

**Project Documentation**

**Electronic Design Automation**

**Name: Mo’men Yasser Wahead ID: 19P7251**

**Name: Abdel Rahman Salah Ahmed ID: 18P9174**

**Task (3)**

* **Testing Plan:**

The Testing plan Is based on the following Criteria:

 Identify the inputs and outputs:

* Inputs: A, B, sel,clk
* Outputs: m, r

 Determine the possible input values and their ranges:

* A and B are signed numbers of size N.
* sel is a control signal that determines the operation to be performed (0 for division, 1 for multiplication).

 Define test cases based on different scenarios:

* Test cases for multiplication:
  + Test with positive numbers: A = 5, B = 3, sel = 1
  + Test with negative numbers: A = -5, B = -3, sel = 1
  + Test with a positive and a negative number: A = 5, B = -3, sel = 1
  + Test with zero: A = 0, B = 5, sel = 1
* Test cases for division:
  + Test with positive numbers: A = 10, B = 2, sel = 0
  + Test with negative numbers: A = -10, B = -2, sel = 0
  + Test with a positive dividend and a negative divisor: A = 10, B = -2, sel = 0
  + Test with zero divisor: A = 5, B = 0, sel = 0

** Create a testbench module to apply the test cases and verify the results:**

* Instantiate the SignedMultiplier module.
* Define the necessary signals and variables for stimulus generation and result checking.
* Apply the test cases by assigning values to the inputs.
* Monitor the outputs of the SignedMultiplier module.
* Compare the obtained results with the expected results for each test case.
* Display or assert the test results.

** Run the simulation and observe the results.**

* Use a Verilog simulator, such as ModelSim or to run the simulation using the testbench.
* Check if the outputs match the expected results for each test case.

To generate stimulus for the testbench, you can use the following approach:

1. Define variables for the input signals (A, B, sel) and the expected outputs (m\_expected, r\_expected).
2. Create a test case generator that sets the input values based on your test scenarios. For example:
   * For multiplication:
     + Positive numbers: A = 5, B = 3, sel = 1
     + Negative numbers: A = -5, B = -3, sel = 1
     + Positive and negative number: A = 5, B = -3, sel = 1
     + Zero: A = 0, B = 5, sel = 1
   * For division:
     + Positive numbers: A = 10, B = 2, sel = 0
     + Negative numbers: A = -10, B = -2, sel = 0
     + Positive dividend and negative divisor: A = 10, B = -2, sel = 0
     + Zero divisor: A = 5, B = 0, sel = 0
3. Apply the input values to the module under test.
4. Simulate the module and capture the output values (m, r).
5. Compare the obtained results with the expected results for each test case.
6. Count the number of passed test cases and calculate the pass ratio.

**Coverage criteria refer to the aspects of the design that you want to ensure are exercised or tested during the simulation. Here we use Branch coverage criteria you can use for this design:**

1. Branch coverage: Ensure that both branches of every conditional statement are executed at least once..

We Have the Following branches

Two main branches:

Sel 0 🡺 Division: 4 cases “Two Positive Numbers”,” Two Negative Numbers”, “One positive and one negative Number”

Sel1🡺 Multiplication: 4 cases “Two Positive Numbers”,” Two Negative Numbers”, “One positive and one negative Number”

We have Done 8 cases

Two Modules one sequential and One combinational

Total Number of cases 15 cases Covered

To implement the self-checking test bench, you can use the following steps:

1. Instantiate the Signed Multiplier module within the test bench module.
2. Define the necessary signals and variables for stimulus generation, result checking, and coverage tracking.
3. Generate the test cases and apply the input values to the Signed Multiplier module.
4. Monitor the outputs of the Signed Multiplier module.
5. Compare the obtained results with the expected results for each test case.
6. Increment the pass count for each passed test case.
7. Display or assert the test results, including the pass ratio.

To handle corner cases with respect to timing and different directives, you can modify the test scenarios to include edge cases and boundary values. For example, you can test inputs that have the maximum and minimum values for the respective data types. You can also introduce delays or timing constraints between stimulus application and result checking to consider timing-related aspects of the design.

**Here's an example test bench structure in Verilog:**

module testbench;

reg [3:0] A, B;

reg sel;

reg [3:0] expected\_m, expected\_r;

divMultiplier #(.N(4)) DUT (

.A(A),

.B(B),

.sel(sel),

.m(m),

.r(r)

);

// Test cases

initial begin

test\_case(4'd5, 4'd2, 1'b1, 4'd10, 4'd0);

#10;

test\_case(4'd7, 4'd3, 1'b1, 4'd21, 4'd1);

#10;

// Check outputs

check\_outputs();

test\_case(4'd6, 4'd2, 1'b0, 4'd3, 4'd0);

#10;

// And so on for remaining test cases

end

// Test case task

task test\_case;

input [3:0] a, b;

input sel;

input [3:0] expected\_m, expected\_r;

begin

A = a;

B = b;

sel = sel;

expected\_m = expected\_m;

expected\_r = expected\_r;

end

endtask

// Output check task

task check\_outputs;

begin

if (m != expected\_m) begin

$display("Output m mismatch!");

end

if (r != expected\_r) begin

$display("Output r mismatch!");

End

end

endtask

endmodule

Statement Coverage:

* The test\_inputs task is used to apply all possible combinations of inputs (A, B, sel) to the DUT. This aims to execute every statement at least once.
* Inputs are chosen to cover edge cases like 0, max values, different sign combinations etc.
* By testing all input patterns, every statement within the always blocks will execute at least once.

Branch Coverage:

* The conditional expressions inside the DUT involve comparisons of inputs and signals.
* The test\_branch task explicitly sets inputs to force both the true and false outcomes of branch conditions.
* For example, swapping A and B inputs tests the true and false cases of A < B comparison.
* This covers all the branch points in the code and helps maximize branch coverage.

Block Coverage:

* The DUT logic is divided amongst multiple always blocks for different functions
* Independent tasks are written to target execution of each block in isolation
* For example, test\_mult\_block only tests multiplication while disabling other blocks
* This helps cover each block and their procedural assignments independently
* Together with statement and branch coverage, most blocks should achieve 100% coverage

The plan focuses on systematically applying different input test patterns using independent tasks to maximize coverage of statements, branches and blocks. Automated scripts will be used to measure and report the final coverage achieved. Let me know if any part of the test plan needs more explanation.