2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

How to test the whole firmware/software when the RTL can't fit the emulator

Horace Chan
Microchip Technology,
8555 Baxter Place,
BC, Canada, V5A 4V7
horace.chan@microchip.com, 604-415-6000

Byron Watt
Microchip Technology
8555 Baxter Place,
BC, Canada, V5A 4V7
byron.watt@microchip.com, 604-415-6000

***Abstract*-Pre-silicon firmware and software (FW/SW) testing is a necessity for all silicon companies. One of the biggest challenge is when the RTL cannot fit inside the emulator. In verification, it is a common practice to black box unused logic in the RTL to reduce gate count. However, adopting this approach in emulation has unique challenges due to the difference in the architecture of an UVM testbench compared to the actual FW/SW. In this paper, we will present our strategy on how to run the full FW/SW on blackboxed RTL in the emulator using a Hybrid Software Simulator (HSWSIM). The paper will compare the benefit of our strategy over using existing solutions, such as virtual platform. Then the paper will describe the architecture of the HSWSIM and outline its two use models, one to test user space FW code and the other to test kernel space FW code. The final section of the paper presents the results of using HSWSIM to bring up the FW/SW in our latest generation chip.**

I. Introduction

Running pre-silicon FW/SW testing using the emulator has many successful stories in various publications. [1,3] demonstrated Android boot up in emulation. [2] proved the production software went through MISRA-C checks in emulation and hence saved a year of development time post silicon. [4,5] applies software driven verification in emulation to address RTL bugs early on the product cycle without writing complex scenarios in the UVM testbench. [6] enabled early development of HW and FW in the same environment to have bugs found and fixed within hours. [7] allows comprehensive power and performance testing in emulation with the production software. [8] demonstrated Linux boot up in the embbeded CPU and had all the major features of the SoC working prior to silicon.

The most prominent benefit of emulation is shifting left the project schedule by allowing testing of the FW/SW much earlier. The FW/SW team can start developing and testing production FW/SW many months before the silicon comes back from the fab. Running production FW/SW code with the RTL uncovers unexpected hardware bugs that slip through the verification coverage. Allowing the project team to discover and fix hardware bugs before the chip tape-out is a huge time and cost saver compared to the alternative of taping out another revision of the chip. To meet the time to market window and deliver a bug free design, it is no longer an option not to have pre-silicon FW/SW testing. The major challenge is how to test the FW/SW in emulation efficiently.

Other than the initial setup of the emulation flow, i.e. compiling the RTL into the emulator, one of the biggest challenge in pre-silicon FW/SW testing is when the chip is bigger than the capacity of the emulator. Technically speaking, you can always buy a bigger emulator that fits designs up to a few billion gates. Given the high cost of the emulators, this is not always a practical solution. In this paper, we are presenting a strategy to cope with the problem when the RTL is too big for the emulator, and yet it is required to test the whole FW/SW as one piece.

Our approach is inspired by an idea used in verification for ages; when the RTL is too big for the simulator, the testbench often blackboxes RTL logic that is not required by the test to reduce the size of the snapshot. Due to the fundamental difference in how a UVM testbench (UVMTB) is constructed and how the FW/SW is built, there are some unique challenges in adopting the blackboxing strategy. We will first review existing approaches on solving the RTL size problem and outline their drawbacks. Then we will present the workflow of our Hybrid Software Simulator (HSWSIM) and its benefits. We will go over the two use modes of the HSWSIM: 1) how to test non-timing critical FW code running in the Linux user space; 2) how to test the FW code running in the Linux kernel

space and how to stress test the performance of the FW. Finally, we will discuss future improvements planned and share our success story on pre-silicon FW/SW testing.

II. CHALLENGES IN HYBRID EMULATION

B. Testing FW/SW on the Emulator

Figure 1 illustrates a generic scenario of testing FW/SW on the emulator, which mimics how a single chip is used in the production environment. Here are some key features of the scenario:

1) A x86 Linux server controls multiple chips via PCIe bus in the production environment
2) The PCIe bus is replaced by the PCIe speed bridge in the emulation environment
3) The embedded CPU inside the chip is also running Linux
4) The SW running on the host Linux communicates with the FW running inside the embedded Linux via TCP sockets or PCIe mailbox.
5) The end-user has no direct access into the FW. The end-user interacts with the FW through the public API of the SW running on the host Linux

Depending on the different feature sets in testcases in the FW/SW test plan, we synthesize and PnR (place and route) the RTL into multiple emulation snapshots of the embedded device with unused logic blackboxed. The biggest problem is that the ARM core would crash if the FW accesses any address space located inside the blackboxed RTL. Since the AXI/AHB slave logic is missing from the blackbox, there is no AHB/AXI response to a pending request, therefore it completely hangs the AXI bus and stalls the CPU. In verification, it is easy to avoid writing to those address since we construct the UVMTB from ground up and have full control on the executions of the UVM configuration sequences. Moreover, it is relatively easy to hook up a dummy AXI/AHB slave behavior model in place of the black box in the simulation environment. However, those two solutions are not readily available to the production FW/SW running in emulation.

C. Existing Solutions

We have considered existing solutions to work around the limited emulation capacity vs the full design size. We tried the first two approaches (summarized below) in the past, however they didn't address our requirements and left us with a bad experience on pre-silicon FW/SW testing. We evaluated the third and fourth approaches early in the project cycle and decided the HSWSIM approach (which we will describe later) best fits our needs:

1. Testing the FW at block level only
   In our previous project, we tested the block level FW in isolation with its corresponding RTL block. This approach allows us to catch block level RTL and FW bugs early, however we would have to defer the full integration testing of the FW until the silicon came back. Given that most complex bugs are usually discovered in the integration testing of the RTL and the FW, this approach didn't meet our goal on code quality and the committed delivery schedule.

2. Use of #ifdef or makefile parameters to partition the FW
   In our previous project, we tried to compile the FW to match the blackboxed RTL snapshots. It was a very involved task that required adding hacks and patches into the FW code. We spent lots of unproductive hours in adding #ifdef into the code, creating mock function to bypass register access into the blackboxed address space and adding many special parameters in the makefile to compile the FW correctly. This approach is highly intrusive to the FW code base. It creates a maintenance nightmare later in the project cycle and it is proven to be unscalable to a larger design

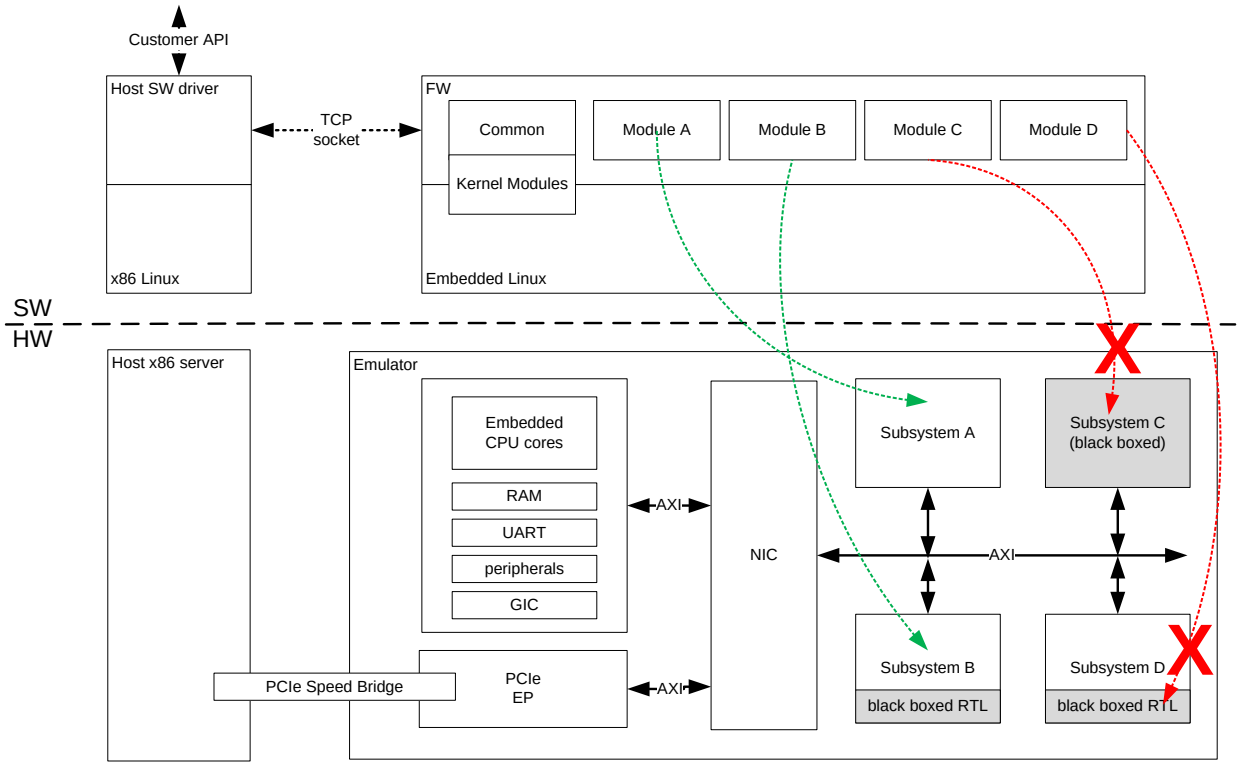

Figure 1. Illustration of testing FW/SW on Emulator

3. Inserting dummy AXI/AHB slave components in the blackboxed RTL
   In the beginning of this project, when we compiled the emulation snapshot, we tried to insert dummy AXI/AHB slave into the blackboxed RTL just like how it is done in the verification testbench. There are two major drawbacks in this approach. First, blackboxing a block of RTL at compile time might be trivial, but it is much more work to create individual blackboxed SystemVerilog modules with embedded dummy AXI/AHB slave functionally. Second, a dummy AXI/AHB slave can only return zero value on a data read, yet we often have to fake some minimum register responses from the blackboxed RTL module, for example when the FW is polling a busy bit. Using a dummy AXI/AHB slave does not solve the problem of how it interacts with the firmware.

4. Use of a Virtual Platform
   Virtual Platform is a proven approach for pre-silicon FW/SW testing as highlighted in the previous papers [3,5]. Depending on the specific test scenario and the availability of SystemC behavioral models, it could be the right choice for a different project. However, in our experience, we found that it is too time consuming to set up the QEMU emulator proxy and create the SystemC behavioral models or the SCEMI transactors from scratch. Also, it does not provide any additional value in helping us to debug our FW. Giving that most of our FW code is running in the Linux user space, and in the beginning of the project where we focus on correct functionality instead of stress the FW performance, we found it more convenient to debug the FW running on the x86 Linux host instead of running it in the embedded CPU. Our approach still allows us to test the FW in embedded CPU later in the project cycle without incurring the overhead of setting up a virtual platform environment.

.

III. HYBRID SOFTWARE SIMULATOR

A. Architecture

Figure 2 outlines the architecture of the Hybrid Software Simulator (HSWSIM). We have a set of scripts to generate the HSWSIM, the SystemVerilog implementation of the registers and the C header files for the FW from the register description files (e.g. IP-XACT). We already have the scripts to generate the required files from previous projects, so it is only incremental work to update the scripts to support the emulation environment. In previous project, we used a pure software simulator to test the basic functionality of the FW code without having the FW interact with the RTL. The pure software simulator supports a shadow register bank to keep track of the state of all the registers in the device. The shadow registers support modeling of the RTL behavior by triggering a callback function when any arbitrary value is written to specific addresses. [6] implements the shadow registers in a dedicated hybrid register server that supports emulator-aware register mirroring. Our solution didn't implement register mirroring due to its complexity and it only provides marginal benefits in certain debug scenarios.

The existing FW header file generation script already outputs bit level register field access functions in a hardware abstraction layer (HAL), such that register access of the FW do not rely on any hardcoded value in the source code. Using hardcoded defined values is a bad coding practice in general. To support emulation, we updated the HAL to redirect the FW register access to the HSWSIM via a TCP socket. Another benefit of abstracting the FW code from the actual hardware register implementation allows better debug control. We can easily change the verbosity of register access log through the built-in logging feature available within the low-level access function.

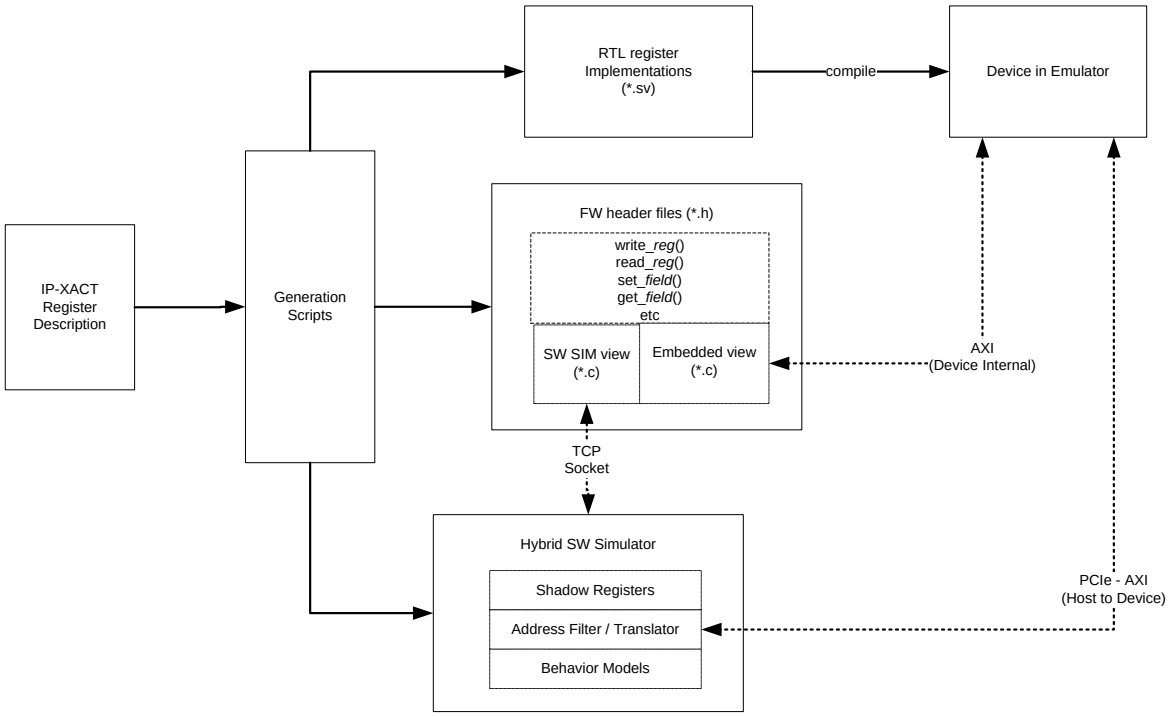

Figure 2. Hybrid Software Simulator Code Generation

It is minimal amount of work to enhance the existing pure software simulator to support hybrid software/emulator simulation. The x86 Linux host already has a connection to the device through the PCIe bus, so that the x86 Linux host has the same visibility into the device address space as the embedded CPU. The only difference is that the

embedded CPU uses a different address mapping than the PCIe address window. The HSWSIM has two main functions when it receives a FW register access from the TCP socket. First it needs to determine whether the address being accessed is blackboxed in the emulator. If it is blackboxed, the HSWSIM behaves like the pure software simulator for this given address and it will terminate the access at the shadow register bank and/or trigger the behavioral model functions. If the address is accessible in the emulator, the HSWSIM will translate the embedded CPU address mapping to the PCIe address mapping and forward the register access to the PCIe bus. [4] uses a similar partition in register access in the real hardware model and in the simulator behavior model. It uses direct memory mapping between the emulator and the software simulator, where we use TCP sockets to communicate between the FW and HSWSIM. The major benefit of the using TCP sockets to facility register access between FW and HSWSIM is that the HSWSIM is agnostic to where the FW is executed. The FW might be running in the x86 host Linux or in the embedded Linux and the HSWSIM couldn't tell the difference.

B. Use Modes

The HSWSIM has two different use modes, depending on which part of the FW is under test, we select the use mode that is most convenient to debug the FW code.

Figure 3 shows the use mode when testing the FW code running in the Linux user space. The goal of the testcase in this early phrase of FW development is to confirm correct functionality of the FW when it interacts with the RTL. The FW code running in user space are mostly non-timing critical functions, therefore the extra latency introduced by register access over the PCIe bus does not affect the correct operation of the FW/SW. In this use mode, both the host side SW layer and the user space SW are running on the x86 Linux host in two separate processes. The production FW/SW communicates with each other using PCIe memory-based mailbox. The HSWSIM simulates the PCIe mailbox with a TCP mailbox. When the FW accesses a register via the HAL layer, the register access is first examined by HSWSIM, it will filter out register access into the blackboxed address space in the emulator. There is only a tiny bare-metal layer program running in the embedded CPU. It boots up the PCIe bus and then puts the embedded CPU to sleep.

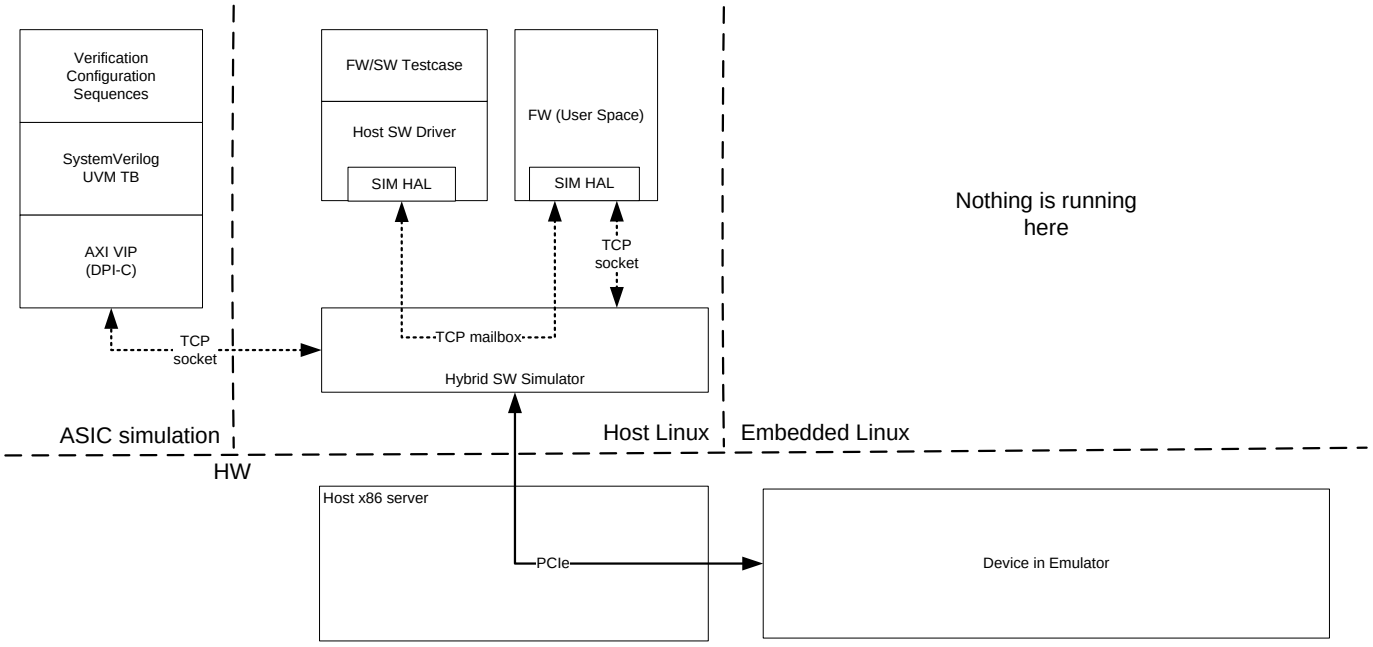

Figure 3. Use Mode on Testing FW (User Space)

There are two major advantages in this approach in debugging the user space FW code. First, running the FW code on the x86 Linux host is a lot more convenient to debug than running it in the embedded CPU. The cross-platform debug turnaround time is slow in the latter setup. After fixing a bug in the C code, not only we have to recompile the FW executable, we need to package the binary along with the embedded Linux into a memory image, then upload it to the flash memory model in the emulator. When the FW is running in the embedded CPU, the choice of the debugger is limited. It can be a plain text-based gdb client or a full-blown memory-hungry GUI-based debugger built on top of Eclipse. When the FW is running in the x86 Linux host, the developers are free to choose their favorite IDE as the GDB front-end attached to the running FW process. Second, in this early phase of FW development, the documentation on how to configure the device is often not up to date. The UVM configuration sequence in the verification testbench is the only golden reference for the FW development team. We can run a modified verification testbench (VerTB) that converts AXI transactions into register accesses into the emulator via the HSWSIM. By mixing and matching bits and pieces of configuration code in the FW and the VerTB, we were able to pin point the configuration errors in the FW much more quickly.

Figure 4 shows the use mode to test kernel space FW code running in the embedded CPU and stress test the performance of the FW. In this phase of FW development, the user space FW code is already debugged and well tested. We can just run the same FW code in the embedded CPU and expect the same result as it is running on the x86 Linux host. The additional FW code is mostly tiny interrupt service routines running in the kernel space of the embedded Linux and the PCIe driver for embedded Linux itself. The FW/SW testcase and the host SW layer are running on the host x86 Linux like in the production environment. The host SW layer communicates with the FW via a PCIe memory-based mailbox and PCIe interrupts. Most of the FW is running the same way as it is in the production environment, except when the FW code accesses a blackboxed register address. When that happens, the HAL layer of the FW will filter the register access and pass it over to the HSWSIM running on the x86 host Linux instead of passing it down to the AXI bus.

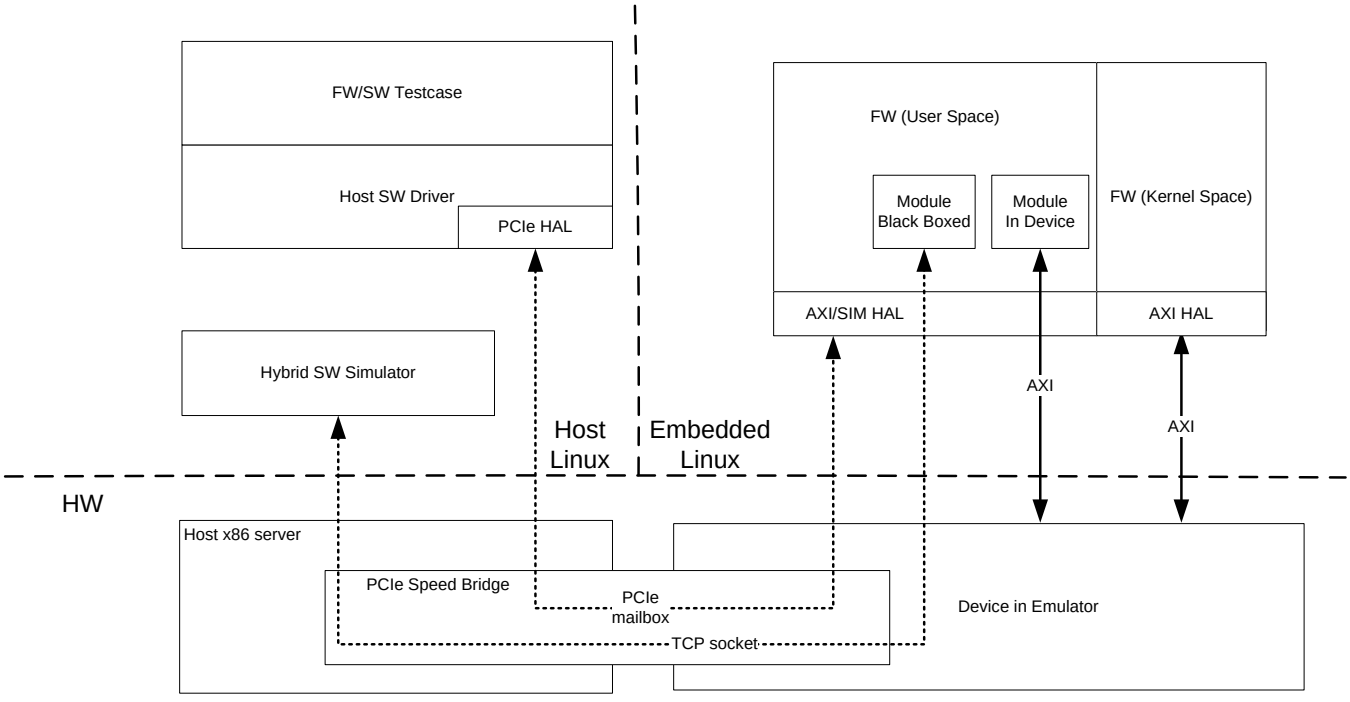

Figure 4. Use Mode on Testing FW (Kernel Space)

Debugging the kernel space FW in this use mode is pretty much the same as debugging it in the actual silicon. We used the same debugging techniques, such as outputting printk to the UART interface, hooking up the DSTREAM probe to the JTAG interface and capturing the PC trace in DS-5 studio. We didn't use any emulator specific debugging techniques, such as dumping the RTL waveform of the CPU or the AXI bus, since all the problems we encountered are found in FW layer. Bringing up the kernel space FW code is a very time-consuming task, fortunately it is only a very small fraction of the code in the FW.

V. RESULTS AND FUTURE ADVANCEMENT

One of the biggest drawback of the HSWSIM is the relative slow speed of register accesses. The execution time of the FW when the it is running in the x86 Linux host is about 10 times slower than when the FW is running in the embedded CPU inside the emulator. The major bottleneck in the register accesses is the PCIe latency. Since a typical FW/SW testcase runs less than a minute on average, it is still acceptable even when the run time slows down to 10 minutes. When the FW is running in the embedded CPU, the access time of the the blackboxed register space in the HSWSIM is very slow. It is about 50 times slower due to the TCP communication overhead on top of the PCIe bus latency. When testing the kernel space FW code, there is only a few register accesses to the blackboxed registers in the HSWSIM, thus the 50x slowdown is not noticeable. In a few occasions, when the kernel space FW testcase runs too slow, we need to restructure the test to eliminate unnecessary access into the blackboxed register space. The PCIe bus in the next generation chip will be upgraded from PCIe Gen2 to Gen3. Once done, it should greatly speed up the register accesses of the HSWSIM.

We have published our pre-silicon achievements in [8]. The design is running at 5MHz in the emulator and Linux boots up in less than 5 minutes. The FW fully brought up all the major silicon features in the emulator. As a bonus, we found three RTL bugs before tape-out. After the silicon came back, we booted up Linux and had the FW running on day 1. After debugging the FW code across the analog, mixed-signal, asynchronous clocks domains, which are not testable in the emulation platform, we brought up the first major feature on day 3. The digital portion of the FW code that is already tested in emulation is essentially bug free. All the major features were up and running by the end of the first week and the SW was released to the customer in less than a month. Thanks to amount of time we spent testing the FW/SW in the emulator before the chip came back, it was the smoothest FW/SW bring up in the company's history for a Rev A device.

REFERENCES

[1] J.Cao, "Pre-silicon Software Development with Protium/Palladium", CDNLive 2017 Silicon Valley
[2] M.Meghana, S.Ramachandran, "Accelerating Time to Market of a Robust Functional Safe Device with Palladium Platform", CDNLive 2017 India
[3] W.Kim, H.Park, S.Choi, S.Kim, "Early Software Development and Verification Methodology Using Hybrid Emulation Platform", DVCON 2017 (Silicon Valley)
[4] F.Thoen, "A pre-Silicon Emulation Platform for Early Software Development of Multi-Mode Modem SoCs", SNUG 2018
[5] D.Bhattacharya, A.Khan, "Hybrid Approach to Testbench and Software Driven Verification on Emulation", DVCON 2018 (Silicon Valley)
[6] B.Cheng, H.Huang, "Improve Firmware Design Schedule by Combining ASIC Simulator and Veloce Emulator Environment", Mentor U2U2018 Silicon Valley
[7] T.Masood, "Multi-level modeling techniques for pre-silicon software development & design verification", Mentor U2U 2018 Silicon Valley
[8] H.Chan "Pre-Silicon SW/FW Testing with Protium S1 – A Case Study", CDNLive 2018 Silicon Valley