief revision
- emotional cognition
- gradual acceptance or rejection of propositions
- probabilistic reasoning

In HaackLang, fuzzy logic typically governs **slow tracks**, allowing gradual updates.

Example:

```
track slow period 4 using fuzzy
```

This models belief stability and slow cognitive revision.

5.3 Paraconsistent Logic: The Logic of Contradiction Without Collapse

Classical logic cannot tolerate contradictions:

- If both A and ¬A are true, classical logic collapses

Paraconsistent logic avoids this collapse, allowing contradictory statements to coexist safely.

Why minds need paraconsistency:

- people hold conflicting beliefs
- emotions contradict facts
- intuition contradicts analysis
- cognitive dissonance exists
- memories conflict

Paraconsistent logic is ideal for subconscious or intuitive processes, which may:

- register contradictory signals
- allow tension between incompatible ideas
- resolve contradictions later

In HaackLang, paraconsistent logic usually governs **syncopated tracks**.

Example:

```
track syncop period 7 using paraconsistent
```

This track captures intuition, gut feeling, and unresolved contradictions.

5.4 Why Coexistence Is Essential

Each logic type specializes in a distinct cognitive role:

Classical logic → fast, crisp reasoning

Used for:

- immediate perception
- quick decisions

- safety checks
- error detection

Fuzzy logic → slow, graded reasoning

Used for:

- belief revision
- evaluating evidence
- emotional intensity
- uncertainty calculations

Paraconsistent logic → contradictory or intuitive reasoning

Used for:

- gut feelings
- dream-like processing
- cognitive dissonance
- unintegrated insights

The combination mirrors human cognition:

- A person can believe something is *mostly true*, while still feeling unsure, while simultaneously holding an intuitive counter-belief.

5.5 Cross-Logic Interactions

The power of HaackLang lies in how these logics interact.

Classical → Fuzzy

Fast perceptions update slow beliefs.

```
if danger.main > 0.8 {
    danger.slow = (danger.slow + 0.8) / 2
}
```

Fuzzy → Paraconsistent

Gradual beliefs influence intuition.

```
if danger.slow > 0.6 {
    danger.syncop = max(danger.syncop, 0.4)
}
```

Paraconsistent → Classical

Unresolved intuition occasionally triggers crisp decisions.

```
if danger.syncop > 0.9 {
    alarm.main = 1.0
}
```

These interactions produce multi-layered cognition.

5.6 Temporal Separation Enforces Logical Coherence

Tracks update at different tempos:

- classical → fast
- fuzzy → medium
- paraconsistent → slow or irregular

This separation prevents logical interference.

Example:

- immediate perception might spike a value on `main`
- slow reflective reasoning will update later
- intuition may oscillate unpredictably

This architecture mimics how thoughts evolve in different layers of mind.

5.7 Logical Coexistence in a Single Variable

A single truthvalue in HaackLang inherently contains multiple logical interpretations:

```
truthvalue trust = {
    main:   1.0,  # classical
    slow:   0.6,  # fuzzy
    syncop: 0.2   # paraconsistent
}
```

This reflects a psychological truth:

- I trust you logically (main = 1.0)
- I have lingering doubts (slow = 0.6)
- Something feels off (syncop = 0.2)

No other language can represent this complexity in a single variable.

5.8 Why HaackLang Treats Logics as Tools, Not Absolutes

Haack's philosophy is fully realized here:

- no logic is supreme
- no logic is universal
- each logic has a specific functional domain
- reasoning emerges from the cooperation of many logics

This is a **computational philosophy**, not just an implementation. HaackLang turns logical pluralism into executable structure.

HaackLang Language Specification — Volume II

Chapters 6–15

This document will contain the full technical and formal specification chapters of HaackLang, beginning with Chapter 6.

Please say "**Begin Chapter 6**" to start populating the volume.

Chapter 6 — Core Semantics: Tracks, Beats, and Polylogical Clocks

HaackLang introduces a new foundational semantic paradigm built on **temporal plurality** and **logical plurality**. Instead of assuming a single universal update cycle and a single universal logic, HaackLang defines program execution as a *polyrhythmic, polylogical system* where different components of truth evolve at different tempos, following different logics. This chapter formalizes the semantic machinery underlying that model.

6.1 Tracks: Independent Logical Timelines

A **Track** is the fundamental temporal unit of HaackLang. Each Track is defined by:

- **A name** (e.g., `main`, `slow`, `syncop`)
- **A period** (how many global ticks pass between updates)
- **A logic** (the inference system used on that Track)

Example:

```
track main   period 1 using classical
track slow   period 4 using fuzzy
track syncop period 7 using paraconsistent
```

Each Track behaves as its own "logical universe," updating propositions according to its rhythm and logic.

Tracks enable:

- fast reasoning (e.g., perception)
- slow reasoning (e.g., reflection)
- irregular reasoning (e.g., intuition)

No other language encodes temporal plurality at the logical level.

6.2 Beats: Track-Specific Update Pulses

A **Beat** is the moment at which a Track updates its values.

A Track fires on a Beat whenever:

```
tick % period == 0
```

Where `tick` is the global clock.

Examples:

- `main` fires every tick (period 1)
- `slow` fires every 4 ticks (period 4)
- `syncop` fires every 7 ticks (period 7)

HaackLang programs produce **temporal interference patterns** as Track beats overlap, drift, and realign.

6.3 The Global Tick

HaackLang operates on a single global clock:

```
global_tick += 1
```

Each global tick causes:

1. Tracks to determine whether they fire
2. Firing tracks to update truth-values using their logic
3. Non-firing tracks to freeze their values
4. Control flows to evaluate according to track-specific rhythms

Temporal plurality does *not* mean temporal chaos — the global tick ensures orderly execution.

6.4 Truth as a Multi-Track Vector

A truthvalue in HaackLang is not a single scalar. It is a **vector of truth-values**, one per Track.

Example:

```
truthvalue fear = 0.5
```

Internally expands to:

```
fear.main   = 0.5
fear.slow   = 0.5
fear.syncop = 0.5
```

If two Tracks update at different times, they produce:

- **temporal divergence** (values drifting apart)
- **temporal convergence** (values syncing back together)
- **oscillation** (wave-like truth evolution)

This reflects human cognition more accurately than flat Booleans.

6.5 Freeze-on-No-Beat Semantics

The core semantic rule of HaackLang:

> **If a Track does not fire on a given tick, its values do not change.**

Formally:

```
p.track[t].next = p.track[t].current
```

This preserves:

- memory
- emotional inertia
- slow belief revision
- delayed intuition
- reflective lag

Without freeze semantics, HaackLang would collapse into synchronous Boolean logic.

6.6 Updating Truth on a Track Beat

If a Track **does** fire:

1. It applies its logical system to update values.
2. It propagates the updated value into shared truthvectors.

Example using classical logic:

```
fear.main = min(fear.main, threat.main)
```

Example using fuzzy logic:

```
fear.slow = fear.slow * 0.8 + stimulus.slow * 0.2
```

Example using paraconsistent logic:

```
fear.syncop = paradox_resolve(fear.syncop, intuition.syncop)
```

Each Track uses a different rule set.

6.7 Inter-Track Referencing and Temporal Pulling

Expressions can reference truth-values from Tracks that did not fire this tick.

Example:

```
if fear.slow > 0.7 && threat.main > 0.9 {
    freeze()
}
```

Here:

- `threat.main` is fresh (fast Track)
- `fear.slow` might be stale (slow Track)

This models:

- fast instincts colliding with old beliefs
- emotional inertia
- reflective lag

The HaackLang interpreter retrieves the most recently updated value when pulling from a slower Track.

6.8 Polyrhythmic Interference

When Track periods differ, they create interference patterns.

Example Track periods:

- `main` : 1
- `slow` : 4
- `syncop` : 7

Their least common multiple (LCM = 28) is the "super-cycle," where all Tracks realign.

Within a super-cycle, Tracks:

- beat together
- beat separately
- synchronize unpredictably

This creates:

- oscillations
- stable attractors
- chaotic intuition spikes
- delayed emotional convergence
- emergent subconscious loops

Such behavior is impossible in classical languages.

6.9 Summary of HaackLang Core Semantics

HaackLang's semantics combine:

- **temporal plurality** → Tracks with unique update tempos
- **logical plurality** → each Track uses its own logic
- **dynamic truth-values** → multi-track truthvectors evolving asynchronously
- **interference patterns** → emergent behavior from rhythm collisions

This produces a computational model capable of representing:

- human-like reasoning
- AGI cognition loops
- emotional/subconscious dynamics
- reflective and intuitive processes
- contradictory or oscillatory beliefs

Ready for **Chapter 7 — BoolRhythm: A Multi-Timed, Multi-Valued Truth Type**?

Chapter 7 — BoolRhythm: A Multi-Timed, Multi-Valued Truth Type

BoolRhythm is the foundational truth type of HaackLang. It replaces the classical Boolean with a **temporal, multi-valued, track-indexed truth structure** capable of representing perception, belief, intuition, hesitation, contradiction, and emotional drift — all within a single variable.

Unlike a Boolean, a BoolRhythm is not a scalar. It is a **truthvector**, where each component corresponds to a logical Track with its own tempo and inference rules.

7.1 Definition of BoolRhythm

A BoolRhythm is defined as:

- a mapping from Track names to truth-values
- stored in a synchronized truthvector
- updated asynchronously according to Track beats

Example declaration:

```
tv danger = 0.6
```

Internally expands to something like:

```
danger = {
    main:   0.6,
    slow:   0.6,
    syncop: 0.6
}
```

Each Track updates independently.

7.2 Truthvector Structure

BoolRhythm truthvectors have these characteristics:

- each Track stores its own truth-value
- each truth-value is a floating number between 0.0 and 1.0
- all Tracks share the same variable name
- each Track inherits the initial value at declaration
- subsequent updates diverge across Tracks

Example expanded truthvector:

```
fear = {
    main:   0.9,
    slow:   0.4,
    syncop: 0.7
}
```

This models:

- fast perception (main)
- slow belief revision (slow)
- intuitive or contradictory affect (syncop)

7.3 Updating Rules for BoolRhythm

BoolRhythm updates follow Track rhythms.

Rule 1: If a Track fires on this tick, update using its logic (classical, fuzzy, paraconsistent).

Rule 2: If a Track does not fire, freeze its value.

Rule 3: Expressions referencing BoolRhythm values pull the latest value per Track.

Rule 4: Cross-track references never force synchronization.

Example:

```
if threat.main > 0.8 {
    fear.main = min(fear.main, threat.main)
}
```

7.4 Cross-Track Truth Calculus

Operations on BoolRhythm apply component-wise across Tracks.

Example conjunction:

```
danger.main   = min(danger.main,   fear.main)
danger.slow   = min(danger.slow,   fear.slow)
danger.syncop = paradox_and(danger.syncop, fear.syncop)
```

Each Track uses its own definition of the operator.

Classical conjunction → crisp minimum Fuzzy conjunction → t-norm or weighted blend Paraconsistent conjunction → contradiction-tolerant blend

7.5 Default Initialization

When writing:

```
tv trust = 1.0
```

All Tracks inherit the same initial value. Later, as Tracks fire at different rates, values drift.

Example after 20 ticks:

```
trust = {
    main:   1.0,
    slow:   0.75,
    syncop: 0.33
}
```

This single variable captures:

- certainty
- hesitation
- subconscious doubt

7.6 Temporal Divergence and Convergence

Because Tracks update at different tempos:

- values drift apart
- values re-sync occasionally
- rhythms create oscillatory patterns

This models:

- rumination
- emotional momentum
- intuitive spikes
- fast vs slow thinking

Example pattern over time:

```
tick 0: fear = {1.0, 1.0, 1.0}
tick 4: fear = {0.9, 0.7, 1.0}
tick 7: fear = {0.8, 0.7, 0.4}
tick 28: fear re-syncs as Tracks align
```

7.7 Access Syntax

BoolRhythm values are accessed via:

```
var.track
```

Example:

```
if trust.slow < 0.5 {
    reassess()
}
```

You may also access the entire truthvector at once:

```
inspect trust
```

This is primarily for debugging and visualization.

7.8 Type Interactions

BoolRhythm interacts with:

- numeric types
- probability types
- context guards
- flow control
- pattern matching

Example coercion to numeric:

```
energy = fear.main * 10
```

Example fuzzy guard:

```
guard slow danger.slow > 0.6 {
    step retreat
}
```

7.9 BoolRhythm Operators

Operators apply per Track.

Supported operators:

- and
- or
- not
- xor
- implies
- equivalent
- blend

Examples:

```
danger = fear and threat
hesitation = not confidence
risk = danger blend uncertainty
```

Blend is unique to HaackLang, enabling weighted mixing:

```
risk.slow = blend(fear.slow, doubt.slow, weight=0.3)
```

7.10 BoolRhythm as a Cognitive Primitive

BoolRhythm is not just a data type. It is:

- a model of mind-like truth dynamics
- the key to emergent cognition
- the enabler of multi-track reasoning
- the basis of temporal and emotional modeling

It makes HaackLang capable of expressing:

- belief drift
- emotional inertia
- intuitive contradiction
- subconscious spikes
- cross-tempo interference
- oscillatory truth

This elevates truth to a dynamic object.

Ready for **Chapter 8 — Expressions, Guards, and Rhythmic Evaluation**?

Chapter 8 — Expressions, Guards, and Rhythmic Evaluation

Expressions in HaackLang are not evaluated under a single universal logic or a single universal time. Instead, every expression inherits the temporal and logical structure of the Tracks on which it is evaluated. This chapter formally defines expression syntax, guard semantics, cross-track evaluation rules, rhythmic precedence, and the interplay between logic and tempo.

HaackLang expressions are designed to model *cognitive processes*, where different components of thought update at different speeds, follow different logics, and may not be mutually synchronized.

8.1 Expression Grammar

HaackLang expressions follow an extended infix grammar, enriched with:

- track-qualified variables (`fear.slow`)
- multi-track variables (`fear`)
- rhythmic operators
- logic-polymorphic operators
- guard-prefixed expressions

Base EBNF Form

```
expr ::= value
       | identifier
       | identifier '.' track
       | expr op expr
       | '(' expr ')'
       | call '(' arglist ')'
       | unary_op expr
       | expr '[' rhythm ]   // rhythmic evaluation
```

Examples

```
fear.main + courage.slow
(perception.main and belief.slow)
not doubt.syncop
blend(fear, trust, weight=0.4)
```

Expressions default to *per-track evaluation*: each operator applies according to the logic of that Track.

8.2 Multi-Track Expression Expansion

If an expression references a multi-track variable without specifying a Track, it expands automatically.

Example:

```
fear + courage
```

Expands internally into:

```
fear.main   + courage.main
fear.slow   + courage.slow
fear.syncop + courage.syncop
```

Each component is evaluated using that Track's logic.

This ensures:

- consistency across Tracks
- independent track-level semantics
- coherent multi-rhythm truth evolution

8.3 Logic-Polymorphic Operators

Operators in HaackLang are polymorphic: their semantics depend on the Track's logic.

Example: AND

- Classical: `min(a, b)` or boolean conjunction
- Fuzzy: t-norm (default: Gödel min, configurable)
- Paraconsistent: contradiction-preserving conjunction

Thus the expression:

```
fear and threat
```

expands into:

- classical AND on main track
- fuzzy AND on slow track
- paraconsistent AND on syncop track

This is an executable expression of logical pluralism.

8.4 Rhythmic Operators

HaackLang defines operators that directly manipulate rhythms.

1. hold

Freezes a value for a number of ticks.

```
x = hold(y, 3)
```

2. pulse

Triggers a value only on certain beats.

```
tv danger = pulse(threat, period=5)
```

3. sync

Forces synchronization across Tracks.

```
aligned = sync(fear)
```

4. echo

Propagates delayed values.

```
memory = echo(perception, delay=2)
```

5. interfere

Produces cross-rhythm interference patterns.

```
intuition = interfere(fear, doubt)
```

These are the core tools for crafting emergent temporal behavior.

8.5 Guards: Conditional Evaluation Across Tracks

Guards in HaackLang control when steps execute.

Basic Guard Syntax

```
guard <track> <condition> {
    step <name>
}
```

Examples:

```
guard main fear.main > 0.8 {
    step flee
}

guard slow trust.slow < 0.4 {
    step reevaluate
}
```

Guard semantics:

- the condition is evaluated *only* on the Track specified
- if the Track does not fire on this tick, the guard is skipped entirely
- guards can stack across Tracks to create multi-tempo control flow

8.6 Multi-Track Guards

Multi-track guards evaluate conditions across multiple tempos.

Syntax:

```
guard (main, slow) fear.main > fear.slow {
    step cognitive_dissonance
}
```

This guard executes only if **both Tracks fire on this tick**.

If their periods are co-prime (e.g., 1 and 4), this creates rare alignment events.

This is crucial for modeling:

- moments of clarity
- sudden insight
- synchronized emotional and reflective awareness

8.7 Rhythmic Precedence

HaackLang expressions are evaluated according to **temporal precedence** rather than just operator precedence.

Temporal precedence rules:

1. Evaluate all expressions on Tracks that fire this tick
2. Freeze all expressions on Tracks that do not fire
3. Resolve multi-track expressions last
4. Apply meta-logic after all per-track evaluation

This ordering models:

- fast instincts overriding slow beliefs
- old beliefs affecting new perceptions
- intuition firing irregularly

8.8 Short-Circuiting in Polylogical Contexts

HaackLang does *not* short-circuit like classical Boolean logic.

Example:

```
if fear and courage {
    act()
}
```

Even if `fear.main` is 0.0 (false), HaackLang still evaluates:

- `courage.main`
- `fear.slow` & `courage.slow`
- `fear.syncop` & `courage.syncop`

Because:

- different Tracks may have different truth-values
- slow Track may still recommend action
- syncop Track may contradict or reinforce

Short-circuiting would ignore valuable cognitive information.

8.9 Meta-Logic on Expressions

After per-track evaluation, HaackLang applies **meta-logic**, responsible for:

- resolving contradictions
- prioritizing Tracks
- collapsing multi-track truthvectors into single decisions if needed
- choosing which cognitive subsystem drives behavior

Example:

```
if @meta(danger) > 0.7 {
    alarm()
}
```

The `@meta` function computes an aggregate value using:

- weighted Track priorities
- context rules
- historical trends

8.10 Rhythmic Evaluation Summary

Expressions in HaackLang:

- evaluate independently per Track
- follow the logic assigned to each Track
- synchronize only when needed
- embrace temporal misalignment

- express cognitive conflict through rhythm
- propagate emergent behavior through beats

Guards extend this by controlling *when* expressions matter, based on tempo.

Together, these features make HaackLang's expression system:

- dynamic
- rhythmic
- pluralistic
- cognitively plausible
- capable of emergent reasoning

HaackLang Language Specification — Chapter 9

Contexts, Scopes, and Cognitive Domains

This document will contain the full text of Chapter 9.

Say "**Begin Chapter 9**" to start writing.

Chapter 9 — Contexts, Scopes, and Cognitive Domains

HaackLang treats reasoning not as a monolithic process but as a constellation of **cognitive domains**, each with its own logic, temporal rhythm, and behavioral rules. Contexts define these domains. They govern how truth-values evolve, how Tracks interact, which operators are active, and which subsystems dominate.

In human cognition, context shapes reasoning:

- The same information produces different conclusions in different emotional states.

- Perception operates under different rules than reflection.
- Intuition contradicts analysis yet both influence behavior.
- Moral reasoning differs from sensorimotor response.

HaackLang formalizes this through **Contexts**, which serve as logic-temporal bubbles governing interpretation.

9.1 What Is a Context?

A **Context** is a declarative block that defines:

- which logics apply
- which Tracks are active
- how truthvectors are interpreted
- how meta-logic resolves conflicts
- the semantics of reasoning within that domain

The simplest form:

```
context perception {
    using track main
    using logic classical
}
```

Contexts are first-class computational environments.

9.2 Context Structure

A context block includes:

- declared Tracks (temporal rhythms)
- assigned logics (reasoning modes)
- context-level variables
- scope resolution rules
- meta-logic policies
- entry and exit hooks

Example:

```
context reflection {
    using track slow
    using logic fuzzy
    priority slow > main

    tv confidence = 0.7

    rule drift {
        confidence.slow = confidence.slow * 0.95
    }
}
```

This creates a domain where:

- `slow` Track dominates
- fuzzy logic governs truth
- confidence drifts slowly

9.3 Contextual Logic Assignment

Each Context may define:

- a single logic for all Tracks
- different logics per Track
- logic overrides for specific variables

Examples:

One logic for all Tracks

```
context cautious {
    using logic fuzzy
}
```

Different logics per Track

```
context cognition {
    using track main using classical
    using track slow using fuzzy
    using track syncop using paraconsistent
}
```

Variable-specific logic override

```
context paradox_zone {
    override logic fear.syncop = paraconsistent
}
```

Logic assignment reflects domains of inference.

9.4 Temporal Inheritance in Contexts

Contexts inherit the global Track list unless overridden.

Example:

```
context intuition {
    using track syncop
}
```

Here, `syncop` fires irregularly, creating:

- nonlinear progression
- sudden insight spikes
- unpredictable evolution

If not overridden, other Tracks still exist — they just do not dominate.

9.5 Scope Hierarchies

HaackLang scopes follow a three-level hierarchy:

1. **Local scope** (inside functions, rules, steps)

2. **Context scope** (variables, rules declared in context)
3. **Global scope** (shared resources, Tracks, meta-rules)

Resolution order:

```
local → context → global
```

Example:

```
context planning {
    tv risk = 0.5

    rule adjust {
        let risk = 0.1   # local
        risk.main = 0.2  # modifies context-level
    }
}
```

Local `risk` shadows context `risk` unless `.main` is specified.

This models how humans hold private thoughts inside a broader reflective domain.

9.6 Nested Contexts

Contexts may nest.

Example:

```
context combat {
    using track main
    using logic classical

    context panic {
        using track syncop
        override logic fear.syncop = paraconsistent
    }
}
```

Nested contexts override the parent's temporal and logical settings.

Nested contexts model:

- sudden emotional takeover
- domain shifts
- local reasoning bubbles

9.7 Dynamic Context Switching

Contexts may be switched at runtime.

Example:

```
if fear.main > 0.9 {
    enter context panic
}
```

Context switching changes the rules of reasoning *without resetting truthvectors*.

This models:

- emotional collapse
- fight-or-flight
- heightened intuition
- cognitive narrowing
- reflective expansion

Exiting a context restores the previous domain.

9.8 Context Fusion

Two contexts may merge if conditions overlap.

Example:

```
fusion awareness = perception + intuition
```

The resulting context has:

- union of Tracks
- blended logics
- merged variables
- fused meta-rules

Fusion is useful for modeling:

- clarity moments
- integrative insights
- cross-domain coordination

9.9 Context-Level Meta-Logic

Contexts define how conflicts between Tracks are resolved.

Example:

```
context diplomacy {
    using logic fuzzy
    priority slow > main

    meta resolve {
        decision = weighted(fear, trust, slow_weight=0.8)
    }
}
```

Meta-logic may:

- weight Tracks
- enforce coherence
- maintain contradictions
- discard contradictions
- blend or filter truthvectors

This represents high-level reasoning strategies.

9.10 Cognitive Domains as Context Templates

Common cognitive contexts:

Perception

- `main` Track
- classical logic
- fast updates

Reflection

- `slow` Track
- fuzzy logic
- gradual drift rules

Intuition

- `syncop` Track
- paraconsistent logic
- irregular updates

Emotion

- amplitude-based updates
- nonlinear decay

Agency

- combines Tracks to generate action
- meta-logic determines dominance

These templates form the backbone of AGI-like architectures.

9.11 Why Contexts Matter

Contexts allow HaackLang to:

- model domain-specific reasoning

- simulate shifts in cognitive state
- constrain or enhance logic
- enforce temporal boundaries
- create emergent cognitive behaviors
- adapt truth evaluations dynamically

Contexts make HaackLang a **psychologically realistic** language.

HaackLang Language Specification — Chapter 10

Meta-Logic: Reasoning About Reasoning

This document will contain the full text of Chapter 10.

Say **"Begin Chapter 10"** to start writing.

Chapter 10 — Meta-Logic: Reasoning About Reasoning

Meta-logic is the crown jewel of HaackLang: the layer of computation where the system reason **about its own reasoning**. It governs cross-track interactions, evaluates logical conflicts, synthesizes divergent truthvectors, and determines how contexts communicate.

Unlike classical meta-logical formalisms, HaackLang's meta-logic is **temporal, pluralistic, and rhythm-sensitive**. Its role is not merely to assign truth, but to orchestrate **coherence**, **conflict tolerance**, **logical switching**, and **cognitive governance**.

This chapter defines the principles, syntax, semantics, and evaluative algorithms of HaackLang’s meta-logic layer.

10.1 What Is Meta-Logic in HaackLang?

In classical logic, meta-logic evaluates statements *about* logical systems. In HaackLang, meta-logic does far more:

- evaluates relationships between Tracks
- resolves contradictions across logics
- selects dominant reasoning modes
- smooths or amplifies truthvector oscillations
- governs context switching
- manages cognitive coherence
- evaluates global reasoning quality

Meta-logic is the **governance layer**: the mind's internal conductor.

In HaackLang syntax:

```
meta resolve {
    # meta-logical rules
}
```

10.2 The Purpose of Meta-Logic

Meta-logic governs cognitive integrative functions:

1. **Conflict Resolution**
   Determines how contradictions between Tracks are handled.

2. **Track Prioritization**
   Chooses which Track has final authority in a given moment.

3. **Logic Switching**
   Changes a Track's logic if the current one yields incoherence.

4. **Coherence Optimization**
   Tunes truthvectors toward more stable cognitive configurations.

5. **Context Governance**
   Decides which context should be active given global conditions.

6. **Emergent Dynamics**
   Enables patterns like rumination, insight, panic, conviction.

Meta-logic manages *the shape of reasoning*.

10.3 The Meta-Logic Cycle (Beat-Level Execution)

Meta-logic fires at the **meta-beat** — a global rhythm occurring less frequently than any Track.

Example:

```
meta-beat 16
```

Meaning: every 16 global ticks, meta-logic executes.

During the meta-beat, it:

1. samples current truthvectors
2. evaluates cross-track coherence
3. resolves contradictions
4. applies stabilizing or destabilizing transformations
5. updates context priorities
6. optionally modifies Track update rules

This is analogous to:

- reflective thought
- self-awareness
- system-level introspection

10.4 Meta-Logic Syntax

A meta-logic block has the following form:

```
meta resolve {
    rule_name(args) -> action
```

```
    rule_name2(args) -> action2
}
```

Rules can:

- evaluate conditions
- compute coherence indices
- modify truthvectors
- switch contexts
- reprioritize Tracks
- apply fuzzy blends or paraconsistent preserves

Example:

```
meta resolve {
    if coherence < 0.4 {
        enter context stabilize
    }
}
```

10.5 Meta-Operators

HaackLang introduces special operators for meta-logic.

10.5.1 @coh(x) — Coherence Evaluation

Computes coherence of truthvector `x`.

```
if @coh(fear) < 0.3 {
    fear.slow = fear.main
}
```

10.5.2 @blend(a, b, w) — Weighted Fusion

Blends two truthvectors with weight `w`.

```
trust = @blend(trust, hope, 0.2)
```

10.5.3 @resolve(a) — Resolve Contradictions

Applies paraconsistent decomposition or classical snapping.

```
belief = @resolve(belief)
```

10.5.4 @dom(track) — Track Dominance

Returns dominance score of Track.

```
if @dom(syncop) > 0.8 {
    enter context intuition
}
```

10.5.5 @meta(var) — Meta-Value of a Variable

Returns global/meta significance of variable (e.g., emotional intensity).

```
if @meta(stress) > 0.7 {
    use logic paraconsistent
}
```

10.6 Meta-Logic and Contradiction Management

Meta-logic provides the system's central mechanism for dealing with contradictions.

Contradictions occur when:

- a truthvector diverges across Tracks
- two contexts disagree
- two logics assign incompatible results
- temporal rhythms produce conflict

Meta-logic determines whether to:

- preserve contradiction (paraconsistent)
- weaken contradiction (fuzzy blend)

- collapse to classical crispness
- escalate contradiction (for insight/panic)

Example:

```
if fear.main == 1.0 and fear.syncop == 0.0 {
    fear.slow = @blend(fear.main, fear.syncop, 0.4)
}
```

10.7 Cross-Track Coherence Functions

HaackLang introduces a coherence metric:

[C(x) = w_1 x_1 + w_2 x_2 + ... + w_n x_n]

Where `x_i` is Track truthvalue and `w_i` its track priority weight.

Example:

```
meta resolve {
    if @coh(trust) < 0.5 {
        trust.slow = trust.main * 0.6
    }
}
```

Advanced coherence functions may consider:

- oscillation amplitude
- temporal derivatives
- cross-track phase alignment

These create meta-level evaluations resembling reflective insight.

10.8 Meta-Logic and Context Switching

Meta-logic decides when the system moves between cognitive domains.

Example: Panic Trigger

```
if @meta(fear) > 0.8 and @dom(syncop) > 0.6 {
    enter context panic
}
```

Example: Return to Reason

```
if @coh(fear) > 0.6 {
    exit context panic
}
```

This enables:

- emotional takeover
- reflective recovery
- intuition bursts
- cognitive shifts

10.9 Meta-Logic Governs Track Priorities

Tracks may be reprioritized dynamically.

Example:

```
meta resolve {
    if stress.main > 0.9 {
        promote syncop
        suppress slow
    }
}
```

Promoting a Track means:

- more frequent updates
- higher blend weight
- increased influence on decisions

Suppressing means the opposite.

This models fight-or-flight, where fast and intuitive Tracks override reflective ones.

10.10 Meta-Logic as Cognitive Governance

Meta-logic serves as:

- **arbitrator** (resolves conflicts)
- **conductor** (orchestrates tracks)
- **thermostat** (maintains coherence)
- **governor** (switches contexts)
- **overseer** (evaluates global reasoning)

It is the computational analog of executive function.

10.11 The Meta-Beat: Temporal Rhythm of Self-Awareness

Meta-logic runs at a slower rhythm than Tracks.

This allows:

- cycles of reflection
- insight generation
- belief revision
- cognitive rebalancing

Meta-beat stability determines:

- coherence
- reasoning quality
- susceptibility to contradiction

A system may modify its own meta-beat interval based on internal conditions.

10.12 Meta-Logic as the Engine of Self-Modification

Meta-logic is the foundation for future chapters on:

- self-modifying code
- evolving track structures
- context evolution
- adaptive logical systems

It will also underpin HaackLang's support for:

- reflective optimization
- dynamic logic switching
- recursive self-evaluation

HaackLang Language Specification — Chapter 11

The Meta-Logical Machinery: Operators, Rules, and Evaluation Models

Chapter 11 — The Meta-Logical Machinery: Operators, Rules, and Evaluation Models

Meta-logic is where HaackLang becomes truly Haackian. If Tracks provide the temporal scaffolding and Contexts provide the cognitive domain boundaries, then **meta-logic provides the machinery by which HaackLang reasons about its own reasoning**.

In the tradition of Susan Haack's logical pluralism, meta-logic in HaackLang is *pragmatic without being anarchic*, *structured without being authoritarian*, and

dynamic without being chaotic. It sits above the Tracks and orchestrates when, how, and why each logic participates.

This chapter defines:

- the full meta-logical operator set
- declarative and procedural meta-rules
- the meta-evaluation model
- cross-track inference and interference
- update rule hierarchy
- the meta-logical cycle (“beat resolution”)
- interpreter-level semantics for meta-logic

11.1 What Meta-Logic Is and Why HaackLang Needs It

In classical programming languages, logic is static and unquestioned. In HaackLang, logic itself becomes fluid and adaptive, so a **higher-order system is required to regulate the logics**.

Meta-logic determines:

- when classical logic dominates
- when fuzzy logic modulates belief
- when paraconsistent logic preserves contradictions
- how conflicts between tracks are resolved
- how truth-values propagate between tracks
- how contexts negotiate their logical constraints
- when to switch logics dynamically

Without meta-logic, logical pluralism collapses into anarchy. With meta-logic, pluralism becomes *orchestrated polylogical reasoning*.

11.2 The Meta-Logical Stack

Meta-logic operates across four levels:

Level 0: Track Logic

The raw logic of each Track (classical, fuzzy, paraconsistent).

Level 1: Context Logic

Context-level override rules:

```
context planning using fuzzy
context introspection using paraconsistent
```

Level 2: Cross-Track Mediation

Meta-logic mediates:

- interference
- weighting
- resonance
- beat harmonization
- conflict resolution

Level 3: Meta-Context Logic

Reasoning *about* context–track relations. Example:

```
if @meta(conflict(perception, intuition)) > 0.6
    boost track slow
```

11.3 Meta-Logical Operators

Meta-logic provides a dedicated operator alphabet for inspecting, manipulating, and reasoning about logical processes.

11.3.1 Structural Meta-Operators

These operators inspect the state of Tracks and Contexts.

Operator	Meaning
`@meta(track)`	access track-level diagnostics
`@meta(context)`	access context-level diagnostics
`@meta(history(x))`	fetch temporal truth history
`@meta(period(t))`	get update period for track t
`@meta(phase(t))`	get phase offset
`@meta(beat)`	return the current global beat

Examples:

```
if @meta(period(slow)) > 3 {
    adjust_trust_level()
}
```

11.3.2 Interference Operators

These model cross-track effects analogous to wave interactions.

Operator	Description
⊕	constructive interference
⊖	destructive interference
⊗	resonance amplification
⊘	damping/suppression

Example:

```
emotion = fear ⊕ memory.trace("wolves")
confidence = resolve ⊘ anxiety
```

11.3.3 Conflict and Coherence Operators

Operator	Meaning
≃	coherence match
≄	coherence mismatch
↯	contradiction spike
↺	reflection loop
⇡	coherence ascent
⇣	coherence descent

Example:

```
if danger.main ≃ danger.slow ⇡ {
    commit_to_course_of_action()
}
```

11.3.4 Meta-Logical Control Operators

These allow the programmer to manipulate reasoning itself.

Operator	Purpose
`meta::boost(track)`	increase weight of a track
`meta::dampen(track)`	decrease weight
`meta::switch_logic(t, L)`	change logic for track t
`meta::freeze(t)`	pause updates
`meta::thaw(t)`	resume updates
`meta::reset(x)`	reset a truth rhythm

Example:

```
if @meta(conflict(main, syncop)) > 0.8 {
    meta::boost(slow)
}
```

11.4 Meta-Rules

Meta-rules govern how meta-logic evaluates and resolves situations.

11.4.1 Structural Meta-Rules

These rules describe how truth propagates across tracks.

Rule: Fast-to-Slow Propagation

```
main influences slow on every 4th beat
slow influences syncop on every 7th beat
```

Rule: Cross-Track Persistence

```
Track values persist until next update window
```

11.4.2 Conflict Resolution Rules

A core feature of HaackLang.

Hierarchy of conflict resolution:

1. **Context Priority**
2. **Track Period Dominance** (shorter period dominates real-time decisions)
3. **Coherence Maximization**
4. **Paraconsistent Preservation** (when contradiction is meaningful)

Example rule:

```
if conflict(main, syncop) {
    if context == survival:
        prefer main
    else:
        preserve contradiction
}
```

11.4.3 Harmonization and Interference Rules

Cross-track interference is governed by:

- beat alignment
- phase overlap
- truth amplitude
- logic weighting

The meta-logic engine determines whether two Tracks:

- reinforce each other (constructive)
- partially cancel (destructive)
- lock into resonance
- diverge chaotically

11.5 The Meta-Logical Evaluation Model

HaackLang's interpreter uses a **7-step meta-evaluation cycle** per global beat.

Step 1 — Snapshot

Read all track states.

Step 2 — Contextualization

Determine current contexts and their priority.

Step 3 — Cross-Track Comparison

Evaluate coherence, conflict, and interference.

Step 4 — Meta-Rule Activation

Apply meta-rules that trigger based on the snapshot.

Step 5 — Logical Adjustment

Boost/dampen/switch tracks based on meta-rules.

Step 6 — Truth Update Scheduling

Determine which tracks update this beat.

Step 7 — Commit

Apply the final meta-logical decisions.

11.6 Update Rule Hierarchy

In resolving multiple possible updates, HaackLang uses this hierarchy:

1. **Meta-Context Rules** (highest)
2. **Context Rules**
3. **Meta-Logic Rules**
4. **Track Logic Rules**
5. **Local Expression Rules** (lowest)

This ensures global coherence without sacrificing local nuance.

11.7 Interpreter-Level Meta-Logic

Architecture

The HaackLang interpreter has three meta-logic components:

11.7.1 The Meta-Inspector

Reads:

- track diagnostics
- phase and period
- coherence measures
- history windows

11.7.2 The Meta-Resolver

Applies:

- conflict resolution
- interference rules
- track harmonization

11.7.3 The Meta-Executor

Executes:

- boosts
- dampens
- logic switches
- freeze/thaw commands
- context reshaping

11.8 Meta-Logic and Emergent Reasoning

The magic of meta-logic is what it produces:

- emergent clarity from conflict

- slow insights crystallizing over beats
- sudden intuitive leaps
- stable contradictions that aid creativity
- coherent reasoning across incompatible logics
- dynamic self-repair of reasoning processes

Meta-logic is the *conductor* of HaackLang's orchestra.

HaackLang Language Specification — Chapter 12

Scopes, Frames, and Cognitive Layers

Chapter 12 — Scopes, Frames, and Cognitive Layers

In classical programming languages, scope is a lexical concern: variables live inside blocks, functions, or modules. But HaackLang operates across **temporal rhythms**, **logical pluralism**, and **cognitive contexts**, so scope must be redefined as a *multi-dimensional structural field*.

In HaackLang, a variable's meaning is determined by:

- **Lexical scope** (traditional block/function/module)
- **Rhythmic scope** (which Tracks update or read it)
- **Contextual scope** (which cognitive domain it belongs to)
- **Meta-logical scope** (how meta-rules treat it)
- **Frame scope** (which cognitive layer currently owns it)

This chapter specifies all five dimensions, defines how they interact, and provides the formal resolution model used by the interpreter.

12.1 The Five-Dimensional Scope Model

HaackLang's scoping system is built on a five-axis model:

Dimension	Controls	Description
Lexical Scope	text structure	Traditional block/function/module boundaries
Rhythmic Scope	track membership	Determines which Tracks update a value
Contextual Scope	contexts	Determines which cognitive domain governs behavior
Meta-Logical Scope	meta-rules	Determines how reasoning about this variable behaves
Cognitive Frame Scope	active cognitive layer	Determines which layer "owns" the variable at runtime

Only HaackLang uses this unified model.

12.2 Lexical Scope (Traditional)

Lexical scope determines where identifiers can be referenced.

Example:

```
let x = 1

context planning {
    let y = x + 2
}
```

`x` is visible inside `planning`; `y` is not visible outside.

HaackLang supports:

- block scope
- function scope
- module scope
- context scope (special)

12.3 Rhythmic Scope: Visibility Across Tracks

Every truthvalue in HaackLang is implicitly multi-track:

```
truthvalue danger
```

expands to:

```
danger.main
danger.slow
danger.syncop
```

Rhythmic scope determines:

- which Tracks update the variable
- which Tracks *read* the variable
- how values propagate between Tracks

Example restricting rhythmic visibility:

```
danger restricted_to [main, slow]
```

This prevents intuition (`syncop`) from modifying or reading it.

12.4 Contextual Scope: Cognitive Domains

Contexts define cognitive domains such as:

- perception
- planning
- introspection
- survival
- memory

Variables can be bound to contexts:

```
context perception {
    truthvalue brightness
}
```

Contexts may override Track behaviors:

```
context survival using track main
```

A variable declared inside a context belongs to that domain unless re-exported.

12.5 Meta-Logical Scope: Reasoning About a Variable

Meta-logical scope determines how the system reasons *about the variable itself*.

Examples:

```
meta sensitive(danger)
meta freeze_on_conflict(trust)
meta propagate(main -> slow, fear)
```

These declarations influence:

- conflict resolution
- logic switching

- cross-track propagation schedules
- reflection loops

12.6 Cognitive Frames: The Mind-Layer Stack

HaackLang implements five cognitive layers as runtime frames:

1. **Perceptual Frame** — immediate sensory-like signals
2. **Evaluative Frame** — fuzzy belief updates
3. **Intuitive Frame** — paraconsistent intuition
4. **Reflective Frame** — slow reasoning and meta-cognition
5. **Executive Frame** — decision-making and action selection

Variables may “belong” to a specific layer.

Example:

```
frame intuitive owns anxiety
```

This affects:

- when updates occur
- which logic is applied
- how contradictions are handled
- cross-frame influence rules

12.7 The Frame Stack Execution Model

Each global beat executes cognitive layers in order:

```
1. Perception → (fast classical logic)
2. Evaluation → (fuzzy logic)
3. Intuition → (paraconsistent logic)
```

```
4. Reflection → (meta-logic)
5. Execution → (classical/fuzzy hybrid)
```

Each layer sees a different “shadow” of the variable.

Example:

```
if danger.main > 0.8    # perceptual frame triggers
if danger.slow > 0.6    # reflective frame triggers
if danger.syncop > 0.9  # intuitive frame triggers
```

12.8 Cross-Scope Resolution Rules

When multiple scopes apply, HaackLang resolves them using a hierarchy:

1. Meta-Logical Scope (highest)

If meta-rules specify behavior, they dominate.

2. Contextual Scope

Context-specific logic overrides Track defaults.

3. Rhythmic Scope

Track-specific truth-values influence update behavior.

4. Lexical Scope

Traditional name resolution.

5. Frame Scope

Frame-specific view of truth.

12.9 Visibility Resolution Algorithm

When a variable is referenced, the interpreter uses this algorithm:

1. Identify the variable's lexical region.
2. Determine active context(s).
3. Determine which Tracks are active for the current beat.
4. Determine which cognitive frame is currently evaluating.
5. Apply meta-rules that modify visibility.
6. Return the appropriate track-specific value.

This allows a single line of code to have different meanings in different layers.

12.10 Scope Inheritance and Overriding

Scopes can inherit and override rules.

Contextual inheritance:

```
context social inherits cognitive
```

Track inheritance:

```
track hyper inherits slow using fuzzy
```

Frame inheritance:

```
reflective inherits evaluative
```

This creates cognitive hierarchies.

12.11 Combined Example

```
track main   period 1 using classical
track slow   period 4 using fuzzy
track syncop period 7 using paraconsistent

context planning using track slow {
    truthvalue decision_confidence restricted_to [slow]
    meta boost_on_alignment(decision_confidence)
}

frame evaluative owns decision_confidence
```

Interpretation:

- Only slow-track logic updates `decision_confidence`
- Only the evaluative frame sees and controls it
- Planning context modulates it with fuzzy logic
- Meta-rule boosts it when tracks align

12.12 Summary

HaackLang's multi-dimensional scoping model allows:

- dynamic cross-track truth propagation
- context-driven logic switching
- meta-logical governance
- cognitive layer specialization

It is the most sophisticated scoping system ever designed for a programming language.

HaackLang Language Specification — Chapter 13

Flow Control in a Polylogical, Polyrhythmic Language

Chapter 13 — Flow Control in a Polylogical, Polyrhythmic Language

Traditional programming languages treat flow control as a matter of evaluating Boolean conditions at a single moment in time. HaackLang breaks this mold by introducing a **polylogical**, **polyrhythmic**, **context-driven**, and **meta-governed** flow control system.

HaackLang's flow control allows:

- different tracks to evaluate conditions with their own logics
- branches to trigger based on rhythmic alignment
- contradictory branches to execute in parallel without explosion
- context-sensitive branching
- meta-logic to override or redirect control flow
- reflective and recursive self-modifying flow

This chapter specifies the full system.

13.1 The Need for Multi-Track Flow Control

Flow in HaackLang must account for:

- multiple truth-values per proposition
- different update tempos
- contradictions across tracks
- context-dependent semantics
- meta-logical governance
- cognitive layer sequencing

In traditional languages, flow is:

```
if condition then block
```

But HaackLang must answer:

- *Which track evaluates `condition` ?*
- *Do other tracks get a say in whether the branch triggers?*
- *What happens if tracks disagree?*
- *Do paraconsistent contradictions fork execution?*
- *Does the current context override track logic?*
- *Do meta-rules intervene?*
- *Should the branch be delayed until its Beat aligns?*

HaackLang solves these through **polylogical branching**.

13.2 Multi-Track Condition Evaluation

A condition in HaackLang evaluates across multiple tracks.

Example:

```
if danger > 0.7 {
    run()
}
```

This expands to:

- `danger.main`
- `danger.slow`
- `danger.syncop`

Each track evaluates the condition using its associated logic.

Classical (main):

```
danger.main > 0.7 → true/false
```

Fuzzy (slow):

```
danger.slow > 0.7 → degree of truth
```

Paraconsistent (syncop):

```
danger.syncop > 0.7 AND possibly ¬(danger.syncop > 0.7)
```

13.3 The Cross-Track Branching Rule

HaackLang defines the **branch activation vector**:

```
B = [b_main, b_slow, b_syncop]
```

Where each element is the track's evaluation.

Branch fire-rules depend on mode:

13.3.1 Classical Mode (default)

Branch fires if:

```
b_main == true
```

13.3.2 Fuzzy Threshold Mode

Branch fires if:

```
b_slow ≥ threshold
```

13.3.3 Polylogical Mode

Branch fires if any of the following are true:

- majority tracks agree
- context assigns priority to a track
- meta-rule activates it

13.4 Contradiction-Preserving Branches

Paraconsistent tracks can produce contradictions. Example:

```
b_syncop = true AND false
```

Classical languages break under this. HaackLang allows **contradiction-preserving branching**:

```
if!! paradox_signal {
    explore_both_paths()
}
```

`if!!` means:

- execute both branches in parallel cognitive frames
- merge results using context or meta-logic

This models human contradictory thoughts.

13.5 Beat-Gated Flow Control (Rhythmic Conditionals)

Each track has a period; flow control may require rhythmic alignment.

```
when@slow danger > 0.5 {
    reevaluate_plan()
}
```

This executes only when:

- slow track's update window opens
- condition evaluates true

Advanced example:

```
if@phase(3) intuition.syncop > 0.8 {
    panic()
}
```

This expresses temporal cognition.

13.6 Context-Sensitive Branching

A context can override track rules.

Example:

```
context survival {
    override track main only
}

if danger > 0.8 {
    flee()
}
```

In survival context:

- only `main` track is consulted
- paraconsistent or fuzzy hesitation is ignored

13.7 Cross-Context Flow Control

Transitions between contexts use:

- priority rules
- meta-rules
- truth thresholds
- contradictions

Example:

```
if @meta(conflict(reason, intuition)) > 0.7 {
    enter context introspection
}
```

The flow enters a new cognitive domain.

13.8 Reflective and Recursive Flow Modes

Flow can reference its own history:

```
loop reflective until stable(belief) {
    reconsider()
}
```

Or recursively inspect its own operation:

```
if @meta(loop_duration) > threshold {
    meta::interrupt()
}
```

13.9 Interruptible Reasoning

HaackLang supports interrupt-driven flow:

```
on contradiction spike {
    pause slow
    boost main
}
```

Or:

```
on alignment(track main, track syncop) {
    ignite insight
```

```
}
```

13.10 Polylogical Loops

Loops may be governed by:

- classical termination
- fuzzy convergence
- paraconsistent oscillation
- rhythm gating

Examples:

Classical loop:

```
while health.main > 0 {
    fight()
}
```

Fuzzy loop:

```
until trust.slow ≥ 0.9 {
    build_rapport()
}
```

Paraconsistent oscillation loop:

```
loop!! until contradiction_resolved(fear.syncop) {
    explore_shadow_beliefs()
}
```

13.11 Multi-Track For-Loops

A for-loop may operate across multiple tracks.

```
for rhythm(main, slow) i in range(10) {
    pulse_memory(i)
}
```

This executes iterations when either or both tracks update.

13.12 Flow Merge Semantics

When branches diverge (e.g., via paraconsistent splitting), they must merge.

Merge occurs via:

- context dominance
- track priority
- meta-logical decisions
- fuzzy blending
- reflective synthesis

Example:

```
merge using reflective {
    integrate_insights()
}
```

13.13 Summary

HaackLang's flow control system is unique in programming language history. It integrates:

- multi-track logic
- rhythm gating
- contradiction-preserving execution
- context-sensitive branches
- meta-logical oversight
- reflective and interruptible flow

It models how minds process decisions over time.

HaackLang Language Specification — Chapter 14

Update Rules, Temporal Dynamics, and Cross-Track Propagation

Chapter 14 — Update Rules, Temporal Dynamics, and Cross-Track Propagation

HaackLang's update model is unlike that of any traditional programming language. Instead of assuming uniform evaluation at discrete time steps, HaackLang implements **multi-rhythmic temporal dynamics**, **per-track update rules**, **cross-track propagation**, **phase-based interference**, and **meta-governed overrides**.

In the human mind, different cognitive layers update at different tempos:

- perception → fast, crisp, constant
- belief → slower, fuzzy, gradual
- intuition → irregular, oscillatory, contradictory
- reflection → deliberate, sporadic
- meta-cognition → episodic and strategic

HaackLang formalizes these as **Tracks**, each with its own update rules, and then specifies **cross-track propagation** that allows reasoning to flow between them.

This chapter defines the full update mechanics, including:

- track-local update rules
- propagation operators
- harmonic alignment and phase offsets
- beats, windows, and firing thresholds
- event-driven updates

- meta-logical overrides
- the Global Beat Engine (GBE) specification

14.1 The Temporal Foundation: Beats and Cycles

HaackLang runs on a global temporal structure:

14.1.1 Global Beat (τ)

The entire system ticks once per global beat. This beat drives:

- track update windows
- context evaluation
- meta-logic checks
- frame sequencing

14.1.2 Track Period (P^t)

Each Track has an integer period:

```
track slow period 4
```

This means:

- slow updates every 4 global beats
- fast updates every beat
- syncop updates every 7 beats

14.1.3 Phase Offset (Φ^t)

Tracks may update not just periodically, but with offset:

```
track intuition period 7 phase 3
```

Meaning:

- intuition updates on beats $\tau \equiv 3 \pmod 7$

14.1.4 Update Window (W^t)

The window is the moment a Track is allowed to recompute. A Track either:

- **fires** (updates), or
- **holds** (keeps previous value)

14.2 Track-Level Update Rules

Each Track uses the logic assigned to it.

14.2.1 Classical Track Updates

For Tracks using classical logic (e.g., `main`):

```
p^t(τ+1) = L_classical(expression)
```

Where `L_classical` evaluates to true/false.

14.2.2 Fuzzy Track Updates

Fuzzy tracks (e.g., `slow`) update via t-norm operations.

```
p^t(τ+1) = T(f(p^t(τ)), input)
```

Where `T` is a t-norm or s-norm.

14.2.3 Paraconsistent Track Updates

These can produce contradictory states.

```
p^t(τ+1) = {true, false} simultaneously when expression supports both
```

Contradiction is stored.

14.2.4 Track Freeze and Thaw

Tracks can be frozen:

```
meta::freeze(slow)
```

Meaning:

```
p_slow(τ+1) = p_slow(τ)
```

14.3 Cross-Track Propagation

Truth moves between Tracks according to propagation rules.

Propagation directions:

- **fast** → **slow** (perception influences belief)
- **slow** → **syncop** (belief influences intuition)
- **syncop** → **main** (intuition may override perception)

14.3.1 Default Propagation Rule

```
p_b(τ+1) = α · p_a(τ) + (1−α) · p_b(τ)
```

Where α ∈ [0,1] is the propagation coefficient.

14.3.2 Propagation Schedules

Propagation only occurs on specific beats:

```
propagate(main → slow) every 4 beats
propagate(slow → syncop) every 7 beats
```

Example syntax:

```
meta propagate(main -> slow) every 4
```

14.4 Harmonic Alignment and Interference

When Track periods share structure, they generate interference patterns.

14.4.1 Harmonic Alignment

Two tracks align if:

$$\tau \% P^a == \Phi^a \quad \text{AND} \quad \tau \% P^b == \Phi^b$$

Alignment produces reinforcement.

14.4.2 Destructive Interference

When track phases oppose:

$$p^a = x,\ p^b = 1 - x$$

They partially cancel:

$$p_{new} = (p^a + p^b)/2 \rightarrow \text{flattening}$$

14.4.3 Rhythmic Resonance

Multiple tracks reinforcing each other form resonance loops. This can trigger:

- insight
- panic
- conviction
- emotional spikes

In HaackLang:

```
on resonance(main, slow, syncop) {
    ignite insight
}
```

14.5 Event-Driven Updates

Some updates ignore rhythm and fire on events.

Examples:

```
on contradiction spike {
    boost slow
}

on alignment(main, intuition) {
    amplify conviction
}
```

14.6 Meta-Governed Update Overrides

Meta-logic can override update behavior.

Example:

```
if @meta(conflict(main, syncop)) > 0.7 {
    meta::dampen(syncop)
}
```

Meaning:

```
reduce propagation into syncop
```

Meta-rules may:

- delay updates
- accelerate belief revision
- suppress intuition
- enhance stability

14.7 The Global Beat Engine (GBE)

GBE is the heart of temporal dynamics.

The GBE must:

1. Advance global time τ
2. Trigger track update windows
3. Trigger context evaluations
4. Run meta-logic
5. Run event checks
6. Apply update rules
7. Emit next global state

14.7.1 GBE Pseudocode

```
tau = 0

loop forever:
    tau += 1
    open_update_windows(tau)
    run_context_logic(tau)
    run_meta_logic(tau)
    run_event_checks(tau)
    apply_track_updates(tau)
    propagate_cross_track()
    commit_state()
```

14.8 Formal Update Algebra

The full update equation for a truthvalue `p`:

[p(τ+1) = U_t(p, τ) + Σ_{a→b} P_{a→b}(τ) + M(τ) + E(τ)] Where:

- `U_t` = track update rule
- `P_{a→b}` = cross-track propagation
- `M` = meta-logic influence
- `E` = events

This generalizes all update behavior.

14.9 Summary

HaackLang's update system is a complete temporal reasoning architecture. It models:

- asynchronous reasoning
- layered cognition
- contradictions
- rhythmic influences
- cross-track feedback
- meta-level regulation

It is the first programming language to integrate temporal logic, cognitive modeling, and rhythmic computation into a unified update formalism.

HaackLang Language Specification — Chapter 15

Interpreter Design and Execution Model

Chapter 15 — Interpreter Design and Execution Model

HaackLang is not merely a syntax and semantics specification — it is a fully realized cognitive execution model. Its interpreter must support:

- multi-track logical evaluation
- asynchronous temporal rhythms
- context-driven logic switching
- meta-logical governance
- contradiction-preserving execution
- cognitive layer sequencing
- reflective and recursive self-analysis

This chapter defines the architecture of the **HaackLang Interpreter (HLI)**, the **Polylogical Evaluation Pipeline (PEP)**, the **Global Beat Engine (GBE)**, the **Cognitive Frame Stack (CFS)**, the **Track Manager**, the **Context Engine**, the **Meta-Logic Engine**, and the **Contradiction-Preserving Execution Engine (CPEE)**.

It concludes with formal execution diagrams that unify HaackLang's temporal, logical, contextual, and cognitive mechanics.

15.1 Overview of Interpreter Architecture

The HaackLang Interpreter (HLI) consists of 7 major subsystems:

1. **Parser & AST Builder** — constructs an abstract syntax tree
2. **Polylogical Evaluator (PEP)** — evaluates expressions across Tracks
3. **Global Beat Engine (GBE)** — drives time, rhythm, and phase
4. **Track Manager** — manages per-track updates and propagation
5. **Context Engine** — manages cognitive contexts and domain-specific logic
6. **Meta-Logic Engine** — orchestrates reasoning about reasoning
7. **Cognitive Frame Stack (CFS)** — manages layered reasoning modes

8. **Contradiction-Preserving Execution Engine (CPEE)** — handles paraconsistent branches

Each subsystem operates in a well-defined temporal order.

15.2 High-Level Execution Loop

The HLI runs continuously through a global loop:

```
while (true) {
    τ += 1
    parse_pending_inputs()
    open_update_windows(τ)
    evaluate_contexts(τ)
    run_meta_logic(τ)
    evaluate_expressions_across_tracks(τ)
    propagate_cross_track_truth(τ)
    run_event_driven_updates(τ)
    execute_branches_and_loops(τ)
    commit_cognitive_frames(τ)
}
```

This loop mirrors cognitive cycles in living reasoning systems.

15.3 Abstract Syntax Tree (AST) Model

HaackLang's AST must encode:

- multi-track truthvalues
- context blocks
- logic-specific operations
- meta-operators
- rhythmic operators
- contradiction-preserving forms
- cross-context flow constructs

Example AST Node Types

- **TruthValueNode** — holds multi-track truth vectors
- **TrackOpNode** — operations tied to specific Tracks
- **ContextNode** — defines a cognitive domain
- **MetaOpNode** — meta-logical operations
- **BeatConditionalNode** — rhythm-gated conditionals
- **ContradictoryBranchNode** — paraconsistent `if!!`
- **ResonanceTriggerNode** — event nodes based on rhythmic alignment

The AST is not linear but **polylogical**, meaning some nodes are evaluated with multiple logics.

15.4 The Polylogical Evaluation Pipeline (PEP)

This is the heart of expression evaluation.

PEP evaluates every expression across:

- classical logic (fast)
- fuzzy logic (slow)
- paraconsistent logic (syncopated)
- meta-logical overrides
- context-specific logic bindings
- rhythmic gating

PEP Stage Overview:

1. **Lexical evaluation** (AST-level)
2. **Logic binding** (assign logic per Track or Context)
3. **Per-Track expression evaluation**
4. **Cross-track state combination** (if needed)
5. **Meta-level inspection**
6. **Meta-level correction** (boost, dampen, reroute)
7. **Propagate results into Track values**

Example Flow:

```
danger > 0.8
↓ evaluate classical
↓ evaluate fuzzy
↓ evaluate paraconsistent
↓ meta checks conflict
↓ meta boosts slow track
↓ produce multi-track result
```

15.5 Global Beat Engine (GBE)

The GBE handles all temporal rhythms.

Responsibilities:

- incrementing global beat τ
- firing Track update windows
- firing Context evaluation cycles
- rhythmic gating of conditionals
- periodic meta-rule checks
- propagating cross-track truth

Beat Algorithm:

```
function global_beat_cycle(τ):
    open_update_windows(τ)
    handle_phase_offsets(τ)
    fire_periodic_triggers(τ)
    propagate_if_scheduled(τ)
    align_resonant_tracks(τ)
```

15.6 Track Manager

The Track Manager controls:

- truth storage for each Track
- update windows
- logic execution per Track
- phase alignment
- freeze/thaw behavior
- propagation schedules
- harmonic resonance detection

Data Structure:

```
struct TrackState {
    value: float or set{true, false}
    period: int
    phase: int
    logic: LogicType
    frozen: bool
    history: list
}
```

Tracks are updated **only** when:

```
τ % period == phase AND frozen == false
```

15.7 Context Engine

Contexts define cognitive domains.

The Context Engine handles:

- context activation and deactivation
- context priority resolution
- context-specific logic overrides
- context-bound variables and frames
- cross-context transitions

Context Prioritization:

Each context has a priority score. Higher-priority contexts override:

- track logic
- meta-logic
- propagation rules

Example:

```
context survival priority 10
context introspection priority 2
```

During survival:

- only main (classical) logic dominates

15.8 Meta-Logic Engine

The meta-logic engine inspects the system continuously.

Responsibilities:

- detecting contradictions
- detecting cross-track misalignment
- resolving conflicts
- switching logics dynamically
- boosting or dampening tracks
- controlling propagation
- freezing or unfreezing Tracks
- triggering reflective loops

Meta-Evaluation Cycle:

1. read Track states
2. compute coherence
3. detect contradiction spikes
4. detect resonance events
5. apply meta-rules

6. adjust interpreter behavior

Meta-logic is the executive control system.

15.9 Cognitive Frame Stack (CFS)

HLI runs cognition through a stack of reasoning modes.

Frames:

1. **Perceptual**
2. **Evaluative**
3. **Intuitive**
4. **Reflective**
5. **Executive**

Each frame has its own logic, update behavior, and access to truthvectors.

Frame Advancement:

On each beat:

```
for frame in [perceptual, evaluative, intuitive, reflective, executive]:
    evaluate_frame(frame)
```

Frames can:

- fork (for paraconsistent branching)
- merge (via meta-rules)
- suspend (via freeze states)

15.10 Contradiction-Preserving Execution Engine (CPEE)

This subsystem handles:

- paraconsistent branches (`if!!`)
- contradictory track states
- oscillatory truth patterns
- ambiguity preservation
- multi-path reasoning forks
- reflective synthesis of contradictions

Execution of `if!!` :

```
if!! condition {
    branch_A()
} else {
    branch_B()
}
```

CPEE:

1. evaluates condition per Track
2. if contradiction exists → forks cognitive frames
3. executes both branches
4. merges results using:
    - context rules
    - meta-logic
    - coherence maximization
    - fuzzy blending

15.11 Memory Model

HaackLang variables store **truthvectors**, not scalars.

Memory structure:

```
struct TruthVector {
    main: float | bool
    slow: float
    syncop: float | {true, false}
```

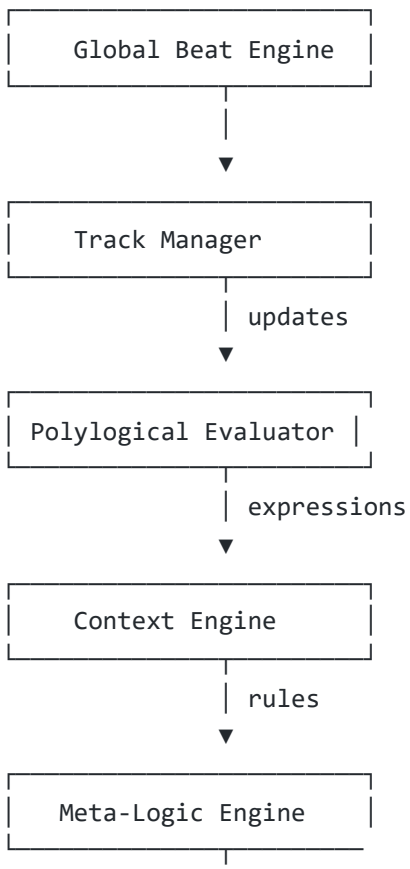
```
    history: list
}
```

Each update stores history for:

- meta-analysis
- reflection
- long-term reasoning

15.12 Putting It All Together: Formal Execution Diagram

Below is the flow of a single beat τ.




```
# Chapter 16 — The HaackLang Virtual Machine (HLVM): A Polylogical, Poly

## Overview
The HaackLang Virtual Machine (HLVM) is the low-level execution substrat
```

- a **polylogical bytecode format**
- a **multi-track register file**
- **rhythmic execution units** bound to Tracks
- a **Global Beat Scheduler (GBS)** that coordinates tick-level timing
- a **Context Authority Layer (CAL)** controlling scope and logic switch
- a **Contradiction-Preserving ALU (CP-ALU)** for paraconsistent operati
- a **Meta-Logical Coprocessor (MLC)** that evaluates coherence and conf

Together, these create a computational model capable of executing multi-

---

16.1 Architecture Goals
The HLVM is designed to satisfy five core requirements:

1. Temporal Pluralism
Multiple Tracks (main, slow, syncop, and user-defined) must update on th

2. Logical Pluralism
Classical, fuzzy, and paraconsistent operations must coexist at the byte

3. Context-Sensitive Execution
Execution semantics change when cognitive contexts gain or lose control.

4. Meta-Level Governability
Meta-logic must be able to inspect, throttle, freeze, reroute, or overri

5. Contradiction Preservation
The VM must be able to represent and propagate contradictory truthvector

---

16.2 HLVM Execution Stack
HLVM execution consists of **four cooperating subsystems**:

1. **GBS — Global Beat Scheduler**
   - Drives the global tick
   - Wakes Tracks on their firing beats
   - Manages update windows

2. **TES — Track Execution Subsystems**
   - One per Track
   - Handles logic-specific ALU operations
   - Maintains register bank slices

3. **CAL — Context Authority Layer**
   - Activates/deactivates contexts
   - Applies priorities and overrides

```
   - Controls scoping rules

4. **MLC — Meta-Logical Coprocessor**
   - Computes coherence, conflict, resonance
   - Adjusts Track weights
   - Freezes/thaws Tracks
   - Issues meta-events

The HLVM executes all four subsystems every global tick, but each operat

---

## 16.3 The HLVM Register Model
Unlike classical VMs, HLVM registers are **multi-track truthvector regis

### **16.3.1 Register File Structure**
The register file is partitioned by Track:
```

R_main[0..255] R_slow[0..255] R_syncop[0..255] R_meta[0..63] # meta-logic coprocessor registers

```
Each logical register `R[i]` is a projection into all tracks:
```

R[i] = { main: R_main[i], slow: R_slow[i], syncop: R_syncop[i] }

```
This mirrors the internal structure of BoolRhythm.

---

## 16.4 HLVM Bytecode Format
HLVM instructions are **track-qualified**, **logic-qualified**, and **te

### **16.4.1 Instruction Word Structure**
```

+--------+-----------+----------+------------+ | OPCODE | TRACK | FLAGS | OPERANDS | +--------+-----------+----------+------------+

```
- **OPCODE** — determines ALU operation
- **TRACK** — specifies which Track executes it (or ALL)
```

```
- **FLAGS** — meta behavior, contradiction-preserving mode, phase gating
- **OPERANDS** — registers or immediates

---

## 16.5 Core Instruction Set

### **16.5.1 Classical Instructions (MAIN Track)**
```

AND rA, rB → rA = rA ∧ rB OR rA, rB → rA = rA ∨ rB NOT rA → rA = ¬rA CMP rA,rB → compare for crisp equality

```
### **16.5.2 Fuzzy Instructions (SLOW Track)**
```

F_AND rA,rB → t-norm(min) or s-norm blend F_OR rA,rB → max/min blend F_NOT rA → 1 - rA F_BLEND rA,rB,w → rA = rA*(1-w) + rB*w

```
### **16.5.3 Paraconsistent Instructions (SYNCOP Track)**
```

P_AND rA,rB → contradiction-preserving conjunction P_OR rA,rB → tolerant disjunction P_BOTH rA → mark rA as {true,false} P_RESOLVE rA → local contradiction resolution

```
These use the CP-ALU.

---

## 16.6 Rhythmic & Temporal Instructions
HLVM exposes the temporal machinery directly.
```

WAIT n → stall until n beats passed UNTIL φ → stall until phase φ of active Track WHEN t,cond → conditional execution gated by Track t's firing

```
These allow programs to explicitly leverage rhythmic truth.

---
```

```
## 16.7 Context Management Instructions
```

CTX_ENTER c CTX_EXIT c CTX_OVERRIDE track,logic CTX_PRIORITY trackA > trackB

```
Context switches do not clear registers — all truthvectors persist.

---

## 16.8 Meta-Logical Coprocessor Instructions
```

META_SNAP rA → force classical snap-to-boolean META_COH rA → coherence score META_BOOST t → increase track weight META_DAMP t → reduce track weight META_FREEZE t META_THAW t META_ROUTE cond → reroute logic through different Track

```
The MLC operates at a slower meta-beat.

---

## 16.9 Contradiction-Preserving Execution
HLVM supports explicit contradiction propagation using *dual-valued regi

A contradictory register holds:
```

R_syncop[i] = {true,false}

```
Propagation rules ensure that contradictions:
- do not explode (no classical collapse)
- propagate only into syncop or meta regions unless explicitly fused
- can be merged via meta-rules when coherence increases

---

## 16.10 The Global Beat Scheduler (GBS)

The GBS manages all rhythmic behavior.
```

```
### **16.10.1 Scheduler Cycle**
```

τ += 1 for each Track t: if τ % t.period == t.phase: wake Track t if τ % meta_period == 0: wake MLC

```
### **16.10.2 Beat-Gated Instruction Window**
Tracks may only commit writes during their firing window.

This enforces:
- slow drift
- intuitive spikes
- reflective lag
- high-frequency perception

---

## 16.11 Execution Examples

### **16.11.1 Fuzzy Belief Revision**
```

F_BLEND R1,R2,0.3 ; belief drift WAIT 4 ; slow track period

```
### **16.11.2 Contradiction-Preserving Dual Branch**
```

P_BOTH R3 ; set contradictory intuition WHEN syncop,cond ; intuition-gated branch

```
### **16.11.3 Meta-Coherence Driven Logic Switch**
```

META_COH R8 CMP R8,threshold IF < THEN META_ROUTE syncop→slow

```
---

## 16.12 Summary
The HLVM provides:
- a temporal, multi-track register model
```

```
- logic-bound ALUs
- rhythmic scheduling
- contradiction-preserving execution
- meta-governed cognitive control

It is the first VM capable of executing *mind-like computations* at the l


# Chapter 17 — HLVM Bytecode Specification & Full Instruction Tables

## 17.1 Overview
Chapter 17 defines the **formal bytecode structure** of the HaackLang Vi
- the **complete bytecode encoding format**
- the **instruction word schema**
- all **core, fuzzy, paraconsistent, rhythmic, context, and meta-logic o
- operand formats and addressing modes
- examples of multi-track execution encodings
- contradiction-preserving instruction encodings

This chapter provides the authoritative low-level specification for HLVM

---

## 17.2 Bytecode Word Format
Each HLVM instruction is encoded as a **32-bit instruction word** with o

### **17.2.1 Base Instruction Word Layout (32 bits)**
```

```
+----------+----------+----------+----------------+ | OPCODE | TRACK | FLAGS |
OPERAND SPEC | | (8 bits) | (4 bits) | (4 bits) | (16 bits) | +----------+----------+-
---------+----------------+
```

```
### Field Definitions
- **OPCODE (8 bits)** – determines the ALU operation or VM action
- **TRACK (4 bits)** – which Track executes it (MAIN = 0x1, SLOW = 0x2, 
- **FLAGS (4 bits)** – gating, contradiction mode, meta-routing
- **OPERAND SPEC (16 bits)** – register indices, immediates, or addressi

If more operands are needed (e.g., 3-register instructions), the HLVM us

---

## 17.3 Track Encoding Table
```

0x0 NONE / INVALID 0x1 MAIN (classical) 0x2 SLOW (fuzzy) 0x3 SYNCOP (paraconsistent) 0x4 META (meta-coprocessor) 0xF ALL_TRACKS (broadcast)

```
Broadcast mode is used for context or meta instructions affecting all tr
```

```
---
```

```
## 17.4 Flag Encoding Table
```

Bit 0 — Phase-gated execution (1 = gated) Bit 1 — Contradiction-preserving mode Bit 2 — Meta-route override Bit 3 — Halt-on-context-switch

```
Flags allow HLVM instructions to respond dynamically to logic and contex
```

```
---
```

```
## 17.5 Operand Encoding
The 16-bit operand field supports several formats:
```

```
### **17.5.1 Register–Register Format (RR)**
```

```
+--------+--------+ | rA (8) | rB (8) | +--------+--------+
```

```
### **17.5.2 Register–Immediate (RI)**
```

```
+--------+-----------------+ | rA (8) | immediate (8) | +--------+----------
+
```

```
### **17.5.3 Three-Operand Extension**
Uses an extension word:
```

WORD 0: OPCODE/TRACK/FLAGS/FORMAT (RRX) WORD 1: rA (8), rB (8) WORD 2: rC (8), UNUSED (8)

```
---
```

```
## 17.6 Opcode Space
OPCODES are divided by subsystem:
```

0x00–0x1F Classical ALU (MAIN) 0x20–0x3F Fuzzy ALU (SLOW) 0x40–0x5F Paraconsistent ALU (SYNCOP) 0x60–0x6F Rhythmic & Temporal Control 0x70–0x7F Context & Scope Control 0x80–0x8F Meta-Logic Coprocessor 0x90–0x9F System & VM Instructions

```
---

# 17.7 Instruction Tables

Below are full tables for every instruction class.

---

## 17.7.1 Classical ALU Instructions (MAIN Track)
```

Opcode | Mnemonic | Description -------+----------+---------- 0x01 | AND | rA = rA ∧ rB 0x02 | OR | rA = rA ∨ rB 0x03 | NOT | rA = ¬rA 0x04 | XOR | exclusive OR 0x05 | CMP | set flags based on crisp compare 0x06 | MOV | rA = rB 0x07 | SET | rA = imm

```
---

## 17.7.2 Fuzzy ALU Instructions (SLOW Track)
```

Opcode | Mnemonic | Description -------+----------+---------- 0x20 | F_AND | fuzzy AND (t-norm) 0x21 | F_OR | fuzzy OR (s-norm) 0x22 | F_NOT | rA = 1 - rA 0x23 | F_BLEND | weighted blend rA,rB using imm 0x24 | F_DAMP | rA = rA * imm (decay) 0x25 | F_GAIN | rA = rA + imm (rise)

```
---

## 17.7.3 Paraconsistent ALU Instructions (SYNCOP Track)
```

Opcode | Mnemonic | Description -------+----------+---------- 0x40 | P_AND | contradiction-preserving AND 0x41 | P_OR | contradiction-preserving OR 0x42 | P_NOT | assign dual-valued negation 0x43 | P_BOTH | set rA = {true,false} 0x44 | P_RES | local contradiction resolve

---

17.7.4 Rhythmic & Temporal Control

Opcode | Mnemonic | Description -------+----------+---------- 0x60 | WAIT | stall until N beats have passed 0x61 | UNTIL_PHASE | wait until Track phase matches imm 0x62 | WHEN | execute only when Track fires 0x63 | RHY_SYNC | force track alignment event

---

17.7.5 Context & Scope Control

Opcode | Mnemonic | Description -------+----------+---------- 0x70 | CTX_ENTER | enter context c 0x71 | CTX_EXIT | exit context c 0x72 | CTX_OVR | override track logic 0x73 | CTX_PRIORITY | set track priority ordering

---

17.7.6 Meta-Logic Coprocessor Instructions

Opcode | Mnemonic | Description -------+----------+---------- 0x80 | META_SNAP | snap-to-classical 0x81 | META_COH | compute coherence score 0x82 | META_BOOST | boost track weighting 0x83 | META_DAMP | dampen track weighting 0x84 | META_FREEZE | freeze track 0x85 | META_THAW | thaw track 0x86 | META_ROUTE | reroute logic through another track

---

17.7.7 System & VM Instructions

Opcode | Mnemonic | Description -------+----------+---------- 0x90 | NOP | no operation 0x91 | HALT | stop VM execution 0x92 | DEBUG | emit debug info 0x93 | INT | raise interrupt

---

17.8 Encoding Examples

17.8.1 Classical AND

OPCODE: 0x01 TRACK: 0x1 (MAIN) FLAGS: 0x0 OPERANDS: rA=5, rB=7 → 0x0507

17.8.2 Fuzzy Blend on Slow Track

OPCODE: 0x23 (F_BLEND) TRACK: 0x2 (SLOW) FLAGS: 0x0 OPERANDS: rA=3, rB=4, imm=0.2

17.8.3 Contradiction Injection

Opcode: 0x43 (P_BOTH) Track: 0x3 (SYNCOP) Flags: 0x2 (contradiction mode) Operands: rA=9

17.9 Summary
Chapter 17 provides the complete, formal bytecode specification for the H
- instruction encodings
- register and operand formats
- track and flag representations
- full opcode tables across all HLVM subsystems

This serves as the definitive reference for compiler implementers and VM

```
---


# Chapter 18 — The HaackLang Compiler (HLC)

## 18.1 Overview
The HaackLang Compiler (HLC) translates high-level HaackLang code—rich w
- **polylogical lowering** (mapping high-level logic to track-specific Al
- **temporal lifting** (deriving rhythmic schedules and update windows)
- **context flattening** (transforming cognitive domains into controlled
- **truthvector expansion** (mapping single variables into multi-track re
- **meta-rule compilation** (lowering meta-logic to meta-coprocessor inst
- **contradiction routing** (ensuring syncop-track ALU receives contradi

HLC is designed like a *cognitive compiler*, bridging declarative cognit

---

## 18.2 Compilation Pipeline Structure
HLC’s pipeline has **nine stages**, each corresponding to a transformati
```

Source → Lexer → Parser → AST → Polylogical Annotator → Rhythmic Scheduler → Context Flattener → Intermediate Representation (PIR) → HLVM Code Generator → Bytecode

```
### Stage Breakdown
1. **Lexing** — tokenize contexts, Tracks, meta-ops, operators, rhythm ta
2. **Parsing** — build AST with nodes supporting multi-track semantics.
3. **AST Normalization** — expand implicit BoolRhythm constructs.
4. **Polylogical Annotation** — assign logic to Tracks and nodes.
5. **Rhythmic Scheduling** — compute update periods & Beat dependencies.
6. **Context Flattening** — reduce nested contexts to CAL operations.
7. **Meta-Lowering** — convert meta-rules to MLC instructions.
8. **PIR Generation** — produce Polylogical IR.
9. **Bytecode Emission** — output HLVM instruction stream.

---

## 18.3 Lexical Model
The lexer recognizes tokens across multiple syntactic dimensions:
- **Context tokens** (`context`, `enter`, `exit`)
- **Track tokens** (`track`, `period`, `phase`)
- **Meta tokens** (`meta`, `@meta`, `meta::freeze`)
- **Rhythmic operators** (`when@slow`, `if@phase(n)`)
- **Truthvector operators** (`blend`, `inspect`, `tv`)
```

```
- **Contradiction ops** (`if!!`, `P_BOTH`)

The lexer must be phase-aware: some tokens have meaning only inside cont

---

## 18.4 AST Structure
Unlike classical ASTs, HaackLang's AST nodes contain **multi-track, mult

### Key node types:
- **ContextNode** — cognitive domain boundaries
- **TrackNode** — declares Track semantics
- **MetaRuleNode** — meta-logic blocks
- **RhythmGuardNode** — beat-gated flow constructs
- **TruthVectorNode** — BoolRhythm variables
- **ContradictionNode** — paraconsistent branches
- **PropagationNode** — cross-track truth propagation

Each node includes:
- logic binding
- track schedule
- phase constraints
- context priority

---

## 18.5 Polylogical Annotation
This compiler phase determines which logic governs each node.

### Example input code:
```haack
if danger > 0.7 { flee() }
```

Annotation output:

```
condition.main   uses classical
condition.slow   uses fuzzy
condition.syncop uses paraconsistent
```

Each operator is expanded into Track-specific subexpressions.

18.6 Rhythmic Scheduling
```

Expressions inherit Track tempos.

Example:

```
when@slow trust > 0.6
```

Produces scheduling metadata:

- Track: SLOW
- Period: from Track declaration
- Phase: inherited or default
- Beat-gate: SLOW update window

The scheduler creates:

- update loops
- hold states (freeze behavior)
- propagation events

18.7 Context Flattening

Nested contexts become **CAL instructions**:

Input:

```
context survival {
    using track main
}
```

Output (flattened):

```
CTX_ENTER survival
CTX_OVR main,classical
... body ...
CTX_EXIT survival
```

Context fusion becomes a sequence of priority reassignments.

18.8 Meta-Lowering

Meta-logic blocks compile into MLC instruction sequences.

Input:

```
meta resolve {
    if @meta(conflict(main,syncop)) > 0.7 {
        meta::dampen(syncop)
    }
}
```

Lowered IR:

```
META_COH R_conflict
CMP R_conflict,0.7
IF_GT → META_DAMP syncop
```

18.9 Polylogical Intermediate Representation (PIR)

PIR is the compiler's internal representation. Each PIR instruction includes:

- track assignment
- logic binding
- contradiction flag
- rhythmic constraints
- operand vector

Example:

```
PIR_AND(main, r1, r2)
PIR_F_BLEND(slow, r3, r4, w=0.2)
PIR_P_BOTH(syncop, r9)
```

PIR is directly mapped to HLVM bytecode.

18.10 Bytecode Generation

The final compiler stage emits HLVM instructions.

Example transformation

Input HaackLang code:

```
if danger > 0.7 { flee() }
```

Produces bytecode sequences:

```
CMP main, R_danger, 0.7
WHEN main
JMP_IF_FALSE L_end
CALL flee
L_end:
```

Fuzzy and paraconsistent variants generate additional bytecodes for other Tracks.

18.11 How HLC Handles Contradictions

Contradictory constructs (e.g., `if!!`) compile to **dual-branch forks**.

Example:

```
if!! paradox_signal {
    explore_both_paths()
}
```

Lowered bytecode:

```
P_BOTH rX
FORK branch_true, branch_false
... merge at L_merge ...
```

These are reconciled later via meta-logic.

18.12 Summary

HLC is a compiler unlike any other:

- It compiles **mind-like constructs**
- It resolves **polylogical logic bindings**
- It maps **tempo, rhythm, and context** into bytecode
- It generates programs executable by a **contradiction-tolerant VM**
- It integrates **meta-logic** as a first-class compilation target

Together, HLC and HLVM complete the HaackLang cognitive computation stack.

Chapter 19 — Execution Traces, Debugging Tools, and Cognitive Introspection

Overview

Chapter 19 defines the complete **debugging, introspection, and execution tracing framework** for HaackLang and the HLVM. Unlike classical debuggers—which step through sequential instructions—HaackLang debugging must account for:

- **multi-track parallel execution**
- **asynchronous beat-driven updates**
- **truthvector evolution** across Tracks
- **context activation and deactivation**
- **meta-logic interventions**
- **contradiction-preserving branches**
- **rhythmic interference and resonance effects**

This chapter introduces specialized debugging tools and introspection operators that allow developers (and researchers) to inspect mind-like

behavior as it unfolds.

19.1 Debugging Philosophy: Minds, Not Machines

HaackLang debuggers are designed to inspect **cognitive computation**, not mechanical computation.

Debugging goals include:

- visualizing how a belief drifts across Tracks
- tracing contradictory intuition spikes
- observing context-switch cascades
- diagnosing meta-logic interference
- understanding how rhythms synchronize or diverge
- replaying execution beat-by-beat

This is cognitive forensics.

19.2 The Three Layers of HaackLang Debugging

HLC and HLVM support debugging on three conceptual layers:

1. Track-Level Debugging

Inspects values within a single logical Track.

2. Context-Level Debugging

Captures transitions between cognitive domains.

3. Meta-Level Debugging

Examines meta-logic actions, coherence metrics, and conflict signals.

All three must be visible to understand system behavior.

19.3 Debugging Primitives in HaackLang

HaackLang introduces specialized primitives for debugging.

19.3.1 `inspect`

Prints the full truthvector of a variable:

```
inspect danger
```

Outputs something like:

```
danger = { main:0.92, slow:0.47, syncop:1 and 0 }
```

19.3.2 `trace`

Begins recording truthvector evolution over multiple Beats.

```
trace fear for 32 beats
```

19.3.3 `debug::break`

A breakpoint triggered by conditions across Tracks:

```
debug::break if fear.syncop is contradictory
```

19.3.4 `debug::watch`

Watchpoints for truthvector thresholds:

```
debug::watch trust.slow < 0.4
```

19.4 The HLVM Debugger (HLDBG)

HLDBG is the low-level runtime debugger for HLVM. It exposes:

- Track stepping
- Beat stepping
- Meta-beat stepping
- Register inspection
- Truthvector snapshots
- Context mapping
- Meta-logic signal logs
- Contradiction registers

HLDBG supports **beat-based execution stepping**:

```
step-beat
```

Unlike machine instructions, HLVM instructions are gated by Beats, so stepping Beat-by-Beat is clearer.

19.5 Truthvector Inspection Tools

Each truthvector has a temporal history stored for debugging.

19.5.1 `dump_truthvector`

Dumps raw truthvector history:

```
dump_truthvector anxiety
```

19.5.2 `plot_truthvector`

ASCII graph of evolution:

```
main:   ▁▂▃▄▅▆▇▇▆▅▄▃▂▁
slow:   ▁▁▂▂▃▃▄▄▅▅▆▆▇▇
syncop: ▇▆▅▄▃▂▁ contradictory
```

19.5.3 Multi-Track Alignment Visualization

Shows periods & phase offsets:

```
main   period=1     phase=0  ──────────────────────
slow   period=4     phase=1  ────────┐       ──────
syncop period=7     phase=3  ────────────────┐
```

19.6 Context Transition Tracing

Debug tool:

```
trace contexts
```

Produces:

```
Beat 12 → enter perception
Beat 46 → enter survival (fear spike)
Beat 81 → exit survival, enter reflection
```

Context transitions may also show causes:

```
Beat 46: @meta(fear) > 0.8 triggered survival
```

19.7 Meta-Logic Introspection

Meta-logic is visible through **meta-log snapshots**.

19.7.1 Meta Snapshot Command

```
meta::snapshot
```

Outputs:

```
coherence(fear)=0.41
conflict(main,syncop)=0.92
```

```
frozen tracks: slow
boosted tracks: syncop
```

19.7.2 Meta-Beat Tracing

```
trace meta-beats for 8 cycles
```

Shows:

- coherence evolution
- conflict spikes
- meta decisions (boost/damp/freeze)

19.8 Contradiction Timeline Tools

Contradictions in the syncop Track are not bugs—they are features.

19.8.1 `trace contradictions`

Lists every contradiction-preserving event:

```
Beat 13: P_BOTH R4
Beat 47: contradiction spike on fear.syncop
Beat 48: coherent resolution attempt via META_COH
```

19.8.2 `visualize contradictions`

ASCII waveform:

```
main:   ──────────────────────────────
slow:   ──────────────────▁▂▄▆█
syncop: ▁0▁1▁0▁1▁0▁1▁ contradiction
```

19.9 HLVM Execution Trace Format

HLDBG can export a trace file:

```
trace export "session.hlx"
```

Trace contents include:

- per-Beat register states
- Track firing map
- context stack
- meta-logic actions
- rhythmic gating events

Example snippet:

```
BEAT 44
TRACK main fired
TRACK slow held
fear.main=0.98
fear.slow=0.63
fear.syncop={true,false}
context=perception
meta: conflict(main,syncop)=0.92 → boost slow
```

19.10 Debugging Rhythmic Interference & Resonance

Debuggers provide **interference maps**.

19.10.1 `trace resonance`

Outputs interference conditions:

```
Beat 28: main+slow+syncop aligned → resonance
Beat 29: intuition spike detected
```

19.10.2 Phase Drift Analyzer

```
analyze phase-drift for 64 beats
```

Shows:

- phase pull
- rhythm decay
- emergent attractors

19.11 Cognitive Introspection Tools

These tools transform raw data into interpretive cognitive artifacts.

19.11.1 Insight Detection

```
detect insights
```

Triggered when:

- coherence rises sharply
- contradictory branches converge
- resonance events synchronize

19.11.2 Emotional State Reconstruction

```
reconstruct emotional-state
```

Uses truthvectors + meta-rules to infer states like:

- panic
- conviction
- hesitation
- insight

19.12 Summary

Chapter 19 formalizes HaackLang's unique approach to debugging and cognitive introspection:

- Tools for tracking truthvector evolution
- Beat-by-beat and Track-by-Track stepping
- Context and meta-logic tracing
- Contradiction timeline analysis
- Resonance visualization
- Cognitive-state reconstruction

These tools allow developers to investigate HaackLang programs not as mechanical processes, but as **dynamic cognitive systems**.

Chapter 20 — The HaackLang Standard Library (HSL)

Overview

The HaackLang Standard Library (HSL) provides essential building blocks for writing cognitive, polylogical, and polyrhythmic HaackLang programs. While HaackLang's language semantics allow expressive modeling of thought-like processes, the Standard Library provides:

- **Predefined contexts** (perception, survival, reflection, intuition)
- **Truthvector utilities** (blend, drift, normalize, snap, contradict)
- **Fuzzy logic helpers** (t-norms, s-norms, graded conditionals)
- **Paraconsistent utilities** (contradiction constructors, resolvers)
- **Rhythmic primitives** (phase tools, beat counters, Track synchronization)
- **Meta-logic macros** (coherence checks, conflict evaluators)
- **Cognitive behavior modules** (fear, trust, risk, belief-dynamics)
- **Debug/introspection helpers** (trace wrappers, resonance detectors)

This chapter defines all standard modules shipped with the language.

20.1 HSL Structure

HSL is divided into logical modules:

```
hsl/
  tracks.hsl
  truthvectors.hsl
  fuzzy.hsl
  paraconsistent.hsl
  rhythm.hsl
  contexts/
      perception.hsl
      survival.hsl
      reflection.hsl
      intuition.hsl
  meta.hsl
  cognition/
      fear.hsl
      trust.hsl
      belief.hsl
      risk.hsl
  debug.hsl
```

Each file defines reusable functions, macros, or context templates.

20.2 Track Utilities Module (`tracks.hsl`)

Provides helpers for manipulating Tracks.

20.2.1 `track::sync`

Forces phase alignment between Tracks:

```
track::sync(main, slow)
```

20.2.2 `track::freeze / track::thaw`

Wrappers for meta-level freeze/thaw:

```
track::freeze(slow)
track::thaw(syncop)
```

20.2.3 `track::period_of / track::phase_of`

Metadata helpers to introspect rhythmic structure.

20.3 Truthvector Utilities (`truthvectors.hsl`)

Provide built-in operations for handling BoolRhythms.

20.3.1 `tv::blend(a, b, w)`

Blend two truthvectors:

```
tv::blend(fear, trust, 0.3)
```

20.3.2 `tv::drift(a, rate)`

Applies fuzzy decay across Tracks:

```
tv::drift(belief, 0.05)
```

20.3.3 `tv::normalize(a)`

Normalizes truthvector magnitudes.

20.3.4 `tv::contradict(a)`

Creates a deliberate contradiction on syncop:

```
tv::contradict(hunch)
```

20.3.5 `tv::snap(a, logic)`

Classical/fuzzy snap:

```
tv::snap(risk, classical)
```

20.4 Fuzzy Logic Module (`fuzzy.hsl`)

Implements fuzzy operators and transformations.

20.4.1 `fuzzy::t_norm(a, b)`

Implements minimum by default.

20.4.2 `fuzzy::s_norm(a, b)`

Implements maximum.

20.4.3 `fuzzy::soft_threshold(a, t)`

Graded conditional logic.

20.4.4 `fuzzy::smooth_step(a, k)`

Smooths truth oscillations.

20.5 Paraconsistent Module (`paraconsistent.hsl`)

Provides functions for working with contradictions.

20.5.1 `para::both(a)`

Marks a truthvector as `{true, false}` on syncop.

20.5.2 `para::resolve(a, mode)`

Resolves contradictions using modes:

- preserve

- classical
- fuzzy-blend

20.5.3 `para::conflict(a, b)`

Computes tension between two truthvectors.

20.6 Rhythmic Module (`rhythm.hsl`)

Functions for accessing and manipulating rhythmic/temporal constructs.

20.6.1 `rhythm::beat()`

Returns current global Beat.

20.6.2 `rhythm::phase_of(t)`

Returns current phase alignment.

20.6.3 `rhythm::wait_for(t)`

Beat-gates code segments.

20.6.4 `rhythm::resonance(a, b, c?)`

Detects rhythmic interference and alignment.

20.7 Context Templates (`contexts/*.hsl`)

HSL ships with canonical cognitive contexts.

20.7.1 Perception Context

```
context perception {
    using track main
    using logic classical
}
```

Fast, crisp, sensory-like reasoning.

20.7.2 Survival Context

```
context survival {
    using track main
    priority main > slow
    meta resolve {
        if @meta(fear) > 0.7 { meta::boost(main) }
    }
}
```

Overrides reflection; boosts instinct.

20.7.3 Reflection Context

```
context reflection {
    using track slow
    using logic fuzzy
}
```

Deliberative thought.

20.7.4 Intuition Context

```
context intuition {
    using track syncop
    using logic paraconsistent
}
```

Handles contradictions and intuitive leaps.

20.8 Meta-Logic Module (`meta.hsl`)

Helpers for coherence, conflict, and meta-behavior.

20.8.1 `meta::coherence(a)`

Returns coherence score across Tracks.

20.8.2 `meta::conflict(a, b)`

Returns conflict magnitude.

20.8.3 `meta::boost_if(a, cond)`

Boosts track weights under condition.

20.8.4 `meta::switch_logic(t, L)`

Switches track logic.

20.9 Cognitive Behavior Modules (`cognition/*.hsl`)

Pre-built behavioral dynamics.

20.9.1 Fear Dynamics

```
module fear {
    rule spike { fear.main = max(fear.main, threat.main) }
    rule drift { fear.slow = fear.slow * 0.95 }
    rule contradiction { para::both(fear.syncop) }
}
```

20.9.2 Trust Dynamics

```
module trust {
    rule build { trust.slow = tv::blend(trust, hope, 0.1) }
    rule degrade { trust.slow *= 0.98 }
}
```

20.9.3 Belief Dynamics

Handles slow revision and coherence.

20.10 Debug Module (`debug.hsl`)

Simplifies debugging tools introduced in Chapter 19.

20.10.1 `debug::trace_var(a, beats)`

Wrapper around execution trace.

20.10.2 `debug::resonance(a, b)`

Visualizes cross-track rhythmic interference.

20.10.3 `debug::context_log()`

Prints context transitions.

20.11 Summary

The HaackLang Standard Library provides:

- essential cognitive contexts
- tools for truthvector and Track manipulation
- fuzzy, paraconsistent, and rhythmic utilities
- meta-logic helpers
- cognitive dynamics modules
- debugging and introspection utilities

Together, these modules allow developers to build sophisticated, mind-like HaackLang programs with minimal boilerplate.

HaackLang Appendices

Appendix A — EBNF Grammar

This appendix defines a reference grammar for HaackLang using Extended Backus–Naur Form (EBNF). It is intended as a guide for parser and tool implementers.

It covers the core language features:

- Tracks and track declarations
- Contexts and meta-logic blocks
- Truthvalue (BoolRhythm) declarations
- Rhythmic guards and flow control
- Expressions, calls, and debugging primitives

A.1 Lexical Elements

```
program       = { declaration | context_decl | track_decl | meta_block }

identifier    = letter , { letter | digit | "_" } ;
number        = digit , { digit } , [ "." , digit , { digit } ] ;
string        = '"' , { character - '"' } , '"' ;

letter        = 'A'..'Z' | 'a'..'z' ;
digit         = '0'..'9' ;
```

A.2 Top-Level Declarations

```
declaration   = var_decl | func_decl | rule_decl | module_decl ;

var_decl      = ( "tv" | "truthvalue" ) , identifier , [ "=" , expr ] ,
```

```
func_decl     = "fn" , identifier , "(" , [ param_list ] , ")" , block ;

param_list    = identifier , { "," , identifier } ;

rule_decl     = "rule" , identifier , block ;

module_decl   = "module" , identifier , block ;
```

A.3 Track Declarations

```
track_decl    = "track" , identifier , "period" , number ,
                [ "phase" , number ] ,
                [ "using" , logic_spec ] , ";" ;

logic_spec    = "classical"
              | "fuzzy"
              | "paraconsistent"
              | identifier ;          (* user-defined logic *)
```

A.4 Context Declarations

```
context_decl  = "context" , identifier ,
                "{" ,
                    { context_item } ,
                "}" ;

context_item  = track_use
              | logic_use
              | priority_decl
              | var_decl
              | rule_decl
              | meta_block
              | statement ;

track_use     = "using" , "track" , identifier , [ "using" , logic_spec

logic_use     = "using" , "logic" , logic_spec , ";" ;

priority_decl = "priority" , track_order , ";" ;
```

```
track_order    = identifier , ">" , identifier , { ">" , identifier } ;
```

A.5 Meta-Logic Blocks

```
meta_block     = "meta" , "resolve" ,
                 "{" , { meta_stmt } , "}" ;

meta_stmt      = if_stmt
               | meta_call , ";" ;

meta_call      = "meta::" , identifier , "(" , [ arg_list ] , ")" ;

arg_list       = expr , { "," , expr } ;
```

A.6 Statements and Blocks

```
block          = "{" , { statement } , "}" ;

statement      = var_decl
               | expr_stmt
               | if_stmt
               | if_paradox_stmt
               | while_stmt
               | until_stmt
               | loop_stmt
               | on_event_stmt
               | meta_block
               | trace_stmt
               | ";" ;          (* empty statement *)

expr_stmt      = expr , ";" ;
```

A.7 Flow Control and Rhythmic Guards

```
if_stmt        = "if" , [ rhythm_guard ] , "(" , expr , ")" , block ,
                 [ "else" , block ] ;
```

```
if_paradox_stmt
             = "if!!" , "(" , expr , ")" , block ,
               [ "else" , block ] ;

while_stmt    = "while" , [ rhythm_guard ] , "(" , expr , ")" , block ;

until_stmt    = "until" , [ rhythm_guard ] , "(" , expr , ")" , block ;

loop_stmt     = "loop" , [ "!!" ] , block ;

on_event_stmt = "on" , event_spec , block ;

rhythm_guard  = "when@" , identifier
              | "if@phase" , "(" , number , ")" ;

event_spec    = "contradiction" , "spike"
              | "alignment" , "(" , track_list , ")"
              | identifier ;

track_list    = identifier , { "," , identifier } ;
```

A.8 Expressions

```
expr          = assign_expr ;

assign_expr   = logic_or_expr , [ assign_op , assign_expr ] ;

assign_op     = "=" | "+=" | "-=" ;

logic_or_expr = logic_and_expr , { "or" , logic_and_expr } ;
logic_and_expr= equality_expr , { "and" , equality_expr } ;

equality_expr = relational_expr ,
                { ( "==" | "!=" ) , relational_expr } ;

relational_expr
             = additive_expr ,
               { ( "<" | "<=" | ">" | ">=" ) , additive_expr } ;

additive_expr = multiplicative_expr ,
                { ( "+" | "-" ) , multiplicative_expr } ;

multiplicative_expr
             = unary_expr ,
```

```
                { ( "*" | "/" ) , unary_expr } ;

unary_expr     = [ ( "not" | "-" ) ] , primary_expr ;

primary_expr   = literal
               | identifier
               | qualified_ident
               | call_expr
               | "(" , expr , ")" ;

qualified_ident
               = identifier , "." , identifier ;   (* e.g., fear.slow *)

call_expr      = identifier , "(" , [ arg_list ] , ")" ;

literal        = number | string | "true" | "false" ;
```

A.9 Debug and Introspection Primitives

```
trace_stmt     = ( "inspect" , expr
                 | "trace" , identifier , "for" , number , "beats"
                 | "debug::break" , "if" , expr
                 | "debug::watch" , expr ) , ";" ;
```

Appendix B — Formal Semantics

This appendix defines the **formal operational semantics** of HaackLang. Unlike traditional languages, HaackLang semantics incorporate:

- **Beat-driven execution**
- **Multi-track truthvectors**
- **Contextual overrides**
- **Meta-logic supervision**
- **Paraconsistent branching**

B.1 Configurations

A program configuration is:

```
C = (P, S, T, τ, Ctx, M)
```

P = program, S = store, T = track table, τ = Beat, Ctx = context stack, M = meta-state.

B.2 Track Semantics

Truthvectors evaluate per Track (classical, fuzzy, paraconsistent). Track fires if:

```
τ ≡ Φ_t (mod P_t) and not frozen
```

B.3 Expression Evaluation

Expressions evaluate differently per Track:

- classical → Boolean
- fuzzy → [0,1]
- paraconsistent → {0,1,{0,1}}

B.4 Beat-Step Rule

At each Beat:

1. τ ← τ+1
2. Determine firing Tracks
3. Execute rhythmic-gated statements
4. Update S on firing Tracks
5. Apply meta-logic
6. Update contexts

B.5 Context Semantics

Contexts push onto Ctx and override logic, priority, and meta parameters.

B.6 Rhythmic Guards

`when@track` executes only if Track fires. `if@phase(n)` fires only at specific phase.

B.7 Paraconsistent Branching

`if!!` forks computation into dual cognitive frames.

B.8 Frame Merging

Meta resolves divergent frames using:

- preserve contradictions
- dominant rules
- fuzzy blending

B.9 Meta Logic

Meta triggers every k Beats and adjusts Track weights, freezes, logic routes, context switches.

B.10 Summary

HaackLang semantics = temporal, multi-track, context-sensitive, meta-supervised cognitive transitions.

Appendix C — Complete HLVM Opcode Tables

C.1 Overview

This appendix defines the **complete opcode set** for the HaackLang Virtual Machine (HLVM). It is the canonical reference for implementers of:

- HLVM interpreters and JIT compilers
- HaackLang compilers and code generators
- Analysis and debugging tools

Each instruction is described by:

- **Mnemonic** — symbolic name
- **Opcode** — 8-bit opcode value (hex)
- **Track Field** — which Track(s) it typically targets

- **Format** — RR (reg–reg), RI (reg–imm), R (unary), SYS (system), META, CTX, etc.
- **Semantics** — high-level behavior, Beat-gating notes

HLVM instructions use a 32-bit base word with optional extensions (see Chapters 16–17).

C.2 Instruction Word Encoding

C.2.1 Base Format (32 bits)

```
+----------+----------+----------+----------------+
| OPCODE   |  TRACK   |  FLAGS   |  OPERAND SPEC  |
| (8 bits) | (4 bits) | (4 bits) |    (16 bits)   |
+----------+----------+----------+----------------+
```

- **OPCODE** — primary operation selector
- **TRACK** — target Track (MAIN/SLOW/SYNCOP/META/ALL)
- **FLAGS** — execution modifiers (Beat-gating, contradiction mode, etc.)
- **OPERAND SPEC** — encodes registers, immediates, or addressing modes

C.2.2 Operand Encodings

- **RR (reg–reg)**

  ```
  +--------+--------+
  | rA (8) | rB (8) |
  +--------+--------+
  ```

- **RI (reg–imm8)**

  ```
  +--------+----------------+
  | rA (8) | imm8 (8)       |
  +--------+----------------+
  ```

- **R (unary)**

```
+--------+----------------+
| rA (8) | unused (8)     |
+--------+----------------+
```

- **EXT (extended)** — multi-word encodings for 3+ operands.

C.3 Track Field Encoding

```
0x0  NONE / INVALID
0x1  MAIN       (classical ALU)
0x2  SLOW       (fuzzy ALU)
0x3  SYNCOP     (paraconsistent ALU)
0x4  META       (meta-coprocessor)
0xF  ALL_TRACKS (broadcast / global)
```

C.4 Flag Bits

Flags are 4 bits wide; each bit has a specific meaning.

```
bit 0  PHASE_GATED   (execute only when phase condition satisfied)
bit 1  CONTRA_MODE   (use contradiction-preserving path if available)
bit 2  META_ROUTE    (results routed via meta-coprocessor)
bit 3  CTX_HALT      (halts after context switch or exception)
```

C.5 Classical ALU Instructions (MAIN Track)

These operate on the MAIN Track's classical Boolean/bitvector representation.

Opcode	Mnemonic	Track	Format	Description
0x01	AND	MAIN	RR	rA ← rA ∧ rB (crisp boolean AND)
0x02	OR	MAIN	RR	rA ← rA ∨ rB
0x03	NOT	MAIN	R	rA ← ¬rA

Opcode	Mnemonic	Track	Format	Description
0x04	XOR	MAIN	RR	rA ← rA XOR rB
0x05	NAND	MAIN	RR	rA ← ¬(rA ∧ rB)
0x06	NOR	MAIN	RR	rA ← ¬(rA ∨ rB)
0x07	MOV	MAIN	RR	rA ← rB
0x08	SET	MAIN	RI	rA ← imm8 (0 or 1 for pure boolean)
0x09	CMP	MAIN	RR	set internal flags based on rA ? rB
0x0A	TEST	MAIN	RR	set flags based on rA ∧ rB

> **Beat Semantics:** Classical ALU instructions commit only when MAIN fires on the current Beat.

C.6 Fuzzy ALU Instructions (SLOW Track)

Operate over fuzzy values in [0,1] on the SLOW Track.

Opcode	Mnemonic	Track	Format	Description
0x20	F_AND	SLOW	RR	rA ← t-norm(rA, rB) (default: min)
0x21	F_OR	SLOW	RR	rA ← s-norm(rA, rB) (default: max)
0x22	F_NOT	SLOW	R	rA ← 1 − rA
0x23	F_BLEND	SLOW	RRI	rA ← (1−w)·rA + w·rB (w from imm8 ∈ [0,1])
0x24	F_DAMP	SLOW	RI	rA ← rA·k (k from imm8 ∈ [0,1])
0x25	F_GAIN	SLOW	RI	rA ← rA + k, clamped to [0,1]

Opcode	Mnemonic	Track	Format	Description
0x26	F_STEP	SLOW	RI	rA ← 1 if rA ≥ θ else 0 (θ from imm8)
0x27	F_SMOOTH	SLOW	RI	rA ← smoothed(rA, k) (e.g., logistic smoothing)

> **Beat Semantics:** Fuzzy updates reflect slow belief drift and commit only when SLOW Track fires.

C.7 Paraconsistent ALU Instructions (SYNCOP Track)

Operate on dual-valued or contradiction-aware representations.

Opcode	Mnemonic	Track	Format	Description
0x40	P_AND	SYNCOP	RR	rA ← paraconsistent AND(rA, rB)
0x41	P_OR	SYNCOP	RR	rA ← paraconsistent OR(rA, rB)
0x42	P_NOT	SYNCOP	R	rA ← paraconsistent negation of rA
0x43	P_BOTH	SYNCOP	R	rA ← {true,false} (explicit contradiction)
0x44	P_RES	SYNCOP	RI	resolve contradiction in rA using strategy imm8
0x45	P_TAG	SYNCOP	RI	tag rA with contradiction metadata (source, weight)

> Typical resolution strategies (imm8): 0 = preserve, 1 = classical, 2 = fuzzy-blend.

C.8 Truthvector & Cross-Track Operations

These instructions treat a register index as a **truthvector handle**, operating across Tracks.

Opcode	Mnemonic	Track	Format	Description
0x30	TV_COPY	ALL_TRACKS	RR	copy all track components of rB into rA
0x31	TV_ZERO	ALL_TRACKS	R	set all components of rA to 0
0x32	TV_ONE	ALL_TRACKS	R	set all components of rA to 1
0x33	TV_BLEND	ALL_TRACKS	RRI	componentwise blend of rA and rB with weight imm8
0x34	TV_NORM	ALL_TRACKS	R	normalize truthvector magnitude (implementation-def.)
0x35	TV_SNAP	ALL_TRACKS	RI	snap rA to logic L (imm8 selects classical/fuzzy/etc)

C.9 Rhythmic & Temporal Control

These opcodes manipulate Beat/phase scheduling and gating.

Opcode	Mnemonic	Track	Format	Description
0x60	WAIT	ALL	RI	suspend local thread for N Beats (imm8)
0x61	UNTIL_PHASE	ALL	RI	block until current Track phase = imm8

Opcode	Mnemonic	Track	Format	Description
0x62	WHEN	ALL	RI	conditionally execute instruction when Track fires
0x63	RHY_SYNC	ALL	RR	align phase of Track rA with Track rB
0x64	RHY_DESYNC	ALL	R	randomly jitter phase of Track associated with rA

> In many implementations, `WHEN` is encoded implicitly via FLAGS; the explicit opcode is provided for clarity.

C.10 Context & Scope Control

Context instructions interface with the Context Authority Layer (CAL).

Opcode	Mnemonic	Track	Format	Description
0x70	CTX_ENTER	ALL_TRACKS	RI	push context id = imm8 onto Ctx stack
0x71	CTX_EXIT	ALL_TRACKS	R	pop top context
0x72	CTX_OVR	ALL_TRACKS	RR	override Track/logic mapping using rA,rB
0x73	CTX_PRIORITY	ALL_TRACKS	RR	set priority ordering between Tracks in rA,rB
0x74	CTX_MASK	ALL_TRACKS	RI	temporarily mask (mute) context id = imm8

`CTX_OVR` and `CTX_PRIORITY` typically operate on small encoded descriptors in registers.

C.11 Meta-Logic Coprocessor Instructions

These opcodes drive the Meta-Logical Coprocessor (MLC).

Opcode	Mnemonic	Track	Format	Description
0x80	META_SNAP	META	RR	compute snapped-to-classical version of truthvector
0x81	META_COH	META	RR	rA ← coherence score of truthvector in rB
0x82	META_CONFL	META	RR	rA ← conflict(main,syncop) for truthvector in rB
0x83	META_BOOST	META	RI	increase weight of Track id = imm8
0x84	META_DAMP	META	RI	decrease weight of Track id = imm8
0x85	META_FREEZE	META	RI	freeze Track id = imm8
0x86	META_THAW	META	RI	thaw Track id = imm8
0x87	META_ROUTE	META	RR	route evaluation from Track in rA to Track in rB
0x88	META_LOG	META	RR	append meta event (in rB) into meta-log buffer

C.12 System, Interrupt, and Debug Instructions

These control HLVM lifecycle, traps, and debugger integration.

Opcode	Mnemonic	Track	Format	Description
0x90	NOP	ALL	SYS	no operation
0x91	HALT	ALL	SYS	stop VM execution
0x92	RESET	ALL	SYS	reset VM state (implementation-defined)
0x93	INT	ALL	RI	raise interrupt id = imm8
0x94	DEBUG	ALL	SYS	debugger hook (breakpoint/trap)
0x95	TRACE	ALL	RI	toggle tracing channel id = imm8

C.13 Reserved and Extension Opcodes

The following opcode ranges are reserved for future extensions and experimental features:

```
0x0B–0x1F  Extended classical operations (vector/matrix, bit-set ops)
0x28–0x2F  Advanced fuzzy ops (custom t-norms, defuzzification)
0x46–0x4F  Advanced paraconsistent ops (multi-source contradictions)
0x50–0x5F  Agent-level and messaging opcodes (multi-agent HLVM)
0x96–0x9F  Implementation-specific system instructions
```

Implementations **must not** repurpose reserved opcodes with conflicting semantics if interoperability is desired. Instead, they should document any custom extensions as profiles on top of this base spec.

C.14 Beat-Gating and Timing Notes

- Any opcode may be **silently skipped** on a non-firing Track; the semantics specify state change only when the target Track is active.
- Rhythmic and context instructions (`WAIT` , `UNTIL_PHASE` , `WHEN` , `CTX_*`) are always interpreted at the Beat level and can alter scheduling for future Beats.

- Meta opcodes execute according to the meta-Beat schedule and may influence the behavior of subsequent blocks of instructions without themselves being Beat-gated (implementation choice).

C.15 Summary

This appendix provided the canonical **opcode catalog** for HLVM, including:

- Binary encoding conventions
- Track and flag encodings
- Classical, fuzzy, and paraconsistent ALUs
- Truthvector and cross-track operations
- Rhythmic, context, and meta-logic control
- System, interrupt, and debug instructions
- Reserved spaces for future extensions

These tables, combined with the formal semantics and grammar in Appendices A and B, are sufficient to implement a fully compatible HaackLang runtime stack.