# A Full-scale System Monitor and Evaluation Solution for SoC Verification

Bin Liu[1] Haibo Shao[2] Xinpan Man[4]
Modem Hardware Group
Intel
Xi'an China
{bin.rocker.liu, haibo.shao, xinpan.man} @intel.com

Xiuqin Zhang[3]
ASIC Design Center
UnilC
Xi'an China
xiuqinx.zhang@intel.com

***Abstract*- Modern SoC size and complexity grow too fast, which brings a big challenge to strive for a qualified verification through the module level to the system level. However, the verification strategy between module level and system level could not take the same strategy and evaluation methods. It is due to the difference of not only the target size for simulation load but also the verification objectives. For chip level, regarding its billion gates body, we figured out new ways to decrease the simulation load and apply C code for processor programming [1]. At the same time, chip level would take more focus on the subsystem integration check, integral system health evaluation, network bandwidth calculation, balance between speed and power and so on. In this paper, a system monitor and evaluation solution based on well-structured and scalable UVM framework will illustrate the mindset how to serve system level debug and evaluation, and finally present the multi-angle measurement for a system by rich monitoring statistics.**

***Keywords- System Monitor; System Debug; UVM; DPI-C; Centralized Solution***

## I. Introduction

In modern SoC development process, function verification is no more the only target before tapeout, and other areas such as performance and power are also involved for a synthesized evaluation. The industry and verifiers keep on fighting against the critical project schedule but with more complex system and more evaluation indicators. Efficiency, abstraction and reuse such verification key factors have already got acceptance, and UVM as a unified methodology illustrated its great contribution. While the power of UVM has not been fully brought to play, especially at SoC system level verification.

The lack of system evaluation and monitor solution made it is too difficult to get an overview about system execution state. Meanwhile, regarding performance and power assessment, EDA vendors have no standard and easily attachable tools. In this paper, it would initiated a customized UVM framework Hawkeye in order to demonstrate a full-scale system monitor and evaluation solution for SoC. In the following, part II would give a background about the concept and the functions of Hawkeye. Part III would draw pictures and give details about the architecture of Hawkeye. Part IV and V would explore each feature of Hawkeye. Part VI and VII summarized the tooling development work and looked ahead the future research direction.

## II. Background

The main reason why such a specific solution is proposed is due to system clarity would be overspread by module and subsystem level verification components. In other words, even all of low level verification components would be reused and integrated in SoC level, the system vision and debug will not be implemented. Because the nature of low level verification target is based on signal check and specific feature validation, when these components are imported to SoC level, the system test would naturally inherit the low level angle of view. This judgement matched the verification daily work since quite some verifiers are still used to directly debugging signal or specific register in SoC level, instead of system vision. However, the inappropriate SoC verification habit reduced the speed of development, and pulled down verification into detailed logic, which could not supply an overview about system health and state.

Referring to software development practice, it is reasonable to gradually change mindset from hardware (HW) to software (SW), or from module level to system level. The SoC level verification is an interesting time point since it is the first time when the system is integrated. Once SoC is able to work at RTL level, FPGA and emulation platform would then prototype based on the same database, which serves SW development and complex system test. At the same time, it is also interesting the early involved software engineers develop the code with different system mindset. Then for HW and SW development, which way is more efficient in SoC level? What would be learnt from SW development? At least, HW verifiers would draw lessons from SW system vision and debug especially when stepping to system verification stage. Unfortunately, EDA simulators still have some blind area to respond the system verification requirement. This is the main reason such an in-house solution Hawkeye is developed and applied in the practical project.

So how to master the overall status of a system? Hawkeye would track those common and major components:

- Processors. There are quite different subsystems running in parallel, and there also locate distributed processors inside them.
- Registers. The processor would access registers. Register value and coverage need to be monitored at system level.
- Memories. Along with registers, memories are also the important blocks and distributed everywhere.
- Clock frequencies. Once getting known about overall clock relationship and frequencies, it is easy to judge each subsystem’s state.
- Synchronous cells (sync-cell). Sync-cell is a vital unit for cross clock domain synchronization. It is helpful to monitor each sync-cell to monitor inappropriate input signal and timing.

Those static components make up the major part of a system, and it is meaningful to monitor and analyze the data from them. Besides those fixed HW resources, those statistics would be extracted by the monitored data:

- Function coverage. It is compulsory to collection function coverage in lower level, but when shifting to SoC level, it is necessary to supply unified command to define function coverage.
- Performance. Besides the function validation, performance is also an indicator.
- VCD/FSDB. For an accurate data dump during a specific time frame, it is necessary to define the events which trigger the time to start and to finish. Waveform database would be the resource for debug and power analysis.

Based on those forms of rich data, function verification would go along with system vision. Hawkeye would accompany with the lower level reused UVM sub-environments to enhance the system debug and evaluation.

## III. Architecture

As it is explained Hawkeye is a customized solution based on UVM, it is naturally simple to be plugged into an existing verification environment. From Fig.1, it depicts the solution is composed of two major function parts. One part is the bus monitor group, and the other part is the distributed monitor utilities.

The two parts played roles differently. Bus monitor group as the name implies, it is to collection bus transactions and then transfer them to several feature blocks. Distributed monitor utilities also charge of monitor tasks but each utility is independent with each other for a specific purpose.

### *A. Bus monitor group*

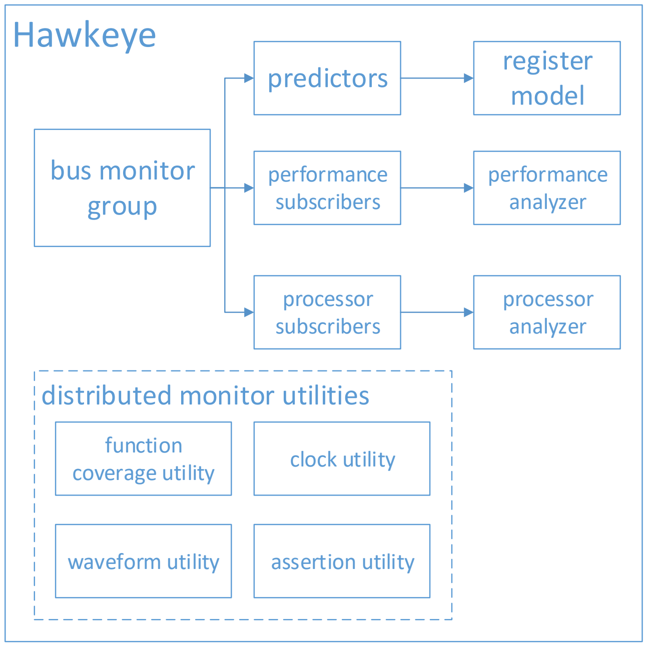
From Fig.1, it would be found bus monitor group is the kernel to collect and transfer data. Each bus monitor mapping to a specific bus located in the system, and then the monitored transactions are fed to such components:

- Register model and predictors. Bus transaction would be parsed by predictors and used to update the register model and coverage.
- Performance subscribers and analyzer. Bus transactions would be interpreted and converted to a common transaction type in performance subscribers, and put forward to the performance analyzer which would calculate and evaluate the performance.
- Processor subscribers and analyzer. Independent with the processor type, the monitored bus transactions would be forward to processor subscribers and converted to a common register operation type, and then put to the processor analyzer. The processor analyzer would broadcast the register access information in the simulation.

### *B. Distributed monitor utilities*

This part is composed of several helpful utilities, and each of them is objective to solve a system specific issue:

- Function coverage utility. In SoC level, function coverage definition needs to be generalized. One reason is that each verifier owns different coding style, and another reason is that long time to compilation will reject the SV covergroup definition directly. Instead, common user interface would be designed for SoC level function coverage definition.
- Clock utility. It is used to use simulation specific Tcl method or to apply script for post process to measure clock frequency. The clock frequency utility is to standardize the clock measurement.
- Waveform utility. Similar with clock measurement, vcs/fsdb database dump used to be triggered in Tcl session. However, for complex scenarios like events trigger, Tcl script is not so friendly to make the dump, and waveform utility will supply convenient user interface for waveform dump.
- Assertion utility. It is common to place sync-cell in SoC for both RTL easy coding and backend easy recognition and synthesis. Although CDC check would explore somewhere lacking of synchronization, but it would not figure out if the synchronized place is fed reasonable input data for successful sampling. Therefore, Hawkeye delivered several sync-cell assertions along with simulation for automatic rule check. Same to the sync-cell utility, Hawkeye also supply several memory assertions to the hundreds memories distributed in the system, and serve online rule check to ensure the system control signals timing and data access correct.



- Figure 1 Hawkeye architecture

The two parts constituted Hawkeye with the purpose of assisting system development. Once Hawkeye is applied in a system, its abundant features will present a high level vision, which matches the way how software is developed.

## IV. Bus Monitor Group

As it is pointed out the core of bus monitor groups is the bus monitor and different feature subscribers. Such framework would best utilize the bus VIP (verification IP) monitor function and further proceed to extract system information. Here it is listed the basic elements composing the bus monitor group:

- Completed SoC register model. With over hundred register blocks, it will integrate over hundred UVM register blocks and thus over thousands registers totally.
- Commonly used bus VIPs. AHB, AXI and OCP bus VIP would be taken and instantiated several times according to the monitor bus type.

- Light weight register map. While generating the full SoC register blocks, an associated array which is indexed by the register address and stores value of the register name will be created for the processor operation analysis.

Based on these elements, the following features would be realized by their specific process with the monitored bus transactions. For easy understanding, it is given the Fig.2 as an example SoC which is composed of common units such as processors, registers and memories. At the same time, there are kinds of bus types attached between the components and the NoC. Then it is to introduce the three features of bus monitor group based on the example SoC, and help understanding how the solution does service in SoC level.

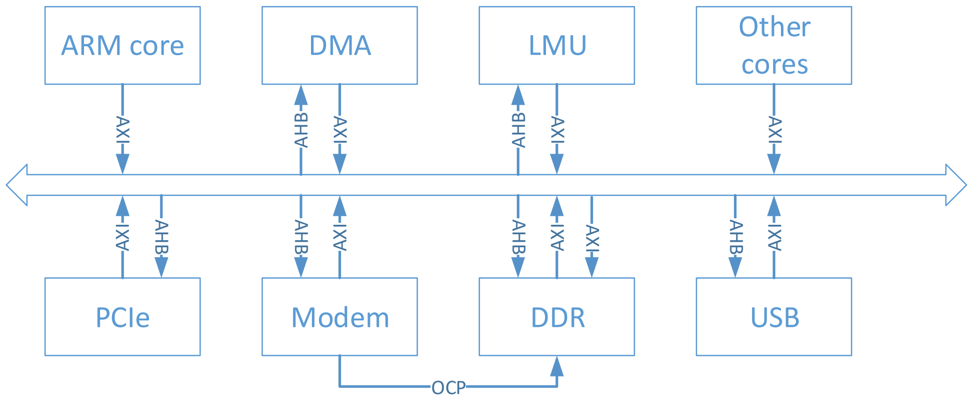


Figure 2 A SoC instituted by NoC and components

*A. Register coverage collection feature*

The basic idea for SoC register coverage is the quite similar with the method taken in module level, but there are some points to be noted here:

- Because of consistent register description XML format files, an automatic flow would be applied with those files to do design automation, generate C header files for SW coding, and UVM register block files for verification. However, when shifting to SoC level, the thousands of UVM register instantiation would be quite time consuming. Therefore, it is necessary to add instantiation switch for each register block, and instantiate only some blocks according to the need of each test case.
- Once Hawkeye binds bus interfaces around each register slave ports, those interfaces would be assigned to Hawkeye environment and each bus monitor would be ready to capture register access transaction. Thus referring to UVM register model application, Hawkeye would instantiate the bus adaptors and predictors along with the singleton SoC UVM RGM (register model). Here the creation and connection for those necessary elements is coded for automatic.
- Specific to each test, it would optionally tell Hawkeye to instantiate which register block, and then transaction data would be used to update the register block and register coverage.
- Regarding for easy maintenance, the backdoor access is not introduced.
- Although it would create a top register map to link up all of sub register blocks' maps, but it is not so practical to use the top register map for coverage. Because the bus monitor is bind at each register block slave port, and then the address is often supplied with shift address. Therefore, for each register block, it is needed to put their map to the corresponding register predictor.

Each test would generate a coverage database with the case specific defined function coverage data, and then once a regression completed, the coverage data would be merged together. It is meaningful to independently analyze the register coverage in SoC level and module level because some register is related with SoC integration for cross module communication. Finally, the module and SoC coverage database would be also merged to evaluate the overall register coverage. Thus, the centralized SoC RGM is the only place to involve all of UVM registers.

Since the RGM structure and data is close related with the project, this paper has to describe the structure and data collection instead of RGM snapshot. As it is explained the singleton RGM is composed of child register blocks, each register block creates the register blocks and coverage. During test run phase, if DMA in Fig.3 receives a register write or read operation, the address and data would be collected by the bus monitor and transmit to the predictor. Finally, the specific address mapped register's coverage would be updated. Along with the regression running thousand SoC tests, all of register blocks' coverage would be collected and merged to centrally reflect the system level register coverage.

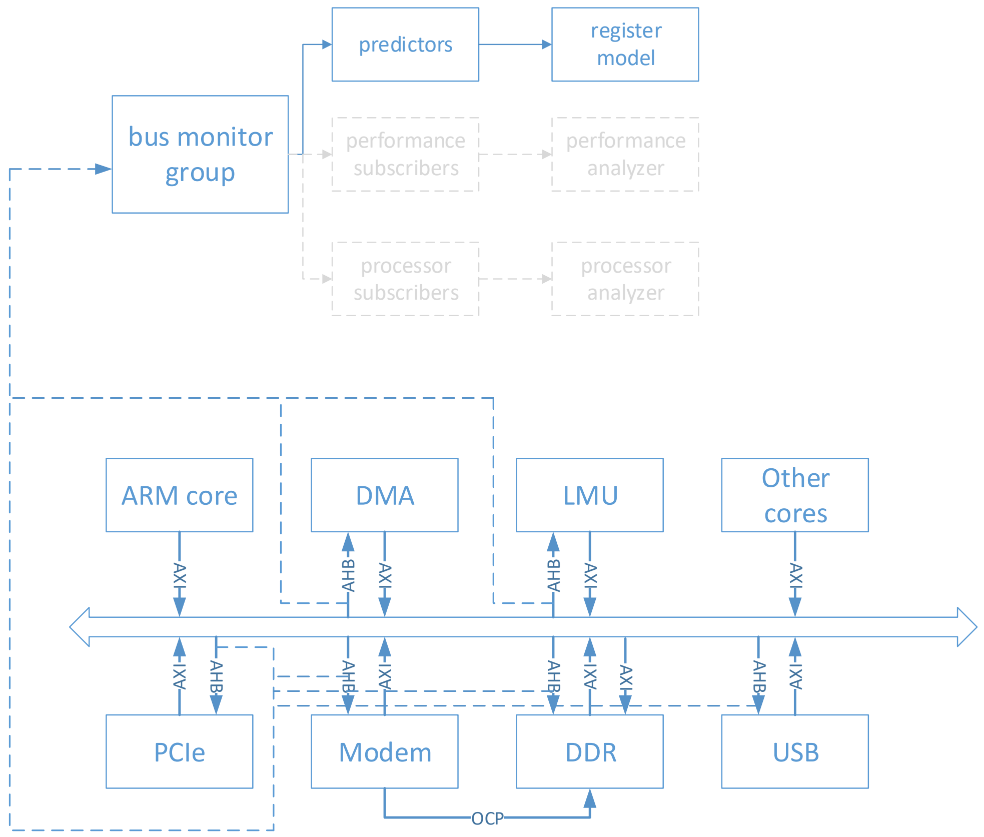


Figure 3 Register coverage collection at SoC level

*B. Processor operation analysis feature*

For any type of processor, it owns high speed bus interface attached to NoC for register and memory access. With the bus monitored transactions, and other necessary signals, it would present a system vision for the scattered processors. Those important signals include:

- Register and memory access bus
- Clock, power and reset
- Interrupt vector

With those necessary input, it would achieve the work frequency, power state, and execution path and interrupt handling. Although all of business processor is delivered with execution parser e.g. tarmac trace for ARM core, it is not so intuitive for the verifiers to locate how far the processor is executing the code. For instance, for a processor is pending while reading a register until the read value is matched, the verifier has to manually watch the bus, map the access address and the code or read the tarmac trace file. Neither of such methods is friendly to help quickly tracing processor execution.

As Fig.4 shows, the processor operation analysis feature is a light weight solution and easy to extensively cover any type of processor. Hawkeye has been applied in a silicon proven SoC project, and it has been utilized to monitor several subsystems in parallel and help demystifying the processors' parallel coordination and synchronization. From the simulation log window, it would generate specific message ID with each processor, and offer friendly readable message format. At the same time, other system signals such as clock and interrupt vector would give more information:

- What speed is the processor running

- If the processor is power off or active running
- If the processor entered interrupt or exited interrupt

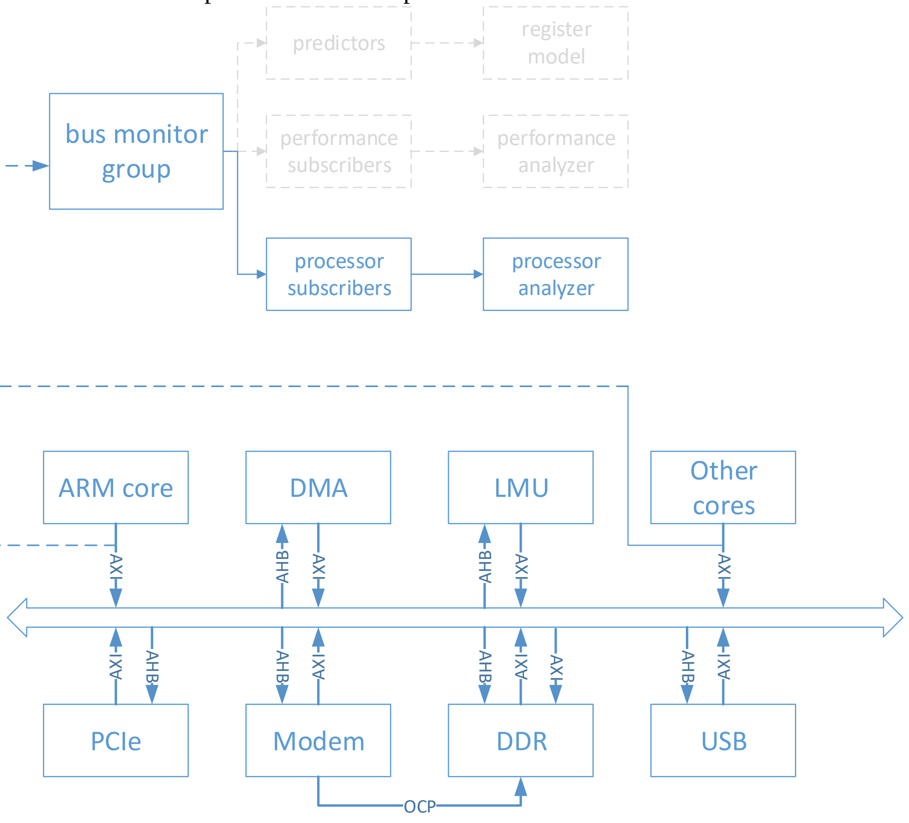


Figure 4 Processor operation analysis at SoC level

Fig.5 gives the processor operation analysis in simulation log window for register access:

```
 @ 164422 ns: uvm_test_top.env.hky_top.busop_subscribers[axi_ap_bus0]
[busop_subscribers[axi0]] WRITE REG/MEM regXXX, ADDR = 0x5d800400 with data
length = 1
 @ 164422 ns: uvm_test_top.env.hky_top.busop_subscribers[axi_ap_bus0]
[busop_subscribers[axi0]] data[0]=0x68070002
 @ 165277 ns: uvm_test_top.env.hky_top.busop_subscribers[axi_ap_bus0]
[busop_subscribers[axi0]] WRITE REG/MEM LMU RAM, ADDR = 0x20008000 with data
length = 4
 @ 165277 ns: uvm_test_top.env.hky_top.busop_subscribers[axi_ap_bus0]
[busop_subscribers[axi0]] data[0]=0x1 data[1]=0x2 data[2]=0x3 data[3]=0x4
 @ 168322 ns: uvm_test_top.env.hky_top.busop_subscribers[axi_ap_bus0]
[busop_subscribers[axi0]] READ REG/MEM regXXX, ADDR = 0x5d800420 with data
length = 1
 @ 168322 ns: uvm_test_top.env.hky_top.busop_subscribers[axi_ap_bus0]
[busop_subscribers[axi0]] data[0]=0x20780401
```
Figure 5 Processor operation analysis log

### *C. Performance analysis feature*

Performance got more and more attention in SoC development, and it is helpful to measure the performance before silicon. As early as possible, RTL simulation is an important performance indicator from system architect to design implementation. If time allowed, it offers the margin to adjust design parameters and finally makes the system overall performance to satisfy the product specification required index. In general, the performance assessment target would be a point to point data path or a multiple points to point path. As Fig.6 shows one case which is to test the performance limits for DDR when multiple devices transmit huge amount data to the DDR destination. Such a case would also point out if the bandwidth distribution to the three data sources is reasonable by the NoC. Moreover, it would help to discover the bottleneck point if the DDR could not be exercised to the throughput limit.

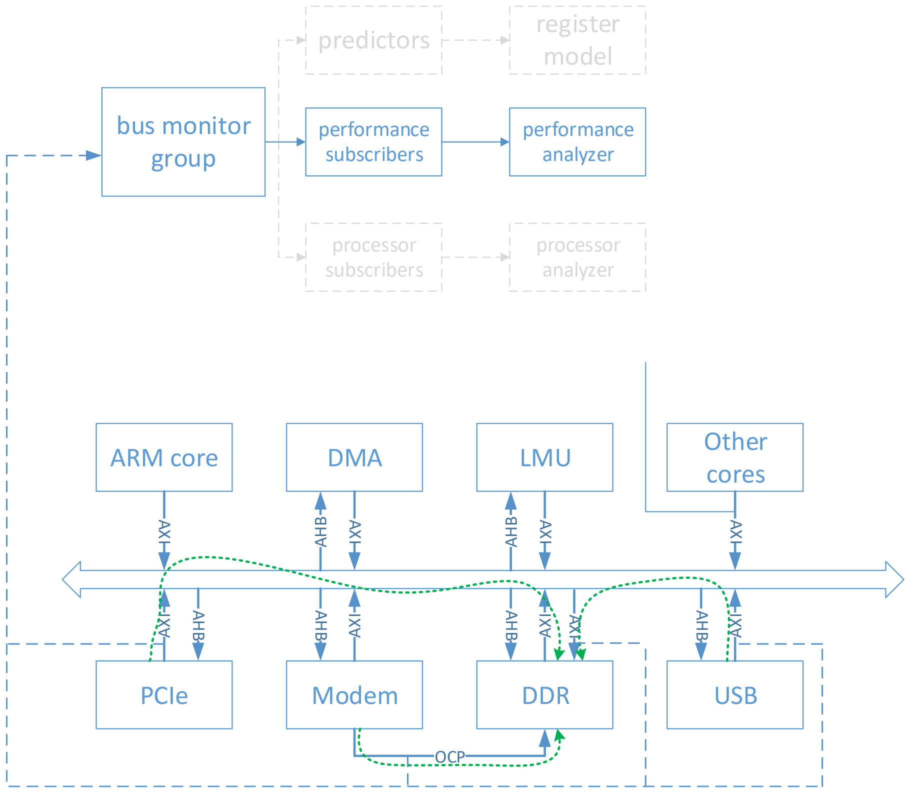


Figure 6 Performance analysis at SoC level

For the performance analysis, there are several indicators which would be calculated based on the monitored bus transactions:

- Throughput. This indicator is to measure the data processing ability during a time frame. For a calculation, the start time, the end time, and transaction information (burst number, length and size) should be given.
- Latency. This indicator is to measure the response speed for each transferred data. For a calculation, each transferred data valid assertion time and ready assertion time should be given.
- Outstanding. This indicator is to measure the multiple burst ID process ability when the outstanding mode is enabled. For a calculation, during each burst valid transition time frame, it is needed to account how many other ID burst asserted.

For those three indicators, Hawkeye would take online calculation and periodically redraw picture. Verifier would also get the throughput by offering a user defined start time and end time, or events defined in C code. Hawkeye has already exported the throughput method with DPI-C interface.

Here Fig.7 & 8 give the drawn performance pictures by Hawkeye to measure a specific bus interface by throughput and latency indicators:

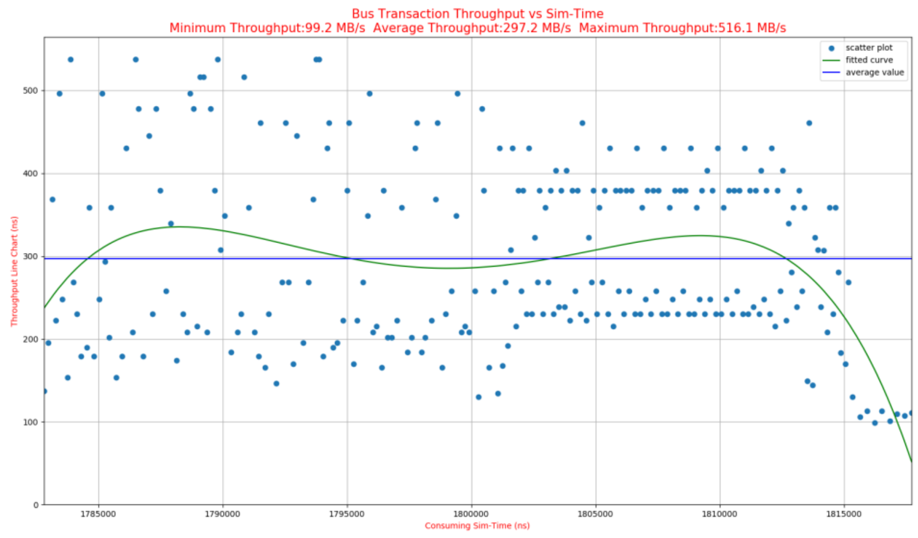


Figure 7 Throughput indicator of specific bus transaction

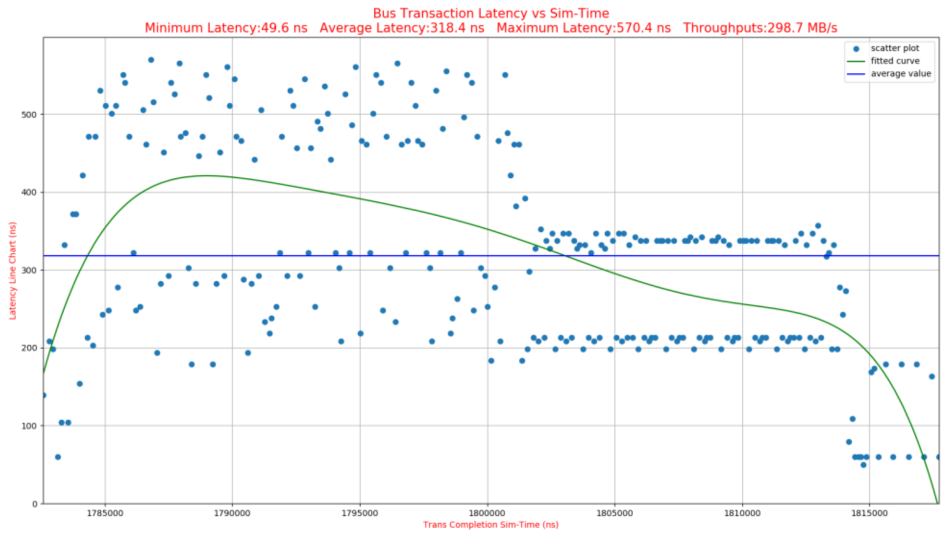


Figure 8 Latency indicator of specific bus transaction

### *D. How-to for adaption*

From the Fig.9, it is found bus monitor group part is available for quick integration with existing environment. Regarding the bus monitors and corresponding different feature subscribers would be automatically created and connected, the user need not to worry about those basic setup. According to project experience and the Fig.9, the adaption steps would be as listed:

- Extend a centralized and user specific Hawkeye configuration class hky_config.
- Hawkeye environment hky_env would be embedded into existing top environment top_env, and the singleton SoC RGM hky_rgm would be created at basic test level.
- By factory configuration uvm_config_db, the specific hky_config and general hky_rgm would be configured to the hky_env and its sub components.
- Moreover, as Hawkeye is plugged into an extended environment, then verifier would choose to enable or to disable Hawkeye features when running simulation. Since Hawkeye is taking a system monitor role, test case result will not be affected if enabling it or not.

- For fine control, test would also give specific runtime option or configure via DPI-C interface to dynamically modify the Hawkeye RGM content. DPI-C APIs are also extensively applied in the other function part of Hawkeye, distributed monitor utilities.

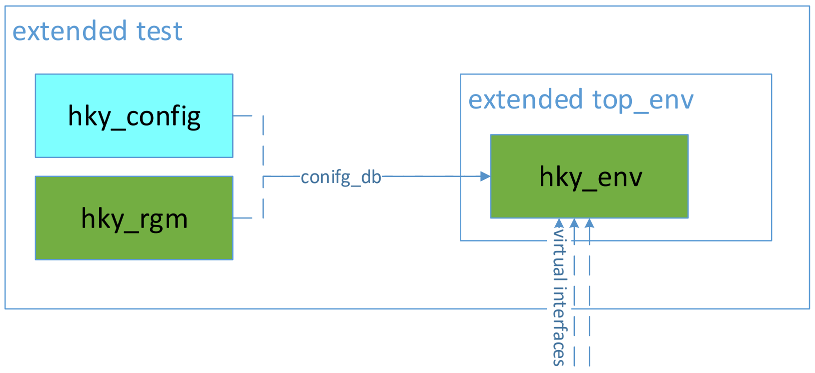


Figure 9 Adaption for Hawkeye bus monitor group

Since the bus monitor groups would attach different kinds of bus monitor to the target blocks, Hawkeye is designed to serve as more automatic as possible. Besides the steps given above to have a light weight UVM environment plugin, the user would just bind the bus interface to the target blocks, and then configure the interface to Hawkeye. It is no more needed to do interface connection or mapping inside Hawkeye and the automation makes the adaption easier. The adaption solution asks the user only to focus on the interface binding and configuration, and it no needs to implement the bus monitor and predictor for register coverage, processor analysis, or performance evaluation. The Hawkeye automatics encapsulated all of bus monitor features, and the solution was initiated for an easily integrated IP.

Fig.10 gives the example code guiding for the Hawkeye adaption:

```
// block bus and interface binding (AHB, AXI or OCP)
bind reg_blk hky_ahb_slave_bind reg_blk_hky_bind (
     .rstn       (hreset_n_i)
     ,.clk       (hclk_i)
     ,.hsel      (hsel_i)
     ,.hprot     (hprot_i)
     ,.hburst    (hburst_i)
     ,.hsize     (hsize_i)
     ,.htrans    (htrans_i)
     ,.haddr     (haddr_i)
     ,.hwrite    (hwrite_i)
     ,.hwdata    (hwdata_i)
     ,.hready    (hready)
     ,.hreadyout (hready_out)
     ,.hrdata    (hrdata_o)
     ,.hresp     (hresp_o)
  );
...
```

```
  uvm_config_db#(virtual svt_ahb_slave_if)::set(uvm_root::get(),
"*hky_top.ahb_slaves[reg_blk]", "vif",`REG_BLK_PATH.reg_blk_hky_bind.intf);
  // top parameters for Hawkeye internal automation
  parameter string hky_ahb_slaves_p[] ={"reg_blk", ...};
  parameter string hky_axi_masters_p[] = {"cpu_bus", "ddr_bus", ...};
  hky_bus_t hky_busop_subscribers_p[string] = '{"cpu_bus":HKY_AXI, ...};
  hky_bus_t hky_perform_subscribers_p[string] = '{"ddr_bus":HKY_AXI, ...};
  ...
  // embedded into existing environment
  class refbox_hky_env extends refbox_env;
    hky_env hky_top; // Hawkeye top environment
    `uvm_component_utils(refbox_hky_env)
    ...
    function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      hky_top = hky_env::type_id::create("hky_top", this);
    endfunction: build_phase
  endclass

  // singleton register block and factory overriding to enable Hakweye
  class refbox_hawkeye_plugin_test extends refbox_base_test;
    hky_user_config hky_cfg; // Hawkeye top user configuration
    hky_reg_model_block hky_rgm; // Hawkeye top singleton RGM
    `uvm_component_utils(refbox_hawkeye_plugin_test)
    ...
    function void build_phase(uvm_phase phase);
      set_type_override_by_type(refbox_env::get_type(),
refbox_hky_env::get_type()); // type override
      set_type_override_by_type(hky_config::get_type(),
hky_user_config::get_type()); // type override
      hky_cfg = hky_user_config::type_id::create("hky_cfg");   // config
object creation
      uvm_config_db#(hky_user_config)::set(this, "*hky_top", "cfg", hky_cfg);
// configuration
      uvm_reg::include_coverage("*", UVM_CVR_FIELD_VALS + UVM_CVR_ADDR_MAP);
      hky_rgm = hky_reg_model_block::type_id::create("hky_rgm"); // RGM
creation
      hky_rgm.build();
      hky_rgm.lock_model();
      hky_rgm.enable_coverage();
      uvm_config_db#(hky_reg_model_block)::set(this, "*hky_top", "rgm",
hky_rgm);  // RGM configuration
      super.build_phase(phase);
    endfunction: build_phase
    ...
  endclass
```

Figure 10 Hawkeye adaption to existing environment

## V. Distributed Monitor Utilities

As the second function part of Hawkeye, the features of distributed monitor utilities serve different purposes. The initiative of such distributed features is also to illuminate the whole system from the state of darkness. The monitored objects involve the function coverage, clock, sync-cell and memory. The elements of distributed monitor utilities are arranged without central blocks, and each utility would be enabled or disabled which is independent with other utilities. As Fig.11 described, the elements include:

- Kinds of corresponding monitor functions.
- Consistent design and verification database.
- Appropriate DPI-C APIs mapping with the utilities.
- Extracted information from the consistent database.

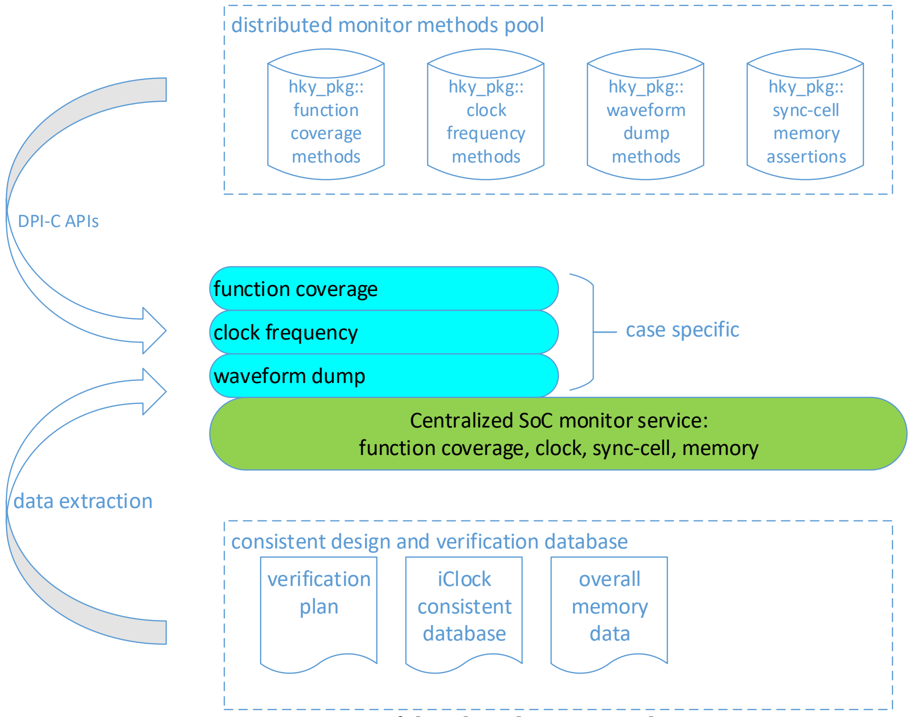

distributed monitor methods pool
hky_pkg:: function coverage methods
hky_pkg:: clock frequency methods
hky_pkg:: waveform dump methods
hky_pkg:: sync-cell memory assertions
DPI-C APIs
function coverage
clock frequency
waveform dump
case specific
Centralized SoC monitor service: function coverage, clock, sync-cell, memory
data extraction
consistent design and verification database
verification plan
iClock consistent database
overall memory data

Figure 11 Compositions of distributed monitor utilities

With the scattered Hawkeye supplied methods, verifiers would call the exported DPI-C APIs in test code to define function coverage, check clock frequency and dump waveform. Meanwhile, with the extracted information from the consistent design and verification database, it would centrally define global function coverage according to the verification plan, key clocks' monitors based on the clock design information, and memories rule check from overall memory data. Therefore, both of Hawkeye supplied methods and the consistent database are the solid basis to enable the distributed monitor utilities.

Fig.12 shows a use case that distributed monitor utilities are made use in the example SoC system. It lists four color circles to present each feature utility, and then it combined and located those utilities into different subsystem. For instances, nearly all of subsystems include the memory utility, sync-cell utility and clock frequency utility. Besides, function coverage utility would be also set globally to collect event and data for specific subsystem. As regards those function utilities are globally called or case specific called, it depends on if the objective point is a global care to take of or a small point to cover.

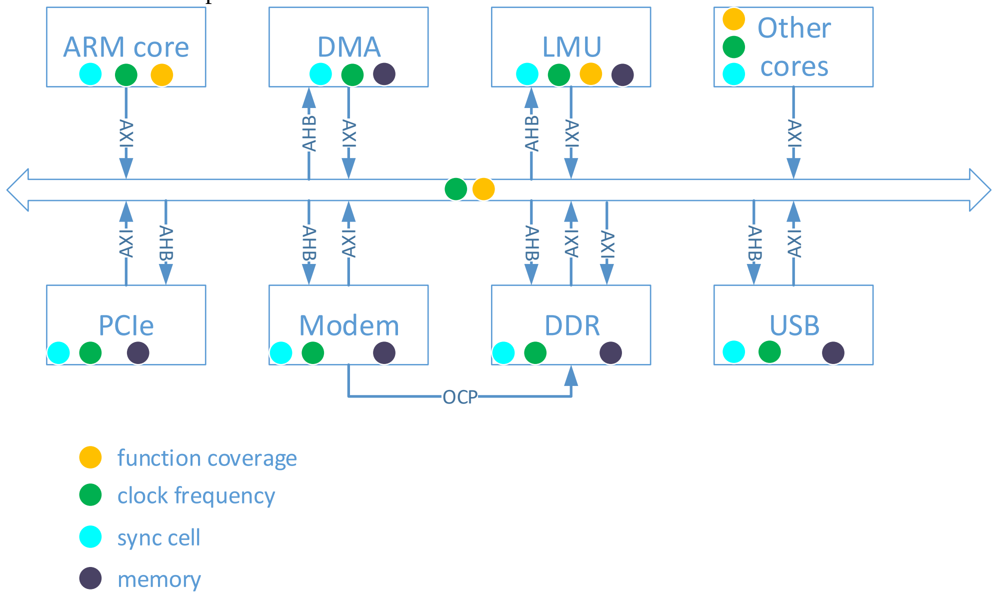


Figure 12 Application case of distributed monitor utilities

### A. *Function coverage utility*

Although the Fig.11 put the verification plan into the consistent database, some verification objectives should be covered in system level, and some should be covered in module level. It would be ideal if all of verification objectives could be defined with SV covergroup in test case, but the simulator such as Synopsys VCS requires long compilation time for covergroup definition and modification. Hawkeye have implemented basic coverage point definition methods by SV and also exported them as DPI-C APIs. C testers would employ those formal APIs to define coverage, and Hawkeye would dynamically create covergroups, which will generate the same coverage data format with module level function coverage. Therefore, with the unified coverage APIs, it not only supplies the uniform coverage interface for better readability and reuse, and also closes the coverage database gap between module level and system level.

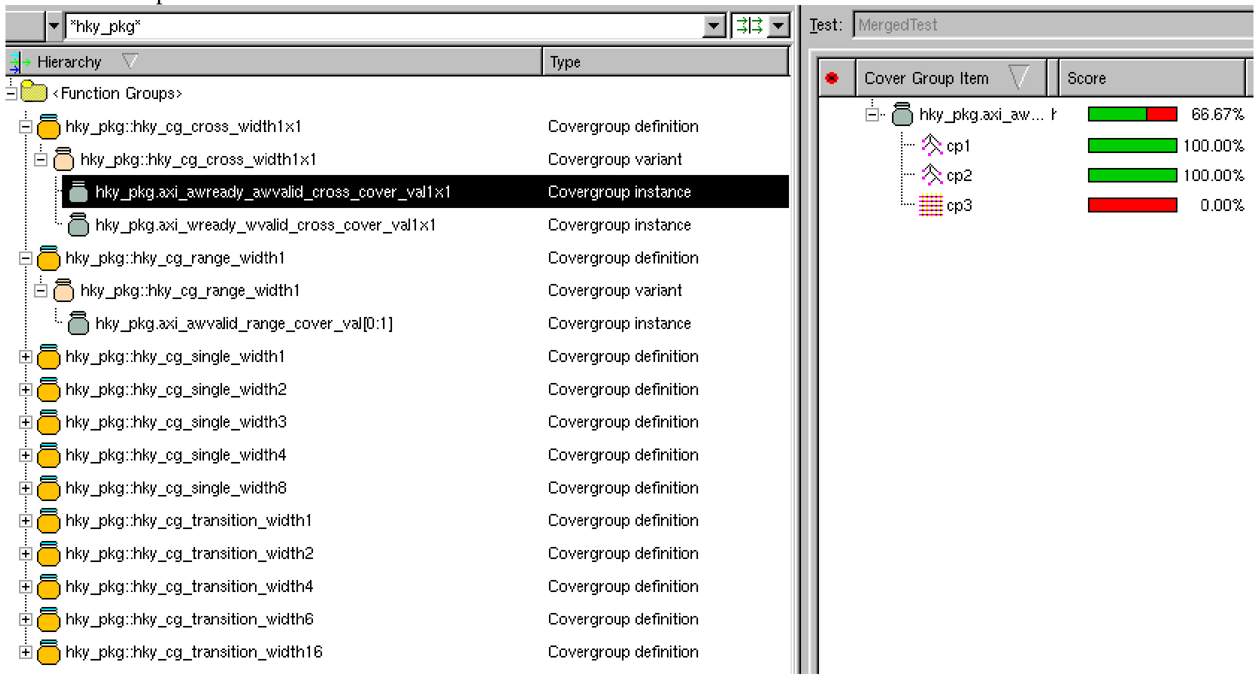
Fig.13 gives the defined coverage DPI-C APIs:

```
  void add_single_coverage(char* sample_e, int sample_width, char* obj, int
obj_width, int binval, char* name, int edge_mode, int condition_val);
  void add_range_coverage(char* sample_e, int sample_width, char* obj, int
obj_width, int binval1, int binval2, char* name, int edge_mode, int
condition_val);
  void add_transition_coverage(char* sample_e, int sample_width, char* obj,
int obj_width, int binval1, int binval2, char* name, int edge_mode, int
condition_val);
  void add_cross_coverage(char* sample_e, int sample_width, char* obj1, int
obj1_width, int binval1, char* obj2, int obj2_width, int binval2, char* name,
int edge_mode, int condition_val);
```

Figure 13 Function coverage definition APIs

APIs' names imply they would be taken to define single data coverage point, data range coverage point, data transition coverage point or cross data coverage point. In general, the exported APIs would satisfy the most need in system level. During the functions implementation procedure, it was blocked by the insufficiency of SV language that it could not support the dynamic sized vector selection, and therefore, we have to copy quite a lot code for the methods to satisfy the possible different signal vector size. At last, system function coverage would be merged with module level coverage as a whole for function coverage measurement, so it requires verifiers to put unique coverage names while calling the APIs. For instance, it is suggested to put the verification objective name + sampled signal name + sampled value to make the covergroup name the one and only.

Fig.14 gives a snapshot that system verifiers applied the function coverage APIs and generated the coverage database in VCS. Later on the data would be fed to function coverage management tooling for the evaluation of verification completion.


*hky_pkg*

Test: MergedTest

| Hierarchy | Type |
|---|---|
| <Function Groups> | |
| hky_pkg::hky_cg_cross_width1x1 | Covergroup definition |
| hky_pkg::hky_cg_cross_width1x1 | Covergroup variant |
| hky_pkg.axi_awready_awvalid_cross_cover_val1x1 | Covergroup instance |
| hky_pkg.axi_wready_wvalid_cross_cover_val1x1 | Covergroup instance |
| hky_pkg::hky_cg_range_width1 | Covergroup definition |
| hky_pkg::hky_cg_range_width1 | Covergroup variant |
| hky_pkg.axi_awvalid_range_cover_val[0:1] | Covergroup instance |
| hky_pkg::hky_cg_single_width1 | Covergroup definition |
| hky_pkg::hky_cg_single_width2 | Covergroup definition |
| hky_pkg::hky_cg_single_width3 | Covergroup definition |
| hky_pkg::hky_cg_single_width4 | Covergroup definition |
| hky_pkg::hky_cg_single_width8 | Covergroup definition |
| hky_pkg::hky_cg_transition_width1 | Covergroup definition |
| hky_pkg::hky_cg_transition_width2 | Covergroup definition |
| hky_pkg::hky_cg_transition_width4 | Covergroup definition |
| hky_pkg::hky_cg_transition_width6 | Covergroup definition |
| hky_pkg::hky_cg_transition_width16 | Covergroup definition |

| Cover Group Item | Score |
|---|---|
| hky_pkg.axi_aw... | 66.67% |
| cp1 | 100.00% |
| cp2 | 100.00% |
| cp3 | 0.00% |

Figure 14 Function coverage database with Hawkeye defined coverage points

## *B. Clock utility*

It is commonly required to measure clock frequency in test case, but it used to be implemented by Tcl script or post proceeded. The clock utility supplies convenient methods to measure clock period and frequency, besides verifiers would also call clock frequency comparison function. Clock utility follows the same manner with function coverage utility that it exported the DPI-C APIs based on the backstage SV methods.

With the exported APIs, it also requires consistent clock information. In this case, the clock information is extracted by the in-house tooling named iClock which includes but not limited with those data such as clock name, path and desired frequency. By APIs and the database, verifiers would keep monitoring the key clocks in system level, and test case would also check specific clocks at any time point.

Fig.15 gives the exported clock utility related APIs:

```
void bget_clock_frequency(char* path, int *freq);
void bget_clock_peroid(char* path, int *prod);
void bcompare_clock_frequency(char* path, int freq, int *ok);
void bcompare_clock_peroid(char* path, int prod, int *ok);
```

Figure 15 Clock frequency and comparison APIs

With these APIs, it is easy to monitor any key clock, and then check if the clock control register configuration is correct. Moreover, holding a full picture of the major clocks' frequency, it is available to evaluate the system status. It would use the clock frequency to check if the system configuration take effects. For instance, if the system went into power down state, the distributed system clocks frequency would be also reduced to zero or a quite slow rate.

### *C. Waveform utility*

The project experience also requires a unified interface to dump waveform. In general, Tcl script would make it but it is difficult to make dump by some specific event. For example, if the event is composed of several signals' specific state, it would be a trouble and long coding for Tcl. Hawkeye also builds the waveform dump methods, and exported them to the system level. Similar with other exported utility APIs, once the verifier is able to define event and then it is easy to dump the waveform during any time frame.

Fig.16 gives the exported waveform related APIs:

```
void create_event(char *name, char *path, int width, int val);
void dump_waveform(char *format, char *event1, char *event2, char *filename)
```

Figure 16 Waveform dump APIs

Verifiers would define an event by signals and conditional value by create_event API, and the API would be called several times for a same named event and define an event which is sensing multiple signals. After the start sampling event and finish sampling event are created, the verifier would call API dump_waveform to dump VCD or FSDB format in a named file. Here the argument event1 and event2 would be given with the previously defined event name or a specific time point such as "100us".

### *D. Assertion utility*

There are over thousands sync-cells and hundreds memories allocated everywhere in the system, but each of them would be checked by different ways in their own module and subsystem level verification. It is also possible quite some of the sync-cells and memories have not been given enough focus and check in the lower level verification. So while stepping to system level verification, it becomes a question if it is required to cover all of the sync-cells and memories. Here Hawkeye has developed several assertions especially for the sync-cells and memories. Commonly, the sync-cell and memory have been already internally located some rule check assertions, but some function test failures pointed out it is not enough yet with those pre-defined internal assertions, and then Hawkeye created some assertions objective to cover the integration control and timing rule check for the sync-cell and memories.

For sync-cell rule check, here it listed two basic requirements and are implemented by SV properties binding with all of sync-cells in the system:

- No glitch is allowed in the data input.
- Data input should be stable for at least 2 cycles.

From the simple rule, it would be considered even CDC check would not discover those dynamic signal drawbacks. However, if the properties are applied in system simulation, it would be easy to cover the rule violation scenarios. It is not suggested to apply formal verification for the thousands cells in system level for the speed concern. While regarding the memory control rule is technology specific, it has to ignore the detailed rules description. The assertion utility would be suggested because the high cost-effective value with several properties binding to the huge amount of target blocks.

### *E. How-to for adaption*

Regarding the adaption, the flow is a little different with the other function part bus monitor group since distributed monitor utilities owns quite independent members. It also has the same benefit as bus monitor group because the maintenance effort is mainly taken centrally. Therefore, once the Hawkeye package is imported in the top level environment and its APIs are exported, then system testbench maintainer would pre-define the global function coverage points, monitored clocks and bind assertions to the common modules. For specific test case, the verifier would then define different function coverage related to test validation points.

Here it is especially given how to use the clock utility for full chip clocks frequency monitor and check. From Fig.11, there is also a solid clock database which is proceeded by *iClock* in-house tooling, and involves all of clock information for the designer, the verifier and the backend. For instance, the verifier would easily extract all of clock names, design paths and target frequencies by *iClock* as shown in Fig.17. Then with the golden references, verifier would take clock utility to monitor important clocks through the test, and report clock frequency mismatch once clock register was not configured properly. A practical way is to keep clock monitor after reset and clock initialization, and disable the monitor if the chip is powered down.

```
#define CLK_CPU_PATH "soc_tb.dut.cpu_subsys.core.clk_cpu"
#define CLK_CPU_FREQ  1200 // MHz
...
```

Figure 17 Extracted Clock Macros

Because of the light weight defined methods and exported APIs, the second function part is also easy to be independently exported to subsystem verification environment and enable any function utility. Meanwhile, it is different with bus monitor group because some utility function would report errors such as clock frequency comparison function and assertions. For such simulation errors, it would require the error taken place's owner to debug the error, and make the responsibility clear.

## VI. Summary

It is important to achieve testbench's and test case's vertical reuse in system level verification, but it needs to pay additional attention system level verification is far more different with module level verification:

- Each subsystem's design and verification need consistent integrated environment in SoC level.
- Besides the original function verification requirement, new system level environment should also make the consistence between tests and function coverage definition. In this case, both of them take the C code, so the system environment, test and function coverage got uniform maintenance.
- At system level, it also requires advanced system debugging feature support. As both of HW integration level and SW test level are raised, the debugging level should be also raised. It is expected the simulator debug is to shift more to the SW style especially for a large complex system. Therefore, the processor operation analysis feature is initiated for this target.
- At the same time, the function is no longer the only focus at system level, but it also includes the performance analysis and power estimation. Therefore, Hawkeye also realized the performance analysis feature to enable real-time monitor and evaluation at any data transmission point, and also analyze the bandwidth balance among multiple data transmission points.
- From the point of project maintenance, the overall Hawkeye solution should be just maintained centrally by one engineer, and regarding nearly hundreds system level verifiers would get benefits with this component, the advantage is very obvious considering the resource input. It is also because of its high configurable feature and scattered utilities, the maintainer would choose some features only or enable the features one by one in critical project phase.

Based on the analysis of Hawkeye and its necessity to system level verification, it should be sensed that Hawkeye's kinds of features are as adhesive compositions while integrating subsystem and module verification components. Hawkeye figured out the possible blind system test coverage areas, and proposed practical and well-structured solution to achieve the verification completion from low level verification to high level verification, and also enhance the system debug vision to make the complex system debug simpler.

## VII. Prospect

The performance analysis feature has been already implemented, and the future research topic would be about power estimation. As the clock frequency and waveform dump utilities have been implemented, and it is considered how to combine those basic utilities to assist the general power analysis flow. Moreover, it is interesting to analyze the already generated power statistics by the power analysis tooling such as PowerArtist, and then apply specific algorithm to train a power estimation model. The objective model is to make accurate online power estimation with RTL simulation.

Meanwhile, no matter if the test information is displayed in simulation log window or performance pictures are drawn, those ways are deeply user defined. As the Hawkeye solution got extensively use in the project, it is expected to adapt the interfaces and features of Hawkeye to the simulation debug tooling Verdi. Via the advanced debug feature, it will be available to mitigate the processor operation analysis feature, performance analysis features, and extracted clock and bus transaction information to the transaction recording window of Verdi. Such a proposal would not only unify a consistent system level debug interface, but also make it easy for system verifier to focus on only one box containing all of tools.

## ACKNOWLEDGMENT

Sincerely appreciating the Hawkeye practices in projects from Weikai Wang and Xuan Shi, their work and feedback accelerated the solution development. We also thank to the assertion checks to the SoC common cells such as sync-cell and memory initiated by Haibo Shao. The performance analysis prototypes implemented by Jiazhen Gou is the basic reference for the performance analysis feature of Hawkeye. Xinpan Man dedicated in the system level register coverage and function coverage collection and analysis by Hawkeye. Xiuqin Zhang adapted Hawkeye to a subsystem and finished the register coverage collection.

## REFERENCE

[1] Bin Liu, Haibo Shao, and Hongbin Yu, “Best Practices over Enhancing SoC Verification Efficiency”, DVCon China, April 2017.