Henry de Lima Carvalho

Ambiente de verificação funcional de um IP-core em UVM

Campina Grande, Brasil

Dezembro de 2018

Henry de Lima Carvalho

Ambiente de verificação funcional de um IP-core em UVM

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 Júnior, D.Sc.

Campina Grande, Brasil

Dezembro de 2018

Henry de Lima Carvalho

Ambiente de verificação funcional de um IP-core em UVM

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: 20/12/2018

Gutemberg Gonçalves dos Santos Júnior, D.Sc.
Orientador

Marcos Ricardo Alcântara Morais, D.Sc.
Convidado

Campina Grande, Brasil
Dezembro de 2018

Dedico este trabalho à minha querida vó Lilita, que hoje repousa no plano espiritual.

Agradecimentos

Agradeço primordialmente à meus pais, Lana e Guilherme, por me prestarem todo apoio quando precisei em toda minha vida, nunca medindo esforços para possibilitar que hoje eu possa estar vivendo e desfrutando essa fase de minha vida.

Agradeço também à minha amada namorada Gabrielle por seu incessável apoio, carinho, amizade e amor. Por ser o ponto de paz em meio ao caos, sempre acreditando em mim e me empurrando para frente, enfatizando sempre minha capacidade quando muitas vezes eu mesmo estava desacreditado.

Ao meu grupo de amigos e colegas de curso que puderam tornar meus dias na universidade mais agradáveis, proporcionando bons momentos de lazer e descontração no ambiente caótico da universidade, compartilhando de seus conhecimentos em épocas de intenso estudo e dividindo angústias nos momentos mais difíceis na graduação.

Aos bons professores que tive oportunidade de conhecer, em especial aos professores Gutemberg, Marcos Morais e Elmar pelas oportunidades de capacitação que influenciaram diretamente e me direcionaram para um caminho profissional promissor.

"A riqueza é produto da capacidade do homem de pensar."

Ayn Rand

Resumo

A verificação funcional é uma etapa fundamental no projeto e concepção de um bloco de circuito integrado digital dedicado, também conhecido como IP-Core, impedindo que falhas e erros de implementação cheguem ao *design* físico e atinjam o consuminor final, comprometendo o correto funcionamento do componente. UVM é uma metodologia composta de uma biblioteca de classes em SystemVerilog que permite a modelagem de um ambiente de testes a nível de transações possibilitando o reuso em diversos IP's. Por se tratar de uma metodologia bastante consolidada e amplamente utilizada tanto no âmbito profissional como acadêmico, foram descritos de forma sistemática os procedimentos de implementação de um ambiente voltado para a verificação funcional de um IP genérico em UVM, demonstrando sua funcionalidade.

Palavras-chaves: Verificação funcional; UVM; Testbench, IP-Core.

Abstract

Functional Verification is a major stage in dedicated digital integrated circuits design, also known as IP-Core, avoiding implementation mistakes and flaws to reach the physical design and the final client as well, compromising the correct functionalities of the component. UVM is a methodology composed by a library of SystemVerilog classes that allow a transaction level modeling environment test, making possible it's reuse in many IP's. Once being a well-established and largely used methodology such in industrial design as academic environments, it was described in a systematic way a verification environment implementation procedures in UVM for a generic IP, demonstrating it's fuctionality.

Key-words: Functional Verification; UVM, Testbench, IP-Core.

Lista de ilustrações

Figura 1 – Fluxo básico do desenvolvimento de um componente de hardware . . . 2
Figura 2 – Exemplo de um *testbench* básico em UVM 10
Figura 3 – Hierarquia das principais classes do UVM 11
Figura 4 – Testbench padrão UVM 18
Figura 5 – Esquema de testbench com utilização de apenas um agent distribuindo transações bilateralmente 29
Figura 6 – Modelo de conexões do scoreboard 35
Figura 7 – Erro emitido devido à um *illegal bin* 40
Figura 8 – Relatório do UVM sinalizando o fim da simulação bem sucedida 46
Figura 9 – Relatório de cobertura indicando porcentagens de cobertura 47

Lista de abreviaturas e siglas

DPI	Direct Programming Interface
DUT	Design Under Test
HDL	Hardware Description Language
IP	Intellectual Property
PEM	Projeto de Excelência em Microeletrônica
SoC	System on Chip
SVA	System Verilog Assertions
TLM	Transaction Level Modeling
UVM	Universal Verification Methodology

Sumário

1 INTRODUÇÃO 1
1.1 IP-Core 2
1.2 Fluxo de Hardware 2
1.3 Linguagem de Descrição de Hardware - HDL 3
1.3.1 SystemVerilog 4
1.3.1.1 SystemVerilog DPI 4
1.4 Register Transfer Level - RTL 5

2 VERIFICAÇÃO FUNCIONAL 6
2.1 Plano de verificação 7
2.2 Testbench 7
2.3 Transaction Level Modeling - TLM 8
2.4 Cobertura 9

3 METODOLOGIA UVM 10
3.1 Hierarquia de classes 11
3.2 Construtores 12
3.3 UVM Phases 12
3.4 Factory 13
3.4.1 Registro 13
3.4.2 Construção 14
3.4.3 Substituição 14
3.5 Conexões 14
3.5.1 Analysis port 15
3.5.2 Analysis export 16
3.5.3 Analysis imp 16
3.6 UVM configuration database 16

4 AMBIENTE DE VERIFICAÇÃO UVM 18
4.1 Interface 19
4.2 Transações 20
4.2.1 Implementação UVM 20
4.3 Sequência 22
4.3.1 Implementação UVM 22
4.4 Sequencer 23
4.5 Driver 24

4.5.1 Implementação UVM 24
4.6 Monitor 26
4.6.1 Implementação UVM 26
4.7 Agent 28
4.7.1 Implementação UVM 29
4.8 Modelo de Referência 30
4.8.1 Implementação UVM 31
4.9 Comparador 34
4.10 Scoreboard 34
4.11 Cobertura 35
4.11.1 Covergroups 36
4.11.2 Coverpoints 37
4.11.3 Bins 38
4.11.3.1 Bins ignorados 39
4.11.3.2 Bins ilegais 40
4.11.4 Cobertura cruzada 40
4.12 Environment 41
4.12.1 Implementação UVM 41
4.13 Teste 42
4.13.1 Implementação UVM 43
4.14 Design Under Test - DUT 43
4.15 Topo 44

5 RESULTADOS 46

6 CONCLUSÃO 48

Referências 49

1 Introdução

A tecnologia de semicondutores alimenta a maioria dos mais novos produtos e inovações atuais. O rápido avanço da tecnologia gera um constate crescimento na complexidade, que por outro lado possibilita produtos incríveis de última geração. Esses chips estão presentes em vários aspectos de nossa vida cotidiana, desde um smartphone, uma transação financeira com cartão de crédito, até um servidor de última geração que computa nossas compras online. Expectativas sobre esses chips crescem em um ritmo equivalente, apesar da complexidade adicional. Por exemplo, consumidores não esperam que os chips que monitoram os procedimentos de segurança em nossos carros falhem durante a vida útil do veículo, ou que seja aceitável não ter acesso à sua conta bancária por um erro de servidor. Desta forma, desenvolver esses circuitos integrados de forma correta têm sido uma façanha da engenharia.

A implementação de um IP-Core consiste na criação de um componente de hardware que desempenhe uma determinada funcionalidade. Esse componente deve permitir a integração em um ambiente onde ele possa interagir com os demais componentes para formar um sistema SoC (*System on Chip*). Para que essa integração seja possível, é necessário que esse projeto tenha a menor quantidade de falhas possível. A confiabilidade vai depender de como é realizado o processo de geração e verificação do IP.

Verificação de circuitos digitais têm se tornado comum uma vez que o número e complexidade dos circuitos vem aumentando. Indústrias comumente estimam que a verificação funcional toma mais da metade do esforço total do projeto, contando mão de obra, cronograma e custos, sendo o processo de verificação o maior obstáculo para completar um novo projeto. O processo de verificação funcional é complexo, demorado e algumas vezes mal compreendido. Como resultado, um bom esforço na verificação representa um dos maiores riscos para o sucesso na conclusão de um projeto.

Existem vários tipos diferentes de verificação usados no projeto e concepção de um novo sistema tais quais verificação funcional, verificação temporal, testes de validação, entre outros. Cada um possui uma finalidade diferente e são usados em etapas e tarefas distintas do projeto. Diferentemente de um teste de validação que foca em cada parte física já concebida, a verificação funcional foca no design antes de cada parte física ser construída. A verificação funcional visa determinar se o design operará conforme sua especificação. Isso requer a definição de uma especificação que indique a operação correta ou como o dispositivo deve funcionar.

1.1 IP-Core

Em microeletrônica, um IP-Core (*Intelectual Property Core*), ou bloco IP, é uma unidade lógica, célula, ou circuito integrado que é propriedade intelectual de alguma parte. Geralmente chamamos de IP um *design* de um módulo que desempenha alguma funcionalidade dedicada que posteriormente poderá ser integrado a um sistema mais complexo como um SoC.

1.2 Fluxo de Hardware

O projeto e concepção de um elemento de hardware, seja ele um IP dedicado ou até mesmo um SoC com arquitetura mais complexa, passa por um fluxo de etapas que envolvem desde o planejamento junto ao cliente, até a análise de amostras físicas. A Figura 1 ilustra de forma básica esse fluxo.

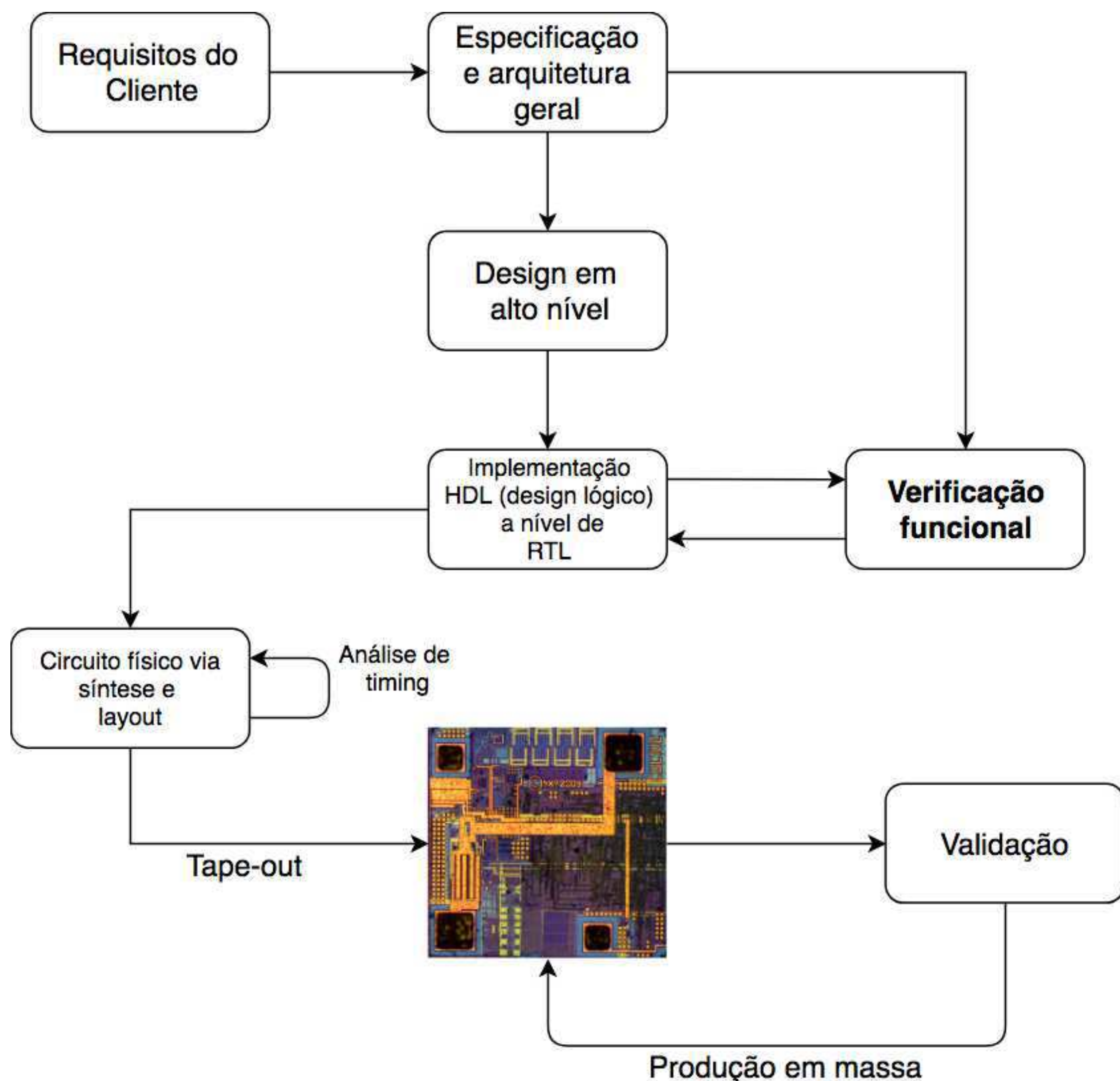


Figura 1 – Fluxo básico do desenvolvimento de um componente de hardware

O primeiro passo referente ao nascimento de um novo projeto de um chip é o alinhamento junto ao cliente para definição dos requisitos e funcionalidades por ele desempenhadas. A empresa propõe uma solução digital para uma necessidade de outra empresa que vá utilizar de seu chip em seus projetos e dessa forma, juntos, eles definem as diretrizes e detalhes de sua funcionalidade.

Uma vez definido com o(s) cliente(s) as funcionalidades do projeto, deve ser elaborado um documento conhecido como 'especificação' que documenta detalhadamente as características, funcionalidades e arquitetura do projeto a ser elaborado. Este documento servirá de base para os projetistas e deve ser seguido à risca na implementação.

Com a especificação em mãos, a implementação em HDL é feita obedecendo o que foi documentado, gerando o que se chama de RTL (*Register Transfer Level*) com a implementação lógica e comportamental do circuito.

A verificação funcional atua a princípio sobre o RTL implementado, verificando e testando através de simulações se o mesmo teve suas funcionalidades corretamente implementadas de acordo com o documento de especificação previamente definido. O processo de verificação funcional requer muito tempo e recurso, estima-se que cerca de 70%, e deve garantir o correto funcionamento do projeto.

Após a garantia de correto funcionamento por parte do RTL, é feita a síntese lógica onde o RTL é implementado em níveis de portas lógicas gerando a *netlist*. Em cima da *netlist* é realizada a análise temporal onde são considerados os atrasos inerentes às portas lógicas e *flip-flops* para assegurar que o circuito ainda funciona a nível de netlist. Com tudo certificado, o projeto vai para o processo de *backend* e *layout* para assim ser mandado para a *foundry*.

A versão preliminar do chip mandada para fabricação chama-se *tape-out*. Nessa etapa, são fabricadas apenas algumas amostras do chip, que retornam para os engenheiros de validação realizarem testes e medições no intuito de detectar algum erro de fabricação ou até mesmo erros que passaram despercebidos pela verificação. Após o aval da validação, o chip pode ser produzido em massa.

1.3 Linguagem de Descrição de Hardware - HDL

Em engenharia computacional, HDL *Hardware Description Language* é uma linguagem computacional especializada em descrever a estrutura e comportamento de circuitos eletrônicos, e mais comumente, circuitos lógicos digitais.

Uma linguagem de descrição de hardware possibilita uma descrição precisa e formal de um circuito eletrônico que permite a análise automatizada e simulação de um circuito eletrônico. Ela também permite a síntese de uma descrição HDL para uma *netlist* (uma

especificação dos componentes eletrônicos físicos e como eles estão interconectados).

Uma linguagem de descrição de hardware se assemelha muito com uma linguagem de programação como C, uma descrição textual formado de expressões, declarações e estruturas de controle. Uma importante diferença entre a maioria das linguagens de programação e linguagens de descrição de hardware é que HDLs incluem explicitamente noções de tempo.

HLDs foram criados para implementar abstrações de RTL (*Register Transfer Logic*), um modelo de fluxo de dados e timing de um circuito. Existem duas principais linguagens de descrição de hardware, VHDL e Verilog, além de SystemVerilog que deriva do Verilog.

1.3.1 SystemVerilog

SystemVerilog, padronizada como IEEE 1800, é uma linguagem de descrição de hardware usada para modelar, projetar, simular, testar e implementar sistemas eletrônicos. SystemVerilog é baseado em Verilog e algumas extensões, e desde 2008 Verilog é parte do mesmo padrão IEEE. É comumente usado na indústria de *design* de semicondutores como uma evolução do Verilog.

A gama de funcionalidades de SystemVerilog pode ser dividida em dois papéis distintos:

1. SystemVerilog para *design* RTL é uma extensão de Verilog-2005; todas os recursos dessa linguagem estão disponíveis em SystemVerilog. Portanto, Verilog é uma subcategoria de SystemVerilog

2. SystemVerilog para verificação usa extensivas técnicas de programação orientada a objeto. Essas construções geralmente não são sintetizáveis.

1.3.1.1 SystemVerilog DPI

SystemVerilog DPI (*Direct Programming Interface*) é uma interface que pode ser usada para integrar SystemVerilog com outras linguagens. Essas outras linguagens podem ser C, C++, SystemC, entre outras. DPIs consistem em duas camadas: Uma camada SystemVerilog e uma camada da outra linguagem. Ambas camadas são isoladas uma da outra. Qualquer que seja a linguagem usada como linguagem externa é transparente e irrelevante para o lado SystemVerilog da interface. Nem o compilador de SystemVerilog nem da outra linguagem precisa analisar o código fonte da outra linguagem. Diferentes linguagens de programação podem ser usadas e suportadas com a mesma camada SystemVerilog.

1.4 Register Transfer Level - RTL

No design de circuitos digitais, *register transfer level* (RTL) é uma abstração do design que modela um circuito síncrono digital em termos de fluxo de sinais digitais (dados) entre registradores de hardware e operações lógicas realizadas com estes sinais. A abstração em RTL é usada em linguagens de descrição de hardware como Verilog, SystemVerilog e VHDL para criar uma representação do circuito em alto nível.

Ao implementar circuitos integrados digitais em alguma linguagem de descrição de hardware, os *designs* geralmente são desenvolvidos em um nível de abstração mais alto do que o nível de transistores ou lógica de gate level. Em HDL o projetista declara os registradores (que grosseiramente correspondem a variáveis em linguagens de programação computacionais), e descreve a lógica combinacional usando construtores que são familiares das linguagens de programação, tais como *if-else*, e operações aritiméticas por exemplo. Esse nível é chamado de *register transfer level*, ou RTL. O termo se dá devido ao fato de que RTL foca em descrever o fluxo de sinais entre registradores.

2 Verificação Funcional

Verificação funcional refere-se à tarefa de verificar se um *design* lógico desempenha suas funcionalidades de acordo com sua especificação. Em linhas gerais, a verificação funcional garante que o design faz o que deve fazer da forma como foi proposto a ser feito. Essa é uma tarefa complexa, e que geralmente toma a maior parte do tempo e esforço na maioria dos projetos de design de sistemas eletrônicos. A verificação funcional é parte de um abrangente processo de verificação de um circuito integrado, que, além da verificação funcional, considera também aspectos não funcionais, tais como *timing*, *layout* e *power*.

O *design* de um chip pode consistir de centenas de milhares de linhas de código. O trabalho do engenheiro de verificação é buscar problemas na implementação do hardware, falhas em sinais ou *bugs*, quando o *design* não funciona conforme a especificação. O engenheiro de verificação identifica e expõe essas falhas executando simulações complexas no *design*.

O processo de verificação funcional pode ser bastante trabalhoso devido à vasta possibilidade de casos de teste que podem existis até em um simples *design*. Entretanto, a verificação pode ser abordada por vários métodos que em conjunto, podem ser de bastante auxílio para um completo e eficaz trabalho de verificação:

- Simulações lógicas que simulam a lógica antes do chip ser fabricado.
- Verificação formal procura provar matematicamente que certos requisitos do projeto sejam atendidos, ou que certos comportamentos indesejáveis não possam ocorrer.
- O processo de Validação testa amostras de chip fabricadas em pequena escala, de forma a garantir que a amostra física está a funcionar conforme especificado em projeto e que não há nenhuma falha no processo de fabricação, ou algum erro funcional que tenha escapado aos olhos da verificação.

Verificação baseada em simulações é amplamente utilizada para simular o circuito. Estímulos são fornecidos para exercitar cada linha no código HDL. Um *testbench* é construído para verificar funcionalmente o design propiciando cenários significantes para checar que dada uma determinada entrada, o *design* desempenha sua funcionalidade especificada. Um ambiente de simulação é tipicamente composto por alguns tipos de componentes:

- O gerador de estímulos gera vetores de entrada que são usados para identificar anomalias que existam entre a especificação e a implementação. Geradores de estímulos

modernos criam estímulos randômicos direcionados que são direcionados para verificar partes randômicas do *design*. A randomização é importante para alcançar uma alta distribuição através da enorme gama de possibilidades de estímulos de entrada. Esse mecanismo permite que o gerador de estímulos crie entradas que revelem *bugs* que não foram diretamente investigados, ou seja, possibilidades que não foram pensadas pelo verificador.

- Os *drivers* traduzem os estímulos produzidos pelo gerador em efetivos sinais de entrada do *design a ser verificado*. Os geradores criam entradas em um alto nível de abstração chamados de transações. Os *drivers* convertem essas entradas em reais entradas tais quais definidas na especificação da interface do *design*.
- O monitor converte o estado do *design* e suas saídas para um nível de abstração de transações, para que possa ser armazenado em um *'scoreboard'* depois.
- O *checker* valida os conteúdos do scoreboard. Isso é feito geralmente confrontando a saída do design com a saída gerada a partir das mesmas entradas, mas produzidas por uma modelagem de alta confiabilidade que execute as mesmas funcionalidades que o *design* deve cumprir.

Diferentes métricas de cobertura são definidas para assegurar que o design foi adequadamente exercitado. Essas incluem cobertura funcional, que indicam se todas as funcionalidades do design foram exercitadas, e *statement coverage*, que aponta se cada linha do HDL foi exercitada.

2.1 Plano de verificação

Um plano de verificação corresponde ao documento que descreve quais funcionalidades do *design* devem ser exercitadas para que seja garantida uma cobertura de verificação aceitável e que deve ser seguido por todo engenheiro de verificação como requisito mínimo para o processo de verificação no fluxo de projeto.

O processo de planejamento de verificação envolve identificar *testcases* que atingem funções específicas do *design* e descrever um conjunto específico de estímulos para serem aplicados no *design*. Essa abordagem na verificação é semelhante à especificação completa de um bloco que uma vez montada, cumprirá todos os seus requerimentos.

2.2 Testbench

Um *testbench* é um ambiente estruturado e implementado para a realização da verificação do DUT (*Design Under Test*). Ele oferece toda infra-estrutura para que es-

tímulos sejam levados ao *design* e posteriormente comparados para assim constatar seu correto funcionamento.

Sua implementação deve ser de preferência em alto nível de abstração, ou seja, a nível de transações. Nesse nível, não há a atenção em detalhes de protocolo de comunicação, que só acontece ao passar para nível de sinais compreendidos pelo *design*. O foco maior se tratando do *testbench* refere-se à comunicação interna entre os blocos e componentes, bem como a transferência de transações entre eles.

Uma transação é o elemento básico da troca de informações entre dois blocos funcionais, sendo um procedimento com início e fim caracterizando-se pelo grupo de dados e instruções imprescindíveis para a realização de determinada operação.

A comunicação entre o *testbench* e o DUT é feito com a conversão dos dados a nível de transação para o nível de sinais compatíveis com as entradas do *design*. É necessária a implementação de um módulo no *testbench* que seja capaz de fazer essa conversão de dados, tarefa geralmente atribuída ao *driver*.

Problemas de verificação algumas vezes podem ser atribuídos à uma má implementação de *testbench*. Portanto o máximo de cautela deve ser tomada em sua implementação, pois possíveis erros nessse ambiente podem gerar ambiguidades no resultado que não necessariamente correspondam a erros de *design*.

O objetivo desse trabalho é a implementação de um *testbench* modelado em UVM para verificação funcional de um IP. UVM é uma metodologia baseada em uma vasta biblioteca de classes em SystemVerilog para implementação de *testbenches* em alto nível. Mais detalhes sobre UVM serão expostos mais adiante.

2.3 Transaction Level Modeling - TLM

TLM (*Transaction-level modeling*) é uma abordagem de alto nível para modelar sistemas digitais onde detalhes da comunicação entre módulos são separadas dos detalhes da implementação das unidades funcionais ou de comunicação da arquitetura. Mecanismos de comunicação tais como barramentos ou FIFO's são modeladas como canais, e podem ser apresentadas a módulos usando classes de interface em SystemC. Transações solicitam sua ocorrência chamando funções das interfaces desses modelos de canais, que encapsulam os detalhes em baixo nível da troca de informações.

No nível de transações, a ênfase é mais na funcionalidade da transferência de dados e menos na implementação em si, isto é, no protocolo usado para transferência de dados. Essa aproximação deixa mais fácil para os *designers* a nível de sistema testarem, por exemplo, diferentes arquiteturas de barramento sem precisar buscar modelos que interajam com qualquer desses barramentos.

TLM também permite maior velocidade de simulação do que interfaces baseadas em pinos e aparece como uma solução promissora em cima do RTL no fluxo de design do SoC.

2.4 Cobertura

Cobertura Funcional é um método usado para mensurar o quanto das funcionalidades do *design* foram de fato exercitadas e testadas. A cobertura é essencial para monitorar a qualidade do teste e indicar onde os mesmos estão deixando de atuar, ajudando o verificador a direcionar seus testes de forma que possa cobrir o máximo possível de funcionalidades do *design*.

Por muitas vezes um teste pode não conseguir cobrir completamente as funcionalidades do dispositivo testado, isso pode ser atribuído a um ou mais fatores:

- Tempo insuficiente para que os estímulos exercitem todas as funcionalidade;.
- Geração de estímulos limitados e insuficientes para todas as funcionalidades possíveis;
- Limitações de *testbench* que não permitem que todas as funcionalidades sejam testadas.

Esses fatores devem ser identificados e corrigidos para que haja a garantia que foi feita uma verificação completa e eficaz, pois uma funcionalidade não testada pode conter algum erro que passará despercebido e consequentemente alcançará o usuário final.

3 Metodologia UVM

UVM (*Universal Verification Methodology*) é uma metodologia para construção de ambientes de verificação baseados em classes em SystemVerilog, tirando vantagens das técnicas de programação orientada a objetos para facilitar o reuso do código. Reuso é o ponto chave do UVM, usar UVM não faz sentido se não tiver como objetivo o reuso completo dos códigos para verificação.

Os blocos construídos em um ambiente de verificação UVM são objetos, isto é, instâncias de classes, ao contrário de módulos, processos e funções familiares aos usuários de Verilog e VHDL. O sentido de usar-se objetos é que eles podem ser substituídos dinamicamente, provendo uma enorme flexibilidade quando se trata de reusar componentes de verificação e testes sem alteração do código fonte original.

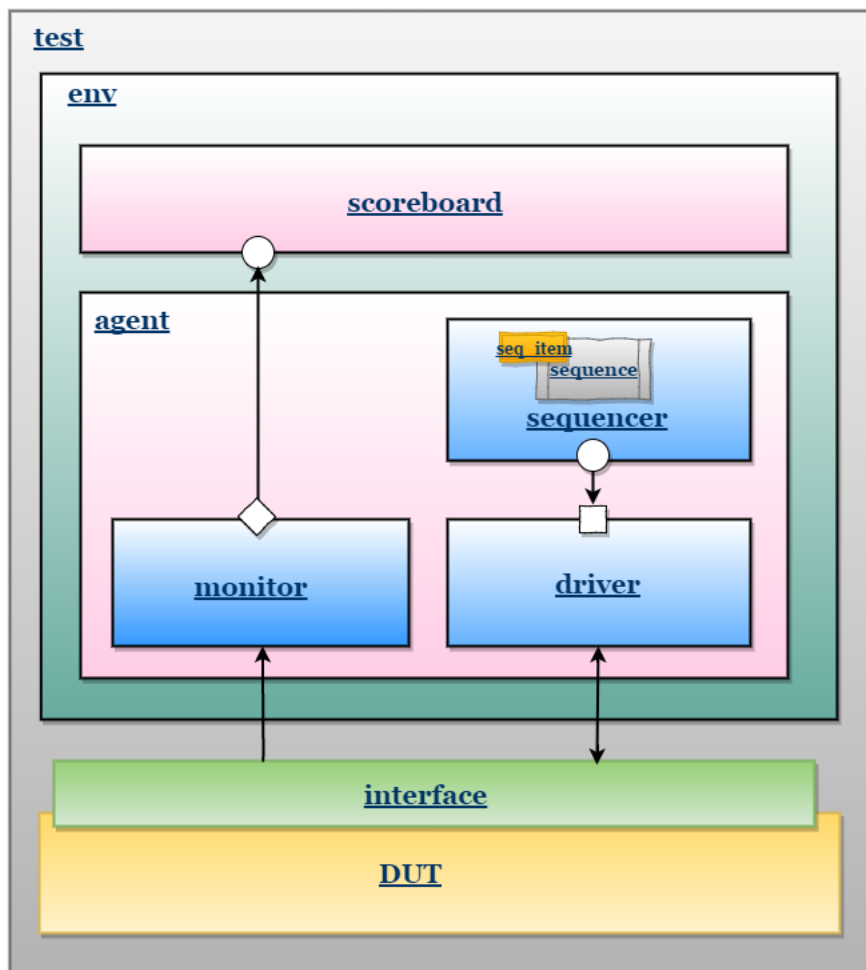


Figura 2 – Exemplo de um *testbench* básico em UVM

O *testbench* UVM trabalha a nível de transações, e cada componente implementado

como objeto de classe tem seu papel no fluxo dessas transações. A Figura 2 representa um *testbench* em UVM na sua forma mais básica, onde as setas representam o caminho das transações entre os componentes representados por blocos. O capítulo 4 aborda a implementação detalhada do *testbench* e deixa explícito o papel e funcionamento de cada parte que compõe este conjunto.

UVM foi desenvolvido pela Acellera, em colaboração com Cadence, Mentor Graphics e Synopsys. A versão 1.0, e subsequentemente a versão 1.1 foram lançadas em 2011, embora uma versão anterior (UVM 1.0EA) esteve disponível em 2010. UVM é implementado inteiramente em SystemVerilog, portanto funciona em qualquer simulador que suporte completamente o padrão IEEE 1800-2009.

3.1 Hierarquia de classes

As principais classes do UVM e suas hierarquias podem ser vistas na Figura 3. A classe `uvm_object` é a classe base de todas as outras classe em UVM.

Figura 3 – Hierarquia das principais classes do UVM

Transações definidas pelo usuário devem ser derivadas de `uvm_sequence_item`. A classe `uvm_component`, que é derivada de `uvm_report_object`, é usada como classe pai para componentes de verificação definidos pelo usuário e possui uma task `run_phase` que

automaticamente invoca o início de uma simulação. Classes básicas para componentes de verificação comums tais como *environments*, *agentes*, *drivers* e *monitors* também são fornecidas.

3.2 Construtores

Todas as classes `uvm_component` precisam de um método `new()` com dois argumentos: `name` e `parent`. Exemplo:

```
10 function new (string name, uvm_component parent)
11   super.new(name, parente);
12 endfunction: new
```

Os construtores de componentes UVM podem fazer mais do que chamar `super.new()`, mas eles precisam ao menos chamar `super.new()`.

3.3 UVM Phases

A classe `uvm_phase` é a base para todas as fases de simulação UVM. Um objeto de `uvm_phase` é passado como argumento em todo método de fase UVM de cada componente, que podem então ser usados para uma variedade de fins. Comumente o objeto de fase é usado para levantar e abaixar objeções para prevenir a mudança para a próxima fase UVM. O objeto de fase também pode ser usado para pular entre fases na execução.

Cada componente UVM possui três métodos de fase como parte de sua herança. O UVM constrói e executa testbenches ao chamar esses métodos de fase em todos os componentes na ordem estabelecida. Seus componentes são definidos sobrecarregando os métodos de fase e delegando ao UVM a responsabilidade de chama-los na ordem certa.

Apesar de não ser necessário, quando for sobrecarregar os métodos de fase, o primeiro passo deve ser chamar a função `super.<phase_method>`. Isso assegura que será respeitado qualquer trabalho feito pelos desenvolvedores de UVM.

Todos os métodos de fase recebem um argumento. Esse argumento deve ser um objeto de fase, ou seja, do tipo `uvm_phase`, e deve se chamar `phase`. Há muitas fases UVM, mas apenas as três principais serão abordadas.

Os métodos devem ser chamados na seguinte ordem:

- `function void build_phase(uvm_phase phase)` – O UVM constrói sua hierarquia de testbench do topo para baixo usando esse método. Os `uvm_components` devem ser instanciados nesse método. Se houver uma tentativa de instanciação de `uvm_components` em outro método, um erro fatal UVM é emitido.

- `function void connect_phase(uvm_phase phase)` – A fase de conexão (*connect*) conecta os componentes.
- `task run_phase(uvm_phase phase)` – A simulação de fato acontece nessa fase UVM e é executado em paralelo com outros `run_phase()` no *testbench*.

3.4 Factory

A UVM *factory* é disponibilizada como um mecanismo configurável para criar objetos a partir de classes derivadas de `uvm_object` (sequências e transações) e `uvm_component` (componentes do *testbech*).

A vantagem de usar a *factory* ao invés de construtores (`new`) é que os tipos de classes que são usados para construir o ambiente de teste são determinados no tempo de execução (durante a *build phase*). Isso torna possível que sejam escritos testes que codificam o ambiente de teste sem a necessidade de editar o código do ambiente de teste diretamente.

Apenas classes derivadas de `uvm_object` e `uvm_component` suportam isso. Existem três passos a serem seguidos ao usar a uvm *factory*.

1. Registro
2. Construção
3. Substituição

3.4.1 Registro

Ao definir uma classe, seu tipo deve ser registrada na *factory*. Para tornar esse trabalho mais fácil, UVM possui as macros pré-definidas:

```
`uvm_component_utils(class_type_name)
`uvm_component_param_utils(class_type_name #(params))
`uvm_object_utils(class_type_name)
`uvm_object_param_utils(class_type_name #(params))
```

Exemplo:

```
class packet extends uvm_object;
`uvm_object_utils(packet)
endclass

class packet #(type T=int, int mode=0) extends uvm_object;
```

```
`uvm_object_param_utils(packet #(T,mode))
endclass

class driver extends uvm_component;
`uvm_component_utils(driver)
endclass

class monitor #(type T=int , int mode=0) extends uvm_component;
`uvm_component_param_utils(driver#(T,mode))
endclass
```

Em todas as classes é possível ver a macro `` `uvm_component_utils(type_name) ``

3.4.2 Construção

Para construir um componente ou objeto, o método `create()` deve ser usado. Essa função constrói o objeto apropriado baseado na classe predomiante. Então quando for construir um componente ou objeto, não se deve usar o construtor *new()*.

Sintaxe:

```
static function T create(string name,
uvm_component parent,
string context = " ")
```

A função `create()` retorna uma instância do tipo de componente `T`.

Exemplo:

```
class_type object_name;

object_name = class_type::type_id::create("object_name",this);
```

Para classes do tipo `uvm_object` não há a necessidade do segundo argumento nos parênteses. Em todo componente que instancia algum outro objeto ou componente, o método `create()` será encontrado na `build_phase`.

3.4.3 Substituição

Se necessário, o usuário pode substituir a classe ou objeto registrado. O usuário pode fazer a substituição baseando-se na *string* de nome ou no tipo da classe. Neste projeto, não houve a necessidade de realizar nenhuma substituição neste sentido.

3.5 Conexões

O UVM fornece uma solução universal para comunicação *interthread* que esconde detalhes. A solução se dá em duas partes básicas:

Portas Objetos que são instanciados nos `uvm_components` para permitir que a tarefa em `run_phase()` se comunique com outras *threads*. Portas emissoras (*put ports*) são usadas para enviar dados e portas receptoras (*get ports*) para receber de outras *threads*.

FIFOs TLM Objetos que conectam uma porta emissora com uma porta receptora. Embora a FIFO TLM possa mover transações, elas podem mover qualquer tipo de dado. FIFOs TLM guardam apenas um elemento. Apesar de ser uma capacidade bastante limitada para uma FIFO, é bem útil para a comunicação *interthread*.

Além disso o UVM fornece duas classes que tornam mais fácil implementar a observabilidade da atividade de outros componentes do *testbench*.

3.5.1 Analysis port

É frequentemente necessário para algumas partes de um *testbench* poderem observar a atividade de outros componentes do *testbench*. *Analysis ports* fornecem um mecanismo consistente para tais observações. Uma *analysis port* é declarada da seguinte forma:

```
uvm_analysis_port#(transaction_type) port_name;
```

No desenvolvimento de algum componente, usa-se uma *analysis port* para tornar dados disponíveis para outras partes do *testbench*. O método `write` da *analysis port* não bloqueia, e portanto, não pode interferir com o fluxo processual do componente.

Qualquer componente que deseje tornar dados de transações visíveis para outras partes do testbench deve possuir um membro do tipo `uvm_analysis_port`, parametrizado para o tipo da transação. Esta *analysis port* deve ser construída durante a execução do método `build_phase` no componente.

Quando um componente tem uma transação e deseja publicá-la, deve chamar o método `write` da *analysis port* com a variável de transação como argumento. Esse método é uma função, portanto, não bloqueante. Ele tem o efeito de chamar o método `write` em todos os *subscribers* conectados.

Monitores designados para observar transações em uma interface seguramente possuem uma *analysis port* através da qual podem ser entregadas transações observadas. Outros componentes podem opcionalmente ter *analysis ports* para expor dados de transações que eles manipulem ou gerem, para que outras partes do *testbench* observem esses dados.

3.5.2 Analysis export

Componentes que trabalham com transações geralmente deixam essas transações disponíveis para outras partes do *testbench* através de *analysis ports*. Um componente de monitoramento ou análise que deseja observar essas transações devem designar sua *analysis port*. Isso é feito conectando uma *analysis export* em cada destinatário na *analysis port* do gerador. A *analysis export* fornece indiretamente o método de escrita chamada pela *anaylis port*. Não há limite para o número de *analysis exports* que podem ser conectados à uma *analysis port*. A declaração de uma *analysis export* é feita da seguinte maneira:

```
uvm_analysis_export#(transaction_type) port_name;
```

A *analysis export* exporta para a *analysis port* do componente conectado a ela, métodos que são definidos em sua hierarquia, ou que foram recebidos de outro componente através de uma *analysis imp*.

3.5.3 Analysis imp

É definida no componente que implementa a função `write`, e que pela qual, componentes se conectarão através de *analysis ports* ou *analysis exports*, e que poderão chamar a função `write` para poder disponibilizar os dados transacionais. Ou seja, a *analysis imp* deixa disponíveis as funções definidas no componente para que outros possam transferir dados. Sua declaração é da seguinte forma:

```
uvm_analysis_imp#(transaction_type, component_name) name;
```

3.6 UVM configuration database

UVM possui uma base de dados interna em que podemos armazenar valores sob determinado nome que podem ser acessadas posteriormente por outro componente do *testbench*. A classe `uvm_config_db` fornece uma conveniente interface no topo de `uvm_resource_db` para simplificar a interface básica usada para instâncias de `uvm_component`.

Tais configurações de base de dados permitem que sejam armazenados diferentes configurações sob diversos nomes que podem potencialmente configurar componentes de *testbench* quando requisitados sem modificar o código do *testbench* em si.

set() Usa-se esta função estática da classe `uvm_config_db` para passar uma variável para a base de dados. No exemplo abaixo, a função `set()` torna a interface virtual sob o nome `my_vif` disponível para todos os componentes abaixo de `uvm_test_top`.

```
uvm_config_db #(virtual my_vif)::set (null, "uvm_test_top.*", "
my_vif", my_vif);
```

get() Usa-se esta função para pegar o valor de uma determinada variável da base de dados. No exemplo abaixo, a função *get()* pega a interface virtual sob o nome de *my_vif* para a interface virtual local.

```
uvm_config_db #(virtual my_if)::get (this, "", "my_vif", my_if);
```

4 Ambiente de verificação UVM

Os blocos que constroem um ambiente de verificação UVM são objetos, ou seja, instâncias de classes, em oposição aos módulos, processos, e funções familiares aos usuários de Verilog ou VHDL. O intuito de usar objetos é que eles podem dar uma enorme flexibilidade no que diz respeito a reutilização de componentes e testes de verificação sem manipulação do código fonte original. É possível utilizar um IP existente e substituir subcomponentes, alterar a sequência de transações gerada, ou extender seu comportamento sem alterações no código fonte e sem a necessidade do autor do IP antecipar essas mudanças de nenhuma forma, além de ter seguido boas práticas de descrição de hardware.

Para a criação do ambiente de verificação funcional do IP em questão é necessária a implementação e configuração de um ambiente que gere sequências de transações, converta essas transações para estímulos compreendidos pelo DUT, monitore os estímulos e compare os resultados com um modelo de referência confiável implementado e integrado ao ambiente de forma que certifique o correto funcionamento do DUT.

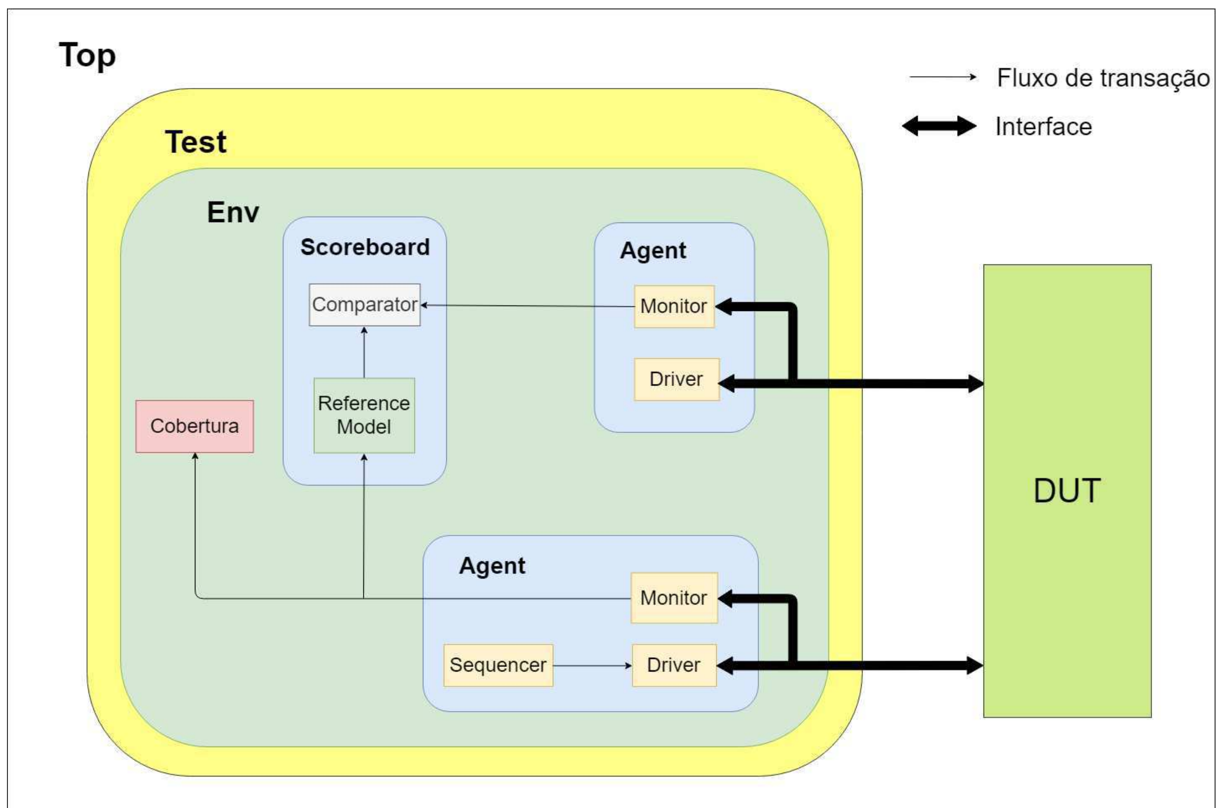


Figura 4 – Testbench padrão UVM

No ambiente projetado, estímulos são gerados por um componente chamado *sequencer* que gera um item de sequência chamado de transação. Essas transações passam por um *driver* que converte os estímulos a níveis de transação em sinais compreendidos

pelo IP, que se apresenta como o *Design Under Test* (DUT), em nosso ambiente de verificação, alimentando-o através de uma interface. Um monitor faz o acompanhamento desses sinais enviados ao IP, ao mesmo tempo em que coleta tais sinais e os enviam para um modelo de referência que nada mais é do que um modelo em alto nível do DUT, que implementa suas funcionalidades com um alto grau de confiança. Uma vez obtendo os resultados de cada lado, as informações convergirão para um comparador que ao confrontar as informações, indicará se o IP testado está executando suas funcionalidades corretamente. Paralelo a isso, um módulo de cobertura analisa os estímulos gerados e avalia o quão bem estimulado o DUT está sendo no sentido de cobrir todos ou a grande maioria dos possíveis cenários de uma aplicação real.

Isso tudo é modelado e montado com o auxílio do UVM. A Figura 4 ilustra a estrutura de implementação do ambiente de verificação construído nesse trabalho. Cada componente necessário na construção deste esquemático será definido e detalhado, tanto conceitualmente, como sua forma de implementação no contexto em questão.

4.1 Interface

Interfaces são uma construção em SystemVerilog criadas especialmente para encapsular a comunicação entre blocos, permitindo um refinamento sutil de um nível de sistema abstrato através de passos sucessivos até níveis mais baixos de RTL e estruturais de design. Interfaces também facilitam o reuso do design. Algumas vantagens podem ser destacadas no uso de interfaces:

- Elas encapsulam a conectividade: uma interface pode ser passada como um único item através de uma porta, portanto substituindo um grupo de nomes por apenas um. Isto reduz o montante de código necessário para modelar as conexões das portas e aumenta a facilidade de manutenção.
- Elas encapsulam funcionalidades, isoladas dos módulos que são conectados via interface. Assim, o nível de abstração e a granularidade do protocolo de comunicação pode ser refinado toalmente independentemente dos módulos.
- Elas podem conter parâmetros, *constraints*, variáveis, funções e *tasks*, processos e *assignments* contínuos, úteis tanto para modelagens em nível de sistema, quanto para aplicações em *testbench*.
- Flexibilidade: Uma interface pode ser parametrizada da mesma forma que um módulo.

Um exemplo de uma interface simples pode ser vista abaixo. Uma implementação de uma pequena interface que parametriza o tamanho do dado e do comando, e recebe como entrada valores de *clock* e *reset*.

```
1 interface my_if #(int DATA_SIZE = 32, int CMD_SIZE = 4) (
2   input logic ACLK,
3   input logic ARESETn);
4
5   logic [ADDR_SIZE-1:0] data;
6   logic [CMD_SIZE] cmd;
7
8 endinterface : my_if
```

O uso de interfaces no projeto aprecem indicadas na Figura 4.

4.2 Transações

A classe virtual que é usada como classe base para transações em um ambiente UVM é `uvm_transaction`. Ela é derivada de `uvm_object`, e adiciona funções para gerenciamento de transações e oferece suporte para gravação de transações. `uvm_transaction` é a classe base para `uvm_sequence_item`. Todas as classes de transações que são criadas e definidas pelo usuário devem derivar de `uvm_sequence_item`, não de `uvm_transaction`.

Transações são usadas como estímulos em um ambiente de teste UVM. O começo e fim de uma transação pode ser gravada e armazenada na transação usando as funções `begin_tr` e `end_tr` respectivamente. Por padrão, o tempo gravado é o tempo de simulação atual. Se um tempo diferente for necessário, pode ser especificado como argumento da função.

As transações transitam no interior do ambiente UVM e compõem os valores e sinais necessários para que o IP que está sendo testado seja estimulado corretamente. A transação implementará todos os sinais presentes na interface do IP. As variáveis ou sinais que compõem essas transações regem a comunicação entre o IP e os demais componentes do SoC. A comunicação respeitando o protocolo garante que haja a sincronização na propagação das informações e garante o sucesso em operações regidas pelo mestre, tais como iniciar um processo em algum bloco, ou até mesmo uma simples escrita ou leitura na memória.

4.2.1 Implementação UVM

O processo de criação de uma transação em um projeto de ambiente de verificação em UVM se faz como se segue.

```
1 class my_transaction extends uvm_sequence_item;
```

```
2
3     rand bit [CMD_SIZE-1:0] cmd;
4     rand bit [DATA_SIZE-1:0] data;
5
6     `uvm_object_utils_begin(my_transaction)
7     `uvm_field_int(cmd, UVM_ALL_ON)
8     `uvm_field_int(data, UVM_ALL_ON)
9     `uvm_object_utils_end
10
11    constraint my_constraint {
12        data != 0x00000000;
13        cmd  != 0xF;
14        cmd  != 0xE;
15    }
16
17    function new(string name = "");
18        super.new(name);
19    endfunction
20
21 endclass: my_transaction
```

Na linha 1 se tem a criação da classe de transação definida pelo usuário que extenda da classe `uvm_sequence_item`. Uma transação nesse contexto não pode derivar da classe `uvm_transaction`, pois a mesma não pode ser usada como um item de sequência. Posteriormente será explicado do que se trata uma sequência.

É interessante tentar minimizar o número de classes de transações distintas. Usar a mesma transação para o *driver* e monitor de um agente. Manter várias transações geralmente requer bem mais esforço do que manter um único padrão de transação.

Nas linhas 3 e 4, é usado o qualificador `rand` para definir as variáveis que precisam ser geradas aleatoriamente em algum momento. A transação pode incluir protocolos e variáveis que precisarão ser aleatórias, porém também pode conter dados que o usuário não vai desejar tal aleatoriedade. Para isso pode ser definida uma *constraint*, que define um comportamento específico para algum membro dessa classe ou alguma limitação dentro de uma aleatorização como é feito nas linhas de 11 a 14.

Após todos os membros e variáveis, definir um construtor que inclua um único argumento em forma de string com o valor padrão de uma string vazia e depois uma chamada à `super.new`. O construtor deve seguir o padrão definido na seção 3.2 tal como nas linhas 17 a 19.

A transação deve ser registrada na factory usando a macro `` `uvm_object_utils `` como primeira linha na classe. Se houver uso das `field_macros`, a transação deve ser registrada depois das declarações de qualquer variável usando a macro `` `uvm_object_utils_begin ``.

4.3 Sequência

O teste cria novas transações e as alimentam ao *testbench*. Isso significa que o testador está fazendo duas coisas: escolher a ordem das transações e alimentá-las ao *testbench*. Isto provoca um problema de reuso. Um projetista futuro poderia pensar que o teste é a solução perfeita para o problema de criar novas transações, mas não poderá usá-la porque o teste tem o efeito colateral de determinar o estímulo.

Podemos sobrecarregar o tipo de transação para controlar a entropia dos dados, mas devemos sobrecarregar a classe testadora inteira para mudar a quantidade de transações e a forma que elas são enviadas. Bons *testbenches* separam a ordem das transações (o estímulo de teste) da estrutura do *testbench*. A estrutura deveria se manter inalterada não importando a ordem das transações.

As sequências UVM separam o estímulo da estrutura do *testbench*. Elas nos permitem criar uma estrutura de *testbench* e então percorrer dados diferentes através dela.

4.3.1 Implementação UVM

A implementação da sequência no *testbench* UVM acontece da seguinte forma:

```
1  class my_sequence extends uvm_sequence #(my_transaction);
2      `uvm_object_utils(my_sequence)
3
4      logic [7:0] aux; //Auxiliar variable just in case
5
6      function new(string name =    my_sequence    );
7          super.new();
8      endfunction: new
9
10     task body();
11         req = my_transaction::type_id::create(   req   );
12         start_item(req);
13         assert(req.randomize());
14         finish_item(req);
15     endtask
16
17 endclass: my_sequence
```

Na linha 1, a criação de uma classe de sequência definida pelo usuário derivando da classe `uvm_sequence`, parametrizando o tipo da transação a ser gerada pela sequência.

Na linha 2 é feito o registro da classe de sequência na *factory* usando a macro `` `uvm_object_utils `` como a primeira linha na classe. O uso de macros de campo não é recomendado, mas se for preferido o uso, a classe deve ser registrada imediatamente depois da declaração de qualquer membro desta classe usando a macro `` `uvm_object_utils_begin ``

como visto na seção 4.2.1.

Após a macro de registro, declarar as variáveis auxiliares se forem necessárias.

Após as variáveis (se houver), definir um construtor que inclua uma única string de argumento com um valor predefinido de uma string vazia, uma chamada para `super.new`. O construtor deve seguir o mesmo padrão de construtor de classes já conhecido e explicado aqui.

O método `body()` da sequência deve executar somente o comportamento puro da sequência. Quando gerando transações da `task body()` de uma sequência, fazê-lo usando o código de procedimento com o padrão genérico descritos nas linhas 10 a 15. As variáveis `req` e `rsp` são herdadas da classe base `uvm_sequence`. O nome da transação não precisa sempre ser `req`, podem ser usados variáveis de nomes alternativos dependendo dos nomes das transações usadas na definição de interface, embora o nome da variável e *string* devem ser o mesmo. *Constraints* podem ser inseridas se necessário.

Sempre instanciar objetos de sequência usando a *factory*. Instanciações devem seguir a seguinte o padrão descrito na seção 3.4.2. O nome na *string* de cada objeto de sequência deve ter o mesmo nome da variável, exceto quando há uma razão específica para a *string* diferir do nome da variável, tal como quando se criam múltiplos objetos de sequência em um laço usando a mesma variável.

Inicializar sequências processualmente chamando o método `start_item`.

Quando criar o objeto de sequência, sempre chamar o método `randomize()` antes de inicializar a sequência. Isso se aplica quando a classe do objeto de sequência não contém nenhum membro rand.

Finalizar sequências processualmente chamando o método `finish_item`.

4.4 Sequencer

O sequenciador (*sequencer*) controla o fluxo das transações do `uvm_sequence_item` gerado por uma ou mais sequências. Um sequenciador é conectado a um *driver* usando um *TLM export*. Um sequenciador gera itens de sequência aleatórios e os manda para o driver. Um sequenciador pode ser configurado com uma sequência padrão, no caso, o sequenciador iniciará a sequência automaticamente. Por padrão, um sequenciador não irá gerar sequências, ou uma sequência padrão deve ser configurada, ou sequências devem ser iniciadas manualmente no sequenciador. Para esse trabalho, o sequenciador entra como uma simples instância no *agent* que será construída e conectada. O *driver* puxa transações (`sequence_itens`) do sequenciador.

4.5 Driver

A classe `uvm_driver` é derivada de `uvm_component`. *Drivers* definidos por usuários devem ser construídos usando classes derivadas de `uvm_driver`. Um *driver* é geralmente usado como parte de um *agent* (visto posteriormente) que irá puxar transações de um sequenciador e enviará essas transações para a interface de conexão com o DUT.

O *driver* deve estar sincronizado com a interface do DUT, o que implica em uma necessidade desse componente de esperar mudanças de sinal na interface. Por conta disso, componentes ou sequências em UVM que fornecem transações para o *driver*, devem prover portas a nível de transação que não bloqueiem a execução do *driver* para que o *driver* sempre seja capaz de reagir imediatamente a sinais na interface do DUT.

Um *driver* deve apenas puxar transações do sequenciador quando necessitá-las. Ter um *driver* puxando transações do sequenciador com antecedência impedirá o sequenciador de ter o controle sobre o que acontecerá no *driver*. Portanto é mais eficiente para o *driver* puxar transações *just-in-time*, dessa forma, permitindo que a sequência gere transações aleatórias usando o estado atual do ambiente de verificação.

4.5.1 Implementação UVM

```
1  class my_driver extends uvm_driver #(my_transaction);
2    `uvm_component_utils(my_driver);
3
4    typedef virtual dut_if dut_if_type;
5    typedef my_transaction mytx_type;
6
7    function new(string name, uvm_component parent);
8      super.new(name, parent);
9    endfunction
10
11   function build_phase(uvm_phase phase);
12     super.build_phase(phase);
13     if(!uvm_config_db#(dut_if_type)::get(this, "", "dut_if", dut_if)) begin
14       `uvm_fatal("NOVIF", "failed to get virtual interface")
15     end
16   endfunction
17
18   function void connect_phase (uvm_phase phase);
19     super.connect_phase(phase);
20   endfunction
21
22   task run_phase (uvm_phase phase);
23     forever begin
24       dut_if_type d_if;
25       mytx_type tx;
```

```
26
27       seq_item_port.get_next_item(tx);   // seq_item_port is a member of
    uvm_driver
28
29       @(posedge dut_if_type.CLK || negedge dut_if_type.RST_n)
30         begin
31           if(!dut_if_type.RST_n) begin
32             d_if.cmd <= 0;
33             d_if.data <= 0;
34           else begin
35             d_if.cmd  <= tx.cmd;
36             d_if.data  <= tx.data;
37           end
38           seq_item_port.item_done();
39         end
40
41     end
42   endtask:run_phase
43
44 endclass:my_driver
```

Criar uma classe driver derivada de `uvm_driver` parametrizando a transação que este componente irá trabalhar.

Registrar a classe *driver* na factory usando a macro ‘`uvm_object_utils` como primera linha na classe.

Após a macro de registro, declarar as variáveis, principalmente, de interface e de transação como nas linhas 4 e 5.

Após as variáveis (se houver), definir um construtor que inclua uma única *string* de argumento com um valor predefinido de uma *string* vazia, uma chamada para `super.new`. O construtor deve seguir o mesmo padrão para construção de classe.

Após o construtor, na linha 11 chama-se o método `build_phase` onde é realizada uma chamada para `super.build_phase` na linha 12 e em seguida faz a tentativa de associar a interface do dut no driver através do método `get`. Caso o método get não obtenha sucesso, dev-se emitir um erro fatal, encerrando a simulação como visto nas linhas 13 e 14.

A chamada do método `conect_phase` nas linhas 18 a 20 é opcional, uma vez que não há conxões de subcomponentes dentro do *driver*.

No método `run_phase` nas linhas 22 a 42, será atribuído valores aos pinos da interface associada ao *driver*. A atribuição desses valores deve ser feita entre os métodos `seq_item_port.get_next_item(tx)` e `seq_item_port.item_done()`, linhas 27 e 38 respectivamente, onde `tx` é o nome da variável de transação declarada.

DICA A conexão física do *driver* geralmente é especificada por meio de um objeto de interface virtual. Esse objeto pode ser configurado usando o mecanismo de configuração, ou pode ser passado para o *driver* por seu *agent*.

DICA Não esquecer de chamar o método `item_done` quando o código tiver terminado de consumir o item de transação.

4.6 Monitor

A classe `uvm_monitor` deriva de `uvm_component`. Monitores definidos por usuários devem ser construídos usando classes derivadas de `uvm_monitor`. Um monitor é normalmente usado para detectar transações na interface e deixa-las disponíveis para outras partes do *testbench* através de uma *analysis port*.

Um monitor não deve atribuir valores a variáveis ou fios na interface SystemVerilog. O monitor deve assistir passivamente uma interface, e dela, criar transações que serão enviadas através de uma *analisys port* para posteriores processamentos. O monitor deve sempre ser escrito como um componente passivo e sua execução nunca deve ser bloqueada por outro componente ou sequência UVM. Além do mais, pode ser associado o uso de SVA (*System Verilog Assertions*) e diretivas de cobertura para conferência de protocolo e obtenção de dados para cobertura.

4.6.1 Implementação UVM

```
1  class my_monitor extends uvm_monitor #(my_transaction);
2    `uvm_component_utils(my_monitor);
3
4    typedef virtual dut_if dut_if_type;
5    typedef my_transaction mytx_type;
6
7    uvm_analysis_port#(mytx_type) aport;
8
9    dut_if_type d_if;
10
11   function new(string name, uvm_component parent);
12     super.new(name, parent);
13   endfunction
14
15   function void build_phase(uvm_phase phase);
16     super.build_phase(phase);
17
18     if(!uvm_config_db#(dut_if_type)::get(this, "", "dut_if", dut_if)) begin
19       `uvm_fatal("NOVIF", "failed to get virtual interface")
20     end
```

```
21
22      aport = new("aport", this);
23
24    endfunction
25
26    task run_phase (uvm_phase phase);
27      forever begin
28
29        mytx_type tx;
30        tx = my_transaction::type_id::create("tx");
31        tx.enable_recording("my_monitor");
32
33        assert(begin_tr(tx, "my_monitor"));
34        @(posedge dut_if_type.CLK) begin
35          tx.cmd  = dut_vi.cmd;
36          tx.data = dut_vi.data;
37        end
38        this.end_tr(tx);
39        aport.write(tx);
40      end
41    endtask:run_phase
42
43
44  endclass:my_monitor
```

Criação e definição da classe monitor derivada da classe `uvm_monitor`, parametrizando o tipo de transação a ser monitorada, realizar o registro na *factory* e declaração das variáveis necessárias.

Na linha 7 há a declaração da *analysis port* parametrizando o tipo de transação que será escrita através do método `write` da porta.

Após as variáveis (se houver), definir um construtor que inclua uma única *string* de argumento com um valor predefinido de uma *string* vazia, uma chamada para `super.new`. O construtor deve seguir a mesma forma do que foi usado até agora neste contexto.

Após o construtor, chama-se o método `build_phase` na linha 15 onde na sequência realiza uma chamada para `super.build_phase` e em seguida faz a tentativa de associar a interface do dut no driver através do método `get` na linha 18. Caso o método get não obtenha sucesso, um erro fatal é emitido, encerrando a simulação. Ainda na *build phase*, a linha 22 faz a construção da *analysis port* anteriormente instanciada.

Na *run phase*, é feita a declaração da variável de transação na linha 29, bem como o uso da *factory* para criação do objeto de transação na linha 30. Em seguida na linha 31 se faz a gravação do fluxo de transação atual através do método `enable_recording` de `uvm_transaction`. Logo após, na linha 33 se faz uso do recurso *assert* de SystemVerilog

para assegurar que a transação será gravada e armazenada através da função `begin_tr`, que recebe como argumentos a transação a ser gravada e uma *string* que no caso será o nome do monitor que chama essa função. Sincronizados por algum sinal de controle de preferência do verificador, geralmente o sinal de *clock* da interface, as variáveis do componente de transação instanciado recebem os valores pegos da interface nas linhas 35 e 36. Finalmente, é marcado o fim da transação através do método `end_tr` na linha 38, e conclui-se fazendo a escrita da transação na analysis port anteriormente instanciada através do método `write` na linha 39.

DICA Um monitor pode ser útil "*stand-alone*", observando a atividade em um conjunto de sinais de tal forma que o resto do *testbench* possa ver essa atividade na forma de objetos de transação completos. Alternativamente, pode compor o *agent*.

DICA Usando uma *analysis port* para enviar suas saídas para o resto do *testebench*, o monitor pode garantir que irá entregar seus dados de saída sem consumir tempo. Consequentemente, o método `run_phase` pode começar imediatamente a trabalhar para receber a próxima transação em sua interface.

DICA A conexão física do monitor é especificada por meio de uma interface virtual. Esse objeto pode ser configurado usando o mecanismo `uvm_config_db` (Seção 3.6), ou pode ser passado para o monitor pelo seu *agent*.

4.7 Agent

A classe `uvm_agent` é derivada de `uvm_component`. Cada *agent* definido pelo verificador deve ser criada como uma classe derivada de `uvm_agent`. Não há uma definição formal para um *agent* em UVM, mas um *agent* deve ser usado para encapsular tudo que for necessário para estimular e monitorar uma conexão lógica com um DUT.

Um típico *agent* contém instâncias de *driver*, *monitor* e *sequencer*. Ele representa um componente de verificação autocontido, designado para trabalhar com uma interface bem definida e específica, por exemplo, um barramento padrão como AMBA AXI. Um *agent* deve ser configurado tanto para um monitor puramente passivo, quanto para um componente de verificação ativo que pode tanto monitorar como estimular sua interface.

Também pode haver *testbenches* onde o verificador pode optar pela instanciação de um único *agent*, dessa forma, ele deve configurar diferentes portas para, de acordo com os sinais de protocolo, diferenciar os sinais de interface e direcioná-los para o componente adequado no fluxo de verificação, ou seja, se for uma transação de entrada do DUT, deve ser direcionado para o modelo de referência, caso seja saída do DUT, direcionar para que o comparador possa confrontar o resultado com a saída do *refmod*. A Figura 5 mostra o modelo de *testbench*, utilizado para verificar o bloco *reed-solomon* no projeto PEM

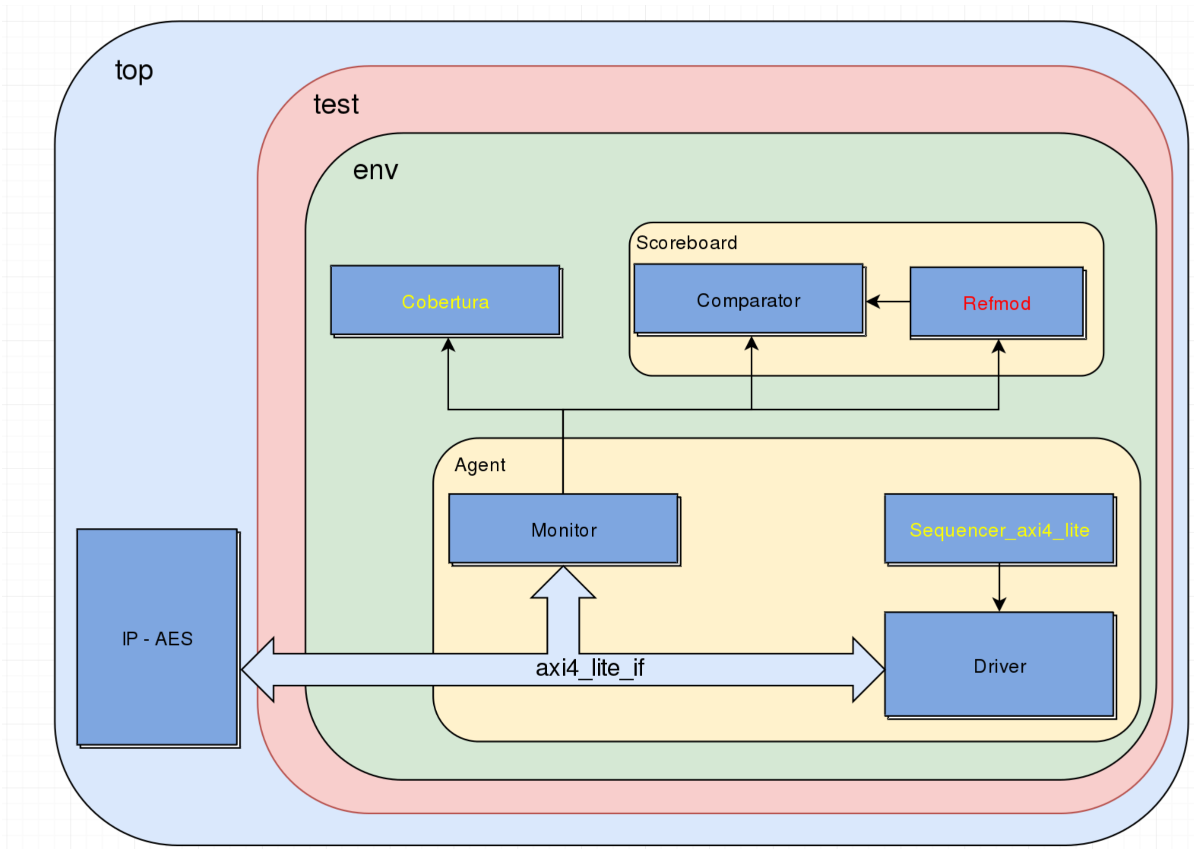


Figura 5 – Esquema de testbench com utilização de apenas um agent distribuindo transações bilateralmente

(Projeto de Excelência em Microeletrônica) da UFCG, ilustra como um único *agent* pode alimentar tanto o comparador, o modelo de referência e o módulo de cobertura ao mesmo tempo.

4.7.1 Implementação UVM

```
class my_agent extends uvm_agent;

    `uvm_component_utils(my_agent)
    typedef uvm_sequencer#(my_transaction) my_sequencer
    uvm_analysis_port #(my_transaction) agtport;

    my_sequencer my_sequencer_h;
    my_driver    my_driver_h;
    my_monitor   my_monitor_h;

    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction: new

    function void build_phase(uvm_phase phase);
```

```
16      super.build_phase(phase);
17      agtport = new("agtport", this);
18      my_sequencer_h = my_sequencer::type_id::create("my_sequencer_h", this
    );
19      my_driver_h = my_driver::type_id::create("my_driver_h", this);
20      my_monitor_h = my_monitor::type_id::create("my_monitor_h", this);
21    endfunction: build_phase
22
23    function void connect_phase(uvm_phase phase);
24      my_driver_h.seq_item_port.connect( my_sequencer_h.seq_item_export );
25      my_monitor_h.aport.connect(agtport);
26    endfunction: connect_phase
27
28  endclass: my_agent
```

Criação da classe agent derivada de `uvm_agent`, registro na factory, declaração de uma analysis port na linha 5 e instanciação dos subcomponentes do agent (sequencer, driver e monitor) nas linhas 7 a 9.

Após as variáveis (se houver), definir um construtor que inclua uma única *string* de argumento com um valor predefinido de uma string vazia, uma chamada para `super.new`. O construtor deve seguir a mesma forma do que foi usado anteriormente.

Na *build phase*, faz-se a construção da porta instanciada no *agent* na linha 17 e nas linhas 18 a 20 há a construção dos subcomponentes encapsulados pelo *agent*.

Na *connect phase* é feita a conexão entre as portas dos componentes instanciados no *agent*. Essa conexão é feita conforme codificado nas linhas 24 e 25,onde usa-se a hierarquia da primeira porta conectando-se à hierarquia da segunda porta que é passada como argumento da função. No exemplo, é feita a conexão da porta `seq_item_port` do *driver* com a porta `seq_item_export` do *sequencer*. Da mesma forma, é feita a conexão da *analysis port* `aport` definida do monitor com a *analysis port* `agtport` definida no próprio agent.

DICA Um *agent* deve representar um bloco de verificação de um IP, um componente reutilizável que pode ser conectado a um à uma determinada interface de um DUT e então formar parte do ambiente UVM.

DICA Todo *agent* ativo deve ter um subcomponente sequenciador capaz de gerar estímulos aleatórios para seu subcomponente `driver`.

4.8 Modelo de Referência

O modelo de referência é um modelo que simula a nível de software as funcionalidades do dispositivo sob verificação. Pode ser modelado tanto em SystemVerilog, quanto

em outras linguagens de mais alto nível. Geralmente modelos de referência são modelados em SystemC ou até mesmo MATLAB e incorporados no ambiente de verificação.

O modelo de referência ou *refmod*, recebe os dados de transação vindos do monitor através de uma *analysis imp*, e alimentado dessa transação, executa as funções em alto nível, gerando então as saídas que posteriormente serão enviadas em forma de novas transações para o comparador, que por sua vez será o responsável por comparar as entradas provenientes do modelo de referência com as entradas oriundas do DUT.

É importante que o modelo de referência tenha um alto grau de confiabilidade nas funcionalidades modeladas por ele, pois ele será a referência que o projeto do DUT terá de seguir. Se o modelo de referência por alguma razão não realizar propriamente as funcionalidades descritas na especificação do projeto, por consequência o ambiente de verificação irá condicionar o *design* ao erro. Portanto a certeza de um modelo de referência bem feito e correto é essencial.

Interessante notar que o modelo de referência recebe suas transações do monitor através de uma porta do tipo `analysis_imp`. Isso se dá devido ao fato de que para enviar essas transações para o *refmod*, o monitor faz uso do método `write` que por sua vez está implementado do componente `refmod`. Por implementar diretamente o método usado por outro componente conectado é que se faz uso da `uvm_analysis_imp`.

4.8.1 Implementação UVM

```
1  import "DPI-C" context function void dut_model (input int data, input int
      cmd, output int dataout);
2
3  class my_refmod extends uvm_component;
4     `uvm_component_utils(my_refmod)
5
6     int data_in, cmd, data_out;
7
8     my_transaction tx_in;
9     my_transaction tx_out;
10
11    uvm_analysis_port #(my_transaction) out;
12    uvm_analysis_imp #(my_transaction, my_refmod) in;
13
14    event begin_refmodtask, begin_record;
15
16    function new(string name = "my_refmod", uvm_component parent);
17        super.new(name, parent);
18    endfunction: new
19
20    virtual function void build_phase(uvm_phase phase);
21        super.build_phase(phase);
```

```
22
23          out = new("out", this);
24          in = new("in", this);
25      endfunction: build_phase
26
27      virtual function write ( axi4lite_master_transaction t);
28          tx_in = my_transaction#()::type_id::create("tx_in", this);
29          tx_in.copy(t);
30          -> begin_readtr;
31  endfunction
32
33
34      virtual task run_phase(uvm_phase phase);
35          super.run_phase(phase);
36          fork
37          read_tr();
38          refmod_task();
39          write_tr();
40          join
41      endtask: run_phase
42
43      task read_tr();
44      forever begin
45          @(begin_readtr);
46          tx_in = my_transaction#()::type_id::create("tx_in", this);
47          data_in = tx_in.data;
48          cmd = tx_in.cmd;
49          ->begin_refmodtask
50      end
51      endtask:read_tr
52
53      task refmod_task();
54      forever begin
55          @(begin_refmodtask);
56          dut_model(data_in, cmd, data_out);
57          ->begin_record
58      end
59  endtask:refmod_task
60
61      task write_tr
62      forever begin
63          @(begin_record)
64          tx_out = my_transaction#()::type_id::create("tx_out", this);
65          begin_tr(tx_out,"refmod");
66          tx_out.data = data_out
67          tx_out.cmd = tx_in.cmd;
68          end_tr(tx_out);
```

```
69     end
70     out.write(tx_out);
71     endtask:write_tr
72
73 endclass:my_refmod
```

É necessário Importar a DPI no SystemVerilog através da função import, permitindo assim o uso das funcionalidades modeladas em software para geração de respostas confiáveis. A importação de uma função modelada em C, por exemplo, é feito na linha 1.

Segue-se normalmente com os procedimentos de criação da classe refmod derivada da classe `uvm_component`, registro na factory e declaração de variáveis auxiliares necessárias. nas linhas 8 e 9 são feitas declarações de uma transação de entrada e uma de saída.

Nas linhas 11 e 12 há a declaração de uma `analysis_imp` de entrada e uma `analysis_port` de saída. Importante observar que são instanciadas duas portas, pois se trata de um componente que não mais pega dados a partir de uma interface, trabalhando assim diretamente a partir do nível de transação capturada através da porta de entrada.

Na linha 14 aparece a declaração de eventos para auxiliar no fluxo de funcionamento do modelo.

Definir um construtor que inclua uma única *string* de argumento com um valor predefinido de uma *string* vazia, uma chamada para `super.new`. O construtor deve seguir a mesma forma do que foi usado anteriormente.

Na *build phase*, após a chamada da `super.build_phase(phase)`, é realizado a construção das portas de entrada e de saída do *refmod* como visto nas linhas 20 a 25.

A implemetntação da função `write` que será chamara pelo monitor para esquever a transação que chegará ao refmod através da `analysis_imp`, justificando assim a escolha dessse tipo de porta para a entrada de transações nesse componente. Nessa função, é feita uma cópia da transação que é passada como argumento na hora da chamada da função, para o componente de transação que foi construído na *build phase*.

Em seguida é definida a *run phase* a partir da linha 35, onde ocorrerão as principais atividades do modelo de referência. Como de praxe, primeiramente se faz a chamada de `super.run_phase(phase)`. Na sequência, na linha 37 é feito um `fork` para que as *tasks* incluídas sejam executadas em paralelo, porém ordenadas por eventos que são disparados dentro de cada *task*. Essa organização não é mandatória, porém para fins didáticos se foi feito dessa formano intuito de de ilustrar o uso dos eventos. No exemplo usado para demonstrar a montagem de um modelo de referência para UVM, foram usadas três *tasks* com funcionalidades diferentes e sincronizadas por eventos. A *task* `read_tr()` faz a leitura da transação de entrada que recebeu a cópia vinda do agent, atribuindo seu valores em

variáveis locais e em seguida dispara o evento para que o modelo de referência entre em execução, enquanto as outras *tasks* aguardam seus respectivos eventos serem disparados. Uma vez detectado o evento, `refmod_task` chama a função modelada com a funcionalidade do bloco implementada em software, e em seguida dispara outro evento para que sejam gravados os resultados na transação de saída. Por fim, com o acontecimento do último evento, `write_tr()` cria a transação de saída, que receberá os valores de resposta da função correspondente ao modelo de referência. Por último, ainda na mesma *task*será feito a escrita dessa transação para a porta de saída do *refmod*.

4.9 Comparador

Os componentes da família `uvm_in_order_*_comparator` podem ser usados para comparar dois canais de transações em um ambiente UVM. Cada comparador fornece um par de analysis exports que agem como receptores dos canais dos fluxos de transação (os fluxos tipicamente originam-se das analysis ports nos drivers e monitores UVM). As transações podem ser do tipo built-in (ex. int, enums, structs) ou classes. Deve-se usar `uvm_in_order_class_comparator` para comparar objetos de classe, e `uvm_in_order_built_in_comparator` para comparar objetos do tipo *built-in*. Em cada caso o tipo é definido por parâmetro. Ambas as versões são derivações da classe pai `uvm_in_order_comparator`.

As transações recebidas são retidas em FIFO *buffers* e comparadas em outra leva. Transações individuais, portanto, podem chegar em tempos diferentes e ainda assim combinarem com sucesso (*match*). Uma conta de `matches` e *mismatches* é mantida pelo comparador.

Neste modelo de *testbench* foi optado por usar o comparador com uma simples instância de um objeto de classe `uvm_in_order_class_comparator`, uma vez que as transações foram definidas como objetos de classes, dentro do *scoreboard*, que será abordado em seguida. A definição de um comparador dentro de um ambiente de encapsulamento é feito da seguinte forma:

```
typedef uvm_in_order_class_comparator #(T) comp_type;
comp_type comp
```

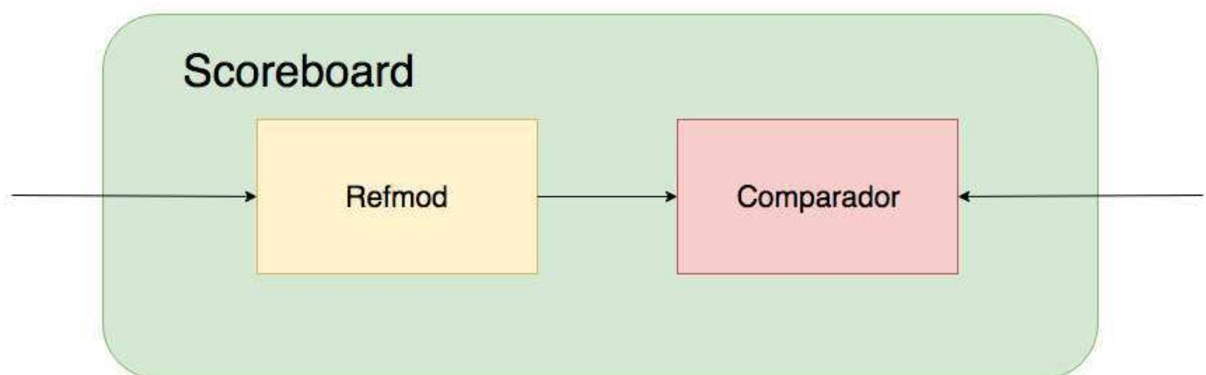
4.10 Scoreboard

A classe `uvm_scoreboard` é derivada de `uvm_component`. Scoreboards definidos pelo usuário devem ser construídos derivando da classe `uvm_scoreboard`.

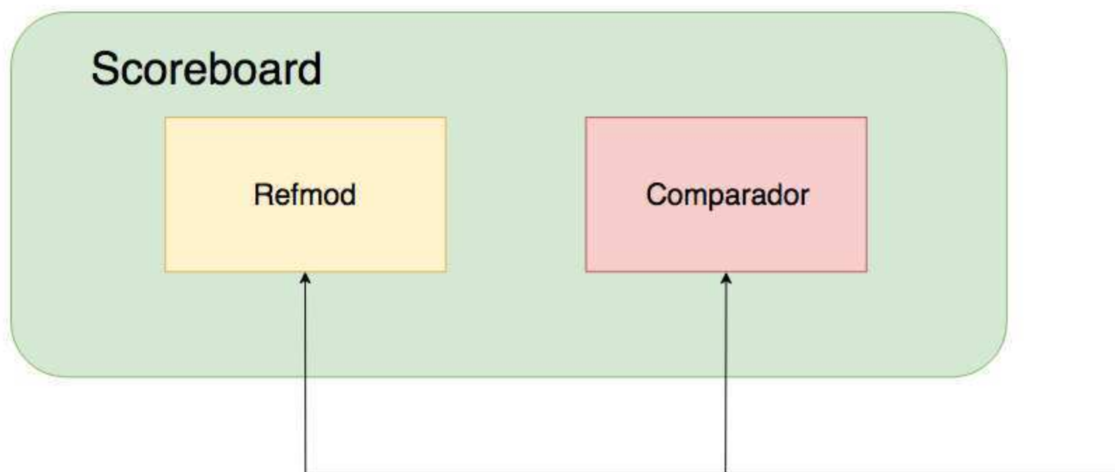
Um *scoreboard* geralmente observa transações em uma ou mais entradas do DUT, computa os efeitos esperados dessas transações, e armazena uma representação desses efei-

tos de forma adequada para uma posterior checagem quando a transação correspondente aparecer (ou não) na saída do DUT.

Ele servirá como encapsulamento do modelo de referência e comparador. Nele, esses componentes serão construídos e conectados entre si de acordo com o fluxo das transações. De acordo com a configuração do *testbench* as conexões dentro do *scoreboard* podem mudar um pouco, a Figura 6 ilustra bem essa diferença para um *testbench* com um ou dois *agents*. Para uma configuração de um agent as transações pré-DUT e pós-DUT vêm do mesmo destino, diferentemente de quando há uma configuração com um agent ativo e um passivo.



(a) Testbench com dois agents



(b) Testbench com úinco agent

Figura 6 – Modelo de conexões do scoreboard

4.11 Cobertura

O modelo de cobertura funcional é definido em termos de estímulo. O DUT estará completamente testado se for executado um conjunto completo de estímulos por ele que

cubra todas, ou boa parte, dos cenários possíveis em termos de sinais.

O bloco de cobertura geralmente é o responsável por finalizar a simulação. Isso pode ocorrer ou quando o *testbench* atingir um nível de cobertura aceitável, ou após un número pré-definido de transações. A simulação é encerrada coma execução da linha `running_phase.drop_objection(this);`.

4.11.1 Covergroups

São usados grupos de cobertura (*covergroups*) para capturar a cobertura funcional. O grupo de cobertura encapsula a especificação de um modelo de cobertura. Cada especificação de grupo de cobertura pode incluir os seguintes componentes:

- Um conjunto de pontos de cobertura (*coverpoints*);
- Cobertura cruzada entre pontos de cobertura;
- Argumentos de verificação formal opcionais;
- Opções de cobertura.

Um grupo de cobertura é definido entre as palavras chaves `covergroup` e `endgroup`. Uma instância pode ser criada usando o operador `new()`.

```
covergroup cg;
...
...
...
endgroup

cg cg_inst = new;
```

O exemplo acima define um grupo de cobertura chamado `cg`. Uma instância de `cg` é declarada como `cg_inst` e criada usando o operador `new`.

´

Sample

A cobertura deve ser disparada para amostragem dos valores de cobertura. A amostragem pode ser feita usando:

- Qualquer expressão de evento de borda de alguma variável;
- Chamando o método `sample()`.

```
covergroup cg @(posedge clk);
...
...
...
endgroup
```

O exemplo acima define um grupo de cobertura chamado `cg`. Esse grupo de cobertura será amostrado automaticamente sempre que houver uma borda de subida no sinal de clock `clk`.

```
initial // or task or function or Always block
  begin
  ...
  ...
  cg_inst.sample();
  ...
  end
```

A amostragem também pode ser feita chamando explicitamente o método `.sample()` no código processual. Isto é usado quando a amostragem de cobertura é necessitada com base em alguns cálculos ao invés de eventos.

4.11.2 Coverpoints

Um grupo de cobertura pode conter um ou mais pontos de cobertura. Um ponto de cobertura pode ser um valor ou expressão integral. Um ponto de cobertura cria um escopo hierárquico, e pode ser rotulado. Se o rótulo é especificado, então ele designa o nome do ponto de cobertura.

```
1  program main;
2  bit [0:2] y;
3  bit [0:2] values[$] =    {3,5,6};
4
5  covergroup cg;
6  cover_point_y : coverpoint y;
7  endgroup
8
9  cg cg_inst = new();
10
11 initial
12 foreach(values[i])
13 begin
14 y = values[i];
15 cg_inst.sample();
16 end
17 endprogram
```

No exemplo acima, o ponto de cobertura `y` é amostrado. O ponto de cobertura é chamado de `cover_point_y`. No relatório de cobertura esse nome será visto. Um grupo de cobertura `cg` é definido e sua instância `cg_inst` é criada. O valor de `y` é amostrado quando `cg_inst.sample()` é chamado. Os possíveis valores totais para `y` são 0,1,2,3,4,5,6 e 7. A variável `y` recebe apenas os valores 3,5 e 6. O mecanismo de cobertura deve reportar que apenas três valores estão cobertos quando há um total de 8 valores possíveis, apresentando assim uma cobertura de 37,5%.

4.11.3 Bins

Bins funcionais de transição são usados para examinar as transições legais de um valor. Systemverilog permite especificar um ou mais conjuntos de valores transitórios ordenados no ponto de cobertura. Os tipos de transições podem ser:

- Transição de valor único;
- Sequência de transições;
- Conjunto de transições;
- Repetições consecutivas;
- Faixa de transições;
- Repetições não consecutivas.

A criação de *bins* explícitos é o método recomendado. Nem todos os valores são interessantes ou relevantes em um ponto de cobertura, desta forma quando o usuário sabe os valores exatos que ele irá cobrir, ele pode usar *bins* explícitos. Os *bins* também podem ser nomeados.

```
1
2  program main;
3  bit [0:2]y;
4  bit  [2:0] values[$]=  {3,5,6};
5
6  covergroup cg;
7  cover_point_y : coverpoint y {
8  bins a = {0,1};
9  bins b = {2,3};
10 bins c = {4,5};
11 bins d = {6,7};
12 }
13
14 endgroup
```

```
15
16 cg cg_inst = new();
17 initial
18 foreach(values[i])
19 begin
20 y = values[i];
21 cg_inst.sample();
22 end
23
24 endprogram
```

No exemplo acima, *bins* são criados explicitamente. Os *bins* são chamados `a`,`b`,`c` e `d`. O mecanismo de cobertura reportará em seu relatório que 3 dos 4 *bins* foram cobertos. Os *bins* `b`, `c` e `d` foram cobertos, e o *bin* `a` aparecerá como um *bin* descoberto, totalizando assim uma cobertura de 75%.

4.11.3.1 Bins ignorados

Um conjunto de valores ou transições associadas com um ponto de cobertura pode ser explicitamente excluído da cobertura sendo especificados como `ignored_bins`.

```
1 program main;
2 bit [0:2] y;
3 bit [0:2] values[$] =    {1,6,3,7,3,4,3,5};
4
5 covergroup cg;
6 cover_point_y : coverpoint y {
7 ignore_bins ig = {1,2,3,4,5};
8 }
9
10 endgroup
11
12 cg cg_inst = new();
13 initial
14 foreach(values[i])
15 begin
16 y = values[i];
17 cg_inst.sample();
18 end
19
20 endprogram
```

No exemplo acima, os valores possíveis para `y` são de 0 a 7. Foi especificado para ignorar valores de 1 a 5 em `ignore_bins` na linha 7. Dessa forma, os valores esperados são 0, 6 e 7. Desses três valores, apenas 6 e 7 foram gerados, totalizando uma cobertura de 66,66%.

4.11.3.2 Bins ilegais

Um conjunto de valores ou transições associadas com um ponto de cobertura podem ser marcados como *bins* ilegais especificando-os como `ilegal_bins`. Todos os valores de transição associados com *bins* ilegais serão excluídos da cobertura. Se um valor ou transição ilegal ocorrer, um erro de execução é emitido.

```
program main;
bit [0:2]y;
bit [2:0] values[$] =   {1,7,3,6,3,4,3,5};

covergroup cg;
cover_point_y : coverpoint y{
ilegal_bins ib = {7};
}
endgroup

cg cg_inst = new();
initial
foreach(values[i])
begin
y = values[i];
cg_inst.sample();
end

endprogram
```

No exemplo acima, os valores possíveis para `y` são de 0 a 7. Foi especificado como ilegal o valor 7 em `ilegal_bins` na linha 7. Dessa forma, os valores esperados são de 0 a 6. Como 7 foi atribuído à `y`, o mecanismo de cobertura emitirá um erro como o mostrado na Figura 7.

Result:

** ERROR **
Illegal state bin ib of coverpoint cover_point_y in
covergroup cg got hit with value 0x7

Figura 7 – Erro emitido devido à um *illegal bin*

4.11.4 Cobertura cruzada

O cruzamento permite monitorar informações recebidas simultaneamente em mais de um ponto de cobertura. A cobertura cruzada é especificada usando a *construct* `cross`.

```
program main;
bit [0:1]y;
```

```
3 bit [0:1] y_values[$]=    {1,3};
4
5 bit [0:1] z;
6 bit [0:1] z_values[$] =    {1,2};
7
8 covergroup cg;
9 cover_point_y : coverpoint y;
10 cover_point_z : coverpoint z;
11 cross_yz : cross cover_point_y, cover_point_z;
12 endgroup
13
14 cg cg_inst = new();
15 initial
16 foreach(y_values[i])
17 begin
18 y = y_values[i];
19 z = z_values[i];
20 end
21
22 endprogram
```

No programa acima, `y` pode ter quatro valores, 0, 1, 2 e 3, bem como `z` que também pode ter os mesmos quatro valores. O produto do cruzamento de `y` e `z` são 16 valores: (0,0), (0,1), (0,2), (0,3), (1,0), (1,1) ... (y,z) ... (3,2), (3,3). Apenas as combinações (1,1) e (3,2) são geradas.

4.12 Environment

Uma classe derivada de `uvm_env` deve ser usada para modelar e controlar o ambiente de teste, mas não inclui os testes em si. Um *environment* pode instanciar outros *environments* de uma hierarquia. O *environment* incluirá todos os principais componentes da metodologia: gerador de estímulos, *driver*, *monitor* e *scoreboard*. Um *environment* é conectado ao dispositivo sob teste (DUT) através de uma interface virtual. O *environment* *top-level* deve ser instanciado e configurado em um teste.

4.12.1 Implementação UVM

```
1 class rs_env extends uvm_env;
2
3   my_agent agent_mst;
4   my_agent agent_slv;
5   my_scoreboard score;
6   my_cover cov;
7
8
```

```
9     `uvm_component_utils(my_env)
10
11    function new(string name, uvm_component parent = null);
12      super.new(name, parent);
13    endfunction
14
15    virtual function void build_phase(uvm_phase phase);
16      super.build_phase(phase);
17      agent = my_agent#()::type_id::create("agent", this);
18      score = my_scoreboard::type_id::create("score", this);
19      cov = my_cover::type_id::create("cov",this);
20    endfunction
21
22    virtual function void connect_phase(uvm_phase phase);
23      super.connect_phase(phase);
24      agent_mst.agtport.connect(cov.resp_port);
25      agent_mst.agtport.connect(score.ap_rfm);
26      agent_slv.agtport.connect(score.ap_cmp);
27
28    endfunction
29
30  endclass
```

No *environment* serão instanciados os componentes que ele engloba nas linhas 3 a 6, que por sua vez serão construídos e conectados através da `build_phase` (linhas 15 a 20) e `connect_phase` (linhas 22 a 26) respectivamente. A fase de conexão pode ser considerada a mais importante do *environment*, pois nela que será definido como será a ligação entre os componentes. Importante que todas as portas dos subcomponentes estejam conectadas de forma apropriada para que nenhuma transação seja perdida e que siga o esquema pré-definido pelo verificador, seja ele qualquer um dos já mostrados anteriormente, por exemplo. O exemplo codificado ilustra a implementação de um ambiente com dois agents que convergem para o scoreboard, onde ocorrerá o confronto de informações.

4.13 Teste

Uma classe derivada de `uvm_test` será usada para representar cada caso de teste. Um teste vai criar e configurar o(s) *environment(s)* necessários para verificar funcionalidades do DUT. Podem haver múltiplas classes de teste associadas com um *testbench*, mas um único objeto de teste desses é criado no início de cada simulação. Essa abordagem separa o *testbench* de *testcases* individuais, aumentado assim sua reusabilidade. A classe `uvm_test` é por si derivada de `uvm_component`. Uma classe de teste é várias vezes definida, mas nunca explicitamente instanciada no módulo de topo do *testbench*.

A *task* `run_test` é chamada em um bloco `initial` no módulo de topo do *testbench*

para instanciar um teste (usando a *factory*) e então rodá-lo. O método `build_phase` do teste cria o *environment* de topo.

A linha de comando de simulação `+UVM_TESTNAME=testname` especifica o nome do teste que será executado (um nome é associado com um teste ao registrá-lo na classe com a *factory*). Se esse *plusarg* não for usado, então o argumento `test_name` de `run_test` pode ser usado para especificar o nome do teste no lugar. Se nenhum nome de teste for fornecido, nenhum teste ou ambiente será criado e uma mensagem fatal será emitida.

4.13.1 Implementação UVM

```
class my_test extends uvm_test;
`uvm_component_utils(my_test)

  my_env env;
  my_sequence seq;

  function new(string name = "my_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = my_env::type_id::create("env_h", this);
    seq = my_sequence::type_id::create("seq", this);
  endfunction

  task run_phase(uvm_phase phase);
        seq.start(env.agent.sequencer);
  endtask : run_phase

endclass
```

No teste devem ser instanciados o *environment* e a *sequence* como é feito nas linhas 4 e 5. Essas, posteriormente, devem ser criadas na `build_phase` da forma que é feito nas linhas 13 e 14. Na linha 18, na `run_phase` devemos inicializar o *sequencer* que por sua vez inicializará a geração de estímulos.

4.14 Design Under Test - DUT

O DUT, do inglês, *Design Under Test*, é o dispositivo que está sendo testado pelo ambiente de verificação funcional. Na prática, nada mais é do que o Verilog, SystemVerilog ou VHDL do IP que está sendo verificado naquele momento. As saídas de resposta aos estímulos recebidos do *testbench* serão comparadas com as respostas do modelo de referência pelo comparador. A comparação entre os resultados deve apresentar similaridades

funcionais, caso contrário há uma grande chance de que correções sejam necessárias no código do IP.

A conexão bem estabelecida do DUT com uma interface compatível com o *testbench* é necessária para a comunicação e transferência de dados entre os componentes. Dessa forma, o bloco a ser testado deve parametrizar seus sinais de entradas e saídas de acordo com a interface e o modelo de transação que está sendo usado no *testbench* para um correto intercâmbio de informações.

Importante frisar que no DUT não há nenhuma linha de código UVM, isso porque o DUT em si não é um elemento UVM, e sim o hardware que foi projetado e está compondo o ambiente para ser testado do jeito que é, portanto não faz sentido adicionar linhas de UVM no bloco.

4.15 Topo

O topo é o módulo SystemVerilog que inicializará todo o ambiente de teste, bem como o DUT. É a camada mais superficial do *testbench* comum a todo projeto Verilog ou SystemVerilog, onde serão importadas as bibliotecas necessárias, será definida uma frequência de relógio que sincronizará o ambiente, o bloco a ser testado será instanciado e o teste inicializado.

```
1  import pkg::*;
2
3  module top;
4
5    logic clk;
6    logic rst;
7
8    initial begin
9      clk = 0;
10     rst = 0;
11     #22 rst = 1;
12
13   end
14
15   always #5 clk = !clk;
16
17
18     my_if#(.DATA_SIZE(32), .CMD_SIZE(3)  dut_if(clk, rst);
19
20     my_ip ip(.if(dut_if));
21
22   initial begin
23
```

```
24     uvm_config_db#(virtual my_if#(.DATA_SIZE(32), .CMD_SIZE(3)))::set(
       uvm_root::get(), "*", "dut_if", dut_if);
25
26     run_test("simple_test");
27   end
28 endmodule
```

A primeira coisa ser feita no módulo de topo é a importação do pacote na linha 1, um arquivo que inclui todos os componentes utilizaods no *testbench*. Em seguida declara-se as variáveis de *clock* e *reset* nas linhas 5 e 6. Logo após é gerado o clock com a frequÊncia desejada nas linhas 8 a 13.

No topo, é necessário que seja instanciada a interface usada, bem como o módulo correspondente ao IP testado, feito nas linhas 18 e 20. Em seguida passamos a interface para a base de dados como visto da seção 3.6 e feito na linha 24. Então o teste é inicializado com o comando `run_test("test_name_string")` na linha 26. A *string* passada como argumento nessa função é apenas para visualização, pois o teste executado através dessa função é passado na linha de comando como dito na seção 4.13.

O topo é a camada mais superficial do *testbench*, encerrando assim, as etapas de implementação do ambiente de teste e deixando-o pronto para ser colocado em execução.

5 Resultados

É de boa prática a configuração de um arquivo *Makefile* para coordenar a compilação dos códigos do *testbench*, módulo HDL do IP e modelo de referência, e então chamar a simulação. Para isso, Geralmente fazemos uso das ferramentas de compilação e simulação da Cadence™e Synopsys™.

A configuração do *Makefile* é bastante relativa e característica de cada projeto, mas deve conter pelo menos os comandos de compilação e simulação. Podem também ser adicionados comandos para visualização de formas de onda e também visualização de relatório de cobertura. No *Makefile* também é importante que sejam passadas as linhas de comando do UVM:

```
+UVM_TR_RECORD +UVM_VERBOSITY=HIGH +UVM_TESTNAME=my_test
```

+UVM_TR_RECORD Argumento utilizado para gravar as transações.

+UVM_VERBOSITY=HIGH Seleciona o nível de verbosidade da simulação.

+UVM_TESTNAME=my_test Indica qual teste deve ser executado na simulação.

Finalizada a simulação, em caso de sucesso, o relatório apresentado na Figura 8 deve aparecer no terminal indicando que a simulação foi concluída com sucesso. No *UVM Report Summary* aparecem informações relevante à simulação onde podem ser observados as contagem de *warnings*, erros e erros fatais inerentes ao UVM, sinalizados como `UVM_WARNING`, `UVM_ERROR` e `UVM` respectivamente.

```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :110027
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[Comparator Match]  9999
[ITEM_DONE]  9999
[MASTER_READ] 71923
[MASTER_WRITE]  8032
[RNTST]     1
[TEST_DONE]     1
[VCS_TR_AUTO]     1
[debug] 10071
$finish called from file "/usr/local/synopsys/vcs-K-2015.09-SP1-1/etc/uvm/base/uvm_root.svh", line 439.
$finish at simulation time            32009500
           V C S   S i m u l a t i o n   R e p o r t
Time: 320095000 ps
CPU Time:     14.640 seconds;       Data structure size:   0.4Mb
Wed Sep 19 22:10:07 2018
[hlcarvalho@jano bench_uvm]$ █
```

Figura 8 – Relatório do UVM sinalizando o fim da simulação bem sucedida

As informações relativas ao *testbench* em si são apresentadas logo em seguida. A primeira informação observada é o `comparator_match`, que nada mais é do que o contador de *matches* dentro do comparador. Como ressaltado anteriormente, cada vez que a saída do DUT e a saída do modelo de referência chegam no comparador em forma de transações apresentando o mesmo resultado, o comparador computa um *match*. O número de *mismatches* computados aparece logo abaixo, porém é omitido quando não há nenhum. O contador de *Match* indica 9999 *matches*, o mesmo número de transações concluídas como indicado no contador `ITEM_DONE`, sinalizando que todas as trasações passaram com sucesso e o IP está seguindo as especificações modeladasde acordo com o critério pré-definido.

Outro resultado interessante de ser analisado pelo verificador é o relatório de cobertura. Nele pode ser observado graficamente a porcentagem de cobertura alcançado como pode ser visto na figura 9.

Cover Group Item	Score	Goal	Weight	At Least
pkg::rs_cover::req_cover	94.42%			
pkg::rs_cover::req_cover	94.42%	100%	1	
addr_X_read_X_prot	100.00%	100%	1	
addr_X_write_X_wstrb	93.95%	100%	1	
cross_encry	55.86%	100%	1	
req_addr	100.00%	100%	1	
req_prot	100.00%	100%	1	
req_read_flag	100.00%	100%	1	
req_start	100.00%	100%	1	
req_wstrb	100.00%	100%	1	

Figura 9 – Relatório de cobertura indicando porcentagens de cobertura

No relatório de cobertura podem ser visto o grau de cobertura tanto dos sinais internos e de interface do IP, como também a porcentagem em grupos de cobertura definidos no bloco de cobertura e detalhados na seção 4.11. No exemplo da Figura 9 retirado do bloco *reed solomon*, do PEM-UFCG, está claro que não podemos considerar o IP 100% verificado, abrindo espaços para melhorias na robustez da verificação.

6 Conclusão

O ambiente de verificação é a base da atividade de verificação de qualquer componente eletrônico digital. A elaboração e implementação de um *testbench* é uma tarefa extensa e cansativa, porém necessária como preparativo primordial na verificação funcional, destacando a importância da compreensão de seu funcionamento para profissionais e estudantes da área.

A metodologia UVM atende as necessidades de verificação funcional tanto a nível acadêmico de pesquisa quanto a nível profissional. Empresas consolidadas na área utilizam UVM na verifcação funcional de seus produtos. O ambiente apresentado aqui é algo extremamente básico, porém é um bom demonstrativo da aplicação da metodologia e capacita o leitor a usar da criatividade e intuição para criação de *testbenches* bastante complexos.

Referências

DOULOS. *UVM Golden Reference Guide*. 2. ed. [S.l.]: Doulos Ltd., 2013.

DOULOS. *The Easier UVM Coding Guidelines*. [S.l.]: Doulos Ltd., 2016.

MEYER, A. *Principles of Functional Verification*. [S.l.]: Elsevier Science, 2003.

PIZIALI, A. *Functional Verification Coverage Measurement and Analysis*. [S.l.]: Springer US, 2007.

SALEMI, R. *The Uvm Primer: A Step-By-Step Introduction to the Universal Verification Methodology*. [S.l.]: Boston Light Press, 2013.

VASUDEVAN, S. *Effective Functional Verification: Principles and Processes*. [S.l.]: Springer US, 2006.

WIEMANN, A. *Standardized Functional Verification*. [S.l.]: Springer US, 2007.

WILCOX, P. *Professional Verification: A Guide to Advanced Functional Verification*. [S.l.]: Springer US, 2007. (Ifip Series).

WILE, B.; GOSS, J.; ROESNER, W. *Comprehensive Functional Verification: The Complete Industry Cycle*. [S.l.]: Elsevier Science, 2005. (Systems on Silicon).