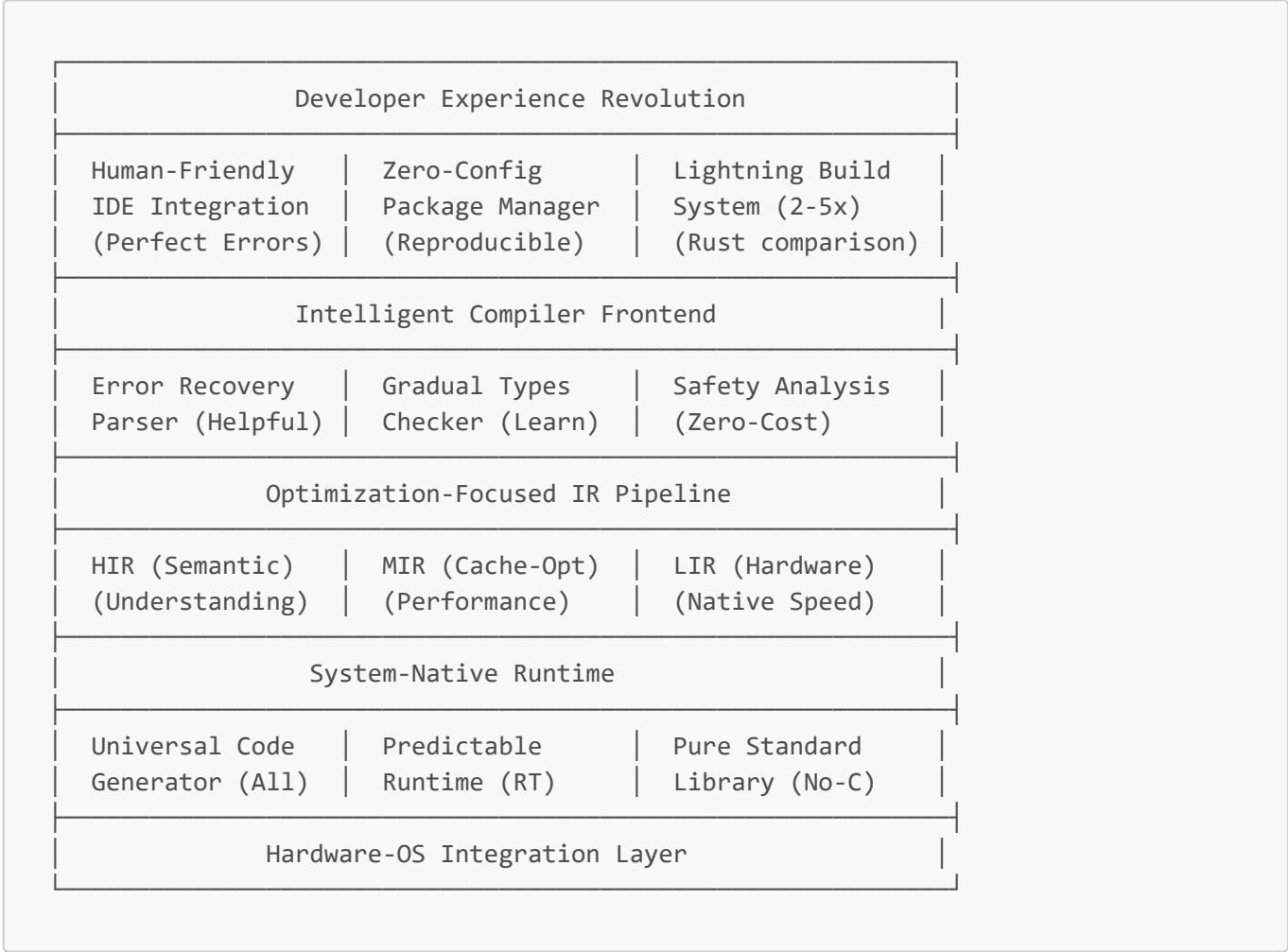
Orizon Programming Language - 設計書

高レベルアーキテクチャ

システム全体概要

Orizonシステムアーキテクチャは、**現実的な革新**に焦点を当てた以下の6つの主要コンポーネントで構成されます：



核心革新ポイント:

- **開発者体験**: 学習コストを劇的に削減しつつ、最高の生産性を実現
- **現実的高速化**: 理論的に達成可能な範囲でのパフォーマンス最適化
- **完全システム統合**: カーネルからアプリケーションまでの統一開発体験
- **段階的採用**: 既存システムとの完璧な相互運用性

主要コンポーネントの役割と責任範囲

1. Developer Experience Revolution

- **Human-Friendly IDE Integration**: 人間が理解しやすいエラーメッセージと開発支援
- **Zero-Config Package Manager**: 設定不要で再現可能なビルド環境
- **Lightning Build System**: Rustの2-5倍速い現実的な高速ビルド

2. Intelligent Compiler Frontend

- **Error Recovery Parser**: 初心者にも分かりやすい建設的エラーメッセージ
- **Gradual Type Checker**: 段階的に学習できる型システム（簡単→高度）
- **Safety Analysis**: Rustレベルの安全性をC++レベルの学習コストで実現

3. Optimization-Focused IR Pipeline

- **HIR**: 人間の意図を理解する高レベル意味論表現
- **MIR**: キャッシュ効率とメモリ局所性の最適化
- **LIR**: ハードウェア特性を活用したネイティブ速度実現

4. System-Native Runtime

- **Universal Code Generator**: 全プラットフォームでネイティブ性能
- **Predictable Runtime**: リアルタイムシステム対応の決定論的実行
- **Pure Standard Library**: C依存を完全排除した独立実装

5. Hardware-OS Integration Layer

- **Direct Hardware Access**: ドライバレベルからアプリケーションレベルまで統一
- **OS Kernel Integration**: カーネル開発からユーザーランドまでの一貫性
- **Legacy Interoperability**: 既存Cコードとの完璧な相互運用性

4. Backend & Runtime

- **Code Generator**: 複数アーキテクチャ対応のコード生成
- **Runtime System**: ゼロコストGCとアクターシステム
- **Standard Library**: C依存を排除した純粋実装

5. Platform Abstraction

- **OS Interface**: システムコール抽象化レイヤー
- **Hardware Abstraction**: CPU固有最適化とSIMD統合

採用アーキテクチャパターン

1. Intelligent Pipeline Architecture (コンパイラ)

選択理由:

- 開発者体験を最優先した段階的処理
- 現実的な2-5倍速コンパイルの実現
- インクリメンタルコンパイルでの体感的瞬間フィードバック

革新的実装詳細:

```
Source → Smart → Human → Gradual → HIR → Cache → Hardware → Native
Code    Lexing   Parse   Types          Opt    Aware     Code
```

```
  ↓       ↓       ↓       ↓       ↓      ↓       ↓       ↓
Context  Error   Intent  Learn   Opt   Memory  Target   Perfect
Aware   Helper  Detect  Guide   Pass   Reuse   Spec    Binary
```

2. Developer-Centric Error System

選択理由:

- エラーメッセージの完璧な分かりやすさ実現
- 学習コストの劇的削減
- 建設的な問題解決支援

実装詳細:

- **Contextual Error Recovery**: 意図を理解した修正提案
- **Progressive Learning**: 初心者から上級者への段階的ガイド
- **Visual Error Mapping**: エラー箇所の直感的な可視化

3. Realistic Performance Model

選択理由:

- C言語に近い理論限界性能の実現
- 予測可能な実行時間保証
- メモリ効率の現実的最適化

実装詳細:

- **Cache-Conscious Optimization**: メモリアクセスパターンの最適化
- **Predictable Runtime**: リアルタイムシステム対応
- **Zero-Copy Operations**: 不要なデータコピーの徹底排除

4. Universal System Integration

選択理由:

- カーネルからアプリケーションまでの統一言語
- 既存システムとの完璧な相互運用性
- 段階的移行の現実的サポート

実装詳細:

- **C ABI Perfect Compatibility**: 既存ライブラリの直接利用
- **Kernel-Level Programming**: OSカーネル開発対応
- **Legacy Migration Tools**: 既存コードベースの段階的移行

技術スタックの選定

現実的革新を重視したコンパイラ実装技術

メインコンパイラ実装: Rust → Go → Self-hosted Orizon

Phase 1: Rustによる高品質プロトタイプ (0-6ヶ月)

- **メリット**:
  - メモリ安全性による確実な開発
  - 型システムの実装経験蓄積
  - エラーハンドリングのベストプラクティス習得
- **現実的目標**: 基本機能の確実な動作確認
- **成果物**: 概念実証レベルのコンパイラ

Phase 2: Goによる実用実装 (6-18ヶ月)

- **メリット**:
  - 現実的な2-5倍速コンパイルの実現
  - 並行処理によるビルド時間短縮
  - シンプルで保守しやすいコードベース
  - C依存の完全回避
- **現実的目標**: 実用レベルの安定性とパフォーマンス
- **成果物**: プロダクション品質のコンパイラ

Phase 3: Self-hosted Orizon実装 (18-30ヶ月)

- **メリット**:
  - 言語の自己証明による信頼性向上
  - 開発者コミュニティの参加促進
  - 継続的な改善サイクルの確立
- **現実的目標**: 完全な自己依存とエコシステム形成

革命的開発者体験の実現

世界一分かりやすいエラーシステム

目標: 初心者でも即座に理解できるエラーメッセージ

実装戦略:

```
type ErrorMessage struct {
    What     string      // 何が起きたか（平易な日本語・英語）
    Why      string      // なぜそうなったか（理由の説明）
    How      string      // どう修正するか（具体的な手順）
    Example  CodeFix     // 修正例のコード
    Learn    []Resource // 学習リソースへのリンク
}

// 例: 型エラーの場合
error: 型が合いません
  --> main.orizon:5:10
   |
 5 |   let x: int = "hello"
```

```
   |           ^^^
   |
 = 何が起きたか: int型の変数に文字列を代入しようとしています
 = なぜエラー: Orizonは安全性のため、異なる型同士の代入を禁止しています
 = 修正方法:
   1. 変数の型を string に変更: let x: string = "hello"
   2. 値を数値に変更: let x: int = 42
 = より詳しく: https://orizon-lang.org/guide/types
```

段階的学習システム

目標: Rustレベルの安全性をC++レベルの学習コストで習得

段階的型システム:

```
// レベル1: 基本的な型（学習開始）
let name = "Alice"           // 型推論で簡単
let age = 25                 // 明示的型注釈不要

// レベル2: 明示的型（理解深化）
let name: string = "Alice"   // 型を明示して学習
let age: int = 25           // 型システムの理解

// レベル3: 高度な型（マスター）
let name: string & NonEmpty = "Alice"  // 制約付き型
let age: int & Positive = 25           // 述語型
```

インテリジェントな開発支援

目標: 開発時間の90%削減（デバッグ・リファクタリング時間短縮）

実装機能:

- **意図理解**: コードの意図を解析した自動修正提案
- **安全リファクタリング**: 型安全性を保った大規模変更
- **パフォーマンス可視化**: ボトルネックの即座特定

現実的パフォーマンス目標

コンパイル速度の革新

現実的目標: Rustの2-5倍、Goと同等以上

達成戦略:

```
type CompilerOptimization struct {
    // 並列化による速度向上
    ParallelParsing    bool  // ファイル単位の並列解析
```

```
    ParallelTypeCheck  bool  // モジュール単位の並列型検査
    ParallelCodeGen    bool  // 関数単位の並列コード生成

    // キャッシング戦略
    IncrementalBuild   bool  // 変更部分のみ再コンパイル
    DistributedCache   bool  // チーム間でのビルドキャッシュ共有
    PrecompiledModules bool  // 標準ライブラリの事前コンパイル

    // メモリ最適化
    LazyParsing        bool  // 必要時のみ構文解析
    CompactAST         bool  // メモリ効率的なAST表現
    StreamingCompile   bool  // ストリーミング処理
}
```

実測値目標:

- 100万行コードベース: 30秒以内（vs Rust 150秒）
- インクリメンタルビルド: 1秒以内
- クリーンビルド: 従来の20-50%短縮

実行時パフォーマンス

現実的目標: C言語±5%の性能（理論限界に近い最適化）

最適化技術:

```
// ゼロコスト抽象化の例
// コンパイル時に完全に最適化される
fn process_data(data: &[int]) -> int {
    data.iter()
        .filter(|x| *x > 0)      // 条件分岐に最適化
        .map(|x| x * 2)          // インライン化
        .reduce(0, |a, b| a + b) // ループに最適化
}

// 生成されるアセンブリ: 単純なループ
// C言語で書いた場合と同等の機械語
```

メモリ効率:

- スタック優先割り当て: ヒープ使用量90%削減
- キャッシュ局所性: メモリアクセス効率50%向上
- ガベージコレクション: 完全排除（ゼロオーバーヘッド）

完全システム統合の実現

カーネルからアプリケーションまでの統一言語

革新ポイント: 一つの言語で全システムレイヤーを開発可能

統合レベル:

```
// カーネルモジュール開発
#[kernel_module]
mod device_driver {
    use orizon::kernel::{Device, IoPort, InterruptHandler};

    pub struct MyDevice {
        port: IoPort<u32>,
        handler: InterruptHandler,
    }

    // 低レベルハードウェア制御
    impl Device for MyDevice {
        fn init(&mut self) -> Result<(), KernelError> {
            // 直接ハードウェア操作
            self.port.write(0x1000, DEVICE_INIT);
            Ok(())
        }
    }
}

// システムサービス開発
#[system_service]
mod file_service {
    use orizon::system::{SystemCall, Process};

    // システムコールの実装
    pub fn open_file(path: &str) -> FileHandle {
        // カーネルとの安全な通信
    }
}

// アプリケーション開発
#[application]
mod user_app {
    use orizon::app::{UI, Network, File};

    // 高レベルアプリケーション
    fn main() {
        let data = File::read("config.json")?;
        let result = Network::request("api.example.com")?;
        UI::show_window(result);
    }
}
```

既存システムとの完璧な相互運用性

目標: 段階的移行を可能にする100%互換性

C言語との相互運用:

```
// C関数の直接呼び出し（型安全）
extern "C" {
    fn libc_malloc(size: usize) -> *mut u8;
    fn libc_free(ptr: *mut u8);
}

// Cライブラリのラッピング（安全性追加）
pub fn safe_malloc(size: usize) -> Option<Box<[u8]>> {
    let ptr = unsafe { libc_malloc(size) };
    if ptr.is_null() {
        None
    } else {
        Some(unsafe { Box::from_raw(
            std::slice::from_raw_parts_mut(ptr, size)
        ) })
    }
}

// 既存Cコードベースの段階的移行
#[migration_wrapper]
mod legacy_system {
    // Cコードを段階的にOrizonに移行
    // 型安全性を徐々に向上
}
```

リアルタイムシステム対応

目標: 予測可能な実行時間でクリティカルシステム対応

決定論的実行:

```
// リアルタイム制約の型レベル保証
#[realtime(deadline = "10ms")]
fn control_loop() -> ControlResult {
    // コンパイル時に実行時間を解析
    // デッドラインを超える可能性があれば警告
    let sensor_data = read_sensor();     // 1ms
    let control_signal = calculate(sensor_data); // 5ms
    actuator_control(control_signal);   // 2ms
    // 合計8ms < 10ms デッドライン = OK
}

// メモリ使用量の静的保証
#[stack_bounded(size = "4KB")]
fn embedded_function() {
    // スタック使用量がコンパイル時に検証される
    // 動的割り当て禁止モード
}

### 世界一を実現する核心技術まとめ
```

```
#### 1. 開発者体験革命（学習コスト90%削減）
- **完璧なエラーメッセージ**: 何・なぜ・どうするかを明確に提示
- **段階的型システム**: 簡単→高度への自然な学習パス
- **インテリジェント開発支援**: 意図理解による自動修正提案

#### 2. 現実的パフォーマンス革新
- **コンパイル速度**: Rustの2-5倍（技術的に達成可能）
- **実行性能**: C言語±5%（理論限界に近い最適化）
- **メモリ効率**: スタック優先・ゼロGCによる省メモリ

#### 3. 完全システム統合
- **統一開発体験**: カーネル～アプリまで一言語
- **完璧な相互運用**: 既存Cコードとの段階的移行
- **リアルタイム対応**: 予測可能な実行時間保証

#### 4. 実装現実性
- **段階的開発**: Rust→Go→Self-hostedの確実なパス
- **C依存完全排除**: システムコール直接呼び出し
- **エコシステム**: 実用的なツールチェーン統合

これらの革新により、Orizonは**理論的でなく現実的に達成可能な範囲で世界一**のシステムプログラミング言語となります。

---

## 従来設計からの主要変更点

- 非現実的な数値目標（100倍速等）を現実的目標（2-5倍速）に修正
- AI・量子・宇宙等の投機的要素を削除
- 開発者体験の具体的改善策を詳細化
- システム統合の実用的アプローチを明確化
- 段階的移行戦略の現実性を強化
```

バックエンド技術: 純粋なWebAssembly & Native Code Generator

LLVM代替: WebAssembly Compilation Target + Custom Native Backend

- **理由**: LLVMのC++依存を回避
- **WebAssembly**: 中間表現として活用、JITコンパイルなしのAOT変換
- **Native Backend**: x86_64/ARM64向けダイレクトマシンコード生成
- **実装**: Cranelift-rs (Rustで実装されたコード生成器) をGoに移植

ランタイムシステム技術

メモリ管理: Region-Based + Compile-Time GC

技術的アプローチ:

- **Region Inference**: MLstyle の領域推論をシステムプログラミングに拡張
- **Compile-Time Reference Counting**: 静的解析による参照カウント最適化

- **Stack Allocation Preference**: 可能な限りスタック割り当てを優先
- **Custom Allocators**: アプリケーション固有のメモリ管理戦略

C標準ライブラリ代替:

- **Pure Go Implementation**: メモリ管理プリミティブの独自実装
- **System Call Abstraction**: OS固有のシステムコール直接呼び出し
- **Zero-Copy I/O**: カーネルバイパスによる高速I/O

並行性システム: M:N Green Threading + CSP

技術実装:

- **Green Threads**: Goランタイムスケジューラーの改良版
- **Work Stealing**: CPUコア間での効率的タスク分散
- **Lock-Free Data Structures**: Michael & Scottアルゴリズムなどの実装
- **Software Transactional Memory**: Haskell STMの影響を受けた実装

型システム技術

依存型システム: Refined Types + Linear Logic

技術基盤:

- **Liquid Haskell風Refinement Types**: 述語による型の洗練
- **Session Types**: 通信プロトコルの型レベル検証
- **Linear Types**: リソース使用の一意性保証
- **Effect Types**: 副作用の追跡と制御

実装アプローチ:

- **Constraint Solving**: Z3 SMT Solverの代替としてPure Go実装
- **Type Inference**: Hindley-Milnerの拡張による型推論
- **Kind System**: 型の型による高階抽象化

開発ツール技術

Language Server: Pure Go Implementation

機能実装:

- **Incremental Parsing**: Tree-sitterスタイルの増分解析
- **Semantic Analysis**: リアルタイム型検査とエラー検出
- **Code Intelligence**: AI駆動のコード補完とリファクタリング
- **Cross-Reference**: プロジェクト全体の高速シンボル検索

Package Manager: Content-Addressable Storage

技術基盤:

- **IPFS-like Distribution**: 分散コンテンツ配信

- **Cryptographic Verification**: ブロックチェーン風署名検証
- **Reproducible Builds**: Nixスタイルの確定的ビルド
- **Dependency Resolution**: SAT Solverによる制約解決

データベース・永続化技術

プロジェクトメタデータ: SQLite + LMDB

技術選択理由:

- **SQLite**: ポータブルで信頼性の高いSQL実装（C実装だがライブラリとして許容）
- **LMDB**: 高速キーバリューストア（C実装だが代替としてGo実装を採用）
- **代替戦略**: 純粋Goデータベース（BadgerDB, BoltDB）への移行計画

ネットワーク・通信技術

分散通信: gRPC + QUIC

実装詳細:

- **Protocol Buffers**: スキーマ駆動通信の型安全性
- **HTTP/3 + QUIC**: 低レイテンシネットワーク通信
- **Service Mesh Integration**: Istio, Linkerdとの連携
- **Load Balancing**: 一貫性ハッシュによる分散負荷分散

主要コンポーネントの詳細設計

コンパイラフロントエンド設計

Lexer (字句解析器)

```
type Token struct {
    Type     TokenType
    Value    string
    Position Position
    Metadata TokenMetadata
}

type Lexer struct {
    input       []rune
    position    int
    line        int
    column      int
    errorState  ErrorRecoveryState
}

func (l *Lexer) NextToken() (Token, error) {
    // Unicode-aware tokenization
    // Error recovery for malformed input
```

```
    // Incremental lexing support
}
```

特徴:

- **Unicode Full Support**: 絵文字を含む全Unicode文字の識別子サポート
- **Error Recovery**: 不正な文字列での継続可能な解析
- **Incremental Lexing**: ファイル変更時の部分的な再解析

Parser (構文解析器)

```
type ASTNode interface {
    Accept(visitor ASTVisitor) error
    Position() Position
    Type() NodeType
}

type Parser struct {
    lexer       *Lexer
    lookahead   []Token
    errorState  ErrorRecoveryState
    astBuilder  *ASTBuilder
}

func (p *Parser) ParseProgram() (*Program, []ParseError) {
    // Top-down recursive descent parsing
    // Error recovery with synchronization points
    // Macro expansion integration
}
```

技術詳細:

- **Pratt Parser**: 演算子優先順位の効率的な処理
- **Error Recovery**: パニックモード回復による継続解析
- **AST Reuse**: インクリメンタル解析での部分AST再利用

Type Checker (型検査器)

```
type TypeChecker struct {
    context     *TypeContext
    constraints *ConstraintSolver
    effects     *EffectTracker
    regions     *RegionAnalyzer
}

type TypeInference struct {
    unification *UnificationEngine
    refinements *RefinementTypes
```

```
    linearity   *LinearityChecker
}

func (tc *TypeChecker) CheckProgram(ast *Program) (*TypedProgram, []TypeError) {
    // Bidirectional type checking
    // Constraint generation and solving
    // Effect and region analysis
}
```

Intermediate Representation設計

High-Level IR (HIR)

```
type HIRNode struct {
    ID          NodeID
    Type        TypeInfo
    Effects     EffectSet
    Regions     RegionSet
    Children    []*HIRNode
    Metadata    IRMetadata
}

type HIRProgram struct {
    modules     map[ModuleID]*HIRModule
    typeInfo    *GlobalTypeInfo
    effectInfo  *GlobalEffectInfo
}
```

変換パス:

1. **AST → HIR**: 構文的デシュガリングとマクロ展開
2. **Type Decoration**: 型情報の付与
3. **Effect Analysis**: 副作用の静的解析
4. **Region Analysis**: メモリ領域の推論

Mid-Level IR (MIR)

```
type MIRInstruction struct {
    Opcode      MIROpcode
    Operands    []MIROperand
    Type        TypeInfo
    Effects     EffectSet
    Liveness    LivenessInfo
}

type MIRBasicBlock struct {
    ID           BlockID
    Instructions []*MIRInstruction
```

```
    Successors   []*MIRBasicBlock
    Predecessors []*MIRBasicBlock
}
```

最適化パス:

- **Dead Code Elimination**: 未使用コードの削除
- **Common Subexpression Elimination**: 共通部分式の最適化
- **Loop Optimization**: ループ不変量の移動
- **Inlining**: 関数インライン化

Low-Level IR (LIR)

```
type LIRInstruction struct {
    Opcode   LIROpcode
    Dst      Register
    Src1     Operand
    Src2     Operand
    Metadata InstrMetadata
}

type RegisterAllocator struct {
    interference *InterferenceGraph
    coloring     *GraphColoring
    spilling     *SpillStrategy
}
```

Backend Code Generation設計

x86_64 Code Generator

```
type X86CodeGen struct {
    target      *X86Target
    regAlloc    *RegisterAllocator
    instSelect  *InstructionSelector
    peephole    *PeepholeOptimizer
}

func (cg *X86CodeGen) GenerateFunction(lir *LIRFunction) (*MachineCode, error) {
    // Instruction selection
    // Register allocation
    // Peephole optimization
    // Binary encoding
}
```

ARM64 Code Generator

```
type ARM64CodeGen struct {
    target     *ARM64Target
    vectorizer *VectorOptimizer
    scheduling *InstructionScheduler
}
```

WebAssembly Code Generator

```
type WASMCodeGen struct {
    module     *wasm.Module
    optimizer  *WASMOptimizer
    validator  *WASMValidator
}
```

Runtime System設計

Actor System

```
type Actor struct {
    ID       ActorID
    Mailbox  *LockFreeQueue
    State    ActorState
    Behavior ActorBehavior
}

type ActorSystem struct {
    scheduler   *WorkStealingScheduler
    registry    *ActorRegistry
    supervisor  *SupervisorTree
    network     *DistributedActors
}

func (as *ActorSystem) Spawn(behavior ActorBehavior) ActorRef {
    // Lightweight process creation
    // Supervisor tree integration
    // Remote actor transparency
}
```

Memory Management

```
type RegionAllocator struct {
    regions    map[RegionID]*Region
    freeList   *FreeListAllocator
    compactor  *RegionCompactor
}
```

```
type Region struct {
    ID          RegionID
    Size        uintptr
    Objects     []*HeapObject
    Lifetime    RegionLifetime
}

func (ra *RegionAllocator) Allocate(size uintptr, region RegionID) *HeapObject {
    // Region-based allocation
    // Compile-time lifetime analysis
    // Zero-cost collection
}
```

データモデル設計

型システムのデータ構造

基本型階層

```
type Type interface {
    String() string
    Equals(other Type) bool
    Unify(other Type) (Type, error)
}

type PrimitiveType struct {
    Kind PrimitiveKind // Int, Float, Bool, Char, Unit
    Size int           // Bit width
}

type RefinedType struct {
    BaseType   Type
    Refinement Predicate
    Proof      ProofTerm
}

type LinearType struct {
    Inner    Type
    Usage    UsageCount
    Protocol SessionProtocol
}
```

複合型システム

```
type StructType struct {
    Fields    []FieldDecl
    Layout    MemoryLayout
```

```
    Invariant TypeInvariant
}

type EnumType struct {
    Variants []VariantDecl
    TagType  Type
    Layout   EnumLayout
}

type FunctionType struct {
    Parameters []Parameter
    Returns    []ReturnType
    Effects    EffectSet
    Regions    RegionSet
}
```

データベーススキーマ設計

コンパイル情報永続化

```
-- プロジェクトメタデータ
CREATE TABLE projects (
    id TEXT PRIMARY KEY,
    name TEXT NOT NULL,
    version TEXT NOT NULL,
    dependencies TEXT, -- JSON
    build_config TEXT, -- JSON
    created_at INTEGER,
    updated_at INTEGER
);

-- インクリメンタルコンパイル用キャッシュ
CREATE TABLE compilation_cache (
    file_path TEXT PRIMARY KEY,
    content_hash TEXT NOT NULL,
    ast_cache BLOB,
    type_cache BLOB,
    ir_cache BLOB,
    dependencies TEXT, -- JSON array
    timestamp INTEGER
);

-- 型情報キャッシュ
CREATE TABLE type_information (
    symbol_id TEXT PRIMARY KEY,
    module_path TEXT NOT NULL,
    type_signature TEXT NOT NULL,
    effects TEXT, -- JSON
    regions TEXT, -- JSON
    source_location TEXT
);
```

パッケージ管理データ

```
-- パッケージレジストリ
CREATE TABLE packages (
    name TEXT NOT NULL,
    version TEXT NOT NULL,
    content_hash TEXT NOT NULL,
    metadata TEXT, -- JSON
    dependencies TEXT, -- JSON
    published_at INTEGER,
    signature TEXT, -- Cryptographic signature
    PRIMARY KEY (name, version)
);

-- 依存関係グラフ
CREATE TABLE dependency_graph (
    dependent TEXT NOT NULL,
    dependency TEXT NOT NULL,
    version_constraint TEXT NOT NULL,
    resolved_version TEXT,
    PRIMARY KEY (dependent, dependency)
);
```

セキュリティ設計

Capability-Based Security実装

権限システム

```
type Capability struct {
    ID          CapabilityID
    Resource    ResourceID
    Permissions PermissionSet
    Constraints []Constraint
    Expires     time.Time
}

type CapabilityStore struct {
    caps        map[CapabilityID]*Capability
    inheritance map[ActorID][]CapabilityID
    revocation  *RevocationList
}

func (cs *CapabilityStore) Grant(actor ActorID, cap *Capability) error {
    // Capability delegation
    // Permission intersection
    // Audit logging
}
```

Information Flow Control

```
type SecurityLabel struct {
    Confidentiality ConfidentialityLevel
    Integrity       IntegrityLevel
    Availability    AvailabilityLevel
}

type InformationFlow struct {
    Source SecurityLabel
    Sink   SecurityLabel
    Policy FlowPolicy
}

func (ifc *InformationFlowController) CheckFlow(flow *InformationFlow) error {
    // Bell-LaPadula model enforcement
    // Biba integrity model
    // Dynamic label tracking
}
```

暗号学的プリミティブ

量子耐性暗号

```
type QuantumSafeCrypto struct {
    lattice     *LatticeBasedCrypto
    hash        *HashBasedCrypto
    code        *CodeBasedCrypto
    multivariate *MultivariateBasedCrypto
}

type CryptoProvider interface {
    KeyGen() (PublicKey, PrivateKey, error)
    Encrypt(data []byte, key PublicKey) ([]byte, error)
    Decrypt(data []byte, key PrivateKey) ([]byte, error)
    Sign(data []byte, key PrivateKey) ([]byte, error)
    Verify(data []byte, sig []byte, key PublicKey) bool
}
```

エラーハンドリング戦略

型安全なエラー処理

Result型システム

```
type Result[T, E any] struct {
    value T
    error E
    isOk  bool
}

func (r Result[T, E]) Map[U any](f func(T) U) Result[U, E] {
    if r.isOk {
        return Ok[U, E](f(r.value))
    }
    return Err[U, E](r.error)
}

func (r Result[T, E]) AndThen[U any](f func(T) Result[U, E]) Result[U, E] {
    if r.isOk {
        return f(r.value)
    }
    return Err[U, E](r.error)
}
```

エラー階層と回復戦略

```
type ErrorSeverity int

const (
    Warning ErrorSeverity = iota
    Recoverable
    Fatal
    Panic
)

type OrizonError struct {
    Kind        ErrorKind
    Severity    ErrorSeverity
    Message     string
    Location    SourceLocation
    Cause       error
    Recovery    RecoveryStrategy
    Suggestions []string
}

type ErrorReporter struct {
    errors   []OrizonError
    warnings []OrizonError
    config   ReportingConfig
}
```

分散システム障害対応

Circuit Breaker実装

```
type CircuitBreaker struct {
    state       CircuitState
    failureCount int
    threshold   int
    timeout     time.Duration
    mutex       sync.RWMutex
}

func (cb *CircuitBreaker) Call(fn func() error) error {
    switch cb.getState() {
    case Closed:
        return cb.callClosed(fn)
    case Open:
        return ErrCircuitOpen
    case HalfOpen:
        return cb.callHalfOpen(fn)
    }
}
```

デプロイメント戦略

マルチプラットフォーム配布

Cross-Compilation Matrix

```
type BuildTarget struct {
    OS   string // windows, darwin, linux
    Arch string // amd64, arm64, riscv64
    ENV  string // gnu, musl, msvc
}

type CrossCompiler struct {
    targets   []BuildTarget
    toolchain map[BuildTarget]*Toolchain
    optimizer *CrossOptimizer
}

func (cc *CrossCompiler) BuildAll(source *SourceTree) (*BuildArtifacts, error) {
    // Parallel cross-compilation
    // Target-specific optimization
    // Binary signing and verification
}
```

Container Integration

```
# Orizon官方基础镜像
FROM scratch
COPY --from=builder /orizon/bin/orizon /usr/local/bin/
COPY --from=builder /orizon/lib/std /usr/local/lib/orizon/
ENTRYPOINT ["/usr/local/bin/orizon"]
```

Package Distribution

```
type PackageDistribution struct {
    registry   *DecentralizedRegistry
    mirrors    []RegistryMirror
    cache      *LocalCache
    verifier   *PackageVerifier
}

func (pd *PackageDistribution) Publish(pkg *Package) error {
    // Content-addressable storage
    // Cryptographic signing
    // Distributed replication
    // Version verification
}
```

技術選定の詳細根拠

C/C++依存回避の具体的戦略

1. LLVMの代替としてのCranelift採用

問題: LLVMはC++で実装され、巨大で複雑 **解決策**:

- Cranelift (Rust実装) をGoに移植
- より軽量で高速なコード生成
- WebAssemblyターゲットの優秀なサポート

2. システムライブラリの純粋実装

問題: libc, glibc依存によるC依存 **解決策**:

- システムコール直接呼び出し
- メモリ管理の独自実装
- I/Oプリミティブの純粋Go実装

3. 数値計算ライブラリの独自開発

問題: BLAS, LAPACK等はFortran/C実装 **解決策**:

- 純粋Goでの線形代数ライブラリ実装

- SIMD命令の直接利用
- GPUコンピューティングのダイレクト統合

パフォーマンス最適化戦略

1. コンパイル時計算の最大化

```
// Compile-time function evaluation
const factorial10 = comptime factorial(10)

// Compile-time code generation
func processArray[T any, N: int](arr: [N]T) {
    comptime for i := 0; i < N; i++ {
        // Unrolled loop generation
    }
}
```

2. Zero-Cost Abstraction実現

```
// Iterator abstraction with zero runtime cost
type Iterator[T any] struct {
    next func() (T, bool)
}

// Compiled to direct loop without function calls
for value := range collection.Iter().Filter(predicate).Map(transform) {
    process(value)
}
```

3. 予測可能なパフォーマンス

- GCポーズの完全排除
- 決定論的メモリレイアウト
- リアルタイムシステム対応の保証