Aluna: Jaqueline Ferreira de Brito
Data: 21 de Março de 2025

RELATÓRIO VERIFICAÇÃO FORMAL E FUNCIONAL SEQUENCE, DRIVER, INTERFACE, MONITOR E DUT

Este relatório apresenta a continuação de uma série de atividades abrangendo a evolução da primeira simulação UVM explorada em aula. A simulação inicial era composta apenas pelos componentes fundamentais: fonte (source), transação (transaction) e sorvedouro (sink). Nesta versão, esses componentes foram reorganizados e integrados à classe test, tornando o ambiente de teste mais modular e flexível.

A arquitetura atual implementa uma estrutura de verificação completa, incluindo Sequence (a_sequence) que gera padrões de estímulos randomizados via macro uvm_do e Sequencer que coordena o fluxo de transações através do protocolo TLM. O ambiente também incorpora componentes de interface como o Driver, responsável por converter transações de alto nível em sinais de baixo nível para o DUT, e o Monitor que observa a interface, detectando atividades válidas e convertendo-as de volta em transações para análise. A Interface Virtual estabelece a fronteira entre o testbench e o DUT (Device Under Test), que implementa uma operação de adição (soma 250 ao valor de entrada).

Além disso, foram implementadas variações para incorporar conceitos como ambientes de teste (env), pacotes (package), filas FIFO (fifo), modelos de referência (reference model), e mais recentemente, a evolução para analysis source, cobertura e agentes, aprimorando a estrutura e a eficiência da simulação. Esta abordagem permite a verificação abrangente do comportamento do DUT através da comparação entre suas saídas reais e as esperadas geradas pelo modelo de referência.

1. INTRODUÇÃO

SEQUENCE (Sequência)

Uma sequência UVM é responsável por gerar estímulos de teste de forma organizada e controlada. Derivada da classe uvm_sequence, ela implementa a tarefa body() que contém a lógica para produzir transações (itens de sequência). As sequências podem ser parametrizadas e configuradas para gerar diferentes padrões de estímulos, permitindo a criação de cenários de teste diversos sem modificar a estrutura do ambiente de verificação.

DRIVER (Atuador)

O driver converte transações abstratas em sinais de hardware. Recebendo itens de sequência do sequenciador através de seq_item_port.get_next_item(), o driver traduz esses objetos em operações de baixo nível nos sinais da interface. Após processar cada transação, ele sinaliza a conclusão com seq_item_port.item_done(). Esta separação entre geração de estímulos e aplicação aos sinais permite maior flexibilidade e reutilização.

INTERFACE

A interface SystemVerilog encapsula os sinais físicos que conectam o ambiente de verificação ao DUT. Ela define os sinais de dados, sinais de controle (como valid e ready), além de clock e reset. A interface atua como um canal de comunicação bidirecional, permitindo que drivers apliquem estímulos e monitores observem atividades. Geralmente instanciada no módulo top, a interface é compartilhada entre componentes através de uma base de dados.

MONITOR

O monitor observa passivamente as atividades na interface e extrai informações significativas. Ele reconstrói transações a partir dos sinais de hardware, permitindo que o ambiente de verificação analise o comportamento do DUT. Quando detecta uma transação válida (tipicamente na borda de clock com sinais de controle ativos), o monitor captura os valores, cria um objeto de transação e o envia através de sua porta de análise para scoreboard, model de referência ou outros componentes.

DUT (Device Under Test)

O DUT é o hardware sendo verificado, tipicamente implementado em RTL (Verilog/VHDL). Ele é instanciado no módulo top e conectado às interfaces de entrada e saída. O DUT é tratado como uma "caixa-preta" pelo ambiente de verificação, que interage

com ele apenas através das suas interfaces. O propósito do ambiente UVM é verificar se, para um determinado conjunto de entradas, o DUT produz as saídas esperadas conforme a especificação.

Esta arquitetura modular do UVM permite construir ambientes de verificação escaláveis, reutilizáveis e mantidos com facilidade, adequados para validar desde componentes simples até sistemas complexos de hardware.

2. DESENVOLVIMENTO

DESCRIÇÃO DOS PRINCIPAIS COMPONENTES

1. j_jb (Transação)
    - Define os dados que fluem pelo ambiente
    - Contém campo randomizado: $1 \leq j < 40$
2. Sequence & Sequencer
    - Geram e coordenam o envio de transações
    - A sequência implementada gera valores aleatórios continuamente
3. Driver
    - Converte transações em sinais físicos
    - Aplica os estímulos ao DUT através da interface virtual
4. Monitor
    - Observa a interface quando 'valid' está ativo
    - Converte sinais físicos de volta em transações
5. RefMod (Modelo de Referência)
    - Replica a funcionalidade do DUT: j + 250
    - Fornece resultados esperados para comparação
6. Sink & Drain
    - Sink: Recebe resultados esperados do modelo de referência
    - Drain: Consome transações pendentes para evitar bloqueios

Módulo top.sv:

- Define sinais de baixo nível (clock, reset) necessários para o funcionamento do DUT
- Cria o test bench físico, incluindo interfaces e conectando-as ao DUT
- permite que os componentes UVM acessem os sinais físicos do DUT através da interface

Após configuração, run_test("test") inicia o fluxo UVM que:

- Constrói a hierarquia de componentes UVM
- Executa as fases UVM (build, connect, run, etc)
- Gerencia sequências e transações

```
// Top-level module: Ponto de entrada do testbench UVM
module top;
```

```
  // Importação de pacotes UVM e pacotes personalizados
  import uvm_pkg::*;
  import test_pkg::*;

  // Gerador de clock (período = 60 unidades de tempo)
  logic clock;
  initial begin
       clock = 0;
       forever #30 clock = ~clock;
  end

  // Gerador de reset: ativo por 2 ciclos de clock
  logic reset;
  initial begin
       reset = 1;
       repeat(2) @(negedge clock);
       reset = 0;
  end

  // Instâncias de interface e DUT
  j_if in(.*);   // Interface de entrada
  j_if out(.*);  // Interface de saída
  dut d(.*); // Design Under Test

  // Bloco de inicialização para configuração do ambiente
  initial begin
       // Configuração de captura de formas de onda (específico do simulador)
       `ifdef INCA
       $shm_open("waves.shm");
       $shm_probe("AS");
       `endif
       `ifdef VCS
       $vcdpluson;
       `endif
       `ifdef QUESTA
       $wlfdumpvars();
       `endif

       // Registro da interface virtual para acesso pelos componentes UVM
       uvm_config_db #(virtual j_if)::set(null,
"uvm_test_top.env_JaquelineB.agent_h_Jaqueline.*", "j_vi", in);

       // Inicia o fluxo de verificação UVM
       run_test("test");
  end
endmodule
```

Módulo agent.svh

O agent é um componente crítico que coordena três elementos essenciais:

1. Sequencer (sequencer_h_JaqueB)
    - Gera sequências de transações (estímulos)
    - Funciona como uma "fábrica de transações" para o teste
2. Driver (driver_h_JaqueB)
    - Recebe transações do sequencer
    - Converte transações em sinais de baixo nível para a interface física
    - Estimula o DUT com valores específicos
3. Monitor (monitor_h_JaqueB)
    - Observa os sinais na interface
    - Converte sinais de volta para transações de alto nível

```
// Classe agent estende uvm_agent, coordena os componentes de estímulo e
monitoração
// Responsável por instanciar e conectar sequencer, driver e monitor em um grupo
funcional
class agent extends uvm_agent;
  `uvm_component_utils(agent)  // Macro que permite registro desta classe na
fábrica UVM

   // Porta de análise que envia transações observadas para outros componentes
(como scoreboard ou coverage)
   uvm_analysis_port #(j_jb) out;

   // Declaração dos três componentes principais que formam um agente completo
   sequencer sequencer_h_JaqueB;        // Gera sequências de transações
   driver_master driver_h_JaqueB;       // Converte transações em sinais de
interface
   monitor monitor_h_JaqueB;            // Observa interface e cria transações

   // Construtor padrão UVM
   function new(string name, uvm_component parent);
       super.new(name, parent);
   endfunction

   // Fase de construção - cria instâncias de todos os componentes
   function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       out = new("Saida", this);  // Cria porta de análise
       // Cria instâncias dos componentes usando a fábrica UVM (permite
substituição)
       sequencer_h_JaqueB = sequencer::type_id::create("sequencer_h_JaqueB",
this);
       driver_h_JaqueB = driver_master::type_id::create("driver_h_JaqueB",
this);
       monitor_h_JaqueB = monitor::type_id::create("monitor_h_JaqueB", this);
   endfunction

   // Fase de conexão - estabelece comunicação entre componentes
   function void connect_phase(uvm_phase phase);
```

```
      monitor_h_JaqueB.out.connect(out);  // Conecta saída do monitor à porta
do agente
      // Conecta driver ao sequencer para permitir fluxo de transações

driver_h_JaqueB.seq_item_port.connect(sequencer_h_JaqueB.seq_item_export);
   endfunction

endclass
```

Módulo driver.svh

Esta implementação do driver usa uma abordagem eficiente com duas tarefas paralelas (fork/join):

1. reset_signals() - Responde imediatamente a qualquer ativação do reset
2. get_and_drive() - Processa transações sequencialmente quando o reset está inativo

O protocolo de handshake (valid=1) é aplicado apenas durante um ciclo de clock para cada transação, seguido de um ciclo com valid=0.

```
// Driver Master: Converte transações de alto nível em sinais para a interface
class driver_master extends uvm_driver #(j_jb);
  `uvm_component_utils(driver_master)  // Registro na fábrica UVM

  // Construtor padrão com nome e referência ao pai
  function new(string name, uvm_component parent);
      super.new(name, parent);
  endfunction

  // Interface virtual que será conectada à interface física
  virtual j_if j_vi;

  // Fase de construção: obtém a instância da interface do banco de dados UVM
  function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      assert( uvm_config_db #(virtual j_if)::get(this, "", "j_vi", j_vi) );  //
Falha se interface não encontrada
  endfunction

  // Fase de execução: gerencia as tarefas principais do driver
  task run_phase(uvm_phase phase);
      // Inicializa sinais com valores indefinidos
      j_vi.valid <= 'x;
      j_vi.j  <= 'x;

      fork
      reset_signals();  // Monitora e responde ao sinal de reset
      get_and_drive();  // Processa transações do sequencer
      join
  endtask
```

```
  // Gerencia os sinais durante o reset
  task reset_signals();
       forever begin
       wait (j_vi.reset === 1);         // Espera reset ativo
       j_vi.valid <= 0;                 // Desativa valid
       j_vi.j  <= 'x;                   // Coloca dados em estado indefinido
       @(negedge j_vi.reset);           // Espera fim do reset
       end
  endtask

  // Obtém transações do sequencer e as converte em sinais para o DUT
  task get_and_drive();
       j_jb tr_sequencer;  // Transação recebida do sequencer

       forever begin
       wait (j_vi.reset === 0);               // Aguarda reset inativo
       seq_item_port.get_next_item(tr_sequencer); // Obtém próxima transação

       // Aplica os sinais na interface no próximo ciclo de clock
       @(posedge j_vi.clock);
       j_vi.valid <= 1;                 // Ativa sinal valid
       j_vi.j <= tr_sequencer.j;        // Aplica valor da transação

       seq_item_port.item_done();       // Notifica conclusão da transação

       @(posedge j_vi.clock);
       j_vi.valid <= 0;                 // Desativa valid por um ciclo
       @(posedge j_vi.clock);           // Espera mais um ciclo antes da próxima
transação
       end
  endtask
endclass
```

Módulo analysis_source.svh

Esta classe atua como geradora independente de estímulos, criando transações aleatórias em intervalos regulares sem necessidade de um sequencer. O fluxo de transações é contínuo e paralelo às outras atividades do testbench, proporcionando estímulos constantes durante toda a simulação.

```
// Analysis Source: Gerador de transações aleatórias para o ambiente de
verificação
class analysis_source extends uvm_component;
   `uvm_component_utils(analysis_source)  // Registro na fábrica UVM

   // Porta de análise para envio de transações para múltiplos consumidores
   uvm_analysis_port #(j_jb) out;

   // Construtor com nome personalizado e referência ao componente pai
```

```
   function new(string name = "source by Jaqueline_Brito", uvm_component
parent);
       super.new(name, parent);
       out = new("Saída", this);  // Inicializa a porta de análise
   endfunction

   // Fase de execução: gera continuamente transações aleatórias
   task run_phase(uvm_phase phase);
       j_jb tr;

       forever begin
       #15;  // Intervalo de 15 unidades de tempo entre transações

       // Cria e randomiza uma nova transação
       tr = j_jb::type_id::create("tr", this);
       assert(tr.randomize());  // Garante randomização bem-sucedida

       `bvm_begin_tr(tr)  // Inicia rastreamento da transação
       `uvm_info("SOURCE", "Sending transaction by Aluna: Brito, Jaqueline",
UVM_LOW)
       out.write(tr);  // Distribui transação para todos os componentes
conectados
       end
   endtask
endclass
```

Módulo DUT.sv

Este DUT opera como um adicionador síncrono com as seguintes características:

1. **Reset síncrono** - Inicializa os registradores na borda do clock
2. **Protocolo de handshake** - Usa sinais valid para indicar dados válidos
3. **Função de processamento** - Adiciona o valor constante 250 à entrada
4. **Latência de um ciclo** - A saída aparece um ciclo após a entrada

O código demonstra um design RTL simples seguindo boas práticas de projetos síncronos.

```
// DUT: Circuito simples que adiciona 250 ao valor de entrada quando válido
module dut (j_if.inp in,         // Interface de entrada usando modport inp
            j_if.outp out);      // Interface de saída usando modport outp

  // Lógica síncrona executada na borda de subida do clock
  always_ff @(posedge in.clock)
       if (in.reset) begin
       // Durante reset: inicializa saídas com zeros
       out.valid <= 0;
       out.j <= 0;
       end
       else if(in.valid) begin
```

```
        // Quando entrada válida: processa e ativa saída
        out.valid <= 1;
        out.j <= in.j + 250;  // Operação principal: soma 250
        end
        else
        // Entrada inválida: mantém saída inválida
        out.valid <= 0;

endmodule
```

Módulo coverage_in.svh

Esta classe de cobertura instrumenta o ambiente para verificar se os valores de entrada j cobrem adequadamente quatro intervalos de valores (0-9, 10-19, 20-29, 30-39). A cobertura só será considerada completa quando cada intervalo for exercitado pelo menos quatro vezes durante o teste.

```
// Coverage In: Monitora cobertura dos valores na entrada do DUT
class coverage_in extends bvm_cover #(j_jb);
   `uvm_component_utils(coverage_in)

   // Define grupo de cobertura para análise de valores j
   covergroup transaction_covergroup;
       option.per_instance = 1;  // Conta cobertura separadamente para cada
instância

       // Monitora distribuição dos valores de j em intervalos de 10 unidades
       coverpoint coverage_transaction.j {
       bins d[4] = {[0:9], [10:19], [20:29], [30:39]};  // 4 faixas de valores
       option.at_least = 4;  // Requer mínimo de 4 ocorrências em cada faixa
       }
   endgroup

   // Configura métodos auxiliares para processar transações j_jb
   `bvm_cover_utils(j_jb)
endclass
```

Módulo env.svh

O ambiente implementa um padrão de verificação "scoreboard" onde as entradas são:

1. Enviadas ao DUT através do agente
2. Simultaneamente processadas por um modelo de referência
3. Os resultados do DUT e do modelo são comparados no sink para detectar discrepâncias.

```
// Environment: Contém e coordena todos os componentes do testbench
class env extends uvm_env;
   `uvm_component_utils(env)
```

```
    // Componentes principais do ambiente de verificação
    agent agent_h_Jaqueline;            // Gera e aplica estímulos
    coverage_in coverage_in_Jaqueline;  // Monitora cobertura dos estímulos
    uvm_tlm_analysis_fifo #(j_jb) agent_refmod;  // Buffer entre agente e modelo
de referência
    refmod refmod_jaqueline;            // Modelo de referência para predição
    sink sink_jaqueline;                // Coletor para comparação e checagem

    // Construtor padrão
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    // Fase de construção: cria todas as instâncias de componentes
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agent_h_Jaqueline = agent::type_id::create("agent_h_Jaqueline", this);
        coverage_in_Jaqueline =
coverage_in::type_id::create("coverage_in_Jaqueline", this);
        agent_refmod = new("agent_refmod", this);
        refmod_jaqueline = refmod::type_id::create("refmod_jaqueline", this);
        sink_jaqueline = sink::type_id::create("sink_jaqueline", this);
    endfunction

    // Fase de conexão: estabelece o fluxo de dados entre componentes
    function void connect_phase(uvm_phase phase);
        // Conecta saída do agente ao monitor de cobertura
        agent_h_Jaqueline.out.connect(coverage_in_Jaqueline.analysis_export);

        // Estabelece o caminho de checagem:
        agent_h_Jaqueline.out.connect(agent_refmod.analysis_export);  // Agente
→ FIFO
        refmod_jaqueline.in.connect(agent_refmod.get_export);      // FIFO →
Modelo de referência
        refmod_jaqueline.out.connect(sink_jaqueline.in);           // Modelo →
Sink
    endfunction
endclass
```

Módulo interface.sv

Esta interface implementa um protocolo simples com:

1. **Sinais de controle globais** (clock, reset) que são sempre inputs
2. **Sinal de handshake** (valid) que indica quando os dados são válidos
3. **Canal de dados** (j[7:0]) para transferência de valores de 8 bits
4. **Modports** que definem claramente a direção dos sinais para entrada e saída

Os modports permitem que a mesma definição de interface seja usada tanto para a entrada

quanto para a saída do DUT, garantindo consistência de sinais.

```
// Interface j_if: Define os sinais para comunicação com o DUT
interface j_if (input logic clock, reset);

  logic valid;      // Sinal de handshake: dados válidos quando valid=1
  logic [7:0] j;    // Barramento de dados de 8 bits

  // Modport para entrada: todos os sinais são recebidos pelo DUT
  modport inp (input clock, reset, input valid, input j);

  // Modport para saída: sinais valid e j são produzidos pelo DUT
  modport outp (input clock, reset, output valid, output j);

endinterface
```

Módulo monitor.svh

O monitor é um componente passivo (não-intrusivo) que observa a interface do DUT e converte sinais de baixo nível em transações de alto nível. Seu papel fundamental é "traduzir" sinais elétricos em objetos de dados estruturados para análise e verificação.

Componentes principais:

1. Porta de Análise: out é uma porta UVM que distribui transações para qualquer número de componentes inscritos (como scoreboard, cobertura, etc.)
2. Interface Virtual: j_vi é a referência que permite acesso aos sinais físicos da interface
3. Ciclo de Captura: O monitor executa continuamente um loop que:
    - Cria uma nova transação (j_jb::type_id::create)
    - Espera pelo momento certo para capturar (@(posedge j_vi.clock iff (j_vi.valid)))
    - Extrai dados dos sinais da interface (tr.j = j_vi.j)
    - Distribui a transação para análise (out.write(tr))

O fluxo de execução:

1. Quando uma transação válida é detectada (sinal valid ativo), o monitor captura o valor de j
2. Ele encapsula esse valor em um objeto de transação j_jb
3. Envia essa transação para a porta de análise, onde outros componentes podem processá-la
4. Volta ao início e espera pela próxima transação válida

Este monitor implementa um protocolo "valid-ready" simplificado onde apenas observa quando o sinal valid está ativo na borda de subida do clock, capturando o valor correspondente do sinal j.

```
// Monitor: Observa a interface e converte sinais em transações
class monitor extends uvm_monitor;
   `uvm_component_utils(monitor)

   // Porta de análise para distribuir transações observadas
   uvm_analysis_port #(j_jb) out;

   // Interface virtual para acessar os sinais físicos
   virtual j_if j_vi;

   // Construtor do componente
   function new(string name, uvm_component parent);
       super.new(name, parent);
       out = new("Saida", this);
   endfunction: new

   // Obtém a interface a partir do banco de configurações
   function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       assert( uvm_config_db #(virtual j_if)::get(this, "", "j_vi", j_vi) );
   endfunction

   // PROBLEMA: Não captura o comportamento durante o reset
   task run_phase(uvm_phase phase);
       j_jb tr;
       forever begin
       wait (j_vi.reset === 0);  // Espera o reset terminar antes de monitorar
       tr = j_jb::type_id::create("tr");

       // Aguarda próxima borda de subida do clock com valid=1
       @(posedge j_vi.clock iff (j_vi.valid));

       `bvm_begin_tr(tr)  // Inicia gravação da transação
       tr.j = j_vi.j;      // Captura o valor atual
       out.write(tr);      // Envia transação para análise
       end
   endtask
endclass
```

Módulo refmod.svh

O modelo de referência (refmod) atua como uma implementação de referência do comportamento esperado do DUT. Suas principais funções são:

1. Predição de Resultados: Processa as mesmas entradas que o DUT recebe e calcula os resultados esperados (j + 250)
2. Base para Verificação: Fornece resultados esperados para comparação com as saídas reais do DUT
3. Sincronização: Simula os atrasos de tempo que ocorreriam no hardware real (#60 unidades de tempo)

O refmod opera em um loop contínuo, obtendo transações de entrada, processando-as conforme a especificação funcional (adicionando 250), e enviando os resultados para comparação no componente sink ou scoreboard.

Este modelo específico implementa uma operação simples de adição, mas a mesma abordagem pode ser expandida para modelar comportamentos mais complexos.

```
class refmod extends uvm_component;
   `uvm_component_utils(refmod)

   // Interfaces TLM para comunicação com outros componentes
   uvm_get_port #(j_jb) in;                    // Porta para obter transações de
entrada
   uvm_blocking_put_port #(j_jb) out;          // Porta para enviar resultados
processados

   // Construtor
   function new(string name, uvm_component parent=null);
      super.new(name, parent);
      in  = new("Entrada", this);
      out = new("Saída", this);
   endfunction : new

   // Fase de execução - implementa a lógica do modelo
   task run_phase(uvm_phase phase);
      j_jb tr_in, tr_out;  // Transações de entrada e saída

      forever begin
      // Obtém próxima transação de entrada (bloqueia se não houver)
      in.get(tr_in);

      #60;  // Simula o tempo de processamento do DUT
      `bvm_end_tr(tr_in);  // Finaliza registro da transação de entrada

      // Cria e processa transação de saída
      tr_out = j_jb::type_id::create("Jaque_out", this);
      tr_out.j = tr_in.j + 250;  // Replica a operação do DUT: adiciona 250
      `bvm_begin_tr(tr_out);  // Inicia registro da transação de saída

      #60;  // Atraso adicional antes de enviar o resultado
      out.put(tr_out);  // Envia transação de saída para o próximo componente
      end
   endtask
endclass
```

Módulo sequence.svh

Esta sequência implementa um gerador simples que:

1. **Gera transações infinitamente**: O loop forever garante que as transações sejam continuamente produzidas até que o teste seja terminado
2. **Utiliza randomização automática**: O macro uvm_do realiza várias operações em

uma única linha:

- Cria uma nova instância da transação j_jb
- Randomiza seus campos conforme restrições definidas na classe j_jb
- Envia a transação para o sequencer
- Aguarda sua conclusão antes de criar a próxima

3. **Simplicidade eficiente**: Sem comportamento condicional ou variações de padrão, esta sequência produz um fluxo constante de valores aleatórios para testar o DUT sob diversas condições

A sequência é um elemento fundamental na metodologia UVM, fornecendo os estímulos que exercitam o design sob teste.

```
// Sequence: Gera um fluxo contínuo de transações randomizadas
class a_sequence extends uvm_sequence #(j_jb);
  `uvm_object_utils(a_sequence)  // Registro na fábrica UVM

  // Construtor padrão
  function new(string name = "a_sequence");
       super.new(name);
  endfunction: new

  // Corpo da sequência: define o comportamento de geração de transações
  task body;
       j_jb tr;

       forever begin
       `uvm_do(tr)  // Cria, randomiza e envia transação para o sequencer
       end
  endtask
endclass
```

Módulo test.svh

Esta classe representa o nível mais alto da hierarquia UVM e orquestra toda a verificação:

1. Configuração: Cria o ambiente de verificação completo (env) que contém todos os componentes necessários

2. Execução: Durante a fase run, cria uma sequência de estímulos (a_sequence) e a associa ao sequenciador dentro do agente

3. Controle de Execução: Como raiz da hierarquia, o teste determina quando a simulação começa e termina

A classe implementa um teste simples que envia um fluxo contínuo de transações aleatórias para o DUT via a sequência a_sequence, proporcionando cobertura ampla sem objetivos específicos de teste.

```
// Classe de teste: Configura e executa o ambiente de verificação
class test extends uvm_test;
   `uvm_component_utils(test)

   // Instância do ambiente de verificação
   env env_h;

  // Construtor padrão
  function new(string name, uvm_component parent);
       super.new(name, parent);
  endfunction

  // Fase de construção: cria o ambiente
  function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       env_h = env::type_id::create("env_JaquelineB", this);
  endfunction

  // Fase de execução: cria e inicia a sequência de estímulos
  task run_phase(uvm_phase phase);
       a_sequence seq;
       seq = a_sequence::type_id::create("sequencer_h_JaqueB");

       // Inicia a sequência no sequenciador do agente
       seq.start(env_h.agent_h_Jaqueline.sequencer_h_JaqueB);
  endtask

endclass
```

Módulo test_pkg.sv

Este pacote implementa uma estrutura UVM completa para verificação, seguindo um fluxo típico:

1. Definição da transação base (trans.svh) que circula pelo ambiente
2. Componentes de estímulo (sequence.svh, driver.svh) que geram e aplicam entradas
3. Componentes de observação (monitor.svh, coverage_in.svh) que capturam atividades
4. Componentes de verificação (refmod.svh, sink.svh) que validam resultados
5. Estrutura de orquestração (agent.svh, env.svh, test.svh) que coordena o teste

```
// Inclusão dos arquivos de macros necessários
`include "uvm_macros.svh"        // Macros padrão da metodologia UVM
`include "bvm_macros.svh"        // Macros específicos do Brazil-IP/UFCG

// Definição do pacote de teste que encapsula todo o ambiente de verificação
package test_pkg;

  // Importação dos pacotes base
  import uvm_pkg::*;             // Classes e funcionalidades UVM padrão
  import bvm_pkg::*;             // Extensões do Brazil-IP
```

```
  // Inclusão hierárquica dos componentes do testbench
  `include "trans.svh"           // Definição da classe de transação j_jb
  `include "sequence.svh"        // Padrões de estímulos para o testbench
  typedef uvm_sequencer #(j_jb) sequencer;  // Definição do tipo de sequenciador
  `include "driver.svh"          // Conversor de transações para sinais
  `include "monitor.svh"         // Conversor de sinais para transações
  `include "agent.svh"           // Agrupamento de driver, monitor e
sequenciador
  `include "coverage_in.svh"     // Monitoramento de cobertura funcional
  `include "refmod.svh"          // Modelo de referência para comparação
  `include "sink.svh"            // Comparador de saídas esperadas vs reais
  `include "drain.svh"           // Coletor de transações não processadas
  `include "env.svh"             // Ambiente que instancia todos os componentes
  `include "test.svh"            // Definição do caso de teste

endpackage
```

Módulo trans.svh

Esta classe define a unidade fundamental de dados que flui pelo ambiente de verificação:

1. Encapsulamento de dados: Contém o valor j que será enviado para o DUT e processado
2. Geração de estímulos: O campo rand permite a criação de valores aleatórios dentro das restrições definidas (1 a 39)
3. Automação UVM: Os macros uvm_object_utils habilitam funcionalidades como:
    - Criação via factory
    - Comparação automática
    - Impressão formatada para debug
    - Serialização para transporte entre componentes

Esta transação é perfeitamente alinhada com os bins de cobertura definidos anteriormente (0-9, 10-19, 20-29, 30-39), permitindo testar o DUT em toda sua faixa de operação.

```
// Transação j_jb: Define a estrutura de dados que circula pelo ambiente de
verificação
class j_jb extends uvm_sequence_item;

      rand int j;  // Campo randomizável que representa o valor a ser
processado pelo DUT

      // Restrições para direcionar a randomização
      constraint a_positive { j > 0; } // Garante apenas valores positivos
      constraint a_small { j < 40; }   // Limita o valor máximo para 39

      // Registro dos campos para funcionalidades de automação UVM
      `uvm_object_utils_begin(j_jb)
      `uvm_field_int(j, UVM_ALL_ON | UVM_DEC)  // Habilita copy/compare/print
no formato decimal
```

```
    `uvm_object_utils_end

endclass
```

FLUXO DE FUNCIONAMENTO

1. Geração de Estímulos:
   - O `Test` inicia a `Sequence` no `Sequencer` do `Agent`
   - A `Sequence` gera transações `j_jb` randomizadas (valores de 1-39)
   - O `Sequencer` coordena a entrega das transações ao `Driver`
2. Conversão de Transações para Sinais:
   - O `Driver` converte as transações em sinais para o DUT
   - Aplica os sinais na interface virtual conectada ao DUT
3. Observação e Captura:
   - O `Monitor` observa a interface e captura os sinais
   - Converte os sinais capturados de volta em transações `j_jb`
   - Distribui as transações via `analysis_port` para análise
4. Verificação e Análise:
   - O `RefMod` (modelo de referência) recebe as transações e calcula o resultado esperado (j + 250)
   - O `Coverage_in` avalia a cobertura funcional das entradas
   - O `Sink` recebe os resultados esperados do modelo de referência
   - (Nota: Um scoreboard completo compararia os resultados do DUT com os esperados)
5. Finalização da Simulação:
   - O `Drain` consome transações pendentes para evitar bloqueios
   - O teste continua até que todos os componentes concluam suas tarefas
   - O fluxo principal de dados segue: Sequence → Driver → DUT → Monitor → RefMod → Sink
   - As transações `j_jb` encapsulam os dados que fluem pelo ambiente
   - Interfaces TLM (portas put/get/analysis) facilitam a comunicação entre componentes
   - Cada componente tem sua responsabilidade bem definida, seguindo o princípio de separação de responsabilidades

SIMULAÇÃO E ANÁLISE DOS RESULTADOS

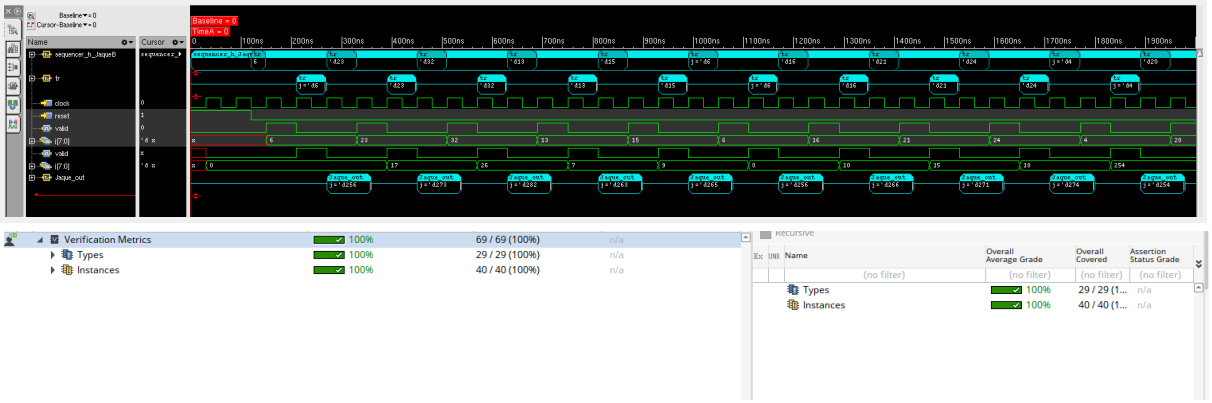
```
[aluno@lad-194 Atividade_1]$ ./coverege.sh
TOOL:  xrun   24.09-s001: Started on Mar 13, 2025 at 15:51:36 -03
TOOL:  xrun(64)     24.09-s001: Started on Mar 13, 2025 at 15:51:36 -03
xrun(64): 24.09-s001: (c) Copyright 1995-2024 Cadence Design Systems, Inc.
xrun: *W,NCEXDEP: Executable (irun) is deprecated. Use (xrun) instead.
Loading snapshot worklib.top:sv ..................... Done
SVSEED default: 1
xcelium> source /usr/local/cds/XCELIUM2409/tools/xcelium/files/xmsimrc
xcelium> source
/usr/local/cds/XCELIUM2409/tools/methodology/UVM/CDNS-1.1d/additions/sv/files/tc
l/uvm_sim.tcl
```

```
xcelium> run
-----------------------------------------------------------------
CDNS-UVM-1.1d (24.09-s001)
(C) 2007-2013 Mentor Graphics Corporation
(C) 2007-2013 Cadence Design Systems, Inc.
(C) 2006-2013 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
-----------------------------------------------------------------

  **********       IMPORTANT RELEASE NOTES          ************

  You are using a version of the UVM library that has been compiled
  with `UVM_NO_DEPRECATED undefined.
  See http://www.eda.org/svdb/view.php?id=3313 for more details.

  You are using a version of the UVM library that has been compiled
  with `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR undefined.
  See http://www.eda.org/svdb/view.php?id=3770 for more details.

       (Specify +UVM_NO_RELNOTES to turn off this notice)

UVM_INFO @ 0: reporter [RNTST] Running test test...
UVM_INFO @ 0: reporter [UVM_CMDLINE_PROC] Applying config setting from the
command line: +uvm_set_config_int=*,recording_detail,1
SDI/Verilog Transaction Recording Facility Version 24.09-s001
SDI2 Transaction Recording API Version 24.09-s001
UVM_INFO ./bvm/bvm_cover.svh(60) @ 210:
uvm_test_top.env_JaquelineB.coverage_in_Jaqueline [coverage_in] Coverage:  0%

UVM_INFO sink.svh(13) @ 330: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 510: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 690: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 870: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 1050: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 1230: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 1410: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 1590: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 1770: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 1950: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 2130: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
```

```
UVM_INFO sink.svh(13) @ 2310: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 2490: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 2670: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 2850: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 3030: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 3210: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 3390: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO ./bvm/bvm_cover.svh(60) @ 3450:
uvm_test_top.env_JaquelineB.coverage_in_Jaqueline [coverage_in] Coverage: 75%

UVM_INFO sink.svh(13) @ 3570: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 3750: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 3930: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 4110: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 4290: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 4470: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 4650: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 4830: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 5010: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 5190: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 5370: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 5550: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 5730: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 5910: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 6090: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 6270: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 6450: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
```

```
UVM_INFO sink.svh(13) @ 6630: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 6810: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 6990: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 7170: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 7350: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 7530: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 7710: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 7890: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 8070: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 8250: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 8430: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 8610: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 8790: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 8970: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 9150: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO sink.svh(13) @ 9330: uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
UVM_INFO ./bvm/bvm_cover.svh(60) @ 9390:
uvm_test_top.env_JaquelineB.coverage_in_Jaqueline [coverage_in] Coverage: 100%

UVM_INFO
/usr/local/cds/XCELIUM2409/tools/methodology/UVM/CDNS-1.1d/sv/src/base/uvm_objec
tion.svh(1268) @ 9390: reporter [TEST_DONE] 'run' phase is ready to proceed to
the 'extract' phase
UVM_INFO ./bvm/bvm_cover.svh(82) @ 9390:
uvm_test_top.env_JaquelineB.coverage_in_Jaqueline [coverage_in] Coverage: 100%

--- UVM Report catcher Summary ---


Number of demoted UVM_FATAL reports  : 0
Number of demoted UVM_ERROR reports  : 0
Number of demoted UVM_WARNING reports: 0
Number of caught UVM_FATAL reports   : 0
Number of caught UVM_ERROR reports   : 0
Number of caught UVM_WARNING reports : 0
```

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

** Report counts by severity
UVM_INFO :   58
UVM_WARNING :0
UVM_ERROR :  0
UVM_FATAL :  0
** Report counts by id
[RNTST]      1
[SINK] 51
[TEST_DONE]  1
[UVM_CMDLINE_PROC]  1
[coverage_in]       4
Simulation complete via $finish(1) at time 9390 NS + 45
/usr/local/cds/XCELIUM2409/tools/methodology/UVM/CDNS-1.1d/sv/src/base/uvm_root.
svh:457      $finish;
xcelium> exit
xmsim: *N,COVCGN: Coverage configuration file command "set_covergroup
-new_instance_reporting" can be specified to improve the scoping and naming of
covergroup instances. It may be noted that subsequent merging of a coverage
database saved with this command and a coverage database saved without this
command is not allowed.

coverage setup:
  workdir  :  ./cov_work
  dutinst  :  top(top)
  scope    :  scope
  testname :  test

coverage files:
  model(design data) :  ./cov_work/scope/icc_2ba9907d_24d443d2.ucm (reused)
  data               :  ./cov_work/scope/test/icc_2ba9907d_24d443d2.ucd
TOOL:  xrun(64)     24.09-s001: Exiting on Mar 13, 2025 at 15:51:38 -03  (total:
00:00:02)
```



Os resultados mostrados no log de execução representam uma execução bem-sucedida do ambiente de verificação UVM. Pontos principais:

1. Métricas de Cobertura

A cobertura funcional evoluiu ao longo da simulação:

- Tempo 210ns: Coverage: 0% - Início da simulação, ainda sem cobertura significativa
- Tempo 3450ns: Coverage: 75% - Três dos quatro bins de cobertura foram atingidos
- Tempo 9390ns: Coverage: 100% - Cobertura total alcançada

Este progresso mostra que a randomização foi eficaz em explorar todo o espaço de teste definido nos bins de cobertura (lembre-se que tínhamos 4 bins: valores 0-9, 10-19, 20-29 e 30-39).

2. Fluxo de Transações

O log mostra 51 mensagens do Sink recebendo transações:

```
UVM_INFO sink.svh(13): uvm_test_top.env_JaquelineB.sink_jaqueline [SINK]
Receiving transaction by Jaqueline_Brito
```

Estas mensagens aparecem em intervalos regulares de 180ns (por exemplo: 330ns, 510ns, 690ns...), demonstrando:

- O fluxo constante de transações pelo testbench
- A correta propagação desde o Driver até o Sink
- O funcionamento do modelo de referência (RefMod)

3. Ausência de Erros

O relatório final UVM mostra:

```
UVM_INFO:    58
UVM_WARNING:  0
UVM_ERROR:    0
UVM_FATAL:    0
```

Isto confirma que o ambiente de verificação e o DUT funcionaram conforme esperado, sem erros de execução ou violações de protocolo.

4. Tempo de Simulação

A simulação foi concluída em 9390ns. Este tempo foi suficiente para:

- Atingir 100% de cobertura
- Processar todas as transações geradas
- Concluir todas as objections UVM normalmente

CONCLUSÃO

A simulação demonstrou progressão consistente da cobertura funcional, começando em 0%, avançando para 75% e finalmente alcançando 100%, indicando exploração completa

dos quatro bins definidos (0-9, 10-19, 20-29, 30-39). Com 51 transações processadas e zero erros ou avisos reportados, o ambiente verificou com sucesso o comportamento do DUT (adição de 250 ao valor de entrada) em toda sua faixa operacional.

Este ambiente UVM completo representa uma solução de verificação modular, reutilizável e escalável, que pode ser adaptada para verificar designs mais complexos através da modificação de componentes específicos sem alterar a arquitetura geral.