

ECE 745: ASIC VERIFICATION

LAB 4: FUNCTIONAL COVERAGE

Introduction

The aim of this laboratory exercise is to get you acquainted with some of the basic coverage features that Questa supports. We will deal with the creation of **covergroups**, **coverpoints** and **cross** coverage options to measure the completeness of checking arithmetic operations.

Again, please note that a basic description of the Execute block that will be used as the DUT (This remains unchanged from the previous laboratory exercises). A full system description can be found on the course website with this lab posting

In this lab we are going to add the coverage constructs in the scoreboard class. It must be noted that coverage can also be added to other parts (the output to memory perhaps) or created as a separate class in itself. You must use coverage for the following:

1. To ensure that the different corners of interest in the DUT, such as special combinations of inputs or types of data (all 0's, all 1's, alternating 1's and 0's etc) are processed.
2. To check the interaction between blocks in the interface between blocks. The coverage constructs would probe and analyze coverage for the different blocks/interfaces of interest within the DUT. It would also attempt to ensure that there is sufficient range in the type of interaction between blocks using cross coverage.

Using coverage provides a simple way to make sure that the Constrained Random Testing methodology you have adopted is actually exercising the corners of the design that of interest to you. We also provide information regarding the storage of the coverage information for a given run using Universal Coverage Database (ucdb) files. Multiple stored ucdb files from different coverage runs can also be combined to create a final ucdb file. The process of doing this combination is also explained. The updated Scoreboard file with coverage incorporated can be downloaded with this lab posting. It is called Scoreboard_coverage.sv

In addition to the above file, please download the two versions of the packet file which represent two different types of data sources for the purposes of Constrained Random Testing. The two files you need are **Packet.sv** & **Packet2.sv**

Also, please ensure you have the rest of the files needed and provided on the course website in the lab posting. Needed files are **Top.v**, **data_defs.v** **ALU.vp**, **Ex_Preproc.vp**, **Arith_ALU.vp**, **Shift_ALU.vp**

Some fundamental requirements for enabling coverage analysis needs to be noted within the Scoreboard_coverage.sv file.

1. The covergroup declaration: In the code you will see the declaration of the covergroup Arith_Cov_Ver1. In this covergroup we blindly cover all possibilities of some of the inputs to the DUT. We use the fields of the pkt_sent packet to determine our coverage value. We do this to be able to determine how comprehensive and useful the inputs to the DUT were. The Arith_Cov_Ver1 covergroup has a lot of overkill because it is almost impossible to meet full coverage. We need to send in 2^{32} values for each data type to get full coverage. We also have the declaration of Arith_Cov_Ver2 and Arith_Cov_Ver3 in the scoreboard. These are smarter measures of completeness of the data being sent in and will be discussed at greater length in the coming sections.

```
#####d#####
covergroup Arith_Cov_Ver1;
    coverpoint pkt_sent.imm;
    coverpoint pkt_sent.src1;
    coverpoint pkt_sent.src2;
    coverpoint pkt_sent.opselect_gen;
    coverpoint pkt_sent.operation_gen;
endgroup
#####
```

2. The allocation of space: Within function Scoreboard::new(...) you will see the allocation of space for the covergroup(s) by performing
Arith_Cov_Ver1 = new();
Arith_Cov_Ver2 = new();
Arith_Cov_Ver3 = new();
3. The sampling of the values to determine coverage. In this case we are going to sample at every clock edge given that we are going to see changes in the input at every clock edge. As things get more complex in terms of timing and you need to cater to stalls and buffering and such, you would be well advised to sample based off of a condition or an event. In this case, the sampling of coverage for Arith_Cov_Ver1 is done by executing Arith_Cov_Ver1.sample(). This causes the revaluation of the coverage measures and determination of newer values for the coverpoint and cross under the covergroup being sampled.
4. Get a hard coverage number for a covergroup: To give the progress of coverage a numeric value we can use get_coverage(). The number represents the percentage of the total conditions specified (within coverpoints, crosses and their bins) that have been met for the current simulation. This returns a real value which requires us to declare a real variable for storage of the coverage information. Ideally, there should be a steady increase in the coverage number towards 100. In this case we measure the coverage for Arith_Cov_Ver1 as
coverage_value1 = Arith_Cov_Ver1.get_coverage();

Coverpoint Arith_Cov_Ver2:

This covergroup assigns labels to many of the coverpoints which proves to be good for debugging when the number of coverpoints goes up in the analysis. Examples for this are: `src1_cov` and `src2_cov`. Moreover, this covergroup provides an example coverpoint called `opselect_cov2` for the case where we only look at the valid inputs `opselect` values into the system. This provides for a more useful analysis of the results with respect to a coverage goal. The `opselect_cov2` is declared as

```
#####
opselect_cov2: coverpoint pkt_sent.opselect_gen {
    bins shift = {0};
    bins arith = {1};
    bins mem = {[4:5]};
}
#####
```

The above construct will result in the creation of bins named `shift`, `arith` and `mem` within the coverpoint. These bins will be displayable within the Questa GUI. We also have two cross coverage points declared as `cx_opsel_opn` and `cx_opselcov_opn`. Here, `cx_opselcov_opn` deals with only the valid input `opselects` for coverage analysis. The aim of `cx_opselcov_opn` is to check if all possible operation values are sent in for each feasible `opselect`. Note that we are still suffering from overkill here given that not all operation values are valid for a given `opselect`.

Coverpoint Arith_Cov_Ver3:

This covergroup improves upon the second version. The coverage constructs determine if interesting data values- all 1's, all 0's, alternating 1's and 0's, have been used. The analysis is performed under improved `src(1/2)_cov` coverpoints. Of particular importance are the bins that use the `iff` construct to determine if a positive and/or negative number has been encountered.

```
#####
src1_cov: coverpoint pkt_sent.src1 {
    bins zero = {0};
    bins allfs = {32'hfffffff};
    bins special1 = {32'h55555555};
    bins special2 = {32'haaaaaaaa};
    bins positive = {[0:1]} iff(pkt_sent.src1[31] == 1'b0);
    bins negative = {[0:1]} iff(pkt_sent.src1[31] == 1'b1);
    //wildcard bins positive = {32'b0????????????????????????????????};
    //wildcard bins negative = {32'b1????????????????????????????????};
}
#####
```

It must also be noted that a multi-dimensional cross (`cross src1_cov, src2_cov, opn_cov;`) is also feasible. Over and above this, we also declare the `addition_cov` cross which attempts to find whether corner cases for additions have occurred. For example, the bin `addfs` determines whether the addition of two `32'hffff_ffff` values has

occurred. This is achieved by accessing specific bins of previously declared coverpoints using the `binsof` and `intersects` constructs.

```
#####
addition_cov: cross src1_cov, src2_cov, opselect_cov, opn_cov {
}
    bins addfs =    binsof(src1_cov.allfs) &&
                   binsof(src2_cov.allfs) && s
                   binsof(opselect_cov.arith) &&
                   binsof(opn_cov) intersect {0};

    bins addspec = binsof(src1_cov.special1) &&
                   binsof(src2_cov.special2) &&
                   binsof(opselect_cov.arith) &&
                   binsof(opn_cov) intersect {0};

    bins addpos =  binsof(src1_cov.positive) &&
                   binsof(src2_cov.positive) &&
                   binsof(opselect_cov.arith) &&
                   binsof(opn_cov) intersect {0};

    bins addneg =  binsof(src1_cov.negative) &&
                   binsof(src2_cov.negative) &&
                   binsof(opselect_cov.arith) &&
                   binsof(opn_cov) intersect {0};
}
#####
```

VIEWING COVERAGE IN QUESTA:

The initialization steps (`setenv MODELSIM modelsim.ini ; vlib mti_lib`) remain the same as the previous labs. Compile the code provided using

```
> vlog *.v *.vp
> // ALL OF THE BELOW SHOULD BE ON ONE LINE
> vlog -mfcu -sv data_defs.v Packet.sv Driver.sv OutputPacket.sv
Receiver.sv      Scoreboard_coverage.sv      Generator.sv      Execute.tb.sv
Execute.test_top.sv Execute.if.sv
```

We can invoke the GUI with coverage for the relevant top level instance using

```
vsim -coverage -novopt Execute_test_top &
```

The “**coverage**” should invoke the GUI in the form shown in Figure. 1 using the coverage palette. Note that the layout of this window might look different for different users. The most important requirement of this layout is the presence of the **Analysis window**. In many cases we would also see the missed coverage window and the instance coverage window. We shall deal with just the analysis window for this laboratory exercise.

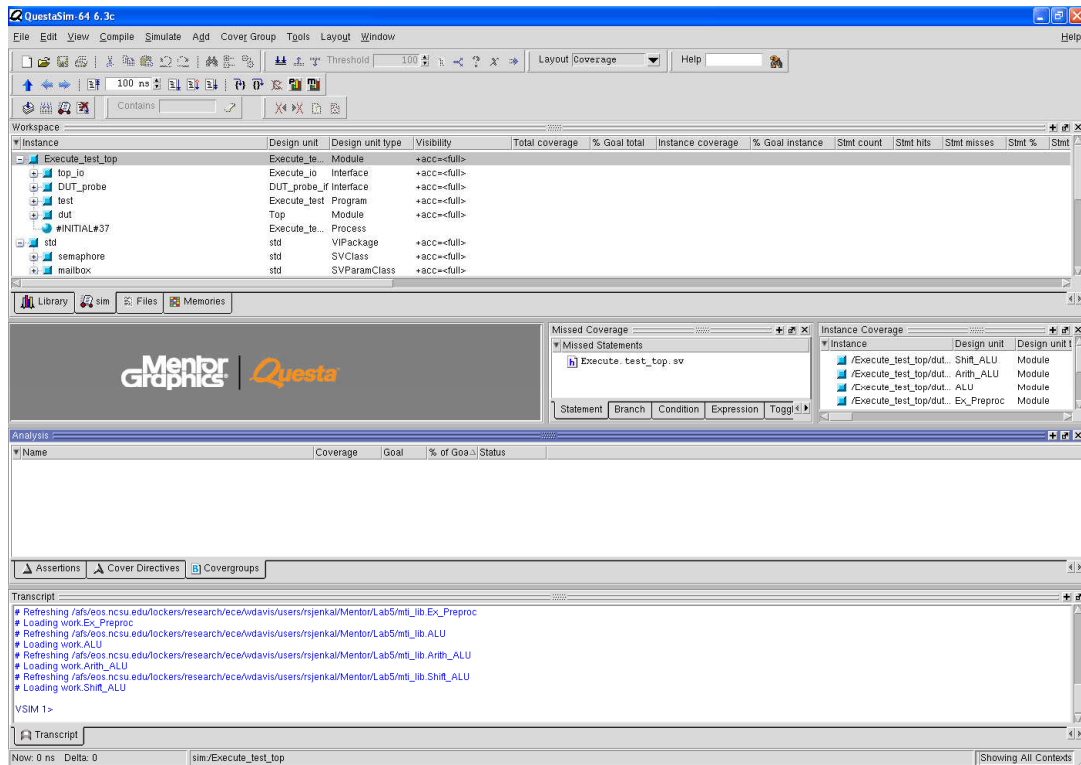


Figure 1: GUI after invocation with coverage option

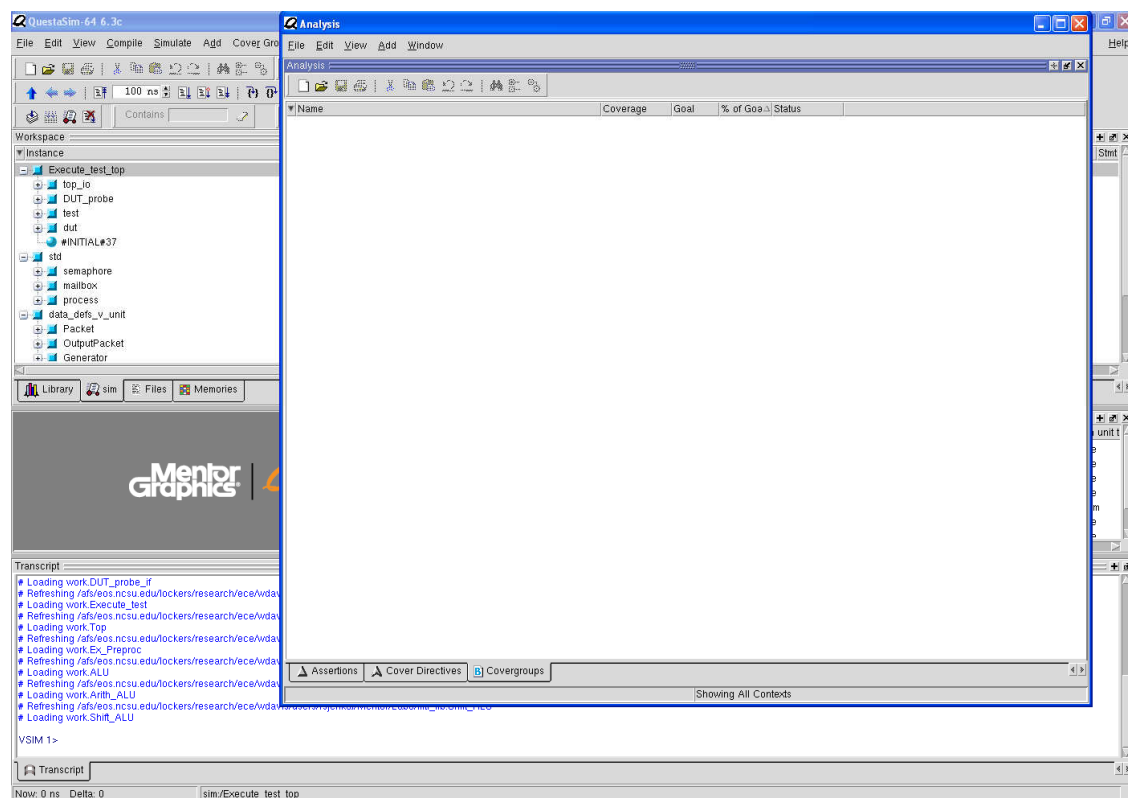



Figure 2: GUI after undocking the Analysis window

It would be best if the Analysis window is undocked using the  button to make it easier to view the coverage graphs. Note that the different options for coverage viewing can always be added and removed by doing *view -> coverage -> <options>*. The undocking of the Analysis window would give us a display of the form shown in Figure 2. Note the presence of assertions, cover directives and covergroups sub- windows. We are only dealing with the covergroups option here. The aim of this window is to provide a graphical representation of the progress in completing the coverage requirements of different covergroups. Initially this window is empty and will only get populated on running a simulation.

We can now run the simulation for, say 2000 ns. After the run, the task now is to obtain the coverage information for the covergroups written within the Scoreboard unit. Generally, the design unit with coverage information in it would already have populated the analysis window. If not, the tool needs to be forced into the analysis mode. For this, we need to perform *view -> coverage -> covergroups* on the main questasim window.

The analysis window would have the color coded representation of the coverage goals achieved till the current simulation time. On expanding the tabs within the covergroups window, expanding the Arith_Cov_Ver3 covergroup, the opselect_cov coverpoint and the addition_cov cross, the analysis window takes the form shown in Figure 3.

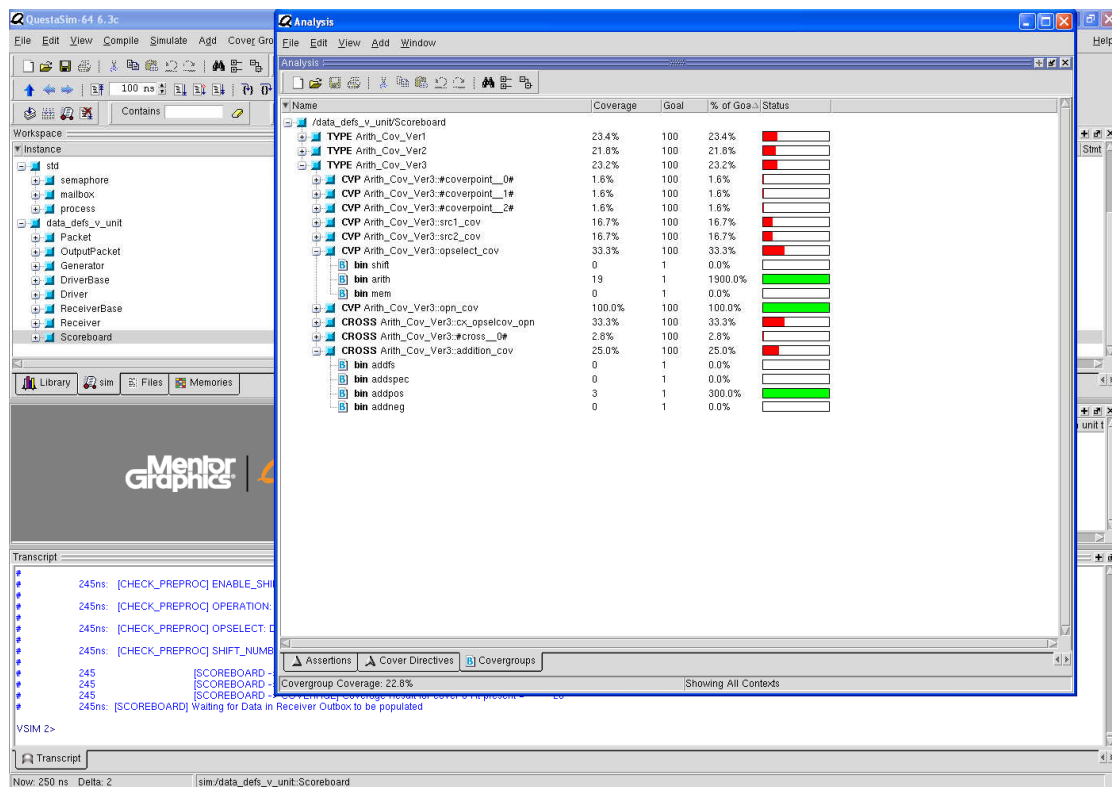


Figure 3: Coverage information using Packet.sv

As can be seen the `opselect_cov` construct which records the occurrence of each possibility of the `opselect_gen` seems to be satisfactory for the arithmetic case. Note that all the labels assigned to coverpoints appear satisfactorily in the analysis window. The `addition_cov` cross shows good coverage only for the addition of positive numbers. This represents an unsatisfactory coverage requirement even for adds. We are yet to exercise the adds when all 0's, all 1's and carry intensive operations are involved. Note also the percentage of the total coverage goal for each of the three covergroups displayed under `[SCOREBOARD -> COVERAGE]` display statements.

To be able to recall this coverage at a later date, it is best to save the coverage information that you have generated for each run to be merged later. This can be done for each simulation (ideally at the end of each run) using the `coverage save` command within the modelsim transcript window. For example, after running 250 ns we can save the coverage generated using

```
VSIM #> coverage save ./arith1.ucdb
```

Let us consider the fact that we want to re-run the simulation with new constraint and we have done this by coding a new `Packet2.sv` file with slightly different constraints. On recompiling and re-running the simulation using `Packet2.sv` for say 2000ns, we get a new set coverage measures as shown in Figure 4.

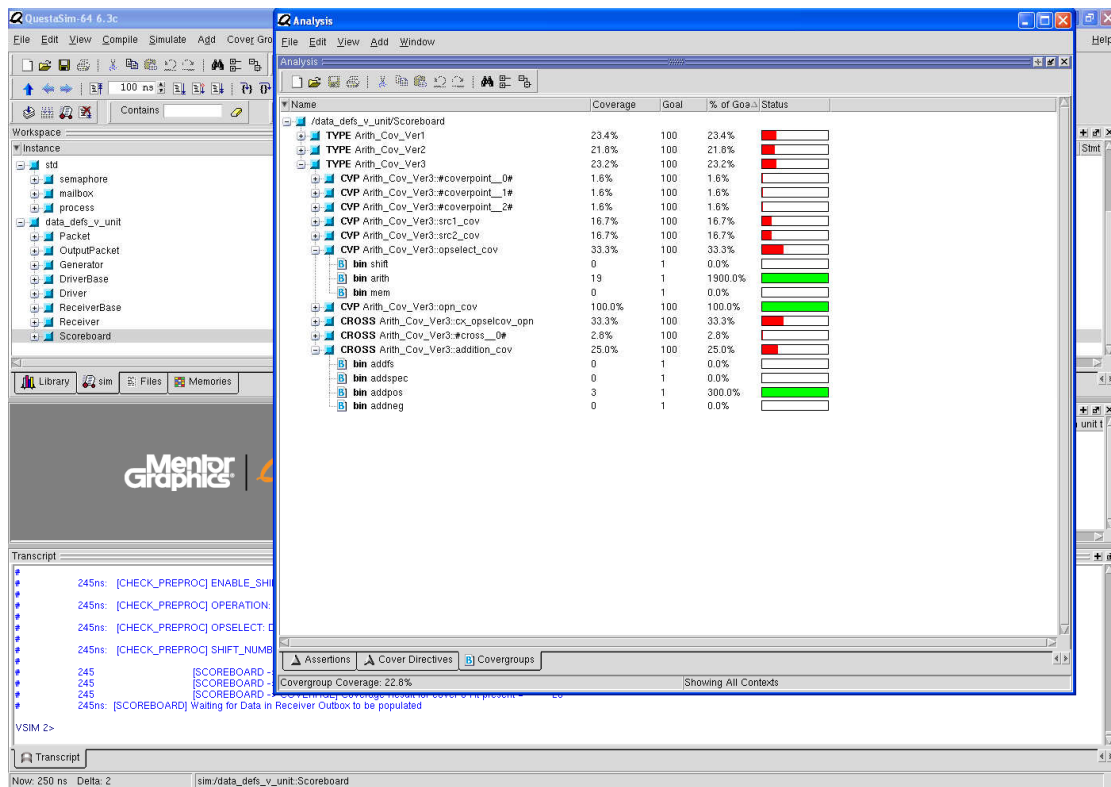


Figure 3: Coverage information using `Packet.sv`

Note also the increased numeric coverage values using the [SCOREBOARD -> COVERAGE] display statements.

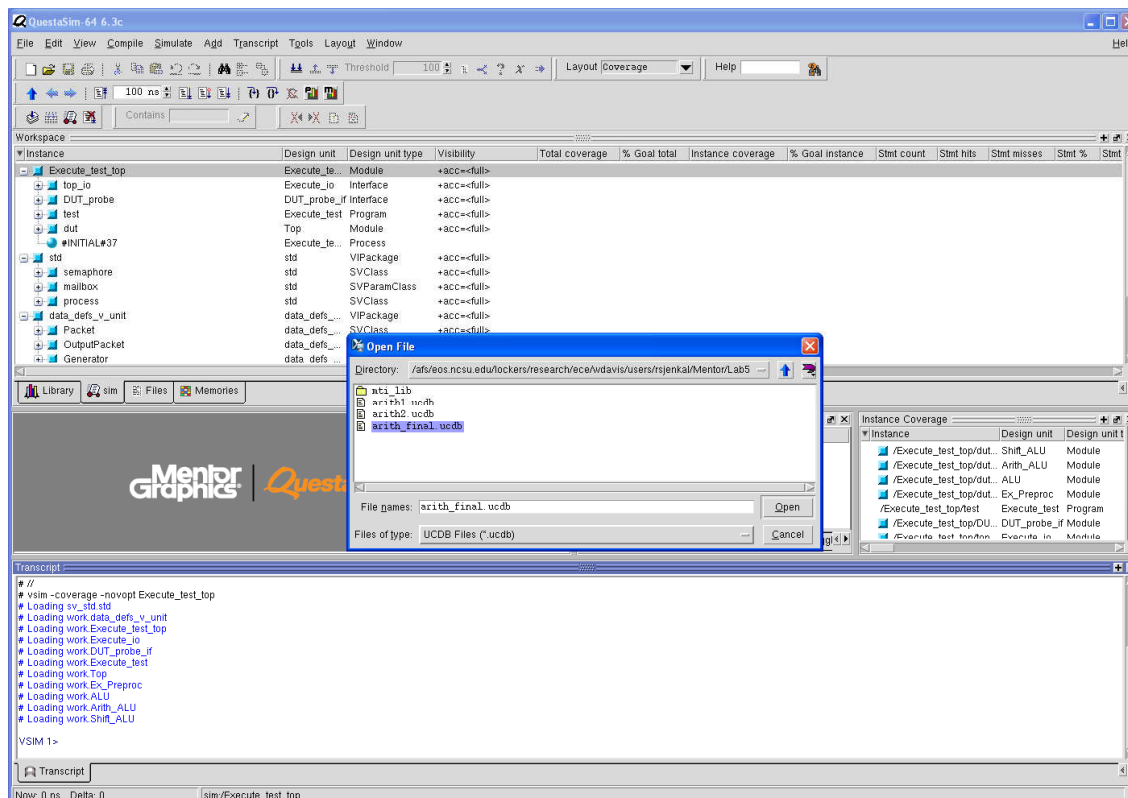
We can save the new coverage information again using

```
VSIM #> coverage save ./arith2.ucdb
```


To be able to deal with these set of multiple coverage database files for a given DUT and test suite, it is best to merge the multiple files into a master UCDB file for a particular cover group. This can be done by running the command (assuming we are creating a merged arith_final.ucdb):

```
VSIM #> vcover merge arith_final.ucdb ./arith1.ucdb ./arith2.ucdb
```

Before we view the combined coverage information, it is important to determine the steps needed when we have already closed the current session and need to come back to view coverage at a later time. To this end, we close the current GUI. We restart the GUI by doing "> vsim -novopt Execute_test_top -coverage". It is a MUST to have the top level design loaded before coverage can be loaded.



To view the final coverage information, we can load the final ucdb file by hitting the *file* -> *open* navigation buttons. This should open the open file window wherein you will have to change "files of type" to UCDB files to view the .ucdb files present. This is shown in Figure 5. Use this facility to load the arith_final.ucdb file. In this case, it should essentially be almost the same result as arith2.ucdb given that it is a superset of

arith1.ucdb. You can see the results using the analysis window at this point. If the coverage information does not appear immediately, it might be useful to attempt highlighting the scoreboard unit under data_defs_v unit. You can also use the expand buttons in the Analysis window to coax the tool. 

Lab5 Submission Requirements:

The addition_cov cross coverage for Arith_Cov_Ver3 is just a small subset of what checks should be performed for arithmetic operations. We are only checking to see if we have sent adds which involve all f's, positives, negatives and special cases. For this lab you must

- Create covergroups as you deem fit for shift and memory instructions such that you consider interesting cases for each case and just valid inputs for each case as well. Remember that not all operations are valid for shifts and memory instructions.
- Create a covergroup for arithmetic operations (you can re-use some of the constructs within Arith_Cov_Ver3) to include checks for all types of operations (AND, OR, XOR etc.). This would involve creation of relevant coverpoints to consider corner cases for each of these operations.
- Add coverage for the input and output for the pre-processor and the ALU to capture the occurrence of relevant variations and their crosses. Remember that probing can be done using virtual interfaces and mailboxes as shown in the previous Labs. Additionally, keep in mind that some of the output signals are asynchronous and hence must be sampled accordingly.
- Attempt to achieve 100% coverage for the above coded cross coverage. You would need to change your number of iterations to do so and make the Generator a smarter unit to provide this coverage as fast as possible. This would also require the displaying the result of get_coverage() as has been shown in the code. It would need something like:

```
coverage_value1 = Arith_Cov_Ver1.get_coverage();
$display($time, "[COVERAGE ARITH1] Coverage Result for cover 1 At
present = %d ", coverage_value1);
```

➤ Merge all of your relevant coverage data into a Lab4.ucdb. Please make sure it is what you expect.

Create a new document Lab4.doc to describe your efforts. Follow a style similar to the template at the end of this document. Expand the tables as necessary but fill in the data to represent your covergroups. You may add more tables depending on your number of covergroups

There is also a very good source of reading and instructional material for coverage and its invocation in Questa at
`/afs/bp.ncsu.edu/dist/questasim63/docs/pdfdocs`

In particular, the “Simulating with Code Coverage” chapter in the `questa_afv_tut.pdf` and the “Code Coverage” chapter in `questa_afv_user.pdf` are good references to have when dealing with coverage issue.

Follow the following steps for submissions (Solaris/Linux only please)

- `mkdir Lab4`
- copy all the SystemVerilog files into the `Lab4` directory
- Move the final `Lab4.ucdb` into the same directory
- Move the report into the same directory
- Zip the file using the command `> zip Lab4.zip Lab4/*`
- Submit the zip using the submit utility on the course webpage.

Lab4: Covergroup Additions

Shift/Memory Instruction Covergroups

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Arithmetic Covergroups

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Input/Output Pre-processor and ALU covergroups

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	

Covergroup Name:	
Coverpoints:	
Coverage Achieved:	
Purpose & Description of Covergroup	