System-Level Random Verification: How it should be done

Madhusudan Rathi
Senior design Engineer, Analog Devices India
Madhusudan.Rathi@analog.com
+91-8197233788

Ashok Chandran
Engineering Manager, Analog Devices India
Ashok.chandran@analog.com
+91-8043002231

***Abstract-* Due to increasing SOC development cost and lower digital gate cost on silicon, Silicon industry is moving towards more general purpose and configurable designs. Aim of developments is to support wider range of applications using same silicon to increase the return on investment (ROI). This trend is resulting in increasing number of use cases and many unknown future applications for the same SOC. Verification of such design is a significant challenge.**

Constrained random verification (CRV) is a standard approach for block level verification to hit multiple known and unknown scenarios, uncover corner cases and find critical design bugs. However, CRV approach is not widely used approach for system level verification. One primary reason is extensive sets of constraint, due to many invalid scenarios present at system level. In addition, there are complex relations and dependencies among blocks at system level, which require constraints debug very complex. Over all constraint writing and debugging constraint failures are excessively difficult at system level.

We had such verification challenge for an SOC. The Design had ~2000 known use cases and many more possible use cases for future. We have adopted CRV approach to verify such highly configurable and complex data path design at system level. With the systematic approach and planning, we have achieved quality verification in reasonable time. We will discuss the learning from such an activity in this paper.

INTRODUCTION

We experienced the challenge of verification of complex and configurable data path design as shown in Fig 1. The design has many programmable muxes and crossbar designs, which enhance flexibility to the design. There are approximately 2000 known use cases. Each use case can result in average 10 unique configurations of design. With so many use cases, verification using directed test approach becomes extremely inefficient.

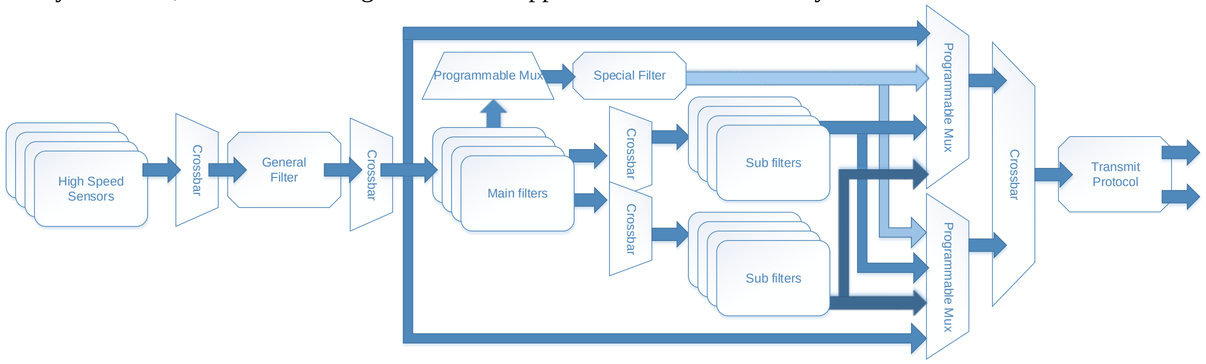

Figure 1.datapath design with programmable muxes and crossbars

Configurable crossbar and programmable mux designs are very much suited for property based formal verification approach. But our data path design has complex clocking scheme. Complex clocking present in design requires many complex properties and huge computation power to prove them. Big data path design requires long time to prove a property, and the formal tool was frequently broken. Complex clocking scheme require gate level simulation to catch synthesis constraints and multicycle issues. Very early, we realized that our design wasn't suitable for formal verification and need to look for alternative approaches.

We have deployed to a UVM based CRV approach for system level datapath verification. In following sections, we will discuss in detail, about the approach used to write system constraints and the experience/learning from system level constraining.

A. Constraint Macros:

By looking at the data path diagram, one can guess there are lots of similar constraints are required. For avoiding typo or copy paste mistakes, we have started creating macros for standard or commonly required constraints. One can review the macro definition, and rest of team will use the macros, without worrying about implementation. It makes the code easy understandable and reusable. Any issue with constraint macro can be solved without editing multiple places. Constraint for instances and size, enables and size, one hot enable are some example of frequently required constraints.

```
`size_const(a_inst,a_size);
`en_const(a_size,a_en);
`onehot_constraint(a_en);
```

B. Understanding SV Language Limitations:

There are some System Verilog language limitations which need to be understood. For Example, if we have a random array of bits and we need to constraint number of one's present in that array.

a) If we select unpacked bit array and use array.sum() function, it does not work. Since return type of sum() function is same as array element type, the sum of bit array will always return a bit and not an integer.
b) $countones(array) is another approach to get sum of bits, but it works only for packed array. But the Packed array randomize all elements together, so we cannot selectively disable/enable rand mode for each array elements.

C. Avoiding Ambiguous Constraints:

Some Statements looks same but have different functionalities. Statement (A->B) and (A==B) looks very much same. Functionality of (A -> B) equals (!A)|B, while (A==B) equals (A&B)|(!A&!B). We need to cautiously select correct statement to avoid invalid scenario generation and failure debug.

There are some syntax which are confusing and functionality is tool dependent. Says A== B|C, tool can assume as

A == (B|C)   or   (A==B) | C.

Another Example

```
(A==2) | ((B==1) & (C==0)) |
(A==3) | ((B==1) & (C==1))
== (D == 1)
```

Tool may take it as a) or b)

```
a)  ((A==2) | ((B==1) & (C==0)) |
    (A==3) | ((B==1) & (C==1)))
    == (D == 1)
b)  (A==2) | ((B==1) & (C==0)) |
    ((A==3) | ((B==1) & (C==1))
    == (D == 1))
```

Such constraints are required for crossbar randomization. While both looks same but functionality is very different based on the parenthesis. It is very difficult to debug such issues at system level in a pool of constraints, and we should apply the correct parenthesis while writing constraint itself, based on intends or requirements.

D. Managing Constraint Solver Speed:

Soon after some constraints are ready, we observed that our system transaction randomization consuming long time. We have to split the fundamental transaction into multiple sub transactions. Partition of transactions was done based on functionality and dependency. We kept separate transactions as following.

1. Protocol transaction: Transmission protocol related parameters like number of transmit lanes, number of transmit links etc. are randomized in this transaction.
2. Datapath transaction: Main data flow parameters are randomized here. After randomization, we will get to know that which all blocks are present in the datapath, how many instances are enabled in the datapath for other blocks, which block is going to receive data from which blocks and etc.

3. Block transaction: Each block may have some parameters which do not impact overall data flow equations but impact other blocks presents in the path. Such parameters are randomized here. Few examples are Filter coefficients, decimations and output data rates.
4. Misc transaction: non datapath parameters like GPIO, interrupts, memory allocations are randomized here.
5. Frequency transaction: Overall system frequency planning related parameters are randomized here. Input band, Mixers modes, mixer frequencies and similar parameters are considered here.

By partitioning the transactions, overall system randomization time is considerably reduced. We have seen ~30 times the performance improvement of tool randomization (system level randomization time reduced to ~1 minute from ~30 minute). Partition of transactions has reduced the overall verification time.

1. Debug time reduced due to partition debug capability improves.
2. It became ease for the code update, if any design/functionality changes during verification cycle.
3. Making the code reusable for next design.

E. *Understanding the solution space:*

Large number of constraints result in frequent tool crash and/or long randomization time. We can reduce such issues to some extend by rewriting constraints in tool friendly ways. Say, for each selected instance, we should have enable index one. For such condition, constraint writing will be as following.

```
foreach(a_inst[i]){a_en[a_inst[i]] ==1;}
```

But it creates order dependency for tool and tool need to solve a_inst before a_en. If we think anew, we can restate condition as each enables the index one means mentioned instance is selected. It makes constraint become the tool friendly by removing order of randomization and result in fast randomization.

```
foreach(a_en[i]){a_en[i] -> I inside {a_inst};}
```

Consider another example, if size is more than one than A and B should be equal and we can code it like below.

```
(size > 0) -> (A==B);
```

But since size is the integer type, there are many value possible and take time for the simulation tool to analyze it. We can rewrite it as if enable than A and B should be equal. Here enable (en) is a bit, so the simulation tool can have only two possible cases.

```
(en) -> (A==B);
```

As the solution space reduces, simulation tools can randomize fast. Minor changes in code can help us in reductions of total randomization time at system level.

F. *Optimizing for Solver Performance:*

On further simulation tool performance improvement side, we found that some tools have optimization for standard constraint statements. Consider an example of instance array of any block. We require no repetition of the enabled instance, and each enabled instances should not be more than max possible instances.

```
foreach(a_inst[i]){
        if(a_en[i]) -> (a_inst[i] <= max);
        foreach(a_inst[j]){
                if((i!=j)&(a_en[i])&(a_en[j])) a_inst[i] != a_inst[j];}
        }

}
```

We can achieve same functionality by using unique function as below and reduce randomization time.

```
unique(a_inst);
foreach(a_inst[i]){(a_inst[i]  > max) -> a_en[i] ==0;}
```

Furthermore we observed that some time redundant constraints help in reduction of tool crash. In the below example, Say mux output a[i] can be selected among b[i], c[i] or d[i]. So a[i] is ored of b[i], c[i] and d[i] and only one out of b[i], c[i] and d[i] is enabled. We added size of a instance equal to sum of size of b, c and d instances. So while constraint a) is sufficient, constraint b) is redundant constraint and helps tool to randomize transactions.

```
a)  foreach(a[i]){a[i] == b[i] | c[i] | d[i]; `onehot_constraint({b[i],c[i],d[i]};);
b)  a_size == (b_size+c_size+d_size);
```

G. Constraint Reusability:

Multiple times, we have to enable or disable the constraints for the given use case or the test scenarios. Having multiple constraint blocks will make easy to control for various combinations from the test or the test bench top and hence improve reusability. On other side, keeping one constraint block for similar functionality, will reduce risk of missing some constraints. It also reduces development time and failures. Therefore, it should be balanced division. We have organized design constraint in way data flows in data path. We kept all constraints related to same design block or interface together. It reduces debug time and makes it easy to adopt any design/functionality changes.

H. Naming Conventions:

At system level verification, we will get request to generate very specific use case scenario for development, review, silicon evaluation or customer support. We do not want to write separate tests for such request every time.We have provided hooks to remove randomization for the variables and/or reduce randomization to create given use case scenario. We have provided run options and config_db to pass such commands. In pre randomization, we implemented check for any such command for rand_mode disable/enable for random variable and constraint blocks and take action accordingly. Moreover, we have created few constraint blocks for frequently required cases. These constraint blocks were disabled by default and enabled only on request via command. Run-time options serve us in saving compile time for each use case and provide the way to use one test to create/run multiple use cases. It also provides quick and easy way to debug randomization failures or functional coverage holes.

I. Run-time control of Constraints:

At system level verification, we will get request to generate very specific use case scenario generation for development, review, silicon evaluation or customer support. We do not want to write separate test for such request every time. We have provided hooks to remove randomization for the variables and/or reduce randomization to create given use case scenario. We have provided run options and config_db to pass such commands. In pre randomization, we use check for any such command for rand_mode disable/enable for random variable and constraint blocks and take action accordingly. Also we have created few constraint blocks for commonly required cases. These constraint blocks were disabled by default unless enable requested via command. Run time options help us in saving compile time for each use case and provide way to use one test to create/run multiple use cases. It also provides quick and easy way to debug randomization failures or functional coverage holes.

J. Constraint Input Check to avoid invalid states:

While run option based commands became handy to use, many will provide wrong or invalid commands by unknowingly. All such commands result in conflicting constraints and randomization failure. These randomization failures are significantly hard to debug for others, and they will start complaining about complexity of system level debug. Certain conditions are invalid according to design or constraint. Such conditions are better known at time of constraint writing. We have added check for such conditions in pre randomization. Which give very specific failure massage about conflicts. It is considerably painless to debug with such failure message than debugging of conflicting constraints failures.

K. Rigorous Functional Coverage:

We cannot be sure about completion of verification without proper functional coverage matrix. Functional coverage measures quality of constraints and give good confidence. Also functional coverage hole will help in finding over-constraints or wrongly constraint condition/block. We have used run options based force randomization approach to easy finding of such conflicts.

SUMMARY/RESULTS

1. There are significant challenges in system level randomization and require deep planning and systematic approach to complete it.
2. We have successfully used system level randomization approach to achieve our verification goals. We have unearthed many scenarios and design bugs using RTG, which were not thought before.
3. Silicon is returned from fab and looks healthy. No issue reported by the silicon Eval team in data path till now for the project.

4. Run option base use case scenario generation is extensively used for silicon debug and accelerate overall time for silicon bring up.

As a whole, deployment of constraint random verification throughout the project from block to system provides significant automation of the verification process. This helps in speeding up the verification and results in optimal trade-off of engineer time vs compute resources in verification.

References