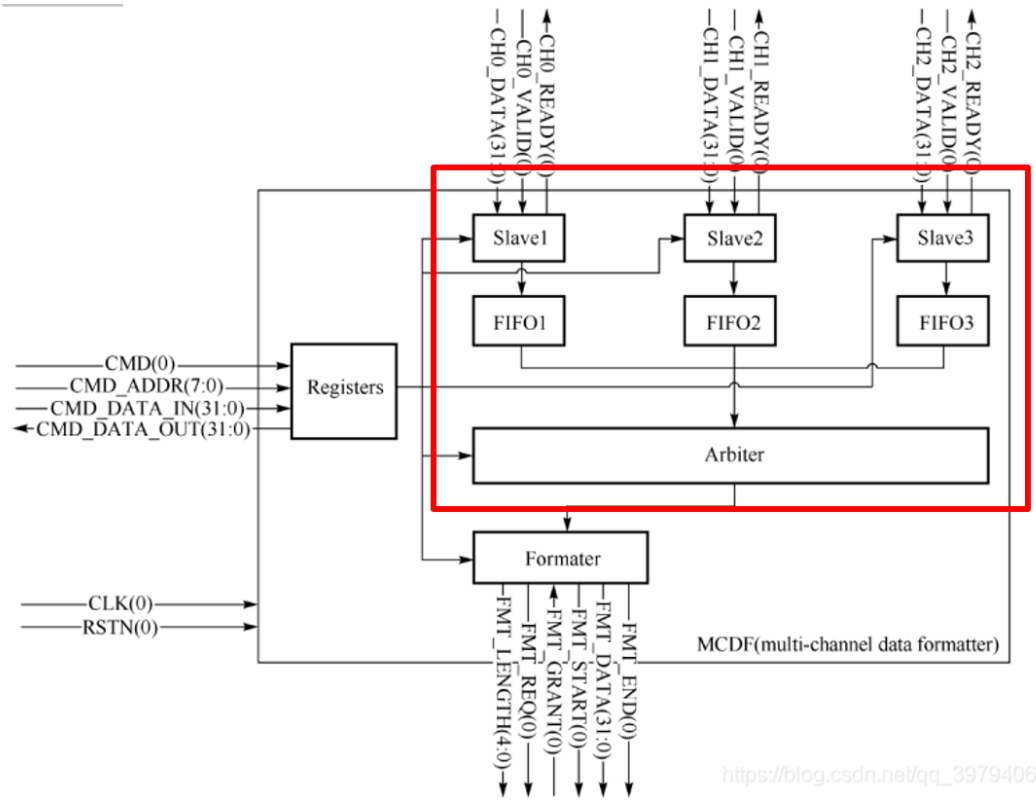
Lab4-Opt 3

Team Member: Zian Wang ziw080@ucsd.edu

Haoliang Wei h5wei@ucsd.edu

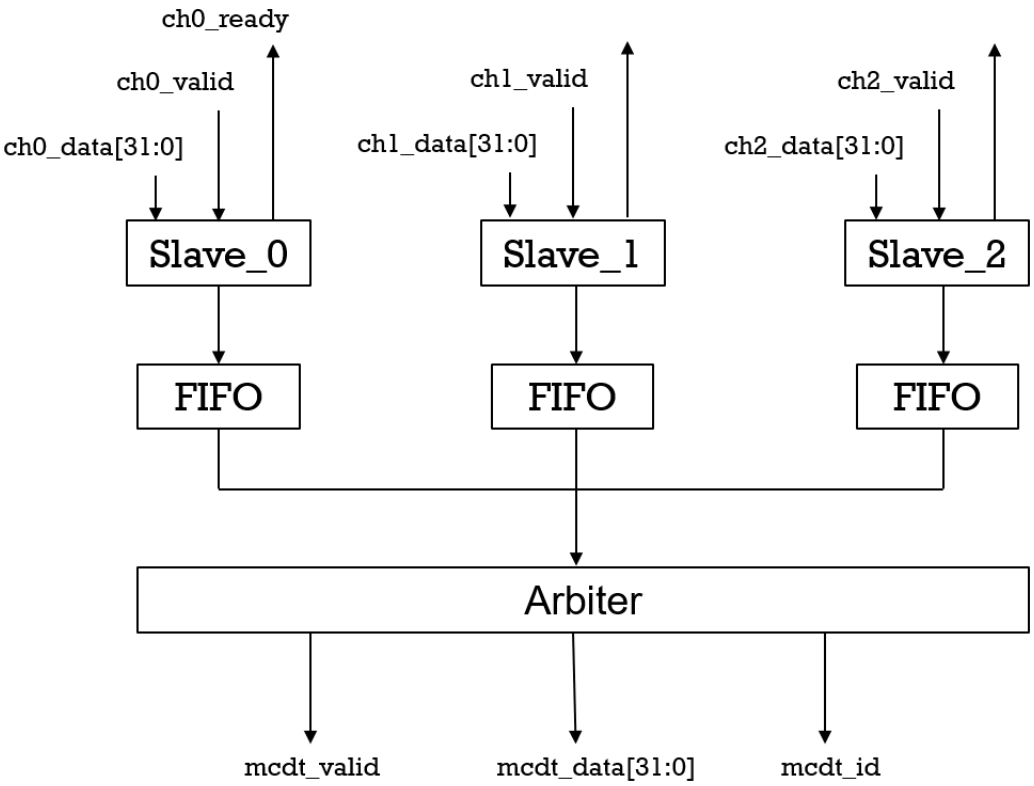
Yizheng Yu yiy003@ucsd.edu

DUT



This is the structure of original DUT. It includes three channels which are used to collect data from outside. The data from outside are stored in fifos. The contact is handshake model. When the data comes, the valid signal is set. If the slave gives ready signal, it means the transaction is done. Then three fifos are connected with an arbiter, which is used to arbiter the priority of three channels when formatter is ready to work. The function of formatter is to package the data with specific capacity. We also have 6 registers. Three is used to control the priority of three channels and close/open the channels. The others are used to set the number of packages for each channel.

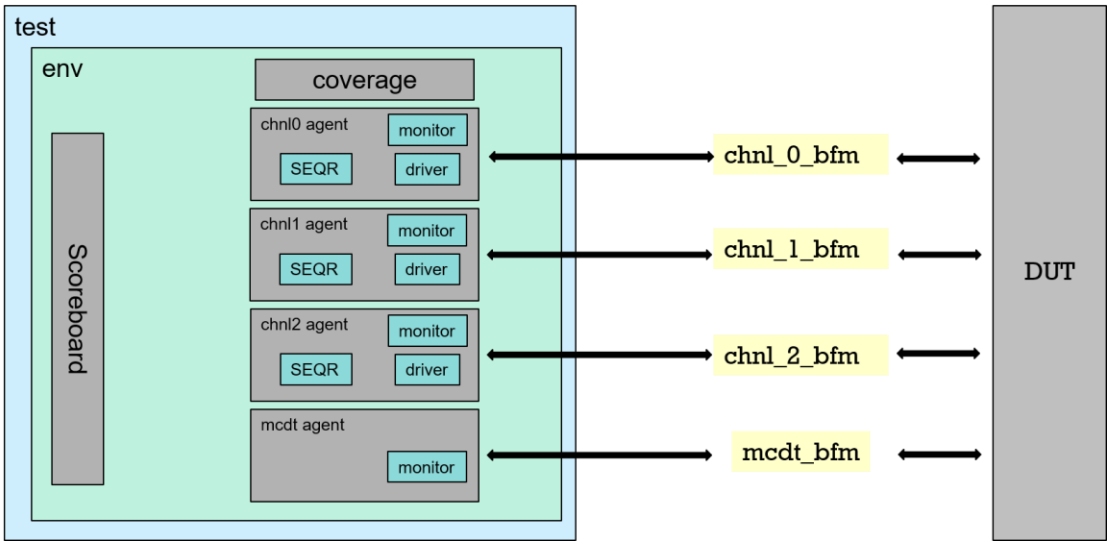
To simplify the DUT, we picked parts of it as our DUT, since the purpose of this lab is to utilize UVM tools.



Here is the structure of the simplified DUT. The output includes valid signal, data and id.

UVM

UVM structure

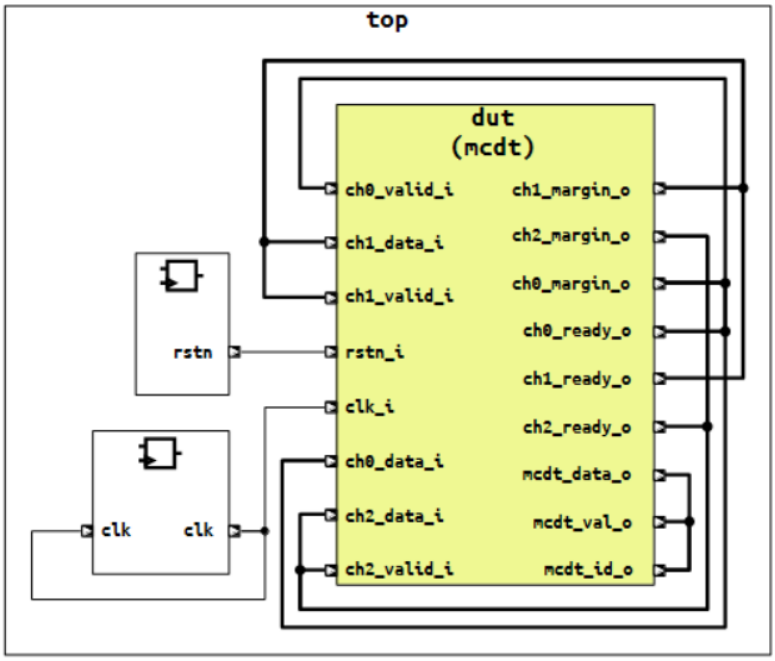


Simulation: Edaplayground: https://edaplayground.com/x/iyue

Source code: Github: https://github.com/wedogg/mcdt_uvm

The source code includes the original systemverilog testbench code and uvm testbench code.

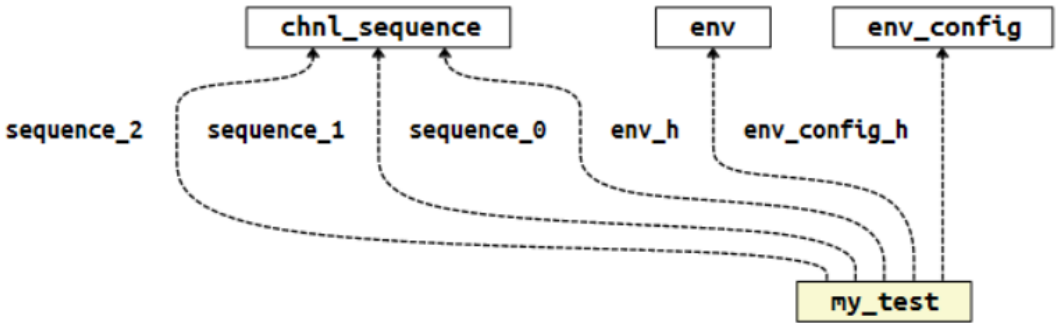
Top.sv



As we can see above, the structure of the test environment includes hardware part DUT and software part UVM, both of which are connected by the bfms.

The bfms are the interfaces between DUT and UVM test environment. The top level structure shown above is done by top.sv module. In top.sv, we build dut instance, interface instances, set bfms and start “my_test”.

My_test.sv

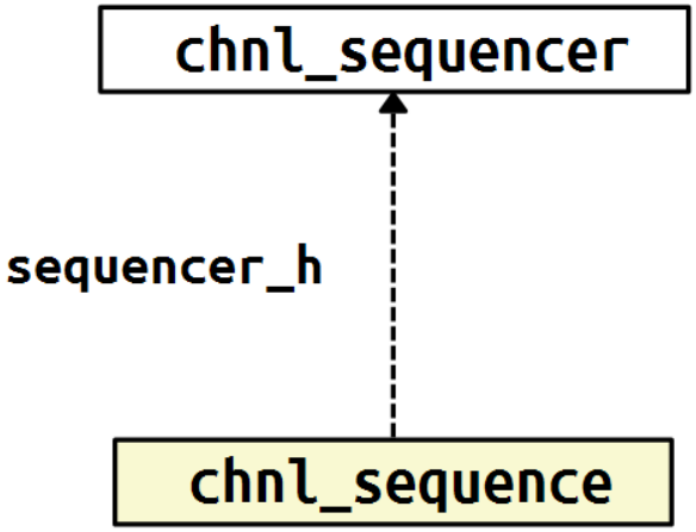


My_test class is the top level class in the uvm test environment. The next level includes env.sv only. So what my_test does is to initialize env. In addition, we need convey the bfms to env class. So we get four bfms from my_test class and send it to env class using env_config class. Otherwise, we create three sequence and start them in this class.

Chnl_sequence_item.sv

Chnl_sequence_item is used to define a type of data which is the transaction data transmitted between different class, such as sequence and driver. It includes data, ch_id (identifies the channel), data_nidles (number of idles between two data), pkt_nidles (number of idles between two packages), and so on. Those signals are defined as rand, so the sequence can randomize it while producing.

Chnl_sequence.sv

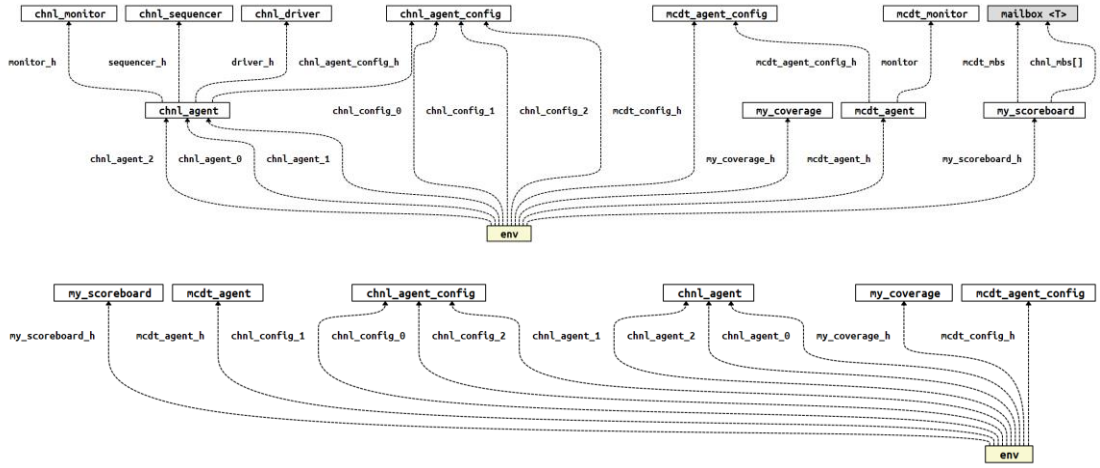


The sequence is used to produce signals we want. It's connected to corresponding sequencer. Through sequencer, the data produced by sequence can be sent to corresponding driver.

Chnl_sequencer.sv

Sequencer is the bridge between sequence and driver. Sequence is used to build data. Driver is used to send data. What sequencer does is to transmit data from sequence to driver.

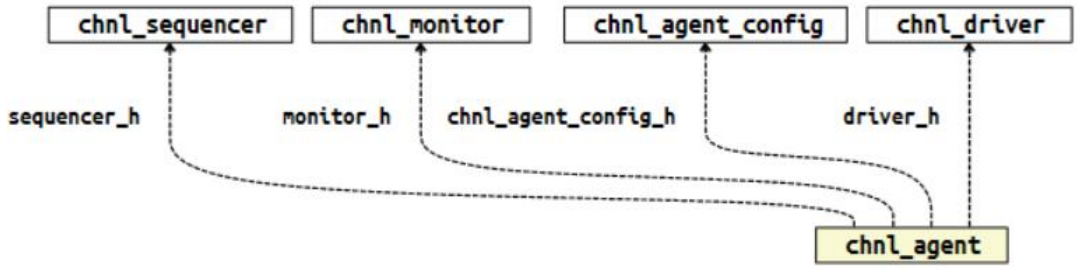
Env.sv



Under env, we have one scoreboard, one coverage, three chnl_agents and one mcdt_agent. So in build phase, we create those instances. Otherwise, we have to do some connections on this level. The scoreboard need to connect to the monitors from the four agents. The coverage should be connected to monitors

of three chnl_agents.

Chnl_agent.sv



This is the channel agent. We use it to produce data for each channel and monitor the results. In this class, we build driver, sequencer and monitor. Sequencer works as the bridge of driver and sequence. So we connect the two structures in connect phase. The function of the driver is to control the order of sending data to dut.

Chnl_agent_config.sv

This class is used to transmit the bfms. The function is quite similar to env.config.

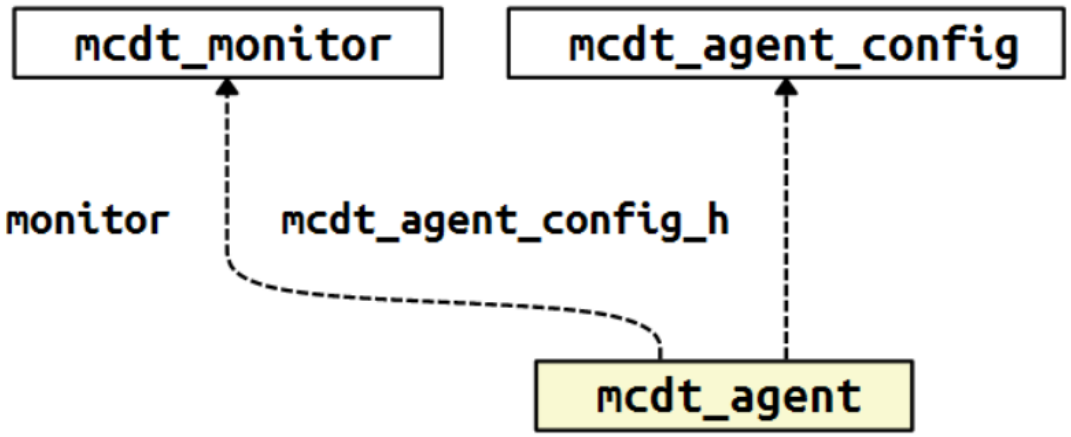
Chnl_driver.sv

Driver is used control the order of sending data, which is achieved by chnl_write task and chnl_idle task. Chnl_write is to provide data and set valid. Also, there is implementation of communication with sequence in the drive task.

Chnl_monitor.sv

Monitor is used to collect the data which has been sent to the driver. So we can get expected data for scoreboard and coverage.

Mcdt_agent.sv

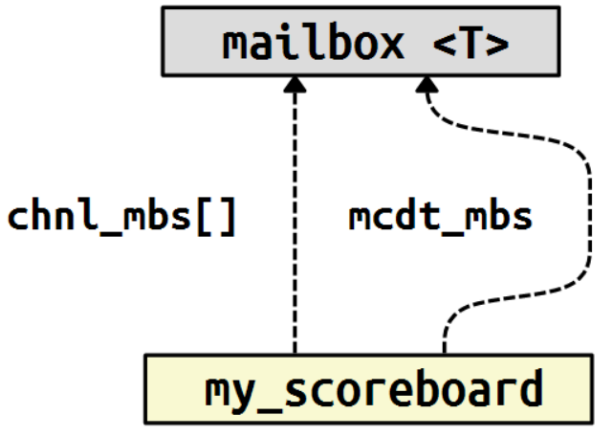


Mcdt_agent is used to monitor results from arbiter only. So it has only monitor, which is used to collect the results. Then send those results to scoreboard. So we can compare those results to the expected results.

Mcdt_monitor.sv

The function of the mcdt_monitor is to collect the results.

My_scoreboard.sv



Scoreboard is to compare the expected results and real results. The expected results are from monitors of chnl_agents, and real results from mcdt_agent. In order to complete the communication with monitors, the scoreboard has to be connected with monitors in env class. They communicate with each other through uvm_blocking_port.

Coverage.sv

This class is to collect the test source, how many data has been covered. It includes three coverpoints.

List of used uvm class:

uvm_test (my_test)

uvm_scoreboard (my_scoreboard)

uvm_monitor (mcdt_monitor and chnl_monitor)

uvm_agent (mcdt_agent and chnl_agent)

uvm_env (env)

uvm_coverage (my_coverage)

uvm_sequencer (chnl_sequencer)

uvm_sequence (chnl_sequence)

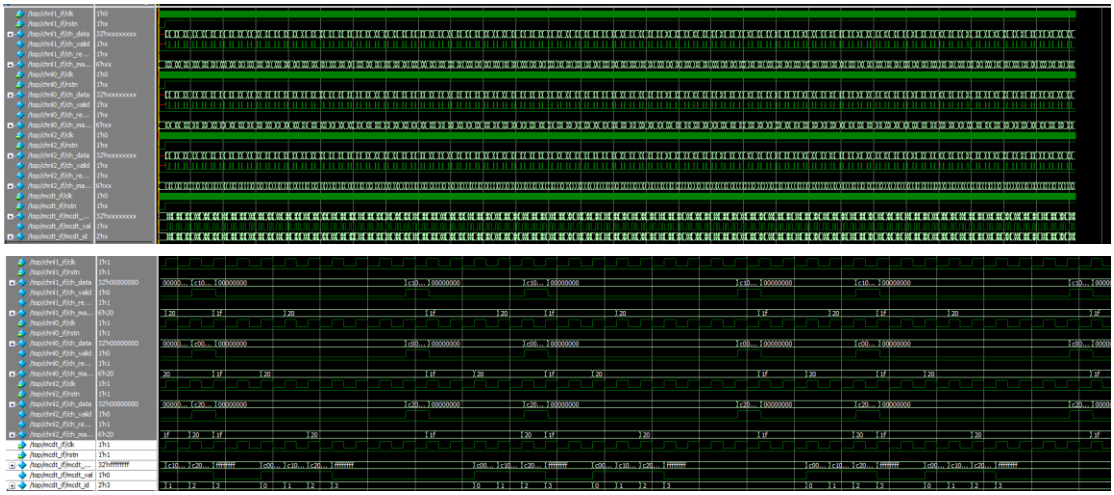
uvm_sequence_item (chnl_sequence_item)

What my_test does?

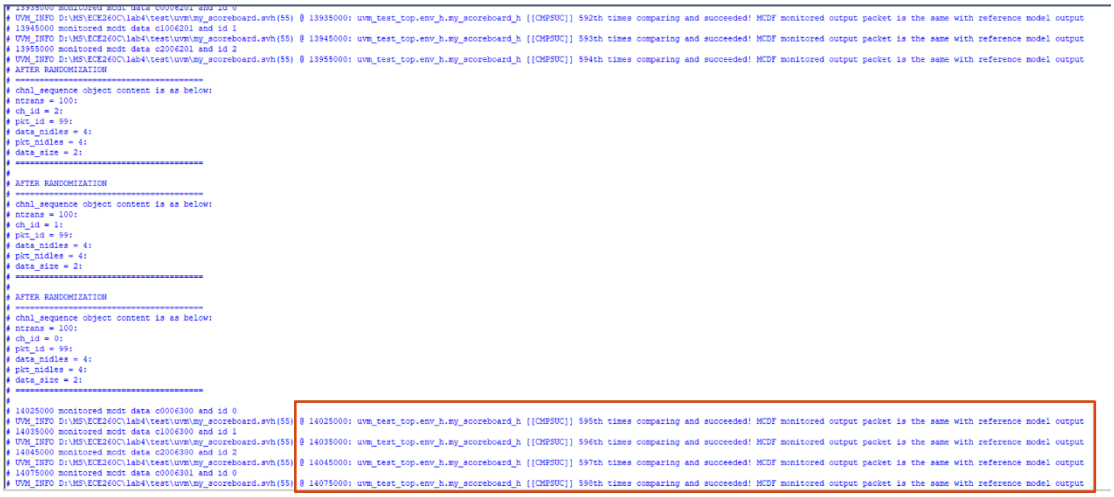
For each channel, it produces a series of continuous number at the same time. For channel zero, it produces data from ‘hc000_0000 to ‘hc000_ffff. For channel one, it produces data from ‘hc100_0000 to ‘hc100_ffff. For channel two, it produces data from ‘hc200_0000 to ‘hc200_ffff. The three channels send data to arbiter at the same time. So the arbiter needs to select one channel to open according to the priority of three channels. The second digit of data stands for the specific channel.

Simulation Results

Waveform



Comparation results display



Take the bottom lines as example. We compare the results with the expected results for each channel. If they compare successfully, it displays “# times comparing and succeeded!” as we can see in the red box.

The messages in the middle show the information of items produced by sequence. We also collect those information for the coverage.

Coverage report

Questa Coverage Report

Number of tests run:	1
Passed:	0
Warning:	1
Error:	0
Fatal:	0

List of tests included in report...

List of global attributes included in report...

List of Design Units included in report...

Coverage Summary by Structure:

Design Scope	Hits %	Coverage %
chnl_pkg	100.00%	100.00%
chnl_sequence/send_trans	100.00%	100.00%
my_coverage	100.00%	100.00%

Coverage Summary by Type:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	3	3	0	1	100.00%	100.00%
Assertions	1	1	0	1	100.00%	100.00%

The coverage report shows the covergroups coverage is 100&. The assertions coverage is 100%.