Lab 3 - Universal Verification Methodology

Overview

This lab is designed as an introduction to Universal Verification Methodology (UVM). UVM is a powerful standardized verification methodology that was architected to be able to verify a wide range of designs. UVM is implemented as a SystemVerilog Class Library. For this lab, you will be using a UVM testbench to verify an ALU. The lab is broken down into two parts. In Part A, you will be completing the primitive testbench given to you by modifying the testbench code. In Part B, you will use the developed testbench to test the given design for bugs. Note: Use only 64-bit ECE LRC servers viz. luigi, wario, kamek or koopa.

Copy the lab3.tar.gz directory to your work directory using:

```
cp -rf /home/ecelrc/students/vhaldiya/TA_verif/lab3.tar.gz .
tar -xvzf lab3.tar.gz
module load mentor/modelsim/2016
```

This will create a lab3 directory with the following sub-directories:

- dut: Contains the Design Under Test (DUT) as a synthesized netlist.
- tb: Contains the testbench (TB) files.
- sim: Work area where the DUT and TB will be compiled by the simulator.
- lib: Verilog library used to synthesize the design. Simulator needs this while compiling the DUT.

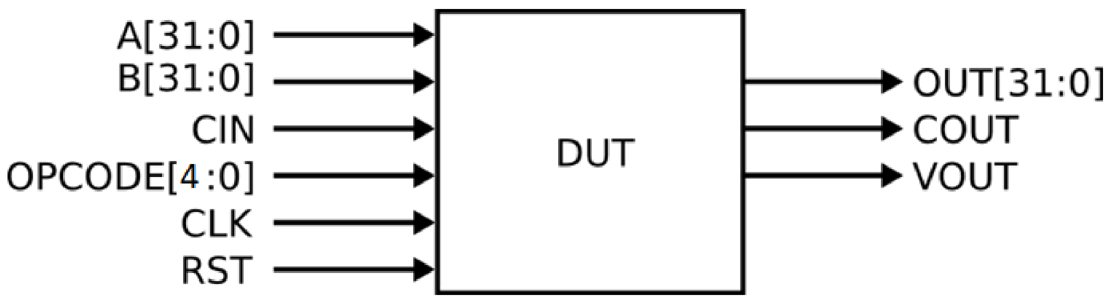
Design Specification

The DUT for this lab is an ALU. You will be treating the given DUT like a black box. For those purposes, you have been provided with a synthesized netlist of the original RTL. The DUT will be tested for bugs using the testbench. Figure 1 shows the input and output ports of the design under test.

Port	Width	Direction	Description
CLK	1	Input	Clock
RST	1	Input	Reset
OPCODE	5	Input	Function Select
CIN	1	Input	Carry In

A	32	Input	Input A
B	32	Input	Input B
OUT	32	Output	ALU Output
COUT	1	Output	Carry Out
VOUT	1	Output	Overflow Out

Figure 1: Design Specification



ALU Operations corresponding to each value of OPCODE[4:0]:

Logic operations (00---)

1. 111: or => A OR B
2. 011: xor => A XOR B
3. 000: not => NOT A
4. 101: and => A AND B

compare operation (A and B are signed numbers) (01---)

1. 100: sle => if A <= B then OUT(31:0) = <0...0001>
2. 001: slt => if A < B then OUT(31:0) = <0...0001>
3. 110: sge => if A >= B then OUT(31:0) = <0...0001>
4. 011: sgt => if A > B then OUT(31:0) = <0...0001>
5. 111: seq => if A = B then OUT(31:0) = <0...0001>
6. 010: sne => if A != B then OUT(31:0) = <0...0001>

arithmetic operation (10---)

1. 101: add => signed addition (A+B -> OUT)
2. 001: addu => unsigned addition (A+B -> OUT)

3. 100: sub => signed subtraction (A-B -> OUT)
4. 000: subu => unsigned subtraction (A-B -> OUT)
5. 111: inc => signed increment (A+1 -> OUT)
6. 110: dec => signed decrement (A-1 -> OUT)

shift operations (11---)

1. 010: sll => logic left shift A by the amount of B[4:0]
2. 011: srl => logic right shift A by the amount of B[4:0]
3. 100: sla => arithmetic left shift A by the amount of B[4:0]
4. 101: sra => arithmetic right shift A by the amount of B[4:0]
5. 000: slr => rotate left shift A by the amount of B[4:0]
6. 001: srr => rotate right shift A by the amount of B[4:0]

The intended specification of the DUT is mentioned below.

- When the RST signal is high, OUT, COUT and VOUT should be set to 0.
- For unused opcodes, the DUT should set OUT, COUT and VOUT to 0.
- Logic shifts should shift in 0's and Arithmetic left shift should shift in 0's.
- Arithmetic right shift should shift in the most significant bit.
- Rotate left shift should shift out the most significant bit to the least significant bit.
- Rotate right shift should shift out the least significant bit to the most significant bit.
- When the logic or rotation shift is performed, A is treated as purely logic information. You don't have to worry about the overflow in these two cases.
- Since this is a 32bit ALU, the amount of shift will never go over 31. Hence, only B[4:0] are used to define the shift amount.
- CIN is a don’t care for all operations apart from add, addu, sub and subu. For add and addu, CIN can be either 0 or 1, however for sub and subu, CIN should be 1.
- For all operations apart from arithmetic operations, COUT should be 0. COUT is the 33rd bit of the computed result for arithmetic operations.
- For add, if the inputs have the same sign (say 0) and the output has the opposite sign (say 1), then VOUT is 1.
- For sub, if the inputs have different signs for A (say 0) and B (say 1) and the output has a different sign than A (say 1), then VOUT is 1.

VOUT Clarifications:

VOUT should indicate the occurrence of an incorrect result for both the arithmetic operations and the arithmetic shift operations. It is supposed to signal the cases of both overflow (when the number is too big or small to be correctly represented by 32 bits) and the cases when there is loss of precision in arithmetic operations. The VOUT signal is referred to as "overflow" informally by us while answering your questions. Please refer to the following rules about the VOUT signal in the case of shifter operations:

1. In the case of logical or rotation operations, VOUT will never be asserted as we do not consider these operations to be arithmetic in nature.

2. In shift left arithmetic, VOUT will be asserted in the case of "Overflow". That is the case when you shift out significant bits. For example, shifting 1100 left by 2 would result in an overflow.

3. In shift right arithmetic, you will not have overflow. However, there still are cases of loss of precision in shift right arithmetic when VOUT is asserted. For example, shifting 0011 right by 1 would assert VOUT as there is loss of precision.

Please refer to the ALU_testcases.pdf file for examples which will help you understand the ALU operations. However, these examples are for a 16-bit ALU, but the design given in this lab is a 32-bit ALU.

Testbench Hierarchy

A picture representing the hierarchy of the given testbench is shown below.

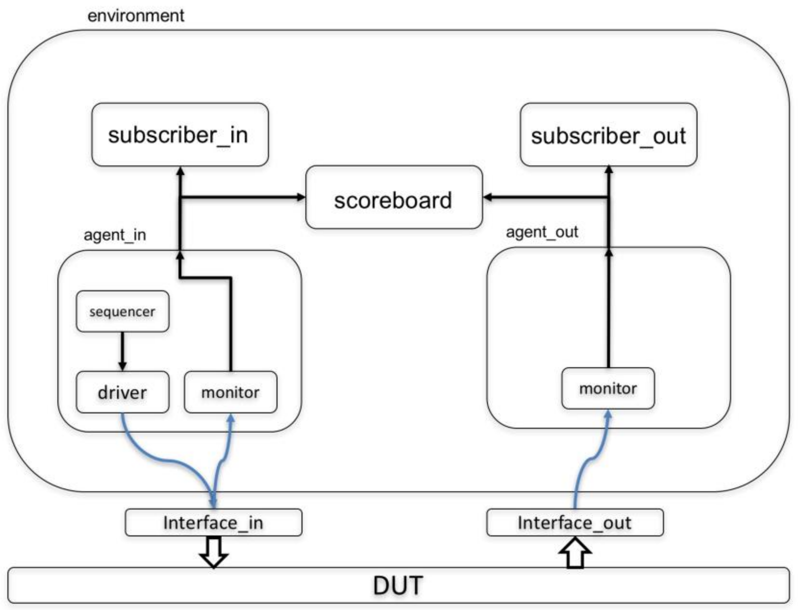


Figure 2: Testbench Hierarchy

Every new test is defined as a child class that is inherited from the parent class *alu_test*. The *alu_test* class instantiates the *alu_env* class which in turn instantiates *alu_agent_in*, *alu_agent_out, alu_scoreboard, alu_subscriber_in* and *alu_subscriber_out* classes. The in and out suffixes correspond to the input side transactions and the output side transactions of the DUT respectively. The inputs and outputs of the DUT are driven and sampled by the testbench using virtual interfaces to the actual interfaces connected to the DUT. A virtual interface is a pointer to an interface. An interface is an abstraction for a bundle of wires. The agent on the input side instantiates *alu_sequencer_in, alu_driver_in* and *alu_monitor_in*. The sequencer provides transactions to the driver and the driver drives the transactions on the DUT through the virtual interface. The monitor samples the virtual interface and creates transactions to be sent to the *alu_subscriber_in* and *alu_scoreboard*. The agent on the output side instantiates an *alu_monitor_out* to sample the output of the DUT and create transactions to be sent to the *alu_subscriber_out* and *alu_scoreboard*. The subscribers contain *covergroups* and *coverpoints* to track the functional coverage. The scoreboard contains a function to compute expected outputs (*getresult*()) and a function to compare the expected outputs to the actual outputs (*compare*()).

The sequencer *alu_sequencer_in* produces sequences of type *alu_transaction_in* defined in sequences.sv. Here, the inputs to be randomized are declared as type rand. Constraint blocks can be added here to constrain these inputs to various values or value ranges. The function *pre_randomize()* can be used to initialize the inputs before *randomize()* is called and the function *post_randomize()* can be used to report or post process the generated random inputs.

The testbench file hierarchy is as follows:

- tb.sv - contains the top level testbench which instantiates other test bench components and also the DUT.
- interfaces.sv - contains the interface definitions for the inputs and outputs of the DUT.
- sequences.sv - contains the class definition for the sequences to be used as transactions by the sequencer.
- modules.sv - contains the class definitions for various components like agent, monitor, sequencer, etc.
- tests.sv - contains the class definitions for various tests which inherit from alu_test class.
- scoreboard.sv - contains the class definition for the scoreboard to check the correctness of the DUT output.
- coverage.sv - contains class definitions for the subscribers to collect coverage.

Part A

In this part, you will be setting up the base testbench using the code provided before moving on to verifying the DUT. You need to complete portions of the testbench (marked with TODO in the sv files inside tb directory) before you can start adding tests to find design bugs. The following are the tasks you are required to do.

1. Instantiating the dut in tb/tb.sv

   (a) In the tb.sv file, instantiate the dut module as dut1 and use the input as dut_in1 and output as dut_out1.

2. Defining the output interface in tb/interfaces.sv

   (a) Complete the definition of the dut_out interface. Think about what the outputs are and define them correctly. **Hint:** The names are case sensitive.

3. Configuring the driver and monitor. In the file tb/modules.sv:

(a) In the class *alu_driver_in*, drive inputs from the transaction on to the virtual interface. You can access the DUT signals hierarchically using the virtual interface handle.

(b) In the class *alu_monitor_in*, sample DUT input values that are driven on the virtual interface and assign it to the created transaction *alu_transaction_in*.

(c) In the class *alu_monitor_out*, sample DUT output values from the virtual interface and assign it to the created transaction *alu_transaction_out*.

(d) In the class *alu_env*, consider setting the UVM verbosity to the maximum. You would want to come back and change this when you test the design.

4. Adding constraints to the input transactions. In the file tb/sequences.sv:

(a) On line 7 and line 29 register the alu_transaction_in and alu_transaction_out objects respectively. **Hint:** Look at what is happening in other classes.

(b) In the class *alu_transaction_in*, you can add constraints. Add validity constraints for the given design.

- Constraint on CIN, such that it is always 1 when arithmetic subtraction is being done.

(c) We strongly suggest adding more constraints in later parts of the lab. Also, it would be a good practice to disable these constraints for some simulations to see the output of the DUT

- You can have multiple constraints on the same input. They can be enabled or disabled using *constraint_mode().*
- You can use a distribution on any input to skew the generated random inputs.
- You can make the constraint solver use an ordering constraint solve A before B.

5. Checking the result. In the file tb/scoreboard.sv:

(a) Implement the *getresult()* function to compute the expected result using the transaction *alu_transaction_in*.

(b) Implement the *compare()* function to check whether the expected result and the actual result (from *alu_transaction_out*) are the same. If there is a mismatch, report it along with the inputs, expected and actual results so that it can be used to ascertain bugs.

6. Measuring coverage. In the file tb/coverage.sv:

(a) In class alu_subscriber_in, add covergroups and coverpoints on the input signals to gather input functional coverage.

(b) In class alu_subscriber_out, add covergroups and coverpoints on the output signals to gather output functional coverage.

Think about how you would want to measure coverage with this given design, and implement that in the above coverage model. Try to cover the corner cases.

You should now have a UVM testbench ready. In the sim directory, run make all to run one of the tests on the testbench. With a successful compilation of the testbench, you should see the testbench generating tests for the DUT and checking the output for correctness. You can now proceed to Part B.

Part B

In this part, you will be using the testbench you developed in Part A to test the DUT for bugs.

1. You are free to use any set of constraints and cover groups.
2. We highly encourage writing new tests or new sequences for different sets of constraints (different input patterns). Alternatively, you could just use the default test and sequences but have many constraints on the inputs depending on the operation.
3. You are also encouraged to read through other parts of the code and make any changes that you find helpful in your approach. Please document any such change that you make.
4. Document your complete approach in the report submission, along with a list of bugs found in the design.
5. You are **NOT asked to fix these bugs, just identify** as many as you can, and report them. **Be very specific when you describe a bug.**
6. Important: We have given a Makefile and a run.do file as an initial setup along with a README. You are free to modify these files to suit your requirements for running the tests. If you do modify any of these files, please modify the README accordingly so that the tests can be run without us having to understand/fix your code. We will use that README to run the tests you have written. Incorrect compilation or errors while running these tests will lead to deduction in points.

Electronic Submission

1. Document your results in a single PDF file lastname_firstname_lab3.pdf that has:
   - Approach to finding the bugs including what constraints were added, how was the coverage measured and which test sequences were used.
   - List of bugs found (be as specific as possible).

2. Submit all the files as a tarball – lastname_firstname.tar.gz on Canvas along- side your report. Do not combine the report into the tarball.

Bonus Policy

1. If you finish and submit your work earlier than the deadline, you obtain bonus in grade for early submission: +5% per day, Maximum +10%

2. If you submit your work later than the deadline, there will be a penalty in grade for late submission: -5% per day, Maximum -25% (Zero credit after the maximum penalty submission)

References

You can refer to the following links for this lab:

https://www.edaplayground.com/: You can register here and use this online tool to quickly check the sanity or behavior of some language constructs.

http://www.testbench.in/

https://www.doulos.com/knowhow/sysverilog/tutorial/

https://www.doulos.com/knowhow/sysverilog/uvm/

https://verificationacademy.com/: You can register here using your utexas.edu mail account.