

# A Novel Framework to Accelerate System Validation on Emulation

**Manoj Sharma Khandelwal, Rinkesh Yadav, Sarang Kalbande, Garima Srivastav**

Samsung Semiconductors R&D India, Bangalore

[manoj.k@samsung.com](mailto:manoj.k@samsung.com), [rinkesh.y@samsung.com](mailto:rinkesh.y@samsung.com), [sarang.mk@samsung.com](mailto:sarang.mk@samsung.com),  
[s.garima@samsung.com](mailto:s.garima@samsung.com)

**Hyundon Kim**

Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, Korea

[hyundon.kim@samsung.com](mailto:hyundon.kim@samsung.com)

*Abstract— this paper discusses about the traditional approach used in Emulators to use run time features like Trigger, Dump, Force, Monitor, and Dump, Load Memory. The pitfalls of using traditional approach, this paper then proposes a Novel framework to use these run time features of the emulator and control it from the software running on the DUT itself. The Novel approach creates an abstraction of the underlying Emulation Hardware for the software user who is validating his SOC which is ported on the Emulator.*

*Keywords—Emulator, DPI, Transactor, Validation, Acceleration*

## I. INTRODUCTION

As Semiconductor industry is growing exponentially, which keeps increasing Design complexity of SoC, it is big challenge for industry to validate its chip on time and stay in competition to the market. Simulation verification of SoC is time consuming so most of the SoC level tests scenarios are moved from Simulation to Emulation to accelerate verification. With each SoC Cycle, more test cases and scenarios run on Emulation as compared to Simulation. With this comes a requirement to have Framework for the Emulation platforms so that the test cases can be ported on to the Emulator seamlessly and in a user friendly manner.

Simulator have a framework which has common system Verilog tasks which does Trigger-Dump, Force-Release, Read-Write to a memory, Get signal values etc. from DUT or Test bench. Emulator does not have these features to be used from DUT directly. These features are only available at run time using commands or software functions from Test bench. There is no way to control them from the Emulator hardware where the DUT is running. This paper discusses on this problem and a novel approach to overcome it.


## II. TRADITIONAL APPROACH

Traditional approach to use these run time features on Emulator was through Commands provided as part of library, but these commands needs manual Intervention and cannot be control through software, it is also not as close to the simulation framework.

For example, if a user wants to start a dump before a failing function in C code running in his design under test and stop it once the function execution complete. This to be done on Emulator we need to either call the API from the C Test bench on the Emulator or using the run time commands which are part of Emulation Framework. There is no way we can control this from the Emulator. In addition, user needs to find exact signal, which he can trigger when the function is executed, or the timestamp at which the function executes to take a dump on Emulation hardware. One more example where the user is limited on emulation side is to model a PLL so he can generate all possible frequencies. The simulation models cannot be directly port as synthesizable code to the Emulator. This limits the number of scenarios, which can be run on the emulation unlike simulation.

A. *Dump Example:*

Let's say you have a function running in software which you need to debug and a take a waveform dump. Traditional approach is to find out the program counter of failing function, put a trigger on it in the run script and run for x amount of time, or find out the time stamp when this function gets executed, run for that time in the run script and then start/stop the dump. For example if we want to take a dump when *cmd\_initialize()* is executed and stop the dump once it is executed. We need to find the address of this function in the memory and trigger on the processor program counter, and then take the dump for some amount of time.



```

void initialize() {
  clock_config();
  reset_config();
  cmd_initialize();
  cmd_status();
}
  
```

Figure 1: Dump Example to take waveform for function *cmd\_initialize()*


*Pitfalls:*

Below are the things user needs to take care in order to take the dump for the function under debug.

- Find out the Hierarchy of signal he wants to trigger on.
- Get the timestamp at which the command is executed.
- Get the runtime for which the *cmd\_initialize()* function is executed.
- Stop the trigger based on the function execution completion.

B. *Force Example:*

Let's say a user wants to force a signal just before the function *cmd\_initialize()* (After *reset\_config()*). Traditional approach to use to keep an expression when the run time reaches the function stop and force the signal and run again.



```

void initialize() {
  clock_config();
  reset_config();
  cmd_initialize();
  cmd_status();
}
  
```

Figure 2: Force Example to force signal before function *cmd\_initialize()*

*Pitfalls:*

Again the user needs to take care of below ,

- Find out the hierarchy of signal where he can trigger for function *reset\_config()* completion.
- Stop the run once the trigger point is hit.
- Force the signal he wants to force.
- Run again after forcing the signal.

### III. NOVEL APPROACH

We need a Framework where Emulator users can control the Run Time Features like Trigger-Dump, Force-Release, Read-write memory, and change frequency output of PLL etc. from the software running in DUT at run time. Emulator has a Transactor-based framework, which can be at advantage to achieve a more novel approach to control run time features of hardware and accelerate the validation.

In This Framework, which we implemented, whenever a register write from CPU in the design under test happens, a DPI call is trigger from the Hardware side of Emulator, which then enables the run time feature on the Emulator using C++ API. For example, as discussed above we can control the dump using a C Dump function. Whenever this function is called, the software running on the CPU will write to a predefined address of a system function register, which already exists in the DUT. Now this generates a transaction at the AXI boundary of the block where the system function register is being implement. Once this happens the address, data and valid signals of the transaction are extracted and fed to a bus functional model. This bus functional model converts these signals to a DPI transaction using import and export commands supported by Emulator. This then in turn triggered the dump on and off API function calls of the hardware using the libraries provided. This way user can control the dump behavior of the Emulator without having need to call any command or function in test bench. User can control it from the software running in DUT. The figure Figure1 gives the top-level implementation we have used in this novel framework.

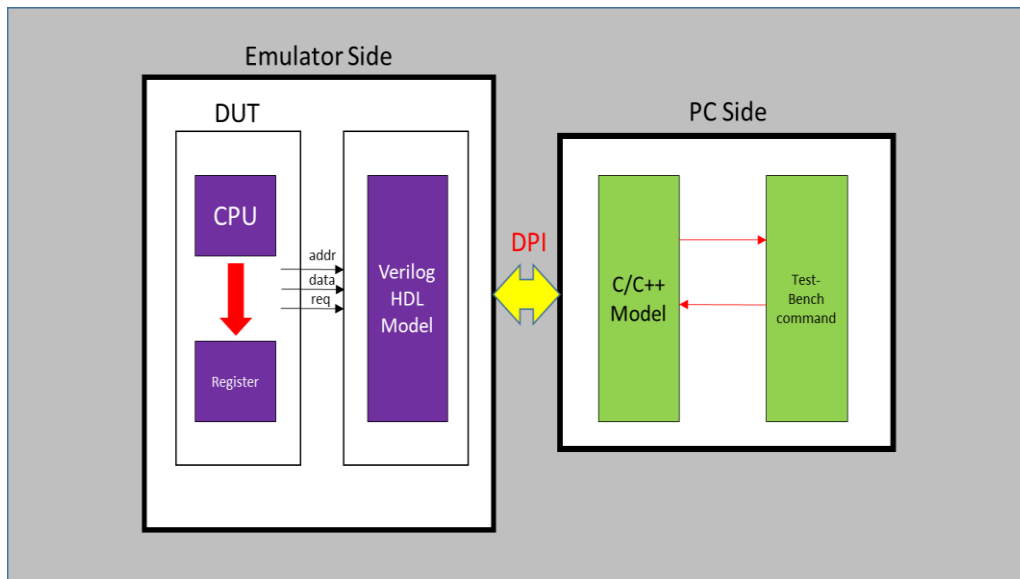


Figure 3: A Novel Framework to Accelerate System Validation on Emulation

Another example, which we discussed in the start, is that of a PLL. We design hardware PLL such that it supports run time changing of PLL Frequency. It includes PLL emulation model, which enables DFS operation with “DPI-C” interface in C environment, and based on the input frequency and Multiplier/Divider values, output frequency is calculated by the C test bench functions. The calculated output frequency is further divided based on the requirements and supplied to respective clock wires at run time. This frequency division helps in optimizing the frequency requirements and help user dynamically shift frequencies similar to the flexibility, which the user has in simulations. User can also now test Dynamic frequency scaling (DFS) feature. In Dynamic Frequency Scaling, the frequency of a microprocessor can be automatically adjusted depending on the actual needs to conserve power.

Below is the sequence of four steps which takes place from DUT Software (C code) to Emulator run time.

1. A register write from the CPU in the DUT is initiated, based on run time C function software wants to control.
2. Bus Functional model then extracts data at the AXI boundary from this register write.
3. After this based on the extracted data a DPI call is triggered from the HW side of Emulator to PC side.
4. This then enables the run time feature on the Emulator using C++ API. This is as per the run time feature step one originally initiated.

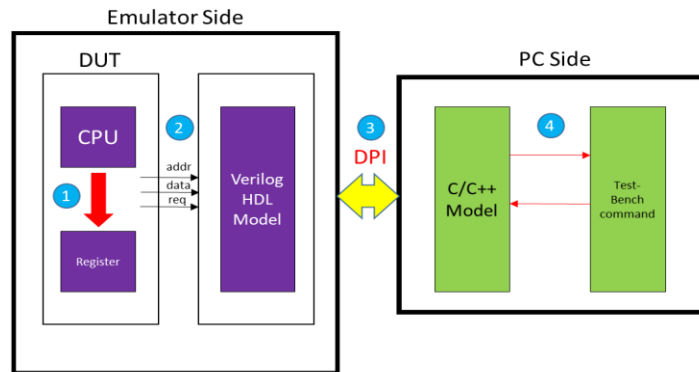


Figure 4: Sequence of Steps in Novel Framework to Accelerate System Validation on Emulation

Step1: CPU write to a Register

In this step the CPU writes to a register based on the function which the user intends to execute. A unique address and the data is mapped based on the user function.

```

void start_dump() {
    volatile uint32_t *addr = (volatile unit32_t *) (unit64_t) (ADDR_BASE + (id&0xff)*4)
    *addr = (unit32_t) ((id&(1<<31)) | (1<<8) | 0x30);
}
  
```

Step2: AXI extraction in BFM

In this step based on the register write CPU does, the address and data are extracted from the AXI bus. This bus is the same bus on which the CPU has initiated a write transaction.

```

OPERATION #( .CPU_NUMBER(`CPU_NUM), .BASE_OFFSET(`HOST_CPU_OFFSET) )
u_operation (
    .reset_n      (reset_n),      // reset
    .osc_clk     (top.systemclock), // clk
    .clk         (top.dut.CLK),
    .addr        (top.dut.ADDR[15:0]),
    .data        (top.dut.WDATA[31:0]),
    .sel         (sel),          // select
  )
  
```

Step3: DPI call from Hardware to Test Bench

In this step based on the extracted data messages are sent from Hardware to DUT using standard DPI call provided by the emulation tool.

```

import "DPI-C" context function void operation(
    input bit [63:0] simtime, input bit [3:0] cpu_id,
    input bit [31:0] length, input bit [63:0] message1,
    input bit [63:0] message2, input bit [63:0] message3,
    input bit [63:0] message4
);
  
```

Step4: Run time Feature control using C++ API

In this step based on the message received in the test bench a C++ API is called, and the corresponding run time feature of the Emulator is triggered

```

extern "C" void operation(
    const svBitVecVal* simtime, const svBitVecVal* cpu_id,
    const svBitVecVal* length, const svBitVecVal* message1,
    const svBitVecVal* message2, const svBitVecVal* message3,
    const svBitVecVal* message4) {
  
```

```

if (is_dump_on) marg_xe_cmd("database -open waveform_result -cont", NULL);
else marg_xe_cmd("database -close waveform_result", NULL);
  
```

#### A. Dump Example:

Let's now consider the Dump Example discussed in the Traditional Approach[II] section. User can now have a function let's say *start\_dump()* and *stop\_dump()* before and after the function which is under debug. A user can also specify the *dump\_time()* and then do *start\_dump()* also. This way user can exactly take dump for the time where user wants to debug the function. These functions to start and stop dump are written with the four step approach discussed in the above novel framework[III].

```

void initialize() {
    clock_config();
    reset_config();
    start_dump();
    cmd_initialize();
    stop_dump();
    cmd_status();
}
  
```

```

void initialize() {
    clock_config();
    reset_config();
    dump_time();
    start_dump();
    cmd_initialize();
    cmd_status();
}
  
```

Figure 5: Novel Approach to take waveform dump for function *cmd\_initialize()*

With this approach the user need not care about below mentioned things when running his software,

- User need not be aware of Hierarchy of signal to trigger.
- The run time for which the function takes.
- The timestamp on Emulator Hardware run time when the function is executed.
- Stopping the Emulator run manually.

### B. Force Example:

Let's now consider the Force Example discussed in the Traditional Approach [II] section. User can now have a function let's say *force\_signal()* to force any signal at runtime. These functions to force a signal at runtime is written with the four step approach discussed in the novel framework[III].

```
void initialize() {  
    clock_config();  
    reset_config();  
    force_signal();  
    start_dump();  
    cmd_status();  
}
```

Figure 6: Novel Approach to force a signal

With this approach the user need not care about below mentioned things before forcing ,

- User need not be aware of Hierarchy of signal to trigger.
- The timestamp on Emulator Hardware run time when the function is executed.
- Stopping the Emulator run manually and forcing the signal.

### C. PLL Example:

Traditional approach to change the clock frequency of the clock on Emulator was through commands provided as part of library, but these commands needs manual Intervention and can't be controlled through registers programmed in the DUT to configure the PLL. Earlier, we used to assign/force PLL frequencies before start of emulation design run, and any subsequent change in frequencies needs a re-run. We need a Framework where Emulator users can control the PLL frequencies using register configuration just like in silicon. Emulator has a transactor based framework which can be leveraged to achieve a more novel approach to control the frequencies at the output of the PLL.

In this framework which we have implemented, whenever a register write from CPU in the design under test happens to configure the clock frequency of PLL, a DPI call is triggered from the Hardware side of emulator. This DPI call holds the multiplier and divider values of the PLL based on which the clock frequency output of PLL is decided. This DPI call then enables the run time feature on the Emulator using C++ API. For example, whenever a divider value of PLL is changed, the divider value signal of the transaction are extracted and fed to a bus functional model. This bus functional model converts this signals to a DPI transaction using import and export commands supported by Emulator. The Output frequency of the PLL is calculated using this Multiplier and divider value in the Test Bench side. This then in turn triggered the clock frequency configuration API function calls of the hardware using the libraries provided. This way user can emulate the PLL behaviour on the Emulator without need to call any command or function in test bench. User can control it from the software running in DUT.

With this approach now user can run scenarios where he has to change the frequencies of different PLL and validate different clock ratios. User can validate dynamic frequency scaling feature of the SOC.

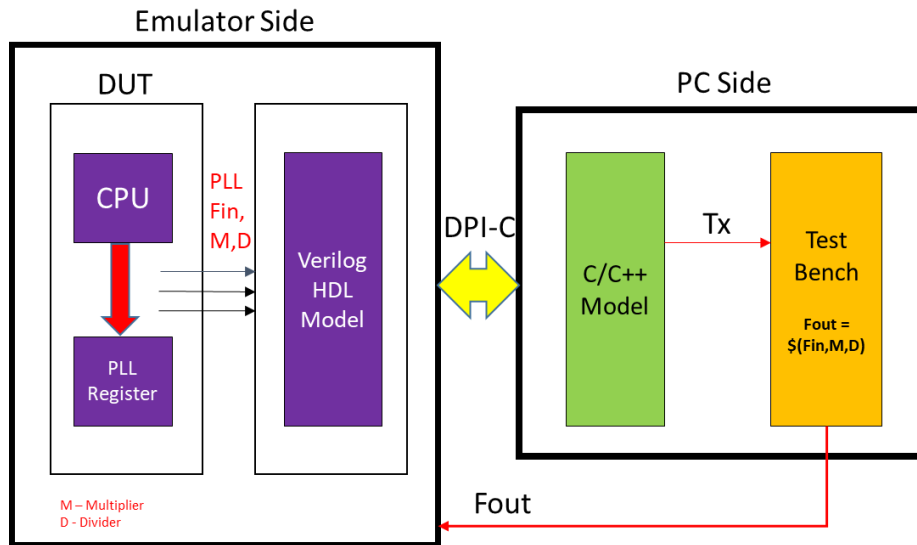


Figure 7: Novel Approach to validate Dynamic Frequency Scaling feature of SOC

#### IV. APPLICATIONS AND LIMITATIONS

This emulation framework is particularly useful to validation users who can now control the emulation hardware from the Software running on DUT itself. Users can use this framework to perform below run time features of Emulation from software itself.

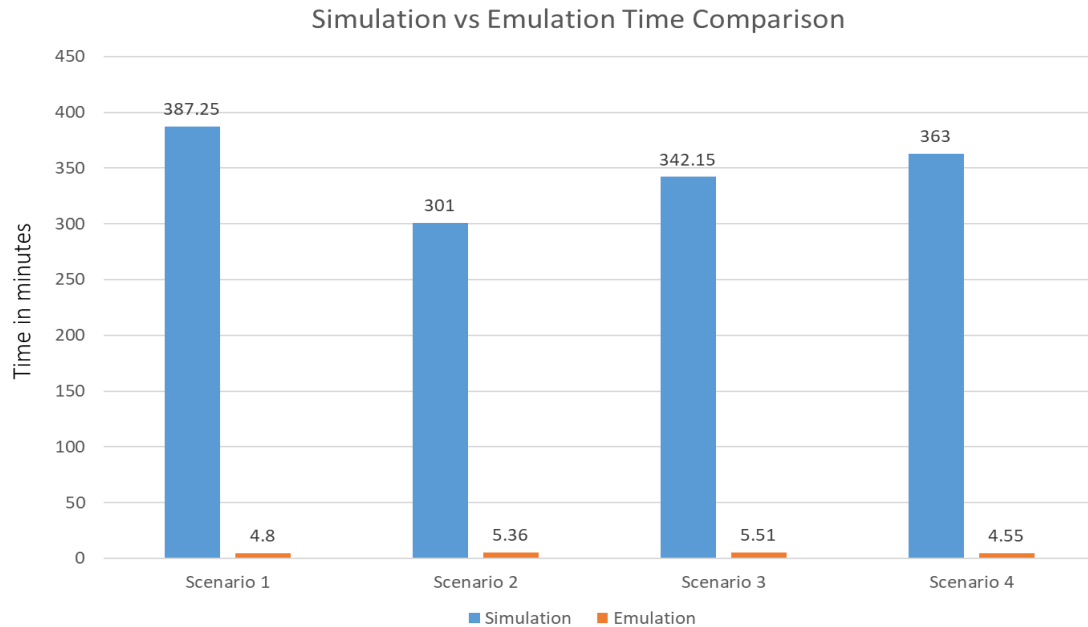
- Trigger and Dump
- Dynamic Frequency shift
- Dump and Load Memory
- Force and Monitor Signals
- Generate System Level Events

This emulation framework comes with below limitations,

- In some scenarios as this functions cause delays in execution, It can impact the functionality if there is dependency on the delays caused by DPI calls.
- The run time performance of the Emulator may be impacted because of DPI calls.

## V. PRELIMINARY RESULTS

For Dynamic Frequency shift scenarios, we have observed approximately X80 improvement in emulation time as compared to simulation. This helps faster verification of the SOC. Below are the snippet of the results for running dynamic Frequency shift scenarios run on Emulator as compared to Simulator



## V. CONCLUSION

This emulation framework is particularly useful to validation users who can now control the emulation hardware from the Software running on DUT itself. Users can debug their code by easily keeping trigger from the software itself. With this framework, they can dynamically control the frequency as in silicon by programming the PLL registers and validate broader frequency range. This framework enables to get access to memory and signals, force signals in DUT from software, reducing validation turnaround times

## VI. ACKNOWLEDGMENT

Thanks to the emulation platform vendors Synopsys and Cadence for their continuous support.

## VII. REFERENCES

- [1] Synopsys ZEBU User Guide
- [2] Palladium User Guide