Training on
IP & SoC Functional Verification Methodology
Using UVM

LAB
Random Variable
Constraint Programming

Objectives

This lab goes through the concept of random variables and constraints. It shows how random variables are created and how they interact together using constraints. The lab will show how random variables can be used to generate input stimulus for test purposes.

Introduction

In constraint programming, the relations between variables are stated in the form of constraints. Constraints do not specify a step or a sequence of steps to execute, but they rather define the properties of the solution which needs to be found.
Constraints are used in verification to declare valid states of test inputs and the randomization is used to generate tests amongst the valid states.

SystemVerilog provides the variable modifier “rand” to declare a random variable and the block construct “constraint *name* { *boolean_expressions;* } ” to declare constraints such as in the following example:

```
class myclass;
      integer a;     // this is a classical non random variable
      rand integer b;   // this is a random variable
      rand integer c;   // this is a random variable
      constraint b_lt_c { b < c; };
endclass
```

Global Explanation

The lab contains the following files

lab.sv	Includes all others.
tb.sv	Simple testbench driving clocks and reset
lab_verif_pkg.sv	Defines the main transactions to be used. You will have to make the variables random.
lab_prog.sv	The main program. You will have to use transactions created in lab_verif_pkg

To help us with this lab, we will keep the *driver* and *monitor* classes from the previous labs declared in the *project_utils_pkg*, as the transaction class (*addertTransactionBase*) that contains the necessary attributes including the action one that indicates whether the transaction is an access register (ACC) or a compute start (COMPUTE), the class implements also a method to print these attributes.

The two transaction classes are as following:

The lab_verif_pkg defines two transaction classes:

```
  //Transaction Class
  class transactionBase ;
    rand data_t      addr;
         data_t      data;
         direction_t dir;
         action_t    action;
  endclass
```

```
  class adderTransaction extends transactionBase;
    // Address is in the range from 0 to 5
    constraint addr_range {
      addr inside {['h0:'h5]};
    }
  endclass
```

Lastly, the *base_labseq* class, which is declared in the *lab_program* scope, represents our sequence base, it is responsible to create the transaction vector. The other sequence posteriorly created should inherits from this one and provide a *body* method (Or another name of your preference) that will analyse in the sequence.

```
//---------------------------------------
// Base Sequence Common Class
//---------------------------------------
class base_labseq;
  adderTransactionBase trans[];
  int count;


  function new (int count = 10);
    this.count = count;
    this.create_trans();
  endfunction

  function void create_trans();
    $display($sformatf("count = %d",count));
    trans = new [count];
    foreach (trans[ii]) begin
      trans[ii] = new;
    end
  endfunction

endclass
```

Instructions

Follow instructions given in “aedv_training_labs_intructions_for_questa.pdf”.
Open the file
  <SANDBOX>/labs-Xdays/labNN-systemverilog_random_generation /lab.sv
Select

[✓ System Verilog]

Step 1 – Understand existing code

- Open the file: <SANDBOX>/labs-Xdays/labNN- random_generation/ project_utils_pkg.sv

Search for LAB-TODO-STEP-1

- Question:
  - What is the difference between *addr* and *in_data* generation?
  - Why *in_data* is always X?

Step 2 – Create random transactions

Search for “LAB-TODO-STEP-2-a”
a) Identify the declaration of the variable: *in_data*.

b) Add “rand” keyword in front of the variable “*in_data*”
Recompile the test and reload the simulation.

c) Change the seed value & Reload the simulation again.
Check the values that are generated.

Search for “LAB-TODO-STEP-2-b”
d) Add a constraint to specify the read only area, do not allow a write in this registers.

```
constraint read_only_area {
      addr inside {'h4, 'h5} -> dir == READ;
}
```

Step 3 – Create sequences of random transactions

- Open the file: <SANDBOX>/labs-Xdays/labNN- random_generation/lab_program.sv

Search for “LAB-TODO-STEP-3-a”
Replace the loop of transaction generation with the generation and driving of the “init_sequence”

```
      init_seq = new;
      init_seq.body();
      foreach(init_seq.trans[ii]) begin
            dri.run(init_seq.trans[ii]);
      end
```

Search for “LAB-TODO-STEP-3-b”

- Analyse the body task, what does it do?
- What does the *with* keyword after the randomize do?

Search for “LAB-TODO-STEP-3-c”

- Create a list of incremental data writing transactions
    - The first transaction takes a random *in_data* value.
    - Each following transaction takes the previous value plus 1 (*in_data* + 1).
    - For all transaction do a valid writing (action == ACC, dir == WRITE)
    - You may be inspired by the *init_labseq*.

Search for “LAB-TODO-STEP-3-d”

- Add a loop of 10 interaction to generate the previous sequence.
- You can draw on the *init_seq* generation
- Re-compile and Re-run

Extra Step – Create more complex sequences

Search for “LAB-TODO-STEP-EXTRA”

- Create a list of Random transactions
- Create a list of transactions with READ generation
    - The first transaction must be a COMPUTE one (action == COMPUTE)
    - All the remain transactions must be a reading
- Add a loop to generate randomly any of the previous sequences (using “**randcase**”)