

# Veryl

SystemVerilogに代わる新しいハードウェア記述言語

初田 直也

PEZY Computing, K.K.

# Veryl

オープンソースソフトウェアとして開発中の新しいHDL

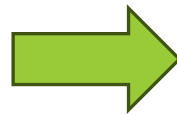
- SystemVerilogとRustをベースにした独自構文
- 可読性の高いSystemVerilogにコンパイル

```
/// 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;
        }
    }
}
```

Veryl

コンパイル



```
// 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, negedge i_rst_n) begin
        if (!i_rst_n) begin
            r_cnt <= 0;
        end else begin
            r_cnt <= r_cnt + 1;
        end
    end
end
endmodule
```

SystemVerilog

# 目次

---

## 開発の動機

- SystemVerilogの課題
- 既存のAlt-HDLの課題

## Veryl: SystemVerilogに代わる新しいHDL

- コンセプト
- 主な機能と利点

## Verylの開発と利用状況

- プロジェクト概況
- 活用事例

## まとめ

# 目次

---

## 開発の動機

- SystemVerilogの課題
- 既存のAlt-HDLの課題

## Veryl: SystemVerilogに代わる新しいHDL

- コンセプト
- 主な機能と利点

## Verylの開発と利用状況

- プロジェクト概況
- 活用事例

## まとめ

# SystemVerilogの課題

---

# SystemVerilogの課題

---

冗長で間違いやすい構文

- Verilog由来の古い構文
- 合成不能記述が容易に混入する

# SystemVerilogの課題

---

冗長で間違いやすい構文

- Verilog由来の古い構文
- 合成不能記述が容易に混入する

コンパイル時チェックが限定的

- チェックの有無や種類はツール依存
- 高度なチェック（例：CDC）には高価な専用ツールが必要

# SystemVerilogの課題

---

冗長で間違いやすい構文

- Verilog由来の古い構文
- 合成不能記述が容易に混入する

コンパイル時チェックが限定的

- チェックの有無や種類はツール依存
- 高度なチェック（例：CDC）には高価な専用ツールが必要

生産性を高めるツールの不足

- フォーマッタ、リアルタイム診断、依存関係管理



# SystemVerilogの課題

---

冗長で間違いやすい構文

- Verilog由来の古い構文
- 合成不能記述が容易に混入する

コンパイル時チェックが限定的

- チェックの有無や種類はツール依存
- 高度なチェック（例：CDC）には高価な専用ツールが必要

生産性を高めるツールの不足

- フォーマッタ、リアルタイム診断、依存関係管理



既存のAlt-HDL（Chiselなど）はどうか？

# 既存のAlt-HDLの課題

---

# 既存のAlt-HDLの課題

---

構文がHDLに適していない

- ベースとなる言語の構文を引き継ぐ（例：Chiselの場合はScala）
- HDL特有の構文は導入できない

# 既存のAlt-HDLの課題

---

構文がHDLに適していない

- ベースとなる言語の構文を引き継ぐ（例：Chiselの場合はScala）
- HDL特有の構文は導入できない

Verilogと言語構造が大きく異なる

- 小さなコードから大量のVerilogが生成
- 生成されたVerilogの可読性が低い

# 既存のAlt-HDLの課題

---

構文がHDLに適していない

- ベースとなる言語の構文を引き継ぐ（例：Chiselの場合はScala）
- HDL特有の構文は導入できない

Verilogと言語構造が大きく異なる

- 小さなコードから大量のVerilogが生成
- 生成されたVerilogの可読性が低い

Verilog（≠ SystemVerilog）が生成される

- 既存のSystemVerilogとの統合が困難

# 既存のAlt-HDLの課題

---

構文がHDLに適していない

- ベースとなる言語の構文を引き継ぐ（例：Chiselの場合はScala）
- HDL特有の構文は導入できない

Verilogと言語構造が大きく異なる

- 小さなコードから大量のVerilogが生成
- 生成されたVerilogの可読性が低い

Verilog（≠ SystemVerilog）が生成される

- 既存のSystemVerilogとの統合が困難



これらの課題を解決する新しいHDLが必要 → Veryl

# 目次

---

## 開発の動機

- SystemVerilogの課題
- 既存のAlt-HDLの課題

## Veryl: SystemVerilogに代わる新しいHDL

- コンセプト
- 主な機能と利点

## Verylの開発と利用状況

- プロジェクト概況
- 活用事例

## まとめ

# Verylのコンセプト

---

合成可能なHDLに最適化された構文

- 生産性と信頼性を高める言語デザイン

可読性の高いSystemVerilogを生成

- SystemVerilogとの高い相互運用性

言語に統合されたツール群

- エディタやCIとの連携



# Verilのコンセプト

---

合成可能なHDLに最適化された構文

- 生産性と信頼性を高める言語デザイン

可読性の高いSystemVerilogを生成

- SystemVerilogとの高い相互運用性

言語に統合されたツール群

- エディタやCIとの連携

# 基本的な構文

## SystemVerilogとVerilogの比較

現代的なプログラミング言語  
でよく見られる機能の採用

**SystemVerilog**

```
// 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
```

**Verilog**

```
/// 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;
    }
}
```

ドキュメンテーション  
コメント

末尾カンマ

SystemVerilogで多用される  
記法の簡素化

ビット幅記法

複合代入演算子

# クロックとリセット

## 専用のクロック・リセット型

- 単一クロックの場合、センシティブティリストを省略可

```
module Counter #(
  param WIDTH: u32 = 1,
)(
  i_clk: input clock ],
  i_rst: input reset ],
  o_cnt: output logic<WIDTH>,
){
  always_ff {
    if_reset {
      o_cnt = 0;
    } else {
      o_cnt += 1;
    }
  }
}
```

Verilog

クロック型  
リセット型

クロックとリセット  
の自動推定

リセット条件の  
専用記法

# クロックとリセット

コンパイル時に極性と同期性を指定可能

- センシティブティリストとリセット条件の自動調整
- 単一のVerylコードからASIC向けとFPGA向けのコンパイル

```
module ModuleA (  
    i_clk: input clock,  
    i_rst: input reset,  
) {  
    always_ff {  
        if_reset {  
        }  
    }  
}
```

Veryl

for ASIC

```
always_ff @ (posedge i_clk, negedge i_rst) begin  
    if (!i_rst) begin  
    end  
end
```

SystemVerilog

for FPGA

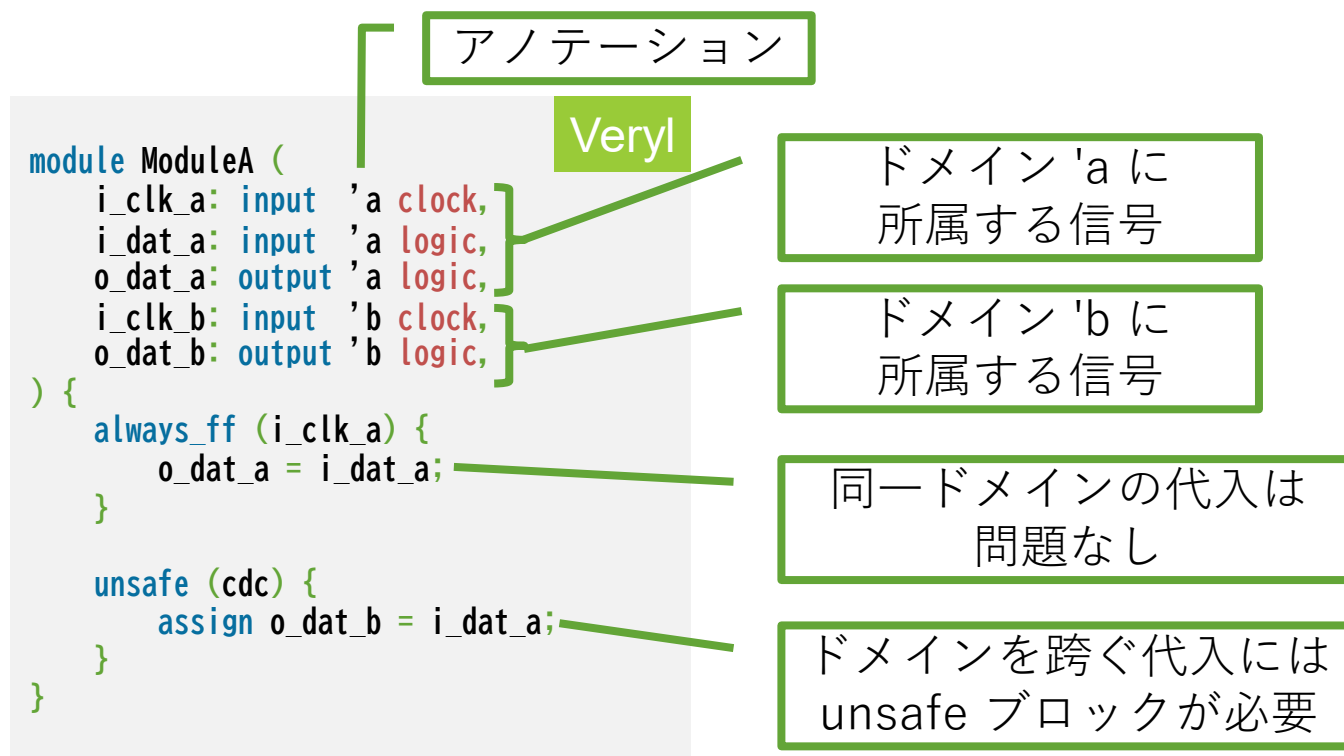
```
always_ff @ (posedge i_clk) begin  
    if (i_rst) begin  
    end  
end
```

SystemVerilog

# クロックとリセット

## クロックドメインアノテーション

- 複数クロックモジュールではアノテーションが必須
- 意図しないクロックドメインクロッシングをエラーとして検出



# インターフェース

## 冗長な記法を簡素化

- modportのデフォルトメンバー指定

```
interface InterfaceA;
  modport master (
    input  accept,
    output command,
    output data
  );
  modport slave (
    output accept,
    input  command,
    input  data
  );
  modport monitor (
    input accept,
    input command,
    input data
  );
endinterface
```

SystemVerilog



```
interface InterfaceA {
  modport master {
    accept : input,
    command: output,
    data   : output,
  }
  modport slave {
    ..converse(master)
  }
  modport monitor {
    ..input
  }
}
```

Verilog

他のmodportの  
反転

全てinput

# インターフェース

## 冗長な記法を簡素化

- インターフェース接続演算子

```
InterfaceA a();  
InterfaceA b();  
  
always_comb begin  
    a.command = b.command;  
    a.data    = b.data;  
    b.accept  = a.accept;  
end
```

SystemVerilog



```
inst a: InterfaceA;  
inst b: InterfaceA;  
  
always_comb {  
    a.master <> b.slave;  
}
```

Verilog

modportに基づく  
代入方向の推定

# ジェネリクス

型パラメータによりコード重複を防ぐ

```
module SramQueueTypeA;  
    SramTypeA u_sram();  
  
    // queue logic  
endmodule  
  
module SramQueueTypeB;  
    SramTypeB u_sram();  
  
    // queue logic  
endmodule  
  
module Test;  
    SramQueueTypeA u0_queue();  
    SramQueueTypeB u1_queue();  
endmodule
```

SystemVerilog



```
module SramQueue::<T: Sram> {  
    inst u_sram: T;  
  
    // queue logic  
}  
  
module Test {  
    // Instantiate a SramQueue by SramTypeA  
    inst u0_queue: SramQueue::<SramTypeA>();  
  
    // Instantiate a SramQueue by SramTypeB  
    inst u1_queue: SramQueue::<SramTypeB>();  
}
```

Verilog

コードの重複

パラメータ化された  
モジュール型

実際のモジュール  
を指定



# Verylのコンセプト

---

合成可能なHDLに最適化された構文

- 生産性と信頼性を高める言語デザイン

可読性の高いSystemVerilogを生成

- SystemVerilogとの高い相互運用性

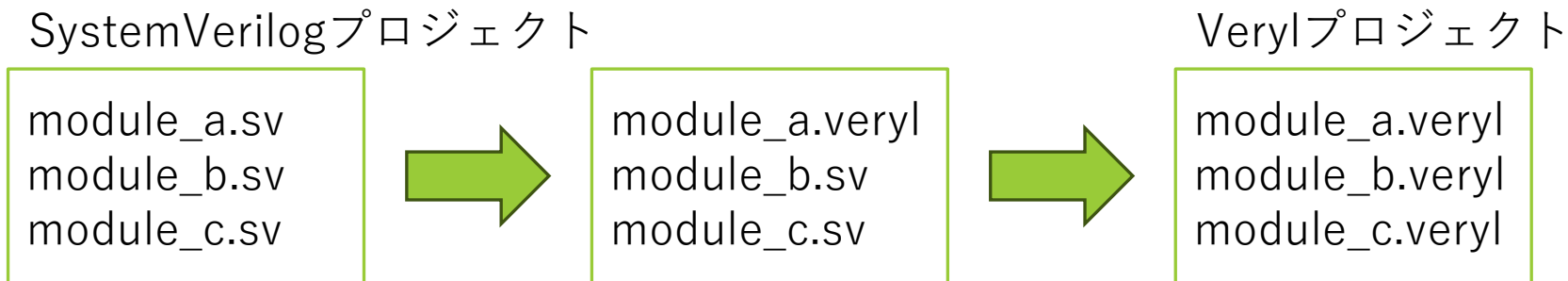
言語に統合されたツール群

- エディタやCIとの連携

# 可読性の高いSystemVerilogを生成

## SystemVerilogとの高い相互運用性

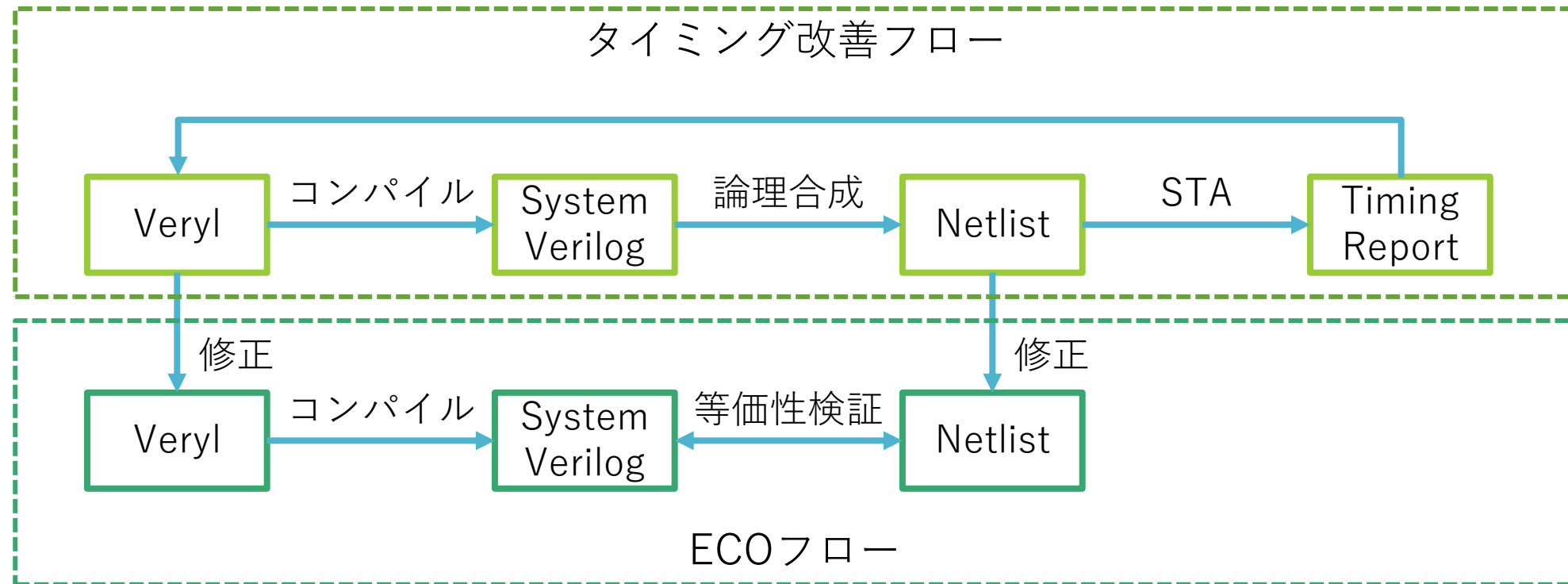
- SystemVerilogのユーザ定義型と完全互換
  - module, interface, package, struct, enum
  - 波形ビューアでstructやinterfaceを使用可能
- 既存のSystemVerilogプロジェクトへ徐々にVerylを導入できる



# 可読性の高いSystemVerilogを生成

## VerilとSystemVerilogの対応付けが容易

- 生成コードを微調整できる
- タイミング改善やpre/post-mask ECOフローの適用可



# Verilのコンセプト

---

合成可能なHDLに最適化された構文

- 生産性と信頼性を高める言語デザイン

可読性の高いSystemVerilogを生成

- SystemVerilogとの高い相互運用性

言語に統合されたツール群

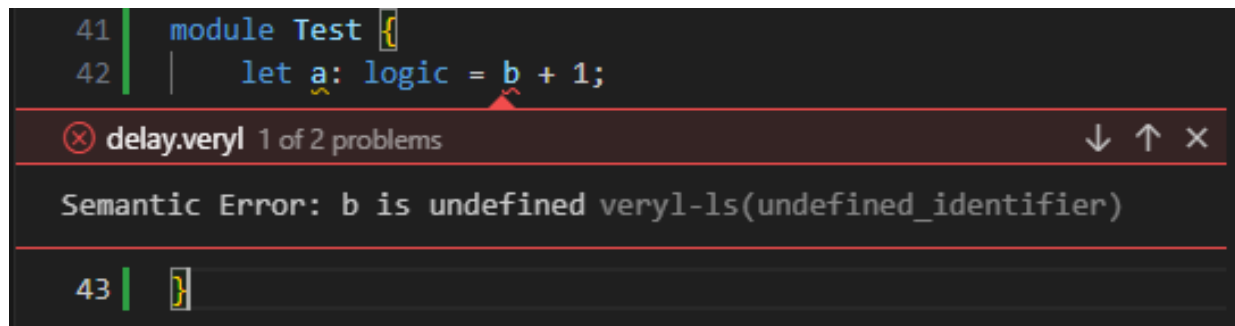
- エディタやCIとの連携

# 言語に統合されたツール群

## リアルタイム診断

- 標準的なLanguage Server Protocolを用いたエディタ統合
- 全てのチェックがリアルタイムに反映される

### Visual Studio Code

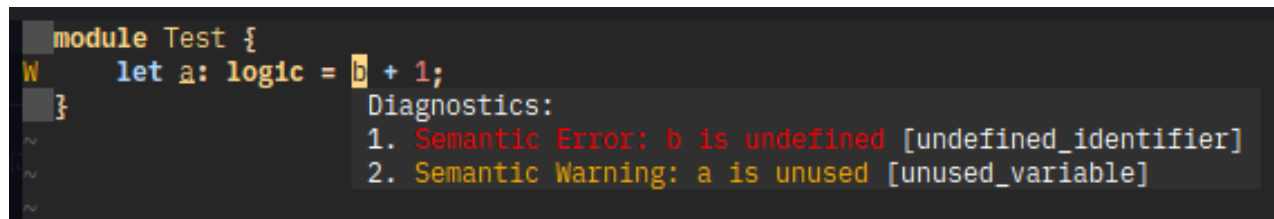


The screenshot shows the Visual Studio Code editor with a Verilog file named `delay.veryl`. The code is as follows:

```
41 module Test {  
42 |   let a: logic = b + 1;  
43 | }
```

A red squiggly line under the variable `b` indicates an error. Below the code, a red error bar displays the message: `Semantic Error: b is undefined veryl-ls(undefined_identifier)`. The error bar also includes a status icon (a red circle with an 'x'), the file name `delay.veryl`, and the text `1 of 2 problems`, along with navigation icons (down, up, and close).

### Vim



The screenshot shows the Vim editor displaying the same Verilog code as the VS Code screenshot. A diagnostics panel is open on the right side of the editor, listing the following issues:

```
module Test {  
W   let a: logic = b + 1;  
}  
~  
~  
~
```

Diagnostics:  
1. Semantic Error: b is undefined [undefined\_identifier]  
2. Semantic Warning: a is unused [unused\_variable]

Visual Studio Codeは、米国Microsoft Corporationの米国およびその他の国における登録商標または商標です。

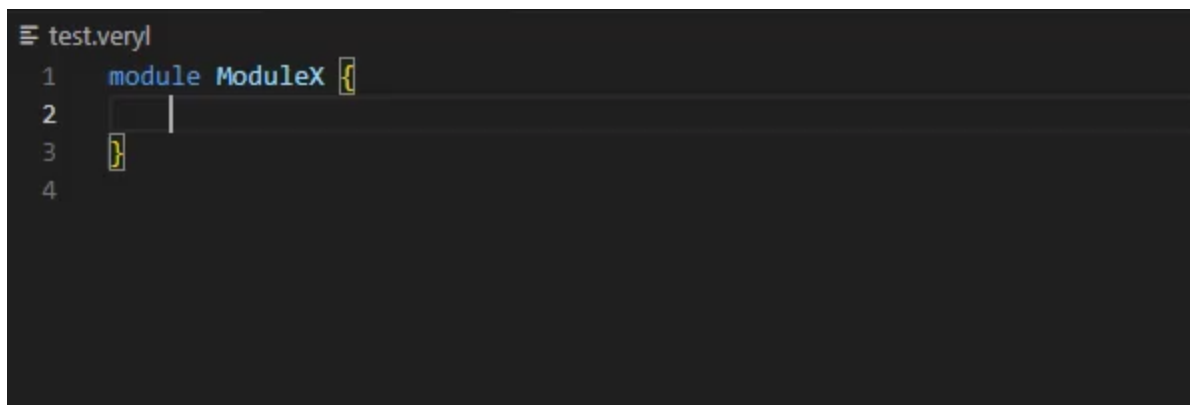
# 言語に統合されたツール群

---

## リアルタイム診断

- 標準的なLanguage Server Protocolを用いたエディタ統合
- 全てのチェックがリアルタイムに反映される

(動画) Visual Studio Codeでのリアルタイム診断



```
test.veryl
1  module ModuleX {
2
3  }
4
```

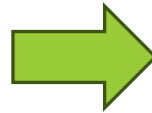
# 言語に統合されたツール群

## ドキュメント自動生成

- マークダウンや波形記述のサポート

```
/// This is a sample module.  
///  
/// ```wavedrom  
/// {signal: [  
///   {name: 'i_clk', wave: 'p.....'},  
///   {name: 'i_dat', wave: 'x.=x..', data: ['data']},  
///   {name: 'o_dat', wave: 'x...=x.', data: ['data']},  
/// ]}  
///```  
pub module Sample #(  
  /// Data Width  
  param WIDTH: u32 = 1,  
)(  
  i_clk: input clock, /// Clock  
  i_dat: input logic<WIDTH>, /// Input Data  
  o_dat: output logic<WIDTH>, /// Output Data  
){}  
`
```

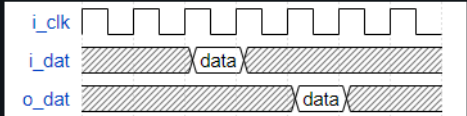
Verilog



**sample\_project**  
0.1.0

Modules  
Sample  
Interfaces  
Packages

**Sample**  
This is a sample module.



**Parameters**

WIDTH	u32	Data Width
-------	-----	------------

**Ports**

i_clk	input clock	Clock
i_dat	input logic	Input Data
o_dat	output logic	Output Data

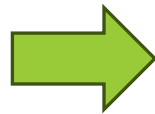
# 言語に統合されたツール群

## 自動フォーマッタ

- クリーンで標準化されたコードレイアウト

```
module Counter #(
  param WIDTH : u32 = 1,
) (
  i_clk: input clock,
  i_rst: input reset,
  o_cnt: output logic < WIDTH >,
){
  always_ff{
    if_reset {
      o_cnt = 0;
    } else {
      o_cnt += 1;
    }
  }
}
```

Veryl



```
module Counter #(
  param WIDTH: u32 = 1,
)(
  i_clk: input clock      ,
  i_rst: input reset      ,
  o_cnt: output logic<WIDTH>,
){
  always_ff {
    if_reset {
      o_cnt = 0;
    } else {
      o_cnt += 1;
    }
  }
}
```

Veryl



# 言語に統合されたツール群

---

## 自動フォーマッタ

- クリーンで標準化されたコードレイアウト

(動画) Visual Studio Codeでのコードフォーマット



The screenshot shows a code editor window titled 'test.veryl' with a dark background. The code is a Verilog module named 'ModuleX'. It contains three variable declarations: 'a' of type 'logic', 'aa' of type 'logic', and 'aaa' of type 'logic<10>'. The code is formatted with automatic indentation and line wrapping. The first line is '1 module ModuleX', the second is '2 {', the third is '3 | var a: logic ;', the fourth is '4 | var aa: logic ;', the fifth is '5 | var aaa: logic<10>;', the sixth is '6 }', and the seventh is '7'. The code is color-coded: 'module' is blue, 'ModuleX' is blue, '{' and '}' are yellow, 'var' is blue, and the variable names and types are white. The line numbers are on the left side of the editor.

```
1 module ModuleX
2 {
3 |   var a: logic ;
4 |   var aa: logic ;
5 |   var aaa: logic<10>;
6 }
7
```

# 言語に統合されたツール群

## ソースマップ

- SystemVerilogからVerilogの位置情報を追跡可能
- 主要なEDAツールのログにVerilog上の位置を追記できる

Synopsys VCS

```
$finish called from file "test.sv", line 12.  
^-- /path/test.verilog:10:8
```

Log

Synopsys Design Compiler

```
Warning: test.sv:67: DEFAULT branch of CASE statement cannot be reached. (ELAB-311)  
^-- /path/test.verilog:127:4
```

Log

AMD Vivado

```
ERROR: [VRFC 10-4982] syntax error near 'endmodule' [/path/test.sv:23]  
^-- /path/test.verilog:18:18
```

Log

# 言語に統合されたツール群

---

## その他の機能

- 標準ライブラリ
- 組み込みの単体テスト
- 依存関係マネージャ
- ツールチェーンマネージャ
- 公式のGitHub Action / Dockerイメージ

# 目次

---

## 開発の動機

- SystemVerilogの課題
- 既存のAlt-HDLの課題

## Veryl: SystemVerilogに代わる新しいHDL

- コンセプト
- 主な機能と利点

## Verylの開発と利用状況

- プロジェクト概況
- 活用事例

## まとめ

# プロジェクト概況

## GitHub

- 作成日 : 2022/12
- コミット : 3330
- リリース : 54
- プルリクエスト : 3 Open, 1258 Closed
- 貢献者 : 16

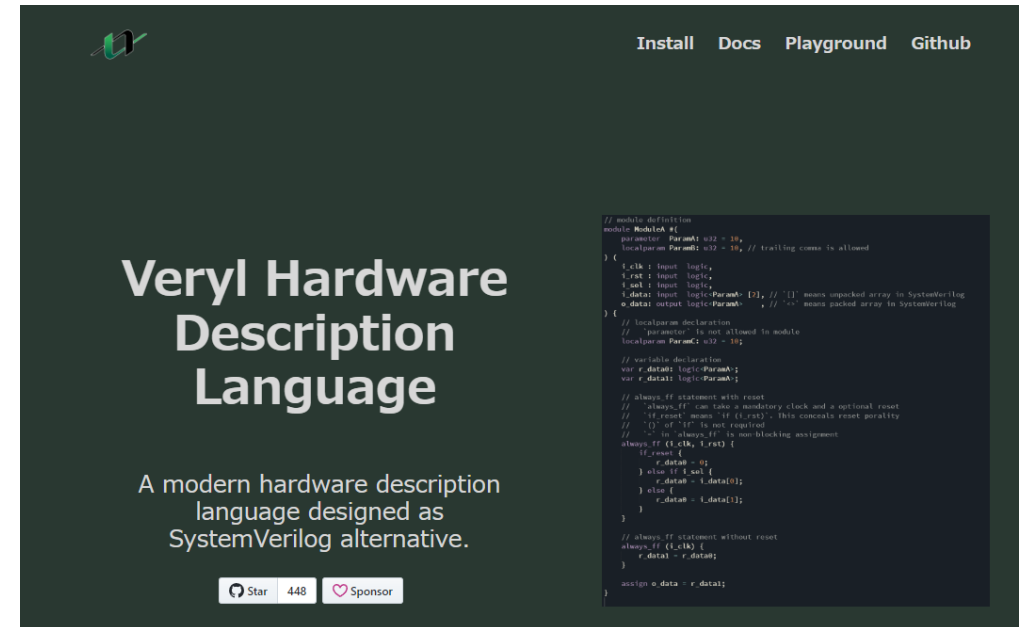
## リソース

- 公式サイト
- 言語リファレンス (日本語版)
- プレイグラウンド

: <https://veryl-lang.org>

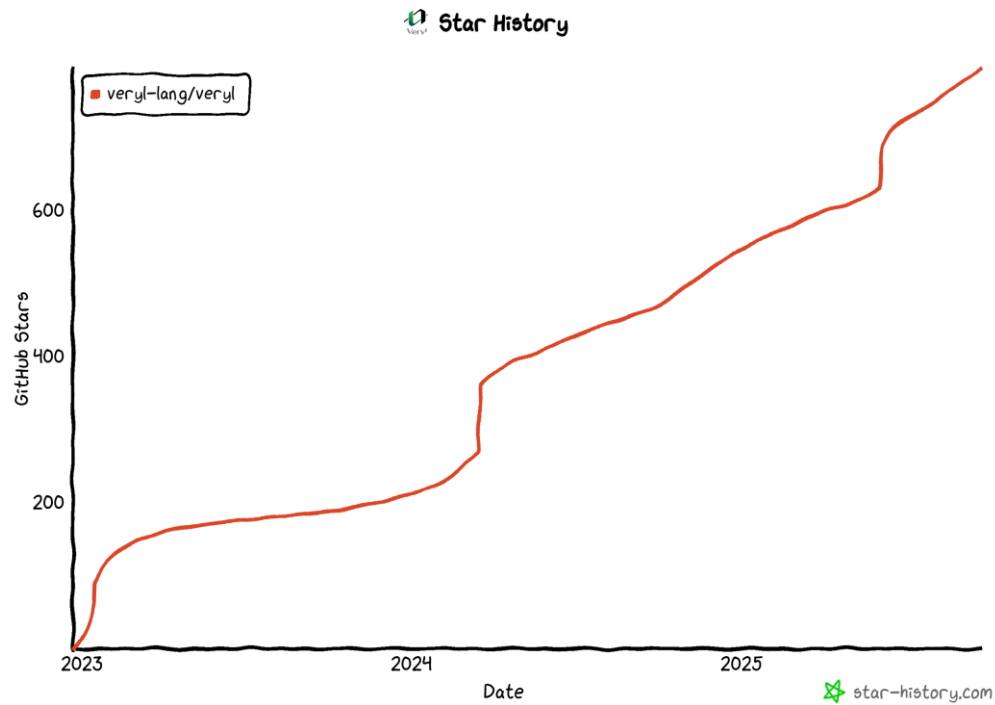
: <https://doc.veryl-lang.org/book/ja/>

: <https://doc.veryl-lang.org/playground/>

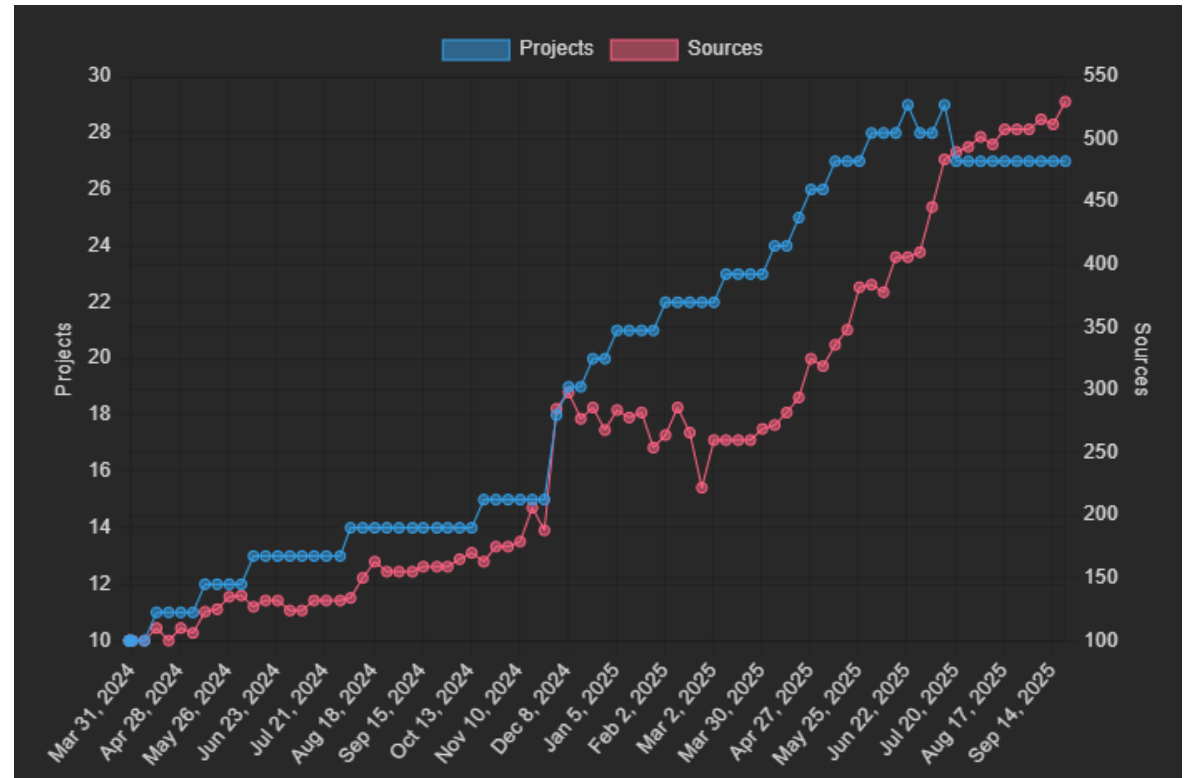


# プロジェクト概況

GitHubスター (2022/12~)



GitHub上のプロジェクト (2024/03~)



# プロジェクト概況

---

## Verylをサポートするプロジェクト

- RgGen：制御レジスタ生成ツール
  - <https://github.com/rggen/rggen>
  - Verylのコード生成をサポート
- GitLab：Gitリポジトリホスティング
  - <https://gitlab.com/>
  - Verylの構文ハイライトサポート

## 今後のサポート見込み

- GitHub
  - GitHub上のファイル数2000以上で構文ハイライト対応
- Qiita
  - 構文ハイライトライブラリへのマージ待ち

# プロジェクト概況

---

## 対外発表

- 2024/8/29 : DVCon Japan 2024
- 2025/6/8 : VLSI Symposium 2025 Workshop
- 2025/6/13 : Satellite Open Source Silicon Workshop
- 2025/6/21 : OSCAR Workshop co-located with ISCA 2025
- 2025/10/7 : Design Solution Forum 2025

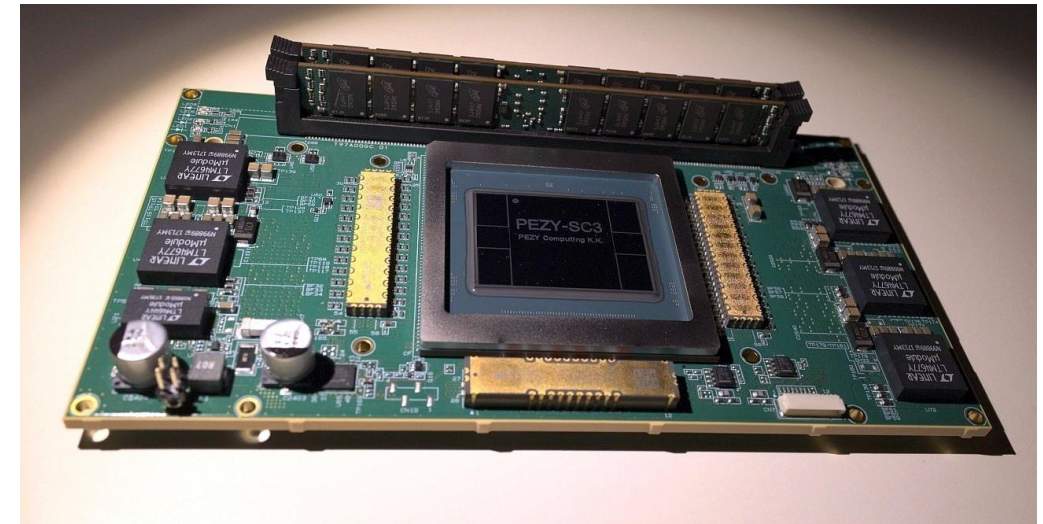


# 活用事例

## PEZY Computing

- HPC/AIアクセラレータPEZY-SCxシリーズの開発
  - 2026年にPEZY-SC4sをリリース予定
- PEZY-SC5の開発にVerylを投入済み
  - 5万行のVerylと600万行のSystemVerilogの組み合わせ
  - 今後も新規実装部分を中心にVerylを増やしていく予定

PEZY-SC3



# 活用事例

## bluecore

- Linuxブート可能なオープンソースのRISC-V実装
  - <https://github.com/nananapo/bluecore>
  - 4千行のVerilogコード
- 書籍「Verilogで作るCPU」を技術書典にて頒布
  - Web版もあり：<https://cpu.kanataso.net>



## rice

- オープンソースのRISC-V実装
  - <https://github.com/taichi-ishitani/rice>
  - 5千行のVerilogコード、RgGenによるCSR自動生成
- VerilogとUVMテストベンチの統合事例

# 活用事例

---

ルレオ工科大学（スウェーデン）

- MIPS32のCPUを実装する実習コース
- 発展課題としてVerylを選択可能

# 目次

---

## 開発の動機

- SystemVerilogの課題
- 既存のAlt-HDLの課題

## Veryl: SystemVerilogに代わる新しいHDL

- コンセプト
- 主な機能と利点

## Verylの開発と利用状況

- プロジェクト概況
- 活用事例

## まとめ

# まとめ

---

## Veryl: SystemVerilogに代わる新しいHDL

- 合成可能なHDLに最適化された構文
- 可読性の高いSystemVerilogの生成
- 言語に統合されたツール群

## 検討中の新機能

- ネイティブシミュレータ
- SDCサポート