Criação, Configuração de Ambiente UVM e Exemplo Completo

Autores	
Odilon de Oliveira Dutra	Unifei

Histórico de Revisões		
18 de dezembro de 2025	1.0	Primeira versão do documento.

Tópicos

Objectives

Objetivo da apresentação

- Apresentar o framework UVM
- Apresentar a preparação do Sistema Operacional para o funcionamento do UVM
- Apresentar a estrutura de um projeto UVM simples
- Aplicado a uma microarquitetura Multi-Cycle Simples
- Preparado para simulação com Cadence Xcelium

Introdução

O que é UVM?

- UVM significa **Universal Verification Methodology**.
- É uma metodologia padronizada para verificação de projetos digitais e *systems-on-chip* (SoCs).
- Baseia-se na linguagem **SystemVerilog** e fornece um framework para:
  - criar componentes de testbench modulares e reutilizáveis,
  - facilitar a integração com o processo de verificação.
- Inclui diretrizes e boas práticas para:
  - desenvolvimento de testbenches,
  - execução de simulações e análise de resultados.
- Tornou-se o padrão para verificação de projetos na indústria de semicondutores.
- É amplamente utilizado por projetistas e engenheiros de verificação para garantir a **correção e funcionalidade** dos seus projetos.

Componentes Fundamentais do UVM

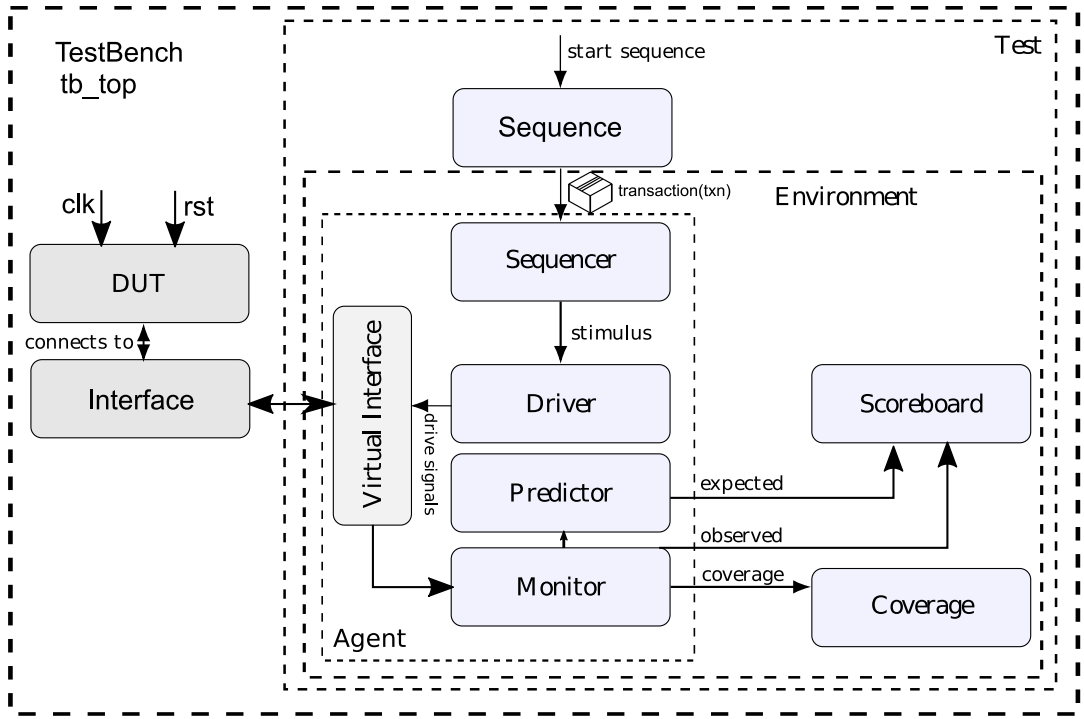
- O UVM contém um conjunto de classes e métodos pré-definidos para criação de testbenches modulares e reutilizáveis.
- Principais componentes:
  - **Componentes de Testbench:**
    - Baseado em classes como `uvm_driver`, `uvm_monitor`, `uvm_scoreboard`, `uvm_agent`, etc.
    - Permitem conexão entre os blocos, sincronização e manipulação de pacotes de dados.
  - **Transações:**
    - Modelam a comunicação entre o DUT e o testbench.
    - Baseadas em classes que podem ser estendidas.
  - **Fases de Simulação:**
    - Controlam a ordem de criação, inicialização e execução dos componentes.

Componentes Fundamentais do UVM

- Principais componentes:
  - **Mensagens e Relatórios:**
    - Infraestrutura para avisos, erros e informações de depuração.
  - **Configuração:**
    - Base de dados para armazenar e recuperar configurações dos componentes.
  - **Cobertura Funcional:**
    - Mecanismo para verificar se todas as funcionalidades do projeto foram exercitadas.
  - **Camada de Abstração de Registradores (RAL):**
    - Facilita a criação e acesso a mapas de registradores.

Ambiente de Verificação UVM

- O ambiente de verificação é construído estendendo classes UVM com prefixo `uvm_`.
- Essas classes já implementam funcionalidades essenciais de comunicação e sincronização.
- UVM passa por revisões constantes: novos recursos são adicionados e antigos descontinuados.
- O manual de referência do UVM traz a hierarquia de classes, funções e tarefas.
- Para novos usuários, a API extensa pode parecer complexa.
- Recomenda-se aprendender o framework em partes.



Setup Genérico

Instalação da Biblioteca UVM

- A biblioteca UVM (Universal Verification Methodology) está incluída na maioria das ferramentas de verificação digital, como:
  - Cadence Incisive / XCelium
  - Mentor Graphics Questa
  - Synopsys VCS
- A biblioteca oferece um conjunto de classes SystemVerilog para criação de testbenches modulares e reutilizáveis.
- Passos para instalar a biblioteca UVM:
  - Instale uma ferramenta de verificação compatível em seu computador.
  - Verifique se a biblioteca UVM já está incluída (na maioria dos casos, está).
  - Caso contrário, baixe a biblioteca no site da Accellera Systems Initiative.
  - Após a instalação, importe as classes UVM no seu código SystemVerilog.
- Use os componentes UVM para criar drivers, monitores, agentes, etc.

Instalação do UVM no Linux

Se for necessária a instalação...

- Passo 1: Baixe a implementação de referência UVM 1.2 no site da Accellera:
  - `http://www.accellera.org/images/downloads/standards/uvm/uvm-1.2.tar.gz`
- Passo 2: Em geral o arquivo `uvm-1.2.tar.gz` estará em sua pasta Downloads.
- Passo 3: Extraia o arquivo via interface gráfica ou no terminal:

```
1 tar -xvf uvm-1.2.tar.gz
```

- Isso criará a pasta `uvm-1.2` no mesmo diretório em que estava o tar.gz.

Instalação do UVM no Linux

- Passo 4: Para incluir o cainho do uvm-1.2 ao PATH do sistema:
  - Para **ncsim**, **-incdir [caminho para uvm-1.2]**
  - Ou defina a variável de ambiente **UVM_HOME** apontando para a pasta **uvm-1.2**.
    - Primeiro, descubra qual shell você está usando:

```
1 echo $0
```

Instalação do UVM no Linux

- Dependendo do shell, use um dos comandos abaixo:
- Para **tcsh**, **csh** e similares:

```
# Choose this for tcsh
setenv UVM_HOME [path_to_your_workarea]/uvm-1.2
```

- Para **bash**:

```
# Choose this for bash
export UVM_HOME=[path_to_your_workarea]/uvm-1.2
```

- Pronto! A instalação do UVM está finalizada e pronta para uso.

Setup XCelium

Variáveis de Ambiente

As seguintes linhas definem variáveis de ambiente necessárias para o correto funcionamento do fluxo de projeto.

```
1  #!/bin/bash
2  #Cadence XCelium - Coloca XCelium 2409 ao PATH do Linux
3  export PATH=$PATH:/apps/cds/XCELIUM2409/bin
4  export PATH=$PATH:/apps/cds/XCELIUM2409/tools/bin
5  #UVM CDNS 1.2 Definition - Mais novo (padrao usa CDNS1.1)
6  export UVM_HOME=/apps/cds/XCELIUM2409/tools.lnx86/methodology/UVM
       /CDNS-1.2
7  # License
8  export LM_LICENSE_FILE="5280@defiant.unifei.edu.br" #porta 5280
       no dominio do servidor licenca
9  export CDS_LIC_FILE="${LM_LICENSE_FILE}"
10 export CDS_LIC_ONLY
```

- Linhas 3 e 4 → Caminho do XCelium 2409 é adicionado ao PATH
- Linha 6 → UVM CDNS 1.2 é setado por **UVM_HOME**
- Linhas 8, 9 e 10 → Definem o endereço e porta que a licença é servida.

Variáveis de Ambiente

- Em geral, essas variáveis de ambiente podem ser salvas num arquivo *bash script (.sh)* que pode ser carregado via source.
- Ou podemos automatizar o processo:
  - Adicionando as linhas mencionadas no slide 17 ao arquivo .bashrc no home de cada usuário.
    - Necessário recarregar o profile com **source /.bashrc**
    - Essa solução exige que o usuário abra o VSCode pelo terminal com: **code &** para que o sistema esteja configurado com as variáveis de ambiente corretamente.
  - Adicionando um *bash script* **cadence.sh** à pasta **/etc/profile.d/**
    - O script cadence.sh terá as mesmas linhas mostradas no slide 17
    - Essa solução generaliza a configuração para todo o sistema e usuários.
    - É necessário reiniciar a máquina ou executar **source /etc/profile**

Estrutura do UVM - Testando uma Microarquitetura Multi-Cycle com UVM

Estrutura de diretórios

```
microarquitetura/
|-- filelist.f
|-- Makefile
|-- README.md
|-- rtl                          # Projeto da microarquitetura multi-cycle
|   |-- ALU.sv                   # Unidade lógica aritmética (ADD, SUB, MUL)
|   |-- control.sv               # FSM de controle da microarquitetura
|   |-- memory.sv                # Memória de 8 posições de 16 bits
|   |-- mux4_registered.sv       # Mux registrado
|   |-- mux4.sv                  # Mux de 4 canais de 8 bits
|   |-- register_bank.sv         # Banco de registradores
|   `-- top.sv                   # Módulo topo (instancia e interconecta blocos)
`-- uvm                          # Diretório do ambiente UVM
    |-- dut_agent.sv             # Agente UVM (driver + monitor + sequencer + predictor)
    |-- dut_cov.sv               # Cobertura funcional
    |-- dut_driver.sv            # Driver: aplica estímulos no DUT
    |-- dut_env.sv               # Ambiente UVM (integra todos os blocos)
    |-- dut_if.sv                # Interface entre o DUT e o testbench
    |-- dut_monitor.sv           # Monitor: observa sinais do DUT
    |-- dut_predictor.sv         # Modelo de referência (golden model)
    |-- dut_scoreboard.sv        # Scoreboard: compara esperado vs. observado
    |-- dut_seq.sv               # Sequências de estímulos
    |-- dut_sequencer.sv         # Sequenciador das transações
    |-- dut_test.sv              # Teste UVM principal
    |-- dut_txn.sv               # Transação (sequence item)
    |-- tb_package.sv            # Package com todas as classes do TB
    `-- tb_top.sv                # Topo do testbench (instancia DUT e chama run_test)
```

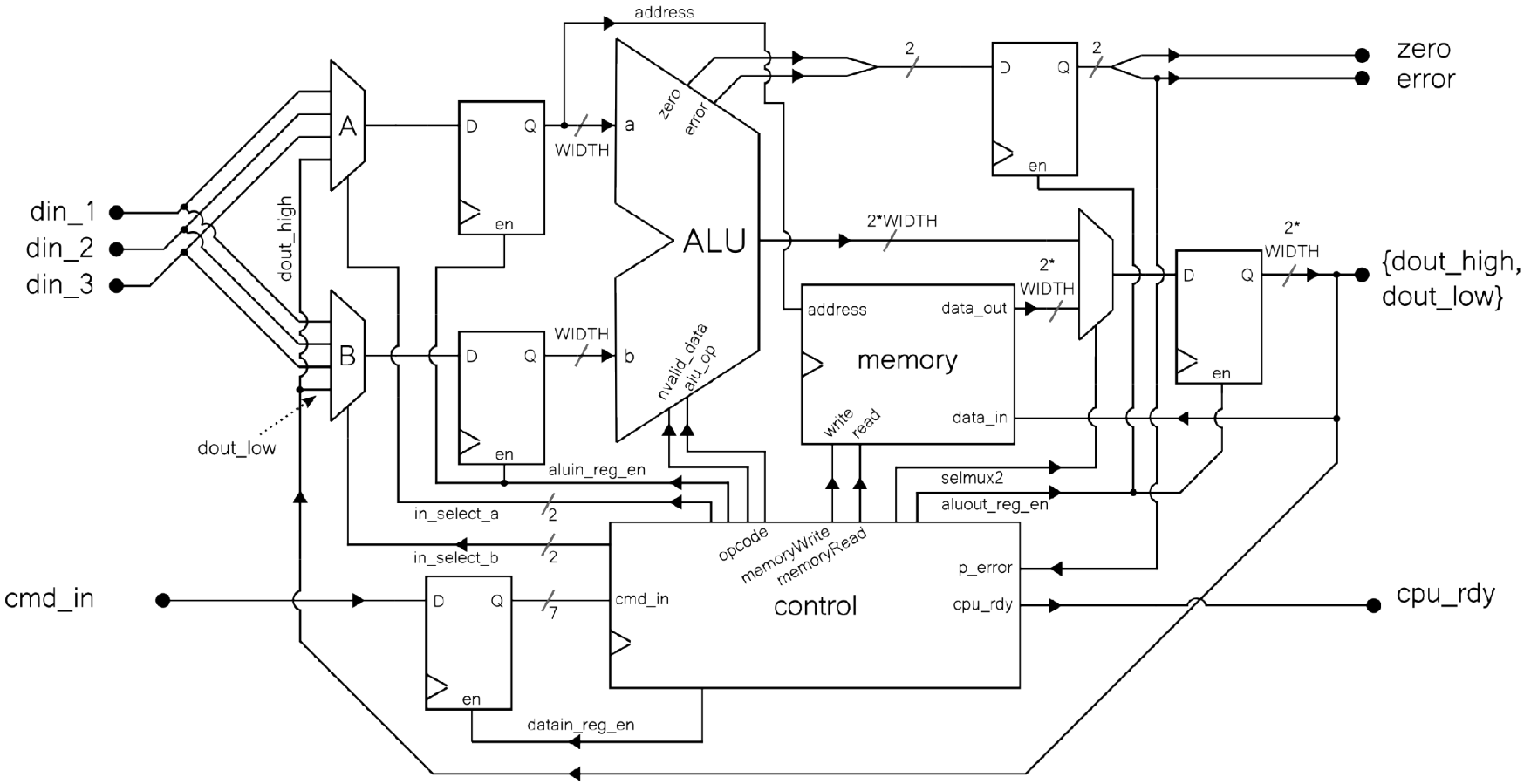
microarquitetura/rtl/top.sv

```
1  //module top
2  module top #(
3      parameter WIDTH = 8
4    ) (
5      input clk,
6      input rst,
7      input [6:0] cmd_in,
8      input [WIDTH-1:0] din_1,
9      input [WIDTH-1:0] din_2,
10     input [WIDTH-1:0] din_3,
11     output [WIDTH-1:0] dout_low,
12     output [WIDTH-1:0] dout_high,
13     output  cpu_rdy,
14     output  zero,
15     output  error
16   );
17
18
```

A implementação da microarquitetura será fornecida.

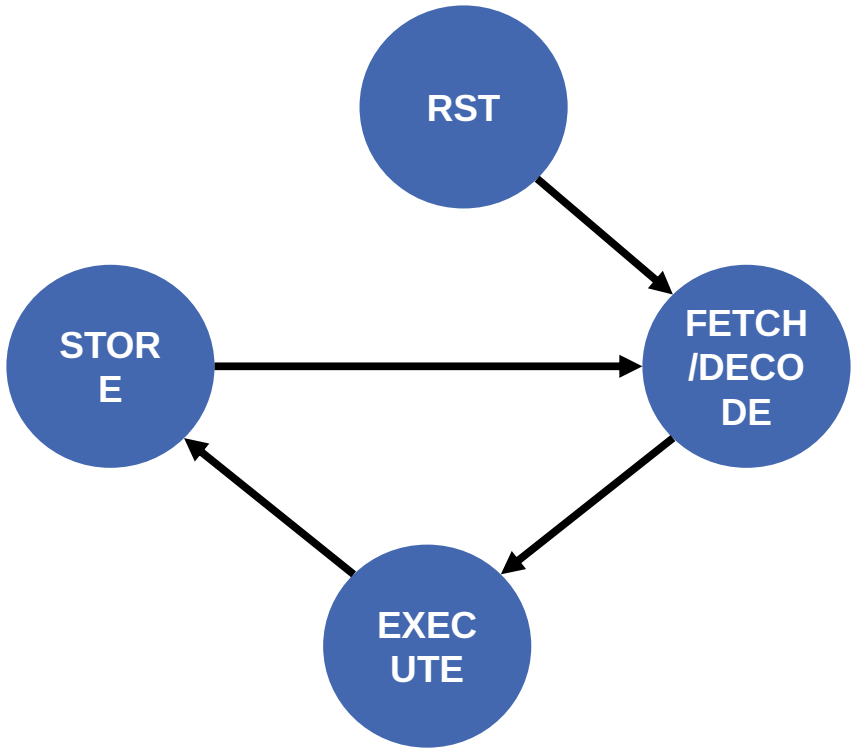
- Módulo topo da microarquitetura Multi-Cycle
- Instancia todos os submódulos e os interconecta.

microarquitetura/rtl/top.sv



A implementação da microarquitetura será fornecida.

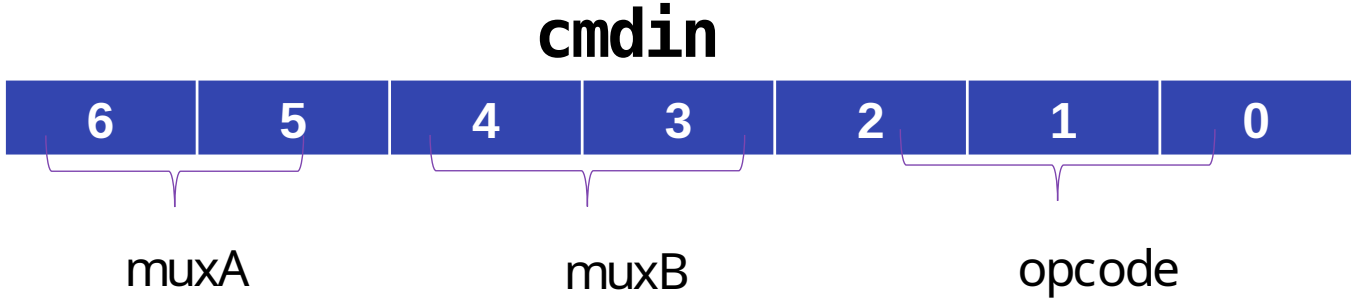
microarquitetura/rtl/control.sv



A implementação da microarquitetura será fornecida.

Microarquitetura - ISA

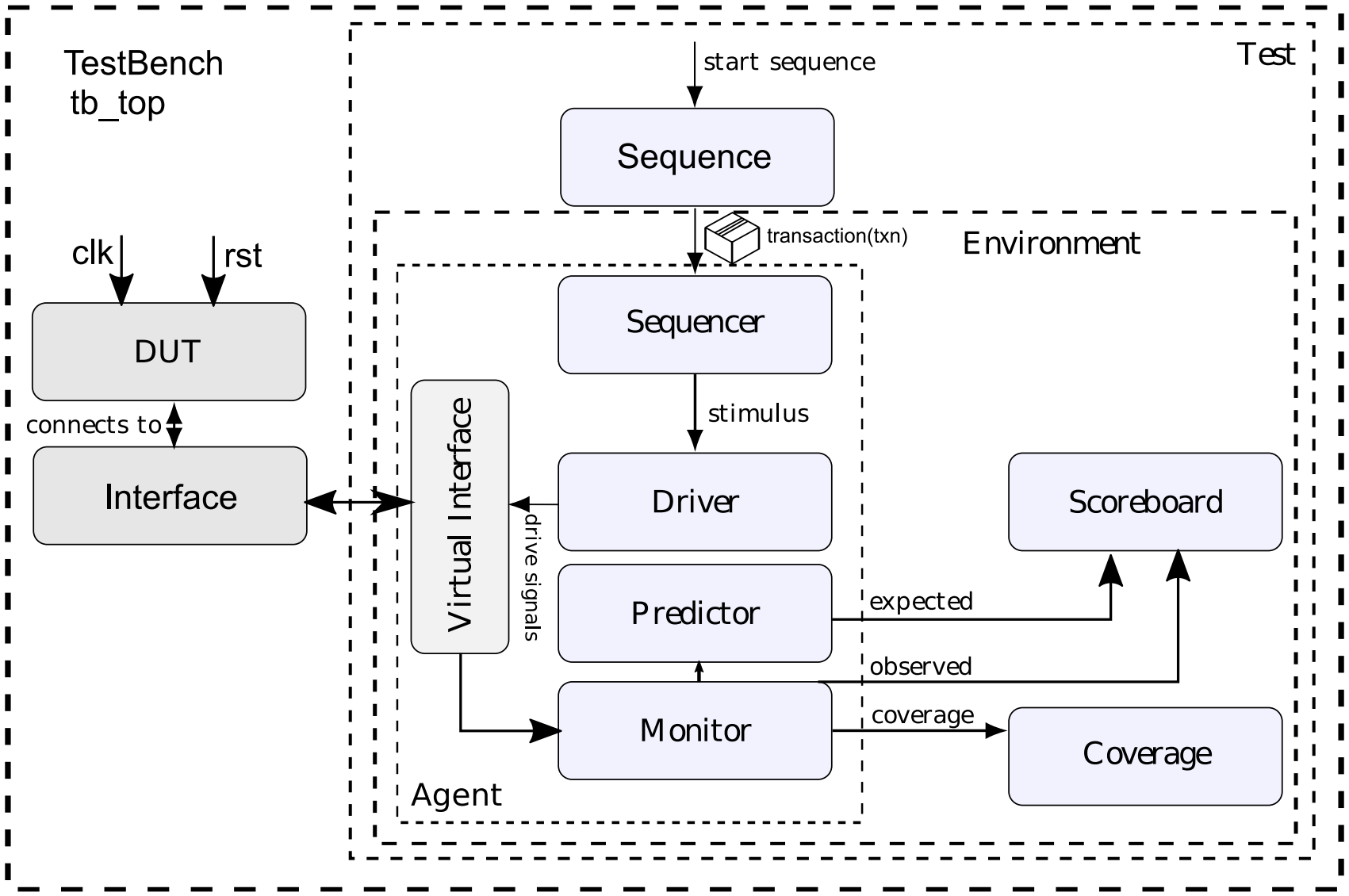
opcode[2]	opcode[1]	opcode[0]	Operation
0	0	0	**Add**
0	0	1	**Sub**
0	1	0	**Mul**
0	1	1	**Div**
1	0	0	**NOP**
1	0	1	**Load**
1	1	0	**Store**
1	1	1	**NOP**



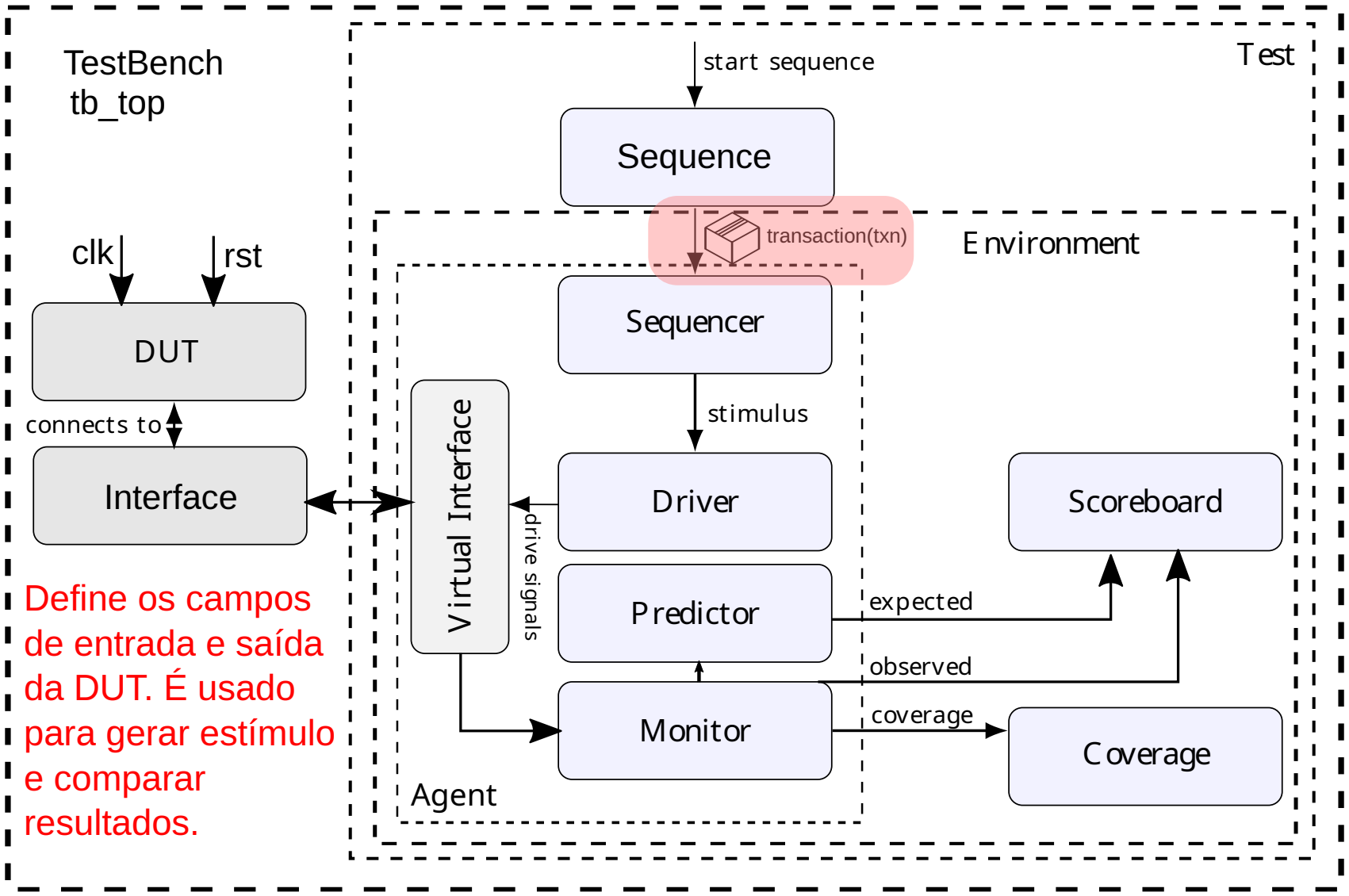
A implementação da microarquitetura será fornecida.

UVM

UVM


TestBench
tb_top
Test
start sequence
Sequence
transaction(txn)
Environment
clk
rst
DUT
Sequencer
connects to
stimulus
Interface
Virtual Interface
drive signals
Driver
Scoreboard
Predictor
expected
observed
Monitor
coverage
Coverage
Agent

UVM Transaction - dut_txn.sv



microarquitetura/uvm/dut_txn.sv

```
1  class dut_txn extends uvm_sequence_item;
2    `uvm_object_utils(dut_txn)
3
4    rand bit [6:0] cmd_in;
5    rand bit [7:0] din_1;
6    rand bit [7:0] din_2;
7    rand bit [7:0] din_3;
8
9    // campos observados / resposta do DUT
10   bit [7:0] dout_low;
11   bit [7:0] dout_high;
12   bit       cpu_rdy;
13   bit       zero;
14   bit       error;
15   logic [1:0] state;
16
17   function new(string name="dut_txn");
18     super.new(name);
19   endfunction
20
```

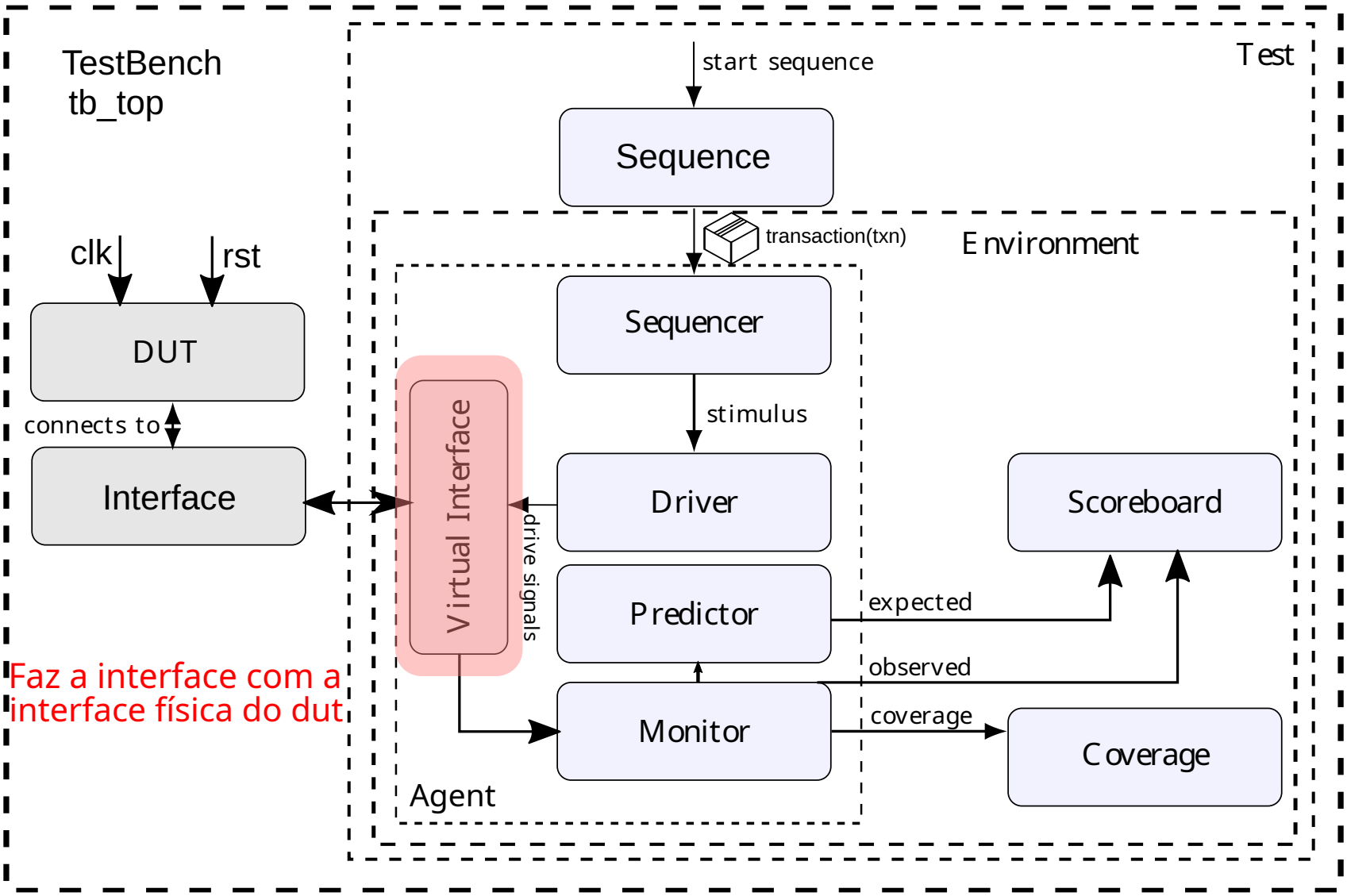
- Declara o pacote UVM e importa dependências necessárias.
- Define a classe `dut_txn` como `uvm_sequence_item`.
- Especifica campos que representam sinais de entrada/saída do DUT.
- Permite que os campos sejam randomizados para geração de estímulos.
- Inclui construtor padrão registrando a classe no UVM factory.

microarquitetura/uvm/dut_txn.sv

```
21    // Metodo copy obrigatorio para propagacao
      ↪ correta
22    function void copy(uvm_object rhs);
23      dut_txn tx;
24      if (!$cast(tx, rhs)) begin
25        `uvm_warning("COPY_FAIL", "Falha ao fazer
          ↪ cast em dut_txn::copy")
26        return;
27      end
28      this.cmd_in    = tx.cmd_in;
29      this.din_1     = tx.din_1;
30      this.din_2     = tx.din_2;
31      this.din_3     = tx.din_3;
32      this.dout_low  = tx.dout_low;
33      this.dout_high = tx.dout_high;
34      this.zero      = tx.zero;
35      this.error     = tx.error;
36      this.cpu_rdy   = tx.cpu_rdy;
37      this.state     = tx.state;
38    endfunction
39
40    function string convert2string();
41      return $sformatf("cmd=%0d din1=%0h din2=%0h
        ↪ din3=%0h -> dout_hi=%0h dout_lo=%0h
        ↪ rdy=%0b z=%0b e=%0b state=%d",
42                       cmd_in, din_1, din_2,
                         ↪ din_3, dout_high,
                         ↪ dout_low, cpu_rdy,
                         ↪ zero, error, state);
43    endfunction
44  endclass
```

- Pode implementar métodos copy, compare e print para depuração (pelo menos o convert2string()).
- Define constraints garantindo valores coerentes com o protocolo do DUT.
- Suporta geração dirigida e aleatória de transações.
- Fornece estrutura para sequences explorarem diferentes combinações de sinais.
- Facilita monitoramento e checagem de comportamento via scoreboards.

UVM Transaction - dut_if.sv

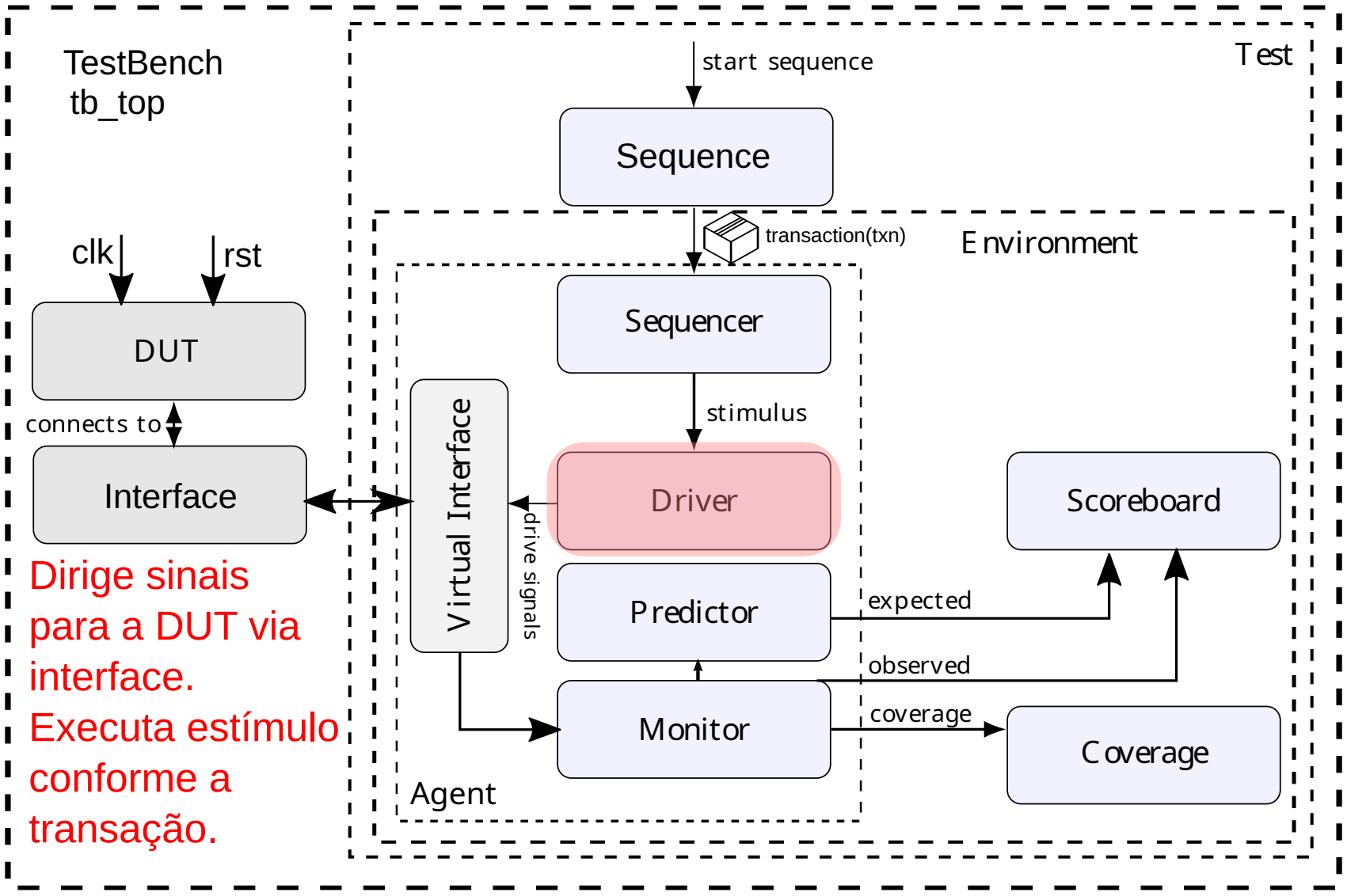


microarquitetura/uvm/dut_if.sv

```
1  //classe dut_if
2  interface dut_if(input logic clk,
   ↪  input logic rst);
3      parameter WIDTH = 8;
4      //logic rst;
5      logic [6:0] cmd_in;
6      logic [WIDTH-1:0] din_1;
7      logic [WIDTH-1:0] din_2;
8      logic [WIDTH-1:0] din_3;
9      logic [WIDTH-1:0] dout_low;
10     logic [WIDTH-1:0] dout_high;
11     logic  cpu_rdy;
12     logic  zero;
13     logic  error;
14     logic [1:0] state;
15 endinterface
```

- Define interface do DUT com sinais de controle e dados.
- Agrupa entradas, saídas e estado interno em um único bloco.
- Fornece ponto de conexão para driver, monitor e virtual interface.

UVM Transaction - dut_driver.sv



microarquitetura/uvm/dut_driver.sv

```
1  `ifndef DUT_DRIVER_SV
2  `define DUT_DRIVER_SV
3
4  class dut_driver extends uvm_driver #(dut_txn);
5    `uvm_component_utils(dut_driver)
6
7    localparam int WIDTH = 8;
8
9    // Interface compartilhada (recebida via agent)
10   virtual dut_if vif;
11
12   // Porta de analise para enviar estímulos ao predictor
13   uvm_analysis_port#(dut_txn) expected_port;
14
```

- Estende `uvm_driver` parametrizado com `dut_txn`.
- Armazena a *virtual interface* usada para dirigir sinais ao DUT.
- Possui `analysis_port` para enviar transações esperadas ao predictor.

microarquitetura/uvm/dut_driver.sv

```
15    // ---------------------------------
16    // Construtor
17    // ---------------------------------
18    function new(string name, uvm_component parent);
19      super.new(name, parent);
20      expected_port = new("expected_port", this);
21    endfunction
22
```

- Construtor registra o driver e inicializa recursos internos.
- Cria a `expected_port` para enviar transações ao predictor.
- Prepara o componente para receber interface e sequência.

microarquitetura/uvm/dut_driver.sv

```
23    // ------------------------------------
24    // build_phase
25    // ------------------------------------
26    function void build_phase(uvm_phase phase);
27      super.build_phase(phase);
28
29      if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif",
      ↪  vif))
30        `uvm_fatal("NOVIF", "Virtual interface nao encontrada para
        ↪  driver")
31    endfunction
32
```

- Executa `build_phase` herdado para configurar o componente.
- Recupera a *virtual interface* via `uvm_config_db`.
- Emite erro fatal caso a interface não seja encontrada.

microarquitetura/uvm/dut_driver.sv

```
33    // ------------------------------------
34    // Metodo auxiliar para aplicar estímulo
35    // ------------------------------------
36    function void apply_tx(dut_txn tx);
37      vif.cmd_in <= tx.cmd_in;
38      vif.din_1  <= tx.din_1;
39      vif.din_2  <= tx.din_2;
40      vif.din_3  <= tx.din_3;
41    endfunction
42
```

- Copia campos da transação para sinais da interface.
- Centraliza a lógica de aplicação de estímulos ao DUT.
- Garante consistência ao dirigir entradas múltiplas do design.

microarquitetura/uvm/dut_driver.sv

```
43    // ------------------------------------
44    // run_phase: recebe, aplica e publica transacoes
45    // ------------------------------------
46    task run_phase(uvm_phase phase);
47      dut_txn tx;
48      forever begin
49        seq_item_port.get_next_item(tx);
50
51        if (tx == null) begin
52          `uvm_error("DRIVER", "Recebeu transacao nula")
53          continue;
54        end
55
56        tx.accept_tr();
57        tx.begin_tr();
58
59        apply_tx(tx);
60
```

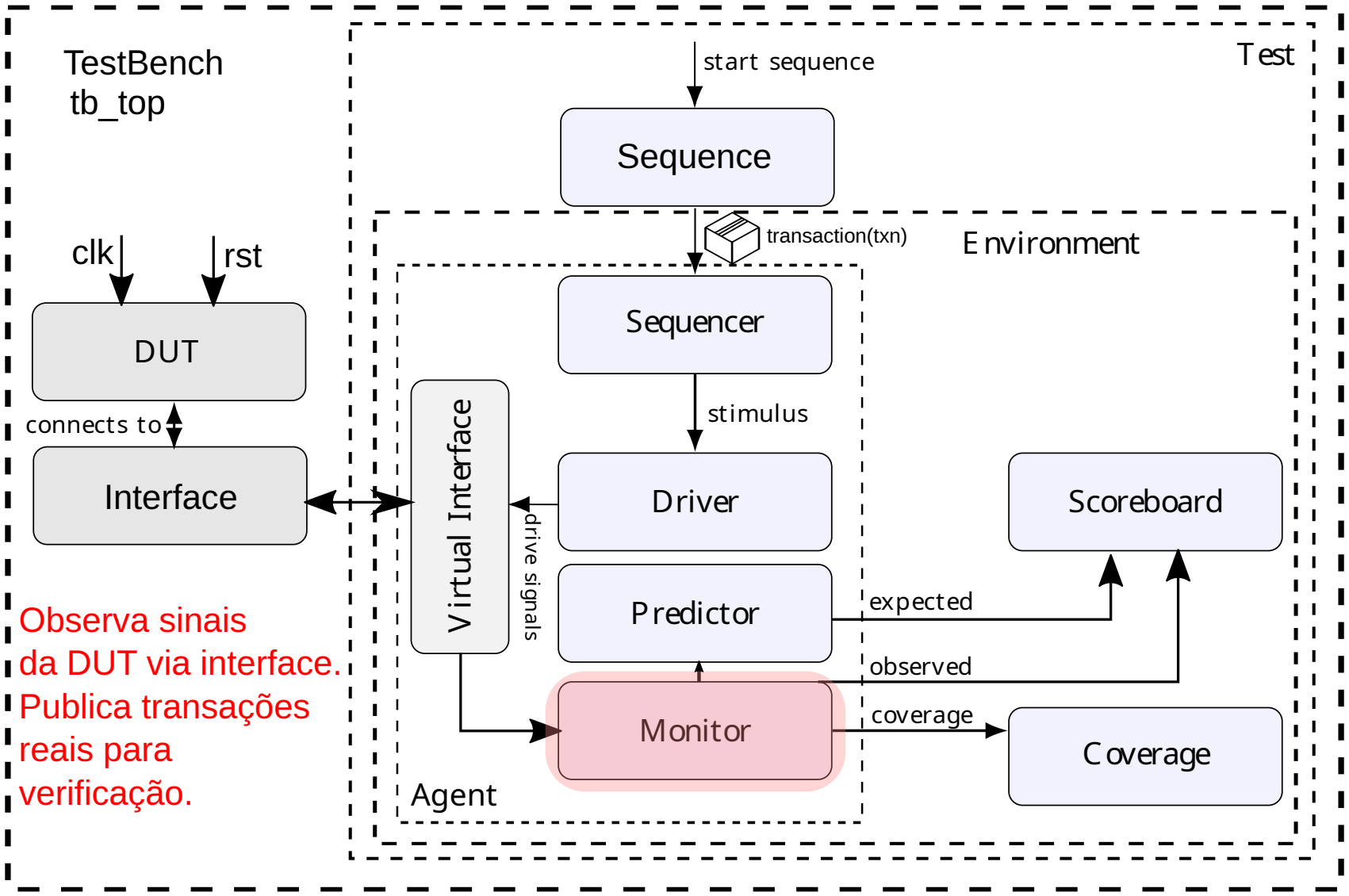
- Obtém transações da sequência continuamente via `get_next_item`.
- Verifica nulidade da transação e reporta erro apropriado.
- Marca início da transação e aplica estímulo com `apply_tx`.

microarquitetura/uvm/dut_driver.sv

```
61          @(negedge vif.clk); // sincronizacao segura
62
63          tx.end_tr();
64          seq_item_port.item_done();
65
66          // Publica transacao para o predictor
67          expected_port.write(tx);
68        end
69      endtask
70
71  endclass
72
73  `endif // DUT_DRIVER_SV
```

- Obtém transações da sequência continuamente via `get_next_item`.
- Verifica nulidade da transação e reporta erro apropriado.
- Marca início da transação e aplica estímulo com `apply_tx`.

UVM Transaction - dut_monitor.sv



microarquitetura/uvm/dut_monitor.sv

```
1  //classe dut_monitor
2  class dut_monitor extends uvm_component;
3    `uvm_component_utils(dut_monitor)
4
5    // Interface virtual
6    virtual dut_if vif;
7
8    // Porta de analise para o predictor
9    uvm_analysis_port#(dut_txn) analysis_port;
10
11   // Fila de estímulos com latência configuravel
12   dut_txn input_q[$];
13   int latency = 3; // padrão: 3 ciclos
14
```

- Estende `uvm_component` e registra o monitor no factory.
- Usa *virtual interface* para observar sinais do DUT.
- Possui fila e `analysis_port` para enviar transações ao predictor.

microarquitetura/uvm/dut_monitor.sv

```
15    // ------------------------------------------------
16    // Construtor
17    // ------------------------------------------------
18    function new(string name, uvm_component parent);
19      super.new(name, parent);
20      analysis_port = new("analysis_port", this);
21    endfunction
22
```

- Construtor registra o monitor no UVM e inicializa sua hierarquia.
- Instancia a `analysis_port`, responsável por enviar transações observadas.
- Deixa o monitor pronto para receber a interface e configurar latência.

microarquitetura/uvm/dut_monitor.sv

```
23    // ------------------------------------------------
24    // build_phase
25    // ------------------------------------------------
26    function void build_phase(uvm_phase phase);
27      super.build_phase(phase);
28
29      // Interface virtual
30      if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif",
      ↪  vif))
31        `uvm_fatal("NOVIF", "Monitor nao recebeu a interface
        ↪  virtual")
32
33    endfunction
34
```

- Executa a `build_phase` padrão para inicializar a hierarquia UVM.
- Recupera a *virtual interface* do DUT via `uvm_config_db`.
- Interrompe a simulação com `uvm_fatal` caso a interface não seja configurada.

microarquitetura/uvm/dut_monitor.sv

```
35    // ------------------------------------------------
36    // run_phase
37    // ------------------------------------------------
38    task run_phase(uvm_phase phase);
39      bit cpu_rdy_prev = 0;
40      bit cpu_rdy_edge = 0;
41      dut_txn stim = dut_txn::type_id::create("stim");
42      dut_txn aligned ;
43
44      forever begin
45        @(negedge vif.clk);
46        // Captura estímulo atual
47        stim.cmd_in = vif.cmd_in;
48        stim.din_1  = vif.din_1;
49        stim.din_2  = vif.din_2;
50        stim.din_3  = vif.din_3;
```

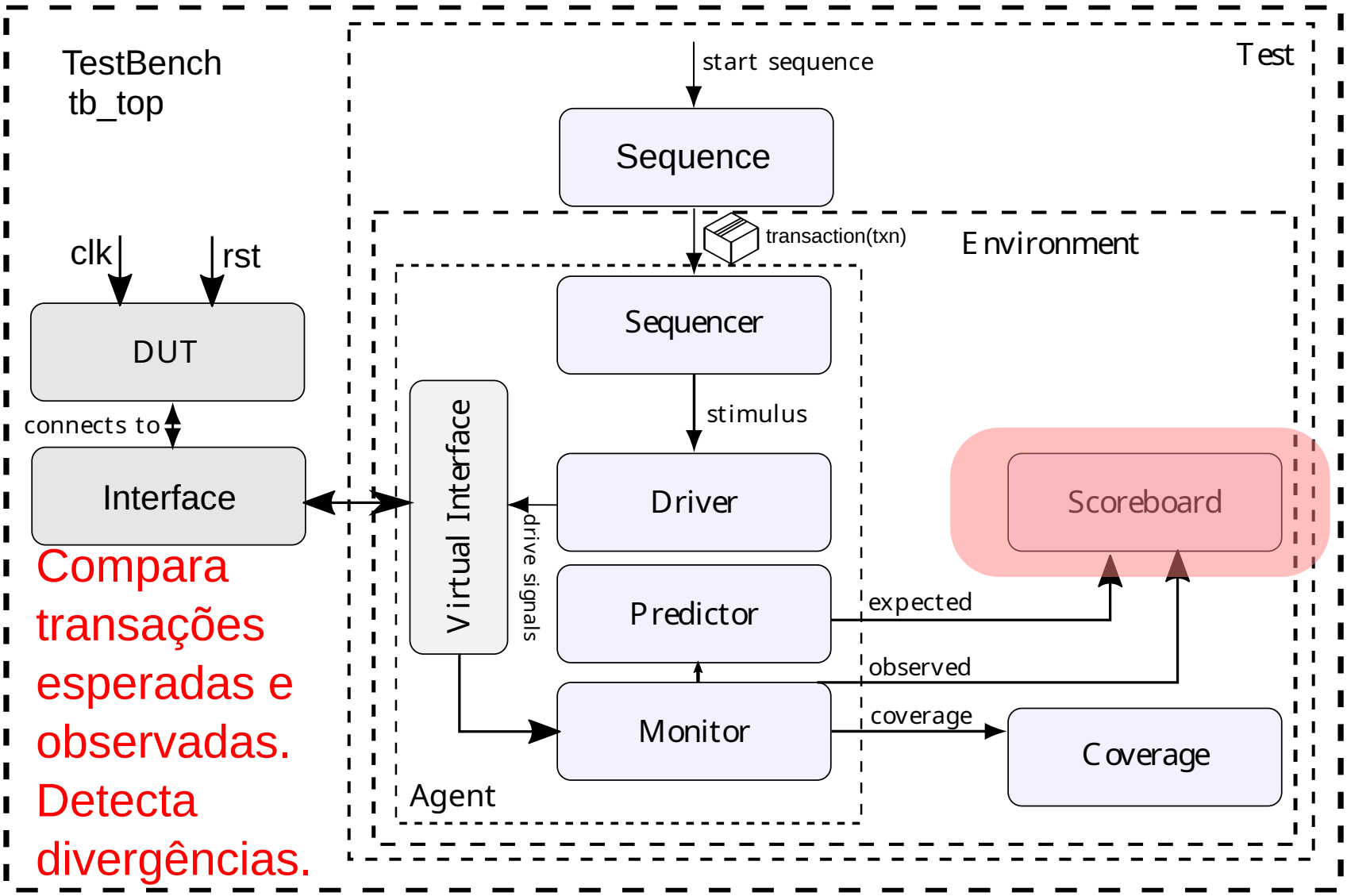
- Monitora borda de `cpu_rdy` e prepara transações de captura.
- Amostra sinais de entrada do DUT a cada `negedge` de clock.
- Atualiza `stim` com os valores observados e insere na fila.

microarquitetura/uvm/dut_monitor.sv

```
51        input_q.push_back(stim);
52        // Detecta borda de subida de cpu_rdy
53        cpu_rdy_edge = vif.cpu_rdy && !cpu_rdy_prev;
54        cpu_rdy_prev = vif.cpu_rdy;
55
56        if (cpu_rdy_edge) begin
57                aligned = input_q.pop_front(); //retira dados de entrada da fila
58            // Junta com os sinais de saída
59            aligned.dout_low  = vif.dout_low;
60            aligned.dout_high = vif.dout_high;
61            aligned.zero      = vif.zero;
62            aligned.error     = vif.error;
63            aligned.cpu_rdy   = vif.cpu_rdy;
64            // Envia para o predictor
65            //Dados alinhados (compensado
66            //para a latência do multi-cycle)
67            analysis_port.write(aligned);
68        end
69      end
70    endtask
71  endclass
```

- Detecta borda de subida de `cpu_rdy` e acessa entrada atrasada.
- Combina essa entrada com as saídas atuais para alinhar fases da operação multiciclo.
  - Microarquitetura tem ciclo de máquina de 3 ciclos de clock.
- Envia a transação alinhada ao `analysis_port` compensando a latência.

UVM Transaction - dut_scoreboard.sv



microarquitetura/uvm/dut_scoreboard.sv

- Compara transações esperadas vindas do *Golden Model* calculadas no **dut_pedictor** com resultados reais do DUT.
- Registra e reporta erros de funcionalidade durante a simulação.

```
1  //classe dut_scoreboard
2  //=====================================================
3  // dut_scoreboard.sv (versão compatível com VCS/UVM 1.2)
4  //=====================================================
5  `ifndef DUT_SCOREBOARD_SV
6  `define DUT_SCOREBOARD_SV
7
8  // ------------------------------------------------
9  // Forward declaration da classe principal
10 // ------------------------------------------------
11 typedef class dut_scoreboard;
12
```

- Arquivo define a classe `dut_scoreboard` para UVM 1.2/VCS.
- Usa guardas de compilação `` `ifndef/`define `` para evitar múltiplas inclusões.
- Contém declaração antecipada (`forward declaration`) da classe principal.

microarquitetura/uvm/dut_scoreboard.sv

```
13 // ------------------------------------------------
14 // Tipos auxiliares (analysis_imp especializados)
15 // ------------------------------------------------
16 class dut_scoreboard_exp_imp_t extends uvm_analysis_imp#(dut_txn, dut_scoreboard);
17   dut_scoreboard parent; // referência explícita
18
19   function new(string name, dut_scoreboard parent);
20     super.new(name, parent);
21     this.parent = parent; // atribuicao correta
22   endfunction
23
24   virtual function void write(dut_txn t);
25     if (this.parent != null)
26       this.parent.write_expected(t); // uso correto do campo 'parent'
27   endfunction
28 endclass
29
```

- Define `dut_scoreboard_exp_imp_t` estendendo `uvm_analysis_imp` para transações esperadas.
- Mantém referência explícita ao scoreboard pai para encaminhar dados.
- Implementa `write()` que chama `write_expected()` no scoreboard.

microarquitetura/uvm/dut_scoreboard.sv

```
30 class dut_scoreboard_act_imp_t extends uvm_analysis_imp#(dut_txn, dut_scoreboard);
31   dut_scoreboard parent;
32
33   function new(string name, dut_scoreboard parent);
34     super.new(name, parent);
35     this.parent = parent; // atribuicao correta
36   endfunction
37
38   virtual function void write(dut_txn t);
39     if (this.parent != null)
40       this.parent.write_actual(t); // uso correto do campo 'parent'
41   endfunction
42 endclass
43
```

- Define `dut_scoreboard_act_imp_t` estendendo `uvm_analysis_imp` para transações reais.
- Mantém referência ao scoreboard pai para enviar dados observados.
- Implementa `write()` que chama `write_actual()` no scoreboard.

microarquitetura/uvm/dut_scoreboard.sv

```
44  // ----------------------------------------------
45  // Classe principal: dut_scoreboard
46  // ----------------------------------------------
47  class dut_scoreboard extends uvm_component;
48    `uvm_component_utils(dut_scoreboard)
49
50          function void write(dut_txn t);
51            // Aqui você decide para onde enviar o dado
52            // Exemplo: se estiver vindo do expected_export
53            write_expected(t);
54
55            // Ou, se estiver vindo do actual_export:
56            // write_actual(t);
57          endfunction
58
59    // Analysis ports para expected e actual
60    dut_scoreboard_exp_imp_t expected_export;
61    dut_scoreboard_act_imp_t actual_export;
62
63    // Filas
64    dut_txn expected_q[$];
65    dut_txn actual_q[$];
66
```

- Estende `uvm_component` e registra a classe no UVM factory.
- Possui métodos `write_expected()` e `write_actual()` para receber transações.
- Contém analysis ports e filas separadas para transações esperadas e reais.

microarquitetura/uvm/dut_scoreboard.sv

```
67    // ------------------------------------------------
68    // Construtor
69    // ------------------------------------------------
70    function new(string name, uvm_component parent);
71      super.new(name, parent);
72    endfunction
73
```

- Construtor chama `super.new` para inicializar o componente UVM.
- Registra o scoreboard na hierarquia do testbench.
- Prepara o componente para instanciar analysis ports e filas.

microarquitetura/uvm/dut_scoreboard.sv

```
74    // ------------------------------------------------
75    // build_phase
76    // ------------------------------------------------
77    function void build_phase(uvm_phase phase);
78      super.build_phase(phase);
79      expected_export = new("expected_export", this);
80      actual_export   = new("actual_export", this);
81    endfunction
82
```

- Executa `build_phase` herdado para inicializar a hierarquia.
- Instancia `expected_export` para receber transações esperadas.
- Instancia `actual_export` para receber transações reais do DUT.

microarquitetura/uvm/dut_scoreboard.sv

```
83    // ---------------------------------------------
84    // Recebimento dos pacotes
85    // ---------------------------------------------
86    function void write_expected(dut_txn t);
87      dut_txn copy = dut_txn::type_id::create("exp_copy");
88      copy.copy(t);
89      expected_q.push_back(copy);
90    endfunction
91
92    function void write_actual(dut_txn t);
93      dut_txn copy = dut_txn::type_id::create("act_copy");
94      copy.copy(t);
95      actual_q.push_back(copy);
96    endfunction
97
```

- `write_expected()` copia e armazena transações esperadas na fila `expected_q` vindas do **dut_predictor**.
- `write_actual()` copia e armazena transações reais na fila `actual_q` vindas do **dut_monitor**.
- Permite que o scoreboard organize dados para comparação posterior.

microarquitetura/uvm/dut_scoreboard.sv

```
98    // ------------------------------------------------
99    // Comparacao
100   // ------------------------------------------------
101   task run_phase(uvm_phase phase);
102   dut_txn exp, act;
103   forever begin
104     wait(expected_q.size() > 0 && actual_q.size() > 0);
105
106     if (expected_q.size() > 100 || actual_q.size() > 100)
107       `uvm_warning("QUEUE_OVERFLOW", "Scoreboard queues estao crescendo deamis -
          ↪  possivel desalinhamento");
108
109     exp = expected_q.pop_front();
110     act = actual_q.pop_front();
111
112     compare_transactions(exp, act);
113   end
114   endtask
115
```

- Monitora continuamente filas de transações esperadas e reais.
- Emite aviso se alguma fila crescer demais, indicando possível desalinhamento.
- Retira transações do início das filas e chama `compare_transactions()`.

microarquitetura/uvm/dut_scoreboard.sv

```
116   task compare_transactions(dut_txn exp, dut_txn act);
117     if (exp.dout_low  !== act.dout_low  ||   exp.dout_high !== act.dout_high ||
118         exp.zero      !== act.zero      ||   exp.error     !== act.error     ||
119         exp.cpu_rdy   !== act.cpu_rdy) begin
120
121       `uvm_error("MISMATCH", $sformatf(
122         "\nExpected -> cmd_in=%h din_1=%d din_2=%d din_3=%d dout_high=%h dout_low=%h
          ↪  zero=%0b error=%0b cpu_rdy=%0b \
123          \nActual   -> cmd_in=%h din_1=%d din_2=%d din_3=%d dout_high=%h dout_low=%h
          ↪  zero=%0b error=%0b cpu_rdy=%0b",
124         exp.cmd_in, exp.din_1, exp.din_2, exp.din_3, exp.dout_high, exp.dout_low,
          ↪  exp.zero, exp.error, exp.cpu_rdy,
125         act.cmd_in, act.din_1, act.din_2, act.din_3, act.dout_high, act.dout_low,
          ↪  act.zero, act.error, act.cpu_rdy))
126     end
```

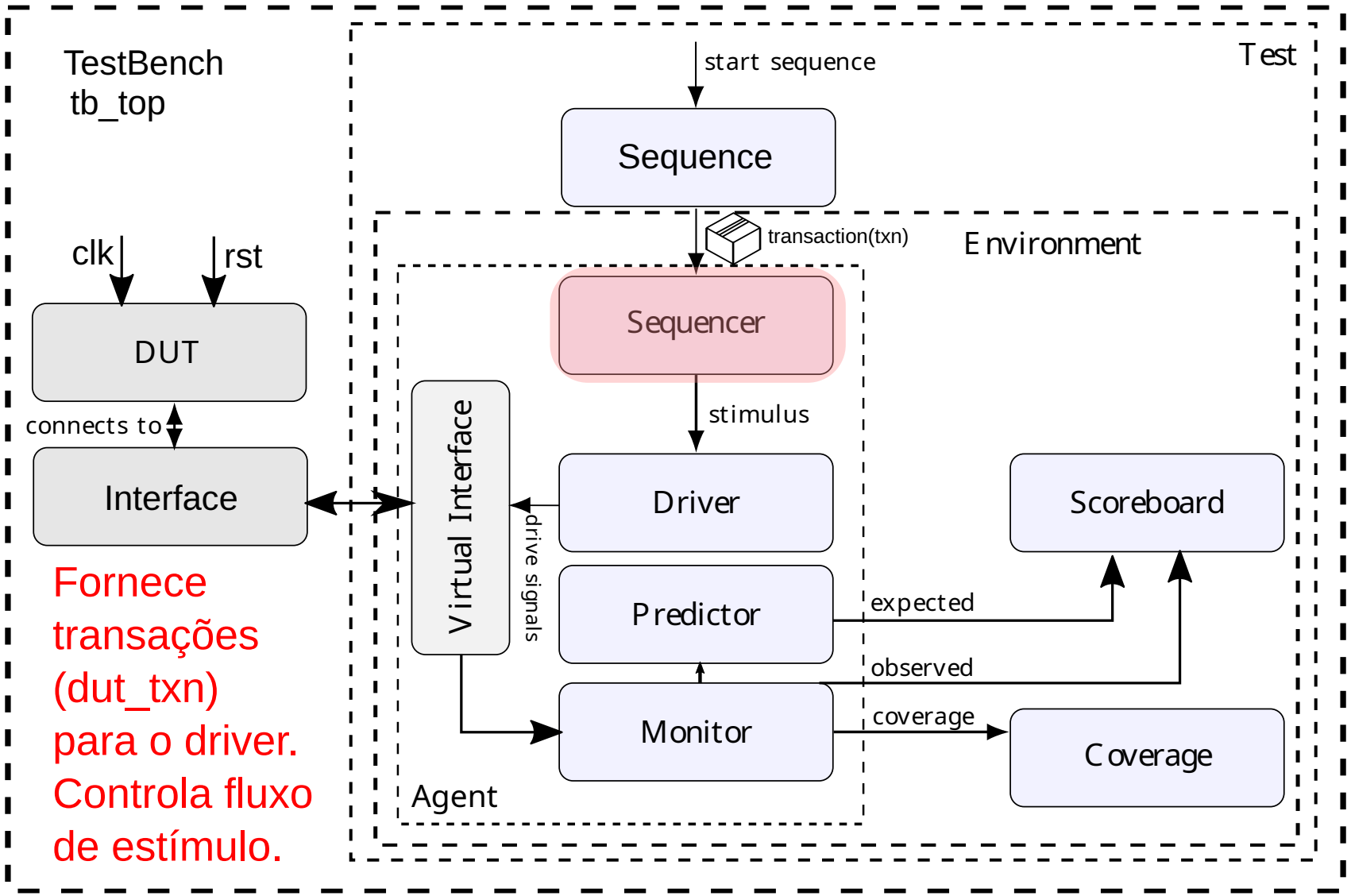
- Compara todos os campos relevantes das transações esperadas e reais.
- Emite `uvm_error` se houver divergência entre esperado e real.
- Emite `uvm_info` se as transações coincidirem corretamente.

microarquitetura/uvm/dut_scoreboard.sv

```
127     else begin
128       `uvm_info("MATCH", $sformatf(
129         "\nExpected -> cmd_in=%h din_1=%d din_2=%d din_3=%d dout_high=%h dout_low=%h
          ↪  zero=%0b error=%0b cpu_rdy=%0b \
130          \nActual   -> cmd_in=%h din_1=%d din_2=%d din_3=%d dout_high=%h dout_low=%h
          ↪  zero=%0b error=%0b cpu_rdy=%0b",
131         exp.cmd_in, exp.din_1, exp.din_2, exp.din_3, exp.dout_high, exp.dout_low,
          ↪  exp.zero, exp.error, exp.cpu_rdy,
132         act.cmd_in, act.din_1, act.din_2, act.din_3, act.dout_high, act.dout_low,
          ↪  act.zero, act.error, act.cpu_rdy), UVM_LOW)
133     end
134   endtask
135 endclass
```

- Compara todos os campos relevantes das transações esperadas e reais.
- Emite `uvm_error` se houver divergência entre esperado e real.
- Emite `uvm_info` se as transações coincidirem corretamente.

UVM Transaction - dut_sequencer.sv

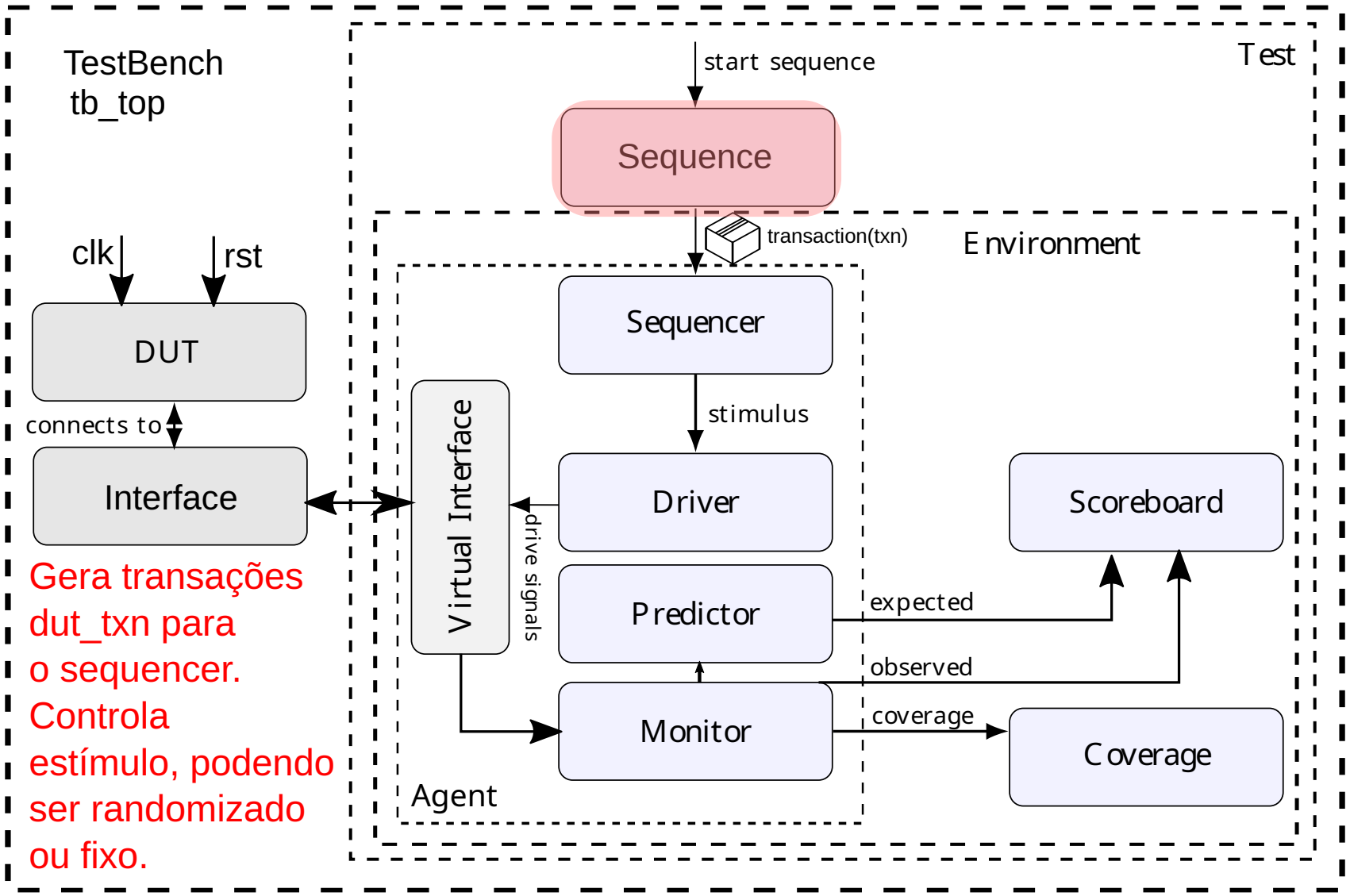


microarquitetura/uvm/dut_sequencer.sv

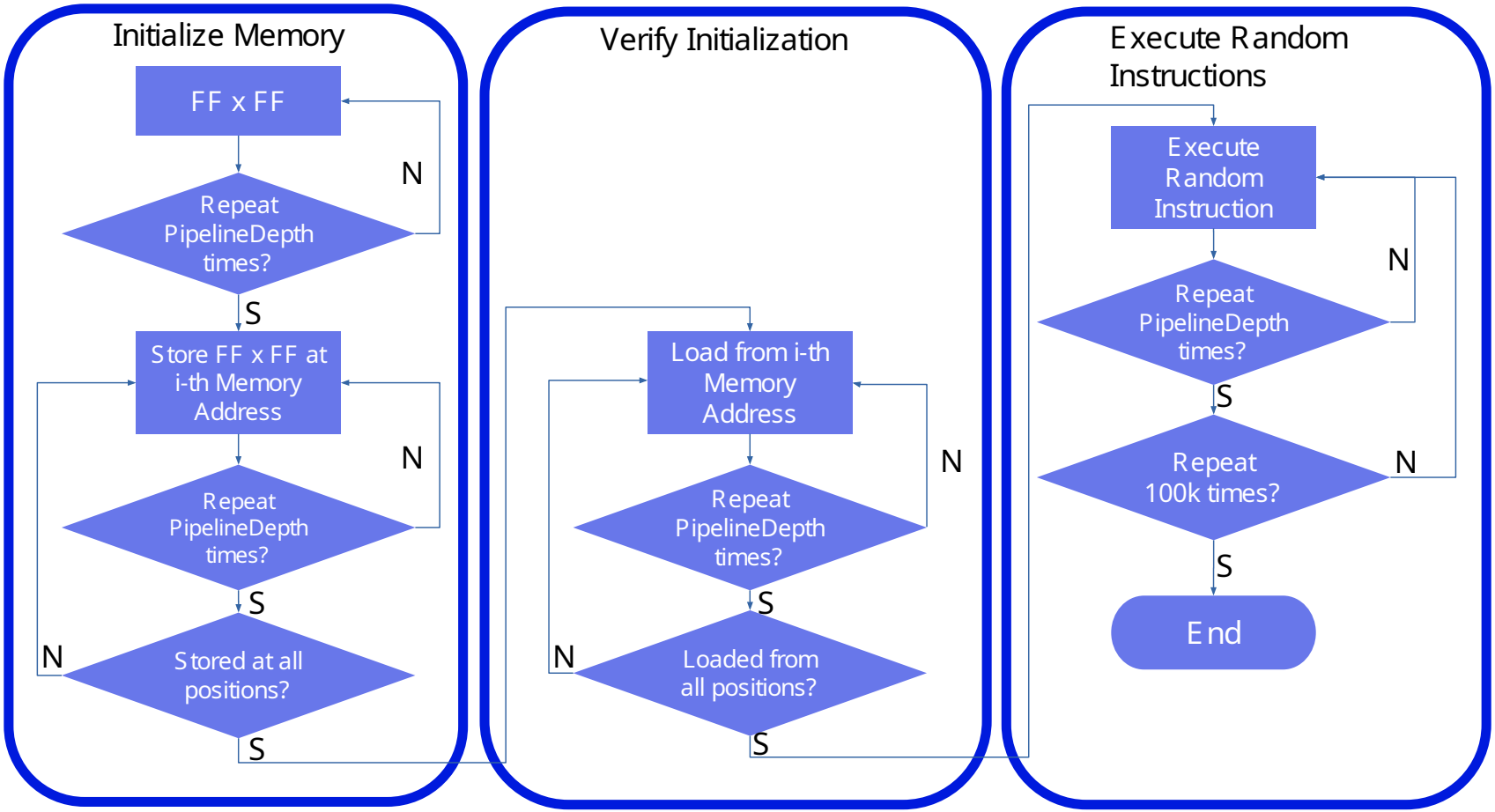
```
1  //classe dut_sequencer
2  //=====================================================
3  // dut_sequencer.sv
4  // Sequencer basico parametrizado por dut_txn
5  //=====================================================
6  class dut_sequencer extends uvm_sequencer #(dut_txn);
7    `uvm_component_utils(dut_sequencer)
8
9    function new(string name, uvm_component parent);
10     super.new(name, parent);
11   endfunction
12 endclass
13
```

- Controla o fluxo de transações entre sequences e o driver
- Herda de `uvm_sequencer#(dut_txn)`
- Atua como canal de comunicação para as transações geradas
- Não contém lógica própria neste exemplo, apenas instanciado pelo driver

UVM Transaction - dut_seq.sv



UVM Transaction - dut_seq.sv



microarquitetura/uvm/dut_seq.sv

```
1  //classe dut_seq
2  class dut_sequence extends uvm_sequence #(dut_txn);
3    `uvm_object_utils(dut_sequence)
4
5    function new(string name="dut_sequence");
6      super.new(name);
7    endfunction
8
9    virtual task body();
10     dut_txn tx;
11     int pipeDepth = 3;
12     bit firstCycle = 1;
13
```

- Sequência de estímulos aleatórios para o DUT
- Cria e inicializa objetos do tipo `dut_txn`
- Executada pelo sequencer a partir do teste principal

microarquitetura/uvm/dut_seq.sv

```
14      // ------------------------------------------
15      // Fase 1 | Inicializa memoria do DUT (256 stores)
16      // Escolhido arbitrariamente o valor de 255*255
17      // ------------------------------------------
18        tx = dut_txn::type_id::create($sformatf("init_store"));
19        tx.cmd_in = 7'b010_0010; // opcode = 010 (multiplicacao = din_2 * din_1;
20        tx.din_1  = 255;         // endereço
21        tx.din_2  = 255;         // pode ignorar
22        tx.din_3  = 0;            // idem
23        // repete 3x para compensar latência do multi-cicle
24        for (int cycle = 0; cycle < pipeDepth; cycle++) begin
25          start_item(tx);
26          finish_item(tx);
27        end
28
```

- Inicializa a memória do DUT com 256 valores de teste (resultado da operação aritmética 255*255 pois não há instruções de carregamento imediato na microarquitetura. Valor arbitrário).
- Cria transação `dut_txn` configurando opcode e operandos.
- Envia repetidamente ao driver para compensar a latência multiciclo.

microarquitetura/uvm/dut_seq.sv

```
29      //Salva 255*255 na memoria (Store)
30      for (int addr = 0; addr < 256; addr++) begin
31        tx = dut_txn::type_id::create($sformatf("init_store_%0d", addr));
32        tx.cmd_in = 7'b000_0110; // opcode = 6 (store), selects din_1 as address,
          ↪  din_2/din_3 ignored
33        tx.din_1  = addr;        // endereço
34        tx.din_2  = 0;           // pode ignorar
35        tx.din_3  = 0;           // idem
36        // repete 3x para compensar latência do multi-cicle
37        for (int cycle = 0; cycle < pipeDepth; cycle++) begin
38          start_item(tx);
39          finish_item(tx);
40        end
41      end
42
43      `uvm_info("SEQ", "Inicializacao da memoria (256 stores) concluida", UVM_LOW)
44
```

- Salva 255*255 em cada endereço da memória do DUT usando opcode store.
- Cria transações `dut_txn` configurando endereço em `din_1`.
- Envia cada transação 3 vezes para compensar a latência multiciclo.

microarquitetura/uvm/dut_seq.sv

```
45      // ------------------------------------------
46      // Fase 2 | Verificacao da memoria inicializada (256 stores)
47      //Le 255*255 da memoria em todas as posicoes
48      // ------------------------------------------
49      for (int addr = 0; addr < 256; addr++) begin
50        tx = dut_txn::type_id::create($sformatf("init_store_%0d", addr));
51        tx.cmd_in = 7'b000_0101; // opcode = 5 (load = mem[addr])
52        tx.din_1  = addr;        // endereco
53        tx.din_2  = 0;           // pode ignorar
54        tx.din_3  = 0;           // idem
55        // repete 3x para compensar latência do multi-cicle
56        for (int cycle = 0; cycle < pipeDepth; cycle++) begin
57          start_item(tx);
58          finish_item(tx);
59        end
60      end
61
62      `uvm_info("SEQ", "Inicializacao da memoria (256 stores) concluida", UVM_LOW)
63
```

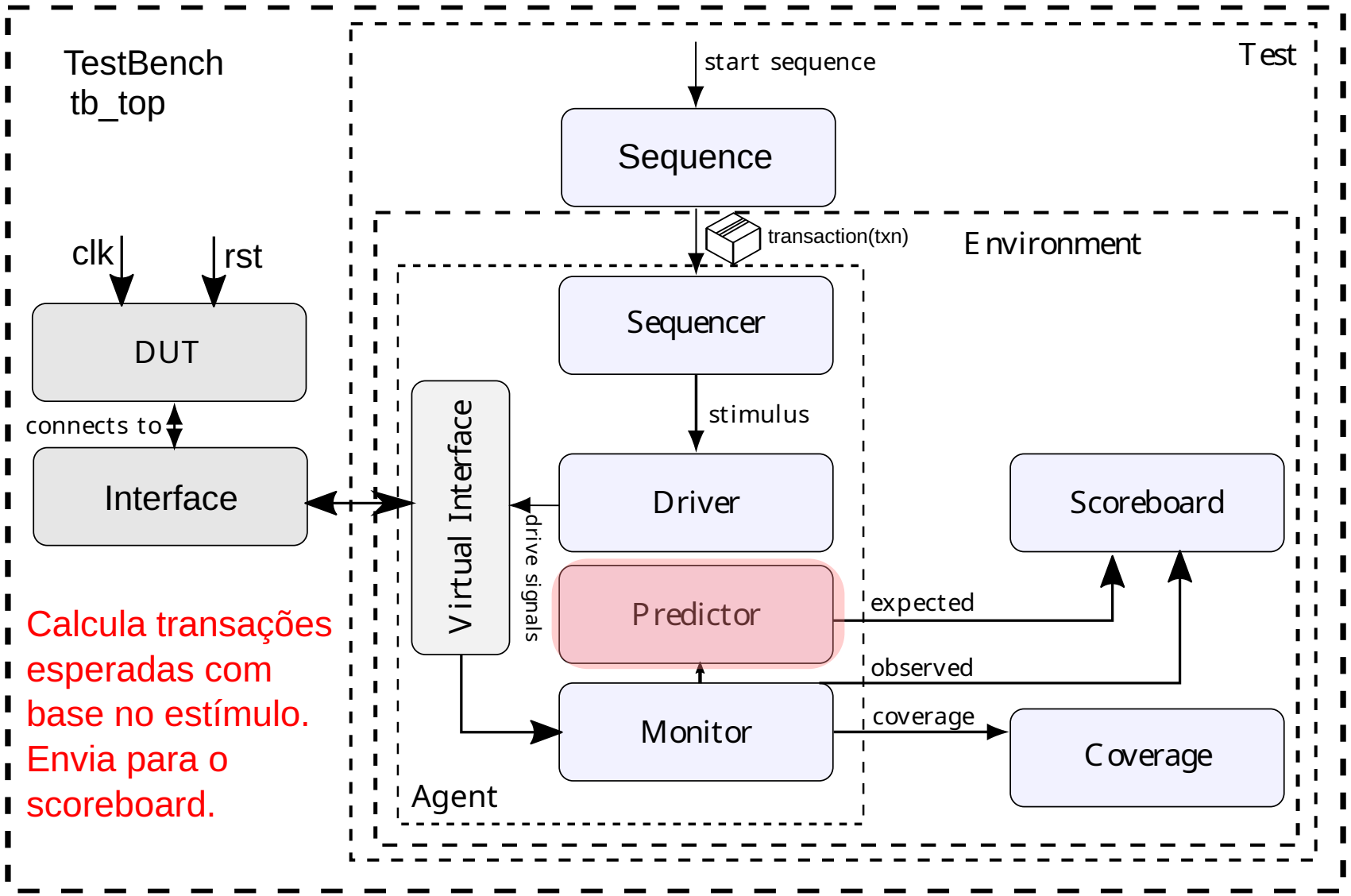
- Lê 255*255 de cada endereço da memória para verificar inicialização.
- Cria transações `dut_txn` configurando opcode load e endereço em `din_1`.
- Envia cada transação 3 vezes para compensar a latência multiciclo.

microarquitetura/uvm/dut_seq.sv

```
64      // ------------------------------------------
65      // Fase 3 | Geracao de estímulos aleatorios - distribuicao normal
66      // ------------------------------------------
67      repeat (100000) begin
68        tx = dut_txn::type_id::create("tx");
69        assert(tx.randomize());
70        for (int cycle = 0; cycle < pipeDepth; cycle++) begin
71          start_item(tx);
72          finish_item(tx);
73        end
74      end
75
76    endtask
77  endclass
```

- Gera 100.000 transações aleatórias usando `randomize()`.
- Cria objetos `dut_txn` com distribuição de valores normal.
- Envia cada transação várias vezes para compensar latência multiciclo.

UVM Transaction - dut_predictor.sv



microarquitetura/uvm/dut_predictor.sv

```
1  //classe dut_predictor
2  class dut_predictor extends uvm_component;
3    `uvm_component_utils(dut_predictor)
4
5    uvm_analysis_imp#(dut_txn, dut_predictor) stim_imp;
6    uvm_analysis_port#(dut_txn) analysis_port;
7
8    localparam int WIDTH = 8;
9
10   // Historico interno
11   // Salva dout_low e dut_high anteriores para
12   // instrucoes que os reutilizam
13   //Comecam como 0 devido a power on reset
14   logic [WIDTH-1:0] dout_low_prev = '0;
15   logic [WIDTH-1:0] dout_high_prev = '0;
16   //Mantem flag de error anterior devido ao
17   //caso de uso em din_high / din_low de
18   //resultados anteriores invalidos (div/0)
19   logic error_flag_prev = '0;
20   //mantem flag de zero anterior para instrucoes
21   //NOP
22   logic zero_flag_prev = '0;
23   //memoria simples para load/store
24   logic [2*WIDTH-1:0] mem_predicted_data [0:7];//255];
25
```

- *Golden Model* que prevê comportamento do DUT.
- Calcula saídas com base em entradas recebidas, para comparação com o monitor.

microarquitetura/uvm/dut_predictor.sv

```
26    function new(string name, uvm_component parent);
27      super.new(name, parent);
28      foreach (mem_predicted_data[i])
29        mem_predicted_data[i] = 16'd65025; //255X255
30      `uvm_info("PREDICTOR", "mem_predicted_data zerada no new()", UVM_LOW)
31          stim_imp = new("stim_imp", this);
32      analysis_port = new("analysis_port", this);
33    endfunction
34
```

- Inicializa histórico interno e memória prevista (`mem_predicted_data`).
- Cria `stim_imp` para receber transações do monitor.
- Cria `analysis_port` para enviar previsões ao scoreboard.

microarquitetura/uvm/dut_predictor.sv

```
35    //select operand A based on cmd[6:5]
36    function automatic logic [WIDTH-1:0] select_A(
37      logic [6:0] cmd,
38      logic [WIDTH-1:0] din_1,
39      logic [WIDTH-1:0] din_2,
40      logic [WIDTH-1:0] din_3,
41      logic [WIDTH-1:0] prev_high
42    );
43      case (cmd[6:5])
44        2'b00: return din_1;
45        2'b01: return din_2;
46        2'b10: return din_3;
47        2'b11: return prev_high;
48        default: return '0;
49      endcase
50    endfunction
51
```

- Seleciona o operando A com base nos bits altos do opcode (`cmd[6:5]`).
- Retorna um dos valores de entrada (`din_1`, `din_2`, `din_3`) ou `prev_high`.
- Garante valor padrão zero caso o opcode não seja reconhecido.

microarquitetura/uvm/dut_predictor.sv

```
52    //select operando B based on cmd [4:3]
53    function automatic logic [WIDTH-1:0] select_B(
54      logic [6:0] cmd,
55      logic [WIDTH-1:0] din_1,
56      logic [WIDTH-1:0] din_2,
57      logic [WIDTH-1:0] din_3,
58      logic [WIDTH-1:0] prev_low
59    );
60      case (cmd[4:3])
61        2'b00: return din_1;
62        2'b01: return din_2;
63        2'b10: return din_3;
64        2'b11: return prev_low;
65        default: return '0;
66      endcase
67    endfunction
68
```

- Seleciona o operando B com base nos bits do opcode (`cmd[4:3]`).
- Retorna um dos valores de entrada (`din_1`, `din_2`, `din_3`) ou `prev_low`.
- Garante valor zero caso o opcode não corresponda a nenhuma opção.

microarquitetura/uvm/dut_predictor.sv

```
69  function automatic logic detectInvalidOperation(
70    logic error_prev,
71    logic [6:0] cmd
72  );
73    // Se a operacao anterior foi invalida (divisao por zero)
74    // e a operacao atual tenta reutilizar o resultado invalido
75    // como operando, entao a operacao atual tambem e invalida
76    if (error_prev) begin
77      if (cmd[6:5] == 2'b11 || cmd[4:3] == 2'b11) begin
78        return 1'b1;
79      end
80    end
81    return 1'b0;
82    endfunction
83
```

- Detecta se operação atual reutiliza resultado inválido anterior (divisão por zero).
- Analisa bits do opcode para identificar operandos dependentes do resultado anterior.
- Retorna 1 se a operação for inválida, 0 caso contrário.

microarquitetura/uvm/dut_predictor.sv

```
84   function void write(dut_txn tx);
85     dut_txn expected_txn = dut_txn::type_id::create("expected_txn");
86
87     logic [WIDTH-1:0] din_1 = tx.din_1;
88     logic [WIDTH-1:0] din_2 = tx.din_2;
89     logic [WIDTH-1:0] din_3 = tx.din_3;
90     logic [6:0] cmd_in = tx.cmd_in;
91     logic [WIDTH-1:0] A, B;
92     logic [2*WIDTH-1:0] alu_result;
93     logic zero_flag, error_flag;
94
95     //define operandos A e B da ALU
96     A = select_A(cmd_in, din_1, din_2, din_3, dout_high_prev);
97     B = select_B(cmd_in, din_1, din_2, din_3, dout_low_prev);
98
99     zero_flag  = 1'b0;
100    error_flag = detectInvalidOperation(error_flag_prev, cmd_in);
101
```

- Cria uma nova transação prevista (`expected_txn`) a partir da entrada.
- Seleciona operandos A e B usando funções `select_A` e `select_B`.
- Inicializa flags `zero_flag` e `error_flag` com base em operação inválida.

microarquitetura/uvm/dut_predictor.sv

```
102     //default ALU conditcion for opcodes alu_result=0 --> flag_zero=1
103     case (cmd_in[2:0])
104       3'b000: begin //sum
105         alu_result = A + B;
106         if (alu_result == 0) zero_flag = 1'b1;
107       end
108       3'b001: begin //subtraction
109         alu_result = A - B;
110         if (alu_result == 0) zero_flag = 1'b1;
111       end
112       3'b010: begin //multiplication
113         alu_result = A * B;
114         if (alu_result == 0) zero_flag = 1'b1;
115       end
116       3'b011: begin//division
117         if (B == 0) begin
118           error_flag = 1'b1;
119           alu_result = {2*WIDTH{1'b1}};
120         end else begin
121           alu_result = A / B;
122         end
123         if (alu_result == 0) zero_flag = 1'b1;
124       end
```

- Calcula resultado da ALU conforme opcode: soma, subtração, multiplicação ou divisão.
- Seta `zero_flag` se o resultado for 0.
- Seta `error_flag` e resultado máximo se divisão por zero ocorrer.

microarquitetura/uvm/dut_predictor.sv

```
125       3'b100, 3'b111: begin //mantem saída e flags anteriores
126         alu_result = {dout_high_prev, dout_low_prev};//nop
127         zero_flag = zero_flag_prev;
128         error_flag = error_flag_prev;
129       end
130       3'b101: begin
131         //load afetara flag zero pois o default da
132         //alu e alu_result=0 --> flag_zero=1
133         //e em caso de operacao anterior invalida ela mantem flags anteriores
134         //e resultado anterior pois nao habilita o ultimo registro
135         if ((cmd_in[6:5] == 2'b11 || cmd_in[4:3] == 2'b11) & error_flag_prev)
136           error_flag = 1'b1;
137         else
138           error_flag = 1'b0;
139         zero_flag = 1'b1;
140         alu_result = mem_predicted_data[A[2:0]];//load
141       end
```

- Opcodes NOP (3'b100,3'b111) mantêm saída e flags anteriores.
- Load (3'b101) atualiza saída da memória prevista e ajusta flags conforme histórico de erro.
- Store (3'b110) mantém flags anteriores e atualiza memória prevista sem alterar saída final.

microarquitetura/uvm/dut_predictor.sv

```
142       3'b110: begin //Store doesnt enable final out register update
143                     //as it registers the last computed value in the
144                     //previous instruction operation
145         alu_result = {dout_high_prev, dout_low_prev};//store
146         zero_flag = zero_flag_prev;
147         error_flag = error_flag_prev;
148         mem_predicted_data[A[2:0]] = alu_result;
149       end
150     endcase
151
```

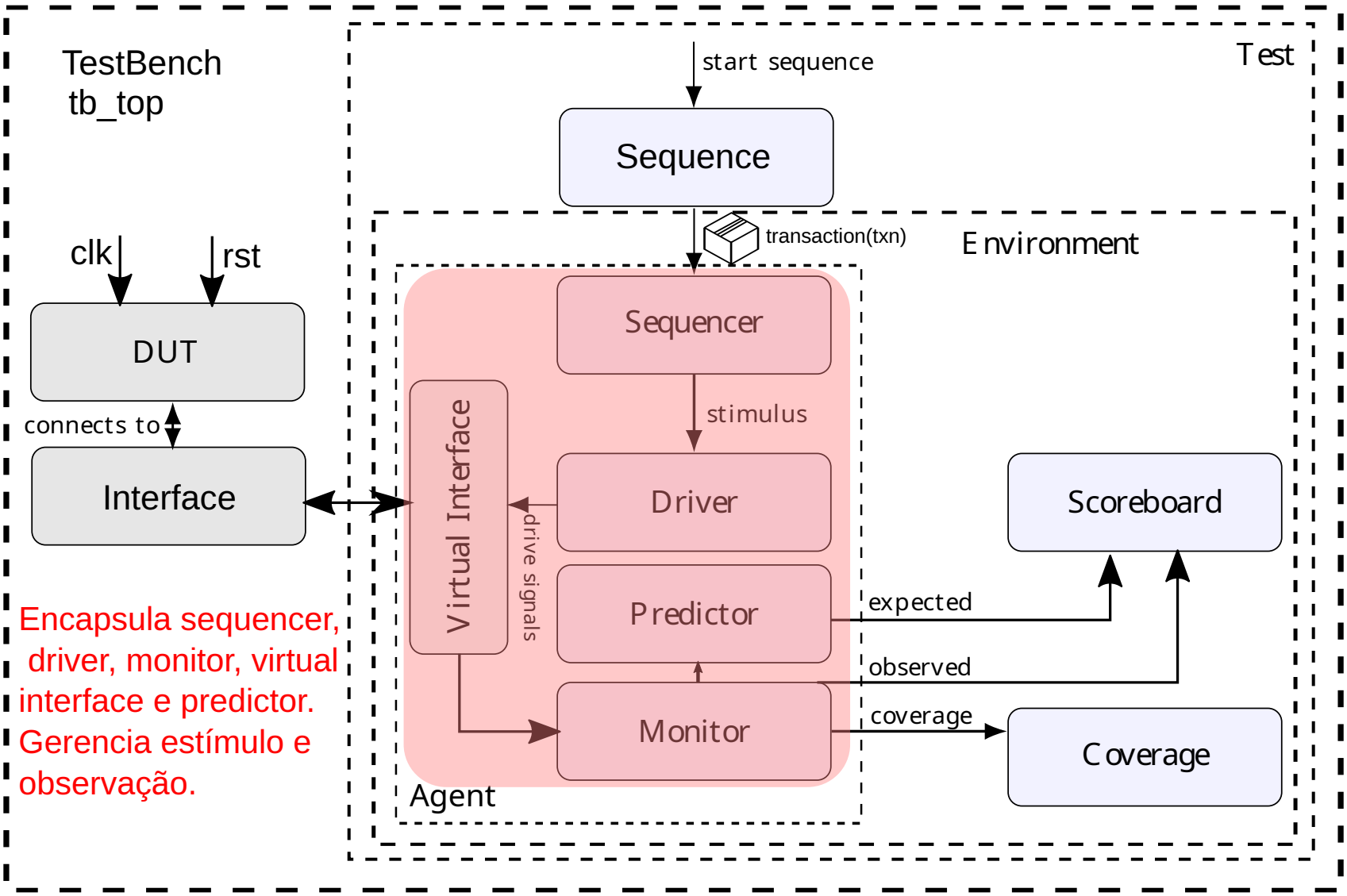
- Opcodes NOP (3'b100,3'b111) mantêm saída e flags anteriores.
- Load (3'b101) atualiza saída da memória prevista e ajusta flags conforme histórico de erro.
- Store (3'b110) mantém flags anteriores e atualiza memória prevista sem alterar saída final.

microarquitetura/uvm/dut_predictor.sv

```
152     dout_low_prev  = alu_result[WIDTH-1:0];
153     dout_high_prev = alu_result[2*WIDTH-1: WIDTH];
154     error_flag_prev = error_flag;
155     zero_flag_prev  = zero_flag;
156
157     expected_txn.cmd_in    = cmd_in;
158     expected_txn.din_1     = din_1;
159     expected_txn.din_2     = din_2;
160     expected_txn.din_3     = din_3;
161     expected_txn.dout_low  = dout_low_prev;
162     expected_txn.dout_high = dout_high_prev;
163     expected_txn.zero      = zero_flag;
164     expected_txn.error     = error_flag;
165     expected_txn.cpu_rdy   = 1'b1;
166
167     analysis_port.write(expected_txn);
168   endfunction
169 endclass
```

- Atualiza histórico interno (`dout_low_prev`, `dout_high_prev`, flags).
- Preenche a transação prevista (`expected_txn`) com resultado e flags calculados.
- Publica a transação prevista via `analysis_port` para o scoreboard.

UVM Transaction - dut_agent.sv



microarquitetura/uvm/dut_agent.sv

```
1  //classe dut_agent
2  `ifndef DUT_AGENT_SV
3  `define DUT_AGENT_SV
4
5  class dut_agent extends uvm_agent;
6    `uvm_component_utils(dut_agent)
7
8    // Subcomponentes
9    dut_sequencer sqr;
10   dut_driver    driver;
11   dut_monitor   monitor;
12   dut_predictor predictor;
13
14   // Interface compartilhada
15   virtual dut_if vif;
16
17   // Ports para o scoreboard
18   uvm_analysis_port#(dut_txn) expected_port;
19   uvm_analysis_port#(dut_txn) observed_port;
20
```

- Agente UVM que encapsula sequencer, driver, monitor e predictor.
- Possui interface virtual (`vif`) compartilhada entre subcomponentes.
- Contém analysis ports para conectar ao scoreboard (`expected_port`, `observed_port`).

microarquitetura/uvm/dut_agent.sv

```
21    // ------------------------------------------------
22    // Construtor
23    // ------------------------------------------------
24    function new(string name, uvm_component parent);
25      super.new(name, parent);
26      expected_port = new("expected_port", this);
27      observed_port = new("observed_port", this);
28    endfunction
29
```

- Construtor chama `super.new` para inicializar a hierarquia UVM.
- Cria `expected_port` para enviar transações previstas ao scoreboard.
- Cria `observed_port` para enviar transações observadas ao scoreboard.

microarquitetura/uvm/dut_agent.sv

```
30    // ------------------------------------------------
31    // build_phase
32    // ------------------------------------------------
33    function void build_phase(uvm_phase phase);
34      super.build_phase(phase);
35
36      sqr       = dut_sequencer ::type_id::create("sqr", this);
37      driver    = dut_driver    ::type_id::create("driver", this);
38      monitor   = dut_monitor   ::type_id::create("monitor", this);
39      predictor = dut_predictor ::type_id::create("predictor", this);
40
41      // Interface compartilhada
42      if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif))
43        `uvm_fatal("NOVIF", "Virtual interface nao encontrada")
44
45      // Injeta a mesma interface nos subcomponentes
46      uvm_config_db#(virtual dut_if)::set(this, "driver",  "vif", vif);
47      uvm_config_db#(virtual dut_if)::set(this, "monitor", "vif", vif);
48    endfunction
49
```

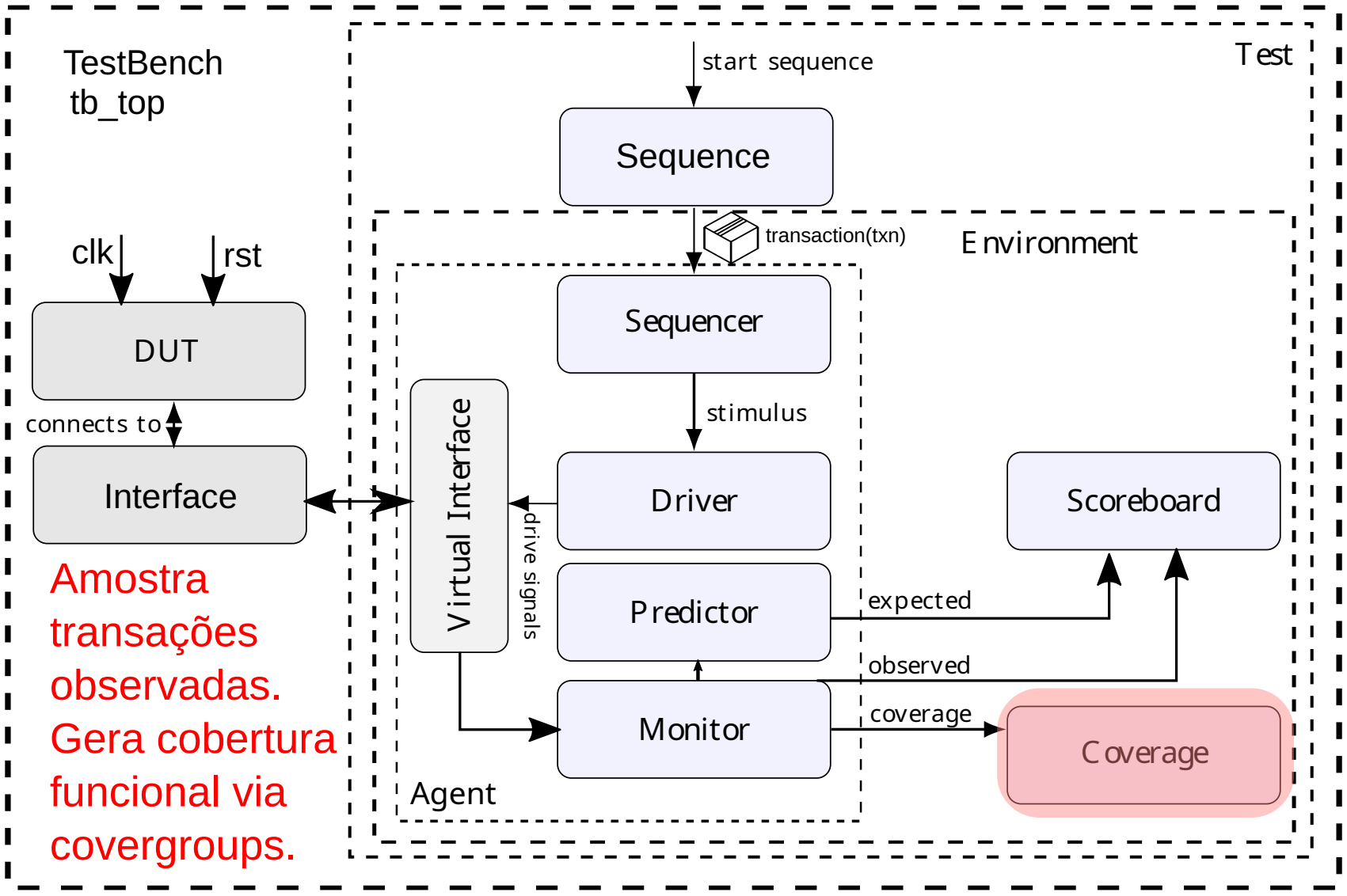
- Instancia sequencer, driver, monitor e predictor usando `type_id::create`.
- Recupera a interface virtual do DUT via `uvm_config_db` e verifica existência.
- Injeta a mesma interface nos subcomponentes driver e monitor para comunicação consistente.

microarquitetura/uvm/dut_agent.sv

```
50    // ------------------------------------------------
51    // connect_phase
52    // ------------------------------------------------
53    function void connect_phase(uvm_phase phase);
54      super.connect_phase(phase);
55
56      // Sequencer -->  Driver
57      driver.seq_item_port.connect(sqr.seq_item_export);
58
59      // Monitor -->  Predictor
60      monitor.analysis_port.connect(predictor.stim_imp);
61
62      // Predictor -->  Scoreboard (esperado)
63      predictor.analysis_port.connect(expected_port);
64
65      // Monitor -->  Scoreboard (observado)
66      monitor.analysis_port.connect(observed_port);
67    endfunction
68
69  endclass
70
```

- Conecta sequencer ao driver para envio de transações (`seq_item_port` → `seq_item_export`).
- Conecta monitor ao predictor para enviar estímulos observados (`analysis_port` → `stim_imp`).
- Conecta predictor e monitor aos ports do scoreboard para resultados esperados e observados.

UVM Transaction - dut_cov.sv



microarquitetura/uvm/dut_cov.sv

```
1  //classe dut_cov
2  // Cobertura funcional do DUT.
3  // Recebe transacoes do monitor via analysis_export.
4  /*
5  | Categoria              | Descricao                              |
6  | ---------------------- | -------------------------------------- |
7  | `cmd_in`               | Operacoes realizadas                   |
8  | `zero`                 | Flag de resultado zero                 |
9  | `error`                | Flag de erro (ex: divisão por zero)    |
10 | `cpu_rdy`              | Sincronizacao de termino da operacao   |
11 | `cross cmd_in, error`  | Erros por tipo de operacao             |
12 | `cross cmd_in, zero`   | Resultados zero por tipo de operacao   |
13 */
14 class dut_cov extends uvm_subscriber#(dut_txn);
15   `uvm_component_utils(dut_cov)
16
17   // Mirror fields from the transaction
18   bit [1:0] cmd_in;
19   bit       zero;
20   bit       error;
21   bit       cpu_rdy;
22   bit[1:0]  state;
23
```

- Subscritor UVM que recebe transações do monitor via `analysis_export`.
- Captura campos do DUT (`cmd_in`, `zero`, `error`, `cpu_rdy`, `state`).
- Define cobertura funcional para monitorar operações, flags e cruzamentos relevantes.

microarquitetura/uvm/dut_cov.sv

```
24    // --- Covergroup ---
25    covergroup dut_cg;
26      coverpoint cmd_in {//boa pratica copair os sinais da txn para
27        bins add    = {0};//variaveis novas e evitar amostragens
28        bins sub    = {1};//sobrepostas
29        bins mul    = {2};
30        bins div    = {3};
31        bins others = default;
32      }
33      coverpoint zero    { bins zero_set = {1}; bins zero_clr = {0}; }
34      coverpoint error   { bins err_set  = {1}; bins err_clr  = {0}; }
35      coverpoint cpu_rdy { bins rdy_set  = {1}; bins rdy_clr  = {0}; }
36
37      cross cmd_in, error;
38      cross cmd_in, zero;
39
40    coverpoint state {
41      bins clear = {0};
42      bins fetch = {1};
43      bins exec  = {2};
44      bins store = {3};
45    }
46  endgroup
```

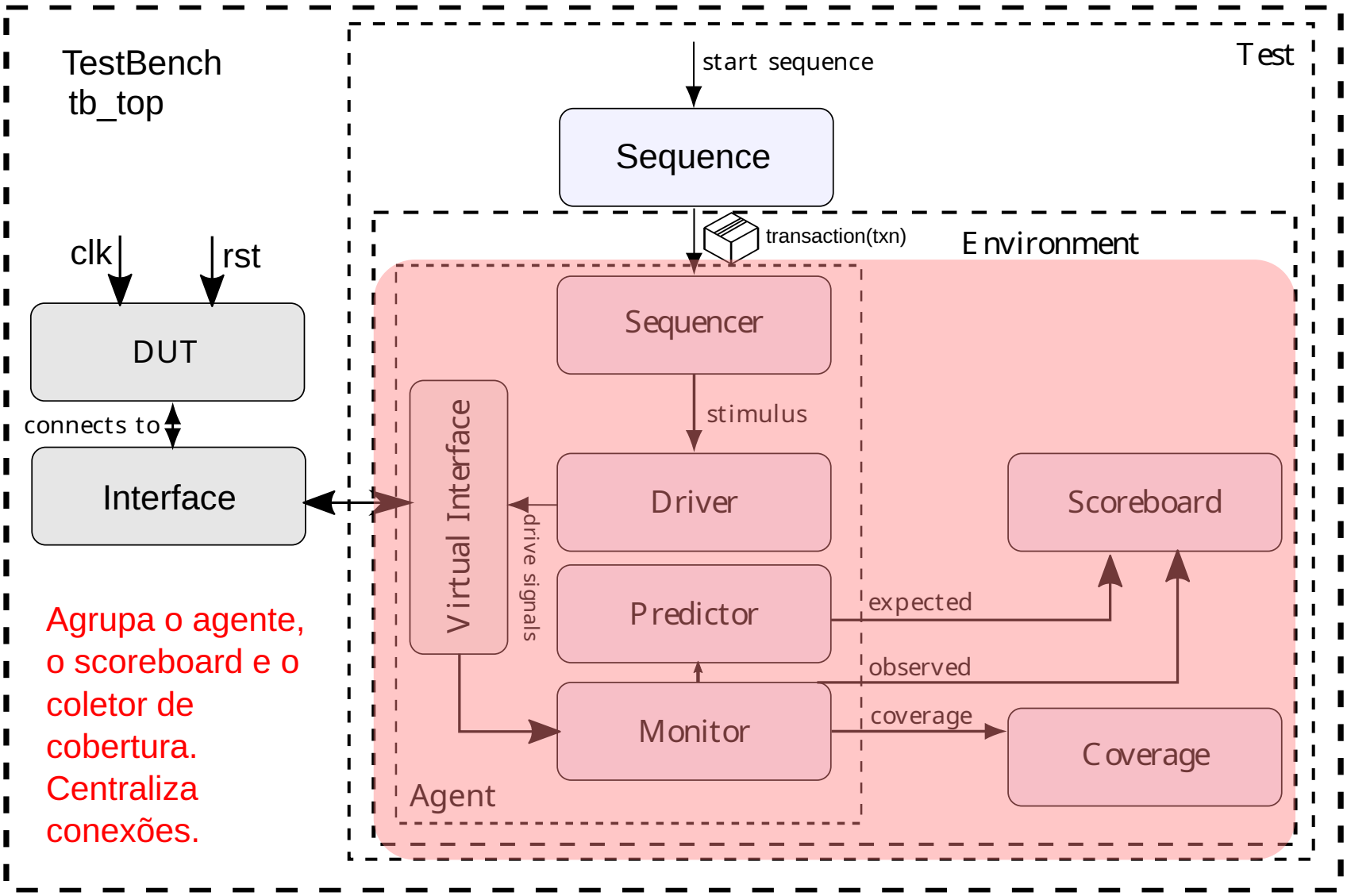
- Define coverpoints para sinais do DUT: `cmd_in`, `zero`, `error`, `cpu_rdy`, `state`.
- Cria bins para cada valor significativo, incluindo default para outros casos.
- Define cruzamentos (`cross`) para analisar combinação de opcode com flags (`error`, `zero`).

microarquitetura/uvm/dut_cov.sv

```
47
48    // --- Construtor ---
49    function new(string name, uvm_component parent);
50      super.new(name, parent);
51      dut_cg = new();
52    endfunction
53
54    // --- write() chamado automaticamente quando monitor publica ---
55    function void write(dut_txn t);
56      cmd_in  = t.cmd_in;
57      zero    = t.zero;
58      error   = t.error;
59      cpu_rdy = t.cpu_rdy;
60      state   = t.state;
61      dut_cg.sample();
62    endfunction
63
64  endclass
65
```

- Construtor inicializa a covergroup (`dut_cg = new()`).
- Método `write()` recebe transações do monitor automaticamente.
- Atualiza campos internos e amostra a covergroup para coleta de cobertura.

UVM Transaction - dut_env.sv



microarquitetura/uvm/dut_env.sv

```
1  //classe dut_env
2  //=====================================================
3  // dut_env.sv
4  // Ambiente UVM generico com agent, scoreboard e (opcional) coverage
5  //=====================================================
6
7  class dut_env extends uvm_env;
8    `uvm_component_utils(dut_env)
9
10   // ---------------------------------------------------
11   // Subcomponentes
12   // ---------------------------------------------------
13   dut_agent       agent;
14   dut_scoreboard  sb;
15   dut_cov         cov;   // opcional
16
```

- Ambiente UVM que encapsula os subcomponentes do testbench.
- Contém o agent (`dut_agent`), o scoreboard (`sb`) e coverage opcional (`cov`).
- Coordena comunicação e integração entre monitor, predictor e scoreboard.

microarquitetura/uvm/dut_env.sv

```
16
17    // ------------------------------------------------
18    // Construtor
19    // ------------------------------------------------
20    function new(string name, uvm_component parent);
21      super.new(name, parent);
22    endfunction
23
```

- Construtor chama `super.new` para inicializar a hierarquia UVM.
- Não cria subcomponentes diretamente; apenas inicializa o componente ambiente.
- Permite futura configuração de parâmetros e integração com testbench.

microarquitetura/uvm/dut_env.sv

```
24    // ------------------------------------------------
25    // Build phase
26    // ------------------------------------------------
27    function void build_phase(uvm_phase phase);
28      bit enable_cov = 1;
29      super.build_phase(phase);
30
31      agent = dut_agent::type_id::create("agent", this);
32      sb    = dut_scoreboard::type_id::create("sb", this);
33
34      // Permite habilitar ou desabilitar coverage via config_db
35      void'(uvm_config_db#(bit)::get(this, "", "enable_cov", enable_cov));
36
37      if (enable_cov) begin
38        cov = dut_cov::type_id::create("cov", this);
39        `uvm_info("ENV", "Coverage collector habilitado", UVM_LOW)
40      end
41      else begin
42        `uvm_info("ENV", "Coverage collector desabilitado", UVM_LOW)
43      end
44    endfunction
45
```

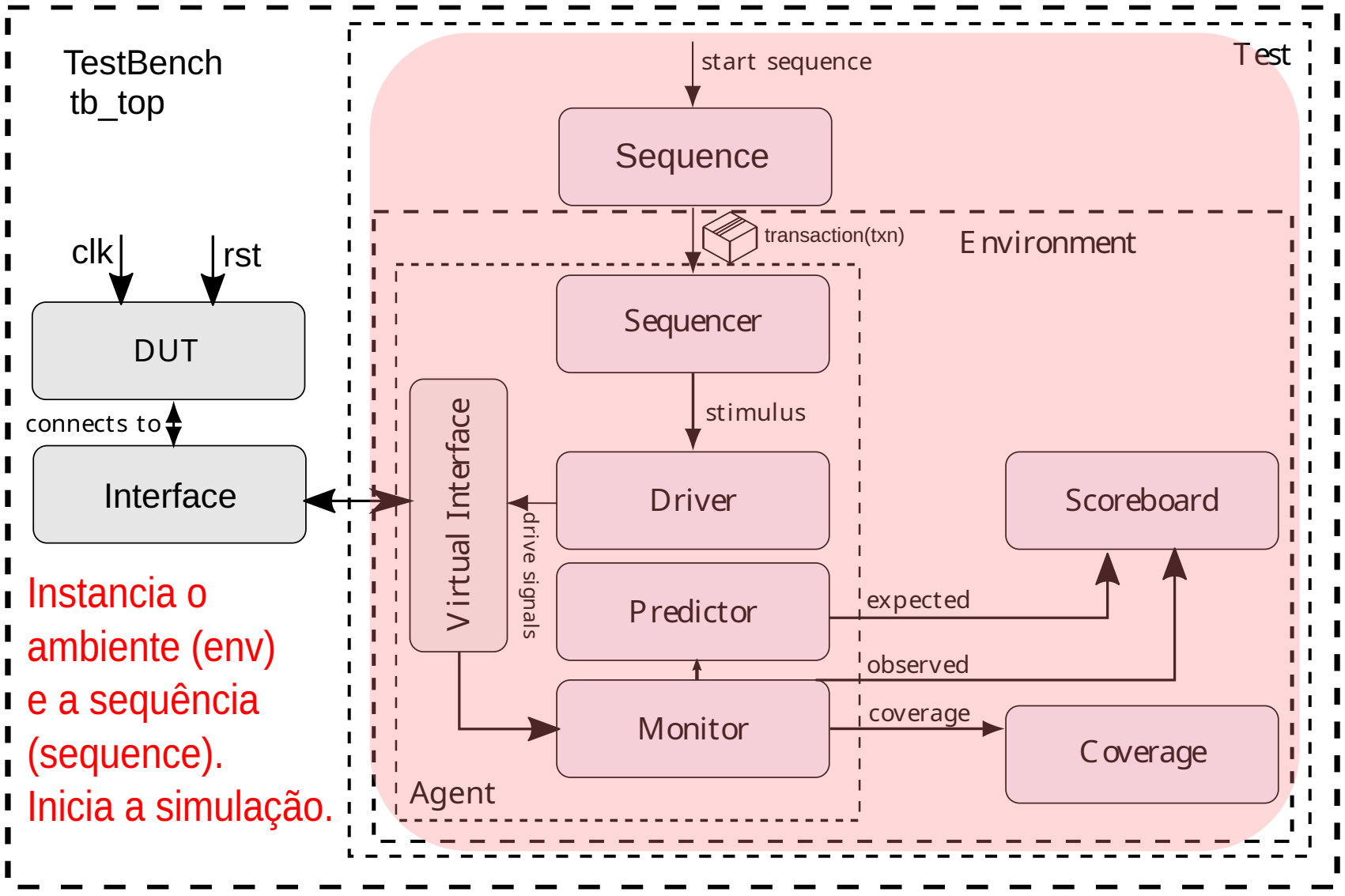
- Instancia subcomponentes: `agent`, `scoreboard` e opcional `cov`.
- Permite habilitar ou desabilitar coverage via `uvm_config_db`.
- Emite mensagens informativas sobre o status do coverage.

microarquitetura/uvm/dut_env.sv

```
46    // ------------------------------------------------
47    // Connect phase
48    // ------------------------------------------------
49    function void connect_phase(uvm_phase phase);
50    super.connect_phase(phase);
51
52    // Conecta o trafego de referência (expected) do predictor -->  scoreboard
53    agent.expected_port.connect(sb.expected_export);
54
55    // Conecta o trafego observado (monitor) -->  scoreboard
56    agent.observed_port.connect(sb.actual_export);
57
58    // (Opcional) conecta o trafego observado ao coverage collector
59    if (cov != null)
60      agent.observed_port.connect(cov.analysis_export);
61  endfunction
62
63  endclass
```

- Conecta o tráfego esperado do predictor ao scoreboard (`expected_port` → `expected_export`).
- Conecta o tráfego observado do monitor ao scoreboard (`observed_port` → `actual_export`).
- Conecta o tráfego observado ao coverage collector se habilitado (`cov != null`).

UVM Transaction - dut_test.sv



microarquitetura/uvm/dut_test.sv

```
1  //classe dut_test
2  `ifndef DUT_TEST_SV
3  `define DUT_TEST_SV
4
5  class dut_test extends uvm_test;
6    `uvm_component_utils(dut_test)
7
8    dut_env       env;
9    dut_sequence  seq;
10
11   // -------------------------------------------------
12   // Construtor
13   // -------------------------------------------------
14   function new(string name, uvm_component parent);
15     super.new(name, parent);
16   endfunction
17
```

- Teste UVM que instancia o ambiente e sequência de estímulos.
- Contém `dut_env` (`env`) e `dut_sequence` (`seq`).
- Construtor chama `super.new` para inicializar a hierarquia UVM.

microarquitetura/uvm/dut_test.sv

```
18    // ------------------------------------------------
19    // Build phase
20    // ------------------------------------------------
21    function void build_phase(uvm_phase phase);
22      virtual dut_if vif;
23      bit enable_cov = 1;
24      super.build_phase(phase);
25
26      // Instancia o ambiente
27      env = dut_env::type_id::create("env", this);
28
29      // Obtem a virtual interface configurada externamente
30      if (!uvm_config_db#(virtual dut_if)::get(null, "", "vif", vif))
31        `uvm_fatal("NOVIF", "Virtual interface nao configurada. Use uvm_config_db::set
          ↪  antes de run_test().")
32
33      // Injeta a interface no agent
34      uvm_config_db#(virtual dut_if)::set(this, "env.agent", "vif", vif);
35
36      // Habilita ou desabilita o coletor de cobertura
37      uvm_config_db#(bit)::set(this, "env", "enable_cov", enable_cov);
38    endfunction
39
```

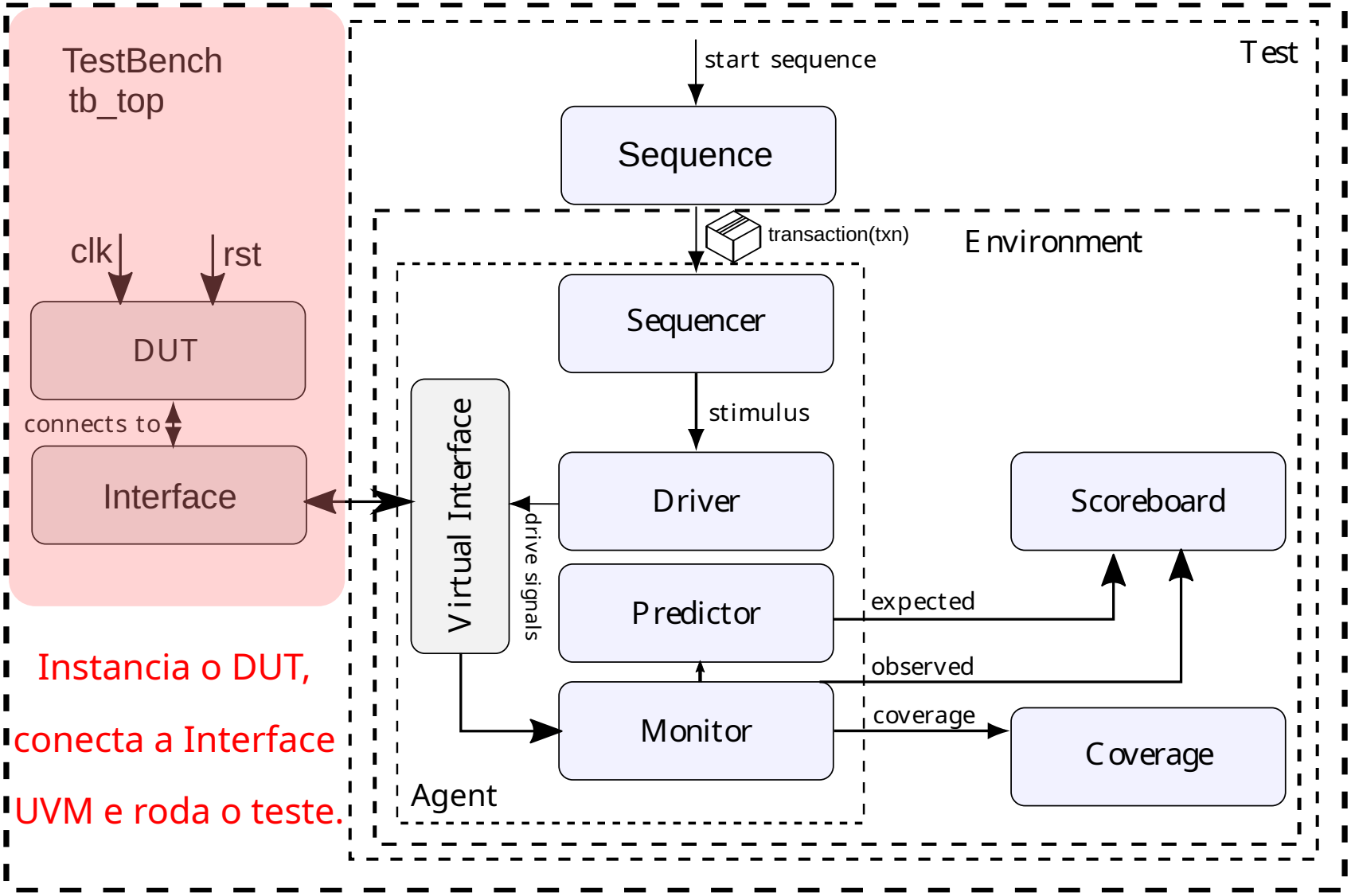
- Instancia o ambiente UVM (`env`) contendo agent, scoreboard e coverage.
- Recupera a interface virtual configurada externamente via `uvm_config_db`.
- Injeta a interface no agent e configura habilitação do coverage collector.

microarquitetura/uvm/dut_test.sv

```
40    // ------------------------------------------------
41    // Run phase
42    // ------------------------------------------------
43    task run_phase(uvm_phase phase);
44      phase.raise_objection(this);
45
46      // Cria e inicia a sequência principal
47      seq = dut_sequence::type_id::create("seq");
48      if (seq == null)
49        `uvm_fatal("NOSEQ", "Sequencia principal nao foi criada.")
50
51      `uvm_info("TEST", "Iniciando sequencia principal", UVM_MEDIUM)
52      seq.start(env.agent.sqr);
53
54      phase.drop_objection(this);
55    endtask
56
57  endclass
58
59  `endif // DUT_TEST_SV
60
```

- Levanta objection para sinalizar início da simulação (`phase.raise_objection(this)`).
- Cria e inicia a sequência principal (`seq.start(env.agent.sqr)`).
- Solta a objection ao final da sequência (`phase.drop_objection(this)`), permitindo finalização.

UVM Transaction - tb_top.sv



microarquitetura/uvm/tb_top.sv

```
1  //classe tb_top
2  `include "uvm_macros.svh"
3  import uvm_pkg::*;
4  import tb_package::*;
5
6  module tb_top;
7
8    logic clk, rst;
9    dut_if dut_if_inst(clk, rst);
10
11   // DUT real
12   top u_dut(
13     .clk      (clk),
14     .rst      (rst),
15     .cmd_in   (dut_if_inst.cmd_in),
16     .din_1    (dut_if_inst.din_1),
17     .din_2    (dut_if_inst.din_2),
18     .din_3    (dut_if_inst.din_3),
19     .dout_low (dut_if_inst.dout_low),
20     .dout_high(dut_if_inst.dout_high),
21     .cpu_rdy  (dut_if_inst.cpu_rdy),
22     .zero     (dut_if_inst.zero),
23     .error    (dut_if_inst.error)
24   );
25
```

- Módulo top-level para simulação
- Instancia DUT e interface
- Configura a interface virtual
- Chama `run_test()`

microarquitetura/uvm/tb_top.sv

```
26    // Clock/reset generation
27    initial begin
28      clk = 0;
29      forever #5 clk = ~clk;
30    end
31
32    initial begin
33      rst = 1;
34      #3 rst = 0;
35    end
36
37    // Conecta o vif ao UVM
38    initial begin
39      uvm_config_db#(virtual dut_if)::set(null,
      ↪  "*", "vif", dut_if_inst);
40
41      // Forca o sinal interno da FSM para a
      ↪  Interface
42      force dut_if_inst.state =
      ↪  tb_top.u_dut.fsm.state;
43      run_test("dut_test");
44
45    end
46
47  endmodule
48
```

- Módulo top-level para simulação
- Instancia DUT e interface
- Configura a interface virtual
- Chama `run_test()`

filelist.f

- Lista todos os arquivos do projeto
- Usado pelo ‘xrun‘ para compilar
- Suporta diretórios com ‘+incdir+‘

```
1  +incdir+./uvm
2  +incdir+./rtl
3
4  #UVM Files
5  uvm/dut_if.sv
6  uvm/tb_package.sv
7  uvm/tb_top.sv
8
9  # RTL  (todos os modulos)
10 rtl/mux4.sv
11 rtl/mux4_registered.sv
12 rtl/ALU.sv
13 rtl/memory.sv
14 rtl/register_bank.sv
15 rtl/control.sv
16 rtl/top.sv
```

Makefile

- Automatiza e evita erros manuais na simulação
- Comandos: ‘make‘, ‘make gui‘, ‘make clean‘
- Usa ‘xrun‘ da Cadence

```
1  # Makefile para simular com Xcelium (xrun)
2  XRUN = xrun
3  UVM_HOME=/apps/cds/XCELIUM2409/tools.lnx86/methodology/UVM/CDNS-1.2
4  XRUN_FLAGS = -uvmhome $(UVM_HOME) -uvm -coverage all -sv -64bit
       -access +rwc -clean -nowarn DLCPTH
5
6  # Arquivo contendo os modulos do projeto
7  FILELIST = filelist.f
8
9  all:
10   $(XRUN) $(XRUN_FLAGS) -f $(FILELIST) +UVM_TESTNAME=dut_test
         +UVM_NO_RELNOTES
11
12 gui:
13   $(XRUN) $(XRUN_FLAGS) -f $(FILELIST) -gui +UVM_TESTNAME=dut_test
         +UVM_NO_RELNOTES
14
15 clean:
16   rm -rf xrun.history xcelium.d INCA_libs *.log *.key *.shm *.vcd
         *.vpd worklib csrc
```

Executando o Teste

Como rodar o MakeFile e Executar o teste

- No Linux, qualquer pasta contendo um arquivo Makefile com sintaxe correta automatiza a execução de uma série de ferramentas. É muito utilizando para compilação de programas pois podem ser descritos inúmeros parâmetros de compilação.
- Para a automação de nossas ferramentas de descrição, compilação, síntese e até mesmo layout, o Makefile também pode ser utilizado, facilitando o processo.

Para rodar o Makefile fornecido, digite no terminal:

```
make clean
make > relatorio.txt
```

Isso gerará o relatório de saída definida pelo scoreboard em **relatorio.txt**. É imprescindível rodar o **make clean** sempre antes do make.

Hands-On

Atividade

O setup UVM apresentado é apenas uma forma de se fazer um teste aplicando UVM.

1. Rode o ambiente UVM fornecido para o teste da microarquitetura com o MakeFile fornecido.
2. Verifique o report gerado.
3. Explique com suas palavras como o UVM funciona (não mais que 2 páginas), enunciando seus componentes e suas funções. Relate também se o ambiente UVM fornecido testa apropriadamente a microarquitetura e se o teste foi bem sucedido.