Scale IP UVC in SOC testbench

Lingkai Shi

Advanced Micro Devices

Lingkai Shi
Email: lgshi@amd.com

Introduction

As chiplet/multicore SOC designs become more common, the challenge of duplicating IPs (Intellectual Properties) grows. While duplicating IP RTL modules is straightforward, scaling IP verification environments is complex. This paper presents a methodology to adapt IP testbenches to SOC environments with minimal modifications, reducing errors and enhancing automation.

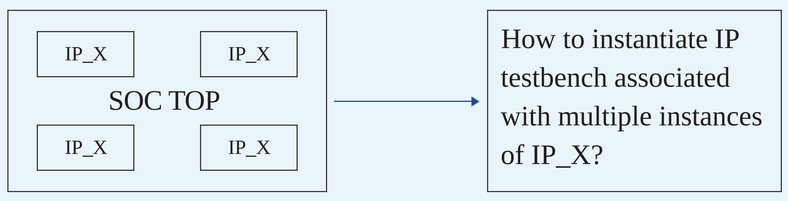

Fig. 1 Scale IP Testbench in SOC

Challenges in Scaling IP UVCs in SOC Testbenches

UVM methodology enables easy duplication of most testbench components in IP, similar to RTL. However, certain components resist scaling:

- **Singletons**
- **DPI**

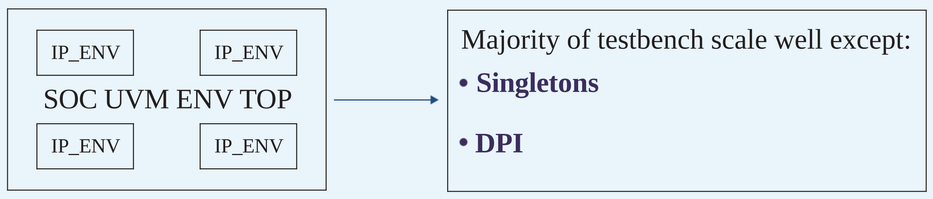

Fig.2 Standard UVM solution to scale the IP testbench env

Singleton Object

This design pattern restricts a class to a single instance and is widely accessible. While this design pattern is ideal for IP verification utilities, their unique nature complicates scaling within SOCs.

```
1. class singleton;
2.     static singleton m object;
3.     typedef enum {RESET, INIT, RUN, FINISH} Status e;
4.     status e m status;
5.     local function new();
6.     endfunction : new
7.     static function singleton get instance();
8.         if (m object==null) begin
9.             $display("object is null \n");
10.            m object = new();
11.        end
12.        return m object ;
13.    endfuntion : get instance
14.    static void function revise m status(Status e input);
15.        m status = input;
16.    endfunction : revise property A
17.endclass : singleton
```

Fig.3 singleton example

```
1.  program main;
2.     singleton objSingle1;
3.     singleton objSingle2;
4.     initial
5.     begin
6.        objSingle1 = singleton::get instance ( );
7.        objSingle2 = singleton::get instance ( );
8.
9.        objSingle1.revise m status(RESET);
10.       objSingle2.revise m status(RUN);
11.
12.       $dis-play($sformat("Testing Singleton object %S, %S \n",objSin-
   gle1.m status,objSingle2.m status));
13.    end
14. endprogram: main
```

Fig.4 singleton test code

Global Variable used in Direct Programming Interface (DPI)

Allows external functions written in C/C++ to be used in SystemVerilog. This causes issues when multiple instances of the same IP are instantiated in an SOC, as illustrated in Fig. 5

```
#include "svdpi.h“
enum Status_e {RESET, INIT, RUN, FINISH};
Status_e m_status;

void dpi_sample(const int I1)
{
 printf(m_status);
  ...
}
```

How to make m_status have different value for different IP instances in DPI call?

Make m_status a multidimension array is not a good idea because it is error prone for memory leak. It will be a nightmare to debug memory leak in SOC tb.

Fig.5 DPI global variable issue in multi-die testbench architecture.

Solutions

Enhancing Singleton

To make a singleton class scalable, introduce a multidimensional array, like m_object in Fig. 3, to static member variables. Each m_object links to an IP instance. Modify the constructor to select the IP instance, using indices like m_object[X][Y] for SOC packageX and dieY.

With minimal code changes, as shown in Fig. 6, only the constructor and m_object references need updating. In the SOC context, the singleton reference should include the index, preserving all singleton advantages. Users identify IP instances to the access function, and all functions remain consistent. For clarity, consider Fig. 7's output, where objSingle1 and objSingle2 represent different IP instances. Each m_status is modifiable individually. Any missed updates trigger a VCS compilation error for correction.

```
1.  `define MAX IP NUM 10
2.  class singleton;
3.    static singleton m object[`MAX IP NUM];
4.       typedef enum {RESET, INIT, RUN, FINISH} Status e;
5.    Status e m status;
6.    local function new();
7.    endfunction : new
8.    static function singleton get instance(int index);
9.      if (m object[index]==null) begin
10.       $display("object is null \n");
11.       m object[index] = new();
12.     end
13.     return m object ;
14.   endfuntion : get instance
15.   static void function revise m status(Status e input);
16.     m status = input;
17.   endfunction : revise property A
18. endclass : singleton
```

Fig.6 multi-dimension singleton example. The revision to the classic singleton is highlighted in yellow.

```
1.  program main;
2.    singleton objSingle1;
3.    singleton objSingle2;
4.    initial
5.    begin
6.       objSingle1 = singleton::get instance ( 0 );
7.       objSingle2 = singleton::get instance ( 1 );
8.
9.       objSingle1.revise m status(RESET);
10.      objSingle2.revise m status(RUN);
11.
12.      $dis-play($sformat("Testing Singleton object %s, %s \n",objSin-
   gle1.m status,objSingle2,m status));
13.   end
14. endprogram: main
```

Fig.7 multi-dimension singleton test code. The difference with classic singleton is highlighted in yellow

Global variable in DPI

To summarize, supporting multiple instances of an IP in a SystemVerilog testbench for DPI global variables can be solved by following steps:

- **Create a wrapper singleton class.**

Begin by designing a singleton class that encapsulates all global variables as member variables.

Introduce a new global variable pointer, termed IP_GLOBAL_CONFIG, to facilitate easy access to the singleton by legacy functions.

Update all references to the original global variable to include this pointer reference. For instance, if the global variable is m_status, modify it to IP_GLOBAL_CONFIG->m_status (refer to line 15 in Fig.8). Although this involves altering numerous lines of code, a simple global search and replace can achieve it. To validate the completeness of these changes, run a testbench compile.

- **Adapt the Singleton IP_GLOBAL_CONFIG to a Multidimensional Array Format**

The last step is to update all DPI calls to include the index of this IP. (Line 14 in Fig 8) With this update the correct instance of singleton can be chosen.

```
1.  class ip global config {
2.     public:
3.       //all original global variables
4.       enum Status e {RESET, INIT, RUN, FINISH};
5.       enum Status e m status;
6.       ~ip global config();
7.       static ip global config *Instance (int index);
8.
9.     private:
10.      ip global config();
11.      static ip global config *ip global config obj[MAX INDEX];
12.  };
13.  ip global config *ip global config::ip global config object[MAX IN-
   DEX] = {NULL};
14. void dpi sample(int ip id, const int I1)
    {
     printf(ip global config::Instance(index)->m status);
     ...
    }
15.  ip_global_config::Instance(index)->m_status = RESET;
```

Fig.8 solution for global variable in DPI. Yellow part is original code

Conclusions

Scaling IP UVCs in SOC testbenches presents unique challenges. By introducing tailored solutions and automation tools, we can streamline the verification process and handle multiple IP instances efficiently. This methodology, backed by successful implementations in flagship projects, paves the way for future advancements in SOC design verification.