Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Kelvin Dantas Vale

RAL - Register Abstraction Layer

Campina Grande
2021

Kelvin Dantas Vale

RAL - Register Abstraction Layer

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Eletrônica

Orientador:
Marcos Ricardo de Alcântara Morais, D. Sc.

Campina Grande
2021

Kelvin Dantas Vale

RAL - Register Abstraction Layer

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Aprovado em / /

Marcos Ricardo de Alcântara Morais, D. Sc.
UFCG

Gutemberg Gonçalves dos Santos Júnior, D. Sc
Professor Convidado
UFCG

Campina Grande

2021

Dedico este trabalho à todos que me acompanharam e me ajudaram até aqui. Até mesmo um único dia diferente na minha história poderia alterar tudo até o dia de hoje.

AGRADECIMENTOS

Agradeço primeiramente à Deus pelo dom da vida e pela contante iluminação deste proveniente.

Agradeço aos meus pais. Minha mãe por, independente das dificuldades na criação de uma criança com deficit de atenção, nunca me deixou desacreditar de mim mesmo e sempre me provar que eu era capaz de alcançar além dos céus. Meu pai, por me apresentar ao incrível mundo da matemática e da física de forma tão lúdica e pelas incontáveis e memoráveis discussões nesta área desde minha infância. Se não fosse pela incrível união de quem vocês são eu jamais chegaria onde estou.

Agradeço à minha irmã casula, que mostrou durante anos que com organização e dedicação pode-se aprender qualquer coisa. O mundo é seu! É só se dedicar o suficiente.

Agradeço ao restante da minha família por nunca terem duvidado de mim e sempre me mostrado que a vida pode ser vivida de um modo mais simples e fácil.

Agradeço a todos do Laboratório XMEN pela incrível oportunidade de trabalhar com vocês e aprender tanto todos os dias.

E, por fim, agradeço à minha amada esposa por ter me aguentado durante todo esse processo de estágio e TCC, que demandou tanto tempo e atenção. E por ter mantido minha cabeça no lugar quando eu mesmo já estava perdido.

RESUMO

Para a metodologia UVM, foi definida uma biblioteca de classes e propostas arquiteturas para a codificação do ambiente de verificação. Visando criar uma ferramenta facilitadora de acesso à registradores, o método RAL foi desenvolvido. Esta se baseia na descrição de um modelo de bloco de memória, por meio do uso de classes pré definidas, e na aplicação das funções fornecidas pelas classes bases.

Palavras-chave: Desenvolvimento de *hardware*, RAL, Register Abstraction Layer, *Hardware Design*, Verificação, UVM.

ABSTRACT

To the UVM methodology, was defined a library of classes and suggested architectures to code a verification enviroment. Aiming to develop a tool to make easier the access of registers, the RAL method was created. This is based in a block memory model description, using the pré defined classes, and the execution of the functions provided from the classes.

Keywords: Hardware development, RAL, Register Abstraction Layer, Hardware Design, Verification, UVM.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

UFCG	Universidade Federal de Campina Grande
DEE	Departamento de Engenharia Elétrica
RAL	*Register Abstraction Layer*
HDL	*Hardware Description Language*
HVL	*Hardware Verification Language*
ASIC	*Application Specific Integrated Circuits*
TDD	*Test Driven Development*
UVM	*Universal Verification Methodology*
SVA	*SystemVerilog Assertions*

SUMÁRIO

1 INTRODUÇÃO

A demanda por aperfeiçoamento tecnológico, tanto por consumidores a nível domésticos quanto empresas de grande porte, impacta diretamente na produção de eletrônicos, forçando assim novos, ou melhorados, produtos a virem ao mercado. De maneira proporcional ao aperfeiçoamento tecnológico, é notória, desde a base do consumo, uma maior necessidade de processamento e memória, como observado nos *smartphones*, que a cada semestre surgem com câmeras melhores, mais memória RAM e processadores com potencial computacional cada vez maior.

Para que a capacidade de processamento dos eletrônicos aumente, é necessário que haja inovação na área de microeletrônica. Com o avanço dos estudos nesta área, foi possível produzir circuitos com transistores cada vez menores e, assim, elevar a densidade dos mesmos em um chip sem mudar a sua área total, gerando um aumento no *throughput* de dados. Porém, a vertente física não foi a única afetada pelas inovações, surgiram também novas metodologias, métodos e ferramentas para auxiliar na criação dos projetos, estas melhorias elevaram a complexidade dos circuitos integrados, permitindo-os atingir, com a mesma quantidade de transistores, um maior poder computacional.

Entretanto, com o aumento da complexidade dos circuitos, há também uma maior chance de ocorrer erros em seu projeto, ou mesmo não poder ser possível a síntese do mesmo. Para sanar este problema, foram testados fluxos de produção que estão constantemente evoluindo e se adaptando. Tradicionalmente, em um fluxo, a equipe comumente se divide em 3 grupos, *frontend*, responsável pela construção lógica e arquitetural do circuito a ser projetado, a verificação, responsável por garantir a correta funcionalidade do circuito projetado e, por fim, a equipe de *backend* responsável por realizar a síntese do circuito para que o mesmo possa ser fabricado.

Tomando como base o fluxo de desenvolvimento acima, após a construção lógica do projeto e antes da sua construção física, há a verificação do projeto. Tal etapa é de suma importância em um fluxo de desenvolvimento, pois a mesma, mesmo que não acrescente lógica ao projeto nem o construa fisicamente, minimiza consideravelmente as perdas por erro de projeto. Ou seja, a verificação garante uma menor perda para a empresa e eleva a confiabilidade do produto final. Tal se comprova a partir da estimativa de que, no fluxo apresentado, mais da metade do esforço total do projeto é dedicado à verificação funcional, contando mão de obra, cronograma e custos.

Para a verificação de um projeto, é necessário que o mesmo seja levado a um estado de estresse, ou por um número elevado de estímulos aleatórios, ou pela entrada de sequências específica de dados que, após a fabricação do chip, não pode haver erro. Com

o objetivo de criar uma metodologia de verificação padrão, diminuindo a chance de erro individual, tendo em vista que a mesma terá a contribuição de muitos engenheiros em sua especificação e pelo fato de não partir do zero a cada novo projeto, foi desenvolvida a metodologia UVM (*Universal Verification Methodology*). Nesta, foram criadas arquiteturas de ambientes de verificação, blocos com objetivos específicos de uso e métodos que auxiliam o verificador a cumprir seu objetivo de maneira mais rápida, segura e eficiente.

Fazendo uso da UVM, alguns tipos de verificação surgiram, tal como a verificação funcional, cujo objetivo é garantir que, independente de como tenha sido escrito o código, a sua funcionalidade macro esteja conforme a especificação do projeto. Para elevar a eficiência na verificação funcional de um bloco com banco de registradores, foi desenvolvida uma ferramenta que possui a capacidade de observar e alterar os valores de um registrador, esta foi denominada de RAL(Register Abstraction Layer), porém, mesmo que haja um grande valor na aplicação desta ferramenta, não há muitos materiais para o estudo, compreensão e aplicação da mesma.

Isto posto, o presente trabalho visa explanar o funcionamento de um ambiente de verificação tradicional em UVM com a aplicação do RAL, bem como a sua construção. Para tal, foi desenvolvido um bloco simples de memória e explanado o passo a passo para a integração da ferramenta ao ambiente e a execução da manipulação de memória.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

O desenvolvimento deste trabalho tem como objetivo o estudo da ferramenta de manipulação de registradores, RAL.

1.1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos para o desenvolvimento desse projeto são:

- Estudar a ferramenta RAL;
- Aplicar a mesma em um ambiente de verificação;
- TO DO

2 EMBASAMENTO TEÓRICO

Para a implementação da ferramenta em análise é necessário possuir um conhecimento prévio da linguagem de programação SystemVerilog e dos componentes de um ambiente UVM. Mesmo que o RAL não influencie diretamente em todos os módulos, sua atuação gera uma reação sistemática.

2.1 UVM

2.1.1 TRANSAÇÃO

Na metodologia UVM, a passagem de informação entre blocos se faz por meio de transações. Estas transações nada mais são do que pacotes de dados com campos e tamanhos pré definidos pelo verificador, podendo ter quantos campos forem necessário, assim como a quantidade de bits em cada um destes.

Definição em código

```
class some_transaction extends uvm_sequence_item;
  rand logic unsigned [x:0] some_field; //x -> Define o tamanho do campo.

  `uvm_object_utils_begin(some_transaction)
    `uvm_field_int(some_field, UVM_ALL_ON|UVM_HEX)
  `uvm_object_utils_end

  function new(string name = "some_transaction");
    super.new(name);
  endfunction : new

  function string convert2string();
    return $sformatf("{some_field = %h}",some_field);
  endfunction

  function void do_copy(uvm_object rhs);
        some_transaction rhs_;

        if(!$cast(rhs_, rhs)) begin
          `uvm_fatal("do_copy", "cast of rhs object failed")
        end
        super.do_copy(rhs);
        some_field = rhs_.some_field;
```

```
24     endfunction : do_copy
25
26 endclass : some_transaction
```

2.1.2 PORTAS

As portas possuem a função de servirem de passagem para as transações. Tradicionalmente, todos os blocos em UVM possuirão portas cujas funcionalidade serão determinadas pela classe da qual foram instanciadas. Algumas classes UVM já possuem portas instanciadas em seu código base, sendo possível utiliza-las por herança. Na codificação de um ambiente, as portas não definidas por padrão comumente usadas são:

- uvm_analysis_port
- uvm_analysis_export
- uvm_analysis_imp

uvm_analysis_port

A *analysis port* envia a transação de dentro da hierarquia de instância, desde a fonte da transação, até o nível mais externo.

uvm_analysis_export

A *analysis export* envia a transação do nível mais externo da hierarquia de instância até o destino da informação.

uvm_analysis_imp

A *analysis imp* é responsável por receber a informação por meio da função *write()*, implementada na descrição da biblioteca UVM.

Para fins didáticos, a imagem abaixo ilustra um exemplo de conexão das portas acima.

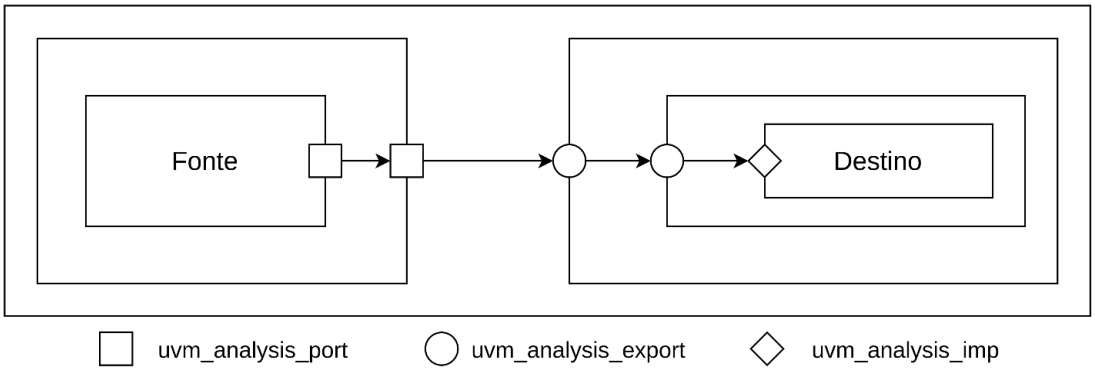

Figura 1 – Arquitetura tradicional UVM

O fluxo inicia quando um bloco, ao montar a transação, a escreve em sua *analysis port* e, por meio das portas conectadas, ativa a função *write()* no bloco de destino. Esta função será diretamente implementada pelo verificador como no seguinte exemplo:

Código da fonte

```
1  class some_class extends some_parent;
2
3    `uvm_component_utils(some_class)
4
5    //Será utilizada a transação definida na explicação da seção anterior.
6    some_transaction transaction_object;
7
8    //Instância da porta uvm_analysis_imp
9    uvm_analysis_port#(some_transaction) some_port;
10
11   .
12   .
13   .
14
15   task main_phase(uvm_phase phase);
16     forever
17     begin
18       @(some_event); //Linha de comando utilizada para travar a execução
     deste código até que o evento seja acionado.
19       begin_tr(transaction_object, "desired_label"); // Macro para que a
     transação seja mostrada na waveform.
20       transaction_object.some_field = some_value;
21       .
22       .//Pode-se atribuir quantos campos forem necessários.
23       .
24       end_tr(transaction_object, "desired_label");
25
```

```
26        some_port.write(transaction_object);
27      end
28    endtask : main_phase
29  endclass : some_class
```

Código do destino

```
1  class some_class extends some_parent;
2
3    `uvm_component_utils(some_class)
4
5    some_transaction transaction_object;
6
7    //Instância da porta uvm_analysis_imp
8    uvm_analysis_imp#(some_transaction, some_class) imp_port;
9
10   .
11   .
12   .
13
14 function void write (some_transaction t);
15      transaction_object.copy(t); //Caso deseje copiar as informações do
       pacote de dados sendo recebido para um objeto de transação interno.
16      .
17      . //Pode-se implementar a lógica necessária para o funcionamento do
18      . // bloco em questão, desde acrescimo de contadores até lógicas mais
       complexas.
19
20   endfunction: write
21 endclass : some_class
```

A conexão das portas ocorre sempre em blocos que contêm outros instanciados internamente. Por exemplo, caso um bloco_1 instancie dois blocos internos, bloco_2 e bloco_3, e também uma porta; para fazer a conexão deve-se seguir o modelo abaixo:

Código conexão de portas

```
1  class bloco_1 extends some_parent;
2
3    `uvm_component_utils(bloco_1)
4
5    some_transaction transaction_object;
6
```

```
7   //Instância da porta uvm_analysis_port
8   uvm_analysis_port#(some_transaction) some_port;
9
10  class_bloco_2 bloco_2;
11  class_bloco_3 bloco_3;
12
13  .
14  .
15  .
16
17  virtual function void connect_phase(uvm_phase phase);
18    super.connect_phase(phase);
19    bloco_2.nome_da_porta_bloco_2.connect(bloco_3.nome_da_porta_bloco_3);
20    bloco_2.nome_da_porta_bloco_2.connect(some_port);
21
22    .//Tendo em vista que não há limites para o número de portas que um
23    .// bloco pode instanciar, esta lista de conexões pode ser estendida
24    .// tanto quanto for necessário.
25
26  endfunction
27
28 endclass : bloco_1
```

Observações

· Para que seja permitida a conexão de duas portas, é necessário que ambas tenham sido criadas com a mesma transação em seus argumentos. Mesmo que sejam criadas duas transações idênticas, deve-se passar apenas uma delas.

· Não é necessário que os campos da transação estejam preenchidos para que a mesma possa ser enviada. Os campos com valores não atribuídos serão passados com todos os bits iguais a 'x'.

2.2 ARQUITETURA DO AMBIENTE

Para a metodologia UVM foi descrita uma arquitetura padrão, bem como foram desenvolvidos módulos bases para a mesma. Tais módulos estão listados abaixo:

- Monitor;
- Driver;
- Sequencer;
- Agent;

- Comparator;
- Coverage;
- Refmod;
- Scoreboard;
- Enviroment.

Monitor

Responsável por monitorar as interfaces que estão ligadas no DUT (*Design Under Test*) e enviar transações com as informações apropriadas para o Comparador, Cobertura e Refmod.

Driver

O *driver* se comunica diretamente com o bloco por meio das interfaces, sendo este responsável por enviar as informações respeitando os protocolos do DUT. Os dados a serem enviados são determinados por meio de transações recebidas do *sequencer*.

Sequencer

A partir de um arquivo de sequencia, outro classe padrão UVM, mas que não possui sua classe representada na arquitetura, o *sequencer* monta as transações com os dados a serem enviados ao DUT e as encaminha ao *driver*.

Agent

Responsável por encapsular e, se determinado e necessário, configurar os blocos sob sua instância. Por padrão, um *agent* encapsula o *driver*, monitor e *sequencer*.

Comparator

Por padrão, o comparador recebe transações em duas de suas portas já estabelecidas em sua classe base. Tais transações são provenientes do monitor e do *refmod*, estas serão comparadas para averiguar se o bloco verificado está funcionando de forma adequada.

Coverage

Na cobertura são estabelecidas condições para que a verificação seja dada por completa. Neste, são definidas quantas portas sejam necessárias para completar o plano de verificação.

Refmod

Também conhecido como *Golden Model*, no *refmod* são definidas pelo menos duas portas, uma para a transação que o monitor irá enviar, contendo os dados de entrada do DUT, e outra para que o *refmod* envie sua resposta ao comparador.

Scoreboard

Assim como o *agent*, o *scoreboard* é responsável por encapsular e configurar, se necessário, os blocos *comparator*, *coverage* e, em algumas arquiteturas, o *refmod*

Enviroment

No *enviroment* são instanciados todos os blocos que serão usados na verificação e, se necessário, configurar os mesmos.

A imagem abaixo ilustra a disposição dos blocos acima descritos.

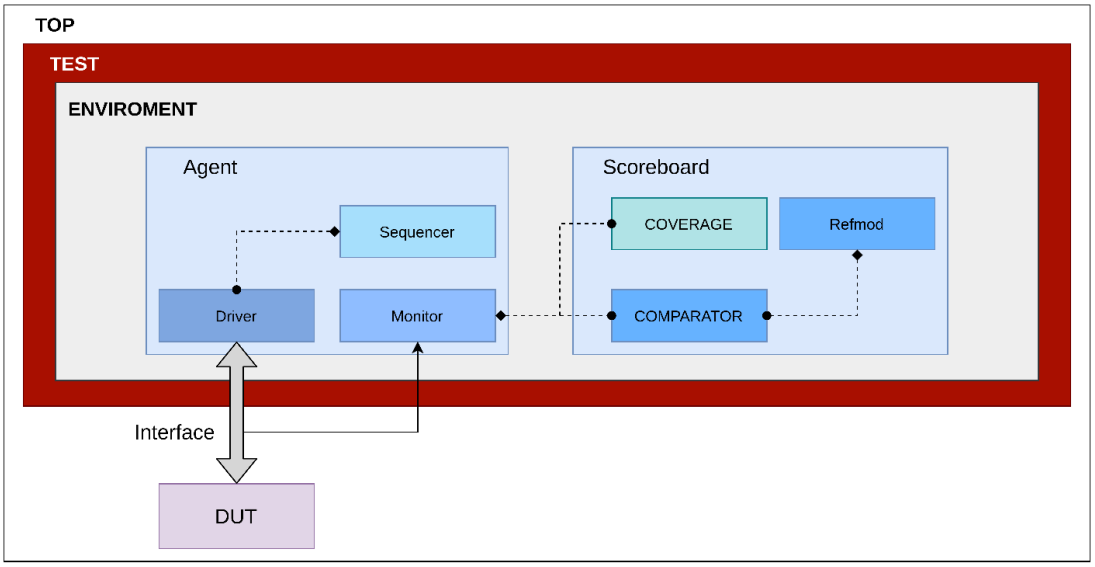

Figura 2 – Arquitetura tradicional UVM

No esquema acima ilustrado, os símbolos abaixo são:

♦ → Porta de saída da transação.

• → Porta de entrada da transação.

3 ESTUDO E DESENVOLVIMENTO RAL

3.1 INTRODUÇÃO

O RAL provê aos usuários uma biblioteca de classes que permitem a abstração de memórias e registradores internos ao DUT. Dependendo do objetivo da verificação, é possível que a mesma seja feita sem o uso do método em análise, porém, mesmo que o RAL não acrescente funcionalidade ao ambiente, este se torna uma ferramenta facilitadora de acesso aos registradores. Entretanto, como será detalhado mais abaixo, alguns objetivos só poderão ser atingidos mediante do uso do mesma.

Para melhor compreender este método, deve-se saber que este possui dois modos de acesso aos registradores, o *frontdoor* e o *backdoor*. No *frontdoor*, o método irá operar mediante a geração de transações para que o *driver* possa enviar, via interface, os dados. Já no modo *backdoor*, as ferramentas RAL irão alterar ou ler instantaneamente o valor de um registrador especificado, sem modificar os sinais da interface nem gerar transações.

3.2 APRESENTAÇÃO DAS CLASSES

As classes que serão utilizadas para o desenvolvimento do método são:

- uvm_reg_field,
- uvm_reg,
- uvm_reg_block,
- uvm_reg_adapter;

uvm_reg_field

É a menor unidade de um registrador. Com a instância desta, pode-se definir quantos campos forem necessário e configurar os mesmos conforme a documentação. Por exemplo, um registrador de controle pode possuir um determinado número de bits para configurar um modo de operação, ou a ativação de determinado bloco, estes bits específicos para algo são os campos do registrador. Tal especificação é necessário para facilitar na leitura do registrador e para determinar se determinado campo pode ser escrito ou não. Para fins didáticos, esta classe será ilustrada conforme abaixo.

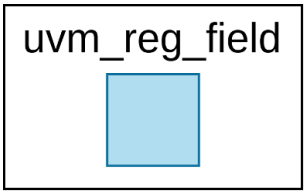

Figura 3 – Ilustração do uvm_reg_field

uvm_reg

É a menor unidade de um bloco de registradores, nesta classe deve-se instanciar a classe anteriormente descrita e definir os campos de um tipo de registrador. A imagem abaixo ilustra a composição do mesmo.

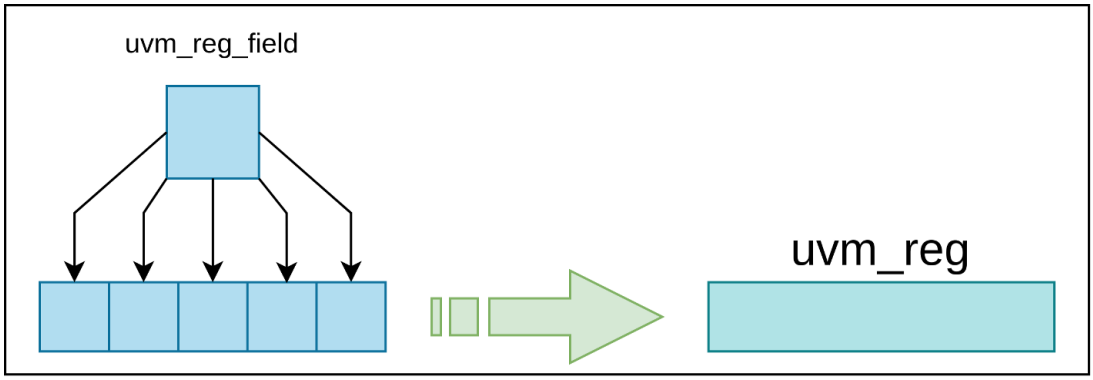

Figura 4 – Ilustração do uvm_reg

Sendo assim, caso em um mesmo bloco de registradores existam dois ou mais tipos de registradores, cada um deles deve possuir sua própria classe definida conforme código abaixo.

Código uvm_reg

```
1 class some_reg_ral extends  uvm_reg ; //Definir classe para posterior instâ
    ncia.
2  `uvm_object_utils( some_reg_ral )
3
4  rand uvm_reg_field some_field_1; //Definir o primeiro campo para o
    registrador.
5  rand uvm_reg_field some_field_2; //Definir o segundo campo para o
    registrador.
6
7  function new( string name = "some_reg_ral" );
8      super.new( .name( name ), .n_bits( x ), .has_coverage(
   UVM_NO_COVERAGE ) );
```

```
9         //No .n_bits() define-se o número de bits do registrador. Ou seja,
    o somatório
10        // do número de bits dos campos deve ser 'x'.
11    endfunction: new
12
13    virtual function void build();
14      some_field_1  = uvm_reg_field::type_id::create( "some_field_1" );
15        some_field_1.configure(
16                        .parent                 ( this ),
17                        .size                   ( y    ),
18                        .lsb_pos                ( 0    ),
19                        .access                 ( "RW" ),
20                        .volatile               ( 0    ),
21                        .reset                  ( 0    ),
22                        .has_reset              ( 1    ),
23                        .is_rand                ( 1    ),
24                        .individually_accessible( 1    ) );
25
26      some_field_2  = uvm_reg_field::type_id::create( "some_field_2" );
27        some_field_2.configure(
28                        .parent                 ( this ),
29                        .size                   ( z    ),//Tal que z + y = x
30                        .lsb_pos                ( y    ),
31                        .access                 ( "RW" ),
32                        .volatile               ( 0    ),
33                        .reset                  ( 0    ),
34                        .has_reset              ( 1    ),
35                        .is_rand                ( 1    ),
36                        .individually_accessible( 1    ) );
37    endfunction: build
38  endclass : some_reg_ral
```

Explanação dos campos

.size() - Determina o número de bits do campo.

.lsb_pos() - Define a posição inicial do campo em questão. Como no código anterior, o primeiro campo vai do bit 0 ao y-1, já o segundo campo vai do bit y ao (y+z-1).

.access() - Configura a permissão do campo. (Explicação abaixo)

.volatile() - Determina se o campo é escrito apenas via interface ou se o mesmo pode ter seu valor alterado pelo próprio bloco. Zero, se o único meio de modificação for por interface; um, se o bloco puder mudar o valor deste.

.reset() - Valor que o campo irá quando o registrador entrar em *reset*.

.has_reset() - Se o bloco possuir *reset*, colocar valor um.

.is_rand() - Se zero, este campo não poderá ser randomizado pelo método "randomize()".

.individually_accessible() - Caso, no barramento que acessa o bloco, seja possível determinar um *byte enable* e este campo seja o único a ocupar um ou mais bytes, então este campo possui a capacidade de ser lido ou escrito individualmente, sem modificar os outros bytes do registrador. Portanto, deve-se colocar valor 1, caso as condições determinadas sejam cumpridas.

Opções para o access

- RO - Apenas leitura/ Escrita não modifica o registrador.
- RW - Leitura e Escrita (Read and Write)
- RC - Registrador zera após leitura/ Escrita não modifica o registrador.
- RS - Após leitura, todos os bits vão para '1'/ Escrita não modifica o registrador.
- WRC - Escrita modifica o registrador/ Registrador zera após leitura.
- WRS - Escrita modifica o registrador/ Registrador modificado para todos bits iguais a '1' após leitura.
- WC - Após escrita, zera o registrador/ Leitura efetuada normalmente.
- WS - Após escrita, modifica os bits do registrador para '1'/ Leitura efetuada normalmente.
- WSRC - Após escrita, modifica os bits do registrador para '1'/ Após leitura, zera o registrador.

- WCRS - Após escrita, zera o registrador/ Após leitura, modifica os bits do registrador para '1'.
- W1C - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for '1', esta irá zerar, se o valor for zero, será mantido o bit anterior/ Leitura efetuada normalmente.
- W1S - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for '1', esta irá para '1', se o valor for zero, será mantido o bit anterior/ Leitura efetuada normalmente.
- W1T - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for '1', esta terá seu valor alternado (toggles), se o valor for zero, será mantido o bit anterior/ Leitura efetuada normalmente.
- W0C - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for zero, esta irá zerar, se o valor for '1', será mantido o bit anterior/ Leitura efetuada normalmente.
- W0S - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for zero, esta irá para '1', se o valor for '1', será mantido o bit anterior/ Leitura efetuada normalmente.
- W0T - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for zero, esta terá seu valor alternado (toggles), se o valor for '1', será mantido o bit anterior/ Leitura efetuada normalmente.
- W1SRC - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for '1', esta irá para '1', se o valor for zero, será mantido o bit anterior/ Após leitura, zera o registrador.
- W1CRS - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for '1', esta irá para '1', se o valor for zero, será mantido o bit anterior/ Após leitura, modifica os bits do registrador para '1'.
- W0SRC - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for zero, esta irá para '1', se o valor for '1', será mantido o bit anterior/ Após leitura, zera o registrador.
- W0CRS - Na escrita, é feita uma operação bit a bit. Nesta, se o valor escrito em uma posição for zero, esta irá para zero, se o valor for '1', será mantido o bit anterior/ Após leitura, modifica os bits do registrador para '1'.
- WO - Apenas a escrita é realizada/ Leitura retorna erro.

- WOC - Após escrita, zera o registrador/ Leitura retorna erro.
- WOS - Após escrita, modifica os bits do registrador para '1'/ Leitura retorna erro.
- W1 - Escreve apenas uma ÚNICA vez após um reset/ Leitura efetuada normalmente.
- WO1 - Escreve apenas uma ÚNICA vez após um reset/ Leitura retorna erro.

uvm_reg_block

Na hierarquia de construção da abstração, esta classe é a última a ser criada. Nesta, deve-se instanciar as classes de registradores codificadas conforme o ítem anterior. A composição desta se encontra ilustrada abaixo:

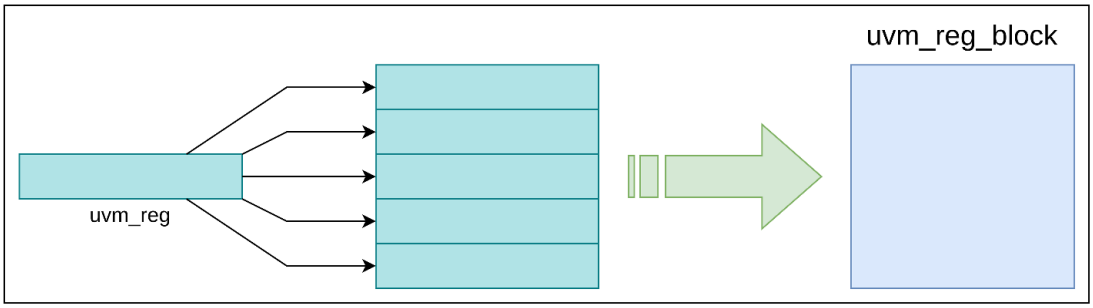

Figura 5 – Ilustração do uvm_reg_block

Caso no bloco haja mais de um tipos de registrador, um para status, um para controle e outros para memória, por exemplo, então é possível a instância dos mesmos. Entretanto, todos os registradores devem possuir o mesmo tamanho de bits, para que os mesmos possam ser ordenados de forma coerente. O código do mesmo está definido, de forma genérica, abaixo.

Código uvm_reg_block

```
1  class some_block extends  uvm_reg_block ;
2    `uvm_object_utils( some_block )
3
4    rand some_reg_ral_1  some_reg;
5    rand some_reg_ral_2  other_reg[1:0];
6    string s;
7    // uvm_reg_map     reg_map;
8
9
10   function new( string name = "some_block" );
11       super.new( .name( name ), .has_coverage( UVM_NO_COVERAGE ) );
12     endfunction
```

```
13
14      virtual function void build();
15
16      default_map = create_map( .name( "default_map" ), .base_addr( 0 ),
17                                .n_bytes( x ), .endian( UVM_LITTLE_ENDIAN ));
18                                //x -> Número de bytes de cada registrador.
19
20
21      some_reg = some_reg_ral_1::type_id::create("some_reg");
22      some_reg.configure( .blk_parent( this ));
23      some_reg.build();
24      default_map.add_reg( .rg( some_reg ), .offset( 0 ), .rights( "RW" ) );
25
26      other_reg[0] = some_reg_ral_2::type_id::create("other_reg[0]");
27      other_reg[0].configure( .blk_parent( this ));
28      other_reg[0].build();
29      default_map.add_reg( .rg( other_reg[0] ), .offset( x ), .rights( "RW" )
        );
30
31      other_reg[1] = some_reg_ral_2::type_id::create("other_reg[1]");
32      other_reg[1].configure( .blk_parent( this ));
33      other_reg[1].build();
34      default_map.add_reg( .rg( other_reg[1] ), .offset( 2*x ), .rights( "RW"
        ) );
35
36      lock_model(); //Fim do mapeamento de endereço.
37    endfunction: build
38
39  endclass : some_block
```

Explanação dos campos

Ao criar um mapa de memória, conforme linha 16 do código acima, deve-se definir alguns parâmetros, sendo estes:

.name() - Declara ao construtor da classe o nome deste objeto.

.base_addr() - Define em que byte este mapa de memória está localizado. Tal é importante pois pode-se desejar criar mais de um mapa de memória para um bloco, ou mesmo este bloco não possuir seu endereço inicial no valor zero.

.n_bytes() - Configura o número de bytes dos registradores que serão alocados no mapa.

.endian() - Define a disposição dos bits, podendo ser *big endian* ou *little endian*.

Ao adicionar um registrador ao mapa, conforme linhas 24, 29 e 34, os parâmetros serão:

.rg() - Passa como argumento o registrador que será adicionado ao mapa.

.offset() - Neste, é determinado qual o endereço, em relação ao inicial do mapa, que o registrador adicionado está localizado. Tendo em vista que o *offset* é dado em bytes, o valor de seu argumento será múltiplo do tamanho de cada registrador, pois não pode haver sobreposição destes. Por isso, como pode-se observar nas linhas indicadas, o *offset* foi dado por 0, x e 2*x.

.rigths() - Campo para configuração das permissões de acesso ao registrador adicionado. As opções válidas, conforme a documentação (MUZZI, 2021), são:

RW - Leitura e escrita.

RO - Apenas leitura.

WO - Apenas escrita.

uvm_reg_adapter

Nesta classe será determinado como as transações que, no modo *frontdoor*, serão passadas ao modulo *driver* do ambiente. A implementação da mesma segue o padrão abaixo:

Código uvm_reg_adapter

```
1 class some_adapter extends  uvm_reg_adapter;
```

```
2   `uvm_object_utils (some_adapter)
3
4   function new (string name = "some_adapter");
5     super.new(name);
6   endfunction : new
7
8   function uvm_sequence_item reg2bus (const ref uvm_reg_bus_op rw);
9     some_transaction tx;
10    tx = some_transaction::type_id::create("tx");
11
12    tx.op = (rw.kind == UVM_READ) ? UVM_READ : UVM_WRITE; //Saber qual
      operação está ocorrendo.
13    tx.add = rw.addr;//Válido para ambos as possibilidades.
14    if (tx.op == UVM_WRITE)
15      begin
16        tx.data_in = rw.data;//data_in -> Dado a ser escrito.
17        tx.write_en = 1'b1;//Sinal de controle de handshake para escrita.
18        tx.read_en = 1'b0;//Sinal de controle de handshake para leitura.
19      end
20    else
21      begin
22        tx.data_in = '0;
23        tx.write_en = 1'b0;//Sinal de controle de handshake para escrita.
24        tx.read_en = 1'b1;//Sinal de controle de handshake para leitura.
25      end
26
27    return tx;
28  endfunction : reg2bus
29
30  function void bus2reg (uvm_sequence_item bus_item, ref uvm_reg_bus_op rw)
      ;
31    some_transaction tx;
32
33    assert($cast(tx, bus_item))
34      else `uvm_fatal("","Erro no cast em some_adapter.sv -> bus2reg()")
35
36    rw.kind = (tx.op == UVM_WRITE) ? UVM_WRITE : UVM_READ; //Se 1,
      UVM_WRITE. Se 0, UVM_READ.
37    rw.addr = tx.add;
38    rw.data = tx.data_in;
39
40    rw.status = UVM_IS_OK;
41  endfunction : bus2reg
42
43 endclass : some_adapter
```

> ***Observação***
>
> · O código acima foi desenvolvido tendo em mente um banco de registradores que possua sinais de *handshake* para controle de escrita e leitura.

A instância de um objeto da classe *uvm_reg_block*, fornecerá ao verificador acesso às ferramentas do UVM RAL, tanto *frontdoor* quanto *backdoor*. Dentre estas estão os comandos '*read()*', '*write()*', '*backdoor_read()*' e '*backdoor_write()*'. Para que os mesmos possam ser executados corretamente, é necessário que haja a configuração de seus mecanismos.

3.3 IMPLEMENTANDO ACESSO *FRONTDOOR*

Para que as funções *read()* e *write()* gerem as transações da forma correta, deve-se, por meio da função 'set_sequencer()' do mapa de memória do *reg_block*, passar um objeto *reg_adapter*, conforme código abaixo:

Código para usar o *reg_adapter*

```
1  class some_class extends some_parent;
2    `uvm_component_utils(some_class)
3
4    some_block reg_block;//Classe definida no código anterior.
5    some_adapter reg_adapter;
6    .
7    .
8    .
9
10   function new(string name, uvm_component parent = null);
11     super.new(name, parent);
12   endfunction : new
13
14   virtual function void build_phase(uvm_phase phase);
15     super.build_phase(phase);
16     reg_block = some_block::type_id::create("reg_block", this);
17     reg_block.build();
18     reg_adapter = some_adapter::type_id::create("reg_adapter",,
       get_full_name());
19     .
20     .
21     .
22
23   endfunction : build_phase
```

```
24
25   virtual function void connect_phase(uvm_phase phase);
26     super.connect_phase(phase);
27     reg_block.default_map.set_sequencer(.sequencer(CAMINHO_PARA_O_SEQUENCER
       .sequencer), .adapter(reg_adapter));
28     .
29     .
30     .
31
32   endfunction : connect_phase
33 endclass : some_class
```

Este código pode ser implementado na classe do *enviroment*, sendo este o topo, apenas será necessário, no *.sequencer()* (linha 27), passar a hierarquia até o sequenciador. Tendo em vista que o *reg_adapter* foi corretamente instanciado e passado conforme detalhado, o arquivo de sequência pose ser escrito tal qual o modelo abaixo.

Código sequência para *frontdoor*

```
1  class some_reg_sequence extends uvm_reg_sequence;
2    `uvm_object_param_utils(some_reg_sequence)
3
4    some_block reg_model;
5
6    function new(string name = "");
7      super.new(name);
8    endfunction
9
10   task body();
11     uvm_status_e status;
12     uvm_reg_data_t rdata;
13
14     write_reg(reg_model.some_reg, status, VALOR_A_SER_ESCRITO);
15     read_reg(reg_model.some_reg, status, rdata);
16   endtask
17 endclass : some_reg_sequence
```

Tendo em vista que a sequência foi implementada corretamente, ao ser dado o *start* na mesma, o *driver* irá enviar os dados conforme determinado nas transações preenchidas na função reg2bus do *adapter*. Com isso, o acesso via *frontdoor* estará concluído.

Este método pode ser utilizado na simplificação do arquivo de sequência, como demonstrado no (SHIMIZU, 2021). Neste site, no tópico 'Sequence without Register Abstraction', o autor compara a codificação da sequência com e sem o método RAL.

Outra aplicação para este tipo de acesso é a previsão do valor que será lido, para a implementação desta é necessário a configuração e instância de um bloco derivado do *uvm_reg_predictor*. Ao poder prever o valor que será lido a partir do modelo de memória, pode-se enviar este como referência ao comparador. Tal não será explanado no presente documento pois não foi possível criar, com êxito, um ambiente de verificação com esta funcionalidade em execução.

3.4 IMPLEMENTANDO ACESSO *BACKDOOR*

Visando a viabilização da execução das funções *backdoor_read()* e *backdoor_write()*, é necessário que sejam realizadas algumas mudanças nas configurações do *reg_block*. Tais mudanças são essenciais pois, nestas, será definido o caminho para os registradores no HDL.

O modelo de código para o *reg_block* com implementação destas mudanças segue abaixo:

Código reg_block para acesso *backdoor*

```
1  class some_reg_block extends  uvm_reg_block ;
2    `uvm_object_utils( some_reg_block )
3
4    rand some_reg   reg_;
5
6    function new( string name = "some_reg_block" );
7        super.new( .name( name ), .has_coverage( UVM_NO_COVERAGE ) );
8      endfunction
9
10     virtual function void build();
11       add_hdl_path  (.path("
     CAMINHO_PARA_O_LOCAL_NO_QUAL_OS_REGISTRADORES_ESTO"), .kind("RTL"));
12         //Nesta função é adicionado o caminho para
13         // o local no qual os registradores estão instanciados.
14         // Um exemplo é: "top.dut".
15
16     default_map = create_map( .name( "default_map" ), .base_addr( 0 ),
17                          .n_bytes( 1 ), .endian( UVM_LITTLE_ENDIAN ));
18
19     reg_ = some_reg::type_id::create("reg_");
20     reg_.configure( .blk_parent( this ), .hdl_path( "
     NOME_DO_REGISTRADOR_NO_RTL" ) );
21       //No .hdl_path é determinado efetivamente qual o registrador
22       // ao qual o modelo se refere.
23       // Um exemplo é: "status".
```

```
24
25      reg_.build();
26      default_map.add_reg( .rg( reg_ ), .offset( 0 ), .rights( "RW" ) );
27
28      lock_model(); // finalize the address mapping
29    endfunction: build
30
31  endclass : some_reg_block
```

As mudanças, em relação ao que já havia sido explicado, são:

> *Mudanças para backdoor*
>
> · Adição da função *add_hdl_path*, que adiciona caminhos, desde o topo, até o local de instância dos registradores que se deseja manipular.
>
> · Na função '.configure()', linha 20, determina-se o valor do .hdl_path( ). Neste, deve-se passar o nome do registrador que está instanciado no RTL.

Ao implementar estas mudanças, será possível utilizar as funções de escrita e leitura via *backdoor*. Estas recebem como argumento um objeto da classe uvm_reg_item, do qual será utilizado apenas o campo '*value[0]*'. Na leitura, o campo em questão irá armazenar o valor lido; na escrita, este deverá conter o valor a ser escrito no registrador.

Segue abaixo um exemplo de execução do mesmo:

Código execução do *backdoor*

```
1  class some_module extends  some_parent ;
2    `uvm_object_utils( some_module )
3
4    rand some_block   reg_block;
5    uvm_reg_item rw;
6
7    function new( string name = "some_module" );
8        super.new( .name( name ), .has_coverage( UVM_NO_COVERAGE ) );
9      rw = uvm_reg_item::type_id::create("reg_item",,get_full_name());
10     endfunction
11
12   virtual function void build_phase(uvm_phase phase);
13     super.build_phase(phase);
14       reg_block = block_reg_ral::type_id::create("reg_block", this);
15       reg_block.build();
16   endfunction
17
```

```
18
19   task run_phase(uvm_phase phase);
20     rw.value[0]       = VALOR_A_SER_ESCRITO;
21       reg_block.reg_.backdoor_write(rw);
22
23       reg_block.reg_.backdoor_read(rw);
24     valor_lido = rw.value[0];
25   endtask
26 endclass : some_module
```

Após a execução do comando de escrita, sem que os sinais da interface variem, o registrador referente ao 'reg_' no DUT irá assumir o valor enviado. Caso esta função seja executada sem que haja o cuidado de estudar previamente a função dos registradores no fluxo de funcionamento do bloco, pode haver uma falha no fluxo de verificação do mesmo.

4 APLICAÇÕES PROPOSTAS

Após o estudo e implementação do RAL, foram testados alguns casos nos quais este método pode auxiliar não só na detecção, como também no reporte de bugs detectados. Este capítulo será dividido entre as aplicações de *frontdoor* e de *backdoor*.

4.1 *FRONTDOOR*

Primeira aplicação:

Por meio da implementação do *frontdoor*, como já citado, é possível otimizar a codificação de sequências com a execução dos comandos *write()* e *read()*. Com esta otimização, o código não se tornará apenas mais fácil de ler, como também será mais intuitiva a operação desejada. Além disto, ao descrever corretamente um bloco de memória, não será mais necessária a preocupação com as permissões de acesso ao mesmo, podendo assim o verificador focar seus esforços no desenvolvimento de outras tarefas.

Segunda aplicação:

Ao obter êxito na implementação do *uvm_predictor*, será possível fornecer um acesso constante aos valores previstos do banco de memória ao modelo de referência. Com isto, o modelo possuirá uma maior confiabilidade em seus cálculos, tendo em vista que um banco de memória com muitos tipos de registradores, cada qual com suas próprias permissões, se torna complexo de implementar, mesmo em alto nível. Sendo assim, ao designar a tarefa de armazenar os valores do banco de registrador do DUT ao RAL, no modelo será simplesmente implementada funções de cálculo.

4.2 *BACKDOOR*

Primeira aplicação:

Com a utilização da função *backdoor_read()*, é possível manter o monitoramento constante dos registradores internos do DUT sem que seja necessário alterar os valores dos sinais da interface. Sendo assim, sempre que algum registrador do RTL mudar de valor, o ambiente poderá se moldar e verificar se a mudança deveria ter ocorrido ou não. Podendo assim elevar o nível de verificação ao ponto de ser capaz de analisar o comportamento da máquina de estados do bloco.

Segunda aplicação:

Após identificar um *bug* no bloco, o verificador reporta o mesmo ao responsável

pelo design deste. Entretanto, normalmente, o verificador, em seu relatório, envia a semente que foi utilizada na geração randômica das transações e em que momento o erro ocorreu, porém, em alguns blocos, é necessário que haja muitas transações específicas até que o erro ocorra.

Entretanto, com a utilização do RAL *backdoor*, é possível ler exatamente os valores de cada registrador do bloco, além da transação passada ao *driver*, no momento em que o erro ocorreu. Estando de posse dessas informações, é necessário apenas que seja executada a função *backdoor_write()* em cada registrador, cada qual com seu respectivo valor, para que o bloco assuma o estado no qual se encontrava quando foi detectado a divergência funcional entre o bloco e a especificação, e enviar ao *driver* a transação salva. Com isto, não só será mais rápido, consumirá menos memória e capacidade de processamento do computador para chegar ao momento do erro, quanto o designer também terá mais informações sobre o que estava acontecendo dentro do bloco no momento da falha.

5 CONSIDERAÇÕES FINAIS

Neste documento, foi discorrido sobre o estudo realizado no método de abstração de registradores, RAL. Foi abordado desde a formação estrutural da metodologia UVM até a implementação do deste método.

Além do embasamento teórico, foram apresentados códigos para que se tenha uma base da qual partir ao se desejar implementar o conteúdo neste explanado.

Diante disto, o objetivo do trabalho foi alcançado, tendo seu fechamento dado por sugestões de aplicação no qual será explorado o potencial da ferramenta.

Com relação a trabalhos futuros, neste foi mencionado o estudo e implementação da classe *uvm_reg_predictor* para que o RAL possa oferecer ao usuário a possibilidade de tornar o ambiente de verificação mais versátil.

REFERÊNCIAS

MUZZI studio. *uvm_reg_map.svh*. 2021. Disponível em: <http://www.studio-muzzi.com/project/docs/UVMdocs_smu/uvm-1.1d/uvm___reg___map_8svh_source.html#l00150>. Acesso em: 20 maio 2021. 26

SHIMIZU, K. *UVM Tutorial for Candy Lovers – 9. Register Abstraction*. 2021. Disponível em: <http://cluelogic.com/2012/10/uvm-tutorial-for-candy-lovers-register-abstraction/>. Acesso em: 24 maio 2021. 29