Lab 6
Sampling, Analyzing and Improving the ALU Coverage

Module ID : CX-301

Design Verification

Instructor : Dr. Abid Rafique

Version 1.2

Information contained within this document is for the sole readership of the recipient, without authorization of distribution to individuals and / or corporations without prior notification and approval.

Copyright 2024 Ⓒ ChiPXPRT Pvt. Ltd.

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	Jan, 2024	Qamar Moavia
1.1	Modified with new exercises	feb, 2025	Qamar Moavia

Table of Contents

Objectives

By the end of this lab, students will be able to answer the following questions:

- What are the advantages of using interfaces in SystemVerilog?
- How do interfaces simplify communication between modules?
- How can arrays be helpful in writing self checking testbenches?

Tools

- SystemVerilog
- Synopsys VCS

Instructions for Lab Tasks

The required files for this lab can be found on the

```
shared_folder/CX-301-DesignVerification/Labs/Lab6
```

The submission must follow the hierarchy below, with the folder name and the file names exactly as listed below.

```
./Lab6/
        ├── Task1/
        │   ├── alu.sv
        │   ├── randtrans.sv
        │   ├── alu_test.sv
        │   ├── top.sv
        │   ├── filelist.f
        │   ├── // screenshots of outputs/waveforms
        ├── Task2/
        │   ├── alu.sv
        │   ├── randtrans.sv
        │   ├── alu_test.sv
        │   ├── top.sv
        │   ├── filelist.f
        │   ├── // screenshots of outputs/waveforms
```

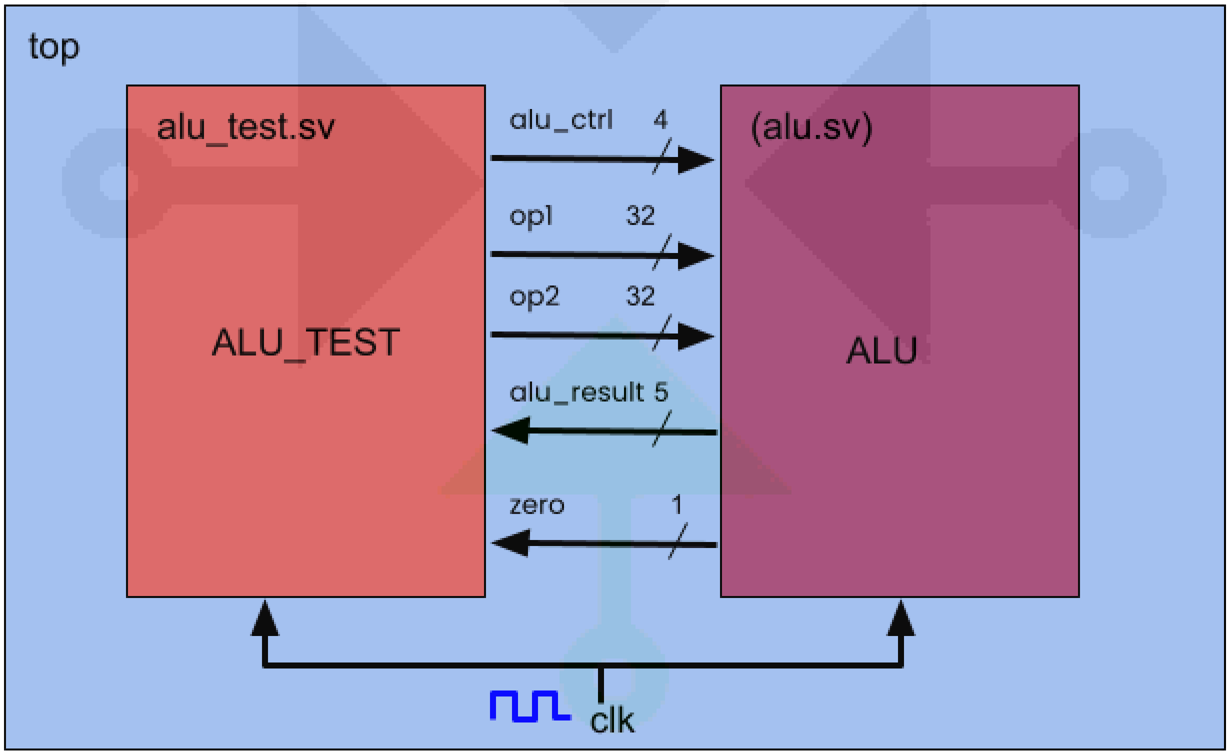
Along with that you also need to upload your solution on the github as well, and share the link.

Lab Overview

In this lab, you are provided with a basic testbench of ALU that we are using in our RISC-V processor. You will add coverage collectors in the ALU testbench to collect the coverage on alu inputs as well as outputs. Following files are shared in the shared folder:

alu.sv	(contains the DUT)
alu_test.sv	(contains the ALU TEST module)
randtrans.sv	(contains the transaction class)
top.sv	(connect the DUT and the TEST module, also generates clk)
filelist.sv	(contains the list of all the files)

Copy those into your working directory.



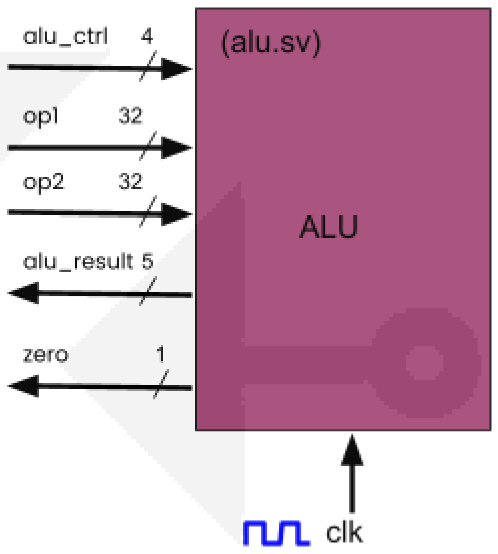
Read the specification first and then follow the instructions in the lab section to add the coverage collector in the `alu_test.sv`

Copyright 2024 © ChiPXPRT Pvt. Ltd.

ALU Specification

- op1, op2 and alu_result are all 32-bit logic vectors. alu_ctrl is a 4-bit logic vector for the ALU operations.
- zero is a single bit, asynchronous output with the value of 1 when out equals 0 Otherwise, zero is 0.
- alu_result is synchronized to the positive edge of clk and takes the following values depending on opcode.

Opcode	Encoding	Output
ADD	4'b0000	op1 + op2
SUB	4'b1000	op1 - op2
SLL	4'b0001	op1 << op2[4:0]
SLT	4'b0010	set if op1 < op2
SLTU	4'b0011	unsigned SLT
XOR	4'b0100	op1 ^ op2
SRL	4'b0101	op1 >> op2[4:0]
OR	4'b0110	op1 | op2
AND	4'b0111	op1 & op2
SRA	4'b1101	op1 >>> op2[4:0]



TASK 1 : Adding Covergroup in the ALU Testbench

Modify the alu_test module to add the coverage collectors for the ALU data inputs op1, op2 as well as the alu_ctrl.

Adding Covergroup

Working in the Lab6/Task1 Directory perform the following:

1. Declare a covergroup in the alu_test module. Set the sampling event to be the positive edge of the clock.
2. Declare the coverpoints for the op1, op2 with the explicit bins as follows:
    a. Vector bins covering the low range values: `(0x00000000, 0x000000FF)`
    b. One scalar bin covering the high range values: `(0xFFFFFF00, 0xFFFFFFFF)`

   c. One scalar default bin for all other values.
3. Declare a coverpoint for `alu_ctrl` using automatic bins.
4. Now, we will have to enable the coverage collection before compiling and running the simulation. To enable the coverage options following flags are to be added in the compile command or the filelist.f

   ```
   -cm line+tgl+cond+branch+fsm+assert
   ```

   Note: These flags are already added in the filelist.f that is provided to you.
5. Since the list of all the files as well as all the flags are already in the `filelist.f`, we can compile the simulation by simply running the following command:

   ```
   vcs -full64 -f filelist.f -o simv
   ```

   Note: The flag `-full64` is not supported in the -f option, so we have to add that in the command line. Other flags like `-sverilog`, `-timescale` etc are currently in the `filelist.f` to make the command more simple.

6. Once, compiled, run the simulation with the following flags:

   ```
   ./simv -cm line+cond+fsm+branch+tgl
   ```

   You should see the following message if the coverage was collected successfully.

   VCS Coverage Metrics: during simulation line, cond, FSM, branch, tgl was monitored.

Analyzing the Coverage using Synopsys DVE

Synopsys provides the **Design Vision Environment (DVE)**, a graphical user interface for debugging and analyzing simulation results. DVE enables users to visualize various types of coverage, such as code coverage (line, toggle, condition, FSM, and branch) and functional coverage (covergroups defined in the testbench).

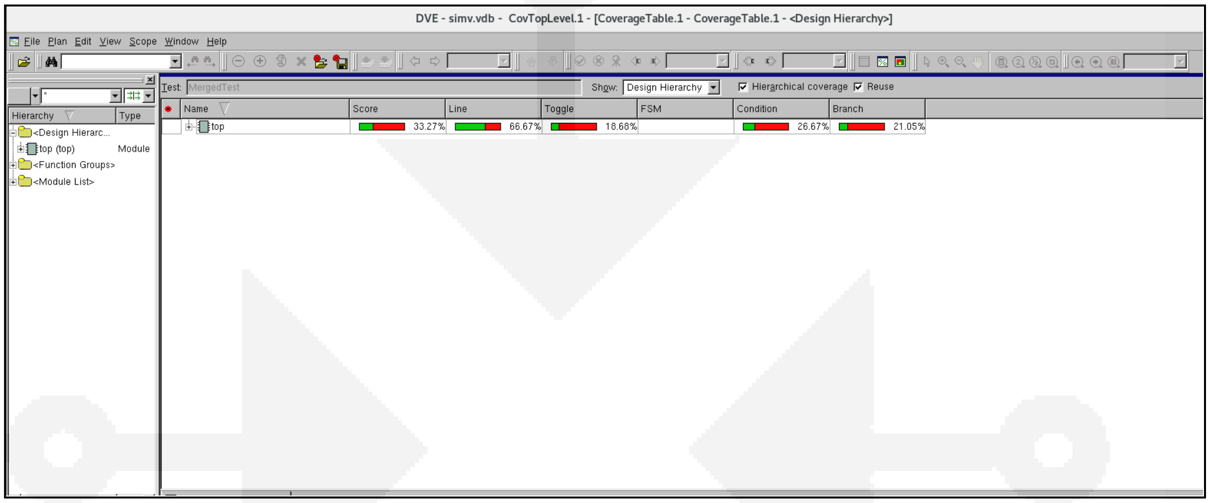
Perform the following steps to launch DVE and analyze the coverage:

7. Invoke DVE by running the following command after simulation:

```
dve -cov -dir simv.vdb &
```

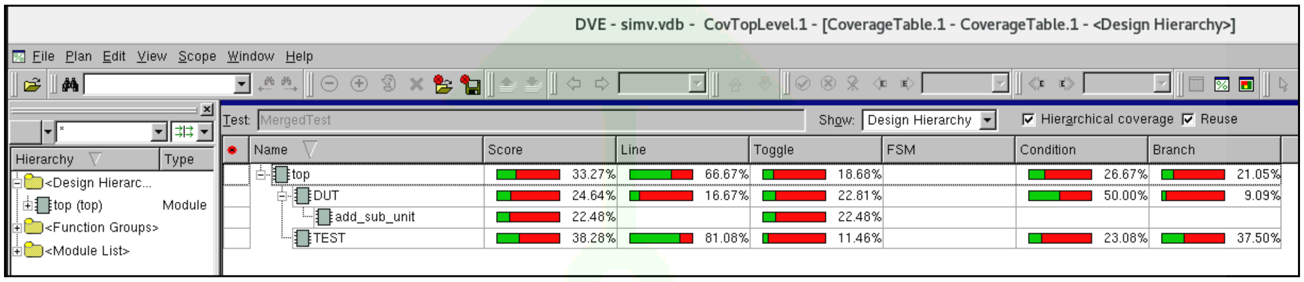
   This command opens the DVE GUI with coverage data loaded from the `simv.vdb` directory.

8. This will open the Synopsys DVE as shown below



Name	Score	Line	Toggle	FSM	Condition	Branch
top	33.27%	66.67%	18.68%		26.67%	21.05%

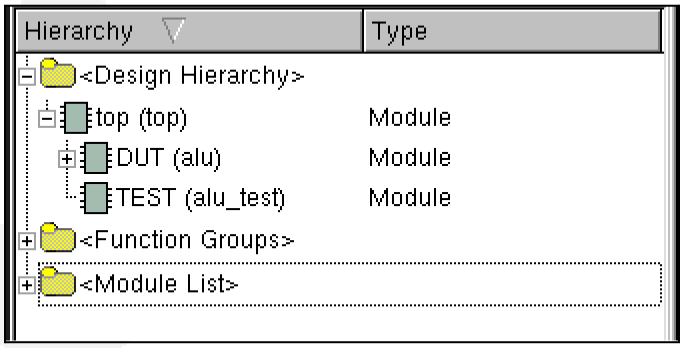
9. If you click the plus sign in the top module in the right window, you will see the **code coverage** results as shown below.



Name	Score	Line	Toggle	FSM	Condition	Branch
top	33.27%	66.67%	18.68%		26.67%	21.05%
DUT	24.64%	16.67%	22.81%		50.00%	9.09%
add_sub_unit	22.48%		22.48%			
TEST	38.28%	81.08%	11.46%		23.08%	37.50%

Analyzing the Code Coverage

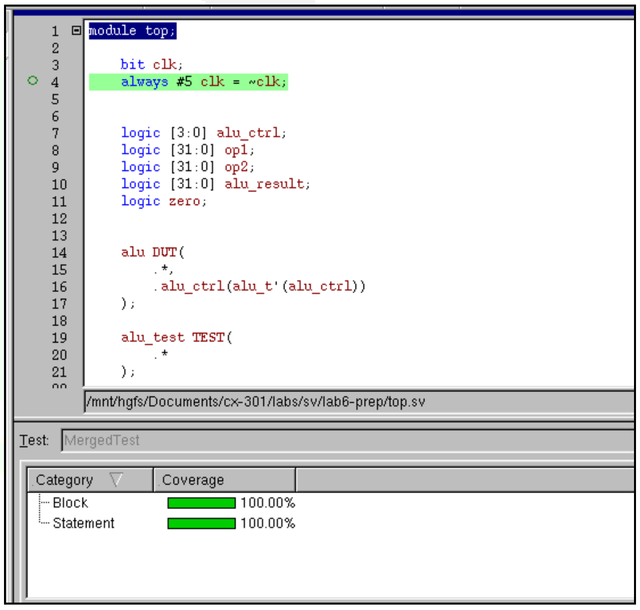
10. On the right side, in the Hierarchy tab, you can see the following tabs:
    a. Design Hierarchy
    b. Functional Groups
    c. Module List



Hierarchy	Type
<Design Hierarchy>	
top (top)	Module
DUT (alu)	Module
TEST (alu_test)	Module
<Function Groups>	
<Module List>	

11. In the Design Hierarchy, you can check the code coverage of any of the modules (`top`, `alu_test`, `alu` etc). Expand the Design Hierarchy and click on the top and you will see the `100%` coverage as you can see here on the right side.

    There is only one procedural block in the top module which is highlighted in green. That's why we get `100%` code coverage for the `top` module.



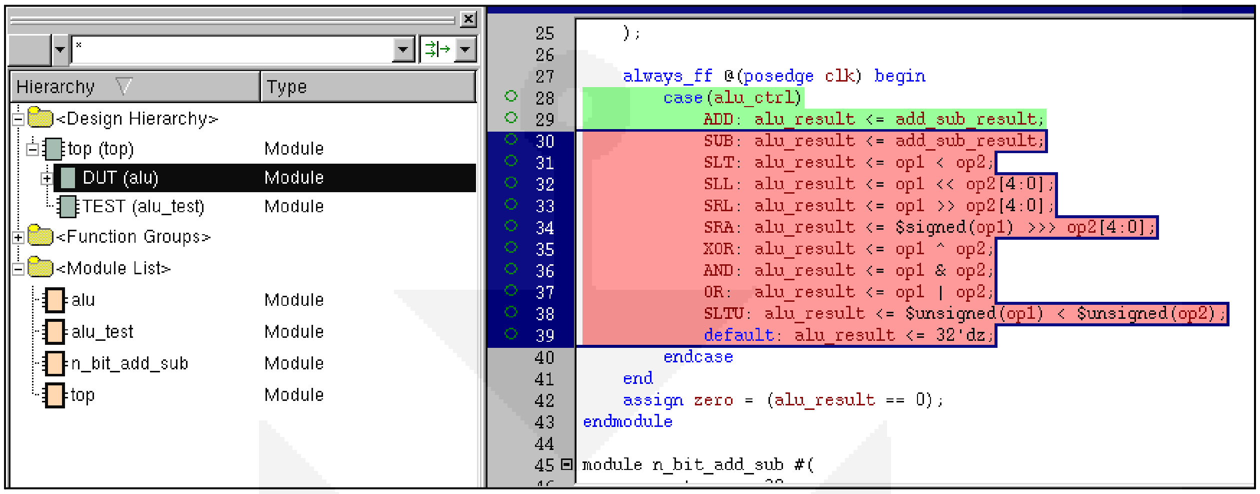
```
module top;

    bit clk;
    always #5 clk = ~clk;


    logic [3:0] alu_ctrl;
    logic [31:0] op1;
    logic [31:0] op2;
    logic [31:0] alu_result;
    logic zero;


    alu DUT(
        .*,
        .alu_ctrl(alu_t'(alu_ctrl))
    );

    alu_test TEST(
        .*
    );
```

/mnt/hgfs/Documents/cx-301/labs/sv/lab6-prep/top.sv

Test: MergedTest

Category	Coverage
Block	100.00%
Statement	100.00%

12. In the Design Hierarchy, now check the code coverage of the `DUT (alu)`. The code coverage for the `alu` module will be very low because the test cases only generate input transactions with `alu_ctrl = 4'b0000` (ADD).



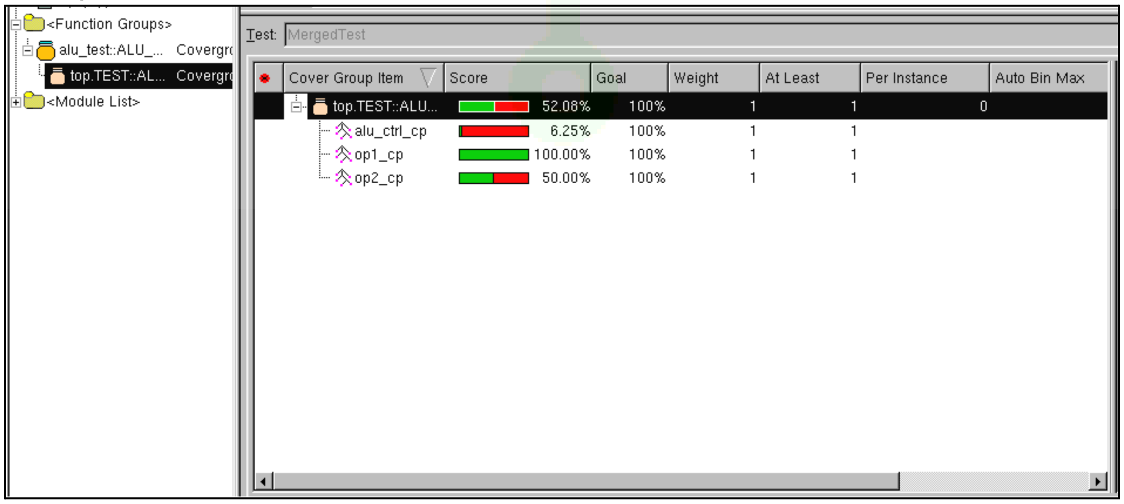
Hierarchy	Type
<Design Hierarchy>	
top (top)	Module
DUT (alu)	Module
TEST (alu_test)	Module
<Function Groups>	
<Module List>	
alu	Module
alu_test	Module
n_bit_add_sub	Module
top	Module

```
    );

    always_ff @(posedge clk) begin
        case(alu_ctrl)
            ADD: alu_result <= add_sub_result;
            SUB: alu_result <= add_sub_result;
            SLT: alu_result <= op1 < op2;
            SLL: alu_result <= op1 << op2[4:0];
            SRL: alu_result <= op1 >> op2[4:0];
            SRA: alu_result <= $signed(op1) >>> op2[4:0];
            XOR: alu_result <= op1 ^ op2;
            AND: alu_result <= op1 & op2;
            OR:  alu_result <= op1 | op2;
            SLTU: alu_result <= $unsigned(op1) < $unsigned(op2);
            default: alu_result <= 32'dz;
        endcase
    end
    assign zero = (alu_result == 0);
endmodule

module n_bit_add_sub #(
```

13. If you don't care about the hierarchy and want to see all the modules at once. You can do so by expanding the Module List. There you can find all the modules of the current simulation environment.

Analyzing the Functional Coverage

14. Now, let's move on to analyze the Functional Coverage. You can do so by expanding the Functional Groups tab and clicking on the Covergroup whose coverage you want to check. In case you have multiple covergroups all covergroups can be seen here in the Function Groups tab.



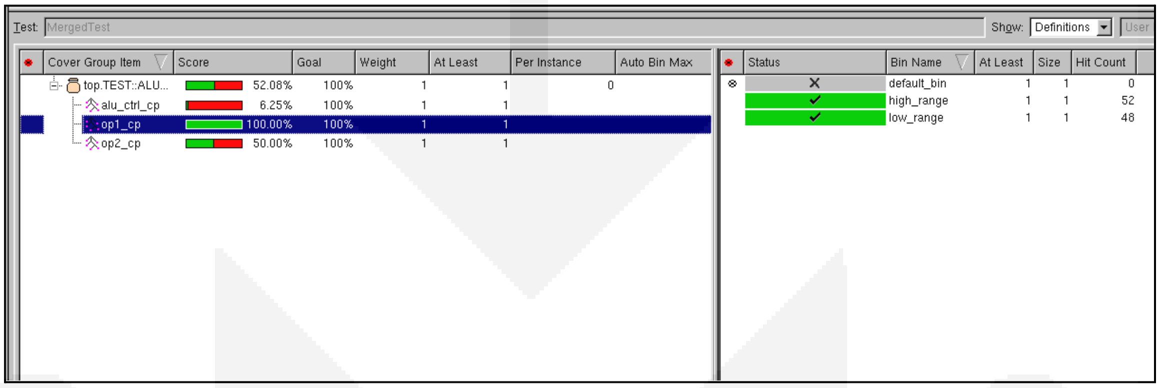
<Function Groups>
alu_test::ALU_... Covergr
top.TEST::AL... Covergr
<Module List>

Test: MergedTest

Cover Group Item	Score	Goal	Weight	At Least	Per Instance	Auto Bin Max
top.TEST::ALU...	52.08%	100%	1	1	0	
alu_ctrl_cp	6.25%	100%	1	1		
op1_cp	100.00%	100%	1	1		
op2_cp	50.00%	100%	1	1		

There you can see the coverage percentage of the overall group as well as all its coverpoints.

15. You can see the detail of each coverpoint as well by clicking on it. It will show the bins of that coverpoint on the right side as you can see below:



Test: MergedTest   Show: Definitions   User

Cover Group Item	Score	Goal	Weight	At Least	Per Instance	Auto Bin Max
top.TEST::ALU...	52.08%	100%	1	1	0	
alu_ctrl_cp	6.25%	100%	1	1		
op1_cp	100.00%	100%	1	1		
op2_cp	50.00%	100%	1	1		

Status	Bin Name	At Least	Size	Hit Count
✗	default_bin	1	1	0
✓	high_range	1	1	52
✓	low_range	1	1	48

16. This coverage specifies the coverage of each coverpoint as well as the overall coverage of the whole covergroup. Currently the weight of each coverpoint is set to 1. So, all the coverpoints have the same weight in the final coverage.
17. Now the issue here is that we are only achieving 52.08% coverage for the covergroup. But, our goal was to achieve 100% coverage.

    We need to modify our testcase to achieve 100% coverage

Improving the Functional Coverage

18. Modify the `alu_test.sv` and add more test cases to achieve 100% functional coverage.

Sampling the Coverage Correctly

Currently, coverage is sampled on every positive edge of the clock, which works well for our current ALU design. However, in modern designs one operation may take multiple cycles. Their sampling at clock edge approach may lead to redundant coverage sampling, resulting in misleading results.

To improve accuracy:

- Use `start` and `stop` covergroup methods to collect coverage only for specific operations, such as random data tests. This helps us to better control the coverage sampling.
- Manually trigger sampling using the `sample()` method once per each operation instead of relying on the clock edge only

TASK 2 : Cross Coverage

Create another directory Task2 in the Lab6 directory. Copy the files from Task1 into the Task2 directory. Working in the Task2 directory perform the following:

1. Create a coverpoint for the alu_result with four bins for value zero, low_range, high_range and default values.
2. Create cross-coverage for `alu_ctrl` and the high/low bins of the `op1` and `op2`. Simulate and Check the cross coverage results.
3. Create cross-coverage for `alu_ctrl` with the with the `alu_result`.
4. Try to reduce the number of cross coverage bins by excluding the irrelevant combinations as defined by the opcode table above.
5. Modify the stimulus to cover some or all of your coverage holes.

Good Luck 🙂