Advanced SOC Randomization Tool for Complex SOC Level Verification

Marvin Mei[1] Chris Weller[2] Michael Sedmak[3] Zhiqiang Ren[4]

AMD (19F, North Build Raycom Infotech Park Tower C, No.2, Science Institute South Rd. Zhong Guan Cun, Haidian District, Beijing, China, Marvin.Mei@amd.com, [2]US-Fort Collins-Mile High Design Center, Fort Collins CO. U.S., Chris.Weller@amd.com, [3]US-Fort Collins-Mile High Design Center, Fort Collins CO. U.S., Michael.Sedmak@amd.com, [4]19F, North Build Raycom Infotech Park Tower C, No.2, Science Institute South Rd. Zhong Guan Cun, Haidian District, Beijing, China, Zhiqiang.Ren@amd.com)

Abstract: With the development of modern SOC design, SOC level verification becomes more and more complex and time-consuming. One proposal is to develop an advanced SOC Randomization tool, which can automatically generate random SOC mixed traffic test scenarios based on our predefined constraints. In order to accomplish this goal, this tool must: I) provide powerful constrained random methods, II) automatically generate C/ASM based test cases which SOC Testbench can load during simulation, III) be able to reproduce the same test scenario with the same seed, and IV) avoid recompiling SOC Testbench if SOC DV engineers have to modify or fix test issues.

I. Introduction

To achieve the above goals, we developed an SOC randomization tool, and it includes a standard constrained random class library based on the object-oriented programming language. By using this library, it can randomly generate SOC scenarios based on the input constraints and provide an easy way to create or modify random SOC test cases without recompiling SOC Testbench. This paper has five chapters to describe the following topics:

- The benefits,
- The detailed introduction about SOC Testbench and SOC Scenarios,
- The detailed steps about how to implement this tool,
- And some good examples to provide a better understanding.

A. The Benefits

System Verilog is good at constrained random, and this is the main reason why SV/UVM is very popular for design verification. However, one of the disadvantages of SV is that DV engineers must recompile the whole Testbench if they want to modify or fix the SV test cases. It is not acceptable for SOC level verification as it costs time to recompile SOC Testbench. To solve this challenge, SOC DV engineers prefer to use C or ASM to develop SOC test cases, but all are not necessarily good at constrained random. Table 1 below shows the details, where Point A can significantly save SOC DV cases' debug time and Point B can allow us to develop more complex tests.

Table 1
SYSTEM VERILOG VS. C VS. ASM

	Point A: No Needs to recompile SOC TB?	Point B: Good at constrained random?
System Verilog	No	Yes
C	Yes	No
ASM	Yes	No

An alternative way is to build a standalone SV/UVM based dummy Testbench, which does not include any RTL design. Based on this Testbench, we can quickly develop complex SV based class and sequence lib, compile the dummy Testbench and simulate the standalone test case separately from the whole SOC environment. In this way, the end-user can reuse System Verilog constrained solver to randomly generate complex SOC scenarios based on the input constraints and print out standard C SOC cases. It can avoid recompiling the whole SOC Testbench, but still takes time.

To achieve both Point A and B list in Table 1, we can use the constrained random class library to generate random C and ASM SOC test cases automatically. In this way, SOC DV engineers can develop random SOC test cases without touching the SOC Testbench. Table 2 below shows the details based on our SOC project, and our method is much better than the others.

Table 2

Time to Regenerate the SOC Cases for 3 Different Methods

Methods	Description	Time to regenerate the SOC cases
1	Develop SV/UVM SOC cases under SOC Testbench	~7 hours to recompile the SOC TB
2	Develop standalone SV/UVM dummy TB, and use it to generate SOC cases	~10 mins to compile and simulate the standalone dummy TB
3	This Tool	2~3 mins to regenerate the cases

II. About SOC Level Testbench and Our Interested SOC Scenario

A. *About SOC Level Testbench*

Figure 1 below shows the basic diagram of our SOC Testbench, where:

I) the blue blocks indicate the RTL designs, including real CPU cores, Cache System, Interconnect Fabric with different masters/slaves' ports, and memory controller.

II) purple blocks indicate the Testbench UVCs and Other verification components, such as, CPU core golden reference model, coherent memory model, Master UVC Originators, and UVC DPIs, which can be used to inject traffic from interconnect master ports by C case.

III) the light blue blocks show SOC test cases, one side is for real CPU core, both the boot ROM and Core ASM test cases are compiled into binary files, and then loaded to coherent memory model right after simulation, the other side is for other different interconnect masters, C test case is developed to control the standard UVC DPI, and then dynamically loaded into SOC Testbench during run time.

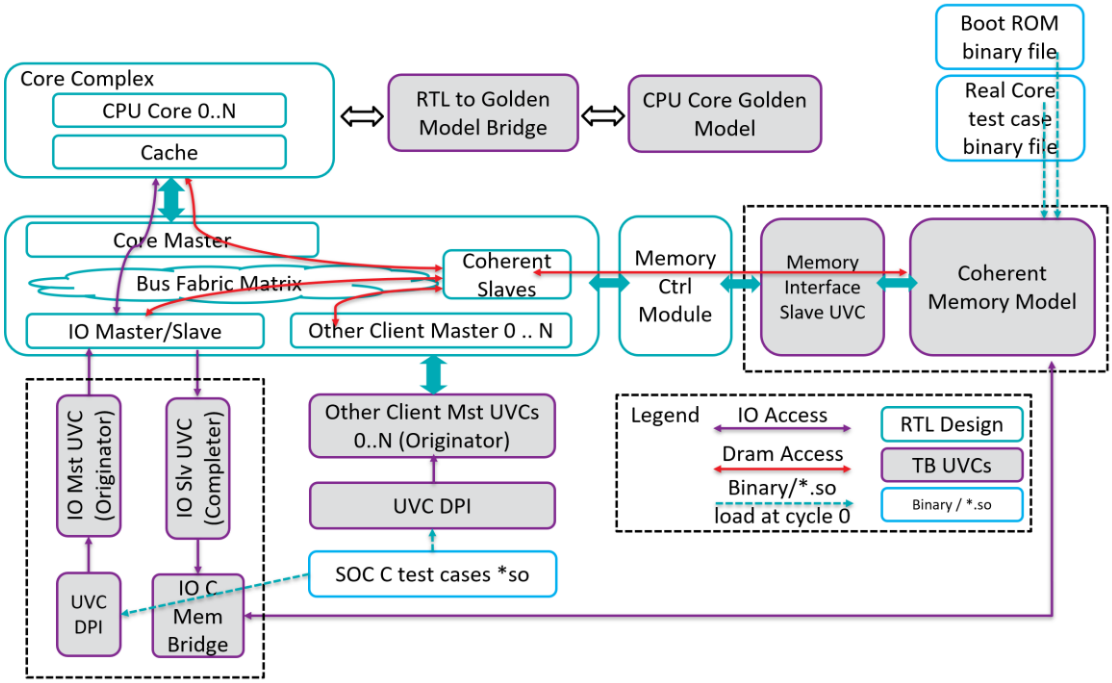

Figure 1. Basic Structure of SOC Testbench

Our SOC verification goal is to develop stress SOC test cases to verify the following four data paths:

- Real Core to IO memory access path and,
- Real Core to Dram Memory access path,
- IO Master to Dram Memory access path and,
- Other Client Masters to Dram Memory access path.

B. *About Our Interested SOC Scenario*

For different interconnect masters, different verification requirements are required to be verified. Table 3 below shows the details. Since it is impossible to manually develop directed test cases to cover all the possible combinations, we hope we can use a common way to parse those different inputs automatically and randomly generate SOC cases. It is the main reason why we want to develop such a tool for SOC level verification.

Table 3

Interested Request Constraints to Different Masters

	Cases	Interested Request Inputs
Real Core Master	ASM Case	• Memory Access Operations (LOAD/STORE/EXCHANGE), with • Operation Sizes (1B/2B/4B…), with • Memory Types (coherent/non-coherent, or cacheable/non-cacheable), with • Address Offset, with • Physical Address Range
IO Master	C Case	• Port Requests (coherent, but non-cacheable read/write), with • Data Payload, with • Address Offset, with • Physical Address Range
Other Client Masters	C Case	• Port Requests ("coherent, but non-cacheable read/write" or "coherent and cacheable Read"), with • Data Payload, with • Address Offset, with • Physical Address Range

Figure 2 shows a classic SOC scenario, and it is one of the good SOC user cases which can be generated by this tool. In this scenario, several masters are enabled to issue massive memory access requests to different, or the same configurable physical address ranges from address 0 to address N with different stepping. For each master's "Whole Traffic", it is divided into several sub-sequences, which include a bunch of memory or IO (for core master) requests, and those sequences are isolated by different "SYNC" macros which are used to communicate with others and get the status from them. Besides, based on various requirements, several different handshake ways are used to make this model more flexible, for example, if master A's SYNC point N is used to sync with master B's SYNC point N, it means all the masters' "traffic" are executed simultaneously. Another example is, if master A's SYNC point N is used to sync with master B's SYNC point N + M, it indicates that master B executes the traffic "M" sequences earlier than master A.

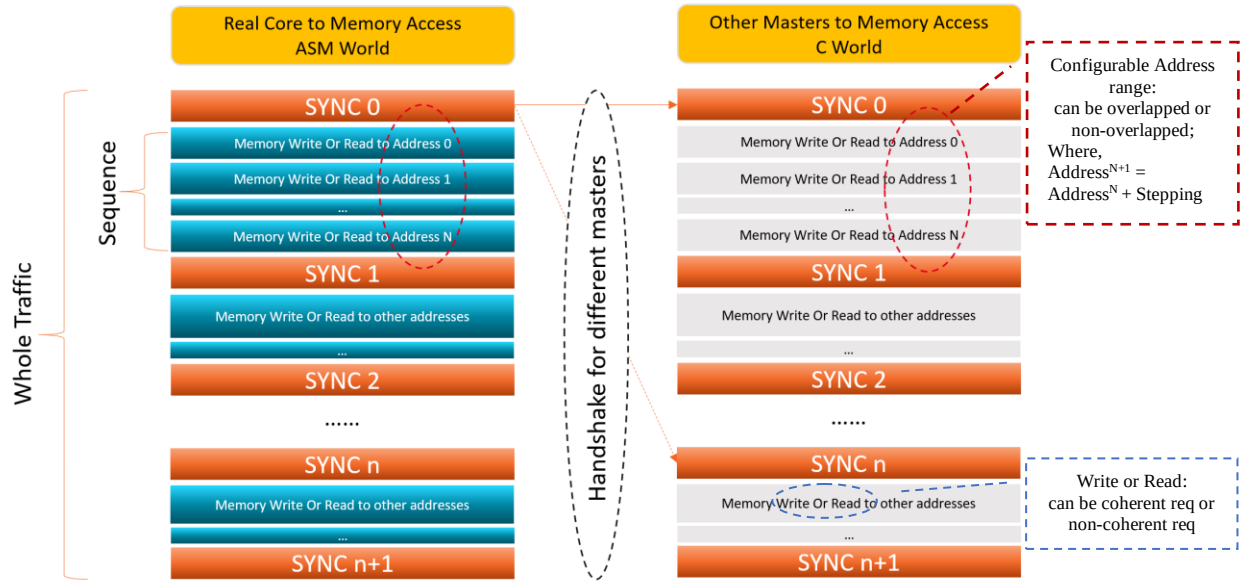

Figure 2. Classic SOC Scenario

III. How to Implement this Tool

A. *Basic Flow*

Figure 3 shows five steps about how to generate the SOC cases by our tool, according to SOC level test plan, we list all the interested SOC feature points, and abstract the traffic constraints for different Interconnect master ports, in step 1, real CPU traffic constraints are input to Core IP level ASM random tool, and output ASM case, and on the other hand, other master clients' constraints are input to this SOC level Randomization tool, and output C case, in order to produce mixed traffic SOC scenario, this SOC tool provides a friendly interface to interact with that Core

level ASM random tool. And then, in steps 2 and 3, our flow calls Assembler to compile ASM case, GCC to compile C case, and output binary/.so files. After that, in step 4, both the generated files are prequalified by CPU Core golden reference model and Master UVC Dummy Testbench separately, then the results are checked in step 4, and only valid and legal cases can be output to SOC Testbench for simulation.

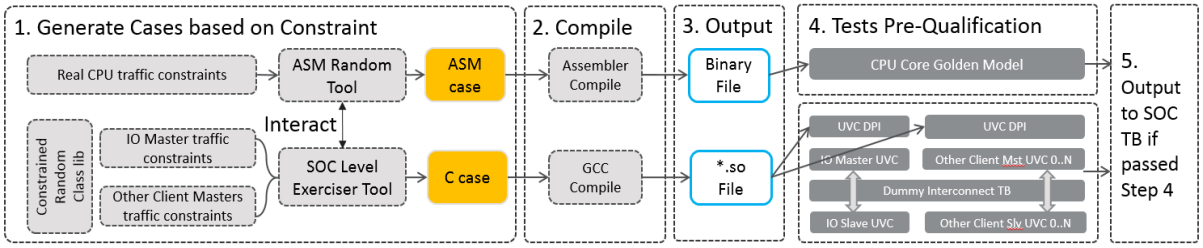

Figure 3. Basic Flow about how to Generate and Qualify SOC Cases

In step 1, we reuse the Core IP level random ASM generation tool, and this tool is capable of creating random assembly test cases as directed by a "generator" class written by the user in the object-oriented programming language. Special commands and routines exist for this class to randomly select and reserve physical DRAM and IO addresses, create page tables and translations, randomly configure threads, and to encode random or specific instructions to be added to the test case. The CPU core golden model is stepped in sync with the generation of the random assembly test case so that the values of architectural registers and memory can be known and referenced by the generation routines.

B. Tool Options

Table 4 below shows the input options which we implemented for this tool, and they can help the end-user to control this tool much more straightforward.

Table 4

Tool Options

Name	Descriptions	Comments
seed	Random seed, which is used to reproduce the same scenario	
project	Indicates SOC project names used to distinguish the requirements for different projects	
num_of_masters	Indicates the number of enabled masters, default is 2, real core master + 1 client master	
num_of_threads	Indicates the number of enabled CPU threads, default is 1, up to 4	
num_operations	The length of "Whole Traffic" shown in Figure 2, default is 800	
num_seq_length	The length of "Sub-Sequence" shown in Figure 2, default is 10	
main_master	Indicates the main master port, which one is going to issue the requests firstly, default is real core master	
handshake_ways	• Mode 0: main master's sync point N -> other master's sync point N • Mode 1 (default): main master's sync point N+M -> other master's sync point N, where M ≥ 0. If num_of_masters > 2, then Way 0 and Way 1 can be easily expanded to: • Mode 0: main master's sync point N -> master 1's sync N -> master 2's sync N … • Mode 1: main master's sync point N+X -> master 1's sync N+Y -> master 3's sync N …, where X ≥ Y.	
address_range_mode	• Non-overlapped: different masters' memory access ranges are non-overlapped • Overlapped (default): different masters share the same memory access ranges	
core_operations	Memory/IO access operations from real Core master	
other_port_operations	Memory/IO access operations from other interconnect masters	
mem_type	For real core master, indicates the default memory type of physical address ranges, default is WB (write back)	
address_stepping	Next Physical Address = Current Physical Address + Stepping: • 0x0: for single line, • 0x20: half cache line size, • 0x40 (default): whole cache line size, • Random	
init_store_data	Initialize data registers before storing, default store value is 0x86	
fake_core_mode	• ON: use fake CPU SOC Testbench, for CPU part's traffic, this tool automatically converts the CPU load/store instructions to interconnect master requests and generate C case only • OFF (default): use real CPU SOC Testbench, this tool generates ASM test case for	

	real core traffic	
self_check_mode	If it turns ON, this tool provides a basic way to • front door or • back door (default), read data back and then do self-compare	

C. The Detailed Implementation Phases

From this section, we will mainly describe Step 1 of the above Basic Flow in detail. As mentioned above, this step is used to parse different input constraints and generate the source codes of SOC cases.

1) **Phase One:** Tool Initialization:

Figure 4 below shows the general steps of this phase, a memory allocation manager from Core IP level's ASM random tool is reused to pick up the tool input options, such as mem_type, address_range_mode, or other different modes, then randomly create, allocate and map the memory access ranges. Besides, two checkpoints are provided to check whether the input options and memory ranges are valid or not. If any check issue is detected, it will fire assertion error and quit the flow immediately.

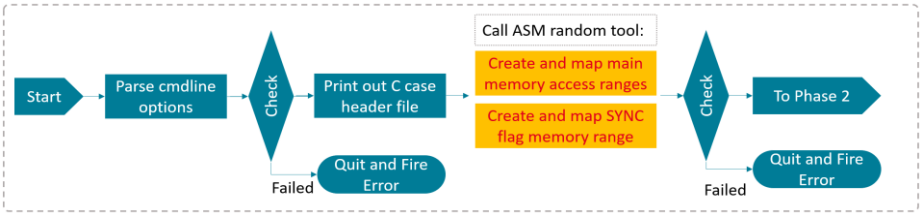

Figure 4. Phase One: Tool Initialization

2) **Phase Two:** Generate SOC ASM Case:

Figure 5 shows how this tool generates SOC ASM case, where the yellow blocks are steps reusing Core IP level ASM random tool functions, they are used to allocate core internal sync macros, do core internal sync with other threads if num_of_threads > 1, and then generate ASM source codes. A globally visible queue is created and maintained in this phase, lots of important information for each core to memory/IO request, such as the physical address, mem type, operation size, sub-sequence index, whole traffic index, and core thread index, are packaged and temporarily stored in this queue. After that, in Phase Three below, those request packages are popped out and used to generate other masters' requests.

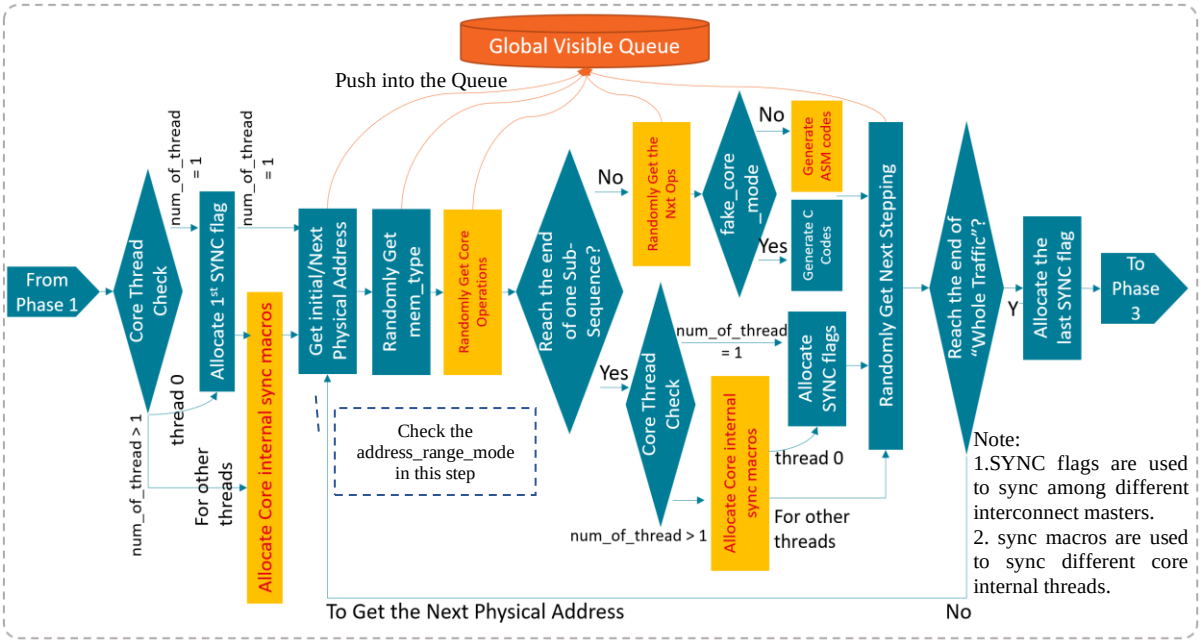

Figure 5. Phase Two: Generate SOC ASM Case

3) **Phase Three:** Generate SOC C Case:

Figure 6 below shows the general steps for Phase Three, it is mainly used to generate SOC C case, a key structure maintained by Phase Two is visible for this phase as well, the expected request packages are popped out strictly in order, then those transactions are used as UVC DPI function inputs and printed out to C file. Besides, this phase provides a sample way to self-compare the final memory data against the expected data if the end-user turns on the self-check mode. However, this way cannot correctly check the data consistency of all the scenarios generated by this tool. Therefore, a better approach is to develop a new SOC scoreboard, but it is beyond the scope of this paper.

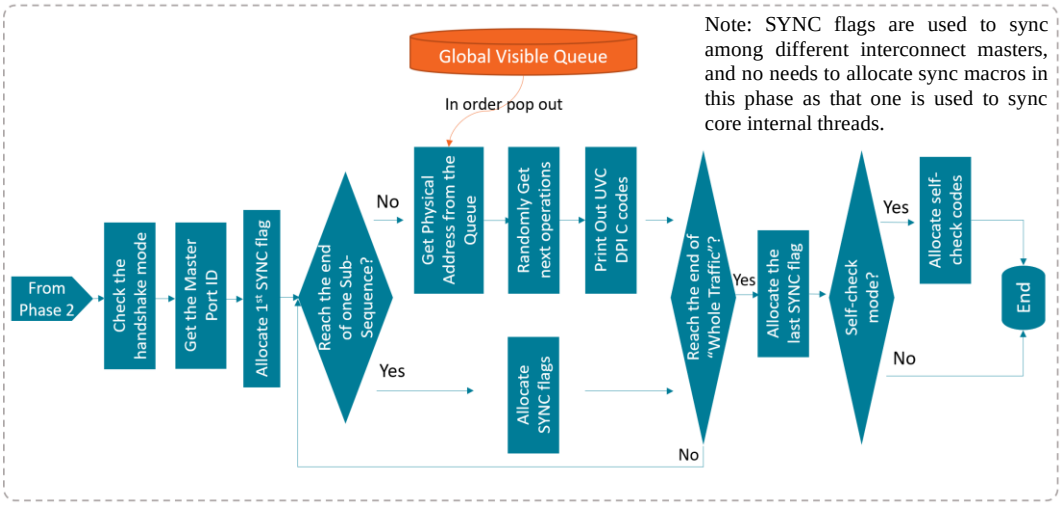

Figure 6. Phase Three: Generate SOC C Case

D. Which language and Why?

There are a lot of object-oriented programming languages, for example, System Verilog, C++, Python, and Ruby. We firstly exclude System Verilog as it is hard to avoid recompiling the Testbench. Due to the following reasons, we finally choose Ruby to implement this tool:

- Ruby is an open-source programming language. It is free to use, copy, modify, and distribute.
- Ruby is a pure object-oriented language, and everything in Ruby is an object, even for an integer.
- There are many useful Ruby-based libraries and examples, which we can take as reference or directly reuse.
- Core IP level ASM random tool is based on Ruby as well, and it is easy to interact with our tool if we use the same language.

IV. Several Good Examples

To better understand this idea, this section shows several Ruby-based code examples, they are all well tested and can be executed if removed the ellipsis dots.

A. Example of the Constrained Random Class Library

There are a lot of useful Ruby Built-In data types, such as arrays, hashes, and ranges, we have enhanced some of them and implemented rich randomization functions. The following Ruby codes show an example about how to improve the constrained random Class library for the "Array" data type, where ellipsis dots indicate the codes omitted from this example. We firstly implemented a function called "rand_with_constraint," which is primarily used to select a legal value based on the input constraints. And secondly, we created an Array called "op_len", and the constraint is "if "option" is 1, then op_len cannot be 0x8 and 0xa". To get a correct value, "op_len.rand_with_constraint" was called, and after 2 rounds of selections, a legal value --- 0x4 was returned, see the "Output log" for more details.

```ruby
#=============================================
class Array # :nodoc:
...
  #---------------------------------------------
  # Select a legal value based on our constraint.
  # Returns nil on failure
  def rand_with_constraint(rng=$rng, &constraint)
    randc_self = self
    puts "Got #{randc_self} with size = #{randc_self.size}"
    while (randc_self.size != 0)
      # Randomly pick up one value from array
      idx   = rng.rand(randc_self.size)
      value = randc_self[idx]
      # Delete it from array
      randc_self.delete_at(idx)
      puts "Got value #{value} with idx #{idx}, deleted it, and new array is #{randc_self}"
      # Return value if legal
      if constraint.call(value)
        return value
      end
    end
    return nil # failed, return nil
  end
...
end

# Select a random value
$rng = Random.new(1000)
# Example about our constraint random lib
op_len = [0x4, 0x5, 0x8, 0xa, 0x1, 0x6]
option = 1
op_len = op_len.rand_with_constraint do |len|
                 if (option == 1)
                   (len != 0x8) && (len != 0xa)
                 else
                   len != 0x4
                 end
               end
puts "Successfully got one valid op_len #{op_len}"
```

Output log

→ruby example.rb
Got [4, 5, 8, 10, 1, 6] with size = 6
Got value 10 with idx 3, deleted it, the new array is [4, 5, 8, 1, 6]
Got value 4 with idx 0, deleted it, the new array is [5, 8, 1, 6]
Successfully got a valid op_len 4

B. Example of How to Reproduce the Same Scenario with the Same Seed

The critical issue for a random tool is how to guarantee it can reproduce the same SOC case with the same seed. A straightforward way to ensure the repeatability of this tool is to create a globally visible variable --- "seed", and then we make sure for each random function, this tool explicitly inputs that global "seed" and uses it as the unique reference to randomize the data results. Figure 7 shows the details.

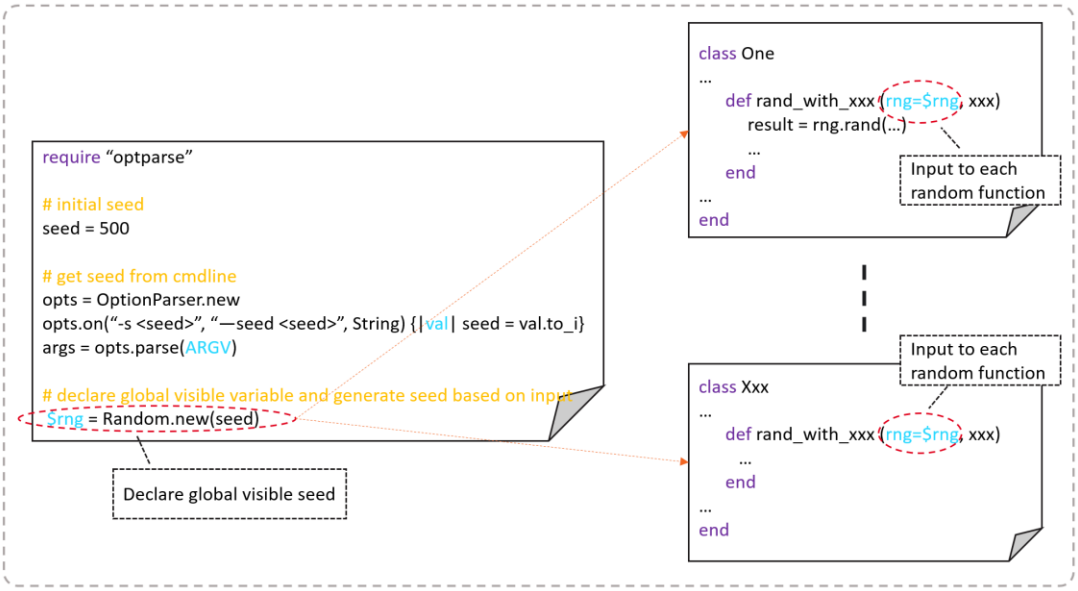

Figure 7. Globally Visible Seed

C. Example about how to Generate Interconnect Master Port's Request

Since all the Master Ports of Interconnect IP reuse the same bus protocol, we can implement a standard class to describe the master request, and the followings are parts of the request fields:

a. Command: Request Commands, i.e., Coherent/Non-coherent reads, Coherent/Non-coherent writes.
b. Address: System Address.
c. Request Length: This field specifies the number of doublewords transferred by one request.
d. Unit ID and Virtual Channel ID: Port ID number and Virtual Channel ID number.
e. Byte Enable and Data Payload used for Write requests only.

Each of the above request fields must follow different constraints. The example below shows how this tool generates a master request transaction based on the input constraints, where ellipsis dots indicate the codes omitted from this example. Here, we firstly declared a class called "Mst_UVC_Req_Cmd". In this class, a Ruby built-in function "initialize" was implemented to get the Class global input parameters, and another function called "gen_c_case()" was applied to parse different constraints for different request fields. Then if no constraint failure, it prints out the standard UVC DPI function to C case. When we want to generate a random master request, the only step is to create a request object, and then call "obj.gen_c_case" function.

V. Results and Future works

By using this tool, SOC DV engineers can quickly develop random SOC test cases to cover complex SOC scenarios without recompiling the whole SOC Testbench. This tool has been deployed to a current SOC project and demonstrated that it could shorten SOC DV's debug time, for example, we can regenerate SOC random case in only 2 or 3 minutes. The next goal is to enhance this tool to better support self-check mechanism or other interested SOC scenarios, such as to randomly insert interrupt, power management transition events or system management commands to the "Whole Traffic" sequence.

Acknowledgment

I would like to express my gratitude to my wife, my daughter and all others who have helped me during the writing of this paper, and gratefully acknowledge the help of Chris Weller and Michael Sedmak from the AMD IP team. I do appreciate their significant contribution to this paper.