Saul Borges Nobre

Implementação de um gerador de ambiente de verificação de IPs

Campina Grande, Brasil

25 de dezembro de 2018

Saul Borges Nobre

Implementação de um gerador de ambiente de verificação de IPs

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Orientador: Gutemberg Gonçalves dos Santos Junior, D.Sc.

Campina Grande, Brasil

25 de dezembro de 2018

Saul Borges Nobre

Implementação de um gerador de ambiente de verificação de IPs

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Trabalho aprovado em:

Gutemberg Gonçalves dos Santos Junior, D.Sc.
Orientador

Marcos Ricardo Alcântara Morais, D.Sc.
Convidado

Campina Grande, Brasil
25 de dezembro de 2018

Dedico este trabalho aos meus amados pais, José Borges da Silva Filho e Maria BetâniaNobre Costa Borges, à minha família e a minha namorada Nathália Guerra de Sousa.

Agradecimentos

Primeiramente, agradeço a Deus pelo dom da vida. Em seguida, aos meus pais, José Filho e Maria Betania, por todo o apoio e suporte que me deram nessa longa jornada ate aqui e por sempre acreditarem que eu seria capaz.

À minha irmã, Débora, que embora não ter convivido comigo nos últimos anos, agradeço por ter me feito, inconscientemente, evoluir tanto no meu lado profissional quanto no lado pessoal.

À minha namorada, Nathália, agradeço por sempre ter estado comigo nas horas que precisei, sempre ter me ajudado a levantar quando cai, sempre ter me incentivado a ser melhor e evoluir. Agradeço por ter sido minha companheira e confidente, e acima de tudo, ter sido paciente.

Agradeço aos Pseudosmitos, grupo de estudantes, que sempre estiveram presentes ao longo dessa caminha, em varias noites de estudos e momentos de descontração. Agradeço , também, por todas as risadas que foram dadas, por todo apoio e incentivo que nos demos ao longo dessa jornada.

Agradeço aos grupos que participei ao longo da graduação, PET, RAMO Estudantil IEEE, X-MEN Lab, que me trouxeram engradecimento profissional e pessoal.

Aos meus professores, por todos os ensinamentos e oportunidades que me foram dadas, em especial ao professor Gutemberg, que me orientou nessa reta final e sempre acreditou no meu potencial.

"First, have a definite, clear practical ideal; a goal, an objective. Second, have the necessary means to achieve your ends; wisdom, money, materials, and methods. Third, adjust all your means to that end"

Aristotle

Resumo

O gerador de ambiente de verificação de IPs é uma ferramenta que deve auxiliar verificadores, de maneira a automatizar o processo de implementação de uma versão *stub* do ambiente. Este basei-se no uso de informações contidas em um arquivo e na determinação de um padrão a ser seguido para o *stub* do ambiente. O ambiente gerado possui um padrão comum e muito utilizado. A implementação deu-se utilizando a linguagem de alto nível Java.

Palavras-chaves: Java, Verificação, IP.

Abstract

The IP verification environment generator is a tool that should aid verifiers in order to automate the process of implementing a textit stub version of the environment. This is based on the use of information contained in a file and on determining a pattern to be followed for the textit stub environment. The generated environment has a common and widely used standard and was implemented using the Java.

Key-words: Java, Verification, IP.

Lista de ilustrações

Lista de tabelas

Lista de Códigos

Sumário

1 Introdução

O processo de desenvolvimento de um IP é composto de etapas que podem ser divididas pelos setores presentes em uma empresa da área: modelagem, responsável por trazer o "modelo" digital do que o IP deverá fazer; design, o qual implementa a funcionalidade do IP em baixo nível; verificação, responsável por checar a corretude do design perante ao "modelo" entregue pela equipe de modelagem; e, por último, *backend*, setor encarregado de fazer a distribuição das células no próprio IP físico.

A totalidade do processo de desenvolvimento, entretanto, não possui um comportamento tão linear como o descrito, visto que muitas vezes as próprias especificações, como o número de bits das portas ou até mesmo a existência ou não de determinada porta, dos blocos que irão compor o IP ainda não foram determinadas de maneira definitiva. Com isso, tem-se versões de modelos e de *design* intermediárias que também passarão por uma verificação.

Essas mudanças podem ser desde a adaptação de algoritmo usado ou de interface. As mudanças de interface ocorrem principalmente na quantidade de bits das entradas e saídas dos blocos que compõem o IP, mas podem abranger também a alteração em nomes de sinais e adição ou retirada de sinais entrada/saídas de blocos. Para cada mudança na interface de algum bloco é necessário a mudança no ambiente de verificação para que o bloco e o ambiente sejam compatíveis e verificados de maneira homogênea.

Porém, tais mudanças demandam tempo dos engenheiros de verificação, e muitas vezes são mudanças mecânicas. Neste contexto e, considerando o fato que um ambiente de verificação possui estruturas básicas que demandam tempo para serem implementadas, é proposto e descrito neste trabalho a ideia e o algorítimo de um gerador, de modo que tanto as mudanças recorrentes no processo de desenvolvimento quanto a criação da estrutura comum sejam automatizadas, tornando assim o desenvolvimento mais eficiente e mais ágil.

O gerador proposto baseia-se de um modelo de *testbench* comum com algumas modificações, como, por exemplo, a implementação da possibilidade de utilizar-se de modelos em alto nível para realizar a verificação ciclo a ciclo, possibilitando uma melhor precisão para os verificadores. Assim, a aplicação proposta possui uma lógica de funcionamento simples, que é ilustrada na Figura 1.

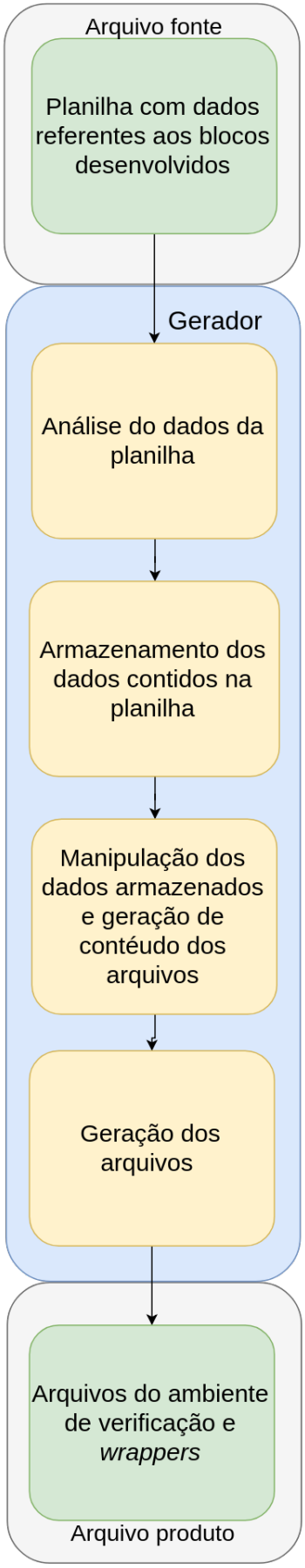

Figura 1 – Lógica de funcionamento do gerador

2 *SystemVerilog*

SystemVerilog, que foi padronizado como IEEE 1800, é um linguagem de descrição e de verificação de hardware usada para simular, testar e implementar sistemas eletrônicos. É comumente usada pela indústria de semicondutores e eletrônica como uma evolução do *Verilog*.

A sua padronização se deu com uma doação da linguagem Superlog para Accellera em 2002. As funções que dão suporte à parte de verificação foram baseadas na linguagem OpenVera doada pela Synopsys. Em 2005, *SystemVerilog* foi adotado como padrão IEEE 1800-2005, e apenas em 2009, o padrão recebeu como acréscimo a base padrão do *Verilog* (IEEE 1364-2005), criando assim o padrão IEEE 1800-2009.

SystemVerilog possibilita o uso de uma linguagem unificada para abstração e especificação de detalhes do design, especificação de *assertions*, cobertura e verificação baseada em metodologias manuais ou automáticas. Esta linguagem oferece também interface de programação de aplicativos (*Application Programming Interfaces - API*) para cobertura e *assertions*, uma API independente do fornecedor para acessar formatos de arquivos de forma de onda registrados, e uma interface de programação direta (*Direct Programming Interface - DPI*) para acessar as funcionalidades registradas.

Neste capitulo serão abordadas e explicadas algumas funções presentes na linguagem, principalmente as mais usadas para possibilitar a implementação dos componentes de um ambiente de verificação, com acréscimo de algumas funcionalidades que podem tornar-se úteis, e que foram implementadas no gerador.

2.1 Tipos de dados

Em *SystemVerilog* podemos ter a representação de quatro valores lógicos, são eles:

- **0**→ Representa o zero lógico, ou condição falsa;
- **1**→ Representa o um lógico, ou condição verdadeira;
- **X**→ Representa um valor lógico desconhecido;
- **Z**→ Representa o estado de alta impedância.

Na maioria dos casos o efeito de valores x e z será o mesmo. Para se declarar uma variável com os quatro estados utiliza-se a palavra-chave `logic`. Esse termo pode ser

usado para declarar objetos e para construir outros tipos de dados partindo de um tipo existente.

2.1.1 Inteiros

SystemVerilog provê um conjunto de dados que podem ser considerados inteiros, podendo dividido-los em inteiros de dois estados e de quatro estados, sendo eles:

- Dois estados: `shortint, int, longint, byte, bit`. Que possuem respectivamente, 16 bits inteiro com sinal, 32 bits inteiro com sinal, 64 bits inteiro com sinal, 8 bit inteiro com sinal ou caractere ASIC;
- Quatro estados: `logic, ref, integer, time`. Desses tipos, os únicos que já possuem tamanhos previamente definidos são `integer`, que possuem 32-bit inteiros com sinal e `time`, 64-bits inteiros com sinal. Nos demais, os tamanhos serão definidos pelo usuário, formando, assim, um vetor.

Alguns desses tipos de dados inteiros também podem ser declarados como "com sinal" (*signed*) e "sem sinal"(*unsigned*). Por exemplo, os dados tipos `byte, shortint, int, interger, e longint`, são tipicamente *signed*, assim como outros tipos tem seus padrões como *unsigned*. Para se declarar algum dos tipos inteiros como *signed* ou *unsigned*, basta utilizar as palavras chaves `signed` e `unsigned` após a declaração do tipo de dado.

```
1  //Inteiros: dois estados
2  signed   int      a;
3  unsigned shortint b;
4  signed   longint  c;
5  signed   byte     d;
6  signed   bit      e;
7
8  //Inteiros: quatro estatos
9  logic    f;
10 ref      g;
11 integer  h;
12 time     i;
```

Código 1 – Tipos de dados inteiros e suas sintaxes

2.1.2 Reais

O tipo `real` seria o equivalente ao `double` em C. Em comparação com os tipos inteiros, existem algumas diferenças. A primeira é a capacidade de representação numérica em ponto flutuante, o que conta como ponto positivo, porém, negativamente, alguns dos

outros artifícios da linguagem não podem ser utilizados como: controle de eventos de borda, seleção de bits, entre outros.

A conversão de um `real` para o tipo inteiro se dá por arredondamento para o valor inteiro mais próximo, ou seja, se a parte fracionada estiver abaixo de 0.5, ele arredondará para zero, e caso esteja igual ou maior, arredondará para cima.

2.1.3 Tipos definidos pelo usuário

Em *SystemVerilog* podem-se declarar tipos de dados de modo que eles venham a ser definidos pelo usuário usando a sintaxe `typedef`. De forma geral, o funcionamento se dá a partir da utilização de um tipo previamente existente, podendo esse ser uma classe ou um tipo de dado, e atribuindo um novo nome a esse.

```
1  class model;
2     typedef real data_r;
3
4     data_r data; // data will be from de type real
5  endclass
```

Código 2 – Sintaxe do uso de `typedef`

Quando se é utilizado `typedef` partindo de um tipo já existente, não é possível acessar os atributos internos deste, ou seja, não possuem referências hierárquicas com o tipo que foi utilizado. Tipos declarados, como interface, tem possível o acesso às portas, pois estes não são considerados uma referência hierárquica. Também é possível usar `typedef` para estruturas de dados como `struct, enum e union`. É importante destacar que a declaração deve ser feita no escopo (*scope*) do bloco.

2.1.4 Constante

Constante é um tipo de dado ou objeto definido no tempo de elaboração, ou seja, não pode ser alterado durante a simulação. Esse tipo pode ser usado para declarar tamanhos de portas em módulos, interfaces, classes, pacotes, etc.

Deve-se notar que o local no qual a constante é declarada afetará seu acesso, como por exemplo, se esta for declarada em uma região de escope (*scope*), o acesso poderá ser feito em qualquer módulo, classe etc, que esteja incluído na simulação. Porém, o mesmo raciocínio não é válido quando definido dentro de uma classe, visto que só terá permissão de acesso à constante as propriedades e os métodos da classe, o mesmo valendo para outros tipos de estruturas.

É importante, ainda, a diferenciação entre `const` e `parameter`. A primeira pode ser definida durante a simulação, ao contrário da segunda, que será definida durante a elaboração dos arquivos.

```
1  //parâmetros que serão globais;
2  parameter PARAM_um   =  'd1;
3  parameter param_dois =  'd5;
4
5  //classes podem ter seus próprios parâmetros sendo passados atras da sintaxe:
6  //#(...);
7  class example_param #(
8     parameter param_tres    =  'd12;
9     parameter param_quatro  = -'sd15; //essa sintaxe é utilizada para atribuir
10                                      //valores com sinal dentro do parâmetro
11 );
12       ...
13       ...
14 endclass
15
16
```

Código 3 – Sintaxe do uso de `parameter`

2.1.5 Estruturas *(structures)*

O tipo estrutura (`struct`) é uma coleção de tipos de dados que pode ser referenciados como um único tipo e seus membros podem ser acessados de forma hierárquica.

```
1  struct (
2     real        code;
3     bit [5:0];  b;
4  ) exemplo;
5
6  exemplo.code = 123;
7
8  typedef struct(
9     logic    valid;
10    int      count;
11 ) exemplo_1;
12
13 exemplo_1 control; // control é do tipo exemplo_1, que é uma struct
14                   //com dois atributos: valid e count.
```

Código 4 – Sintaxe do uso de `struct`

Estruturas de dados possuem o padrão *"unpacked"* tornando-as dependentes do *packing* do tipo de dados presentes dentro de sua declaração. Quando uma estrutura é declarada explicitamente como *packed*, os bits correspondentes são agrupados juntos na memória, sem espaços.

2.1.6 *Packed* e *unpacked* vetores

Para o tipo vetor, não existe nenhuma palavra-chave a ser usada em sua implementação, utilizando-se dos tipos mais básicos, ditos como não agregados e que foram mencionados anteriormente. Esses termos são designados apenas para referência de como os vetores são declarados, como, por exemplo, se suas dimensões são declaradas antes do nome da variável, isso a torna um vetor *packed*, e caso seja delcarada após, a torna *unpacked*.

A diferença entre tais métodos é a forma de como os dados serão salvos na memória, pois vetores *packed* são armazenadas de forma sequencial e, em contrapartida, os *unpacked* podem ou não ser salvos sem espaços entre seus elementos na memória.

Vetores declarados na forma *packed* têm como dimensão máxima 65536 bits e só podem ser utilizados para tipos de dados simples, como: `bit, logic e reg`. Por outro lado, os vetores *unpacked* podem ser de qualquer tipo e possuem a dimensão máxima de 16777216 elementos.

Uma variável do tipo vetor pode ter numa mesma declaração tamanho *packed* e *unpacked*. Quanto à velocidade de variação, esta segue o mesmo padrão de C, no qual a dimensão mais a direita varia mais rapidamente, salientando que as dimensões *packed* variam mais rápido que as *unpacked*.

Para acessar um elemento ou valor de um vetor, é suficiente a utilização do *index* do elemento; o mesmo sendo válido para as dimensões *packeds*, seguindo o mesmo critério de C, ou seja que a dimensão que varia mais rápido, sempre estará mais a direita, assim o acesso se dá a partir da inserção do *index* após todas as dimensões *unpacked*.

```
1  bit [7:0] clear;              //variavel packed
2  real data [255:0];            //variavel unpacked
3  real [10:0] values [16:0];    //variavel unpacked e packed
4
5  values[0]      = 1;           //atribui 1 para todos os valores packed, cados 0-10;
6  values[0][1]   = 2;           //atribui 2 para um único elemento
```

Código 5 – Sintaxe do uso de `packed e unpacked`

2.1.7 Vetores dinâmicos

Vetores dinâmicos não possuem um tamanho definido em sua declaração, ou seja, podem variar de tamanho durante a simulação. É importante ressaltar que não é possível a utilização desse artifício para vetores com dimensões exclusivamente *packeds*.

Para declarar os vetores de maneira dinâmica, é necessário apenas colocar colchetes vazios na dimensão mais a esquerda. O construtor utiliza-se do `new[]` para definir o

tamanho do vetor desejado. O vetor passa, ainda, a possuir dois métodos já existentes da própria linguagem: `size()`, que retorna o tamanho atual do vetor e, `delete()`, que deleta o vetor deixando-o com zero elementos.

2.2 Classes

SystemVerilog também possui suporte ao uso de classe. As variáveis contidas numa classe são chamadas de propriedades, e as `task` e `functions`, ou seja, sub-rotinas, são chamadas de métodos. A classe, assim como em C, comporta-se como um tipo de dado que pode ser instanciado, e esta instanciação é chamada de objeto.

Uma classe também pode conter os próprios parâmetros, ou seja, no momento da sua instanciação pode-se passar parâmetros que serão utilizados em suas propriedades.

O acesso das propriedades e dos métodos acontece da maneira como é demostrado no Código 6. Os construtores de classe em *SystemVerilog* utilizam-se de palavra-chave que torna um dos métodos, chamado de `new`, em construtor. Assim, enquanto é criada é chamada a função `new()`. Como o construtor se comporta como uma função, ele pode receber argumentos em sua chamada, como mostrado em Código 6.

```
1  class model ;
2     //propriedade da classe
3     bit [2:0] pos;
4     logic     valid;
5
6     //construtor da classe ou inicialização
7     function new();
8        pos = 3'd0;
9        valid = 1;
10    endfunction
11
12    //métodos
13    function void reset();
14       pos = 'd1;
15       valid = 0;
16    endfunction
17
18    function real cal (input real a, input real b);
19       cal = a*b;
20    endfunction : cal
21
22 endclass : model
```

Código 6 – Sintaxe do uso de `packed e unpacked`

Uma outra maneira de acessar as propriedade da classe, ao fazer uma implementação dentro da classe, é o uso da palavra-chave `this`, a qual evita possíveis ambiguidades no momento da implementação e simulação.

2.3 Processos

Em *SystemVerilog* os processos podem ser divididos em: procedimentos estruturados, declaração de bloco e controle de temporização.

O foco desta seção é percorrer, de forma geral, as funções de *SystemVerilog* que estarão presentes no código resultante do gerador, motivo pelo qual, só serão mencionadas as funções que foram utilizadas no modelo gerado.

Os procedimentos estruturados podem ser realizados através de: `initial`, `always`, `final`, *task* e *function*. `initial` é um procedimento que só será executado uma única vez, sendo utilizado, de forma geral, para inicialização de variáveis ou códigos que devem ser executados somente uma única vez. Sua sintaxe pode ser vista no Código 7. *task* e *function* serão explicados adiante.

```
1  module example();
2
3     initial begin
4        //códigos que serão executados apenas uma vez no inicio da simulação
5        //...
6     end
7  endmodule
```

Código 7 – Sintaxe do uso de `initial`

Declarações de blocos são uma maneira de agrupar declarações de modo que comportem-se como uma única declaração. Dessa forma, temos dois tipos: blocos sequenciais (`begin-end`) e blocos paralelo `fork-join`.

Em blocos sequenciais todas as declarações devem estar contidas entre `begin` e `end`, e serão executas pela ordem que foram implementadas, o mesmo sendo válido para atrasos colocados dentro do bloco.

Blocos paralelos são declarados atraves de `fork` e `join, join_any, join_none`. Todas as declarações internas serão executadas ao mesmo tempo, de forma que, nesse bloco, pode-se colocar um atraso para prover uma ordem temporal na sua execução. Este tipo não pode ser utilizado dentro de funções. Como mencionado anteriormente, existe mais de uma sintaxe para fechar o bloco, cada uma com sua devida peculiaridade:

- `join`: O processo do bloco só será finalizado após todos os processos lançados pelas declarações internas do bloco terem sido concluída;
- `join_any`: O processo do bloco será finalizado quando um dos processos lançados pelas declarações internas do bloco for concluído;

- `join_none`: Os processos do bloco serão executados em paralelo com o processo que iniciou o bloco (processo pai), e o bloco só iniciará seus processos internos após o processo pai ter realizado uma atribuição bloqueante.

Os tipos de blocos não são mutualmente excludentes, ou seja, dentro de um bloco paralelo podem conter blocos sequenciais e vice-versa, como pode ser visto em sua sintaxe no Código 8.

```
1  //Neste exemplo é possível ver um loop dentro de um loop
2  //para o primeiro fork ele irá ser concluido após qualquer
3  //um dos seus processos terminarem, neste caso tem-se só
4  //um processo. O segundo fork ele irá esperar que todos
5  //os seus subprocessos sejam finalizados.
6  fork
7     fork
8        $display("loop dentro de um loop");
9     join_any
10
11    fork
12       $display("loop dentro de um loop");
13    join
14 join
```

Código 8 – Sintaxe do uso de `fork-join`

A linguagem *SystemVerilog* fornece também estruturas que possam controlar os processos, como aguarda-lo terminar ou finaliza-lo enquanto esteja sendo executado. As declarações `wait, wait fork e wait_order` dão o controle para esperar que um processo seja encerrado.

A simples declaração `wait(expressão)` irá esperar pelo fim do processo "expressão". Por sua vez, `wait fork` irá esperar que todos os processos, sem levar em consideração os subprocessos iniciado pelo `fork`, sejam concluídos.

2.3.1 Condicionais

Estruturas condicionais em *SystemVerilog* são semelhantes às existentes em C, ou seja, elas são feitas através de estruturas de `if-else`, com as mesmas capacidade de intercalação e sequências, como pode ser visto no Código 9.

2.3.2 *Case*

Há, ainda, em *SystemVerilog*, as estruturas de *case*, as quais são uma estrutura que se baseia na ideia do `switch-case` de C, que compara quando uma expressão é igual a outra, realizando assim, uma operação desejada, como é mostrado no Código 10.

```
1  //Exemplo de estrutura de condição em sequência
2  if (idx > 0) begin
3     if (val < 0) begin
4        result = idx+val;
5     end else begin
6        result = idx-val;
7     end
8  end else if(idx == 0) begin
9     result = result + idx;
10 end
```

Código 9 – Sintaxe do uso de `if-else`

```
1  logic [6:0] pos;
2  logic [15:0] values;
3
4  case (values)
5     16'd0: result = 7'd1;
6     16'd1: result = 7'd2;
7     16'd2: result = 7'd3;
8     16'd3: result = 7'd4;
9     16'd4: result = 7'd1;
10    16'd5: result = 7'd1;
11    16'd6: result = 7'd1;
12    16'd7: result = 7'd1;
13    16'd8: result = 7'd1;
14    16'd9: result = 7'd1;
15    default result = 'x;
16 endcase
```

Código 10 – Sintaxe do uso de `case`

2.3.3 *Loops*

Loops, ou estruturas de repetição, também são suportados por *SystemVerilog*. Estas estruturas são: `for-loop`, `forever`, `repeat`, entre outras, porém, somente as mencionadas serão detalhadas no decorrer do trabalho.

Cada uma dessas estruturas tem sua peculiaridade, o que define, caso a caso, qual delas é a melhor opção. A estrutura `for-loop` é a única que contém em sua declaração uma inicialização de variável, a qual servirá para a condição de parada. O `forever`, por sua vez, não tem condição de parada, ou seja, é utilizada principalmente em situações nas quais se deseja *loops* infinitos. É importante salientar que sua utilização deve ser acompanha de algum controle de tempo ou outra função que consiga desabilitar o *loop*, para que assim evite repetições infinitas com zero de atraso, o que resultaria no travamento do escalonador da simulação com aquele processo.

A estrutura `repeat` tem a peculiaridade de receber o número de vezes que a sua declaração terá que ser executada, sendo, dessa forma, útil em situação na qual já sabe

quantas vezes será necessária a repetição. Um exemplo do uso dessas estruturas pode ser encontrado no Código 11.

```
1  //exemplo da estrutura de for-loop
2  for (int i = 0; i < MAX; i++) begin
3     //codigo que deverá ser executado MAX vezes
4     //..
5  end
6
7
8  //exemplo da estrutura de forever
9  forever begin
10    //código que deverá ser repetido infinitas vezes
11    //..
12 end
13
14 //exemplo da estrutura repeat
15 repeat(10) @(posedge clk) //a thread que esteja sendo executa esperará por 10
16                           // eventos de clk, para assim de o procedimento na
17                           // na execução do código
```

Código 11 – Sintaxe do uso das estruturas de *loops*

2.3.4 `task` e `function`

`task` e `function` são estruturas que também podem ser chamadas de sub-rotinas, permitindo a execução de procedimentos comuns em diferentes lugares do código. Ainda, realizam a função de "modularização" do código, fazendo com que grandes processos possam ser divididos em processos menores, possibilitando que o *debug* seja realizado mais facilmente.

As principais diferenças entre `task` e `function` são:

- As declarações contidas dento de `function` serão executadas em uma unidade de tempo de simulação, já a `task`, pode conter uma declaração de controle tempo;
- Uma `function` não pode iniciar uma `task`, porém uma `task` pode inciar outras `task` e também `function`;
- Uma `task` não possui retorno, ou seja, sempre será "`void`", ao contrário de `function`, que pode ou não ter retorno.

O padrão no momento de declaração de uma `task` é que ela será do tipo `static`, sendo assim, todos os seus valores sendo declarados de maneira estática. Esta tarefa também pode ser declarada de maneira que venha a ter um armazenamento automático, utilizando-se da palavra-chave `automatic` em sua declaração ou declarando-a dentro de

estruturas que são, por padrão, `automatic`. A seu turno, `task` que não são estáticas possuem suas variáveis internas e suas declarações alocadas dinamicamente. Quando declaradas como `automatic` não se pode acessar atributos internos de maneira hierárquica.

O uso de `function` não deve conter nenhuma declaração de controle de tempo, uma vez que não deve fazer uso de chamada de `task`, a menos que esta `task` possua alguma declaração de controlador de tempo.

```
1  //exemplo de função com retorno do tipo
2  //void
3  function func_01 (input logic a, output int b);
4     ...
5  endfunction
6
7  //exemplo de função com retorno
8  function logic [5:0] func_02 (input logic a, output logic b);
9     ...
10    return c
11 endfunction
12
13 //exemplo de determinar o retorno da função
14 //utilizando o nome da função
15 function logic [5:0] func_03 (input logic a, output logic b);
16    ...
17    func_03 = 2 * a + b;
18 endfunction
```

Código 12 – Sintaxe do uso de `function`

Como pode ser visto no Código 12, a declaração pode incluir o tipo de dados de retorno ou ter uma declaração implícita, que será considerada do tipo *void*. A declaração de uma `task` se dá de maneira semelhante, vide Código 13.

```
1  //Exemplo de declaração de uma task. É importante mencionar
2  //que a task pode ter algum mecanismo de controle de tempo,
3  //sendo assim possivel a declara de um @(posedge ...)
4  task my_task (input logic a, input logic b);
5     ...
6     ...
7     b = a;
8  endtask
```

Código 13 – Sintaxe do uso de `task`

Os argumentos para essas estruturas podem ser passados por valor, que é o modo mais comum de passar argumentos para sub-rotinas. Este consiste em passar um objeto para a função. Uma outra maneira é a passagem por referência, na qual diferente da anterior, que copia o valor passado para o escopo da função, aquele modo acessa diretamente o argumento original. Para tanto, deve-se usar a palavra-chave `ref`.

A passagem por referência tem condições para ser utilizada: os tipos e tamanhos que foram declarados tem que ser os mesmos que estão sendo utilizados pelo argumento passado, e não podem ser do tipo `static`. Assim, os únicos elementos que podem utilizar-se desse artificio são: variável, uma propriedade de uma classe, membro de uma estrutura *unpacked* ou um elemento de um vetor *unpacked*. Importante notar que o uso de passagem por referência deve ocorrer apenas em variáveis *automatic*.

As `function` e as `task` podem possuir em sua declarações valores padrão para seus argumentos, ou seja, caso não seja atribuído nenhum valor a algum deles, esses terão o valor utilizado na implementação. Um outro artifício que pode ser usado no momento da sua instanciação é a atribuição dos argumentos usando os seus próprios nomes, evitando, assim, uma possível inversão na ordem de instanciação, como demostrado no código 14.

```
1  //exemplo de função com valores padrões de arugmentos
2  task write(int valid = 0, int data, int o_ready=0);
3     ...
4  endtask
5
6  //exemplo de instanciação utilizando os nomes dos argumentos
7  //e não a ordem
8  write writa_inst(
9     .valid   (1),
10    .data    (13),
11    .o_ready ()
12 )
```

Código 14 – Sintaxe do uso de `task` para argumentos com valores padrões e pasagem de argumento com o nome

2.4 *Clocking block*

Como será mencionado mais adiante, existem estruturas que podem ser utilizadas para conectar as portas e especificar sinais pelos quais o *testbench* irá se comunicar com o dispositivo que está sendo testado. Porém, no tocante ao controle de tempo, através de um *clock*(em tradução literal: relógio), o *clocking block* é bastante eficiente quando comparado a outras estruturas.

Visando ter esse controle de sincronização, tem-se a estrutura de um *clocking block*, que é declarado entre as palavras-chaves `clocking-endclocking`, fazendo com que os *clocking blocks* se tornem uma chave importante em metodologias que baseiam-se em ciclos, dando a possibilidade dos testes tenham noção de ciclo e de transação.

Clocking blocks separam detalhes de tempo e de sincronização das informações estruturais, funcionais e elementos procedurais de um *testbench*. Com isso, é possível realizar as operações de eventos síncronos, amostragem de sinais e *drives* síncronos.

Em suma, com essa estrutura é possível determinar o momento em que deverá ser feita a leitura de uma sinal e/ou o momento que deverá ser atualizados os sinais com novos valores. Desta forma, um dos seus principais usos é evitar que ocorra uma condição de corrida no seu *testbench*. A sua estrutura é simples, porém é importante mencionar que existem outras opções além das vistas no Código 15, que podem ser encontradas no manual de *Systemverilog*.

```
1  //exemplo de clocking block. clk é o sinal que o clocking block deverá usar
2  //como base para realizar as mudanças do sinal. Os sinais podem ser declarados
3  //como inout, input e ouput.
4  clocking cb @(posedge clk);
5     default input #1step output #1step //os valores de #1step referen-se ao
6                                        //atraso ou adiantamento dos sinais
7     //declaração das portas da interface
8     inout i_valid;
9     inout i_data;
10    ...
11    inout o_valid;
12    inout o_valid;
13 endclocking :cb
```

Código 15 – Sintaxe do uso de *clocking block*

2.4.1 Sincronização e comunicação: *mailboxes*

Quando há necessidade de fazer uma comunicação entre os processos, pode-se utilizar uma classe já existente de *Systemverilog*, `mailbox`. Essa classe funciona como uma caixa de correios, na qual alguém (processo) pode colocar algo dentro e outra pessoa (outro processo) pode retirar o que foi colocado. Algo importante que torna-se útil para sincronização entre os processos é o fato que quando a *mailbox* está cheia, o processo que tentar colocar algo dentro irá entrar em estado de suspensão, até que outro processo venha e retire um elemento de dentro da *mailbox*, realizando, com isso, a sincronização entre dois processos.

A classe `mailbox`, por ser uma classe própria de *Systemverilog*, possui os seguintes métodos:

- Construtor: `new()`;
- Inserir item dentro da *mailbox*: `put()`;
- Tentar inserir um item dentro da *mailbox* sem bloquear o processo: `try_put()`;
- Retirar um item: `put()` ou `peek()`;
- Tentar retirar um item sem bloquear o processo `try_put()` ou `try_peek()`;

- Retornar a quantidade de itens dentro da *mailbox*: `num()`.

```
1  program test();
2     mailbox t2d; //para a declaração de uma mailbox de um tipo especifico
3                  //usar a seguinte sintaxe:
4                  //         mailbox #(<tipo_de_dado_definido>) t2d;
5
6     initial begin
7        drive = new(t2d);
8
9        int a = 23;
10
11       t2d = new(1); //pode-se incluir ou não o número de espaços que terá a
12                     //mailbox, neste caso ela terá um único espaço. Caso deseje
13                     //um numero infinito de espaços não deve ser inserido nenhum
14                     // número
15
16       t2d.put(a);
17    end
18
19 endprogram
20
21
22 class drive;
23    mailbox  t2d;
24    int      a;
25
26    function new(mailbox in_t2d);
27       this.t2d = in_t2d;
28    endfunction
29
30    task run();
31       t2d.get(a);
32    endtask
33 endclass
```

Código 16 – Sintaxe do uso de *mailbox*

2.5 Construtores hierárquicos

Em *SystemVerilog* existem diferentes construtores hierárquicos, ou estruturas, que podem ser acessados de maneira hierárquica ou são utilizados para encapsular dados, funcionalidades, temporização etc. Alguns deles são: módulos (`module`), programas (`programs`), interface (`interface`) e pacotes (`packages`). Cada uma dessas estruturas têm suas peculiaridades, que serão explanadas mais adiante.

2.5.1 Módulos

Módulos (`module`) são as estruturas básicas em *SystemVerilog* para um *design*. Possuem como principal objetivo encapsular dados, funcionalidades e temporização de

objetos digitais, ou seja, os componentes de baixo nível, sejam portas lógicas simples ou complexas. De forma geral, existem duas maneiras de fazer a declaração de um módulo, ANSI e não-ANSI, ambas maneiras são demostradas no Código 17.

```
1  //declaração com o padrão não-ANSI
2
3  module example(a,b,c,d,e);
4     input    [3:0] a;
5     input    [5:0] b;
6     input    [10:0] c;
7     output   [8:0] d;
8     output   [15:0] e;
9     //...
10
11 endmodule
12
13 //declaração com o padrão ANSI
14
15 module example(
16    input    [3:0] a,
17    input    [5:0] b,
18    input    [10:0] c,,;
19    output   [8:0] d,
20    output   [15:0] e
21 );
22
23    //...
24 endmodule
```

Código 17 – Exemplo do uso dos padrões ANSI e não-ANSI

2.5.2 Programas

Enquanto o foco de módulos é a descrição do hardware propriamente dito, os programas (`programs`) são voltados ao suporte no *testbench*, pois para isto não é necessário o foco em nível de hardware, não sendo relevantes fios, estruturas hierárquicas etc. Assim, os blocos chamados de programas (`programs`) possuem os seguintes objetivos:

- Proveem um ponto de entrada para o execução dos `testbenchs`;
- Criam um escopo para encapsular dados, *task* e funções;

Junto com os *clocking blocks*, os programas proveem interações livres de condições de corrida entre o design e o *testbench* e ainda possibilitam abstrações de ciclos e transações. Um exemplo da implementação de um programa é mostrado no Código 18.

```
1  //exemplo de uso da estrutura de program
2  program test #(
3     parameter MAX = 'd10,
4     //..
5     parameter SIZE = 'd5
6  )(
7     input clk,
8     input [15:0] dado,
9     //..
10    int   count,
11 );
12
13    //aqui pode ser chamada os construtores do ambiente (environment) do
14    //testbench
15    initial begin
16       //..
17    end
18 endprogram
```

Código 18 – Exemplo de uso de `program`

2.5.3 Interface

A sua principal função é a comunicação entre blocos, abrangendo tanto alto nível quanto baixo nível. Uma interface também pode conter parâmetros, constantes, funções e *tasks*, ou seja, essa também pode comportar-se como uma classe e consegue realizar a comunicação entre blocos de maneira suave. Um exemplo de sua implementação e uso pode ser visto no Código 19

Uma interface pode ainda ser declarada como virtual, fazendo com que se possa manipular os sinais da interface sem referenciar os sinais atuais. Estes tipos de interfaces também podem ser passados como argumentos de *tasks*, funções ou métodos. A declaração de uma interface como virtual é feita como é mostrado no Código 20

É importante salientar que uma interface virtual não pode ser usada como porta e nem em `assigns`, ou lista de sensibilidade. Para que sua utilização seja possível nestes cenários, a interface deve conter ou um *clocking block* ou um *driver* que esteja ligado a uma interface que não seja virtual.

2.5.4 Pacotes

Um outro mecanismo para compartilhar parâmetros, dados, *tasks* e funções entre módulos, interfaces e programas, são os pacotes (ou `packages`). A declaração de um `package` cria um escopo que contém as declarações que serão compartilhadas. Como pode ser visto no Código 21.

```
interface block_if (
   input logic clk

);
   //declaração dos sinais
   wire i_valid;
   wire i_data;
   //...
   wire o_valid;
   wire o_data;

   //...
endinterface

module top():
   block_if if_b; //declaração de uma interface para o block

   block a_block(
      .i_valid(if_b.i_valid),
      .i_data(if_b.i_data),
      //...
      .o_valid(if_b.o_valid),
      .o_data(if_b.o_data)
   )
endmodule
```

Código 19 – Exemplo de uso de `interface`

```
virtual blcok_if vif_b;
```

Código 20 – Declaração de um interface como virtual

```
//exemplo de implementação de um package
package example;
   class util_1;
      int a;
      real b;

      function new(input int in_a, input real in_b);
         this.a = in_a;
         this.b = in_b;
      endfunction

      function sum(input int c, output real d);
         d = a + b + c;
      endfunction
   endclass
endpackage : example
```

Código 21 – Declaração de um pacote

3 Comunicação entre *SystemVerilog* e Python

O gerador foi desenvolvido partindo do princípio que o modelo de referência estaria em Python e poderia ser comparado com o DUT ciclo a ciclo. Para que isso fosse possível, foram desenvolvidas cascas (*wrappers*) que fariam as traduções de uma linguagem para outra.

Como ponte da linguagem de alto nível para a linguagem de baixo nível foi utilizado C. Assim, foi possível utilizar a DPI-C (*Direct Programing Interface - C*) que realiza a comunicação entre C e *SystemVerilog*, e é uma ferramente desta linguagem. Para a comunicação entre Python e C foi utilizada a API (*Application Programming Interface*) de Python para C, juntamente com a API da biblioteca *Numpy* de Python para C.

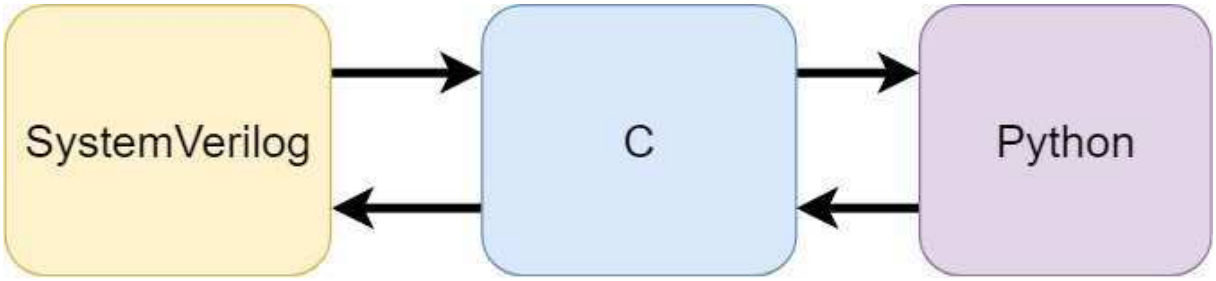

Figura 2 – Fluxo do transição de dados

Assim, esse tópico irá tratar das funções que possibilitaram essa comunicação entre *SystemVerilog* e uma linguagem de alto nível, através de um código exemplo, que pode ser encontrado no Anexo 8.1.2.

Como pode ser observado no Anexo 8.1.1, pode-se enviar qualquer tipo de dados de *SystemVerilog* para C. A maioria desses tipos podem ser enviados através de uma `struct`, e os tipos que não são passíveis de envio neste molde, utilizam funções próprias existentes na biblioteca da DPI (como o caso de um vetor dinâmico). Em suma, em *SystemVerilog*, torna-se simples o uso de uma linguagem "estrangeira", sem o uso de nenhuma função especial, com exceção do `import`.

A parte que deve-se ter uma maior atenção é o código que encontra-se em C. Nesse, estão contidas algumas funções necessárias para traduzir os tipos de dados de *SystemVerilog* para C, e as funções da API-C de Python.

As funções da API-C de Python possuem uma complexidade maior quando comparadas com as funções da DPI-C de *SystemVerilog*. Visando uma maior praticidade na explanação das funções da API-C, as funções que foram utilizadas serão explicadas sucin-

tamente através de tópicos, e sua demonstração de uso estará no Anexo 8.1.2 (IEEE. . . ,)(FOUNDATION, 2018).

- **Py_Initialize()**: Função que irá inicializar o interpretador de Python. Caso algum interpretador já tiver sido inicializado, ao chamar esta função pela segunda vez, não ocorrerá nada;
- **PyObject**: É um tipo que contém todas as informações necessárias para que o C trate um ponteiro de um ponteiro de um objeto da biblioteca como um objeto de Python. É o tipo básico de todos os objetos da API.
- **PyObject_CallOject(PyOject *obj, NULL)**: Chama um objeto. O objeto que puder ser chamado, chamará, consequentemente, seu construtor;
- **Py_BuildValue(const char *format, value)**: De forma geral, essa função serve para criar um objeto `PyObject` com um valor de um tipo de dado conhecido (`int`, `float`, `string`, etc);
- **PyObject_CallMethodObjArgs(PyObject *obj, PyObject *method, PyObject *arg1, PyObject *arg2, NULL)**: Chama um método de um objeto previamente inicializado e com a referência em *obj* e seus argumentos;
- **PyTuple_GetItem(PyObject *tuple_obj, int pos)**: Retorna o elemento na posição *pos* da tupla *tuple_obj*. O retorno se dá através de um `PyObject`;
- **PyList_New(int/Py_ssize_t dim)**: Retorna uma nova lista em caso de sucesso. Esta lista deverá ser armazenada em um ponteiro do tipo PyObject;
- **PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)**: Coloca o valor que está em *item* dentro de *list* na posição *index*;
- **PyTuple_Pack(Py_ssize_t n, PyObject *item1, PyObject *item2, ...)**: Retorna uma nova tupla de tamanho *n*, inicializada com *item1, item2, ...* O retorno deve ser armazenado dentro de um PyObject.

Foram explicadas algumas funções utilizadas para a confecção dos *wrappers*. Caso o leitor deseje adquirir um conhecimento mais profundo sobre o assunto, deve consultar a documentação de referência externa. Um código exemplo encontra-se em Anexo 8.1.3 e faz parte da comunicação com o *SystemVerilog*, juntamente com o código em Python, demonstrando assim, a conexão completa dos *wrappers*.

4 Ambiente de verificação

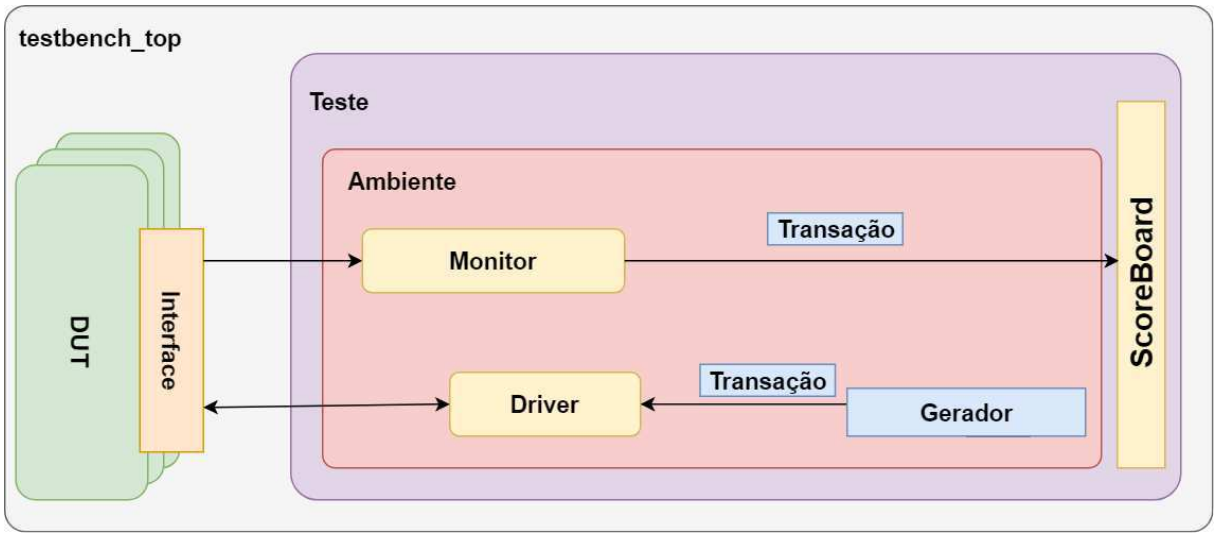

Figura 3 – Exemplo de uma topologia de *testbench*

O ambiente de verificação possui uma estrutura padrão que visa manter o dispositivo sobre teste DTU (*Device Under Test*) sobre constante monitoramento. De forma geral, pode-se dividir nos seguintes componentes: *driver*, monitor, interface, transação, ambiente, gerador, *checker/scoreboard*, teste e topo. É importante destacar que alguns desses componentes encontram-se contidos dentro de outros, como é o caso do *driver* e monitor que estão contidos dentro do ambiente.

Cada um dos componentes possui uma função específica. O *driver* tem como função receber os estímulos (transação) advindas de um gerador de dados e atribuir tais valores de entradas para o DUT e para o modelo de referência; o monitor é responsável por monitorar os sinais de saída de ambos, DUT e modelo de referência, através da interface para que assim seja possível obter uma comparação entre eles. A referida comparação é feita pelo componente chamado de *checker*, no qual basicamente deverá pegar as saídas e compará-las, incrementando um contador de erro ou emitindo um alerta quando uma discrepância entre os sinais comparados existir ou algum sinal adquirir um valor ilegal. A transação define a atividade no nível de pino, ou seja, se comporta como um pacote de dados para que o *driver* tenha as informações necessárias para realizar sua atividade, também podendo ser utilizado para armazenar valores de saída e transacioná-los para o monitor.

A transação recebe os dados que serão utilizados no *driver*, advindo de outro componente, o gerador. Este componente tem a função de produzir estímulos, partindo de princípios que tornem esses dados válidos e de maneiras pseudoaleatórias.

Os componentes anteriormente mencionados podem ser agrupados como agente (*agent*), ambiente (*environment*) e teste. O grupo/componente agente é composto por componentes menores que são específicos a uma interface ou a um protocolo, no caso: gerador, *driver* e monitor. Já o ambiente, conterá os componentes de alto nível, como os agentes e o *scoreboard*. O teste, que também pode ser chamado de *program test*, é responsável por configurar o *testbench*, iniciar os processos de construtores dos componentes do *testbench* e iniciar os estímulos. O último componente é o que conecta todos os outros, *testbench top*, ou topo do *testbench*. Esse é o o arquivo mais externo, que conecta o DUT ao *testbench*, consistindo em instâncias do DUT, *test* e interface, na qual a instância da interface conecta o DUT ao *testbench*.

Todos esses componentes, com exceção do teste e do topo do *testbench*, são implementados como classes, comportando-se, assim, como parte de alto nível do *testbench*. O teste é construído usando um tipo de bloco chamado *program*, o qual é suportado pela linguagem *SystemVerilog*, como já foi mencionado anteriormente. O propósito segundo o manual de referência da linguagem (IEEE. . . ,): é prover um ponto de entrada para execução dos *testbenchs* e criar um escopo que encapsule todo os dados, *tasks*, e funções do *program*.

O topo do *testbench* é implementado com um nível mais baixo de abstração, utilizando o `module`, que neste caso é utilizado para ter a possibilidade de se conectar com os blocos ou componentes de verificação, com os blocos de design, ou DUT.

5 *Testbench* modelo

O *TestBench* gerado segue um modelo que é de acesso público em (SYSTEMVERILOG..., 2016), o qual foi escolhido por utilizar-se de uma ambiente completo com praticamente todos os componentes de um *testbench*.

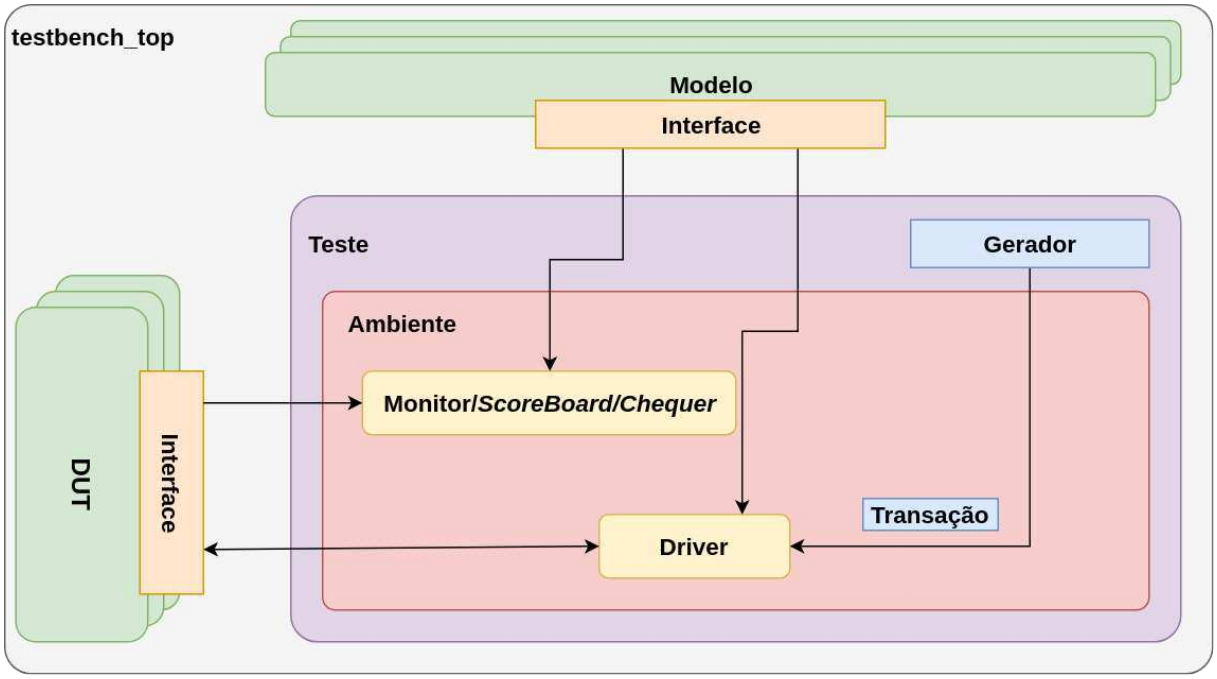

Figura 4 – Topologia de *testbench* utilizada no gerador

Este ambiente possui uma organização que torna o arquivo `testbench.sv`(vide 8.2.1) o arquivo de topo. Este arquivo irá conectar o DUT ao ambiente de verificação, utilizando-se, para tanto, de um interface virtual, visando a possibilidade de manipulação dos valores dos sinais sem que estes sejam alterados instantaneamente no DUT.

Neste arquivo, por ser o arquivo de topo, irá conter a geração do *clock*. No exemplo encontrado, ele gera um *reset* que será utilizado apenas no início da simulação. Visando testes que exercitem mais o DUT, o ambiente gerado terá seus *reset* ativados por controle do teste, assim podendo ser ativado quando o teste desejar.

Uma outra adaptação em relação ao arquivo é que, no exemplo, não existe um modelo de referência propriamente dito. Assim, para aumentar as possibilidades de testes quanto à capacidade do ambiente gerado e poder ter um leque maior de modelos suportados, ou seja, o modelo não se limitar a um modelo que seja implementado em *SystemVerilog* e ser possível comparar algoritmos complexos que são possíveis de ser implementados em linguagens de alto nível, criou-se *wrappers*. Estes são capazes de realizar a comunicação de um modelo de referência em Python (linguagem de alto nível) com o

ambiente de verificação. Com os *wrappers*, o modelo de referência passa a ser um modelo instanciado no topo, que terá sua própria interface.

Assim, neste arquivo são instanciados todos os DUT e todos os modelos de referências necessários para realizar os testes, juntamente com suas respectivas interfaces.

A instanciação das interfaces acontecem no `testbench.sv`, mas sua declaração acontece dentro do arquivo `interface.sv`. A declaração da interface do bloco é feita da maneira ANSI, recebendo os parâmetros do bloco e os sinais de *reset* e do *clock* como entradas para a interface. Seguindo o padrão, a interface possui sua declaração de parâmetros, seus dois sinais e, internamente, a declaração dos sinais ao bloco que ela corresponde. Atentando ao detalhe que os sinais aqui foram declarados como *wire*, visando a possibilidade do uso de `inout`, pois julgou-se melhor a implementação de um único *clocking block* em que este tem os sinais declarados de maneira `inout` e sem a declaração de um `modport`. Fazendo assim, com que um único bloco comporte-se como se tivesse a utilidade para o *driver* e para o monitor, deixando, assim, a necessidade da instanciação de dois `modport`, um para cada componente.

Dessa forma, a estrutura final da interface, que pode ser vista no Código 8.2.2, ficou apenas com a declaração dos sinais e de um *clocking block*.

A próxima instanciação no arquivo `testbench.sv` é a instanciação do teste. O arquivo de declaração de teste não pode ser gerado completamente, pois, em razão de sua característica peculiar para cada bloco, torna-se difícil generaliza-lo. Assim, pensou-se em montar uma estrutura básica, só para conter objetos e declarações que tornam-se obrigatórios devido ao resto dos componentes do *testbench*.

Com isso, o *stub*(vide 8.2.3) do teste acabou sendo composto por um bloco de `initial`, no qual tem-se a instanciação do ambiente (*environment*) sendo passado para o seu construtor todas as interfaces necessárias e um bloco de `fork`, no qual mantém-se a repetição do ambiente, até que este detecte a condição para terminar.

O construtor do ambiente contém a instanciação de outros componentes, como o monitor e o drive. O construtor do drive irá receber a interface do DUT (que servirá como meio de propagação dos valores do drive para as outras interfaces) e o monitor receberá as interfaces virtuais do DUT e o do modelo. Para manter o sincronismo entre o teste o ambiente, utiliza-se uma variável declarada como `mailbox`, que é instanciada no teste e passada através do construtor do ambiente para o drive, assim, quando o teste colocar uma transação ele só irá colocar a próxima depois que o drive tirar o item já existente. Além do seu construtor recebendo parâmetros, tem-se a declaração de uma `task` chamada de `run` para ser invocada pelo teste e iniciar os processos do monitor e do drive e o comando para finalizar a simulação, `$finish`. Os processos iniciados por essa `task` estão dentro de um `fork-join_any`, juntamente com o processo de outra *task*, responsável por checar se

a condição de término do teste foi atingida, no caso de as quantidades de ciclos desejada terem sido atingida no monitor.

Para dar suporte a essas necessidades, no ambiente também são declarados alguns tipos através do uso de `typedef`(vide 8.2.4), como os tipos das interfaces virtuais dos blocos para que assim seja possível o construtor receber tais informações.

A declaração do *driver* é feita no arquivo `nome_block_drive.sv`(vide 8.2.5). O seu construtor recebe a interface do DUT e o `mailbox`, assim como a quantidade de ciclos que o drive terá que operar. A classe `drive` foi implementada com o construtor e uma *task* drive, que tem com principal funcionalidade retirar o item do `mailbox` e realizar a atribuição dos sinais da interface do DUT, tornando-a uma classe simples.

A classe monitor nesse modelo terá duas funções extras (vide 8.2.6): checar as saídas e contabilizar os erros. Ainda, tem como entradas do seu construtor as interfaces virtuais do modelo de referência e do DUT. A classe possui um único método monitor que irá acessar os dados advindos das interfaces virtuais e compará-los, e, caso estes dados divirjam, deverá incrementar um valor que servirá como *flag* para detectar se ocorreu algum erro entre o DUT e modelo.

Todos os dados são armazenados em um objeto de uma classe chamada *transaction* (em tradução livre: transação), a qual contem propriedades variáveis equivalentes aos sinais que o drive é responsável por atribuir valores, pode ser observado seu modelo no Anexo 8.2.7.

O seu construtor não receberá nenhum parâmetro, mas a classe terá método de *init*, para inicializar valores internos e `randomize_static`, para randomizar variáveis/sinais. É importante salientar que a parte de `constraints` deve ser implementada pelo usuário do código, ou seja, não será gerado com o gerador.

6 Gerador de *TestBench*

Visando a praticidade e eficiência no decorrer de um projeto de um IP, no qual se requer uma verificação a cada mudança, seja ela de interface, na quantidade de bits internos para algoritmos de convergência ou em sua arquitetura, pensou-se na implementação de gerador para o ambiente de verificação, fazendo com que o engenheiro tenha seu tempo melhor aproveitado.

Para que a utilização do gerador implementado seja possível, utiliza-se uma planilha na qual deve conter informações cruciais quanto aos sinais do bloco que se deseja verificar, como: nome do bloco; nome dos sinais; quantidade de bits de cada sinal; a quantidade daqueles bits que serão inteiros e se a porta terá bit para a representação do sinal ou não, para o uso em conjunto com um modelo de referência em Python. Esta última informação é necessária devido ao fato de que o modelo em Python deverá operar em *floating point*, tornando necessário que o valor no sinal seja convertido de maneira correta através de aritmética de ponto fixo.

Assim, o funcionamento do algoritmo irá percorrer a planilha e armazenar as informações em variáveis das classes do gerador. Tendo assim, fluxo de dados e resultados como mostrados na Figura 5.

O fluxograma possui o seguintes elementos:

a) Arquivo em formato de planilha onde estão contidas as informações dos blocos e suas interfaces;

b) Conjuntos de objetos que contém as informações contidas nas abas do arquivo fonte;

c) Conjunto de objetos que contém as informações contidas nas linhas de cada aba do arquivo fonte;

d) Conjunto de objetos que contém as informações contidas em cada células das linhas que foram armazenadas;

e) Objetos inicializados com as informações completas adquiridas da planilha, como, por exemplo, nome do bloco, tamanho do sinal, direção do sinal etc;

f) *Strings* construídas partindo da análise dos dados obtidos no processo anterior;

Cada classe possui métodos que servem para armazenar e manipular as informações para que seja possível a impressão dos arquivos de *testbench*. Seguindo o fluxo de dados no gerador um objeto deve ser criado, inicialmente, da classe `SpreadsheetReader`, a

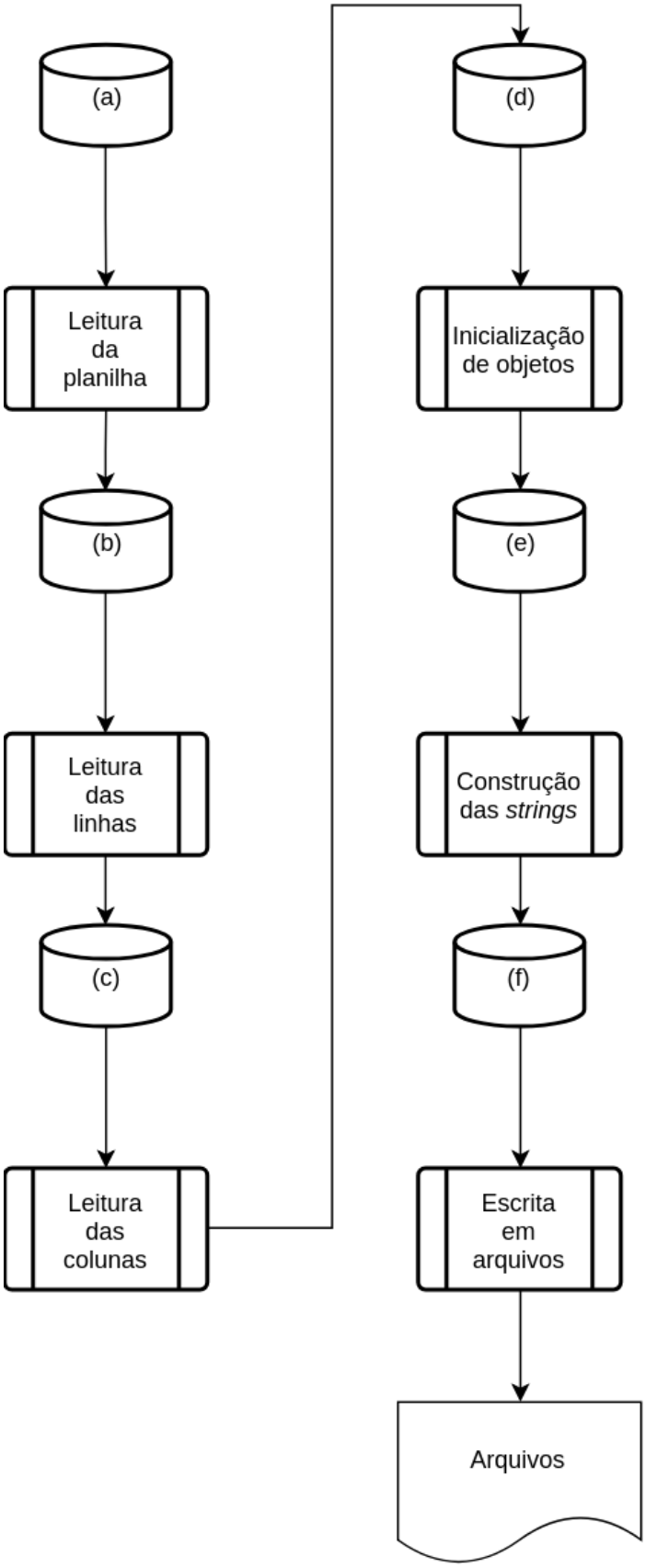

Figura 5 – Fluxograma dos dados no gerador

qual possui os métodos e os atributos mostrados na Tabela 1. O construtor desta classe recebe o caminho para onde se encontra a planilha que deverá ser utilizada para obter as informações necessárias.

SpreadsheetReader	
Atributos	Métodos
mPath mInputStream mWorkbook maBlock	SpreadsheetReader(string) parseWorkbook(int) getBlocks

Tabela 1 – Atributos e métodos da classe *SpreadsheetReader*

A classe é composta por um atributo que armazenará o caminho para a planilha juntamente com o nome da planilha em um objeto do tipo `InputStream`, que faz parte de um pacote de Java para entrada e saída de dados. Assim, esse objeto receberá os *bytes* referentes ao arquivo da planilha no sistema. A classe ainda contém um atributo do tipo `XSSFWorkbook`, que possui métodos para manipular a planilha, sendo, este tipo de objeto, próprio de bibliotecas de Java. O último atributo é de um tipo de outra classe implementada para o gerador, o qual também é `ArrayList`. Este atributo referenciará a aba da planilha (útil para caso em uma única planilha encontram-se vários blocos declarados em diferentes abas), em que cada elemento do vetor é uma aba da planilha.

O método da classe `parseWorkbook` visa à atribuição dos valores na planilha para objetos, mais especificamente do tipo `ParsedBlock`, ou seja, ele "adiciona" as abas das planilhas nas posições do vetor `maBlock`. A classe ainda tem um outro método, `getBlocks`, que tem como funcionalidade o retorno do vetor, que possui as informações contidas nas abas em cada elemento.

Para adicionar os elementos ao vetor é necessário chamar o construtor da classe `ParsedBlock`, que possui atributos e métodos mostrados na Tabela 2. O construtor recebe um objeto do tipo `XSSFSheet` e irá analisar as linhas de cada aba e adicionar em um vetor do tipo `ParsedSignal`, que servirá como objeto, contendo informações das linhas nas quais devem estar os sinais e seus dados. Para que seja possível ter esse acesso, se utiliza o tipo `XSSFRow`, que faz parte da biblioteca que trata a interação de Java com arquivos `.xlsx`. Com isso, o construtor da classe `ParsedSignal` receberá um objeto que conterá informações de uma das linhas da planilha e acessará as informações contidas nas células das linhas armazenando o conteúdo em seus atributos.

A classe `ParsedBlock` possui mais um método, `toBlock`, que converterá as informações do objeto `ParsedBlock` em `Block`. O `toBlock` chama um dos métodos da classe `ParsedSignal`, que tem como funcionalidade converter os sinais, no caso, os elementos que estão dentro do atributo `maSignal`, para um objeto da classe `Signal`, que será utilizado no construtor da classe `Block` sendo este, o retorno do método.

ParsedBlock	
Atributos	Métodos
mName	ParsedBlock(XSSFSheet)
maSignal	parseWorkbook(int)
maIgnore	toBlock

Tabela 2 – Atributos e métodos da classe `ParsedBlock`

ParsedSignal	
Atributos	Métodos
mIo	ParsedSignal(XSSFRow)
mName	getConstants
mNbw	getSignalType
mSigned	getDirection
mType	getRole
mNs	getSigned
mObs	parseOptions
mDefault	toSignal

Tabela 3 – Atributos e métodos da classe `ParsedSignal`

`ParsedSignal`, uma outra classe, possui atributos de informações que estão contidas em cada linha da planilha, na qual cada atributo será inicializado no construtor através de funções de bibliotecas de Java, que servem para manipular células de planilhas e obter os valores de cada célula.

Os métodos da classe acima referida, têm como objetivo criar objetos para que seja possível a construção dos dados de retorno da função `toSignal`. Assim, como mostrado na Tabela 3, a classe possui os seguintes métodos que serão detalhados a seguir.

- **`getConstants`**: é utilizado para criar, inicializar e retornar um vetor de objetos da classe `SignalTypeConstant`, que armazena as informações sobre o tamanho dos sinais e constata se há bits dedicados à parte inteira ou não;
- **`getSignalType`**: é utilizado para constatar de que tipo será o sinal, se terá ou não partes *packed* e se terá que utilizar lógica de *fixed point* na sua comunicação com o modelo. Utiliza-se do método `getConstants` para conseguir construir o seu objeto de retorno;
- **`getDirection`**: retornará através de um *enum* se o sinal é de entrada ou saída do bloco;
- **`getRole`**: identificará a natureza do sinal, se ele é de *reset*, *clock* ou dados/controle do bloco;

- **`getSigned`**: utilizará da informação existente na planilha sobre como o sinal irá se comportar sobre o seu sinal, ou seja, se ele possuirá ou não *bit* para representatividade do sinal;
- **`parseOptions`**: servirá para obter informações de um campo onde se pode colocar peculiaridades do sinal. Este campo varia mais de bloco para bloco.

Todos os métodos detalhados são utilizados no método `toSignal`, que retornará um objeto do tipo `Signal`. Para tanto, se faz o uso do seu construtor, este, que por sua vez, recebe o retorno de quase todos os métodos da classe e também instancia um objeto do tipo `SignalTypeHub` e do tipo `Name`, ambos necessários, junto com os métodos, para a inicialização do objeto do tipo `Signal`.

Assim, pode-se perceber que as principais classes do gerador são as classes `Block, Signal, SignalType, SignalTypeConstant`. Essas classes armazenam as informações referentes aos sinais de cada bloco da planilha, fazendo com que o acesso aos seus atributos através de métodos seja necessário e frequente.

A classe `SignalTypeConstant` possui os métodos e atributos mostrados na Tabela 4, os quais irão montar constantes para serem utilizadas nos arquivos como `#define`, para os arquivos de *wrappers*, e `parameter` para arquivos em *SystemVerilog*. O construtor da classe tem como parâmetros o nome do sinal, a quantidade de bits e se a quantidade de bits representa o sinal completo ou só a parte inteira. Com isso, o método `getConstantName` consegue retornar uma `string` com o nome da constante a ser definida.

SignalTypeConstant	
Atributos	Métodos
mName mValue mType	SignalTypeConstant(String, int, EnumSignalTypeConstant) getConstantName setConstantType findByConstantType

Tabela 4 – Atributos e métodos da classe `SignalTypeConstant`

`SignalType` é a classe que utiliza diretamente de objetos originados da classe `SignalTypeConstant`. Seus atributos possuem características que armazenam as *strings* referentes às constantes utilizadas no código, no caso também envolvendo informação. Caso se utilize de SystemC como modelo de referência, também pode-se adicionar as informações referentes ao tipo de sinal: `sc_bv, sc_fixed_fast`, etc. Entre seus atributos há um vetor do tipo `SignalTypeConstant`, no qual serão armazenadas as informações de quantidade de bit de cada sinal juntamente com o nome da sua respectiva constante.

O seu construtor recebe atributos sobre os tipos do sinal em *SystemVerilog*, com possibilidade de receber os tipos de *SystemC*, um vetor de inteiros que conterá as dimensões para o uso em *SystemVerilog*, um vetor que deverá estar em sincronismo com o vetor

anterior que determina se a dimensão do sinal será *packed/unpacked* e, por fim, um vetor com informações para a construção das constantes.

SignalType	
Atributos	Métodos
mCustomName	SignalType(String, EnumScBaseType, ArrayList<Integer>, ArrayList<EnumPackedness>, ArrayList<SignalTypeConstant>)
mSvBaseType	getConstantByType(EnumSignalTypeConstant)
mScBaseType	getConstantExist(EnumSignalTypeConstant)
mScTemplate	getParameterName(String)
mScFlatTemplate	getDimensionDeclaration
maDimension	getTemplateDeclaration
maSvPackedness	
maConstant	

Tabela 5 – Atributos e métodos da classe `SignalType`

A classe possui os seguintes métodos(vide Tabela 5):

- **`getConstantByType`**: tem como objetivo filtrar as contantes pelo tipo desejado (tamanho do sinal em *bit*, tamanho da parte inteira em *bits* do sinal, etc);
- **`getConstantExist`**: retorna se um determinado tipo de constante foi declarado, tornando assim, seu valor de retorno *True/False*;
- **`getTemplateDeclaration`**: é utilizado para compor as declarações de `typedef`;
- **`getDimensionDeclaration`**: é utilizado na construção da declaração dos sinais em *SystemVerilog*, para retornar de acordo com os valores das partes *packed* e *unpacked*.

A próxima classe é a `Signal`, na qual seus atributos formam um conjunto de informações quanto ao sinal (vide Tabela 6). Os atributos `mName` e `mDefault` são declarados como *strings*, e `mhOption` é um `HashMap`. Todos esses tipos já existem previamente em Java, tornando, assim, o conjunto de atributos que não tiveram seus tipos criados. Esse último atributo tem como objetivo conter informações sobre a última coluna, a qual foi pensada para opções que venham a se fazer necessárias sobre possíveis peculiaridades do bloco. Para o uso, é necessário um padrão que se tornará uma chave, e esta pode ou não conter valor, fazendo com que seja bastante útil para seu uso.

Os outros atributos são de tipos que foram implementados para auxiliar o armazenamento de informações advindas da planilha, como: `mPortDirection`, `mRole` e `mSigned`,

Signal	
Atributos	Métodos
mName mDefault mPortDirection mRole mSigned mhOption	Signal(Name, EnumSignalDirection, EnumRole, EnumSignedSignal, HashMap) getDeclaration getSignalDeclaration getStrBaseType getTypedefDeclaration

Tabela 6 – Atributos e métodos da classe `Signal`

que são declarados como um tipo específico de `enum`, que representam a direção do sinal, seu papel no bloco (se o sinal é de entrada ou saída, *clock* ou *reset*) e se o sinal possui ou não *bit* de sinal, respectivamente. As informações que as classes `SignalType` podem conter (ou armazenar) são acessadas através do atributo `mType`, que, assim como os outros atributos, é inicializado no construtor da classe.

Os atributos desta classe possuem um foco para retornar as declarações dos sinais. Assim, as declarações necessárias para o *testbench* e seus componentes foram implementadas como métodos desta classe. O método `getDeclaration` retornará a declaração dos sinais em *SystemVerilog*, utilizando das informações de dimensões e *packed/unpacked* contidas em seu tipo no atributo da classe. Supondo uma interação com *SystemC*, os métodos `getTypedefDeclaration` e `getStrBaseType` retornam *strings* para que seja possível a declaração dos tipos que serão utilizados nos *wrappers* para *SystemC*.

Dessa forma, com todos os elementos da variável `aBlock` inicializados devidamente, temos no programa todas as informações úteis contidas na planilha sobre todos os blocos que estejam devidamente descritos. Partindo deste ponto, pode-se utilizar os métodos e atributos contidos no tipo de `aBlock` para que os arquivos comecem a serem gerados.

O método do objeto `aBlock` invocado é `createFiles`, que irá chamar o método `setupFiles`, da própria classe. Este, por sua vez, constrói todos os arquivos através da inicialização dos atributos que referem-se aos arquivos em si. O primeiro método chamado irá invocar um método em particular de cada atributo, `write`, que tem como função receber uma *string*, a qual irá conter todo o conteúdo do arquivo. Para que esta última ação seja possível, foram implementadas classes para cada grande componente, como: driver, monitor, *wrappers* etc. Assim, Block possui atributos referentes a todos os componentes, e suas classes possuem métodos que retornam a *string*, que serão escritos nos arquivos.

Alguns métodos foram implementados para que os referidos dados fossem gerados antes que a função de escrita no arquivo fosse utilizada. Por padrão eles seguem a característica: `generate<nome do arquivo>`. Dentro desse métodos são construídos e inicializados os atributos correspondentes aos componentes como: `mEnvironment`, `mPyWrapper`, etc.

As classes dos componentes possuem métodos que irão utilizar das informações contidas nos seus atributos, que são passadas através do seu construtor que, por sua vez, recebe um objeto do tipo `Block`. Este objeto conterá todas as informações referentes a um bloco que estava na planilha. Com essas informações é possível gerar a *string* que será impressa nos arquivos, pois se optou por seguir um padrão predefinido como *testbench* e, assim, com as informações que foram adquiridas da planilha é possível determinar como será o código final.

7 Considerações Finais

Com este trabalho foi possível desenvolver um gerador de ambiente para uma equipe de verificação, o qual consiste em uma ferramente capaz de aumentar a eficiência da equipe no decorrer do processo de verificação. Sendo assim, é necessário, após a implementação do gerador, apenas a atualização da planilha, que é usada como fonte de informação com as mudanças de interface que acontecem ao longo de um projeto em microeletrônica.

Ao longo deste trabalho foram explicadas muitas das funções que foram colocadas no ambiente verificação gerado com objetivo de justificar o uso de tais funções servindo, assim, como uma introdução para quem deseja aprender mais sobre a linguagem *System-Verilog*. Foi explicado, ainda, o funcionamento do algorítimo algoritmo como um todo do gerador e como as classes foram organizadas, de modo que se o leito desejar, é possível a implementação de um gerador sem precisar necessariamente replicar os códigos.

Como projeto futuros para serem acrescentados a esse gerador, tem-se a implementação da geração dos *scripts* de compilação e simulação, sem necessitar que sejam feitos manualmente pelo verificador, implementação de UVM (*Universal Verification Methodology*) no ambiente de verificação, realização de carregamento carregamento nas ferramenta de controle de versão, para que cada bloco no início do projeto já contenha o *stub* do ambiente e para agilizar a atualização do repositório de acordo com as mudanças que ocorrem no decorrer do projeto e, mais adiante, a implementação de um interface gráfica para o uso.

8 Anexo

8.1 Códigos exemplos dos *wrappers* de comunicação Python - *SystemVerilog*

8.1.1 Código exemplo em *SystemVerilog* usando DPI-C

```
module tb_dpi();

   //Struct que é reponsável pela comunicação entre SystemVerilgo e C,
   //cada elemento aqui deve está na mesma posição que seu respectivo
   //elemento na stuct em C, com o seu tipo equivalente.
   typedef struct{
      int       i;
      int       FstD;
      int       SndD;
      int       ThdD;
      int       FthD;
      bit       X;
      bit       [4:0]XVector;
      real      RealVector[5];
      string    m;
      string    l;
      string    str;
      int       obj;
   } cStruct_t;


   //Enum para o switch-case em C para determinar o tipo de dados
   //que está sendo enviado
   typedef enum {
      kString, kListNp, kTuple, kBit, kBitVector, kReal
   } DataType_t;

   //Importanto função de C responsável por colocar valores no python
   import "DPI-C" context c_set_func_python
            = function sv_set_func_python(
                                          inout cStruct_t io_struct,
                                          inout DataType_t dataType,
                                          inout int dyn_arr[][][][]
                                       );

   //Importanto função de C responsável por receber os valores do pyhotn
   import "DPI-C" context c_get_func_python
            = function sv_get_func_python(
                                          inout cStruct_t io_struct,
                                          inout DataType_t dataType,
                                          inout int dyn_arr[][][][]
                                       );

   //Importantdo função de C responsável por fechar o interpretador
   //de python
```

```
46  import "DPI-C" context end_python = function sv_end_python ();
47
48  //Importando função de C reponsável por abrir o interpretador
49  //de python e inicializar o objeto que será link do modulo python
50  import "DPI-C" context init_python = function sv_init_python ();
51
52  //importanto função de C responsável por interpretar um open array
53  //de SystemVerilog em C e converte-lo para um tipo reconhecido
54  //pelo python
55  import "DPI-C" context get_open_array
56          = function sv_get_open_array(inout int dyn_arr[]);
57
58  cStruct_t cStruct;   // Instanciação da variável da struct
59  DataType_t dataType; // Instanciação da variável do enum
60
61  int data[];
62  int data_know[3][3][3][3];
63
64  initial begin
65
66     static int DataSize = 5;
67
68     //Chamando função de inicialização do interpretador python
69     void'(sv_init_python());
70
71     //ENVIANDO UM VETOR REAL SV -> C -> PYTHON
72     $display("Enviando um vetor real \n");
73     dataType  = kReal;                          //definindo o tipo de
74                                                 //dado que será enviando
75     cStruct.RealVector[0] = 3.1;
76     cStruct.RealVector[4] = 4.5;
77     void'(sv_set_func_python(
78                               cStruct,       //chamando função em C
79                               dataType,
80                               data_know
81                             ));
82     $display("Sent");
83
84     //RECEBENDO UM VETOR REAL SV <- C <- PYTHON
85     $display("Recebendo um Vetor Real \n");
86     dataType  = kReal;
87     void'(sv_get_func_python(
88                               cStruct,  //chamando função em C
89                               dataType,
90                               data_know
91                             ));
92     $display("Dado recebido: %f", cStruct.RealVector[0]);
93     $display("Dado recebido: %f", cStruct.RealVector[4]);
94
95     //Enviando um Bit SV -> C -> PYTHON
96     $display("Enviando um Bit \n");
97     dataType  = kBit;
98     cStruct.X = 1'b1;
99     void'(sv_set_func_python(
100                              cStruct,
101                              dataType,
102                              data_know
103                            ));
```

```
104         $display("Sent");
105
106         //RCEBENDO UM BIT SV -> C -> PYTHON
107         $display("Recebendo a BIT \n");
108         dataType  = kBit;
109         cStruct.X = 1'b0;
110         void'(sv_get_func_python(
111                                  cStruct,
112                                  dataType,
113                                  data_know
114                                 ));
115         $display("------> %b", cStruct.X);
116         $display("Sent");
117
118         //ENVIANDO UMA TUPLE SV -> C -> PYTHON
119         $display("Enviando uma tupla");
120         dataType  = kTuple;
121         cStruct.m = "Primeiro elemento da tupla";
122         cStruct.l = "Segundo elemento da tupla";
123         void'(sv_set_func_python(
124                                  cStruct,
125                                  dataType,
126                                  data_know
127                                 ));
128         $display("Enviado");
129
130         //Recebendo uma tupla SV -> C -> PYTHON
131         //Importante notar que para receber uma tupla
132         //deve-se ter um conhecimento previo de que tipo
133         //sera cada elemento da tupla, neste exemplo
134         //espera-se uma string como retorno da tupla
135         $display("Recebendo uma tupla \n");
136         dataType  = kTuple;
137         void'(sv_get_func_python(
138                                  cStruct,
139                                  dataType,
140                                  data_know
141                                 ));
142         $display("STRING R: %s", cStruct.str);
143         $display("Recebido");
144
145         void'(sv_end_python()); //funçao em C para finalizar
146                                 //interpretador python
147
148         $finish;                //função de sistem do SystemVerilog
149                                 //que finaliza a simulação
150     end
151 endmodule
```

8.1.2 Código exemplo em C servindo com *wrappers*

```
1 #include <stdio.h>
2 #include <Python.h>
3 #include <pythonrun.h>
4 #include <ctype.h>
5 #define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
```

```
6  #define DPI_COMPATIBILITY_VERSION_1800v2005
7  #include <arrayobject.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <getopt.h>
12 #include <vpi_user.h>
13 #include <svdpi.h>
14 #include <string.h>
15
16 //para multiplos modelos
17 #ifdef NUMBER_PYTHON
18    PyObject *pBlockObject[NUMBER_PYTHON];
19 #else
20    PyObject *pBlockObject[1];
21 #endif
22
23 //estrutura que serve para transladar alguns tipos de dados entre o SystemVerilog
24 typedef struct {
25    int             i;
26    int             FstD;
27    int             SndD;
28    int             ThdD;
29    int             FthD;
30    unsigned char   X;
31    svBitVecVal     XVector[SV_PACKED_DATA_NELEMS(5)];
32    double          RealVector[5];
33    char            *m;
34    char            *l;
35    char            *str;
36    int             obj;
37 } mystruct_t;
38
39 //enum utilizada no exemplo para selecionar qual tipo de dado esta sendo
40 //enviado ao Python ou recebido do Python
41 typedef enum _DataType {
42   kListNp, kTuple, kBit, kBitVector, kReal
43 } DataType_t;
44
45
46 //Função utiliada para inicializar o obejeto que deseja-se em Python
47 void init_python(){
48
49         char pydir[200];
50         char pwd[200];
51
52    PyObject *pUnicodeBlockName, *pBlockWrapper, *pBlockClass;
53
54    //inicializa um interpretador em Python
55         Py_Initialize();
56
57         PyRun_SimpleString("import sys");
58    strcpy(pydir, "sys.path.append('");
59         if (getcwd(pwd, sizeof(pwd)) != NULL)
60                 strcat(pydir, pwd);
61         strcat(pydir, "/../tb");
62         strcat(pydir, "')");
63         PyRun_SimpleString(pydir);
64
```

```
65      //cria uma string contendo o nome do arquivo que tem o código em python
66      //que se deseja importar
67      pUnicodeBlockName = PyUnicode_FromString("exemplo");
68           assert(pUnicodeBlockName != NULL);
69
70           pBlockWrapper = PyImport_Import(pUnicodeBlockName);
71           if(pBlockWrapper == NULL){
72                   PyErr_Print();
73                   exit(1);
74           }
75
76      //cria um objeto que referência uma classe dentro do arquivo importado
77      //anteriormente
78            pBlockClass = PyObject_GetAttrString(pBlockWrapper, "classe_py");
79            assert(pBlockClass && PyCallable_Check(pBlockClass));
80
81      //Inicialilza o objeto anteriormente criado, chamando seu construtor
82           pBlockObject = PyObject_CallObject(pBlockClass, NULL);
83           if(pBlockObject  == NULL){
84                   PyErr_Print();
85                   exit(1);
86           }
87
88      printf("\n PYTHON INITIALIZED \n");
89  }
90
91  //função dedicada a procedimento de leitura de dados em Python
92  void c_get_func_python( mystruct_t *io_struct,
93                          DataType_t *dataType,
94                          const svOpenArrayHandle dyn_arr){
95           char *msg="";
96      void *pArrayElement;
97      long pBit;
98
99      //objetos de python em C que serão utilizados ao longo deste código
100          PyObject *pBlockMethod, *pMethodReturn, *pTupleObject, *pTupleRepr;
101     PyObject* pUtf8String;
102
103     //objeto criado de maneira a se assemelhar com as caracterísitcas
104     //de um numpy array
105     PyArrayObject* pMethodArray;
106     PyObject *pMethodReturnNp;
107
108     Py_ssize_t t = 3;
109     npy_intp   i = 1;
110     npy_intp   j = 2;
111
112     switch(*dataType){
113        case(kListNp):
114
115                pBlockMethod = Py_BuildValue("s", "get_test_list_np");
116                 assert(pBlockMethod != NULL);
117
118                pMethodReturnNp = PyObject_CallMethodObjArgs( pBlockObject,
119                                                              pBlockMethod,
120                                                              NULL);
121                if(pMethodReturnNp == NULL){
122                        PyErr_Print();
123                        exit(1);
```

```
124             }
125
126         pMethodArray = (PyArrayObject *)pMethodReturnNp;
127         pArrayElement = PyArray_GETPTR2(pMethodArray, i, j);
128
129         io_struct->i = *(int *)pArrayElement;
130
131         break;
132     case(kTuple):
133
134             pBlockMethod = Py_BuildValue("s", "get_test_tuple");
135              assert(pBlockMethod != NULL);
136
137             pMethodReturn = PyObject_CallMethodObjArgs( pBlockObject,
138                                                         pBlockMethod,
139                                                         NULL);
140             if(pMethodReturn == NULL){
141                     PyErr_Print();
142                     exit(1);
143             }
144             if(!PyTuple_Check(pMethodReturn)){
145                     PyErr_Print();
146                     exit(1);
147             }
148
149         pTupleObject = PyTuple_GetItem(pMethodReturn, t);
150             if(pTupleObject == NULL){
151                     PyErr_Print();
152                     exit(1);
153             }
154
155         pTupleRepr = PyObject_Str(pTupleObject);
156             if(pTupleRepr == NULL){
157                     PyErr_Print();
158                     exit(1);
159             }
160
161         pUtf8String = PyUnicode_AsUTF8String(pTupleRepr);
162             if(pUtf8String == NULL){
163                     PyErr_Print();
164                     exit(1);
165             }
166
167         //convertentdo conteúdo da tupla para um representação em string
168         msg = PyBytes_AsString(pUtf8String);
169             if(msg == NULL){
170                     PyErr_Print();
171                     exit(1);
172             }
173
174         io_struct->str = msg;
175         break;
176
177     case(kBit):
178             pBlockMethod = Py_BuildValue("s", "get_test_bit");
179              assert(pBlockMethod != NULL);
180
181             pMethodReturn = PyObject_CallMethodObjArgs( pBlockObject,
182                                                         pBlockMethod,
```

```
183                                                     NULL);
184             if(pMethodReturn == NULL){
185                     PyErr_Print();
186                     exit(1);
187             }
188
189         pBit  = PyLong_AsLong(pMethodReturn);
190         if(pBit == -1){
191            PyErr_Print();
192            exit(1);
193         }
194
195         io_struct->X = pBit;
196
197         break;
198
199       case(kReal):
200
201             pBlockMethod = Py_BuildValue("s", "get_test_real");
202              assert(pBlockMethod != NULL);
203
204             pMethodReturn = PyObject_CallMethodObjArgs( pBlockObject,
205                                                     pBlockMethod,
206                                                     NULL);
207             if(pMethodReturn == NULL){
208                     PyErr_Print();
209                     exit(1);
210             }
211
212         pMethodArray = (PyArrayObject *)pMethodReturn;
213
214         for(npy_intp i = 0; i < 5; i++){
215            pArrayElement = PyArray_GETPTR1(pMethodArray,i);
216            io_struct->RealVector[i] = *(double *)pArrayElement;
217         }
218
219         break;
220
221       default:
222          printf("ERROR");
223          break;
224    }
225 }
226
227
228 //função dedicada a procedimento que enviar informações ao Python
229 void c_set_func_python( mystruct_t *io_struct,
230                         DataType_t *dataType,
231                         const svOpenArrayHandle dyn_arr){
232
233    int FirstDim  = io_struct->FstD;
234    int SecondDim = io_struct->SndD;
235    int ThirdDim  = io_struct->ThdD;
236    int FourthDim = io_struct->FthD;
237
238
239    double Value;
240
241          PyObject *pBlockMethod, *pMethodReturn, *pTupleObject;
```

```
242     PyObject *pSetArguments,*pData, *pBit;
243     PyObject *pListNp, *pListNp2, *pListNp3, *pListNp4;
244     PyObject *pRealList;
245
246     //tipos da própria biblioteca da DPI para que seja possível a troca de dados
247     //contidos em open array
248     svBit a;
249     svBitVecVal *ValueToPython[SV_PACKED_DATA_NELEMS(5)];
250
251     PyObject *First  = Py_BuildValue("s", io_struct->m);
252     PyObject *Second = Py_BuildValue("s", io_struct->l);
253
254     switch(*dataType){
255        case(kListNp):
256           //enviando um open array para o Python
257                pBlockMethod = Py_BuildValue("s", "set_test_list_np");
258                 assert(pBlockMethod != NULL);
259           if(pBlockMethod == NULL){
260              PyErr_Print();
261              exit(1);
262           }
263
264           pListNp = PyList_New(FirstDim);
265           if(pListNp == NULL){
266               PyErr_Print();
267               exit(1);
268           }
269
270           for(int x = 0; x < FirstDim; x++){
271              pListNp2 = PyList_New(SecondDim);
272              if(pListNp2 == NULL){
273                  PyErr_Print();
274                  exit(1);
275              }
276
277              for(int z = 0; z < SecondDim; z++){
278                 pListNp3 = PyList_New(ThirdDim);
279                 if(pListNp4 == NULL){
280                     PyErr_Print();
281                     exit(1);
282                 }
283                 for(int k = 0; k < ThirdDim; k++){
284                    pListNp4 = PyList_New(FourthDim);
285                    if(pListNp4 == NULL){
286                        PyErr_Print();
287                        exit(1);
288                    }
289
290                    for(int l = 0; l < FourthDim; l++){
291                       pData = Py_BuildValue("i", *(int *)svGetArrElemPtr(dyn_arr,
292                                                                          x,
293                                                                          z,
294                                                                          k,
295                                                                          l));
296                       assert(pData != NULL);
297
298                       if(PyList_SetItem(pListNp4, l, pData) == -1 ){
299                           PyErr_Print();
300                           exit(1);
```

```
301                     }
302                 }
303                 if(PyList_SetItem(pListNp3, k, pListNp4) == -1 ){
304                     PyErr_Print();
305                     exit(1);
306                 }
307              }
308
309              if(PyList_SetItem(pListNp2, z, pListNp3) == -1 ){
310                  PyErr_Print();
311                  exit(1);
312              }
313           }
314
315           if(PyList_SetItem(pListNp, x, pListNp2) == -1 ){
316               PyErr_Print();
317               exit(1);
318           }
319        }
320
321        pMethodReturn = PyObject_CallMethodObjArgs(pBlockObject,
322                                                   pBlockMethod,
323                                                   pListNp,
324                                                   NULL);
325             if(pMethodReturn == NULL){
326                     PyErr_Print();
327                     exit(1);
328             }
329
330        break;
331
332     case(kTuple):
333        //enviando dados como uma tupla para o Python
334
335             pBlockMethod = Py_BuildValue("s", "set_test_tuple");
336              assert(pBlockMethod != NULL);
337
338
339        pTupleObject = PyTuple_Pack(2 , First, Second);
340        if(pTupleObject == NULL){
341            PyErr_Print();
342            exit(1);
343        }
344
345        pMethodReturn = PyObject_CallMethodObjArgs(pBlockObject,
346                                                   pBlockMethod,
347                                                   pTupleObject,
348                                                   NULL);
349             if(pMethodReturn == NULL){
350                     PyErr_Print();
351                     exit(1);
352             }
353
354        break;
355
356     case(kBit):
357        //enviando um bit para o Python
358
359             pBlockMethod = Py_BuildValue("s", "set_test_bit");
```

```
360                 assert(pBlockMethod != NULL);
361
362             pBit = Py_BuildValue("i", io_struct->X);
363
364             pMethodReturn = PyObject_CallMethodObjArgs(pBlockObject,
365                                                        pBlockMethod,
366                                                        pBit,
367                                                        NULL);
368                 if(pMethodReturn == NULL){
369                         PyErr_Print();
370                         exit(1);
371                 }
372             break;
373
374
375         case(kReal):
376             //enviando um valor real para Python
377
378             pBlockMethod = Py_BuildValue("s", "set_test_real");
379                   assert(pBlockMethod != NULL);
380             if(pBlockMethod == NULL){
381                PyErr_Print();
382                exit(1);
383             }
384
385             pRealList = PyList_New(5);
386             if(pRealList == NULL){
387                 PyErr_Print();
388                 exit(1);
389             }
390
391             for(int l = 0; l < 5; l++){
392
393                pData = Py_BuildValue("d", io_struct->RealVector[l]);
394                assert(pData != NULL);
395
396                if(PyList_SetItem(pRealList, l, pData) == -1 ){
397                   PyErr_Print();
398                   exit(1);
399                }
400             }
401
402             pMethodReturn = PyObject_CallMethodObjArgs(pBlockObject,
403                                                        pBlockMethod,
404                                                        pRealList,
405                                                        NULL);
406                 if(pMethodReturn == NULL){
407                         PyErr_Print();
408                         exit(1);
409                 }
410
411             break;
412
413         default:
414             printf("\n\n\nERROR\n\n\n");
415             exit(1);
416
417     }
418 }
```

```
419
420 //função para finalizar o interpretador de Python
421 void end_python(){
422    Py_Finalize();
423    printf("\n PYTHON FINALIZED \n");
424 }
```

8.1.3 Código exemplo em Python que será acessado via o código em 8.1.2

```
1  import numpy as np
2  import ctypes
3
4  class de_custom:
5     def __init__(self):
6          print("started python");
7
8          self.instance_name = "DE_test_DPI";
9          self.test_string   = "String from Python"
10         self.test_int      = 1000
11         self.test_bit      = 0
12         self.test_Real     = np.array([1.3 , 2, 3, 4])
13         self.test_list     = [11, 12, 13, 10, 15]
14         self.test_list2    = [21, 22, 23]
15         self.test_list_np  = np.array([self.test_list,self.test_Real])
16         self.test_tuple    = (1000, 1001, 2001, "string test tuple")
17
18    def get_test_bit(self):
19       print("BIT PYTHOM: ", self.test_bit)
20       return self.test_bit
21
22    def get_test_list_np(self):
23       return self.test_list_np
24
25    def get_test_real(self):
26       return self.test_Real
27
28    def get_test_tuple(self):
29       return self.test_tuple
30
31    def set_test_list_np(self, *iListNp):
32       self.test_list_np =np.array(iListNp)
33       print("NUMPY ARRAY EXAMPLE: ")
34       print(self.test_list_np)
35
36    def set_test_tuple(self, iTuple):
37       self.test_tuple = iTuple
38       print("TUPLE EXAMPLE: ")
39       print(self.test_tuple)
40
41    def set_test_bit(self, iBit):
42       self.test_bit = iBit
43       print("BIT EXAMPLE: ")
44       print(self.test_bit)
45
46    def set_test_real(self, *iReal):
47       self.test_Real = np.array(iReal[0])
```

```
48        print("REAL RECEIVED: ")
49        print(self.test_Real)
50
```

8.2 Modelos utilizados para o *testbench* gerado

8.2.1 Modelo do `testbench.sv`

```
1  //Os arquivos podem ser incluido manualmente, ou serem eleborador junto
2  //pelo mesmo comando da ferramenta de simulação, como o objetivo do
3  //gerador é também gerar o script de simulação, então os includes serão
4  //omitidos do código
5  //`include "interface.sv"
6  //include "random_test.sv"
7  //------------------------------------------------------------------------
8
9  module block_name_tb_top;
10
11    //clock and reset signal declaration
12    bit clk;
13    bit reset;
14
15    //clock generation
16    always #5 clk = ~clk;
17
18    //Criando as intâncias das interfaces necessárias para as instâncias do
19    //DUT e do modelo de referência
20    block_interface #(<parameters>) block_rtl_if_00(clk,reset);
21    //              .
22    //              .
23    //              .
24    //Partindo do principio que o modelo irá está em Python
25    block_interface #(<parameters>) block_py_if_00(clk,reset);
26
27    //Instância do teste, recebendo todas as interfaces que foram
28    //declaradas
29    test #(<parameters>) test(block_py_if_00, block_rtl_if_00);
30
31    //As instância que seguem podem ser multiplicadas caso seja
32    //necessário
33    //
34    //Instanciação do DUT e a conexão com sua respectiva interface
35    block_name DUT (
36       .i_valid(   block_rtl_if_00.cb.i_valid),
37       .i_data(    block_rtl_if_00.cb.i_data),
38    //     .
39    //     .
40    //     .
41       .o_valid(   block_rtl_if_00.cb.o_data)
42    );
43
44    block_name_py py_model (
45       .i_valid(   block_py_if_00.cb.i_valid),
46       .i_data(    block_py_if_00.cb.i_data),
47    //     .
```

```
48      //      .
49      //      .
50         .o_valid(   block_py_if_00.cb.o_data)
51      );
52
53    //enabling the wave dump
54    initial begin
55      $dumpfile("dump.vcd"); $dumpvars;
56    end
57  endmodule
```

8.2.2 Modelo da `interface.sv`

```
1   interface block_name_interface #(<parameter>)(input logic clk,reset);
2
3      //Declarando os sinais do bloco
4      wire  [TAM_DATA-1:0] i_data;
5      wire  i_valid;
6      //      .
7      //      .
8      //      .
9      wire  [TAM_DATA-1:0] o_data;
10     wire  o_valid;
11
12
13     //Clocking block
14     clocking cb @(posedge clk);
15     default input #1 output #1; //com os valores desejado
16        inout  i_data;
17        inout  i_valid;
18        //      .
19        //      .
20        //      .
21        inout  o_data;
22        inout  o_valid;
23     endclocking
24
25  endinterface
```

8.2.3 Modelo do `test.sv`

```
1   program #(<parameters>)test(  block_name_interface intf_rtl,
2                                 block_name_interface intf_model);
3
4
5
6     //declarando tipos da transação e da interface virtual
7     typedef virtual block_name_interface #(<parameters>) block_name_vif_t;
8     typedef block_name_transaction #(<parameters>) block_name_item_t;
9
10    //declarando o ambiente
11    block_name_env #(<paramters>) block_name_env;
12
13    //declarando a mailbos para comunicação
14    mailbox #(block_name_item_t) test2drv_item_mb;
```

```
15
16    //declarando item da transaction
17    block_name_interface_t item;
18
19    initial begin
20      //criando o ambiente
21      block_name_env = new(intf_rtl, intf_model);
22
23      //lógica do teste a ser implementada
24      //...
25    end
26  endprogram
```

8.2.4 Modelo do `environment.sv`

```
1   class block_name_environment;
2
3     //instanciaas do gerador e do monitor
4     block_name_driver #(<parameters>) m_drv;
5     block_name_monitor #(<parametesr>) m_mon;
6
7     //declarando tipo da transação
8     typedef block_name_transaction #(<parameters>) block_name_item_t;
9
10    //mailbox
11    mailbox #(block_name_item_t) test2drv_item_mb;
12
13    //virtual interface
14    typedef virtual block_name_interface #(<parameters>) block_name_vif_t;
15
16    //instâncias da interface no ambiente
17    block_name_vif_t m_vif_model;
18    block_name_vif_t m_vif_rtl;
19
20    //quantidade de ciclos a serem simuladas
21    int test_cycles;
22
23    //construtor
24    function new(block_name_vif_t vif_model,
25                 block_name_vif_t vif_rtl,
26                 mailbox #(block_name_item_t) test2drv_item_mb,
27                 int test_cycles);
28      //recebe a interface do test
29      this.m_vif = vif_model;
30      this.m_rtl = vif_rtl;
31      this.test2drv_item_mb = test2drv_item_mb;
32      this.test_cycles = test_cycles;
33
34      //criando driver e monitor
35      m_drv = new(vif_rtl, test2drv_item_mb, test_cycles);
36      m_mon = new(vif_rtl, vif_model, test_cycles)
37    endfunction
38
39    //task do ambiente
40    task run();
41      fork
42      m_drv.run();
```

```
43      m_mon.run();
44      post_test();
45      join_any
46    endtask
47
48    task post_test();
49      wait(m_mon.transaction_cnt == test_cycles);
50      wait(m_drv.no_transactions == m_mon.transaction_cnt);
51    endtask
52
53    //run task
54    task run;
55      fork
56        test();
57        post_test();
58      join_any
59      disable fork;
60      $finish;
61    endtask
62
63  endclass
```

8.2.5 Modelo do `driver.sv`

```
1   class block_name_driver;
2
3     //uado para contar o número de iterações que já
4     //transcorreram
5     int no_transactions;
6     int test_cycles;
7
8     //declarando tipos da transação e da interface virtual
9     typedef virtual block_name_interface #(<parameters>) block_name_vif_t;
10    typedef block_name_transaction #(<parameters>) block_name_item_t;
11
12    //criando a interface virtual
13    block_name_vif_t m_vif_rtl;
14
15    //criando item da transação;
16    block_name_item_t block_name_item;
17
18    //mailbox
19    mailbox #(block_name_item_t) test2drv_item_mb;
20
21    //construtor
22    function new( block_name_vif_t block_name_vif,
23                  mailbox #(block_name_item_t) test2drv_item_mb,
24                  int test_cycles);
25      //recebendo a interface
26      this.m_vif_rtl = block_name_vif;
27      //recebendo a mail box
28      this.test2drv_item_mb = test2drv_item_mb;
29      this.test_cycles = test_cycles;
30    endfunction
31
32    //Reset task
33    task reset;
```

```
34      wait(!m_vif_rtl.rst);
35      $display("--------- [DRIVER] Reset Started ---------");
36      //sinais que devem ser resetados;
37      wait(m_vif_rtl.rst);
38      $display("--------- [DRIVER] Reset Ended ---------");
39    endtask
40
41    //task drive
42    task drive;
43        test2drv_item_mb.get(block_name_item);
44        //sinais que deverão receber a informação da transação
45        no_transactions++;
46    endtask
47
48    task main;
49      no_transactions = 0
50      forever begin
51        fork
52
53          begin
54            forever reset();
55          end
56
57          begin
58            forever drive();
59          end
60        join_any
61        disable fork;
62      end
63    endtask
64
65  endclass
```

8.2.6 Modelo do `monitor.sv`

```
1   class block_name_monitor;
2
3     //declarando tipos da interface virtual
4     typedef virtual block_name_interface #(<parameters>) block_name_vif_t;
5
6     //instâncias da interface no ambiente
7     block_name_vif_t m_vif_model;
8     block_name_vif_t m_vif_rtl;
9
10    //declaração de número de ciclos
11    int test_cycles;
12
13    //construtor
14    function new(block_name_vif_t vif_model,
15                 block_name_vif_t vif_rtl,
16                 int test_cycles);
17
18      this.m_vif_model = vif_model;
19      this.m_vif_rtl = vif_rtl;
20      this.test_cycles = test_cycles;
21    endfunction
22
```

```
23    task monitor();
24      //sinais que deveram ser comparados na saída
25    endtask
26
27
28    task run;
29      fork
30        forever monitor();
31      join
32    endtask
33
34  endclass : monitor
```

8.2.7 Modelo do `transaction.sv`

```
1   class block_name_transaction;
2     //declração dos itens conditos na transação, geralmente
3     //entradas do bloco
4     //
5     rand logic [9:0] i_data;
6
7     //deep copy method
8     function void do_copy(
9       block_name_transaction rhs);
10      this.i_data = rhs.i_data;
11      //...
12      //
13    endfunction
14
15  endclass
```

Referências

FOUNDATION, P. S. *Python/C API Reference Manual*. 2018. Disponível em: <https://docs.python.org/3.6/c-api/index.html>. Citado na página 21.

IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. [S.l.]. Citado 2 vezes nas páginas 21 e 23.

SYSTEMVERILOG TestBench. 2016. Disponível em: <https://www.verificationguide.com/>. Citado na página 24.