NTNU | Department of Electronic Systems

TFE4171 Design of Digital Systems 2
Lab 2

Delivery time:Sunday 26th February, 23.59

Learning outcome

Constrained randomisation:

- Classes for data structures
- Stimulus generation Coverage:
- Functional coverage
- Bins
- Cross coverage
- Covering transitions
- Coverage options

Reference literature

[1] IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language

[2] Ashok B. Mehta, SystemVerilog Assertions and Functional Coverage, 3rd edition, Springer, 2020.

[3] Ashok Mehta, Introduction to SystemVerilog, Springer, 2021.

[4] Chapter 18 in The Power of Assertions in SystemVerilog (Cerny et al., 2015).

[5] http://www.doulos.com/knowhow/sysverilog/tutorial/constraints/

1 PartA

Create a directory for this exercise in you home directory. After logging in, execute the

mkdir lab-2 command and then enter this

directory by

cd lab-2

Now you can copy the code for this first task by

cp -r /home/courses/desdigsys2/2023s/dd2master/lab2/partA .

and enter the partA directory. Under this directory the first version of the partA case is in the v0 directory. You can either make new version in new directories (v1, v2, ...) as you change the source code, or apply more advanced revisioning tools if you prefer such. The ex1-1.v file defines a module of a state machine controller that is clocked with the clk signal, reset data out with the rst=1 signal, and has the following active functionality:

- When validi=1 in three consecutive clk cycles, compute *data out* = $a * b + c$ where a is data in three clk cycles ago, b is data in two clk cycle ago, and c is the previous (one clock cycle go) data_in. Also set valido=1 when this occurs to flag valid data out. Else valido=0 which means data out is not valid.

The test-ex1-1.sv file contains the instantiation of the ex1 1 module together with a rudimentary testbed and also the binding of the modules to assertions in a property module. The property module is defined in the ex1-1property.sv file.

a) First run the simulation with no assertion checks with ./run check 0 to familiarize yourself with the functionality. Then define the check1 property and assertion to check that data out is correct after rst=1 is

asserted. Run the simulation with ./run check 1 and report results and comment.

b) Define the check2 property and assertion to check that the valido flag is correct after validi has been asserted for three consecutive cycles. Run the simulation with ./run check 2 and report results and comment.

c) Define the check3 property and assertion to check that the valido flag is 0 for any other combination of validi=1 such as two, one or none in a row. It is smart to only include the "fail" printout since you will not need dozens of passes cluttering up the output. Run the simulation with ./run check 3 and report results and comment.

d) Define the check4 property and assertion to check that the value of the data_out is correct with respect to the specification. Run the simulation with ./run check 4 and report results and comment.

e) Now extend the testbed (stimuli) by removing the comments in the stimuli code in the file test-ex1-1.sv and run the valid check test (check2) by the ./run check 2 command. You should now see several assertion fails. Analyze these fails, and find out what is the reason for them. Report and comment.

f) The functional fault that was exposed in the prior task shall now be fixed by changing the state machine functionality. First it is smart to make a revision of the code, either you can use a revision system or you can make a new directory copy by: cd ..; mkdir v1; cd v1; cp ../v0/* .

   Make a state diagram with transitions of the original state machine to understand how it works. Now you should probably add one or two more states in order to fix the working with overlapping valid sequences.

   This is helpful, but optimization/clever design makes it possible to revise the functionality even within the same number of states. This is not strictly expected, though, an increase in the number of states will be accepted for the exercise. Code the new state machine in the same file in the v1 directory. Now try verifying the functionality by running the check2 assertion again. Repeat this until you see correct behaviour.

Now you can run all the assertions checks with the command ./run_check 5 to verify that they all (hopefully) pass. Report and comment on both the new functionality and verification results

2 PartB

Now you can copy the code for Lab-2 partB by

```
cp -r /home/courses/desdigsys2/2023s/dd2master/lab2/partB .
```

and enter the directory for partB: `cd partB`

a) Use your editor of choice(vim, vi emacs, gedit..) to open alu packet.sv. Make a struct called ”data t”, that contains three random variables a, b, op.

b) Now we are going to construct the class ”alu_data”. Initialize the struct inside the class ”alu data” as random and call it ”data”.

c) Make a task ”get” that has three variables a,b and op, and that is used to get the random variables contained in data t.

d) Make three constraints c1, c2 and c3 and constrain a inside 0-127, b inside 0-255 and op inside 0-6.
Save and exit.

3

If you want to compile your design just run this bash script in the terminal(not inside your editor) ./simulate.sh At this stage it will compile but the simulation might fail or will not return anything. If you want to exit after compilation use ctrl-c.

a) Open the file alu tb.sv and make a vector of alu data called test data with the parameter-length ”NUMBERS”.

b) Make an ”initial begin” called ”data_gen” that contains a for-loop from 0 to NUMBERS. Put a delay of 20 before the loop. Generate random signals inside the loop using the ”new” operator and the built in ”randomize” function. Use the get function(from alu packet) inside the loop to send the random signals to the alu. Lastly, put a delay of 20 inside the loop. Run the simulation. You shall now be able to generate random input to the VHDL alu.

4

a) Make an enumeration called opcode that contains the opcodes in the ALU (ADD, SUB, MULT, NOT, NAND, NOR, AND, OR).

b) Make a covergroup called alu_cg that triggers on the positive edge of clk. Now you will make a couple of coverpoints inside the covergroup:

- op, that uses the opcode enumeration
- a, that has the bins: zero, small(1-50), hunds[](100, 200), large(200 and larger) and others(as default). The hunds[] should cover each case and both in order to show coverage for each and for both.
- Crosscoverage of a and b.

c) Initialize the covergroup.

d) Sample the covergroup. This can be done in two ways, either by declaring a fixed sampling edge for the group directly, or by using the sample() task in an always-block. Choose one of these for your design.

e) Use ./simulate.sh to compile and run. Make sure everything is working accordingly. Deliver code, log file, and coverage txt file on Blackboard.

5 PartC

Go back to the lab-2 directory, and create and enter a new directory partC. To this place copy the corrected and verified state machine code from partA and the alu code and wrapper from partB.

a) Now you shall rewrite this state machine in such a way that it uses the alu of partB for the addition and multiplication operation. Create a new top level module where you instantiate both the state machine and 2 alu's (since there will sometimes be both an add and a mult at the same time), and connect them with signals and buses for the control and data during addition and multiplication. You might have to rewrite the code a bit in order to be able to use two alu instances.

b) Validate the new state mechine and alu combination module with the full assertions set from partA and the last and most comprehensive test pattern set. Comment the result. If any errors are detected, find them and fix them. Repeat until correct.

c) Run the coverage analysis from partB on the new combination state machine and alu but use the same test pattern set as for the assertions above. What results would you expect? Would you expect the coverage to be better or worse? Comment on this.