Successful Application of System Level Emulation for Server CPU Validation

George Powley

Intel Corporation
Hudson, MA, USA

www.intel.com

ABSTRACT

Over the last decade, emulation has become a mainstream methodology for chip validation. In this paper, we present a successful methodology deployed to validate a large multi-core server CPU using system level emulation. An emulation model maps RTL onto a hardware platform and provides simulation speeds that are several orders of magnitude greater than simulation. We discuss how emulation model development was divided into a manageable set of tasks; how emulation was useful in pre-Si bug hunting – in some cases, before simulation based verification; how emulation filled a gap in the validation plan that could not be covered by simulation and helped find critical bugs that, would have escaped to silicon; and how emulation features were utilized to improve overall efficiency. We present an effective debug methodology using emulation and conclude the paper with areas of improvement and future trends.

Table of Contents

1. Introduction 3
2. Server Emulation Challenges 5
3. Delivering a System Level Emulation Model 6
   - Static RTL Checks 6
   - Nightly Emulation Model Regression 6
   - Staging Plan for Emulation Builds 7
   - Constantly Evolving Regression Suite 7
   - Nimble Crisis Management 7
   - Flow Automation 7
   - Job Queuing 8
4. Emulation Usage Models and Debug Methodology 8
   - Choosing the Right Emulation Mode 9
   - Optimizing for Capacity 9
   - Optimizing for Performance 10
   - Efficient Debug 11
   - Enabling Self Checking Mechanism 11
5. Results 11
6. Conclusions 12
7. Acknowledgements 12
8. References 12

Table of Figures

Figure 1: Intel Xeon Processor high level block diagram [1] 3
Figure 2: “Shift Left” enabled by system level emulation 4

Table of Tables

Table 1: Emulation modes of operation 9
Table 2: Memory implementation options 10

1. Introduction

The size and complexity of multicore server CPUs have pushed us to the limit of what can be achieved in a system level RTL simulation. For example, the currently available Intel Xeon processor includes up to 15 multi-threaded cores, 37.5 MB of last-level cache, two memory controllers, four memory channels, and multiple QuickPath Interconnect and PCIe links [1]. The CPU contains complex Power Management (PM) features, which require long flows of interaction between hardware and firmware. Validating cross products of PM flows and multi-processor operations, like interrupts and locks, require simulations of hundreds of millions of cycles. At RTL simulation speeds, each test would require weeks of simulation time. System level emulation promises execution speeds that are several orders of magnitude faster than RTL simulation, reducing test time from weeks to minutes. Emulation achieves this speedup by mapping the RTL onto a hardware platform implemented with FPGAs. Clearly, system level emulation is required to enable efficient validation of RTL in the pre-Si timeframe.

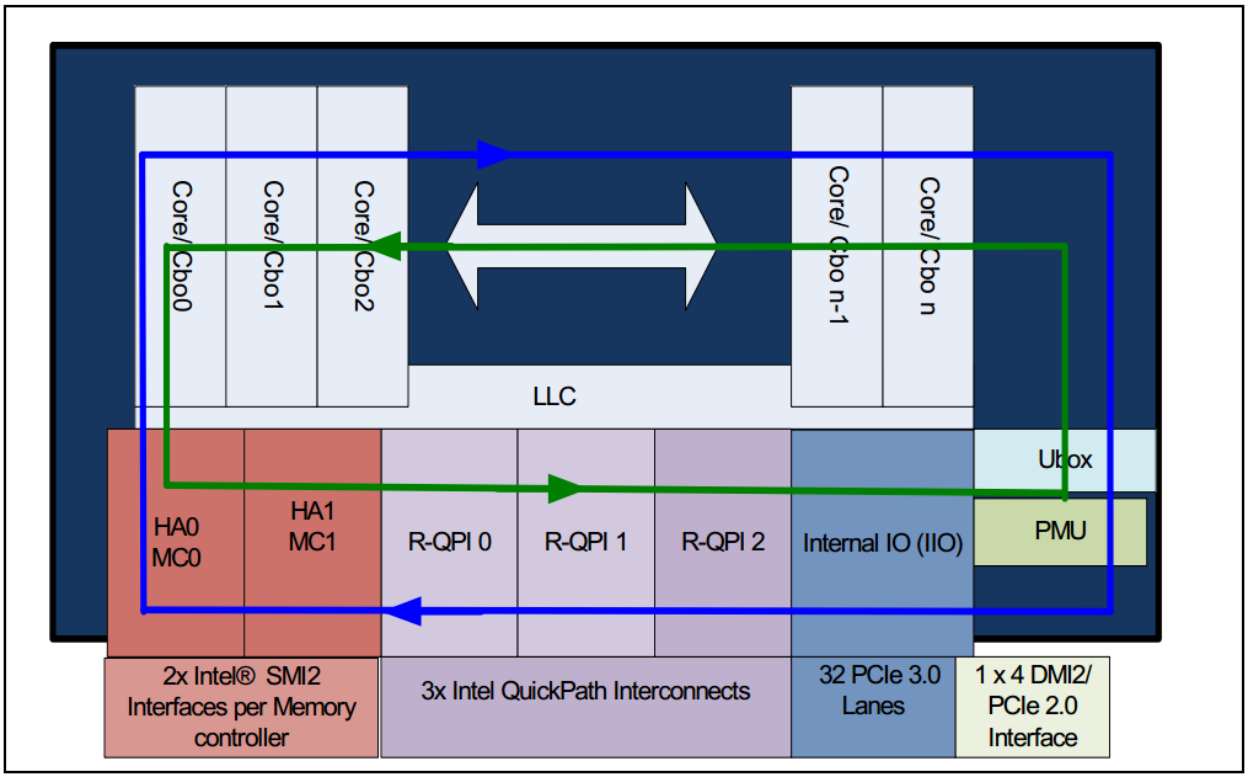

Figure 1: Intel Xeon Processor high level block diagram [1]

In addition to improving the quality of RTL, system level emulation can reduce product time-to-market by reducing the amount of time spent in post-Si validation. A “shift left” approach moves work typically executed in post-Si to the pre-Si timeframe. This work can range from post-Si tool enabling to high volume manufacturing test pattern checkout. By providing a system level emulation model to post-Si engineers before the design tapes out, the post-Si team can resolve bring up issues before silicon arrives in the lab. In addition, enabling post-Si content on emulation can expose RTL and firmware bugs that other validation content miss.

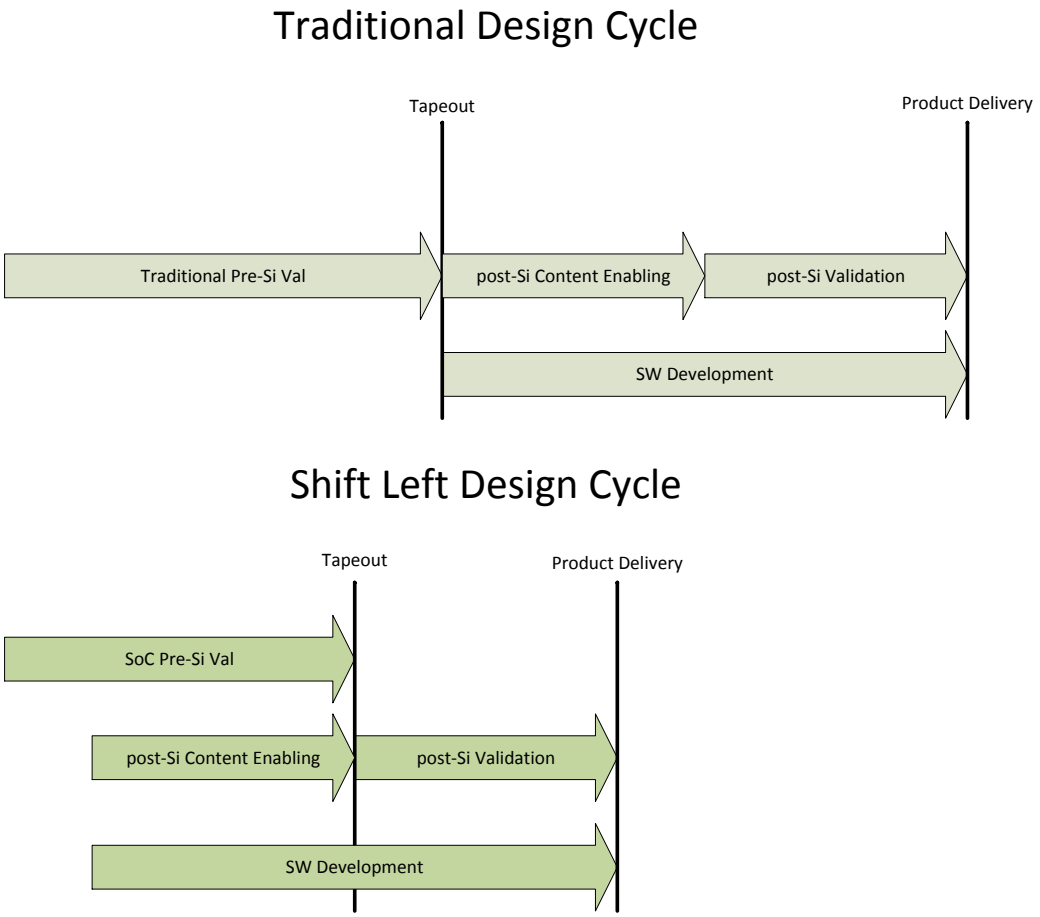

Figure 2: "Shift Left" enabled by system level emulation

Finally, another challenge in bringing a CPU to market is firmware and software development. In many cases, the development of firmware and software are the critical path to bringing a new design to market. System level emulation provides a benefit here as well. Since emulation provides an accurate RTL level model of the system, we can validate firmware and software with a higher level of confidence than using software based virtual prototypes alone. Running firmware and software on a system level emulation model can also expose RTL bugs that may be missed otherwise.

Firmware is essentially low-level software specifically written to interact with hardware and provide a specific function, for example reset or PM flow control. Firmware can be hardcoded into ROMs in silicon or loaded from some form of memory during system initialization. In the case of hardcoded ROMs, it is very important to validate the functionality of firmware in pre-silicon. Loading firmware from memory does provide flexibility to fix bugs in post-Si, however validating the firmware in pre-Si will reduce the time-to-market impact of firmware bugs.

Examples of software required for a server CPU include BIOS, drivers, and user applications. Each of these can be developed and validated in a pre-Si environment. In fact, most software is already validated on software based virtual prototypes before silicon is available. The advantage

of using emulation for HW/SW co-validation is emulation provides a more accurate model of the system under development. Adding emulation to the HW/SW validation toolset provides more validation coverage of HW and SW and reduces overall system time-to-market.

Implementing shift left for firmware and software development does require planning and attention to detail. Planning is required to line up the availability of required RTL, firmware, and software early enough to build a system level emulation model. If these components do not arrive in early enough, the project will not have time to run cycles on emulation and achieve the benefits promised by emulation. Attention to detail is required because the HW and SW are still in development. HW features important to SW, like register addresses, may be in flux. It is very important to ensure firmware and software versions are compatible with the RTL version.

In summary, there are numerous benefits to applying system level emulation to the design and validation of server CPUs, including:

- Improve the quality of the first silicon by finding post-Si quality RTL bugs before tapeout
- Reduce the cost of bringing a design to market by eliminating additional steppings
- Shorten the time to market through "shift left" of post-Si activities
- Improve system quality through HW/SW Co-Validation of RTL, software, and firmware

In order to realize all of the benefits of system level emulation, we developed a robust emulation model using the Synopsys ZeBu Emulation platform and delivered it to multiple internal pre-Si and post-Si customers. This paper describes some of the challenges and solutions of applying system level emulation to an Intel Xeon processor.

2. Server Emulation Challenges

There are a number of challenges in delivering a system level emulation model. These challenges must be overcome to realize the benefits described in the previous section. The first issue we encountered is the delivery of required components (RTL, firmware, software, and content) to meet the shift left requirements of emulation. A number of different teams deliver these components, and in the past, the components were delivered closer to tapeout than required to support the goals of emulation. We worked with the different teams to make emulation requirements known and adjusted schedules to support emulation where resources allowed. In cases where schedules could not be adjusted, like firmware or software development, appropriate workarounds (i.e. injections) were put in place to provide the required functionality.

One requirement for delivering an emulation model is having emulation friendly RTL. Emulation friendly RTL covers many different aspects of RTL quality, including:

- Pass the emulation front-end compile without syntax errors. The emulation compiler is stricter than the simulation compiler, so RTL that passes the simulation build is not guaranteed to build in emulation.
- Limit addition of unnecessary capacity to the emulation model. Adding instrumentation code to provide visibility to registers in memory arrays can force the array to be implemented with registers instead of BRAM, which are more efficient.
- Provide synthesizable and functionally correct behavioral models of analog circuits. For example, RTL for simulation can use #delay statements, but emulation does not support

#delay statements. A separate, synthesizable RTL "view" for emulation must provide similar functionality.

Since the RTL comes from many different designers, across different design teams, we needed an automated solution to checking for emulation friendly RTL.

The size of the design was also a challenge for system level emulation of server CPU models. Some of the issues we encountered include:

- Long compile time for large models
- Emulation HW capacity required for large models
- Maintaining emulation model speed on large models

In the next section, we describe how we dealt with these design challenges.

3. Delivering a System Level Emulation Model

Early delivery of the first emulation model is critical because it allows validation teams to start execution and begin the shift left process. As the complexity and design size increases, planning plays a big role in how soon the emulation team can deliver the model to its customers. During this project, we deployed multiple methodologies to enable delivery of the emulation model within 7 days of RTL release for a new design and within 1 day of RTL release for an incremental change.

Static RTL Checks

The RTL design was highly complex, containing thousands of unique Verilog modules, written by multiple design teams. To compile a design of this complexity quickly and successfully for emulation, it is very important to flag the potential issues to the designers through static checks during RTL check-in. Based on prior project experience and findings on the current project, RTL lint rules were developed to flag some coding styles known to be non-emulation friendly. In addition, we used the ZeBu compilation tool flow to provide an additional static check. Synopsys provide hooks to stop the front-end synthesis flow after analysis, elaboration, or synthesis. This capability allowed a trade-off between runtime and check quality. We selected the stop after elaboration flow, since it took less time and compute resources, and allowed us to run the check for each RTL turn-in. By running these checks on every RTL turn-in, it greatly reduced the number of model build errors caused by RTL changes.

Nightly Emulation Model Regression

To keep pace with RTL releases, even when emulation models were not being refreshed, a full emulation model build was done for every official RTL release. After the RTL model was released, we built an emulation model and ran some simple regressions tests. This "nightly" model build and test flow was fully automated. By doing these checks, we ensured that the RTL was emulation ready when it was time to do a production build. Performing a nightly build also helped identify RTL issues missed by the static checks. For example, if certain coding style caused issues in the back-end of the emulation model build, it was identified early and gave us time to work with the emulation field support team and make necessary changes.

Staging Plan for Emulation Builds

Even with the RTL checks in place, quick delivery of emulation model requires an efficient assembly line type of setup. With full understanding of the inputs required at each intermediate step of the build process, it was easy to parallelize the tasks and start a task early enough if it requires more time. For example, behavioral modeling for emulation needs to happen in parallel with understanding debug requirements from customers that require compile time enabling. But, anything which is needed for runtime enabling can wait. This type of planning helped in making sure that all resources were effectively utilized and that all inputs were ready when the requirement came. It also reduced frequent model spins significantly and helped us in achieving the hard targets set for emulation model delivery.

Constantly Evolving Regression Suite

Once the model was compiled, the quality checks were done by running a set of tests that included basic operations like reset, memory access, and design feature testing. This regression suit was constantly enhanced as test content matured and more features and configurations were enabled in the emulation model. If a certain test was prone to fail more often, it was added in regression test suit to build confidence in the model quality.

Nimble Crisis Management

Despite all the planning and preparation, the crisis do happen which require quick thinking, planning, and collaboration. During execution, we faced a major problem where the tests were not functional at the default clock frequency reported by the Zebu compiler. The problem was partly due to tool issues in timing closure and partly due to non-emulation friendly modeling of analog circuitry in the clock path. The behavioural model of such RTL had various kinds of loops: combinational loops, loops between data and control pin of a flop, or loops between D and Q pin through a latch. We worked with Synopsys to enhance the tool for better loop reports after synthesis. This was mapped back to RTL names, analyzed in detail, and fixed with input from the RTL owners. As a proactive fix, use of VCS Congruency mode is recommended for future projects. VCS Congruency mode can help identify HW loops in simulation, where they are easier to identify and fix, and prevent the long debug cycle required to identify and fix HW loops in emulation.

Flow Automation

Flow automation is an essential part of delivering a system level emulation model. In this project, we provided emulation models for multiple unique configurations. Each model required different build configurations. To make the flow more manageable, the setup was divided in three broad classes:

- RTL: This section specified RTL source, defines, and any other RTL specific attributes. The information was provided by the RTL simulation build flow and converted into the format required by the ZeBu build tools.
- Test Environment and Debug: These files were related to transactors, trackers, and various debug options, like signal probes and triggers. Typical refresh varied from few times in project life cycle for transactors, to few times every model release for debug options.

- Build Config: The build config files contain options that guide the synthesis tool. These config files are used to tune the model size and performance, among other things. Typically, these files were only modified in collaboration with Synopsys.

Within each section, there were lot of variations depending on the model type. Emulation tool flow fully supported TCL. This capability was highly leveraged to merge the requirements of various model types into single set of setup files. Finally, we could choose the right model by setting variables to indicate the desired model.

The emulation tools also provided an integrated method to run user specific scripts at various points of the compile process. This further simplified the automation as we could do any pre or post compilation work in a much more seamless fashion.

Job Queuing

The emulator tool flow generally offers a basic queuing mechanism. At Intel, we adapted our internally developed job queuing system to work with emulation. The queuing system supports sharing a pool of emulators between projects, and sharing emulation capacity allocated to a project among different functional team, with configurable priorities for both. The queuing system also tracked the emulator usage to measure emulation capacity utilization. It also provided checks to detect and remove runaway jobs to avoid emulator cycle waste.

4. Emulation Usage Models and Debug Methodology

A number of different internal customers, with a wide range of validation goals, utilized the system level emulation model. Our goal was to deliver a common emulation environment to both pre-Si and post-Si customers, using a limited number of unique model configurations. Some models delivered include:

- 1 socket models with different numbers of cores (capacity vs. coverage trade-off)
- 2 socket model with reduced number of cores
- Models for SKUs with various core counts

Each of these models included fuse settings, a warm reset sequence to bring the model out of reset, and a mini-BIOS program to configure the system. Emulation customers selected the model that matched their requirements, made changes to the configuration if needed, and ran their content on the model.

Common to all emulation use models is the need to debug failures. Failures ranged from the model hanging in reset to data corruption while executing test content. Some of the common areas for debug include:

- Model bring up and test failures due to emulation specific RTL issues
- FSM and firmware hangs during the reset sequence
- Test failures caused by model configuration issues (fuse, firmware, and/or mini-BIOS)
- Test failures due to RTL and/or firmware bugs

Operational and debug efficiency becomes an important factor in emulation because it is an expensive resource. The goal is to extract as much value from emulation cycles as possible. This covers a wide range of possibilities, discussed below. Most of these options come with trade-offs which have also been explained.

Choosing the Right Emulation Mode

Emulation can be accomplished in multiple modes, which are summarized in the table below.

Verification mode	Specification	ZeBu-Server
HDL co-simulation	Commercial HDL simulators	5K-100KHz
Signal level co-simulation	C/C++, SystemC	100K-500KHz
Transactional level co-simulation	C/C++, SystemC	200K-20MHz
Test vectors emulation	Vector files	100K-500KHz
Stand-alone emulation	Synthesizable testbench	2MHz-20MHz
In-Circuit emulation	Connected to target system	2MHz-20MHz

Table 1: Emulation modes of operation

We chose C-cosim mode in place of ICE (In Circuit Emulation) mode because it allowed effective HW usage across various model types and various projects. In ICE setup, emulation speed can be faster, but it requires a dedicated HW testbench connected to the emulator through external cables. With ICE, the emulator is tied down to one type of design and preparing the HW for a different design requires downtime and human intervention. With C-Cosim, the testbench remains in SW and interacts with the emulator via DPI calls. This mode allows emulator to be setup in a pool and shared by multiple projects. The emulation HW is utilized on demand and once the execution finishes, it goes back in the pool and becomes available for next test or design. Synopsys reports there has been a general trend to move to Co-emulation mode because the benefits provided by a cloud like setup overcome any performance limitations.

Optimizing for Capacity

Capacity optimization is another important factor in improving overall efficiency. Fitting more logic in the emulation HW can help maximize the ROI of emulation. This again has a trade-off. To fit a bigger design, FPGA fill density needs to be increased, but compile time also increases. If the FPGA fill rate is increased further, compile predictability suffers. In some cases, increasing capacity could have a negative effect on model performance, so this needs to be carefully balanced. ZeBu emulation tools provide excellent analysis of design transformation at every compile stage. By analyzing the logs carefully, it is easier to identify the capacity bottlenecks and address them with either compile time options or RTL recoding. The tool has built-in capabilities like zFastStatBrowse, which provides a graphical view of the design after synthesis.

Similarly, the tool provides a scripting utility to extract the useful info from logs and present it in a user friendly format after backend compilation.

One main area where we saw significant capacity gain was how ZeBu tools auto-inferred memories in the design. There are multiple FPGA memory components that can be targeted by the compile process.

	Registers	RAMLUT	BRAM	Design Memory
Capacity guidance	<kb	<10kb	<500kb	>500kb
# of instances	200k/FPGA	100k/FPGA	286/FPGA	Module dependent
Size	1b	64b	36Kb	H/W dependent
# of ports		128	128	8
Synchronous	✓	✓	✓	✓
Asynchronous	✓	✓	✓	
Speed	+++	+++	+++	++
Runtime accessibility	Will be visible as individual registers	zMem	zMem	✓

Table 2: Memory implementation options

In our analysis, we found that memories were not always written in FPGA friendly style. ZeBu synthesis can infer the memories automatically in the design, but in some cases the RTL coding style prevented the tool from selecting the best implementation possible for capacity optimization. Information about capacity bottlenecks is easily extracted from the compilation log files. After a bottleneck is identified, ideally, it can be address using compile time options to change the FPGA memory implementation, but in some cases it requires recoding the RTL.

Optimizing for Performance

Performance optimization is another area that can greatly influence the emulation efficiency. There are two aspects of optimizing for performace:

The first part involves improving the compile flow so that the tool is able to generate a higher clock frequency. ZeBu tool flow provides multiple advanced features like custom partitioning,

localized clocks, and the ability to handle false paths and multi-cycle paths. These can help improve the default performance of the model. The tool also has provisions for detailed log file generation, which can help identify performance bottleneck. This information is used to provide specific custom commands to help improve the performance. The tool also has built-in capability to offer capacity vs. performance trade-offs for many features.

The second part involves analysis of runtime performance so the throughput can be improved. Several factors can have an impact on the throughput. Phase relationship between various primary clocks can have a negative impact on how fast emulation can run. ZeBu tools provide a few alternatives, like clock tolerance, to address this problem. Frequent interaction between the emulation HW and software testbench can significantly reduce throughput. ZeBu software uses a proprietary ZEMI-3 implementation to model HW/SW interaction and multiple features, like streaming DPI calls to reduce synchronization points.

Efficient Debug

The ability to debug a design using the same tools as simulation is what sets emulation apart from FPGA prototyping. During the execution of this project, emulator debug features were utilized effectively to debug emulation test failures without the need to duplicate them in a simulator. We followed a debug methodology of using log files to first narrow down the problem area and then, capture a full signal waveform for in-depth debug. For the first level of debug, we added DPI based trackers and checkers on key interfaces. We also used flexible probes to generate waveforms for a large number of signals over millions of cycles, without a significant performance drop. These probes were added on key interfaces of the design. In most cases, the tracker output alone would allow us to narrow in on a section of the design and time window for a test failure. Once the problem area was identified, ZeBu provides a tool called Combination Signal Access (CSA), which requires a dynamic probe based capture of the design. This capture runs at a slower speed, but does not require a model recompilation. The capture generates a signal database, which is post processed using the CSA engine to calculate the values of combinational nodes in the design. The flow generates a Fast Signal Database (FSDB) file with RTL signal names. This FSDB is loaded in Verdi and used by designers for debug just like any other simulation issue.

Enabling Self Checking Mechanism

Self-checking testcases are highly effective in improving emulation usage. It avoids human intervention to validate for pass/fail and it allows the testbench to terminate the job as soon as it has passed. To enable self-checking, a DPI based test end checker was developed. When the test passed condition was met, the test end checker sent a DPI call to the testbench. In the testbench code, it checked if the flag was for test pass or fail and triggered the test exit routine. The testbench also had a timeout feature. If the test did not complete in a pre-specified amount of time, the testbench would stop the test and report a test failure.

5. Results

The application of system level emulation on this project was very successful. We built on the success of previous Intel Core and Xeon Server emulation efforts, and expanded the use of emulation in terms of model configurations, customers, content, and bugs found. In some cases, emulation was the only platform used to validate features in pre-Si, like long running firmware

flows that include interactions across the entire design. Therefore, a number of complex and critical bugs were found in emulation that would otherwise have escaped to silicon.

In addition to finding pre-Si RTL bugs, emulation enabled the shift left of post-Si tool development and HVM test vector preparation. Improving the quality of these items help to reduce the time spent in post-Si, thereby reducing time-to-market.

Finally, the emulation model was successfully applied to post-Si bug reproduction. After identifying the characteristics of a bug seen on a platform, we made an effort to reproduce the bug on the emulation model. The amount of effort required to reproduce the bug varied depending on stimulus required and bug complexity. Reproducing a bug in emulation provided designers with the full signal level visibility of the system, which greatly improved the ability to debug the issue, and also provided an environment to test the bug fix.

6. Conclusions

Emulation is a required capability to enable full-chip pre-Si validation and product shift left. In this paper, we have shown a methodology that achieved the desired benefits of emulation on a server CPU design. We see emulation becoming even more critical in future projects, which means the methodology of delivering an emulation model must become even more robust. Based on the experience in this project, a major recommendation for improving emulation model quality is validating emulation view of analog circuits in simulation using VCS Congruency [2]. Another recommendation is planning up-front to share validation monitors and checkers between emulation and simulation. This will reduce overall project effort in collateral development, improve the quality of validation, and reduce time in debugging emulation failures.

There are a number of emulation capabilities we did not take advantage of, since they were maturing during the lifetime of our project. For example, SystemVerilog Assertions (SVA), UPF based power aware emulation, and saved state based post-run debugging are available today. A future trend is a strong focus on tight integration with industry tools for seamless transition between simulation, emulation, debug, FPGA prototyping, and virtual platforms. This focus will enable the vision of a validation continuum to address the hardware and software validation needs of complex system designs.

7. Acknowledgements

I would like to acknowledge the Intel team who built, delivered, and supported the emulation models. This team included Emma Beckman, Richard Davies, John Domogalla, Mike Kantrowitz, Suvansh Kapur, Janine Mereb, Yura Pyatnychko, Ziad Rawas, Javier Salvatierra, Alex Samachisa, and Zan Yang. I would also like to acknowledge Ribhu Mittal from Synopsys, who provided ZeBu support on the project and also contributed to this paper.

8. References

[1] "Intel Xeon Processor E7 v2 2800/4800/8800 Product Family Datasheet Volume Two", March 2014, https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-e7-v2-datasheet-vol-2.html

[2] Spigelman, Rafi, "Hardware Congruency – Introducing Hardware Semantics for RTL Simulations", SNUG Isreal 2013, http://www.synopsys.com/news/pubs/snug/2013/israel/B2_Spiegelman_paper.pdf