Synopsys Users Group

Unified Verification Framework Automation and Test Standardization with UVM

Bin Liu, Libo Tian, Yu Guo, Yan Wu, Yongqi Sang

Baseband SoC Development Group

Intel Inc.

Xi'an, Shaanxi, China

www.intel.com

ABSTRACT

As the SoC design becomes more complex and larger, the verification effort has been mainly affected by several factors. Those factors include the verification completion standard, verifiers' experience and verification flow harmonization. If the verification strategy took more dependency on verifiers' experience, the company would call for more training and professional working way. However, in most of cases, a large SoC verification team is composed of different level of skilled engineers. Comparing spending more into training, directing new hands to a unified verification methodology would help building general verification mindset. Several years past, now the Universal Verification Methodology (UVM) has been the main stream verification methodology. The verifiers mastered standardized means to build test framework, but it could not well satisfy project schedule. This paper introduced a test framework automation tool, which is based on UVM and Verification IP (VIP) and to serve higher level environment automation. After the mechanism is given, the paper gave the project application case and the benefits from this tool.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

It is helpful with a unified language and methodology to be applied in verification, and verifiers are no more worried about which methodology to choose and the best suitable simulator. In fact, the methodology reconciliation took over a decade, and now the verifiers as beneficiary could focus more on the design verification itself instead of methodology choices. From the methodology evolution progress, it inspired the verification industry should not only unify the methodology but also supply flexible test framework to deal with kinds of design.

Yet the different designs made verifiers to build up design specific verification environment. It is good to see most of verifiers apply the UVM for a 'house', but it is still unsatisfied every verifier has to be firstly a construction worker, and then secondly to be its owner of the house. With different experience, it would take different time and efforts for the verifiers to build a house. The test framework obviously becomes a conflict factor with the project schedule. The project manager will well understand the design complexity and estimate the verification effort regarding the design complexity, but it is hard for the manager to consider the time to build a verification environment. If it could be, the manager is willing to see the environment construction time to be shortened as much as possible.

Simultaneously, the block level test case reusability and readability are much worse than the chip level. The bad reuse quality made trouble for maintenance. Looking into the various block level verification environment, it is reasonable to understand why the poor quality is born. The causes lay on:

- Different verification environment architecture
- Different UVM application habits
- Different sequences with different VIPs
- Different compilation and simulation script

It is desirable to see verifiers could build unified style of 'houses', but it could not prevent verifiers from making different appearance and interior for the houses. Then for design A's verifier and design B's verifier, it is not practical to exchange the two verifiers to maintain the other one's verification environment. With different application manners, it will introduce much effort to understand the verification environment's structure and test cases.

Even if test cases are created, it could not be equalized the verification objective could be fully covered. Without a unified test framework, the verification management missed guidelines to be embedded into the environment. Then verifiers would contribute different function coverage for design verification. Therefore, the verification completion will be questionnaire and challenged.

2. Background

Before giving the solution proposal, it is meaningful to look back the general verification MTB (Module Test Bench) establishment flow. When a new design is assigned to the verifier, he would read the design specification and master the functions, design boundary and register description. Then it is time to collect the VIP personally to prepare for the verification environment. Before the test framework could be operated, it requires the effort to learn how to integrate the VIPs and how to generate the register model. Moreover, it asks more debug time for the UVM environment top-down creation and connection. Once those VIP elements are successfully planted, the verifier would write high level sequences and coordinate the VIP agents. Besides the verification environment buildup, the later phase regression run needs customized script for server job submission and coverage collection. After the MTB is grown in a previous project, it would be maintained and adapted to a new project. For the new verifier, he should first get understood the environment and the test scenarios.

This is the general flow to build a MTB from the very beginning. Through the long and scattered process, there is quite effort taken to ensure the testbench basics:

- The verifier has to collect the available VIP himself.
- The learning curve to master the new VIPs before integration.
- There is no consistent register model creation and integration flow.
- Inestimable debug time for testbench integration.
- Non-uniform script for testbench compilation, simulation and regression.
- Varied MTB and test sequences increased the maintenance costs explicitly.

To extract the difficulties from decreasing the main efforts above, it would be summarized as below:

1. Verification environment construction time
2. Test framework and test case reuse
3. Function coverage completion

Respectively correspond the conflicts factors given in the paper abstract: verifier's experience, verification flow harmonization and verification completion standard. The UVM, as a singleton methodology, has not fully satisfy the verification teamwork. Along with the project execution, the overall verification standard and framework generalization were urgently needed to be unified. Before the verification industry push forwards the higher level test standard, the companies have waked up to the requirement, and contributed their own solutions.

This paper gives a solution 'Pangu' to ease the conflicts listed with the project, and the objective is to serve the verifier convenience for test framework fast build up, test case unification and centralized the function coverage management.

3. Unified Verification Framework Automation

To give a complete solution, Pangu as the internal testbench automation tool is developed. Before giving more details, those key words are put here for understanding the core of Pangu:

- Pangu: the testbench automation tool.
- uTB (unified testbench): the generated testbench by Pangu automation.
- uIF (unified interface): the interface which is to bridge unified command to specific bus VIP agent's sequence or item.
- uTB command: the unified command set which is used to create the test scenario.
- uNet: the unified network which transfers the uTB commands based on AHB protocol.

It would help understanding the mechanism while splitting the generated uTB framework and the automation script. Because the tool development process also follows the split strategy and is processed independently. First, it is necessary to prove the feasibility of uTB framework. Then the automation script would refer to the original model as a prototype. Therefore, this paper would first introduce the uTB architecture and then explain the Pangu script organization.

3.1 uTB Architecture

Imagine it is assigned a new module, before automating the uTB, those information should be extracted from the design specification:

- Standard bus type and number
- Clock and reset number and synchronization relationship
- Non-standard interface and number
- Register description file

Those extracted parameters would be fed to Pangu and then a specific uTB will be generated. The general framework could be drawn as the Figure 1 below:

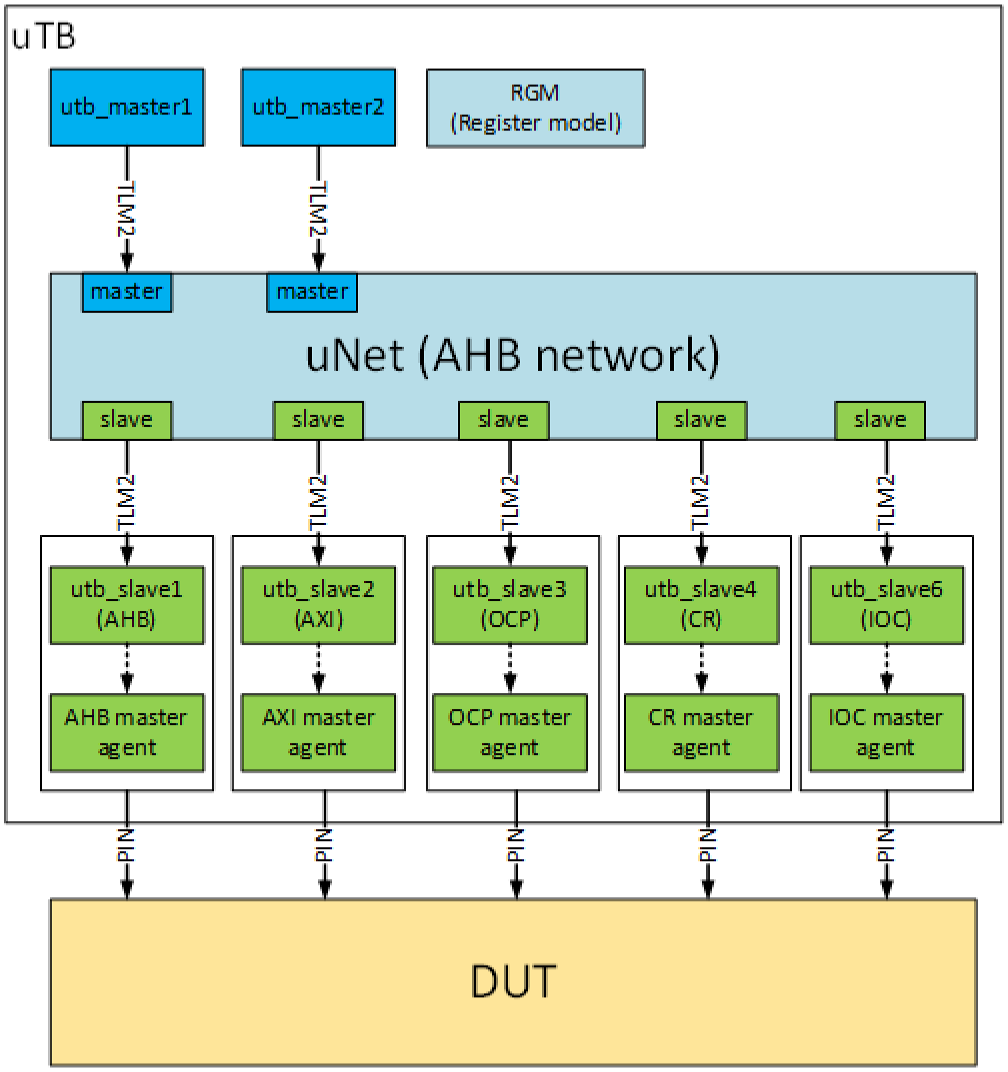


Figure 1 Specific uTB framework

Combined with previous uTB keyword definitions, it could be found several core elements composing the uTB:

- uTB master
- RGM (Register Model)
- uNet (AHB network)
- uTB slave
- Specific VIP master agent

The critical point of uTB architect is the unified command trigger and layer conversion. The uTB master could be taken as the processor or a general master which sends out the unified command. The command is composed of target slave address, command type, and the arguments. The command would be converted via the uNet and reach uTB slave. The uTB slave would further parse the command and finally translate it to the specific VIP master agent sequence or item. This is a full path from the uTB master to the VIP master agent, and similarly, the response data path would be translated along the inverted layer path. This solution realizes the feasibility of unified command set, and data consistency.

From the Figure 1, the DUT (Design under Test) is extracted those interface types:

- AHB slave interface
- AXI slave interface
- OCP slave interface
- Clock and reset
- Miscellaneous interface

To mimic the parallel slave interface (AXI and OCP), it is requested to create two uTB masters. With the register description file based on XML (Extensible Markup Language), the RGM could be created. If it is to further explore the uTB unified command conversion layers, it would be explained as the Figure 2.

When a command is given to the uTB master, it would issue a TLM2 socket item to the uNET AHB master. The uNet is configured based on the Synopsys AHB bus system, and available to receive and to response by TLM2 socket item. The transaction between uTB master and uNet AHB master is done via TLM2 instead of AHB hardware pins. Then the uNet master would translate the TLM2 socket to AHB hardware bus pins inside the visualized AHB network, and hardware bus event would be transferred to the AHB slave and then translated to TLM2 socket item. The TLM2 socket items stored in AHB slave agent would be fetched by the uTB slave. uTB slave would combine the sequential TLM2 socket items and reformatted them as a recognized uTB command object. The uTB command object which should include the command type and arguments. The arguments could consist address, data and other optional arguments. The command object will be parsed and mapped to the specific VIP master agent. Finally, the VIP master agent would stimulate the DUT via the interface pins.

To summarize the streamline command parse and composition, it could be divided into four layer conversion as Figure 2:

- Layer1: TLM2 socket item to AHB pin
- Layer2: AHB pin to unified command object
- Layer3: Unified command object to specific VIP sequence or item
- Layer4: VIP sequence or item to bus interface pin

From the development view, the layer1 and layer2 compose of the core uTB command transition and are developed centrally. The layer3 is customized according to different VIP since it is a mapping phase from the uTB command to the VIP sequence or item. The layer4 is already implemented by the VIP driver itself. From the VIP adaption to the uTB framework concern, only layer3 needs to be implemented. The layer3 is mainly about the development of uTB command parse and mapping to the VIP sequence or item.

Based on the uNet, it is available to instantiate multiple uTB masters, and supported to trigger different data access request from those masters. This way makes it applicable to trigger interleaved data transactions to the same slave agent or parallel data transactions to the different slaves.

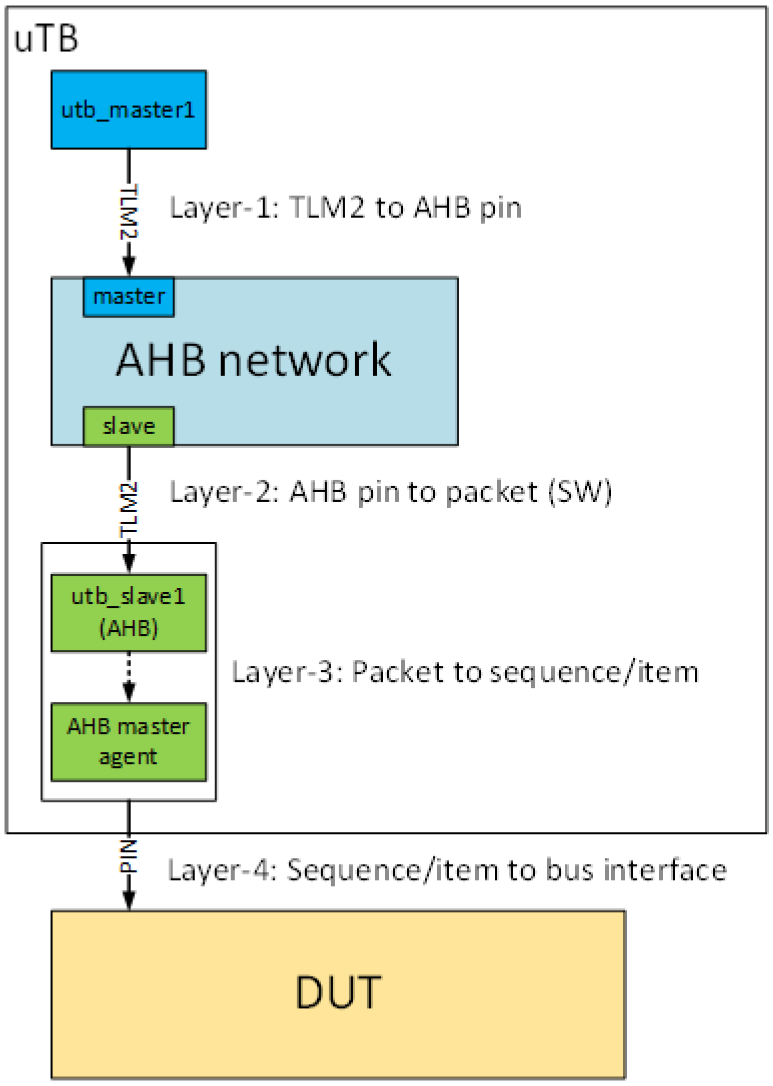


Figure 2 Unified command conversion layer

3.2 Pangu Script

As the uTB is proven its architect unity, Pangu is developed for the uTB automation. The automation flow could be described as Figure 3:

1. The uTB common package and available VIPs are the basic components.
2. The HAS (Hardware Architecture Specification) would be extracted for design parameters as customized input to Pangu.
3. With the elements above, Pangu would generate the specific uTB.

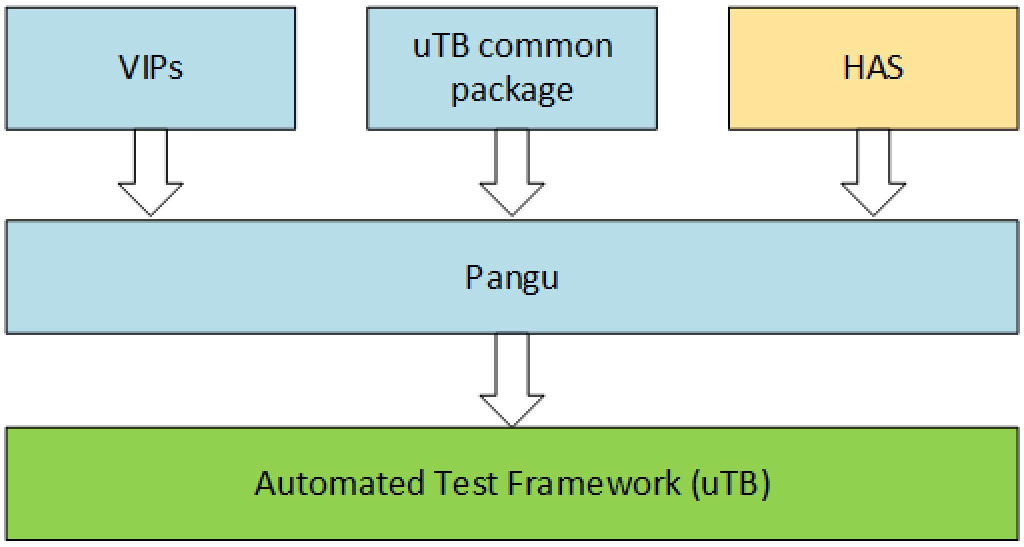


Figure 3 uTB automation flow

As an automation tool, the core development technics is based on Python and Mako [1]. To simplify the software relationship and development strategy, the Pangu software architect could be described as Figure 4. From the design input to the generated uTB, the data flow could be divided as three steps:

- Step1: Extract the design input from the GUI (Graphic User Input) or Excel form, and arrange them into the data pool class.
- Step2: The template types are divided as four kinds: configuration, environment, register and test. Each template type is together with a corresponding data class which excavates useful data from the central data pool. With the extracted data, each template could automate the related files.
- Step3: The main generator is a coordinator which combines all of template engines, and finally organize all of generated files as an integral uTB suite.

With the design parameters, Python script and Mako templates, the generated uTB suite would cover those content:

- All necessary VIP element link
- UVM register model
- uTB UVM top environment with all of element instances and the register model
- Configuration files which could be customized for each VIP type or instance
- Basic test
- Automated connected hardware testbench which instantiates the DUT and invokes the UVM test.
- Makefile script

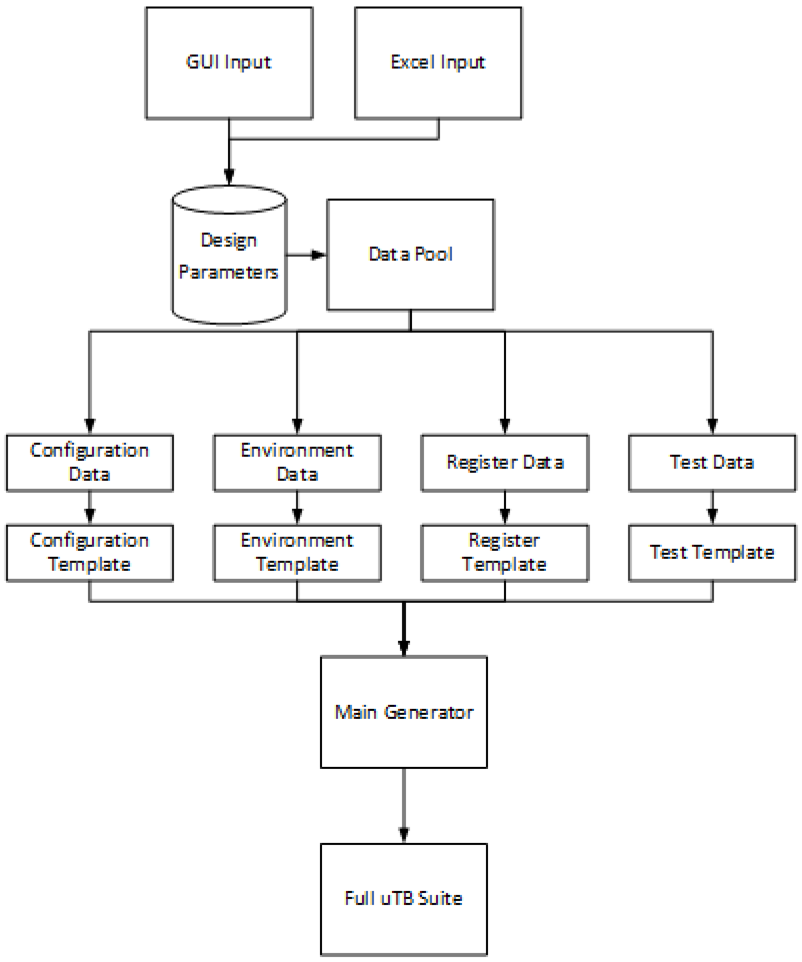


Figure 4 Pangu software architect and data flow

3.3 Conformity of Pangu and uTB

In the practical uTB architect and Pangu tool development process, there exists interdependent relationship between them. First of all, it is only possible to bear the basic uTB architect and prove its feasibility, Pangu would then have chance to trigger the script development. After Pangu reaches the initial software release standard, it is time to apply Pangu to generate more specific uTBs and get the actual feedback. The feedback would in turn help the tool to be further improved.

Therefore, it is a coupling and spiral development relationship between Pangu and uTB. Figure 5 gives an overall uTB automation layer relationship:

- In the bottom layer, the unified command set and uNet communication network compose the test standardization basics. Mako template and HAS parameters would generate the uTB environment.
- In the upper layer, commercial VIP, in-house VIP and user defined VIP all ensure the bottom drive units.
- With the generated uTB and VIPs, the centralized configuration and unified test cases contribute the maintainable tests.
- Based on those elements above, uTB could be created by Pangu.

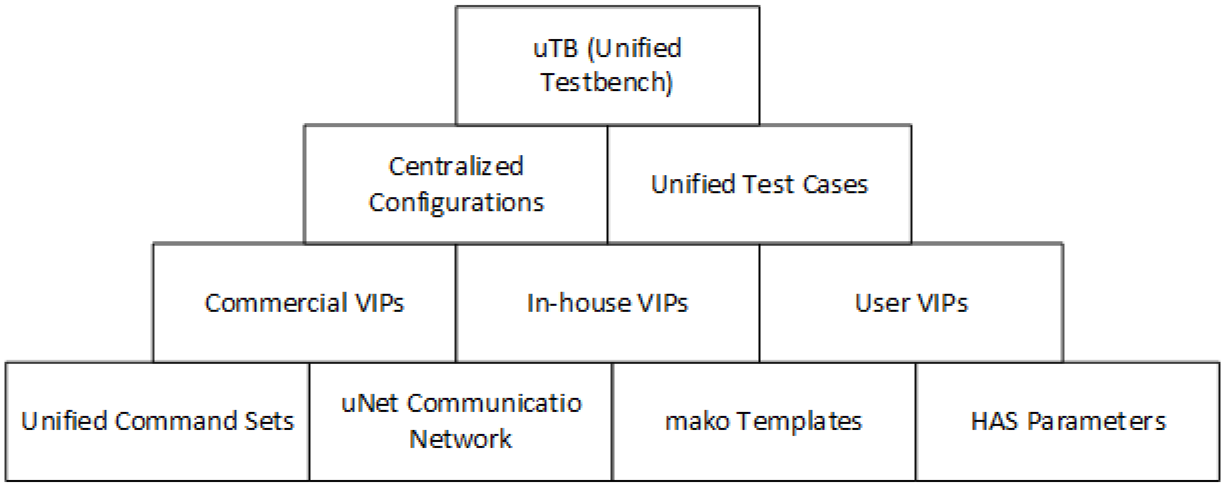


Figure 5 Coupling of Pangu and uTB

4. Test Standardization

As it is explained in the background description, uTB is not only about the parameterized architecture, but also about the test standardization. The test unity is realized based on the uNet communication network and the unified command set. The uNet communication network is an AHB bus system which transfers the composed sequential packet data number, and the unified command set makes the uTB test scenario easily understandable.

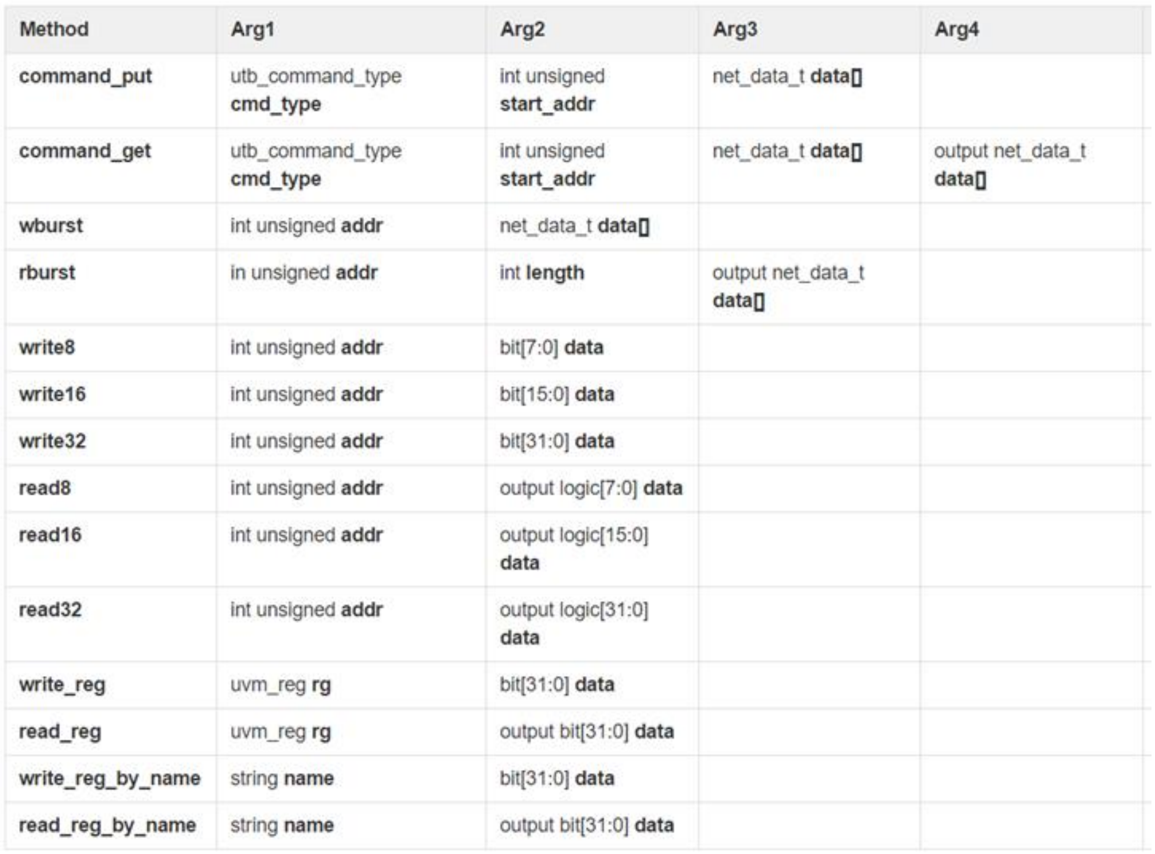
Table 1 gives the unified command set. Those command set could be divided as three types:

- Data access commands
- Register access commands
- Other commands

It is feasible for the uTB master to send those commands to different uTB slaves. Besides register and data access, it is also available to configure the clock frequency or pin value. Behind those command set, uTB master would pack different commands referring to the standard command packet format and transfer it via TLM2 socket to the AHB network master agent. Then the following layer conversion is described as Figure 2.

With the generalized commands, the uTB tests ask for less effort to create, maintain and read. This also make it possible for little UVM experienced engineer to create test scenario, which is quite like the C test. Simultaneously, it is precisely because of the command consistency, the UVM based command set is also adapted to C layer. This makes convenience to write UVM test or C test with the same command set, and the only difference is about the test language.

Table 1 Unified command set



Method	Arg1	Arg2	Arg3	Arg4
command_put	utb_command_type **cmd_type**	int unsigned **start_addr**	net_data_t **data[]**	
command_get	utb_command_type **cmd_type**	int unsigned **start_addr**	net_data_t **data[]**	output net_data_t **data[]**
wburst	int unsigned **addr**	net_data_t **data[]**		
rburst	in unsigned **addr**	int **length**	output net_data_t **data[]**	
write8	int unsigned **addr**	bit[7:0] **data**		
write16	int unsigned **addr**	bit[15:0] **data**		
write32	int unsigned **addr**	bit[31:0] **data**		
read8	int unsigned **addr**	output logic[7:0] **data**		
read16	int unsigned **addr**	output logic[15:0] **data**		
read32	int unsigned **addr**	output logic[31:0] **data**		
write_reg	uvm_reg **rg**	bit[31:0] **data**		
read_reg	uvm_reg **rg**	output bit[31:0] **data**		
write_reg_by_name	string **name**	bit[31:0] **data**		
read_reg_by_name	string **name**	output bit[31:0] **data**		

5. Centralized Function Coverage Management

Since the uTB framework is highly structured, and this provides possibility to manage the function coverage centrally. In general, the function coverage would be divided as those types below:

- Register coverage
- Bus protocol coverage
- I/O toggle coverage
- Design internal coverage

For the register coverage, bus protocol coverage and I/O toggle coverage, they could be monitored and achieved from the VIP monitor instantiated in uTB, and for the design internal coverage, and it is the verifier responsibility to refine the coverage item and to map them with the verification objective. Therefore, the centralized function coverage management diagram could be described as the Figure 6. The centralized top coverage manager would collect the global coverage dynamically with the running case.

Here the coverage collection could be done during the case run or merged with multiple regression cases offline. Therefore, the top coverage monitor would reflect two type statistics: current coverage and incremental coverage. The current coverage would specify the coverage contribution by the running case, and the incremental coverage would indicate the overall coverage by the regression run. Either of the coverage would be helpful to direct the random stimulus generation and the dynamic constraint set. Thus it is a feedback loop from the coverage monitor to the top coverage manager, and then to the uTB master random stimulus generation.

The stimulus biasing strategy would be applied in the post verification phase when the design reach to the stability region. Before triggering the stimulus biasing method, the verifier would have already ran enough random case but there is still some function coverage holes exposed. The left 20% function coverage hole is hard to be hit by the random case due to the much loose constraint of stimulus. For the left 20% coverage, some verifiers would choose to analyze the possibility of coverage hit, and create direct case manually, which is often time consuming. Moreover, some deep design internal coverage cannot be estimated the precise external stimulus, and this means the ordinary means is out of order.

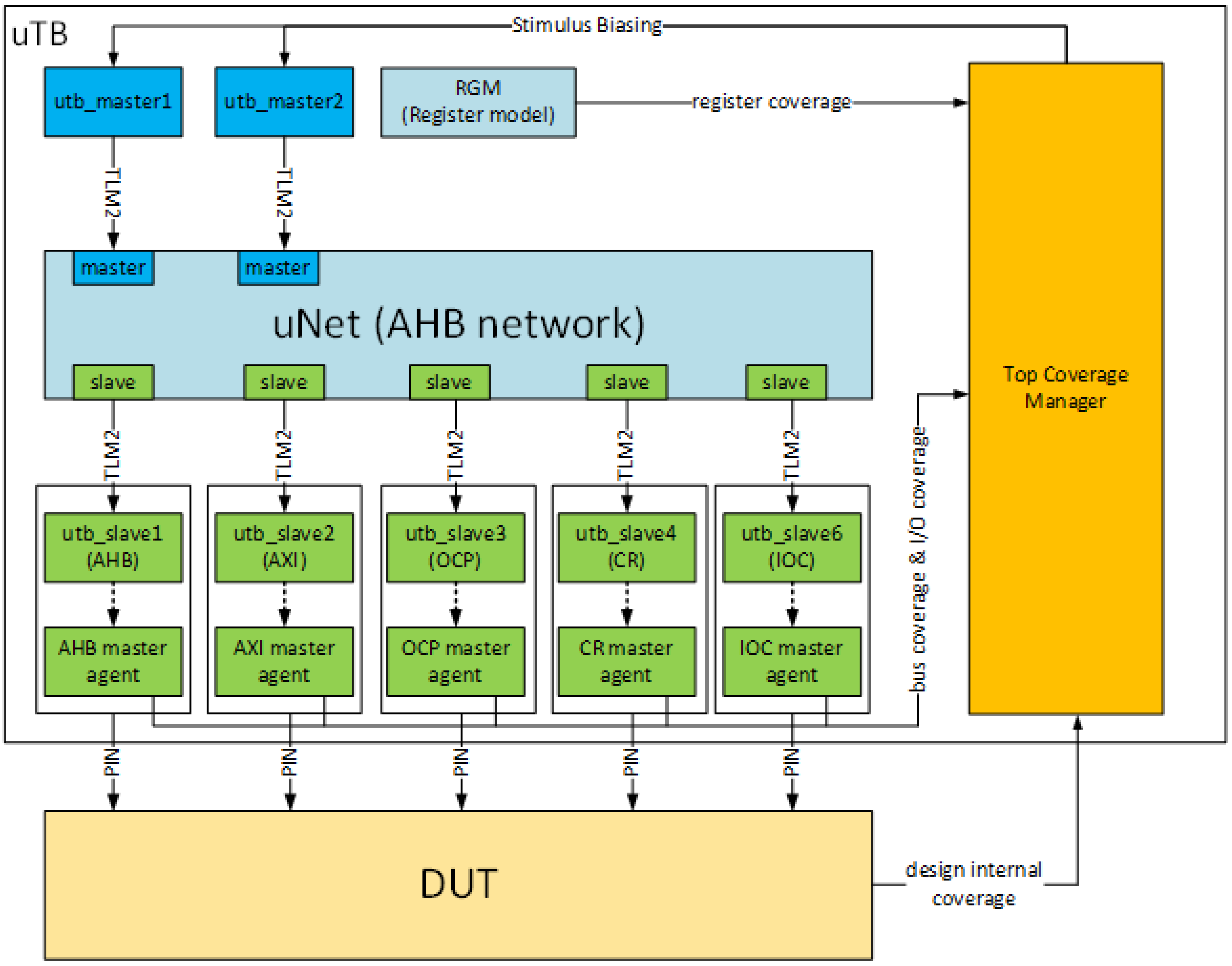


Figure 6 Centralized function coverage management

For this case, it is time to give further constraint and orient the possible stimulus for the coverage hole. The feedback loop from the top coverage manager to the uTB master is also called function coverage driven loop. The top coverage manager would give helpful combined sequence or over constraint items based on those data:

- Historical coverage and related element sequences
- All of available sequences or items

With the coverage and sequence/item database, the coverage manager would calculate the possible sequence combination and give a more clear direction. In this way, the function coverage and test sequence promote each other.

6. Case Application

Before the completion of this paper, Pangu has been applied in the project to support uTB creation for several design modules. A case application would be more conducive to how to use this tool. The Figure 7 gives an LPDDR4 controller device verification environment. In the environment, those element compose together for the uTB:

- LPDDR4 controller instantiated as DUT.
- LPDDR4 PHY and Denali memory model are connected with LPDDR4 controller as data transition path.
- One uTB master, uNet, one AXI uTB slave and one AHB uTB slave are automated to play the external master role.
- Checker and DFI4.0 monitor are manually created for data check.

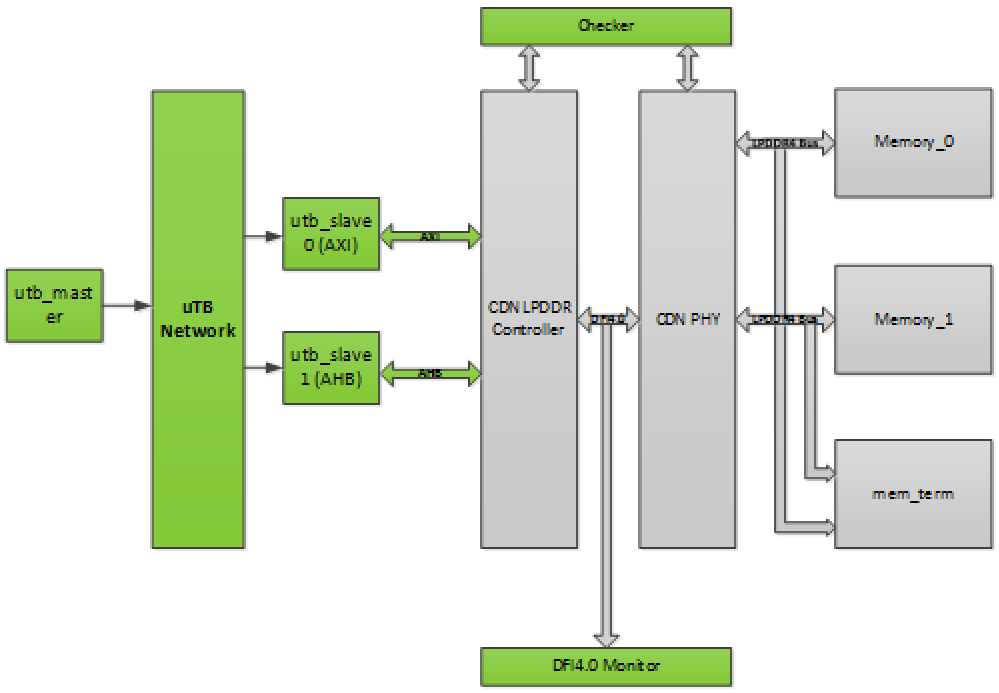


Figure 7 LPDDR4 controller uTB environment

The interesting point lays on how to create the uTB by Pangu. The uTB automation steps follow this way:

1. Read the LPDDR4 controller design specification and extract the design parameters.
2. Get the LPDDR4 controller register files.
3. uTB would be automated by the design parameters, register file and Pangu.
4. LPDDR4 PHY and Denali memory model would be connected inside uTB to complete the data path.
5. The generated Makefile could be used to compile the DUT and uTB.
6. The generated basic test case could be ran first as use case reference.

Once the uTB is created, the verifier could directly write test sequence by the unified command with little environment debug effort. An example test code is given in Table 2:

Table 2 Example code of a standard test

```
class lpddr4_basic_test extends lpddr4_base_test;
  …
  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    phase.raise_objection(this);
    // clock frequency set as 100MHz
    env.utb_master1.command_put(CLK_SET, ‘h1000_F000, {100});
    // reset assertion after 100 ns
    env.utb_master1.command_put(RESET, 'h1000_F000, {100});
    // write register by name
    env.utb_master1.write_reg_by_name("reg_dataport", 'h9ABCDEF0);
    // read register
    env.utb_master1.read_reg(env.rgm.cp_host_sdmmc.reg_dataport,
'h9ABCDEF0);
    // AHB for register address
    env.utb_master1.wburst('h2000_F000, {'h11223344, 'h55667788,
'h99AABBCC});
    env.utb_master1.rburst('h2000_F000, 4, data);
    // AXI for data address
    env.utb_master1.wburst('h4000_F000, {'h11223344, 'h55667788,
'h99AABBCC});
    env.utb_master1.rburst('h4000_F000, 4, data);
    // IOC
    env.utb_master1.command_put(IOC_SET, 'h5000_F000, {1, 1, VAL_1});
    env.utb_master1.command_put(IOC_CHECK, 'h5000_F000, {3, 2, VAL_1,
VAL_1});
    phase.drop_objection(this);
  endtask: run_phase
endclass: lpddr4_basic_test
```

In the *run_phase()* task, the first two commands are to set clock frequency and to trigger reset. This is implemented by the command *command_put()* with correct arguments. The next two commands are to write and read register with commands *write_reg_by_name()* and *read_reg()*. Then there are four commands for data write and read. It is noticeable the first pair of write and read commands are to access the slave address 0x2000F000, and the second pair of write and read commands are to access the slave address 0x4000_F000. The two slave addresses are mapped to different slaves. One slave is AHB slave, and the other one is AXI slave. For the uTB master command, it is no different for the command sent to different bus slaves, and this way gives a direct and simple way for stimulus generation. Finally, the last two commands are to configure and check I/O pins of DUT.

From the test code, it could be conceivable that this method is easy to learn and understand. Just because of the command standardization, the uTB has been also extended with adaptive C language interface. Then for verifiers, both of UVM test or C test are supported and easy to maintain.

7. Conclusions

It is easy to manage and maintain the SoC verification environment because most of SoC level verifiers are testbench users instead of architects. However, this is not true for MTB build up and most of verifiers need to build MTB themselves and write tests. Different experience and understanding of verification make the MTB difference and test manner diversity. As it is pointed out, if MTB build and debug time could be shorten, then the saved time would definitely help the design stability and project schedule. Pangu was initiated based on the MTB automation request, and the purpose is also to deliver a unified testbench.

The unified verification framework is not only the basis of testbench automation, but also the basic of test standardization. To serve the command unity, it is necessary to adapt kinds of VIP to the uTB. If the uTB slave bridge has not been developed, the verifier would implement the specific slave bridge himself. At the same time, it would be also available to connect all of sequencers inside the VIP agents and write legacy UVM virtual sequences for more flexible control. This way also complement the immobilization of uTB unified command set.

Pangu supplies an integral solution for testbench automation and test unity, and the standard test framework further makes it possible to generalize the module level verification process, and also the reuse of environment and test from module level to chip level.

8. Acknowledgment

In the development process, uTB framework validation is concurrently implemented with Pangu tool. We appreciate Haibo Shao for the uTB prototype initiative, and Yan Wu, Libo Tian, Yu Guo and Yongqi Sang for the uTB elements accomplishment. For the Pangu tool development, we would thank to Weikai Wang and Xuan Shi for the Python/Mako template execution, and Wenqiang Ren for the register model generation flow definition. Besides the Pangu release, it would be also grateful for those first batch of Pangu users, and they are Rongli Jin, Yingran Huang and Zhao Liu.

9. References

[1] Mako template language website http://www.makotemplates.org/