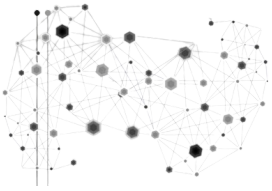


CI INOVADOR – POLO UFCG – TRILHA DIGITAL

PROJETO FINAL
INTRODUÇÃO A VERIFICAÇÃO

Campina Grande, PB

FEVEREIRO / 2025

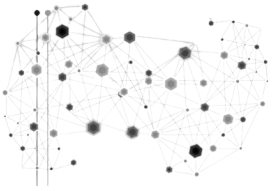
IDENTIFICAÇÃO

Discente: JAQUELINE FERREIRA DE BRITO

Docente: DRª. JOSEANA MACÊDO FECHINE RÉGIS DE ARAÚJO

Curso: ESPECIALIZAÇÃO EM MICROELETRÔNICA

Período: 1



CI INOVADOR
RESIDÊNCIA EM MICROELETRÔNICA

GOVERNO FEDERAL
BRASIL
UNIÃO E RECONSTRUÇÃO

SUMÁRIO

1. INTRODUÇÃO

O objetivo foi garantir a verificação das operações da ULA (ADD, SUB, MUL, DIV), com ênfase especial em casos críticos e condições de contorno. Este projeto apresenta a progressão das metodologias de verificação aplicadas a uma ULA de 8 bits, começando com uma abordagem básica e culminando na implementação completa em UVM (Metodologia Universal de Verificação).

A verificação foi organizada em quatro níveis de complexidade crescente:

1. Testbench Simples: Implementação fundamental com 18 testes direcionados à funcionalidade.
2. Self-Checking com Vetores: Abordagem estruturada utilizando 100 vetores de teste predefinidos.
3. Self-Checking: Metodologia híbrida com 100 testes combinando casos determinísticos e aleatórios.
4. UVM: Estrutura de verificação completa incluindo: Monitor, Sequencer, Sequence Items, Scoreboard e Driver.

O objetivo foi assegurar a verificação completa das operações da ULA (ADD, SUB, MUL, DIV), com foco especial em casos críticos e condições de contorno.

2. OBJETIVO

Desenvolvimento de três abordagens de verificação para ULA de 8 bits:

1. **Testbench Simples**
    - Estrutura básica de verificação
    - 18 testes (8 determinísticos + 10 aleatórios)
    - Assertions fundamentais
    - Interface direta com DUT
2. **Testbench Self-Checking**
    - Sistema de auto-verificação
    - 100 testes (12 determinísticos + 88 aleatórios)
    - Verificação automatizada
    - Geração de relatórios detalhados
3. **Testbench Self-Checking com Vetores**
    - 100 testes predefinidos
    - 25 testes por operação (ADD, SUB, MUL, DIV)
    - Verificação determinística
    - Ideal para regressão

Conjunto de Testes

Verificação mínima por operação:

- **ADD**: Soma com/sem carry
- **SUB**: Subtração normal/caso A=B
- **MUL**: Multiplicação normal/por zero
- **DIV**: Divisão exata/com resto

Análise de Resultados

Para cada implementação:

- Validação funcional
- Testes de casos de erro
- Documentação de I/O

- Análise de formas de onda

Subprojeto 2: Verificação Ambiente UVM

Implementação de blocos funcionais:

1. **Monitor**
    - Observação de sinais
    - Captura de transações
    - Interface com scoreboard
2. **Sequencer**
    - Coordenação de estímulos
    - Gerenciamento de transações
    - Controle de fluxo
3. **Sequence Items**
    - Definição de padrões
    - Restrições aleatórias
    - Estruturação de testes
4. **Scoreboard**
    - Verificação de resultados
    - Análise de comportamento
    - Estatísticas de teste
5. **Driver**
    - Interface com DUT
    - Controle de timing
    - Aplicação de estímulos

Resultados e Análise

- Documentação de testes
- Análise de cobertura
- Formas de onda
- Eficácia da verificação

Análise Comparativa

Avaliação das metodologias:

1. **Complexidade**
    - Simples → UVM
    - Curva de aprendizado
    - Reusabilidade
2. **Eficiência**
    - Cobertura de testes
    - Automação
    - Manutenibilidade
3. **Aplicabilidade**
    - Casos de uso
    - Escalabilidade
    - Integração

Este projeto visa estabelecer uma compreensão profunda das diferentes metodologias de verificação, desde abordagens básicas até frameworks profissionais como UVM.

3. DESENVOLVIMENTO E RESULTADOS

3.1 SUBPROJETO 1:

- O design é sincronizado por clock.
- O tipo de reset é ativo alto. (Reset ocorre quando o valor = 1)
- Envie a entrada no ciclo atual, e o DUT (Dispositivo sob teste) dará a saída no próximo ciclo.
- Não suporta transações consecutivas (back-to-back).
- Apenas quatro operações são suportadas (ADD, SUB, MULT, DIV).
- A entrada A deve ser sempre maior ou igual à entrada B.

1. Especificações do design da ULA:

Port Name	Type	Property	Size
Clock	Input	Wire	1 bit
Reset	Input	Wire	1 bit
A	Input	Wire	8 bits
B	Input	Wire	8 bits
ULA_Sel	Input	Wire	4 bits
ULA_Out	Output	Reg	8 bits
CarryOut	Output	Reg	1 bit

ULA_Sel	Operação
4’b0000	A + B
4’b0001	A - B
4’b0010	A * B
4’b0011	A / B
4’b0100	Reserved
4’b1111	Reserved

2. Design ULA em SystemVerilog

```
//------------------------------------------------------------------------------
// Projeto: Simple Arithmetic and Logic Unit (ALU 1.0)
// Aluna: Jaqueline Ferreira de Brito
// Data: 23/02/2025
// Descrição: ULA síncrona com reset ativo alto e latência de 1 ciclo
//------------------------------------------------------------------------------

// Definindo a escala de tempo para a simulação. O tempo de simulação será 1ns por unidade de tempo.
`timescale 1ns/1ps

module ULA (
	input  logic          clock,	// Clock de entrada
	input  logic          reset,	// Reset ativo alto
	input  logic [7:0]  A,	// Entrada A (deve ser >= B)
	input  logic [7:0]  B,	// Entrada B
	input  logic [3:0]  ULA_Sel, // Seletor de operação (4 bits para selecionar entre 16 operações possíveis)
	output logic [7:0]  ULA_Out, // Resultado da operação
	output logic          CarryOut	// Sinal de carry, utilizado em operações como ADD
);

	// Declaração de variáveis internas para armazenar o próximo valor de resultado e carry
	logic [7:0] next_result;	// Próximo valor do resultado
	logic         next_carry;	// Próximo valor do carry

	// Lógica combinacional: avalia as operações com base no seletor ULA_Sel
	always_comb begin
	// Inicializa os valores de next_result e next_carry com os valores atuais de ULA_Out e CarryOut
	next_result = ULA_Out;	// Mantém o valor anterior de ULA_Out
	next_carry  = CarryOut;	// Mantém o valor anterior de CarryOut

	// Se reset não estiver ativo, realiza as operações
	if (!reset) begin
		// Seleção da operação baseada no código do seletor ULA_Sel
		case (ULA_Sel)
		4'b0000: begin	// Operação de adição (ADD)
			{next_carry, next_result} = {1'b0, A} + {1'b0, B}; // Adiciona A e B, considerando o carry
		end
		4'b0001: begin	// Operação de subtração (SUB)
			next_result = A - B;  // Subtrai B de A
			next_carry  = 1'b0;   // Não há carry em uma subtração
		end
		4'b0010: begin	// Operação de multiplicação (MUL)
			next_result = A * B;  // Multiplica A e B
			next_carry  = 1'b0;   // Não há carry em uma multiplicação
		end
		4'b0011: begin	// Operação de divisão (DIV)
			// Se B não for zero, realiza a divisão. Caso contrário, retorna '1' (indicando erro)
```

```
                        next_result = (B != '0) ? A / B : '1;
                        next_carry  = 1'b0;   // Não há carry em uma divisão
                end
                default: begin
                        next_result = '0;   // Caso padrão, resultado igual a zero
                        next_carry  = '0;   // Carry igual a zero
                end
                endcase
        end
        end

        // Registrador de saída: sincroniza os valores com o clock
        always_ff @(posedge clock) begin
        // Se reset estiver ativo, zera os valores de saída
        if (reset) begin
                ULA_Out  <= '0;       // Zera o resultado
                CarryOut <= '0;       // Zera o carry
        end else begin
                ULA_Out  <= next_result; // Atualiza o resultado com o valor calculado
                CarryOut <= next_carry;  // Atualiza o carry com o valor calculado
        end
        end

endmodule
```

A. Implementação dos Testbenchs:

a. Testbench Simples

```
//------------------------------------------------------------------------------
// Projeto: Testbench simples ULA
// Aluna: Jaqueline Ferreira de Brito
// Date: 23/02/2025
// Descrição: Testbench simples com testes determinísticos e aleatórios
//------------------------------------------------------------------------------

// Definindo a escala de tempo para a simulação. O tempo de simulação será 1ns por unidade de
tempo.
`timescale 1ns/1ps

module ULA_simples_tb;

        // Sinais do testbench
        logic          clock;         // Sinal de clock
        logic          reset;         // Sinal de reset
        logic [7:0]  A;               // Entrada A (8 bits)
        logic [7:0]  B;               // Entrada B (8 bits)
        logic [3:0]  ULA_Sel;         // Seletor de operação (4 bits)
        logic [7:0]  ULA_Out;         // Resultado da operação (8 bits)
        logic          CarryOut;      // Sinal de carry (indicativo de overflow ou carry)

        // Instanciação do DUT (Design Under Test - a ULA a ser testada)
        ULA dut (.*);  // Conecta automaticamente todas as entradas e saídas do DUT
```

```
        // Geração de clock
        initial begin
        clock = 0;                  // Inicializa o clock com valor 0
        forever #5 clock = ~clock;  // Alterna o clock a cada 5ns, com período de 10ns
        end

        // Task para aplicar estímulos e esperar resultado
        task automatic apply_test(
        input logic [7:0] in_a,     // Valor para a entrada A
        input logic [7:0] in_b,     // Valor para a entrada B
        input logic [3:0] op        // Operação a ser selecionada pelo ULA_Sel
        );
        @(posedge clock);           // Aplica os valores na borda positiva do clock
        A = in_a;                   // Atribui o valor a A
        B = in_b;                   // Atribui o valor a B
        ULA_Sel = op;               // Atribui a operação ao seletor

        @(posedge clock);           // Espera um ciclo de clock para o resultado ser
calculado
        #1;                         // Pequeno delay para estabilização do valor
        endtask

        // Processo de teste
        initial begin
        // Identificação do testbench
        $display("//---------------------------------------");
        $display("// Iniciando testbench ULA Simples");
        $display("// Data: 23/02/2025");
        $display("// Aluna: Jaqueline Ferreira de Brito");
        $display("//---------------------------------------\n");

        // Reset inicial
        reset = 1;                  // Ativa o reset
        A = 0;                      // Zera A
        B = 0;                      // Zera B
        ULA_Sel = 0;                // Zera o seletor de operação
        repeat(2) @(posedge clock); // Mantém o reset por 2 ciclos de clock
        reset = 0;                  // Desativa o reset
        @(posedge clock);           // Espera um ciclo após o reset

        // Testes de Adição
        $display("\n=== Testes de Adição ===");
        // Teste 1: Soma sem carry
        apply_test(8'h30, 8'h20, 4'b0000); // A = 0x30, B = 0x20, Operação de adição
        $display("ADD: %h + %h = %h (Carry=%b)", A, B, ULA_Out, CarryOut);

        // Teste 2: Soma com carry (overflow)
        apply_test(8'hFF, 8'h01, 4'b0000); // A = 0xFF, B = 0x01, Operação de adição com carry
        $display("ADD: %h + %h = %h (Carry=%b)", A, B, ULA_Out, CarryOut);

        // Testes de Subtração
        $display("\n=== Testes de Subtração ===");
        // Teste 1: Subtração simples (carry sempre 0)
        apply_test(8'h50, 8'h20, 4'b0001); // A = 0x50, B = 0x20, Operação de subtração
        $display("SUB: %h - %h = %h", A, B, ULA_Out);
```

```
        // Teste 2: Subtração com A = B
        apply_test(8'h30, 8'h30, 4'b0001); // A = 0x30, B = 0x30, Operação de subtração
        $display("SUB: %h - %h = %h", A, B, ULA_Out);

        // Testes de Multiplicação
        $display("\n=== Testes de Multiplicação ===");
        // Teste 1: Multiplicação simples
        apply_test(8'h04, 8'h03, 4'b0010); // A = 0x04, B = 0x03, Operação de multiplicação
        $display("MUL: %h * %h = %h", A, B, ULA_Out);

        // Teste 2: Multiplicação por zero
        apply_test(8'h10, 8'h00, 4'b0010); // A = 0x10, B = 0x00, Operação de multiplicação
        $display("MUL: %h * %h = %h", A, B, ULA_Out);

        // Testes de Divisão
        $display("\n=== Testes de Divisão ===");
        // Teste 1: Divisão exata
        apply_test(8'h20, 8'h04, 4'b0011); // A = 0x20, B = 0x04, Operação de divisão
        $display("DIV: %h / %h = %h", A, B, ULA_Out);

        // Teste 2: Divisão com resto
        apply_test(8'h0F, 8'h04, 4'b0011); // A = 0x0F, B = 0x04, Operação de divisão
        $display("DIV: %h / %h = %h", A, B, ULA_Out);

        // Testes Aleatórios
        $display("\n=== Testes Aleatórios ===");
        repeat(10) begin
                logic [7:0] temp_a, temp_b;
                logic [3:0] op;

                // Gera valores aleatórios para A, B e a operação
                temp_a = $urandom_range(1, 255); // Gera A entre 1 e 255
                temp_b = $urandom_range(1, temp_a); // Garante que B seja menor ou igual a A
                op = $urandom_range(0, 3);          // Gera uma operação aleatória (0 a 3)

                apply_test(temp_a, temp_b, op);     // Aplica os valores gerados

                // Exibe o resultado da operação aleatória
                case(op)
                4'b0000: $display("ADD Aleatório: %h + %h = %h (Carry=%b)", A, B, ULA_Out,
CarryOut);
                4'b0001: $display("SUB Aleatório: %h - %h = %h", A, B, ULA_Out);
                4'b0010: $display("MUL Aleatório: %h * %h = %h", A, B, ULA_Out);
                4'b0011: $display("DIV Aleatório: %h / %h = %h", A, B, ULA_Out);
                endcase
        end

        // Fim dos testes
        $display("\n=== Fim dos Testes ===");
        $display("Testbench concluído em %0t", $time);
        repeat(2) @(posedge clock);         // Aguarda mais 2 ciclos antes de finalizar
        $finish;                            // Finaliza a simulação
        end
```

```
        // Verificações contínuas
        always @(posedge clock) begin
        // Verificações de consistência do comportamento da ALU
        if (!reset) begin
                assert(A >= B) else
                $error("Violação: A deve ser maior ou igual a B (A=%h, B=%h)", A, B);

                assert(ULA_Sel inside {[0:3]}) else
                $error("Operação inválida detectada: %h", ULA_Sel);

                assert(!(ULA_Sel == 4'b0011 && B == 0)) else
                $error("Tentativa de divisão por zero");
        end
        end

endmodule
```

b. testbench self checking

```
//------------------------------------------------------------------------------
// Projeto: ULA Testbench self checking
// Aluna: Jaqueline Ferreira de Brito
// Data: 2025-02-23 20:56:59
// Descrição: Testbench auto verificado com testes determinísticos e aleatórios
//------------------------------------------------------------------------------

`timescale 1ns/1ps // Definindo a unidade de tempo e a precisão para 1ns/1ps

// Importa os pacotes necessários para o testbench
`include "ula_enhanced_pkg.sv"
`include "testbench_utils_pkg.sv"
import ula_enhanced_pkg::*;
import testbench_utils_pkg::*;

// Módulo do testbench com verificação automática
module ULA_self_checking_tb;

        // Declaração de sinais de entrada e saída para o testbench
        logic          clock;         // Sinal de clock
        logic          reset;         // Sinal de reset
        logic [7:0]  A;               // Operando A
        logic [7:0]  B;               // Operando B
        logic [3:0]  ULA_Sel;         // Seleção da operação na ULA
        logic [7:0]  ULA_Out;         // Resultado da ULA
        logic          CarryOut;      // Sinal de carry (transbordo)

        // Instância do gerenciador de testes
        TestManager test_mgr;

        // Contadores de estatísticas de operações
        int num_adds = 0;
        int num_subs = 0;
        int num_muls = 0;
        int num_divs = 0;
```

```
int num_invalids = 0;
int num_errors = 0;
int num_tests = 0;

// Instância dos monitores e verificadores
PerformanceMonitor perf_monitor;
TimingChecker timing_checker;
enhanced_error_stats_t enhanced_errors;

// Definição de constantes
localparam MIN_OPS = 2; // Número mínimo de operações válidas a serem realizadas

// Enumeração das operações disponíveis
typedef enum logic [3:0] {
ADD = 4'b0000, // Operação de adição
SUB = 4'b0001, // Operação de subtração
MUL = 4'b0010, // Operação de multiplicação
DIV = 4'b0011  // Operação de divisão
} op_t;

// Estrutura para armazenar estatísticas de erros
typedef struct {
int invalid_ops;      // Operações inválidas
int timing_violations; // Violações de timing
int overflow_errors;   // Erros de overflow
int a_less_than_b;    // A < B (violação de ordem)
int div_by_zero;      // Tentativa de divisão por zero
int result_mismatch;   // Erro de comparação de resultados
int carry_mismatch;   // Erro no carry
int unknown_values;   // Valores desconhecidos
} error_stats_t;

// Instância de estatísticas de erro
error_stats_t errors;

// Instanciação do DUT (Dispositivo sob teste) ULA
ULA dut (.*);

// Geração do sinal de clock
initial begin
clock = 0;
forever #5 clock = ~clock; // Clock com período de 10 ns (5 ns para cada borda)
end

// Função para converter a operação em string para exibição
function automatic string op_to_string(logic [3:0] op);
case(op)
       ADD: return "ADD";
       SUB: return "SUB";
       MUL: return "MUL";
       DIV: return "DIV";
       default: return "INV"; // Retorna "INV" para operações inválidas
endcase
endfunction
```

```
        // Função para gerar um operando B válido, considerando a operação
        function automatic logic [7:0] gen_valid_b(input logic [7:0] a, input op_t op);
        logic [7:0] b;
        if (op == DIV) begin
                b = ($urandom % a); // Gera um valor de B menor que A para divisão
                return (b == 0) ? 8'h01 : b; // Evita divisão por zero
        end else begin
                return ($urandom % (a + 1)); // Gera um valor de B aleatório
        end
        endfunction

        // Task para verificar o resultado da operação
        task automatic check_result(input logic [7:0] a, input logic [7:0] b, input logic
[3:0] op, input int test_num);
        time start_time;
        string op_str;
        string op_symbol;
        logic [8:0] full_result;
        logic [7:0] expected_result;
        logic expected_carry;

        // Define o símbolo da operação
        case(op)
                ADD: op_symbol = "+";
                SUB: op_symbol = "-";
                MUL: op_symbol = "*";
                DIV: op_symbol = "/";
                default: op_symbol = "?"; // Operação inválida
        endcase

        // Verificação de valores desconhecidos
        if ($isunknown({a, b, op})) begin
                $display("Teste %0d: Valores de entrada desconhecidos", test_num);
                errors.unknown_values++;
                enhanced_errors.unknown_values++;
                return;
        end

        // Cálculo do resultado esperado
        case (op)
                ADD: begin
                full_result = a + b;
                expected_result = full_result[7:0];
                expected_carry = full_result[8]; // Carry ocorre no resultado
                end
                SUB: begin
                expected_result = a - b;
                expected_carry = 0; // Não há carry em subtração
                end
                MUL: begin
                full_result = a * b;
                expected_result = full_result[7:0];
                expected_carry = 0; // Não há carry em multiplicação
                end
                DIV: begin
```

```
            expected_result = a / b;
            expected_carry = 0; // Não há carry em divisão
            end
        endcase

        @(posedge clock);
        A = a; // Atribui valores aos operandos
        B = b;
        ULA_Sel = op; // Define a operação a ser realizada
        start_time = $time;
        op_str = op_to_string(op); // Converte operação para string

        @(posedge clock);

        // Exibe informações detalhadas do teste
        $display("\n=== Teste %0d (%s) ===", test_num, op_str);
        $display("Cálculo: 0x%h %s 0x%h = 0x%h (%s)", a, op_symbol, b, expected_result,
            (expected_carry ? "com carry" : "sem carry"));
        $display("Valores em decimal:");
        $display("A = %0d, B = %0d, Resultado esperado = %0d", a, b, expected_result);
        $display("Tempo de execução: %0t ns", $time - start_time);

        if (CarryOut || expected_carry)
            $display("*** Operação com carry/overflow ***");

        // Verifica se o resultado está correto
        if (ULA_Out !== expected_result) begin
            $display("Status: FALHOU");
            $display("Resultado obtido: 0x%h (%0d) - INCORRETO", ULA_Out, ULA_Out);
            errors.result_mismatch++; // Incrementa o erro
            num_errors++;
        end else begin
            $display("Status: OK");
            $display("Resultado obtido: 0x%h (%0d)", ULA_Out, ULA_Out);
        end

        // Adiciona o resultado ao TestManager
        test_mgr.add_result(
            $sformatf("Teste %0d (%s)", test_num, op_str),
            $sformatf("0x%h %s 0x%h = 0x%h", a, op_symbol, b, expected_result),
            (ULA_Out === expected_result),
            $time - start_time,
            $sformatf("0x%h", expected_result),
            $sformatf("0x%h", ULA_Out),
            (CarryOut || expected_carry) ? "Com carry/overflow" : "Sem carry/overflow"
        );

        // Atualiza as estatísticas de operações realizadas
        case (op)
            ADD: num_adds++;
            SUB: num_subs++;
            MUL: num_muls++;
            DIV: num_divs++;
            default: num_invalids++;
        endcase
```

```
num_tests++; // Incrementa o número total de testes
endtask

// Task para rodar os testes determinísticos
task automatic run_deterministic_tests();
int test_count = 0;
$display("\n=== Testes Determinísticos ===");

// Testes para adição
check_result(8'h30, 8'h20, ADD, test_count++);
check_result(8'hFF, 8'h01, ADD, test_count++);
check_result(8'h80, 8'h80, ADD, test_count++);

// Testes para subtração
check_result(8'h50, 8'h20, SUB, test_count++);
check_result(8'h30, 8'h30, SUB, test_count++);
check_result(8'hFF, 8'h01, SUB, test_count++);

// Testes para multiplicação
check_result(8'h04, 8'h03, MUL, test_count++);
check_result(8'h10, 8'h10, MUL, test_count++);
check_result(8'hFF, 8'h02, MUL, test_count++);

// Testes para divisão
check_result(8'h20, 8'h04, DIV, test_count++);
check_result(8'h0F, 8'h04, DIV, test_count++);
check_result(8'hFF, 8'h02, DIV, test_count++);

num_tests = test_count;
endtask

// Task para rodar os testes aleatórios
task automatic run_random_tests(int num_random_tests);
int remaining_tests;
int test_count;
logic [7:0] temp_a;
logic [7:0] temp_b;
op_t op;

begin
        $display("\n=== Testes Aleatórios ===");
        remaining_tests = num_random_tests;
        test_count = num_tests;

        while (remaining_tests > 0) begin
        // Determina a operação a ser realizada de forma aleatória
        if (num_adds < MIN_OPS) op = ADD;
        else if (num_subs < MIN_OPS) op = SUB;
        else if (num_muls < MIN_OPS) op = MUL;
        else if (num_divs < MIN_OPS) op = DIV;
        else op = op_t'($urandom % 4);

        temp_a = $urandom; // Gera valor aleatório para A
        if (temp_a == 0) temp_a = 8'h01; // Garante que A não seja zero
```

```
            temp_b = gen_valid_b(temp_a, op); // Gera B válido com base em A e operação

            // Chama a task para verificar o resultado
            check_result(temp_a, temp_b, op, test_count++);

            remaining_tests--;
            end

            // Verifica se o número mínimo de operações foi realizado
            assert(num_adds >= MIN_OPS) else
            $error("Número insuficiente de adições: %0d", num_adds);
            assert(num_subs >= MIN_OPS) else
            $error("Número insuficiente de subtrações: %0d", num_subs);
            assert(num_muls >= MIN_OPS) else
            $error("Número insuficiente de multiplicações: %0d", num_muls);
            assert(num_divs >= MIN_OPS) else
            $error("Número insuficiente de divisões: %0d", num_divs);
    end
    endtask

    // Função para exibir relatório de erros
    function void print_error_report();
    $display("\n=== Relatório Final - %s ===", "2025-02-23 20:56:59");
    $display("Usuário: %s", "jaquedebrito");

    $display("\nEstatísticas de Testes:");
    $display("Total de testes: %0d", num_tests);
    $display("Adições: %0d", num_adds);
    $display("Subtrações: %0d", num_subs);
    $display("Multiplicações: %0d", num_muls);
    $display("Divisões: %0d", num_divs);
    $display("Operações Inválidas: %0d", num_invalids);

    $display("\n=== Relatório de Desempenho ===");
    perf_monitor.print_report();

    $display("\n=== Estatísticas de Erro ===");
    $display("Operações inválidas: %0d", errors.invalid_ops);
    $display("Violações de timing: %0d", errors.timing_violations);
    $display("Erros de overflow: %0d", errors.overflow_errors);
    $display("Violações A < B: %0d", errors.a_less_than_b);
    $display("Divisões por zero: %0d", errors.div_by_zero);
    $display("Resultados incorretos: %0d", errors.result_mismatch);
    $display("Carrys incorretos: %0d", errors.carry_mismatch);
    $display("Valores desconhecidos: %0d", errors.unknown_values);
    endfunction

    // Processo principal de teste
    initial begin
    // Formatação do tempo para exibição
    $timeformat(-9, 2, " ns", 20);

    // Inicializações do sistema
    test_mgr = new("2025-02-23 20:56:59", "jaquedebrito");
```

```
perf_monitor = new();
timing_checker = new();
enhanced_errors = '{default: 0};
errors = '{default: 0};

// Cabeçalho do relatório inicial
$display("//----------------------------------------");
$display("// Iniciando testbench ULA with self checking");
$display("// Data: 23/02/2025");
$display("// Aluna: Jaqueline Ferreira de Brito");
$display("//----------------------------------------");

// Reset inicial
reset = 1;
A = 0;
B = 0;
ULA_Sel = 0;

// Espera o clock para liberar o reset
repeat(2) @(posedge clock);
reset = 0;
@(posedge clock);

// Executa os testes determinísticos e aleatórios
run_deterministic_tests();
run_random_tests(10); // Realiza 10 testes aleatórios

// Relatórios finais
print_error_report();
test_mgr.print_report();

$display("\nTestbench concluído em %t", $time);
repeat(2) @(posedge clock);
$finish;
end

// Verificações contínuas durante o clock
always @(posedge clock) begin
if (!reset && !$isunknown({A, B, ULA_Sel})) begin
        // Verifica as condições de entrada
        assert(A >= B) else
        $error("Violação: A deve ser maior ou igual a B (A=%h, B=%h)", A, B);

        assert(ULA_Sel inside {[0:3]}) else
        $error("Operação inválida detectada: %h", ULA_Sel);

        assert(!(ULA_Sel == DIV && B == 0)) else
        $error("Tentativa de divisão por zero");
end
end

// Cobertura funcional
covergroup ULA_cov @(posedge clock);
option.per_instance = 1;
```

```
        // Cobertura de operações
        cp_op: coverpoint ULA_Sel {
                bins ops[] = {[0:3]};
                illegal_bins invalid = {[4:$]};
        }

        // Cobertura de carry
        cp_carry: coverpoint CarryOut iff (ULA_Sel == ADD) {
                bins no_carry = {0};
                bins with_carry = {1};
        }

        // Cobertura de resultado zero
        cp_zero_result: coverpoint ULA_Out {
                bins zero = {0};
                bins non_zero = {[1:$]};
        }

        // Cobertura para divisão válida
        cp_division: coverpoint (ULA_Sel == DIV && B != 0) {
                bins valid_div = {1};
        }

        // Cobertura para A maior que B
        cp_a_gt_b: coverpoint (A > B) {
                bins a_greater = {1};
                bins a_equal = {0};
        }
        endgroup

        ULA_cov cov;

        initial begin
        cov = new(); // Inicializa a cobertura
        end

endmodule
```

c. Testbench self checking with vectors

```
//------------------------------------------------------------------------------
// Projeto: ULA Testbench self checking with Test Vectors
// Data: 23/02/2025
// Aluna: Jaqueline Ferreira de Brito
// Descrição: Testbench auto verificado com vetores e verificações aprimoradas
//------------------------------------------------------------------------------

`timescale 1ns/1ps
`include "ula_enhanced_pkg.sv"  // Inclusão de pacotes personalizados
`include "testbench_utils_pkg.sv"
import ula_enhanced_pkg::*;  // Importação do pacote de verificações aprimoradas
import testbench_utils_pkg::*;  // Importação de pacotes utilitários

module ULA_test_vectors_tb;  // Módulo do testbench para a ULA
```

```
// Sinais do testbench
logic         clock;  // Sinal de clock
logic         reset;  // Sinal de reset
logic [7:0]  A;       // Entradas de 8 bits para o A
logic [7:0]  B;       // Entradas de 8 bits para o B
logic [3:0]  ULA_Sel;  // Seleção de operação da ULA (4 bits)
logic [7:0]  ULA_Out;  // Saída da ULA (resultado)
logic         CarryOut;  // Sinal de carry (se ocorrer durante a operação)

// Contadores para o número de operações realizadas
int num_adds = 0;  // Contador de adições
int num_subs = 0;  // Contador de subtrações
int num_muls = 0;  // Contador de multiplicações
int num_divs = 0;  // Contador de divisões
int num_invalids = 0;  // Contador de operações inválidas
int num_errors = 0;  // Contador de erros gerais
int num_tests = 0;  // Contador de testes realizados

// Monitoramento de timing
logic timing_monitor_enabled = 1;  // Flag de monitoramento de timing
time operation_start_time;  // Tempo de início de operação
time last_operation_time;  // Tempo da última operação
int timing_violations = 0;  // Contador de violações de timing

// Arquivo de vetores de testes
integer file;  // Handle para o arquivo
string line;  // Linha lida do arquivo

// Estruturas para monitoramento e verificação
error_stats_t errors;  // Estrutura para estatísticas de erro
PerformanceMonitor perf_monitor;  // Monitor de desempenho
TimingChecker timing_checker;  // Verificador de timing
enhanced_error_stats_t enhanced_errors;  // Estrutura para erros aprimorados

// Instanciação da ULA (Unidade Lógica Aritmética)
ULA dut (.*);  // Instância do DUT (Device Under Test)

// Geração do clock
initial begin
clock = 0;  // Inicializa o clock
forever #5 clock = ~clock;  // Gera um clock de 5ns
end

// Task para monitoramento de timing detalhado
task automatic monitor_operation_timing(
input time start_time,  // Tempo de início da operação
input string operation,  // Nome da operação
input int test_num  // Número do teste
);
time end_time, delta;
end_time = $time;  // Tempo de término da operação
delta = end_time - start_time;  // Calcula o tempo de execução da operação

// Verifica se há violações de timing
if (timing_monitor_enabled) begin
```

```
        if (delta != 10) begin
        timing_violations++;  // Incrementa as violações de timing
        $error("Violação de timing detectada no teste %0d (%s): %0t ns",
               test_num, operation, delta);  // Exibe erro
        end
end

// Atualiza as estatísticas de desempenho
perf_monitor.update_timing(delta);
last_operation_time = end_time;  // Atualiza o tempo da última operação
endtask

// Função para converter a operação para string
function automatic string op_to_string(logic [3:0] op);
case(op)
        4'b0000: return "ADD";  // Adição
        4'b0001: return "SUB";  // Subtração
        4'b0010: return "MUL";  // Multiplicação
        4'b0011: return "DIV";  // Divisão
        default: return "INV";  // Operação inválida
endcase
endfunction

// Task para verificar o resultado da operação
task automatic enhanced_check_result(
input logic [7:0] a,  // Valor A
input logic [7:0] b,  // Valor B
input logic [3:0] op,  // Operação
input logic [7:0] expected_result,  // Resultado esperado
input logic expected_carry,  // Carry esperado
input int test_num  // Número do teste
);
time start_time;
string op_str;
string op_symbol;

// Determina o símbolo da operação
case(op)
        4'b0000: op_symbol = "+";  // Adição
        4'b0001: op_symbol = "-";  // Subtração
        4'b0010: op_symbol = "*";  // Multiplicação
        4'b0011: op_symbol = "/";  // Divisão
        default: op_symbol = "?";  // Operação desconhecida
endcase

// Verifica se os valores de entrada são desconhecidos
if ($isunknown({a, b, op})) begin
        $error("Teste %0d: Valores de entrada desconhecidos", test_num);
        errors.unknown_values++;  // Incrementa erro de valores desconhecidos
        enhanced_errors.unknown_values++;  // Incrementa erro aprimorado
        return;
end

@(posedge clock);
A = a;  // Atribui o valor de A
```

```
        B = b;  // Atribui o valor de B
        ULA_Sel = op;  // Seleciona a operação
        start_time = $time;  // Registra o tempo de início
        op_str = op_to_string(op);  // Converte operação para string

        @(posedge clock);

        // Exibe os resultados de forma formatada
        $display("\n=== Teste %0d (%s) ===", test_num, op_to_string(op));
        $display("Cálculo: 0x%h %s 0x%h = 0x%h (%s)",
               a, op_symbol, b, expected_result,
               (expected_carry ? "com carry" : "sem carry"));
        $display("Valores em decimal:");
        $display("A = %0d, B = %0d, Resultado esperado = %0d",
               a, b, expected_result);
        $display("Tempo de execução: %0t ns", $time - start_time);

        // Verifica se houve carry na operação
        if (CarryOut || expected_carry)
               $display("*** Operação com carry/overflow ***");

        // Verifica se o resultado está correto
        $display("Status: %s", (ULA_Out === expected_result) ? "OK" : "FALHOU");
        $display("Resultado obtido: 0x%h (%0d)", ULA_Out, ULA_Out);

        // Verifica se o resultado obtido é o esperado
        if (ULA_Out !== expected_result) begin
               errors.result_mismatch++;  // Incrementa erro de mismatch
               num_errors++;  // Incrementa contador de erros
        end

        // Verifica o carry, se aplicável
        case (op)
               4'b0000, 4'b0001, 4'b0010: begin
               if (CarryOut !== expected_carry) begin
                      errors.carry_mismatch++;  // Incrementa erro de carry
                      num_errors++;  // Incrementa contador de erros
               end
               end
        endcase

        // Monitora o tempo da operação
        if (!timing_checker.check_timing($time - start_time))
               enhanced_errors.timing_violations++;  // Incrementa violação de timing

        monitor_operation_timing(start_time, op_str, test_num);  // Monitora o timing
        num_tests++;  // Incrementa o contador de testes
        endtask

        // Task para aplicar os vetores de teste
        task automatic apply_vector(
        input logic [7:0] in_a,  // Valor A
        input logic [7:0] in_b,  // Valor B
        input logic [3:0] op,  // Operação
        input logic [7:0] expected_result,  // Resultado esperado
```

```
        input logic expected_carry,  // Carry esperado
        input int test_num  // Número do teste
        );
        enhanced_check_result(in_a, in_b, op, expected_result, expected_carry, test_num);  //
Chama a task de verificação
        endtask

        // Função para imprimir relatório completo ao final dos testes
        function void print_enhanced_report();
        $display("\n=== Relatório Final - %s ===", "2025-02-23 19:56:07");
        $display("Usuário: jaquedebrito");

        // Exibe as estatísticas de teste
        $display("\nEstatísticas de Testes:");
        $display("Total de testes: %0d", num_tests);
        $display("Adições: %0d", num_adds);
        $display("Subtrações: %0d", num_subs);
        $display("Multiplicações: %0d", num_muls);
        $display("Divisões: %0d", num_divs);
        $display("Operações Inválidas: %0d", num_invalids);

        perf_monitor.print_report();  // Imprime o relatório de desempenho

        // Exibe estatísticas de erro
        $display("\n=== Estatísticas de Erro ===");
        $display("Operações inválidas: %0d", errors.invalid_ops);
        $display("Violações de timing: %0d", errors.timing_violations);
        $display("Erros de overflow: %0d", errors.overflow_errors);
        $display("Violações A < B: %0d", errors.a_less_than_b);
        $display("Divisões por zero: %0d", errors.div_by_zero);
        $display("Resultados incorretos: %0d", errors.result_mismatch);
        $display("Carrys incorretos: %0d", errors.carry_mismatch);
        $display("Valores desconhecidos: %0d", errors.unknown_values);
        $display("Erros de formato de vetor: %0d", errors.vector_format_errors);

        // Exibe erros aprimorados
        $display("\n=== Erros Adicionais ===");
        $display("Violações de setup: %0d", enhanced_errors.setup_violations);
        $display("Violações de hold: %0d", enhanced_errors.hold_violations);
        $display("Erros de estabilidade: %0d", enhanced_errors.stability_errors);
        $display("Erros de reset: %0d", enhanced_errors.reset_errors);
        $display("Violações de desempenho: %0d", enhanced_errors.performance_violations);

        // Exibe estatísticas de timing
        $display("\n=== Estatísticas de Timing ===");
        $display("Total de violações de timing: %0d", timing_violations);
        $display("Última operação completada em: %0t", last_operation_time);
        endfunction

        // Processo principal de teste
        initial begin
        // Inicializações
        $timeformat(-9, 2, " ns", 20);

        // Inicializa monitores
```

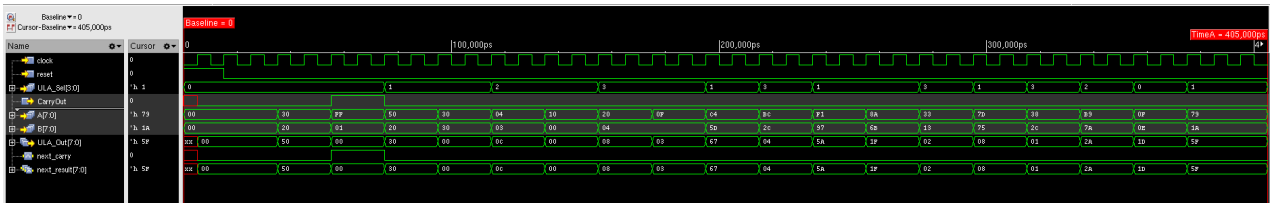
```
        perf_monitor = new();
        timing_checker = new();
        enhanced_errors = '{default: 0};
        errors = '{default: 0};
        operation_start_time = 0;
        last_operation_time = 0;
        timing_violations = 0;

        // Cabeçalho
        $display("//----------------------------------------");
        $display("// Iniciando testbench ULA Test Vectors");
        $display("// Data: 23/02/2025");
        $display("// Aluna: Jaqueline Ferreira de Brito");
        $display("//----------------------------------------");

        // Reset
        reset = 1;
        A = 8'h01;
        B = 8'h01;
        ULA_Sel = 4'b0;

        repeat(2) @(posedge clock);
        reset = 0;
        @(posedge clock);

        // Processa vetores de teste a partir de arquivo
        file = $fopen("ULA_vectors.txt", "r");
        if (file == 0) begin
                $display("Erro: Não foi possível abrir o arquivo ULA_vectors.txt");
                $finish;
        end

        while ($fgets(line, file)) begin
                logic [3:0] op;
                logic [7:0] a, b, expected_result;
                logic expected_carry;
                string op_comment;

                // Remove espaços antes e depois
                while (line.len() > 0 && (line[0] == " " || line[0] == "\t"))
                line = line.substr(1, line.len()-1);

                while (line.len() > 0 && (line[line.len()-1] == " " || line[line.len()-1] ==
"\t" || line[line.len()-1] == "\n"))
                line = line.substr(0, line.len()-2);

                // Pula comentários
                if (line == "" || line.substr(0, 2) == "//")
                continue;

                // Parse da linha
                if ($sscanf(line, "%h %h %h %h %b // %s",
                        op, a, b, expected_result, expected_carry, op_comment) != 6) begin
                if (line.len() > 0) begin
                        $display("Erro no formato do vetor: %s", line);
```

```
                        errors.vector_format_errors++;  // Erro de formato de vetor
                end
                continue;
                end

                apply_vector(a, b, op, expected_result, expected_carry, num_tests);  // Aplica
o vetor de teste

                // Atualiza os contadores por tipo de operação
                case (op)
                4'b0000: num_adds++;
                4'b0001: num_subs++;
                4'b0010: num_muls++;
                4'b0011: num_divs++;
                default: num_invalids++;
                endcase
        end

        $fclose(file);  // Fecha o arquivo de vetores

        // Imprime relatório final
        print_enhanced_report();

        $display("\nTestbench concluído em %t", $time);
        repeat(2) @(posedge clock);
        $finish;
        end

        // Verificações contínuas durante o clock
        always @(posedge clock) begin
        if (!reset && !$isunknown({A, B, ULA_Sel})) begin
                assert(A >= B) else
                $error("Violação: A deve ser maior ou igual a B (A=%h, B=%h)", A, B);

                assert(ULA_Sel inside {[0:3]}) else
                $error("Operação inválida detectada: %h", ULA_Sel);

                assert(!(ULA_Sel == 4'b0011 && B == 0)) else
                $error("Tentativa de divisão por zero");
        end
        end

        // Cobertura funcional
        covergroup ULA_cov @(posedge clock);
        option.per_instance = 1;

        cp_op: coverpoint ULA_Sel {
                bins ops[] = {[0:3]};  // Cobertura para operações
                illegal_bins invalid = {[4:$]};  // Operações inválidas
        }

        cp_carry: coverpoint CarryOut iff (ULA_Sel == 4'b0000) {
                bins no_carry = {0};  // Cobertura para sem carry
                bins with_carry = {1};  // Cobertura para com carry
        }
```

```
        cp_zero_result: coverpoint ULA_Out {
                bins zero = {0};  // Resultado zero
                bins non_zero = {[1:$]};  // Resultado não zero
        }

        cp_division: coverpoint (ULA_Sel == 4'b0011 && B != 0) {
                bins valid_div = {1};  // Cobertura para divisão válida
        }

        cp_a_gt_b: coverpoint (A > B) {
                bins a_greater = {1};  // A > B
                bins a_equal = {0};  // A == B
        }
        endgroup

        ULA_cov cov;

        initial begin
        cov = new();  // Inicializa o grupo de cobertura
        end

endmodule
```

B. Resultados dos testbenchs

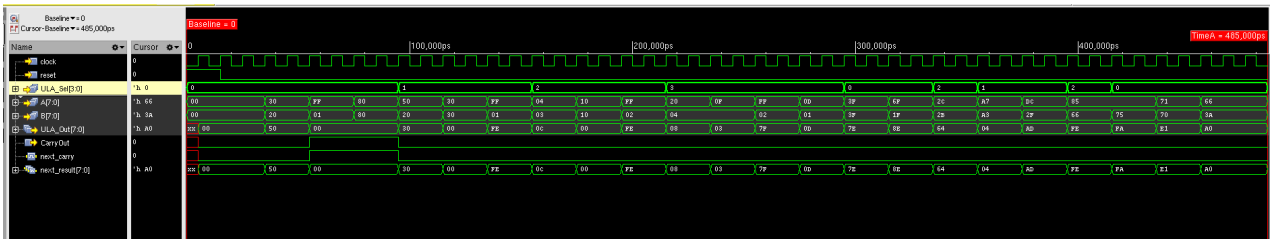
a. Testbench simples:



```
//----------------------------------------
// Iniciando testbench ULA Simples
// Data: 23/02/2025
// Aluna: Jaqueline Ferreira de Brito
//----------------------------------------


=== Testes de Adição ===
ADD: 30 + 20 = 50 (Carry=0)
ADD: ff + 01 = 00 (Carry=1)

=== Testes de Subtração ===
SUB: 50 - 20 = 30
SUB: 30 - 30 = 00

=== Testes de Multiplicação ===
```

```
MUL: 04 * 03 = 0c
MUL: 10 * 00 = 00

=== Testes de Divisão ===
DIV: 20 / 04 = 08
DIV: 0f / 04 = 03

=== Testes Aleatórios ===
SUB Aleatório: c4 - 5d = 67
DIV Aleatório: bc / 2c = 04
SUB Aleatório: f1 - 97 = 5a
SUB Aleatório: 8a - 6b = 1f
DIV Aleatório: 33 / 13 = 02
SUB Aleatório: 7d - 75 = 08
DIV Aleatório: 38 / 2c = 01
MUL Aleatório: b9 * 7a = 2a
ADD Aleatório: 0f + 0e = 1d (Carry=0)
SUB Aleatório: 79 - 1a = 5f

=== Fim dos Testes ===
Testbench concluído em 386000
Simulation complete via $finish(1) at time 405 NS + 0
```

b. Resultado testbench self checking

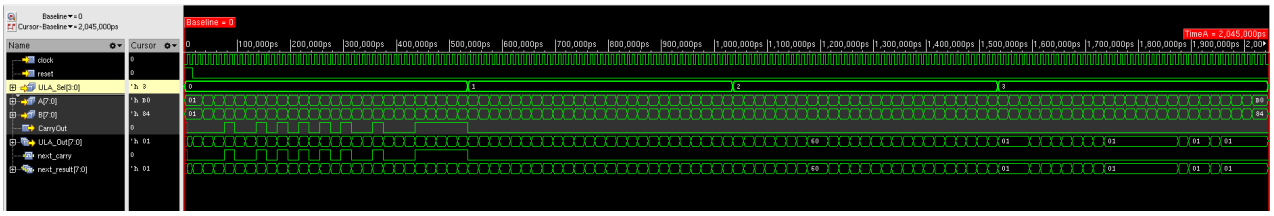



```
xcelium> run
//---------------------------------------
// Iniciando testbench ULA with self checking
// Data: 23/02/2025
// Aluna: Jaqueline Ferreira de Brito
//---------------------------------------

=== Testes Determinísticos ===

=== Teste 2 (ADD) ===
Cálculo: 0x80 + 0x80 = 0x00 (com carry)
Valores em decimal:
A = 128, B = 128, Resultado esperado = 0
Tempo de execução: 10.00 ns ns
*** Operação com carry/overflow ***
Status: OK
Resultado obtido: 0x00 (0)

=== Teste 5 (SUB) ===
Cálculo: 0xff - 0x01 = 0xfe (sem carry)
```

```
Valores em decimal:
A = 255, B = 1, Resultado esperado = 254
Tempo de execução: 10.00 ns ns
Status: OK
Resultado obtido: 0xfe (254)

=== Teste 8 (MUL) ===
Cálculo: 0xff * 0x02 = 0xfe (sem carry)
Valores em decimal:
A = 255, B = 2, Resultado esperado = 254
Tempo de execução: 10.00 ns ns
Status: OK
Resultado obtido: 0xfe (254)

=== Relatório Final - 2025-02-23 20:56:59 ===
Usuário: jaquedebrito

Estatísticas de Testes:
Total de testes: 22
Adições: 8
Subtrações: 5
Multiplicações: 5
Divisões: 4
Operações Inválidas: 0

=== Relatório de Desempenho ===

=== Relatório de Desempenho ===
Tempo mínimo de operação: -1.00 ns
Tempo máximo de operação: 0.00 ns
Tempo médio de operação: 0.00 ns
Total de operações: 0

=== Estatísticas de Erro ===
Operações inválidas: 0
Violações de timing: 0
Erros de overflow: 0
Violações A < B: 0
Divisões por zero: 0
Resultados incorretos: 0
Carrys incorretos: 0
Valores desconhecidos: 0

=== Relatório de Testes ===
Data/Hora: 2025-02-23 20:56:59
Usuário: jaquedebrito
Total de testes: 22
Testes passou: 22
Testes falhou: 0

Resultados detalhados:

Teste 0: Teste 0 (ADD)
Cálculo: 0x30 + 0x20 = 0x50
Status: PASSOU
```

```
Tempo: 10.00 ns
Notas: Sem carry/overflow

Teste 1: Teste 1 (ADD)
Cálculo: 0xff + 0x01 = 0x00
Status: PASSOU
Tempo: 10.00 ns
Notas: Com carry/overflow

Teste 4: Teste 4 (SUB)
Cálculo: 0x30 - 0x30 = 0x00
Status: PASSOU
Tempo: 10.00 ns
Notas: Sem carry/overflow

Teste 5: Teste 5 (SUB)
Cálculo: 0xff - 0x01 = 0xfe
Status: PASSOU
Tempo: 10.00 ns
Notas: Sem carry/overflow

Teste 8: Teste 8 (MUL)
Cálculo: 0xff * 0x02 = 0xfe
Status: PASSOU
Tempo: 10.00 ns
Notas: Sem carry/overflow
```

c. Testbench self checking with vectors




```
=== Teste 96 (DIV) ===
Cálculo: 0x09 / 0x08 = 0x01 (sem carry)
Valores em decimal:
A = 9, B = 8, Resultado esperado = 1
Tempo de execução: 10.00 ns ns
Status:        OK
Resultado obtido: 0x01 (1)

=== Teste 97 (DIV) ===
Cálculo: 0xbc / 0xad = 0x01 (sem carry)
Valores em decimal:
A = 188, B = 173, Resultado esperado = 1
Tempo de execução: 10.00 ns ns
Status:        OK
Resultado obtido: 0x01 (1)

=== Teste 98 (DIV) ===
```

```
Cálculo: 0xd4 / 0xd3 = 0x01 (sem carry)
Valores em decimal:
A = 212, B = 211, Resultado esperado = 1
Tempo de execução: 10.00 ns ns
Status:        OK
Resultado obtido: 0x01 (1)

=== Teste 99 (DIV) ===
Cálculo: 0xb0 / 0x84 = 0x01 (sem carry)
Valores em decimal:
A = 176, B = 132, Resultado esperado = 1
Tempo de execução: 10.00 ns ns
Status:        OK
Resultado obtido: 0x01 (1)

=== Relatório Final - 2025-02-23 19:56:07 ===
Usuário: jaquedebrito

Estatísticas de Testes:
Total de testes: 100
Adições: 25
Subtrações: 25
Multiplicações: 25
Divisões: 25
Operações Inválidas: 0

=== Relatório de Desempenho ===
Tempo mínimo de operação: 10.00 ns
Tempo máximo de operação: 10.00 ns
Tempo médio de operação: 10.00 ns
Total de operações: 100

=== Estatísticas de Erro ===
Operações inválidas: 0
Violações de timing: 0
Erros de overflow: 0
Violações A < B: 0
Divisões por zero: 0
Resultados incorretos: 0
Carrys incorretos: 0
Valores desconhecidos: 0
Erros de formato de vetor: 0

=== Erros Adicionais ===
Violações de setup: 0
Violações de hold: 0
Erros de estabilidade: 0
Erros de reset: 0
Violações de desempenho: 0

=== Estatísticas de Timing ===
Total de violações de timing: 0
Última operação completada em: 2025.00 ns

Testbench concluído em       2025.00 ns
```

```
Simulation complete via $finish(1) at time 2045 NS + 0
```

C. Análise dos resultados:

a. testbench simples

Este testbench combina testes fixos e aleatórios:

- 8 testes determinísticos (2 para cada operação)
- 10 testes aleatórios
- Assertions básicas
- Saída em formato simples e direto
- Sem pacotes adicionais
- Total: 18 testes

b. testbench self checking

Este testbench combina testes fixos e aleatórios:

- 12 testes determinísticos
- 10 testes aleatórios
- Usa TestManager
- Relatórios detalhados
- Timing checks
- Total: 22 testes

c. Testbench self checking with vectors

Este testbench lê um arquivo .txt que contém 100 vetores fixos

- Usa vetores predefinidos
- 100 testes (25 de cada operação)
- Usa TestManager
- Relatórios detalhados
- Timing checks
- Total: 100 testes

3.2 SUBPROJETO 2

a. Explicação dos módulos das classes UVM solicitado:

1. Módulo top (tb.sv)

O módulo **top** é o principal ponto de entrada para a simulação, servindo como testbench para o Design Under Test (DUT), que neste caso é a unidade aritmética e lógica (ULA). Ele

configura a simulação, instância o DUT e gerencia a execução de testes com a ajuda do UVM.

- **Inclusão de arquivos:**
  - Os `include estão sendo usados para incluir arquivos SystemVerilog em um módulo top.
  - Cada arquivo incluído é uma referência a um componente ou bloco de código que será utilizado no teste ou simulação UVM (Universal Verification Methodology).
  - Esses arquivos contêm definições de classes, módulos e componentes necessários para executar um teste completo de verificação de um design.

```
//---------------------------------------------------------
//Include Files
//---------------------------------------------------------
`include "interface.sv"
`include "sequence_item.sv"
`include "sequence.sv"
`include "sequencer.sv"
`include "driver.sv"
`include "monitor.sv"
`include "agent.sv"
`include "scoreboard.sv"
`include "env.sv"
`include "test.sv"
```

- **Importação do pacote UVM:**
  - ***import uvm_pkg::*;*** Importa o pacote UVM (Universal Verification Methodology) e todos os seus componentes.
  - O * indica que todas as classes, funções e outros elementos definidos no pacote UVM podem ser usados diretamente no código.
  - O pacote contém as classes principais e utilitários do UVM, como ***uvm_sequence***, ***uvm_driver***, ***uvm_monitor***, entre outros.
  - Ao importar esse pacote, é dado acesso a todas as funcionalidades de verificação UVM, permitindo a criação de testes estruturados e modulares.
  - ***"uvm_macros.svh":*** `include é uma diretiva de pré-processamento que inclui o conteúdo do arquivo especificado no código. O arquivo ***uvm_macros.svh*** contém macros úteis do UVM, como ***uvm_info***, ***uvm_error***, ***uvm_component_utils***, entre outras.
  - Essas macros ajudam a simplificar o código, fornecendo atalhos para certas funcionalidades do UVM, como registrar componentes, gerar mensagens de log, e outras ações de verificação.

```
`timescale 1ns/1ns

import uvm_pkg::*;
`include "uvm_macros.svh"
```

- **Instanciação de Componentes:**
  - A interface ***alu_interface*** é instanciada, conectando a ULA ao sistema de simulação.
  - O ***alu dut*** é configurado, conectando seus sinais de entrada e saída à interface.

```
module top;

  //-------------------------------------------------------
  //Instantiation
  //-------------------------------------------------------

  logic clock;

  alu_interface intf(.clock(clock));

  alu dut(
       .clock(intf.clock),
       .reset(intf.reset),
       .A(intf.a),
       .B(intf.b),
       .ALU_Sel(intf.op_code),
       .ALU_Out(intf.result),
       .CarryOut(intf.carry_out)
  );
```

- **Configuração de banco de dados UVM (*uvm_config_db*):**
  - A interface ***alu_interface*** é registrada no banco de dados UVM (***uvm_config_db***), permitindo que os componentes UVM acessem a interface.

```
//-------------------------------------------------------
//Interface Setting
//-------------------------------------------------------
  initial begin
       uvm_config_db #(virtual alu_interface)::set(null, "*", "vif", intf );
  end
```

- **Execução de teste:**
  - Inicia a execução do teste chamando ***run_test("alu_test")***.

```
//-------------------------------------------------------
//Start The Test
//-------------------------------------------------------
 initial begin
      run_test("alu_test");
 end
```

- **Geração de Clock:**
  - Espera 5 unidades de tempo após a inicialização do clock antes de executar a primeira instrução.
  - Gera um clock que espera 2 unidades de tempo antes de executar a próxima instrução, isso quer dizer que cria um período de 4 unidades de tempo (2 para a transição de 0 para 1 e 2 para a transição de 1 para 0), fazendo com que o sinal clock oscile a cada 2 unidades de tempo.

```
//-------------------------------------------------------
//Clock Generation
//-------------------------------------------------------
  initial begin
       clock = 0;
       #5;
       forever begin
       clock = ~clock;
       #2;
       end
  end
```

- **Limitação de tempo:**
  - Limita a simulação a 5000 unidades de tempo.

```
 //-------------------------------------------------------
 //Maximum Simulation Time
 //-------------------------------------------------------
  initial begin
       #5000;
       $display("Sorry! Ran out of clock cycles!");
       $finish();
  end
Gera os arquivos de waveform (.vcd).
```

- **Gera os arquivos de waveform (.vcd)**

```
//-------------------------------------------------------
//Generate Waveforms
//-------------------------------------------------------
  initial begin
```

```
        $dumpfile("d.vcd");
        $dumpvars();
    end
```

2. interface.sv

A interface ***alu_interface*** define os sinais de comunicação entre o testbench e o DUT.

- Define os sinais de entrada e saída da ULA.
    - **Sinais:**
        - reset: Sinal para reiniciar a ULA.
        - a, b: Operandos de 8 bits da ULA.
        - op_code: Código de operação (4 bits).
        - result: Resultado da operação (8 bits).
        - carry_out: Flag de carry.

A interface facilita a conexão entre o DUT e o testbench, garantindo que os sinais sejam adequadamente compartilhados.

```
interface alu_interface(input logic clock);

  logic reset;

  logic [7:0] a, b;
  logic [3:0] op_code;
  logic [7:0] result;
  bit carry_out;

endinterface: alu_interface
```

3. test.sv

A classe alu_test organiza e controla as fases de verificação utilizando o UVM.

- Fases do teste:
    - Construtor: A classe **alu_test** herda de ***uvm_test.***
    - O macro uvm_component_utils(*alu_test*) é utilizado para registrar a classe no UVM, o que permite que o UVM a reconheça e utilize em sua execução.

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

- Cria uma instância do ambiente de simulação (***alu_env***):

- Registrar as sequências de reset e de teste.
- A variável do tipo *alu_env*, é o componente que configura o ambiente de simulação.
- As variáveis **reset_seq** e **test_seq** armazenam sequências de testes, que são usadas para executar e controlar as interações com o DUT:
- **alu_base_sequence** → responsável pelo reset da ULA.
- ***alu_test_sequence*** → gera operações da ULA.

```
alu_env env;
alu_base_sequence reset_seq;
alu_test_sequence test_seq;
```

- O construtor ***(new)*** inicia a classe e chama o construtor da classe base ***(super.new(name, parent))***. O macro ***uvm_info*** exibe uma mensagem no log, o que ajuda na depuração e no rastreamento da execução.

```
function new(string name = "alu_test", uvm_component parent);
      super.new(name, parent);
      `uvm_info("TEST_CLASS", "Inside Constructor!", UVM_HIGH)
 endfunction: new
```

- **Fase de construção:**
  - Durante a fase de construção ***(build_phase)***, o método *build_phase* da classe base é chamado, e a variável *env* é instanciada utilizando o método *type_id::create*, que cria um novo objeto do tipo *alu_env*.
  - ***build_phase*** instância os componentes do ambiente de simulação, como o agente de comunicação (***alu_agent***) e o placar de pontuação (***alu_scoreboard***).

```
 function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      `uvm_info("TEST_CLASS", "Build Phase!", UVM_HIGH)

      env = alu_env::type_id::create("env", this);
 endfunction: build_phase
```

- **Fase de conexão *(connect_phase)*:**
  - É onde os sinais e as conexões entre os componentes do ambiente são estabelecidas.

```
function void connect_phase(uvm_phase phase);
```

```
      super.connect_phase(phase);
      `uvm_info("TEST_CLASS", "Connect Phase!", UVM_HIGH)
  endfunction: connect_phase
```

- **Fase de execução *(run_phase)*:**
  - É o núcleo do teste, onde as ações são realizadas. Durante essa fase, as sequências de reset e de teste são executadas.
  - A execução do ***reset_seq*** inicializa o DUT, enquanto o ***test_seq*** realiza operações repetidas, garantindo que a ULA seja testada sob várias condições.

    Objeção no UVM:

  - As objeções ***(phase.raise_objection() e phase.drop_objection())*** indicam que o teste está em execução, evitando que a simulação termine prematuramente.
  - **phase.raise_objection(this)**: A objeção é levantada, indicando que o teste está em execução e deve ser mantido até que a objeção seja removida.
  - **Execução do reset_seq**: A sequência de reset (***reset_seq***) é instanciada e iniciada usando **start()**, que provavelmente a envia para o sequenciador do agente ***(env.agnt.seqr)***. Após isso, há uma pequena pausa com ***#10***; (delay de 10 unidades de tempo).
  - **Execução do test_seq**: Um loop é usado para repetir a execução de ***test_seq*** 100 vezes. Cada instância de ***test_seq*** é criada e iniciada de maneira semelhante ao ***reset_seq***.
  - ***phase.drop_objection(this)***: Após a execução dos testes, a objeção é removida, sinalizando que o teste terminou e que a execução pode continuar.

```
//-------------------------------------------------------
//Run Phase
//-------------------------------------------------------
  task run_phase (uvm_phase phase);
      super.run_phase(phase);
      `uvm_info("TEST_CLASS", "Run Phase!", UVM_HIGH)

      phase.raise_objection(this);

      //reset_seq
      reset_seq = alu_base_sequence::type_id::create("reset_seq");
      reset_seq.start(env.agnt.seqr);
      #10;

      repeat(100) begin
```

```
        //test_seq
        test_seq = alu_test_sequence::type_id::create("test_seq");
        test_seq.start(env.agnt.seqr);
        #10;
        end

        phase.drop_objection(this);

  endtask: run_phase


endclass: alu_test
```

4. env.sv

A classe ***alu_env*** configura o ambiente de simulação para o DUT ALU, criando e conectando os componentes necessários para a verificação (como o agente ***alu_agent*** e o placar de pontuação ***alu_scoreboard***). Ela também executa as fases básicas do ciclo de vida UVM, como a construção e a conexão dos componentes. As fases de execução estão configuradas, mas a lógica de execução em si está contida no agente e no placar de pontuação.

- **Componentes do ambiente:**
  - A classe ***alu_env***, que é parte do ambiente de simulação UVM, é derivada da classe ***uvm_env*** e registrada no UVM através do macro ***uvm_component_utils(alu_env)***.
  - Ela contém duas variáveis:
    - agnt (Agente), do tipo ***alu_agent***, interage diretamente com o DUT e realiza as operações de verificações.
    - scb (Placar de Conexão), do tipo ***alu_scoreboard***, verifica e valida os resultados gerados pelo DUT, os resultados da simulação.

```
class alu_env extends uvm_env;
  `uvm_component_utils(alu_env)

  alu_agent agnt;
  alu_scoreboard scb;
```

- **Construtor:**
  - O construtor ***(new)*** é responsável por inicializar a classe ***alu_env***, *inicializa os componentes agnt e scb*, e invoca o construtor da classe base ***(super.new(name, parent))***.
  - Uma mensagem de log é gerada através do macro ***uvm_info***, indicando que o construtor foi chamado.

```
function new(string name = "alu_env", uvm_component parent);
      super.new(name, parent);
      `uvm_info("ENV_CLASS", "Inside Constructor!", UVM_HIGH)
 endfunction: new
```

- **Fase de Construção (*build_phase*) cria instâncias dos componentes do ambiente:**
    - Durante a fase de construção, a função ***build_phase*** da classe base é invocada, resultando na criação de instâncias dos componentes ***agnt*** e ***scb*** através do método ***type_id::create***.
    - Consequentemente, o ambiente é configurado com o agente de comunicação com o DUT (***agnt***) e o placar de pontuação responsável pela validação dos resultados da simulação (***scb***).

```
 function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      `uvm_info("ENV_CLASS", "Build Phase!", UVM_HIGH)

      agnt = alu_agent::type_id::create("agnt", this);
      scb = alu_scoreboard::type_id::create("scb", this);
 endfunction: build_phase
```

- Fase de Conexão **(*connect_phase*):** Estabelece a comunicação entre o monitor do agente e o placar de pontuação.
    - Na fase de conexão, a porta de monitoramento do agente ***(agnt.mon.monitor_port)*** é ligada à porta de placar do scb (***scoreboard_port***).
    - Essa ligação permite que o placar receba e confira os sinais que o agente monitora.

```
function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      `uvm_info("ENV_CLASS", "Connect Phase!", UVM_HIGH)

      agnt.mon.monitor_port.connect(scb.scoreboard_port);
 endfunction: connect_phase
```

- A fase de execução (run_phase) é onde a verificação efetivamente acontece.
    - A função run_phase da classe base (***super.run_phase(phase)***) é chamada.

```
task run_phase (uvm_phase phase);
      super.run_phase(phase);
 endtask: run_phase
```

5. agent.sv

A classe alu_agent define um agente UVM para interagir com o DUT ALU. Ela é composta por três principais componentes: o driver (***drv***), o monitor (***mon***) e o sequenciador (***seqr***), que são instanciados e conectados durante as fases de construção e conexão do ciclo de vida UVM. O objetivo deste agente é controlar o fluxo de dados e interações com o DUT durante a execução dos testes.

- **Herança:**
  - A classe alu_agent, que é derivada da classe uvm_agent, é um tipo de componente UVM que facilita a interação com o DUT e a execução de sequências de teste. Para registrar essa classe no sistema UVM e torná-la reconhecível durante a simulação, utiliza-se o macro ***uvm_component_utils(alu_agent)***.
  - Além disso, a classe declara três variáveis:
    - **drv**: um componente do tipo ***alu_driver***, responsável por enviar sinais para o DUT.
    - **mon**: um componente do tipo ***alu_monitor***, que monitora os sinais do DUT e captura dados para análise posterior.
    - **seqr**: um componente do tipo ***alu_sequencer***, que gerencia as sequências de itens de teste.

```
class alu_agent extends uvm_agent;
  `uvm_component_utils(alu_agent)

  alu_driver drv;
  alu_monitor mon;
  alu_sequencer seqr;
```

- O construtor (***new***) inicializa o objeto ***alu_agent***.
  - Primeiramente, ele invoca o construtor da classe base ***super.new(name, parent)***.
  - Utiliza o macro ***uvm_info*** para registrar uma mensagem de log indicando a execução do construtor.

```
  function new(string name = "alu_agent", uvm_component parent);
       super.new(name, parent);
       `uvm_info("AGENT_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new
```

- **Fase de construção (*build_phase*):**
  - O método ***type_id::create*** é utilizado para instanciar os componentes ***drv, mon*** e ***seqr***. Esse método cria as instâncias ***alu_driver***, ***alu_monitor*** e ***alu_sequencer***, que são essenciais para o funcionamento do agente.

- A função chama ***super.build_phase(phase)*** da classe base para executar as operações necessárias nessa fase de construção.

```
function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      `uvm_info("AGENT_CLASS", "Build Phase!", UVM_HIGH)

      drv = alu_driver::type_id::create("drv", this);
      mon = alu_monitor::type_id::create("mon", this);
      seqr = alu_sequencer::type_id::create("seqr", this);
endfunction: build_phase
```

- **Fase de conexão (*connect_phase*):**
  - O driver (***drv***) é conectado ao sequenciador (***seqr***), permitindo a troca de itens de sequência entre eles. A função ***drv.seq_item_port.connect(seqr.seq_item_export)*** estabelece essa conexão.

```
function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      `uvm_info("AGENT_CLASS", "Connect Phase!", UVM_HIGH)

      drv.seq_item_port.connect(seqr.seq_item_export);
endfunction: connect_phase
```

- **A fase de execução *(run_phase)*:**
  - É responsável por iniciar as operações do agente.
  - A função ***super.run_phase(phase)*** é chamada para garantir que as operações da classe base sejam executadas. Mas neste código específico, não há lógica adicional implementada.

```
task run_phase (uvm_phase phase);
      super.run_phase(phase);
endtask: run_phase
```

6. driver.sv

A classe ***alu_driver*** é um driver UVM responsável por controlar a interação com o DUT. Ela busca e executa itens de sequência, aplicando os sinais apropriados ao DUT, como valores de entrada e códigos de operação. O driver usa a interface ***alu_interface*** para se comunicar com o DUT e aplica os valores do ***alu_sequence_item*** para simular operações de teste.

- **Herança:**
  - A classe ***alu_driver*** herda de ***uvm_driver#(alu_sequence_item)***, o que a torna um driver especializado para trabalhar com itens do tipo ***alu_sequence_item***.

- O registro da classe no sistema UVM é realizado através do macro ***uvm_component_utils(alu_driver)***, permitindo sua utilização durante o ciclo de vida da simulação.
- A classe também contém a variável virtual ***vif***, do tipo virtual ***alu_interface***, que possibilita a conexão com o DUT, e a variável item, do tipo ***alu_sequence_item***, que armazena os dados da sequência a ser transmitida ao DUT.

```
class alu_driver extends uvm_driver#(alu_sequence_item);
  `uvm_component_utils(alu_driver)

  virtual alu_interface vif;
  alu_sequence_item item;
```

- A classe ***alu_driver*** é inicializada pelo construtor (***new***), que invoca o construtor da classe base ***super.new(name, parent)*** e utiliza o macro ***uvm_info*** para exibir uma mensagem informativa, sinalizando a execução do construtor.

```
 function new(string name = "alu_driver", uvm_component parent);
       super.new(name, parent);
       `uvm_info("DRIVER_CLASS", "Inside Constructor!", UVM_HIGH)
 endfunction: new
```

- **Fase de construção (*build_phase*):**
  - A função ***build_phase*** da classe base é chamada para iniciar as operações.
  - O código busca acessar a interface ***alu_interface*** através do banco de dados de configuração UVM (***uvm_config_db***).
  - Caso não consiga obter a interface, um erro será gerado utilizando ***uvm_error***.
  - A interface ***vif*** será então utilizada para estabelecer a conexão com o DUT.

```
 function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       `uvm_info("DRIVER_CLASS", "Build Phase!", UVM_HIGH)

       if(!(uvm_config_db #(virtual alu_interface)::get(this, "*", "vif", vif))) begin
       `uvm_error("DRIVER_CLASS", "Failed to get VIF from config DB!")
       end
 endfunction: build_phase
```

- **Fase de conexão (*connect_phase*):**
  - É chamada para fazer as conexões necessárias entre os componentes UVM.
  - A função ***super.connect_phase(phase)*** da classe base é chamada.

```
 function void connect_phase(uvm_phase phase);
       super.connect_phase(phase);
       `uvm_info("DRIVER_CLASS", "Connect Phase!", UVM_HIGH)
 endfunction: connect_phase
```

- Fase de execução (***run_phase***):

- O driver entra em um loop infinito (***forever***) onde ele cria um novo ***alu_sequence_item***, obtém o próximo item da sequência com ***seq_item_port.get_next_item(item)***, aplica os sinais ao DUT com o método drive, e sinaliza que o item foi processado com ***seq_item_port.item_done()***.

```
task run_phase (uvm_phase phase);
      super.run_phase(phase);
      `uvm_info("DRIVER_CLASS", "Inside Run Phase!", UVM_HIGH)

      forever begin
      item = alu_sequence_item::type_id::create("item");
      seq_item_port.get_next_item(item);
      drive(item);
      seq_item_port.item_done();
      end
endtask: run_phase
```

- O método drive, que aplica os valores do item de sequência ao DUT, aguarda a borda positiva do clock ***(@(posedge vif.clock))*** para então aplicar os valores do item às variáveis da interface ***vif***.
    - Assim, o valor de reset do item é aplicado a ***vif.reset***, e os valores de ***a, b*** e ***op_code*** do item são aplicados às suas respectivas variáveis da interface.

```
task drive(alu_sequence_item item);
      @(posedge vif.clock);
      vif.reset <= item.reset;
      vif.a <= item.a;
      vif.b <= item.b;
      vif.op_code <= item.op_code;
endtask: drive
```

7. monitor.sv

A classe alu_monitor é responsável por monitorar e coletar os sinais de entrada (operandos e código de operação) e saída (resultado) do DUT em cada ciclo de clock. Esses dados são então enviados para o scoreboard, que valida os resultados. A classe alu_monitor opera dentro do ciclo de vida UVM e aguarda a desativação do sinal de reset antes de iniciar a amostragem e monitoramento das entradas e saídas.

- **Herança:**
    - A classe ***alu_monitor*** herda de ***uvm_monitor***, que é usada para monitorar o DUT.
    - O macro ***uvm_component_utils(alu_monitor)*** registra a classe no sistema UVM.
    - A variável ***vif*** é uma interface virtual do tipo ***alu_interface***, que conecta o monitor ao DUT e possibilita a amostragem dos sinais.

- A variável ***item*** é do tipo ***alu_sequence_item***, que armazena os dados amostrados durante a simulação.
- A variável ***monitor_port*** é uma porta de análise (***uvm_analysis_port***) do tipo ***alu_sequence_item***, usada para enviar os itens amostrados para um scoreboard ou outro componente de verificação.

```
class alu_monitor extends uvm_monitor;
  `uvm_component_utils(alu_monitor)

  virtual alu_interface vif;
  alu_sequence_item item;

  uvm_analysis_port #(alu_sequence_item) monitor_port;
```

- O construtor (***new***) inicia a classe ***alu_monitor*** chamando o construtor da classe base ***super.new(name, parent)***.
  - Para gerar uma mensagem informativa no log de simulação, utiliza-se o macro ***uvm_info***.

```
function new(string name = "alu_monitor", uvm_component parent);
       super.new(name, parent);
       `uvm_info("MONITOR_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new
```

- Fase de construção (***build_phase***):
  - A função ***super.build_phase(phase)*** é chamada para executar as operações de construção da classe base.
  - Em seguida, é criada uma instância de ***monitor_port***, que será usada para enviar os itens amostrados.
  - A interface ***vif*** é então recuperada do banco de dados de configuração UVM (***uvm_config_db***).
  - Se a interface ***vif*** não for encontrada, um erro será gerado com a macro ***uvm_error***.

```
  function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       `uvm_info("MONITOR_CLASS", "Build Phase!", UVM_HIGH)

       monitor_port = new("monitor_port", this);

       if(!(uvm_config_db #(virtual alu_interface)::get(this, "*", "vif", vif))) begin
       `uvm_error("MONITOR_CLASS", "Failed to get VIF from config DB!")
       end
  endfunction: build_phase
```

- A função "***connect_phase***" é chamada, porém não estabelece conexões específicas no

código apresentado. Tipicamente, essa fase seria responsável por conectar componentes entre si ou integrar a interface com outros módulos.

```
function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      `uvm_info("MONITOR_CLASS", "Connect Phase!", UVM_HIGH)
 endfunction: connect_phase
```

- **A fase de execução (*run_phase*):**
  - É a parte central do monitor, responsável por mostrar os sinais.
  - Ele opera em um loop infinito (***forever***) para monitorar o DUT continuamente.
  - O monitor aguarda a desativação do sinal de reset ***(wait(!vif.reset))***.
  - Uma vez que o reset é desativado, na borda de subida do clock, o monitor amostra as entradas (***a, b, op_code***), armazenando os valores no item, e a saída (***result***).
  - O item amostrado é então enviado através de ***monitor_port.write(item)*** para um scoreboard ou outro componente que realize a verificação.

```
task run_phase (uvm_phase phase);
      super.run_phase(phase);
      `uvm_info("MONITOR_CLASS", "Inside Run Phase!", UVM_HIGH)

      forever begin
      item = alu_sequence_item::type_id::create("item");

      wait(!vif.reset);

      //sample inputs
      @(posedge vif.clock);
      item.a = vif.a;
      item.b = vif.b;
      item.op_code = vif.op_code;

      //sample output
      @(posedge vif.clock);
      item.result = vif.result;

      // send item to scoreboard
      monitor_port.write(item);
      end
 endtask: run_phase
```

8. sequencer.sv

A classe ***alu_sequencer*** é um sequenciador simples dentro do UVM, responsável por gerar e enviar itens de sequência do tipo ***alu_sequence_item***. Durante as fases de construção e conexão, o sequenciador se prepara para iniciar a execução de sequências, embora a execução real das sequências não esteja explicitada no código fornecido. O código configura o sequenciador para

trabalhar dentro do ciclo de vida UVM, com as fases de construção e conexão configuradas para enviar informações de log. Em resumo, a classe ***alu_sequencer*** é essencialmente um componente que prepara o ambiente para gerar e sequenciar as operações no DUT.

- **Herança**:
  - A classe ***alu_sequencer*** herda de ***uvm_sequencer#(alu_sequence_item)***, o que significa que ela é um sequenciador de itens do tipo ***alu_sequence_item***.
  - Esses itens representam a sequência de operações (como entradas e comandos) que o sequenciador irá gerar e enviar para o driver.
  - O macro ***uvm_component_utils(alu_sequencer)*** registra a classe no sistema UVM, permitindo que ela seja utilizada como um componente UVM.

```
class alu_sequencer extends uvm_sequencer#(alu_sequence_item);
  `uvm_component_utils(alu_sequencer)
```

- O construtor (***new***) chama o construtor da classe base ***super.new(name, parent)*** para inicializar o sequenciador.
  - O macro ***uvm_info*** é utilizado para gerar uma mensagem informativa no log da simulação para indicar que o construtor foi executado.

```
function new(string name = "alu_sequencer", uvm_component parent);
       super.new(name, parent);
       `uvm_info("SEQR_CLASS", "Inside Constructor!", UVM_HIGH)
  endfunction: new
```

- Fase de construção (*build_phase*):
  - É responsável por construir ou inicializar os componentes do sequenciador.
  - A função ***super.build_phase(phase)*** chama a fase de construção da classe base ***uvm_sequencer***, garantindo que as funcionalidades de construção da classe pai sejam executadas.
  - O macro ***uvm_info*** é utilizado novamente para gerar uma mensagem informativa no log de simulação indicando que a fase de construção foi executada.

```
  function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       `uvm_info("SEQR_CLASS", "Build Phase!", UVM_HIGH)
  endfunction: build_phase
```

- Fase de conexão (*connect_phase*):
  - É responsável por conectar as portas de comunicação entre os componentes.

- Neste caso, não há uma implementação específica dentro do método ***connect_phase***, mas este método normalmente é usado para conectar o sequenciador a outros componentes (como o driver ou o monitor).
- O ***super.connect_phase(phase)*** chama a implementação da classe base ***uvm_sequencer***.
- O macro ***uvm_info*** novamente gera uma mensagem informativa no log de simulação indicando que a fase de conexão foi executada.

```
function void connect_phase(uvm_phase phase);
       super.connect_phase(phase);
       `uvm_info("SEQR_CLASS", "Connect Phase!", UVM_HIGH)
  endfunction: connect_phase
```

9. sequence_item.sv

A classe alu_sequence_item define e restringe os sinais de entrada (***reset, a, b, op_code***) e de saída (***result, carry_out***) para garantir que os valores dos operandos e do código de operação estejam dentro de limites aceitáveis. Um construtor é fornecido para inicializar a classe, permitindo seu uso em sequenciadores UVM. O sequenciador pode usar instâncias de ***alu_sequence_item*** para gerar e controlar as operações enviadas ao DUT (Design Sob Teste). Os itens de sequência gerados por essa classe serão consumidos pelo driver para executar a operação e monitorar os resultados.

- **Herança:**
  - A classe ***alu_sequence_item***, derivada de ***uvm_sequence_item*** (classe base para objetos que representam itens de sequência no UVM)
  - É registrada no sistema UVM através do macro u***vm_object_utils(alu_sequence_item)***.
  - Esse registro possibilita o uso da classe ***alu_sequence_item*** em sequenciadores e outros componentes UVM.

```
class alu_sequence_item extends uvm_sequence_item;
  `uvm_object_utils(alu_sequence_item)
```

- As variáveis ***reset, a, b*** e ***op_code*** são declaradas como ***rand***, o que significa que podem receber valores aleatórios durante a simulação, respeitando as restrições que serão definidas posteriormente.
  - ***reset*** é um sinal de controle para reiniciar a operação.

- ***a*** e ***b*** são operandos de 8 bits utilizados na operação da ALU.
- ***op_code*** é um código de operação de 4 bits, representando a operação que será executada pela ALU (como soma, subtração, etc.).
- As variáveis result e carry_out são definidas como saídas:
  - result armazena o resultado da operação da ALU.
  - ***carry_out*** é o bit de transporte resultante da operação (geralmente utilizado em operações como soma).

```
rand logic reset;
rand logic [7:0] a, b;
rand logic [3:0] op_code;

logic [7:0] result; // output
bit carry_out; // output
```

- As restrições são usadas para definir o intervalo de valores válidos para as variáveis aleatórias.
  - ***input1_c***: A variável ***a*** pode ter valores entre 10 e 20.
  - ***input2_c***: A variável ***b*** pode ter valores entre 1 e 10.
  - ***op_code_c***: O ***op_code*** pode ter os valores 0, 1, 2 ou 3 (representando as operações da ALU, como soma, subtração, etc.).

```
constraint input1_c {a inside {[10:20]};}
constraint input2_c {b inside {[1:10]};}
constraint op_code_c {op_code inside {0,1,2,3};}
```

- A classe é inicializada através do construtor ***new***, que chama o construtor da classe base ***uvm_sequence_item (via super.new(name))***.
  - Um argumento pode ser passado para o construtor para definir o nome da instância, que tem como valor padrão "***alu_sequence_item***".

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

10. sequence.sv

O arquivo define duas classes principais de sequências UVM que controlam o envio de pacotes de teste para o DUT (Design Under Test). Ambas utilizam a metodologia UVM para gerar e gerenciar sequências de teste, mas cada uma tem uma função específica:

alu_base_sequence: Essa sequência é responsável por gerar um pacote de reset. O ***reset_pkt*** é um item de sequência que ativa o sinal de reset.

alu_test_sequence: Essa sequência gera pacotes de operação da ALU, sem ativar o reset.

- **alu_base_sequence:** Seu objetivo é garantir que o DUT comece em um estado conhecido (reset ativo) antes da execução de qualquer teste.
    - A classe ***alu_base_sequence*** herda de ***uvm_sequence***, que é a classe base para sequências dentro do UVM.
    - Ela contém um objeto do tipo ***alu_sequence_item*** chamado ***reset_pkt***.
    - Este objeto será utilizado para gerar um pacote de reset que será enviado ao DUT.

```
class alu_base_sequence extends uvm_sequence;
  `uvm_object_utils(alu_base_sequence)

  alu_sequence_item reset_pkt;
```

- O construtor da classe chama o construtor da classe base ***uvm_sequence*** e registra uma mensagem de log usando ***uvm_info***.

```
function new(string name= "alu_base_sequence");
  super.new(name);
  `uvm_info("BASE_SEQ", "Inside Constructor!", UVM_HIGH)
endfunction
```

- O método ***body*** gera um pacote de reset e o envia ao sequenciador
    1. Um novo item de sequência ***reset_pkt*** é criado.
    2. O item é iniciado com ***start_item***.
    3. O item é aleatoriamente randomizado com a restrição de que o sinal reset deve ser igual a 1 (***reset==1***).
    4. O item é finalizado com ***finish_item***, o que envia o item para o sequenciador.

```
task body();
  `uvm_info("BASE_SEQ", "Inside body task!", UVM_HIGH)

  reset_pkt = alu_sequence_item::type_id::create("reset_pkt");
  start_item(reset_pkt);
  reset_pkt.randomize() with {reset==1;};
  finish_item(reset_pkt);

endtask: body
```

- ***alu_test_sequence:*** Antes de iniciar a execução dos testes da ALU, o DUT já foi resetado. Essa sequência gera pacotes de teste com operações da ALU (sem reset).
    - A classe ***alu_test_sequence*** herda de ***alu_base_sequence***, ou seja, ela é uma sequência de teste que estende a base da sequência de reset.
    - A classe define um novo objeto item do tipo ***alu_sequence_item***, que será usado para representar uma operação de teste da ALU.

```
class alu_test_sequence extends alu_base_sequence;
  `uvm_object_utils(alu_test_sequence)

  alu_sequence_item item;
```

- O construtor chama o construtor da classe base (***alu_base_sequence***) e registra uma mensagem de log.

```
function new(string name= "alu_test_sequence");
  super.new(name);
  `uvm_info("TEST_SEQ", "Inside Constructor!", UVM_HIGH)
endfunction
```

- O método ***body*** gera um pacote de teste da ALU e o envia ao sequenciador.
    1. Um novo item item de tipo ***alu_sequence_item*** é criado.
    2. O item é iniciado com ***start_item***.
    3. O item é aleatoriamente randomizado, mas dessa vez com a restrição de que ***reset==0***.
    4. O item é finalizado com ***finish_item***.

```
task body();
  `uvm_info("TEST_SEQ", "Inside body task!", UVM_HIGH)

  item = alu_sequence_item::type_id::create("item");
  start_item(item);
  item.randomize() with {reset==0;};
  finish_item(item);

endtask: body
```

11. scoreboard.sv

A ***alu_scoreboard*** é responsável por validar os resultados das operações realizadas pelo DUT. Ela recebe transações do monitor e as compara com os resultados esperados, gerando mensagens de erro ou sucesso.

- A ***alu_scoreboard*** estende ***uvm_scoreboard***, que é uma classe base do UVM usada para validação de resultados.

```
class alu_scoreboard extends uvm_scoreboard;
```

- ***scoreboard_port:*** É um *port* de análise, que recebe transações do monitor e as armazena na scoreboard para comparação.
- ***transactions[$]:*** Uma fila dinâmica(***queue***) que armazena as transações (***alu_sequence_item***) a serem analisadas.

```
uvm_analysis_imp #(alu_sequence_item, alu_scoreboard) scoreboard_port;
alu_sequence_item transactions[$];
```

- O construtor ***new*** chama o construtor da classe base e exibe uma mensagem de log

```
function new(string name = "alu_scoreboard", uvm_component parent);
  super.new(name, parent);
  `uvm_info("SCB_CLASS", "Inside Constructor!", UVM_HIGH)
endfunction: new
```

- O método ***build_phase*** inicializa ***scoreboard_port***.
    - O *port* de análise ***scoreboard_port*** é instanciado.
    - Não há necessidade de buscar interfaces ou recursos da configuração, pois a scoreboard apenas recebe e analisa transações.

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  `uvm_info("SCB_CLASS", "Build Phase!", UVM_HIGH)

  scoreboard_port = new("scoreboard_port", this);
endfunction: build_phase
```

- O ***write*** recebe um ***alu_sequence_item*** do monitor e o adiciona à fila transactions.

```
function void write(alu_sequence_item item);
  transactions.push_back(item);
endfunction: write
```

- O método ***run_phase*** processa continuamente as transações, retirando a mais antiga e chamando ***compare*** para validação.
    - Um loop infinito processa continuamente as transações da ALU.
    - Aguarda até que existam transações (***transactions.size() !***= 0).
    - Retira a transação mais antiga (***pop_front()***) e a compara com o esperado.

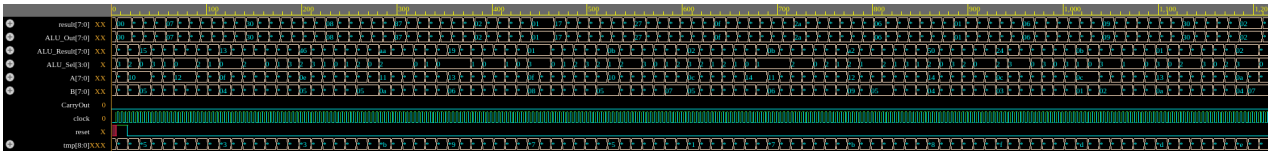
```
task run_phase (uvm_phase phase);
```

```
  super.run_phase(phase);
  `uvm_info("SCB_CLASS", "Run Phase!", UVM_HIGH)

  forever begin
       wait((transactions.size() != 0));  // Espera até haver transações na fila
       alu_sequence_item curr_trans = transactions.pop_front();
       compare(curr_trans); // Chama o método de comparação
  end
endtask: run_phase
```

- O método ***compare*** calcula o resultado esperado da ALU com base no ***op_code***, compara com o resultado gerado pelo DUT e exibe mensagens de erro ou sucesso.
    - Calcula o resultado esperado de acordo com op_code.
    - Compara com ***curr_trans.result***, que contém o resultado gerado pelo DUT.
    - Se houver divergência, um erro UVM (***uvm_error***) é gerado.
    - Se os valores forem iguais, uma mensagem de sucesso (***uvm_info***) é exibida.

```
task compare(alu_sequence_item curr_trans);
  logic [7:0] expected;
  logic [7:0] actual;

  case(curr_trans.op_code)
       0: expected = curr_trans.a + curr_trans.b; // Soma
       1: expected = curr_trans.a - curr_trans.b; // Subtração
       2: expected = curr_trans.a * curr_trans.b; // Multiplicação
       3: expected = curr_trans.a / curr_trans.b; // Divisão
  endcase

  actual = curr_trans.result;

  if(actual != expected) begin
       `uvm_error("COMPARE", $sformatf("Transaction failed! ACT=%d, EXP=%d", actual,
expected))
  end
  else begin
       `uvm_info("COMPARE", $sformatf("Transaction Passed! ACT=%d, EXP=%d", actual,
expected), UVM_LOW)
  end
endtask: compare
```

b. Explicação dos Resultados



1. **Fase de Construção(Constructor):**

Esta fase é onde:

- Os objetos são criados na memória
- A hierarquia começa a ser construída de cima para baixo
- Cada componente é instanciado usando ***new()***
- O tempo ***(@0)*** indica que isso acontece no início da simulação

```
UVM_INFO test.sv(15) @ 0: uvm_test_top [TEST_CLASS] Inside Constructor!
UVM_INFO env.sv(14) @ 0: uvm_test_top.env [ENV_CLASS] Inside Constructor!
UVM_INFO agent.sv(15) @ 0: uvm_test_top.env.agnt [AGENT_CLASS] Inside Constructor!
UVM_INFO scoreboard.sv(16) @ 0: uvm_test_top.env.scb [SCB_CLASS] Inside Constructor!
UVM_INFO driver.sv(14) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Inside Constructor!
UVM_INFO monitor.sv(16) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Inside Constructor!
UVM_INFO sequencer.sv(11) @ 0: uvm_test_top.env.agnt.seqr [SEQR_CLASS] Inside Constructor!
```

2. **Fase de Build**:

Nesta fase:

- Os componentes são configurados
- As conexões TLM são criadas
- Os objetos filhos são criados
- Configurações são aplicadas

```
UVM_INFO test.sv(24) @ 0: uvm_test_top [TEST_CLASS] Build Phase!
UVM_INFO env.sv(23) @ 0: uvm_test_top.env [ENV_CLASS] Build Phase!
UVM_INFO agent.sv(24) @ 0: uvm_test_top.env.agnt [AGENT_CLASS] Build Phase!
UVM_INFO driver.sv(23) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Build Phase!
UVM_INFO monitor.sv(25) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Build Phase!
UVM_INFO sequencer.sv(20) @ 0: uvm_test_top.env.agnt.seqr [SEQR_CLASS] Build Phase!
UVM_INFO scoreboard.sv(25) @ 0: uvm_test_top.env.scb [SCB_CLASS] Build Phase!
```

3. **Fase de Connect**:

Nesta fase:

- As conexões entre componentes são estabelecidas
- Portas TLM são conectadas
- Análise de portas é conectada
- Comunicação entre componentes é configurada

```
UVM_INFO driver.sv(37) @ 0: uvm_test_top.env.agnt.drv [DRIVER_CLASS] Connect Phase!
UVM_INFO monitor.sv(40) @ 0: uvm_test_top.env.agnt.mon [MONITOR_CLASS] Connect Phase!
UVM_INFO sequencer.sv(30) @ 0: uvm_test_top.env.agnt.seqr [SEQR_CLASS] Connect Phase!
UVM_INFO agent.sv(38) @ 0: uvm_test_top.env.agnt [AGENT_CLASS] Connect Phase!
UVM_INFO scoreboard.sv(37) @ 0: uvm_test_top.env.scb [SCB_CLASS] Connect Phase!
UVM_INFO env.sv(36) @ 0: uvm_test_top.env [ENV_CLASS] Connect Phase!
UVM_INFO test.sv(36) @ 0: uvm_test_top [TEST_CLASS] Connect Phase!
```

4. **A hierarquia completa no resultado:**

```
uvm_test_top (Teste Principal)
      │
      ├── env (Ambiente)
      │     │
      │     ├── agnt (Agente)
      │     │     ├── drv (Driver)
      │     │     ├── mon (Monitor)
      │     │     └── seqr (Sequenciador)
      │     │
      │     └── scb (Scoreboard)
```

Cada componente tem um papel específico:

- **Test**: Controla todo o ambiente de teste
- **Environment**: Contém todos os componentes de verificação
- **Agent**: Gerencia os componentes que interagem diretamente com o DUT
- **Driver**: Converte transações em sinais para o DUT
- **Monitor**: Observa os sinais do DUT e cria transações
- **Sequencer**: Controla a ordem das transações de teste
- **Scoreboard**: Verifica se os resultados estão corretos

As mensagens mostram que cada fase é executada em ordem e que todos os componentes são inicializados corretamente, o que é crucial para um ambiente de verificação funcional

adequado.

5. **Comparação**:

```
UVM_INFO scoreboard.sv(105) @ 25: uvm_test_top.env.scb [COMPARE] Transaction Passed! ACT= 48,
EXP= 48
```

Onde:

- @ 25 é o tempo da simulação
- [COMPARE] indica uma comparação de resultado
- ACT= 48 é o valor atual (resultado obtido)
- EXP= 48 é o valor esperado (resultado previsto)

Alguns resultados na saída:

```
ACT= 48, EXP= 48  (@ 25)
ACT= 21, EXP= 21  (@ 41)
ACT=  4, EXP=  4  (@ 49)
ACT=  7, EXP=  7  (@ 65)
ACT= 27, EXP= 27  (@ 73)
ACT= 55, EXP= 55  (@ 97)
ACT= 10, EXP= 10  (@ 113)
```

6. **Estatísticas Finais:**

```
** Report counts by severity
UVM_INFO :  479
UVM_WARNING :  0
UVM_ERROR :    0
UVM_FATAL :    0
```

7. **Análise dos resultados totais:**

```
** Report counts by id
[AGENT_CLASS]  3
[BASE_SEQ]   102
[COMPARE]   149       <- Número total de comparações
[DRIVER_CLASS]        4
[ENV_CLASS]    3
[MONITOR_CLASS]       4
[RNTST]        1
[SCB_CLASS]    4
[SEQR_CLASS]   3
[TEST_CLASS]   4
[TEST_SEQ]   200
```

1. **Sucesso do Teste**:
    a. Todas as 149 comparações passaram

   b. Nenhum erro ou aviso foi reportado
   c. O teste executou completamente até o final
2. **Valores Testados**:
   a. Valores pequenos (ex: 1, 2, 4, 6)
   b. Valores médios (ex: 21, 27, 48)
   c. Valores maiores (ex: 135, 170, 180)
   d. Mostra uma boa cobertura de diferentes magnitudes de números
3. **Padrão de Execução**:
   a. As comparações ocorrem em intervalos regulares
   b. Cada resultado é verificado duas vezes (padrão comum em testes UVM)
   c. O tempo de simulação avança consistentemente
4. **Aspectos Positivos**:
   a. Consistência nos resultados
   b. Todas as comparações bem-sucedidas
   c. Boa variedade de valores testados
   d. Execução ordenada e sistemática
5. **Cobertura**:
   a. 149 comparações diferentes
   b. 200 sequências de teste executadas
   c. Boa distribuição de valores testados
   d. Teste abrangente da funcionalidade da ULA

c. Análise comparativa

1. Testbench UVM da ULA:
   - É um testbench baseado em UVM (Universal Verification Methodology)
   - Estrutura mais complexa e hierárquica
   - Componentes principais:
     - Test class
     - Environment
     - Agent
     - Driver
     - Monitor
     - Sequencer

    - Scoreboard
  - Foco em verificação baseada em transações
  - Usa sequências para estímulos
2. Testbench Simples ULA:
  - Testbench tradicional mais simples
  - Características principais:
    - Testes determinísticos
    - Testes aleatórios limitados
    - Verificações básicas
  - Estrutura:
    - Task para aplicar testes
    - Verificações simples de assert
    - Testes de operações básicas da ULA
3. Testbench Test Vectors ULA:
  - Testbench baseado em vetores de teste
  - Características:
    - Leitura de vetores de arquivo
    - Estatísticas detalhadas de erros
    - Cobertura funcional
  - Recursos avançados:
    - Estruturas para tracking de erros
    - Verificações mais complexas
    - Relatórios detalhados
4. Testbench Self-Checking ULA:
  - Testbench self-checking mais sofisticado
  - Características:
    - Verificações automáticas
    - Estruturas de dados complexas
    - Cobertura funcional detalhada
  - Recursos:
    - Enumerações para operações
    - Estruturas para resultados esperados
    - Estatísticas detalhadas de erros

Comparação das principais características:

1. Complexidade:
    - UVM: Alta (mais complexo)
    - Simples: Baixa
    - Test Vectors: Média
    - Self-Checking: Média-Alta
2. Reusabilidade:
    - UVM: Alta
    - Simples: Baixa
    - Test Vectors: Média
    - Self-Checking: Alta
3. Verificações:

```
// UVM
UVM_INFO scoreboard.sv(105) @ 25: uvm_test_top.env.scb [COMPARE] Transaction Passed!

// Simples
assert(A >= B) else $error("Violação: A deve ser maior ou igual a B");

// Test Vectors
enhanced_check_result(in_a, in_b, op, expected_result, expected_carry, test_num);

// Self-Checking
enhanced_check_result(a, b, op, actual_result, actual_carry, test_num);
```

4. Cobertura:

```
// Test Vectors e Self-Checking têm cobertura similar:
covergroup ULA_cov @(posedge clock);
	cp_op: coverpoint ULA_Sel { ... }
	cp_carry: coverpoint CarryOut { ... }
	cp_zero_result: coverpoint ULA_Out { ... }
	...
endgroup
```

5. **Relatórios**:

```
// Self-Checking e Test Vectors têm relatórios detalhados:
function void print_error_report();
	$display("=== Relatório Detalhado de Erros ===");
	$display("Operações inválidas: %0d", errors.invalid_ops);
	...
endfunction
```

1. **UVM Testbench**:

- Melhor para projetos grandes
- Quando reusabilidade é importante
- Para verificação complexa

2. **Testbench Simples**:
   - Para testes rápidos
   - Verificação inicial
   - Projetos pequenos
3. **Test Vectors Testbench**:
   - Quando há muitos casos de teste
   - Necessidade de reprodutibilidade
   - Testes regressivos
4. **Self-Checking Testbench**:
   - Verificação detalhada
   - Necessidade de relatórios detalhados
   - Cobertura funcional importante

4. CONCLUSÃO

A análise comparativa das quatro abordagens de verificação revelou que cada uma delas contribui com aspectos positivos para o processo, e que a implementação progressiva facilita o aprendizado e aumenta a confiabilidade.

- A cobertura da verificação evoluiu do Testbench Simples (18 testes), passando pelo Self-Checking com Vetores (100 casos) e Self-Checking (100 testes), até o UVM, que maximizou os resultados com um framework estruturado.
- Cada abordagem contribuiu com aspectos específicos: o Simples foi ideal para depuração inicial e validação rápida; o de Vetores foi excelente para testes de regressão e casos específicos; o Self-Checking equilibrou determinismo e aleatoriedade; e o UVM forneceu um ambiente profissional e escalável.
- Os resultados consolidados incluem 218 testes executados, zero discrepâncias, timing consistente e cobertura completa de casos críticos.
- A abordagem progressiva permitiu uma complexidade gradual, facilitando o aprendizado, além de ter promovido a verificação complementar entre as metodologias, resultando em maior confiabilidade através da redundância e estabelecendo uma base sólida para desenvolvimentos futuros.

Com isso, a implementação progressiva demonstrou que cada abordagem agrega um valor único ao processo de verificação, e que o ambiente UVM oferece a solução mais completa e profissional.