# WebAssembly 3.0: 2025年12月時点におけるアーキテクチャ、表現形式、およびエコシステムの包括的技術報告書

## 1. 序論：WebAssembly 3.0の策定とユニバーサルランタイムへの進化

2025年12月現在、WebAssembly（Wasm）は当初の「ブラウザ内での高速実行」という目的を超越し、クラウドネイティブ、エッジコンピューティング、そして組み込みシステムに至るまで、あらゆる環境で動作する「ユニバーサルなバイナリフォーマット」としての地位を確立しています。W3C WebAssemblyワーキンググループは、2025年12月8日にWebAssembly Core Specification Version 3.0の草案を公開しました [1]。このマイルストーンは、単なるバージョン番号の更新にとどまらず、ガベージコレクション（WasmGC）、64ビットメモリアドレス空間（Memory64）、そしてコンポーネントモデルの標準化といった、長年議論されてきた機能の実用化を意味しています [2]。
本報告書は、2025年末時点でのWebAssemblyの仕様、特にテキスト表現（WAT）とバイナリ表現（WASM）の構造的詳細、メモリ管理のパラダイムシフトをもたらすWasmGC、そしてモジュール間連携の革新であるコンポーネントモデルについて、技術的な観点から包括的に分析するものです。これにより、システムアーキテクトやコンパイラ開発者が、現在のWebAssemblyエコシステムを深く理解し、次世代の分散システム設計に応用するための基礎資料となることを目指します。

## 2. WebAssemblyの設計思想とCore Specification 3.0の現状

WebAssemblyは、スタックベースの仮想マシン用バイナリ命令形式として設計されました。その設計目標は、高速で効率的な実行、安全性、ハードウェア独立性、そしてオープンなデバッグ可能性にあります [1]。2025年のWebAssembly 3.0では、これらの基本原則を維持しつつ、より高度な言語機能と大規模なメモリ空間への対応が強化されています。

### 2.1 セキュリティと分離モデル

WebAssemblyのセキュリティモデルは、ホスト環境からの厳格な分離（サンドボックス化）に基づいています。Wasmモジュールは、ホストのメモリやリソース（ファイルシステム、ネットワークなど）に直接アクセスすることはできず、明示的にインポートされた関数やテーブルを通じてのみ相互作用が可能です [1]。この「capability-based security（ケイパビリティベースのセキュリティ）」は、コンポーネントモデルにおいてさらに強化され、サプライチェーン攻撃に対する堅牢な防御策として機能しています。コードは検証され、メモリセーフな環境で実行されるため、データ破損やセキュリティ侵害のリスクが最小限に抑えられます [1]。

### 2.2 バージョン3.0における主要な機能拡張

2022年に完了したバージョン2.0（SIMD、Bulk Memory Operationsなどを導入）に続き、3.0では以下の機能が標準化（Phase 5）または実装完了段階にあります [3]:

- **Garbage Collection (WasmGC):** 線形メモリ（Linear Memory）とは異なる、ホストVM（ブラウザのJSエンジンなど）のガベージコレクタによって管理される構造体や配列を定義可能にします。
- **Memory64:** 従来の32ビットアドレス空間（4GB制限）を撤廃し、64ビット整数によるメモリアドレッシングを可能にします。これにより、巨大なデータセットやAIモデルのオンメモリ処理が可能になります [2]。
- **Relaxed SIMD:** 厳密な決定論的動作を緩和し、特定のハードウェア（x64のSSE/AVXやArmのNEON/SVE）のネイティブ性能を最大限に引き出すためのベクトル演算命令セットです [3]。
- **Tail Calls (末尾呼び出し最適化):** 再帰的な関数呼び出しにおいて新しいスタックフレームを作成せず、ジャンプとして処理することでスタックオーバーフローを防ぎます。これは関数型言語のサポートに不可欠です [6]。

以下の表は、2025年時点での主要な機能提案のステータスをまとめたものです。

| 機能名 | ステータス (2025年12月) | 概要 |
|---|---|---|
| **Garbage Collection** | Phase 5 (Standardized) | 管理されたメモリ構造（struct, array）の導入 [6] |
| **Tail Call** | Phase 5 (Standardized) | 末尾再帰の最適化 [6] |
| **Relaxed SIMD** | Phase 5 (Standardized) | 非決定論的な高速ベクトル演算 [3] |
| **Multiple Memories** | Phase 5 (Standardized) | 1つのモジュールで複数の線形メモリを使用可能 [6] |
| **Memory64** | Phase 4/5 | 64ビットアドレス空間のサポート [6] |
| **Exception Handling** | Phase 4/5 | ゼロコスト例外処理（exnrefの導入）[6] |

# 3. WebAssemblyテキスト形式 (WAT) の詳細分析

WebAssemblyテキスト形式（WebAssembly Text Format: WAT）は、バイナリ形式と1対1に対応する中間表現であり、人間が可読・編集可能なS式（S-expressions）として定義されています [8]。これはLispに似た構造を持ち、フラットな命令列を木構造として表現することを可能にします。

## 3.1 S式によるモジュール構造

WATにおける最小単位はモジュールであり、(module...)というS式で全体がラップされます。モジュール内には、関数、メモリ、グローバル変数、エクスポート、インポートなどの定義がネストされたS式として記述されます。

コード スニペット

```
(module
  (func $add (param $lhs i32) (param $rhs i32) (result i32)
    local.get $lhs
    local.get $rhs
    i32.add)
  (export "add" (func $add))
)
```

この例では、funcキーワードで関数を定義し、$addという識別子を付与しています。パラメータと戻り値の型は明示的に宣言され、関数本体にはスタックマシンの命令が記述されています [10]。

## 3.2 命令記述の二面性：Linear vs. Folded

WATには、命令を記述する際に2つのスタイルが存在します。

1. スタックマシン形式 **(Linear):** バイナリ形式の命令順序をそのまま記述する方法です。オペランドをスタックに積み（push）、命令がそれを消費（pop）します。
   コード スニペット
   ```
   local.get $a
   local.get $b
   i32.add
   ```

2. **S式形式 (Folded):** 命令をネストさせて記述する方法です。視覚的に処理の流れが追いやすくなりますが、アセンブラによってフラットな命令列に変換されます。
   コード スニペット
   ```
   (i32.add (local.get $a) (local.get $b))
   ```

   この柔軟性により、開発者は低レベルなスタック操作のデバッグと、高レベルなロジックの記述の両方に対応できます [8]。

## 3.3 型システムとWasmGCによる拡張

Core WebAssembly（MVP）の型システムは、数値型（i32, i64, f32, f64）とベクトル型（v128）、および基本的な参照型（funcref, externref）に限定されていました [8]。しかし、Wasm 3.0におけるWasmGCの導入により、テキスト形式の型定義能力は飛躍的に向上しました。

### 3.3.1 構造体と配列の定義

WasmGCでは、typeセクション内でstructやarrayを定義できるようになりました。これにより、メモリレイアウトをWasmレベルで記述できます。

- 構造体 **(Struct):** フィールドのリストを持ち、各フィールドはミュータビリティ（可変性）を持つことができます。
  コード スニペット
  ```
  (type $Point (struct (field $x (mut f64)) (field $y (mut f64))))
  ```

  上記は、2つの可変な64ビット浮動小数点数を持つPoint構造体の定義です [13]。

- 配列 **(Array):** 同一の型を持つ要素のシーケンスです。
  コード スニペット
  (type $IntArray (array (mut i32)))

  これは可変な32ビット整数の配列を定義しています [13]。

これらの新しい型は、従来の線形メモリ上のオフセット計算ではなく、オブジェクト参照として扱われます。例えば、構造体のインスタンスを生成するにはstruct.new命令を使用し、フィールドへのアクセスにはstruct.getを使用します。これらの命令もWAT上で表現可能です [14]。

### 3.4 制御構造と識別子

WATでは、$identifier形式の識別子を使用することで、関数や変数を数値インデックス（0, 1, 2...）ではなく名前で参照できます。アセンブル時にこれらは自動的に数値インデックスに解決されます。制御構造にはblock、loop、ifがあり、構造化された制御フローを提供します。ラベルを用いた分岐（br、br_if）も可能で、これにより複雑なロジックも効率的に表現されます [10]。

## 4. WebAssemblyバイナリ形式の詳細構造

WebAssemblyのバイナリ形式（.wasmファイル）は、転送サイズとデコード速度を最適化するために設計された高密度なフォーマットです。2025年現在、基本となるCore Moduleのバイナリ構造はバージョン1（MVP）の骨格を維持しつつ、新しいセクションIDやオペコードの追加によって拡張されています。

### 4.1 モジュールヘッダとマジックナンバー

すべてのWasmバイナリファイルは、以下の8バイトのヘッダで始まります [16]。

- **Magic Number (4バイト):** 0x00 0x61 0x73 0x6D (\0asm)。これがファイルをWasmモジュールとして識別させます。
- **Version (4バイト):** 0x01 0x00 0x00 0x00 (リトルエンディアンで1)。WebAssembly 3.0の仕様下であっても、Core Module形式のファイルである限り、後方互換性を保つためにバージョン番号は1のまま維持されるのが一般的です。

### 4.2 LEB128可変長整数エンコーディング

バイナリサイズを削減するため、Wasm内のすべての整数（セクションサイズ、インデックス、即値など）は**LEB128 (Little Endian Base 128)** 形式でエンコードされます [19]。

- 仕組み**:** 整数を7ビットごとのグループに分割し、各バイトの最上位ビット（MSB）を「続きがあるか」のフラグとして使用します。
- 例**:** 数値が小さければ1バイトで表現でき、32ビット整数であっても値が小さければ数バイトで済みます。
  - 符号なし（ULEB128）: インデックスやサイズに使用。
  - 符号付き（SLEB128）: i32.constなどのリテラル値に使用。

### 4.3 セクション構造とID一覧

ヘッダの後には、一連のセクションが続きます。各セクションはSection ID（1バイト）、Size（ULEB128

)、Content（バイト列）で構成されます。標準的なセクションIDは以下の通りです [18]。

| ID | セクション名 | 内容と役割 |
|---|---|---|
| 0 | **Custom** | デバッグ情報、ソースマップ、ツール固有のメタデータ。配置順序は自由。 |
| 1 | **Type** | 関数シグネチャ、およびWasmGCの構造体・配列型の定義。 |
| 2 | **Import** | 外部からインポートする関数、メモリ、テーブル、グローバルの宣言。 |
| 3 | **Function** | Typeセクションのシグネチャと Codeセクションのボディを紐付けるインデックス。 |
| 4 | **Table** | 関数参照などを格納するテーブルの定義（funcref, externref）。 |
| 5 | **Memory** | 線形メモリの定義（初期ページ数、最大ページ数）。 |
| 6 | **Global** | グローバル変数の定義。 |
| 7 | **Export** | 外部に公開する関数、メモリなどの宣言。 |
| 8 | **Start** | モジュールロード時に自動実行される開始関数のインデックス。 |
| 9 | **Element** | テーブルの初期化データ（関数参照のリストなど）。 |
| 10 | **Code** | 関数の実際のバイトコード本体。 |
| 11 | **Data** | 線形メモリの初期化データ。 |
| 12 | **DataCount** | Bulk Memory操作のためのデータセグメント数（検証を高速化するため）。 |
| 13 | **Tag** | 例外処理用のタグ定義（Exception Handling提案による追加）。 |

セクションID 1〜13は、Customセクションを除き、この順序で出現しなければなりません。これにより、ストリーミングコンパイル（ダウンロードしながらのコンパイル）が可能になります。例えば、関数の型情報（Type）やインポート情報（Import）は、関数ボディ（Code）が到着する前に処理できるため、コード生成を並行して行えます。[1]

## 4.4 命令（オペコード）のバイナリ表現

Wasmの命令は通常1バイトのオペコードで表現されます。2025年時点での主要なオペコード体系は

以下のようになっています [24]。

- 基本命令:
  - 制御構造: block (0x02), loop (0x03), if (0x04), end (0x0B)。
  - 変数操作: local.get (0x20), local.set (0x21)。
  - 数値演算: i32.const (0x41), i32.add (0x6A)。
- プレフィックス命令:
  命令セットの拡張に伴い、1バイトでは不足するため、特定のプレフィックスバイトを使用する命令群が増えています。
  - **SIMD (Vector):** プレフィックス 0xFD。128ビットベクトル演算に使用。
  - **Atomic:** プレフィックス 0xFE。スレッド間同期に使用。
  - **GC (Garbage Collection):** プレフィックス 0xFB。構造体や配列の操作に使用。

# 5. WebAssembly Component Model：次世代の相互運用性

2025年のWebAssemblyエコシステムにおいて、最も重要な技術的進歩は**Component Model（コンポーネントモデル）**の標準化と普及です。これは従来のCore Moduleの限界を打ち破り、言語間の壁を取り払うための新しいバイナリ仕様です [27]。

## 5.1 Core ModuleとComponentの違い

従来のCore Moduleは、数値（整数・浮動小数点数）のみを引数や戻り値として扱うことができました。文字列やリストのような複雑なデータをやり取りするには、メモリを共有し、ポインタを渡す「グルーコード」を手動で書く必要があり、これが言語間連携の大きな障壁となっていました。
Component Modelは、モジュールの上にさらに高レベルなラッパー層を提供します。

- 共有ナッシング **(Shared-Nothing):** コンポーネント間ではメモリを共有せず、明確に定義されたインターフェースを通じてのみ通信します。これによりセキュリティと分離性が向上します [29]。
- 高レベル型**:** 文字列（String）、レコード（Record）、バリアント（Variant）、リスト（List）などのリッチな型をネイティブに扱えます。
- **Canonical ABI:** これらの高レベル型を、Core Moduleが理解できる低レベルな数値型に変換（Lowering）したり、逆に復元（Lifting）したりするための標準的なメモリレイアウトと呼び出し規約を定めています [30]。

## 5.2 コンポーネントのバイナリ構造

Component Modelは、Core Moduleとは異なるバイナリ構造を持ちます。マジックナンバーは同じですが、**Layer**フィールドの値が異なります [32]。

- **Core Module:**
  - Magic: 0x00 0x61 0x73 0x6D
  - Version: 0x01 0x00 0x00 0x00
- **Component:**
  - Magic: 0x00 0x61 0x73 0x6D
  - Version: 0x0d 0x00 (2025年現在、ドラフト版として0x0dなどが使用されることが多い

が、正式化に向けてバージョン管理されている)
    - **Layer:** 0x01 0x00 (このフィールドにより、ランタイムはこれがコンポーネントであることを識別します)。

## 5.3 コンポーネント固有のセクションID

コンポーネントバイナリには、Core Moduleには存在しない独自のセクションIDが割り当てられています。これにより、モジュールのネストやインスタンス化を定義します [33]。

| ID | セクション名 | 役割 |
|---|---|---|
| 0 | Custom | メタデータ。 |
| 1 | **Module** | Core Moduleそのものを内包(埋め込み)します。 |
| 2 | **Instance** | モジュールや他のコンポーネントをインスタンス化します。 |
| 3 | **Alias** | インスタンスのエクスポートを内部的なインデックスとしてエイリアスします。 |
| 4 | **Type** | Component Model用の型定義(インターフェース型など)。 |
| 5 | **Component** | ネストされたコンポーネントの定義。 |
| 6 | **Function** | Canonical ABIに基づく関数の定義(Lifting/Lowering)。 |
| 10 | **Import** | 高レベルインターフェース(WITベース)のインポート。 |
| 11 | **Export** | 高レベルインターフェースのエクスポート。 |

この構造により、例えばRustで書かれたコンポーネントの中に、C言語で書かれたライブラリ(Core Module)を内包し、外部にはPythonから呼び出し可能なインターフェースを公開するといった構成が可能になります。

## 5.4 WIT (WebAssembly Interface Type)

コンポーネントのインターフェース定義には、**WIT**というIDL(インターフェース記述言語)が使用されます [36]。

コード スニペット

```
package my-org:calculator;

interface operations {
```

```
  add: func(a: u32, b: u32) -> u32;
  log: func(msg: string);
}

world my-app {
  import operations;
  export run: func();
}
```

このWITファイルから、wit-bindgenなどのツールを使用して、各言語用のバインディング（グルーコード）が自動生成されます。開発者は複雑なABIを意識することなく、ネイティブな関数呼び出しのようにWasmコンポーネントを利用できます [38]。

# 6. WasmGCによるメモリ管理の革新

WasmGCは、WebAssembly 3.0における最大の技術的転換点の一つです。従来のWasm（MVP）では、C++やRustのような手動メモリ管理言語が前提であり、JavaやKotlinのようなガベージコレクション（GC）を必要とする言語は、Wasmバイナリ内に独自のGC実装とランタイムを含める必要がありました。これはバイナリサイズを肥大化させ、効率を低下させる要因でした。

## 6.1 WasmGCの仕組みと型階層

WasmGCでは、Wasmランタイム（ブラウザやサーバーサイドランタイム）が持つ既存のGC機能を利用できるようにします。そのために、新しい型階層（Heap Types）が導入されました [12]。

- any: すべての参照型のトップ型。
- eq: 同値比較可能なオブジェクトの型。
- struct: 構造体型。メモリレイアウトが静的に定義されます。
- array: 配列型。
- i31: 31ビット整数をポインタとして扱うための型（参照カウントのオーバーヘッドなしに小整数を扱う最適化）。

## 6.2 構造体のバイナリエンコーディング

WasmGCの型定義は、Typeセクション（ID 1）で行われます。従来の関数型（0x60）に加え、以下の型コンストラクタが追加されています [39]。

- **Struct Type (0x5F):** 構造体を定義します。これに続いてフィールド定義のベクタが配置されます。
- **Array Type (0x5E):** 配列を定義します。

## 6.3 GC命令セット

GCオブジェクトを操作するための命令は、0xFBプレフィックスを使用します [24]。

- struct.new (0xFB 0x00): 構造体の新しいインスタンスをヒープ上に割り当てます。
- struct.get (0xFB 0x02): 構造体のフィールド値を読み取ります。
- array.new (0xFB 0x10): 配列を割り当てます。
- ref.cast: 実行時の型キャストを行います。

この仕組みにより、Kotlin/WasmやDart（Flutter Web）は、数MBにも及ぶランタイムを同梱することなく、軽量なバイナリとしてブラウザ上で高速に動作することが可能になりました。Google Sheetsの計算エンジンがWasmGCへ移行し、JavaScriptと同等以上のパフォーマンスを実現した事例は、この技術の実用性を証明しています [7]。

# 7. エコシステムとツールチェーン (WASI, Tooling)

WebAssemblyの技術仕様を支えるエコシステムも、2025年には大きく成熟しています。

## 7.1 WASI (WebAssembly System Interface) の進化

WASIは、WasmがOSの機能（ファイル、ネットワーク、時刻など）にアクセスするための標準インターフェースです。

- **Preview 2 (0.2.x):** 2024年に安定化し、2025年現在広く利用されています。Component Modelを全面的に採用し、モジュール化（wasi:cli, wasi:httpなど）が進みました [6]。
- **Preview 3 (0.3.x):** 2025年後半から2026年にかけての策定が進んでおり、最大の特徴は「ネイティブ非同期（Async）サポート」です。これにより、ポーリングなしで効率的な非同期I/Oが可能になります [42]。

## 7.2 必須ツールチェーン

- **wasm-tools:** Wasmバイナリの検証、閲覧、変換を行うためのデファクトスタンダードツールです。wasm-tools printでバイナリをWATに変換したり、component newでCore Moduleからコンポーネントを作成したりできます [43]。
- **wit-bindgen:** WIT定義から各言語（Rust, C, Go, Java, Pythonなど）のバインディングコードを生成し、Component Modelの利用を容易にします [38]。
- ランタイム: **Wasmtime**（Bytecode Alliance主導）がComponent ModelとWasmGCの参照実装として最先端を走っています。**Wasmer**やブラウザエンジン（V8, SpiderMonkey, JavaScriptCore）もこれら新仕様への対応を完了しています [3]。

# 8. 結論

2025年12月時点におけるWebAssemblyは、単なるブラウザ向けの技術から、セキュアでポータブルな計算基盤へと完全に変貌を遂げました。WebAssembly 3.0における**WasmGC**の導入は、JavaやKotlinといったマネージド言語の効率的な実行を可能にし、**Component Model**の標準化は、言語の壁を超えた真の再利用可能なソフトウェアコンポーネントの構築を実現しました。
テキスト表現（WAT）とバイナリ表現（WASM）は、これらの複雑な機能を収容するために拡張されましたが、その根底にある「コンパクトで、検証可能で、高速である」という設計思想は揺らいでいません。今後、WASI Preview 3による非同期処理の統合や、スレッドサポートの更なる成熟により、クラウドネイティブおよびエッジコンピューティング領域でのWebAssemblyの支配的な地位はさらに強固なものとなるでしょう。

引用文献


1. Introduction — WebAssembly 3.0 (2025-12-08), 12月 21, 2025にアクセス、https://webassembly.github.io/spec/core/intro/introduction.html

2. Wasm's Identity Crisis: What the 3.0 Release Tells Us About WebAssembly's Uncertain Future - RedMonk, 12月 21, 2025にアクセス、https://redmonk.com/kholterhoff/2025/10/17/wasms-identity-crisis/
3. Wasm 3.0 Completed - WebAssembly, 12月 21, 2025にアクセス、https://webassembly.org/news/2025-09-17-wasm-3.0/
4. WebAssembly Core Specification - W3C, 12月 21, 2025にアクセス、https://www.w3.org/TR/wasm-core-2/
5. Wasm 2.0 Completed - WebAssembly, 12月 21, 2025にアクセス、https://webassembly.org/news/2025-03-20-wasm-2.0/
6. The State of WebAssembly – 2024 and 2025 - Uno Platform, 12月 21, 2025にアクセス、https://platform.uno/blog/state-of-webassembly-2024-2025/
7. WebAssembly Production 2025: WasmGC and Enterprise Adoption - byteiota, 12月 21, 2025にアクセス、https://byteiota.com/webassembly-production-2025-wasmgc-and-enterprise-adoption/
8. Understanding WebAssembly text format - MDN Web Docs, 12月 21, 2025にアクセス、https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Understanding_the_text_format
9. Conventions — WebAssembly 3.0 (2025-12-08), 12月 21, 2025にアクセス、https://webassembly.github.io/spec/core/text/conventions.html
10. Understanding the WebAssembly Text Format WAT - KodeKloud Notes, 12月 21, 2025にアクセス、https://notes.kodekloud.com/docs/Exploring-WebAssembly-WASM/Getting-Started-with-WebAssembly/Understanding-the-WebAssembly-Text-Format-WAT
11. The WebAssembly Text Format - Nish Tahir, 12月 21, 2025にアクセス、https://nishtahir.com/the-wasm-text-format/
12. Types — WebAssembly 2.0 + tail calls + function references + gc (Draft 2024-12-09), 12月 21, 2025にアクセス、https://webassembly.github.io/gc/core/syntax/types.html
13. How to log elements in `ref struct` and `ref array` at js side? · WebAssembly gc · Discussion #493 - GitHub, 12月 21, 2025にアクセス、https://github.com/WebAssembly/gc/discussions/493
14. ArrayRef in wasmtime - Rust, 12月 21, 2025にアクセス、https://docs.wasmtime.dev/api/wasmtime/struct.ArrayRef.html
15. StructRef in wasmtime - Rust, 12月 21, 2025にアクセス、https://docs.wasmtime.dev/api/wasmtime/struct.StructRef.html
16. Wasm Introduction (Part 1): Binary Format | by CoinEx Smart Chain | Medium, 12月 21, 2025にアクセス、https://coinexsmartchain.medium.com/wasm-introduction-part-1-binary-format-57895d851580
17. Learning WebAssembly #2: Wasm Binary Format - Tomas Tulka's Blog, 12月 21, 2025にアクセス、https://blog.ttulka.com/learning-webassembly-2-wasm-binary-format/
18. Understanding the WebAssembly Binary Format - KodeKloud Notes, 12月 21,

2025にアクセス、
https://notes.kodekloud.com/docs/Exploring-WebAssembly-WASM/Getting-Started-with-WebAssembly/Understanding-the-WebAssembly-Binary-Format
19. LEB128 - Wikipedia, 12月 21, 2025にアクセス、https://en.wikipedia.org/wiki/LEB128
20. LEB128 - Grokipedia, 12月 21, 2025にアクセス、
https://grokipedia.com/page/LEB128
21. Values — WebAssembly 3.0 (2025-12-08), 12月 21, 2025にアクセス、
https://webassembly.github.io/spec/core/binary/values.html
22. Modules — WebAssembly 3.0 (2025-12-08), 12月 21, 2025にアクセス、
https://webassembly.github.io/spec/core/binary/modules.html
23. WebAssembly - MDN Web Docs - Mozilla, 12月 21, 2025にアクセス、
https://developer.mozilla.org/en-US/docs/WebAssembly
24. WebAssembly Opcode Table - GitHub Pages, 12月 21, 2025にアクセス、
https://pengowray.github.io/wasm-ops/
25. Change History — WebAssembly 3.0 (2025-12-03), 12月 21, 2025にアクセス、
https://webassembly.github.io/spec/core/appendix/changes.html
26. Instructions — WebAssembly 3.0 (2025-09-26), 12月 21, 2025にアクセス、
https://webassembly.github.io/spec/core/binary/instructions.html
27. What's New in WebAssembly 3.0 and Why It Matters for Developers - Medium, 12月 21, 2025にアクセス、
https://medium.com/wasm-radar/whats-new-in-webassembly-3-0-and-why-it-matters-for-developers-db14b6fb0d6c
28. The State of WebAssembly (Wasm) in 2025, 12月 21, 2025にアクセス、
https://hansrajrana.space/blog/the-state-of-webassembly-wasm-in-2025
29. Why the Component Model?, 12月 21, 2025にアクセス、
https://component-model.bytecodealliance.org/design/why-component-model.html
30. Revolutionizing Distributed Software with WebAssembly Component Model | by ritesh rai | Wasi Articles | Medium, 12月 21, 2025にアクセス、
https://medium.com/wasi-articles/revolutionizing-distributed-software-with-webassembly-component-model-412574d2881a
31. What is the WebAssembly Component Model? - F5, 12月 21, 2025にアクセス、
https://www.f5.com/company/blog/what-is-the-webassembly-component-model
32. component-model/design/mvp/Binary.md at main - GitHub, 12月 21, 2025にアクセス、
https://github.com/WebAssembly/component-model/blob/main/design/mvp/Binary.md
33. Modules — WebAssembly 1.1 (Draft 2022-04-05) - W3C, 12月 21, 2025にアクセス、
https://www.w3.org/TR/2022/WD-wasm-core-2-20220419/binary/modules.html
34. component-model/design/mvp/Explainer.md at main - GitHub, 12月 21, 2025にアクセス、
https://github.com/WebAssembly/component-model/blob/main/design/mvp/Explainer.md
35. wasm_encoder - Rust - Docs.rs, 12月 21, 2025にアクセス、
https://docs.rs/wasm-encoder

36. WIT Reference - The WebAssembly Component Model, 12月 21, 2025にアクセス、https://component-model.bytecodealliance.org/design/wit.html
37. WIT.md - WebAssembly/component-model - GitHub, 12月 21, 2025にアクセス、https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md
38. Components | wasmCloud, 12月 21, 2025にアクセス、https://wasmcloud.com/docs/developer/languages/go/components/
39. Binary Format · Issue #37 · WebAssembly/gc - GitHub, 12月 21, 2025にアクセス、https://github.com/WebAssembly/gc/issues/37
40. 249149 – [Wasm-GC] Adjust struct.new opcode - WebKit Bugzilla, 12月 21, 2025にアクセス、https://bugs.webkit.org/show_bug.cgi?id=249149
41. The WebAssembly Component Model, Part 1 - NGINX Community Blog, 12月 21, 2025にアクセス、https://blog.nginx.org/blog/wasm-component-model-part-1
42. Roadmap - WASI.dev, 12月 21, 2025にアクセス、https://wasi.dev/roadmap
43. WebAssembly Text Format (WAT), 12月 21, 2025にアクセス、https://component-model.bytecodealliance.org/language-support/wat.html
44. wit-component - WebAssembly - Lib.rs, 12月 21, 2025にアクセス、https://lib.rs/crates/wit-component
45. WASI and the WebAssembly Component Model: Current Status - eunomia, 12月 21, 2025にアクセス、https://eunomia.dev/blog/2025/02/16/wasi-and-the-webassembly-component-model-current-status/