* 1. Testbench file:

module dynamic\_array\_tb;

    // Declare dynamic arrays

    int dyn\_arr1[];

    int dyn\_arr2[];

    initial begin

        // initialize dyn\_arr2 array elements with (9,1,8,3,4,4)

        dyn\_arr2 = '{9,1,8,3,4,4};

        // Allocate 6 elements in dyn\_arr1 using new

        dyn\_arr1 = new[6];

        // Initialize dyn\_arr1 with its index as value using foreach

        foreach (dyn\_arr1[i]) begin

            dyn\_arr1[i] = i;

        end

        // Display dyn\_arr1 and its size

        $display("dyn\_arr1: %p", dyn\_arr1);

        $display("dyn\_arr1 size: %0d", dyn\_arr1.size());

        // Delete dyn\_arr1

        dyn\_arr1.delete();

        // Display dyn\_arr2 before any operations

        // $display("Original dyn\_arr2: %p", dyn\_arr2);

        // Reverse dyn\_arr2

        dyn\_arr2.reverse();

        $display("Reversed dyn\_arr2: %p", dyn\_arr2);

        // Sort dyn\_arr2

        dyn\_arr2.sort();

        $display("Sorted dyn\_arr2: %p", dyn\_arr2);

        // Reverse sort (descending)

        dyn\_arr2.rsort();

        $display("Reverse Sorted dyn\_arr2: %p", dyn\_arr2);

        // Shuffle dyn\_arr2

        dyn\_arr2.shuffle();

        $display("Shuffled dyn\_arr2: %p", dyn\_arr2);

    end

endmodule

* 1. Transcript:

dyn\_arr1: '{0, 1, 2, 3, 4, 5}

dyn\_arr1 size: 6

Reversed dyn\_arr2: '{4, 4, 3, 8, 1, 9}

Sorted dyn\_arr2: '{1, 3, 4, 4, 8, 9}

Reverse Sorted dyn\_arr2: '{9, 8, 4, 4, 3, 1}

Shuffled dyn\_arr2: '{8, 1, 4, 3, 9, 4}

* 1. Design file:

module counter (clk ,rst\_n, load\_n, up\_down, ce, data\_load, count\_out, max\_count, zero);

parameter WIDTH = 4;

input clk;

input rst\_n;

input load\_n;

input up\_down;

input ce;

input [WIDTH-1:0] data\_load;

output reg [WIDTH-1:0] count\_out;

output max\_count;

output zero;

always @(posedge clk) begin

    if (!rst\_n)

        count\_out <= 0;

    else if (!load\_n)

        count\_out <= data\_load;

    else if (ce)

        if (up\_down)

            count\_out <= count\_out + 1;

        else

            count\_out <= count\_out - 1;

end

assign max\_count = (count\_out == {WIDTH{1'b1}})? 1:0;

assign zero = (count\_out == 0)? 1:0;

endmodule

* 1. Verification plan:

| **Label** | **Design Requirement Description** | **Stimulus Generation** | **Functional Coverage** | **Functionality Check** |
| --- | --- | --- | --- | --- |
| COUNTER1 | When rst\_n is asserted low, count\_out should be 0 and zero should be 1, max\_count = 0. | Directed at the start of simulation | - | Checker ensures count\_out = 0, zero = 1, max\_count = 0. |
| COUNTER2 | When load\_n is low, data\_load should be loaded into count\_out. | Directed + random valid data\_load | - | Checker verifies count\_out == data\_load when load\_n == 0. |
| COUNTER3 | When ce = 1, up\_down = 1, counter increments each clock cycle. | Directed increment loop | - | Checker validates correct counting sequence and max\_count assertion at max. |
| COUNTER4 | When ce = 1, up\_down = 0, counter decrements each clock cycle. | Directed decrement loop | - | Checker validates correct counting sequence and zero assertion at 0. |
| COUNTER5 | If ce = 0, counter must hold its value regardless of up\_down. | Directed with ce = 0 | - | Checker ensures count\_out is stable when ce == 0. |
| COUNTER6 | When incrementing reaches 2^WIDTH - 1, max\_count should be 1. | Directed + edge condition testing | - | Checker asserts max\_count == 1 only at count\_out == max. |
| COUNTER7 | When decrementing reaches 0, zero should be 1. | Directed + edge condition testing | - | Checker asserts zero == 1 only at count\_out == 0. |
| COUNTER8 | Verify counter behavior under random sequences of load\_n, up\_down, and ce. | Constrained random tests | - | Scoreboard checks behavior against expected model. |
| COUNTER9 | Ensure correct priority of load\_n over ce. | Mixed scenario tests | - | Checker confirms load\_n overrides counting when active. |

* 1. Package file:

package counter\_pkg;

  // Parameter for data width

  parameter int WIDTH = 4;

  // Class to generate constrained stimulus for counter

  class counter\_txn;

    // Signals (inputs to DUT)

    rand bit rst\_n;

    rand bit load\_n;

    rand bit ce;

    rand bit up\_down;

    rand bit [WIDTH - 1:0] data\_load;

    constraint c\_rst\_n {

        rst\_n dist {0 := 5, 1 := 95}; // 5% active (0), 95% inactive (1)

    }

    constraint c\_load\_n {

        load\_n dist {0 := 70, 1 := 30}; // 70% active (0), 30% inactive (1)

    }

    constraint c\_ce {

        ce dist {0 := 30, 1 := 70}; // 30% active (0), 70% inactive (1)

    }

    constraint c\_up\_down {

      up\_down dist {0 := 50, 1 := 50}; // 50% active (0), 50% inactive (1)

    }

    function void reset (rst\_n = 1, load\_n = 1, ce = 0, up\_down = 0, data\_load = 0);

        this.rst\_n    = rst\_n;

        this.load\_n   = load\_n;

        this.ce       = ce;

        this.up\_down  = up\_down;

        this.data\_load = data\_load;

    endfunction

  endclass : counter\_txn

endpackage : counter\_pkg

* 1. Testbench file:

import counter\_pkg::\*;

module counter\_tb;

    // TESTBENCH VARIABLES

    int local\_error\_count = 0;

    int error\_count = 0;

    int pass\_count  = 0;

    // DUT

    logic clk;

    logic rst\_n;

    logic load\_n;

    logic up\_down;

    logic ce;

    logic [WIDTH - 1:0] data\_load;

    logic [WIDTH - 1:0] count\_out;

    logic max\_count;

    logic zero;

    // GOLDEN MODEL

    logic [WIDTH - 1:0] golden\_count\_out;

    logic golden\_max\_count;

    logic golden\_zero;

    counter #(.WIDTH(WIDTH)) DUT (

        .clk(clk),

        .rst\_n(rst\_n),

        .load\_n(load\_n),

        .up\_down(up\_down),

        .ce(ce),

        .data\_load(data\_load),

        .count\_out(count\_out),

        .max\_count(max\_count),

        .zero(zero)

    );

    counter\_txn txn;

    always #5 clk = ~clk;

    task wait\_cycles(input int num\_cycles);

        repeat (num\_cycles) @(negedge clk);

    endtask

    task assert\_reset();

        rst\_n = 0;

    endtask

    task deassert\_reset();

        rst\_n = 1;

    endtask

    task golden\_model();

        if (!rst\_n) begin

            golden\_count\_out = 0;

        end else if (!load\_n) begin

            golden\_count\_out = data\_load;

        end else if (ce) begin

            if (up\_down) begin

                golden\_count\_out = golden\_count\_out + 1;

            end else begin

                golden\_count\_out = golden\_count\_out - 1;

            end

        end

        golden\_max\_count = (golden\_count\_out == {WIDTH{1'b1}});

        golden\_zero = (golden\_count\_out == 0);

    endtask

    task check\_result();

        local\_error\_count = 0;

        if (count\_out !== golden\_count\_out) begin

            $error("[ERROR] count\_out mismatch. Expected: %0d, Got: %0d", golden\_count\_out, count\_out);

            local\_error\_count++;

        end

        if (max\_count !== golden\_max\_count) begin

            $error("[ERROR] max\_count mismatch. Expected: %b, Got: %b", golden\_max\_count, max\_count);

            local\_error\_count++;

        end

        if (zero !== golden\_zero) begin

            $error("[ERROR] zero mismatch. Expected: %b, Got: %b", golden\_zero, zero);

            local\_error\_count++;

        end

        if (local\_error\_count == 0) begin

            pass\_count++;

            $display("[PASS] Outputs match expected values.");

        end else begin

            error\_count++;

            $display("[FAIL] Total mismatches in this check: %0d", local\_error\_count);

        end

    endtask

    initial begin

        $display("Starting counter testbench...");

        clk = 0;

        rst\_n = 1;

        txn = new();

        // COUNTER1

        $display("\nCOUNTER1\n");

        assert\_reset();

        golden\_model();

        wait\_cycles(1);

        deassert\_reset();

        check\_result();

        // COUNTER2

        $display("\nCOUNTER2\n");

        repeat (10) begin

            assert (txn.randomize(load\_n, data\_load));

            drive\_inputs();

            golden\_model();

            wait\_cycles(1);

            if (~load\_n) begin

                check\_result();

            end

        end

        txn.reset();

        // COUNTER3 & COUNTER4

        $display("\nCOUNTER3 & COUNTER4\n");

        repeat (10) begin

            assert (txn.randomize(ce, up\_down));

            drive\_inputs();

            golden\_model();

            wait\_cycles(1);

            if (ce) begin

                check\_result();

            end

        end

        txn.reset();

        // COUNTER5

        $display("\nCOUNTER5\n");

        repeat (10) begin

            assert (txn.randomize(up\_down));

            drive\_inputs();

            golden\_model();

            wait\_cycles(1);

            check\_result();

        end

        txn.reset();

        // COUNTER6 & COUNTER7

        $display("\nCOUNTER6 & COUNTER7\n");

        repeat (50) begin

            assert (txn.randomize(ce, up\_down));

            drive\_inputs();

            golden\_model();

            wait\_cycles(1);

            if (max\_count) begin

                check\_result();

            end

            if (zero) begin

                check\_result();

            end

        end

        txn.reset();

        // COUNTER8

        $display("\nCOUNTER8\n");

        repeat (10) begin

            assert (txn.randomize());

            drive\_inputs();

            golden\_model();

            wait\_cycles(1);

            check\_result();

        end

        txn.reset();

        // COUNTER9

        $display("\nCOUNTER9\n");

        repeat (10) begin

            assert (txn.randomize(load\_n, ce));

            drive\_inputs();

            golden\_model();

            wait\_cycles(1);

            if (~load\_n & ce) begin

                check\_result();

            end

        end

        // Display completion message

        $display("Simulation Completed: %0d test cases executed.", pass\_count + error\_count);

        $display("Test Summary: Passed = %0d, Failed = %0d", pass\_count, error\_count);

        $stop;

    end

    task drive\_inputs();

        rst\_n     = txn.rst\_n;

        load\_n    = txn.load\_n;

        ce        = txn.ce;

        up\_down   = txn.up\_down;

        data\_load = txn.data\_load;

    endtask

endmodule

* 1. Do file:

vlog counter\_pkg.sv

vlog -f src\_files.list +cover -covercells

vsim -voptargs=+acc work.counter\_tb -cover

add wave \*

coverage save counter\_tb.ucdb -onexit

run -all

* 1. Waveform snippet:

صورة تحتوي على نص, لقطة شاشة, عرض, برمجيات

قد يكون المحتوى المعد بواسطة الذكاء الاصطناعي غير صحيح.

* 1. Transcript:

COUNTER1

[PASS] Outputs match expected values.

.

.

.

COUNTER9

[PASS] Outputs match expected values.

Simulation Completed: 48 test cases executed.

Test Summary: Passed = 48, Failed = 0

* 1. Bugs:

No bugs detected.

* 1. Coverage report:

> vcover report counter\_tb.ucdb -details -annotate -all -output coverage\_rpt.txt

==============================================================================

Branch Coverage:

Enabled Coverage Bins Hits Misses Coverage

---------------- ---- ---- ------ --------

Branches 10 10 0 100.00%

==============================================================================

Statement Coverage:

Enabled Coverage Bins Hits Misses Coverage

---------------- ---- ---- ------ --------

Statements 7 7 0 100.00%

==============================================================================

Toggle Coverage:

Enabled Coverage Bins Hits Misses Coverage

---------------- ---- ---- ------ --------

Toggles 30 30 0 100.00%

* 1. Design file:

module ALSU(A, B, cin, serial\_in, red\_op\_A, red\_op\_B, opcode, bypass\_A, bypass\_B, clk, rst, direction, leds, out);

parameter INPUT\_PRIORITY = "A";

parameter FULL\_ADDER = "ON";

input clk, cin, rst, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in;

input [2:0] opcode;

input signed [2:0] A, B;

output reg [15:0] leds;

output reg signed [5:0] out;

reg red\_op\_A\_reg, red\_op\_B\_reg, bypass\_A\_reg, bypass\_B\_reg, direction\_reg, serial\_in\_reg;

reg signed [1:0] cin\_reg;

reg [2:0] opcode\_reg;

reg signed [2:0] A\_reg, B\_reg;

wire invalid\_red\_op, invalid\_opcode, invalid;

//Invalid handling

assign invalid\_red\_op = (red\_op\_A\_reg | red\_op\_B\_reg) & (opcode\_reg[1] | opcode\_reg[2]);

assign invalid\_opcode = opcode\_reg[1] & opcode\_reg[2];

assign invalid = invalid\_red\_op | invalid\_opcode;

//Registering input signals

always @(posedge clk or posedge rst) begin

  if(rst) begin

     cin\_reg <= 0;

     red\_op\_B\_reg <= 0;

     red\_op\_A\_reg <= 0;

     bypass\_B\_reg <= 0;

     bypass\_A\_reg <= 0;

     direction\_reg <= 0;

     serial\_in\_reg <= 0;

     opcode\_reg <= 0;

     A\_reg <= 0;

     B\_reg <= 0;

  end else begin

     cin\_reg <= cin;

     red\_op\_B\_reg <= red\_op\_B;

     red\_op\_A\_reg <= red\_op\_A;

     bypass\_B\_reg <= bypass\_B;

     bypass\_A\_reg <= bypass\_A;

     direction\_reg <= direction;

     serial\_in\_reg <= serial\_in;

     opcode\_reg <= opcode;

     A\_reg <= A;

     B\_reg <= B;

  end

end

//leds output blinking

always @(posedge clk or posedge rst) begin

  if(rst) begin

     leds <= 0;

  end else begin

      if (invalid)

        leds <= ~leds;

      else

        leds <= 0;

  end

end

//ALSU output processing

always @(posedge clk or posedge rst) begin

  if(rst) begin

    out <= 0;

  end

  else begin

    if (bypass\_A\_reg && bypass\_B\_reg)

      out <= (INPUT\_PRIORITY == "A")? A\_reg: B\_reg;

    else if (bypass\_A\_reg)

      out <= A\_reg;

    else if (bypass\_B\_reg)

      out <= B\_reg;

    else if (invalid)

        out <= 0;

    else begin

        case (opcode)

          3'h0: begin

            if (red\_op\_A\_reg && red\_op\_B\_reg)

              out <= (INPUT\_PRIORITY == "A")? |A\_reg: |B\_reg;

            else if (red\_op\_A\_reg)

              out <= |A\_reg;

            else if (red\_op\_B\_reg)

              out <= |B\_reg;

            else

              out <= A\_reg | B\_reg;

          end

          3'h1: begin

            if (red\_op\_A\_reg && red\_op\_B\_reg)

              out <= (INPUT\_PRIORITY == "A")? ^A\_reg: ^B\_reg;

            else if (red\_op\_A\_reg)

              out <= ^A\_reg;

            else if (red\_op\_B\_reg)

              out <= ^B\_reg;

            else

              out <= A\_reg ^ B\_reg;

          end

          3'h2: out <= A\_reg + B\_reg;

          3'h3: out <= A\_reg \* B\_reg;

          3'h4: begin

            if (direction\_reg)

              out <= {out[4:0], serial\_in\_reg};

            else

              out <= {serial\_in\_reg, out[5:1]};

          end

          3'h5: begin

            if (direction\_reg)

              out <= {out[4:0], out[5]};

            else

              out <= {out[0], out[5:1]};

          end

        endcase

    end

  end

end

endmodule

* 1. Verification plan:
  2. Package file:
  3. Testbench file:
  4. Do file:
  5. Waveform snippet:
  6. Transcript:
  7. Bugs:
  8. Coverage report: