Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica

Pedro Henrique Silva Cavalcante

Análise das metodologias de verificação UVM e COCOTB, utilizando um IP Core AES

Campina Grande
2020

Pedro Henrique Silva Cavalcante

Análise das metodologias de verificação UVM e COCOTB, utilizando um IP Core AES

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

Área de Concentração: Eletrônica

Orientador:
Gutemberg Gonçalves dos Santos Júnior

Campina Grande
2020

Pedro Henrique Silva Cavalcante

Análise das metodologias de verificação UVM e COCOTB, utilizando um IP Core AES

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

Aprovado em / /

Gutemberg Gonçalves dos Santos Júnior
UFCG

Marcos Ricardo de Alcântara Morais
Professor Convidado
UFCG

Campina Grande

2020

Dedico esse trabalho a todos os professores que tive e a meus familiares.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, por terem feito de tudo para que eu pudesse hoje estar onde estou.

Agradeço também a toda minha família por terem me dado muito amor e apoio durante toda a jornada e por sempre acreditarem em mim.

A minha companheira Fernanda que esteve comigo desde o primeiro dia dessa longa jornada que foi a minha graduação, me dando muito amor e motivação para seguir em frente.

Agradeço a todos os amigos que fiz durante a graduação em especial a Carlos e Mike que aguentaram meus abusos e com os quais vivenciei momentos de muita alegria e aperreios.

A todos que fazem parte do laboratório XMEN em especial a Klynger Dantas, José Iuri e Matheus Andrade, responsáveis pelo incentivo para o desenvolvimento deste trabalho e indispensáveis para o seu término dentro do prazo.

Ao professores que tive durante toda minha vida, parte do meu sucesso devo a eles, gostaria de levantar o nome de alguns, meu professor de matemática do fundamental Prof. José Freire que fez eu me apaixonar por matemática, Prof. Ivaldy José, Prof. Jorge Beja, Prof. Helder Barbosa por ter conseguido colocar o assunto de eletromagnetismo na minha cabeça, Prof. Edmar Gurjão e em especial os professores Gutemberg Junior, Marcos Morais e Elmar Melcher por terem sido meus tutores para minha carreira profissional.

RESUMO

Nesse trabalho é descrito a criação e comparação de dois ambientes de verificação funcional utilizando metodologias distintas, sendo elas o UVM e cocotb. São descritas as principais características de cada uma das metodologias aplicadas. Para fim de comparação utilizou-se um IP-Core AES no qual houve a análise dos resultados em ambos os casos. Apesar de ainda contar com problemas, o cocotb se mostrou suficiente e com potencial para desempenhar bem em atividades de verificação.

Palavras-chave: cocotb, metodologia de verificação, Python, SystemVerilog, UVM, verificação funcional.

ABSTRACT

This work describes the creation and comparison of two functional verification environments using different methodologies, namely UVM and cocotb. They are all the main characteristics of each of the applied methodologies. For comparison, an IP-Core AES was used, there was no analysis of the results in both cases. Despite still having problems, the cocotb is suspicious and has the potential to score well in verification activities.

Keywords: cocotb, functional verification, Python, SystemVerilog, UVM, Verification Methodology.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

UFCG	Universidade Federal de Campina Grande
DEE	Departamento de Engenharia Elétrica
TLM	*Transaction Level Modeling*
IP	*Intellectual Property*
RTL	*Register Transfer Level*
ASIC	Circuitos Integrados de Aplicação Específica
EDA	Electronic Design Automation
API	*Application Programming Interface*
OVM	*Open Verification Methodology*
DUT	*Design Under Test*

SUMÁRIO

1 INTRODUÇÃO

Com o avanço do desenvolvimento da microeletrônica, muitos circuitos adquirem uma grande complexidade e seu desenvolvimento representa um grande desafio de engenharia. Nesse contexto, os setores de desenvolvimento de *hardware* e circuitos integrados buscam aperfeiçoar seu fluxo de desenvolvimento e adotar metodologias padronizadas.

Em um fluxo tradicional de construção de *chips*, é comum a existência de três equipes, o *frontend*, responsável pela construção lógica e arquitetural do circuito a ser projetado, a equipe de verificação, responsável por garantir que a correta funcionalidade do circuito projetado e, por fim, a equipe de *backend* responsável por realizar o projeto da concepção física do circuito projetado.

Para garantir que as equipes atuem corretamente e o fluxo de desenvolvimento flua de acordo com o esperado foram desenvolvidas diversas metodologias e propostas que alinham todas as equipes envolvidas. A adoção correta dessas práticas viabiliza o rápido desenvolvimento assim como aumenta a confiabilidade do produto final, evitando dessa forma que os projetos falhem ao fim da produção e precisem serem recomeçados ou os *chips* refabricados.

Dentro de um fluxo de projeto de um chip as indústrias comumente estimam que a verificação funcional requer 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 fatores 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 de como o dispositivo deve funcionar.

Tendo em vista o fluxo de verificação funcional foi proposto, como trabalho de conclusão de curso, a análise e descrição de duas metodologias de verificação utilizadas pela indústria de desenvolvimento de circuitos integrados, sendo elas as metodologias

UVM e o COCOTB.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

A execução desse trabalho objetiva a análise comparativa das metodologias cocotb e UVM para a verificação funcional.

1.1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos para o desenvolvimento desse projeto são:

- Utilizar como caso de uso um IP core AES;
- Construir um ambiente de verificação para cada uma das metodologias;
- Aproximar ao máximo os dois ambientes de verificação, a fim de facilitar possíveis comparações;
- Levantar características distintas entre as metodologias;
- Levantar vantagens e desvantagens de cada um dos *testbenches*;
- Avaliar a adoção de cocotb para um fluxo de verificação.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 VERIFICAÇÃO FUNCIONAL

Verificação funcional é uma das etapas do desenvolvimento de microeletrônica, essa tem o objetivo de buscar equivalência entre o que foi especificado e o RTL implementado.

Dado a complexidade crescentes dos blocos IP/ASIC modernos, tornou-se impraticável realizar todos os casos de teste, fazendo com que seja preciso montar estratégias mais eficazes para cobrir o máximo de testes possíveis dado o tempo e recurso alocado.

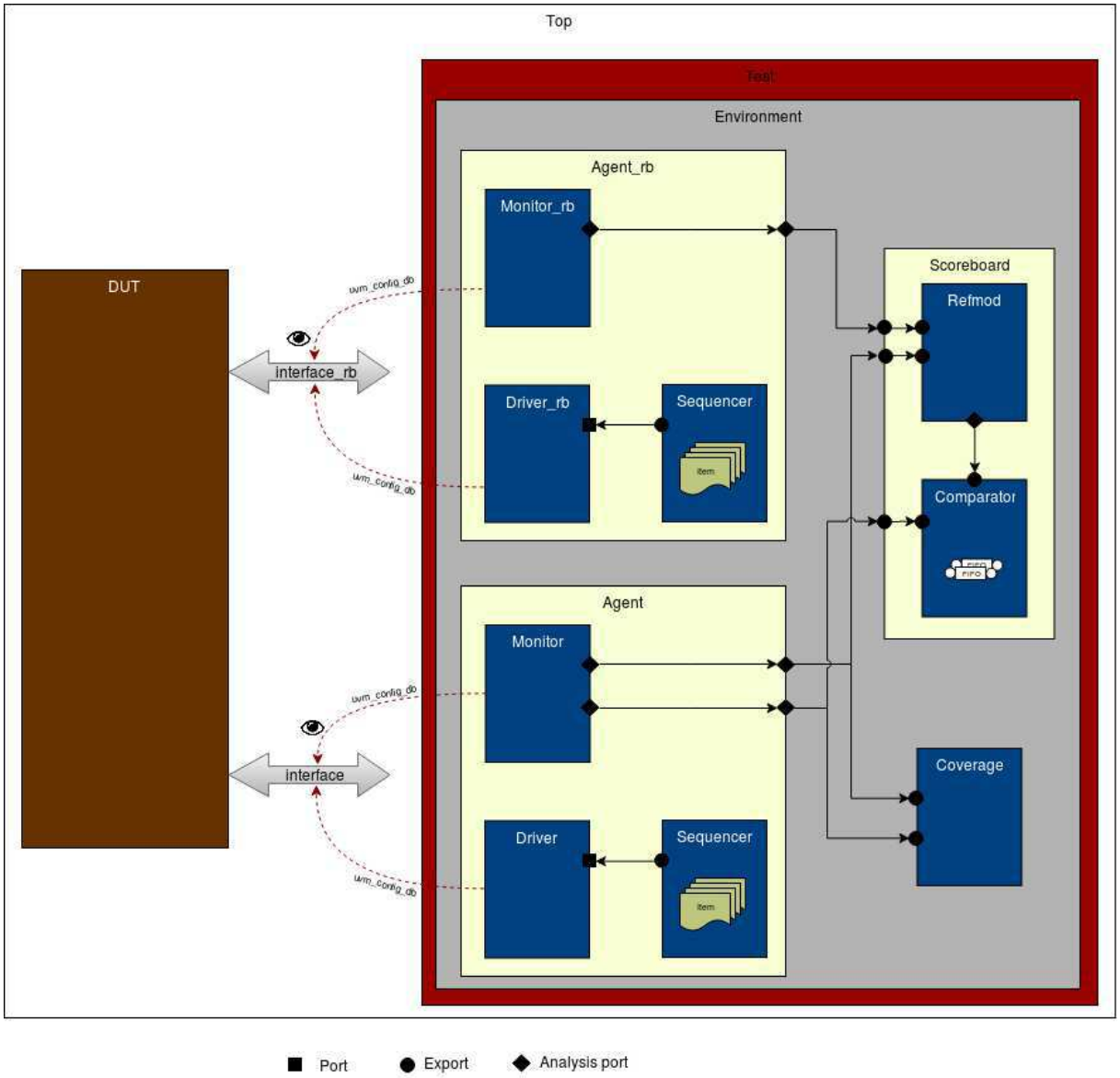
Como forma de otimizar o processo de verificação funcional diversas metodologias foram criadas a fim de padroniza-la e facilita-la, após diversas propostas de metodologias, a UVM acabou se tornando a mais aceita na industria, seja por sua utilização ser possível em todas as ferramentas de EDA das maiores desenvolvedoras do mercado, como também por sua grande capacidade de modularização e reutilização.

2.2 METODOLOGIA DE VERIFICAÇÃO UNIVERSAL - UVM

A metodologia UVM é um padrão para permitir o desenvolvimento e a reutilização de forma mais rápida dos ambientes de verificação. Sendo composto por um conjunto de bibliotecas de classes definidas usando a sintaxe e semântica de SystemVerilog, a ideia principal por trás do UVM é ajudar as empresas a desenvolver estruturas de *testbench* modulares, reutilizáveis e escaláveis, fornecendo uma estrutura de API que pode ser implementada em vários projetos.

O UVM é derivado principalmente da metodologia *Open Verification Methodology* (OVM), porém como o OVM não possuía compatibilidade com todas as EDAs do mercado acabou caindo em desuso. Por sua vez, o UVM tem suporte para vários fornecedores de EDA como *Synopsys*, *Cadence*, *Mentor*, entre outras. A biblioteca de classes UVM fornece utilitários genéricos como bancos de dados de configuração, TLM e hierarquia de componentes, além de recursos de automação de dados como cópia, impressão e comparação. Ele traz uma camada de abstração em que cada componente no ambiente de verificação tem uma função específica. Por exemplo, um objeto de classe *driver* será responsável apenas por direcionar os sinais para o RTL, enquanto que um monitor simplesmente monitora a interface do design e não direciona os sinais para essa interface (CHIPVERIF, 2020).

Figura 1 – Diagrama de um ambiente de verificação em UVM.



Fonte: O próprio autor

Como pode-se perceber no diagrama da Figura 1 existem diversos componentes que formam o *testbench* com UVM, que serão melhores discutidos na subseções seguintes.

2.2.1 *TESTBENCH* TOPO

O modulo topo do *testbench* é responsável por instanciar o DUT e a interface bem como por iniciar o modulo teste do UVM. Ele é um centro do *testebench* onde contem tudo o que é necessário para realizar a simulação, é nesse bloco que o sinal de *clock* e *reset* são definidos, bem como qualquer lógica, sendo *wrappers* ou outros componentes, a mais que seja necessária para se simular o RTL.

2.2.2 *DESIGN UNDER TEST* - DUT

O *design under test* (DUT) ou *design under verification* (DUV) nada mais é do que a representação do RTL o qual será verificado, instanciado no modulo topo e conectado a interface. A passagem dos sinais entre o ambiente de verificação e o DUT unicamente e realizada por meio da interface. De fato o ambiente não estimula nenhum sinal diretamente ao DUT, e este também não responde diretamente ao ambiente.

2.2.3 INTERFACE

A interface em SystemVerilog é um conjunto de sinais que representam o bloco a ser verificado, sendo agrupados de acordo com conveniência do projeto. Nessa instância é possível operar com os sinais por meio de *modports*, que são repensáveis por direcionar os sinais como entradas ou saídas, além de ser possível realizar operações logicas e aritméticas nesses sinais. A principal função da interface como comentada anteriormente é de conectar o DUT com o ambiente de verificação, será através dela que os estímulos serão passados do ambiente para o DUT, bem como as respostas do DUT serão entregues ao ambiente de verificação.

2.2.4 TESTE

Um caso de teste é um padrão para checar e verificar recursos e funcionalidades específicas do RTL. No plano de verificação estão listados todos os recursos e outros itens funcionais que precisam ser verificados e os testes necessários para cobrir cada um deles. Sendo que quanto mais complexo é o projeto, mais casos de testes serão necessários para se verifica-lo.

Em vez de escrever o mesmo código para diferentes casos de teste, colocamos todo o *testbench* em uma "caixa" denominada ambiente e usamos o mesmo ambiente com uma configuração diferente para cada teste. Cada caso de teste pode apresentar pequenas alterações nos outros blocos do ambiente bem como a execução de diferentes sequências.

2.2.5 AMBIENTE (*ENVIRONMENT*)

Um ambiente UVM contém vários componentes de verificação reutilizáveis que define sua configuração padrão conforme a aplicação. Por exemplo, um ambiente UVM pode ter vários agentes para diferentes interfaces com um *scoreboard* comum, uma cobertura funcional e *checkers* adicionais.

Ele também pode conter outros ambientes menores que foram verificados no nível de bloco e agora estão integrados em subsistemas. Isso permite que certos componentes e sequências usados na verificação de nível de bloco sejam reutilizados no nível de sistema.

2.2.6 AGENTES

Um agente encapsula um sequenciador, *driver* e monitor em uma única entidade instanciando e conectando os componentes por meio de interfaces TLM. Uma vez que o UVM tem tudo a ver com configurabilidade, um agente também pode ter opções de configuração como o tipo de agente UVM ativo ou passivo, diferenciando ambos pela presença ou ausência do sequenciador respectivamente.

2.2.7 *DRIVER*

O *driver* tem a função de receber todas as transações criadas no sequenciador e traduzi-las em sinais tais quais serão enviadas ao DUT por meio da interface, como a interface permite o mesmo *driver* também recebe as transações de resposta vindas do DUT. No *driver* é criada a lógica necessária de *handshake* dos sinais de controle para que os dados sejam enviados para o DUT no momento correto.

2.2.8 MONITOR

O monitor como o próprio nome diz monitora a interface, sempre que houver um *handshake* entre o DUT e o *driver* através da lógica de controle, havendo esse *handshake* o monitor reescreve os sinais em novas transações e a os manda para o modelo de referência para que este tenha entradas iguais ao DUT.

2.2.9 SEQUENCIADOR

O sequenciador é responsável por gerar as transações de dados como objetos de classes para assim poder envia-las ao *driver* o qual irá trabalhar com elas de acordo com o que foi anteriormente apresentado.

2.2.10 *SCOREBOARD*

O *scoreboard* tem como função comparar e pontuar acertos e erros provenientes do RTL tomando como base o modelo de referência. Por exemplo, os valores de gravação de uma leitura no registrador devem corresponder, tanto para o DUT como para o modelo de referência, a garantia de que esses dados foram passados ao comparador é de responsabilidade do *scoreboard*.

2.2.11 MODELO DE REFERÊNCIA

O modelo de referência como o próprio nome já diz é a referência do bloco de que os cálculos executados pelo design estão corretos. Ele é responsável por realizar a mesma lógica que o design realiza de acordo com a documentação podendo ou não instanciar

um outro código em escrito numa linguagem de alto nível, como C++ ou Python, a qual será responsável por realizar as operações matemáticas. O modelo por ser feito em uma linguagem de mais alto nível tem maior garantia de estar respondendo corretamente, de acordo com o que era previsto na documentação, aos estímulos gerados do que o design que é feito em uma linguagem de descrição de hardware.

2.2.12 COMPARADOR

O comparador tem o importante e simples trabalho de comparar os resultados do DUT e do modelo de referência reportando-os se sucesso como *Match* ou falha como *Mismatch*.

2.2.13 COBERTURA

Sabe-se que para validar um teste de verificação é preciso testar uma grande quantidade de estímulos de forma aleatória, mas apenas gerar estes estímulos aleatórios nem sempre é sinônimo de validação de um verificação, é bastante comum querer que o sequenciador realize certos estímulos para assim podermos validar por completo nossa verificação. É justamente esse o papel da cobertura na verificação: verificar todas as transações que deseja-se com relação ao funcionamento do bloco e a partir disto verificar se as respostas obtidas são esperadas. Assim percebe-se que a cobertura funciona como uma espécie de contador que vai observando os estímulos desejados e guardando esses valores. A cobertura amostra esses valores em termos de porcentagem, ou seja caso tenhamos observado metade dos estímulos desejados, podemos dizer que a cobertura atingiu o valor de 50%. Portanto caso a cobertura atinja o valor de 100% pode-se afirmar que a verificação foi concluida.

2.3 *COROUTINE BASED COSIMULATION TESTBENCH* - COCOTB

O COroutine based COsimulation TestBench, ou cocotb, é um ambiente de verificação de RTLs escritos em SystemVerilog e VHDL. Trata-se de um *framework* baseado em código aberto, que permite a verificação com a utilização de simulador arbitrário. A função de um simulador externo é limitada apenas a uma simulação de nível lógico e todas as transações são traduzidas para Python usando a Verilog Procedural Interface (VPI) (CIEPLUCHA; PLESKACZ, 2017).

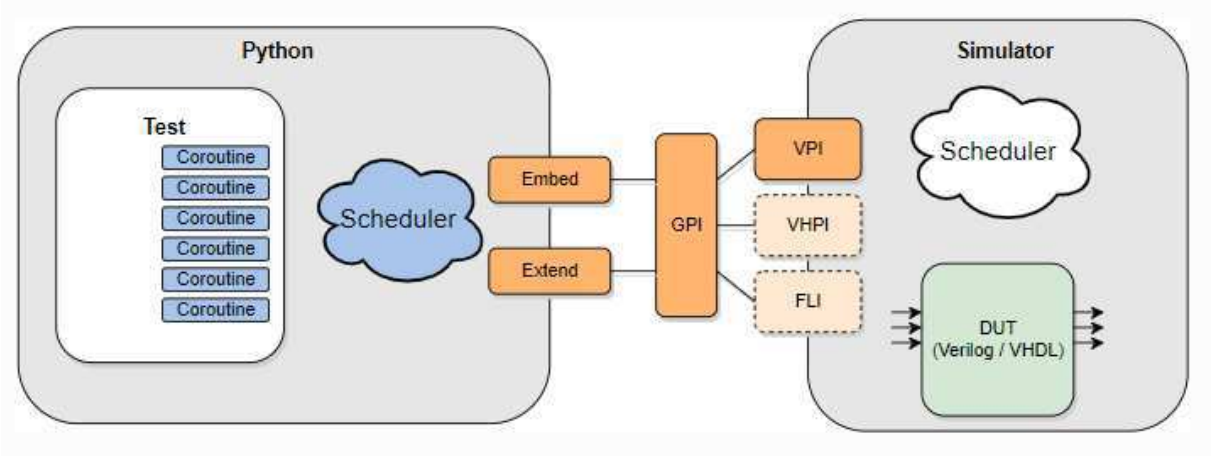
Com a separação das partes relativas a simulação dos sistemas digitais, a HDL é utilizada apenas no intuito de representar o sistema logicamente, enquanto que toda a parte de geração de estímulos, coordenação de testes e análise de cobertura passa a

ser executada utilizando uma linguagem de alto nível e de propósito geral que é o caso do Python. A utilização desse tipo de ferramenta facilita a construção de programas mais complexos com recursos que não estão disponíveis em linguagens HVL, como a manipulação de funções de primeira classe, que podem ser passadas como argumento de outras funções.

2.3.1 ESTRUTURA DE FUNCIONAMENTO

Um testbench de cocotb típico não requer nenhum código RTL adicional. O Design Under Test (DUT) é instanciado como o nível superior no simulador sem qualquer código de wrapper. cocotb direciona estímulos para as entradas do DUT (ou mais abaixo na hierarquia) e monitora as saídas diretamente do Python. O exemplo da Figura 2 demonstra como é o funcionamento da biblioteca

Figura 2 – Interação entre o simulador e o ambiente cocotb.



Fonte: (VENTURES, 2014)

Um teste em cocotb é simplesmente uma função em Python. A qualquer momento, o simulador está avançando no tempo ou o código Python está em execução. Além das funções temporais, é possível obter apontadores para os sinais do DUT através da hierarquia dos blocos e mudar diretamente seus valores.

O exemplo a seguir ilustra a utilização de um simples teste de um mux em cocotb.

```
1  # mux_tester.py
2  import cocotb
3  from cocotb.triggers import Timer
4  from cocotb.result import TestFailure
5
6  @cocotb.test()
7  def mux_test(dut):
8
9      dut.L0_i <= 0
10     dut.we_lp_i <= 0
```

```
11      dut.readout_mode_i <= 1
12      dut.L0_i <= 1
13      yield Timer(1, "ns")
14      if dut.we_lp_muxed_o != 1:
15          raise TestFailure("Failure!")
16
17      dut.readout_mode_i <= 0
18      yield Timer(1, "ns")
19      if dut.we_lp_muxed_o != 0:
20          raise TestFailure("Failure!")
```

2.3.2 CO-ROTINAS

Testbenches construídos em cocotb fazem uso de co-rotinas. Enquanto a co-rotina está em execução, a simulação é pausada. A operação utiliza da palavra-chave *await* para bloquear a execução de outra co-rotina ou passar o controle da execução de volta para o simulador, permitindo que o tempo de simulação avance.

Em comparação as linguagens comuns de HVL, a utilização de uma co-rotina se assemelha as palavras chaves *task* em SystemVerilog em que é possível consumir tempo de simulação dentro do seu escopo. Outro ponto de flexibilidade no uso de co-rotinas é a possibilidade da execução em paralelo através do método *fork*. As co-rotinas devem ter pelo menos uma chamada de evento de modo a sincronizar a execução com o simulador. Um exemplo de uso de co-rotinas pode ser visto abaixo.

```
1  import cocotb
2  from cocotb.triggers import RisingEdge
3
4  @cocotb.coroutine
5  def test_helper(dut):
6      dut.member <= 1
7      yield RisingEdge(dut.clk)
8
9  @cocotb.test()
10 def test(dut):
11     yield test_helper(dut)
```

2.3.3 TRIGGERS

Os *triggers* são usados para indicar quando o cocotb deve retomar a execução da co-rotina. Para usar um gatilho, uma co-rotina deve aguardar por ele. Isso fará com que a execução da co-rotina atual pause. Quando o gatilho disparar, a execução pausada será retomada.

A partir dos *triggers* é possível que o cocotb se comunique com o simulador utilizado por meio da interface VPI ou VHPI. Dessa forma, enquanto o código em Python estiver executando a simulação não avança e após a liberação dos gatilhos que ela avança

em termos temporais. Existe uma variedade de *triggers* disponíveis na biblioteca, os mais comuns são:

- *Timer(tempo, unidade)*: espera que um certo tempo de simulação passe.
- *Edge(sinal)*: aguarda um sinal para mudar de estado (borda ascendente ou descendente).
- *RisingEdge(sinal)*: aguarda a borda ascendente de um sinal.
- *FallingEdge(sinal)*: aguarda a borda de queda de um sinal.
- *ClockCycles(signal, num)*: espera por algum número de *clocks* (transições de 0 a 1).

O código abaixo ilustra a utilização de *triggers*:

```
async def coro_inner(clk):
    await Timer(1, units='ns')
    await FallingEdge(clk)
    return "Hello world"


task = cocotb.fork(coro_inner(dut.clk))

result = await Join(task)

assert result == "Hello world"
```

2.3.4 FERRAMENTAS DE *TESTBENCH*

Com a utilização de co-rotinas e *triggers* a biblioteca pode se expandir para oferecer algumas funcionalidades típicas para utilização em *testbenches*. Essas ferramentas auxiliam a tarefa de construir comportamentos complexos.

Logging

O cocotb possui uma biblioteca padrão para a utilização e geração de *logs*. Cada DUT, *monitor*, *driver* e *scoreboard* (bem como qualquer outra função usando o decorador de co-rotina) contém um *logging.Logger*, e cada um pode ser configurado para seu próprio nível de severidade. Dentro de um DUT, cada objeto hierárquico também pode ter níveis de severidade individuais definidos.

```
class AvalonSTPkts(BusMonitor):
...
    @coroutine
    def _monitor_recv(self):
        ...
        self.log.info("Received a packet of %d bytes" % len(pkt))
```

Barramentos

Os barramentos são simplesmente definidos como uma coleção de sinais. A classe *Bus* irá agrupar automaticamente qualquer grupo de sinais com nomes semelhantes dentro de um `dut.<bus_name><separator><signal_name>`

Drivers

A classe *Driver* permite que um teste acrescente transações para realizar a serialização de transações em uma interface física. Dentro da biblioteca existem alguns exemplos com uma variedade de barramentos, um deles pode ser visto no código abaixo.

```
class EndianSwapperTB(object):

    def __init__(self, dut, debug=False):
        self.dut = dut
        self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)

def run_test(dut, data_in=None, config_coroutine=None, idle_inserter=None,
             backpressure_inserter=None):

    cocotb.fork(Clock(dut.clk, 5000).start())
    tb = EndianSwapperTB(dut)

    yield tb.reset()
    dut.stream_out_ready <= 1

    if idle_inserter is not None:
        tb.stream_in.set_valid_generator(idle_inserter())

    # Send in the packets
    for transaction in data_in():
        yield tb.stream_in.send(transaction)
```

Monitors

A classe Monitor é uma classe base que se espera que seja derivada para seu propósito específico. Deve-se criar uma função `_monitor_recv()` que é responsável por determinar 1) em quais pontos da simulação chamar a função `_recv()`, e 2) quais valores de transação passar para serem armazenados na fila de recebimento de monitores. Os monitores são bons para ambas as saídas do DUT para verificação e para as entradas do DUT, para conduzir um modelo de teste do DUT para ser comparado ao DUT real.

```
class BitMonitor(Monitor):
    """Observes single input or output of DUT."""
    def __init__(self, name, signal, clock, callback=None, event=None):
        self.name = name
        self.signal = signal
```

```
6           self.clock = clock
7           Monitor.__init__(self, callback, event)
8
9       @coroutine
10      def _monitor_recv(self):
11          clkedge = RisingEdge(self.clock)
12
13          while True:
14              # Capture signal at rising edge of clock
15              yield clkedge
16              vec = self.signal.value
17              self._recv(vec)
```

3 BLOCO AES

O bloco selecionado para ser usado como caso de teste para o desenvolvimento desse trabalho foi o bloco de criptografia desenvolvido no Laboratório de Excelência em Microeletrônica do Nordeste - XMEN.

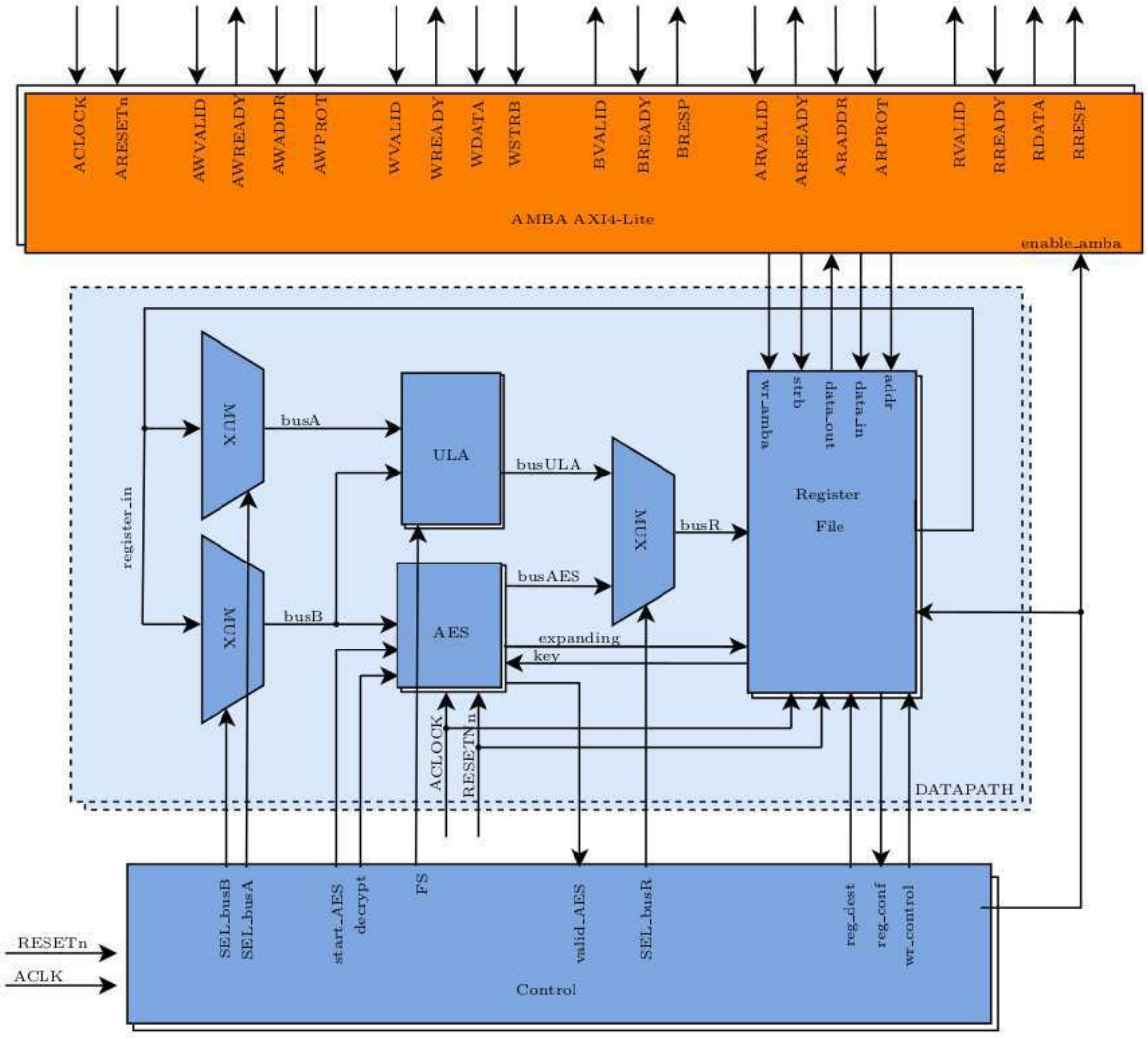
A escolha do AES se deu por ele ser um bloco com documentação bem definida, o que facilita o trabalho do verificador, além disso o bloco tem características que implicariam em uma verificação mais completa, tais como a presença de vários modos de operação e por ser um bloco que utiliza um protocolo de comunicação conhecido do mercado, no caso o AXI4_lite, este é uma versão mais simplificada do AXI4, porem já é suficiente para a aplicação.

Tendo sinais de entrada e saída, *handshake valid-ready*, strobe para melhor otimizar o acesso de registrados e um sinal de proteção para dar segurança aos dados, faz com que o AXI4-LITE seja um dos mais aceitos no mercado.

O *Advanced Encryption Standard* (**AES**), é um algoritmo de criptografia de chave simétrica, desenvolvido pelo governo dos EUA e anunciado pelo NIST (Instituto Nacional de Padrões e Tecnologia dos EUA) como U.S. FIPS PUB (FIPS 197). O AES cifra/decifra informações de 128 bits, com chaves de tamanhos variados, podendo ser de 128, 192 ou 256 bits.

O **IP** aqui apresentado, implementa esse algoritmo, oferecendo um hardware dedicado para transmitir e receber dados com segurança, porém, só permite chaves com tamanhos de 128 bits. O IP ainda pode ser configurado para processar vários modos de operação, sendo eles: **ECB, CBC, PCBC, CFB, OFB e CTR**. A comunicação é feita através da interface **AMBA AXI4-Lite**. A Figura 3 mostra sua arquitetura.

Figura 3 – Arquitetura do AES-128.



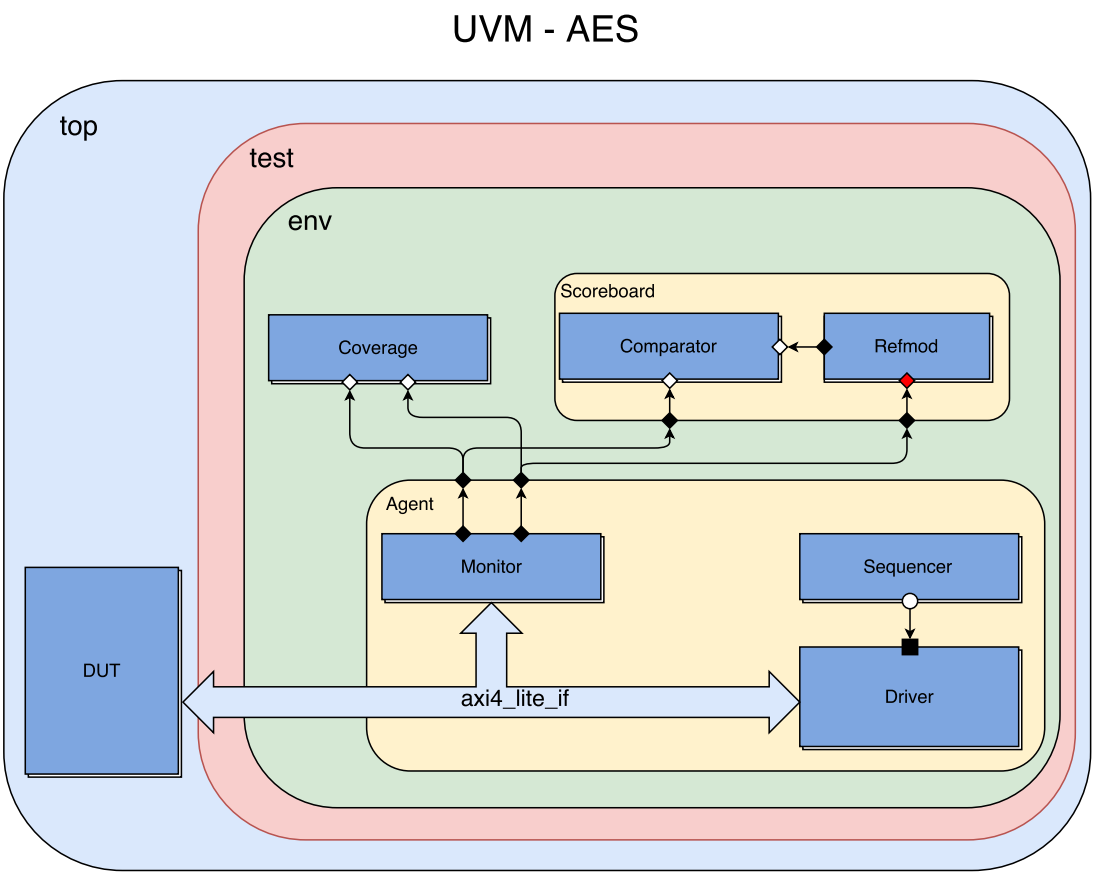
Fonte: (ABRANTES, 2018)

4 TESTBENCH UVM

Primeiramente foi desenvolvido o ambiente de verificação para o AES com UVM, nesse capítulo será descrito todo o *testbench* e os resultados obtidos.

A arquitetura do *testbench* seguiu uma linha bem tradicional e pode ser vista na Figura 4, além do ambiente de verificação, feito exclusivamente em *SystemVerilog* o modelo de referência foi desenvolvido em C++ e conectado ao ambiente via *Direct Programming Interface* (DPI) esse e todos os outros componentes desenvolvidos serão descritos na próxima seção.

Figura 4 – Arquitetura de *testbench* UVM desenvolvido.



Fonte: O próprio autor

4.1 COMPONENTES

4.1.1 TRANSAÇÃO

Sendo a unidade básica para um *testbench* UVM, já que esse é implementado a nível TLM, a transação foi definida a fim de facilitar a comunicação via AXI4_lite, foi decidido acrescentar, além dos sinais conhecidos do protocolo, um sinal "rw"para indicar se será feita uma solicitação de leitura (rw = 2'b10) ou escrita (rw = 2'b01).

```
class axi4lite_master_transaction #(int DATA_WIDTH = 32, int ADDR_SIZE = 32)
    extends uvm_sequence_item;

  rand bit [ADDR_SIZE-1:0]      awaddr;
  rand bit [2:0]                awprot;
  rand bit [DATA_WIDTH-1:0]     wdata;
  rand bit [(DATA_WIDTH)/8-1:0] wstrb;
  axi4_resp_eb                  bresp;

  rand bit [ADDR_SIZE-1:0]      araddr;
  rand bit [2:0]                arprot;
  bit      [DATA_WIDTH-1:0]     rdata;
  axi4_resp_eb                  rresp;

  rand bit [1:0]                rw;

  constraint constraints {
    bresp  != AXI4_RESP_B_EXOKAY;
    rresp  != AXI4_RESP_B_EXOKAY;
    rw     != 2'b0;
  }

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

...

endclass
```

AXI4lite_master_transaction.svh

Algumas *constraints* foram adicionadas a fim de eliminar alguns tipos de respostas que não são utilizadas no AXI4_lite apenas no AXI4, sendo esse uma versão mais completa e com mais sinais do AXI4_lite.

4.1.2 AGENTE

Sendo esse um dos blocos principais de um *testbench* UVM, foi desenvolvido para gerar, estimular e monitorar a interface AXI4_lite optou-se por criar apenas um agente já que a interface possui canais independentes para leitura e escrita.

```
class axi4lite_master_agent#(int DATA_WIDTH = 32, int ADDR_SIZE = 32) extends
    uvm_agent;
```

```
2   `uvm_component_param_utils(axi4lite_master_agent#(.DATA_WIDTH(DATA_WIDTH), .
      ADDR_SIZE(ADDR_SIZE)))
3
4   typedef axi4lite_master_transaction#(.DATA_WIDTH(DATA_WIDTH), .ADDR_SIZE(
      ADDR_SIZE)) master_transaction_type;
5   typedef uvm_sequencer#(master_transaction_type) master_sequencer;
6   typedef axi4lite_master_driver#(.DATA_WIDTH(DATA_WIDTH), .ADDR_SIZE(ADDR_SIZE))
       master_driver_type;
7   typedef axi4lite_master_monitor#(.DATA_WIDTH(DATA_WIDTH), .ADDR_SIZE(ADDR_SIZE)
      ) master_monitor_type;
8
9   uvm_analysis_port#(master_transaction_type) agt_resp_port;
10  uvm_analysis_port#(master_transaction_type) agt_req_port;
11
12
13  master_sequencer sequencer;
14  master_driver_type driver;
15  master_monitor_type monitor;
16
17  function new(string name, uvm_component parent);
18    super.new(name, parent);
19  endfunction
20
21  function void build_phase(uvm_phase phase);
22    super.build_phase(phase);
23
24    agt_resp_port = new("agt_resp_port", this);
25    agt_req_port = new("agt_req_port", this);
26
27
28    sequencer = master_sequencer::type_id::create("sequencer", this);
29    driver = master_driver_type::type_id::create("driver", this);
30    monitor = master_monitor_type::type_id::create("monitor", this);
31  endfunction
32
33  function void connect_phase(uvm_phase phase);
34    super.connect_phase(phase);
35    driver.seq_item_port.connect(sequencer.seq_item_export);
36    monitor.resp_tx_port.connect(agt_resp_port);
37    monitor.req_tx_port.connect(agt_req_port);
38  endfunction
39
40 endclass
```

AXI4lite_master_agent.svh

Como pode ser visto nosso código do agente é responsável apenas por criar e conectar os sub-componentes *Driver*, *Monitor* e *Sequencer*, para o ultimo foi utilizado a classe própria da biblioteca UVM, assim não faz-se necessário comentar sobre a implementação.

O desenvolvimento desse agente, assim como os sub-componentes, foi desenvolvido para ser completamente reutilizável, sendo essa uma das premissas da metodologia UVM, assim qualquer bloco que se comunique através de um protocolo AXI4_lite pode ser estimulado e monitorado por esse mesmo agente. Apesar de que o bloco verificado não

ter suporte para leitura e escrita em paralelo, o agente desenvolvido e seus subcomponentes estão preparados para tal funcionalidade.

4.1.3 *DRIVER*

Para desenvolvimento do *driver* se fez preciso um estudo detalhado do protocolo AXI4_lite, nesse componente deve-se garantir todas as relações de *handshake* entre o bloco a ser verificação e o ambiente de verificação, aqui que encontramos o primeiro desafio no processo de verificação, já que dependendo do protocolo a implementação do *driver* pode se tornar complexa, no código abaixo podemos ver as principais tarefas do *driver* desenvolvido.

```
1  typedef virtual axi4_lite_if axi4_lite_vif;
2
3  class axi4lite_master_driver#(int DATA_WIDTH = 32, int ADDR_WIDTH = 32) extends
       uvm_driver #(axi4lite_master_transaction);
4      `uvm_component_utils(axi4lite_master_driver)
5
6      ...
7
8      virtual task drive_write_address(axi4lite_master_transaction tr);
9          if(tr.rw[0]) begin
10             @(posedge vif.ACLK);
11             vif.awaddr  = tr.awaddr;
12             vif.awprot  = tr.awprot;
13             vif.awvalid = 1'b1;
14             if(!vif.awready) @(vif.awready);
15             @(posedge vif.ACLK);
16             vif.awvalid = 1'b0;
17         end
18     endtask : drive_write_address
19
20     virtual task drive_write_data(axi4lite_master_transaction tr);
21         if(tr.rw[0]) begin
22             @(posedge vif.ACLK);
23             vif.wdata  = tr.wdata;
24             vif.wstrb  = tr.wstrb;
25             vif.wvalid = 1'b1;
26             if(!vif.wready) @(vif.wready);
27             @(posedge vif.ACLK);
28             vif.wvalid = 1'b0;
29         end
30     endtask : drive_write_data
31
32     virtual task receive_write_response();
33         if(tr.rw[0]) begin
34             fork
35             @(posedge vif.ACLK iff (vif.awvalid && vif.awready));
36             @(posedge vif.ACLK iff (vif.wvalid  && vif.wready));
37             join
38             vif.bready = 1'b1;
39             if(!vif.bvalid) @(vif.bvalid);
40             @(posedge vif.ACLK);
```

```
41              vif.bready = 1'b0;
42          end
43      endtask : receive_write_response
44
45      virtual task drive_read_address(axi4lite_master_transaction tr);
46          if(tr.rw[1]) begin
47              @(posedge vif.ACLK);
48              vif.araddr  = tr.araddr;
49              vif.arprot  = tr.arprot;
50              vif.arvalid = 1'b1;
51              if(!vif.aready) @(vif.aready);
52              @(posedge vif.ACLK);
53              vif.arvalid = 1'b0;
54          end
55      endtask : drive_read_address
56
57      virtual task receive_read_data();
58          if(tr.rw[1]) begin
59              @(posedge vif.ACLK iff (vif.arvalid && vif.aready));
60              vif.rready = 1'b1;
61              if(!vif.rvalid) @(vif.rvalid);
62              @(posedge vif.ACLK);
63              vif.rready = 1'b0;
64          end
65      endtask : receive_read_data
66  endclass
```

AXI4lite_master_driver.svh

4.1.4 MONITOR

Assim como o *driver*, o monitor foi implementado obedecendo o protocolo AXI4_lite e pode ser também utilizado para qualquer bloco que se comunica através desse mesmo protocolo.

O monitor tem o papel de recuperar os sinais transmitidos através da interface transformando-os em transações, para isso deve-se monitorar os *handshakes* ocorridos entre os canais do barramento.

Pode ser visto entre as linhas 34 e 39 do código abaixo o monitoramento do canal de endereço de escrita no qual é avaliado o *handshake ready/valid*, dado a ocorrência do mesmo é armazenado em uma transação o endereço de escrita e o sinal de proteção.

```
1  class axi4lite_master_monitor #(int DATA_WIDTH = 32, int ADDR_SIZE = 32) extends
       uvm_monitor;
2    `uvm_component_param_utils(axi4lite_master_monitor#(.DATA_WIDTH(DATA_WIDTH), .
       ADDR_SIZE(ADDR_SIZE)))
3
4    ...
5
6    task run_phase(uvm_phase phase);
7      forever begin
8
```

```
9       @(posedge dut_if.ACLK) begin
10        if (!dut_if.ARESETn) begin
11          if (tx) begin
12            // abort
13            end_tr(tx);
14            // TODO: tx.abort = 1'b1; resp_tx_port.write(tx);
15          end
16        end
17        if (dut_if.arvalid && dut_if.arready) begin
18          tx = master_transaction_type::type_id::create("tx");
19          begin_tr(tx, "req");
20          tx.read = 1'b1;
21          tx.addr = dut_if.araddr;
22          tx.prot = dut_if.arprot;
23          end_tr(tx);
24          begin_tr(tx, "rsp");
25          req_tx_port.write(tx);
26        end
27        if (dut_if.rvalid && dut_if.rready) begin
28          tx.read = 1'b1;
29          tx.data = dut_if.rdata;
30          tx.resp = axi4_resp_eb'(dut_if.rresp);
31          end_tr(tx);
32          resp_tx_port.write(tx);
33        end
34        if (dut_if.awvalid && dut_if.awready) begin
35          tx = master_transaction_type::type_id::create("tx");
36          begin_tr(tx, "req");
37          tx.read = 1'b0;
38          tx.addr = dut_if.awaddr;
39          tx.prot = dut_if.awprot;
40
41        end
42        if (dut_if.wvalid && dut_if.wready) begin
43          tx.data = dut_if.wdata;
44          tx.wstrb = dut_if.wstrb;
45          end_tr(tx);
46          begin_tr(tx, "rsp");
47          req_tx_port.write(tx);
48        end
49        if (dut_if.bvalid && dut_if.bready) begin
50          tx.resp = axi4_resp_eb'(dut_if.bresp);
51          end_tr(tx);
52          resp_tx_port.write(tx);
53        end
54      end
55    end
56  endtask
57
58endclass
```

AXI4lite_master_monitor.svh

4.1.5 MODELO DE REFERÊNCIA

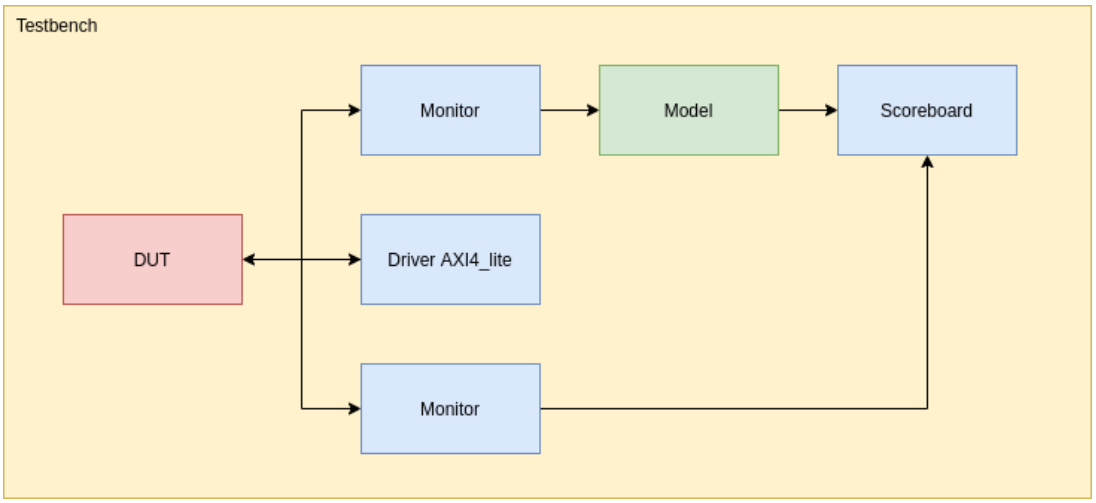
Para servir como modelo de referência para a verificação funcional foi utilizada a biblioteca crypto++, sendo essa uma biblioteca de classes C++ de código aberto e gratuita de algoritmos e esquemas criptográficos, assim obteve-se um conjunto métodos e classes para modelagem do bloco AES. Para acesso ao modelo escrito em C++ utilizou-se de uma funcionalidade própria do SystemVerilog chamada de *Direct Programming Interface* (DPI), essa permite chamadas diretas de função entre o SystemVerilog e o idioma estrangeiro, a transferência de dados entre os dois domínios se da por meio de argumentos de função e retorno.

A utilização da biblioteca Crypto++ além de ter minimizado o tempo de desenvolvimento do ambiente de verificação fez com que fosse possível ter um modelo muito mais confiável já que é uma biblioteca bem estruturada e com uma grande quantidade de desenvolvedores.

4.2 *TESTBENCH* COCOTB

No *testbench* em cocotb não há necessidade de criação de diversos blocos como feito em UVM, a arquitetura se resume ao modelo, *driver*, monitor e teste Vale ressaltar que para o desenvolvimento desse trabalho não se preocupou em garantir a verificação completa do bloco escolhido e sim perceber as características de um *testbench* em cocotb.

Figura 5 – Arquitetura de *testbench* cocotb desenvolvido.



Fonte: O próprio autor

Na Figura 5 vemos o diagrama de como ficou estruturado o *testbench* feito em cocotb, apesar de ser visto um bloco de *scoreboard* e dois blocos de monitores, a implementação de ambos ocorre no próprio código de teste, não se faz necessário a criação de

classes separadas já que diferente do UVM, que necessita de classe de comparador e classe de monitores, o processo do *scoreboard* é uma simples de comparação de duas transações.

Foi decidido criar a transação no ambiente em cocotb pois apesar de ser opcional se mostrou bem útil e facilitaria diversas operações dentro do processo de verificação. Abaixo podemos ver como ficou a implementação da transação, vale dar atenção a função *BinaryValue* que foi utilizada para converter as variáveis da transação para tipo binário fazendo ser possível operações com seus bits de forma independente.

```
...

#AXI4-Lite Transaction class
class AXI4LiteTransaction(Randomized):
    '''AXI4-Lite Transaction'''

    def __init__(self, DATA_WIDTH = 32, ADDR_SIZE = 32, ADDR_MAX = 20):

        Randomized.__init__(self)

        self.awaddr = None
        self.awprot = None
        self.wdata  = None
        self.wstrb  = None
        self.bresp  = BinaryValue()
        self.araddr = None
        self.arprot = None
        self.rdata  = BinaryValue()
        self.rresp  = BinaryValue()
        self.rw     = None

        #delay as a random variable
        self.add_rand("awaddr",list(range(ADDR_MAX*4)))
        self.add_rand("awprot",list(range(8)))
        self.add_rand("wstrb" ,list(range(2**4)))
        self.add_rand("araddr",list(range(ADDR_MAX*4)))
        self.add_rand("arprot",list(range(8)))
        self.add_rand("rw"    ,list([1,2]))

    def post_randomize(self):
        self.wdata = random.randint(0,0xFFFFFFFF)

        self.awaddr = BinaryValue(value=self.awaddr, n_bits=32, bigEndian=False)
        self.awprot = BinaryValue(value=self.awprot, n_bits=3, bigEndian=False)
        self.wdata  = BinaryValue(value=self.wdata, n_bits=32, bigEndian=False)
        self.wstrb  = BinaryValue(value=self.wstrb, n_bits=4, bigEndian=False)
        self.araddr = BinaryValue(value=self.araddr, n_bits=32, bigEndian=False)
        self.arprot = BinaryValue(value=self.arprot, n_bits=3, bigEndian=False)
        self.rw     = BinaryValue(value=self.rw, n_bits=2, bigEndian=False)

    def print_tr(self):
        print (" AWADDR = ", self.awaddr," \n", "AWPROT = ", self.awprot," \n",
    "WDATA = ", self.wdata," \n", "WSTRB = ", self.wstrb," \n", "BRESP = ", self.
    bresp," \n", "ARADDR = ", self.araddr," \n", "ARPROT = ", self.arprot," \n", "
    RDATA = ", self.rdata," \n", "RRESP = " , self.rresp," \n", "RW = ", self.rw)

```

```
44      def __eq__(self,other):
45          if (isinstance(other, AXI4LiteTransaction)):
46              return self.awaddr == other.awaddr and self.awprot == other.awprot
        and self.wdata  == other.wdata and self.wstrb  == other.wstrb and self.bresp
        == other.bresp and self.araddr == other.araddr and self.arprot == other.arprot
         and self.rdata  == other.rdata and self.rresp  == other.rresp and self.rw ==
        other.rw
47          else:
48              return False
49
50  ...
```

axi4_transaction.py

A estrutura de teste se da de uma forma bem simples já que todo o *testbench* fica reunido em apenas um arquivo, abaixo podemos ver um trecho do código do teste para o modo ECB do AES, nele fazemos a configuração do DUT setando o período do relógio e definindo o *reset*, após as configurações, instanciamos o nosso *driver* isso pode ser visto na linha 17 do código, aqui no *driver* que temos nossa primeira grande vantagem do uso do cocotb, enquanto com UVM a construção do *driver* foi o primeiro grande desafio dado que necessitou-se de um grande estudo sobre o protocolo para o seu desenvolvimento, para o cocotb a classe de *driver* do AXI4_lite já vem pré implementada na própria biblioteca, sendo preciso apenas fazer uma importação da mesma.

```
1   ...
2
3   @cocotb.test()
4   async def test_ECB(dut):
5       """ Test AES-IP mode ECB"""
6
7
8       aes_model = AES_model()
9
10      cycles = 0
11      match = 0
12      mismatch = 0
13      setup_dut(dut)
14      await reset(dut)
15
16      axim = AXI4LiteMaster(dut, "S_AXI", dut.S_AXI_ACLK)
17      tr = AXI4LiteTransaction()
18
19      tr_out_dut = AXI4LiteTransaction()
20      tr_out_model = AXI4LiteTransaction()
21
22      while (True):
23
24          while (True):
25              tr.randomize()
26              if (tr.awaddr.integer < 0x48):
27                  break
28
29          tr_out_dut = copy.copy(tr)
30
```

```
31          if ((cycles % 100) != 0):
32
33              if (tr.rw.integer == 1):
34                  bresp = await axim.write(tr.awaddr, tr.wdata, tr.wstrb)
35                  aes_model.put(tr,tr_out_model)
36                  tr_out_dut.bresp.integer = bresp
37              elif (tr.rw.integer == 2):
38                  rdata,rresp = await axim.read(tr.araddr)
39                  aes_model.put(tr,tr_out_model)
40                  tr_out_dut.rdata.integer = rdata
41                  tr_out_dut.rresp.integer = rresp
42              else:
43                  pass
44
45          else:
46
47              tr.awaddr.integer = 0x48
48              tr.wdata[31]      = 1 # Set bit start
49              tr.wdata[2:0]     = 0 # Set ECB mode
50              tr.wdata[3]       = 0 # Set encrypt mode
51              tr.wstrb.integer  = 0xF
52              tr.rw.set_value(1)
53              bresp = await axim.write(tr.awaddr, tr.wdata, tr.wstrb)
54              aes_model.put(tr,tr_out_model)
55
56          if(tr_out_model == tr_out_dut):
57              match += 1
58          else:
59              mismatch += 1
60
61              print("MISMATCH - expected output:")
62              tr_out_model.print_tr()
63              print (" received:")
64              tr_out_dut.print_tr()
65
66          cycles += 1
67          if (cycles >= 10000):
68              break
69
70      if(mismatch == 0):
71          raise TestSuccess("========== PASS ==========")
72      else:
73          raise TestFailure("========== FAIL ==========")
```

test_aes.py

Nesse código temos um teste de encriptação do modo ECB, para realizar o teste são geradas transações randômicas a fim de preencher os registradores internos do bloco, após o preenchimento é gerada uma transação direcionada de escrita no registrador de comando definindo a configuração de modo, encriptação ou decriptação e define para 1 o bit de *start*, todas as transações geradas são enviadas para o modelo e DUT para que ao fim suas saídas sejam comparadas.

Para o modelo de referência foi utilizada uma biblioteca do Python chamada de

Crypto, onde nessa temos um modelo de auto nível do AES, contendo a maioria dos modos de operação. Foi criado uma classe para simular o modelo junto ao banco de registradores.

O funcionamento desse modelo se da pela utilização de três funções onde apenas uma delas, a função *put*, é utilizada pelo teste, ela tem o trabalho de receber a transação e caso for de leitura retornar o dado a ser lido, caso seja de escrita guardar o dado no banco de registrador, se for feito uma escrita no registrador de comando uma operação é iniciada podendo ser ela de encriptação ou decriptação em cada um dos modos do AES.

Parte desse modelo desenvolvido pode ser visto no código a seguir.

```
...

class AES_model:

    def __init__(self):

        self.reg_bank = [None] * 20

        for i in range(20):
            self.reg_bank[i] = BinaryValue(value=0,n_bits=32, bigEndian=False,
    binaryRepresentation=BinaryRepresentation.TWOS_COMPLEMENT)

    def _write(self, addr, data, strb):
        for i in range(4):
            if strb[i]:
                self.reg_bank[floor(addr/4)][i*8+7:i*8] = data[i*8+7:i*8].binstr

    def _read(self, addr):
        return self.reg_bank[floor(addr/4)]

    def put(self, tr_in, tr_out):

        tr_out = copy.copy(tr_in)
                    ...

                        if mode ==  0: #ECB
                            encryption_suite = AES.new(key.buff, AES.MODE_ECB)
                            cipher = encryption_suite.encrypt(plainText.buff)

                            cipher_data = BinaryValue(n_bits=32*4, bigEndian=
    False)

                            cipher_data <= cipher
                            for i in range(4):
                                for j in range(4):
                                    self.reg_bank[i+12][j*8+7:j*8] = cipher_data
    [(i*32)+(3-j)*8+7:(i*32)+(3-j)*8].binstr
                        elif mode == 1: #CBC
                            ...
                    ...
```

aes_model.py

4.3 ANÁLISE COMPARATIVA UVM VS COCOTB

Com a implementação dos *testbenches* utilizando ambas as metodologias expostas nesse documento é possível perceber algumas diferenças fundamentais entre elas, essas diferenças impactam a forma como o verificador constrói seus códigos e seus testes. É importante ressaltar que o cocotb é uma ferramenta ainda em construção e que não conta com o aporte ferramental que o UVM dispõe, entretanto ele traz algumas ideias novas e interessantes para a verificação de *hardware*.

A principal diferença observada entre as metodologias é o tratamento com tipos binários. Com o suporte nativo a esses tipos, o SystemVerilog dispõe de mecanismos mais naturais a linguagem para tratar com variáveis desse tipo. Devido a tipagem fixa, diferente do Python que a tipagem é dinâmica, a atribuição de valores e a quebra de *arrays* de *bits* é facilitada no UVM.

No cocotb a utilização de tipos binários vem a partir da biblioteca `BinaryValue`. A utilização da biblioteca é extremamente penosa em termos de velocidade de simulação, a medida que mais operações são feitas com os *bits* das variáveis. Muitas vezes a utilização de máscaras de *bits* é bastante comum para alterar diretamente alguns valores específicos dentro de uma palavra maior. Essa utilização torna os códigos menos claros em termos de intenção, sendo um ponto a favor da utilização da metodologia UVM. Um exemplo de uso da biblioteca `BinaryValue` aplicada ao modelo de referência do AES é demonstrada abaixo.

```
1 key = BinaryValue(n_bits=32*4, bigEndian=False)
2 for i in range(4):
3     for j in range(4):
4         key = BinaryValue(key.binstr + self.reg_bank[i][j*8+7:j*8].binstr)
```

Um fator de agilidade no desenvolvimento em cocotb é a disponibilidade de extensões e módulos. Existem uma variadade de *drivers* e *monitors* que facilitam a realização de operações com os sinais do bloco e possibilita uma maior flexibilidade na criação da estrutura de testes e de *testbench*. A obtenção desses módulos em SystemVerilog é bastante cara, e por muitas vezes envolve a aquisição de *Verification IPs* dificultando e encarecendo o custo do projeto.

Ambas as metodologias possuem ferramentas e possibilitam a verificação completa, entretanto o cocotb peca em termos de escalabilidade do projeto. Em projetos maiores, a utilização da ferramenta em Python necessitará de muitas adaptações e desenvolvimentos que já são padrão quando se trata de UVM. Algumas necessidades básicas de projetos de verificação de *hardware* não são trivialmente solucionadas com o cocotb, a exemplo do gerenciamento de cobertura e de regressões. Como a biblioteca ainda está em desenvolvimento, espera-se que num futuro próximo todas essas necessidades sejam atendidas e torne mais fácil a sua utilização em ambientes de verificação mais complexos.

5 CONSIDERAÇÕES FINAIS

Através do desenvolvimento desse trabalho percebeu-se que a complexidade da metodologia UVM é extremamente maior que a do cocotb, um dos fatores que caracteriza isso é extensa especificação da linguagem SystemVerilog e grande quantidade de classes presente na biblioteca UVM chegando a mais de 300 classes.

Um outro ponto é o fato de ser improvável que os alunos de graduação tenham experiência anterior em SystemVerilog ou UVM, se fazendo um necessário um longo período de adaptação.

Por ser desenvolvido com a linguagem Python, cocotb tem tido uma boa aceitação já que Python é simples, fácil de aprender e muito poderoso, além de uma grande biblioteca padrão fornecendo muitos recursos ao verificador.

Pode-se concluir que a metodologia UVM é poderosa, mas complicada, sua implementação demanda um tempo maior frente ao cocotb, porém ainda propicia uma confiança maior por parte do verificador, parte disso se da pelo fato de que a metodologia cocotb ser recente e ainda estar em desenvolvimento. O processo de verificação é algo caro e de extrema importância, a metodologia UVM se mostrou nos últimos anos muito eficaz e consistente, porém metodologias baseada em Python devem crescer cada vez mais e serem escolhidas para a verificação de uma parcela considerável de projetos de *design*.

REFERÊNCIAS

ABRANTES, R. F. R. *Implementação em Hardware de Encriptação e decriptação com utilização do Advanced Encryption Standard (AES)*. [S.l.], 2018. 24

CHIPVERIF. *UVM Tutorial*. 2020. Disponível em: <https://www.chipverify.com/uvm/uvm-tutorial>. 13

CIEPLUCHA, M.; PLESKACZ, W. New constrained random and metric-driven verification methodology using python. In: *Proceedings of the design and verification conference and exhibition US (DVCon)*. [S.l.: s.n.], 2017. 17

VENTURES, P. *COCOTB 1.0 documentation*. 2014. Disponível em: <http://cocotb.readthedocs.org/en/latest/introduction.html.> Acesso em: 11 de Novembro de 2020. 18