Смотрим на современный инструмент для FPGA

Вступление

Область разработки для ПЛИС, довольна консервативна и неповоротлива. Поскольку она узкоспециализирована, то новые инструменты и среды появляются редко, а старые инструменты имеют свои слабости в самой своей основе и перекладывать их на новые рельсы уже ни кто не будет.

Поэтому появление нового инструмента всегда маленькое, но событие. Сегодня мы посмотрим на Veryl-lang. Интересно, что на самом деле это не столько язык, сколько среда для разработки. Такой маленький мирок со своими современными и удобными функциями.

```
Veryl еще сыроват, поэтому использовать его в рабочих задачах еще не представилась возможность. Но судя по темпам развития, его можно будет рассматривать в ближайшем будущем.
```

Обзор инструментов

Составными частями этого мирка являются следующие очевидные для остального ИТ мира вещи:

1. Собственно сам язык
2. Транспайлер в SystemVerilog
3. Линтер\Форматер\Языковой сервер
4. Интеграция с VSCode, vim\neovim
5. Пакетный менеджер
6. Встроенная документация

Сходу отмечу плюс: Наличие линтера/языкового сервера. Казалось что такого в 2024 году? Вы когда ни будь видели нормальный линтер для Verilog или SytemVerilog (с HDL насколько я знаю та же проблема)? Если посмотреть, то их особо и нет. Да он не такой умный как RustAnalyzer, но его наличие еще в сыром продукте это жирный плюс.

Что же в основе?

Veryl и его инструменты написаны на Rust. Кстати автор dalance, так же является разработчиком языкового сервера для SytemVerilog svls, который тоже написан на Rust. Вот так любовь одного Японского разработчика к Rust и ПЛИС делает этот мир чуточку лучше. Rust имеет строгие правила работы с данными и кодом, поэтому написать сложное и комплексное приложение на нем будет проще, банально в нем будет меньше странных вылетов и багов. Rust показал как нужно делать современный язык, сразу включая в него пакетный менеджер, удобство работы с языком из коробки, но при этом оставляя гибкость для тех кому это действительно нужно. Поэтому вся экосистема крутиться вокруг Rust инструментов. Что бы воспользоваться Veryl нужно установить Rust и после этого можно начинать:

```
cargo install veryl veryl-ls
```

И все - ни каких танцев с компиляторами, MXE (Yosys), Cmake (Verilator), makefiles, скриптами(ICarus) ни какой тяжелой Java(Chisel)... Поэтому хоть на Linux, хоть на Windows - одна строчка и поехали.

Первый проект

Создадим проект - мы же не хотим писать ручками CMakeFile попутно разбираясь с ним (Verilator), поэтому снова одна команда:

```
veryl new hello
```

Все теперь у нас есть папка hello с одним файлом настроек в формате Toml: `Veryl.toml`. Файл содержит почти стандартный набор параметров для Rust проектов. Это имя проекта, версия и все. Остальные варианты настроек можно посмотреть в документации. Там есть такие вещи как список авторов, настройки сгенерированного кода и зависимости проекта.

```
[project]
name = "hello"
version = "0.1.0"
```

Имя проекта и версия. Создаем папку `src` и начинаем писать код `src/hello.veryl`

```
module Hello (){
    initial {
        $display("Hello world!");
    }
}
```

Тем кто знает Verilog или SytemVerilog здесь все будет понятно. У нас ест модуль `Hello` и все что он делает при инициализации отправляет сообщение в консоль `Hello world`.

Хорошо, сгенерируй нам SystemVerilog файлик:

```
veryl build
```

Получаем на выходе:

```
hello
│   hello.f
│   Veryl.lock
│   Veryl.toml
├──dependencies
```

```
└──src
        hello.sv
        hello.veryl
```

Так как по умолчанию он генерирует файлы рядом с исходниками, то он заботливо создал файл `hello.f` в котором лежит список всех сгенерированных файлов

```
hello\src\hello.sv
```

Мне такое не нравится поэтому попросим его класть их в отдельную папку `target` для этого добавим в `Veryl.toml`

```
...
[build]
target = {type = "directory", path="target"}
```

и еще раз соберем.

```
.
│   hello.f
│   Veryl.lock
│   Veryl.toml
├──dependencies
├──src
│       hello.veryl
└──target
        hello.sv
```

Отлично теперь это можно удобно скопировать себе

Что-то посложнее

Попробуем задачку чуть сложнее - расчет среднего за окно, шагающее и скользящее. В комментариях кода есть пояснения некоторых особенностей, стоит обратить внимание.

Начнем с шагающего:

```
/// Это документация, а не комментарий. И она встроена по умолчанию
// А это комментарий к коду
/// Шагающее среднее за SIZE
pub module StepAverage #(
    parameter WIDTH: u32 = 8, /// Разрядность входных и выходных данных
    parameter SIZE: u32 = 4, /// Размер окна для расчета, только степень двойки
    localparam CNT_SIZE: u32 = $clog2(SIZE), // Почти все функции SystemVerilog
доступны. Об этом будет позже в недостатках
```

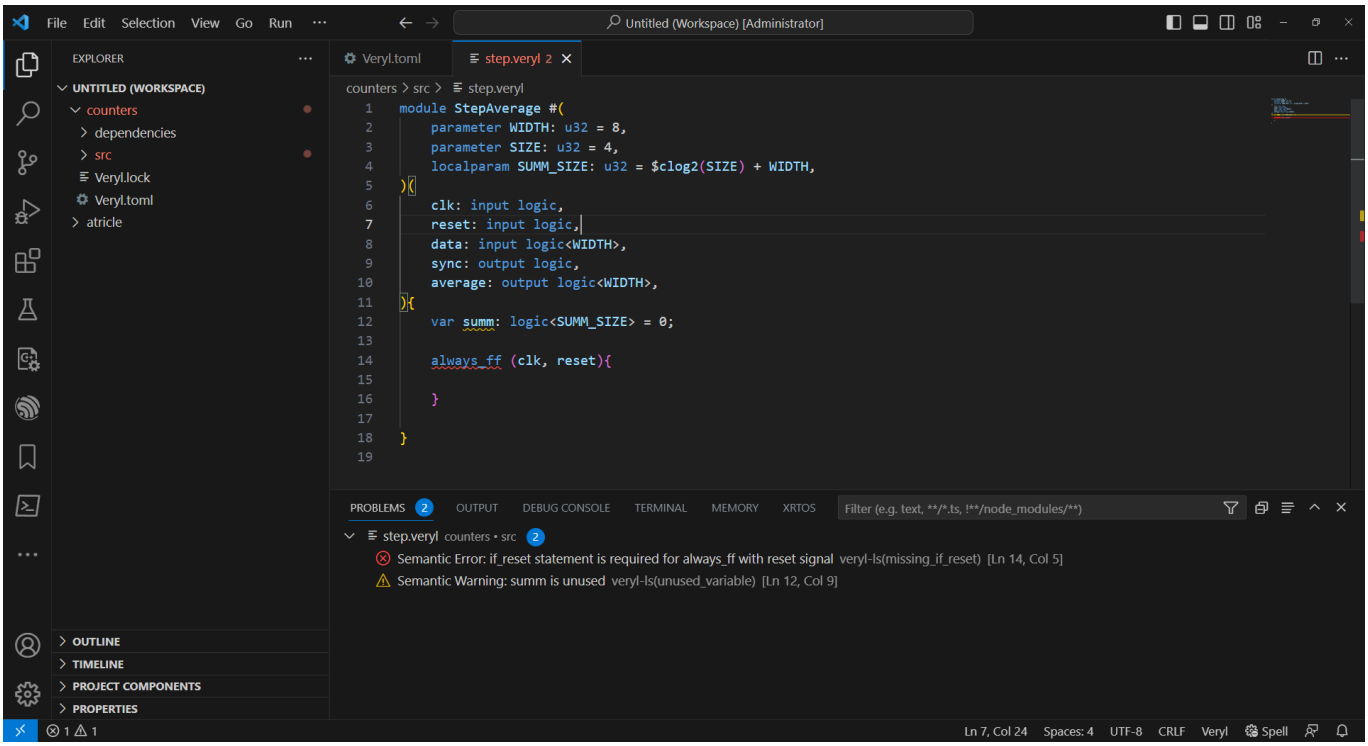
```
    localparam SUM_SIZE: u32 = $clog2(SIZE) + WIDTH,
)(
    clk: input logic, /// Клоки данных
    reset: input logic, /// Сигнал сброса. Уровень определяется по умолчанию для
проекта
    data: input signed bit<WIDTH>, /// Знаковые входные данные
    sync: output logic, /// Сигнал синхронизации среднего
    average: output signed logic<WIDTH>, /// Среднее за окно
){
    var sum: logic<SUM_SIZE> = 0;
    var cnt: logic<CNT_SIZE> = 0;

    /// Основной блок расчетов
    always_ff (clk, reset){
        if_reset{ // А вот тут хитро. Второй параметр always_ff должен быть reset,
поэтому это условие обязательно, без него ошибка.
            sum = 0;
            cnt = 0;
            average = 0;
            sync = 0;
        } else {
            sum += data;
            cnt += 1;
            sync = 0; // Интересно, что все присвоение внутри always_ff будет
неблокирующие, об этом написано в документации. Как по мне это минус, что это
нельзя контролировать, бывает нужно.
            if cnt == SIZE{ // Интересный момент, что для оператора меньше\больше
нужно использовать "<:"\">:".
                average = sum / SIZE;
                sync = 1;
                sum = 0;
                cnt = 0;
            }
        }
    }
}
```

Как вы заметили синтаксис вдохновлен Rust, но не перегружен, это скорее ограничения самой платформы (ПЛИС), поскольку нельзя ввести такие понятия как время жизни. Сразу скажу, что для generics структуры не предусмотрены и будут ли не могу сказать.

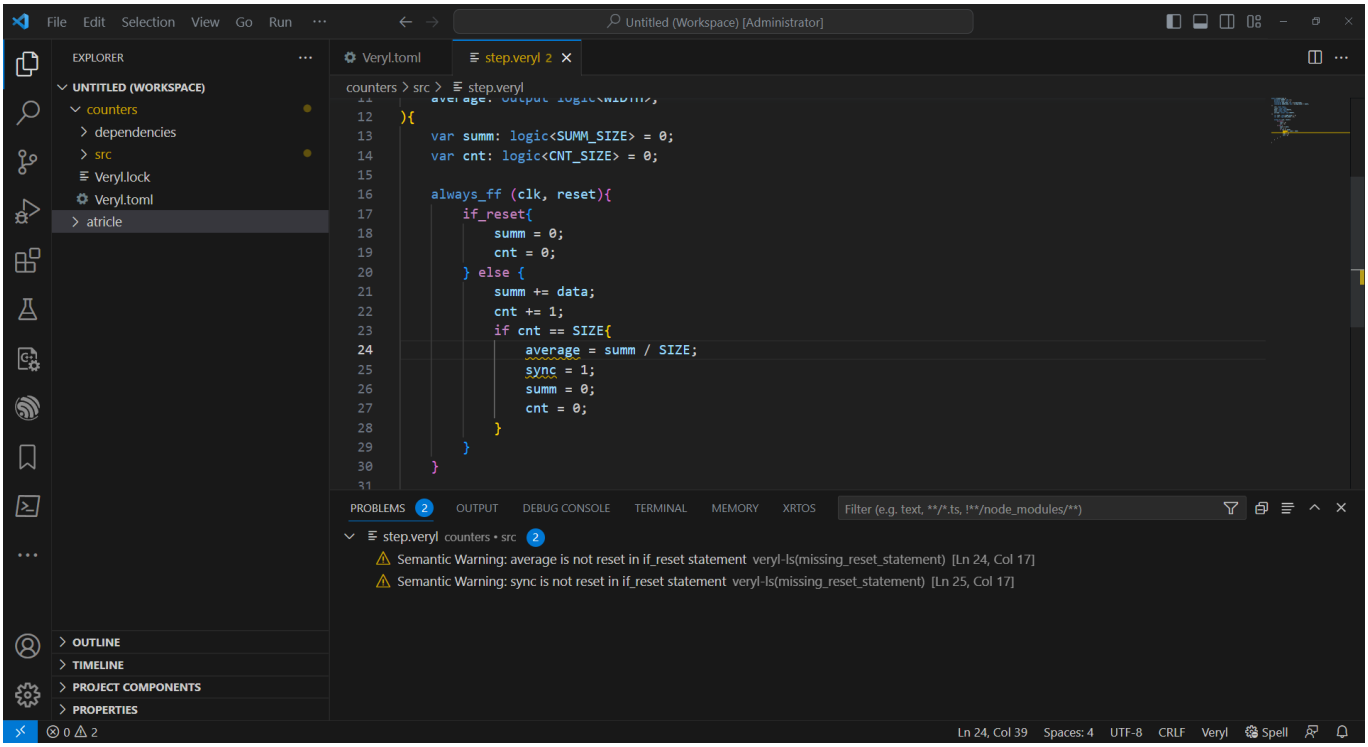
Походу написания кода можно видеть как линтер подсказывает:

- Синтаксические ошибки
- Стилистические предупреждения
- Предупреждения о возможных логических ошибках



Тут линтер подсказывает, что нужно использовать `if_reset` и что sum не используется.

А дальше кое-что интересное. Линтер сообщает, что переменные не сбрасываются при reset.



Линтер работает шустро и без тормозов, очень приятно.

Теперь реализуем скользящее окно.

```
/// Скользящее среднее за окно
pub module SlidingAverage #(
    parameter WIDTH: u32 = 8, /// Разрядность входных и выходных данных
    parameter SIZE: u32 = 4, /// Размер окна. Только степень двойки
    localparam CNT_SIZE: u32 = $clog2(SIZE),
```

```
    localparam SUM_SIZE: u32 = $clog2(SIZE) + WIDTH,
)(
    clk: input logic,
    reset: input logic, /// Сигнал сброса
    data: input signed bit<WIDTH>, /// Знаковые входные данные
    average: output signed logic<WIDTH>, /// Среднее за окно на каждом шаге
) {
    var cache: logic<WIDTH> [SIZE] = {0 repeat SIZE}; //Unpacked массив

    always_ff(clk, reset){
        if_reset{
            cache = {0 repeat SIZE}; //repeat - очень приятно, наглядно говорит,
что 0 должен быть повторен N раз.
            average = 0;
        } else {
            cache[msb-1:0] = {data, cache[msb-1:1]}; //msb - это индекс старшего
бита. выводиться автоматически, но не всегда.
            average = 0;
            for i: u32 in 0..SIZE{ // Вот тут линтер требует указать тип, иначе
ошибка.
                average += cache[i];
            }
        }
    }
}
```

Ну что же теперь взглянем, на то что сгенерирует Veryl. Шагающее окно:

```
/// Это документация, а не комментарий. И она встроена по умолчанию
// А это комментарий к коду
/// Шагающее среднее за SIZE
module average_StepAverage #(
    parameter  int unsigned WIDTH     = 8                  , /// Разрядность
входных и выходных данных
    parameter  int unsigned SIZE      = 4                  , /// Размер окна для
расчета, только степень двойки
    localparam int unsigned CNT_SIZE  = $clog2(SIZE)       , // Почти все функции
SystemVerilog доступны. Об этом будет позже в недостатках
    localparam int unsigned SUM_SIZE = $clog2(SIZE) + WIDTH
) (
    input  logic                    clk    , /// Клоки данных
    input  logic                    reset  , /// Сигнал сброса. Уровень
определяется по умолчанию для проекта
    input  bit   signed [WIDTH-1:0] data   , /// Знаковые входные данные
    output logic                    sync   , /// Сигнал синхронизации среднего
    output logic signed [WIDTH-1:0] average /// Среднее за окно
);
    logic [SUM_SIZE-1:0] sum;
    assign sum = 0;
    logic [CNT_SIZE-1:0]  cnt ;
    assign cnt = 0;
```

```
    /// Основной блок расчетов
    always_ff @ (posedge clk) begin
        if (!reset) begin // А вот тут хитро. Второй параметр always_ff должен
быть reset, поэтому это условие обязательно, без него ошибка.
            sum    <= 0;
            cnt     <= 0;
            average <= 0;
            sync    <= 0;
        end else begin
            sum <= sum + (data);
            cnt  <= cnt  + (1);
            sync <=  0; // Интересно, что все присвоение внутри always_ff будет
неблокирующие, об этом написано в документации. Как по мне это минус, что это
нельзя контролировать, бывает нужно.
            if (cnt == SIZE) begin // Интересный момент, что для оператора
меньше\больше нужно использовать "<:"\">:".
                average <= sum / SIZE;
                sync    <= 1;
                sum    <= 0;
                cnt     <= 0;
            end
        end
    end
endmodule
```

Скользящее окно

```
/// Скользящее среднее за окно
module average_SlidingAverage #(
    parameter  int unsigned WIDTH     = 8                  , /// Разрядность
входных и выходных данных
    parameter  int unsigned SIZE      = 4                  , /// Размер окна.
Только степень двойки
    localparam int unsigned CNT_SIZE  = $clog2(SIZE)       ,
    localparam int unsigned SUM_SIZE = $clog2(SIZE) + WIDTH
) (
    input  logic                    clk    ,
    input  logic                    reset  , /// Сигнал сброса
    input  bit   signed [WIDTH-1:0] data   , /// Знаковые входные данные
    output logic signed [WIDTH-1:0] average /// Среднее за окно на каждом шаге
);
    logic [WIDTH-1:0] cache [0:SIZE-1];
    assign cache = {{SIZE{0}}};

    always_ff @ (posedge clk) begin
        if (!reset) begin
            cache   <= {{SIZE{0}}};
            average <= 0;
        end else begin
            cache[((SIZE) - 1) - 1:0] <= {data, cache[((SIZE) - 1) - 1:1]};
            average          <= 0;
            for (int unsigned i = 0; i < SIZE; i++) begin
```

```
                average <= average + (cache[i]);
            end
        end
    end
endmodule
```

Как видим не появилось, ни каких оверхедов. Код красиво отформатирован, все комментарии и документация сохранены. Все как мы просили, так и было сгенерировано.

Теперь нюансы на внимательность

- `reset`
  - Ни где не указано `active_low` или `active_high`. Потому что это выбирается на уровне проекта, для всего проекта. Что с одной стороны удобно - в проекте везде всегда одинаково, с другой всегда найдутся не довольные этим. Настройка `reset_type = "sync_low"`. Еще может быть `async`.
  - `reset` не был помещен в `always_ff`, поскольку он `sync`.
- `clk` Та же история что и с `reset` в настройках проекта указывается `clock_type = "posedge"`
- Как уже было сказано в комментариях все присвоения неблокирующие `<=`

Структура проекта

Определение модуля через ключевое слово, имя, описания парметров и цепей. Все как в SystemVerilog. Для вызова модуля ни чего сложно, но мне нравиться, что используется ключевое слово и двоеточие, для отделения имени инстанса от имени модуля. Честно я регулярно забываю в SytemVerilog, что идет сначала имя модуля или имя инстанса. Удобно, что если имена параметров совпадают, то их можно просто указать. А вот соответствие типов он не проверяет, от части это логично... Создаим папку `src/package` и положим туда два модуля `moduleA.veryl` и `moduleB.veryl` а в `src/calls.veryl` вызовим `moduleB`

```
module ModuleA #(
    parameter A: u32 = 10,
    parameter B: string = "",
    parameter C: i32 = -10,
    parameter D: u32 = 0
)(
    clk: input logic
){

}
```

```
module ModuleB #(
    parameter A: u32 = 4,
    parameter D: string = ""
    )(
        clk: input logic
```

```
){
    inst instB: ModuleA #(
        A,
        B: "строка",
        D
    ) (
        clk
    );
}
```

```
module ModuleCall(
    clk: input logic,
){
    inst modB: ModuleB(
        clk
    );
}
```

Если внимательно посмотреть, то можно заметить, что мы ни где не указывали, откуда брать вызываемый модуль. А все потому что для Veryl вся папка исходников одна глобальная область видимости. С одной стороны это не плохо, так например он запрещает создавать два модуля с одинаковым именем, в разных папках\файлах. К тому же так проще использовать проект в каком нибудь Verilator, достаточно указать папку и он сожрет её целиком и не подавится из-за одинаковых имен. С другой стороны в современном мире хотелось бы видеть `import` и разбиение проекта на части. Есть еще работа с `package`, но классов нет, поэтому туда сейчас можно вписать только `localparam`, `function` и типы. Обращение к пакетам: `PackageRoot::WIDTH`

```
pub package PackageRoot{
    struct StructA{
        a: logic,
    }
    localparam WIDTH: u32 = 16;
    function sum(a: input logic<WIDTH>, b: output logic<WIDTH>){
        b = a + b;
    }
}
```

Ну и сгенерированный код для все этого:

```
module hello_ModuleA #(
    parameter int unsigned A = 10 ,
    parameter string       B = "" ,
    parameter int signed   C = -10,
    parameter int unsigned D = 0
) (
    input logic clk
```

```
);

endmodule
```

```
module hello_ModuleB #(
    parameter int unsigned A = 4 ,
    parameter string       D = ""
) (
    input logic clk
);
    hello_ModuleA #(
        .A (A             ),
        .B ("строка"),
        .D (D             )
    ) instB (
        .clk (clk)
    );
endmodule
```

```
module hello_ModuleCall (
    input logic clk
);
    hello_ModuleB modB (
        .clk (clk)
    );
endmodule
```

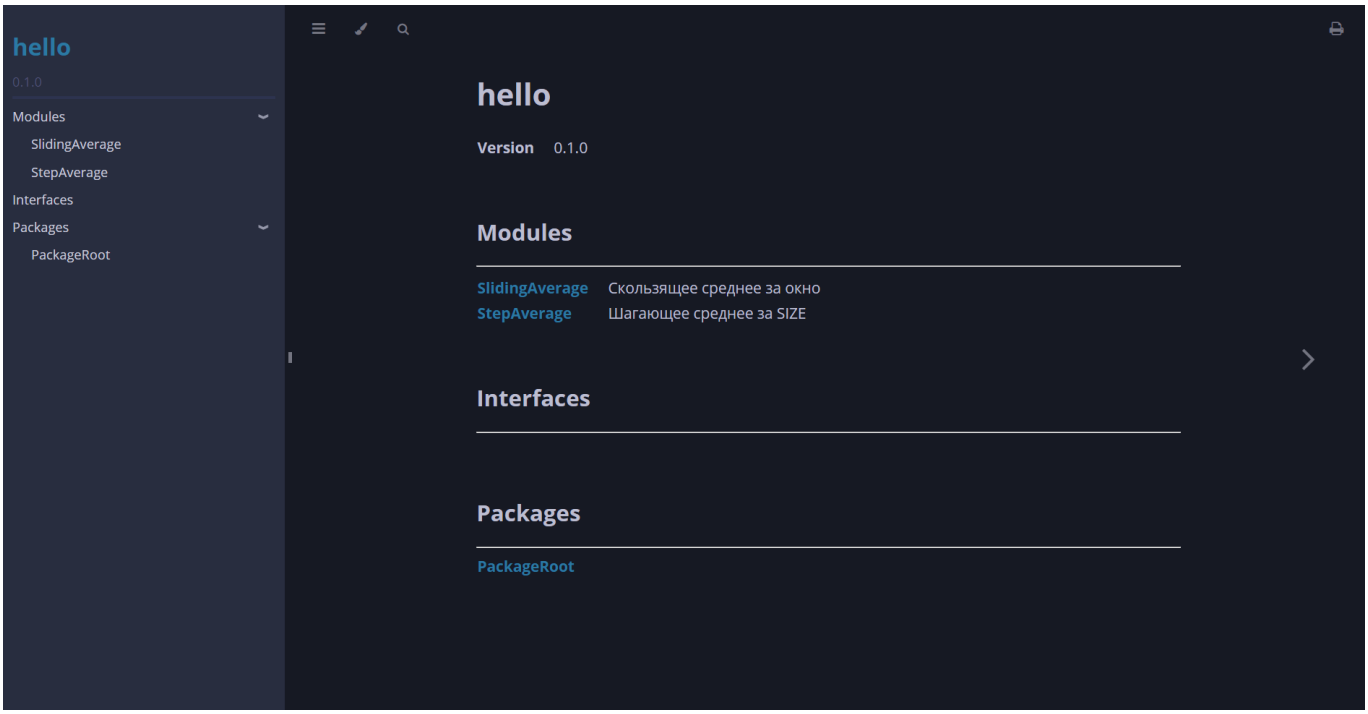
```
package hello_PackageRoot;
    typedef struct packed {
        logic a;
    } StructA;
    localparam int unsigned             WIDTH = 16;
    function automatic void sum(
        input  logic        [WIDTH-1:0] a    ,
        output logic        [WIDTH-1:0] b
    ) ;
        b     = a + b;
    endfunction
endpackage
```

Документация

Сразу есть возможность сгенерировать документацию в виде сайта. Для этого просто

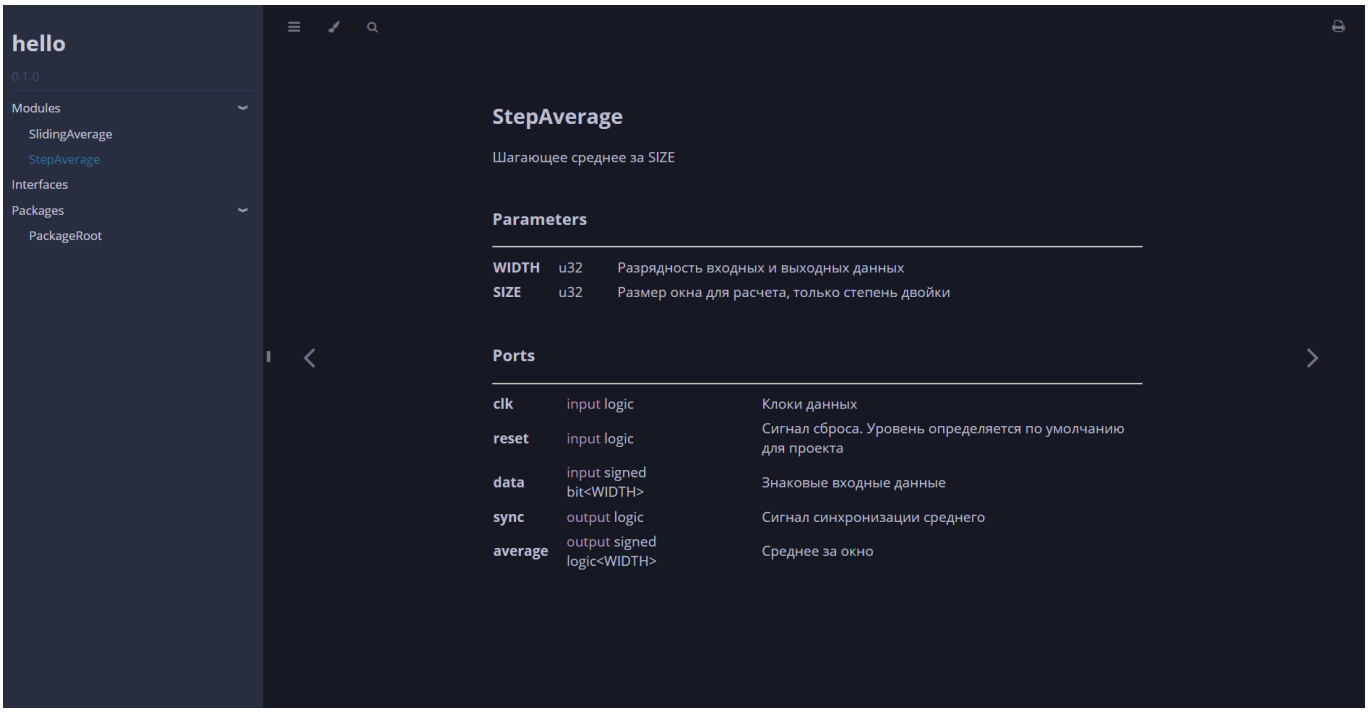
```
veryl doc
```

и создается папка `doc` в ней нам интересен файл `index.html`.



Как видно в документацию попали только те модули и пакеты для которых было указано, что они `pub`.

Откроем `StepAverage`



Стандартное сгенерированное описание. Не такое интересное как у TerosHDL, но уже что-то.

Пример сырости Veryl

Простой пример, входные данные беззнаковые и необходимо их положить в переменную большей разрядности так как будто они знаковые (в доп. коде)

```
module BreakingTest (
    clk: input logic,
    data: input logic<8>,
){
    var inner: signed logic<16>;
    always_ff(clk){
        inner = $singed(data);
    }
}
```

А вот нельзя. Такую функцию как `$signed` он почему-то не знает и считает, что имя пересекается с ключевым словом `signed` для переменных.

```
Error:   × veryl check failed
Error: undefined_identifier (link)
  × singed is undefined
    ╭─[src\tests.veryl:6:1]
 6 │     always_ff(clk){
 7 │         inner = $singed(data);
   .                 ──┬──┬──
   .                   └──┴── Error location
 8 │     }
   ╰────
  help:
```

Ну что же придется делать руками:

```
...
    inner = {data[7] repeat 8, data[msb:0]};
...
```

А вот теперь он не смог рассчитать размер `data`

```
Error:   × veryl check failed
Error: unknown_msb (link)
  × resolving msb is failed
    ╭─[src\tests.veryl:9:1]
 9 │         // inner = $singed(data);
10 │         inner = {data[7] repeat 8, data[msb:0]};
   .                                     ┬┬
   .                                     └┴── Error location
11 │     }
   ╰────
  help:
```

Ну что же придется вручную всё

```
inner = {data[7] repeat 8, data[7:0]};
```

И на выходе получаем:

```
module hello_BreakingTest (
    input logic         clk ,
    input logic [8-1:0] data
);
    logic signed [16-1:0] inner;
    always_ff @ (posedge clk) begin
        // inner = $singed(data);
        // inner = {data[7] repeat 8, data[msb:0]};
        inner <= {{8{data[7]}}, data[7:0]};
    end
endmodule
```

Скорее всего это какая-то ошибка, или нужно знать какой-то алиас на эту функцию, но я не смог найти этого в документации.

Итоги

- Проект интересный и комплексный, сразу создает свою среду обитания. Да так делают многие проекты, но использовать python для работы с плис, для меня звучит бредово. Python слишком гибкий\динамический, а хочется какой-то надежности и основательности.
- Простота установки и использования. Многие проекты грешат не простым процессом установки и это под Линукс, что там под Windows, даже смотреть страшно.
- Хотелось бы протестировать пакетный менеджер, но мне такое не очень нужно, поэтому не смотрел.
- Есть возможность вызова SystemVerilog модулей, функций, пакетов и интерфейсов прямо из Veryl кода: `$sv::ModuleSV`
- Пока сыроват. Зато опенсорс и проект в который можно залезть руками, относительно быстро разобраться что и как и дописать чего не хватает.
- Еще не умеет подсказывать параметры и аргументы модулей. Но сами имена модулей подсказывает

Мне явно интересен этот проект. Писать для ПЛИС и так нудное занятие и такие инструменты вселяют надежду в светлое будущее. Как вы относитесь к свежим (и местами сырым) проектам, развивающим средства разработки для ПЛИС - делитесь в комментариях.