IC Verification Improvement By Using The New Random Testcase Generation Methodology

Ngo Thanh Nhan
Integrated Circuit Design Research and Education Center – Vietnam National University
Ho Chi Minh City, Vietnam
nhan.ngothanh@icdrec.edu.vn

Abstract— **A new methodology to generate testcases for IC verification is proposed. The methodology offers new way to create the desirable number of test cases, which have the same template, but differ with each other in internal data. So it is definitely advantageous for users to randomize easily a group of similar cases. Moreover, the proposed method also supports users' define library. The advantage of this library is that users could collect a group of repeatable code lines and store them as a template in the library. By this way, it is really convenient for users in case of the group of repeatable code lines is used from time to time in different testcases. Following that, the particular compiler is proposed to make the testcase writing become straightforward and brisker.**

Keywords-methodology, syntax, testcase, random.

I. Introduction

To implement a verification system to check the validity of a design, there are many parts that verifiers need to do, such as environment organization, testbench, script, testcases. In all of them, testcases actually are the crucial part, which consumes not only a lot of time but also much more effort of verifiers. Therefore, it is really beneficial for verifiers if there is any method or tool that favors them in speeding up the testcase writing. In reality, it is necessary to check cases that verify the same function of the design, but the data is different. These testcases could be considered as a group of testcases. In the common way, the particular script is used to solve this issue. The solution is that the verifiers have to write script to duplicate these similar testcases. In this paper, the new random testcase generation methodology is proposed, which offers verifiers the advantageous way in creating testcases. Following that, the particular syntax is defined and used in the combination with the other official programming languages in the testcase file. Moreover, the existence of library, which verifiers could store the repeatable group of code lines as a special function and share them among members of verification team. It is really worthy when verifying huge projects, and makes the testcase generation become easier and significantly convenient.

II. Implementation

In this part, the general structure of methodology is presented. Following that, the methodology has architecture which consists 4 main parts as shown in Fig. 1. They are:

- Inheritable Library
- Input testcase
- Synthesis
- Output testcase

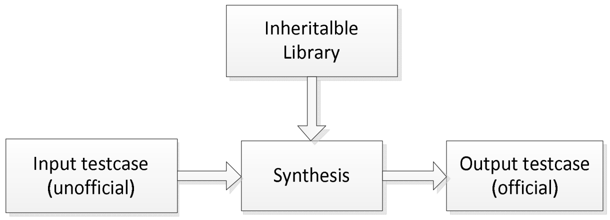

Figure 1. General structure of methodology

By this method, the testcase which is used to run design verification is the output testcase. Input testcase is used as a seed to generate the group of testcases that has same format, but differs each other in data. Using the inheritable library, it is actually brisker to create testcase. The synthesis tool has responsibility to generate the desirable output testcase file from the input file by decoding the input file and accessing elements in library. More detail of them is devoted deeply in the next sessions.

A. *Input testcase*

Input testcase is the testcase written with the combinational syntax between the standard programming language and the particular syntax defined by proposed methodology. Following that, the input testcase is shorter and more straightforward to manage. Detail of particular syntax is discussed in the next session.

B. *Inheritable Library*

To test an IP core, an enormous number of testcases are usually used. However, it is easy to realize that these testcases are actually generated from minor groups of codes called repeatable parts. It means that by using the library of repeatable parts, almost all testcases could be released. The convenience of the library is inheritability. It is absolutely beneficial for the

Integrated Circuit Design Research and Education Center (ICDREC) large projects when many people attend to verify as the team work.

C. *Synthesis*

Synthesis is the central process unit used to transform an informal testcase into a formal one. On the way, the input testcase and the library are inputs of the synthesizing unit. After the process, the group of desirable testcases, which are included in testbench, is generated. Synthesis tool could be written by any programming languages such as C/C++, Python, Perl [1], Tcl, et al.

D. *Output testcase*

It could be a testcase or a group of testcase belonging to the expectation of user. The output testcase is available to use in IP verification flow. For IC verification, it usually is a task which is called by testbench.

III. LIBRARY BUILD

Library keeps the repeatable codes in file whose name is known as command, which used at input testcase. Therefore, input testcase could be easy to access library through calling command by the particular syntax defined in session IV. Because of the repeatable codes has same structure but differs each other in data inside. Therefore, they are usually saved in the library with special marks, which are available to replace by expected data. The synthesis tool looks for the mark in order to supply the suitable data to that place. "*##i*" with *i* begins from 1, is used to mark special places in library file.

IV. PARTICULAR SYNTAX

The special syntax is defined in the proposed methodology that supports user in randomizing internal data of testcase. Next, some syntax is devoted as example to expose the idea of methodology. They are basically the key words that the synthesis tool based on them to process.

A. *Array*

The new type of array is defined. Array stores the data values that user want to change in their testcase. There are two kinds of array. They are random array and incremental array. By using random array, the data value is chosen accidentally from array. In contrast, the growing method is used to get the particular data when exercising the incremental array. The declaration of these arrays is expressed as follows.

Declaration of random array:

@[array_name],[val_1],…,[val_i],…,[val_n]

@[array_name],**range**,[val_1],[val_n]

Declaration of incremental array:

@[array_name],**incr**,[val_1],…,[val_i],…,[val_n]

@[array_name],**incr**,**range**,[val_1],[val_n],[incr_const]

To get value from array, symbol "*" and "?" is used before the *array_name* in the input testcase file. When using "*[array]", the synthesis tool takes any value from the indicated array following random or incremental method, and replaces "*[array]" by the value. When using "?[array]", the value gotten at latest time of executing "*[array]" is reused.

B. *Variable*

New defined structure of variable supports partition randomization for value of variable. The flexibility in randomizing the value of variable is the notable stress of this data type. It is definitely powerful when user wants to randomize a compounding number. The declaration of variable is shown as follows.

&[name]=[val]

&[name]=[val]-*[array1]-*[array2]-…-[val]

The second method allows randomizing variable value by compounding way. Following that, the getting value is constituted by many different parts. Each part could be randomized its value. For example,

@a,1,5

@b,2,6

&N=3-*a-*b

After synthesis, N could receive one in four values 312, 316, 352 or 356.

C. *Library Access*

The syntax "%[library_file_name],[arg]" is defined to access library and get the suitable content. For example,

% test,10,20

It means that exploring file name "test" in the library, replacing "##1","##2" by "10","20" respectively; then pasting all to output testcase file.

D. *Iterativeness*

1) *Sequential Iterativeness*

Sequential iterativeness shortens group of code that is repeatable. Following that, user only writes

%loop, [N]

<content>

%endloop

for implementing N times loop of particular content. The repeatability is executed following the order of code line.

2) *Random Iterativeness*

Difference with sequential iterativeness, random method also implements N times loop of particular content, but it only gets one line in the content each time, processes and pastes to output testcase. This structure is usually used to choose randomly any command in the group of indicated commands, which is put between two keys "%random" and "%endrandom", and repeat N times but any command one time.

%random, [N]

%[command_1],[arg]

…

%[command_n],[arg]

%endrandom

A note is that N could be an absolute number or random number by using "*array" or "?array".

V. Practical Example

To clarify the idea of proposed methodology, the practical example, which uses this methodology to generate expected testcase, is presented. In the example, a group of testcases generated from the input testcase. The number of output testcases depends on user's suggestion. Fig. 2 shows the content of the sample testcase. Following that, AHB Master VIP implements the access to memory. Requirements are that creating N testcases which have the same content of sample testcase with "ADDR" in the range of 8 bit number and "DATA" is one of three values 32'h55, 32'h5a, 32'haa.

```
task test;
integer xferAttr, readHandle, readBufHandle, bufHandle,
cmdStatus, readData;
begin
 $display("This is ...");
xferAttr = `DW_VIP_AMBA_XFER_SIZE_32;
//write data to memory
tb_top.master_vip.write(`VMT_DEFAULT_STREAM_ID, ADDR,
DATA, xferAttr, bufHandle);
tb_top.master_vip.block_stream(`VMT_DEFAULT_STREAM_ID, 0,
cmdStatus);
//read data from memory
tb_top.master_vip.read(`VMT_DEFAULT_STREAM_ID, ADDR,
xferAttr, readHandle);
tb_top.master_vip.get_result(`VMT_DEFAULT_STREAM_ID,
readHandle, readBufHandle);
tb_top.master_vip.get_buffer_data(readBufHandle, readData);
//check data value
if (readData!==DATA) begin
$display("\nExpect data: %d  Actual data: %d\n", DATA, readData);
end
end
endtask
```

Figure 2. Sample testcase

It is no matter if there is only one testcase with particular couple of address and data. However, if the goal is testing the access of AHB Master VIP to memory in a huge range of address with different data, it is more complex. Besides, the demand of generating testcase in different scenarios still is considered. Realizing from the sample testcase, the code could be split into different groups of code lines which could use again and again in testcases. Therefore, we could define some files in the library as follows.

declaration.v

```
integer xferAttr, readHandle, readBufHandle, bufHandle,
cmdStatus, readData;
```

write_data.v

```
//write data to memory
tb_top.master_vip.write(`VMT_DEFAULT_STREAM_ID, ##1, ##2,
xferAttr, bufHandle);
tb_top.master_vip.block_stream(`VMT_DEFAULT_STREAM_ID, 0,
cmdStatus);
```

read_data.v

```
//read data from memory
tb_top.master_vip.read(`VMT_DEFAULT_STREAM_ID, ##1, xferAttr,
readHandle);
tb_top.master_vip.get_result(`VMT_DEFAULT_STREAM_ID,
readHandle, readBufHandle);
tb_top.master_vip.get_buffer_data(readBufHandle, readData);
```

check_result.v

```
//check data value
if (readData!==##1) begin
$display("\nFAIL: Expect data: %d  Actual data: %d\n", ##1,
readData);
end
```

Using the methodology particular syntax and the library, the input testcase could be edited simply as shown in Fig. 3 to meet the requirements. Synthesis tool reads input testcase file, runs following the request of N output testcases and releases expected result.

```
@addr,range,0,255
@data,32'h55,32'h5a,32'haa
task test;
%declaration
begin
$display("This is ...");
xferAttr = `DW_VIP_AMBA_XFER_SIZE_32;
%write_data,*addr,*data
%read_data,?addr
%check_data,?data
end
endtask
```

Figure 3. Input testcase using particular syntax

As another example, in order to generate testcase that implements N times of writing, reading and checking data, the syntax of sequential iterativeness could be used. Following that, the group of codes between the keys of “%loop” and “%endloop” is repeated N times. Therefore, input testcase could be composed as in Fig. 4.

```
@addr,range,0,255
@data,32'h55,32'h5a,32'haa
&N=5
task test;
%declaration
begin
$display(“This is ...”);
xferAttr = `DW_VIP_AMBA_XFER_SIZE_32;
%loop, &N
  %write_data,*addr,*data
  %read_data,?addr
  %check_data,?data
%endloop
end
endtask
```

Figure 4. Input testcase using iterative structure

VI. CONCLUSION

The paper presents simply the idea of methodology that generates testcase more convenient and brisker. The flexibility of creating testcase, which is formed by any programming language, actually makes the strong point for the methodology. Besides, the project verifiers in the same project could share their work together by using inheritable library. It helps to save a lot of effort and working time. As a result, it increases the performance and reliability of verification. In addition, based on the methodology, developers could define other advanced syntax that could improve the effectiveness of methodology. It could make the particular syntax and synthesis tool be smarter, more flexible and powerful.

ACKNOWLEGMENT

I am grateful to my colleagues for their comments during the time when I implemented this idea.

REFERENCES

[1] Johan Vromans, “Programming Perl”, Squirrel Consultancy.