一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

ハードウェア記述言語 Veryl の設計と実装

初田　直也†

† 株式会社 PEZY Computing　〒101–0052 東京都千代田区神田小川町 1–11 千代田小川町クロスタ 5 階
E-mail: †hatta@pezy.co.jp

あらまし　ディジタル回路の設計には Verilog や SystemVerilog などのハードウェア記述言語（HDL）が広く使用されているが，言語の古さに起因する設計生産性の低さが問題となっている．代替として提案されてきた HDL の多くはプログラミング言語上のドメイン固有言語として構築されており，HDL に適した構文を導入できないうえ，生成される Verilog の可読性が低いため，既存の設計フローとの共存が難しい．そこで我々は，HDL に特化した構文を持ち，可読性の高い SystemVerilog へ変換可能な新しい HDL である Veryl を開発している．Veryl の処理系は中間表現に基づく意味解析器，シミュレータ，論理合成を統合しており，記述から検証・見積りまでを処理系単体で完結できる．本論文では Veryl の言語設計と処理系の実装について述べる．

キーワード　ハードウェア記述言語，SystemVerilog，コンパイラ，オープンソースソフトウェア

Design and Implementation of Veryl: A Hardware Description Language

Naoya HATTA†

† PEZY Computing, K.K.　Chiyoda Ogawamachi Crosta 5F, 1–11 Kanda-Ogawamachi, Chiyoda-ku, Tokyo, 101–0052 Japan
E-mail: †hatta@pezy.co.jp

Abstract　Hardware description languages (HDLs) such as Verilog and SystemVerilog are widely used for digital circuit design, but their outdated language design limits productivity. Most existing alternative HDLs are built as domain-specific languages embedded in programming languages; they cannot introduce HDL-specific syntax, and the generated Verilog has low readability, making them difficult to coexist with existing design flows. We are developing Veryl, a new HDL that features syntax specialized for hardware description and transpiles into human-readable SystemVerilog. The Veryl toolchain integrates an IR-based semantic analyzer, a simulator, and logic synthesis, enabling a complete design cycle within a single toolchain. This paper describes the design and implementation of Veryl.

Key words　hardware description language, SystemVerilog, compiler, open source software

1. はじめに

ディジタル回路の設計には，主に Verilog，VHDL，SystemVerilog といったハードウェア記述言語（HDL）が広く使用されている．これらの言語の起源は古く，例えば Verilog と VHDL はいずれも 1980 年代に開発されたものである．そのため，近年のプログラミング言語で一般的となった高度な抽象化やコード再利用のための言語機能を欠いている．SystemVerilog は，Verilog にこうした機能を加えて拡張した言語であるが，言語仕様が大幅に複雑化した結果，EDA ツールによる仕様のサポートは不完全であり，有用な言語機能であっても使用できるとは限らない．また，合成可能な記述とシミュレーション専用の記述が言語上区別されないため，回路設計者は合成に使用できる記述を慎重に選別する必要がある．

さらに，近年のプログラミング言語では自動フォーマッタやリンタが言語標準で提供され，エディタと連携してリアルタイムにフィードバックを返すなど，開発支援ツールによって生産性を高める仕組みが一般的となっているが，伝統的な HDL でそのような開発環境を構築することは困難である．このように，HDL では言語とツールの両面に起因する設計生産性の低さが問題となっている．

この問題に対し，プログラミング言語のエコシステムを活用する新しい HDL（以下，Alt-HDL）が提案されてきた．Chisel [1] や Amaranth [2] に代表されるこれらの言語は，特定のプログラミング言語のドメイン固有言語（DSL）として HDL を構築するアプローチをとる．記述されるソースコードはベースとなるプログラミング言語のものであるため，その言語の持つ抽象化機能に加え，開発支援ツールやライブラリといった既存

のエコシステムの恩恵を受けられるという利点がある．

しかしこれらの Alt-HDL には，実際のハードウェア設計に用いるうえで以下の課題がある．第一に，ベース言語の構文の制約を受けるため，任意のビット幅の表現やクロック・リセットといった HDL に頻出する要素に特化した構文を導入できない．第二に，多くの Alt-HDL は Verilog へ変換するトランスパイラとして実装されているが，生成される Verilog は DSL の抽象化機能を展開した結果であり，可読性が低くデバッグが困難である．第三に，DSL の記述と生成される Verilog との対応が単純でないため，Verilog の特定の箇所のみを変更したい場合に，DSL のどこをどのように修正すればよいかを予測することが難しい．このため，Verilog ソースコードの部分的な修正を必要とするタイミングクロージャや pre/post-mask ECO（Engineering Change Order）といった ASIC 設計フローへの適用が難しい．

そこで我々は，HDL に特化した構文を持ち，可読性の高い SystemVerilog へ変換可能な新しい HDL である Veryl [3] を開発している．Veryl は SystemVerilog のキーワードと意味論をベースに，Rust [4] や Go [5] など近年のプログラミング言語で採用されている構文上の改善を取り入れた言語である．Veryl の記述と生成される SystemVerilog との対応は明確であり，既存の SystemVerilog コードベースとの高い相互運用性を持つ．

さらに Veryl の処理系は，単なるトランスパイラではなく，中間表現（IR）に基づく意味解析器，シミュレータ，論理合成による面積・遅延・電力の見積り機能を統合している．これにより，外部の EDA ツールに依存せずに，記述・検証・見積りからなる設計サイクルを処理系単体で完結できる．

Veryl はオープンソースソフトウェアとして開発されており，PEZY Computing の次世代プロセッサ（Veryl 記述約 35 万行）や RVA23 プロファイル準拠のアウトオブオーダ RISC-V コア Heliodor [6] など，実規模の設計での利用実績を持つ．

なお，Veryl の初期の言語設計は文献 [7] で発表している．本論文はこれに，その後追加された言語機能と，IR ベース意味解析器・シミュレータ・論理合成を含む処理系の実装および評価を加えたものである．以下，2. で Veryl の言語設計を，3. で処理系の実装を述べ，4. で評価と採用事例を示す．

2. Veryl の設計

2.1 設 計 方 針

Veryl は以下の 3 つを設計方針とする．

（1） 合成可能な HDL に特化した構文：合成可能な回路の記述に最適化した構文を採用する．シミュレーション専用の記述はテストベンチ内に限定され，合成対象のコードには現れないことが保証されるため，回路設計者は合成に使用できる記述を選別する必要がなく，記述の信頼性と可読性が高まる．

（2） 可読性の高い SystemVerilog の生成：変数や `always_ff` 宣言といった基本的な記述が，生成される SystemVerilog と明確に対応するよう言語を設計する．これにより，抽象化機能を用いた場合も生成結果の構造は予測可能となり，特定の SystemVerilog 箇所と Veryl 記述の対応を追跡できる．また Veryl のモジュール・インターフェース・パッケージ・構造体などのユーザ定義型は SystemVerilog のそれと等価であり相互に参照できるため，既存の SystemVerilog コードベースおよび設計フローと相互運用できる．

（3） 言語に統合されたツール群：フォーマッタやリンタ，シミュレータなどの開発支援ツールを処理系の一部として提供し，追加の環境構築なしに高い生産性を得られるようにする．

本章では方針 1，2 に基づく言語設計を説明し，方針 3 を実現する処理系の実装は 3. で述べる．

2.2 基 本 構 文

Veryl の構文は SystemVerilog のキーワードをベースに，近年のプログラミング言語で見られる構文上の改善を取り入れたものである．図 1 に同一のカウンタモジュールを SystemVerilog と Veryl で記述した例を示す．なお，図 1(a) は図 1(b) の Veryl 記述からコンパイラが生成する SystemVerilog とほぼ一致しており，2.1 で述べた，生成 SystemVerilog と明確に対応する設計方針を反映している．以下では，この例に現れる基本的な構文の違いを説明する．

まず，`///`で始まるコメントはドキュメンテーションコメントである．これは通常の`//`で始まるコメントと構文上区別され，処理系によるドキュメント自動生成に用いられる．

パラメータ宣言やポート宣言では，最終要素の末尾にカンマを置くこと（末尾カンマ）を許容する．これにより要素の追加・削除に伴うカンマの調整が不要になるだけでなく，バージョン管理システムにおける不要な差分の発生を抑制できる．

変数宣言やポート宣言では，`var r_cnt: logic<WIDTH>;` のように型を変数名の後に記述する．これは近年のプログラミング言語で広く採用されている形式であり，構文解析が単純になるほか，型推論を用いた省略記法へも自然に拡張できる．また SystemVerilog では `[]` を変数名の前に置くか後に置くかで packed 配列と unpacked 配列を区別するが，Veryl では packed 配列を`<>`，unpacked 配列を `[]` として区別し，幅を直接指定できるようにすることで `[WIDTH-1:0]` のような冗長な記法を不要にした．

SystemVerilog ではブロッキング代入とノンブロッキング代入が演算子として区別されるが，Veryl では代入演算子を`=`に統一し，ブロッキングとノンブロッキングのいずれになるかは，記述される文脈（`always_ff` または `always_comb`）によって決まる．これにより，`always_comb` 内に誤ってノンブロッキング代入を書くといった代入種別の取り違えを防げるうえ，ノンブロッキング代入においても `+=`のような複合代入演算子を自然に使用できる．

2.3 クロックとリセット

SystemVerilog においてクロックとリセットは単なる変数として扱われるが，Veryl では `clock` と `reset` という専用の型を用意し，通常の変数と区別している．これによりモジュール内のクロックとリセットが 1 つずつである場合，図 1(b) のように `always_ff` におけるクロックとリセットの指定を省略できる．複数のクロックを用いる設計であっても，単一のクロックしか持たないモジュールが大半を占めるため，多くの記述

```
// Counter
module Counter #(
    parameter WIDTH = 1
)(
    input  logic             i_clk  ,
    input  logic             i_rst_n,
    output logic [WIDTH-1:0] o_cnt
);
    logic [WIDTH-1:0] r_cnt;

    always_ff @ (posedge i_clk or
                 negedge i_rst_n) begin
        if (!i_rst_n) begin
            r_cnt <= 0;
        end else begin
            r_cnt <= r_cnt + 1;
        end
    end

    always_comb begin
        o_cnt = r_cnt;
    end
endmodule
```

(a) SystemVerilog

```
/// Counter
module Counter #(
    param WIDTH: u32 = 1,
)(
    i_clk: input  clock       ,
    i_rst: input  reset       ,
    o_cnt: output logic<WIDTH>,
){
    var r_cnt: logic<WIDTH>;

    always_ff {
        if_reset {
            r_cnt = 0;
        } else {
            r_cnt += 1;
        }
    }

    always_comb {
        o_cnt = r_cnt;
    }
}
```

(b) Veryl

図 1　SystemVerilog と Veryl によるカウンタモジュールの記述例

Fig. 1　A counter module written in SystemVerilog and Veryl.

```
module ModuleA (
    // signals in clock domain 'a
    i_clk_a: input  'a clock,
    i_dat_a: input  'a logic,
    o_dat_a: output 'a logic,
    // signals in clock domain 'b
    i_clk_b: input  'b clock,
    o_dat_b: output 'b logic,
) {
    // safe: within the same domain
    assign o_dat_a = i_dat_a;

    // crossing domains requires
    // an unsafe (cdc) block
    unsafe (cdc) {
        assign o_dat_b = i_dat_a;
    }
}
```

図 2　クロックドメイン注釈の記述例

Fig. 2　An example of clock domain annotation.

```
// prototype of SRAM modules
proto module SramProto #(
    param WIDTH: u32 = 8,
)(
    // port definitions (omitted)
);

// queue independent of SRAM types
module SramQueue::<T: SramProto> {
    inst u_sram: T;
    // queue logic (omitted)
}

module Top {
    // queue with SRAM by vendor A
    inst u0: SramQueue::<SramVendorA>;
    // queue with SRAM by vendor B
    inst u1: SramQueue::<SramVendorB>;
}
```

図 3　ジェネリクスとプロトタイプの記述例

Fig. 3　An example of generics and prototype.

を簡略化できる．

また，リセットの極性と同期・非同期は，SystemVerilog への変換時にプロジェクト設定で指定できる．SystemVerilog ではリセットの極性に応じて `if (!i_rst_n)` や `if (i_rst)` といった条件を書き分ける必要があるが，Veryl ではリセット専用の `if_reset` 構文を提供し，設定に応じたリセット条件をコンパイラが生成する．これにより，同一の Veryl 記述から，例えば非同期負論理リセットを用いる ASIC 向けと同期正論理リセットを用いる FPGA 向けの SystemVerilog を生成し分けることができる．

さらに Veryl は，クロックドメイン注釈によるクロックドメインクロッシング（CDC）の検査を言語機能として提供する．図 2 に示すように，複数のクロックを持つモジュールでは各ポートに `'a` のような注釈を付け，所属するクロックドメインを明示することが必須である．コンパイラは信号のドメインを追跡し，ドメインを跨ぐ代入をエラーとして検出する．同期化回路を経由するなどの意図的な CDC は，`unsafe (cdc)` ブロックで囲むことで明示する．従来の SystemVerilog フローにおいて CDC 検査は高価な専用ツールを必要としていたが，Veryl ではコンパイル時に標準で検査される．

2.4　ジェネリクスとプロトタイプ

SystemVerilog ではパラメータオーバーライドによって，モジュールやインターフェースのパラメータをインスタンス時に変更できる．しかし，インスタンスするモジュールの種類などパラメータとして指定できないものはオーバーライドできず，関数に対するパラメータオーバーライドもできない．そこで Veryl ではジェネリクスを導入し，モジュール・インターフェース・パッケージ・関数・構造体などを型パラメータによって汎用化できるようにした．

図 3 に SRAM を記憶域として使用するキューの例を示す．SRAM は IP ベンダーやプロセスごとに異なるモジュール名を持つため，SystemVerilog では，使用する SRAM ごとにキューの記述を複製するか，`generate` 文やマクロで書き分ける必要があり，いずれの方法も煩雑である．Veryl ではモジュール名を型パラメータ `T` として外部から与えることで，キューのロジックと SRAM モジュールを分離し，再利用性の高い記述ができる．

このとき，型パラメータには境界（bound）としてプロトタイプ（`proto`）を指定する必要がある．プロトタイプはモジュールなどが持つべきパラメータとポートを規定する一種のシグ

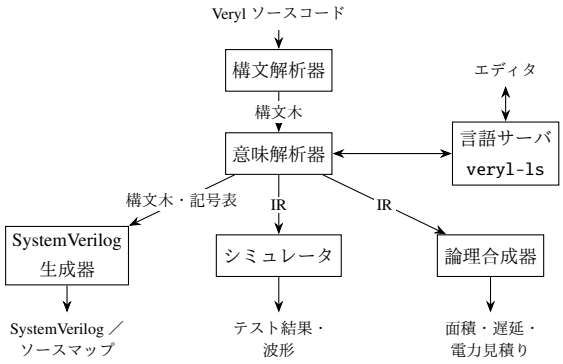

図 4　Veryl 処理系の構成
Fig. 4　Architecture of the Veryl toolchain.

ネチャであり，型パラメータに渡される実引数がプロトタイプを満たすことをコンパイラが検査する．これにより，ジェネリックな記述の誤用を，インスタンス化の展開を待たずに検出できる．

2.5　その他の言語機能

このほかにも Veryl は，列挙型のエンコーディング（one-hot，グレイコードなど）の指定，構造体コンストラクタ，名前付き引数による関数呼び出し，変数宣言における型推論，モジュールやインターフェースの別名定義といった，記述性を高める言語機能を備える．言語仕様の全体はオンラインの言語リファレンス [3] を参照されたい．

3.　Veryl の実装

3.1　コンパイラ構成

Veryl の処理系は Rust で実装された単一のコマンドラインツール `veryl` として提供され，プロジェクト生成（`new`），整形（`fmt`），検査（`check`），SystemVerilog 生成（`build`），テスト（`test`），論理合成（`synth`），ドキュメント生成（`doc`），ライブラリの発行と取得（`publish`，`update`）などのサブコマンドを持つ．これに加えて，Language Server Protocol [8] に基づく言語サーバ `veryl-ls` を提供し，Visual Studio Code や Vim などのエディタからコンパイラと同等の解析結果をリアルタイムに利用できる．

図 4 に処理系の構成を示す．コンパイラ内部は，構文解析器，意味解析器，各種バックエンド（SystemVerilog 生成器・シミュレータ・論理合成器）から構成される．構文解析器はパーサジェネレータ parol [9] によって文法定義から生成される．意味解析器は構文木から記号表を構築して名前解決と型検査を行い，さらに後述する中間表現（IR）を構築して詳細な検査を実施する．SystemVerilog 生成器は構文木のコメントや整形をできる限り保持したまま SystemVerilog を出力し，あわせて生成コードと Veryl 記述の対応を示すソースマップを生成する．シミュレータと論理合成器は，意味解析器が構築した IR を入力として動作する．

3.2　IR ベース意味解析

Veryl コンパイラは，識別子の重複や未定義，信号方向と代入・参照の不一致，ビット幅の不一致といった一般的な検査に加え，未代入変数や意図しないラッチ生成など，従来の SystemVerilog フローでは高価なリントツールや時間のかかる論理合成で検出されていたエラーをコンパイル時に検出する．これらの検査は言語サーバ `veryl-ls` にも組み込まれており，回路設計者はエディタでの編集中にもエラーをリアルタイムに把握できる．

当初これらの検査は構文木上で実装されていたが，パラメータオーバーライドやジェネリクスを考慮した検査が難しく，また検査の粒度も限られていた．そこで現在の処理系では，パラメータを解決した後の回路構造を表す IR を構築し，その上で意味解析を行う．IR 上ではすべての信号がビット単位で追跡されるため，例えば条件分岐の一方の経路でのみ信号の一部のビットが代入されない，といったビット単位のラッチ生成を検出できる．また IR の導入によって解析の実装が単純化され，解析性能も大幅に向上した．

3.3　ネイティブシミュレータ

Veryl の処理系は，意味解析器の構築する IR から直接実行コードを生成するネイティブシミュレータを内蔵する．従来の Alt-HDL における検証は，生成した Verilog を Verilator [10] などの外部シミュレータに入力する形で行われてきたが，ネイティブシミュレータによって外部ツールのセットアップなしにテストを実行できる．テストベンチは Veryl ソースコード中に `#[test]` 属性を付けて記述し，`veryl test` コマンドによって実行される．2 値シミュレーションに加えて，不定値を含む 4 値シミュレーションにも対応する．

シミュレータの実行コードの生成方式には，コンパイル速度と実行速度のトレードオフが存在する．Verilator をはじめとする既存の高速シミュレータは C++ コードを経由した事前コンパイル方式をとるため，実行開始までに長いコンパイル時間を要する．そこで Veryl のシミュレータは 2 つのバックエンドを組み合わせる．1 つはコンパイラ基盤 Cranelift [11] を用いたバックエンドであり，最適化の程度は低いもののコンパイルが高速であるため，シミュレーションをほぼ即座に開始できる．もう 1 つは C コードを経由して GCC で最適化するバックエンドであり，バックグラウンドで最適化バイナリの生成を進め，完成した時点で実行中のシミュレーションを最適化バイナリへ動的に切り替える．これにより，起動の速さと実行の速さを両立している．性能の評価は 4. 1 で述べる．

3.4　論 理 合 成

`veryl synth` コマンドは，IR から軽量なゲートレベル合成を行い，指定したトップモジュールの面積・クリティカルパス遅延・電力の見積りを報告する．セルの面積・遅延・電力の値は，SKY130 [12]，ASAP7 [13]，GF180MCU [14]，IHP SG13G2 [15] といったオープン PDK が公開する Liberty データに基づいてキャリブレーションされており，抽象的なゲート数ではなく実際のシリコンの比率に基づく見積りが得られる．これはサインオフ用ではなく設計初期の探索を目的とした機能であり，記述の変更が面積やタイミングに与える影響を，論理合成ツールを起動することなく即座に確認できる．ただし，この見積

表 1　評価に用いたツールおよび対象設計のバージョン・リビジョン
Table 1　Versions and revisions of the tools and target design used in the evaluation.

名称	バージョン／リビジョン
Veryl	revision `49435563`
Verilator	`v5.050`
Heliodor	revision `a30afc7`

表 2　Heliodor による Linux ブートのシミュレーション時間（秒）．「比」は Verilator を基準とした Veryl の実行時間比（Veryl/Verilator）を表す．
Table 2　Simulation time of Linux boot on Heliodor in seconds. The ratio is Veryl time relative to Verilator time (Veryl/Verilator).

		Xeon Gold 6434			1950X		
構成	実行	Verilator	Veryl	比	Verilator	Veryl	比
1 コア	初回	162.7	88.0	0.54	431.6	159.3	0.37
	2 回目	78.1	63.1	0.81	300.2	116.3	0.39
2 コア	初回	499.4	287.0	0.57	1183.4	682.5	0.58
	2 回目	287.5	248.2	0.86	914.4	671.7	0.73
4 コア	初回	2941.7	1726.3	0.59	3933.0	2202.0	0.56
	2 回目	2227.9	1568.9	0.70	2728.7	2200.7	0.81

りが設計判断に有用であるためには，見積り値が実際の論理合成結果と十分に相関している必要がある．この点の詳細な評価は今後の課題であるが，予備的な簡易評価では，オープンソースの論理合成ツール Yosys [16] による合成結果との誤差は 10〜20% 程度であった．

3.5　その他のツール

処理系はこのほかに，開発を支援するツール群を統合している．

既存資産の移行支援として，SystemVerilog ソースコードを等価な Veryl へ変換するトランスレータ（`veryl translate`）を提供する．Veryl は SystemVerilog との相互参照が可能であるため，既存プロジェクトの一部のファイルのみを Veryl へ段階的に書き換えていくことができる．

ライブラリ利用の支援として，パッケージ管理機能を提供する．プロジェクト定義ファイルに git リポジトリの URL とバージョンを記述するだけで依存ライブラリを取得でき，ライブラリ内で定義されたモジュール・インターフェース・パッケージを使用できる．

またドキュメンテーションコメントと，コンパイラが解析したモジュールやポートなどの構造情報から，統一された形式のドキュメントを自動生成する（`veryl doc`）．コメント中には波形（WaveDrom [17]）や図（Mermaid [18]）の記述を埋め込むことができる．

さらに，ツールチェーン自体の配布・更新を管理するインストーラ `verylup` を提供しており，処理系のバージョンをプロジェクトごとに固定・切替できる．

4.　評価と事例

4.1　シミュレータ性能

ネイティブシミュレータの性能を，オープンソースの SystemVerilog シミュレータとして広く使用されている Verilator [10] と比較した．ワークロードには，Veryl で記述されたアウトオブオーダ RISC-V コア Heliodor [6] による Linux ブートを用い，1・2・4 コアの構成（それぞれ約 1,000 万・1,400 万・2,000 万サイクル）で測定した．Veryl と Verilator のいずれもコンパイル済み成果物をキャッシュして再利用するため，キャッシュが存在しない初回実行と，存在する 2 回目以降の実行を分けて測定した．比較対象は Verilator v5.050 である．Verilator は 4 値シミュレーションにも対応しているが，これは実験的な機能であるため，本評価では両者を 2 値モードで測定した．測定は世代の異なる 2 つの CPU（Intel Xeon Gold 6434 および AMD Ryzen Threadripper 1950X）上で行った．評価に用いたツールおよび対象設計のバージョンとリビジョンを表 1 に示す．

表 2 に結果を示す．初回実行では，Verilator が実行時間の多くを C++ コードのコンパイルに費やすのに対し，Veryl は Cranelift による高速なコンパイルで即座に実行を開始できるため，構成とマシンに応じて 41〜63% 短い時間で完了した．コンパイル済み成果物を再利用する 2 回目以降の実行においても，Veryl はすべての構成で Verilator より高速で，実行時間は 14〜61% 短かった．これらの優位性は世代の異なる 2 つの CPU のいずれでも一貫して見られ，Veryl のシミュレーションエンジンが特定のマイクロアーキテクチャに依存せず十分に最適化されていることを示唆している．

4.2　採 用 事 例

Veryl は 2022 年 12 月からオープンソースソフトウェアとして開発されており，2026 年 7 月時点でコミット数は約 5,000，コントリビュータは約 30 名，GitHub のスター数は 950 を超えている．以下に実規模の設計における採用事例を示す．

産業利用としては，PEZY Computing が開発中の次世代 HPC/AI 向けプロセッサの設計に採用されており，約 600 万行の SystemVerilog コードベースと組み合わせて約 35 万行の Veryl 記述が使用されている．これは 2.1 で述べた，既存の SystemVerilog 資産との相互運用性を重視した設計方針の有効性を示すものである．

オープンソースの設計としては，前述の Heliodor [6] が挙げられる．Heliodor は Veryl で記述された 2-way スーパースカラのアウトオブオーダ RISC-V コアであり，RVA23 プロファイルに準拠し，最大 8 コアの SMP 構成で Linux をブートできる．RISC-V Architectural Compatibility Tests (ACT) v4 をパスしており，Veryl が実用規模のプロセッサ設計に耐えることを示している．また，Linux ブート可能な RISC-V コアである bluecore [19] は，その開発過程を解説した技術同人誌『Veryl で作る CPU』[20] が頒布されており，教材としても利用されている．

教育利用としては，スウェーデンの Luleå 工科大学におけるディジタル設計の講義で Veryl が採用されている．2025 年度は希望者向けの発展課題で実装言語として採用されたが，2026

年度は全受講者が最初から Veryl を用いて課題に取り組んでいる．

また Veryl を取り巻くエコシステムも成長しており，CSR ジェネレータの RgGen [21] や，Rust でテストベンチを記述するフレームワークの Marlin [22] など，Veryl に対応した外部ツールも登場している．

5. 関 連 研 究

プログラミング言語の DSL として HDL を構築する Alt-HDL としては，Scala を基盤とする Chisel [1] や SpinalHDL [23]，Python を基盤とする MyHDL [24] や Amaranth [2] などが挙げられる．これらはベース言語の強力な抽象化機能を利用できる一方，1. で述べたとおり，HDL に特化した構文を導入できず，生成される Verilog の可読性にも課題がある．

独自の構文を持つ HDL としては，Bluespec SystemVerilog [25] や Spade [26] が挙げられる．Bluespec はルールベースの高い抽象度を持つ言語であり，Spade は Rust に影響を受けた構文を持ち，パイプラインを第一級要素として扱う言語である．これらは RTL より高い抽象度の意味論を導入するため，生成される Verilog とソースコードの対応を追うことは難しい．これに対し Veryl は，抽象度を SystemVerilog と同等に保ち，生成コードとの対応を追跡可能に保つことで，既存の設計フローとの相互運用性を優先している点が異なる．

また，これらの既存言語の処理系はいずれもコード生成までを担当し，検証や合成は外部ツールに委ねている．シミュレータと論理合成までを処理系に統合し，単体で設計サイクルを完結できる点は Veryl に特徴的である．

6. ま と め

本論文では，HDL に特化した構文を持ち，可読性の高い SystemVerilog へ変換可能な新しいハードウェア記述言語 Veryl の設計と実装について述べた．Veryl は，合成可能な回路の記述に最適化された構文により信頼性と可読性を高めるとともに，生成される SystemVerilog との明確な対応により既存の設計フローとの相互運用性を確保している．処理系は IR に基づく意味解析器，シミュレータ，論理合成を統合し，外部ツールに依存せず記述・検証・見積りの設計サイクルを完結できる．ネイティブシミュレータは Verilator との比較において，初回実行で 41〜63% 短い実行時間を示した．Veryl はオープンソースソフトウェアとして開発されており，産業・オープンソース・教育の各分野で採用が進んでいる．今後は，シミュレータのさらなる高速化や検証機能の拡充を進めるとともに，より幅広い採用を目指して開発を継続する．

謝　　辞

Veryl の開発に貢献しているすべてのコントリビュータ，ならびにコミュニティの皆様に感謝する．

文　　献

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: Constructing hardware in a Scala embedded language,” Proceedings of the 49th Annual Design Automation Conference (DAC), pp.1216–1225, 2012.
[2] “Amaranth: A modern hardware definition language and toolchain based on Python,” https://github.com/amaranth-lang/amaranth.
[3] “Veryl: A modern hardware description language,” https://veryl-lang.org/.
[4] “The Rust programming language,” https://www.rust-lang.org/.
[5] “The Go programming language,” https://go.dev/.
[6] “Heliodor: An RVA23-compliant multicore out-of-order RISC-V core in Veryl,” https://github.com/dalance/heliodor.
[7] N. Hatta, T. Ishitani, and R. Shioya, “Veryl: A new hardware description language as an alternative to SystemVerilog,” DVCon Japan 2024. arXiv:2411.12983, 2024.
[8] “Language Server Protocol,” https://microsoft.github.io/language-server-protocol/.
[9] “parol: LL(k) and LALR(1) parser generator for Rust,” https://github.com/jsinger67/parol.
[10] W. Snyder, “Verilator: Speedy reference models, direct from RTL,” Presentation to University of Massachusetts Amherst, https://www.veripool.org/papers/Verilator_Modeling_UMass2017b_pres.pdf, 2017.
[11] “Cranelift,” https://cranelift.dev/.
[12] “SkyWater Open Source PDK,” https://github.com/google/skywater-pdk.
[13] L.T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, “ASAP7: A 7-nm finFET predictive process design kit,” Microelectronics Journal, vol.53, pp.105–115, 2016.
[14] “GlobalFoundries 180nm MCU Open Source PDK,” https://github.com/google/gf180mcu-pdk.
[15] “IHP Open Source PDK,” https://github.com/IHP-GmbH/IHP-Open-PDK.
[16] C. Wolf, “Yosys Open SYnthesis Suite,” https://yosyshq.net/yosys/.
[17] “WaveDrom: Digital timing diagram everywhere,” https://wavedrom.com/.
[18] “Mermaid: Diagramming and charting tool,” https://mermaid.js.org/.
[19] “bluecore: A RISC-V core written in Veryl,” https://github.com/nananapo/bluecore.
[20] nananapo, “Veryl で作る CPU,” ミーミミ研究室，https://cpu.kanataso.net/，2024.
[21] “RgGen: Code generation tool for control and status registers,” https://github.com/rggen/rggen.
[22] “Marlin: Hardware testing/simulation in Rust,” https://github.com/ethanuppal/marlin.
[23] “SpinalHDL: A high level hardware description language,” https://github.com/SpinalHDL/SpinalHDL.
[24] K. Jaic and M.C. Smith, “Enhancing hardware design flows with MyHDL,” Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp.28–31, 2015.
[25] R. Nikhil, “Bluespec System Verilog: Efficient, correct RTL from high level specifications,” Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), pp.69–70, 2004.
[26] F. Skarman and O. Gustafsson, “Spade: An HDL inspired by modern software languages,” Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications (FPL), pp.454–455, 2022.