2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION INDIA

Towards early validation of firmware using UVM simulation framework

Reducing Time to Revenue

Amaresh Chellapilla, System Validation Engineer, Intel Technology India Pvt Ltd, Bangalore, India
(amaresh.chellapilla@intel.com)

Pandithurai Sangaiyah, Engineering Manager, Intel Technology India Pvt Ltd, Bangalore, India
(pandithurai.sangaiyah@intel.com)

This paper demonstrates a novel approach to achieve early firmware validation which helps in ensuring preparedness and gaining confidence for post silicon validation & debug of a semiconductor chip or product. The framework simply reuses pre silicon RTL verification environment along with test framework. What it demands is additional resource and time investment during pre-si validation of RTL and as ROI enables us with an optimized and functionally stable firmware well ahead of silicon availability. Utilization of this approach would help unearth critical and corner RTL bugs, reduce the time to revenue, provide window to improve post-si validation quality by enabling engineers to focus on critical product values. The approach is perused from idea to implementation and demonstrated through use cases in this paper.

Keywords—Early firmware validation; Shift-Left time-to-revenue; post silicon readiness

I. INTRODUCTION

As ever the post silicon validation and debug challenges continue to be daunting and exacerbated because of increasingly heterogeneous designs which are inevitable so as to compensate for diminishing power and performance improvements from node shrink. As the design architectures adapt complexity so do the SW and FW that are going to surface on semiconductor products and thus emphasizing the need to re-visit the priorities in a chip's product cycle which traditionally has been weighted around and until tape-outs. Today it calls for a focus on post tape-out frame which eventually causes the time to revenue. As a result many new tools and flows have materialized to help maximize the post-si efficiency; some of the most commonly known and adapted ones being the virtual platforms like Simics solutions and complex FPGA solutions for emulating the target HW. Nevertheless the solutions may often cost us more lab spaces, additional HWs, new licenses and new skill development.

The proposed approach allows a step further towards addressing a part of the challenge which is felt obvious but hardly isolated as a separate entity among all the ingredients involved in post silicon validation. It suggests on validating the complete Firmware destined for the product ahead in time even before the first silicon is available including the programming sequences and configuration settings. This is a step introduced in pre-silicon validation stage preparing towards post-si validation helping deal with many high probabilistic uncertainties in the time-to-revenue window. Above all, the approach simply reuses the pre silicon UVM framework, used for RTL simulation, to integrate the 'C' based firmware utilizing System Verilog's DPI (Direct Programming Interface) import and export features, requires a C compiler like GCC (GNU Compiler Collection) co-existing with the RTL simulator which is an easily available trait in simulators now a days. So the cost to company for adapting this approach is considerably less as what it demands in terms of cost is more time from engineers during pre-silicon validation stage, and a one-time effort of bringing up the setup to enable working with FW in RTL environment and FW development team to indulge ahead of time. Besides, the approach exploits the open source 'C' code coverage tools like GCOV and LCOV to extract the Firmware code coverage achieved during its validation through RTL test bench. While discussing the implementation details in further sections it has been responsibly taken care to notify of other benefits of adapting this approach.

II. METHOD TO QUALIFY THE FIRMWARE IN RTL SIMULATION ENVIRONMENT

A. *Bird's-eye view – UVM and Firmware amalgamation*

On an abstract level the environment through this approach can be assembled as follows where 'C' based firmware is integrated with UVM framework. A logical mux would differentiate the firmware sequences and settings from System Verilog sequences driven on to the RTL DUT. Except for this change rest all the system Verilog entities in the RTL test bench environment would act unchanged and irrespective of the source of sequences.

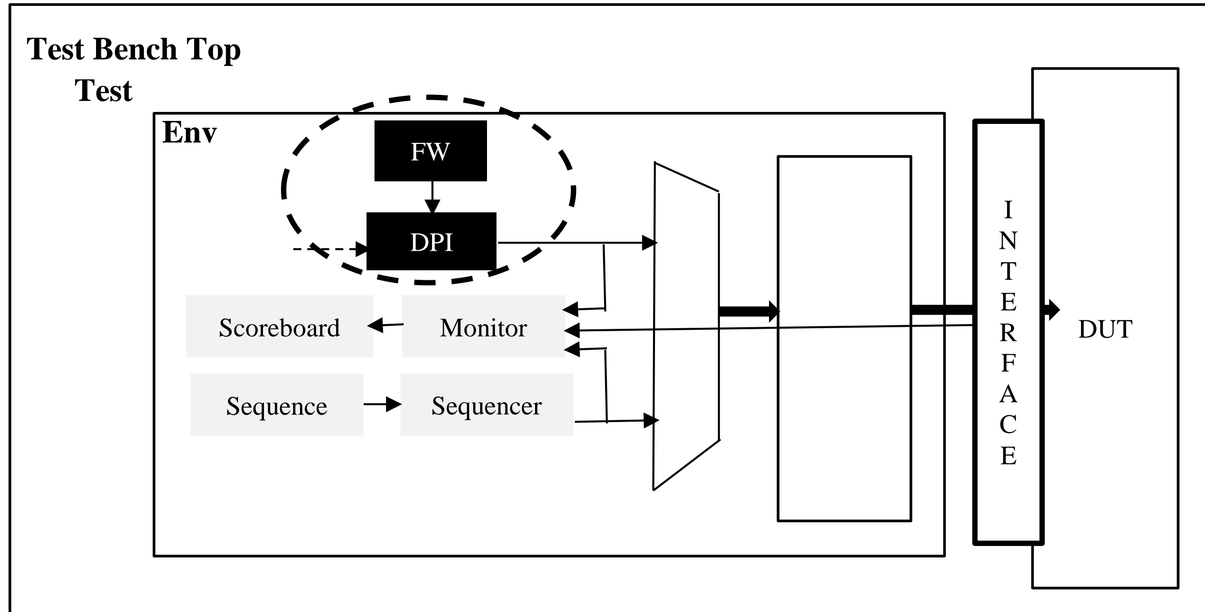


Figure 1 UVM and Firmware Amalgamation

B. *Firmware's abstraction layer*

The outcome of the proposed approach is an integral framework which allows to integrate the firmware with system Verilog RTL simulation environment, but the implementation of the same may differ from one developer to another. Usually for a considerably complex sub system (like DDR) the firmware would consist of several APIs targeting exclusive functionalities. To simplify the discussion it may be fair enough to distribute the firmware APIs into two categories – boot time and run time. Speaking of run time sequences, those would be dedicated and most of the time may not be bundled in any specific order i.e. the APIs called on need basis such as frequency hops, power down, self-refresh entry/exit (in case of DDR), watchdog reset, etc. Boot time sequences unlike the run time ones always need to follow an order of execution as defined by the system architecture. The reason behind discussing the core difference between types of sequences is to find out a better way of minimizing the DPI-C imports into System Verilog.

There may be about 25 or more individual APIs in firmware for complex systems. In order to import those to System Verilog we would need to define them as 'extern C' functions/tasks which is fine as we can do this by defining an 'extern C' function/task surrounding the target function/task.

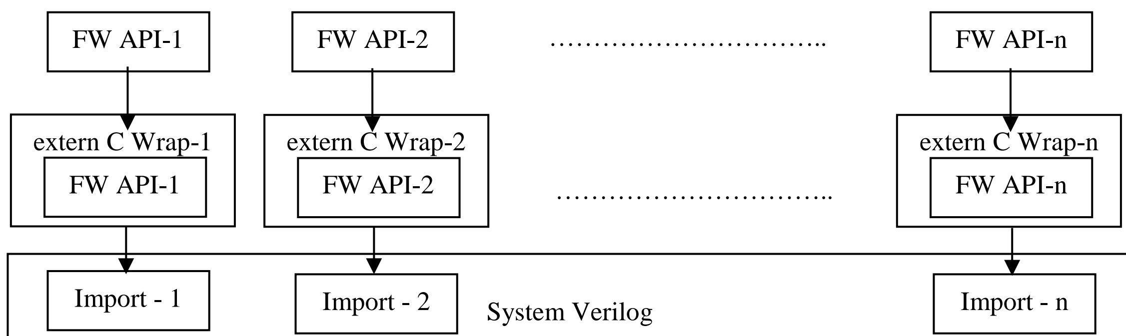


Figure 2 FW to SV Data Flow without Optimization

System Verilog	System Verilog

Boot sequence:
function int sys_init()
    ……..
    Call Import-1
    Call Import-2
    …..
    ….
    Call Import-p
    …..
endfunction
 	
Import 1, 2,…, p functions can be gathered into one import
	Run time sequences:
	function int sys_run1()
	    ……..
	    Call Import-q
	    …..
	endfunction
	
	Function int sys_run2()
	    ……….
	    Call Import-r
	    …….
	endfunction

Figure 3 Individual Imports for FW APIs (No Optimization case)

As shown above, in case of boot sequence there are multiple individual imported ‘C’ APIs called in an order. We can instead create a single ‘C’ API which can wrap all the boot targeted APIs into one and eventually a single DPI-C import would do the work. This is one context where the ‘C’ wrapper resolves the problem of crowded SV DPI imports. Before looking into the implementation change, let’s look into another problem that the same wrapper resolves.

Firmware APIs usually need input parameters/function arguments such as a target frequency that we need to system to boot to, a set of initial register configurations which may be encapsulated as a C structure or array for example, and in case of run time sequences there may be several previous and next values (or frequencies for frequency hops), or a specific system control parameter to drive the firmware into a needed direction, and the list goes on. Passing the arguments to the ‘C’ firmware APIs from System Verilog is going to pose huge numbers of challenges and difficulties and at some points it may not even be possible just because the definition of DPI-C does not support the data types for such parameters.

System Verilog

```
typedef struct{
        byte p;
        bit [0:1][0:2] q;
        int r
}sv_struct-1;

typedef struct{
        bit[0:2][0:2] z
}sv_struct-2;

//Import-1 : ‘C’ FW API-1
import “DPI-C” context function void c_func-1
(input sv-struct-1 sv_S-1);
//Import-2: ‘C’ FW API-2
import “DPI-C” context function void c_func-2
(input sv-struct-2 sv_S-2);
```

Complex arrays and structures may have to be created in SV in compatible to C data types

‘C’

```
#include “svdpi.h”

typedef struct{
        char p;
        //Impl specific
        PACKED_ARRAY(2*3, q);
        int r;
} c_struct-1;

typedef struct{
        PACKED_ARRAY(2*2, z);
}c_struct-2;

//FW API-1
extern void c_func-1 (const c_struct-1 *ptr){
// structure passed by reference
……..
}
//FW API-2
extern void c_func-2 (const c_struct-2 *ptr){
}
```

Figure 4 Example of complex Data Type Compliance requirement (No Optimization case)

In order to resolve the discussed challenges it would be wise to initialize the parameters needed by the firmware in the firmware wrapper (which is in ‘C’) itself so that the data type restrictions for System Verilog to ‘C’ compatibility would not be in picture at all.

With a ‘C’ type firmware wrapper addressing the above two challenges, the DPI interfacing would be easier from SV side.

System Verilog

```
//No structures need to be created and initialized at
//SV side
//Instead of importing individual FW APIs import
//just the C wrapper func defined for boot
//For run time functions make the import easier with
//only simple or no function arguments from SV

//Import for Boot time
import "DPI-C" context function void
c_wrapper_boot_func (int freq);

//Import for Run time
Import "DPI-C" context function void
c_wrapper_run_func_q();
Import "DPI-C" context function void
c_wrapper_run_func_r();

//Boot
function int sys_init()
        int boot_freq = x;
        c_wrapper_boot_func (boot_freq);
endfunction

//Run time
function int sys_run1()
        ….
        c_wrapper_run_func_q();
        …
endfunction
```

‘C’ Firmware Wrapper

```
#include "svdpi.h"

//Declare and initialize the structure variable in 'C'
//wrapper
struct{
        char p;
        PACKED_ARRAY(2*3, q); //Impl specific
        int r;
        PACKED_ARRAY(2*2, z);
} big_c_struct = {8,{1,2,3,4,5,6},3,{1,2,3,4}};
const big_c_struct *ptr;

//Define the wrapper boot function combining all
//needed FW APIs, pass parameters from C structure
extern void c_wrapper_boot_func (int freq){
// structure passed by reference
Call FW API-1: c_func-1(ptr->z)
Call FW API-2: c_func-2(ptr->p, ptr->q, ptr->r)
……
Call FW API-p: …
}

//Define wrapper function for run time APIs with
//parameters from C structure, easing DPI calls from
//SV
extern void c_wrapper_run_func_q (){
Call FW API-q: c_func-q (ptr->…)
}

extern void c_wrapper_run_func_r (){
Call FW API-r: c_func-r (ptr->…)
}
```

Figure 5 Optimized FW Wrapper easing SV imports for FW APIs

Let’s take a note that this firmware wrapper is not a part of firmware but a part of our proposed framework. So we have not touched firmware for serving any of our purpose, rather we are using firmware as it is.

With the above optimized implementation our initially discussed data flow as shown in Figure 2 from ‘C’ to System Verilog through DPI reduces to following.

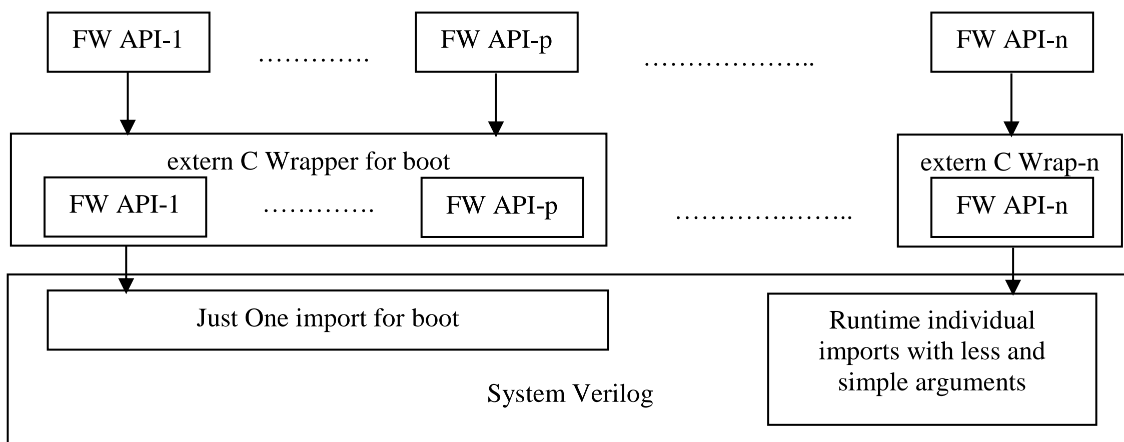


Figure 6 FW to SV Data Flow with Optimization

C. UVM Imports firmware APIs and exports SV APIs

We have discussed the use of DPI-C imports till this point. DPI-C exports also show the usefulness in our approach where our primary goal is to touch the firmware code as less as possible. There is a high possibility that the firmware of any complex sub system (like DDR) may at some points depend on external (to the sub-system) signals as part of APIs. There can be multiple ways to handle such situation but the goal here is to touch the firmware code as less as possible to match the behavior on silicon at RTL level. One easier way as shown in Figure 7, but not helping towards our goal, is to break the API into pieces where such external signal dependency is there and import each such pieces of APIs individually as it would fit in System Verilog test bench. External dependency in a sophisticated firmware is usually implemented through extern or global functions and those are called inside main APIs when needed. So we should break the main API taking these global function calls as break points, also shown in Figure 7.

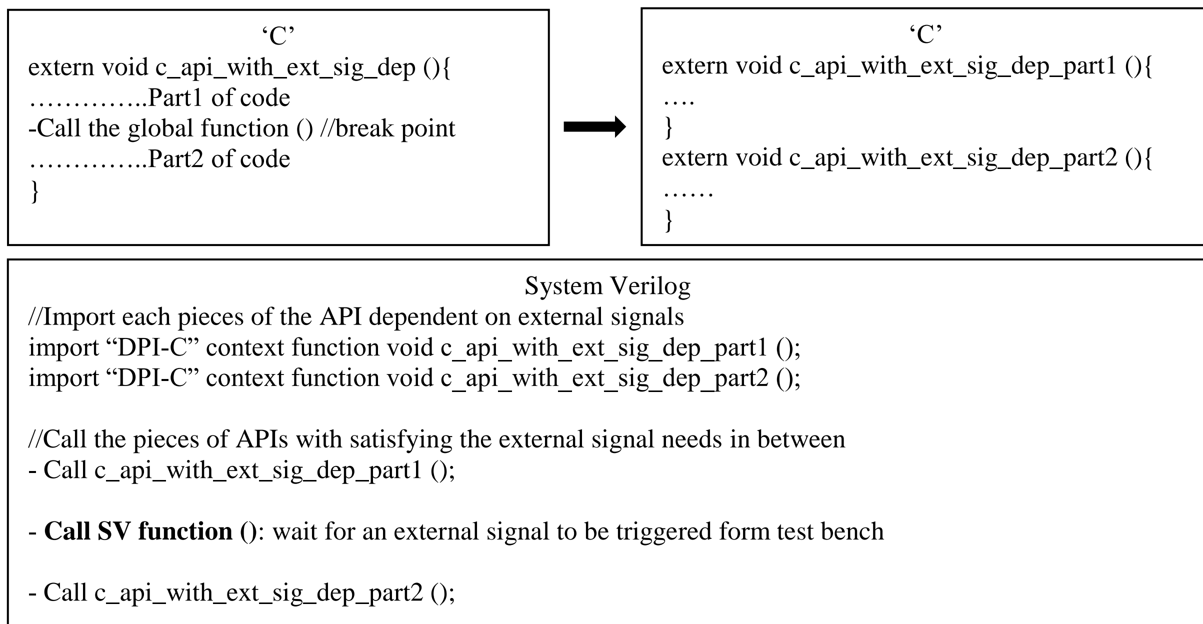


Figure 7 Broken firmware APIs imported into SV to fulfil external signal dependency

Instead we can use DPI export utility and call SV function fulfilling the external signal dependency inside firmware at the points where the global functions are called as shown in Figure 8.

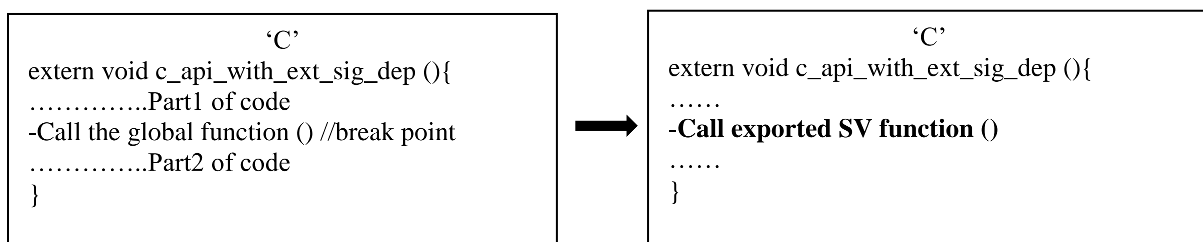


Figure 8 Function exported from SV called in 'C' firmware to fulfil external signal dependency

D. Firmware accessing Registers through UVM RAL

The register writes and reads from the firmware needs to be routed to the register abstraction layer (RAL) of System Verilog which must have been implemented in UVM simulation environment. This would make it possible to write to and read from the registers in RTL level and thus allow the firmware to reach to the DUT. Firmware is eventually all about DUT register accesses both at boot and run time. Though the implementation is dependent on the developers the path from firmware to RAL through DPI would be similar to what is shown in Figure 9.

E. Firmware code coverage extraction, enhancement and optimization

Integrating the firmware with UVM environment provides a way to get the firmware code coverage using any of the available coverage tools for 'C' code entities. The open source tools GCOV or LCOV have proven to be compliant to all the needs to extract the firmware code coverage, analyze, enhance the test plan and optimize the firmware itself. Figure 9 shows at an abstract level the complete picture of the FW integrated with the UVM environment along with the coverage utility.

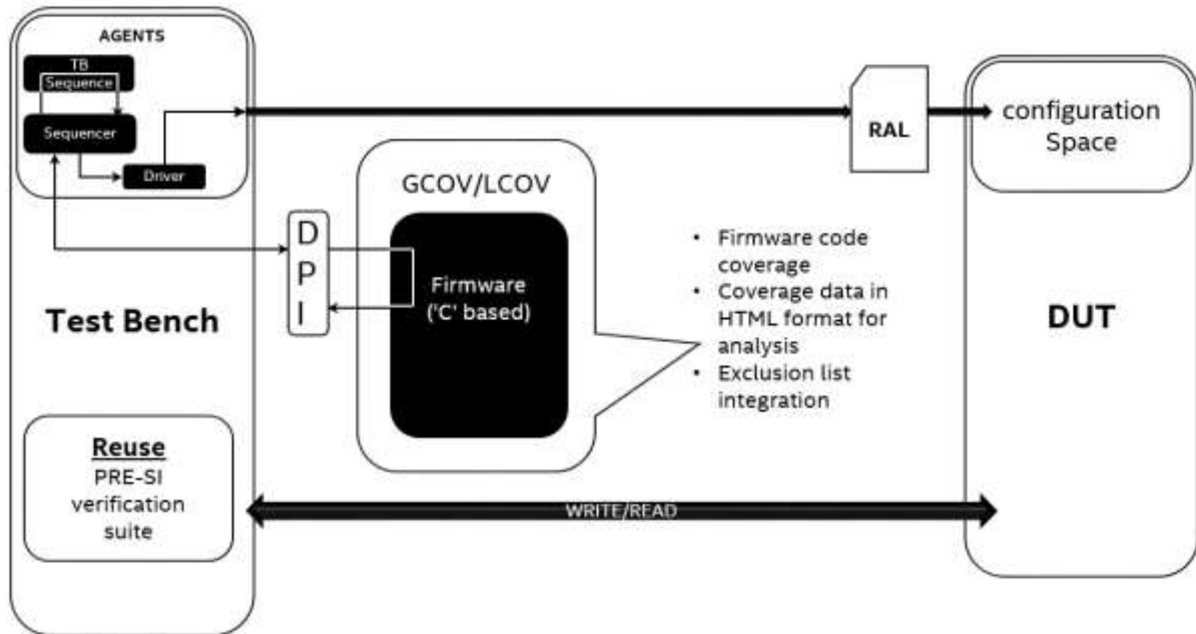


Figure 9 Firmware to SV RAL through DPI and GCOV/LCOV as the coverage utility

All the implementation aspects that the approach proposes have been discussed. Thus the firmware gets functionally validated quite ahead of silicon, optimized in terms of memory foot print, improvised in terms of covering maximum RTL corners and stable enough to be used in FPGA emulation platforms and eventually on the silicon.

Firmware is functionally ready for silicon!

III. Important Guidelines Before Implementation

An important characteristic of this approach is "re-use" of UVM simulation framework. It would be a wise choice to first make sure that the RTL simulation test bench environment (including UVM agents, and test cases) is stable. Following guidelines would help get a healthy start for firmware validation in pre-silicon environment.

- Integrate the firmware into UVM simulation environment only after the environment itself is stable. Otherwise it would be difficult to isolate the issues between environment and firmware code and eventually would require increased effort.
- Provide necessary provision in the UVM simulation framework to allow or include certain features like random delays, wait cycles, etc that may be needed when working with firmware sequences and settings.
- Do necessary upgrades in UVM checkers, monitors and scoreboards to consider firmware's precise cycle timings which might have been assumed to be liberal during conventional RTL simulation.

IV. Advantages Towards Post Silicon Preparedness

We now know that the approach is not complex to understand or implement. Still to utilize it one would also require ample stack of advantages that the approach can provide in order to adapt. Advantages act as wild cards in the initial phase of adaption of new methods into main stream against the very obvious risk factors emerging out of newness. So as many of those we can have as less would be the threat to the method's existence. We discuss all kind of benefits that the proposed approach provide starting from quantitative to qualitative.

A. *Re-Use UVM framework*

The methodology re-uses the UVM framework available for RTL simulation at pre-si stage. Firmware validation using this approach is usually commenced after the RL simulation environment is stable enough functionally so that not much of work needs to be done towards enabling the environment. The following minimal effort is sufficient to bring the firmware validation framework to working state.

- Create an abstract 'C' wrapper around the firmware consisting of top level 'C' APIs
- Implement imports using DPI-C context, integrate into existing UVM drivers and SV test cases as applicable.
- (Optional) Implement exports using DPI-C context, integrate into 'C' firmware APIs as applicable
- Include GCC instructions to compile 'C' firmware and to link the shared objects to SV objects.

Discussed flow is a one-time effort. Once achieved it can keep on providing its benefits till the end of post silicon phase and beyond.

B. *Increased probability of discovering corner RTL bugs*

Simulating RTL with firmware recommended settings and sequences provides an opportunity to find RTL issues that might be hidden and not found yet because of sequence differences. As firmware would usually have much stable and specification compliant programming sequences and settings much closer to actual silicon it is probable that same test cases when executed with firmware may lead to unearth some corner scenarios. Having this capability to stabilize the RTL to a plausible higher extent is a welcome trait. Additionally the memory foot print of the firmware can be calculated and optimized well ahead of time.

C. *Consistency across platforms*

It is a usual practice with SoC validation and FPGA emulation teams to leverage the sequence drivers from the IP teams who must have validated those in the RTL simulation environment. So it would not pose any difficulty is getting all the teams to adapt the firmware sequences and settings into their environment of validation especially because the firmware would be now a qualified one from IP team in RTL simulation flow. This establishes a solid consistency across platforms (IP RTL simulation, SoC RTL simulation, FPGA emulation platforms, and eventually Silicon validation platform) to have exactly the same sequences and settings used and validated throughout the product cycle.

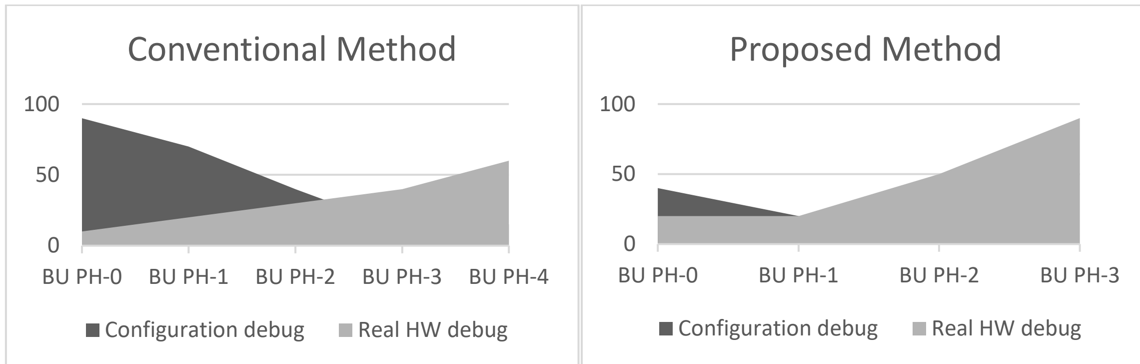
The consistency thus achieved would further streamline the validation flows across different domains, ease the re-generation of issues on any platforms, and create more debug capabilities.

D. *Reduction in time-to-revenue*

Commencing the validation on first silicon with a functionally proven and optimized firmware is for sure a 'dream come true' occasion for post silicon engineers. Among many of its advantages the time it saves on silicon validation is significant. Any issue seen on silicon can now be delegated to HW features or traits that are out of scope of pre-silicon such as delays from analog components, electrical behavior nuances, etc which need to be converged in lab through investigation.

A considerable reduction in silicon bring-up or power ON can be achieved as there would very negligible time required to be spent on dealing with design configuration and sequences which have already been optimized. It has been witnessed to save about ~60% of debug time from configuration space and allow that to be spent on debugging real silicon issues. A left-shift in power ON thus achieved is a left-shit in time-to-revenue.

Please notice the less number of BU phases in below chart with the proposed method in comparison to that of conventional method.



E. Tool to help in debugging post-si issues

The existence of firmware pre-si validation framework opens up additional debug capabilities that we can have for any critical post silicon issues that may trend towards weeks or months of effort. It would be an obvious argument that the time taken for RTL (+ firmware) simulation would not be suitable during the post silicon phase. Given that a critical issue pending root cause for more than week would never hesitate using additional debug capability in hand as this can provide more insight in terms of waveforms, test bus traces, and performance monitor output.

There is an equal probability that firmware may need to implement new sequences during post silicon phase due to a requirement realized never before. In such cases the pre-si validation framework that the approach suggests comes quite handy if the sequence is complex and covers a considerably longer functional path. The change or addition can be integrated within no time into the framework and simulation cycle can be executed with suitable test case. Within hours we can have the behavior in terms of logs, wave forms in hand while simultaneously the sequence can be run on silicon and validated.

F. Boost in Emotional confidence and Creating Opportunities

This trait being offered by the proposed approach is a qualitative advantage emerging out of the new capabilities. With a given knowledge that the firmware is functionally validated, correct by construction and optimized before we commence the silicon validation, an engineer's state of mind is definitely going to be at far positive end compared to that of having an immature firmware. The 'Emotional Confidence' that is presented at this stage opens up many other opportunities among the engineers involved in bring-up or validation. Following are some of the capabilities which have been witnessed.

- Given the process node is stable, and electrical behavior of the silicon has been qualified, the power ON for the chip including boot and beyond can be achieved on Day 0 itself which would call for a celebration.
- This drives the possibility towards quick lab exit from power ON eventually left shifting time-to-revenue.
- It opens up a huge window for post silicon engineers to spend time on actual silicon issues rather than indulging in fixing firmware issues. This may lead to innovations and initiatives which can further help minimize the cycle time or improve efficiency of post silicon validation.

V. RESULTS

The approach has given excellent results in terms of firmware readiness before silicon, helping out sophisticated FPGA/emulation cycles with relatively stable firmware. It also encapsulates pack of advantages that pre-si world offers in terms of observability, debug capabilities, coverage analysis and controllability. A comparison between 3 projects w.r.t firmware's number of line of code change after silicon, time taken for specific firmware intense features in power ON, reduction in time taken to enable other dependent sub systems for silicon has been shown here to give an idea on how the proposed approach helps in left-shift of the time to revenue.

For this exercise DDR sub system was considered as the ingredient as it is fairly complex and does consist intense firmware content. The feature that is in itself complex in DDR sub system is DDR PHY and the training sequences involved. The comparison data below is basically centered on DDR PHY bring-up but also covers data partly from DDR controller.

Table 1 Reduction in lines of code change after silicon

Projects	# of lines of code before silicon	# of lines of code changed after silicon until power ON	% of change done after silicon
Base Project (*no firmware validation during RTL simulation, only partial validation in emulation*)	3432	921 (addition of APIs/sequences, changes in APIs/sequence orders, deletion of sequences, changes in register setting values, change in array sizes)	27%
Project-1 (with firmware validated during RL simulation)	3201	195 (Addition of code to support specific PHY pad trainings for an issue root caused during ATE characterization, changes in some register settings after assessment of silicon behavior)	6%
Project-2 (with firmware validated during RTL simulation)	4350	519 (Code change due to introduction of new frequency plan for some customers, change in other register settings)	12%

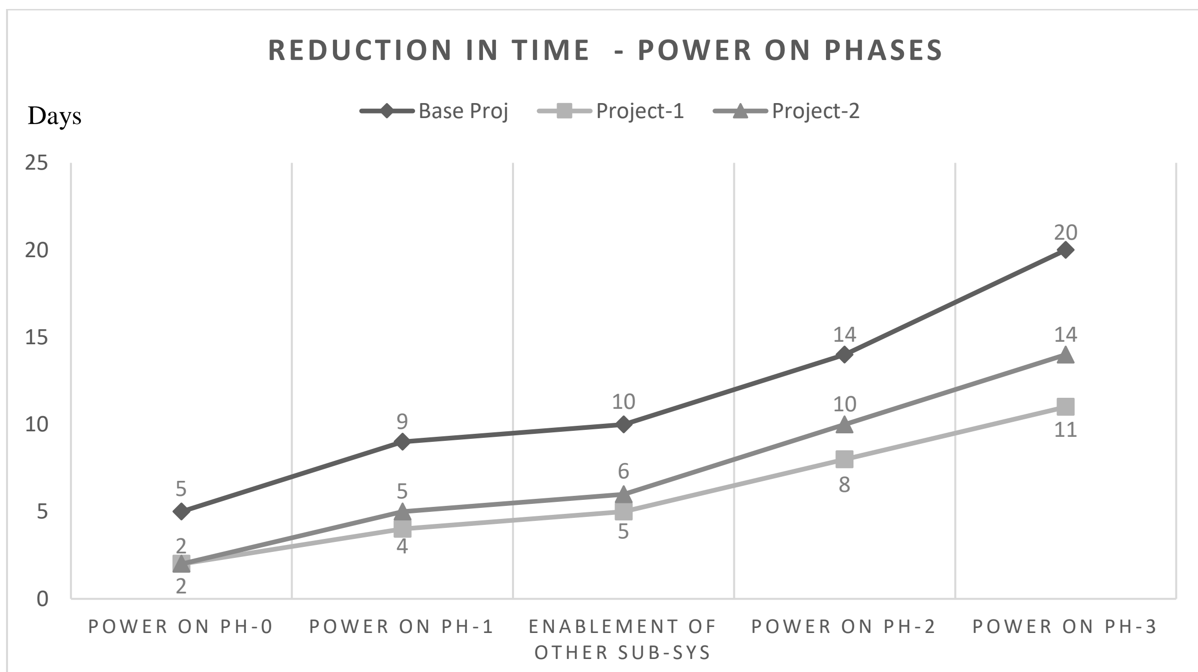


Figure 10 Silicon Power ON phases - reduction in time

The reduction in required number of lines of code change as shown in Table 1 is a result of stabilizing the firmware functionally (sequences as per specification, register settings as close as possible to silicon behavior, optimized memory foot print, higher firmware coverage, and efficient emulation with comparatively much stable firmware) well ahead of silicon. The time spent on firmware debug during silicon is thus minimized and this

effectively brings down the time required to complete the silicon power ON phases. Figure 10 shows the time taken in different phases of silicon power ON with a healthy comparison between projects having firmware not validated during pre-si RTL simulation and those which have. There is a gain of at least ~40% to 50% time in power ON when comparing similar projects (in this case having almost same firmware).

Apart from the above discussed quantitative advantages there are also many qualitative advantages; one of which we have already discussed (in IV.F) as "emotional confidence" before commencing silicon validation with a well-defined, optimized and stabilized firmware.

CONCLUSION

There are a few approaches in the industry now which help in improving the post silicon validation domain. This approach of validating the firmware, targeted on silicon, has opened up lot of opportunities of having visibility and controllability on silicon though it is virtual. Provided this approach is adapted it is quite probable that the over the period the post silicon world can leverage most of the observation from pre-si level. With the current era of new technologies like artificial intelligence and machine learning more advantages can be induced from this approach which is the next possible direction of improvising such beneficial approaches to further heights.

REFERENCES

[1] IEEE Standard for SystemVerilog – Unified hardware Design, Specification, and Verificationlanguage; IEEE Std 1800™ – 2017

[2] Using the GNU Compiler Collection (GCC) - https://gcc.gnu.org/onlinedocs/gcc/

[3] LCOV Code Coverage – WIKI document foundation - https://wiki.documentfoundation.org/Development/Lcov