System level random verification: How it should be done

Madhusudan Rathi
Senior Design Engineer, Analog Devices India

Ashok Chandran
Engineering Manager, Analog Devices India

Agenda

- Introduction
- Constraint development
- Tool performance optimization
- Control and debug
- Guideline for reuse
- Results and summery

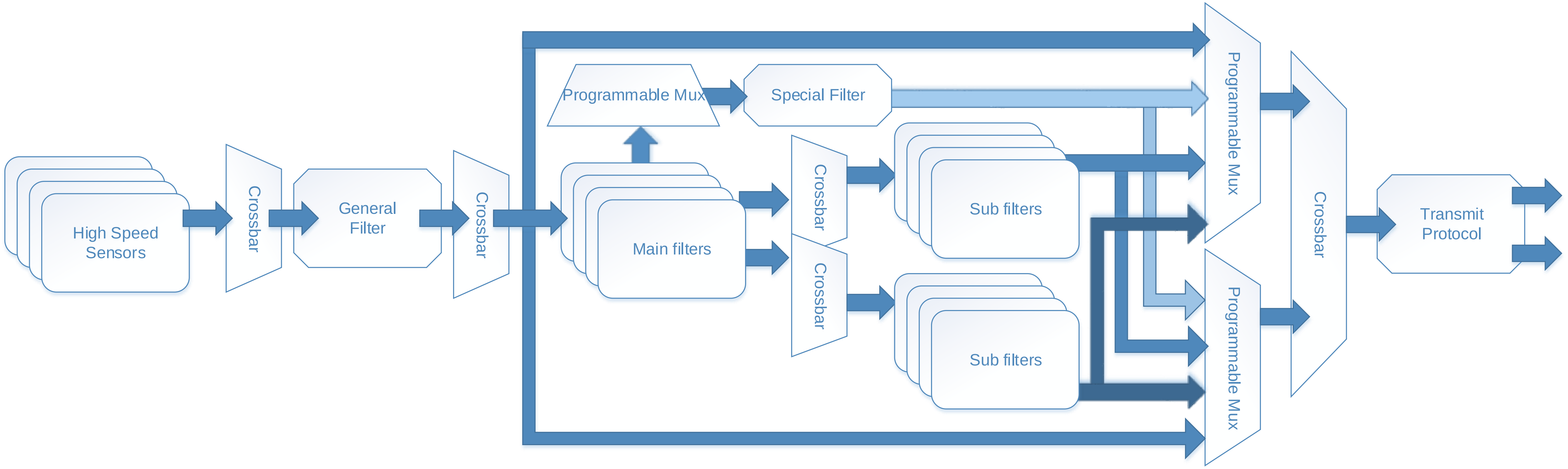
Introduction

- Industry trends
  - Trends reveal more general purpose configurable designs.
  - Support broader range of applications with the same silicon.
  - Increasing number of use-cases and many unknown future applications.

- System level verification
  - Typically restricted to connectivity checks between sub-blocks.
  - Additionally, use-cases scenario verification using directed test approach.

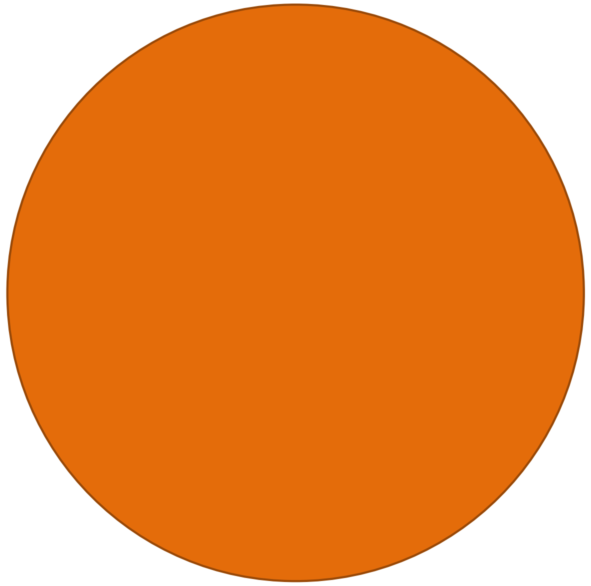
- Constraint random verification(CRV)
  - Widely accepted standard approach for block level verification.
  - Helps in exercising known/unknown scenarios.

Design block diagram



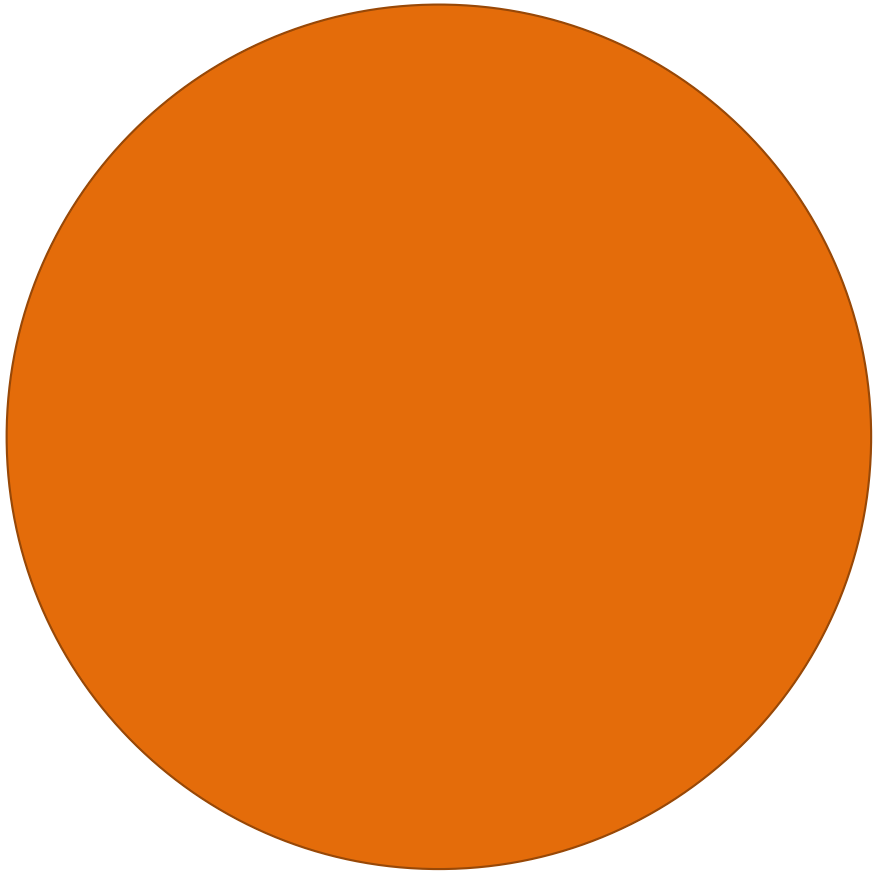
Need for system level randomization

- ~2000 major known use-cases and many more possible use-cases for the future.



Need for system level randomization

- ~2000 major known use-cases and many more possible use-cases for the future.
- Each use-case results in 10 unique possible combinations of data flow configuration.



Need for system level randomization

- ~2000 major known use-cases and many more possible use-cases for the future.
- Each use-case results in 10 unique possible combinations of data flow configuration.
- Each data flow configuration have the 100s of other parameters for randomization.

Need for syste[illegible] randomi[illegible]

- ~2000 major known use-cases and many more possible use-cases for the future.
- Each use-case results in 10 unique possible combinations of data flow configuration.
- Each data flow configuration have the 100s of other parameters for randomization.
- Required end to end checks for each configuration for signal integrity and deterministic latency.

More than 20,000,000 known design configurations to verify at system level

Need for system level randomization

- For massive number of known use-cases, directed test approach is extremely inefficient.
- Complex data path clocking scheme makes DUT unsuitable for formal runs.
  - Require complex properties and huge computation power.
- Signal processing and integrity checks for data path.
- Requirement for gate level simulation to catch synthesis constraints and multicycle issues.
- System level CRV approach is a must for thorough verification.

Challenges with CRV approach at system level

- Development
  - Extensive sets of constraints.
  - Require systematic planning.
- Execution
  - Huge solution space for constraint solver.
  - Significant debug efforts.
- Sign off
  - Adaptation to late design changes and new use-cases.
  - Appropriate coverage bins for functional coverage.

Challenges with CRV approach at system level

- Development
  - Extensive sets of constraints.
  - Require systematic planning.
- Execution
  - Huge solution space for constraint solver.
  - Significant debug efforts.
- Sign off
  - Adaptation to late design changes and new use-cases.
  - Appropriate coverage bins for functional coverage.

How much of the development efforts will be reusable in future projects?

Agenda

- Introduction
- ☛ **Constraint development**
- Tool performance optimization
- Control and debug
- Guideline for reuse
- Result and summery

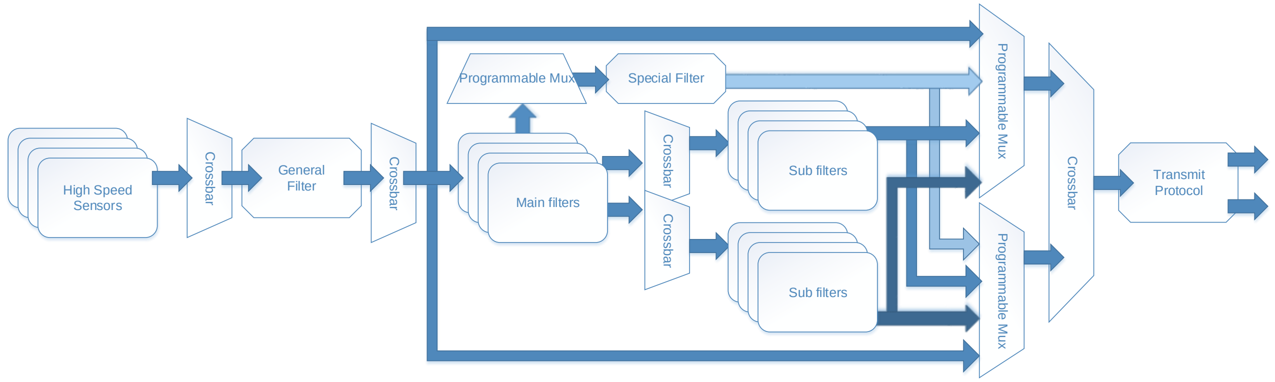
Use of constraint macros

```
`size_const(a_inst_arr,a_size);

`en_const(a_size,a_en);

`onehot_const(a_arr);
```

Number of enabled instances should match with size variable.

Size should be zero, if disabled.

One hot constraint for mux select



Use of constraint macros

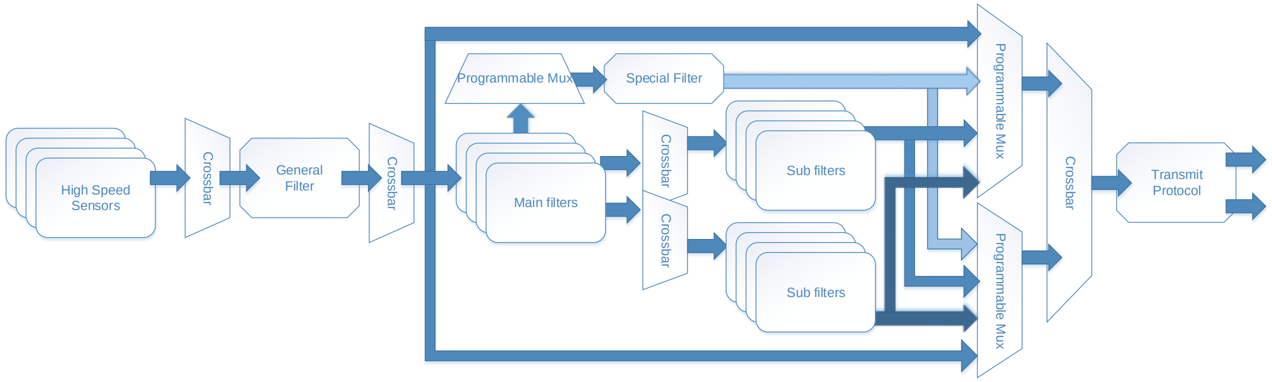
```
`size_const(a_inst_arr,a_size);

`en_const(a_size,a_en);

`onehot_const(a_arr);
```

Number of enabled instances should match with size variable.

Size should be zero, if disabled.

One hot constraint for mux select

– Platform approach for uniformity in coding.
– Avoid typo and copy paste errors.
– Easy to update and audit.



Recognize SV language limitations

```
rand bit a;
rand bit b;
constraint a_to_b {a -> b;}
constraint a_to_b {a == b;}
```

Logical equivalent to (!a)|b

Logical equivalent to (ab)|(!a!b)

a	b	a -> b	a == b
0	0	True	True
0	1	True	False
1	0	False	False
1	1	True	True

Recognize SV language limitations

```
rand bit a_inst_arr[a_max-1:0];
a_size == a_inst_arr.sum();
a_inst_arr[n].rand_mode(0);
```

Unpacked array of random bits.

☒ Sum function will return bit and not integer type.

Control for individual element works.

```
rand bit[a_max-1:0] a_inst_arr;
a_size == $countones(a_inst_arr);
a_inst_arr[n].rand_mode(0);
```

Packed array of random bits.

$countones return integer value.

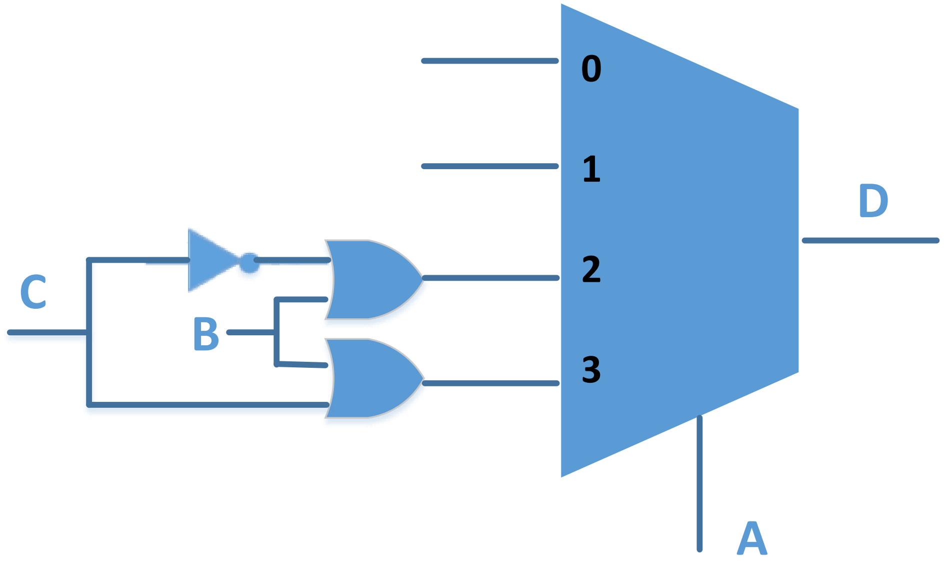
☒ Not possible for packed array.

Understanding of language limitation will enable reduction of global development and debug time.

Avoid ambiguous constraints

Looks good !!

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



Avoid ambiguous constraints

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

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));
```

- Are you able find the difference between the two?
  - X | Y == Z means (X | Y) == Z or X | (Y == Z)
  - Both are correct and interpretation depends on tool.
- Can we debug such issues quickly?
- Eliminate ambiguities with proper brackets as per intends.

X	Y	Z	(X | Y) == Z	X | (Y == Z)
0	0	0	True	True
0	0	1	False	False
0	1	0	False	False
0	1	1	True	True
1	0	0	False	True
1	0	1	True	True
1	1	0	False	False
1	1	1	True	True

Agenda

- Introduction
- Constraint development
- ☛ **Tool performance optimization**
- Control and debug
- Guideline for reuse
- Result and summery

System level constraint issues

- Large solution space for system will consume significant computation power.
  - 100s of parameter to randomize.
- Inter blocks dependencies for valid combinations.
  - Like number of monitors depend on sensors count.
- Implementation complexity/trade off driven design constraints.
  - Ex. Max. data rate at filter input.
- Frequency planning for valid outputs.
  - Depends on input bands and mixers.

Divide and conquer

- Partition of randomization based on functionalities and dependencies.
  - Protocol and external interaction (transmit lanes, lane rates…).
  - Data path, crossbar and muxes (number of filters, data source & destination…).
  - Frequency planning for system (input bands, mixer frequencies…).
  - Non data path parameters (GPIOs, interrupts…).
- Implementation details
  - Individual, layered transactions for each partition.
    - » Used layered approach for reuse.
  - Transactions are randomized and passed to rest of transactions.
    - » System sequence will order randomization among transaction and share as required.
- Result
  - See ~30 times performance improvement.
  - Help in debug for randomization failures.
  - Easy to update and maintain.

Understanding the solution space with examples

Aim: do some constraint, if any enabled

```
rand bit a_en;
rand int unsigned a_size;
.
(a_size>0) -> constraint_statement;
(a_en) -> constraint_statement;
```

☒ Integer has many possible values.

☑ Bit has only 2 possible values.

Understanding the solution space with examples

Aim: do some constraint, if any enabled

```
rand bit a_en;
rand int unsigned a_size;
.
(a_size>0) -> constraint_statement;
(a_en) -> constraint_statement;
```

☒ Integer has many possible values.

☑ Bit has only 2 possible values.

Aim: randomly selected instances should be enabled

```
rand int a_inst_arr[];
rand bit a_inst_en[a_max];
.
foreach(a_inst_arr[i]){
      a_inst_en[a_inst_arr[i]] == 1;}
.
foreach(a_inst_en[i]){
      a_inst_en[i] -> i inside{a_inst_arr};}
```

☒ Order dependency for tool, may result in frequent tool crash.

☑ Tool friendly due to removing order for randomization.

Optimizing for the solver performance

- Recommendations
  - Split system randomization in multiple steps.
  - Search for possibilities in solution spaces reduction.
  - Avoid indirect dependencies by rewriting constraints.
  - Provide guidance to solver for order of randomization.
  - Tools may have optimization for standard constraint statements.
  - Redundant constraints may also help in performance improvement.

Solution space (not the number of constraints) determines randomization time for the tool.

Agenda

- Introduction
- Constraint development
- Tool performance optimization
- ☛ **Control and debug**
- Guideline for reuse
- Result and summery

Run-time control over constraints

- Problem
  - Multiple request for the specific use-case scenarios during the project.
  - Separate test for each request will be overkill.
  - Require control over random variables and constraint blocks.
- Solution
  - Enable or disable randomization in pre randomization phase.
  - Use $plusargs based run-time options and uvm_config_db to receive commands.
    - Sample commands: +set_sensor_mode=1 +set_main_filter_size=2 +set_bypass_path
    - Ease to update and native method to provide support for various commands.
    - Use macro defines for uniformity in run-time options.
  - Separate constraint blocks for commonly required cases.

Run-time control over constraints

- Run-time options based control became savior.
  - Used by architecture, design, verification and silicon eval teams.
  - Used for development, verification, mix signal cosim, coverage closure, power analysis, silicon debug and use-cases run.
- Result
  - Help in serving many requested use-cases without writing alternative tests.
  - Save compile time for each required use-case generation.
  - Verification easiness
    - Used to narrow down cause for randomization failures.
    - Identifying constraints for the required functionality.
    - Functional coverage analysis.

Functional coverage debug

- Problem
  - Functional coverage holes.
    - Indicates given scenario is not covered.
- Cause
  - Overly or wrongly constrained conditions and blocks.
  - Insignificant runs for functional coverage sampling.
- Strategy to debug
  - Use run-time options to force required combination.
  - Conflicting constraint failure from tool help in identifying possible constraint issues.
  - If no constraint failure means more runs are needed.

Conflicts and invalid conditions

- Multiple times the bunch of commands result in invalid combinations.
  - Result in randomization failure due to conflicting constraints.
  - Significant unproductive efforts for various teams to understand and debug such failures.
- Early detection of run-time commands issue
  - Error and conflicting conditions are known upfront to developers.
  - Put conditional checks for them before randomization.
    - Use rand_mode() return value for identification of run-time options.
  - Result in very specific failure message for such conditions.

Agenda

- Introduction
- Constraint development
- Tool performance optimization
- Control and debug
- ☛ **Guideline for reuse**
- Result and summery

Constraint reusability concerns

- Identifying constraint for the functionalities in “a sea of constraints”.
- Refurbishing constraints for design changes.
- Missing constraint statements and blocks.
- Constraint block override.
  - Language supports and useful in few cases.
    - » Extended class can override base class constraints.
  - Extremely error prone, if not intended.
  - Difficult to avoid issue without upfront planning.

constraint block will be fully ignored by tool.

It will override previous constraint block.

```
constraint abc_const{
   condition_1;
}
.
.
constraint abc_const{
   condition_2;
}
```

Guideline for reuse

Guideline	Improvement
One constraint block for similar functionalities.	• Reduce risk of missing constraints. • Ease updating of code for design changes.
Name each constrained block using the dependent variables and specific functionality.	• Avoid constraint blocks override issue. • Help in finding
Ordering the constrained blocks as per order of programming or data flows in design.	• Mitigate missing constraints issues. • Promising way to debug and search constraints.
Use enums and macros instead of hardcoding value	• Easy understanding and debug for constraint. • Require minimum change for spec changes
Appropriate comments before constrained blocks and code.	• Helps to understand and reuse. • Effortless debug for constraint issues. • Provide templates for similar functionality.

Efficiency in finding & understanding

$\Rightarrow$ **Helps in update & debug**

$\Rightarrow$ **Enable wider adoption & reuse**

Agenda

- Introduction
- Constraint development
- Tool performance optimization
- Control and debug
- Guideline for reuse

☛ **Result and Summery**

Results

- **Challenges:** There are significant challenges in system level randomization and require deep planning and a systematic approach to complete it.
- **Quality within time:** We have successfully used a system level randomization approach to achieve our verification goals
  - Uncovered 60+ bugs in system before tape-out.
  - We have unearthed many scenarios and design bugs using CRV, which were not thought before.
  - Extensive silicon testing did not report any issues.
- **Impact:** Run-time option based use-case scenario generation is extensively used for silicon debug and decreases overall time for silicon bring up.

Summery

- System level CRV approach provides <u>significant automation</u> to verification process.
- <u>Optimal trade-off</u> between Engineer time vs Computer resources.

Thank You !!

Thank you !!

Backup slides

System level CRV approach v/s portable stimulus

- Limited advantages
  - Due to aggressive schedule and insignificant returns, block level approach is not used in project.
  - Other teams (software, silicon eval) were not ready to invest efforts.
  - PS was not industry standard and hence had doubt about reusability in subsequent projects.
- Immature tool support.
  - A couple of PS tools were evaluated and experienced inadequate results.
  - Anticipated various tool issues and time consuming tool debug during project cycle.
- Significant efforts
  - PS require significant efforts for team to learn and understand.
  - SV is well known and minimum efforts required for team to understand, develop and debug.
  - Since we need to code large number of constraints in limited time, native approach is preferred.

Finally we are working to bring up PS setup for the project.

Examples for standard constraint statements

Aim: randomly selected instances should be unique and enabled.

```
rand int a_inst_arr[];
rand bit a_inst_en[a_max];
.
foreach(a_inst_arr[i]){
   if(a_inst_en[i]) -> (a_inst_arr[i] <= a_max);
   foreach(a_inst_arr[j]){
       if(i!=j) & a_inst_en[i] & a_inst_en[j]{
          a_inst_arr[i] != a_inst_arr[j];}
   }
}
.
Unique(a_inst_arr);
foreach(a_inst_arr[i]){
      (a_inst_arr[i] > max) -> a_inst_en[i] == 0;}
```

☒ Complex loops create issue and suffer with tool performance hit.

☑ Tool may have optimization for standard constraint blocks.

Examples of redundant constraint

Aim: each mux output should have respective one and only one input.

```
rand int a_size,b_size,c_size,d_size;
rand bit a[max],b[max],c[max],d[max];
.
foreach(a[i]){
   a[i] == (b[i] | c[i] | d[i]);
   `onehot_constraint({b[i],c[i],d[i]});
}

a_size == (b_size + c_size +d_size);
```

Constraints for mux output and only one selection for input.

☑ Redundant constraint on size to help tool.