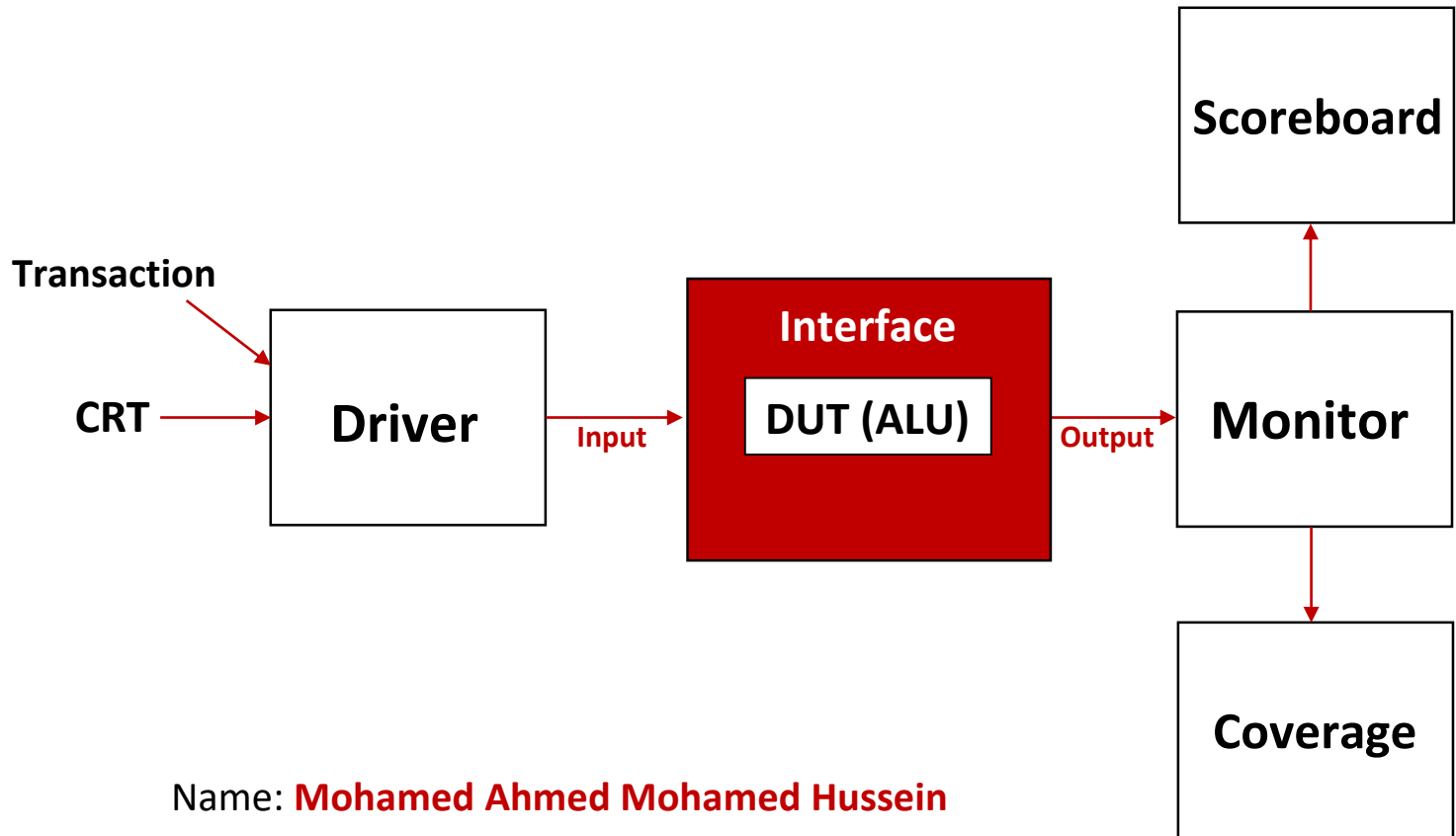


Verification Task 1



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

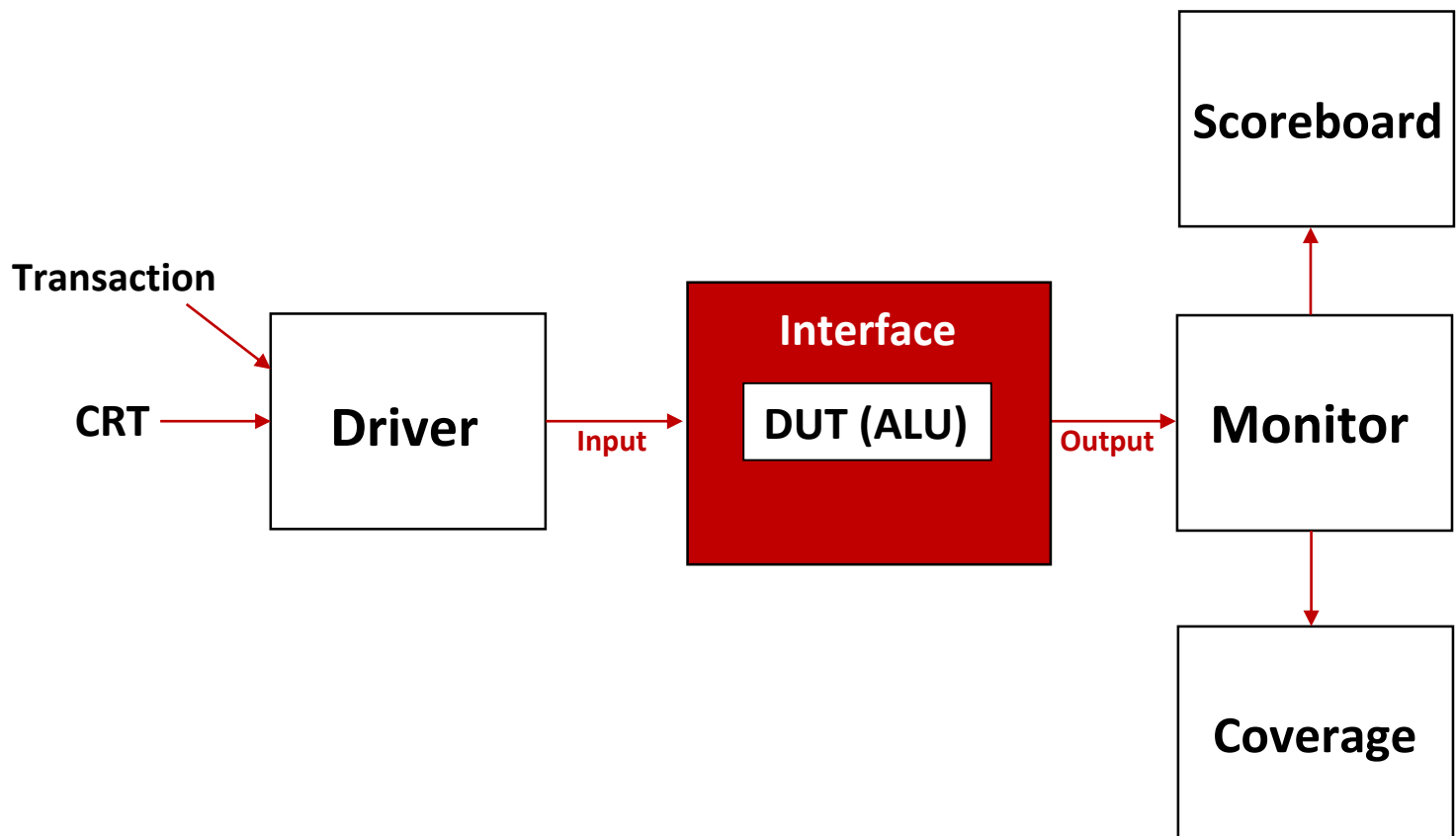
Detect arithmetic, logic, and flag-related bugs

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})	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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)	<ul style="list-style-type: none"> 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 <ul style="list-style-type: none"> ADD incorrect XOR incorrect AND incorrect Carry randomly asserted <p>And this scenario happens multiple of times</p>	Use arithmetic and logical operators: + ^ &
4	Carry bit is meaningless, Carry should only be valid for ADD	<ul style="list-style-type: none"> Wrong carry behavior 	De-assert carry signal in all logical operations and make it valid in add operation

Coverage Analysis

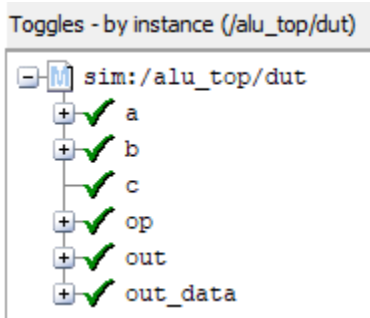
- Code Coverage:
 - Statement:

Statements - by instance (/alu_top/dut)		Statements - by instance (/alu_top/dut)	
DUT.sv			
13	assign a = aluif.a;	1004	10'b1111101000: out_data = 5'b010100;
14	assign b = aluif.b;	1005	10'b11111010101: out_data = 5'b01010;
15	assign op = aluif.op;	1006	10'b11111010110: out_data = 5'b111110;
20	assign {c, out} = out_data;	1007	10'b11111010111: out_data = 5'b00010;
22	always @(a, b, op) begin	1008	10'b11111011000: out_data = 5'b10101;
24	10'b0000000000: out_data = 5'b01010;	1009	10'b11111011001: out_data = 5'b01001;
25	10'b0000000001: out_data = 5'b00111;	1010	10'b11111011010: out_data = 5'b00110;
26	10'b0000000010: out_data = 5'b01010;	1011	10'b11111011011: out_data = 5'b01111;
27	10'b0000000011: out_data = 5'b00000;	1012	10'b11111011100: out_data = 5'b10110;
28	10'b0000000100: out_data = 5'b00111;	1013	10'b11111011101: out_data = 5'b01000;
29	10'b0000000101: out_data = 5'b01110;	1014	10'b11111011110: out_data = 5'b00111;
30	10'b0000000110: out_data = 5'b00000;	1015	10'b11111011111: out_data = 5'b00111;
31	10'b0000000111: out_data = 5'b00110;	1016	10'b11111000000: out_data = 5'b10111;
32	10'b0000001000: out_data = 5'b00010;	1017	10'b11111000001: out_data = 5'b10010;
33	10'b0000001001: out_data = 5'b00010;	1018	10'b11111000010: out_data = 5'b01000;
34	10'b0000001010: out_data = 5'b00000;	1019	10'b11111000011: out_data = 5'b10000;
35	10'b0000001011: out_data = 5'b00010;	1020	10'b11111000100: out_data = 5'b11000;
36	10'b0000001100: out_data = 5'b00011;	1021	10'b11111000101: out_data = 5'b00110;
37	10'b0000001101: out_data = 5'b00011;	1022	10'b11111000110: out_data = 5'b01001;
38	10'b0000001110: out_data = 5'b10001;	1023	10'b11111000111: out_data = 5'b01111;
39	10'b0000001111: out_data = 5'b00011;	1024	10'b11111010000: out_data = 5'b01011;
40	10'b0000010000: out_data = 5'b00100;	1025	10'b11111010001: out_data = 5'b01101;
41	10'b0000010001: out_data = 5'b00100;	1026	10'b11111010010: out_data = 5'b10000;
42	10'b0000010010: out_data = 5'b00000;	1027	10'b11111010011: out_data = 5'b10101;
43	10'b0000010011: out_data = 5'b01000;	1028	10'b11111010100: out_data = 5'b01100;
44	10'b0000010100: out_data = 5'b00101;	1029	10'b11111010101: out_data = 5'b00100;
45	10'b0000010101: out_data = 5'b00101;	1030	10'b11111010110: out_data = 5'b01011;
46	10'b0000010110: out_data = 5'b00000;	1031	10'b11111010111: out_data = 5'b10010;
47	10'b0000010111: out_data = 5'b00111;	1032	10'b11111000000: out_data = 5'b11011;
48	10'b0000011000: out_data = 5'b00110;	1033	10'b11111000001: out_data = 5'b10010;
49	10'b0000011001: out_data = 5'b00110;	1034	10'b11111000010: out_data = 5'b10111;
50	10'b0000011010: out_data = 5'b00000;	1035	10'b11111000011: out_data = 5'b01111;
51	10'b0000011011: out_data = 5'b00110;	1036	10'b11111010000: out_data = 5'b11100;
52	10'b0000011100: out_data = 5'b00111;	1037	10'b11111010001: out_data = 5'b11011;
53	10'b0000011101: out_data = 5'b11111;	1038	10'b11111010010: out_data = 5'b11001;
54	10'b0000011110: out_data = 5'b00000;	1039	10'b11111010011: out_data = 5'b00001;
55	10'b0000011111: out_data = 5'b11000;	1040	10'b11111010000: out_data = 5'b11101;
56	10'b0000100000: out_data = 5'b01000;	1041	10'b11111010001: out_data = 5'b11011;
57	10'b0000100001: out_data = 5'b01100;	1042	10'b11111010010: out_data = 5'b01110;
58	10'b0000100010: out_data = 5'b00000;	1043	10'b11111010011: out_data = 5'b01111;
59	10'b0000100011: out_data = 5'b01000;	1044	10'b11111010000: out_data = 5'b11110;
60	10'b0000100100: out_data = 5'b01001;	1045	10'b11111010001: out_data = 5'b00000;
		1046	10'b11111010010: out_data = 5'b01111;
		1047	10'b11111010011: out_data = 5'b01111;

- 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:

/alu_coverage_pkg/alu_coverage	100.00%				
TYPE alu_Cross_Group	100.00%	100	100.00...		auto(1)
CVP alu_Cross_Group::A_CP	100.00%	100	100.00...		
bin a_data_0	1028	1	100.00...		
bin higher_half	6127	1	100.00...		
bin a_data_max	469	1	100.00...		
default bin a_data_default	7376	-	-		
CVP alu_Cross_Group::B_CP	100.00%	100	100.00...		
bin b_data_0	953	1	100.00...		
bin higher_half	6876	1	100.00...		
bin b_data_max	424	1	100.00...		
default bin b_data_default	6747	-	-		
CVP alu_Cross_Group::OP_CP	100.00%	100	100.00...		
bin op_add	3693	1	100.00...		
bin op_xor	3714	1	100.00...		
bin op_and	3837	1	100.00...		
bin op_or	3756	1	100.00...		
CVP alu_Cross_Group::OUT_CP	100.00%	100	100.00...		
bin auto[0]	1385	1	100.00...		
bin auto[1]	830	1	100.00...		
bin auto[2]	1005	1	100.00...		
bin auto[3]	920	1	100.00...		
bin auto[4]	887	1	100.00...		
bin auto[5]	656	1	100.00...		
bin auto[6]	1000	1	100.00...		
bin auto[7]	832	1	100.00...		
bin auto[8]	1035	1	100.00...		
bin auto[9]	827	1	100.00...		
bin auto[10]	840	1	100.00...		
bin auto[11]	829	1	100.00...		
bin auto[12]	785	1	100.00...		
bin auto[13]	916	1	100.00...		
bin auto[14]	938	1	100.00...		
bin auto[15]	1315	1	100.00...		
CVP alu_Cross_Group::C_CP	100.00%	100	100.00...		
bin auto[0]	11521	1	100.00...		
bin auto[1]	3479	1	100.00...		
CROSS alu_Cross_Group::BOUNDARY_ADD_C	100.00%	100	100.00...		
bin ADD_BOUNDARY	6	1	100.00...		
CROSS alu_Cross_Group::CARRY_C	100.00%	100	100.00...		
bin CARRY_SIG	668	1	100.00...		

Challenges and Lessons Learned

Alignment of driver and monitor in combinational dut testing

Time	Action
t	Driver updates internal signals
$t \rightarrow t+\Delta$	Continuous assign updates interface
t+1	Monitor samples stable inputs
t+1	Scoreboard checks same transaction