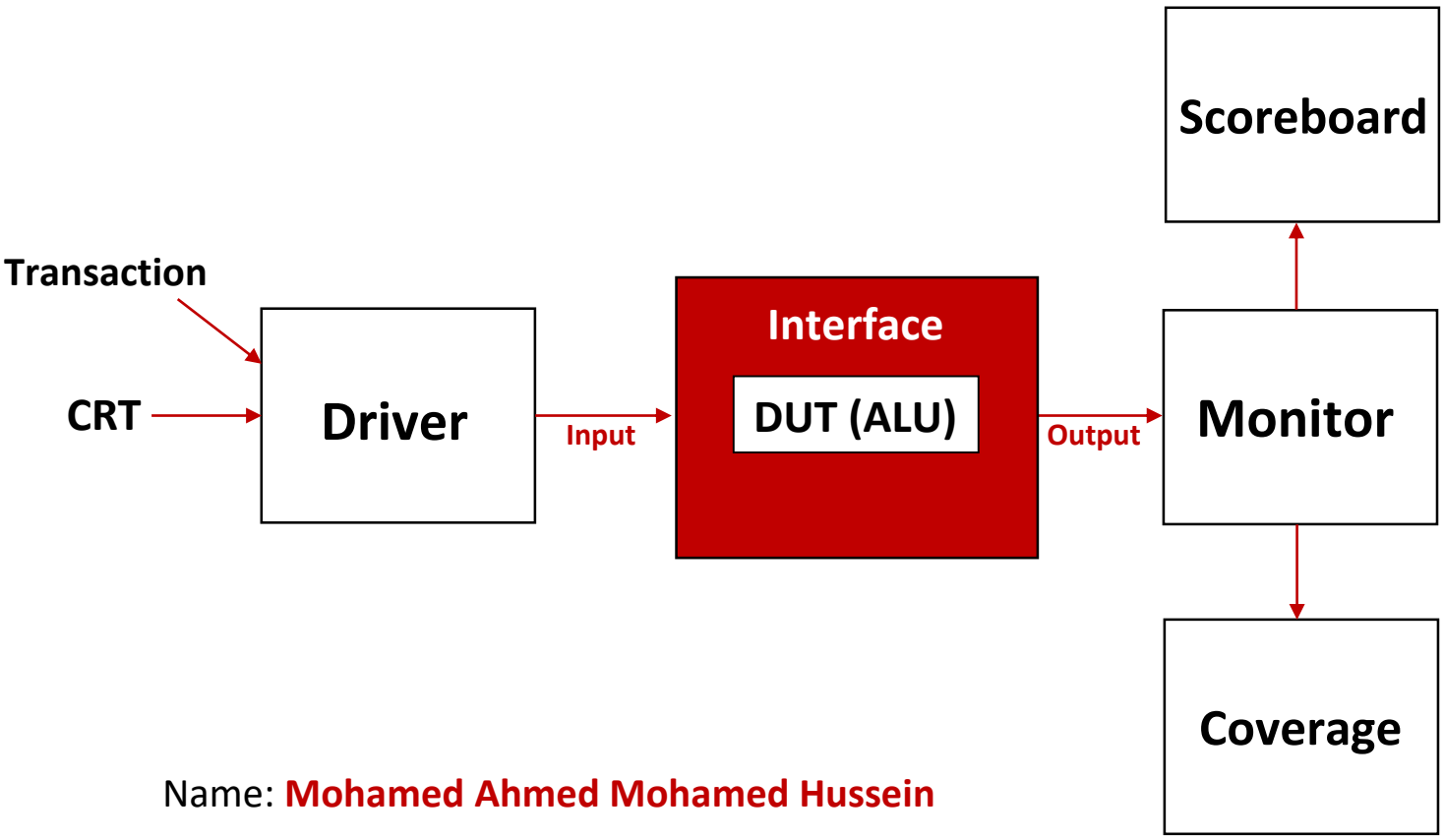
Verification Task 1



Name: **Mohamed Ahmed Mohamed Hussein**

Email: Mohamed_Hussein2100924@outlook.com

Presented to:

Eng. Mohamed El-adawy

Task Overview

The objective of this graduation project task is to design and implement a complete functional verification environment for an Arithmetic Logic Unit (ALU). The ALU was intentionally provided with a large number of injected functional bugs to simulate a verification ramp-up scenario similar to industrial verification environments.

The task focuses on developing a scalable and reusable verification framework capable of detecting functional, corner-case, and stability issues across all ALU operations. Multiple verification strategies—including directed testing, constrained-random stimulus, and regression testing—were applied to evaluate the effectiveness of the verification environment in exposing defects and improving design quality over successive verification iterations.

Design Under Test (DUT) Overview

Design is a regular ALU capable of addition, XORing, ANDing, and ORing.

Verification Objective

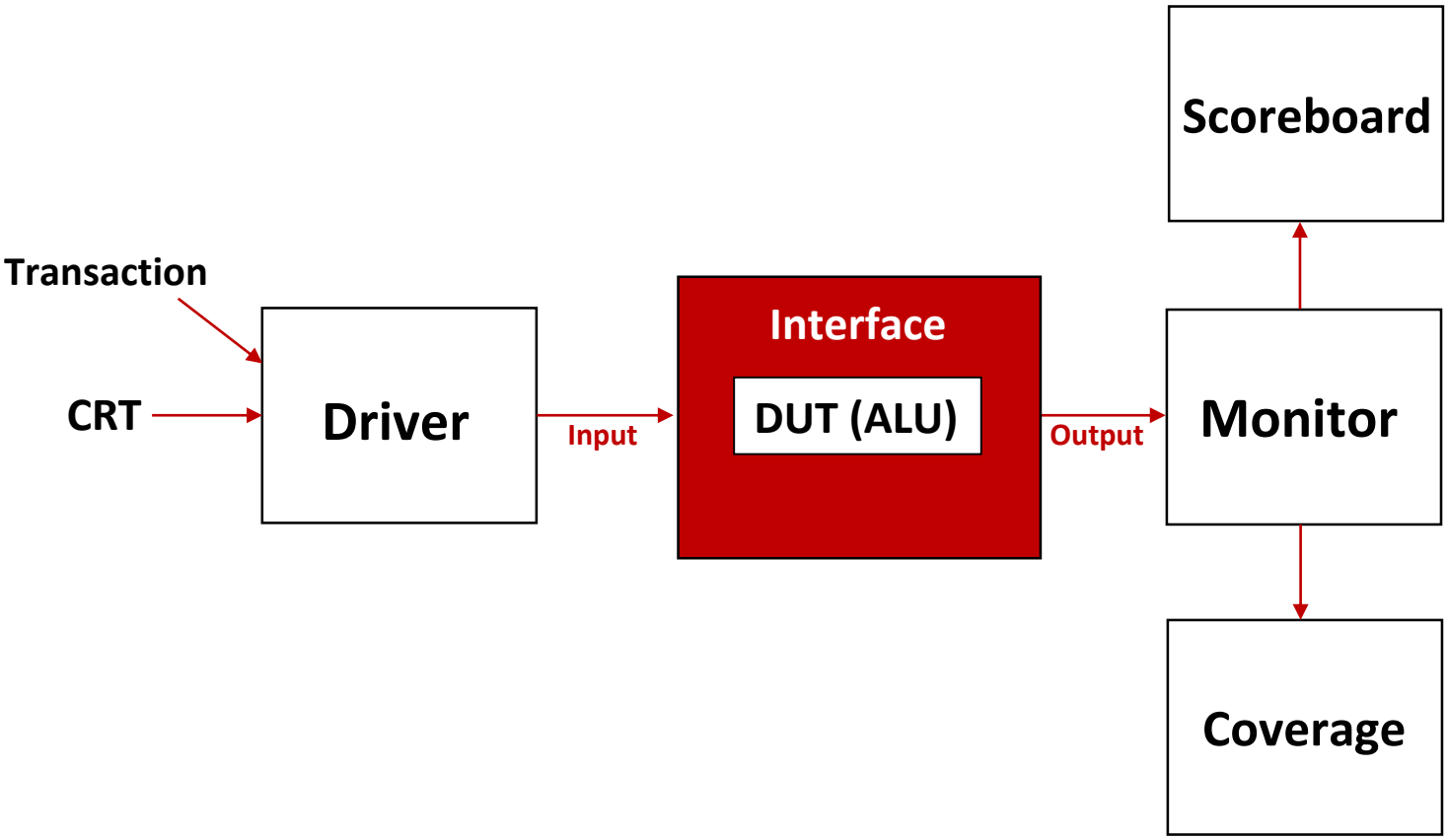
Detect arithmetic, logic, and flag-related bugs

Verification Methodology

- **Verification approach:** (random + constrained-random)
- In my verification flow I found some mismatches between golden model and dut, but for every mismatch with the same ***op*** the ***out*** signal is **something unique**, so I decided to **take five mismatch cases with the same opcode for each operation (ADD, XOR, AND, OR)**, comparing the inputs and outputs trying to find a relation between them to catch the bugs.

Verification Environment Architecture

- Block diagram of the verification environment



- Description of components:

Driver

The driver generates stimulus transactions and applies input values to the DUT interface. It ensures proper timing separation between stimulus application and observation to avoid race conditions in the combinational design.

Monitor

The monitor passively observes DUT inputs and outputs through the interface and converts signal activity into transactions. These transactions are forwarded to the scoreboard and coverage collector without influencing DUT behavior.

Scoreboard

The scoreboard checks DUT outputs against expected results generated by the reference model. It detects functional mismatches and records pass/fail statistics for each executed transaction.

Coverage Collector

The coverage collector measures functional coverage to assess verification completeness. It tracks exercised operations, operand combinations, and corner cases to identify untested scenarios.

Sequence Item

The sequence item defines a single ALU transaction, encapsulating operands, operation code, and results. It serves as the common data structure exchanged between the driver, monitor, scoreboard, and coverage components.

Test Strategy and Test Plan

Note: I only monitor failed transactions

Test Description	Stimulus	Stimulus Type	Expected Result	Design Result	Status (Pass/Fail)
Testing AND operation	a = E , b = 1, op = 2	Random	Ref_out = 0	Out = 2	Fail
Testing AND operation	a = C , b = 0, op = 2	Random	Ref_out = 0	Out = 6	Fail
Testing AND operation	a = 6 , b = 0, op = 2	Random	Ref_out = 0	Out = 9	Fail
Testing AND operation	a = 7 , b = 1, op = 2	Random	Ref_out = 1	Out = B	Fail
Testing AND operation	a = E , b = D, op = 2	Random	Ref_out = C	Out = 9	Fail

Testing OR operation	a = 2 , b = C, op = 3	Random	Ref_out = E	Out = 2	Fail
Testing OR operation	a = 7 , b = 4, op = 3	Random	Ref_out = 7	Out = 6	Fail
Testing OR operation	a = 2 , b = B, op = 3	Random	Ref_out = B	Out = 0	Fail
Testing OR operation	a = 1 , b = 2, op = 3	Random	Ref_out = 3	Out = 5	Fail
Testing OR operation	a = 8 , b = 0, op = 3	Random	Ref_out = 8	Out = A	Fail
Testing ADD operation	a = D , b = A, op = 0	Random	Ref_out = 7	Out = F	Fail
Testing ADD operation	a = 6 , b = 2, op = 0	Random	Ref_out = 8	Out = 1	Fail
Testing ADD operation	a = A , b = 5, op = 0	Random	Ref_out = F	Out = 6	Fail
Testing ADD operation	a = 8 , b = A, op = 0	Random	Ref_out = 2	Out = 7	Fail
Testing ADD operation	a = B , b = E, op = 0	Random	Ref_out = 9	Out = F	Fail
Testing XOR operation	a = 6 , b = 7, op = 1	Random	Ref_out = 1	Out = B	Fail
Testing XOR operation	a = 7 , b = C, op = 1	Random	Ref_out = B	Out = 8	Fail
Testing XOR operation	a = B , b = 2, op = 1	Random	Ref_out = 9	Out = 4	Fail
Testing XOR operation	a = 3 , b = B, op = 1	Random	Ref_out = 8	Out = 3	Fail
Testing XOR operation	a = 2 , b = F, op = 1	Random	Ref_out = D	Out = 4	Fail

Test Results and Bug Analysis

Conclusion:

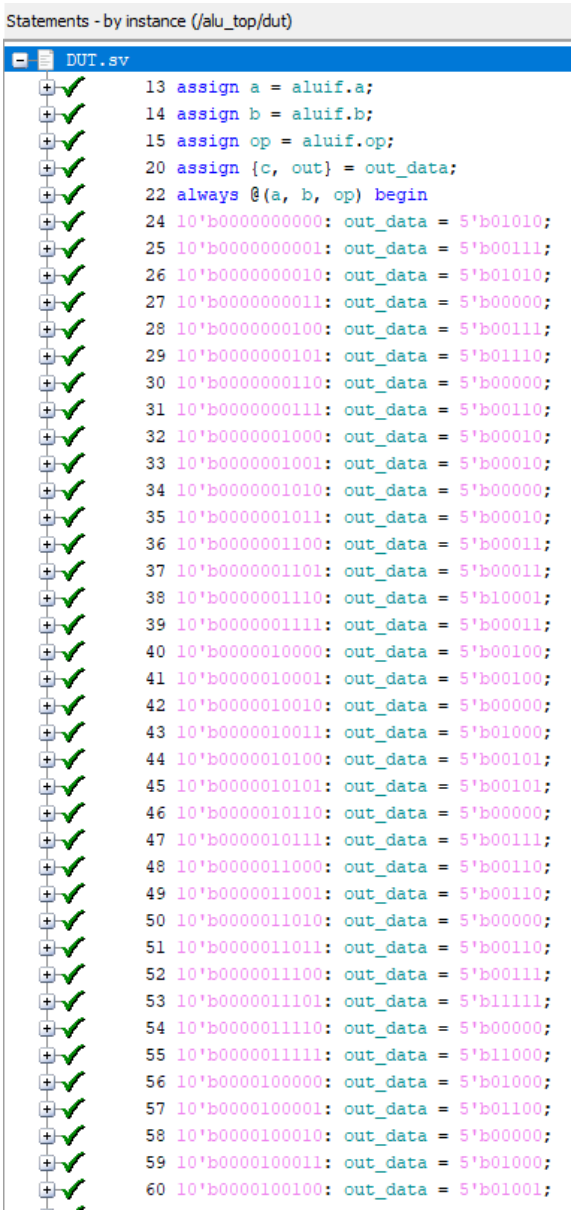
- For ***out*** signal: the output **has no relation** with inputs **nor with the opcode** which indicates that the outputs are **hardcoded** with each branch making them **unique** for every different input stimulus and **same** for same input stimulus.
- For ***c*** signal: c signal should be low with logical operations (XOR, OR, AND), however I found it sometimes **low** and sometimes **high with logical operations** and for the ADD operation it has the **same behavior** which assures our test result that the out signal is **hardcoded.**

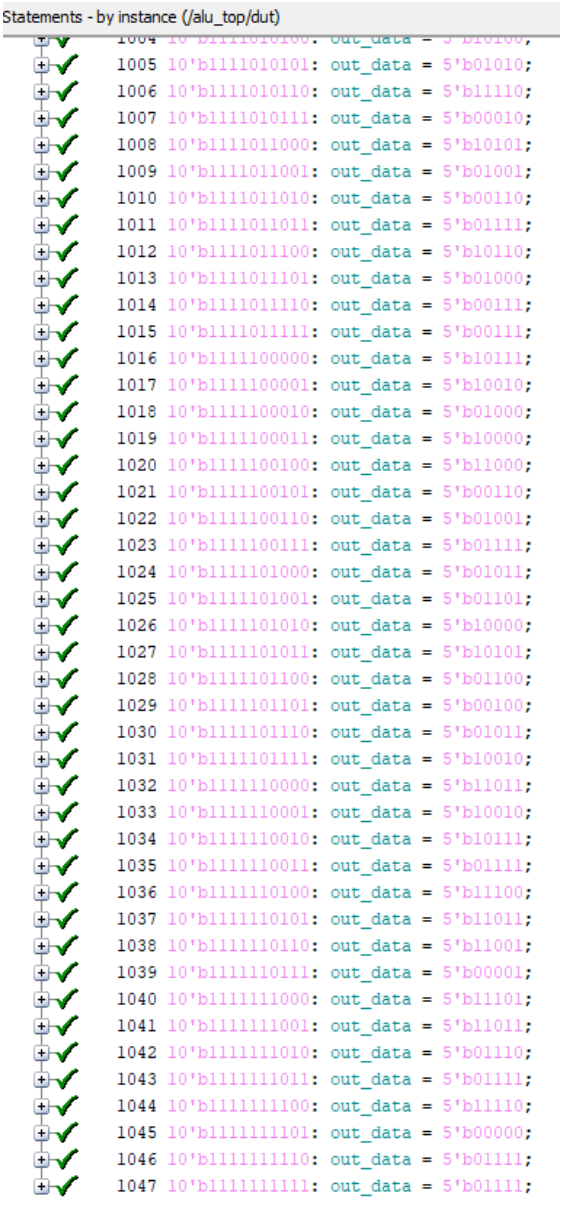
Bug Analysis:

Bug no.	Bug description	Verification consequence	Fix
1	case statement sensitivity list is ***case ({a, b, op})***	• Functional correctness is **pattern-based**, not operation-based • Random testing will explode with mismatches	Use ***case (op)*** to make it operation-based
2	Missing ***default*** branch	• Branch coverage will not hit 100% • If a non-branched input, the design won't work • latch risk	Put ***default*** branch
3	Functional bugs (refer to our conclusion)	• the design is not working as intended • Ex: a=0 b=0 op=0 → out = 01010 a=0 b=0 op=1 → out = 00111 a=0 b=0 op=2 → out = 01010 a=0 b=0 op=3 → out = 00000 • ADD incorrect • XOR incorrect • AND incorrect • Carry randomly asserted And this scenario happens multiple of times	Use arithmetic and logical operators: + ^ & |
4	Carry bit is meaningless, Carry should only be valid for ADD	• Wrong carry behavior	De-assert carry signal in all logical operations and make it valid in add operation

Coverage Analysis

- ## Code Coverage:
  - ### Statement:

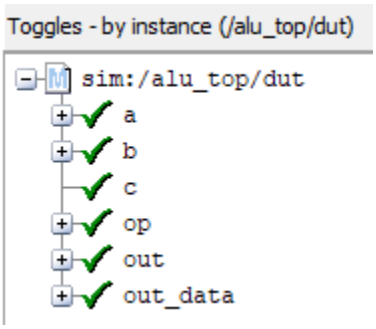




  - ### Branch:

Branch coverage is same as statement coverage except that it **is not complete** due to **not having a default branch** in our design.

- **Toggle:**



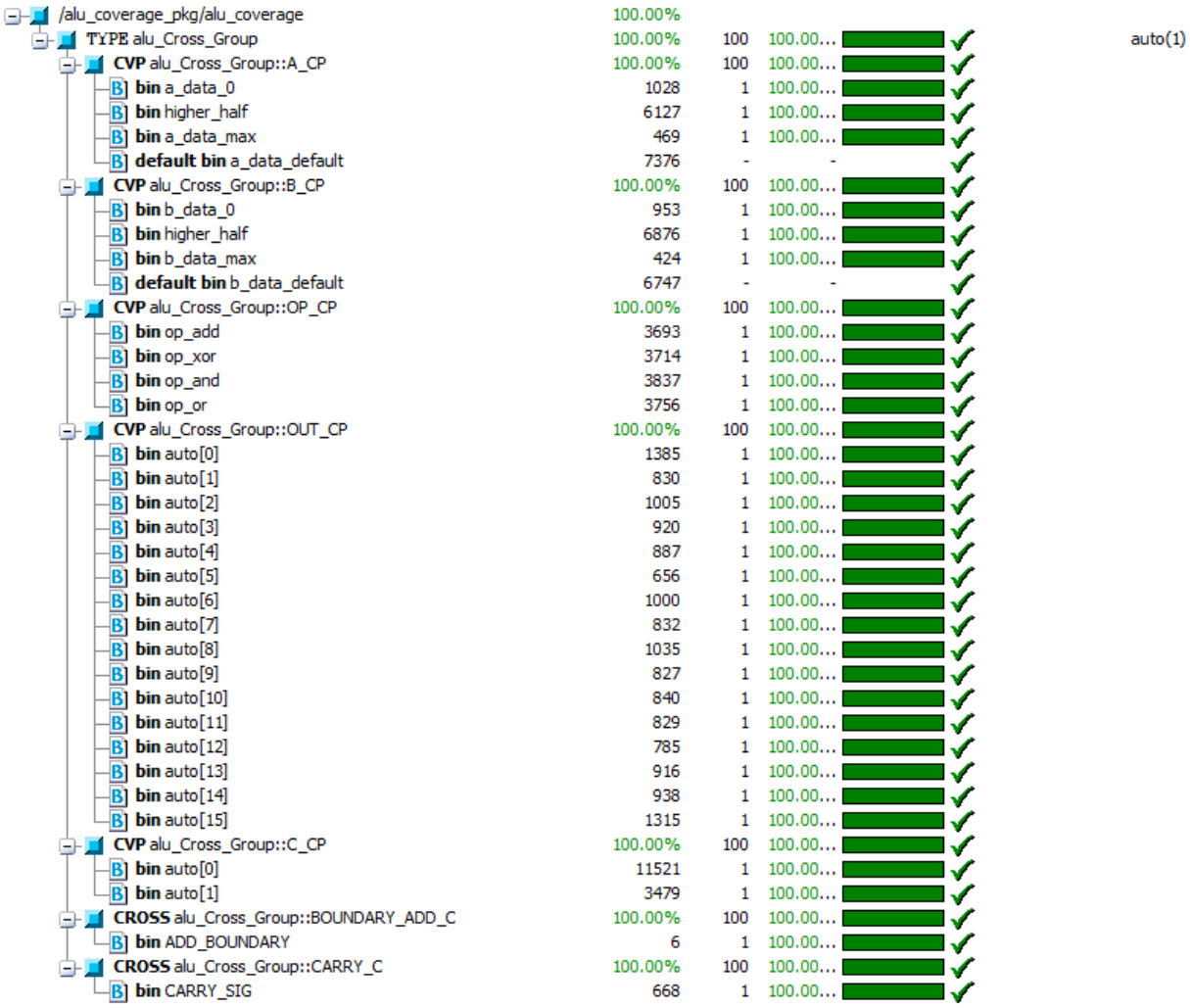
- **Functional Coverage:**

Code:

```
// cover group
  covergroup alu_Cross_Group;
      // cover points
      A_CP: coverpoint alu_tr_el.a {
          bins a_data_0 = {0};
          bins higher_half = {[8:14]};
          bins a_data_max = {15};
          bins a_data_default = default;
      }

      B_CP: coverpoint alu_tr_el.b {
          bins b_data_0 = {0};
          bins higher_half = {[8:14]};
          bins b_data_max = {15};
          bins b_data_default = default;
      }

      OP_CP: coverpoint alu_tr_el.op {
          bins op_add = {2'b00};
          bins op_xor  = {2'b01};
          bins op_and = {2'b10};
          bins op_or = {2'b11};
      }

      OUT_CP: coverpoint alu_tr_el.out;
      C_CP: coverpoint alu_tr_el.c;

      // cross coverage
      BOUNDARY_ADD_C: cross A_CP, B_CP, OP_CP {
          bins ADD_BOUNDARY = binsof(A_CP) intersect {15} &&
                              binsof(B_CP) intersect {15} &&
```

```
                                binsof(OP_CP) intersect {2'b00};
            option.cross_auto_bin_max = 0;
        }

        CARRY_C: cross A_CP, B_CP, OP_CP {
            bins CARRY_SIG = binsof(A_CP.higher_half) &&
                                binsof(B_CP.higher_half) &&
                                binsof(OP_CP) intersect {2'b00};
            option.cross_auto_bin_max = 0;
        }
    endgroup
```

Achieved:



Challenges and Lessons Learned

Alignment of driver and monitor in combinational dut testing

Time	Action
t	Driver updates internal signals
t → t+Δ	Continuous assign updates interface
t+1	Monitor samples stable inputs
t+1	Scoreboard checks same transaction