Is it a software bug? Is it a hardware bug?

Horace Chan, Mame Maria Mbaye, Sim Ang
Microchip Technology,
8555 Baxter Place,
BC, Canada, V5A 4V7
horace.chan / mamemaria.mbaye / sim.ang @ microchip.com
604-415-6000

***Abstract*-Pre-silicon firmware and software (FW/SW) testing on emulation platforms drastically shorten the time to market by shifting left the project schedules. As a result, the FW/SW is integrated with the RTL in a much earlier stage when the RTL code is less stable. When a test failure occurs, sometimes it is very hard to determine whether the issue is originated in the software code or the hardware code. In this paper, we will present our strategy on how to triage the root cause of the bug, provides tips and tricks on the implementation of the software/hardware co-verification utilities used in our process. We will also include some pointers on how to reuse the emulation debug flow for post-silicon debug.**

I. Introduction

According to the annual verification trends survey in DVCON [1], emulation has become one of the fastest growing sectors in the EDA industry. Over 40% of the design projects perform pre-silicon software and hardware co-verification to allow shifting left the development schedule of the software and identify software/hardware integration issues before tapeout to avoid costly bug fix in the silicon. The emulation team and the software team have to work with pre-tapeout RTL code that is less stable and not yet have sufficient verification coverage. Sometimes the verification teams and emulation teams consume the new RTL release almost at the same time. Emulation and software testing might even run into RTL bugs before the verification team.

In addition to the bugs that could have been caught by the verification team, running the real software on the hardware stress test the RTL in a near production environment. It uncovers corner case bugs that take too long to run in simulation. It also uncovers software and hardware integration bugs that the verification testbench is unable to model the stimulus accurately. In one of our projects, the emulation test caught a FIFO overflow RTL bug related to consecutive counter rollover condition. Counter rollover events take too long to simulate, so the verification testbench usually deposit the counter value to fake a one off the rollover event, hence did not cover this error case. In another project, the emulation test caught a wiring bug creating a memory aliasing in an obscure piece of RAM. The real software can swap test the full address space in the emulation platform running at one tenth of the silicon clock rate, where the same test would take a few hundred years to run in simulation.

When the software testcases fail in the emulation platform, after the software developers do their due diligence to debug the software code but still could not find the problem, there is a reasonable doubt to suspect the problem may be in the RTL. This type of problem that cross the software and hardware boundary is the hardest to debug, since very often the software team and the hardware team live in their respective isolated world, each have their different design and debug flow. It is the job of the emulation team who understands both the inner operation of the software and complexity of the hardware to find the root cause of the problem, acts as the referee to assign the bug to the software team or the hardware team to follow up with the fixes. An emulation engineer is a full stack expert who knows both the software and the RTL equally well, a polyglot who speaks in both C/C++ and SystemVerilog,

There are many debugging tools and methodology from both the software and hardware domains at the emulation team's deposal to investigate the problem. The main advantage of debugging in emulation platform is the test runs in a controlled and reproducible environment. The emulation platform is a digital representation of hardware, the emulation engineer has full visibility into the executing software and the internal states of the hardware. Once a failed test is identified, it can be run as many times as it takes to reproduce the failed result again and again. Each run narrows down the observable point of failure until the root cause is pinpointed. Waveform and SystemVerilog assertions don't lie. At the end of day, if we spend enough time investigating the problem, we will have a conclusive answer on whether it is a software bug, a hardware bug, or a new "feature".

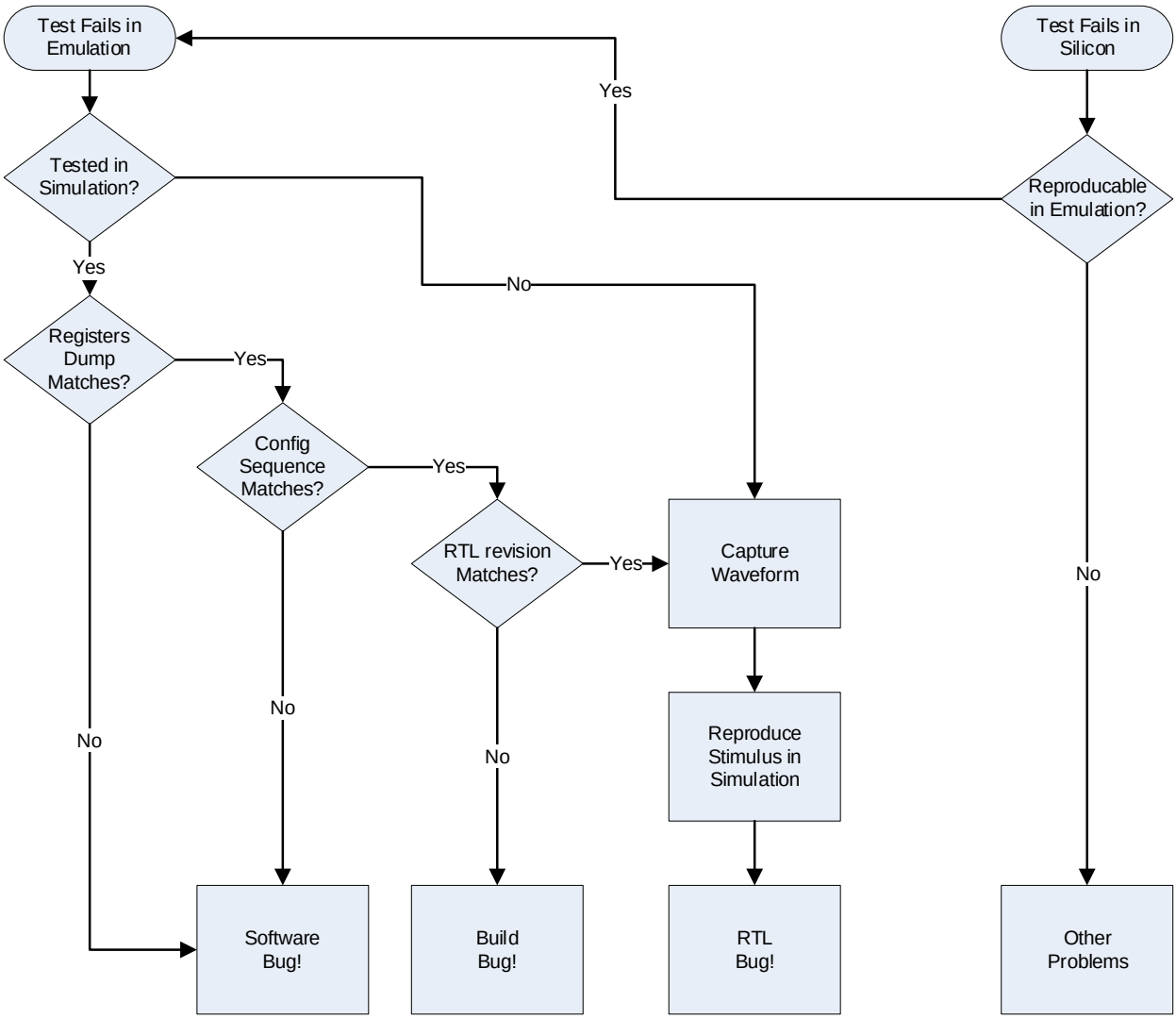

Figure 1. Software/Hardware co-debug decision tree

II. IS IT A SOFTWARE BUG?

Although we use the generic term bugs in this paper, most of time the problem occurs due to lack of communication between the software and the hardware team [2]. The hardware/software interface (HSI) is often subject to changes in the middle of the development cycle. When the hardware team is working under pressure to deliver the RTL to meet the tapeout schedule, documentation becomes an afterthought. The verification team who are familiar with the RTL source code and works closely with the hardware team can keep up to date with the changes, but the software team is left behind with an out-of-date document. The verification testbenches and testcases becomes the golden reference on how to configure and operate the hardware. When the software team and the hardware team have a disagreement on what the HSI should be, it is usually the system architect who has the final say on whether the software or the hardware or both should change. The role of the emulation team is to present the evidence if there is a disagreement in how the software and hardware should implement the HSI.

When the emulation team encounter a software test failure, the first thing to do is to narrow down the point of failure and reproduce the same error signature with a smaller testcase that is easier to debug. The emulation engineer should cross reference the emulation test plan and verification test plan to determine whether the feature is already covered in simulation. If it is an emulation only stress test, then it is very likely the test exercises a new scenario that is not covered by simulation, thus it could be a new RTL bug. If the test scenario is already covered by

simulation and the verification team does not report any failure, the next step is to check what is the difference between the simulation testbench and the software testcase. Very often the problem is an undocumented magic bit that the software team is not aware of. It is very time consuming and tedious chore to compare the configuration settings manually and it is prone to human error and misses the difference. Therefore, we developed some tool to help us automated the debug process. Although the tools take care of the mechanical work of finding the mismatch. The emulation engineer must isolate the error scenario into a simplified software testcase targeting the RTL block in question, and work with the verification engineer of the said RTL block to come up with an equivalent verification testcase.

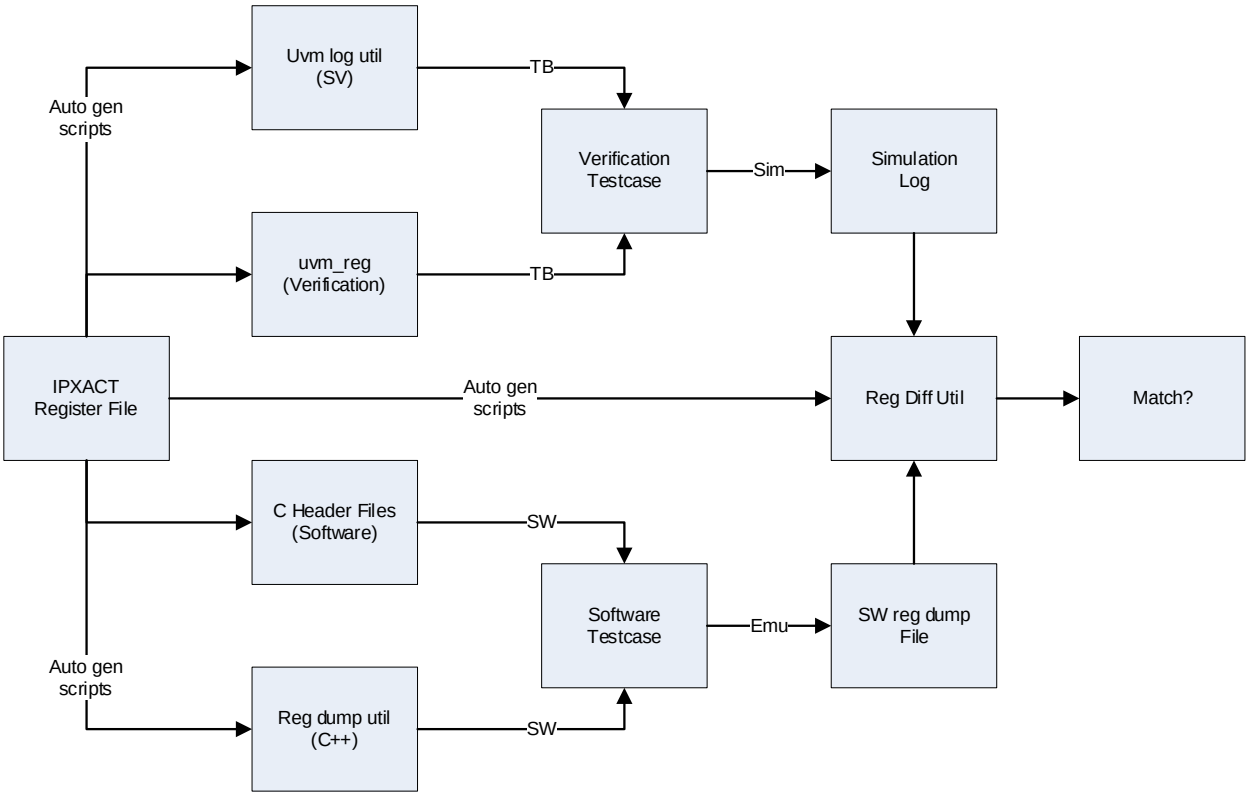

Figure 2. Compare registers in simulation vs emulation

The register address space of a modern day SoC devices is often constructed by aggregating the IPXACT register definition file (or an equivalent register data format) of the individual RTL blocks. The software C header files that define the register address offsets are generated directly from the device level IPXACT register map. The software code accesses the registers through the header files offset directly or through a host adaptation layer API. If it is the former case, we will need to generate a register dump utility from the IPXACT register map to read the register value from the device after it is configured. The utility can select what address range or which RTL block to dump. If it is the later case, the host adaptation layer API can print all the register accesses from the software to a log file, then we use a post processing script to filter out the registers in the specified RTL block. In the verification testbench side, the uvm_reg block is usually generated from the IPXACT register file. However, UVM does not come with a standard way to log the uvm_reg accesses. It is important that all the testbenches adapts a common uvm_reg log library to standardize the output format of the register accesses in the simulation log file. The final step to find the register value mismatch is through a simple text processing script to compare the register value in the software register dump file and the simulation log file.

Sometimes, even when the register values are the same in both the simulation testbench and the emulation testcase, the order of register access may make a difference to the behavior of the hardware. One approach to tackle this problem is to compare the register accesses sequence between the simulation and emulation log files. However due to the timing difference in the execution and the order of execution, an automated comparison is not always feasible.

The emulation engineer may have to spend lots of time to going through the difference manually. We have developed technics to run the SystemVerilog UVM configuration sequence directly on the emulation platform [3] and configure device using the software in the simulation testbench [4] to cross reference the difference in the register operations. The host adaptation layer (HAL) API in the software can be programmed to filter out register access to the RTL block under investigation. A striped down version of the SystemVerilog testbench is connected to the HAL via Linux port and translate the AXI transactions from the testbench into memory read/write to the emulation platform. A modified software testcase will have to co-ordinate the configuration knobs of the SV UVM config sequence to match the software setup and call the SV UVM config sequence in lieu of software configuration function. If the device supports idempotent operations, that is the same set of configurations can overwrite itself without any side effects, we can skip the complex HAL co-ordinations, simply run the SV UVM config sequence after the software has setup the device and observe the difference. We can also use same address filter mechanism in reverse to configure the DUT in simulation using the software testcase. However, the software usually has lots of unnecessary register accesses and compare to the testbench that has omniscient view of the DUT, as a result the simulation runs much slower than usual.

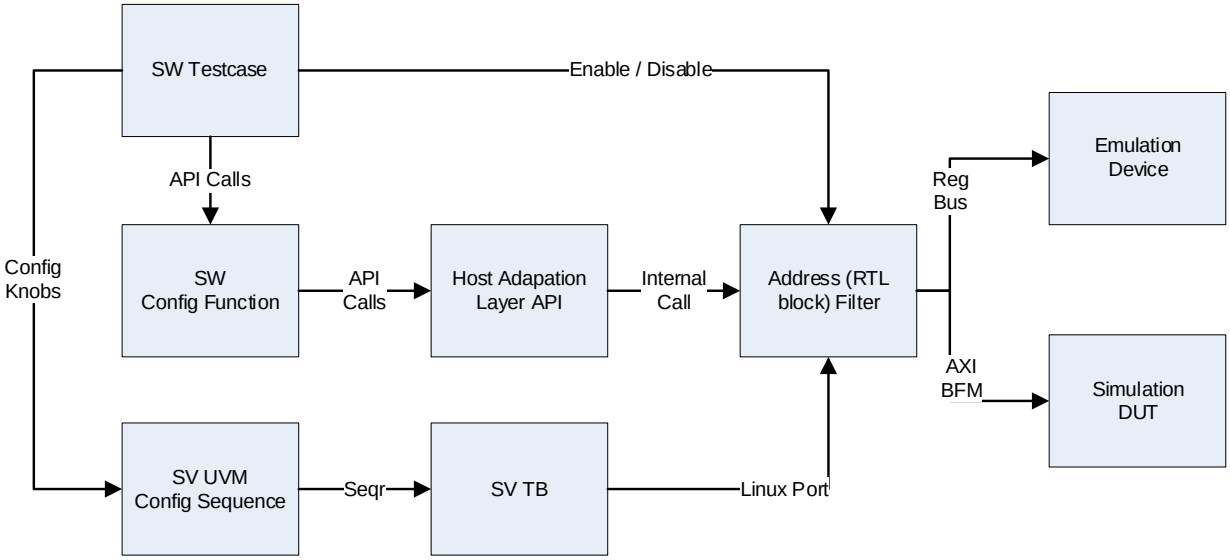

Figure 3. Mix and match SW config function and SV UVM config sequence

III. CAN WE TRUST THE EMULATION BUILD?

Other than software bug or hardware bug, sometimes the bugs originated from the emulation build. One of the most common mistakes is the emulation team picks up an incompatible RTL version when put together the components to compile the top-level design. Ideally, the verification team will qualify the RTL releases first, then deliver a complete device level package with all the RTL blocks to the emulation team for further emulation testing. Unfortunately, due to schedule pressure, the verification and emulation team often consumes new RTL releases at the same time and there is not much co-ordination between the two teams. The software and hardware team may have different release management systems and they may not even use the same revision control software. As a result, the emulation environment may have incompatible RTL releases which may cause software testcase failures.

Manually check the RTL releases against the release note to make sure the verification testbench, the emulation environment and the software header files are aligned with each other is very tedious, time consumption and prone to human error. We have developed an automated solution to safeguard against this type of error. The IPXACT register definition of each RTL component has a read only revision id that is updated automatically with the release information when the RTL code is checked in to the revision control system. The release id is embedded in the design instead of an external piece of information floating around in the repository which often overlooked by the emulation engineers. We implemented the revision id in RTL using the built-in keyword substitution in subversion and using the smudge and clean filters to mimic the keyword substitution in git. Please follow the example in [5] on

how to setup the git filters. It only supports C/C++ file extensions by default, we have to add the .sv and .v extensions to the .gitattribute file.

```
// svn example
`define KEYSUB_SVN(K) rev_svn(K);
function int unsigned rev_svn(string r);
    string r2;
    r2 = r.substr(11, r.len() - 3);
    return r2.atoi();
endfunction

rev_id = `KEYSUB_SVN("$Revision: 1234 $")

// git example
`define KEYSUB_GIT(K) rev_git(K);
function int unsigned rev_git(string r);
    string r2;
    r2 = r.substr(11, r.len() - 3);
    return r2.atohex();
endfunction

rev_id = `KEYSUB_GIT("$Revision: e12a8ea124f2cb8760673ea6e609200ef3ae34f8 $")
```

The svn revision id is an incrementing integer number updated on every check into the svn server. We use the git hash value as the revision id in git, which is a unique value on each commit. The auto generated register diff utility from previous section will automatically compare the revision id of the verification log and the emulation log. The software has built-in debug utility to compare the revision id read from the device against the auto-generated expected revision id from the IPXACT register map release. The RTL release check is fully automated, and any error is flagged to the emulation team preventing us from wasting debug cycle in the wild goose chase of such a mundane problem.

IV. IS IT A HARDWARE BUG?

After we rules out all the possibilities of software or build problems, it is very likely the problem lies within the RTL code. It is time to dump some waveform and debug the RTL like a typical verification engineer. Armed with the undisputed truth of the waveform and SystemVerilog assertions, we can enlist the help of the designers and verifiers of the RTL block in question. It will be the matter of time before we can get to the bottom of the problem.

Many emulation engineers in our team have verification background and we are familiar with investigating RTL issues using waveform and SystemVerilog assertion. Although the debug techniques of emulation are very similar to day-to-day work of the verifiers, but there are differences in the use model thanks to run speed of the emulation is much faster than simulation. When we run a testcase in verification, we usually need to wait a few hours to sometimes a couple of days to see the simulation result and the corresponding waveform. We learned to be careful with what is included in the test and tends to plan on what signals we need to balance the scope of the waveform and the simulation speed. The emulation platform can dump a lot more waveforms much faster compare to the verification testbench thus it has a different use model than the simulation testbench.

There are two types of emulation platform, the FPGA based (Protium) and custom processor based (Palladium). The FPGA platform runs 10x ~ 50x faster than the custom processor platform and it costs less than half the price. It takes 1-2 days to compile a new build targeting the FPGA platform compares to half an hour targeting the custom processor platform. Adding new signal probes to the FPGA platform requires a recompile, while signal probes can add at run time for the custom processor platform. Although the maximum number of signals the system can probe is very large, several thousand signals per FPGA, in the FPGA platform, there is still a hard limited. On the other hand, the custom processor platform can probe every single signal in the design.

The emulation team usually have more resources of the FPGA platforms and it is used to run the daily software testing. The FPGA build should always be compiled with some signal probes provides more debug insight into the internal state of the RTL. There is no need to probe the registers accessible by the software, since that information can easily obtain by register reads. The probed signals should focus on the internal states of the device which is not observable by the software, such as the FIFO fill level, back pressure tokens, packet size, the arrival and departure time of the packets, etc. The emulation engineer also has to choose what signals conditions will trigger the waveform capture. We recommend setting the waveform trigger on the interrupt ports and other error event signals of the RTL block. In general, we are not interested in the waveform of a passing testcase where RTL runs without any problem. We only need to examine the waveform when something is wrong in the RTL.

The custom processor platform is usually reserved for initial bring up and debug complicate problems. On top of supporting waveform dumping, the platform also supports SystemVerilog assertion. The emulation build can compile with the SV assertions from the verification testbench to catch invalid or illegal conditions close to the source of the problem in each RTL block. The platform supports two waveform capture mode, full-vision mode which captures all signals in the device and dynamic probes mode which only captures the specified signals. The platform has a large amount of memory used as a circular buffer to store the waveform. If the buffer is full, it be configured to flush the captured waveform to the disk or simply discard the earlier values. Depending on the investigate stage, the emulation engineer can trade breadth for depth or sacrifice emulation speed to continuously flushing the captured waveform to the disk.

Due to the high price of the emulation platform, it is a limited resource in the organization, and it must be used efficiently. Running regression tests in batch mode maximize the system utilization and processing cycle, but it is not suitable for debug investigation. Interactive debug to explore different configuration settings and traffic stimulus provide most value in understanding the complicate interaction between the software and hardware. We have adopted the following development guideline in our software testcase to shorten the debug turnaround time. The device compiled into the emulation platform must support running a new testcases without restarting the emulation tool and redownload the build. The initial brings up time of the emulation tool sometimes is longer than the run time of the testcase. When trying out different configuration settings, we try to avoid make changes to the code and recompile the testcase. Each recompile does not take very long, maybe just 30 seconds to a minute, but don't underestimate the time wasted on recompile, it really adds up. It is better to use command line arguments to reconfigure the testcase without recompiling the code. It is even better if the testcase supports an interactive prompt letting the user to change the settings in run time, pause the testcase at a specific test stage, loop over the error condition on demand. It may take more time to develop the testcase with an interactive debug prompt, but it will save more time and effort in a long run. The key is to isolate the error stimulus on the emulation platform as soon as possible and pass it on to the verification team to reproduce the error condition in simulation testbench.

V. POST SILICON DEBUG

After the chip comes back from the fab and the software is running on the real silicon, very often the emulation environment is no longer maintained as the engineers have moved on to work on post-silicon tasks. The software testcases slowly add new features that cannot be run on emulation platform without substantial re-work. One day, a corner-case bug hits the fan, and the emulation team is called into action to triage whether it is a software or hardware issue once again.

It is important to keep the silicon software testcase backward compatible with emulation environment. Due to the size limitation of the emulation platform, it may not contain the full device. For example, the silicon supports 8 slices of the RTL blocks, but only 1 instance is included in the emulation setup. The post silicon testcase will use the full device, but it should be easily configurable to run on the emulation setup with reduced capacity. It could be as simple as making the slice number configurable in the testcase. The emulation platform often does not have the analog or mixed signal blocks like in the silicon. Some emulation environments would choose to include behavior models of those blocks, but we simply black box those blocks. As result any access to those address space would hang internal AXI bus. In pre-silicon testing, the software has not included the code related to those blocks. The post-silicon software should have a kill switch to disable accessing to the those address space, so it can still run on the emulation platform. The silicon is 10 times faster than the emulation platform. Any time out delays in the software testcase should be configurable so that the test will not run forever in the emulation platform. The goal is to run the same software testcase binary on both the silicon and emulation platform without recompile.

If the problem is reproduced in the emulation, it is a digital problem, then we can just follow the steps in previous sections to root cause the problem. If the problem is not reproducible, it is very likely an analog, timing, or yield problems which it is outside the scope of this paper.

VI. Conclusion

In an ideal world where the verifiers catch all the bugs in simulation and the designers always keep the document up to date, the emulation teams do not have to worry much about debugging. Unfortunately, the emulation team always have to deal with the most difficult bugs that fall through the crack. In this paper we presented a workflow to identify the origin of the bug. We also presented the automated tools we developed to auto debug some common issues seen in software/hardware co-verification, such as an undocumented magic bit, out of date document of the configuration sequence, incompatible RTL releases in compiling the emulation build. At last, we provided some rules of thumb guideline on how to probe waveform in the emulation platform and how to make the software testcase emulation debug friendly.

With the help of the automated debug tools, we were able to cut the debug turnaround time from half a day to half an hour. In our first emulation project, we neglected the emulation infrastructure after tape-out, as a result it took us 3 days to clean up the software testing environment in order reproduce a post silicon issue in emulation. The same three days is way more valuable after the silicon comes back and the customer is waiting on the first deliverable than three days during the relatively quiet period between tape-out and the parts come back. We keep the emulation infrastructure up to date in the following two projects, we were able to reproduce silicon issues in less than a day.

It is hard to quantify the amount of time saved with the debug techniques presented in the paper, since debugging is more of an art than a science. Some of the content may even sounds like common sense to experienced emulation engineers. We hope this paper lays down the foundation to build a systemic approach to software/hardware co-debug in emulation.

References

[1] H. Foster, “Trends in Functional Verification”, DVCON 2021
[2] J. Rensch, “What Your Software Team Would Like the RTL Team to Know”, DVCON 2020
[3] H. Chan, B. Watt, “How to test the whole firmware/software when the RTL can’t fit the emulator”, DVCON 2019
[4] H. Chan, B. Vandegriend, “Hardware/Software co-verification using Speman and SystemC with TLM ports”, DVCON 2012
[5] https://github.com/turon/git-rcs-keywords