

The Art of Pre-Silicon Validation



The Art of Pre-Silicon Validation WG
Michael S. Bair, Chair

2024 Edition

For internal use only
Intel Confidential



Intel®, Intel386™, Intel486™, Pentium®, Intel® Xeon™, Intel® NetBurst™, MMX™, and Itanium™ are trademarks owned by Intel Corporation.

*Third-party brands are the property of their respective owners.

Copyright © 2002-2024 Intel Corporation

The latest released version of this document is available on the Intel intranet at:

<https://goto/artofval>

Acknowledgements:

This document would not have been possible without the ongoing support and contributions of many people.

Our thanks go to the people who really got this effort off the ground: Matt Kupperman, Michael Bair, Bob Fisch, Kalpana Kothapally, Mike Miller; key content contributors: Mark Savoy, Phil Atkinson, Matt Plavcan; Paul Schwabe, our technical editor; and finally, our managers: Susan Meredith, Blair Milburn and Bob Bentley.

A great debt of gratitude goes out to those Validators who took time to sort out our methodologies and put pen to paper, more or less, and to create the chapters within. The following are the chapter authors of *The Art of Pre-Silicon Validation*:

Leo Al-Aqrabawi	Bob Grim	Eddy Ong	Erik Samuelson
Phil Atkinson	Darrin Hancock	Leslie Ong	Mark Savoy
Michael Bair	Doug Hergatt	Adeboye Oshin	Carl Seger
Neriya Bar-levav	Kalpana Kothapally	Maria Pineda	Mike Smith
Erik Berg	Soon Seng Loh	Matt Plavcan	Garret Staus
Igor Beskorovainy	Matt Kupperman	Kavitha Ramasamy	Mike St. Clair
Gerry Chen	Sailaja Madduri	Erik Reeber	Chun Tan
Matt Chidester	Jean-Philippe Martin	David Rogers	Wei Kang Teh
Bob Fisch	Prakash Math	Stacey Ross	Amber Telfer
Lance Geiger	Mike Miller	Allan Rudwick	Annette Upton
Gautam Ghare		Steven Saar	Celia Wall
Andrew Gibson			Colin Wood

Cover art by Sergey Sokolov; screenshot from the movie *Hackers*, © 1995 MGM/UA.

Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	5/26/2005	First release version.	Michael Bair	Matt Kupperman / DEG Methodology WG
1.1	8/8/2005	Additions of latest chapters and chapter revisions. Chapters now start on Right-Hand Page	Michael Bair	Matt Kupperman / DEG Methodology WG
1.2	1/9/2006	Updates to following chapters: Introduction to the Art of uAV, Reviewing Execution Testplans, Interaction with Post-Si Val; Minor Updates to other chapters.	Michael Bair	Matt Kupperman / DEG Methodology WG
1.3	2/6/2007	Addition of the Art of uAV Methodology Forum chapter. Major overhaul to Execution Test Plan Methodology chapter.	Michael Bair	Matt Kupperman / DEG Methodology WG
1.4	3/18/2010	Addition of chapter: Leading a uAV Team (pre-L2 review).	Michael Bair	
1.5	6/8/2011	Cleanup of out-of-date content, addition of new methods to existing chapters, addition of new chapter "Checkers". New cover too!	Michael Bair	Mike F. Miller and Mark Savoy
2.0	6/15/2012	Complete Morph of Book into "The Art of Pre-Si Validation" Contains new chapters for various Val disciplines, many updates to previous uAV chapters	Michael Bair	CCDO Validation Staff
2.1	8/2016	Updates to most chapters, addition of many chapters, reorganization of sections.	Michael Bair	DDG-OR Val Staff
2.2	2020	Addition of 'Validation Platforms' chapter, update to 'Checkers' chapter	Michael Bair	DDG-OR Val Staff
2.3	12/14/2022	Completion of Validation Platforms chapter; Small fixes	Michael Bair	
2.4	*2023-2024*	*IN PROGRESS* Lots of small chapter updates; new checking, data flow, and security chapters	Michael Bair	

1 Contents

Section 1: Introduction and Theory

1 Introduction to Pre-Silicon Validation	7
2 Validation Disciplines	33
3 The Life of a Project	55
4 Validation Planning	81
5 Leading a Validation Team	115
6 Stimulus	141
7 Checking (In Prog)	167
8 Coverage	205
9 The Validation Mindset	235

Section 2: Global Concepts

10 Validation Platforms.....	249
11 Test Environments	279
12 Data Flow within the Val Environment	299
13 Testplan Writing	311
14 Regressions.....	343
15 Triage	369
16 Debug.....	381
17 The Life of a Bug	407
18 Indicators	423
19 Evil Validation.....	455

Section 3: IP Validation

20 IP Validation.....	465
21 Dynamic Microarchitecture Validation	475
22 Introduction to Formal Verification.....	487
23 Formal Property Verification	499
24 Protocol Formal Verification	525
25 Symbolic Simulation	545
26 FV Theory Fundamentals	565
27 Mixed Signal Validation	587

28 Becoming the Microarchitecture Expert	611
------------------------------------------------	-----

Section 4: DFX Validation

29 Design for Test Validation	631
-------------------------------------	-----

30 Design for Debug Validation	647
--------------------------------------	-----

Section 5: Firmware Validation

31 Microcode Validation	661
-------------------------------	-----

32 Embedded FW Validation	671
---------------------------------	-----

Section 6: Product Validation

33 Integration Validation (In Prog).....	681
------------------------------------------	-----

34 Architecture Validation	691
----------------------------------	-----

35 Feature Architecture Validation	701
------------------------------------------	-----

36 Reset and Power Management Validation	713
------------------------------------------------	-----

37 Performance Validation (TBD).....	729
--------------------------------------	-----

38 Power Performance (PnP) Validation (TBD)	731
---------------------------------------------------	-----

39 Security Validation	733
------------------------------	-----

40 Fullchip Validation	773
------------------------------	-----

41 Becoming the Architecture Expert (TBD).....	789
------------------------------------------------	-----

Section 7: Post A-Step

42 The Post A-Step World	791
--------------------------------	-----

43 Interaction with Post-Silicon Validation	805
---------------------------------------------------	-----

44 Post-Silicon Escapes	823
-------------------------------	-----

45 ECO Validation	837
-------------------------	-----

46 Stepping Validation.....	853
-----------------------------	-----

Section 8: Appendices

47 Glossary	861
-------------------	-----

48 The Art of Val Methodology Forum.....	869
------------------------------------------	-----

The Art of Pre-Si Val: Chapter 1

Introduction to Pre-Silicon Validation

By: Mike F. Miller

1 Abstract

This is an introduction to Pre-Silicon Validation and *The Art of Pre-Silicon Validation*. This chapter gives background information on the importance of Validation, context for the rest of the document, and insight into the fundamental concepts and themes of Validation that are significantly expanded on in the rest of *The Art of Pre-Silicon Validation*.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	05/03/2005	First draft in the new format (previous versions were not in the proper format and have no revisions)	Mike F. Miller	Matthew Kupperman, Michael Bair
1.1	05/19/2005	Changes and additions from L1 reviews	Mike F. Miller	Matthew Kupperman, Michael Bair, Mark Savoy
1.2	08/02/2005	Some changes from the L2 review and some formatting changes.	Mike F. Miller	Bob Bentley
1.3	08/23/2005	Rest of changes from L2 review	Mike F. Miller	Bob Bentley
1.4	09/08/2005	A couple of minor content changes	Mike F. Miller	Michael Bair
1.5	11/17/2005	L3 review – numerous grammar and formatting changes	Mike F. Miller	Paul Schwabe
1.6	6/8/2011	Updating and revising with new examples and areas where new discoveries have been made	Mike F. Miller	Michael Bair
2.0	5/18/2012	Revising text to broaden scope and move some points to later chapters	Mike F. Miller	Michael Bair
2.1	12/29/2015	Minor revisions to text and expanding scope discussion to multiple teams within presi validation	Mike F. Miller	Michael Bair
2.2	2/16/2016	Updating various areas incorporating feedback from discussions sessions with experienced validators	Mike F. Miller	Michael Bair
2.3	6/23/2022	Update to cost models	Mike Tucknott	Michael Bair
2.4	12/16/2022	Updates to a few sections	Michael Bair	Michael Bair
2.5	2/18/2024	New sections on bug rate and bucket hysteresis	Michael Bair	Michael Bair

3 Contents

1 Abstract.....	7
2 Revision History.....	8
3 Contents.....	9
4 Purpose.....	11
5 Background Information.....	11
5.1 Validation and Verification definitions	11
5.2 Scope of discussion.....	11
6 Introduction to Pre-Silicon Validation	12
6.1 Purpose of <i>The Art of Pre-Silicon Validation</i>	12
6.2 Why is Validation important?	12
6.3 Validation Goals.....	13
6.4 When is Validation Done?	14
6.5 Validation Cost Models	15
6.5.1 Hardware change cost model	15
6.5.2 Software change cost model.....	16
6.5.3 Cost of the Validation team.....	16
6.5.4 Cost of detecting and root causing bugs over time	17
6.5.5 The Cost of Proving the Design	18
6.6 What is High Quality Validation?.....	18
6.7 Where do bugs come from?.....	20
6.8 Beyond Pre-Silicon Validation.....	21
6.9 Themes in Validation	22
6.9.1 Validation as a Career	22
6.9.2 The three pillars of Validation	23
6.9.3 Validation is Risk Management.....	25
6.9.4 Bug Rate.....	26
6.9.5 Bucket Hysteresis	27
6.9.6 Ownership and Responsibility.....	28
6.9.7 Closed Loop Feedback.....	28
6.9.8 Continuous, Aggressive Improvement	29
6.9.9 Destructive Engineering.....	30

6.9.10	Expertise.....	30
7	Summary.....	31
8	Future Work.....	31
9	References.....	32

4 Purpose

This chapter introduces the basic concepts in Pre-Silicon Validation and discusses the fundamental themes and concepts that are elaborated on in the other chapters of *The Art of Pre-Silicon Validation*.

5 Background Information

5.1 Validation and Verification definitions

Outside Intel, *validation* and *verification* have very specific meanings¹:

- Validation is that a product solves the right problems to satisfy user needs
 - Often this job is primarily a Silicon/Platform task
- Verification is that a product confirms to requirements and standards
 - Often this job is primarily a Pre-Silicon task

Most people agree that the two are interrelated, and we agree so much that internally we do not use them as separate terms, but use the term validation to refer to both activities. This is important because both are critical to project success.

- If you demonstrate a product meets the specifications, but does not satisfy user needs, no one will buy it.
- If you demonstrate a product meets user needs, but fails to meet specifications, some unknown user or business need will arise and show your product to be deficient and unsaleable.

To ensure that engineers are always thinking about both, we generally do not disambiguate between the two terms. For our purposes, *validation* is the set of activities Validators perform to verify that a product is correct according to reference specifications, compatible with relevant software, and meets the needs of the consumer.

5.2 Scope of discussion

In general, we restrict the discussion on validation to the logic or higher abstraction levels of the design including some of the firmware/software that runs on it. Lower abstraction levels like schematics and specific layout structures are outside the scope. We will explicitly point out exceptions. The focus is primarily on Pre-Silicon Validation as opposed to Post-Silicon Validation. Pre-Silicon Validation works primarily with a simulated or emulated model, where Post-Silicon Validation works on actual platforms and parts.

¹ The IEEE standard 1012 abstract has a good example of external definitions: <https://ieeexplore.ieee.org/document/8055462>

6 Introduction to Pre-Silicon Validation

6.1 Purpose of *The Art of Pre-Silicon Validation*

The primary goal of this document is to convey knowledge about **why** we have particular tasks and methodologies in place. As part of this process, we explain what those tasks and methodologies are. It is critical to understand **why** these methodologies exist and **how** they support the goals of the team and the company.

The Art of Pre-Silicon Validation intends to provide insight to new and experienced Validators. The document is useful for a variety of audiences and tasks: ramping new Validators, exploring methodology directions, communicating methodology to other teams, and reminding ourselves why we do things **this** way and not **another** way.

We do not recommend reading *The Art of Pre-Silicon Validation* all at once. If you are new to Validation, start with [Validation Disciplines](#) overview chapter. Next, read the theory-oriented chapters. Follow that up with chapters relating to your discipline or your current project phase. It is impossible to assert that a particular methodology is best for all situations, so discuss these chapters and concepts with your manager and teammates to gain a broader view and insight into your particular situation.

We have arranged these chapters to provide an overview of general Validation practices first, followed by in-depth chapters specific to each Validation discipline. While the focus is on processor designs, once you understand why we do things, you can adapt your activities to match your project's unique needs. Some activities and concepts span multiple chapters, so you will likely need to read multiple related chapters to get all the information that is most useful to you.

The Art of Pre-Silicon Validation is evolving documentation. If you find errors, omissions, or areas that you have questions or disagreements about, please send any of the authors feedback.

6.2 Why is Validation important?

Validation is important because being wrong has significant penalties. Consider the circuit problem in the CougarPoint chipset in early 2011. The problem was caught very quickly after release, but recalling the parts cost \$700 million and another \$300 million in lost revenue²! The problem was a small circuit reliability problem that only affected a small percentage of users, but recalling was important to protect our brand integrity. As seen in the CougarPoint example, non-catastrophic failings can be expensive. This is not only true of customers who use our parts but also our OEM/ODM³ customers who expect that our parts “just work” and will be easy to incorporate into their platforms and solutions. If our parts have small bugs, are hard to debug or are unreliable, public perception declines, along with the prices our customers are willing to pay us.

²http://newsroom.intel.com/community/intel_newsroom/blog/2011/01/31/intel-identifies-chipset-design-error-implementing-solution

³ OEM/ODM stands for Original Equipment Manufacturer/Original Device Manufacturer. We use this broadly to refer to the companies who incorporate our products into theirs to make end-user products.

6.3 Validation Goals

You can consider goals on a number of different levels. We will start at the corporate level and work our way down to the specific goals for individual Validators. Each goal supports and contributes to the larger goals. If you think you have a goal that does not support the goals, mission, or activities of your larger group, investigate if you have the correct goal⁴.

The goal of Intel is to **increase long-term shareholder value**⁵.

The goal of a Design team is to produce a design that factories (fabs) will build that **maximizes the value of the silicon** processed by the fab.

The contribution of Validation is **to assist the Design team** in producing designs and collateral that will increase the value of our fab output and to **protect shareholder value** against design flaws that might result in financial loss.

Specifically for Pre-Silicon activities, the high-level goal is to **improve and prove the quality of the design**. The “design” is a broad range of considerations, not just the current stepping. Understanding the overall product family including proliferations and subsequent products prevents shortsighted behaviors that undermine long-term value. One must also consider the design in the context of how the customers use the design and how it delivers value to them⁶. There are many specific goals for individual Validators that support this, including:

- Find all, or as many as possible, of the bugs in my assigned area as quickly as possible, or at least before first silicon.
- Find bugs in general, not just in my assigned area.
- Find bugs at the right level (unit vs. IP vs. fullchip vs. platform).
- Understand the design.
 - Provide continuity of knowledge (not just how and why does it work, but why was it designed that way).
 - Quickly analyze and root cause bugs as they appear.
 - Identify ways to make the design better and prevent harmful changes.
- Create high quality collateral (testplans, tests, proofs, checkers, coverage, test environments, APIs) for validating the design, including proliferations and subsequent iterations of the design.
- Continuously improve and increase effectiveness in all activities.
- Enable the Design team by identifying issues and increasing the quality of their work.

⁴ For more details on thinking about missions, goals, and indicators, see Eliyahu M. Goldratt's book *The Goal*.

⁵ From the Business Practice Excellence class that all Intel employees are required to take.

⁶ For example, saving \$1 in a SOC design is not an improvement if it adds \$2 to the cost of the motherboard.

- Give the Design team confidence that when they make mistakes and code bugs, we will find those bugs.
- Work with IP⁷ teams to ensure their IP is validated utilizing the understanding of how the IP will interact with the rest of the design.
- Tapeout something we are proud of.
- Enable derivative teams through flexible design, high quality validation collateral and other support activities.
- Contribute to the architecture of the product.

There are other goals that individuals in Pre-Silicon Validation have. Keep in mind that the individual goals should always support the organization's goals. If your goal or current activity is not supporting the company goal of increasing long-term shareholder value, it probably is not a good goal.

The teams within Validation work together using a wide variety of approaches and disciplines to achieve the goals. While this document focuses on Pre-Silicon Validation, keep in mind there are other groups that work in complementary ways to achieve the goals of the product development team and the company.

6.4 When is Validation Done?

Validation has the unusual position of never being “done” for a non-trivial project. The space of possible inputs is too large to test every possible combination of states and sequence of inputs for even the smallest designs. Even if we could, determining that the behavior is “correct” for each of those cases is incredibly prohibitive without anything but a perfect formal specification of the requirements (which is incredibly rare).

Even if all bugs are removed from a design, showing that they have been removed is difficult, because unlike other engineering activities, we do not know the denominator of the work. For example, the layout team knows that when n of n units are converted to layout, they are done. This is nice since the layout manager can make indicators and predict the completion of the work based on the rate at which the team is completing subtasks. Unfortunately, we do not know how many bugs exist in the model. So while we know how many are removed, we do not know how many bugs remain.

There are good guidelines on how many bugs to expect in a design and reasonably accurate models can predict bug rates over time on a project. Unfortunately, these models are only 5-10 percent accurate. Assuming your design contains 5000 bugs, if your estimate were off by only 5%, that would yield 250 escapes, which is probably unacceptable for post-silicon validation. Therefore, we use other methods to determine completion of validation activities. More information is in section 6.9.3 Validation is Risk Management, the [The Life of a Project](#) chapter and the [Indicators](#) chapter.

⁷ IP stands for “Intellectual Property”, but in our context it refers to a block or cluster delivered by a team other than the team responsible for integrating it into a design that will be fabricated.

6.5 Validation Cost Models

Being cost effective is an important aspect of engineering. Validation is no exception. It is important to understand how our work affects the cost of the products that we produce.

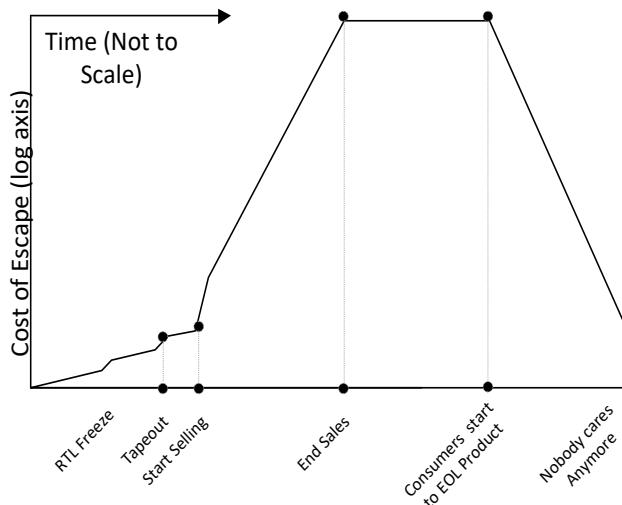
6.5.1 Hardware change cost model

One of the primary costs is the total cost that the company experiences as the result of fixing a bug. As Fred Brooks discusses in *The Mythical Man-Month*, the cost of change (for our discussion, change because of a previously undiscovered bug) increases over time. During the early part of a project, changes do not require much rework (if any). Only the RTL is affected, so the cost is very low. Later in the design cycle, changes to RTL cause schematic change and layout changes, so the costs start rising. When the part is sent to the fab (called tapeout), the cost of changing the design to fix bugs includes the cost of building new masks and manufacturing new parts. Once we start selling parts, a

serious bug may require a recall of parts at a significant expense to the company, an expense that grows with the number of parts sold. Much later in the lifetime of the part, customers start replacing the part with other solutions, so exposure to bugs drops. This can easily be 5-10 years after we started selling the parts. At some point in the distant future, nobody cares if there is a bug, so we do not need to fix it. The diagram above gives a general idea of the magnitude of the cost profile.

Obviously, there is a huge advantage in finding bugs as early as possible in the design process. This reduces the amount of change later in the process, yielding a lower cost of development.

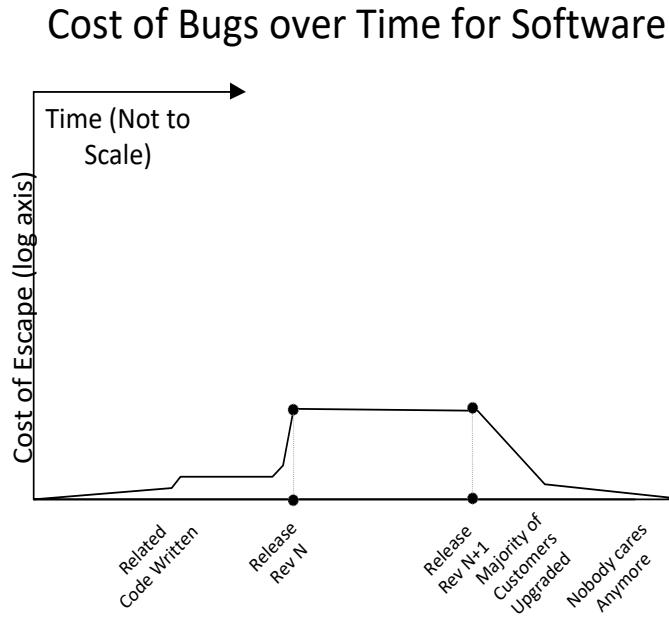
Cost of Bugs over Time for Hardware



6.5.2 Software change cost model

The cost of change for software is significantly different from hardware. This is particularly important as many Validators write software (test generators, checkers, and visualization tools) and need to keep the cost models separate. For several reasons, the cost model for change in software developed internally is flatter than hardware. First, there is no layering of work as exists in the RTL to schematics to layout chain, which can take months to complete. Second, software usually has lower coupling; i.e. changes in one place are less likely to affect other pieces. Third, software is more easily replaceable at low cost to the company. Finally, version N+1 completely replaces version N such that it is rare that version N requires any additional support after version N+1 is released. For many internal products, the costs are negligible for releasing a new version of a tool, making a nearly flat cost model for making changes to the software.

Recent developments in software engineering intentionally flatten the cost model further by doing things like building highly decoupled systems, deploying aggressive automated unit testing and continuous integration and release. This enables them to implement and deploy changes at exceptionally low cost. For more information on software cost models and their impact to development styles, read *Extreme Programming Explained* by Kent Beck or some of the other recent books on agile software development.



6.5.3 Cost of the Validation team

To meet the goal of maximizing the value of the fab output, we must find the right balance of reducing risk with the costs associated with reducing risk. The cost of staffing a Validation team is one of the largest costs associated with reducing risk, so minimizing those costs is important.

The Validation team has two tasks, improving and proving the design. Even if a design was flawless, in order to have confidence to ship the design to customers, we must expend the costs associated with proving the design.

Proving costs include constructing [Validation Platforms](#) and DUTs, and running stimulus, checkers and coverage⁸. These costs tend to be easy to identify, model and track. Improving costs

⁸ See section 6.9.2 The three pillars of Validation for more details on these three.

include debugging failures, filing bugs and constructing debug aids that would be unneeded if the design was bug-free. These costs tend to be very difficult to predict with any significant precision, but previous projects and good engineering judgment can provide good guidelines. In general, increasing the efficiency of improvement tasks is more valuable as the unpredictability leads to risk. For example, spending time to have a checker produce good error messages is very valuable because it reduces debug time later. That debug may occur at a point when latency may be critical to the project timeline.

The efficiency of the team both short and long term should be priority for both managers and individual contributors. Because our business model requires delivering increasingly complex products on a log scale (Moore's Law) with near-constant costs, the efficiency of Validators must increase on a log scale as well. This is a huge challenge that requires a relentless pursuit of efficiency utilizing techniques such as automation, standardization, abstraction and frequently re-examining tradeoffs. This is also where experienced validators can engage with the Architecture and Design teams to prevent bugs as discussed in section 6.9.1.

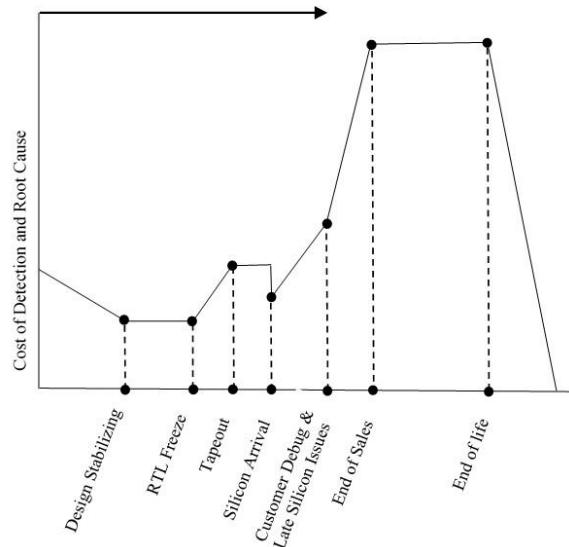
6.5.4 Cost of detecting and root causing bugs over time

From the previous two sections on change cost models, one might draw the conclusion that finding a bug as early as possible is desirable to minimize the cost of design change. However, the three main costs associated with root causing a bug (cost to develop collateral, cost to debug the failure, and cost of system time) change over the life of the project.

Early on, it is imperative to get solid validation collateral (stimulus, checkers, coverage, formal proofs) in place as soon as possible to drive the cost of bug finding down. In the figure it shows that cost starts higher and comes down until the point of design stabilization; anything that Validation can do to speed up this process is valuable.

Mostly, this entails getting quality collateral into place as early as possible. There may be added maintenance cost to adapt to a changing design, for example, an early test environment may need to work around missing or defective design features to reach the area of interest. While this cost is real, it should not dissuade Validation from getting collateral in place as early as possible.

As the project progresses towards tape out, collateral development and maintenance costs come down and the collateral increases in quality. Better collateral results in more efficient debug. The cost of debug remains approximately constant due to an increase in collateral efficiency. System time costs start to increase as higher complexity bugs require more cycles to hit. At some point, all the easy bugs are found, and the validator must develop new collateral causing all three vectors to rise in cost.



A noticeable outcome is that the number of bugs found per week tends to ramp as additional Validators are added, but then becomes a near-constant value and does not drop until tapeout. However, the types of bugs found changes dramatically during this time. Early in the project, the team finds simple, inexpensive to detect bugs. Over time, team finds increasingly complex bugs. The rate of detection is roughly constant because the capabilities of the team to detect complex bugs increases and maintenance costs drop as the design stabilizes.

After silicon arrives, the cost of debug increases due to the loss in visibility. Collateral and system time costs generally go down during this same period (post silicon hits their “easy” bugs). As the project progresses towards Project Release Qualification (PRQ) all vectors start increasing. If the bug is not caught by Intel, the cost increases again as customers might charge Intel to rent their systems or apply penalties to Intel for the customer going Lines Down (Lines Down: When a customer’s factory stops production of a product until an issue is resolved). Once the product reaches End of Life (10 years for some products) the cost trends towards 0 as Intel and the customer agree that providing a root cause and fix does not provide a value.

6.5.5 The Cost of Proving the Design

As Validation transitions from improving the design (mostly bug finding) to proving the design (driving deeper coverage, checking, etc), in some sense the ROI calculation for Validation changes dramatically. Throughout periods where bug finding is high, the output of validation activities seems clear (bugs!) and thus ROI seems straightforward. When bug finding is harder, it may appear that Validation is doing lower-ROI activities, since the return may not be as visible. Hitting deeper coverage conditions, filling checking holes, and lowering project risk is not as tangible when compared to bug finding.

The cost for this phase of validation can be very high and yet may yield few bugs. Without an incoming bug rate project management may think that it is time to put validation effort elsewhere. It is critical that project leadership understands this aspect of validation and that expectations are set early within the project.

6.6 What is High Quality Validation?

You can do any task with different levels of quality. Note that the final quality of the product is different from the quality of validation. It is possible to produce a high quality product without high quality validation due to luck or very good Designers. Given appropriate resources, it is very difficult to produce a poor quality product while having high quality validation. Having a high quality product does not guarantee marketplace success. Timna (a P6 core with integrated memory controller) was a high quality product, but because it used very expensive memory technology, it failed to meet the customer’s requirements for low platform cost and failed to produce revenue for the company.

While we encourage everyone to aspire to a high level of quality, one must take into account resources, requirements and sensitivity to escapes (see section 6.9.3 Validation is Risk Management). Even if it is acceptable to do a lower quality job, it is important to understand and document the tradeoffs. For convenience of discussion, we will split the spectrum of quality into three sections:

- Exercise: running stimulus on the design to find bugs and improve the design. The tests exercise the primary usage models of the product.
- Validation: running stimulus on the design to find bugs, improve the design, and show the absence of bugs in the design using coverage and other evidence. The tests and coverage show that the part operates as required.
- High Quality Validation: is going beyond normal validation to show that the design is robust beyond the current usage models and requirements.

The difference between exercise and validation is quite significant. The difference is between simply finding bugs and using a methodical process to find bugs and show the absence of bugs. Limiting the work to exercise will result in bugs escaping for any area that the Validators did not think of or areas that the tests did not exercise due to incorrect test construction. Exercise is appropriate for some phases of a project (for example, see [The Life of a Project section: IP RTL 0.5 – Validation Activities](#)) but is rarely sufficient for taping out a project or for shipping to customers.

The difference between validation and high quality validation is an expansion of the space that is tested. High quality validation shows that the design is robust under all conditions, even those that currently cannot happen. For example, not all possible sequences of ucode are possible in any given processor. However, high quality validation tests that ucode sequences that do not currently exist on the processor to confirm that they operate correctly. This is particularly important when the block is going to be reused on later projects, or on other projects where a different context can expose latent issues.

Naturally, the costs of doing high quality validation are more expensive than normal validation, which in turn is more expensive than just doing exercise. Initially, it may seem financially unwise to do high quality validation due to the additional expense to test for correct behavior on conditions that will not actually happen in the design. However, there are two factors to consider.

The first factor is that while the product is in the design process, changes to the design may significantly affect what is or is not possible. If you remember the original cost graph, finding these bugs before they become possible has some benefit, as they are less expensive to repair earlier in the project.

The second factor is that it is extremely rare for a project to exist in isolation. Usually there are follow-on projects called proliferations or derivative products that directly use an older project's design, bugs and all, to produce a new product. Proliferations may change process technologies to a smaller feature size, or a different recipe. Both proliferations and derivative products invariably change the design, such as modifying cache sizes or adding features either from scratch or integrating a different set of IPs. These changes modify the design such that previous bugs that were impossible to hit become possible. The cost of fixing a latent bug in the proliferation is higher than in the initial design. The bug and logic must be reexamined in both products. Extra testing must be done on the proliferation in areas that were believed to be working. Finally, Validators on proliferations or derivative products are usually less familiar with the design as they were not involved with the original product, resulting in higher costs to successfully eliminate the defect and demonstrate the correctness of the design.

There is significant benefit to projects doing high quality validation to keep the total costs to Intel minimized. This is particularly true for large flagship designs that are leveraged by smaller

proliferation designs frequently staffed at much lower headcount levels and less able to pay the costs of latent bugs.

6.7 Where do bugs come from?

While bugs have a variety of sources, these sources are not magical or unpreventable. Understanding the sources of bugs is not an academic exercise. Understanding enables you to predict bugs, do substantially better validation with less effort and provides guidance to prevent the creation of bugs.

Data collected after the first Pentium 4 tapeout indicated that bugs came from many sources, but five general areas covered half of the bugs. The remaining 29 categories covering the other half⁹:

1. Goof – Typos, cut and paste errors, careless coding when Designers were in a hurry, and situations where the Designer was counting on tests to find bugs instead of being proactive in thinking about the problem.
2. Miscommunication – Includes Architects not communicating expectations clearly to Designers, misunderstandings between Designers. In this dataset this was particularly common between microcoders and RTL Designers. In today's SoCs, miscommunication between IPs is still common.
3. Microarchitecture – Problems with the microarchitecture definitions provided by the Architecture team or low-level microarchitecture work done by Designers.
4. Logic/Microcode change – When logic or microcode is changed and the Designers did not consider the full impact of the change.
5. Corner Case – The Designer's implementation did not cover some rare or unusual situation.

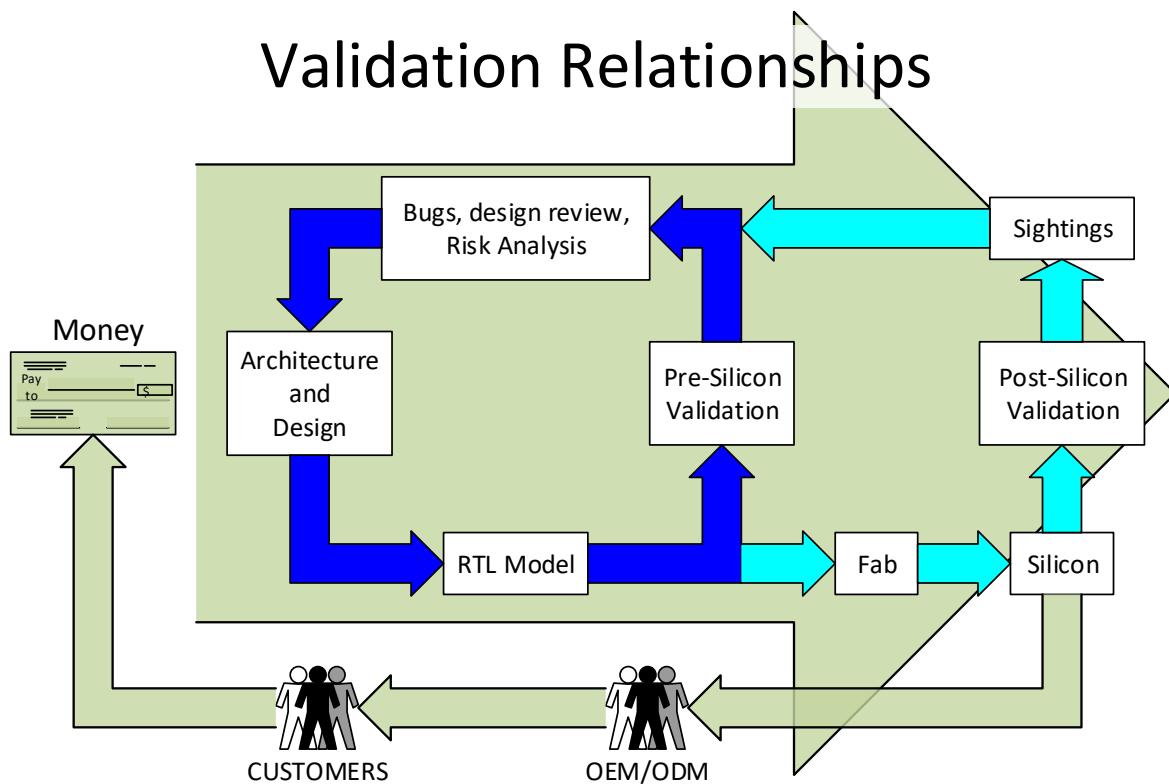
The primary sources of bugs are simple mistakes or misunderstandings. While one might expect these bugs would be easy to discover, this is not the case. For example, on a later Pentium 4 processor we had a silicon escape that was a cut-and-paste style error. When increasing the number of buffers in a unit from 6 to 8, the Designer missed doing the 0-5 to 0-7 change in one place that was rarely activated. The Validators missed that the two buffers were not identical to the other six, so we missed the bug until after it reached silicon. Other bug analysis data from later Pentium 4 projects indicated that the percentage of the “goof” category was even higher than reported on the first Pentium 4 design.

The complexity of the machine clearly has an impact on the number of bugs. The more complex (and large) the design is, the more opportunities exist for goofs, microarchitectural mistakes, miscommunication and unanticipated interactions. Reducing design complexity reduces the number of bugs and reduces validation effort and project risk. That does not necessarily mean less features. Some architecture and microarchitectural choices result in designs that have higher cross-unit coupling, hard to hit conditions or other unnecessary interactions as side-effects. Validators should work with Architecture and Design to balance complexity with other product attributes.

⁹ Contact Mike F. Miller for a copy of the paper

6.8 Beyond Pre-Silicon Validation

Projects are composed of a wide variety of groups all working together to improve and prove various aspects of the design. While this document focuses on Pre-Silicon Validation, it is helpful to know the other groups in your organization and their roles and responsibilities. There are three groups that Pre-Silicon Validators frequently interact with: Architecture, Design, and Post-Silicon Validation. There are also groups with fewer interactions with Validation: the fab, platform software teams, external customers include OEM/ODMs like Dell and IBM, and Joe down the street. Let us take a quick look at the relationships of these groups.



Notice the three circular loops. The inner dark blue loop shows Pre-Silicon Validation giving input and feedback to the Design and Architecture teams and getting models back from them (the Architecture team assists the Design team in developing the design). Observe a similar but longer light blue loop with Design and Post-Silicon Validation teams. Finally, there is the long, light green loop with OEM/ODMs, and end customers, which pays for all our Design, Validation, factories etc.

For Pre-Silicon Validation, the better we do things the first time, the fewer iterations we need to make around the inner dark blue loop, and indirectly around the larger light blue loop. Consider that the light blue loop is a four to six month latency because fabricating a chip and deploying it into labs is a slow and expensive process. The quality of the feedback we give directly affects the quality of the silicon and the product.

There are a number of other customers and interactions between the Pre-Silicon Validation team and other teams which are covered in other chapters.

6.9 Themes in Validation

As you read the rest of *The Art of Pre-Silicon Validation*, you will notice general trends or themes; topics that are revisited in multiple chapters. Some are specific to Validation and some are just good engineering practices. We have attempted to highlight some of these themes in this section. Most of these themes will make more sense when applied to real examples from your projects.

6.9.1 Validation as a Career

Validation is a topic that is rarely taught in engineering schools. In the past when large Validation teams were formed, they were formed almost exclusively of recent college graduates (RCGs) with little or no engineering experience. Because of this, Validation was viewed as an “entry-level” position to be done by inexperienced engineers. This stigma stuck to the whole field of Pre-Silicon Validation. Engineers starting in Validation left Pre-Silicon Validation as they gained experience for jobs that were perceived as higher prestige. These Validators were replaced with more inexperienced, fresh RCGs who needed to be trained. This cycle perpetuated the image of Validation as a “stepping stone” to other engineering activities.

In DDG we have largely eliminated this false perception of the Validation job as being an “entry-level” position. Only experienced Validators are able to do the most critical activities, particularly evaluating risk correctly and being experts in their area. Their experience enables them to drive the kind of aggressive improvement that our industry requires (see section 6.9.8). They can provide feedback to the Architecture and Design teams that junior Validators cannot. We have established a career path that can take a Validator in a technical position all the way to the principal engineer level, providing benefit to the company at the same level as counterparts of the same grade in the Design and Architecture teams.

The ability to have a Validation career path stems from a viewpoint that there are three ways to deal with bugs – prevent, detect, survive. Junior Validators are concentrated on the “detect” portion (hence the perception by some that detection of bugs is all that Validators do). Detecting is the most straightforward of the three (find a bug via a failing test and file it). The other two can only be done well by experienced engineers.

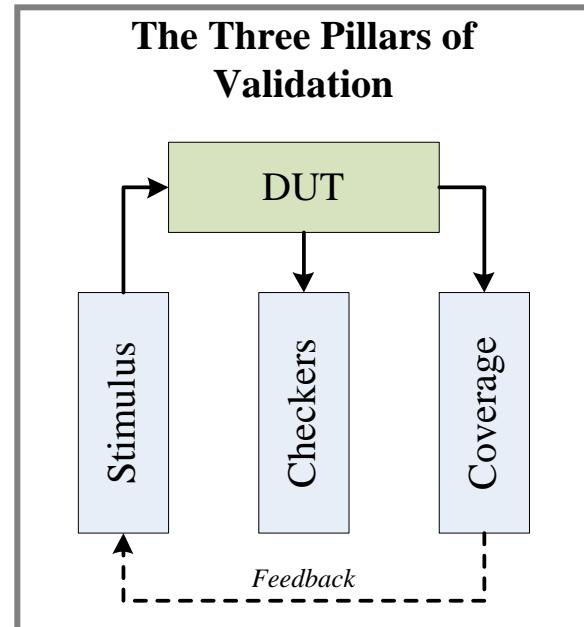
Preventing bugs from entering the design by working with Designers and Architects to avoid unnecessary risk requires experience, a well-rounded view of tradeoffs, and good engineering judgment. As seen in the cost models, preventing a bug is usually the least expensive way of dealing with one.

Surviving a bug implies that the bug has made it into some form of silicon, either in-house engineering samples or potentially products that we are actively selling. Surviving means that we can deal with the bug with minimal impact to the company. For example, experienced Validators work with Designers to make sure that high-risk areas or microarchitectural features have backup “defeature modes,” which are enabled via a firmware patch. While using these defeature modes often trades off performance to gain correctness, it enables the company to “survive” bugs with minimal financial impact or bad press.

It is important to point out that senior Validators are not simply “movie critics” who complain about the bugs in the design. They are proactively involved with the Architects and Designers making suggestions, offering solutions, providing input on risk and costs, and are engaged in the creative process of the design throughout the entire lifecycle of the product. For more information on developing a career path see [Leading a Validation Team section: Developing a technical career path.](#)

6.9.2 The three pillars of Validation

Most of the validation collateral (what the Validation team produces) falls into one of three categories: stimulus, checking and coverage. Each of these may look different for various Validation groups, but all groups have them and each is critical for success. The overall quality of validation is limited by the quality of the lowest of these three areas. Balancing resources for each of these key elements is particularly crucial, and ignoring any one of them will result in bugs escaping to silicon. While these pillars are more obvious in dynamic validation, formal methods also have these three, but stimulus and coverage are much more exhaustive, making them particularly valuable for many situations. See the chapter [Formal Property Verification](#) to understand those methods.



6.9.2.1 Stimulus

Stimulus is frequently referred to as tests or input. Stimulus is the content that you use to exercise the design under test (DUT). Good stimulus will quickly exercise a large amount of functionality in a minimal amount of time. If you do not have good stimulus, you will not exercise important conditions where bugs exist. In dynamic validation, stimuli are tests along with other tools to inject state or transactions into the model. More discussion of how to build good stimulus is in the [Stimulus](#) chapter.

6.9.2.2 Checking

Checking is verifying that the behavior of the DUT matches some reference model or expectation. For example, an instruction architectural checker will verify that the results of an increment-register instruction results in the register value being one more than it was before the execution of the instruction. There are many different checkers that are deployed in a project and include:

- Self-checking tests
- Assertions in the source code

- Forbidden statements in coverage monitors
- Mini-checkers in (and outside) the test environments (TEs)
- Interface checkers
- Protocol checkers
- Architectural checkers

Checking alone is insufficient. If the stimulus does not exercise the DUT where a bug exists, the checkers will not be able to detect the problem. Similarly, even if the stimulus causes the bug to occur, without a checker capable of detecting the failure, the bug will not be found. For more information on checkers and their development, see the [Checking](#) chapter.

6.9.2.3 Coverage

Coverage confirms that the tests exercise the DUT in some particular way. The coverage space is defined in a testplan so it can be tracked and reviewed by customers as evidence of the correctness of the design (see the [Testplan Writing](#) chapter). For simple DUTs and some black box testing, this may be done by ensuring that particular stimulus has been fed into the DUT and the simulation passed the checker. For more complicated designs and white-box testing, it is not reasonable to assume that some stimulus exercises the DUT in a predictable way. For these cases, it is necessary to monitor the DUT to detect and record when the DUT exercises cases of interest. Coverage is used to identify gaps in the stimulus and make changes in the stimulus to close those gaps. Failure to collect and use coverage results as feedback to improve stimulus quality will result in a failure to exercise important cases, allowing bugs to escape. For more information on coverage, start with the [Coverage](#) chapter.

6.9.2.4 Optimizing ROI among the three pillars

There is special relationship among the three pillars. You can achieve similar Validation results using different combinations of effort for the three. For instance, there is a balance between stimulus and checking: there are situations where an increase in checking allows for a decrease in stimulus, and vice-versa. Within the design, errors can occur where the bug never manifests itself in an architecturally visible way, but microarchitecturally can be identified to be incorrect. The stimulus has to work ‘extra hard’ to hit the bug as well as to make sure that the bug manifests itself in a way to visibly cause a failure. If effort is placed on microarchitecture checking that can catch the bug at the point it occurs (deep within the machine), less effort is needed to be placed on ensuring the test can flush the buggy instance out to a visible point. Examples of situations requiring extra effort to see the bug: data corruption could occur down a mis-predicting branch path, memory ordering violations, cache coherency bugs, and cache inclusion violations.

A similar balance occurs between stimulus and coverage. The greater effort you put into making your coverage space ‘perfect’, the less effort you need to put into making sure your stimulus is doing the right thing since coverage will tell you exactly what works and what does not. Conversely, you can spend a lot of time validating your stimulus, or writing stress engines, and opt for less coverage.

The Validation team uses these relationships to optimize the ROI of our stimulus, checking, and coverage efforts.

6.9.3 Validation is Risk Management

Validation, particularly Pre-Silicon Validation, is about risk management. Our job is to reduce the risk of the product not being logically functional. We are not attempting to eliminate risk, which would take forever. We manage risk to an acceptable level. What that level is will vary between products and sometimes changes during a project based on business and customer needs. There are an infinite number of activities that we can perform to find and eliminate bugs. There is a strictly finite amount of time and resources available. Identifying the maximum return on investment (ROI) for activities that mitigate or reduce risk (not necessarily the activities that find the most bugs) is *critical*. The mark of an experienced Validator or manager is that they can correctly identify the highest ROI activities for reducing project risk.

For example: as you read through *The Art of Pre-Silicon Validation*, the writers may advocate doing more things than you have resources to do. Make a priority list of the things you would like to do and draw a ZBB line where you run out of resources. Then, you and your managers need to decide if the risk of not doing the items below the ZBB line is acceptable for your project. Otherwise, more resources or time will need to be added. As your project progresses, feedback from bugs, coverage and changes in the availability of resources will require that you revisit your priorities and make adjustments based on the new information.

Another part of risk management is determining when you have “done enough” so you can tape out or ship a product. There is no single answer for this and you need to understand your customers’ needs and tradeoffs. For tapeout, Post-Silicon Validation is your customer and you need to consider that because Post-Silicon Validation has incredibly higher test throughput (GHz speeds versus Pre-Silicon’s Hz, or possibly kHz) but less visibility, some bugs may actually be lower cost and higher efficiency to detect in the Post-Silicon testing space. For shipping, your customers’ requirements and your market position significantly affect what bugs are acceptable to work around, errata or take “acceptable losses” on. For example, if you are building an embedded processor for a single client, it may be acceptable to tell them “Don’t do instruction X followed immediately by Y.” If your client can avoid this behavior in their code, the bug is harmless and does not affect your revenue. If you are building a part where you or your clients do not control the software, that same bug will prevent you from shipping.

Market position is another factor in how much impact bug escapes have on company revenue. Intel is a very high profile company, and our high average selling price (ASP) is tied to customer’s perception of our brand. Intel’s brand value was valued at \$35 billion dollars in 2021, the seventeenth highest¹⁰. As part of protecting long-term shareholder investment, it is critical that we produce products that meet our customers’ expectations of quality (lack of bugs, compatibility, ease of debug, etc.). This ensures that our brand value, and thus our ASPs and profit margins remain high.

¹⁰ Interbrand rankings: <https://interbrand.com/best-global-brands/>

High ROI activities may not be high-bug activities. On the Pentium 4 client projects, very little validation work was done on the Machine-Check Architecture (MCA) features¹¹, despite the fact that it was generally known that there are many bugs in MCA. Why is this acceptable? Because the cost of being wrong and having a MCA Silicon escape was minimal (see all the MCA-related errata, and lack of a recall) when compared to memory cluster bugs where escapes caused us to make additional steppings in multiple client projects. We focused our resources on low-bug memory cluster areas instead of the high-bug MCA feature, because the risk of financial impact (extra steppings or recalls) was greater in the low-bug memory cluster.

6.9.4 Bug Rate

Since the beginning of time, project managers have wondered *when will the bug rate go down?* The answer is both simple and complex.

The maximum bug rate on a project, measured in bugs per week or even bugs per quarter, is a factor of how much RTL code is changing, how buggy the code is in terms of ‘bugs per changed line of code’, and how fast validation can find it in terms of how much validator time is needed to find and debug one bug and how many validators exist.

All of those factors can change from project to project. There have been projects where individual validators found well over 100 RTL bugs within a year... for multiple years in a row. In such situations the RTL change was high, the complexity was high, and the gating regressions were not good enough to keep the model from accepting newly coded RTL bugs. Contrast this with a project where the validation environment is solid, where the gating regression is robust, and where the RTL code change is limited to local feature changes... in such a situation, the bug rate would be much lower.

One of the constraints for maximum bug rate is the size of the validation team. In theory, during maximum bug finding times of the project, simply adding more validators could raise the bug rate. However, that might not be useful if the design team cannot keep pace with fixing all of the incoming bugs.

When does the bug rate go down? Three main factors (and probably lots of minor factors) determines when bug rates fall.

First: When RTL stops changing. When RTL stops being modified (with the exception of bug fixes), the bug rate will take its first major drop. This is because new bugs are introduced with RTL change; when RTL stops changing, there are no new bugs being introduced. If RTL coding ramps down slowly, the drop in the bug rate will happen slowly. If RTL coding stops all at once, the bug rate drop will happen just as dramatically. Note: this drop might be masked if your validation team is debug-limited; meaning, they were spending all their time debugging ‘newly

¹¹ MCA is a feature where the processor tracks reliability information such as ECC errors and makes this information available to the OS. In cases where the design fails to correct an error, MCA will cause the CPU to go to an OS handler to attempt to recover, or notify the user that data corruption or some other unrecoverable error occurred. For more details, consult the IA-32 Intel Architecture Programmer’s Reference Manual

introduced' bugs... and immediately shift all of their resources to debugging in the next two classes of bugs!

Second: When new feature enabling is completed. The majority of bugs come from feature enabling. When validation is still spending much of their time simply getting microarchitectural and architectural features working, expect a significant number of bugs from this effort. When validators complete their feature enabling, the bug rate will take its second major drop.

Third: When validators run out of things to do, as measured by the number of changes they are turning into the model repo. During the execution phase of a project, validators are turning in new tests, sequences, checkers, stressors, configuration, constraints, and anything else they can think of that will find more bugs. Some of this is because the testplan called for it. Some is because a coverage hole revealed a gap. Some may be because a bug escaped to a different team, and this team is adding more content to find other bugs in the area. Sometimes the validator just thinks of a new thing to do. When all of these stop happening, the bug rate takes its final big drop. This moment can be considered the true end of validation execution.

At that point though, the bug rate is not likely at zero. What happens next is called the *bug tail*. The bug tail is a bug rate that continues after validation is done executing, where bugs are being found due to failures occurring due to the continued running of tests using the *existing content*. In other words, the bug tail exists simply because you are letting random cycles find deeper and deeper bugs. Things can be done to speed up the bug tail (genetic and AI algorithms to make existing content find deeper bugs sooner, for example), but it often comes down to just time and cycles.

6.9.5 Bucket Hysteresis

Test failures are grouped together by failure signature in bins known as 'buckets'. Buckets are another way to measure your regression health. Running 100 tests and having 10 tests fail with exactly the same bucket signature means, hopefully, that debugging and fixing one of them will fix them all. Conversely, have 10 test fail with 10 different buckets means that you *must* debug them all and that they have a reasonably high likelihood of different root causes.

In teams that continually run tests, undebugged failures can begin to pile up, especially if the pass rate is poor. If 10 test fail for every 100 run, after you have run 1000 tests you might have 100 failures. Again, if they are all of the same bucket then there is some likelihood that a single fix might make them all go away. But if they are different buckets... well, you have a lot of debug and fix work in front of you. So, bucket count is a better indicator of the amount of debug effort you are facing and a better indicator of the quantity of bugs that will be exposed.

As mentioned in section 6.9.4 project managers want the bug rate to fall, so they are very interested in seeing the bucket count drop as well. But, this isn't that simple, and that is because Validation typically has a *bucket hysteresis curve*.

When there is a high pass rate, and when the bucket count is low, it is Validation's job to *create more failures!* They need to enable more features, add more configurations, add more checkers, add more randomness, and so on to increase the stress on the system and find more bugs. When the failure rate becomes too high or the bucket count too high, Validation must stop adding extra stress and work to get the failure rate back under control and get the bucket count down (lest the

health of the overall DUT be poor... and have everyone assume that failures are due to ‘known bugs’ and not their new code!). Sometimes teams manage this back-and-forth relationship in a way that keeps the bug count relatively constant. Sometimes bucket counts can shoot up quickly and thus the team needs to do a major focus shift to ‘debug and fix’, also known as ‘get healthy’.

Get healthy periods appear to stall project execution, but they are better to do earlier than later because anytime you are carrying too high of a failure rate and bucket count it can cause spurious failures in your gating regressions... and everyone assumes it is ‘not their own fault’. If you do not trust your gating regression and your overall DUT health, you are in trouble. Extraneous failures act like an anchor on a boat: you may want to go fast, but they will slow you down!

Bucket hysteresis is part of the job. It is Validation’s job to drive the failure rate higher... and then to drive the failure rate lower! And continue this cycle throughout the project until they expend all of their ideas for bug finding.

6.9.6 Ownership and Responsibility

A consistent theme in the engineering world is that ownership and responsibility are critical to successfully completing tasks with quality. Items that are deemed *everyone’s responsibility* can sometimes end up with nobody performing that item. Historically that has happened with edicts such as *quality is everyone’s responsibility* and *everyone is a security validator*. These are conceptually fine but they rarely lead to action. It is critical as a team to recognize these collective responsibility situations and turn them into concrete action. For example, if the team deems that Test Environment quality will be a focus for a project, then it should also determine systems or actions to make it happen, such as a system to require code reviews or pair programming or unit testing.

Shared responsibility is less risky if there is an obvious indicator that monitors the situation. As mentioned in section 6.9.5, many teams share the ownership of debug and keeping the number of open buckets below some threshold. On any given day multiple team members might not work on debug and may think that others will cover; in such a case the un-debugged failures may rise significantly. In such a situation, if there is an indicator, the team can realize this and take appropriate action to get back on course.

6.9.7 Closed Loop Feedback

Creating closed loops of feedback is good engineering practice. As a Validator, it is important that you systematically verify that everything works, and verify that if something is not working that you will be alerted via some mechanism. For example, we write tests to exercise microarchitctural conditions and use coverage monitors to verify that the tests exercised the conditions. In a similar way, a Validator writing a checker should write unit tests and possibly full test cases for the checker to verify that it does indeed report incorrect behavior, or use a tool to ensure checker functionality like Certitude¹².

¹² Certitude is a tool for injecting faults in simulations to verify that your checker can detect the faults.

Many important processes have loop methodologies. For example, when a bug is filed in a tracking database, the Designer is notified. When the Designer fixes the bug, it goes back to the Validator who validates that the bug is fixed. This methodology loop reduces the risk of the bug not being fixed or fixed incorrectly by involving the bug filer after the fix has been made to *confirm* that the solution is valid. For more details on this process, see [The Life of a Bug](#). You can take this too far and end up doing very low ROI activities. So balance this with trust in the abilities of your team and tools to do the right thing¹³.

Another important feedback loop is for *bug escapes*. A bug escape is any bug found downstream from where it *should have been found*, which can be subjective. Two commonly discussed (and measured) escapes are:

- IP bugs being found by SoC teams
- Logic bugs being found in silicon

There are many forms of escapes; the important way to look at them is to ask *should this bug have been found previously by an ‘upstream’ activity?* And taken from the opposite angle, *why did this bug escape my activities and get found downstream by others?* Closed loop escape analysis should be done to understand why escapes happen and to fix whatever systems or collaterals allow escapes.

By ensuring that you have closed loops of feedback in your processes, you significantly reduce the risk of bugs going undetected in both the design you are testing and in the tools that you are using to test the design. Remember that bugs in your tools and infrastructure can hide bugs in the design and deficiencies in your understanding.

6.9.8 Continuous, Aggressive Improvement

Moore’s Law states that the number of available transistors doubles every two years. It has some profound impacts on Validators, not just Designers and folks driving our process technology. Because of Moore’s law, our ability to validate transistors needs to increase at a similar rate, or headcount must rise on each project on a log scale. Obviously increasing headcount in that way is not financially possible. While changes in our product architecture and increasing level of abstraction helps reduce the rate of change we are exposed to, the pace of improvement required is still very high.

Because the projects require that we deal with continuous significant scope growth with fixed headcount, you cannot assume that what worked on the last project will work on the next one. You must significantly increase you and your team’s ability to deal with increasing scope and complexity on each project. This means being able to create abstraction layers and automation, maximizing reuse, finding and becoming proficient in new tools and methodologies, using feedback to continuously make improvements and eliminate waste, and innovation to find breakthrough methods.

¹³ For more on this balance, see “The Speed of Trust” by Steven Covey

The history of the technology industry is littered with the remains of companies who failed to innovate, improve and adapt to a fast-changing world. This is the industry that we are a part of and need to be aware that we must be constantly growing and adapting to be relevant.

6.9.9 Destructive Engineering

The mindset of a Validator needs to be different from that of other engineers. Validators are often accused of being destructive, negative, and evil in their thinking. While this may seem bad, there is a reason for this mindset that is beneficial.

Thomas Edison said that he found 2000 ways “not to make a light bulb” before he found the one way to make a light bulb. As Validators we will run 2000 passing tests that tell us “how not to improve the design” before we find the one way to improve the design via a failing test that exposes a bug. This doesn’t mean that the 2000 passing tests are useless; we can use them to guide our testing to untested areas via coverage analysis, but getting the one failing test, like the one working light bulb is frequently the focus of our activities.

Finding bugs (improving the design) usually requires the Validator to be creative. Once the design is working, Validators still need to find the remaining bugs. These bugs will continue to exist in uncovered (or under-covered) areas of the design, and areas of the design that are not used frequently. This causes Validators to focus on “gap areas” and write tests that use the design in unusual ways. For example, multiprocessor validation tests use some of the worst performing synchronization algorithms, because it forces the processors to do more work (and hit more conditions) than a “properly” coded synchronization algorithm would. Many validation tests are not “well behaved” programs and rarely perform any useful functions. Conversely, it is also true that Validation tests real-world use cases and algorithms as well. See [The Validation Mindset](#) for further discussion.

6.9.10 Expertise

The process of validation is comparing the design under test to a reference model and flagging the differences. These reference models may be as specific as the IEEE floating-point specification, the IA32 Programmers Reference Manual, or as simple as the expected behavior of the unit as understood by the Validator. There are usually multiple levels of reference models for any particular area of functionality. Validators need to have a clear and complete understanding of the reference models and the implementation of those models in order to make the comparison.

Being an expert in your area of the design has many other benefits. Being an expert provides prestige to you and to the Validation team. Frequently, Validators know their areas as well as, or better than the Designers who wrote the code. Because of this, Pre-Silicon Validators are called in as experts to be able to quickly make important decisions, or debug critical failures. Proactive experts take this further and create tools, training and collateral that aids debugging of that area by others. For more detailed information on this topic see [Becoming the Microarchitecture Expert](#)

7 Summary

Validation is a challenging and rewarding activity that requires solid engineering skills and creativity. We hope that *The Art of Pre-Silicon Validation* will help enable you as a Validator or a Validation manager to understand why and how to do validation.

8 Future Work

Create guidance on how to prevent bugs. What architecture or design items are inherently more difficult to validate? This comes up in section 6.7. One example is that latency is generically bad – is this a theme that we can discuss in the intro?

How do we recognize and reward the prevention of bugs?

More discussion on not all bugs being equal in value or impact. Possible additions to sections 6.4 and 6.5.1. Maybe more on bugs that prevent customer (post-silicon validation) progress?

Increasing fab latencies are putting a lot of pressure on resolving issues pre-silicon instead of post-silicon. This trend appears semi-permanent. There is also the 2-step PRQ goal. Should there be some discussion of that in this chapter?

As we own more of the software aspects of the platform it puts pressure on the product teams to produce pre-silicon platforms appropriate for software development and validation. Is that something we should also discuss here and the IP or product teams' relationships with the SW teams.

Not as much discussion about the importance of hitting schedule. It affects customers, cadence and product competitiveness. In addition, there are opportunity costs associated with this.

Cost of bug discussion does not really emphasize prevention and early designer discovery. Add a “code turnin” point to the graphs to make that more clear?

Discuss in more detail when high quality validation is not appropriate?

Requirements based validation and/or use case validation. When or where should this be discussed?

What specific skills do validators need to have? How do we cultivate them? One particular skill is SW engineering to build good, reusable collateral.

Ratio of time spent on environment issues versus debugging and DUT issues. This varies between areas a lot (particularly as one goes up the hierarchy). Should we attempt to cover this topic?

Bugs are social creatures. It might be able to be added to section 6.7, but it is not a great fit currently.

We do not currently have a chapter on cost and effort estimation. Would be nice so we could introduce the idea here and then reference the chapter. There is a lot of need for validators to engage with that process.

Should there be a section on “what a successful project looks like” or “what is success for a validation team”?

9 References

Validating the Intel Pentium 4 Processor by Bob Bentley and Rand Gray. Published Q1, 2001.
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=941434&tag=1

Bob Colwell's *At Random* columns in IEEE Computer society magazine from 2001-2005. We particularly like the columns from April, May, June and October of 2002 as well as August and September of 2003. <https://ieeexplore.ieee.org/author/37271738200>

The Art of Pre-Si Val: Chapter 2

Validation Disciplines

By: Matthew P. Kupperman and Mike F. Miller

1 Abstract

Effective Validation of a product requires utilizing many Validation Approaches and Validation Disciplines across multiple levels in the design. This chapter provides a framework for understanding Validation Disciplines, an introduction to disciplines and a discussion of how DDG currently maps those disciplines into a team structure.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	8/01/2012	Initial Draft	Matthew Kupperman	Michael Bair, David Rogers, Kalpana Kothapally, Erik Samuelson, Wei Kang Teh, Annette Bunker, Rajnish Ghughal, Steven Saar, Steve Petersen, Chun Tan, Mike Libby, Robert Beers
2.0	2/8/2016	Restructured to cover Validation Approaches and Team Structure	Mike F. Miller	DDG management
2.1	3/16/2016	Restructured Design Hierarchy, Validation Approaches. Many enhancements based on discussion group feedback	Mike F. Miller	Michael Bair, Lance Geiger, John Faistl, Matthew Chidester
2.2	12/18/2022	Added Perf, PnP, FuSa Val; small fixes	Michael Bair	Michael Bair

3 Contents

1 Abstract.....	33
2 Chapter Revision History	33
3 Contents.....	34
4 Purpose.....	36
4.1 Why do we need this chapter?.....	36
4.2 What does this chapter cover?	36
4.3 What does this chapter not cover?	36
5 Background Concepts.....	36
5.1 What is a block?	36
5.2 White and Black Box Validation.....	36
6 Validation Disciplines	37
6.1 Design Hierarchy and Validation.....	37
6.2 Validation Approaches.....	39
6.2.1 Interconnection Validation.....	39
6.2.2 Feature Validation.....	40
6.2.3 Implementation Validation.....	40
6.2.4 Cross Feature Validation	41
6.3 Validation Disciplines.....	42
6.3.1 Dynamic Validation	42
6.3.2 Formal Verification.....	43
6.3.3 Integration Validation	43
6.3.4 Firmware Validation	44
6.3.5 Mixed Signal Validation.....	44
6.3.6 Reset and Power Management Validation.....	45
6.3.7 Intel Architecture Validation	45
6.3.8 Design for Test Validation.....	46
6.3.9 Design For Debug Validation	46
6.3.10 Security Validation	47
6.3.11 Performance Validation.....	47
6.3.12 Power Performance (PnP) Validation	47
6.3.13 Functional Safety (FuSa) Validation	47

6.4	Team Structure	48
6.4.1	DDG 2016 Example	49
7	Summary.....	52
8	Future Work.....	52
9	References.....	53

4 Purpose

4.1 Why do we need this chapter?

Chips are very complex devices, and because of Moore’s law, they continue to increase in complexity. They contain a wide variety of features with different purposes and characteristics. Validators use a number of disciplines, or areas of specialty, each targeting different problems or utilizing a particular approach to meet this challenge. Understanding your Validation Discipline helps you understand unique properties about your work. Understanding the other Validation Disciplines helps you understand validators outside your immediate team and interact with them more productively.

4.2 What does this chapter cover?

This chapter has an overview of design hierarchy, Validation Approaches, Validation Disciplines and application of the Validation Disciplines to an example team structure.

4.3 What does this chapter not cover?

This chapter is just an overview. Subsequent chapters go into more depth on nearly every topic introduced here.

This chapter does not discuss cross-group interactions to keep the scope sufficiently bounded to not become exceptionally long.

This chapter does not discuss [Validation Platforms](#) (simulation, emulation, etc.) except in cases where a discipline has a unique platform. While tradeoffs are different within an approach or discipline depending on what platforms are available, the discussion is high-level enough that we ignore those details.

5 Background Concepts

5.1 What is a block?

For discussion purposes, we use the term “block” to refer to a unit of the design. It could be as small as a handful of RTL files, or as large as an entire chip. It can be either RTL or Firmware (FW). This allows discussion of validation concepts independent of a specific implementation or scope details.

5.2 White and Black Box Validation

Each presilicon validation team approaches the problem of finding bugs and providing confidence in the design in different ways. By attacking the validation problem from different angles, the team as a whole succeeds.

Consider a design under test to be a box of logic. Black box testing looks at the specification of the inputs and outputs of the box, and tests to those specifications. In the case of IA-32 validation, our [Architecture Validation](#) (AV) team relies primarily on black box testing. They take the Software Developer's Manual and test that the processor behaves as specified. Most of their tests are portable from one IA-32 processor to another.

White box testing takes the cover off the box and looks at how things are implemented. The white box Validator looks at internal logic and looks for bugs that exist in that logic. The Validator writes tests and checkers that test the implementation of the specification. As a result, the white box Validator is intimately familiar with the inner workings of the block.

Both black box and white box testing need to be done to fully validate a complex design. White box will find bugs that black box testing will not and vice versa. A simple example is a queue-full boundary condition. Black box testing would not know or care about the scheduling queue used to feed micro-operations to the core. If there is a bug when the queue fills, black box testing would not target it, but white box testing would. Conversely, if an IA-32 instruction does not behave the way the specification says it should, white box testing might not catch the problem. White box testing derives its test plans from the bottom up of a design. Black box testing starts with the architecture and drives down to the design.

6 Validation Disciplines

To understand validation disciplines, it is helpful to have a broader understanding of the problem structure, and validation approaches. This knowledge is applied to understand Validation Disciplines and how teams are formed to efficiently validate designs.

6.1 Design Hierarchy and Validation

One of the results of Moore's Law is that we have ever-increasing size of our designs. This introduces ever-increasing complexity as we integrate more functionality into our products. A single team sufficiently large to create all the code for a modern SOC from scratch would require thousands of engineers. Managing and coordinating a team of that size with variable workload in different areas of the design would be an impossible task. One of the primary mechanisms for dealing with the complexity of the product is creating hierarchy and encapsulating functionality. By isolating the development and validation of blocks from each other, it decouples the people doing the work so they can work independently. With sufficient standardization, they can even reuse blocks between projects, reducing effort. This hierarchy allows some blocks, usually called IPs¹⁴, to be developed in different organizations or external companies and then integrated into projects that are fabricated and sold to customers.

¹⁴ IP stands for "Intellectual Property" but in our context, it generally refers to a block designed and packaged for use in multiple products.

Hierarchy enables scaling up our designs, but also introduces new problems. While blocks are individually coded and validated, they must also be integrated into a product to be sold¹⁵. Integration of the IPs and validating that the product as a whole operates as needed is a challenging problem. To understand hierarchy better, consider this highly simplified example of a client product containing both a IA Core chip and a PCH chipset.

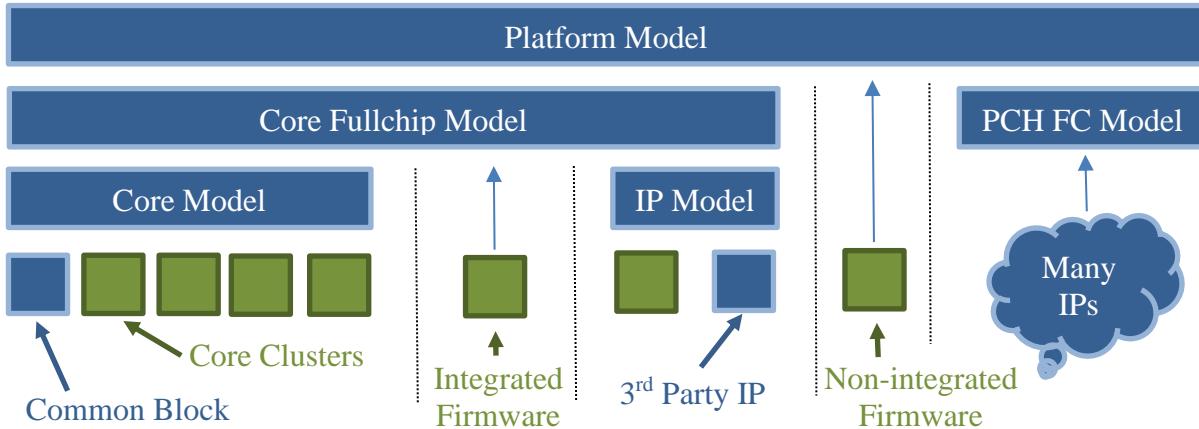


Figure 1: Composition and Models of a client product

Working in the example from bottom left to top right:

A large IP like a core is usually composed of multiple blocks built by the processor designers which are validated and then integrated, possibly with some common components from another team into an IP which is delivered to an fullchip team (or perhaps a subsystem team) for further integration. It is then integrated into fullchip model that represents everything on a single die. Often a more comprehensive model of multiple chips is built to test platform interactions.

An integrated firmware will be validated in a software development environment like FORE¹⁶ or simics¹⁷ which is similar to a cluster environment. It is then integrated into the RTL model. The “interfaces” and the “features” to be validated may be different than pure hardware, but the activities are fundamentally the same.

For some situations, a non-Intel 3rd party IP is used. Usually we need to combine it with some custom code to make it interface correctly with the rest of the design. This custom code will need to be validated and then integrated with the 3rd party IP. This work is done independently of the rest of the chip and then delivered as an Intel-compatible IP to the fullchip team who integrate it into the fullchip model.

In some cases firmware is developed independently of the RTL model and only integrated later, sometimes with a chipset if the firmware requires both chips to be fully modeled to run correctly.

¹⁵ Some business models sell IPs to other companies, but eventually somebody needs to fabricate the design and sell chips to end customers to make money. Otherwise the business model is unsustainable.

¹⁶ FORE is a high-performance firmware development environment that allows testing of firmware without the associated RTL.

¹⁷ Simics is a high performance system simulator that enables running software on a simulated platform. <http://www.windriver.com/products/simics/>

For very large blocks, the models required to do IV/AV are expensive and slow, but it is also very powerful at identifying issues that slow down or block the post-silicon teams because of the high fidelity of all components. The platform AV work would be restricted in scope to features that utilize connections between the CPU complex, Chipset and nonintegrated FW. Revalidating features that do not cross those boundaries is very low ROI.

In conclusion, at each level of the hierarchy, validators need to examine their block, what validation was done at the level under them, and what their customers at the next level up in the hierarchy need. Then the validators can determine how different approaches should be applied at that level of the hierarchy. The appropriate combinations of approaches at each level of the hierarchy results in high quality, cost effective validation for the entire product.

6.2 Validation Approaches

In an ideal situation, you could black box test every possible combination of inputs on your design at the user interface against a perfectly written specification. If you could do that, then you would ship your product with complete confidence. Real-world product development is very far away from that ideal. To manage that disconnect we break the problem down into smaller, more manageable aspects both in hierarchy and approach.

Four approaches collaboratively help segment the validation space and maximize validation ROI.

- *Interconnection Validation:* Focusing on the communication interfaces between sub-blocks inside the block being validated is usually lower cost than debugging failures using Feature Validation stimulus.
- *Feature Validation:* By isolating a specific externally visible feature of a block, validators can focus on that aspect to more easily create content and drive out bugs.
- *Implementation Validation:* By identifying key implementation details within a block, validators can focus stimulus, checking and coverage on high-risk areas.
- *Cross Feature Validation:* Combining knowledge from Feature Validation and Implementation Validation, high-risk combinations of feature behaviors are targeted to close holes in the other approaches.

These approaches are rarely used in isolation of each other, but allow for parallel execution of many activities. The mix of the four approaches varies widely depending on the level of the hierarchy, quality of the design, performance of the models, quality of specifications, and complexity of features at the block interface. Getting the right mix of the four approaches is key to cost-effective validation.

6.2.1 Interconnection Validation

The goal of Interconnection Validation is to verify that a block which has already been validated is correctly connected (sometimes called integrated) with other blocks. The primary source for defining correct behavior is that the block's connections and communication mechanisms with other blocks are functional. This is a “white box” approach



Figure 2: Interconnection Validation focuses on the connections between the block being integrated and other blocks. The connections in the red circle in this trivial example.

as the validator is concerned with the implementation of each of the communication mechanisms, but it is also “black box” in that the validator generally ignores implementation details within the boxes being integrated. Interconnection Validation is usually focused on breadth of coverage, making sure that each of the mechanisms is basically functional, not attempting to target unusual corner case conditions unless there are demonstrated issues in that area. This is because other validation work will usually cover those cases and the expense of going after them usually results in poor ROI. Integration validation requires that most (if not all) of the directly interacting blocks be available and healthy enough to integrate. In practice, Validators working on integrating a block are generally able to work independently of Validators on other blocks as the staging of the integration and standardized interfaces hides most dependencies. A team testing power management interactions while integrating a PCIE IP can mostly do that in independently from someone doing a similar test with a memory controller IP. Section 6.3.3 on Integration Validation and the [Integration Validation](#) chapter goes into more depth on this topic.

6.2.2 Feature Validation

Feature Validation’s goal is to verify that a block both conforms to a specification and satisfies customer need for a particular feature. Customer needs should be clearly defined in an architectural specification¹⁸, but frequently need to be augmented or clarified. Part of a validator’s job is to identify issues and work with Architects to clarify and complete specifications ensuring that everyone has a clear understanding of how the design ought to operate¹⁹. By splitting the external behaviors into individual features, it allows validators to more easily focus on a feature or small group of related features. This allows for parallelism in the work as well as manageable scope. It does potentially create gaps between features, which Cross Feature Validation attempts to resolve. Because Feature Validation starts with customer requirements or external interface specifications it is a “black box” activity. In practice, validators will also utilize implementation knowledge to enhance their debug, stimulus, checking and coverage.

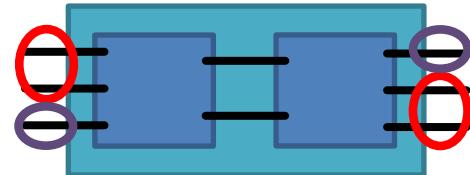


Figure 3: Feature Validation focuses on specific subsets of interface behaviors. For example, the “red feature” and the “purple feature” would be the two Feature Validaiton activities

6.2.3 Implementation Validation

One of the largest drawbacks to pure black box validation is that it is very inefficient in detecting implementation bugs. Looking at the implementation frequently identifies very interesting cases that are likely to result in bugs. As a trivial example, consider a simple 32-bit adder. The external

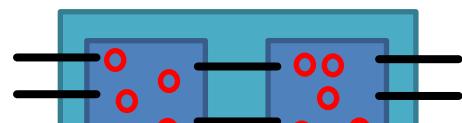


Figure 4: Implementation Validation' focuses on select conditions within blocks.

¹⁸ By “architecture specification”, we include any documented interface that customers (including internal customers like post-silicon debuggers or fab testers) will use to interface with the block.

¹⁹ In the absence of a collaborative environment, some validators have been known to write a spec on the side.

requirements would indicate you treat all possible 2^{64} inputs as equally interesting (possibly with the exception of inputs that result in the carry-out bit being set). Alternatively, knowing that the adder implementation is a set of identical 8-bit carry-lookahead adders would allow you to construct a radically smaller set of stimulus (~600 inputs) to exhaustively test the adder. The downside is that if the implementation is not what you think it is, or it changes, the stimulus is worse than random. The Validator must carefully weigh substantial expense of developing and maintaining implementation-specific infrastructure versus the benefit it provides.

Frequently the low-level design is planned and documented in an implementation specification²⁰. In those cases, the documentation is invaluable in understanding the design details. That is obviously not the only source, as the specification may be incomplete, incorrect or insufficiently clear. Part of any Validation activity is working with Architects, Designers, other Validators and sometimes customers to understand fully the correct behavior. Implementation Validation takes engineer time to examine the implementation and uses judgment to evaluate what is interesting and/or presents risk. This is often a reasonable cost as it can significantly increase the quality of stimulus by directing it to high-value areas. See the [Dynamic Microarchitecture Validation](#) and subsequent chapters for more information on Microarchitecture Validation which is Implementation Validation applied to the lowest levels of the design.

6.2.4 Cross Feature Validation

Cross Feature Validation²¹ is specifically to close gaps that exist between separate Feature Validation activities that were done independently of each other. In contrast to Feature Validation, Cross Feature Validation is breadth focused, trying to mix and match running different features together to find bugs in how the features interact. It is a black box activity because externally visible features are the focus, but for practical purposes, the selection of what to combine is informed by implementation knowledge of the design as well as previous history of where failures have occurred. For example, the majority of DFT²² features are never used at the same time as IA features²³ and do not share hardware, so we do not make effort to test them together. Cross Feature Validation requires that the validators work closely with Feature Validators to combine validation stimulus from different features.

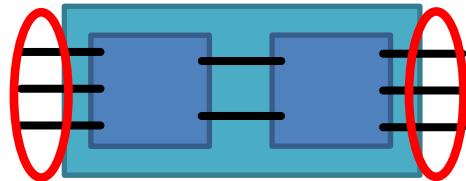


Figure 5: Cross Feature Validation considers the full spectrum of features and their interactions exposed at the block interface.

²⁰ Sometimes referred to as a “High Level Architectural Specification” (HAS) or “Microarchitectural Specification” (MAS). Both tend to contain significant implementation details for a block.

²¹ Some people refer to this as concurrency, but this can be confusing to others as that term is sometimes used specifically to refer to multiple-source memory traffic validation.

²² See section 6.3.8 Design for Test Validation

²³ See section 6.3.7 Intel Architecture Validation

6.3 Validation Disciplines

Validation Disciplines are areas of specialty within Validation that currently are relevant to our products. Understanding disciplines will help you understand the unique challenges and opportunities that exist across Pre-Silicon Validation as well as some of the motivations behind team structure that are explored later. These Disciplines are not mutually exclusive, some are complementary, some are sub-disciplines and some overlap in functionality.

Disciplines change over time and not all projects require all disciplines. For example, Back on the Pentium 4 designs, we had separate Feature Validation Teams for Multi-threading Validation and Multi-processor Validation because those features were new, very broken, and required focused effort to learn how to effectively validate them. Teams were formed with specific, dedicated resources to understand the features and then develop methodologies and tools to solve the problems with validating the features. As the features became healthy, those teams trained up the larger validation organization to recognize and deal with the issues associated with those features and those disciplines no longer needed a separate team. This is a very good thing. We continue to add features to our designs. If we had to permanently fund a new team for every new feature, our costs would be horrible! Every Validator needs to find ways on every project to reduce, or eliminate work while retaining quality to keep total effort flat or down with increasing complexity.

6.3.1 Dynamic Validation

Dynamic Validation is the technique taking a block of the design and running it to determine what its behavior is for a given set of stimulus. The behavior is compared to some reference or expectation to determine if the behavior is correct by some form of automated checking. Finally, coverage of interesting events that occurred during the run is collected to ensure that the stimulus had the desired effect on the DUT. The majority of validation is done using this technique, because the design team's output is easily consumed by Dynamic Validation tools and other techniques have scalability issues. Dynamic validation may not seem like a unique "discipline" because most other disciplines build upon the knowledge and skills associated with Dynamic Validation.

To do dynamic validation on a block, you need some form of Test Environment that mimics the behaviors of the blocks around it, or the real world. Once validated, the block is integrated into a larger model with more blocks until an entire system is composed. Generally, because larger models contain more of the design, they are slower and more expensive to build and run.

The [Test Environments](#) chapter discusses the issues and complexities of simulation TEs and reuse of that content. Dynamic Validation is used at all levels of the design hierarchy including validation of Firmware and Software. All of the Validation

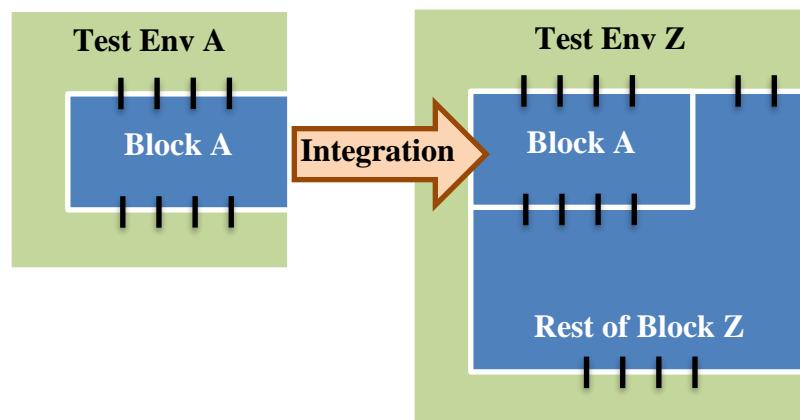


Figure 6: Integration

Approaches work well with Dynamic Validation, and it is used in most of the other Validation Disciplines. Dynamic Validation is done on all [Validation Platforms](#).

6.3.2 Formal Verification

Formal Verification is a technique that provides a different toolset than Dynamic Validation. With Dynamic Validation, if a bug exists, the right stimulus must be applied to excite the bug, and a checker must detect that something has gone wrong. In contrast, formal methods use mathematics to in effect apply all possible inputs and calculate all possible outputs and determine if any of these violate the specified correct behavior. Formal approaches can be applied in several ways. First, formal tools are used early in the design cycle to model proposed protocols and quickly identify issues. Additionally, Formal Property Validation enables validators to validate properties of low-level RTL. Finally, an approach known as Symbolic Simulation supports validation of RTL correctness. In each of these approaches, a variety of implementations that yield different levels of quality are possible, ranging from bounded model checking where a “good enough” limit is set, to full proofs. In many cases the Formal Verification collateral can also be used in Dynamic Validation as additional checkers which produce very helpful error messages when the design deviates from expectations.

While there are limitations to Formal Verification techniques, they are extremely powerful tools that can be used at several different stages of the design process. All validators should understand enough of these techniques to know when to apply them to their work. The chapters [Introduction to Formal Verification](#), [Formal Property Verification](#), [Protocol Formal Verification](#) and [Symbolic Simulation](#) provide a good basis for understanding what is possible using these techniques.

6.3.3 Integration Validation

The primary goal of integration validation is to bring a block into the larger model as quickly and inexpensively as possible to enable backend physical design and other validation activities. While it was not always called out as a separate discipline, in the SOC environment where project teams receive IPs from multiple teams, the role of the Integration Validator becomes increasingly distinct and critical to project success. While this sounds straightforward, numerous pitfalls require unique skills and experience to avoid. The [Integration Validation](#) chapter discusses these in detail.

Integration validation generally relies on two approaches: Interconnection Validation and Feature Validation. This is because their goals are highly time and resource sensitive. This varies between IPs and over time, so in practice, a combination of the two approaches is used to gain the benefits of both as described in the table below:

Comparison of Validation Approaches for Integration Validation		
Approach	Benefits	Drawbacks
Interconnection Validation (interface is white box)	Good for identifying all cases, even those outside normal feature use.	Can be hard to test cases independently. Undocumented interconnection requirements can hide bugs.

Feature Validation (interface is black box)	Usually easier to acquire or reuse existing stimulus. Can hit undocumented interconnection requirements.	Some interconnection primitives might not be tested if the feature tests do not happen to cover them.
-------------------------------------------------------	----------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------

Figure 7: Approaches to Integration Validation

Integration Validators also have a unique relationship role in the organization. They are not experts in the IP, but they are responsible for that IP in the context of the SOC. This means that they are responsible for evaluating risk of an IP or features in that IP as they integrate it and communicating that to the IP team and the other SOC validation teams who are impacted. They are the validation team's connection to the IP delivery team. Building a good working relationship with that team is very important and can take years in some cases. Good relationships with the IP team results in quick resolution of problems, open communication and evaluation of risks, effective staging and balancing of work for the good of the entire company.

6.3.4 Firmware Validation

Today's chips must provide a variety of complex functions. Firmware performs tasks ranging from coordinating the operation of multiple hardware units to providing functionality beyond what can reasonably be implemented in hardware. Firmware in Intel CPUs also has the advantage that it can be patched even after the design has been manufactured. Since firmware is not hardware, it requires a different set of tools to validate it. For example, their validation environments are usually software simulators of the processors that the firmware runs on, like FORE²⁴ or simics²⁵. Focusing on a specific firmware allows the validators to become experts on that firmware and develop specific tools that are well suited to discovering bugs in that firmware. Chapter [Microcode Validation](#) and [Embedded FW Validation](#) cover these disciplines.

6.3.5 Mixed Signal Validation

In our products, the design is split into analog and digital parts. The majority of our discussion in the *Art of Pre-Silicon Validation* is about validating the larger digital portion written as RTL or firmware source code. However, the analog part is also critically important because it includes clocks, I/O pads, sensors, voltage regulators and other circuits that are required for the chip to work.

The Mixed Signal Validators validate the functional interactions between the analog and digital parts. Mixed Signal Validation employs dynamic validation and formal verification (where beneficial) on relatively small models, focusing on the interactions between the digital controls and the analog blocks. The analog blocks are challenging to work with because they are usually

²⁴ FORE is a high-performance firmware development environment that allows testing of firmware without the associated RTL. See also [_LINK_FW validation_LINK_](#) for more information about why this is so important.

²⁵ Simics is a high performance system simulator that enables running software on a simulated platform. <http://www.windriver.com/products/simics/>

behavioral RTL written from the schematics instead of being schematics synthesized from the RTL. That modeling is not proven to be correct. This can lead to discrepancy between the model and schematics because there is no automation to verify they are functionally equivalent. To address this problem, in addition to using the standard digital simulation tools, the validators use mixed signal simulation tools that can simulate RTL and schematics together. Validators doing Mixed Signal Validation must have knowledge of both digital design and analog circuit concepts. This is an example of a validation discipline created around a problem that requires unique skills and tools. For more details see chapter [Mixed Signal Validation](#).

6.3.6 Reset and Power Management Validation

Intel's customers expect our products to consume progressively less power to improve battery life and thermally fit into smaller form factors. To meet this critical power management challenge without sacrificing performance or responsiveness, we build increasingly complex features into our platforms. At a high level, these features are implemented to match their specifications and must not break the functional correctness of the normal operation of the platform. In Intel platforms, the reset implementation is very tightly coupled with the power management functionality. Because the problems became nearly identical, we have combined the validation those two features. This is an example where we previously had two distinct disciplines, but as they converged, we merged them into a single discipline.

Most power management features are global in nature, with communication and control that spans across the entire chip, including external devices and software. These flows require a combination of hardware and firmware, which requires validation at higher levels of the model hierarchy that contain all this code. Moreover, many power management flows have microarchitectural interactions with complex features such as locks and interrupts, requiring validation of feature cross-products. While Power Management Validation requires all three approaches, within DDG the discipline is focused primarily on the Feature Validation approach to ensure that the customer-facing controls are correct.

Several attributes of Reset and Power Management Validation make it unique relative to other Feature Validation activities. The first is that the execution time required to run the RPM flows is very high, making efficient simulation difficult. This can require creativity in breaking up flows, or creating acceleration mechanisms to validate critical parts of the flows faster. Second, modeling of power is specified by Unified Power Format (UPF) files, not RTL and requires special Multiple Power Plane (MPP) models to simulate the supply of power. Finally, some power management require saving and restoring machine state to function correctly after a power event. Power modeling requires special handling by the model (simulation or emulation), test environments and checkers. Enabling and debugging with these additional modeling capabilities requires awareness and tools to work efficiently. Chapter [Reset and Power Management Validation](#) has details on this discipline.

6.3.7 Intel Architecture Validation

Intel Architecture Validation is a Feature Validation activity that focuses specifically on the external programmer specifications, which have some unique properties. A well-defined and

validated design that meets the published specifications enables software programmers and end-users to write useful and productive programs that function as expected when run on the target CPU. The goal of Intel Architecture Validation is to ensure the product is an IA processor and will run software as expected. This is particularly critical for Intel because one of our key value propositions to customers is backwards compatibility with our previous products so they can run their old software on our new platforms. Chapter [Architecture Validation](#) covers this in more depth. Currently DDG does not do much Intel Architecture Validation as it is done primarily on a Core IP, but we have included it for completeness and there are a few tasks that require interactions between a Core IP and non-Core components to be Intel Architecture compatible which we validate.

6.3.8 Design for Test Validation

Design for Test (DFT) Validation uses a Feature Validation approach for hardware features designed for enabling efficient High-Volume Manufacturing (HVM) or tester debug. The goal of DFT Validation is to validate that both the design and, perhaps more importantly, the proposed usage models of these features will meet HVM's requirements. Enabling HVM is critical because testing manufactured chips for defects is a non-trivial part of the per-unit cost of our products. The DFT Validators maintain close ties with their customers, the HVM and tester teams, to facilitate understanding of feature capabilities and usage models by both groups, and often engineers from all these teams fund the effort. While DFT Validation is similar to other Feature Validation based disciplines in many ways, it also has many unique challenges. First, DFT features have very specific customers and unique properties. Second, the architectural definitions and usage models are often unclear and evolve with the product. Third, DFT logic is often implemented late in the design process. Finally, DFT features do not lend themselves well to random testing. Having dedicated DFT validators enables the specialization needed to overcome the challenges in this area. See chapter [Design for Test Validation](#) for more details.

6.3.9 Design For Debug Validation

Debug features are critical to the Post-Silicon phase of the project, and Design For Debug (DFD) Validators are responsible for ensuring their functionality. As modern SOCs continue to incorporate more and more logic that used to exist on separate chips, seeing what is happening inside the chip becomes more difficult. These features are key in providing visibility so that functional failures can be efficiently identified and root caused. Debug features are also increasingly being made available to Intel's OEM customers for their use debugging their platforms.

The DFD Validators take a Feature Validation approach, considering the post-silicon debuggers to be the customers of the features. They reuse Dynamic Validation infrastructure. DFD Validation plays a critical role in the success of our projects by taking part in defining the right debug features, ensuring the correct operation of those features, and supporting post-silicon debuggers. For more details, see chapter [Design for Debug Validation](#).

6.3.10 Security Validation

Security is a vital aspect of our product designs. As our products are used in more valuable and strategic applications, the security standards continue to increase. To protect our own assets, as well as those of our customers and third parties who use our platforms, we have implemented a number of security features that require validation. Validation of these features, a relatively new discipline, is focused on enabling new usage models through enhanced security. Previously, security-related validation activities were handled under the Intel Architecture Validation discipline, but as the requirements grew, it became necessary to create a separate discipline for this purpose. Chapter [Security Validation](#) discusses this discipline in detail.

6.3.11 Performance Validation

Performance is a key factor in evaluating our products. It is generally measured by the speed at which our CPUs can execute instructions and complete tasks, but there are also other intermediate measures such as the amount of bandwidth the CPU has for transferring data and the speed at which data can be retrieved from memory.

Performance Validation has traditionally involved a combination of Architecture and Validation activities, without the strict adherence to a standard validation test plan. In the past, tests and benchmarks were run and compared to high-level expectations (when available), and any discrepancies were debugged to identify bugs or improve our understanding of the architecture and micro-architecture.

With each new product generation, we are increasing the rigor of both performance specifications and performance validation. For more information, see the [Performance Validation](#) chapter.

6.3.12 Power Performance (PnP) Validation

Power Performance is another key factor in evaluating our products. It refers to how efficiently a product uses power while completing tasks, such as how much power is consumed in relation to its performance. For many computing products, the long-term cost of ownership is heavily influenced by the cost of power and cooling. Power Performance Validation ensures that the design meets specifications for conserving power, such as shutting down clocks when they are not in use and lowering or shutting off power domains when they are not needed. This discipline also looks for unnecessary power usage, or "power wasters," beyond the specification, in order to optimize power efficiency. For more information, see the [Power Performance \(PnP\) Validation](#) chapter.

6.3.13 Functional Safety (FuSa) Validation

Functional Safety (FuSa) Validation is an emerging validation discipline within Intel. FuSa Validation is done to ensure that a product or system functions safely and reliably under all intended conditions. FuSa Validation is critical in applications where a failure could have serious consequences, such as in the automotive, aerospace, and medical industries. In these industries there are regulatory bodies and standards committees that set standards and requirements. FuSa

Validation works at both a feature-level and a product-level to verify that the product meets those standards and requirements

6.4 Team Structure

Forming good teams is a very challenging management activity. There are many conflicting goals and interrelated issues to consider and the situation is constantly changing. Because of this, team structures rarely look identical or some “ideal” form.

The organizational structure of a team is dependent on many factors.

- Project that the team is working on. Projects differ in what is in the design (IPs, FWs, etc.), ownership of portions of the design, and customer requirements for features. These factors significantly alter the validation tasks and even approaches required. In some cases, entire disciplines may be unneeded by a team.
- Team background. Teams and individuals that have been working together for a long time are more productive. Teams usually stay within a discipline, but in some cases will switch disciplines as a team. Keeping teams intact, even across unrelated projects or even disciplines, is very valuable. The history of a team will often explain its current structure.
- Individual skills and desired career path: A validator working in a discipline, or area of the design is more productive staying in that discipline or area of the design. However, the career needs of individuals (both individual contributors and managers) are important as well. Organizing the team differently can create opportunities for individuals.
- Team Size. Each team is around ten direct reports as directly managing more individuals is not effective. Upper-level managers may appear in the organizational charts to have fewer reports as they may only have a handful of team managers reporting to them, but they also have Individual Contributors (ICs) that report directly to them because those ICs are working on cross-team tasks.
- Horizontal cross-project teams. Some activities are organized “horizontally” across multiple project teams (called vertical teams). This allows disciplines that do not have critical mass within a project to allow teams to specialize in that discipline by getting critical mass from multiple projects. Examples include non-simulation model-building²⁶ and the Security Center of Expertise (SeCOE). Teams should strive to be fully engaged with these horizontal teams and avoid replicating their work.

Because of all these factors, teams will have differences to optimally organize and coordinate the work for each project. The examples that follow are not an encouragement to organize your team identically, but rather demonstrate how different factors contribute to team formation.

²⁶ Emulation, FPGA and Virtual Platform models are all examples where work is shared with the project team and the PreSilicon Platforms Acceleration group (PPA) inside SVE which provides services for all PEG projects.

6.4.1 DDG 2016 Example

Below is a discussion of the existing team structure of DDG during early 2016 when the team was working on three major projects; the Broxton (BXT) family²⁷, BaseIA²⁸ and Cannon Lake Desktop (CNL-S)²⁹. The diagram below shows the organization, which we will discuss roughly from top to bottom, left to right. Some of the teams we will discuss as groups.

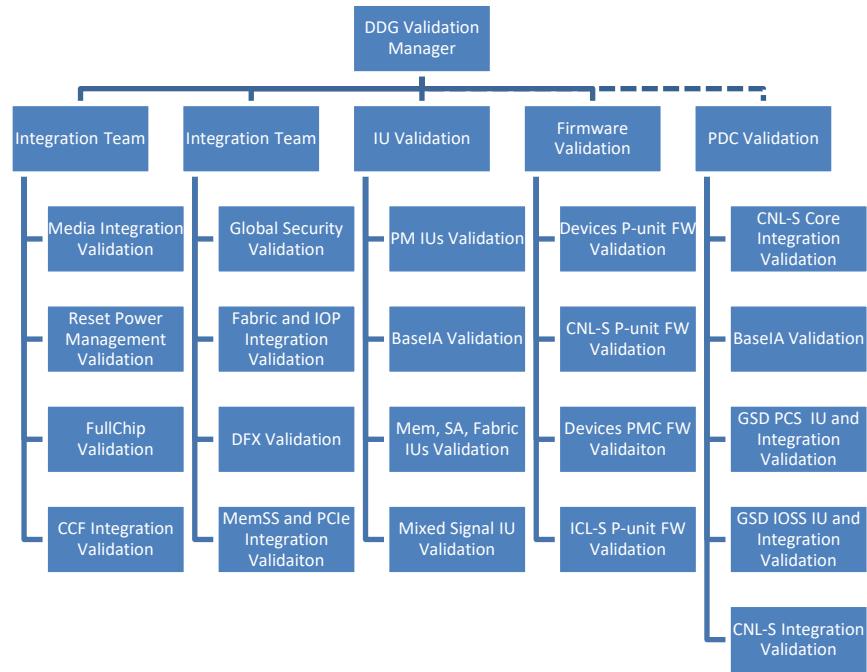


Figure 8: Historic Validation Org Example

6.4.1.1 Integration (and Feature Validation) Teams

Because there are too many teams doing Integration Validation and Feature Validation to put under a single manager, we split them up under two “Integration” Teams. There is a mix of both Integration Validation teams and Feature Validation (and Cross Feature Validation) teams under each manager. The concern addressed is that if all the Integration Validation was under one manager and all the Feature Validation under another, it would result in less collaboration between validators using different approaches.

²⁷ Broxton is a family of SOC projects all on 14nm using Atom cores and with full integration of the “south”, so it was a true single-die SOC. Products in the family owned by this group are Broxton (BXT or BXT-M), BXT-P Gemini Lake (GLK) and Goldsand (GSD).

²⁸ BaseIA is a family of IPs providing the bulk of the “north” as an IP to other customers, both internal and external to Intel for use in a variety of lower-power products.

²⁹ Cannon Lake Desktop (CNL-S, S for “socket”) is a “big-core” Client product without an integrated PCH. It is a derivative of Cannon Lake.

6.4.1.1.1 Integration Validation Teams

The names of the Integration Validation teams generally give away the IP that they are doing Integration Validation on, and are for a mix of CNL-S and BXT products.

- Media Integration Validation – IPU, I-Unit, GT (graphics) and Display IPs. The same team usually integrates GT and Display as they communicate with each other and the IP teams share infrastructure. Having experts knowledgeable in both is beneficial.
- Fabric and IOP Integration Validation – Some of the “uncore” parts of the chip.
- CCF Integration – The Core/Ring/GT interconnect.
- MemSS and PCIe Integration Validation – The Memory Subsystem and PCIe are delivered as IUs and have nontrivial changes for CNL-S.
- Core Integration Validation – The IA Cores. This team also does some Intel Architecture Validation work as needed. This team is in Penang, so it shows up at a different place in the team hierarchy because having local managers is important for both business reasons and successful cross-site collaboration.

6.4.1.1.2 Feature Validation Teams

There are three separate Feature Validation teams that closely follow the Feature Validation portion of their disciplines across multiple projects; Reset and Power Management Validation, Global Security Validation and DFX Validation. DFX Validation combines the Design For Test Validation and the Debug Feature Validation disciplines into a single team as they have always worked very closely together and have some overlap in tools and methodology.

6.4.1.1.3 Fullchip Validation

The Fullchip Validation team is unusual in that it is organized around a level (or two) of the integration hierarchy, working with fullchip or platform level models instead of a specific approach or discipline. The team focuses on several areas for the BXT family and CNL-S projects:

- Feature Validation for features that do not have their own team on Fullchip models
- Cross Feature Validation on Fullchip models
- Integration of critical non-integrated firmware
- Delivery of non-simulation fullchip models (Emulation and FPGA)

Most of these tasks require collaborating with other groups. Cross Feature Validation activities coordinate with other Validators within the organization for content and debug. The team works with FW teams to do Integration Validation and Feature Validation with production FWs. The model builders work with the horizontal PPA³⁰ organization to get emulation and FPGA models created and delivered.

³⁰ Pre-Silicon Platform Acceleration (PPA) group focuses on emulation, FPGA and Virtual Platform technologies to select the best technologies and integrate them into the development and validation environment of products.

The chapters [Architecture Validation](#) discusses the approach in much more details and [Fullchip Validation](#) dives into the unique opportunities and challenges of the last layer of integration before the design is handed off to the Post-Silicon team.

6.4.1.2 IU Validation

In DDG, IP Validation is called IU³¹ Validation. The primary focus of IU Validation is Implementation Validation of an RTL-based block or a combination of blocks that can then be integrated as a single entity (called IU) into multiple products. The goal is to flush out all design bugs inside the IU by emulating the interfaces. An IU is validated using either dynamic or formal technique or a combination depending on how well the microarchitecture lends itself to these techniques. IUs can be entirely digital or mixed signal.

In general, each IU has a validation team associated with it with the intent of developing in-depth experts who can drive the required quality. The exact composition of each team varies based on the complexity of the IU. This expertise has tremendous benefits to both the validators and the entire design org, as described in [Becoming the Microarchitecture Expert](#).

We have intentionally silo'ed all the IU Validation together in part because they share the Implementation Validation approach. Unlike the Integration Validation and Feature Validation teams, we want the IU validation teams to practice interacting with the other validation teams as if they are delivering to an external team. If the IU teams are too closely associated with the Integration Validation teams, it is easy to take shortcuts in communication and quality that will limit the usability of the IU to a specific product. This does not serve us well if a need arises (as it often does) to service multiple products, particularly if they are not owned by the same organization.

6.4.1.3 Firmware Validation

In DDG we have several examples of firmware that are built by the Architecture team³² and integrated directly into our RTL models. Other firmware and software is created outside our organization and is not integrated directly into our RTL models. Those organizations have their own validation teams and their firmware is integrated into our environment as needed by the FullChip Validation team (see section 6.4.1.1.2).

The DDG integrated Firmware Validation teams share both a Validation Approach and a discipline with unique tools, so they are grouped together. The firmware that needs to be validated by our organization changes from project to project, just like the IPs that need to be integrated. To get greater affinity with the design, we organize the validators into groups working on a particular firmware on a particular family of products.

³¹ Because IPG officially owns delivery of all “IPs” at Intel Corporation, the term “Integration Unit” (IU) is used to refer to blocks that are created in groups outside IPG to avoid confusion with IPG-created IPs.

³² For historical reasons, the Architecture team owns microcode, pcode and PMC FW development for our products instead of the Design team.

We currently do not own any microcode validation, but that was a significant part of our Firmware Validation efforts for many projects. Chapter [Microcode Validation](#) talks about our experiences and learnings from that work. The chapter [Embedded FW Validation](#) discusses our current environment, working primarily with power management firmware.

6.4.1.4 Penang Design Center Validation

For well over a decade, the DDG validation and design teams located in Oregon, have partnered with the Penang Design Center (PDC) in Malaysia. While working cross-site has challenges, it is often necessary to keep engineers constantly busy working on revenue-generating projects. In this case, the Pentium 4 projects required more heads than were available in Oregon. The PDC team was finishing a project and were assigned to help on the Pentium 4 line with the Oregon team. Because our teams have worked together for so long, the inefficiencies of cross-site are reduced and we are able to accomplish more together. Naturally, we attempt to keep work co-located as much as possible for efficiency, but many areas are split across multiple sites, largely due to team availability and skills.

The Goldsand team put the Implementation Validation and Integration Validation work of IUs into the same team, unlike Broxton and CNL-S. This increases efficiency for the Goldsand project, but will decrease the efficiency for using those IUs outside of Goldsand. In this case, that is a good tradeoff, as the IUs are not planned for reuse with other customers. This was also how we organized validation in other earlier projects when we did not reuse blocks between multiple projects. This is another example of how project priorities affect organizational structure.

7 Summary

Validation Approaches and Validation Disciplines provide us a broad understanding of the extensive set of techniques to address the increasing complexity and variety within our designs. Teams are formed in part around commonality in these Validation Disciplines as well as the unique properties of the projects we work on, and the skills that we bring both individually and as teams.

8 Future Work

It would be good to talk about how we feed bug escape information from “higher” levels of validation

What about other disciplines like static code analysis and code inspection? Both seem to need some discussion. Pcode has static checks on the code and so does RTL though spyglass/lintra.

Formal Verification section is a bit weak, work with FV chapter owners to clean it up some more.

We do not call out Modelbuild or DA as disciplines. Currently that does not seem in scope because they are both indirect contributors to validating the product, and that is consistent with the rest of the chapters. Long-term it might be nice to include them, but other items are higher priority.

There is no discussion of virtual teams in the team section. Need to explain why we matrix and not re-manager frequently.

There is no discussion of how our group interacts with external groups. These include VTs, VJT and CDG or even business groups. There has been a lot of interest in this topic and it might make a good chapter by itself.

It would be helpful to have a second organizational structure example. Audio seems like a good candidate as it is internally asymmetric, has firmware and 3rd party IPs.

Having people specialize to near-extreme levels is a large company/team benefit. Currently our company does not value this deep specialization as much anymore, and people need to adjust their thought processes and behaviors to match. This chapter is probably not the right place for that discussion.

More “historical color” examples in the early sections would make better reading.

Had an interesting discussion about fabric validation. Although fabrics do not require a lot of validation, they do require substantial collateral and/or expertise because traffic flows through them and other validators need to debug “through” the fabrics to get to bugs. Inside DDG we have not been managing this well because we don’t have good abstractions for the routing logic and have moved the Integration Validation work around from person-to-person too many times to build up a team of people who can efficiently help with that debug process.

9 References

This section intentionally left blank.

The Art of Validation: Chapter 3

The Life of a Project

By: Bob Fisch [and Andrew Gibson](#)

1 Abstract

This chapter provides an overview of how Intel's products are brought into being, assigned to a Design team, built, validated and ultimately delivered to our customers. The chapter begins with a discussion of Intel's Product Overview Proposal (POP) process. The POP defines the product vision, market, capabilities and business opportunities. If sufficient financial and strategic incentive exists for Intel to build the product, the engineering teams engage to refine the schedules and costs. Upon POP process approval, the product moves to an engineering team that operates in accordance with the Platform and Engineering Group (PEG) Product Life Cycle (PLC). The PLC defines the deliverables and receivables of the various teams. This chapter carefully examined the Validation aspects of the PLC in 2 different sections: IP PLC and SOC PLC.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
0.1	11/12/2003	First draft	Bob Fisch	Matt Kupperman / DPG-N Validation MWG
0.2	1/22/2004	Revision of first draft	Bob Fisch	Rick Zucker
0.3	3/26/2004	Updates from Rick's review	Bob Fisch	Rick Zucker
0.4	5/17/2004	Final update from Rick's review	Bob Fisch	
1.0	5/17/2004	Level 2 version. Copy of version 0.4	Bob Fisch	Paul Schwabe
1.1	5/18/2004	Same as 1.0 – fixed page 1 version no.	Bob Fisch	Paul Schwabe
1.2	6/24/2004	Initial revision based on Paul's comments	Bob Fisch	Paul Schwabe
2.0	6/24/2004	template update	Bob Fisch	Paul Schwabe
2.1	3/9/2011	Updated with more recent information	Michael Bair	CCDO Art of Validation WG
2.2	12/7/2011	Updates to move it to Art of Validation	Phil Atkinson	CCDO Art of Validation WG
3.0	2/11/2016	New organization of this topic. Introduce POP and PEG PLC	Andrew Gibson	Michael Bair

3 Contents

1 Abstract.....	55
2 Revision History.....	56
3 Contents.....	57
4 Purpose.....	59
4.1.1 Why do we need this chapter?.....	59
4.1.2 What does this chapter cover?	59
4.1.3 What does this chapter not cover?	59
5 Background Concepts	59
6 The Life of a Project.....	59
6.1 POP Process	59
6.1.1 Introduction to POP Process.....	59
6.1.2 SOC POP Milestones	60
6.2 PEG Product LifeCycle (PLC) Concepts.....	63
6.2.1 PLC Introduction	63
6.2.2 PLC Milestones.....	63
6.2.3 PLC Components	63
6.2.4 The PLC lifecycle.....	64
6.3 PLC Milestones for IP Validation	66
6.3.1 Introduction.....	66
6.3.2 IP Tech Readiness – Validation Activities.....	67
6.3.3 IP RTL 0.0 – Validation Activities.....	67
6.3.4 IP RTL 0.5 – Validation Activities	67
6.3.5 IP RTL 0.8 - Validation Activities.....	69
6.3.6 IP RTL 1.0 Validation Activities.....	71
6.4 SOC Validation Milestones	72
6.4.1 Introduction	72
6.4.2 SOC VAL TR	72
6.4.3 SOC VAL 0.0	74
6.4.4 SOC VAL 0.3	75
6.4.5 SOC VAL 0.5	76
6.4.6 SOC VAL 0.8	77

6.4.7	PLC SOC Val 0.8 Exit Criteria	77
6.4.8	SOC VAL 1.0	77
6.5	Post-Silicon and Stepping Validation	78
7	Summary.....	80
8	Future Work.....	80
9	References.....	80

4 Purpose

4.1.1 Why do we need this chapter?

Validators need to understand the different phases of the projects so they know how to work with teams responsible for delivering to Validation, meet the expectations of their customers and prioritize their own deliverables. This chapter provides an overview of Validation priorities through the course of the product, and attempts to provide the reasoning that underpins the Validation strategy. The description of the PEG PLC defines terminology that Validators will benefit from understanding in their everyday work. Similarly, Validators must understand the POP process to understand the product definition and thereby understand what needs to work in the hands of our customers.

4.1.2 What does this chapter cover?

The focus of this chapter is the description of the Pre-Si Validation activities that take place during the progression of the project lifecycle. The reader will learn when the Validation team writes its first test, writes the testplans, and, for example, shifts focus to validation of global features such as Power Management. As a secondary focus, the document also attempts to describe the activities that interact with Pre-Si Validation. For example, Validation relies on RTL delivery, SOC Integration Validation relies on the underlying quality of the IP Validation, SW Validation and Structural Design (SD) rely on the quality of validation to ensure they meet their schedule commitments.

4.1.3 What does this chapter not cover?

This chapter does not cover a detailed list of activities the Validation teams do during the different project phases. For that information, please see the chapters for each individual Validation discipline.

5 Background Concepts

6 The Life of a Project

6.1 POP Process

6.1.1 Introduction to POP Process

The POP (Product Overview Proposal) process is used to merge input from stakeholders (including Intel's customers, Marketing, Finance, Competitive Analysis, Architecture and Design Teams) to define the external requirements of a future product. These product external requirements are referred to as Landing Zone features. Intel uses similar but slightly different POP processes for the development of SOC products versus IPs that we implement ourselves. We will describe these in separate sections

Validation engagement in the POP process is typically limited to the project Validation manager who interfaces with the Validation team to understand the complexity and costs of proposed features. Understanding the POP process is valuable for all Validators because it is important to know why Intel is making an investment in a project. The Validation strategy would probably be very different between two projects where, for example, the focus was to improve performance versus to reduce the product cost. Understanding the POP milestones and terminology will help Validators understand why we are building a product.

6.1.2 SOC POP Milestones

6.1.2.1 SOC POP Milestone Introduction

The SOC POP (Product Overview Proposal) milestones, known as POP L1, L2, and L3, take a product from business conception (L1) to architectural definition (L2) to an approved/funded project on the official Intel roadmap (L3). This section describes each of these milestones and attempts to explain the inputs and outputs of each milestone alongside how the Validation team interacts with this process.

6.1.2.2 SOC POP L1

The theme of SOC POP L1 is to lay down a vision for a new product based upon business, usage, customer requirements and/or technology opportunity. Product planners and business teams collect feedback from the customers of our existing products and they provide input to the process with requests for features such as increased performance, lower power for new form factors, or a reduced Bill of Materials (BOM) to enter new market segments. Some of the new technology requirements are predictable such as the need to support a memory technology at the required cost/performance in the time window when the product is in the market. Corporate strategic initiatives drive other technologies and features into our products. For example, new features to support a secure/trusted computing model are a requirement to enhance the Intel brand.

In practice, the Marketing/Technology and Planning teams generate many great ideas for a new product, but these must be balanced against what the market is willing to pay and what the engineering teams can accomplish. While the volume of predicted sales increases with the addition of new features, so does the time/cost to develop and manufacture the product. The outcome of SOC POP L1 is a document that balances these considerations. The document includes the following assessments:

- Define the Product Vision, Value Proposition, Preliminary Usages, Business Opportunity, Customer Requirements, Key Performance Indicators (KPIs), Bill of Materials (BOM) and any Strategic Imperative.
- Complete Market Research, Customer Feedback and Competitive Analysis.
- Identify an SOC intercept and schedule
- A Landing Zone (LZ) specification that defines the IPs required by the features of the proposed product, and a feasibility analysis for any new IPs.

- Finance Sanity Check (e.g. product cost, volume/ASP, cost targets) and a Business Unit (BU) commit to secure resources needed for POPL2 completion.
- A preliminary estimation of Business Segmentation, Schedule and Resources. Note: The Business Segmentation attempts to answer whether the product is for the Devices, Client or Server markets. This is a more complex question than it first appears for products used across segments.

The Validation Project Manager engages with the POP L1 process to provide feedback on schedule and resource requirements. Validators contribute their technical feedback to the Validation manager.

6.1.2.3 SOC POP L2

SOC POP L2 consumes the data generated by the L1 Milestone and makes decisions as to what features are included in the product. The fundamental outcome of the milestone is that the Engineering Teams have a framework in place to enable product development. To a first approximation, SOC Validation TR (section 6.4.2) runs concurrently with SOC POP L2. The major features of the SOC are usually defined before the milestone completes, and the engineering team needs to make a speculative start to identify unforeseen risks in these areas. It is a challenge for the Validation team to start speculative work on non-approved products (because they are busy working on approved ones).

Some of the key outcomes of the SOC POP L2 milestone include:

- Identify customers and/or end-users for the product. End-user value proposition comprehended based on product's inclusion in Intel platforms.
- Value proposition for the product aligned to the LZ, and all SOC related LZ Requirements (IP/HW/FW/SW) closed. SOC architecture is placed under change control from the platform perspective.
- All SoC LZ requirements responded to by engineering team. This requires the SOC engineering teams to comprehend the IP schedule/commitments and the cost/schedule for integration.
- Engineering Resources secured through A0 T/I.
- Updates estimation of product cost, schedule and resource needs are aligned to the graded Landing Zone.

6.1.2.4 SOC POP L3

SOC POP L3 is where a product gains an official placement on the Intel product roadmap (The product becomes Plan of Record (POR)). The product features and schedule may be discussed with Intel customers (under strict legal non-disclosure arrangements). To exit POP L3 a detailed implementation plan of the program through the development and production phases is approved and funding is allocated.

- All SOC related LZ Requirements (including related FW/SW) are placed under strict change control with engineering responses documented. SOC architecture is under strict change control.
- The overall implementation plan is put into place and approved including schedule and success metrics.
- Program risks are identified and only med and low risks and knowledge gaps exist.
- Program financial analysis completed and approved. Program resources & funding approved through PRQ per desired POR.
- Customer communication plan approved.

At the close of SOC POP L3 the project development typically moves into rapid execution.

6.1.2.5 POP IP Planning

A fundamental assumption of the SOC POP process is the pre-existence of IPs with features required by the product. In reality, several IPs are co-developed on the same schedule as the SOC. Nonetheless, the POP planning for the creation and delivery of an IP takes place in the five stages described in the table below. IP0, IP1 and IP2 are planning stages where the IP engineering team engages to set the schedule and engineering costs. The initial development of the IP takes place after IP2, and IP3 is when the IP team commits to deliver to a product on a specified schedule. When a later product wants to reuse the IP in substantially the same state, they engage with the IP team to close the IP3 Milestone.

Milestone	LZ	Objective	IP Coverage
IP0	Estimate	Add to IP roadmap as under investigation (FW, SW, & SIP). Start of longer lead TR work; pathfinding	IP family
IP1	Establish	Create cross BU agreement on IP timing and requirements (FW, SW, and SIP). All TR activities identified	IP Family
IP2	Freeze; change control start	Lock LZ so IP delivery team(s) can commit to the broader roadmap feature-set (FW, SW, SIP). TRs complete. Requirements go in to change control. 1 st pass of customer feedback. Enables SOC POP L1	IP Family
IP3	Commit	Establish as POR (SIP & FW/SW covering SIP features). Commits to headcount, funding, features, and schedule. Enables SOC POP L2	Product Specific

IP 3.2	Commit for SW/FW non-HW specific	FW/SW commits for HW independent features. Timing TBD.	Product specific
--------	----------------------------------	-----------------------------------------------------------	------------------

6.2 PEG Product LifeCycle (PLC) Concepts

6.2.1 PLC Introduction

The PLC is a set of requirements, durations and handoffs needed to guide predictable product development. It is a roadmap of deliverables and receivables that attempts to bring together the ingredients of the design at the right times to produce a product for our customers on a predictable schedule. The PLC is broken into components such as RTL and Validation, and milestones are defined on a schedule that supports timely handoffs between components.

6.2.2 PLC Milestones

Most PLC Components divide activities into 3 or 4 milestones. Projects with shorter durations may have fewer milestones, and conversely longer projects may have more milestones. The Milestones are typically named 0.0, 03, 0.5, 0.8 and 1.0, although a few of the components have milestones with other numbers. In almost all cases, 1.0 means all planned activities are completed.

6.2.3 PLC Components

The key PLC components that interact with Pre-Si Validation are described in the table below. This table, extracted from the 1.3.4 release of the PLC specification, describes the themes of the milestones assigned to each of the components. The Post-Si Validation Component is missing from the PLC Milestone schedule since it was only recently defined (and has not made it into the literature).

Components	Description
HAS	High-Level Architecture Specification. This document is typically written by the Architecture team, and is used to describe new or incremental features added to the design.
IP RTL	The IP team delivers RTL to the SOC team. The IP RTL milestones include validation and back-end design requirements
SOC RTL	The SOC RTL team integrates IP RTL and performs quality checks at the SOC/Integration level
SD	Structural Design. The Structural Design activities consume SOC RTL, synthesizes the logic into gates, and produces the layout.

SOC Val	The SOC Val team consumes SOC RTL and validates the system-level functionality.
PSS	The Pre-Silicon Software activities consume the validated SOC models to enable system software including IP Firmware and BIOS.
MFG	Manufacturing develops the test-content patterns and flows required for elimination of defective parts.
Post-Si Val	Post Si Validation of all HW/FW features to support B0 tapein, and then PRQ. Enable and validate SW stack. Enable and manage customer engagement.

Pre-Si PLC: Overview of Category Themes

REVISION 1.3.4

	0.0	0.5	0.8	1.0
HAS	<ul style="list-style-type: none"> HAS 0.2 - Provide information for projects to start planning resourcing needs and timelines 	<ul style="list-style-type: none"> Enable all features for Rev 0.5 RTL Fully define an IP block to allow FED to begin 	<ul style="list-style-type: none"> Complete specification to enable Rev 0.8 RTL 	 Change Control
IP RTL	<ul style="list-style-type: none"> Build initial full chip validation environment, with a shell at the IP level 	<ul style="list-style-type: none"> Code features and functionality per HAS 0.5 pertaining to DFX, PSS 0.5, and power management Code all interfaces 	<ul style="list-style-type: none"> Code all features 100% (including DFX and PM) with all test plans written Full design convergence with IP level constraints with guardbands using fully featured RTL 	 Tape-in ready quality
SoC RTL	<ul style="list-style-type: none"> Code and connect hierarchy and floor-plan Integrate all negotiated IPs 	<ul style="list-style-type: none"> Ensure 100% of Globals are feature complete (including fabrics) per HAS 	<ul style="list-style-type: none"> Clean all RTL flows Drive to 100% feature complete RTL 	 Predictable tape-in execution on final RTL
SD	<ul style="list-style-type: none"> Conduct initial floor plan and die size estimations Provide initial constraint definition and flow verification 	<ul style="list-style-type: none"> Run first full cycle of all partition and fullchip flows on RTL Have layout-based timing quality at SD 0.5 	<ul style="list-style-type: none"> Converge full SD partition and fullchip with fully featured RTL Ensure all tools and flows are functional, run, and results are reviewed 	 Design supports tape-in criteria
SoC VAL	<ul style="list-style-type: none"> Approve validation strategy and install core validation infrastructure 	<ul style="list-style-type: none"> Complete basic validation on 0.5 IPs Pass all Globals acceptance tests (chassis, DFX, PM, PSS 0.5, emultn.) 	<ul style="list-style-type: none"> Complete all freeze gating tests 	 All tape-in gating tests completed
PSS	<ul style="list-style-type: none"> Create pre-Si SW development Plan Define IP/FW dependencies for each UC/OS Track 	<ul style="list-style-type: none"> Enable pre-Si systems (HFPGA, SLE, n-1) with basic secured boot for SW/FW development and deployment 	<ul style="list-style-type: none"> Finish all HW/SW co-validation (have 100% of IPs ready) <i>Note: this is a freeze gating milestone</i> 	 SW quality checkpoint: Go/No-Go Gate for A0 TI
MFG	<ul style="list-style-type: none"> Content porting and creation planning Final PHI/coverage targets established Binning targets and Qual plans documented 	<ul style="list-style-type: none"> MFG test content validation and coding activities begin HVM reset validation complete Binning targets established 	<ul style="list-style-type: none"> Full HVM content coding activities enabled HVM reset passing, released in all modes Tester hardware design complete 	 Full silicon test content finalized; Test Program shell complete

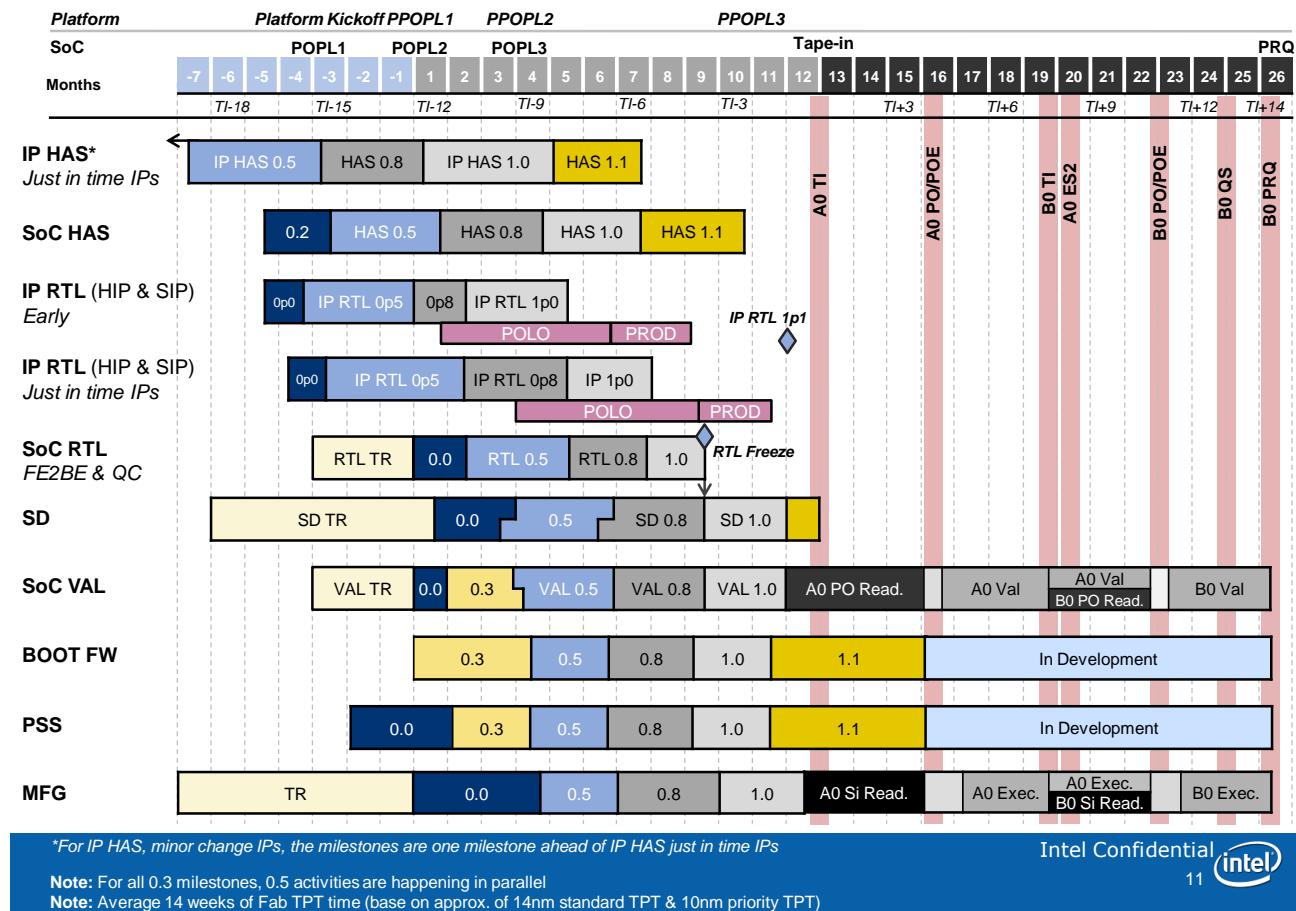
6.2.4 The PLC lifecycle

The prior discussion of PLC Milestones and Components is the precursor to understanding the lifecycle of the standard 26 month project. Intel will undertake longer and shorter projects, but this lifecycle is the reference from which deviations are best understood. The first thing a good Validator will notice about the 26 month project is that it takes 33 months to complete. The 26 months is the duration that commences with the closing of POP L2, and the associated funding of the product through to the PRQ (product release qualification) of the project. PRQ means the part

has been certified as having met the Intel standards for a product. The product launch typically follows the PRQ date by a few weeks after the product has been manufactured in the volumes needed to meet customer requirements.

Base PLC overview: 26 month POPL2 - PRQ

REVISION 1.3.4



6.2.4.1 Base PLC IP RTL Schedule

The IP RTL schedule includes a mix of RTL and Validation milestones and deliverables. These are described in section 6.3 of this chapter. IP teams may take internal milestones with finer granularity (often called IP Val milestones), but these are not official PLC milestones. The IP RTL 0.5 schedule keys off the completion of IP HAS 0.5. The prior IP RTL 0.0 schedule runs concurrently with IP HAS 0.5, and is when the Validation teams interact with Architecture to make sure the HAS contains the degree of specification required by Validation. The IP RTL 0.5 feeds into SOC RTL 0.5, IP RTL 0.8 into SOC RTL 0.8 and IP RTL 1.0 into SOC RTL 1.0. The IP RTL 1.0 drop is 5 months from tapein: 2 months are needed to integrate the RTL and complete SOC quality checks, and the final 3 months for SD 1.0 and full-chip timing convergence.

6.2.4.2 Base PLC SOC VAL Schedule

The SOC Val Milestones have a complex relationship with the SOC RTL milestones upon which they depend.

- SOC RTL and VAL TR run in parallel. When the RTL repository opens (12 months from tapein) the 2 teams work together to integrate the IP RTL 1.0 drops and maintain the health of the model.
- At month 1.5 the SOC RTL and Val teams work together to complete the IP RTL 0.5 integration. The expectation is that the IP integration completes after 4 weeks at month 2.5 of the schedule. The SOC Val team use the next 4 weeks to complete SOC Val 0.3. This is where the Val team brings the new IP out of reset and drive the health to the point where a basic DOA test is passing. At this point the RTL model is consumed by the SD and PSS 0.5 milestones. A passing test is an indication that clks, reset signals, sideband messaged and firmware protocols are sufficiently healthy to indicate that the first SD model can be built for a working design. Similarly, the FPGA models used by the PSS teams want to build a model with the expectation that it will come out of reset.
- The SOC Val 0.5 milestone completes 1 month into the RTL 0.8 milestone. The expectation is that SOC Val 0.5 health is first gained using the IP RTL 0.5 drop and then repeated within the same milestone using the 0.8 IP RTL. Completion of this activity ends SOC Val 0.5, and the validated database can now be consumed by the PSS and SD 0.8 milestones.
- SOC Val 0.8 completes by running tests on the SOC model containing the 1.0 IP RTL drops. Successful completion of SOC Val 0.8 corresponds to freeze of the RTL model, and SD uses the database for final synthesis for tapein.
- The post A0 tapein PLC schedule will be discussed in chapter 9 of this document

6.3 PLC Milestones for IP Validation

6.3.1 Introduction

The intent of this section is to describe the validation lifecycle of an IP and provide insight into the activities a Validation team uses to reach the objectives. By contrast, the PEG PLC milestone definitions describe what must be accomplished for a successful handoff to an SOC team, but not how it is accomplished. This section is written using the terminology of the PLC, where the five milestones are labeled IP RTL TR, IP RTL 0.0, IP RTL 0.5, IP RTL 0.8 and IP RTL 1.0. These milestone definitions cover the development of the RTL and the validation, whereas the SOC milestones are separated for RTL and Validation.

This section of the document is written for the development of a new IP as opposed to the modification of an existing IP to support the requirements of a new project. For pre-existing IPs the PLC expectations are that new features will be validated in accordance with the milestone definitions, but existing features are delivered 1 milestone ahead of new features. For example, 0.8 validation quality is expected in the 0.5 drop for existing features.

6.3.2 IP Tech Readiness – Validation Activities

During tech readiness, the Validation team engages with the Architecture team to discuss, understand and define the microarchitecture of the IP. This engagement enables Validators to appreciate the decisions and tradeoffs that led to the final definition of the IP. The insight they gain from this process is valuable to Validation because it exposes weaknesses that require extra focus from the validation effort. The Validation team must understand what is written in the IP HAS, and make sure everything they need to validate the IP is documented.

The TR phase is where the Validation team does its IP planning and readies its technical infrastructure. Items specific to Validation that need to be ready include model build tools, Test Environments, coverage tools, test generators and coverage collection tools. Some of these tools can take a long time to create, so identifying the requirements needs to happen very early. For example, an IP with thousands of control registers will need an automated way of validating the functionality against the specification, and the team will need to develop the infrastructure or learn an existing tool during TR in preparation for validation execution.

The Validation team plays an important role in defining the microarchitecture. When ideas are proposed about microarchitectural possibilities, experts from Validation weigh in on the merits of these ideas and evaluate whether there are cases to consider that might have been overlooked. This may be something as simple as requesting the reuse of an existing standard interface or protocol. On the other extreme, when a large new protocol is proposed, the Validation team may write a high-level formal model to flush out problems before RTL coding starts.

6.3.3 IP RTL 0.0 – Validation Activities

The PLC specification does not call for any validation as part of IP RTL 0.0. The RTL team delivers a shell for the IP with the pin connectivity to the SOC team. It makes sense for the Validation team to review the shell to make sure all interfaces are in place and have the correct signal names.

6.3.4 IP RTL 0.5 – Validation Activities

6.3.4.1 Planning and Organization

During IP RTL 0.5, the Validation team completes its first pass at understanding the microarchitecture and enables Design to make forward progress. Testing done to demonstrate basic functionality of the RTL is referred to as exercise (as opposed to validation). Both directed tests and directed random tests are used for exercise. A staging plan co-created by the Design and Validation teams is used as a blueprint for much of Validations exercise activities during IP RTL 0.5. Indeed, recent projects have formalized the relationship between Architecture, Design, and Validation during this stage by creating *Virtual Teams* (VTs). The VTs own features within the staging plan from implementation to exercise to turnin to the RTL repository. Within the VT, exercise plans are written and executed. These exercise plans dictate the validation activities during IP RTL 0.5 that allow the VT to claim high-level functionality for the RTL.

6.3.4.2 Testing

Directed tests are written to set up specific situations of interest to show that the RTL generally does what it is supposed to do. The focus is on breadth over depth, although a particularly risky area can be cause for more detailed exploration to expose any fundamental but subtle bugs early. Self-checking is one approach to use for directed tests to demonstrate that the conditions of interest are actually being set up. If your project is starting with a mature TE and a reasonable level of checking, skip directed testing in place of more efficient directed random testing.

Directed random testing during exercise provides a higher level of stress for the RTL than directed tests. This testing is an opportunity to get the test generation environment and tools working. The directed random tests run against the model during IP RTL 0.5 are of relatively low complexity, since basic functionality needs to be established before investing resources in finding bugs that are more interesting.

Validation creates turnin-gating test suites as part of the VT activity – RTL code cannot be turned in without an exercise test being included in the turnin-gating regression. The purpose of these tests is to prevent change (RTL or TE) from entering the model that lowers the quality of the design. The pass rate of the regressions outside of the model should be largely protected in the sense that a bad turnin should not drop the full regression health by more than a few percent.

The test failure rates are high during IP RTL 0.5, and resources do not exist to debug all failures. Triage takes place to limit debug to those failures that will do the most to improve model health. This is very subjective, but the guiding principle is to do what is most likely to help Design make forward progress.

6.3.4.3 Bugs

Since the RTL is immature at this stage, a large number of bugs exist. There are differing attitudes about the amount of bug filing that should be done during IP Val 0.5. One view is that if a feature is not completely coded or has had no testing, then the overhead of filing and dispositioning bugs can be counterproductive to both the Designer and the Validator. The opposing view is to file all bugs, regardless of feature maturity when found, because this ensures a continual flow of communication between Validation and Design. Furthermore, the documentation of these bugs is valuable and merits the associated overhead.

One approach used is that bugs in a feature are not filed until exercise of that feature is complete, after which time bugs in that feature do get filed. Within a VT, the Validator communicates constantly with the Designer, multiple times per day, to inform them of what seems to be working and what seems to be broken and to plan their next steps. In this model, bugs in the code are not filed within HSD, although the VT may utilize a lightweight tracking tool. This approach tries to strike a balance between documenting bugs found and efficiently getting the basic functionality implemented. Even with this approach, “interesting” bugs are filed even if exercise is not yet complete in that area.

For bugs that are filed and fixed during IP RTL 0.5, bug validation is very lightweight. Generally, we are satisfied with rerunning the failing test and seeing it pass. Interesting bugs get more scrutiny before being marked validated. The amount of such scrutiny depends on the nature of the bug.

6.3.4.4 Validation Infrastructure

The IP Validation team must make investments during IP RTL 0.5 to be efficient in future milestones Examples include:

- Test management – how to manage versioning, how to determine the version of each test to run on each model;
- Random template and weight file management:
 - how to manage updates
 - how to determine the versions of the templates and weight files to use for each model
 - how to use templates and weight files created by others,
- Coverage collection methodology;
- Document archiving – where to put documents such as exercise plans, testplans, checklists.

6.3.4.5 PLC requirements for handoff to SOC Integration/Validation

SOC teams have specific requirements for the IP RTL 0.5 drop that may fall outside of what the IP Validation team considers their highest priority (their highest priority is typically finding bugs). The following list of deliverables was extracted from the PLC 1.3.4 requirements, and offers explanations as to why these deliverables are required.

- Basic exercise of all interface-level functionality. This includes power, resets (boot), clocks, DFx and security as well as data interfaces. This is probably the key requirement for the SOC team. Validation of the IP at the integration level must demonstrate the IP boots correctly, responds to PM requests, control registers can be read and written, and basic data flows are functional. Without this level of functionality the IP does not meet the spirit of IP RTL 0.5
- An integration level validation environment with full ACE support including test islands. The SOC team must be able to reuse relevant parts of the IP validation environment (checkers, tests, monitors). This requires the IP Test Environment to support an SOC mode where the BFM's used at IP-level are replaced by real RTL at the integration level.
- IP integration test plans and tests should be provided to the SOC team. When these tests pass at the integration level there is a good indication that the IP is integrated correctly and the system fabrics of the SOC support the IP.
- Basic performance tests must be delivered so that benchmarking can take place at the integration level.
- RTL/Validation collateral passes emulation/FPGA build checks to enable SOC emulation/FPGA builds. Significant delays to the program are incurred if the SOC team requires additional RTL drops and integrations to meet this requirement.
- Internal forces must be minimized and documented

6.3.5 IP RTL 0.8 - Validation Activities

6.3.5.1 IP RTL 0.8 Overview

The fundamental deliverable from the IP RTL 0.8 milestone is a design database that is RTL complete with validation infrastructure in place and working. Although the PLC does not make the requirement explicit, a good way to think about this milestone is that 80% of the final testing

and or coverage is done. Falling short of this makes it very difficult to avoid large RTL changes as part of the 1.0 drop when Validation must be 100% complete. Any large changes in the 1.0 IP drop to SOC require greater effort for back-end timing convergence and creates an associated slip to the project schedule. In addition to completing the RTL coding for remaining features, the IP Design team will make changes in the design based on feedback from back-end timing convergence efforts. While these RTL changes may only have a minor effect on internal logic, they can introduce new bugs. Even a simple signal rename is a source of instability, since such changes often break TEs, coverage monitors, and checkers.

6.3.5.2 Test Plans

Validation writes testplans and conducts thorough reviews of these testplans with other Validators, Designers, and Architects. The testplans are a deliverable to the SOC integration team so they have an opportunity to review validation at the IP-level and can make effective decisions as to what should be re-validated at SOC.

6.3.5.3 Testing and Coverage

Testing emphasis leans heavily towards random or directed random testing, although a small number of areas might continue to rely on directed tests.

Coverage collection can begin once coverage monitors are available. Validators utilize random test generators and weight files to generate tests for collecting coverage. Coverage analysis helps identify holes in testing. Holes can be filled in many ways: creating new weight files, improving the test generators and test environments, writing injectors, running tests from other units or clusters, or using coverage from other Validation teams.

6.3.5.4 Driving Model Health

During IP RTL 0.8, it is imperative that all failure signatures be debugged. All bugs are filed, regardless of whether the bugs are in the RTL or in the checkers. Historically, the bar for validating bugs rises in this phase of the project. Initially, it may suffice to rerun the failing test. By the end of the .8 milestone, validation is much more thorough requiring actions such as creating a test that sets up the failure scenario and demonstrates correct functionality, running a batch of directed random tests targeting the failure area, or possibly writing new testplan entries and coverage monitors to gain additional coverage in the logic associated with the failure.

6.3.5.5 Validation Deliverables to SOC

The IP 0.8 validation deliverables specified by the PLC are:

- 80% functional coverage. The implicit assumption is that breadth validation is complete, but some corner case and deep conditions will be covered as part of the 1.0 milestone.

- Validation environment collateral fully functional. The intent of this requirement is to enable validation of all flows at the SOC level, and as such requires BFM_s to be 100% complete and all checkers coded.
- RTL/FW functionality must support secure boot flows. Testing performed utilizing SW drivers supporting PSS 0.8 features. These requirements are typically mandated by teams enabling software flows on the SOC FPGA model.
- 0.5 Deliverables are assumed to persist as part of 0.8

6.3.6 IP RTL 1.0 Validation Activities

IP RTL 1.0 is the handoff of the fully validated RTL to the SOC team. The Validation team makes every reasonable effort to ensure the IP design does not contain bugs. This IP is consumed for final back-end synthesis, and the schedule does not permit anything more than small design changes that can be made manually to the netlist. The IP Validation team's responsibility is to drive the remaining coverage, review the testplans for completeness and make certain late bugs are carefully validated.

6.3.6.1 Testplan Reviews

Once a testplan has reached its intended goals (coverage targets or tests written and run), a thorough review is usually held. This review is held with other Validators, Designers, and Architects to decide if more testing is needed. The primary focus is on areas with low coverage or low testing, although other areas of interest – historically buggy areas, newly implemented features – are also examined. A checklist is used to document these areas of concern for the review. Upon the completion of any ARs that arise from the review, the area is deemed done.

This rigorous review process confirms that each Validator has done due diligence to validate her piece of the design and has received a stamp of approval from Design and Architecture. The completion of all these reviews is the primary evidence Validation uses to assert that the RTL is of extremely high quality functionally.

6.3.6.2 Bug Validation

Bugs uncovered at this late stage of the project are, by definition, hard to hit (or at least they should be). Finding a bug provides a data-point indicating that test generators are not as effective in this area of the design (or that the tests just became effective), and that bugs are in existence where validation tools are weak. Finding a bug at this stage of the project requires the Validator to contemplate why it was found now (what changed in the environment to expose this bug). The Validators must convince themselves the tools are good enough to provide effective validation of the fixed RTL. Additional coverage monitors are added to make sure the tests are reaching the conditions where the bugs were found.

6.3.6.3 Evil Validation

Although Validators should always be thinking of evil ways to tease out more bugs, the focus during most of the project is on getting the testplan written and executed, including lots of debug. Generally, resources are limited for expanding evilness, but in certain cases it makes sense to invest in writing new injectors, writing new checkers, and otherwise exploring interesting boundary cases. Towards the end of the project, depending on the progress of Design, there may be increased opportunity for evil exploration. If such opportunity arises, Validators do spend some time exploring the boundaries of the design for lurking bugs. However, such activity is limited if it jeopardizes the primary task of delivering a quality IP on schedule.

6.3.6.4 IP 1.0 Validation Deliverables to SOC

There are no specific additional deliverables to the SOC for IP 1.0. The presumption is for the 1.0 to be incremental change over the 0.8 with the remaining bugs found and fixed.

6.4 SOC Validation Milestones

6.4.1 Introduction

The SOC Validation Milestones receive their own designation within the PEG-PLC framework (TR, 0.0, 0.3, 0.5, 0.8 and 1.0). From the PLC perspective the key inputs to the validation milestone are IP delivery and SOC RTL (specifically, the RTL team is expected to integrate the IP into the SOC repository so that Validation can do its job). The SOC Validation team is responsible for meeting milestones that allow the SD team to drive back-end convergence with high confidence in the RTL quality, and for enabling the PSS team who rely on the functionality of specific flows.

This section of the document is written with reference to the validation activities of a SOC project with significant scope. Projects with smaller scope where, for example, only one of two non-intrusive IPs are changed will likely eliminate some of the milestones based on the premise that they are re-using methodology and validation collateral from the parent project.

6.4.2 SOC VAL TR

6.4.2.1 Define the Validation Strategy

As an outcome of TR, the SOC Validation Team must set its validation strategy for the project. To set the right strategy requires an understanding of the methodologies used on the prior project (most SOC designs start with a prior generation product). The strategy for the current project also requires an assessment of the product's schedule and risks, along with an analysis of the features/IPs that are changing. This information can be analyzed to set the right strategy and drive the methodology changes with the largest return on investment.

Validators gather information from the SOC Validation team on which this SOC is based, and from the IP providers who can provide insight into the schedule and risks in their area of expertise.

In many cases, Validators have good contacts with the engineers who work in their area of responsibility within the other teams, and they have some degree of awareness of the different approaches attempted in prior generations. In this case, the Validator needs a few meetings with their counterparts to understand the methodology details, and the organization of the collateral. The next step is to review the testplans, tests, checkers, BFM_s, and then run and debug some tests as a way of ramping expertise. In situations where the SOC validation effort is not familiar, the Validator may need to perform a tour of duty for a few weeks where they learn by doing the validation alongside other experts. Validators also meet on a regular cadence with the Integration Architect responsible for each IP/feature. The Validator is responsible for a thorough review of the HAS.

The overall validation strategy requires the team to look at each of the SOC validation disciplines and optimize globally. The prior project may have had good reasons for supporting a dozen or more integrations DUT_s, but the overhead cost may make such a choice not appropriate for a project with lower scope. Similarly, the TR process must set a strategy for what is to be covered in emulation versus simulation based on the specifics of the project and the availability of resources.

6.4.2.2 Validation Management TR

While the PEG PLC provides high-level guidance for what needs to be done in each milestone, the TR phase is where the individual projects turn the guidelines into concrete goals. The details of a specific project may require some validation to be completed on a faster schedule. For example the global reset validation team needs to deliver test templates for new reset flows before the features can be tested by individual IPs. A staging plan is built to meet schedule with the dependencies identified. Progress in the project is measured by execution to the staging plans as well as technical indicators such as pass rates and coverage. The technical indicators and associated goals are established as part of TR.

6.4.2.3 Interaction with suppliers, customers and external teams

The Validation team engages with their suppliers to make sure validation requirements have been communicated. For example, the RTL team needs to be aware whether the Validation team plans to maintain global power management functionality throughout the project. If so, the RTL team needs to make sure power management flows and firmware code is delivered prior to when they plan to turnin a new IP. In another example, the Validation team must communicate to the IP providers and the RTL teams regarding the requirements for FPGA and emulation models (coding guidelines or LINT rules that must be met must to enable these platforms).

The Validation team must also understand the requirements of its customers and partners. For example, the performance validation team often needs to run benchmarks with specific DDR frequencies and configurations, and so these must be enabled with the appropriate priority.

The Pre-Si Validation team often forms a strong partnership with the Post-Silicon Validation team. This collaboration can range from Post-Si Validators undertaking a tour of duty in critical areas, to forming a joint Validation team.

6.4.2.4 Methodology Improvements

The TR process often identifies opportunities to improve validation infrastructure, and these are best implemented early in the project before change jeopardizes the schedule. The changes can vary in scope from, for example, modification to the gating regression based on the needs of the new project to a major recoding of the test environment to support a fundamental shift in Intel methodology. Validation is under pressure to make improvements as quickly as possible so changes are not disruptive when the RTL is changing

6.4.3 SOC VAL 0.0

The transition from TR to SOC Val 0.0 takes place the moment the RTL model is enabled for the new project. The integration validation team takes on the tasks of bringing the validation infrastructure to life, and working with the RTL team to prepare for new IPs or features to enter the model.

6.4.3.1 Infrastructure Readiness

The immediate priority in SOC Val 0.0 is to enable the validation environment. The activities typically include:

- Importing and enabling infrastructure not contoured within the database inherited from the parent project. Examples include testplans, tests, testlists, indicators and coverage monitors.
- Enabling the full regression suite and driving the pass-rate to the same level as the parent project to establish the baseline health.
- Building the initial emulation model and enabling basic tests

6.4.3.2 Preparing and enabling new RTL

In reality the health of IP RTL 0.0 drops varies widely. The RTL may be a pinlist associated with an empty hierarchy, or may have been modified from a prior project to support the chassis requirements and global flows of the new project. The Validator needs to understand what is needed to maintain the health of the global flows (reset, PM etc.) when this IP is integrated to the model. For example, the gating regression will likely include a test that brings the SOC out of reset. To prevent the new IP from breaking the reset flow, many things will need to happen: the system fabric (and associated routers) must be hooked up correctly, changes to the PMC/Punit and associated Firmware must be made to support the new IP, support for new fuses must be coded, and clock connectivity and hookup to relevant power domains must be done. The Validator needs to understand the ingredients required by these flows, and make sure the constituent pieces are validated for this functionality.

6.4.3.3 Additional PLC Requirements

The PLC adds the following specific exit requirements for SOC Val 0.0 beyond the themes mentioned so far:

- PSS Acceptance Tests defined in Collaboration with Platform Team with minimum required as Basic Boot.
- Performance test cases specified to meet performance LZ requirements.

6.4.4 SOC VAL 0.3

The key outcome of the SOC Val 0.3 milestone is the turnin and enabling of the 0.5 IPs with a basic test. The significance of the milestone is that it is an indicator to other teams that the RTL is in-place and is alive. This is an ideal starting point for the SD work and building of FPGA or emulation models. In the case of the model builds, the Val 0.3 milestone is the first opportunity to build the models and expect them to be functional.

6.4.4.1 Enabling Validation Collateral

Unfortunately, when a new or updated IP is added to the RTL model the validation infrastructure requires significant effort to enable. The Test Island (where the Test Environment interacts with the IP) often requires many iterations to compile successfully as different paths/hierarchies are resolved. New validation collateral (such as BFMIs) that once worked flawlessly may require updated versions of other collateral, and the problems begin to grow across the chip. In the ideal state, the IP integration validator has the support of a TE expert, an ACE expert and domain experts who have a breadth of knowledge to debug the gating tests. More often than not, the integration validator needs to arrange time to meet with these experts and have them assist with the debug after much of the initial triage is complete.

6.4.4.2 Feedback to IP Provider

While the trauma of the integration is fresh on the mind, the Validator must provide feedback to the IP team about what worked well and what needs improvement. The IP teams need to know when they are doing a great job providing reusable collateral and when their validation approaches allow basic bugs to escape to the integration level.

6.4.4.3 PLC Requirements

The PLC requirements for Val 0.3 focus on the boot IPs. In general, it make sense to prioritize the boot IPs, but the SOC Validation team must always focus on all IPs because that is the correct handoff to the SD team. The PLC stipulated the following requirements for SOC Val 0.3 exit:

- Exercise basic boot and CPU reset. Read and Write relevant registers in boot IPs.
- Build the emulation model & execute Basic Boot/CPU Reset flows.
- Exercise Basic Boot/CPU Reset using RTL 0.3 model in power-aware (UPF) simulation.
- Add/Maintain Emulation & FPGA RTL gating model builds and regressions.

- Maintain gating regressions for IPs/Globals that are not changing from the parent project.
- Full chip single socket idle latency tests added to gating regression.

6.4.5 SOC VAL 0.5

The essential theme of SOC Val 0.5 is breadth validation: every feature passing a basic test. Validators who have done this before know that this includes enabling global features (such as power management and DFX) which require significant effort.

6.4.5.1 Testplans

Integration testplans are typically the top priority for the Validators once the 0.5 IP integration is complete. Different approaches to writing integration testplans are under development at this point of time, but there is general agreement that the content of the testplans should include all flows associated with the IP. It is necessary to go beyond making sure interface connections are correct, because running tests on IP flows at the integration level exposes misplaced assumptions between different IPs/features. A full replication of the IP testplan is not required, but running the basic tests at SOC finds bugs. The testplans written in Val 0.5 are for the full POR set of features, and not just the features defined/delivered in the 0.5 drop. The PLC HAS schedule supports this activity.

6.4.5.2 Breadth Regressions

The tests written for breadth validation of a feature are often appropriate for addition to the gating regressions. The basic tests prevent backslide of the feature health when subsequent RTL drops (intermediate or 0.8) are turned into the model. The Validator must understand the relationship between the breadth tests and the interfaces and interface protocol under test. All interface protocol are validated as part of Val 0.5: it is not acceptable to learn that basic protocols or wiring is missing beyond this point in the schedule. Examples of the breadth tests include access to the IOSF primary and sideband fabrics, read/write properties of registers (including security policies), cold and warm reset, DFX features enabled, entry and exit of low power states and basic performance tests. The breadth regression are often needed in simulation because the test is used to gate turnins to the model. When appropriate, the breadth validation should be replicated on the emulation model. The breadth tests can be used as a metric for the health of model releases.

6.4.5.3 Val 0.5 PLC Requirements

The PLC has an extensive list of requirements for Val 0.5 completion (it is extremely challenging to meet all of these requirements in the allotted time):

- All SoC Test Plans written & reviewed and tests prioritized for freeze and tape-in.
- Basic validation completed on 0.5 IPs. IP integration validation completed for 0.8+ IPs with Regression. Note: the PLC schedule has the 0.8 IPs integrated into the model at the end of Val 0.5. The PLC expectation is that Val 0.5 regressions are successfully rerun with the new IP drops.

- Basic tests passing for all global features (e.g. chassis DFX, power management, RAS/Error PSS 0.5, emulation/FPGA build checks)
- Enable Boot/Reset Flow for the Platform Validation.
- Progressive performance capabilities tested/demonstrated (50-60% performance test plans written & running).
- PSS 0.8 and PSS 1.0 Acceptance Tests defined in collaboration with Platform Team.
- Always-alive Power-Aware Emulation & FPGA Models (~ weekly cadence)
- DOA tests passing on the GLS DC (or Xprop) model
- RTL Signal force and jams documented,
- Security: SDL S1 & S2 milestone complete.

6.4.6 SOC VAL 0.8

The theme for SOC Val 0.8 is the transition from exercise to validation. The key concept is the completion of the 80% highest priority line items in the testplan. All checkers must be implemented, and coverage complete as it pertains to the testplan entries. In Val 0.5 global flows such as power management were exercised with a small number of basic tests. At the close of Val 0.8 these features must be validated such that all wake mechanisms are tested, and there is a significant cross product with the non-PM flows. What is not expected in Val 0.8 are the deepest cross-product tests. The bugs found beyond Val 0.8 should fall into the category of “When A, B and C happen in the same time window.....”. The completion of SOC Val 0.8 takes place after the RTL 1.0 (final) IP RTL integrations. The milestone gives the green light to the Structural Design team to start the final synthesis for tapein.

<THIS SECTION IS NOT WRITTEN. Likely discussions include bugeco validation, elimination of workarounds. Describe depth of testing (Security, power gating, PNP, Boot ROM)>

6.4.7 PLC SOC Val 0.8 Exit Criteria

- All freeze gating tests passing. Minimum of 80% of new and 90% of legacy content regression passing. This applies to simulation and emulation. Testing includes requirements such as Performance and PSS tests
- All PSS Acceptance Requirements met
- Bug rate consistent with RTL Freeze, and no approved bug fixes more than a few days old.
- Failing tests debugged
- Execute GLS (or Xprop) to level that represents 95% functionality and cold boot`

6.4.8 SOC VAL 1.0

The completion of SOC Val 1.0 send the message to project management that all planned validation activities are complete, and the Validation team has signed off that this design should be manufactured and brought back into the silicon lab with the expectation of highly functional silicon. It is a big commitment!

6.4.8.1 The Tapein Green Light

The Validation manager will, in most cases work with all Validators and domain managers/leads to give the green light to tapein. Any reasonable person understands the Validation Manager cannot commit to zero bugs escaping to silicon, but on the other hand he or she needs to give a clear GO/NO-GO direction. A “tentative GO” is a NO-GO. The Validation manager needs to have the confidence that the planned validation work has been completed. This plan was put together with the knowledge that this day would come, and if we executed to the plan the decision to give a GO should be straightforward. Ultimately, the success of the Pre-Si Validation team on first silicon is whether the Post-Si activities are blocked from exercising features due to RTL or complex firmware bugs. If the Post-Si Validation team can exercise all functionality, they are in a position to find and file the deep bugs or bugs that can only be found with system-level ingredients (usually SW). These bugs can be fixed on the B0 stepping and the SOC can go onto a timely PRQ. Conversely, bugs that prevent features from being enabled on A0 may require a new stepping to enable the feature, and thus the subsequent stepping is used to find the deep bugs and the program is delayed.

6.4.8.2 Val 1.0: Complete Planned Validation

<NOT WRITTEN> Key theme is that Validators need to dot the I's and cross the T's. Discuss reviews.

6.4.8.3 Bug Validation

<Not written>Key theme he is that when a bug is found we ask “Why now?” and what other bugs in this category could be hiding elsewhere in the system

6.4.8.4 Evil Validation or Paranoia Validation

<Andrew will write this later>

There is management theme to be discussed here: The manager role is to make sure discipline is enforced (Tehcnical Contributors can get caught up in what they are working on and forget other things). The challenge is preventing the discipline from derailing into lengthy (and often pointless) checklists: the manager needs to rely on the expertise and judgment of the technical team to make the right decisions. This is resolved with meaningful interactions (and a level of trust that accumulates over the duration of the project (and beyond).

6.5 Post-Silicon and Stepping Validation

<This section needs to be completely rewritten>

The Post-Silicon stage refers to all project activities after tapeout. Once tapeout occurs, the nature of project activity changes radically, as does the organization of the project. In this section, the typical areas of Post-Silicon activity are outlined.

Once initial tapeout occurs, there is a waiting period where the ability to make progress is limited until silicon returns. This is a time when much training occurs for Post-Silicon tasks and when tools required for Post-Silicon undergo their final preparations before being deployed. Members of the Validation team participate in all these tasks, including the delivery of training courses.

Teams are formed for fault grading, pattern debug, speedpath debug, Low Yield Analysis (LYA), System Validation (SV), Compatibility Validation (CV), and Circuit Marginality Validation (CMV). By the time first silicon arrives, most if not all these teams are active.

The fault grade team writes tests that have the ability to expose processors malfunctioning due to localized manufacturing defects in the circuits. The pattern debug team is responsible for getting a selected set of tests, including the fault grade tests, functional for use with the testers, thereby enabling identification of faulty parts. Speedpath debug identifies circuit design causes for frequency-limiting failures that occur when applying good tests to parts. LYA looks for systemic reasons why parts are faulty.

SV, CV, and CMV do testing on platforms to demonstrate functionality of integrated systems. SV tests explore interesting microarchitectural areas. CV runs applications looking for anomalies. CMV works on the same platforms as SV and CV but focuses on provoking transient speedpaths.

The Validation team most commonly helps with fault grading, pattern debug, SV, and CV because these are the areas most dependent on microarchitectural knowledge without requiring extensive knowledge of circuit details. However, all these Post-Silicon debug teams do get some contribution from the Validation team.

Design and Validation loan people out to these teams as appropriate. Those members of Design and Validation not loaned out work on future steppings. Designers change circuits to improve timing in order to increase frequency. They must also propose fixes to any Post-Silicon bugs and find suitable implementations of these fixes in layout. Finally, they must evaluate the feasibility of adding proposed new features and implement those that are approved. These tasks resemble those of the presilicon execution stage.

For each stepping, a decision must be made as to whether the work for that stepping is complete and if the design is of suitable functional quality. The Validation team must contribute evidence regarding this functional quality. The requirements from Validation depend on what changes were made to the stepping. When major changes to the RTL have been enacted on a stepping, part or all of Validation might be asked to do a full revalidation to the same level of quality as the initial stepping. For bug fix steppings, Validation might focus testing on the buggy area but otherwise do a lightweight level of revalidation. For steppings involving little or no RTL change, Validation might spend little to no effort on revalidation. Decisions depend on available Validation resources, and an evaluation of risk accompanies any specific decision made. Each stepping is unique, and the amount of new validation and revalidation done must be evaluated in the context of that stepping.

From a Validation perspective, the key Post-Silicon goal is to identify silicon of essentially perfect functional quality in order to grant PRQ. Certain extremely arcane bugs can be tolerated in granting PRQ, and such cases are normally documented as errata. If a given stepping cannot grant PRQ, design changes are made in a new stepping, which permits forward progress towards granting PRQ on some future stepping. The other Post-Silicon goal of interest to Validation is to hit targets for

customer-visible DPM. Together these goals ensure that the parts that reach consumers meet the highest standards of functional quality.

See [The Post A-Step World](#) for a deeper look into the Post-Si world.

7 Summary

A project evolves through several milestones in both the IP and SOC world.

Kudos to those of you in the Art of Val forum who actually read through to this point. Give yourself a nice pat on the back!

8 Future Work

- This chapter should cover the roles and responsibilities of arch, planning, design and business groups.
- This chapter should cover how do you manage the project pivots. Particularly when you have a POC and then need to turn it into a product? The infrastructure required to do a POC is different than product, but management doesn't always see that. eDRAM and McIver are seeing this

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 4

Validation Planning

By: [Michael Smith](#)

1 Abstract

Validation planning is the process of identifying all tasks and objectives required to verify the functionality of a design and qualify it for production. The planning process involves identifying scope, gathering requirements, setting strategy, writing testplans and creating staging plans. The primary output of the planning process is a written testplan that codifies the validation Plan of Record (POR). After testplans are written, staging plans are created to map out and order the tasks that need to be done to execute the items in a testplan.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	05/01/2016	First draft of the document	Michael S. Smith	Michael S. Bair, Maria Pineda, Amber Telfer, Eric Adkinson, Lance Geiger, Colin Wood, Jae Ko, Gerry Chen

3 Contents

1 Abstract.....	81
2 Revision History.....	81
3 Contents.....	82
4 Purpose.....	85
4.1 Validation Planning	85
4.2 What is covered in this document?	85
4.3 What is not covered in this document?	85
5 Background Concepts.....	85
6 Validation Planning.....	86
6.1 Establishing Scope	86
6.1.1 Hierarchical Scopes.....	86
6.1.2 Discipline Scopes	89
6.1.2.1 Discipline Sub-scopes	89
6.1.3 Mapping scopes of ownership	89
6.1.4 Adapting scopes	90
6.1.5 Using scope for progress tracking	90
6.2 Gathering Requirements.....	90
6.2.1 Top-Down Product Requirements.....	90
6.2.2 Bottom-Up Product Requirements	91
6.2.3 Approaches for Validation Requirements.....	92
6.2.3.1 Specification-Based Validation	92
6.2.3.2 Requirements-Based Validation	93
6.2.3.3 Pre-Silicon Validation Requirements	93
6.2.4 General Requirement Sources	94
6.2.4.1 Features	94
6.2.4.2 Specifications	94
6.2.4.3 Requirement Sets.....	95
6.2.4.4 Other Sources of Information	95
6.2.4.5 Requirement Gaps	96
6.2.5 Gathering Requirements by Scope.....	97
6.2.5.1 Low Level Scopes	97

6.2.5.2	Subsystem Level and Integration Scopes	97
6.2.5.3	Fullchip Scopes	98
6.2.6	Ramping on Requirements	98
6.2.7	Learning the Design.....	98
6.2.7.1	Learn the Big Picture	98
6.2.7.2	Understand the RTL implementation.....	99
6.2.7.3	Understand the History.....	99
6.2.8	Requirement Review.....	99
6.3	Setting a Strategy	100
6.3.1	Documenting Strategy	100
6.3.2	Elements of Strategy.....	101
6.3.2.1	Defining Ownership	101
6.3.2.2	Defining the Execution Strategy	102
6.3.2.3	Defining Validation Test Environment.....	103
6.3.2.4	Defining Tools, Flows and Methodology (TFM)	104
6.3.2.5	Collateral Development	106
6.3.2.6	Progress Tracking Strategy	106
6.3.2.7	Multi-Stepping and Multi-Die Derivative Strategy	108
6.3.3	Strategy Approval	109
6.4	Writing a Validation Testplan	110
6.5	Creating a Staging Plan.....	110
6.5.1	Staging Plan Tasks.....	111
6.5.2	Collaboration and Joint Staging Plans	111
6.5.3	Staging Plan Format	111
6.5.4	Staging Plan Development	112
6.5.5	Staging Plan Execution.....	112
6.5.6	Staging Plan Maintenance	113
6.6	Other Planning.....	113
6.6.1	Forecasting	113
6.6.2	Effort estimates	113
6.6.3	Reprioritization, Effort Reductions, Schedule Pull-Ins	113
7	Summary.....	113
8	Future Work	114

9 References.....	114
--------------------------	------------

4 Purpose

4.1 Validation Planning

“Plans are nothing; Planning is everything” – Dwight D. Eisenhower

Validation faces the difficult task of finding bugs and demonstrating that a complex design is ready for production. Planning is essential to success, and upfront efforts to identify and organize the strategy, objectives and methods of validation will directly affect the quality of the design. Approaching this challenge with ad-hoc methods and with no clear strategy is shortsighted and introduces risk to the project. This document describes the steps of an organized planning process that will enable high quality Validation:

- 1) Establishing Scope
- 2) Gathering Requirements
- 3) Setting Strategy
- 4) Writing a Testplan
- 5) Creating a Staging Plan

4.2 What is covered in this document?

This document describes the validation planning process for a project in terms of establishing scope, gathering requirements, setting a validation strategy, writing a testplan and creating a staging plan.

The planning stages set forth in this document do not need to be part of a formal or official planning process. However, the concepts, methods and procedures described here have grown to be part of validation practices established over many project generations. This document strives to recognize and capture these planning practices as recommendations for validation teams.

4.3 What is not covered in this document?

Writing a Validation Plan is one of the key stages of Validation Planning, but it is only briefly mentioned in this document. Testplan content, drafting, reviews, maintenance and reuse are not covered here. Readers are directed to review [Testplan Writing](#) to complete the discussion on Validation Planning.

5 Background Concepts

Validation planning requires a solid understanding of validation concepts, and this document frequently cites concepts described in other chapters. Readers are encouraged to refer to other chapters as needed.

6 Validation Planning

Validation planning begins in the Technical Readiness (TR) phase of the PLC in a project (see [The Life of a Project](#)) and involves establishing scope, gathering requirements, setting a strategy and writing a testplan. Staging plans are also developed to identify, prioritize and organize all the tasks needed to carry out validation. This chapter provides details on how each of these activities is accomplished.

6.1 Establishing Scope

Dividing ownership and establishing scope is an essential part of validation planning. Establishing scope provides a focused starting point for all subsequent steps of validation planning. Requirements are gathered based on scope, strategies are set based on scope and testplans and staging plans are created based on scope. Scope is one of the first things that should be clearly stated when communicating with stakeholders of the project on validation activities.

Establishing scope occurs during the technical readiness (TR) phase of the project PLC. Validation managers play a central role in defining scope because scope ownership directly influences the size and types of validation teams that are organized for the project.

The first step of establishing scope is break down the validation that a team will own. To be more accurate, the first step is to break down the validation that *multiple* validation teams will own. As complexity increases, the more difficult it becomes for a single team to handle all validation. Validation uses a ‘divide and conquer’ approach to tackle the challenge of an infinite validation space by breaking up validation into categories of manageable scope and applying strategies that reduces validation complexity within that scope. As discussed in the [Validation Disciplines](#) chapter, Pre-Silicon Validation divides the scope of ownership across:

- Design Hierarchies
- Validation Approaches
- Validation Disciplines

Establishing scope is the process of mapping out validation scopes of ownership across these domains. Every team should understand their scope in terms of the hierarchy they will validate, the disciplines they employ and the approaches they use.

The motivation for dividing Pre-Si Validation effort among the scopes listed above is to simplify and optimize validation. The tools for testing and validating in various scopes can be fundamentally different. By dividing into scopes of ownership, validation teams can reduce the number of methods they use and can centralize expertise to operate more efficiently.

6.1.1 Hierarchical Scopes

When establishing scope, Pre-Silicon Validation teams initially divide the scope of ownership along hierarchical lines that match design hierarchy. This *hierarchy-first* approach acknowledges that Pre-Silicon Validation **must be aligned** to the hierarchical development of a design, so that features can be validated as soon as possible after they are implemented. It also enables validation to divide ownership along logical boundaries that use similar validation methods. In a hierarchy-

first approach, all aspects of validation are planned based on a hierarchical context. Teams gather requirements, establish strategies, write testplans and create staging plans all from the context of a hierarchical scope of ownership.

Dividing ownership on a *hierarchy-first* basis stands in contrast to a *feature-first* organization. Many project stakeholders including project planners, Architects and Post-Silicon Validation teams operate with a feature-first approach that organizes around design features rather than hierarchy. While Pre-Silicon Validators primarily operate on a hierarchy-first basis, they must remember that the success of a design is ultimately determined by how features work in the hands of customers. Pre-Silicon Validators should not limit their attention to the hierarchy they own, and be prepared to step outside of their hierarchical scope of ownership if needed.

Pre-Silicon Validation teams contributing to a Microprocessor or System on Chip (SoC) project might divide validation ownership among the following design hierarchies:

- **SoC Fullchip*** (System) Full SoC/CPU RTL model with IPs and cores
- **Uncore Subsystem** (Subsystem) RTL model with connected IPs, no cores
- **IP Subsystem** (Subsystem) RTL model of a collection of related IPs
- **IP / IU**** (Component) Standalone RTL model of a functional IP block
- **SubIP**** (Component) Standalone RTL model of a logic block

* South Complex PCH teams also use the designation SoC Fullchip for the full RTL model, although no CPU processing cores are present as they are not part of the chip design.

** SoC/CPU validation teams often do not have direct validation ownership for these hierarchical scopes.

Note in the differentiation between component, subsystem and system hierarchical scopes. Components are validated as standalone blocks in isolation from other system components, while subsystems include a variety of components that are integrated and tested together. The SoC Fullchip system hierarchy represents the entire chip design.

The practice of dividing validation ownership along hierarchical lines is not unique to Pre-Si Validation. Other validation organizations also follow this approach, although they may use different terms for each level. Examples of hierarchical scopes generally recognized outside of Pre-Si Validation include:

- **Platform:** i.e. SoC/CPU die + Chipset + Software + Firmware
- **Component/System:** i.e. SoC/CPU
- **Subcomponent:** i.e. IA core, Memory Controller, Power Management Agent etc.

Each hierarchical scope serves a purpose and introduces constraints on the validation methods that can be used. The following detailed examples of hierarchical scopes used in Pre-Silicon Validation are shared here to provide insight into how they are used.

SoC Fullchip: SoC Fullchip (FC) validation contains the full system design. Validation at the FC level is compute-intensive and costly, but it is the only place that all components of the system can be tested together. In the case of a CPU design, FC includes all IPs and all IA or Graphics processing cores and validation uses IA-based test generators and testing tools that are different from those used at other levels. In the case of a South complex PCH, the FC design does not contain cores and testing techniques are similar to those used in a subsystem.

Subsystem: Often it is prudent to integrate and validate a collection of IPs together as a *subsystem* before they are put into a SoC FC design. Subsystems contain a collection of IPs that are integrated together with surrounding logic in a design configuration that matches the product design configuration. Subsystem validation is useful for groups of IPs that have interdependent logic or interdependent flows. Stimulus is typically generated by driving traffic packets directly onto interfaces using test sequences. Subsystem level validation makes it possible to test and validate these IPs together with more control and less compute overhead than FC.

The Uncore subsystem is a unique subsystem validation scope that usually only exists for SoC CPU projects. It contains nearly all of the system design, except that processing cores are removed (hence the name *Uncore*). Uncore subsystem testing is useful for validating most of the system hardware and global flows and for validating IP integration without paying the compute expenses of FC.

IP / IU: IP-level validation involves components that can be independently validated before it is integrated into a SoC. IP components typically support a specific set of features, and get their name from the Intellectual Property or *IP* they represent. IPs are designed in a modular fashion with standard interfaces to facilitate integration into the system. IP-level validation enables engineers to thoroughly test and stress all aspects of the IP component before it is integrated into a system. IP-level testing emphasizes white box testing of all aspects of the implementation and typically uses low-level stimulus injections on the IP interfaces. Many IPs are developed and validated by teams or organizations that are independent from the team responsible for a SoC/CPU design. The term *IU* or *Integration Unit* is used for IPs that are developed by the SoC/CPU, to differentiate them from IPs that are developed externally.

SubIP: SubIP-level validation is nearly identical to IP validation scopes, except that SubIP components tend to be small blocks of logic that form common building blocks for IPs. Like IPs, SubIPs are validated in a standalone environment. Unlike IPs, SubIPs do not receive the same degree of dedicated validation focus when they are integrated. SubIPs might lack standard interfaces, or they might contain logic that defines a standard interface. SubIPs tend to be validated with custom test environments instead of the highly standardized test environments of IPs.

When discussing hierarchy and scope, terms like *system* and *component* may have different meaning depending on the audience. Validators are reminded to always be clear in their communications dealing with hierarchical scope.

In the remainder of this document, the term *hierarchical scope* is used interchangeably with the term *level*. For example, the Fullchip or IP hierarchical scope may simply referred to as *FC-level* or *IP-level*. In addition, the terms *design*, *design under test* or *DUT* may be used generically to refer to the portion of a design belonging to a team's hierarchical scope of ownership.

6.1.2 Discipline Scopes

Aside from dividing validation ownership across hierarchical scopes, Pre-Silicon teams also divide scope ownership among several specialized disciplines. Validation disciplines described in the [Validation Disciplines](#) chapter include:

- Dynamic Validation
- Formal Validation
- Integration Validation
- Firmware Validation
- Mixed Signal Validation
- Reset and Power Management Validation
- Intel Architecture Validation
- Design for Test Validation
- Design for Debug Validation
- Security Validation

As with hierarchical scopes, the objective of dividing ownership across specialized disciplines is to simplify the validation effort by aligning teams around common sets of tools and validation methods.

6.1.2.1 Discipline Sub-scopes

Discipline scopes can be further subdivided into domains of expertise. For example, integration validation nearly always divides ownership into separate scopes for each IP that requires integration. In addition, many disciplines divide ownership into separate sub-scopes based on features. For example, dynamic validation may subdivide validation ownership scopes based on features such as control registers, fuses or cross-feature testing.

6.1.3 Mapping scopes of ownership

The full description of a scope of ownership combines hierarchical scopes with discipline scopes (or sub-scopes). Projects should map out all anticipated scopes of ownership in this way so that teams can begin to plan with specific scopes of ownership established.

For example, a project list of ownership scopes might include:

- Display IP Dynamic Validation of Feature XYZ
- Uncore Subsystem Control Register validation
- Uncore Subsystem PCIe Integration validation
- Fullchip Intel Architecture validation
- ...

The purpose of mapping out scopes of ownership is to provide direction to each team as they begin the next stages of validation planning. It also provides a high-level view of ownership so that gaps or areas of overlap can be easily identified.

It is not necessary to have all combinations of hierarchy scopes and discipline scopes. For instance, mixed signal validation might not make sense at a Fullchip hierarchy. Similarly, formal validation might not map onto a standard design hierarchy.

Some scopes of ownership can be combined. For instance, a single power management validation team might own validation across Uncore and Fullchip hierarchical scopes. On the other hand, a team that owns both of these scopes may want to treat the scopes separately because the validation environment, tools and flows are likely to be different.

6.1.4 Adapting scopes

As validation techniques and environments evolve, it may be possible that entirely new domain or hierarchical scopes need to be created. Firmware validation is an example of a domain scope that grew into a distinct scope of validation ownership as it became more efficient to develop dedicated validation tools, flows and methodologies around this domain.

6.1.5 Using scope for progress tracking

Once validation scopes have been established and assigned, progress should be tracked by scope. Validation data like testplans, coverage and pass rates should be developed, stored, organized and maintained on a per-scope basis so that progress can be tracked by scope.

Organizing validation data by team instead of scope is a common pitfall. This may limit visibility into the health and progress of each discipline or hierarchy where unique validation tools and methods are being used, or where unique features or IPs are being tested. This can be problematic for teams that own several scopes, or can cause complications when validation ownership is handed off to teams that do not own the same scopes as the original team.

6.2 Gathering Requirements

Gathering Requirements is a stage of planning where Validators learn about the design in their validation scope of ownership and work to identify, understand and collect all of the fundamental expectations for how the design is required to behave. The requirements of a design are a primary input to the validation planning process, so it is important to understand where they originate and where they can be found.

6.2.1 Top-Down Product Requirements

All top-down product requirements originate from the POP process. The POP process identifies a set of features that the product must support to be competitive in the market and to meet business demands. Features that are approved as POR for the project are captured in the product *landing zone* (LZ).

The landing zone represents more than just the silicon product that will be marketed to customers. The landing zone actually describes a *platform*, because the LZ must be supported by an entire

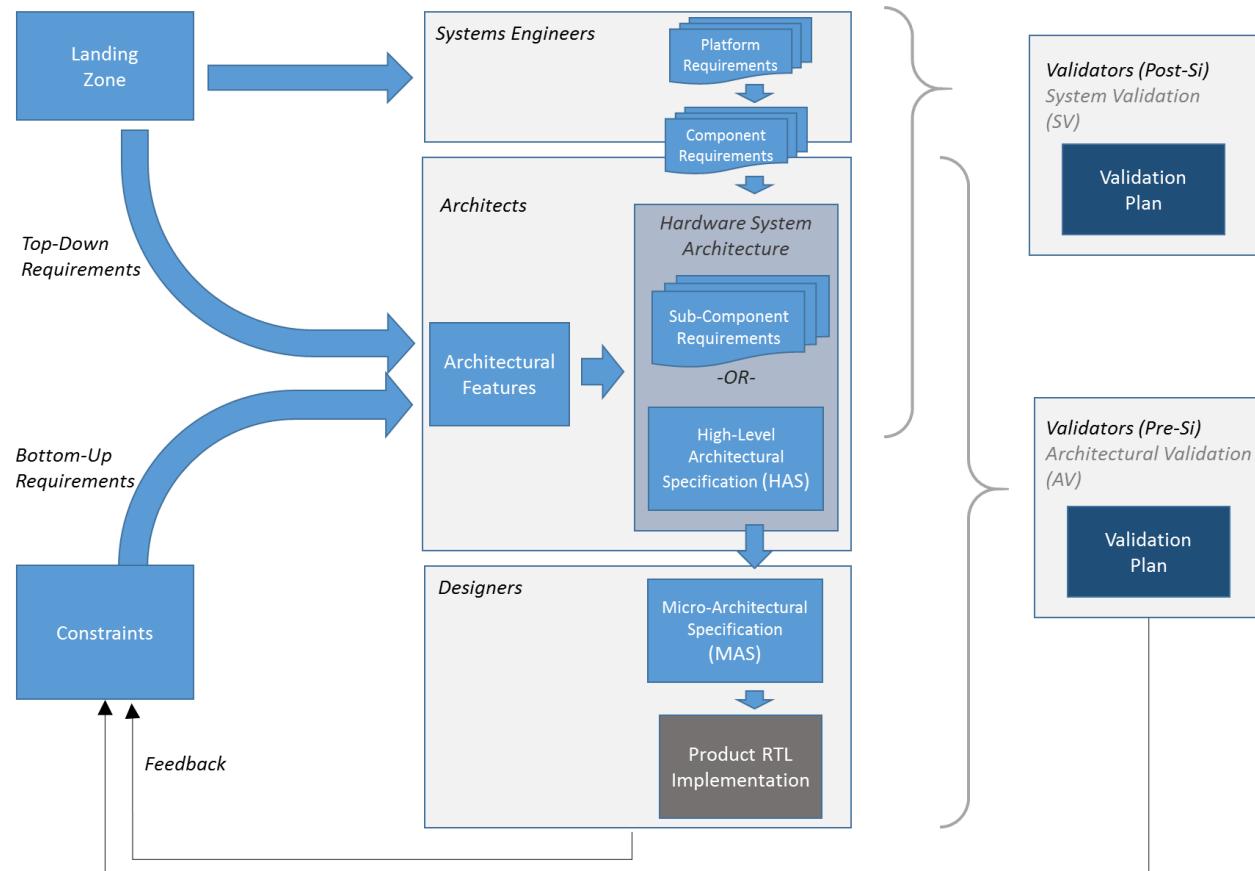
platform of software, firmware and hardware ingredients. Systems engineers break down the landing zone into platform requirements for each ingredient. These requirements are known as *platform-level requirements*. In the meantime, project Architects focus on the silicon product itself and work to break down the landing zone features into hardware design requirements. These requirements are known as *component-level requirements*, and are usually captured by Architects as a set of *architectural features*. At this point, all initial top-down high-level hardware requirements for the product have been defined for the actual silicon chip or SoC.

The next step is to define the *hardware system architecture*. Architects identify hardware sub-components that can be used to support the architectural features, and work to define a hardware system architecture that combines and connects the subcomponents into a working system. A hardware system architecture might be a system on chip (SoC) comprised of processing cores, embedded firmware, a memory controller, a power management agent and other functional blocks that are connected together by a fabric with standard interfaces. These subcomponents might be modular in design, but Architects need to specify which features each block must support, how they are to be connected and arranged in the system, and how they are to communicate with each other to support both basic functional operation and architectural features. Architects generally capture details of the hardware system architecture and *sub-component requirements* of the product in written specifications.

Designers are expected to deliver RTL designs that meet the system and component-level requirements, and Validators are charged with evaluating the designs to ensure they are capable of basic functional operation and satisfy all top-down product and platform requirements, all component-level architectural features and all hardware system architecture and sub-component requirements.

6.2.2 Bottom-Up Product Requirements

Bottom-up requirements are product requirements that do not originate as top-down requirements from the product landing zone. There are many challenges, pressures and realities to overcome in developing a functional, competitive product. Implementation and development constraints can drive the definition of new architectural features, force changes to the hardware system architecture or influence the sub-component requirements of a product. Examples of bottom-up constraints include design implementation constraints, cost optimizations or validation complexity. Bottom-up product requirements are usually invisible to the end customer, but are equal in importance to tops-down requirements for developing functional, competitive products. As with top-down requirements, Architects capture approved bottom-up features and requirements as architectural features or in written specifications.



6.2.3 Approaches for Validation Requirements

As just described, there are many layers of requirements for a design. Systems engineers and product Architects document the expected behavior of the system and its components either as sets of distinct requirements or in written specifications. Validation uses this as input to validation planning, so the methods used to express requirements can have a non-trivial impact on the approach that engineers use to verify a design. This has led to two generally recognized styles of validation: Specification-based validation and requirements-based validation.

6.2.3.1 Specification-Based Validation

Specification-based validation is a method of engineering where written specifications are the primary inputs to validation planning. In specification-based validation, validation targets the design requirements described in written specifications. Validators read specifications to identify the key features of a design and derive test scenarios to verify that the features function correctly in the design implementation. With specification-based validation, Validators do not just test the documented scenarios, but they look for boundary cases that will stress the design with the goal to expose bugs.

6.2.3.2 Requirements-Based Validation

Requirements-based validation (RBV) is a method of systems engineering (SE) where specific requirements are provided as the primary inputs to validation planning. In requirements-based validation, validation targets a specific set of requirements to verify. RBV is more commonly used by Post-Silicon validation teams, as dedicated systems engineers have begun breaking down the Landing Zone into discrete sets of platform and component-level requirements. These requirements are often expressed in terms of market-driven customer *use cases*.

Requirements-based validation works well for products that are to support a well-defined set of customer use cases. Products that go into fixed-function devices fit this category. When product use cases are well-defined and all functional expectations of the system and its components are clearly understood, it can reduce validation scope and simplify validation planning.

RBV based on use cases alone can be hazardous because use case requirements focus on demonstrating that certain scenarios work, rather than focusing on boundary condition requirements under which they might not work.

6.2.3.3 Pre-Silicon Validation Requirements

Pre-Silicon Validators primarily deal with requirements defined at the hardware system architecture or sub-component level. Validation is specification-based because most requirements are provided in the form of architectural features or written specifications. Specifications can contain a blend of top-down and bottom-up requirements with no clear delineation between the two. Pre-Silicon Validation gives equal attention to top-down and bottom-up requirements when evaluating specifications.

Whenever applicable, Pre-Silicon Validation should consider use case requirements when defining the testing scenarios in a testplan. In the meantime, Pre-Silicon Validation will continue to use specification-based validation as long as written specifications are used to capture component and implementation level requirements. The following barriers would need to be overcome before Pre-Silicon could adapt RBV:

- **Requirement Availability:** Architects currently provide hardware system architecture and subcomponent-level requirements in the form of specifications. Component-level requirement sets are not available to Pre-Silicon Validation.
- **Requirement Completeness:** Requirements derived from top-down requirements of the platform fail to include bottom-up requirements required for design and validation. Requirements are often expressed in terms of user experiences and usages that involve a fully developed hardware and software. This is a challenge because testing hardware with real software is not possible in many pre-Si validation environments. Also, requirements are seldom broken down into the implementation-level details that are essential for validating at the hardware system architecture and subcomponent levels. The machine microarchitecture contains a variety of FSMs, datapaths, flow control structures, queues, logic elements and protocols that are invisible at a platform feature level, but need to be validated. Top-down feature requirements may not reflect all the system requirements needed for basic operation and are not a complete set of requirements.
- **Requirement Adequacy:** Use case requirements often are expressed in terms of representative features or product usage scenarios that must work. The focus is on

affirming certain use cases work, and does not place adequate focus on the boundary condition requirements that are important to Pre-Silicon Validation. This poses risk, because testing may not flush out corner case bugs that may exist in the hardware. Secondly, some products are intended to be used by customers as a platform ingredients in open-ended ways, and customer use case expectations may not all be knowable in advance. Customers may not disclose all of their use cases to a product team, and may only provide general requirements. In such cases, use case requirements are inadequate. Validation must plan on verifying behavior beyond known use cases. Client microprocessor and SoC designs often fit this category, which is why product Architects typically capture requirements as broad behaviors described in specifications rather than in requirement sets.

6.2.4 General Requirement Sources

The following are common sources of requirements for Pre-Silicon Validation.

6.2.4.1 Features

As described earlier, architectural features are defined by Architects as they translate landing zone requirements into a hardware system architecture. Architects typically record features in a database. These feature descriptions in these records are typically high-level, and may contain documentation links that provide significant information and details on how the feature will be implemented or what specific requirements must be met.

6.2.4.2 Specifications

Architects also capture requirements in written specification. Specifications describe the features, functionality or expected behaviors of a design. Specifications might use written descriptions, tables, diagrams, images or other media to represent design requirements. Validators are expected to carefully read specifications to become experts in understanding all functional requirements of the design.

There are three common types of specifications used as requirement sources for pre-silicon validation: High-Level Architectural Specification (HAS) and Micro-architectural Specification (MAS) and external specifications.

6.2.4.2.1 High-Level Architectural Specification (HAS)

A High-level architectural specification or HAS is used to describe the expected behavior of the design at a high level. A HAS describes the overall architecture for a part of the design and outlines key flows, features and functionality. Architectural details that are visible or relevant to software or other agents in the system are described, and low-level details are often omitted and it is left up to Designers to create an implementation that meets the specification. There are usually multiple HAS documents that are written for a project, where each might describe some feature or domain.

Examples of HAS's include: Global Flows HAS, Power Management HAS, IP HAS, IP Integration HAS or a HAS for specific features.

6.2.4.2.2 Micro-Architectural Specification (MAS)

Another form of written specification is a micro-architectural specification, or MAS. The MAS is typically written by a Designer to describe the details of the micro-architectural implementation for a design. A MAS may not contain many high-level requirements, but it does provide details and requirements of the hardware implementation. For instance, a HAS may state bandwidth and ordering requirements for a datapath, while a MAS may describe the implementation-level details of arbiters, queues, virtual channels or flow control structures that are used to meet the HAS design. A Validator would be expected to ensure that all of these hardware structures work as expected in a variety of conditions and may need to target test cases around each of these structures to verify that nothing about the hardware implementation would cause a violation of the HAS requirement.

6.2.4.2.3 External Specifications or Standards

There are a variety of external specifications used by Validators. Many parts of the design involve industry standard interfaces or protocols. Examples include the DDR4 JEDEC Memory Standard, the PCI Express Standard or the Advanced Configuration and Power Interface (ACPI) Specification. Validators are expected to know the external specifications related to their portion of the design.

6.2.4.3 Requirement Sets

As described earlier, Architects or Systems Engineering (SE) teams may capture design requirements into a formalized set of requirements. Requirements are generally captured in a database as sets of market-driven customer *use cases*. Use cases are generally platform-level requirements that avoid implementation-level details, but they can be broken down into more detailed sets of requirements for a system component or even for subsystems. Architects for SoC and CPU products have traditionally not provided requirement sets that go down to the component or subcomponent level, opting for written specifications instead.

6.2.4.4 Other Sources of Information

Features, specifications and requirement sets are not the only sources that document the expected behavior of a design. Other sources of information are described here.

6.2.4.4.1 Issues

Oftentimes Architects, Designers or Validators identify issues with a design feature or requirement and file official issues in a database to document the issue and track resolution. Issues may be as simple as a clarification on an architectural feature. They may involve complex architectural problems that create a scenario where a design requirement cannot be met. In any case, open issues will be tracked and resolved by Architects. A variety of documentation and other details are created

and linked to the issue in the database in the process of closing the issue. This documentation is a valuable source of requirements for validation.

6.2.4.4.2 ECOs

Designers file Engineering Change Orders (ECOs) as they seek to implement new features, fix bugs or make any change to the design late in the schedule. A committee of Architects, Designers, Validators and program managers evaluates the impact of the proposed design changes. The ECO must be approved before the change is implemented. The ECO process creates documentation on the expected behavior of a design, which constitutes a new design requirement for validation.

6.2.4.4.3 Reference and Historical Documentation

Reference materials come in many shapes and forms. Sometimes historical documentation provides insight into requirements or expected behavior, especially for legacy features. Validators might use the internet to find online documentation of industry-standard specifications. Alternatively, Validators may want to consult with validators or the testplans from an IP or earlier projects to gain possible insight in to the expected behavior of their own design. In doing this, Validators should understand that similar features might exist across products, but their implementation might be completely different and the requirements or behavior described in another project's documentation may not apply to the current project. In any case, reference or historical documents can be valuable as a learning tool or helping to build understanding around a feature or product.

6.2.4.5 Requirement Gaps

Unfortunately, it is not uncommon for there to be gaps in requirements. As touched on earlier, requirement gaps can come in the form of availability, completeness or adequacy gaps.

Availability gaps can happen with legacy features. Legacy features are features that have existed for several product generations, and a product team may choose to not re-document all expected behaviors and requirements of legacy features. Documents may be out of date, or it might be impossible to locate the requirements for legacy features.

Completeness gaps can happen when Validators begin to learn the design and start to have questions about how the design is supposed to work. Sometimes, there are questions for which there are no documented answers, and the process of working with Designers and Architects produces an answer that becomes the specification or requirement. Completeness gaps can usually be closed as stakeholders resolve questions and issues.

Adequacy gaps can happen when validation meets all requirements, but the design remains of poor quality. This can happen when requirements are inadequate for qualifying the design for production.

Validators are encouraged to work directly with Architects or other authors of specifications or requirements to resolve gaps, resolve questions, and to provide direct feedback on the content or quality of specifications. When architectural experts are not available, Validators will need to employ their engineering skills to understand what the author meant when a specification was

written. Validators should do all they can to understand the requirements, and resolve questions and support improvements to the specification.

6.2.5 Gathering Requirements by Scope

Validators should gather all requirements applicable to their scope. They should never assume that requirements can be found in one document or specification. Requirements can be spread across multiple documents. For example, a Validator for display integration may need the following documents to understand the architectural requirements and expectations of the design:

- Display IP G-Unit HAS
- Display Integration HAS
- Display-Graphics Global Flows HAS
- Display MAS
- IOSF System Fabric Specification
- Display Serial Interface Specification
- Global Reset HAS
- Global Power Management HAS
- Global Security HAS

6.2.5.1 Low Level Scopes

For IP and SubIP scopes, Validators should identify any HAS or MAS documents that exist that describe the IP or SubIP in detail. Validators should look for:

- Implementation-level details about the functionality of all supported features
- Descriptions of external interfaces and all transactions supported on the interface
- Descriptions of all significant blocks, structures, FSMs and datapaths in the design and their functionality
- Descriptions of control registers and configuration details
- Descriptions or diagrams containing low-level details of flows

6.2.5.2 Subsystem Level and Integration Scopes

For subsystem level and integration scopes, Validators should identify any Integration HAS or Global Flow HAS documents that might exist describing the subsystem or IP(s) that are being integrated. Validators should look for:

- Details on hardware interfaces of the subsystem, including which blocks are connected to each other via the interfaces.
- Details about which agents or IPs communicate and interact with each other over each interface, which types of transactions they use, and in which flows they are used.
- Details on how IPs participate with each other to support product features.
- Details on how IPs are used in global flows such as reset exit or power management events.

6.2.5.3 Fullchip Scopes

For SoC or Fullchip scopes, Validators should identify any HAS documents dealing with global flows, architectural features filed as official feature records and product feature requirements.

Fullchip scopes should also be acquainted with the Intel 64 and IA-32 Architectures Software Developer's Manual (referred to as the Programmer's Reference Manual or PRM). This multi-volume reference describes the basic architecture, programming guides and instruction sets that all IA-based systems must support.

6.2.6 Ramping on Requirements

Validators should thoroughly read and research the requirements collateral they gather and become experts on the functional requirements of the design. Validators should also develop perspective on the relationship is between low-level requirements and the high-level features defined by the product landing zone.

Where possible, Validators are also encouraged to work directly with Architects or other author(s) of specifications or requirements to resolve any questions that might arise, and to provide direct feedback on the content or quality of the specification.

6.2.7 Learning the Design

Ramping up on the design is an essential part of gathering requirements. While learning the design, Validators often find important testplan scenarios that they derive from the design implementation. As strange as it may sound, the design implementation itself can be a source of requirements!

It may not even be possible to investigate the RTL design early on, but it is important to develop the habit of understanding of both the architectural features of the design *and the implementation*. The design is constantly evolving, so the habit of learning the design should continue well into execution. Validators should continually evaluate design changes to see if validation requirements emerge.

Validators can use the following top-down approach to learn the design.

6.2.7.1 Learn the Big Picture

Begin learning about the design by understanding the role that your piece of the design plays in the broader system. Developing this Big Picture understanding will help you see where to focus validation efforts. Start by gaining a good understanding of the high-level architecture of the fullchip system:

- Learn about the architecture. Validators working on Intel designs should familiarize themselves with the overall architecture, major functional blocks and should know where they can find specifications for standard interfaces used in the design.
- Learn about high-level flows and how information and data is passed and processed through the system. Learn by reading specifications and by participating in chalk talks with Architects or senior Validators.

Next, delve into the lower levels:

- Learn about flows in your piece of the design. Read the HAS and MAS (if available) and participate in discussions on the micro-architecture with Architects, Designers, and senior Validators. Learn about the main FSMs and interfaces of the design.
- Learn about interactions with neighboring blocks in the design by reading relevant HAS specifications, and talking with the micro-architecture experts.

6.2.7.2 Understand the RTL implementation

The lower the hierarchical scope of ownership is, the more likely you will need to do logic-level white box testing of the design implementation. This will require you to delve into the design RTL. Reviewing RTL files can be overwhelming and time consuming, but it is extremely important to know the implementation. You can do this independently, or with the help of the RTL owner. Keep in mind that interactive code reviews with Designers can be helpful for both parties.

As you browse through the RTL, try to understand the overall organization of the functional blocks in the design. Search for feature descriptions and try to answer the following questions:

- What are the RTL source files in which design features are implemented?
- What data paths or control logic exist for features?
- What are the key signals that reflect the state of the design?
- If it is a pipelined design, what are the different pipe stages through which the logic is flowing?
- What interfaces does the DUT use to interact with other agents in the system? Where is the logic that drives the interface?

Apart from reading the RTL, running tests interactively or capturing waveforms and tracing back the logic clone in a simulator can also be very valuable. Again, this is an activity that might not be able to start until later, but it is a good practice.

Further discussion on learning microarchitecture and becoming an expert can be found in [Becoming the Microarchitecture Expert](#)

6.2.7.3 Understand the History

Learning about how a design or feature was implemented on a prior project can shed light on to bottom-up validation requirements. Learning what is changing and why can be instructive. For instance, a design may decide to implement a feature involving communication between two agents by using a new opcode on an existing interface. This design choice may simplify hardware, but the introduction of a new opcode may introduce complexity to the validation requirement for the interface.

6.2.8 Requirement Review

During the requirement-gathering stage in the planning process, Validators are to identify all the sources of requirements and work to understand those requirements. These requirements are the

primary inputs for the validation planning, so it is important to ensure nothing has been missed. Once Validators have gathered requirements, it is worthwhile to review them with Architects and Designers to confirm all requirements are represented and understood. Reviews can be a formal or informal process.

6.3 Setting a Strategy

After Gathering Requirements comes Setting a Strategy. A validation strategy should be comprised of a concise set of strategic policies that will enable Validators to meet their objectives. Setting a strategy involves making decisions on how validation principles such as [Stimulus](#), [Checking](#) and [Coverage](#) will be applied to a specific validation scope.

Strategy is not limited to Stimulus, Checking and Coverage. This section will aim to identify all essential elements of a validation strategy and provide considerations when defining a strategy. Essential components of a validation strategy include:

- Defining Ownership
- Defining Execution Strategy
- Defining the Validation Test Environment
- Defining Tools, Flows and Methodology
- Developing Collateral
- Defining Execution and Tracking
- Defining Strategy for Multi-Stepping and Multi-Die Derivatives

These elements of strategy will influence all subsequent planning. It provides a ‘stake in the ground’ around which more detailed plans can be made. Strategy will affect *what* you validate by influencing the types of test scenarios that are put into a testplan and affect *how* you validate by establishing the tools and methods used to test and check the design.

6.3.1 Documenting Strategy

Creating written strategy documentation is useful for setting direction and level setting among validation stakeholders. Strategy documentation does not need to be lengthy. The key points can usually be expressed in brief strategy declarations or statements. The sections below contain examples of strategy declarations.

Some elements of strategy will be established at the project level and some at the team level. A project may choose to create official documentation of strategic PORs to be used by a broad set of validation teams. Project-level strategy documentation may be posted in a database or posted to a shared project website. Projects are encouraged to provide templates for capturing strategy PORs. Project-level POR documentation usually provides background information on the options considered in addition to the strategy declarations. Team-level strategy is usually captured as part of testplan background information (see testplan section below) and does not usually provide information on other options considered.

6.3.2 Elements of Strategy

This section describes the elements of a validation strategy in more detail.

6.3.2.1 Defining Ownership

The first element of strategy is defining ownership. Defining ownership is similar to establishing scope, except that it goes further to define the boundaries of ownership in more concrete terms. A strategy that defines ownership will state:

- team name
- the product
- the validation scope of ownership
- what is covered in the validation scope of ownership
- what is not covered
- assumptions that are made about validation ownership by other teams

The ownership definition should also touch on which validation approaches will or will not be used (see definitions of interconnection, implementation, feature and cross feature validation approaches in the [Validation Disciplines](#) chapter).

Clearly articulating the boundaries of ownership for a validation team in this way is critical. It helps avoid gaps, prevents unnecessary overlap, breaks down assumptions and helps facilitate understanding between complementary validation teams that have similar tasks. The process of defining ownership may require negotiation between teams, a process commonly known as establishing Roles and Responsibilities (R&Rs).

Here is an example of defining ownership:

The PCIe integration team covers the Uncore subsystem and FC-level integration of the PCIe IP into all desktop systems of the Broadwell client product family. Validation focus is directed at integration-level behavior and SoC-level interactions and flows between the PCIe IP and other SoC agents. Special attention is given to connectivity of the IP and proper functionality of the IP as it participates in the testing of global flows and features, including reset and power management flows. The team will share ownership with the Power Management team in enabling PCIe in global power management flows, although PM cross feature validation will be limited and handled primarily by the PM team. It is also assumed that IP-level validation has modelled SoC-level features and flows in unit-level testing and that IP-level has done robust validation of any features, flows, interactions and microarchitecture elements that are implemented entirely within the IP, including interface compliance to PCIe and IOSF specifications. It is assumed that analog circuits in the Analog Front-End (AFE) of the PHY layer has been validated by mixed signal teams at the IP/Sub-IP level, as integration validation will only use behavioral modeling of the AFE. PCIe-related DFT conditions are not in the scope of validation and are expected to be covered by DFT Validation.

6.3.2.2 Defining the Execution Strategy

Validators need to define an execution strategy that describes how they intend to test the design. For many validation scopes, execution strategy is generally framed as being either *coverage-driven execution*, *content-driven execution* or a blend of these two.

6.3.2.2.1 Coverage-Driven (Oriented) Execution

The [Coverage](#) chapter discussed background principles of coverage. There are two scenarios where the importance of coverage is amplified for the overall validation strategy. One scenario is when validation teams rely heavily on a random testing for stimulus, and there is no way to know if the random tests are triggering specific events in the design. The other scenario is when validation needs to exercise a broad variety of hardware structures in the design implementation and there is no way to determine which elements of the design are being exercised. These scenarios often go hand-in-hand for white-box validation at the IP level where random testing is frequently used to test functional events and exercise logic structures in the design. This approach adheres to the philosophy that covering a broad set of functional events with random testing is a superior method for validating a design than using directed tests. In these cases, running tests and monitoring test results does not provide sufficient feedback to indicate that a feature or hardware element in the design has been tested. Coverage of the target events cannot be implied and explicit coverage is needed. This drives a validation focus on coverage as the primary indicator of progress during execution, and is known as a *coverage-driven* or *coverage-oriented* execution strategy. Testplans built upon this execution strategy will contain coverage-based entries that describe the coverage events that need to be hit by testing to verify a design.

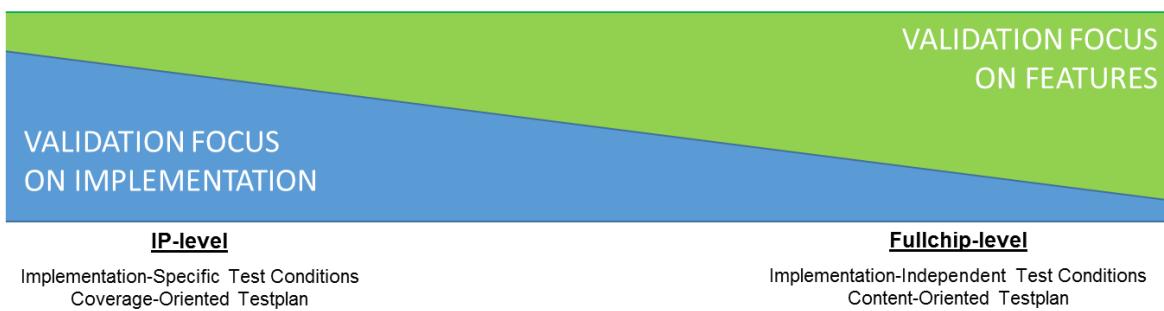
6.3.2.2.2 Content-Driven (Oriented) Execution

For some validation teams, coverage plays a smaller role in the overall validation strategy. There are two scenarios where the importance of coverage can be de-emphasized. One scenario is when self-checking directed testing (or even directed-random testing) is used to validate new features and a passing test is sufficient feedback to infer that the feature has been tested. The other scenario is when a suite of legacy tests is carried over from established designs and is used to validate legacy sets of features. If Validators can reasonably conclude that a test or regression is sufficiently capable of verifying the proper behavior of the design or that a feature has been properly tested, then there is arguably no need for explicit coverage. This drives a validation focus on test run status and regression pass rates as the primary indicator of progress during execution and is known as a test *content-driven* or *content-oriented* execution strategy. Testplans built upon this execution strategy will contain *feature-oriented* entries that describe the design features that are to be tested, along with the specific tests that will be used. Content-driven execution is frequently used in several discipline scopes including Intel Architecture, Design for Test and Design for Debug Validation.

A caveat to content-driven execution is that there is always a danger that inherited or directed/directed-random tests may not function as expected on the current design. A test may be run on the DUT with passing status but fail to test the intended feature. Validators must be quite sure that they can avoid false passes if they choose to use a content-oriented approach instead of using coverage.

6.3.2.2.3 Blended Execution Strategy

In practice, Pre-Silicon Validation teams typically use a blend of coverage-oriented execution and content-oriented execution. Generally, lower hierarchical scopes of validation lean heavily towards coverage-oriented execution while higher hierarchical scopes of validation shift the balance towards content-oriented execution. Of course, there are exceptions. For example, a new IP might use coverage extensively to verify the hardware implementation, but a mature IP with little or no design change might use existing content regressions to verify the design is still stable when it is delivered to a new product.



Blending execution strategies will produce a mix of the type of validation entries that are defined in a testplan, so testplan tools especially should support entries for both coverage-oriented conditions and content-oriented conditions. Validation tools should not prohibit the use of either approach.

An example of execution strategy declaration is provided here:

The imaging subsystem team will use a combination of direct testing and random testing and maintain all tests in a regression. Most feature testing will target implicit conditions where tests are designed to fail if the targeted flows do not complete. While coverage will be used to collect feedback on a few explicit events described in the testplan, no extensive effort will go into defining coverage events for implicit conditions.

6.3.2.3 Defining Validation Test Environment

The next element of strategy is defining the validation test environment. The validation team should consider topics discussed in the [Validation Platforms](#) and [Test Environments](#) chapters and apply them to the current validation scope.

As part of defining the validation environment, a validation team must decide:

- What platform will be used? VCS simulation, Emulation, Formal Environment etc.
- What DUT(s) or model(s) will be used, and why they will be used?
- What are the requirements of the test environment?

For IP and SubIP teams, the DUTs or models will include the corresponding IP and/or SubIP design. For subsystem and fullchip teams, determining a model can be more of a challenge, as there are many factors to consider.

First, a model must be used that has all the components involved in the scope of validation ownership. For example, in the case of IP integration, the team will need to test all interactions of their IP with other agents in the system. The DUT or model the IP integration team requires needs the memory component, clock units, a power management agent, a sideband network, a primary fabric, a fuse puller and many other components. Teams should evaluate their model requirements and define the model that best allows them to accomplish their objectives.

Determining a DUT or model is also complicated by external constraints. Model performance, compute capacity, compute availability and maintenance costs all factor into consideration. Teams evaluate whether it is necessary to create and support new models that include or exclude additional areas of the design. After the DUT or model is defined, ownership and scope may need to be adjusted to the hierarchical scope.

A strategy that defines the validation environment will clearly state what type of validation platform will be used and what DUT(s) or model(s) will be used, and why they are used. Some details of the test environment can be provided, but descriptions should be kept brief.

An example of a validation environment strategy declaration is provided here:

Power Management validation will use both VCS-based simulation and large-box emulation to validate digital logic in the front-end RTL. Two models will be used:

Pre-Si Uncore simulation model: This model has no IA/Gfx cores in the DUT, but contains the fabrics, the power control block and all IPs connected to the primary system fabric. UPF is enabled in this model and will be used to model power delivery. Core emulators will model core/GFx behavior. This model is used to verify basic plumbing for PM flows and handshakes with all IPs except core, and can utilize/reuse low power sequences delivered from each IP.

Pre-Si FC model simulation: This model contains all elements of the design as the Uncore model, but also includes 2 IA cores and 1 Gfx cores in DUT. UPF is enabled. This model will be used to test full PM flow interactions, especially those involving cores and GT and will take advantage of fullchip test generators that create code to run on the cores to create more advanced test scenarios.

6.3.2.4 Defining Tools, Flows and Methodology (TFM)

Defining the strategy for tools, flows and methodology (TFM) is the point at which a validation team begins to take all of the concepts of [Stimulus](#), [Checking](#), [Coverage](#) and [Debug](#) discussed in earlier chapters and applies them to their scope of ownership. The validation team must establish and define their own set of practices.

TFM strategy is often inherited from prior projects. Decisions to update, reuse or change TFM strategy are frequently made on a project-wide basis so that teams will converge on common solutions. Validators should not re-document project-level TFMs that have already been

established and documented as project PORs. Validation should understand where global TFM PORs are documented and provide links in their strategy document.

When local tools, flows or methodologies are used, Validators should document the TFMs in their strategy. The primary goal is to document the tools and high-level approach. The strategy declaration should answer the following questions:

Stimulus

- Will testing include random tests, directed tests, formal methods or a combination?
- Are there design modes or configurations that will be tested?
- What test templates, injectors, tools or test generators will be used to create stimulus?
- What test lists or regressions will be run, and how often? What tools or flows will be used to manage regressions?

Checking

- What checking tools or methods will be used?
- Will assertions, static checking, dynamic checks, post processing, test self-checking be used?
- What types of checking will be used?
- Will testing involve transaction scoreboards, compliance checkers, or checkers for performance or ordering?

Coverage

- What tools will be used for collecting and analyzing coverage?
- What is the coverage objective? Is 100% coverage expected to be hit for all conditions?

Debug

- What tools, flows or methods will be used for debug?

An example of a validation TFM strategy declaration is provided here:

Stimulus: Directed IA test templates and random test generators will both be used to create ASM-based test content for stimulus. Stimulus will focus on exercising features in the FC functional scope of ownership. SW/OS and MRC is only minimally modeled. FC simulation will use few/no internal injections for stimulus, although traffic from external interfaces may be generated using BFM s. Tests will be launched in the HDK environment using the trex/simgress execution stack. Tests covering all features will be added to regression lists, and regressions will be launched on a regular basis (~weekly) either manually or using Granite automation. Gating tests for key features will also be added to turn-in regressions to maintain model and feature health.

Checking: Validation will rely primarily on self-checking behavior in the directed tests and tests from the random test generators for checking. Tests will report failures or cause test hangs when features do not behave as expected. This self-checking will be complemented by design assertions, compliance monitors and various other dynamic system verilog scoreboard checkers that will be enabled in all FC test runs.

Coverage: Self-checks in directed tests will provide sufficient feedback to demonstrate implied coverage of the features the tests target. Similar principles apply to test generators

using feature based configuration files. Consequently, little effort will go into creating coverage monitors for implicit events. Functional coverage will be written for explicit events (not otherwise observable) using QuickCov methodology. Coverage results will be collected and aggregated by CASA. Coverage analysis will be done using Verdi coverage analysis tools.

Debug: Failures from regressions or manually launched tests will be reported in Triage for debug and resolution. nWave/nTRACE FSDB debug tools will be supported by project tools for enabling debug.

6.3.2.5 Collateral Development

Validators should define their strategy for procuring or developing validation collateral. The strategy should answer questions such as:

- Where will test content come from? Will it be inherited, reused, adapted from an IP or prior project or developed from scratch? Will it be developed for reuse by others?
- Will collateral in the test environment be inherited, reused, adapted from an IP or prior project or developed from scratch?
- Which behavioral models, BFM's or other forms of verification collateral (VC) are required and how will they be obtained? Will they need to be developed? Do licenses need to be secured?
- Are there internal or external dependencies on other teams or suppliers for validation collateral?

6.3.2.6 Progress Tracking Strategy

The validation strategy needs to define how progress will be tracked and reported. Many things can be used to evaluate the health of a design and measure validation progress. When a coverage-oriented execution strategy is used, progress tracking might be tightly coupled to coverage reports. Similarly, when a content-oriented execution strategy is used, progress tracking might be tightly coupled to test results and regression pass rates. Blended execution strategies might rely on a combination of coverage, test status and pass rates to determine progress. In any of these cases, completion of testplan items should also be reflected in progress tracking. Because there are so many factors to consider, tracking validation progress can get complicated.

Validators should have the right strategy and tools to track and report progress. Progress tracking should always be based on the scope of ownership. A good strategy allows Validators to express their confidence in the design, track progress against project schedule, and indicate when they have qualified the design for production.

Several options for tracking validation progress are discussed here.

6.3.2.6.1 Ad Hoc Tracking and Analysis

Ad hoc methods of tracking validation progress can vary widely. Teams might develop custom scripts to extract regression data, check on the results of test runs or reports on coverage.

Alternatively, they might evaluate and report progress based on their direct knowledge of current validation activities without using any specialized scripts or tools. Ad hoc methods can be customized, adapted or optimized for specific scopes of ownership.

Ad hoc methods of analyzing and tracking progress rely on the judgment and expertise of individual engineers and the teams doing the work. Consequently, they have the potential to provide a more accurate view of progress than any generic automated tool can provide. For the same reason, ad hoc progress tracking methods are highly subjective and may fail to detect risks or report problems in validation execution. Ad hoc methods also make it difficult to compare progress against other teams. Ad hoc progress tracking methods should be scrutinized to ensure they represent a full picture of the progress and quality of validation.

6.3.2.6.2 Quality of Validation Metrics

Some projects create standard metrics for tracking validation progress and reporting the quality of validation. With this approach, objective measurements are taken for things like pass rates and coverage and a standard formula is applied to determine the quality of validation. Quality of Validation metrics are based on real data sources and can be helpful for establishing a progress tracking in a standardized way across many validation teams for a project. On the other hand, quality of validation metrics may not map well to every scope of ownership and teams usually need to apply customization to metrics. For instance, a team that uses large regressions of random tests may need to place a higher weight of emphasis on regression pass rates in their metrics than teams that use directed tests for validation.

6.3.2.6.3 Traceability

Traceability can also help with progress tracking. Traceability is the practice of linking validation results with testplan entries or design requirements. Traceability shows direct relationships between validation objectives and validation results and helps demonstrate that validation has met specific design requirements.

Examples of traceability include:

- Testplan Entry ↔ Test result linking (content-driven validation)
- Testplan Entry ↔ Coverage result linking (coverage-driven validation)
- Requirement ↔ Test Condition linking (requirements-based validation)

Traceability can play a central role in progress tracking. For example, if a content-oriented test condition is linked to a test and the test is run with passing results, the Validator can conclude that the design requirement has been met. Likewise, if a coverage-oriented test condition is linked to the results of a coverage monitor, and analysis shows that all conditions have been covered, then the Validator can conclude that the design requirement has been met. In a similar fashion, progress tracking tools that support traceability can automatically generate testplan progress indicators based on testing results.

The term ‘fully-connected flow’ refers to full linking between requirements, testplan entries, and test or coverage results. Fully connected flows requires advanced tools – some of which may only support content-driven validation or coverage-driven validation. Validation teams may opt on a progress tracking strategy that uses partial traceability between the items they care about most.

Requirements-based validation facilitates fully connected flows involving requirement traceability because exact requirements are specified as inputs to validation. Fully-connected flows involving requirement traceability is more difficult to implement for the specification-based validation used by Pre-Silicon Validation because requirements are not always expressed as specific requirements in written documents.

6.3.2.6.4 Mixed Tracking Methods

Validation may use a combination of the progress tracking methods listed above. For instance, teams might use traceability to link test results to testplan entries so that a quality of validation formula can be used to track overall progress, but they might use ad hoc analysis to determine whether they have met the criteria for completing a PLC milestone.

It may be advantageous to change strategy for tracking progress during different phases of a project. For example, it may be prudent to manually track progress for testplan conditions as basic features are exercised through the PLC VAL 0.5 Milestone, then shift towards more advanced forms of tracking progress for the PLC VAL 0.8 Milestone as test run automation tools, coverage and regressions are deployed.

6.3.2.7 Multi-Stepping and Multi-Die Derivative Strategy

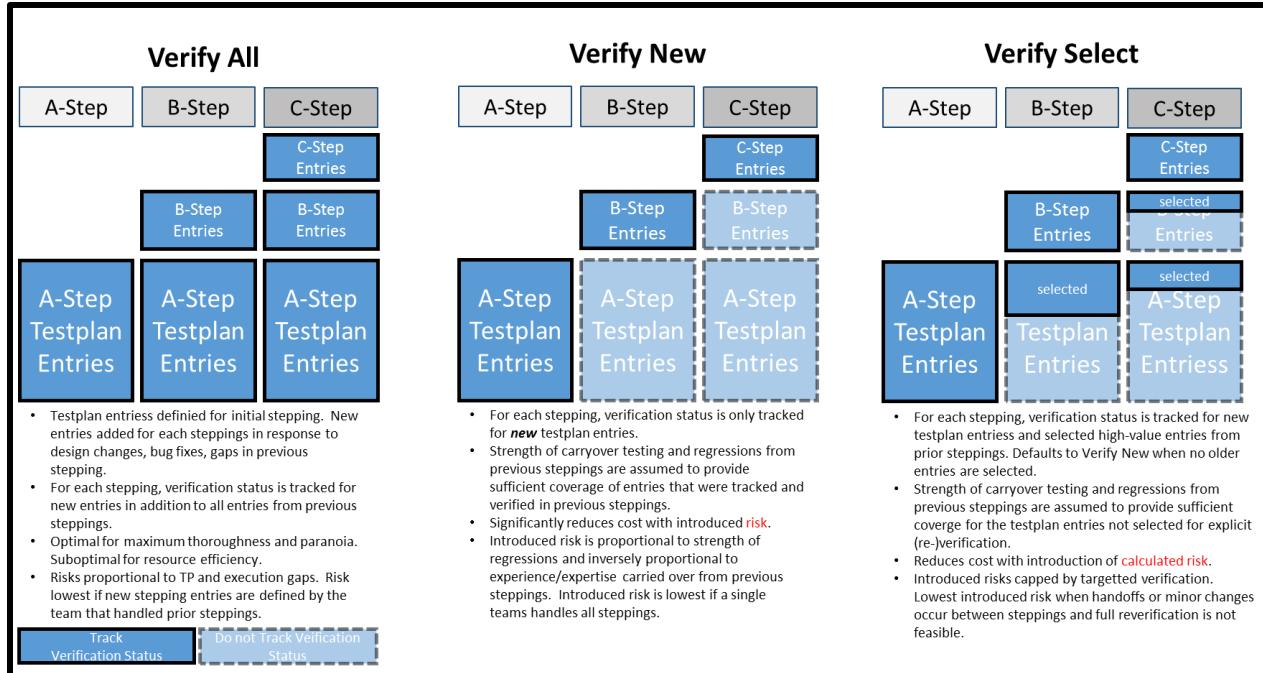
Validation should determine a strategy for handling steppings in terms of what is changing, and which entries in a testplan need to be re-verified and which do not. The following multi-stepping strategy options exist:

Verify All: New conditions will be added for the new stepping in response to design changes, bug fixes and gaps in the previous stepping. All testplan entries from the previous stepping will be tracked for re-verification on the new stepping.

Verify New: Only the new conditions added for the current stepping will be tracked for verification. Regression testing from the previous stepping will be run.

Verify Select: Verification status will be tracked for new testplan entries and selected high-value conditions from prior steppings.

The benefits and risks of each option are captured in the diagram below.



For derivative products, the same options exist, although the stepping strategy for the derivative does not need to be the same as the derivative strategy. For example, one project may use Verify New for each stepping. The next derivative may use a Verify Select strategy for its first stepping and re-verify selected testplan entries from the base product before reverting to a Verify New strategy for each of its derivatives.

Selecting testplan entries to be verified in a Verify Select multi-stepping or multi-die strategy should use the following criteria:

- Testplan entries should be selected for re-verification if they are needed to re-establish and *demonstrate the strength of the regressions* inherited from earlier systems or steppings. Testplan entries selected for re-verification should add value to our indicators and provide confidence that tests and regressions are at or above the levels inherited.
- Testplan entries should be selected to *lower the risk profile for areas with change*. Areas with major changes will typically need new testplan entries. Areas with minor change may warrant targeted retesting (e.g. a non-changing IP may connect to a fabric in a new way). Some areas have major hardware changes, yet existing test condition descriptions are still valid and can be reused and re-tracked on the new system.

6.3.3 Strategy Approval

Strategy should be defined during the Tech Readiness (TR) phase of a project and is to be approved by validation management during a strategy review at the start of execution as part of PLC Val 0.0. Project-level strategy PORs should be posted in a database or shared project website.

6.4 Writing a Validation Testplan

The most significant output of the validation process is producing a written testplan. A written testplan acts as the official Plan of Record (POR) for a validation team and outlines the set of validation objectives that will guide all validation activities during the execution phase of a project.

Although it is a significant part of Validation Planning, this chapter does not cover Validation testplan writing. See [Testplan Writing](#) for a full description of testplan contents, drafting, reviews, maintenance and reuse.

6.5 Creating a Staging Plan

Once the testplan is written, it provides a blueprint and set of goals for validation during execution, and even provides some high-level background on how the goals will be accomplished. However, just like the blueprint on a construction project, a testplan does not break down the tasks that need to be done or the order in which they should be completed. This is the purpose of a staging plan.

While some projects do not require a formal staging plan from validation, the staging of tasks is an essential concept. It provides a way to identify and prioritize tasks and map out dependencies. Validators are encouraged to develop a staging plan (official or unofficial) as the final part of validation planning.

The items in a staging plan are different from the entries in the testplan. Staging plans are used to provide an organized, logical *staging* of tasks that enable the testplan to be executed. Often they are combined with design staging plan tasks so that all design and validation dependencies can be identified. Consider the following simplified example for PCIe controller validation:

Testplan Conditions:

- PCIe Link training to L0 state
 - {32B, 64B, 128B} upstream memory {read, write} from PCIe device
1. Staging Plan Tasks: Install PCIE VC/BFM in test environment, connect to design in testbench.
 2. Install IOSF Primary and SB VC/BFM in the test environment, connect to design.
 3. Implement basic reset modeling, confirm with IOSF ISM initialization and PCIE training.
 4. Create base sequences and tests for PCIe testing.
 5. Test upstream posted memory transactions (writes), then nonposted (reads)
 6. Enable coverage monitors and coverage collection tools. Code coverage condition(s).

As shown in the example above, a staging plan may contain some test scenarios from the testplan, but this is usually because it represents a significant stage of development or because they highlight a dependency that forces tasks to be completed in a certain order.

Staging plans are a highly effective tool for mapping out tasks, recognizing dependencies and prioritizing work.

6.5.1 Staging Plan Tasks

Staging plans can be started even before testplans are started and are used throughout execution. Staging plans are particularly helpful in clearing dependencies during the initial validation enabling work that occurs between VAL 0.0 and 0.5 in the PLC Milestones, as validation infrastructure is being developed and new design features are being introduced. The following items are examples of the types of tasks usually found in a validation staging plan:

- Develop random test generators, directed test templates, other forms of stimulus
- Develop and deploy checkers
- Code the coverage test cases documented in the testplan into coverage monitors
- Enable BFM's, tools or components of the test environment necessary to enable validation
- Basic exercise of select flows or features to demonstrate significant milestones or test environment health
- Adding tests to gating regressions or standard regressions

The team (and project management) may show high interest in staging plans early on a project, but once the majority of the validation infrastructure is in place, they typically shift attention to other activities and execution indicators such as pass rates, coverage and bug counts to drive or determine the health and status of validation. As such, staging plans do not typically include the background tasks, or tasks that occur late in execution:

- Debugging failures
- Filing Bugs
- Validating bug Fixes
- Training
- Documentation
- Coverage analysis and follow-on improvements to tests/checkers/coverage

6.5.2 Collaboration and Joint Staging Plans

Staging plans are most useful when a group of people is working in parallel towards a common goal, and where there are interdependencies across their tasks requiring tight collaboration.

There are many dependencies between validation and design during the initial stages of RTL development and exercise. As a result, Designers will often collaborate on joint staging plans with Validation. This produces a highly effective environment where the details of development can be discussed and resolved to drive progress towards a common goal.

Projects may assign ownership of enabling basic exercise to either Validation or Design. For projects that assign this enabling work to a virtual team (VT) between Design and Validation, a joint staging plan is highly recommended.

6.5.3 Staging Plan Format

Unlike testplans, staging plans have a limited lifespan and are “by the team and for the team”. Consequently, they tend to be as lightweight as possible – often put together as simple line items in a spreadsheet, with no significant emphasis put on style or language. A typical staging plan will

list the task (as a 1-2 sentence description), the task owners, the ETA, which milestone the task belongs to, status and notes.

Some projects and teams have used advanced project-planning tools for staging plans. Tools might offer Gantt charts, advanced dependency mapping or the use of a HSD database to document tasks. Such tools tend to come with some effort overhead when entries are moved, edited or changed. This is a disadvantage to staging plans, because they evolve rapidly.

Staging plans should avoid becoming too detailed. 20-50 items should be sufficient for even major enabling efforts.

6.5.4 Staging Plan Development

Developing a staging plan can start even before testplan writing. Staging plan development can start as soon as validation tasks are identified, which can start when setting a strategy.

There is no formal process for drafting staging plans, and there is a lot of discovery along the way. Usually an engineer or small group of engineers will start by writing down all the tasks they can think of and then go through a few rounds of refinement. The key is to start with something. Validators should add anything that they think should be a task, even if the task is not well understood. Notes can be added to items that need to be investigated further. Over time, teams will develop a sense the type of tasks that will best facilitate their progress.

Once an initial staging plan is in place, it provides a framework for further planning. The plan will go through a lot of change as teams remember or discover items to add. Each new task has the potential to displace or move other tasks based on dependencies. Over time, the plan will stabilize as fewer tasks are added.

The key to a useful staging plan is understanding dependencies. Internal or external factors might dictate when various validation tasks can begin. If tasks are ordered correctly and dependencies are properly identified, the critical path can be found and work can be prioritized to accelerate the overall effort.

6.5.5 Staging Plan Execution

Staging plan items can start before a testplan is complete. Validators go to work on the staging plan, working to complete all tasks in order. When tasks are blocked by a dependency, they work to resolve that dependency if possible, and might work on other unblocked tasks in parallel.

Occasionally, there are scenarios where a task is not blocked, but should not be started. For instance, suppose there is a high amount of open bugs for a feature. In this case, it might not be the right time to begin running random tests targeting the feature even though the staging plan says to do that next. Doing this will only produce a high failure rate for an area of the design that will soon change. Similarly, developing low-level checkers for a design feature that is about to change may lead to unnecessary rework updating the checker for the new design implementation. This would be a very inefficient usage of time and resources. It may be prudent to work on something else in the staging plan until the RTL stabilizes.

6.5.6 Staging Plan Maintenance

Staging plans are very dynamic. Teams using staging plans should meet regularly to discuss progress, where the frequency depends on the speed and urgency of the tasks. Staging plan meeting (a.k.a. syncs) are used to provide updates, discuss dependencies and prioritize tasks. The focus is always on the next steps to drive progress. Inordinate amounts of time should not be spent on things that do not add value such as fine-tuning ETA dates.

The staging plan should be updated, reordered, modified or amended as needed. They should be posted to a location (such as SharePoint) that all team members can access.

6.6 Other Planning

Validation is often involved in a variety of other planning tasks. While it is impossible to anticipate all such activities, some of the more common examples are listed here.

6.6.1 Forecasting

Validators may be asked to provide forecasts for compute resources including provisioning of licenses, allocating disk space or forecasting compute demand for netbach.

6.6.2 Effort estimates

Validators are often asked to provide effort estimates for their validation activities or for validating proposed new features. Estimates are often given in *person weeks* or *person quarters* of effort, which is the amount of time it would take one person to perform the tasks if they were working on it full time. Sometimes this requires the Validator to provide their analysis and a detailed breakdown. Effort estimates generally do not include debug time, but should account for all enabling tasks – not just the actual testing itself. This helps management as they work to determine schedule and balance resources for project planning.

6.6.3 Reprioritization, Effort Reductions, Schedule Pull-Ins

Validators might be asked to join planning efforts associated with product priority calls, effort reductions or schedule pull-ins. Planning may involve reprioritization of tasks, risk identification, critical-path identification, adjustments to high-level and low-level staging plans and the elimination of some tasks.

7 Summary

The total of logical state combinations that are possible in a complex design is immense. Blending such a vast state space with boundless combinations of temporal behaviors creates a validation space that is virtually infinite. Unless an organized approach is used to break down the problem of an infinite validation space, it would be difficult to deliver products to market that meet the highest

standards of quality. By following the formula of establishing scope, gathering requirements, setting a strategy, writing a testplan and creating staging plans, the members of a validation team can establish and refine an achievable set of objectives that satisfy the true purposes of validation. Validation plans are dynamic and must adapt to the natural and unpredictable changes that arise. As Validators execute to their plans and apply a validation mindset along the way, they can operate with assurance that their efforts will produce high-quality results.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 5

Leading a Validation Team

By: [Michael Bair](#)

1 Abstract

Managers and technical leaders of Validation must effectively and efficiently drive validation of a project. They must also provide a framework to individual and team proficiency and career growth. They need to know how to hire good Validators, how to ramp the team, how to train newcomers and how to best use veterans. Validation staff members must have intimate knowledge of the status of their teams. They also need to understand the types of problems that will arise over the life of a project and how to deal with them using tradeoffs with schedule, goals, quality, and efficiency.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/13/2009	Initial Draft	Michael Bair	Matt Kupperman / CCDO uAV Staff
1.1	2/26/2010	First round of feedback edits	Michael Bair	Matt Kupperman / CCDO uAV Staff
1.2	3/9/2010	Second round of feedback edits	Michael Bair	Matt Kupperman / CCDO uAV Staff
1.3	3/15/2010	Feedback on "awards", and prep for L2 reviews	Michael Bair	Phil Atkinson
1.4	3/14/2011	Updated with feedback from the L2 review	Michael Bair	Phil Atkinson
2.0	9/9/2011	Update to cover all of Validation	Michael Bair	Matt Kupperman / CCDO Val Staff

3 Contents

1 Abstract.....	115
2 Chapter Revision History	115
3 Contents.....	116
4 Purpose.....	118
4.1 Why do we need this chapter?.....	118
4.2 What does this chapter cover?	118
4.3 What does this chapter not cover?	118
5 Background Concepts.....	118
5.1 Nomenclature: managers, staff, technical leads	118
6 Leading a Validation Team.....	119
6.1 Being a Leader	119
6.1.1 The responsibilities of Validation Staff	119
6.1.2 Traits of a good Validation staff member	120
6.1.3 Growing and maintaining your technical capability as a manager	121
6.2 Creating a team	122
6.2.1 Hiring good Validators.....	122
6.2.2 Organizing the parts into a team.....	122
6.2.3 Developing a consistent Validation Mindset.....	123
6.2.4 Standardization	124
6.2.5 Sit your team together.....	125
6.2.6 Developing a technical career path.....	126
6.2.7 Utilizing technical leaders	128
6.3 Validating a project	129
6.3.1 Planning for a project	129
6.3.2 Following the plan.....	131
6.3.3 Awarding good work	134
6.3.4 Managing the non-Validation side of the project	135
6.3.5 Managing positively versus managing aggressively	136
6.3.6 Transitioning from project to project.....	137
6.4 Special conditions faced in Validation.....	137
6.4.1 Extremely long project lifecycles	137

6.4.2	Validation output is confidence	138
6.4.3	Cross-site.....	139
6.4.4	Failure is an option	139
7	Summary.....	140
8	Future Work.....	140
9	References.....	140

4 Purpose

4.1 Why do we need this chapter?

Leadership is paramount in the engineering world to simultaneously execute projects, grow team efficiency and capability, and promote organization health. Teams with good management execute projects predictably, efficiently, and consistently. Their teams mature, gaining capability and efficiency, leading to better project execution in the future. Moreover, their teams tend to be happier and more motivated. Intel's greatest resource is its people; leaders must develop, refine, and safeguard this resource.

4.2 What does this chapter cover?

This chapter presents and explores the responsibilities of Validation leadership, management practices that we find particularly beneficial or interesting in the context of validation, and special challenges in leading a Validation team executing major projects. The chapter describes how to build the team and how to use that team to execute a project.

4.3 What does this chapter not cover?

This chapter does not cover the vast majority of typical good management practices taught in management courses. This chapter will not make you a good manager. This chapter is similar to other management training: used in the right context and with good judgment, it will help you to better execute a project and mature your team. Used with poor judgement... well, any tool in the hands of subpar management can be devastating.

5 Background Concepts

5.1 Nomenclature: managers, staff, technical leads

Within this document, the following terms and definitions are used:

- *Manager*: Any manager within the Validation team, including FLMs (First Line Managers) as well as managers at higher levels within Validation.
- *Technical Lead*: Any individual contributor within the organization that sets the technical direction within the team (with greatly varying scope between senior and more junior technical leads)
 - *The term “Technical Lead” does not refer to the official IAG Technical Lead program*
- *Staff (or Staff member)*: The managers and senior technical leads that own direction setting and accountability for the entire team

6 Leading a Validation Team

Success, on tough projects, derives from solid leadership and solid individual contribution. Failure typically comes from poor leadership alone.

6.1 Being a Leader

6.1.1 The responsibilities of Validation Staff

As a Validation Staff member your responsibilities, in priority order, are:

1. Executing your current project
2. Developing the career path of your team members
3. Increasing the capability of your team
4. Increasing the efficiency of your team
5. Preparing for the next project

All of these responsibilities are intertwined; for example: the second, third, and fourth all directly help the fifth responsibility: preparing for the next project. Another example: increasing the efficiency of your team also helps the execution of the current project.

First and foremost, you must execute the current project. If Intel has a top value, it is *results orientation*, and your main result is the quality completion of a project. There are two aspects to project execution; the first aspect is your ‘day job’: fulfilling the duties of the Validation team by completing the tasks assigned to it. The second aspect is to help guide the other parts of the project to completion using the expertise of your team. For instance, when new features are proposed, Validation helps project management measure the risk of the new feature prior to doing any real work on it.

Your second responsibility is the career path of your team. More than any methodology or tool, a team that sees their current position as a great place to take root and grow is vital to current and future success. Beyond team success, proper focus on individuals’ career growth builds job satisfaction. A career path in Validation can take on many forms (see 6.2.6), but all forms foster expertise and increasing interaction with other teams.

You must also increase the capability and efficiency of your team. At a first order, you need your team technically capable of completing the task at hand. Further capability gains allow your team to take on more kinds of tasks with less risk. For instance, having some expertise in software gives your team an added boost if your project takes on further ownership of software validation. In addition, you need to bring up the capabilities of more members of your team for “bench strength”: if you lose your expert, you will have good backup and will be well on the path to growing a new expert. Efficiency gains allow you to do more with your current team – this has always been good, but decreasing ASPs (average selling prices) will drive increased cost pressure within CPU project teams.

Finally, even though our project life cycles are very long, your next project requires consideration right now. As mentioned earlier, your team is your most important asset, so prepare them for the next project by increasing their efficiency and capabilities and by giving them a career path. Make

your decisions and create your infrastructure with the future in mind. As will be shown later, there are ways to increase efficiency dramatically in future projects with little effort now.

6.1.2 Traits of a good Validation staff member

Validation managers must be technically capable Validators; they might not file a bug or write a test environment, but they must understand the low-level methodologies employed by the team as well as the day-to-day issues facing Validators. Managers coordinate and track progress as well. Technical leads set the technical direction of the team; managers ratify and enforce that direction. In the absence or shortage of technical leads, Validation managers assume technical leadership duties as well, and thus own technical direction setting. See section 6.1.3 regarding how to grow and maintain your technical capability as a manager.

As a Validation staff member, you are responsible to teach, justify, defend, propagate, and hold people accountable for your chosen methods and policies. Engineers can tell the difference between a boss who understands and believes in the policies being driven, versus one who does not grasp the reasoning behind the policy or has not fully bought into it. This responsibility applies to methods and policies developed by Validation staff (almost 100% on NHM) as well as to those developed by members throughout the Validation team and ratified by the staff (vast majority of HSW). If you cannot teach/justify/defend/propagate/hold people accountable for some policy, then you should not have ratified it in the first place. Moreover, disagree and commit is important too – when you commit to a method or policy, you had best defend and enforce that policy as if it were your own idea!

You also have a shared responsibility with the other Validation staff members to make inter-team decisions. To do this, you must understand the status of the other teams and the issues they face. This shared responsibility may mean doing extra work to enable other teams, taking ownership of additional areas into your team, or loaning or permanently moving your engineers. You will have to weigh the risk added to your schedule with the project benefit gained by helping the other team, a tough challenge if you do not have a clear picture of the other team's status and their role in the project goals. To achieve a fluid system that allows load balancing across many groups within Validation, it helps to be under a single point of management and to have mutual respect among the Validation staff members. This kind of fluidity is impossible in a team that has leaders that protect their fiefdoms. First-line Validation managers must view failures in neighboring Validation sub-teams as partly their responsibility; therefore, they should always be watching for problems brewing outside their own team and act accordingly for the good of the project. Similarly, having a single point contact for each discipline helps with load balancing between different disciplines within Validation. Again, a sense of shared responsibility for the project and an understanding of how each team contributes to project success is paramount.

On the technical side, you need to have a detailed understanding of what your engineers are doing and the problems they are facing. There are many reasons for this, beyond helping them through roadblocks and using this understanding for performance management. When Validation staff members come together, they should compare notes on the issues their engineers are seeing. This helps to identify project trends early (for instance, Designers requiring too much turnin debug help), to share solutions (one team already dealt with the problem), and to increase long term efficiency through synergy. A manager might be able to unblock another team by reordering his

team's work. When staff compares notes, they can also scrub each others' effort estimates and inject reality into each others' plans. Be careful: while keeping abreast of your engineers' work, avoid unnecessary micromanagement.

Validation staff members need to follow basic engineering leadership rules: help remove roadblocks for your engineers, help them find resources, and drive urgency to the right tasks (which changes over the course of the project).

All Validation staff members should promote interaction with groups well outside of their ownership spheres. The CPU projects you own create a wealth of opportunities for this: bringing in other teams to help, participate, or give feedback on what you are doing, or to lend your teams' experience or time to other projects. This can be something as simple as a 2-hour review of another team's testplan or as much as giving another team your top TE (Test Environment) engineer for a full year to help their effort and to return with even greater expertise. These opportunities exist between Validation sub-teams within the same project, within other projects, in emulation, in Post-Si, and in countless other places.

Validation staff are often the buffer between Validation and other teams, such as Design. Friction can naturally take place between Validation and Design given the nature of their jobs (Validation is trying to find flaws in Design). Staff members can encourage disagreements while still fostering good partnership between the two teams to complete a top quality design. In conjunction with this, you must also communicate Validation impact in a language that others can understand and interpret within the context of the project. Section 6.3.4 expands on this further.

Lastly, a Validation staff member must be open to criticism. There is a lot to be learned through the feedback from others. Likewise, you must also learn to give constructive feedback and do so often. It also helps to keep a close eye on what other teams are seeing. Pay close attention to the post-mortem presentations of other Validation teams and attempt to learn from their successes and avoid making similar mistakes.

6.1.3 Growing and maintaining your technical capability as a manager

Managers can grow and maintain technical capability in a number of ways. First line managers without prior validation experience can grow expertise by managing a smaller team and doing technical work with half of their time. Do not have managers own critical technical tasks; instead, have your ICs own all critical technical tasks and position technical managers to own less-critical work, such as writing lower priority testplans, writing coverage monitors, and doing debug. Having managers own technical work will require a greater number of managers to cover the entire Validation team and will thus have some disadvantages, but in the short term (~1 project) it is the best way to ramp newcomers to Validation as managers. After a project with this structure, the team can restructure into a more ideal state for the next project (see section 6.2.2)

Beyond owning a portion of the technical work, there are plenty of ways to support your engineers that can help maintain your technical knowledge. Managers should read all of their reports' bug write-ups and make sure they are following guidelines and are up to standards. From doing this, you will also become more familiar with the features and more readily see bug trends as they emerge. Set up and attend chalk talks, and attend technical training that your reports take, both for architectural knowledge and for validation tools. Managers, especially in the absence of technical

leads, can own task forces that solve technical problems and thus become immersed in technical details.

When your team matures, you begin shifting more of the leadership and basic management tasks to interested members of your team. This promotes understanding within your team of the management side of the project (building bench strength), and gives you an opportunity to delve into other technical leadership opportunities within the project.

6.2 Creating a team

6.2.1 Hiring good Validators

Your hiring effort should target candidates with the correct behavior skills for Validation. Many technical skills and knowledge gaps can quickly be overcome after hiring whereas behavior is hard to change. Focus on behaviors such as teamwork, discipline, commitment to task, flexibility, initiative, integrity, problem solving, quality of work, attention to detail, and the ability to deal with ambiguity (see [The Validation Mindset](#)). Internships are the best way to judge the quality of a candidate. You should assume that the behaviors of the intern reflect his future behaviors as a full time Validator. When hiring via an interviewing process, employ behavioral interviewing techniques directly by having a dedicated behavioral interview, and when asking technical questions watch for the candidate's behaviors.

While behavior is primary, also look for software skills in candidates. Validation's software infrastructure is increasing at a rate much faster than the increase in CPU size, and software skills are necessary for many aspects of Validation, especially to maintain quality tools. Last, but not least, a candidate should understand CPU microarchitecture or analog circuitry and be able to work intelligently through design-related problems.

After hiring, you need to emphasize training. Training areas include methodology (see [The Art of Val Methodology Forum](#)), microarchitecture, mixed-signal, formal methods, test environment languages, tools, etc. Creating good training infrastructure, such as boot-camps, labs, classes, presentations, and online training, requires a lot of time. This time is recouped in two ways: first, if you have training materials already created, you do not have to put together some quick-and-dirty training every time you bring in someone new. Over the course of a project, you will realize this time adds up. Second, if given high quality training, your newer Validators will ramp faster and make a larger impact towards Validation's goals. You will also have a baseline from which to create training on your next project, rather than having to do it all from scratch.

6.2.2 Organizing the parts into a team

The ideal way to organize the Validation team is to have a single top level manager as well as having each discipline managed by separate management structures (within reason, of course) also with their own manager. FLMs should be technical experts that manage large teams (8-12 engineers). Having a single top-level manager gives you a decision making point, which is especially useful when you need to do major load-balancing across sub-teams. Having FLMs be technical experts allows them to understand the workflow and issues their engineers are facing, and understand the best way to avoid and handle problems. It also helps in training engineers when

their direct manager can fill in technical knowledge gaps during normal 1:1s and other such meetings.

The team needs to grow technical leaders to handle some of the leadership load of the team. Technical leaders will take on methodology and tool direction setting, they will spot technical problems early since they are deep in the trenches of technical work, they will help create and implement solutions to those problems, and they should be utilized to drive the toughest task forces. As your technical leadership ranks grow, they take on more responsibility for technical training in the team as well. More on this in section 6.2.7.

You should strive for a low turnover rate within your team, but you will always want a small stream of new people entering your group. This can be RCGs, employees transferring from other disciplines, and loaners from other teams. You should cross-pollinate between the Validation disciplines as much as is feasible; this spurs BKM sharing and innovation, and opens communication channels. A senior team with low turnover rate can bring elevated levels of proficiency and efficiency, but there is always an added risk of complacency with respect to trying revolutionary ideas from outside the team – and complacent teams eventually lose their leading edge!

When your team is junior, you need to have more bureaucracy to keep track of Validation status. This makes certain what is perceived as done is actually done and keeps the project from building on an unstable foundation. This has subtle benefits as well: it forces the management to remain extra focused on the execution aspects of the team, helping to maintain their technical expertise. It also forces newer members to articulate more of what they are doing, and the simple act of writing things down often makes them think it through. Other examples of heightened bureaucracy include extra reviews (such as code reviews and testplan reviews) as well as finer granularity of checklists, task lists, and effort tracking.

Validation often needs experts in many different technical domains to use as a resource for the entire team. Horizontal workgroups fulfill this need. A horizontal workgroup is made up of engineers across multiple organizations; for instance, multiple Validation sub-teams. They can include members from multiple Validation disciplines as well as teams outside Validation. Workgroups are typically created to resolve a problem or create a point of expertise. These workgroups can develop tool and method PORs for the team, create guides and training for those tools and methods, and have “office hours” for engineers to ask questions. As an added bonus, horizontal workgroups establish and develop technical leaders.

6.2.3 Developing a consistent Validation Mindset

Validation requires a different mindset than other engineering activities (see [The Validation Mindset](#)). To begin with, Validators must always assume there is *something wrong* with the RTL model and must exert themselves in all ways possible to find bugs. This alone sets them apart from most engineers, who are intent on creating working things, not proving something is broken. Furthermore, we develop our Validators’ mindsets with the tenets laid out within this textbook – high level ideas such as why we use checking and coverage, concepts such as ‘Validation is never done’, and even fine granularity training such as how to write a testplan or to properly target random tests.

Be wary of forces that can corrupt or influence this mindset. Design, our closest partner, is often a corrupting force – not because they are trying to be subversive, but simply because they believe that their design is correct and because they also must meet their milestones. This is a very different mindset than that of Validation (see also section [Dynamic Microarchitecture Validation section: As Design goes, so goes uAV](#)). Attaching your Validators too closely to the Design team is likely to put them in a position where they consistently see and hear the Design mindset, instead of seeing and hearing the Validation Mindset.

Consistent Validation Mindset enables some aspects of standardization. Validators must understand that the best productivity gain for Validation is often reached by individuals helping the entire team become more efficient, not by a bunch of individuals each making them self more efficient. This idea is further developed while discussing standardization in section 6.2.4.

Again, be wary of forces that corrupt this team mindset, even within a discipline. FV and DFTV contain sub-teams with different methodologies and tools. This is OK, but we wary of allowing the teams to extend these differences beyond what is necessary. For example, the further secluded IP Validation sub-teams are from one another increases the likelihood of differing mindsets between those sub-teams. Sub-teams with different mindsets begin to think of themselves as having “special cases” and rightfully justified to divert from the standard methods and practices chosen by the IP Validation team. This incoherency can cause a nightmare for IP Validation staff, not to mention adding risk to the project (these aspects of standardization are dealt with in section 6.2.4 as well as section 6.3.1.3)

6.2.4 Standardization

Validation teams are reaching peaks of around 150 engineers. Some of the larger sub-teams within Validation, such as IP Validation, have reached peaks of 50 to 60. Teams of this size have inherent inefficiencies due to requiring an enormous amount of communication and multiple levels of management. Conversely, large teams can be exploited for their innovative and load-balancing potential. Standardization is the key to minimizing inefficiency while exploiting this potential upside.

The larger the team, the more varied the tools and methods tend to become without strictly enforced standardization. This creates a higher workload for DA teams having to support more tools and methods. This also inhibits load balancing due to the lower efficiency of having engineers move to jobs where they have to relearn *everything*: not only the new microarchitecture and test environment in the part of the chip they just moved to, but also new methods and tools used by that team or the engineer they are now helping.

Large teams with strictly enforced standardization greatly enhance your ability to load balance. Standardization allows you to more accurately measure and compare the progress of your teams. Because the members of the team are doing similar work, the indicator comparisons are meaningful. Moreover, having all of the team use the same methods and tools (test language, testplan style, formal tools, coverage type and depth, etc) eases engineer transitions. Your engineers will still have to ramp on new microarchitecture and test environments, but they ramp much faster when they already speak the same ‘language’ as the new team, ie, use the same methods and tools.

Innovation benefits greatly from standardization. The scrutiny and brainstorming a tool/method receives is proportional to the number of users of that tool/method. Standardizing on a minimized set of tools focuses the entire team's innovative strength on those tools, and the entire team thus receives any resulting efficiency or capability gain. The following is an example: we currently have multiple disciplines using the Triage tool for failure bucketing and assigning debuggers. We created a tool within Triage that creates a single debug trace for each failure bucket. Since we are standardized, the tool applies to most of the Validation team. Prior to using Triage, every team bucketed failures in ad-hoc ways, and thus a single tool for creating debug traces would not have been possible.

The following is the most important lesson about large teams: you must impress upon your engineers that their contribution is job #1, but that their best way to help the team is to increase everyone's efficiency, not just their own. The larger your team, the less impact an individual can make to the entire team simply by increasing his own performance. Far more important is the impact that an individual can make on the performance of an entire team. In a team of 10, if one person suddenly becomes 50% more efficient (wow!), the entire team just gained 5% efficiency (great!). In a team of 100, the team just gained 0.5%. That engineer worked incredibly hard, and yet the team only gained a small amount. If that engineer would have found a way to make the team just 1% better instead of making himself 50% better, he could double his overall contribution to the team performance.

Obviously, the need for standardization and the relative importance of increasing the efficiency of the team (versus the individual) makes certain behaviors more desirable in potential Validators. For instance, strong teamwork and communication skills are highly desirable in large teams. Conversely, candidates that favor individuality and freedom are less likely to flourish in a large team with standardization unless they are committed to making individual breakthroughs and transferring them to the rest of the team.. On the management side, it takes great skill to teach Validators how they can stand out from the crowd when everyone is doing the same thing. It really comes down to the bottom line: a Validator's impact to the team. In large teams, the best way to impact a team is to increase the efficiency and capability of the entire team, not just the individual.

6.2.5 Sit your team together

The entire Validation team should sit together; this is for a number of reasons. First, when your engineers are under their heaviest workload, you want them sitting nearest other engineers that are using similar tools and facing similar problems. Proximity facilitates sharing and learning and greatly increases the efficiency of a team. For Validation, the critical project stage is during the execution of the IP RTL and SOC VAL milestones (see [The Life of a Project](#)), and the major work being done during those periods is debug, testplan/test/coverage/proof writing, and driving and analyzing coverage. During Execution, it is important to have the team sitting together and working together on similar tasks with similar goals. Within Validation, sit each discipline together as well. Because it makes sense to "form" the team prior to entering Execution state, we have the Validation team sit together for the entire project.

Second, have the entire Validation team sit together because you want them to develop a Validation Mindset, as stated in section 6.2.3. Sitting together lowers the communication barrier and helps the team to start thinking coherently. This furthers your ability to drive standardization across the

entire team (not just a discipline or within a sub-team), and to have the team see and understand the results as the team becomes more efficient.

Third, if you are serious about building a career path in Validation, you need to start here. Having people sit disparately gives them no sense of community within Validation. Nobody considers a group with a bunch of scattered people a “team”; yet having a team is essential to driving Validation as a career path. If you have exemplary Validators on your team, proximal seating allows your whole team to witness their actions and more readily emulate them. Again, you want to do this across the entire Validation team, not just within the sub-teams.

6.2.6 Developing a technical career path

Technical leaders increase the efficiency of your team and identify and solve major problems. They feel a heightened level of responsibility for the quality of their DUT, their team, and the project. They also relieve some of the tasks from Validation staff, ranging from simpler things like training to critical jobs such as driving task forces. When utilized correctly, the *rate of increase in technical expertise and leadership is proportional to the quantity of technical expertise and leadership*. This means that teams that are more capable are likely to increase their capability faster than teams less capable. There are caveats: if your technical leads do not feel responsible for driving the capability and efficiency of the entire team, this certainly will not hold true. This is yet another reason to make sure your engineers have a consistent Validation Mindset and act like a team.

To give structure to the process of developing technical leadership, it is important to create Validation technical career paths. There are many different technical career paths in Validation. Some Validators stay in one area for a long time, become a microarchitectural expert (see [Becoming the Microarchitecture Expert](#)), influence Architecture and Design in that area, and execute Validation tasks in that area very quickly. Some Validators prefer to move around the chip, becoming familiar with many parts of the CPU and eventually tackling features that have global effects in the CPU. Still another path is to focus on the methodologies or tools used by the entire team in an attempt to bring greater efficiency to all Validators. The Test Environment has become Validation’s most critical infrastructure, creating a career path for high-quality TE coders and Software Architects. Other Validation technical leaders may excel at giving other teams, such as the Post-Si Validation team, expert help during their critical phase of the project.

To help develop your technical leaders and the technical career path, you need to first create a career path guide. This career path guide is a set of expectations for different grade levels. Though it is not a checklist for promotion, it does detail the criteria for focal ranking and rating sessions, and is highly useful in career development planning and discussions. This document is a great framework for describing growth paths and higher-grade job roles to aspiring technical leaders. In DDG, we have created technical career path documents for each Validation discipline, though they contain many similarities. The DDG IP Validation technical career path document is [here](#).

Mentorship programs are also effective in developing technical leaders. If you have a solid set of technical leaders already within your team, using them as mentors will have three beneficial effects. First, it will increase the rate of development of the mentees. Second, mentoring will provide the mentors an opportunity to develop the behavioral skills essential to being a leader. Third, it will increase the sense of community in the team - nothing exemplifies *great place to*

work more than having senior members of your team showing more junior members that they are willing and wanting to have the junior members become more senior.

As mentioned previously in section 6.2.2, horizontal workgroups, especially for tools and methodologies used across the Validation team or sub-team, help to establish and develop technical leaders. Validators in these workgroups will need to learn how to interact with the team to gather information (what different flows will this tool need to support) as well as to train the team on new tools and methods.

Managers play a critical role in developing technical leaders. The process begins with realizing that one of your engineers is capable of technical leadership. As part of your management routine to assess Validation status, ask probing questions of your engineers. In-depth technical questions are good, as well as abstract questions such as “what is the health of feature X” and “are we ready for tapeout” and “what worries you the most”. For more senior members, you can ask “where are we inefficient” and “what could we be doing better”. Figure out who can answer the questions and start looking for patterns of individuals whose answers make sense. Look at their work: for instance, read all bug reports, testplans, and written papers. Find out how their code reviews went. Get feedback from other managers and technical leaders both inside and outside of Validation. Experienced Designers can give you good insight on the technical and leadership capability of a Validator. Talk to the engineers individually and see what their general interests and career goals are, and provide them with examples of opportunities within their job and in the future. Begin providing both tactical and strategic coaching, and possibly point them towards experts to answer questions. When discussing short term tasks, tie those tasks to the long term objectives of the project, the team, and their career. Begin steering challenging problems towards them for them to resolve, and encourage them to be looking for challenges they see and to go address them.

To foster development of technical leaders, you must create an environment where constructively challenging the status quo is encouraged and does not carry repercussions. For instance, when following a process, team members may see inefficiencies in the process. If they can publicly challenge the process/policy in a controlled fashion, this will help increase the efficiency and the quality of those processes/policies. If they cannot, then they will violate the policy/process quietly, and when many engineers do this they will likely do it in many different directions, leading to complete lack of standardization. Creating an environment for constructive challenging is tough – engineers will often *un*-constructively challenge the status-quo, thus requiring coaching by management on challenging constructively versus un-constructively. Furthermore, some engineers like change simply for change’s sake, and in the worst case, some engineers simply want to be different from everyone else. These examples require solid coaching for the sake of the team as well as for the individual’s career in engineering. Validation staff members need to be vigilant that they are not rewarding engineers that are successful via subverting standard methods and practices.

To further spur development of technical leaders, you might consider the following options when prospective technical leaders are in the position to make a technical decision. First, instead of telling them which direction to take, use questions to help them think through their issue and come around to making the right decision on their own. This is tough and requires practice! Second, if it is a contained situation and you think that the engineer can benefit from it, consider letting him make a decision even when it has a high probability of failure. Failure is an excellent tool for learning in an environment where engineers are not punished and can reflect on the original issue with hindsight. Third, if the best decision (in the manager’s mind) does not really make a major difference over the engineer’s choice, then go with the engineer’s choice. Sometimes losing a small

percentage of goodness, especially if it can be corrected later, is worth it if it inspires engineers to feel ownership and responsibility for their methods and tools.

Encourage your prospective technical leaders to reflect on their previous actions and decisions. Ask them how things could have been done differently, and have them propose improvements to their own actions or methods of decision-making. Thoughtful reflection maximizes learning from their own history.

Some situations discourage technical career growth. In groups with a high percentage of managers, there is an appearance that the best career direction is in management. This appearance comes from multiple reasons – firstly, management-heavy groups tend to be that way due to the lack of technical leads, and thus engineers will see that management is making all decisions and driving the direction of the team (both important tasks connected to career growth). Secondly, management-heavy groups tend to be more vertical and have greater movement in their management ranks. While grade change is typically secret, there is no secret to managers taking on larger teams or moving up within a vertical organization, and thus it is seemingly the only visible career movement. Manage this situation carefully, since there may be times that your team requires extra management. The goal is to switch from a high percentage of management to low as soon as it is practical (see sections 6.1.3 and 6.2.2).

6.2.7 Utilizing technical leaders

Technical leadership permits project execution with somewhat less bureaucracy. It also allows management to transfer some tasks and responsibility to non-managers. Here are some examples:

- Creation and delivery of training (methods, tools, microarchitecture, Validation principles)
- Cross-project methodology convergence
- Primary contact for customers, such as Design, Architecture, other Validation teams.
- Solving major problems
- Driving task forces
- Creating PORs and team direction (possibly within constraints set by Validation staff)
- Team decision making and priority setting
- Building the team (interviewing, GPTW activities, etc)

There are a couple of balances that shift as technical leaders develop. When technical people are not senior, the manager should rely on their engineers' technical information, but the manager makes the decision. As the technical leader develops, she will make decisions – within the bounds that managers set.

If you are doing your job well as a manager, and people are becoming more senior, you should have to spend less time managing them, and you should be able to move on to other areas or take on further tasks. There is a management balance reversal that takes place: you transition from engineers being dependent on their managers for success to where the engineers begin guiding their managers. This in turn makes managers more effective because their people will bring “high value” problems for them to work on and solve.

The plus side of utilizing technical leaders properly is that all of these usages: decision making, handling big problems, driving task forces, are exactly the practices that will encourage the technical leads to remain in Validation and remain on the technical career path.

6.3 Validating a project

The project stages offer varied challenges for Validation to undertake, as well as opportunities to further enhance the team.

6.3.1 Planning for a project

To prepare for a project, you need to assess the challenge in front of you and assess the assets that you have to tackle that challenge. The challenge is the overall workload of the project: the quantity of overall change and the number of major new features. Your assets are your team: its strengths and its weaknesses and any special skills it might have. As you analyze this information, the first decision you should make is how much ‘project management’ needs to be done at this early stage. For instance, if it looks like the early project targets are well beyond the capabilities of your team, it is important to drive scope control within Architecture and Design to bring the workload within reason.

6.3.1.1 Funding early Validation tasks and seeding Validators strategically

Early on within a project, you should decide where to fund pathfinding and spearheading efforts. For instance, changing the TE software language is something that requires major early funding. Even without changing the TE language, you should invest heavily in understanding the TE environment if you are inheriting it from a prior project without local expertise. Another common example: if the project is likely to take on some brand new technology, seed a technical expert with the Architecture and Design group early to enforce a high level of goodness in the new technology as well as to prepare Validation methods and collateral to target this new technology.

As you begin to position Validators in various clusters/units, you need to weigh their individual wishes (wanting to stay within their current expertise envelope or wanting to spread out) with the needs of the project. For instance, you may need to migrate many Validators from one area to another due to where the majority of design change is occurring. You should also seed technical leads by finding tough areas and putting them there early on. Spread them out as well, to maximize their effect on your team and the project. Strive to get experienced Validators into all sub-teams.

While brainstorming critical areas that will require extra Validation focus, also consider what areas have a high probability of having late-project major additions. For instance, if a feature gets rejected during Tech Readiness because there is a question regarding its importance at launch time, it may likely reappear later when those questions are answered. Choose one or two features that are likely candidates as late addition, and allow the Validation teams covering those areas a little more headcount slack than the other teams. You cannot do a lot, but sometimes simply understanding what could be coming gives you a head start.

6.3.1.2 Choosing methods and tools

When choosing methods, carefully review the methods of other Validation teams within Intel; beware your team having a “not invented here” syndrome. Critique what was planned on your last project and what actually happened, and why. Recognize if you can fix or avoid things that went wrong and if you can ensure and magnify things that went right. Remember that change for change’s sake is not good, and that if your team knows how to execute one methodology with high efficiency, moving to another methodology with only a small additional benefit is not worth it.

After the analysis to choose methods and tools, write down what you learned along the way, what you have chosen, why you have chosen it, any assumptions you are making, indicators to measure success or failure, and backup plans if they are necessary. This write-up is a great way to explain your whole decision process to future projects. It is also a great way to remind you of the process and reasoning a year later when you are knee-deep into the project.

6.3.1.3 Enforcing rigid initial standardization

As you plan your project, standardize everything to the greatest degree possible. Rigid standardization is the key to efficiency and to innovation. As you create your methodology and tool PORs, standardize them as well. Knowing that you will have to make tactical decisions as the project progresses, create a framework within which to make those decisions. For instance, “if coverage monitor coding and coverage does not meet a certain point by this milestone, then we will sort the conditions into two priorities, and allow the lower priority condition analysis to continue past tapeout”. Another example of a prepared tactical direction is: “in the event that layout thinks they can hit a very early tapeout, we need exactly 12 weeks’ notice prior to the new tapeout date so that we can shift focus to reset and DFX testing and finish those in time for the early tapeout”. Of course, you cannot perceive every scenario; this is why you create rigid standardization up-front: because you will have to loosen up and have special cases as the project goes on. In fact, there will come a time near tapeout when your attitude should be “do whatever is needed to get things done”. If you start with loose methodologies, “do whatever is needed to get things done” will be complete chaos. If you start rigid, then “do whatever is needed to get things done” will still be performed in a structured, disciplined manner. This is something for which to strive. Then, when you start your next project, you revert to rigid standardization yet again.

6.3.1.4 Choosing indicators

Project preparation includes planning the indicators you will use on the project. Indicators are used for two main purposes: to provide feedback to project management on the progress of your team, and to motivate action within your team or from your customers and suppliers. For Validation progress, you want to choose indicators that turn abstract confidence into a concrete measurement of health and progress. Progress towards testplan completion (tests written, proofs completed, coverage hit) are indicators used to measure progress. To drive behavior in Design, the HOM (Health of the Model) indicator is excellent.

If possible, choose the minimal set of indicators to best measure your progress and drive the most important behaviors. One great example of this is the coverage indicator used within DDG. DDG IP Validation found less need to track testing and checking status: Validators naturally enjoyed

writing tests and checkers, and it did not require extra management. Furthermore, missing tests would often show up as missing coverage. Coverage, however, required an extra level of discipline, and therefore was monitored closely by IP Validation staff. Thus, it was an indicator used to drive Validator behavior to balance the time spent on testing and checker writing with coverage driving, and it was also used as an indicator of overall Validation progress for use internally and by project management. The assumption here is that the testplan that the coverage is based on is of high quality and has been reviewed by Design stakeholders. Basing indicators on poor quality coverage is a recipe for disaster.

Standardization of indicators is important. Within a project, indicators for multiple teams doing similar work should be the same to aid in comparison of progress and quality, and thus to help in load balancing. From project to project, indicator standardization allows Validation's customers and suppliers and project management to become comfortable with the indicators and to understand what the indicator means from their point of view. Across projects and teams standardization is also important for project comparisons and to identify trends (bug rate increasing, etc). Consider using a tried and true indicator prior to choosing something new simply for the sake of continuity.

Remember, you are using indicators to help manage your team and the project – do not let the indicators manage you! There have been times when it made sense to ignore indicators, but this is rare, and it is really an art. For an insanely long but thorough chapter on indicators, please see [Indicators](#).

6.3.2 Following the plan

Beyond managing your employees or doing technical work, Validation staff members have a massive load of work during the project to ensure Validation success. The following policies and BKMs support Validation staff in accomplishing their task.

6.3.2.1 Validation staff meetings

The Validation staff should meet weekly. Within this meeting, all sorts of things will get reviewed and decided by the staff members. Execution progress is reviewed weekly. Other process items like choosing awards are done at less-frequent intervals such as bi-weekly. Validation staff should question each other's indicators and dig deep into the details – this is the way to level-set everyone and get understanding of everyone's true progress. This is also the place to discover global trends, where multiple teams see the same issue before any one of them individually decides to raise a flag. This facilitates early identification and solving of problems.

The Validation staff meeting is the place for tactical decision making; it allows these types of decisions to be made with consistency across the team. If one of the sub-teams wants to have a special case and do something in a non-standard way, the Validation staff meeting is the place to review it. The Validation staff members can decide that the special case is warranted, or tell the team to work harder to meet the standard. In some cases, all sub-teams may want to switch to the new process or policy, for example: “The OOO is abandoning perfmon testing until Execution phase. We should all do this”.

This meeting provides the opportunity to treat Validation as an organization: to scrutinize the Org Health Survey results and take action, to do BKM sharing and brainstorm solutions to sub-team problems, to fund global needs and horizontal projects, and to drive process and tool change pioneering. Most of all, it gives every Validation staff member responsibility over the entire Validation program. This responsibility makes tough decisions, such as load balancing, possible.

The Validation staff meeting is also the place where technical experts can make proposals for new methods or tools that will influence the rest of the team. Validation staff can then decide to fund the proposal, request more data, or redirect the technical expert down a different path.

For those sub-teams within Validation that are large enough to warrant a sizable management staff (such as IP Validation), all of the above applies to those staff meetings as well. For instance, the IP Validation staff meeting is the place to discuss load-balancing within IP Validation, special cases, etc.

6.3.2.2 Regulating schedule pressure

During TR and FED (see [The Life of a Project](#)), Validation will have more work than it can possibly do, and during FED you will likely feel way behind since you will see a project's worth of work in front of you and because you will be in a race with Design (new RTL coding versus TE and exercise of the new code). Manage this carefully. Though it is important to get through FED quickly and with high quality, you must carefully monitor individuals' workload. Keep it light during FED, because you are going to ramp up to a much higher pace in Execution. The Execution phase is the one where high pressure can be placed on the majority of the team for long periods of time. This is the point in the project where many items have to come together for success, and sometimes this takes excessive work on the part of many groups to achieve. Keep in mind, some sub-groups, such as TE coders, will have their high pressure time during a different stage of the project. For TE coders, the end of TR, all of FED, and the beginning of Execution is typically their frantic stage.

One extra note on this topic: when your project is farther behind, it requires *more* bureaucracy – which is counterintuitive. Bureaucracy is work, and adding more work to a project behind schedule seems wrong, but it is necessary. When behind, all levels of management require a much finer granularity of information with which to make decisions – at the company level, the project level, and the Validation level. Do not fight it: offer the most correct information as efficiently as possible. Those who dally or misrepresent their status eventually pay a price for it. The cost of not having detailed, accurate data is poorly made decisions, leading to schedule slips, extra steppings, or cancelation. Be detailed and forthright with your status.

6.3.2.3 Quality tenets to strive for

Some quality tenets must be enforced when executing a project. One such tenet is to drive out failures from your proofs and test runs as early as possible. If failures are caused by RTL bugs, you want to get them fixed as soon as possible to search for bugs hidden underneath. If your failures are due to test environment bugs or test problems, take the time to fix them immediately. Though time in Execution is a premium, the amount of time lost in debugging false failures easily surpasses the time needed to fix a bug, and false failures add risk that Validators maybe eventually

ignore a failure from an RTL bug. Just as importantly, when you enter Post-Si steppings and tick projects, your Execution practices must be polished. You should be able to run a regression where every failure is important, proofs are stable, and you should be able to gain coverage quickly. Intel requires great precision and quality in its tick projects, and thus you need to have Execution fine-tuned as you transition into your tick projects.

Along with eradicating false failures, strengthen your turnin gating regressions to be solid. As the project progresses, gating regressions should continue to raise the bar on the quality of code turned in. This is critical to avoid validation setbacks on the road to tapeout, and especially critical during stepping and tick validation where far less scrutiny is applied by Validation.

Validation staff should also proactively analyze where the Validation gaps are surfacing based on bugs or escapes to higher level DUTs or to Post-Si. Using this data, Validation staff can see trends and plug any gaps that they see as early as possible. In the situation where Validation is unable to respond to the gap, Validation staff should inform Post-Si Validation about any risks that exist due to these gaps.

6.3.2.4 Load Balancing

It is imperative to know the status of your teams. What is done, what isn't done, are tests running with *everything* enabled, what is the failure rate with everything enabled, is everything being debugged, what percentage of time is being spent on debug, what checkers haven't been written or enabled, what are the major coverage holes, what are the minor coverage holes, etc. And you need to know this for every single area. Moreover, you need to understand things beyond the indicators – for instance, are there major flows that are not being stressed? Are any workarounds still enabled? What are the gut feelings of your Validators? If you are lucky enough to have worked with some of your Validators for multiple projects, how does their concern match with that of the previous project, and how did *that* project turn out?

With this information, you can begin thinking about how to load balance to achieve consistent quality across the CPU. There is no need for perfect equality – even 10% or 15% difference in quality is OK, but since absolute quality is hard to measure, be careful. If all of your teams are coming up short, you might still consider shifting effort around in order to get similar quality in all areas or to get some areas especially clean. For instance, if you are entering Post-Si with sub-par quality, the Post-Si team will get the most distance with the CPU if reset, analog, and DFX capabilities are working. From a functional logic point of view, Post-Si Validation always trips up over front-end and ooo cluster bugs first, so driving a little extra quality there could help Post-Si progress.

When choosing how to load balance, take into account the typical factors: the greater distance you have to move a Validator, the greater his ramp time will be. If you move a senior team member, the ramp time will be less, but the damage to his original team will be greater. The farther from tapeout, the more return a relocated Validator will accomplish after his ramp. If some of your Validators will 'disappear' during Post-Si due to helping with Post-Si debug, you might not want to shortchange their teams since they would then have multiple losses. Along with all of this, you always want to consider the future. You might want to make a move permanent – if the area was behind leading to A-step tape-in, it might be understaffed for all future steppings and ticks as well.

6.3.2.5 Tape-in strategies

As you approach tape-in, assess the relative progress and quality that each team is likely to reach at tape-in. Be forthright with project management; even if you change your tape-in goals, give your status with respect to your original goals as well as any new goals. For instance, “cluster xx has hit 95% of its high priority coverage and is at 65% of the full coverage”. This illustrates your solid progress to your new goal and yet reminds them of the risks involved and the work to do in the future. The same goes for the Post-Si Validation team; somewhere near tape-in hold a hand-off meeting with them, explaining Pre-Si status and detailing what hasn’t been done or what has been done with less quality.

Being forthright in this manner may be your hardest job. Nobody likes to admit to doing a low quality job. It is your duty to the project and to customers such as Post-Si Validation to be candid; not doing so is doing them a disservice and can have disastrous effects on the project.

If you have chosen to take back-offs prior to tapeout, you must still validate to 100% quality as soon as possible. Many Validators think that once silicon is in Post-Si Validation’s hands that Pre-Si Validation has little to offer to the project. This is absolutely wrong. The fastest PRQs are achieved when Pre-Si drives to completion, even if done shortly after tape-in. Some gnarly microarchitecture or mixed-signal bugs that Pre-Si can find may not show up in Post-Si until near PRQ. ‘Easier’ bugs that escape to silicon are likely to cause Post-Si Validation some level of struggle to make progress on that stepping. This makes it even more imperative that the Bstep silicon that they receive is top quality, lest the schedule to PRQ be at risk. With that in mind, Pre-Si Validation needs to focus on getting all portions of the chip up to 100% fully validated. This will require juggling multiple jobs at once, as Pre-Si Validators should also be involved in Post-Si debug and investigating/validating Post-Si bugs and fixes.

Your initial schedule should never plan to push some validation tasks off, or “snowplow”, until Bstep; this should be taken as a back-off and only when all other strategies fail. Pushing any validation to Bstep adds risk to the program by adding instability to A-step Post-Si Validation. Post-Si Validation has an incredibly aggressive schedule, and if A-step is subpar, it immediately puts PRQ schedule at risk.

6.3.3 Awarding good work

Recognition is important. The more you care about your team, the more you will want to get this right. You need to award the obvious things such as brilliant ideas, quality work, or going above and beyond. You will also need to award the non-obvious things such as informed risk-taking that does not pan out or doing something in the name of standardization and long-term quality. Knowing what kind of recognition motivates individuals helps. Some people are motivated by public recognition, some by personal acknowledgment.

You should use awards to advertise important Validator behaviors or habits that you want others to emulate. Bug/Behavior of the Week (BOTW) is a good example where Validation (or sub-team) staff can showcase a Validator for high quality work they have done. Examples of BOTW are: finding bugs by inspection, going the extra mile to prove something was a bug when told by experts that it was not, writing an excellent bug report, investigating and solving an environment problem that is affecting the team, or anything else that you would encourage others on the team to emulate.

Some Validators excel at the everyday tasks in Validation such as testplan writing, proof writing, test writing, protocyte coding, etc. When you see significant or speedy progress made on one of these tasks, you may want to give an “excellence in execution” award to highlight the recipient’s effort. Awards do not have to be monetary. A simple verbal acknowledgement of a job well done goes a long way, and in some cases, carries more weight than a formal award. When you see a great performance, applaud it right away.

There are numerous cautionary notes to giving awards. For instance, avoid giving awards based only on the overall impact to the team and/or the project. This can lead to senior people in the team, or team members who own tools that are particularly fruitful, getting the vast majority of the recognition. Rather, give awards based on the actions of the individual. Did the individual take a well-informed risk? Did they go above and beyond to get the job done right? Were their actions something you would like to see more?

Avoid giving formal awards to whole teams; it inevitably devalues the award due to disparity amongst those receiving it. Some of the recipients may feel that others were not as deserving in the award. The inverse is also true: when giving awards to too few people non-recipients may feel like their contributions were neglected. This is a delicate balance. When in doubt, lean to the direction of giving formal awards to single individuals or the select few that were instrumental in the good deed.

Avoid giving awards that might de-motivate your team. For instance, certain practices devalue an award: giving the award in a circular pattern throughout your team, giving awards for time on the job, or giving too many formal awards. Take care that awards pass the “snicker” test. Do not give an award if it is likely to cause the audience to ridicule it. Sometimes it may be appropriate to give recognition privately, such as in a 1:1 or via email. If you feel that public recognition would not be appropriate (maybe it fails the snicker test, or you feel that another person in the team would feel demotivated by the recognition for whatever reason), it does not mean the recognition should not happen. Instead, do so in private. It is incumbent upon leaders in Validation to keep their team members feeling valued and motivated.

6.3.4 Managing the non-Validation side of the project

At the beginning of the project, establish the roles and responsibilities of Validation and your customers and suppliers (Design, Arch, Post-Si Val, etc). Do this in coordination with your customers and suppliers. In less structured settings, your customers/suppliers may find this unusual. In settings where Validation does not command respect, they may even be inclined to disregard this suggestion. This makes it even more important. This task forces the teams to come to agreement with one another, or in the worst case, it exposes a major inter-team disconnect that requires immediate attention and possibly moderation by project management.

Begin this task by defining who your customers and suppliers are, the collateral and services that each provides, and the expectations, SLAs, and how to track such things. Example roles and responsibilities that you want to address are: division of ownership of test writing, checker writing, and debug through different stages of the project. Get this right early on; misunderstandings can create bad feelings when the teams are under real pressure. And most important of all, the expectations you setup had damn well better be something that allows the team to grow, establish respect, expertise, and add unique value.

During this early part of the project, and later when pressure has risen, Validation staff members will often be a buffer between Design and Architecture and individual Validators. In a mutually supporting environment, Validation staff members can seek out mutually beneficial tweaks to each team's roles to speed through some phase of a project or to deliver some feature sooner and with higher quality. For instance, load balancing of debug: if Validation is ahead, they may take on a greater debug load, and vice-versa. In a negative, less respectful environment, Validation staff members need to be tough: they should not, for instance, let Design dictate Validation's priorities. Nonetheless, you must continue to look for ways the teams can work together for the good of the project. Regardless of how new or inexperienced you are as a Validation staff member, you will need to grow a spine and get tough fast in this environment. There is no excuse for your team to be run over by another team, regardless of the situation. Do not let that happen. It is OK (and important) to admit when the Validation team or sub-team is behind schedule – but spend your time working constructively with the other teams to get back on schedule, not to defend the position that you are in. Simply put, this is not easy.

Work hard to gain a rapport with project management. Validation is in a unique position as the ultimate gate to quality as well as typically being the long pole to tape-in. In this position, you often have the best perspective when some part of the project is not coming together correctly or on schedule. If you have a good relationship with project management, you can give them this information to help them take early actions. When presenting project-related problems to management, present the information within a framework that contains the situation and its history, the likely repercussions in terms of schedule, risk, and effect on Post-Si Validation, and possible fixes and their costs. This will help project management understand the situation swiftly and hopefully take appropriate action.

In addition, Validation must provide input to project wide decisions and status. If new features are being discussed, assess and provide their cost and risk to the project. If the project is considering an earlier or later tape-in date, provide the downsides, upsides, and what Validation can reorder internally to help. The same goes for adding another stepping or tick-product. Validation has special knowledge and skills to offer the project, and thus should act accordingly.

6.3.5 Managing positively versus managing aggressively

Depending on the current status and situation of your team, you may manage them in vastly different ways. The same is true for the outside world: your suppliers and customers.

When your team is behind schedule yet working hard and doing the right things, you will want to switch to a mode where your management of the team is very positive and where you are always looking for ways to offload non-essential tasks and remove roadblocks for them. In that situation, you might manage your customers and suppliers in a very different manner, much more aggressive and tough, where you push back against new requests and you quickly squelch anyone's negative opinions of your team's progress.

On the other hand, when your team (or individuals) are behind or not meeting customer SLAs due to poor engineering or relaxed work ethic, you might take a much tougher management approach with your team and become involved at a much finer granularity with their daily work and decision making. In this situation, you will often seek input from customers and suppliers and find ways to better measure your team's progress in meeting its goals.

6.3.6 Transitioning from project to project

Validators rarely leave the Validation team in the middle of a project. Perhaps the busy schedule precludes looking for other opportunities or even thinking about such a change. However, the transition between projects offers a *relatively* high amount of relaxed time. This time is necessary since driving a team at full steam for too long is likely to burnout the team. During this time, individuals often take the time to think about their life, their careers, and opportunities open to them. This is healthy. The downside is that it also opens the door to a large percentage of your team leaving to work elsewhere in Intel, outside Intel, or even return to school.

Validation can use this transition period and individual “soul searching” to its advantage. Engineers love learning and challenges. During this transition time, Validation staff needs to offer a large set of varied opportunities to its engineers. First off, now is a great time for Validation career-path discussions and technical leadership chats. This tells Validators that it is OK to start thinking beyond their current tasks and that there are opportunities within Validation for them to grow. More importantly, it subliminally tells them that Validation leadership *wants* them to grow, and that is a major factor in an engineer’s work satisfaction.

Secondly, now is a great time to offer voluntary opportunities. Set up technical talks with outside speakers giving talks about things connected with or having no relationship whatsoever with CPUs. Offer tasks that previously fell below ZBB – tool work and efficiency experiments. Drive efforts to find out exactly what all of the other Validation teams within Intel are doing. Publicize opportunities to write technical papers. Poll your team for some of these ideas and more.

All of these ideas facilitate keeping your Validators interested, spurring their imaginations, and showing them that Validation is a great place to have a career.

6.4 Special conditions faced in Validation

CPU projects and the nature of Validation create some unique or extraordinary engineering conditions; these conditions create or magnify problems for Validation staff. The following sections explore some of these conditions, their problems, and how Validation staff members engage those problems.

6.4.1 Extremely long project lifecycles

Major CPU projects tend to take 5 or 6 years from the beginning of Pathfinding to CPU launch. Validation has very limited engagement during Pathfinding, much more in Tech Readiness, and full engagement during FED and Execution, as well as during the Post-Si and stepping phases of the project. Due to project overlap, Validation begins a major project family approximately every 4 years. See [The Life of a Project](#) for more details.

Engineers need data to compare methods and tools. However, the project length makes experimentation in Validation damn near impossible. Proper experimental procedure calls for changing one variable and rerunning a test. If the test duration is a full project – 4 years – then running 5 tests would take 20 years! Moreover, each project introduces so many radical new

features and methods that it is hard to attribute change to any one given factor. So how do you introduce change into Validation methodology with less experimentation?

A proposed methodology change that precludes experimentation should be subjected to rigorous thought experiments, a form of brainstorming. Use ‘thought experiments’ covering the span of a project or even multiple projects to speculate positive and negative results. Validation staff members should debate and contest these results, and eventually use this speculation to make strategic decisions. To accomplish this with confidence you need experts on validation: deep historical knowledge of previous project execution and results across many teams and deep knowledge of the quiet relationships/interworking of disparate portions of Validation methodology. Along with this, it helps to have perspective, foresight, intuition, and a firm belief in fairly arguing both sides of a debate. Moreover, you need as much data from other projects with similar experiences as possible. If you do not have these kinds of experts, then find them somewhere! If after doing the thought experiments and debate a team decides to proceed, they should then do a pilot in one area to gain real-world experience on the new approach, before betting the whole team and project on it. Potentially keep the old way of doing things enabled as a fall back. Set up acceptance criteria on whether/when to deploy or expand.

Another inherent downside to doing so few projects (over time) is that the total number of experiments run is limited. This reduces the search space, and increases the possibility of missing important breakthroughs that could fundamentally improve Validation quality and efficiency. Therefore, place extra emphasis on fully analyzing options and executing the right experiments. Furthermore, if you are using incremental change to converge multiple groups’ methods and tools, beware! Incremental change of even 20% per project (seemingly a lot) will take 20 years to complete – and that is assuming no backsliding!

With the limited ability to create a large quantity of experiment data, it is imperative to utilize experiments run in other projects throughout Intel. This takes discipline: many engineers’ natural inclination is to disregard others’ data. This is exacerbated by the massive differences in projects across Intel as well as the differences between the methodologies of the teams executing them. However, there is information to be gleaned from other projects, regardless of how different their teams and projects are. It takes a disciplined team to search these out and take the data for what it is worth. This is one more expectation to place on your technical leaders.

Long project life cycles also magnify another engineering problem: late additions to the project. It is very tough to know 5 years ahead of product launch exactly what consumers will want and what the industry will support. This creates the need to have course corrections midway through project execution. These course corrections may require major feature changes or additions, and Validation is likely to be heavily affected. It is important for Validation to understand this environment, to prepare for such eventualities, to give good feedback to project management when these changes are proposed, and to deal with the change as fluidly as possible. Section 6.3.1.1 explores options for preparing for this kind of issue.

6.4.2 Validation output is confidence

Design criteria for tape-in are understandable: layout must be complete and timing must meet the spec. Validation, however, has a much less tangible output: confidence. Throughout the execution

of the project, Validation must deliver an indicator of confidence in the design and a projection of completion date.

Validation must change this vague output into something concrete that truly illustrates the health status of the design and the progress rate of the team. Validation should choose indicators that are meaningful to themselves and customers and drive the correct behavior within the team and with customers and suppliers. This topic is explored in greater depth in section 6.3.1.4.

6.4.3 Cross-site

Cross-site work is very common in the industry. It causes inefficiency by inhibiting communication. Validation, which is a communication-heavy discipline, suffers accordingly. Validation staff must recognize the inefficiencies this creates and try to choose roles for the cross-site teams that require less frequent communication. At the same time, you also have to teach your teams the best-known methods for cross-site communication – the proper way to create training material, how to run meetings, how to ask questions and make requests, etc. This is not easy, and the importance of this effort should not be taken lightly.

One extra note regarding cross-site: Validation typically chooses to seat Validators together instead of seating them near their respective Designers (see section 6.2.5). Cross-site is one circumstance that Validation violates this rule. If given the choice, it is preferable to have Validation and Design for the same block located within the same site even at the expense of splitting Validation across sites. However, within each site Validation should still be co-located.

6.4.4 Failure is an option

The DDGteam has not yet failed on a CPU. With each passing project, Intel relies more on the team to produce and launch a quality CPU on schedule. This responsibility has increased to the point where most team members have taken it to heart that failure is not an option. There is a benefit to this: team members will throw their souls into making these projects come to fruition. There is a downside as well: engineers become overly averse to risk and potential waste.

Risk aversion is death to a company pushing the envelope of innovation. Within Validation, it stifles innovation and efficiency gains, which eventually increases the cost of validating the CPU, and makes the final product less profitable and compelling. If the capabilities of Validation are not scaling as fast as the projects themselves, Validation will become the bottleneck that will push back on features that could make the product more profitable and compelling.

Aversion to potential waste sounds like a positive attribute: aren't we striving for efficiency? Yes, but innovation requires failure. Wary of wasting time, DDGengineers might stop experiments early, as soon as there is a hint of failure. Often, experiments need to continue through to the end to understand their true ramifications. Not doing this leads to less innovation and eventually stagnation.

Validation staff must counter this trend by encouraging risk taking, performing rigorous tracking of new method/tool performance, and having a backup plan in case of failure. Moreover, Validation staff must be cognizant that successful projects are not necessarily indicative of world-class methods, and therefore should always keep their eyes open wide to new ideas.

7 Summary

The quality of Validation leadership, more than any other factor, will determine Validation's success in executing projects and in developing their team. Validation staff is faced with some unusual problems, but some of the ideas in this chapter will help good leadership become great leadership, execute quality projects, and create an awesome Validation team.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 6

Stimulus

By: [Michael Bair](#)

1 Abstract

Stimulus is the input driven into a device under test (DUT) to exercise the design, typically with the intention to find bugs. Stimulus can range from targeting an exact condition with a directed test to running completely random tests. Good stimulus is measured by the amount of functionality exercised versus the investment required by engineers to write the test and debug failures, as well as the compute investment. The requirements of the project and of the various [Validation Disciplines](#) often dictate the types of stimulus that will be written.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/27/2004	First release of document	Michael Bair	Matt Kupperman / DPG-N uAV MWG
1.1	4/1/2005	Completed L1 review revision	Michael Bair	Matt Chidester
1.2	5/16/2005	First L2 Revision	Michael Bair	Paul Schwabe
1.3	7/30/2005	Revision follow first L3 review	Michael Bair	Paul Schwabe
1.4	6/6/2011	Additions and update of content	Erik W. Berg	Michael Bair
2.0	9/12/2011	Update from "Random Testing" to "Stimulus". Reordering, massive reduction in size, and general cleanup.	Michael Bair	Matt Kupperman / CCDO Val Staff
2.1	8/28/2016	Additions/changes from Adeboye Oshin from DDG Art of Val sessions	Michael Bair	Mike Miller/Michael Bair

3 Contents

1 Abstract.....	141
2 Revision History.....	141
3 Contents.....	142
4 Purpose.....	144
4.1 Why do we need this chapter?.....	144
4.2 What does this chapter cover?	144
4.3 What does this chapter not cover?	144
5 Background Concepts	145
5.1 Test Environment.....	145
5.2 Bus Functional Models	145
5.3 Test Generator.....	145
5.4 Directed	145
5.5 Random	145
5.6 Seed	146
5.7 Constraint/Bias/Weight/Knob/Parameter	146
5.8 Coverage	146
5.9 Template.....	146
5.10 Sequences.....	147
5.11 Algorithmic Test Generators	147
5.12 Random Instruction Test (RIT).....	147
6 Stimulus.....	147
6.1 Directed Stimulus vs. Random Stimulus.....	148
6.2 Randomness Spectrum	150
6.3 Dynamic random vs. randomly created static instances	152
6.4 One vs. many stimulus generators	153
6.5 Differences between Pre-Si and Post-Si test generation	156
6.5.1 Failure detection	156
6.5.2 Test build time vs. test run time ratio	156
6.6 General desired properties of stimulus	156
6.6.1 Reproducibility	156
6.6.2 DUT-specific configuration.....	157

6.6.3	The right level of randomness control	157
6.6.4	Feature cross-product.....	158
6.6.5	Mutually Exclusive features	159
6.6.6	Changing contexts during generation	159
6.6.7	Overstressing vs. relaxing.....	159
6.6.8	Quiescing vs. relaxing.....	160
6.6.9	Monotony monotony monotony!.....	160
6.6.10	Ease of use.....	160
6.6.11	Debug-ability.....	161
6.6.12	Extendibility for new features	161
6.6.13	Efficient setup	161
6.6.14	Ability to work in multiple environments	162
6.7	Effective use of a random stimulus generator.....	162
6.7.1	Never turn off the background noise	162
6.7.2	Do not turn everything on full blast	163
6.7.3	Maximize the use of the collective knowledge of your peers	163
7	Summary.....	164
8	Future Work.....	164
8.1.1	Distribution of test between the test environment and the Test	164
8.1.2	Types of randomness	164
8.1.3	Biasing and User Input.....	164
8.1.4	Stimulus generator Outputs	165
8.1.5	Genetic feedback.....	165
8.1.6	Input coverage	165
9	References.....	166

4 Purpose

4.1 Why do we need this chapter?

Stimulus is the first pillar of validation. Stimulus currently provides the most efficient and cost-effective method of finding bugs (as opposed to finding all bugs by inspection or formal proofs). Also, Stimulus is arguably the only Validation Pillar that can stand alone; you can find bugs with tests even without checking and coverage (test hangs, for example), but checking or coverage without stimulus are useless.

Writing stimulus, or tests, has historically been a large percentage of Validation time. The test writing effort would have grown dramatically with the ever-increasing complexity of hardware if it were not for multiple efforts to make test writing more efficient. One of those efforts was to move as much testing to *random testing* as possible. Other efforts include treating test infrastructure as important software – software that should live multiple generations, be easily maintained, and be extendible as new features are implemented.

The ideal solution for test infrastructure or a random test generator may never be reached by any team; moreover, how each team prioritizes its tactical and strategic stimulus requirements may vary. This chapter will explain the tradeoffs, nuances and other good stuff learned over the course of many projects and in varying circumstances.

4.2 What does this chapter cover?

This chapter covers the tradeoffs in the range between directed tests and random tests. It delves further into the properties of good random test generators, tradeoffs between different styles of generators, and pitfalls to watch out for when coding a random test generator. The chapter serves as an introductory tool for writing a test generator and uncovers the issues behind some of the “Holy Wars” between different test generator camps.

The chapter also covers ideas for how to use a random test generator. This might be the most useful portion of the chapter as there are already many test generators in existence. So, how to *run* them might be more important than how to *write* them.

4.3 What does this chapter not cover?

This chapter does not cover how to write testplans or test conditions for directed or random testing, nor does it cover the exact depth of conditions that should exist within a testplan or coverage monitor.

This chapter does not cover a step-by-step process of how to code a random test generator or what language or environment to use, nor does it tell you how to choose between a stand-alone test generator, a test generator that is part of a test environment and made up of sequences, or something in between.

It should be noted that this chapter concerns itself mostly with discussing stimulus as it pertains to *dynamic* validation, not Formal Verification. Formal Verification also has stimulus, checking, and coverage as the 3 pillars of Validation, but they are of a different fashion. For example, a SAT

check has explicit checks (like checkers), the tool itself runs an infinite set of stimuli (up to a bound), and the coverage is 100% minus any assumptions created by the Validator. For more information on Formal Verification, see the [Validation Disciplines](#) chapter.

5 Background Concepts

5.1 Test Environment of Cluster Test Environment

A test environment (TE, sometimes called a Cluster Test Environment - CTE) is software that connects to a DUT and provides stimulus to that DUT. The stimulus may be directed or random. Test environments take input from the user in the form of biasing, hand-written sequences, or possibly from test generator output and applies that input to the DUT. For more information see the [Test Environments](#) chapter.

5.2 Bus Functional Models

A Bus Functional Model (BFM) is the functionally equivalent implementation of a device used to generate (or respond to) stimulus at an interface (in lieu of real design). There are various reasons to utilize a BFM including speed, stability, and familiarity across generations of product development.

5.3 Test Generator

A test generator is stand-alone software that is capable of creating multiple tests. This chapter focuses on random test generators that use random choices to create different tests every time they run. Outside of this chapter, you might encounter test generators that are written to reproduce the same large set (for instance, thousands) of directed tests every time it runs.

A random test generator produces a test instance which can then be run on a DUT. The test instance may still include some randomness, possibly provided by the test environment.

5.4 Directed

Directed, within the scope of this document, indicates that everything is pre-defined. The exact same events and results should occur every time.

5.5 Random

Random, within the scope of this chapter, indicates that choices are being made, possibly within some bounds and with some distribution, where the outcome might be different every time and does not follow any pattern.

5.6 Seed

A seed is an input parameter to a random test generator that sets up all of the random choices within the test. Given the same set of input parameters and the same seed a random test generator should reproduce the same test (see 6.6.1 for more information on reproducibility). In general, a Validator can set up his input biasing with certain targets in mind, then run thousands of random tests with that same bias by simply changing the seed.

Specifically, here is the definition we use for seed:

SEED: An input used as the initial value for a pseudo-random number generation algorithm. Specifying the same seed to the same algorithm will produce the same sequence of random values.

5.7 Constraint/Bias/Weight/Knob/Parameter

Bias is the user input bounds and distribution values that direct how random choices will be made within a test generator or test environment. For instance, with biasing parameters set to 60% ADD instructions and 40% SUB instructions, the bounds are such that only ADD or SUB instructions will be chosen, that is, the test would never include a MULT instruction. The distribution of the choices would come out to approximately 60 percent ADD instructions and 40 percent SUB instructions given a large enough body of choices.

You will hear these terms used, mostly interchangeably, in phrases like “the user called for a large number of ADD instructions in his bias setup,” or “No ADDs should occur since this command line parameter has been turned off.”

User input bias is typically a form of constraint, but should not be confused with other constraints that a test environment uses for correctness and general operation. For instance, a typical test environment constraint might be *“if optype == read_type, opcode must be inside {READ32, READ64, READ128}”* which is not defining user bias but simply defining which opcodes to use when the user specifies optype “read_type”.

5.8 Coverage

Coverage is the concept of recording what conditions a test has hit to evaluate what the overall goodness a test suite has, or to look for deficiencies within a test generator or test suite. See [Coverage](#) chapter for more details.

5.9 Template

In this chapter, *template*, refers to a test “shell” that sets up a certain state automatically allowing the test writer to spend time writing the important portion of the test instead of the state setup. For instance, if you plan on writing a protected mode assembly test that runs a bunch of ADD instructions, you can use a protected-mode template to get into the right mode of operation, and add the targeted code, that is, the ADDs.

5.10 Sequences

A sequence is an ordered set of operations that can be used to target architectural or microarchitectural conditions. The ordering of the sequence may be strict or loose; loose ordering implies that the operations may execute in different orders. Sequences can also be used to set up certain conditions or state (for instance, a test generator may use a pre-defined sequence to get the DUT into a certain mode). The common paradigm with sequences is that as users learn more about what sequences of operations are needed to stimulate conditions within the DUT, they should code these sequences into the test generator's or test environment's sequence libraries, so they can be reused by multiple tests.

5.11 Algorithmic Test Generators

Algorithmic test generators create tests that follow one or more testing algorithms. They can vary from having little randomness to having as much randomness as possible and still fit within the algorithm. A good example of a testing algorithm is to have a 2 thread test, where one thread writes a pattern to memory and the second thread reads that pattern and checks it. Algorithmic test generators are much simpler to code and tend to find bugs very quickly, but they tend to lose bug-finding efficiency quickly since their operations are very constrained to whatever algorithm the test generator follows. Teams that use algorithmic test generators typically require many different algorithms to go after their coverage space.

5.12 Random Instruction Test (RIT)

A RIT test generator can be generalized as a test generator that creates tests *one operation at a time*. Since the RIT tool can choose any sequence of operations, the testing space that it theoretically can hit is infinite. The downside of this is that a simple sequence such as “20 nops in a row” becomes almost impossible to hit, since the likelihood of choosing the same instruction 20 times in a row (when you have hundreds of instructions to choose from) is very small. Most contemporary RIT tools have pre-defined sequences built in to the tool. These sequences allow the tool to efficiently hit conditions that require specific sequences of operations, while still allowing “absolute” randomness of operations around the sequence. This creates a middle-ground between algorithmic and RIT test generators that typically maintains the best of both worlds.

6 Stimulus

Stimulus is one of the Three Pillars of Validation, along with checking and coverage. (See [Introduction to Pre-Silicon Validation section: The three pillars of Validation](#)) Stimulus is typically seen as the base pillar of Validation – both checking and coverage are reliant on stimulus as an input.

6.1 Directed Stimulus vs. Random Stimulus

Stimulus ranges from cheap, throwaway tests to long-term investments in complex test environments and stimulus generators. Certain types of projects, or stages within a project, lend themselves to quick-and-dirty stimulus that will not-necessarily stand the test of time. For example, when first exercising a new design, simple directed tests that do basic operations have a lot of return for little investment. As the design stabilizes, their value typically drops significantly, and new tests need to be written to target conditions of greater complexity.

In areas where the condition space is finite, such as some aspects of [Design for Test Validation](#), the efficient way to hit this space, and to continue hitting this space, is to code the conditions into directed tests. These directed tests range from simple to complex.

For validation of general functional areas, the size of the condition space is functionally infinite. After a few rounds targeting the simplest conditions, continuing to attack the condition space with directed testing would be inefficient. Moreover, with a space so large, Validation cannot determine every condition requiring testing and checking. Thus, Validation relies mostly on random testing, with [Coverage](#) feedback. Coverage gives confidence that the random tests are hitting simpler conditions, and then it is up to the randomness (and the allotted compute cycles) to hit more complex conditions. This does not preclude the need (or desire) for directed testing; indeed there are many times when the ability to quickly create a directed or mostly-directed test is immensely helpful (reproducing a Post-Si failure, doing ‘what-if’ analysis, etc).

Random testing quickly becomes the stimulus of choice when targeting a complex condition space of near-infinite size (see Figure 9).

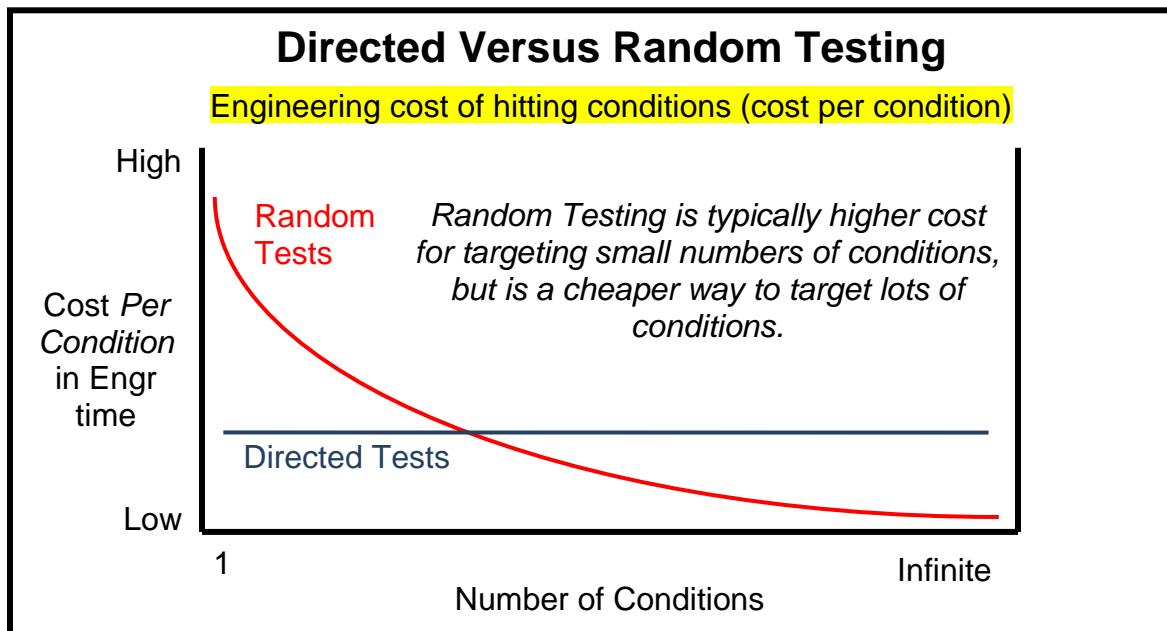


Figure 9

Random tests have a much harder time performing self-checking. Self-checking is a beneficial addition to a test generator. It provides the only checking in environments where there is limited checking (such as Emulation or the Post-Silicon environment), and increases the overall checking in environments such as the Pre-Silicon environment. Random tests have a much harder time performing self-checking than directed tests due to the randomness in which it chooses operations. For example, if a test generator is creating a multi-threaded test that does data sharing among threads, some data values might be indeterminate – and thus cannot be self-checked. In this case, it requires some form of run-time or post-processing checking to determine correctness.

When deciding between a random or directed test for a given task, or when creating a stimulus strategy for a project, consider the following questions along with the observations previously put forth within this section:

- Do I need to hit this condition predictably and/or efficiently; for instance, as part of a turnin-gating regression?
- Is the condition I am trying to hit just a ‘base’ condition, i.e., do my tests need to hit more than this condition?
- Do I need to hit the condition ASAP; for instance, in testing a Post-Si bug fix, or getting basic exercise on a new feature prior to releasing to a larger audience?
- What kind of infrastructure have you inherited? Is it individual directed tests, a directed-test generator, individual random templates, or a monolithic random test generator?
- What are the long term stimulus requirements? Will the stimulus be packaged as part of design soft-IP delivered to another group? Does the stimulus need to live for multiple generations of products and be easily updated for new features?

Taking into account the different properties of directed and random tests, and the specific situation you are working in, these questions and the information in this chapter should help you to make good choices.

Some of the advantages/disadvantages are spelled out in the following table.

Directed Stimulus	Random Stimulus
<u>Advantages:</u> <ul style="list-style-type: none"> - Lower engineering cost for small numbers of tests/conditions. - Debugging is typically straightforward – the intention of the test is ‘known’. - Self-checking of test intent and correctness is relatively easy and straightforward - Less likely to cause random failures in turnin regression gating lists 	<u>Advantages:</u> <ul style="list-style-type: none"> - Lower engineering costs for large numbers of tests/conditions - May hit conditions <i>beyond</i> what Validation can conceive.
<u>Disadvantages:</u> <ul style="list-style-type: none"> - Engineering cost is directly proportional to the condition space. 	<u>Disadvantages:</u> <ul style="list-style-type: none"> - More difficult to debug due to complexity, requiring broader understanding of the DUT. - Greater chance of false failures due to illegal interactions of feature space.

<ul style="list-style-type: none"> - Stimulus is strictly limited to the conditions Validation conceives. If Validation did not come up with the condition, it will not be hit. 	<ul style="list-style-type: none"> - Self-checking may become quite complex. Typically checking is moved outside of the test to the test environment. - The intention of the test is not known, which may add considerable time to debug
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 1

6.2 Randomness Spectrum

A great way to conceptualize a completely random stimulus generator is one that applies unconstrained random data to every input of the system. As compute resources approach infinite, one can theoretically exercise every possible condition with this technique, even in a near-infinite condition space. This is similar to the concept of an infinite number of ‘randomly typing monkeys’ eventually recreating a Shakespeare work.

The downside of a completely random stimulus generator is that we do not have infinite compute capacity, even within Post-Si Validation; thus, we utilize constrained random testing. When attempting to hit a specific condition or set of conditions, a random stimulus generator can be constrained to hit the condition quicker, in terms of compute cycles, than a completely random one. The combined intelligence of the user (knowing what conditions need to be hit) and a smart stimulus generator (knowing how to resolve complicated constraints provided by the user and by the system (for *legality*)) creates a compute-efficient solution to testing a near-infinite condition space.

This, however, comes with the following costs:

- 1) The user has to figure out what conditions to hit (it’s a big space!)
- 2) The user must know how to make the test generator hit them (biasing can be tricky)
- 3) Smarter test generators require more engineering time to build/maintain, and are naturally bigger, buggier, and with more conflicting constraints. This can cause testing holes!

Though these may sound relatively trivial they are not, and they end up resulting in a significant amount of engineer time. The engineering time is still far less than the time it would take to identify and code directed tests for every condition in the system (infinite!), and the upside is that your computing requirement drops from near-infinite to something relatively manageable. In Figure 10 you can see a graphical representation of constrained random testing as the “sweet spot” on the engineering and compute cost curves.

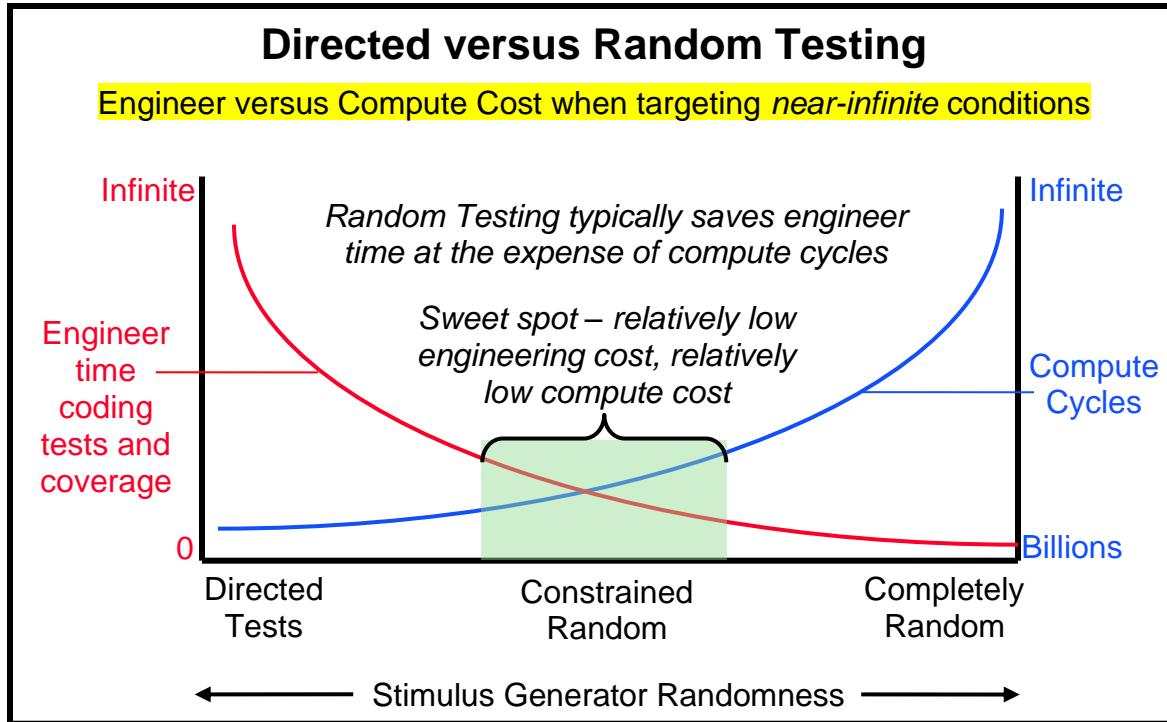


Figure 10

Validation typically uses [Coverage](#) to minimize the engineering cost of constrained random testing. Instead of identifying the entire condition space, it identifies a representative subset of the condition space to use as a proxy for the entire condition space. If the constrained random stimulus generator performs well on this proxy, it is likely to do a good job on the entire space given sufficient compute cycles.

Having coverage monitors running with the random tests allows Validation to minimize the risk caused by #2 or #3 in the list above. If a Validator does not create proper biasing for the stimulus generator, or if the stimulus generator has internal bugs that create testing holes, the coverage feedback alerts Validation to the problem.

Within constrained random testing, Validation has control to shift cost back and forth between engineering and compute cost. To lower compute cost, use engineering time to make smarter, more efficient stimulus generators, smarter biasing, and greater depth of coverage monitoring to make up for the added risks. To lower engineering time, dumb down your stimulus generators and their biasing input, and simplify your coverage space, and make up for it with compute cycles.

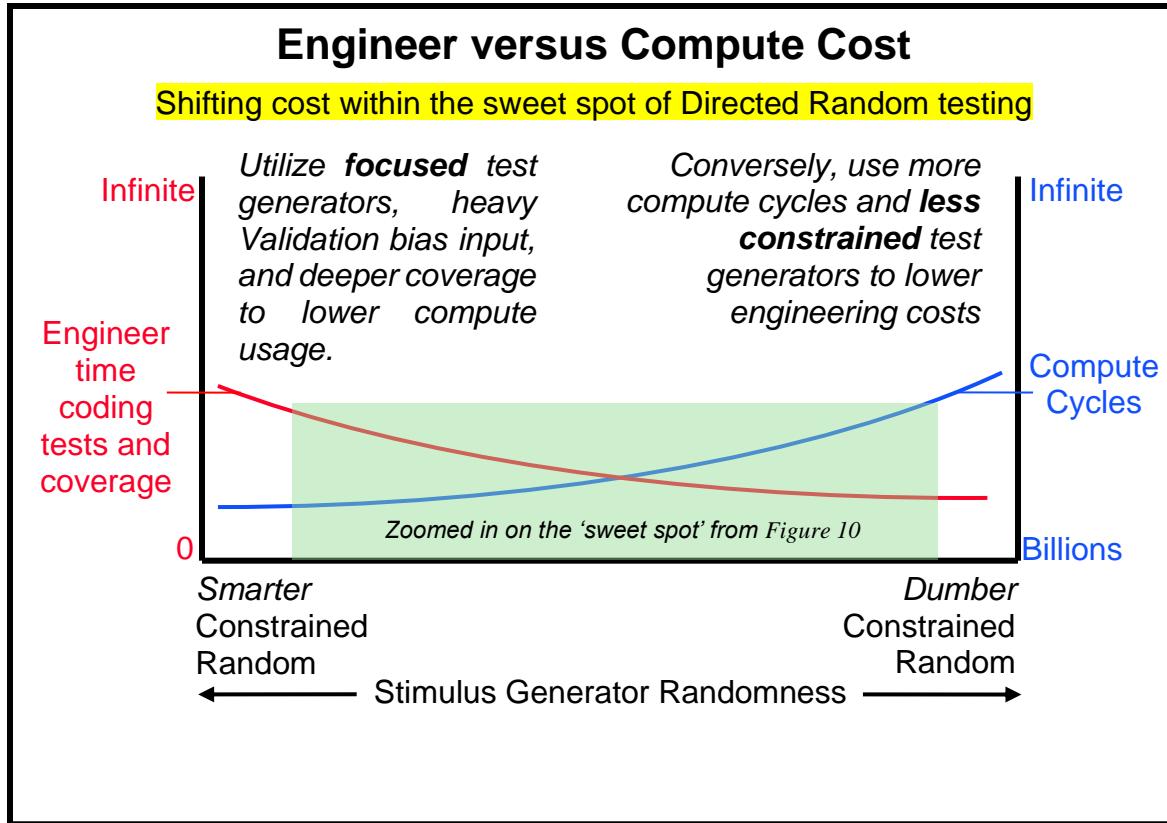


Figure 11

In summary, the tradeoff between constrained random and completely random testing is efficiency versus completeness. In truth, the efficiency gain is enormous versus the relatively small engineer cost associated with constrained random testing. Because of this, Pre-Silicon Validation seldom uses completely random testing; instead we use constrained random testing.

For the remainder of this chapter, the term “random test” implies “constrained random test”.

6.3 Dynamic random vs. randomly created static instances

There are two ways in which a random test can create its randomness, dynamically or statically. A dynamic random test makes random decisions *while the test is executing*, possibly using feedback from the DUT to further increase testing efficiency. A static random test makes its random decisions at build time, effectively creating a directed test to be run on the DUT. This precludes having any capability of using feedback from the DUT in its decision making. Static random tests have the nice side-effect of usually being easier to debug, that is, you do not have to understand what random decisions are being made since you are effectively debugging a directed test running on a DUT. To match this, a dynamic random test can output a “directed-like” log of everything it did during a test run to facilitate debug.

In truth, random tests can fall anywhere between purely dynamic and purely static. A good example of a dynamic-static hybrid test is where the test content is created statically before run

time, yet it starts injector subroutines that make random choices depending on DUT interactions. Other hybrid tests can be found in clusters that include instruction decoders, where static assembly tests are created at build time while the random test environment creates random clears and fetches at runtime.

Which method is better, dynamic or static? In truth, both are fine. It is recommended that at least some small portion of the test or environment is dynamically random, such that it can use feedback from the DUT to make the test that much more efficient or evil.

One further note on this: In the past, (and likely in the future) we have been interested in keeping “mostly static” versions of our random tests around for use in places such as model release regressions. The reason we keep a “mostly static” version is because a regression simply attempts to make sure that those features that worked in the past do not break on future models. In the case of a static random stimulus generator this is quite easy – we simply create a static test instantiation from the generator, and add it to the regression. In the case of dynamic tests, this is much harder. We would attempt to “seed” the random test, as in pass it the same seed as a previously passing version, but due to the way that our team has coded our dynamic stimulus generators we found that they were much less predictable from model to model. Why? Because we coded the test, the CTE, and the DUT-triggered injectors to use the same randomization macros. Thus, changes in the model timing for some small injector might cause major changes in the test itself, such as the test choosing an entirely different mode of operation.

Special writers note: There is really a third style here – indeterminable dynamic random. This type of dynamic test has the test output itself (such as assembly code) use indeterminable environment randomness to alter itself along the way. For instance, in the presence of a hardware random number generator or of a TimeStampCounter, you can imagine the test using these values and making decisions with them such that the test changes depending what it sees. These tests are reproducible simply by forcing the state that the test uses to the desired value. In general, you can think of these as just “dynamic random” for the purpose of all of these discussions. These, in general, should be avoided – they do not add much to the overall stress of the test and they have horrific issues with reproduction.

6.4 One vs. many stimulus generators

One of the toughest tradeoffs a Validator makes is in structuring a test suite. Finding the optimum stimulus ‘coverage’ versus cost is not simple. The following are some points to consider:

- Overlap between multiple stimulus generators has benefit, typically because two different stimulus generators targeting the same thing will target it in different ways, and possibly find different bugs
- At least one stimulus generator needs to cross-product all features to hit the potential array of interaction bugs
- As stimulus generators try to do more of ‘everything’, they typically become more complex and expensive
- With tight budgets, having multiple stimulus generators target the same functionality is not practical

A test suite can be made up of a ‘mega’ stimulus generator that tries to do everything, or possibly many smaller, more focused, stimulus generators. This section discusses why the mega stimulus generator is preferable but not always practical. For the purposes of these sections the term *stimulus generator* can mean either of these two things:

- 1) A monolithic, independent test generator (such as the core tool *Café*)
- 2) A test that randomly picks items out of sequence libraries and puts them together

A ‘mega’ stimulus generator is defined as follows: a stimulus generator that tries to do *everything*, or really close to everything. In the space of a monolithic, independent test generator it would mean that this test generator understands every feature that the DUT contains and attempts to create stimulus to validate those features. In the space of a stimulus generator that uses sequence libraries, a ‘mega’ stimulus generator simply means that it will mix and match any and all sequences together.

With no constraint on effort, Validation would always choose to create mega stimulus generators that target all features. However, Validation is always effort-constrained, and so it makes sense to put serious thought into how to structure the stimulus suite.

The answer is not simple. To begin with, assume the answer for any given DUT should be “create a mega stimulus generator”. The reason for this is that you do not want feature-interaction bugs to fall through the cracks – if stimulus generator 1 stresses features A/B/C and stimulus generator 2 stresses features X/Y/Z; for instance, a bug in the B-X interaction space might not be caught. Moreover, if the coverage space does not enumerate all possible feature interactions, coverage will not reveal the testing hole. Figure 12 illustrates this with a bug that is a cross-product of nearby coverage conditions – which are being hit by two different stimulus generators. Thus, the bug can escape between the two stimulus generators, *though all the nearby coverage conditions are hit*. See [Coverage section: Targeting beyond the enumerated coverage space](#) for more information.

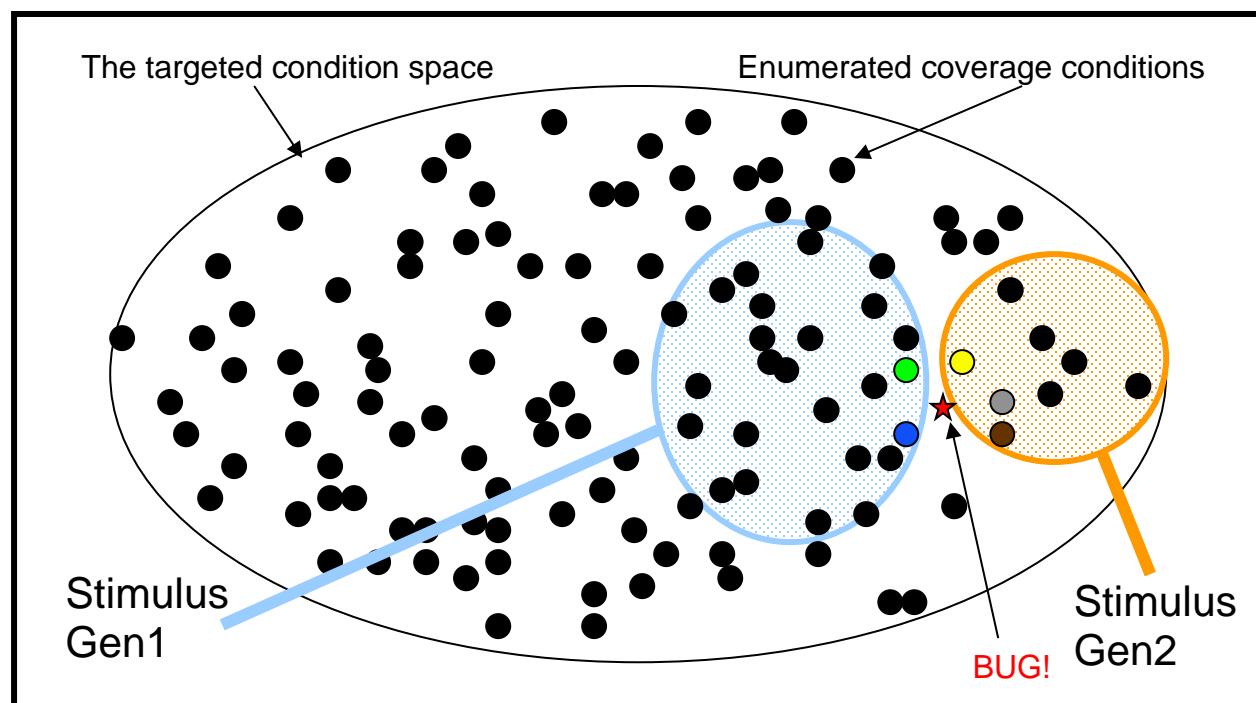


Figure 12

After you start with the assumption of “creating a mega stimulus generator”, then walk through the following caveats and examples to see if your situation calls for more, simpler, stimulus generators.

The first thing to consider is whether all the features within your DUT interact. If they do, or there is heavy interaction between the units within your DUT (typically considered ‘strong coupling’), then it is likely that you need one mega stimulus generator so that the features can be tested exhaustively together. However, if they do not have heavy interaction, then you can break them into smaller groups of coupled features and consider breaking your test generation along those lines.

The next thing to consider is that there are desired properties for your stimulus package beyond exhaustively testing all features together. For example, your overall stimulus package should be:

- easily maintained
- extendible as new features are implemented
- capable of creating focused traffic

(See Section 6.6 for these and other desired properties of your stimulus package). If two feature sets are hard to put together within the same stimulus generator, or if the two feature sets will make the stimulus generator controls more confusing to the user, consider keeping them in separate stimulus generators. A simple example of this is fullchip test generation, where the majority of core internal conditions rarely interact with uncore internal conditions. Most core conditions are covered by architectural RIT-style scenarios, and the user control is at the IA architectural and core micro-architectural level. Conversely, efficient test generation in the uncore cares very little about the instructions being executed, but cares much more about the system configuration, traffic types, and the various DRAM and MMIO address space setups. Satisfying both of these requirements would make for a very complicated stimulus generator, and placing the controls for all of this in the hands of the user would likely create confusion.

Another consideration is the lifetime of the stimulus package. Will this package be utilized and upgraded over the course of many projects? If so, it makes more sense to spend effort creating a mega stimulus generator that can support everything – the long term cost is better than supporting a bunch of smaller stimulus generators, especially when new DUT features are added project after project. A project might consider a tactic of creating a bunch of simple stimulus generators to get through the short term, but begin funding a mega stimulus generator for the long term and to eventually take the place of the smaller stimulus generators.

One addition note on this subject: one would think that moving to a sequence-library-based random stimulus generation would make it much easier to mix-and-match stimulus such that this is all a moot point, i.e., *mega stimulus generators should be easy when using sequences*. In practice, it is still complex, since not all sequences play nicely with others – for instance, power management and Package-S-State (warm reset, etc) sequences often create havoc for other sequences. Smart resource management and system constraint modeling are needed to properly mix-and-match random sequences. This is sometimes beyond the effort that Validation is willing to expend, so decisions are often made to, for example, test warm reset outside of the mega stimulus generator.

6.5 Differences between Pre-Si and Post-Si test generation

The following sections discuss a few areas where there are important differences between Pre-Si and Post-Si environments that have a big effect on test generation.

6.5.1 Failure detection

Pre-Silicon validation, in general, relies on its ability to have full visibility into the internal state of the machine so that it can install low level checkers. Post-Silicon, on the other hand, is reliant on the test itself and the platform it runs on to make an error visible. The vast majority of the time this is done via the test doing “self checking”. Self checking in the world of Post-Silicon often relies on keeping track of all data that has ever been touched so that it can be checked at the end of the test. For instance, doing two different move operations in a row to a register is a bad idea – the first one can never be checked since its data is being overwritten. A good self-checking test will first save the register data away in a reserved spot in memory before overwriting with the next move. The additional code to do this save operation is called *visibility code*. After the test completes, a post-processor can go back and check all of the reserved memory spaces to see if all of the data calculated throughout the test was correct. The problem with visibility code is that it is very invasive within the test. Suddenly, in the middle of two ‘move’ instructions, the test needs to place a store operation!

This fact that the stimulus generator needs to place visibility code inside of the test to enable self-checking causes the tests to be less stressful than they otherwise could be.

6.5.2 Test build time vs. test run time ratio

In both Pre-Silicon and Post-Silicon validation, the total simulation time that it takes to create and run a test is the “cost” of running that test. If it takes one minute to build a test of 300 instructions in Pre-Silicon and two hours to run that test that is a very efficient build versus run time cost ratio. However, tests run very quickly in Post-Silicon. That same 300 instruction test might run in less than a second on a Post-Silicon platform. That would be a very poor build versus run time ratio! In this situation, Post-Silicon validation would be more limited by the number of tests it could build instead of the number of tests it can run. So, test build speed is critical for Post-Silicon validation.

6.6 General desired properties of stimulus

The following sections contain items to consider when creating or choosing stimulus generators and/or test environments.

6.6.1 Reproducibility

In the case of a test failure that requires debug, or in the case of wanting to recreate some condition that has been hit by a random test, it is necessary to have the random test reproduce the same behavior and output given the same input. In the cases where the random test creates, at build time,

a static test, this might not be necessary since the static test might still exist and Validators can use that directly to reproduce failures or conditions. Thus, this becomes especially important for dynamic random tests that do not leave behind static test instantiations.

Reproducibility is also useful to those developing the random stimulus generator. As a piece of software that has bugs, reproducing the same result is important for finding, analyzing, and fixing bugs in the generator itself.

In general, Validation requires the following sorts of things for reproducing failures or conditions:

- Same test generator version, including libraries
- Same user input
- Same random seeds
- Same DUT version
- Same test build tools

6.6.2 DUT-specific configuration

All generators have some configuration-specific knowledge that changes their behaviors: fuse settings, security policies, which IPs exist in the system, and configuration register addresses. For a stimulus generator to be useful for multiple products or generations of a single product it must understand the differences between them. For example, if the cache size doubles between generations one and two of a product, the stimulus generator might want to choose data addresses in a slightly different manner to stress the new boundary conditions. The easiest way to do this is with DUT-specific configuration.

A DUT-specific configuration is a library that includes all information the stimulus generator needs about a DUT. For example, the file should contain all information specific to the architecture or micro-architecture. Then the user simply informs the stimulus generator which configuration to use and the test is tailored to that specific DUT.

The following is a short list of possible configuration information

- IP list
- Fuse settings
- Feature/Mode capabilities
- Legal/Illegal instructions
- Cache and AddressMap breakdowns
- Pipeline depths and buffer sizes

6.6.3 The right level of randomness control

When using a random stimulus generator, a Validator should have the ability to *direct* or *free* most features and sub-features that are exposed to the user. By directing a set of features, the Validator

is attempting to focus in on a narrow range of conditions that target hard-to-hit or rare conditions. By freeing a set of features, the Validator is telling the stimulus generator to allow full random potential, for instance, all legal values. This capability is recommended for all features such that the user can direct a few while freeing the rest. For instance, to create a targeted memory test, a Validator might direct the stimulus generator to only use a small set of addresses and memory types. At the same time, the Validator might not care what instruction causes the memory operation, so he might want the stimulus generator to choose freely from all memory related instructions. Thus, it is an important thing to be able to free any set of inputs while directing another set. It is important that completely freed features should meet a full random potential – if the Validator says that register A can be any value then the stimulus generator had better allow it to be any value and not constrain it to a subset.

There was an interesting “miss” case similar to this in the P4 Post-Silicon history. The case boiled down to the following: The Post-Silicon random stimulus generator Pipe-X created “reasonable” values for architectural register ECX when setting up string operations. After this, it would decide whether or not the string would see an exception before finishing. If so, it would attempt to go back and fully randomize the ECX value, since it no longer required a “reasonable value”. The ECX value should have had the potential to be any value at this point – but Pipe-X failed to randomize it properly, and thus it allowed a Post-Silicon escape: If ECX was within a certain set of values in a string exception case, the CPU would get data corruption.

Good control depth in a stimulus generator is useful to work around blocking failures as well as to work around incomplete (or blocked) features during the FED phase of RTL development. Furthermore, being able to focus on a subset of features can allow you to narrow down your failure during limited visibility debug such as Post-Silicon.

6.6.4 Feature cross-product

It is important for Validators to test interactions between the features they own as well as interactions between their features and many others. Thus, a stimulus generator should not have limitations in its mixing of one feature with another. For instance, if the stimulus generator supports running string instructions, and it supports all paging modes, then it had better support string instructions within all of its paging modes. This capability is hard to measure within a stimulus generator since it is rare that anyone is writing coverage monitors for all cross-products. This does not mean we do not want to see this type of testing happen; we simply don’t have time to add coverage conditions down to that level. Section 6.4 explored this same topic from the point of view of structuring an overall test suite.

Feature cross-products seems easy enough, but when you start mixing together features like “looping” and “strings” and “interesting addresses”, suddenly it becomes a very tough problem. Usually it can be solved; however, it might add undue complexity to your stimulus generator, which requires more engineer time and generally leads to more software bugs within the stimulus generator.

In cases where the test generation intelligence is distributed, such as in sequence libraries, it is paramount that library contributors do their utmost to make their sequences compatible with others’; moreover, the more they can be ‘interwoven’, the better.

One last item to consider: stimulus generators, test environments, and sequence libraries may all do their utmost to make all features compatible; i.e., any two features can be targeted to test together. This, however, is not enough – the user of the stimulus generator must still ask for this to happen, or at least not forbid it from happening. See Section 6.7 for more discussion on this.

6.6.5 Mutually Exclusive features

Some features/sequences do not work together: they are *mutually exclusive*, or *mutex*. For instance, a certain power management flow cannot occur with certain IP configurations or fuse settings – this is a configuration-based mutex. Or, two different sequences which utilize the same resource cannot run in parallel; they need to be serialized – this is a temporal or resource-based mutex.

One temptation when creating stimulus generators is to “not handle” these mutex situations; ie, to simplify the test generator to only handle features/configurations that do not cause mutex issues. In order to strengthen the overall stimulus generator or test environment, it is better to build the knowledge of these constraints into the stimulus generator such that these conditions may be utilized whenever the structural or temporal restrictions allow.

6.6.6 Changing contexts during generation

One downside of monolithic stimulus generators is that the user input is typically in the form of a single bias file for the entire test. The results is that, within the bounds of the user’s biasing and the natural randomness of the stimulus generator, the entire test is focused in one ‘direction’. Moreover, this may also mean that every core running in a system, or every IO port, etc, is doing the same thing. This makes it extremely unlikely to have one core beating repeatedly on one feature while another core is beating repeatedly on something different. Or, that for half of the test you utilize mostly cacheable operations, and then switch to almost completely uncached operations for the second half.

To facilitate these types of situations, stimulus generators should be coded in a way that any agent (such as an IA core, or an IO agent) can be given separate bias controls. Moreover, they should be coded such that any agent can switch from one bias control to another in the middle of the test. This allows for the stimulus generator to take bias controls with very different targets and mix them within the same test, typically creating very interesting results within the DUT. See section 6.7 for expansion of this concept.

6.6.7 Overstressing vs. relaxing

Numerous boundary conditions occur within our CPUs, at points in time where certain portions of the DUT might “relax”. For instance, when a pipeline clears of operations, much of the hardware might power-down its clocks. If you throw so many operations into your test such that this rarely happens, you might never end up hitting conditions similar to “uop appears in pipe just as pipe was powering down and something bad happens”. Allowing the stimulus generator to pause could be as simple as variable amounts of time between generated transactions or it could be as heavy-handed as to completely suspend traffic on one/some/all drivers in the DUT. “Relaxing” is

agnostic of the results of the pause in traffic and the stimulus generator will resume after some period of time.

6.6.8 Quiescing vs. relaxing

Though similar in nature, quiescing is much stronger than merely allowing the stimulus generator to relax or pause. Quiescing can be thought of as “relaxing with a purpose”. The stimulus generator or portions of it will be quiesced until some other condition in the machine is met.

Quiescing has several flavors. They include stopping traffic:

- 1) On all drivers until the machine is empty
- 2) Until a clock LCB powerdown
- 3) Until a PCIe link transitions from L0 to L1
- 4) Until a PCU-initiated flow is triggered by the inactivity

Quiescing will stop traffic to create a desired effect and the stimulus generator can't resume sending traffic until the condition is met. Copious debug messages should be sent to log files so that debuggers can easily identify where hanging tests are stuck if they are waiting for a quiesce condition.

6.6.9 Monotony monotony monotony!

Random tests, simply by their nature, rarely exhibit the behavior where they do the same thing over and over again. In some cases though, this is a very important feature to have. While it might seem foolish to run “add add add add add add add add...” (This is the famous Blair Milburn test) it might actually cause some very interesting conditions when the machine completely fills with them. Furthermore, this sort of thing is very useful when you have multiple agents that access the same bus or fabric. You can have one agent simply be a monotonous agent that simply sends operations forever, trying to break the other agents by strangling their access to the fabric. This is called a “harassing agent”, and is very important in the world of CPU and SOCs that we live in.

When building sequences, the middle road between full randomness and monotony is to code a given sequence such that it allows the possibility of generating either extreme. In uncore testing, a sequence that only generates coherent opcodes will send a variable number of fully random combinations of any coherent opcode that can be sent across IDI from the core emulator to the uncore. The sequence should be coded so that some percentage of the time it will choose only one of those opcodes and send it repeatedly. Repeated calls of the sequence will vary the output between fully random and the monotony without any additional input.

6.6.10 Ease of use

If you build a decent stimulus generator there is a likelihood that other people will want to use it or you will have to proliferate it to a subsequent project. Moreover, you might actually want to try and get others to use it because you feel you have the best stimulus generator in town. The first impression of a stimulus generator often comes from how easy it is to use and whether or not the user interface makes sense.

Any TE changes that may break the generator should be blocked at turnin thus preventing any cratering of the random stimulus generator upon a new model release. The penalty for adding seeded random tests to a turnin regression is that the behavior of the seeded random tests will change as the TE changes. The pass rate of random unseeded runs of the stimulus generator needs to be maintained >99% for this to not cause instability in the regressions themselves.

6.6.11 Debug-ability

When running a stimulus generator, it is important that Validators receive output from the stimulus generator to help the debug process of DUT failures, missed conditions, and internal stimulus generator bugs. The following are debug aid examples from various stimulus generators.

- A report file output of the exact operations of the test and what they expect to do
- A log file of all random choices made throughout the stimulus generator's production of a test instance. The user can give categories and weightings for what types of features for which to collect information.
- An output of the final numbers and percentages of choices made, to be compared against the original user input biasing.

6.6.12 Extendibility for new features

There will always be new additions to our architecture and micro-architecture. Keep this in mind as you are coding your stimulus generator such that it is not impossible to add new capabilities in the future. There is always another new feature around the corner. You cannot be expected to know the next new feature beforehand, but you can certainly keep your code modular and understandable such that you can update it when the time comes. Fragile code is not something to have in a stimulus generator!

Having said this, it is important to note that the IA32 architecture does not lend itself to modularity. While there are many ways to group instructions or features, it seems that there are so many quirks and caveats within the architecture that assembly stimulus generators rarely can take advantage of much modularity. It is still a good idea to try though!

The same is not true for cluster testing that may not rely on IA32 instructions. New test functionality may be added in cluster sequence libraries that specifically addresses new functionality added to the RTL. This new test traffic may then get randomly chosen from the sequence library and run with the legacy sequences.

6.6.13 Efficient setup

It is incredibly important that stimulus generators, especially in the Pre-Silicon world, make efficient use of run-time cycles. Portions of the test instance that are merely setup and not intent-based should be minimized as much as possible. Just saving 10,000 cycles per test by getting around a few MSR writes is an awful lot when you realize that you might run the stimulus generator a few million times.

As always, there are caveats. If your test does its setup code randomly, then it might be useful to have this inefficiency since it is important to get to the same destination state through many paths. One of the paths might find a bug!

6.6.14 Ability to work in multiple environments

In the Pre-Silicon validation world, it is very useful for a fullchip stimulus generator to be able to produce silicon friendly tests when needed (or all the time). This is useful for High Volume Manufacturing (HVM) fault grading content, since your tests might hit interesting microarchitectural conditions that they would like to see hit in the Post-Silicon silicon test world to find silicon faults. Furthermore, during initial Post-Silicon system bring-up, it might be useful as an information tool to be able to create short tests, find out what they do Pre-Silicon on a simulator, and then run them in the lab to attempt to diagnose a system failure.

It is also useful for stimulus generators to run in the Pre-Silicon world. This might help Pre-Silicon validation to find extra bugs before tapeout; more importantly, it will find the major bugs that would impair the Post-Silicon stimulus generator from quickly peeling the bug onion on silicon.

In general, this applies to fullchip stimulus generators only, although some cluster stimulus generators make useful Post-Silicon tests via hookup to DAT facilities.

6.7 Effective use of a random stimulus generator

This section will introduce you to some of the best known methods for effectively using random stimulus generators or random test environments.

6.7.1 Never turn off the background noise

It is natural to have a particular target in mind when coding a basic test or when biasing a random stimulus generator. This target may come from many sources: a testplan entry that you wish to complete, a coverage hole you are trying to fill, or an area where escapes have occurred and you want to do more extensive testing. Armed with your stimulus generator and microarchitecture knowledge, you will know what types of features you will need to control to hit this target. So far, so good! However, the natural tendency is for users to ‘turn off’ everything that they do not think is particularly useful to focusing in on the target area. *Why bother turning on Floating Point instructions if you are targeting Integer instruction control logic?* This natural tendency comes from an instinct to be efficient: turning on unnecessary features may slow down or actually work against the stimulus generator’s ability to hit your target.

In practice, the opposite is true. Many projects’ worth of data at multiple DUT levels (cluster/fullchip) has shown us that the best controls for hitting coverage and bugs are those that always leave everything on *to a small degree*. In these cases the user controls the features to target the given area, and leaves everything else to either a default value or some low noise value. The noise value might be extremely small, but they should be on enough to cause interactions to

happen. These interactions, while rare, are enough to put the DUT into a different operating space and hit conditions that otherwise would not have been hit.

It is useful to note here that this is made extremely efficient for the user if the default bias for stimulus generators has everything on to some small degree. This way, the user can concentrate on the knobs specific to the features being targeted, and can leave everything else to the default value.

This concept is similar for test environments that are based on sequences. Users may target a test by utilizing a small group of sequences specific to the targeted feature, but they should always leave a small random possibility that other sequences will be called as well.

6.7.2 Do not turn everything on full blast

After reading the previous section, the natural question is whether all features should be turned on above the noise level, and in fact simply have a main bias file which turns all features on ‘full blast’ to try to hits bugs and coverage as fast as possible. This also was attempted on previous projects, and it also had poor results. When a stimulus generator is told to do ‘everything’ at once, it typically does nothing well. The DUT will be pulled in every direction at once, and will rarely have a chance to setup certain conditions. Sections 6.6.7, 6.6.8, and 6.6.9 discuss the importance of monotony and on quiescing the DUT. This sort of thing cannot happen with the stimulus generator trying to do everything at once. Have ten operations of the same type in a row is very unlikely when the stimulus generator is being asked to choose from many types that all have high bias values.

This issue does not exist to the same degree with sequence library-based stimulus generators, but it does exist. Certain conditions within the DUT require the same sequence to run many times to properly setup. With all sequences weighted at a high value, this may not occur. The fix for this is to create a new sequence that does all of the work to get the DUT to that state; but this would require the users to know every required sequence!

6.7.3 Maximize the use of the collective knowledge of your peers

In any given DUT, from the smallest cluster test environment to a complete system, there are almost always multiple Validators working and creating test content, often in the form of bias controls or sequences. This collateral is the embodiment of the Validator’s *intent* as well as their knowledge of how to fulfill that intent. In some cases, it takes Validators many iterations to figure out how to bias a stimulus generator or write a sequence to properly target a condition. Once this effort is successful, no other users should have to go through the same effort to hit the same conditions. Instead, the bias file or sequence should be added to a standard library.

For bias files, the reason to have a standard library is to allow Validation teams to use them as ‘building blocks’. These building blocks have Validation intent and knowledge. A library of these may contain many Validators’ intent and knowledge. If the stimulus generator can utilize many bias files at once (such as on different threads, different IO devices, or simply changing bias files on-the-fly), it can cause excellent interactions between different features – with little extra learning outside of their area by the individual Validators.

The same is true for sequences: if all Validators place their sequences into a standard library, it is available for all to use. It also places extra responsibility on the sequence writers: they must try to make their sequences as robust as possible by adding as few artificial constraints as possible. This allows the sequence to work well with other sequences and to not inadvertently turn other sequences off or limit them. It then becomes important that all users have all sequences within the library enabled with a small noise weight.

These practices attempt to maximize the effectiveness of constrained random testing by maximizing the use of the collective knowledge and intent of the Validation team.

7 Summary

Stimulus is the pillar of Validation that most directly affects overall validation quality. Tests can be directed or random, but typically become mostly random for the vast majority of functional validation work. Random tests allow Validators to test outside of their knowledge of the DUT. There are numerous desirable properties in a random stimulus generator, such as debugability, controllability, reproducibility, and extensibility.

There are many portions of the overall testing effort that do not have simple answers, such as whether to create one “master” stimulus generator or many smaller generators. These issues need to be sorted out on a case by case basis.

8 Future Work

The following sections have been identified as good candidates for future additions into this chapter. Many of them are specific to actual stimulus generator implementation.

8.1.1 Distribution of test between the test environment and the Test

What portion of the entire test infrastructure belongs in the CTE itself and what is best left in individual tests and stimulus generator.

8.1.2 Types of randomness

Perhaps a talk about distribution, for example uniform, Gaussian, and left-heavy

8.1.3 Biasing and User Input

There are many interesting ways to create and use bias files. For instance, the ability to use multiple bias files in the creation of one test instance. BKMs exist for use input, such as allowing for command-line overrides to internal knobs. “Recipes”, that are user-written code chunks or macro calls, are also useful forms of stimulus generator input.

An effective random stimulus generator provides its user one or more interface(s) through which biasing can be applied. Two commonly implemented interfaces are through command-line arguments and bias files. There are pros and cons to each, with many stimulus generators implementing more than one interface. In deciding which interface a stimulus generator implements, there are a few key attributes to consider. These attributes are intended to minimize “reinventing the wheel” and facilitate frictionless proliferation of established goodness, with respect to targeting specific scenarios. The most important characteristics to consider is Modularity.

A biasing interface should provide the user the ability to create test scenarios from one or smaller biasing components. One such scenario, in CPU cluster, is to have one core running snoop biasing while another running power management biasing. The ability to bias the subcomponents of a greater subsystem in isolation, using independent biasing inputs (i.e. individual bias files), will minimize the need for a Validator to be the microarchitecture/stimulus generator expert (Disclaimer: you still need to know a thing or two) to adequately setup complex interactions.

Another useful interface for random stimulus generators to have is the ability to execute user-written code chunks. This provides infinite potential to extend the coverage of a stimulus generator, well beyond its original design.

When combined, these two interfaces significantly improves the usability of a random stimulus generator.

8.1.4 Stimulus generator Outputs

There are a few options for stimulus generator outputs. For instance, in CTEs, different SMT threads can have their test code interwoven or in separate files.

8.1.5 Genetic feedback

What are the pros and cons of genetic feedback, the current state of the art, and its ramifications on the next stimulus generator you write?

8.1.6 Input coverage

It might be possible to get coverage information on your stimulus generator without having model-related coverage using input coverage.

Due to the relative complexity of a typical DUT and the various stimulus generators designed to validate them, there are inherent interdependencies between the hundreds (or even thousands) of knobs used to target validation test cases. Since every user of a stimulus generator cannot be an expert, one might not be aware of or may easily overlook any one of the many dependencies built into a complex stimulus generator. As such, the ability to dump coverage information of what instructions, features, or sequences are contained in a static generated test code, provides instant feedback to the user. This significantly reduces the feedback loop generally required to create bias inputs needed to target more complex test cases.

9 References

The IAG Coverage Validation Working Group tutorial: Test Generation. Michael Bair, 2004

The Art of Pre-Si Val: Chapter 7

Checking (In Prog)

By: Stacey Ross and [Neriya Bar-levav](#)

1 Abstract

This chapter is WIP.

This chapter attempts a thorough dissection of functional checking. It will define the vocabulary of checkers so that future discussion builds on a rational framework. Although Validators spend significant time working directly or indirectly with checkers, we frequently fail to analyze what goes into, and what comes out of, checkers. This chapter will attempt to frame that discussion as well. Finally, there will be some discussion on planning checkers and some real-world examples, including discussion of tradeoffs.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	6/1/2011	Completed First Draft	Stacey Ross	Michael Bair / CCDO Art of uAV WG
1.1	6/6/2011	Revision following feedback	Stacey Ross	Michael Bair
1.2	5/16/2012	Revision to be less uAV-centric	Stacey Ross	Michael Bair
1.3	3/1/2016	Updated with latest methodologies	Stacey Ross	Michael Bair
1.4	12/20/2019	Addition of post-processing; cleanup	Michael Bair	Michael Bair
2.0	2/15/2023	Complete Rewrite	Neriya Bar-levav	Michael Bair

3 Contents

1 Abstract.....	167
2 Chapter Revision History	167
3 Contents.....	168
4 Purpose.....	171
4.1 Why do we need this chapter?.....	171
4.2 What does this chapter cover?	171
4.3 What does this chapter not cover?	171
5 Background Concepts	171
5.1 TE	171
5.2 DUT	171
5.3 Checking rule.....	172
5.4 Bug	172
5.5 Failure.....	172
5.6 False Failure	172
5.7 False Pass	172
5.8 Failure effect.....	173
5.9 Escape.....	173
5.10 Reference Model	173
5.11 Validation Collateral (VC)	173
5.12 Monitors and Trackers	173
5.13 Inline versus offline checking	173
6 Checking	174
6.1 Basics: The Act of Checking and the Components of a Checker.....	174
6.1.1 Collecting DUT Inputs and Responses	174
6.1.2 Detecting Checking Events.....	175
6.1.3 Generating the Expected Response from Checking Rules	175
6.1.4 Comparing the Expected Response with the Actual Response	175
6.1.5 Reporting an Error	175
6.2 Serial Vs. Query-Based Components and Checkers	175
6.2.1 Simplified concepts of Serial and Query-Based Checking	176
6.2.2 Using an example to explain Serial and Query-Based Checking.....	176

6.2.3	Serial Checking Pros, Cons, and Concepts	178
6.2.4	Query-Based Checking Pros, Cons, and Concepts	178
6.2.5	Mixing serial and Query based checking.	179
6.3	Checking events	179
6.3.1	Timeout/latency checks	180
6.3.2	Periodic checks.....	180
6.3.3	Hang checks	181
6.3.4	End-of-Test checks.....	181
6.4	Checking rules	182
6.4.1	Source of Checking rules.....	182
6.4.2	Timing rules	184
6.4.3	Ordering rules.....	184
6.4.4	Value rules.....	184
6.4.5	Exact Rules Vs. Statistical Rules	185
6.4.6	Conditional Rules.....	185
6.4.7	Implicit Rules: Deriving Negative (Illegal) Cases from Positive Specs ...	185
6.4.8	Rule Gray Zones: Undefined Behaviors (AKA Fuzziness)	186
6.4.9	Separating and layering	188
6.5	DUT response.....	188
6.5.1	Interfaces	189
6.5.2	Selecting internal or External DUT response	189
6.5.3	Performance consideration when selecting how to collect DUT responses.	189
6.6	Generating the legal expected response	190
6.6.1	Which component should generate the expected response?	190
6.6.2	Generating the expected DUT response.....	191
6.7	Transforming Data for Checker Consumption.....	193
6.8	Checker requirements	194
6.8.1	Useful error messages.....	195
6.9	Checker Taxonomy: Categorization and Classification.....	196
6.9.1	Correctness Checkers	198
6.9.2	Completeness Checkers.....	199
6.9.3	Performance Checkers	200

6.9.4	Security Checkers.....	200
6.10	Testing your Checker.....	200
6.10.1	Unit Tests.....	200
6.10.2	Checker Code Coverage or Checker Coverage	201
6.11	Understanding Checking Escapes.....	201
6.12	Using a Checker as a Debug Aid.....	202
6.13	Trust	202
7	Summary.....	203
8	Future Work.....	203
9	References.....	203

4 Purpose

4.1 Why do we need this chapter?

Checking is one of the three pillars of Validation (see [Introduction to Pre-Silicon Validation section: The three pillars of Validation](#)). Successful Validation requires quality checking, and the efficiency of Validation depends on informed early ROI choices on checking, as well as intelligent implementation. Choices made during checker architecture and development will influence the quality of the final product, the timeframe in which bugs are found, the ease in debugging, and the effort that is required in making architectural changes.

4.2 What does this chapter cover?

This chapter is limited to discussing logical *correctness* checkers, that is, those used to detect logic bugs within Pre-Silicon logic simulation or hardware-accelerated/emulation models. This chapter describes different types of these checkers and illustrates their differences with respect to a few important metrics, such as the efficiency of finding bugs, completeness of the checker, and development effort.

4.3 What does this chapter not cover?

This chapter does not cover checking of non-logic (such as analog or timing). Also omitted: coverage checking, such as frequency coverage alarms. This chapter is also not covering other ways of finding DUT bugs, such as: code review, static analyzers (linters) output and wave from reviews or Formal proves.

5 Background Concepts

5.1 TE

Test Environment. Testbench is a generally accepted synonym, although some methodologies draw a distinction between the two. The distinction usually places the testbench as the “top level” container for the test environment. The Test Environment may include the design and run alongside it, but may also be written to run standalone with other Testbenches (so they can test each other and test protocols). See [Test Environments](#).

5.2 DUT

Device Under Test. The part of the validation system that eventually becomes a product, potentially in addition to a small amount of non-synthesizable support RTL to support connecting to test environments. Typically speaking, this is the functionality described by the RTL model. One could envision systems where a testbench is wrapped around a piece of TE collateral; in that case, the DUT is the TE.

5.3 Checking rule

A checking rule is a rule that the checking operation check. The rule may define what is the legal set of DUT responses, or what, when and how should the DUT react to inputs. notice that while the later require to monitor the DUT inputs, so the output can be decoded as following or not following the rule, the former require not input monitoring. Since the later is more common, and more complex, and for sake of simplicity, this article will only discuss the later, i.e. rules that correlate the DUT response to inputs.

5.4 Bug

A bug is a scenario in which the DUT response is not following a well-defined rule. the checker target is to identify a divergence between the expected behavior (as defined by the rule) and the actual DUT response. (See [The Life of a Bug](#) for more information.) This definition merely defines the presence of the bug, not its location. The bug could reside in the DUT or in the assumption of the DUT or even in the rule itself, as defined in the Architectural Specifications.

5.5 Failure

Any scenario in which the Verification collaterals indicate unexpected behavior. the failure may be in a form of an Error message, or in the form of unhandled SW error. The failure may indicate a DUT or Spec bug. Or a bug in the TE. Failures can also indicate environmental problems, where the simulation or other collateral has a programmatic issue not related to functionality (e.g. full disks, license issues, etc).

5.6 False Failure

A “false failure” is a failure whose root cause is not a bug in the DUT or Spec. In general, false failures incur unnecessary cost to the validation effort, while “true” failures reveal bugs that would have incurred costs to the project if not found. Therefore, it is to the benefit of the validation cost to reduce the rate of false failures.

5.7 False Pass

Conversely, a “false pass” is a test run that is successfully exercising a bug but is not causing the test run to fail. These are very dangerous because they bestow false confidence to the Design and Validation team.

5.8 Failure effect

Failure reported by the checker can have several effects on the rest of the simulation or analysis. The impact is usually defined by the simulation configuration and can be ranged from stopping any farther simulation or analysis or complete ignoring the failure.

5.9 Escape

An escape is a bug not caught at the lowest simulation hierarchy, where it could be observed. In a general sense, it is better to identify bugs at the lowest possible DUT hierarchy, e.g.: in an IP-level model instead of a full-chip model. Exceptions to this generality exist based on cost, as later discussed. Escape may also refer to capturing bugs later than expected, e.g., finding the bug much later than the time the bug was introduced in the DUT, See also [Introduction to Pre-Silicon Validation section: Closed Loop Feedback](#) and [Post-Silicon Escapes](#) for more background on escapes.

5.10 Reference Model

Sometime defined as **Abstract Functional Model** or **Golden model**.

These three terms mean roughly the same thing: a system NOT contained in the DUT whose intent is to simulate the expected DUT response according to the DUT specification. in some cases, there is no discrete reference model, but the test or test generation engine generate both the DUT response and the stimulus.

5.11 Validation Collateral (VC)

Validation collateral is a general term encompassing all things created primarily for use by Validation. Frequently, VCs include passive and active elements; that is, monitoring and stimulus elements. This term frequently refers to functional checking delivered by a specialized third-party team.

5.12 Monitors and Trackers

A Monitor is a piece of validation collateral designed to collect the DUT low-level signal or event activity and packetize it. A tracker is a piece of validation collateral that adds virtual information to collected packets or packs several packets into a larger general packet (e.g., connecting responses to requests). See also [Test Environments section: Monitors](#) and [Test Environments section: Trackers](#)

5.13 Inline versus offline checking

Inline checking is checking that is run concurrently with the execution of a test, typically within a simulator or emulator. In most cases, inline checkers are compiled as part of the DUT, though that is not a requirement.

Offline checking is checking that happens after test execution completes. This is also known as *post-processing checking*. The information consumed by offline checkers can be in the form of test log files, FSDBs, and TLM databases.

The concept of inline versus offline (post-processing) is applicable to all *passive* validation components: checkers, coverage monitors, trackers, and debug aids.

6 Checking

6.1 Basics: The Act of Checking and the Components of a Checker

Checking is the act of finding bugs by comparing the actual DUT response to the legal and/or expected DUT response as defined by the Arch or uArch rules. If the rule is met – the checking is successful, and if the rule is not met – the checking will produce an error. Errors include some meaningful message about the rule that was broken and the actual and expected values that were found. Typically, errors will cause the testrun to be considered a *failure*.

Creating failures is a critical part of validation. As mentioned in [Introduction to Pre-Silicon Validation section: Bucket Hysteresis](#), checking is one example of addition stress that Validation can place on a DUT to expose more bugs.

Checking is implemented within a piece of validation collateral (VC) software called a **checker**. The checker will utilize information collected from the DUT along with specification rules to determine if the DUT is doing something wrong. When it decides that the DUT has done something wrong, it will flag an error.

The following section will briefly outline the basics of checker components, and the subsequent section will dive into far greater detail.

6.1.1 Collecting DUT Inputs and Responses

Checkers gather information from the DUT to perform checking. This can include any combination of DUT inputs, DUT internal state, and DUT outputs. DUT internal state and outputs are collectively known as **DUT responses**, since the updating of DUT state and the setting of DUT output are the *responses to the DUT inputs*.

DUT inputs and responses can come in many forms, ranging from basic signals to more complex compositions such as transactions, sequences of transactions, entire memory images, or even complete architectural and/or micro architectural state. DUT inputs and responses can be observed directly from the DUT, or refined and packetized by a set of collectors and monitors to a higher level of abstraction.

Collecting DUT inputs and responses for the purposes of checking sounds straightforward but is actually quite complex. A later section called **add link here** goes over critical concepts regarding the collection and refinement of DUT responses.

6.1.2 Detecting Checking Events

Checkers use DUT inputs and responses to detect **checking events**, the points at which the checker must decide if there is a failure. A typical checking event is a DUT output, such as when a read operation sent into a DUT receives its data response back from the DUT. At such a point, the checker would check the data returned from the DUT. Checking events can be based on sequences of DUT responses, DUT inputs, periodic events, or even the lack of DUT outputs (for instance, in the case of deadlock/livelock/timeout situations).

6.1.3 Generating the Expected Response from Checking Rules

Checkers utilize **checking rules** to define the expected response during checking events. Rules typically come from specs. Most rules define the DUT response as a transform function that takes in to account the DUT inputs, prior state, and time. This means that to produce the legal expected response of the DUT, one must collect all the function parameters, then take them thru the “transform function”.

For checks of simpler behavior, such as a basic request-acknowledge (req-ack) protocol, the transform function is often encoded directly within the checker (*after Ack assertion, the Ack must be seen within 10 cycles*). In more complicated situations, this is usually implemented in a dedicated VC known as a **reference model** (put a link here if there is a distinctive place that we define reference models in this doc).

6.1.4 Comparing the Expected Response with the Actual Response

Checkers have a component that compares the expected response calculated by the checking rules with the actual response as seen in the DUT state and outputs. The comparison might be simple (*expected == actual*) or something far more complicated that requires special cases. This component must decide when a mismatch has occurred and report an error.

6.1.5 Reporting an Error

Checkers have a component that handles the reporting of errors. This ranges from very simple (print a phrase that includes a keyword such as ‘error’) to more complicated (collect more data, analyze it, and dump this to output files such that a human or machine can consume it and root-cause the error quickly). The reporting logic may also decide if further checking, or even continuation of the testrun, is appropriate.

6.2 Serial Vs. Query-Based Components and Checkers

Checker components come in two forms: *serial components* or *query-based components*. Serial components execute their function in a time-ordered, one-at-a-time manner. Query-based

components execution their function in an un-ordered fashion which could potentially be parallelized.

For components that are detecting checking events, a serial version of the component would likely monitor a serialized stream of input and detect each time a checking event occurs. A query-based version of the same component would likely query a database for all events that match the requirements of being a checking event.

For components that are generating expected responses, a serial version of the component would monitor the serialized stream of input, updating its state as every input operation occurs, and prepare its expected outcomes for all times along the way. A query-based version of the same component would generate an expected response by doing time-based queries of its input traces.

Typically, all components in a checking system act as either a serial component or a query-based component, but this is not a requirement.

In a checking system that utilizes all serial components, the concept of ‘time’ is less important for some components because the time will always be the *current time* (or ‘now’). In a system with any query-based components all components are required to track time.

6.2.1 Simplified concepts of Serial and Query-Based Checking

These over-simplified examples of how serial and query-based checking work are here to give readers the most obvious definition of these checkers before diving into the subtleties.

6.2.1.1 Serial Checking

After initializing itself, a typical serial checker runs the following algorithm:

- 1) Wait for a new operation to arrive from the monitored ports
- 2) Check the new operation versus the current state tracked in the checker
 - a. Flag any errors found!
- 3) Update the current state with the new operation
- 4) Go back to step 1

6.2.1.2 Query-Based Checking

Query-based checkers typically do the following:

- 1) Query the database for all checking events.
- 2) For each checking event:
 - a. Query the database to create the expected response for the specific checking event
 - b. Flag any error found!

6.2.2 Using an example to explain Serial and Query-Based Checking

The concept of serial versus query-based is best explained with an example!

For this example, we'll use a data storage system that stores and retrieves data within a memory structure based on an address (like memory, disk, caches, TLBs, control-registers, etc). The checker is a simple read-data checker that checks that every read operation retrieves the expected data from the storage system.

This checker will use the following component definitions:

- *Checking events* will occur on every read operation.
- *Expected responses* for every read operation will be based on prior write operations to the same address as the read operation.

The next section explains how to make the *components* serial or query-based, and the subsequent section explains how to make the entire checker serial or query-based.

6.2.2.1 Example: Serial vs. Query-Based; per Component

The next four sections explain what these two components would look like using the two forms.

Checking event as a serial component:

The component would iterate through (in time order) all collected inputs and responses from the DUT. For all ‘read operations’, it would capture the address of the operation. When the data response occurs, it would then trigger a check on that address/data combination at that time.

Checking event as a query-based component:

The component would query the database for all read operations. For each read operation, it would trigger a check on the read address/data combination given the time of the operation.

Expected response generation as a serial component:

The component would create simple ‘memory model’ which tracks addresses with data and when they were written. The component would iterate through (in time order) all collected inputs to the DUT. For all ‘write operations’, it would update its memory model for that address with the write data and the time of the write.

Expected response generation as a query-based component:

This component will be triggered by a checking event. If the checking event saw a load to address X at time Y, this component would then query that database for the last write event to address X prior to time Y.

6.2.2.2 Example: Serial vs. Query-Based; Per Checker

As mentioned previously, most checking systems employ components that are either all serial or all query-based. If you take that into account, the example becomes simpler to understand:

Checking event and expected response as serial components:

The components monitor incoming data from the DUT. As write operations happen, the expected response component updates its memory model for that address to be the newly written data. As read operations occur, it triggers a checking event and the expected response is the *current* data

within the memory model for that address. The concept of *time* isn't necessary for the checks since the check is happening at the current time. Most inline checkers have no concept of time; time is simply recorded when the error message is sent to the system.

Most SVTB/UVM checkers are serial checkers. If you study these checkers you will find that they rarely track time, except maybe for debug aids. Time is always the current simulation time.

Checking event and expected response as query-based components:

The checking event component queries the database for all read operations in the system. Then, for each read, it the expected response component queries the database for the last write prior to the read's time to the same address of the read. Query-based checking systems pay much more attention to the concept of *time* since there is no 'current time'.

6.2.3 Serial Checking Pros, Cons, and Concepts

Serial components can run both inline with a simulation or as offline as post-processing.

Serial components naturally excel in execution that requires a high-degree of tracking the time ordering of events. For example, the more your component has to imitate DUT behavior to overcome gray zones (see 6.4.8), the more it becomes useful to use serial components since DUT imitation typically requires significant monitoring and reacting to the timing of DUT events.

[SHOULD WE INCLUDE AN EXAMPLE?]

Serial Components tend to track more data. Since serial components tend to operate in a mode of "get one new input. Check that input versus state. Update state" the state can become very large.

6.2.4 Query-Based Checking Pros, Cons, and Concepts

Query-based components require all inputs (DUT inputs, responses, checking events) to be collected prior to the start of checking; therefore, they can only run offline as post-processors.

Query-based checking systems pay much more attention to the concept of *time* since there is no 'current time'.

Query-based components sometimes have more debug-information at hand, such as the txns that created the expected response.

Query-based components, if coded in a way that saves little state and requires a ton of queries, can quickly become performance problems if they have a logN problem (ie, do not scale linearly with input size)

Query-based components can be easily parallelized.

Query-based checking can have an advantage over serial checking in places where gray-zones exist; having the big picture view of all time can help the checker determine which direction the DUT took and can thus tighten checking. (see [Example for checking rule with grey zone with Query based checker](#))

When the spec defines the behavior of a system and not the behavior of a single input, i.e., when the DUT expected response is defined by the response from many inputs and outputs. One such

example is system performance. Checking system performance by serially checking the performance of each transaction can be very hard, but collecting all the system outputs, and comparing them to the desired performance can be very simple.

6.2.4.1 Example for checking rule with grey zone with Query based checker.

Consider the above memory system, with a small twist.

There can be many pending read and write requests, read request may complete in different order, with the following rule: when there are several reads and writes pending to the same address. Read operation can return with old data (i.e. before the writes that are currently executing) or new data (i.e. the data updated by an on-the-fly write) but must be coherent, i.e. if the 1st read reaching the DUT return with new data, all read operation starting after it, must return the new data.

So if we have this order of access (all to the same address, all starting before the write completed) read_1; write; read_2;read_3. And then complete in the following order: read_2_cmp, read_1_cmp,read_3_cmp,write_cmp

A serial checker, using the read completion as a checking event, will have to implement complicate logic to define if read_2_cmp, returning with old data, is legal or not. It will have to save it and decide only after read_1_cmp arrive.

Query based checker can collect all the access for the same address and easily solve the grey zone problem.

6.2.5 Mixing serial and Query based checking.

A checker architecture that mixes the concepts of both serial and query-based checking can be used to gain the advantages of both architectures. Such a hybrid architecture would collect all of the DUT inputs and responses ahead of time, similar to query-based checking, but then run its checks in a serial fashion by performing the checks on the operations in the exact ordering that they occurred. This brings the advantage of the exact ordering, which would allow the checker to mimic micro-architectural behavior and thus overcome certain gray-zone problems. At the same time the checker can utilize query-based concepts to look over the entire trace to overcome *other* gray-zone problems. In the example above, the serial based checker can collect all the read completion, in order, and then query them after all the pending writes completed.

6.3 Checking events

In section 6.1.2 we introduced the concept of *checking events*, the points at which the checker must run a check and decide if there is a failure. That section said: *Checking events can be based on sequences of DUT responses, DUT inputs, periodic events, or even the lack of DUT outputs (for instance, in the case of deadlock/livelock/timeout situations)*. You may use a different checking event depending on what style (serial versus query-based) of checker you write. For instance, if your serial read data checker uses “read data return” as its checking event, it might miss the case where a read request starts but read data never returns. Further checking would need to be added such as a check that says that all read requests will see data returns (typical protocol compliance).

A query-based check might tackle this in a different way, using the read request as the checking event. This example isn't to advocate for a given style, but more to advocate for the engineer to consider the boundary cases.

While most checking events are linked to DUT inputs or outputs, there are these special types as well:

- Timeout checks
- Periodic checks
- Hang checks
- General end-of-test checks

6.3.1 Timeout/latency checks

Many transactions within our DUTs come in the form of “A triggers B”. One example is a request getting an acknowledge. Another is a read request receiving a data return. The standard check in most of these cases is that “If A happens, make sure B eventually happens”, along with further checking of the B transaction (for instance, is the read data return correct). One additional check to consider is *does B happen soon enough after A?* This is a timeout or latency check. The purpose is to make sure that events happen within a timed window.

This type of check is **required** in places where there is a clear timing spec, such as “*when A occurs, B must happen with X cycles*”. This type of check is **important** in places where there is not a specific timing spec but latency is still important, such as the length of time it takes to run certain powerup sequences. This type of check is **useful** in catching unexpected behavior within your DUT by coding worst-case checks like “*there should never be a case where the FSM remains in state ABC for 50 consecutive cycles*”.

These checks can be problematic when there is no clear timing spec but instead the microarchitecture says something vague such as “*to maximize performance, the XYZ IP should complete most ABC operations within 100 cycles*”. The check is easy to code, but when it gets false failures due to special cases it becomes a gray-zone nightmare (see 6.4.8).

The following are more complicated versions of timeouts that can also be used as hang checks:

- *This transaction has to complete before another 1000 transactions complete.*
- *This transaction has to complete before the interface is idle for 50 cycles 3 times.*

6.3.2 Periodic checks

Another checking event similar to a timeout event is a periodic check, where the checker periodically (every X number of clock cycles or X number of nanoseconds) comes alive and looks for certain things.

6.3.3 Hang checks

This type of checking event is monitoring activity within the DUT and seeing if operations are completing and if new operations are starting. This is a type of timeout check that specifically targets the situation where *something happened in the system and everything is stuck in a livelock or deadlock*. Livelocks and deadlocks are defined as:

- Livelocks: operations continue to flow through the DUT, but they never make forward progress towards completing because they continue to have conflicts with one another (typically over resources)
- Deadlocks: operations have ground to a halt within the DUT due to prioritization conflicts causing stalls or situations that have led to credits being lost or not returned

Another similar condition is Starvation. Starvation is defined as:

- One or more operations are not progressing due to prioritization of other endless stream of operations.

Some DUTs may have hardware logic that recognizes livelock and deadlock situations and attempts to break the lock by changing priorities and operation flow.

Creating checking events for deadlock, livelock and starvation detection is tough. The easiest versions of these are general timeout checks (make sure all read requests finish within X cycles), where the checking event is the original request and check fails if the operation is not complete within a certain amount of time. More complicated versions of the checks watch multiple interfaces and internal queues and FSMs and typically continually watch (and count) when things are happening on the interface and how long it has been since something has successfully happened.

6.3.4 End-of-Test checks

The end of the test is checking event for checks like “make sure all operations complete” and that “FSMs have returned to idle” and other such things. This is typically utilized in testbenches which *quiesce* themselves; i.e., they stop sending new operations, wait for operations to complete, and put the DUT in a state where it isn’t doing anything. Achieving a quiesced state is useful to aid your checkers: if you can assume your DUT will always quiesce, then you can create checks that say “*all operations must finish*”. In contrast, if you do not quiesce your DUT, and you simply kill your test while operations are still active, checkers will have a lot harder time telling whether an operation didn’t complete due to a bug or due to the simulation stopping abruptly.

A lot of DUTs have a *flush phase* to accomplish quiescence. In theory, a flush phase is as simple as “stop sending new operations into the DUT” and “waiting for pending operations to complete”. It is rarely that easy. A lot of IPs are internally active, meaning that at any given time they may wake up and doing something. For instance, a Display Engine might request more data to fill its buffers. A memory controller may send refresh operations to its memory DIMMs. IPs may reach a timer threshold and request a transition to a lower power state. All of these situations make it hard to perfectly achieve a quiesced state every time. Some teams have taken to setting defeatures and turning off timers when they enter their flush phase to lessen the amount of internally generated activity from an IP.

As you can see, the flush phase itself is waiting for items to finish, and thus becomes a natural point for checking. If the flush phase cannot quiesce the DUT, that in itself is an indication that the system is hung. And so the engineer has to decide which ‘checks’ to put into the flush phase itself (for instance, waiting for all operations to complete) and which checks it puts after the flush phase (for instance, make sure the gated clocks turned off).

6.3.4.1 considerations regarding flash phase

Waiting for all the DUT to “quite down” may be very long. (depending on the “depth” of the DUT) this time might be even longer if there are a few periodic events that may extend this phase even longer. The validation team should keep an eye on the length (in wall clock time) of this phase, to ensure it is not wasting simulation time. If not monitored, this phase can grow over time and reduce the validation effectiveness.

Another issue this stage may pose is that it may hide starvation scenarios. Since starvation of an operation relies on endless stream of higher priority operations, the flash stage of the test will stop the endless stream, and allow the starved operation to complete, hiding the buggy DUT.

6.4 Checking rules

Every checking operation refers to a rule or set of rules. Each rule defines DUT legal behavior in terms of DUT external interfaces or internal state. Rules are best when derived from specifications; when specs are not available rules can be derived from implementation. Rules can refer to a wide set of DUT behavior including timing, ordering, and values. Rules may define exact behavior (*this operation must get this specific value*) or ranges (*this operation must get one of these values*) or statistical behavior (*the average request response must be less than X*). Some rules always apply while some may apply to specific configuration or set of inputs, or specific time windows.

Rules define the DUT response in specific granularity and precision. i.e., what is the DUT response that we can apply the rule on. It can vary from signal behavior in specific cycle, thru the response to a protocol level and Architectural state, and up to memory bandwidth. See section 6.7 for more about modeling levels and precision.

The following sections explore rules in greater detail.

6.4.1 Source of Checking rules

The specification is the primary and most important source of rules. The spec defines the architectural intent for the DUT. Spec rules define the DUT response on architecturally defined DUT interfaces.

In addition to specification rules, checking can also apply implementing rules. Implementation rules are derived from the microarchitecture Spec (MAS) written by the Designer. They define the expected behavior of the DUT internal state and buses. Checking implementation rules is also known as *low level checking* and *white-box checking*. In theory, implementation rule checking is unnecessary; ie, the implementation is irrelevant as long as the overall DUT meets its spec. In

practice, there can be significant value added by checking implementation rules... but there are disadvantages as well.

6.4.1.1 The Value of Implementation Rules

The first advantage of checking implementation rules is that they help find specification bugs faster, i.e.: in more frequent scenarios. The theory is that the bug may occur often within your tests but the bug will not always manifest itself as a specification mismatch on the DUT outputs. Or, put another way, a bad thing might happen internally, but there may be plenty of cases where the bad thing gets cleared up prior to making it to the DUT outputs. Checking implementation rules typically causes checking of intermediate points inside the DUT; having intermediate points equates to more places to catch bugs. Some examples:

Example 1: Reading from a non-initialized memory can lead to x on the Dut interfaces, but since this might be very rare – adding a rule on memory initialization values can expose the bug early.

Example 2: On occasion, a buggy FSM goes into an invalid state. Within a few cycles, the FSM gets cleared back to a good state and continues. Only if a certain transaction interacts with the FSM while it is in the illegal state will the bug manifest itself on the DUT outputs. Having direct checking of the FSMs states will find the bug sooner... waiting for the FSM problem *to mix with other potentially rare cases such that a DUT output breaks a specification rule* may cause the bug to be found much later.

A second advantage derived from implementation rules is ease of debug. Checks of the implementation may expose the bug in the DUT closer to the root cause.

Example: A specification rule may define the final value of memory after an instruction. Checking this rule should be easy: the checker would simply compare DUT memory state after the instruction execution. However, the instruction execution may involve hundreds of steps, and thus finding the source of the failure may be very hard. Adding implementation rules can ease the debug by flagging errors closer to the point of the buggy code. Lowering the cost of debug can significantly improve the cost of Validation.

6.4.1.2 Disadvantages of implementation rules

The disadvantages of checking implementation rules lay in three points.

- The implementation rule may define a DUT response on an internal state... but that internal state itself may be poorly defined.
- The implementing rules tightly rely on the design implementation which may change for many reasons throughout the project (e.g.: due to timing or physical design needs) and thus require checker tuning.
- Since implementation rules are internal they have a higher likelihood of having special casing; meaning, the rule may not be applicable in some cases (e.g.: the logic knows that the rule would be broken in this case but no wrong behavior will occur)

All the above may lead to fragile checkers. Fragile checkers lead to much higher maintenance cost ([<add link>](#)) and loss of trust ([<add link>](#)), negating any gain in finding bugs sooner or debug latency.

Finally, there is an assumption that all implementation rules are backed up by higher-level specification rules. This is due to the fact that an implementation can be buggy and the implementation checker may have the same misconception of how the feature is going to work. In effect, the implementation and the checker mimic each other... and both are incorrect. The only way to catch this is to have solid specification checking. Never skimp on your specification checking, even if you think it is covered by implementation rules!

6.4.1.3 When to use implementation rules

Validation must weigh the advantages and disadvantages of utilizing implementation rules in checking. First and foremost, your checker strategy should prioritize spec-based rules prior to implementation rules. Second, checking of implementation rules makes far more sense in bottom-level DUTs (where RTL is validated for the first time). Validation at integration DUTs generally does far less checking of implementation rules. Finally, the greater the complexity of the DUT, and the greater likelihood of bugs getting ‘missed’, the more likely you need to supplement your spec-based checking with checking of implementation rules. They are powerful tools to ‘get evil’ on your DUT (see also [Evil Validation section: Checking at Lower Levels](#)).

6.4.2 Timing rules

Timing rules define the timing of DUT responses. Timing rules are commonly defined in one of the following ways:

- By a time measurement
 - *An ACK must be returned to the requesting agent within 100ns of the request*
- By a quantity of periodic events
 - *Data will follow the Valid signal by 2 clock cycles*

6.4.3 Ordering rules

Ordering rules define the order of the DUT responses. The order may define the strict legal ordering, e.g.: *The completion responses for write commands must be returned in the order the write commands arrived.*

6.4.4 Value rules

Value rules define the exact value and data of a given DUT response. The value can be specific, e.g.: “*value must be A*”, or a range of values, e.g.: “*value must be in the range [A-W]*”.

6.4.5 Exact Rules Vs. Statistical Rules

The checking can be greatly affected by how the rules define the expected behavior of the DUT. Rules can be divided into “exact rules” and “statistical rules”. Exact rules define exactly what is a legal or illegal DUT Response. Statistical rules define the expected distribution of the DUT responses.

Note: Typically, functional specs define exact rules and performance specs define statistical rules.

6.4.6 Conditional Rules

Rules may be *conditional*, meaning that the rule only applies in certain situations. This is further broken down into two types because it affects how checks are coded.

Static Conditional Rules

Static conditional rules are conditional rules that do not change *over time* within a test. Meaning, for a given context (a specific DUT, or a specific fuse recipe), the conditional rule will always be in effect. The rule either applies for the specific DUT, or does not.

For example: a rule defines the value of the parity bit but only if the parity detection logic is not fused off. In each simulation, the rule will be either disabled or enabled for the entirety of the test; the rule will not change along the way.

Dynamic Conditional Rules

Dynamic conditional rules are conditional rules that may change over time within a test; there are time windows in which the rule applies, and times it does not. The condition may change due to DUT configuration that may change during the test (e.g., dynamic modes set by control registers), or checks that are enabled while the DUT is in a certain state.

For example: a rule that describes legal transactions on a link is applicable only when the link is in active mode, but not in the training phase. In each test run this rule may be relevant or irrelevant in different time windows.

Note: The dynamic nature of such rules are one source of *checking gray zones*.

6.4.7 Implicit Rules: Deriving Negative (Illegal) Cases from Positive Specs

One issue with deriving rules from specifications is that most specifications define the *positive case*, ie, what is *supposed to happen*. Having the positive case defined is sufficient when coding positive validation collateral such as coverage, where you can code the exact case that is specified. Having the positive case defined for rules such as value rules is sufficient as well: an error should be flagged when the checking event occurs and the checked value is NOT the defined value from the spec. However, having only the positive case specified is problematic for ordering rules and occasionally timing rules.

Specs often define sequences of events. For example: event_A should be followed by event_B, which should then be followed by event_C. This definition requires a lot more detail prior to being able to check the sequence, such as:

- Can another event_A occur prior to the first sequence finishing? IE, can you have multiple instances of the same sequence running in parallel?
- Can *more* event_Bs happen within the sequence? Is it an error if we see event_A->event_B->event_B->event_C?
- Can extraneous event_Bs and event_Cs occur, outside of the A-B-C sequence?
- What other events are illegal during the sequence? Is it legal to see event_D between event_A and event_B?
 - And, when considering other events, can configuration changes and other such things happen *during* the sequence? If not, then that is an example of an *Illegal Event* that should be checked to not happen during the sequence. If it *is legal* to happen during the sequence, then stimulus/checking/coverage had better be coded to make it happen.

To derive rules from this sequence spec, further definition is required... and should be documented in the spec!

There are also positive specs that define what *can* happen without defining what *must* happen. For example, in PCI ordering rules there are rules that some operations must be able to overtake other operations. Completion operations should not be blocked behind read operations; instead, a completion operation should be able to pass a read operation. This rule is critical for PCI forward progress... but is very hard to check. The spec does not say that if a completion is behind a read that it must pass the read, since if they are both making forward progress through the fabric there is no reason for the completion to pass the read. Only when the completion is blocked does the mechanism come in to play; and thus the rule derived from this spec is likely to be more complicated than the original spec:

- Original spec: Completions must be able to overtake Reads
- Checking Rule: Completions cannot be stuck at any arbiter specifically due to being stuck behind a Read

When specifications only define the positive case and that leads to ambiguity when deriving checking rules, the specifications require further definition!

[note to mbair and Neriya – if we add a section on sequence checking – put a link from here!]

6.4.8 Rule Gray Zones: Undefined Behaviors (AKA Fuzziness)

There are situations where rules become *fuzzy*, where the DUT has entered a *gray zone* and a rule may become a “Don’t Care” or become much harder to define.

Gray zones can be the result of the following factors:

- Transitioning dynamic rule conditions, e.g.: change of the DUT configuration or mode, in which the expected DUT response is “don’t care” or just very loose.
- Rule that defines very high Precision in time, order, or data that the checking does not implement.

- Gap in Architectural or Microarchitectural Spec allowing implementation freedom.
- Hazard cases where the spec allows implementation freedom.

The following is a common example of a hazard case: a standard write-to-read checker on an interface is typically coded to say that *a read operation to address X should always return the data written by the most recent write operation to the same address*. That check is relatively straightforward to code. However, the check may involve *hazards*. If, for example, a write operation is allowed to occur on the interface between the read request being initiated on the interface and the return of the data... should the returned data be the newly written data, or the data that existed prior to the initiation of the read? The spec may simply say *it has to be one or the other*. So how should a checker deal with that?

Any such gray zone requires the checker to do one of the following:

- a. Reduce the grey zone by, for example, by sampling internal DUT responses, such that the checker can correctly choose between the set of responses it already determined were legal. Think of this in the following way: the checker already knows that the answer must be X or Y; if it looks at some internal DUT state, it can then use further checking code to decide whether it should check for X or Y.
- b. Align to the specific DUT implementation; meaning code a lot of extra *modeling* code within the checker such that it mimics the same calculations or timing or other logic within the DUT, and can thus determine an exact response.
- c. If there are multiple valid responses, check that the response is one of them, and do not do further effort to narrow it to a single correct response.
- d. Refrain from checking the DUT response in gray-zone cases.

All of these options have downsides; all of these options can lead to escapes. Selecting (a) or (b) may lead reduction of the checker stability or accuracy. In addition, they may increase the checker fragility. (b) in particular is something the validation community strives to do as little as possible but may be your best option when (c) has far too many possible responses built up over time.

Case Study:

For the sake of documentation, the following is an example where (b) was the clear winner over the other options. A simple *Core Monitor Hit* checker in the Client CCF (cache/core/fabric) IP checks all transactions to see if they should produce a ‘Monitor Hit’ event or not. (Monitor is an IA32 operation that sets address-based wake events for cores). This checker should theoretically do the following:

- 1) Track all incoming set_monitor opcodes and keep a database of their addresses
- 2) Track all incoming clear_monitor opcodes and clear their addresses from the database
- 3) Track all other operations and see if their addresses hit an active monitor in the database.
 - a. If they do: expect hit
 - b. If they do not: expect miss

What makes this checker hard is that the spec says that logic is allowed to flag monitor hits when there are not real hits. There is a performance penalty, but the situation is legal. Taking advantage

of that, the implementation is allowed to optimize their own database for tracking monitor addresses... and do interesting things on overflows, where on overflow conditions certain cores are assumed to *always get hit*.

Given the set of options that a checker can use to deal with this gray area rule, (c) doesn't work at all because the set of responses quickly becomes near-infinite. Moreover, tracking some internal state (a) was not helpful either. But (b), where the check simply mimicked that the RTL had a database of size X and all overflows resulted in more operations expecting hits, was created without an exorbitant amount of modeling code.

6.4.9 Separating and layering

Checking rules define the expected behavior of the DUT. When implementing TE checking of the rules, one may decide to separate the checking implementation to several checkers. Splitting the checking can:

- Improve the modularity of the checker.
E.g.: allow using different part of the checking in different Testbench or different stages of the projects.
- Improve the checker stability and maintenance.
E.g.: checking simpler rules even if the more complicated are not (yet) checked. (for example when handling grey areas <add link>)
- Allow better usage of Validation techniques.
E.g.: implement one rule as SVA, and another as post process checker

There are several ways to split the checking:

- According to DUT interfaces
- According to configurations or DUT status
- According to Arch and uArch rules
- According to rule definition of the DUT response (time, order, data or state)

6.5 DUT response

The DUT response represents the actual behavior of the DUT to which the checking operation applies rules. The response can be observed on DUT interfaces (internal or external) or DUT internal state.

Selecting how to collect the DUT response, how to manipulate it, and how to maintain the data until the checking point will be influenced by several factors, according to the following order.

1. The terms and elements defined in the rule. i.e.: if the rule defines the legality of a data item – we would like to manipulate the DUT response, as collected on the DUT interfaces, or internal state, until it matches the form of the rule data item.
2. The ease of back tracking the data from the checker failure point to the DUT response, and the ease of debug. It would be best to be able to easily match the failing scenario

back to the DUT interfaces and internal state, or to another trusted/understood interface such as a tracker file.

3. Performance of the checking

6.5.1 Interfaces

DUT response can be observed on the DUT interfaces. The interfaces can be architectural interfaces, or uArch interfaces (i.e., internal interfaces). Interfaces may be a set of DUT buses or DUT states such as configuration registers or readable memories. The DUT response could be the memory dump of a DMA device.

6.5.2 Selecting internal or External DUT response

Most checking rules define the expected DUT response on external buses. However, some implementations present the option to sample the interfaces also on internal interfaces. For example, sampling an internal bus or register file. In some cases the internal data will be simpler to sample, it may have the DUT response better organized, it may be closer in timing to the checking event, or it may be synchronized to other information required by the rule checking. All of this would make it much simpler to utilize the internal responses than the external response. However, this may lead to several issues:

1. The response is not the actual DUT response, and the checking may miss a bug. For example, the internal bus may behave correctly, but the values occasionally get corrupted or a transaction dropped before reaching the external bus.
2. The internal data requires strict alignment to the implementation, which will lead to fragile checkers.

The disadvantages of using internal interfaces mirror the disadvantages of doing implementation checking as discussed in section 6.4.1.2. Both the issues listed above should drive the checking to sample, except for rare cases, external DUT responses.

6.5.3 Performance consideration when selecting how to collect DUT responses.

Like any SW, checker can be written in effective or ineffective way. The performance aspects are

1. Memory, i.e. how much memory is needed to perform the checks. The memory consumptions can be derived from the consumption of data, e.g.: processing of the data stream of dut responses. And from the storing of information used for the checking. for example: maintaining lists of all outstanding access until the checking events.
2. CPU, i.e. how many actions are needed to evaluate the checking rules.

The main impact of the checker performance are:

1. The compute impact of performing the check. This impact any regressions and running of the checks.

2. The development and Debug TAT. This impact the development, debug and maintenance tasks.

6.6 Generating the legal expected response

As stated in <add link> checking includes the following actions:

1. Identifying the checking event
2. Deciding if there is an expected DUT response.
3. Evaluating the expected DUT response
4. Comparing the expected response to the actual DUT response
5. Reporting the error

Tasks 1,4 and 5 are implemented in the checker VC.

Task 2 and 3 can be implemented any of the following:

- The checker
- The reference model.
 - When the inputs are observed (a.k.a “ScoreBoard”)
 - At the checking event
- The stimulus engine

6.6.1 Which component should generate the expected response?

When the rules are simple, the task of generating the expected response (#2 and #3) is done in the Checker VC.

When the logic and algorithm of tasks 2 and 3 are complicated, they should be implemented by a dedicated “reference model” VC.

This separation has a few advantages.

1. The separation leads to two less complex VCs: the reference model and the checker. Each VC can focus on their own specific tasks such that each is easier to code, debug, and maintain.
2. The reference model can be re-used.
 - a. Taken from a high-level Arch modeling of the DUT.
 - b. Or shared between two test benches (once as a reference model, and once as an active BFM)

In rare cases the stimulus itself should generated the expected response. See section 6.6.1.1 for more depth on that concept.

6.6.1.1 Having the Stimulus generate the expected response

It would seem intuitive that stimulus generators, that are deciding what inputs to drive into the DUT, would be a logical place to generate the expected response as well. For instance, if the stimulus generator knows that it is sending an “ADD of 2 plus 3” that it should expect a response of “5”. This is not the recommended strategy in most cases, for a few reasons:

- Keeping components decoupled typically allows them to remain smaller and less complicated
- Decoupling checking components from stimulus ensures the checking components are portable to places that do not utilize the same stimulus. For example: it may be desirable to port the checker from IP to SoC; it is unlikely that the same stimulus generator would be ported as well.

There are times when there needs to be some connection between the stimulus generator and the response generator. If a checker needs to understand the *intention* of the stimulus generator then the stimulus generator is required to leave breadcrumbs for the checker to consume. For example, the stimulus may create a ‘high bandwidth’ scenario that it expects is checked for performance. Simply looking at the DUT inputs would not be enough information for the checker to deduce when to check for high bandwidth, so the stimulus generator needs to leave indications of its intentions for the checker to consume. The stimulus could leave a log with clues such as “cycle 20000: Running 500 back-to-back non-conflicting read operations with the intention of hitting full bandwidth”.

More examples of connection between stimulus and checking is described in [Special Case: Directed Tests](#)

6.6.2 Generating the expected DUT response

The checking rules define the DUT response as a function of the following factors: DUT state, inputs, and periodic events.

$$DUT\ response = F(DUT_{state}, Inputs, clock)$$

For the checker to generate the expected DUT response it should track all the effecting factors and apply the rules.

Note: not all rules require all factors.

6.6.2.1 Dut inputs

The DUT inputs. (as covered in [Interfaces](#)) Include all the inputs that effect the checking rules. The collation of inputs may arrive as a set of signals, transactions, set of transactions, memory images or flows.

6.6.2.2 DUT state.

The DUT State is a function of the following factors: DUT initial state, inputs, and periodic events.

$$DUT\ State = F(DUT_{init\ state}, Inputs, clock)$$

Note: not all DUT state is affected by all factors.

This means that to calculate the expected DUT response, the checker needs to hold the DUT initial state (i.e., the default reset values), track all the inputs that may change the state, and track all the periodic events that may change the state.

If the DUT function is simple (e.g., there is no periodic events factors) then it will be easy to track.

When the DUT state is complicated, one may decide to split the checking by sampling the DUT state from the DUT.

- Checker 1 checks that the DUT state is correct. It takes the DUT initial state and all the inputs over time, generates an expected DUT state, and compares it with the actual DUT state.
- Checker 2 checks ‘other things’ that require the DUT state as an input. It can use either the ‘expected DUT state’ from checker 1, or the actual DUT state that is being sampled.

The engineer that coding or debugging checker 2 can assume the DUT state input is correct because checker 1 would catch any difference between expected and sampled DUT state. This is another example of separating responsibilities that is critical for simplifying our environments. This method is often named “trust and verify”

6.6.2.3 Periodic events

The clock factor in the DUT response (and DUT state) are any periodic event that may affect the DUT behavior. It can be as simple as defining the number of clock toggles between the input and output (e.g.: the DUT response is 5 cycles after the input) or something far more complicated. The periodic events can sometimes be the only input factor in the DUT response calculation, for example, the DUT may hold a timer that counts down X cycles, then sends a transaction on its outputs. Such checking rules may create very cycle accurate checkers. See section 6.7 for more about the precision levels of checking.

And one way of simplifying checkers is separation of checking rules by using the separation and layering technique (see [Separating and layering](#))

6.6.2.4 Special Case: Directed Tests

Directed tests sometimes contain their own checking. This occurs when:

- The test needs specific checking that no other checker provides.
- No other tests require this type of checking, so there is no value in sharing.

Outside of that space, checking should generally be kept separate from tests.

Common cases of this kind of checking are destructive or survivability modes checking.

For example: consider a survivability mode that allows user to move an internal FSM to a new mode (to be used in post Si debug or patching.) this checking is very specific, covering a very

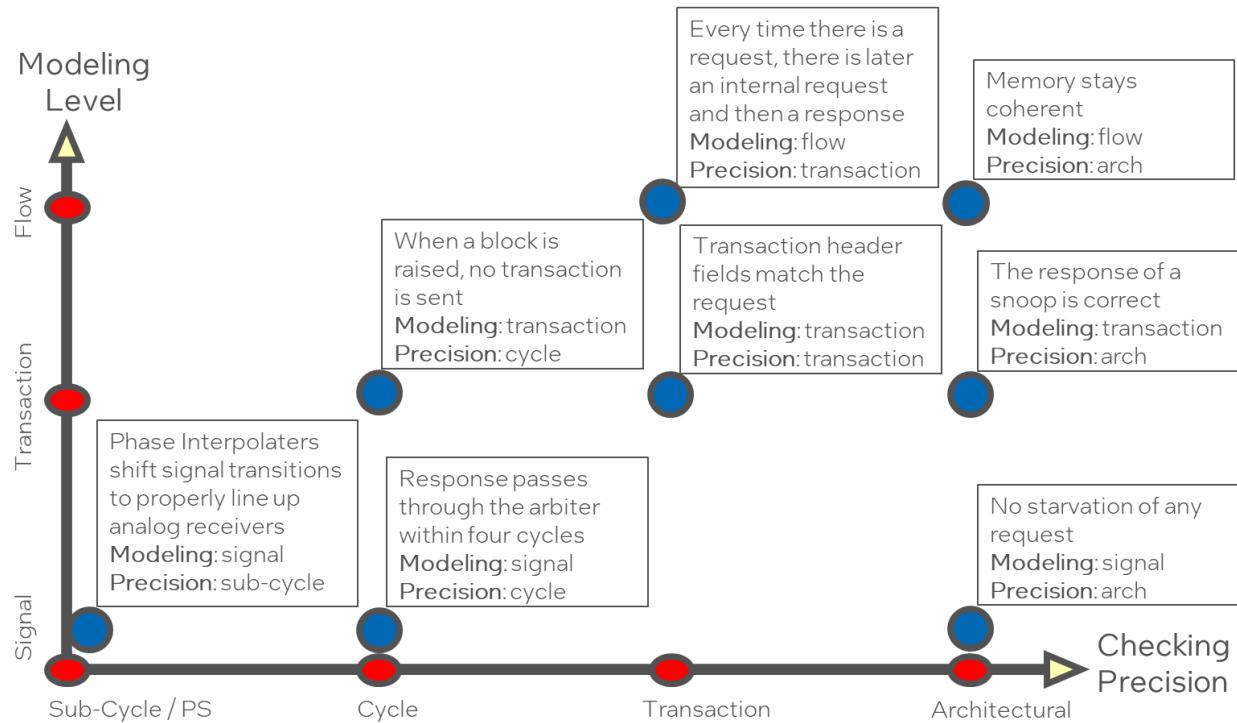
unique scenario, building a dedicated data base and checking rules. Building a self check test such that the test will bring the DUT to the desired “hang”, use the dedicated mechanisms to inject the correct FSM state and check the DUT response, is the correct solution.

Note: Many tests that run on silicon platforms, such as *Café* and *Raccoon*, run self-checking algorithms directly on the target platform since it would be very labor intensive to take traces from those platforms and process the checking separately. Conversely, when those same test generators run on simulation platforms, it is far more efficient to offload the checking to standard pre-si checkers. Find more information in the [Validation Platforms](#) chapter.

6.7 Transforming Data for Checker Consumption

Components within a test environment work best when the data that they consume comes to them at the right modeling level and precision. This allows the component to consume the data directly without transforming the data. Care must be taken to not architect a checker that requires data of vastly different precisions (*this part of the check needs cycle-accuracy, this other part simply deals with transaction ordering*) or different modeling levels (*this part of the checker relies on signal-level knowledge; that part of the checker relies on flow-level knowledge*). If you have checking rules that are of two different modeling levels or two different precisions, those rules probably belong in two different checkers. Checkers should have a natural best precision and best modeling level. You can typically tell this from the description of the checker alone – the description should give you some clue to the precision and modeling level required.

The diagram below, copied from [Data Flow within the Val Environment section: Modeling Level and Precision](#), gives you an idea of different styles of checkers and their precision and model levels.



The entire [Data Flow within the Val Environment](#) is critical reading to understand how and where to do the transforms such that the data input to the checker meets the needs of the checker without require significant transformations.

6.8 Checker requirements

Checker requirements, by priority, are:

1. Cover all the assumed legal space of inputs.
2. Find the bug as early as possible in the development process.
3. Find the bug as early as possible in terms of simulation time.
4. Make the bug as identifiable and easy to root cause as possible
 - a. See section 6.8.1 *Useful Error Messages*
5. Be cost effective, i.e.:
 - a. Minimize the number of false failures.
 - b. Be insensitive to design implementation choices and changes
 - c. Allow separate development of VC and DUT
6. Efficiency in terms of compute
7. Cover the illegal space of inputs; ie, handle the ‘negative space’
8. Reusability

6.8.1 Useful error messages

When the checker observes a divergence, the best thing it can do for the Validator is to output a useful error message. The bare minimum checker output would be the “expected” data and the “observed” data.

However, it is not sufficient for the error message to be descriptive to a human. It must also be contain information that is useful and well-formatted for triaging and bucketing systems. This will allow Validators to quickly disposition certain classes of failures and increase efficiency. For example, consider the following two fictitious error messages:

```
UVM_ERROR @ 4000000: Bad address 0x1122fb11 observed!

UVM_ERROR @ 4000000: Core sent an address that decodes to a region that is invalid
or doesn't exist.

Core: 2. Address: 0x1122fb11. Region: VGA. Region Valid: FALSE
Following is a table of known regions:
...

```

The first message is almost completely information-free. We do not know what rules it is using to determine that it should fail, and there is data printed in the middle of the message that will have to be cleaned up during triaging, in order to prevent redundant buckets.

The second message is multi-line. The first line gives a reasonably abstract description of the error, so triaging can cleanly bucket the failure with other similar failures. The subsequent lines contain deeper data used to give the debugger a head-start.

One more aspect of useful errors: the checker should be smart enough to know whether it should immediately kill the simulation as a result of the particular error. For normal functional failures, such as a data miscompare, it is generally undesirable to kill the simulation immediately as a result. However for cases where the test environment has determined that it has diverged from the DUT and is unlikely to recover, the checker should force the simulation to terminate. In practice, this is the difference between using an UVM_ERROR and an UVM_FATAL. This occurs in the post-processing world as well: there are errors found where the checker decides it should *check no further*.

Keep in mind that error messages should be as stable as possible once released. Altering error messages might harm customers of those error messages that are not necessarily known to the checker coder.

In summary, error messages should contain:

- A unique, but abstract, error signature that can be used in bucketing (see [Triage section: Buckets](#))
- Debug information to jumpstart the debug process
- The simulation time/cycle of the failure
- The file location and line number of the checker that fired
- Severity (Warn/Error/Fatal)

This applies to checkers that return error ‘objects’ as well; the objects must contain the items in the above list.

6.9 Checker Taxonomy: Categorization and Classification

This section will teach you from the ground-up how to *conceptualize and define* a checker. It starts with the basics and builds on many different categories of classification for checkers. The result should be a greater understanding of checkers in general as well as understanding that certain ‘choices’ need to be made prior to jumping into checker coding; moreover, this will give engineers a better vocabulary for describing checkers than our current overly simplistic terms such as “low level” and “high level” and so on.

This section defines 3 levels of checker taxonomy. Some checker types utilize all 3 levels, some use just two levels. These levels will be defined, and examples will be given. You will find that there is overlap between different types, where a specific check covers two types.

This taxonomy is useful beyond learning all of the types of checkers. Walking through the taxonomy list while writing (or reviewing) a validation plan will cause you to ask yourself *should there be a ‘continuity checker’ here? What about deadlock detection?*...and so on, through all of the checker types.

A taxonomy is helpful in other ways as well:

- Simplifying the Selection of best checker structure technology and inputs per given checking task.
- Lower validation holes (escapee) count by identifying each checker known limitations.
- Lower Effort by defining when to implement each checker (Vs. feature/project stability)
- Drive modular checkers set.
 - By splitting checking rule to several best suit checkers types.
- Reducing effort by readymade templets
 - And “golden” examples for each.
 - And perhaps use AI to create readymade checkers.
- Reduce ramp up time/effort of checkers ownership by unifying checker Arch.

The first level of the taxonomy includes 4 classifications of checkers:

- 1) Correctness
- 2) Completeness/Fairness/Deadlock
- 3) Performance
- 4) Security

Protocol and Timing		
Correctness	Req-Ack protocols	All handshakes checked at the signal level
	Single-transaction Correctness	The TXN, by itself, is self-consistent (header parts, length, rsvd fields, etc)
	Timing on an interface	Signal B will assert within 10ps of Signal A asserting
	Powerup, Reset, initialization checks	FSMs, interfaces, etc have these. Credit swap, etc
	Signal Propagation; Multi-Cycle-Paths; Isolation	
	Ordering	Did the sequences come out in the same order they went in? Were ordering rules followed?
Transformation		
	State Updates	Txn causes state to be updated (memory, cache, register, FSM..)
	Causality	Event/Txn causes some different sequence to start.
	Data Transform/Creation	Parity, ECC, scrambling, encode/decode, and all arithmetic/logic
Data Persistence		
	Read Data Checkers	Did the read of the register/buffer/cache/memory get the right data (coming from past writes/resets/forces)
	Save-Restore, Retention	Data remains through certain powered-off states
	Cache Coherency	Cache hierarchy remains consistent, data not lost, MESI protocols followed, memory appears as coherent as if no caches existed
	Memory Ordering	Cross-address and cross-instruction ordering maintained
Sequence and State		
	Illegal sequence of ops	General sequence checking; checks for illegal operations
	Illegal op during state	State based checks; "cannot see opcode ABC between entry and exit points", which is really just another sequence.
Completeness		
	Continuity	Txn coming in Port 1 eventually comes out Port 2
	Txn Flow Completeness	A txn finishes all of its spawned parts, including responses
	Livelock Detection	Transactions keep moving, but nothing finishes
	Deadlock detection	Transactions have stopped moving. Everything is stalled/dead
	Arbiter Fairness	Can operations get through arbiters in a fair way
Performance		
	Bandwidth	Are we meeting the bandwidth expectations at an interface
	Latency (txn perf)	Are transaction latencies reasonable? Look at min/max/mean/median/etc
	Latency (isoch)	Certain ops must finish within X time or the machine breaks
	Latency (power transitions)	The latency of power transitions fixes how deep we can go
	Power (clock gating)	
Security	Power (power gating)	
	Rights management	
	Side Channels	

6.9.1 Correctness Checkers

Correctness checkers are a first-level classification of checkers that watch events within the DUT and check that they are correct. These are broken down into the next level of sub-classes:

- Protocol and Timing checks
- Transformation checks
- Data Persistence checks
- Sequence and State checks

6.9.1.1 Protocol and Timing Checks

These checkers do signal-level

6.9.1.1.1 Req-Ack protocols

6.9.1.1.2 Single-transaction Correctness

6.9.1.1.3 Timing on an interface

6.9.1.1.4 Powerup, Reset, initialization checks

6.9.1.1.5 Signal Propagation; MCP; Isolation

6.9.1.1.6 Ordering

6.9.1.2 Transformation Checks

6.9.1.2.1 State Updates

6.9.1.2.2 Causality

6.9.1.2.3 Data Transform/Creation

6.9.1.3 Data Persistence

6.9.1.3.1 Read Data Checkers

6.9.1.3.2 Save-Restore, Retention

6.9.1.3.3 Cache Coherency

6.9.1.3.4 Memory Ordering

6.9.1.4 Sequence and State

6.9.1.4.1 Illegal sequence of ops**6.9.1.4.2 Illegal op during state**

6.9.2 Completeness Checkers

6.9.2.1 Continuity Checkers

A *continuity checker* checks that items that enter a block eventually exit that block. It flags errors when something gets ‘dropped’. The items may change along the way through the block, they may split into multiple items that eventually come out of the block, and in some cases it may be legal that they do not leave the block.

Continuity checking can happen at a very low level, such as *all items that enter this arbiter eventually come out of it*. The same for a fifo, queue, etc. Formal verification is often the best avenue for thorough continuity checking.

Continuity checking also happens at a high level as well, typically in the form of *transactions coming into the XYZ IP should eventually leave it*. Post-processing checking is an excellent method for high level continuity checking.

6.9.2.2 Txn Flow Completeness

6.9.2.3 Livelock Detection

A *livelock* is a situation where the DUT cannot make forward progress, but the critical items that *should* be making forward progress continue to move about the machine and do things, but never ‘complete’. Operations may be entering and leaving queues, read operations may be getting data, but they will end up doing the same thing again and again. Typically this is due to one set of operations causing another set of operations to have to restart, and vice-versa.

Livelock detection checkers watch for critical signs of forward progress and flag when no forward progress has been made after some timeout length.

6.9.2.4 Deadlock detection

A *deadlock* is a situation where the DUT cannot make forward progress and the critical items that *should* be making forward progress are stuck and cannot move. This can happen for a number of reasons, such as: critical resources ‘disappearing’; one type of operation blocking other types of operations from completing; pointers getting corrupted... the list goes on and on.

Deadlock detection checkers watch for critical signs of forward progress and flag when no forward progress has been made after some timeout length.

Livelock and deadlock detection are both considered *hang detection*

6.9.2.5 Arbiter Fairness

6.9.3 Performance Checkers

6.9.3.1 Bandwidth

6.9.3.2 Latency (txn) (perf)

6.9.3.3 Latency (isoch)

6.9.3.4 Latency (power transitions)

6.9.3.5 Power (clock gating)

6.9.3.6 Power (power gating)

6.9.4 Security Checkers

6.9.4.1 Rights management

6.9.4.2 Side Channels

6.10 Testing your Checker

There are two predominant ways to test that your checker is working properly.

- Unit tests
- Checker code coverage or checker coverage

Both bring unique value in terms of making sure your checker (and your overall environment) are working as expected. It is recommended that you do both. The following sections explain these in detail.

It should be noted that it is also valuable to do code reviews of your checker.

6.10.1 Unit Tests

Unit testing, and in fact test-driven development (TDD), bring value to checker development and maintenance via:

- You can verify that every unique error, and every pathway to get to each error, can be caught by the checker
- The turnaround time of unit tests is very fast

- When you decide to refactor your code, your collection of unit tests can quickly provide feedback as to whether your checker is no longer functioning properly (especially if they are written at the level of ‘checker inputs’).

Unit tests are the best way to show that the checker catches a specific error.

6.10.2 Checker Code Coverage or Checker Coverage

Checker code coverage is automated code coverage of your checker. *Checker coverage* is checker code coverage of the conditional statement immediately prior to each unique error. Take this piece of code:

```

132:     check_op = op_is_write_type and addr_is_out_of_bounds
133:     if check_op:
134:         machine_check_occurred = saw_mc_1 or saw_mc_2
135:         if not machine_check_occurred:
136:             ERROR("Did not see MC on out-of-bounds write")

```

When doing code coverage of this checker, it would report whether you hit all of these lines. In fact, it might be hard to hit line 136 unless you actually caught the error.

Checker coverage for this code would only care whether you have hit line 135. The purpose of checker coverage is to say whether, in a ‘live environment’ (ie, your testruns), has the condition code of every single check been executed. If it has not, it could mean there is a bug in the checker, or it could mean that the stimulus has not put us into a situation that required the check. Both, of course, are interesting to Validation! Checker coverage is a great way to minimize code coverage down to the critical interesting code lines.

Checker coverage is the best way to show that the checker is actively checking a specific rule in a live environment.

6.11 Understanding Checking Escapes

The QoV2.0 (Quality of Validation: <https://goto.intel.com/qov>) chapter on checking quality focuses on measuring and reviewing your checking plan to make sure that you do not have bug escapes due to checking misses. To frame this discussion, the chapter grouped checking escapes into these 6 categories:

1. No checking planned: there was an incomplete accounting for behaviors in the specifications (either HAS or MAS) when creating checking plan
2. Incomplete checking: checking was planned but did not account for all aspects of a feature or piece of functionality. This could include missing fields, a missing opcode, missing leg of a state machine, etc
3. Unclear specification: the checker was purposely coded to allow/expect a behavior that, in reality, should have been an error

4. Checking disabled: checking existed for the fault, but was disabled when an error condition was encountered, either statically or dynamically
5. Checking bug: all planning had accounted for the check, code was written and active when the fault was encountered, but the checking code was written in a way that missed the error
6. Checking trusts/matches RTL uarch: the checking used hints from, or coded the algorithm of, the planned design microarchitecture of the architectural feature. This caused the checker to follow the same faulty path as the errant design

Categories 1 and 2 focus on creating a complete plan for checking, without missing any necessary checks. Having a review process for your checking plan (or overall validation plan) is critical to catch these types of escapes.

Category 3 requires disciplined HAS/MAS reviewing, as well as an eye for questioning vague or possibly conflicting specifications when writing the checker. Machine readable specs and/or AI might be able to help here as well.

Categories 4, 5, and 6 might be found using unit tests and checker coverage. If your unit tests set up the failure and yet the checker fails to hit it, it is a sure sign something is wrong. If you run regressions and yet your checker coverage shows that you do not hit the checker trigger condition, it is possible that there is a bug in the checker or that the checker is being disabled somehow.

6.12 Using a Checker as a Debug Aid

In some platforms (see [Validation Platforms](#)) it does not make sense to run all checkers all the time. For instance, in the emulation platform it can run so many cycles that running a lot of checkers against emulation traces would be compute cost prohibitive. But... if the emulation test run fails some high level checks, it might be worthwhile to run more checks against the emulation traces to narrow down and help debug the problem that the original checker flagged.

For example, if the emulation test flagged a simple “*test did not complete correctly*” error, well, that can mean a lot of different items could have gone wrong. That might be a good time to run a round of checkers versus the emulation trace to see what problems that the next set of checkers can unearth.

In such a methodology these checkers are not being used to fail the test – the test already failed. Instead, the checkers are being used as a mechanism to narrow in on *what went wrong*. In some sense, they are helping us get to a more fine-grained bucket that something coarse like “*test did not complete correctly*”.

6.13 Trust

(TBD)

- Trust is built over time
 - Every false error ruins the trust

- Every bug caught builds trust
- Missed bugs don't have as much of a negative vibe as a false positive
 - A missed bug usually means: "We need some more work to catch this"
- Better cover less, but be sure about it rather than covering more but with a lower degree of certainty

7 Summary

(Original) Checking is one of the three pillars of Validation. Checkers are validation collateral whose purpose is to identify bugs. Several aspects of a checker directly affect the short-term productivity and long-term efficiency of the Validation team. There are many classes of checkers, ranging from low-level checkers such as SVAs to high-level checkers such as architectural models. When planning checking for a project or feature, there are important factors to consider that will determine the type of checkers to write.

8 Future Work

Fill out more detail in the 'Checker Requirements' section. Give more detail.

From Art of Val page:

- Discuss sequence Checking - why it is hard (easy to identify what 'should' happen, hard to identify the illegal space)
- Add to 'error messages': lots of new learnings here from post-proc initiatives and how non-validators are reacting positively to some types of error messages
- Pros and cons of defining the legal conditions (and all else is forbidden) versus defining the illegal conditions directly.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 8

Coverage

By: [Michael Bair](#) and [Igor Beskorovaynyy](#)

1 Abstract

Coverage is a tool that tells whether a test or a formal proof has hit a specific condition. Coverage is typically used with methodologies such as random testing that do not have a high probability of hitting their intended targets on their own. Both Pre-Silicon and Post-Silicon Validation use coverage. Coverage can be used for a range of areas, from architectural and microarchitectural conditions to software coverage. Coverage typically targets conditions within the DUT, but is also used to verify test environment capabilities.

Coverage provides a metric of completeness to Validation, which is typically used in conjunction with other metrics to gauge Validation progress and in some cases RTL health.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/04/2003	First release version.	Michael Bair	Matt Kupperman / DPG-N uAV MWG
1.1	1/7/04	1 full round of L3 review	Michael Bair	Paul Schwabe
2.0	3/30/05	Rewrite and addition of new sections	Michael Bair	Matt Kupperman / DPG-N uAV MWG
2.1	5/16/05	Post-L1 update	Michael Bair	Geetha Ravishankar
3.0	7/15/05	Rewrite to add Coverage Depth	Michael Bair	
3.1	12/20/09	Small Additions/Updates	Michael Bair	
4.0	5/1/12	Rewrite to cover all of validation, FV, Sleuth, and to pare down and simplify	Michael Bair	Brad Kelly, Bob Grim, Allan Rudwick, CCDO Val staff
5.0	3/15/16	Rewrite of the Functional Coverage section. New sections on TE Coverage and Coverage Reuse.	Igor Beskorovaynyy	Michael Bair

3 Contents

1 Abstract.....	205
2 Revision History.....	206
3 Contents.....	207
4 Purpose.....	209
4.1 Why do we need this chapter?.....	209
4.2 What does this chapter cover?	209
4.3 What does this chapter not cover?	209
5 Background Concepts	209
5.1 Random test	209
5.2 Test generator	209
5.3 Condition	209
5.4 Onion Peeling	210
6 Coverage.....	210
6.1 An introduction to coverage	210
6.1.1 What is coverage?	210
6.1.2 Why use coverage?	210
6.1.3 Targeting beyond the enumerated coverage space.....	211
6.1.4 The relationship of stimulus, checking, and coverage	215
6.1.5 Condition Complexity and Coverage Depth	216
6.2 Types of coverage	217
6.2.1 Architectural coverage	217
6.2.2 Functional coverage (aka microarchitectural coverage).....	218
6.2.3 Software / code coverage	221
6.2.4 Input coverage (aka TE input coverage).....	221
6.2.5 TE coverage	222
6.2.6 Toggle coverage	222
6.3 Implementing coverage-based Validation.....	223
6.3.1 Developing coverage conditions	223
6.3.2 Writing the coverage monitor	224
6.3.3 Writing and running tests	224
6.3.4 Running coverage monitors with tests	225

6.3.5	Analyzing coverage	225
6.3.6	Feedback to the testplan.....	226
6.3.7	Exit review and the high-water mark.....	226
6.3.8	Coverage during stepping validation.....	226
6.4	Coverage Reuse.....	227
6.4.1	QuickCov Methodology.....	227
6.4.2	IP Coverage.....	228
6.4.3	SOC Simulation Coverage.....	228
6.4.4	SOC Emulation Coverage.....	229
6.4.5	Post-Silicon Coverage	229
6.4.6	Coverage and PLC	230
6.5	Coverage goals and important clarifications	231
6.5.1	Coverage indicators	232
6.5.2	Prioritizing coverage versus other activities	232
6.5.3	Coverage is not the end result.	233
6.6	Other uses for coverage	233
6.6.1	Test-suite minimization	233
7	Summary.....	234
8	Future Work.....	234
9	References.....	234

4 Purpose

4.1 Why do we need this chapter?

Coverage is one of the three pillars of Validation, but it is an often misunderstood and misused aspect of Validation. There is a delicate balance that Validation must strike when allotting effort to each of the pillars because too much emphasis on coverage can lead to teams cutting corners on stimulus and checking while too little emphasis can lead to bug escapes due to un-covered conditions.

This chapter provides Validators with an introduction to coverage methodology. It also explains *why* we need coverage. It will also document some of the surprises, nuances, and pitfalls within the methodology.

4.2 What does this chapter cover?

This chapter will cover the definition of coverage-based Validation, many types of coverage, and many nuances of coverage-based Validation.

4.3 What does this chapter not cover?

This chapter does not cover all of the tradeoffs of coverage-based Validation versus other styles of Validation. It does not discuss when it is or is not appropriate to use coverage-based Validation. It also does not cover the specific tools used for coverage.

5 Background Concepts

5.1 Random test

See [Stimulus section: Directed Stimulus vs. Random Stimulus](#).

5.2 Test generator

See [Stimulus section: Test Generator](#).

5.3 Condition

A condition is a logical (and possibly temporal) set of events that exist within a system. The system, in the case of Pre-Silicon Validation, is the RTL model. Simple examples of conditions include:

- See a 32-byte write transaction

- See signal X assert
- See Core A sending an inter-processor interrupt when Core B is in a sleep state.

5.4 Onion Peeling

Onion peeling is the term used to define a serialized set of bug finding and fixing that it takes to find most of the bugs in an area. For instance, as a Validator finds a bug in area X, he finds that he cannot proceed with a greater depth of testing in the area until the bug is fixed because the majority of these tests will fail due to the first bug. He cannot get to the next layer until the first bug is fixed. When you consider that an area may have many layers of bugs to go through, you realize that doing serialized finding-then-fixing-then-finding may have extremely bad repercussions to project schedule.

6 Coverage

Coverage is one of the Three Pillars of Validation, along with [Checking](#) and [Stimulus](#). (See [Introduction to Pre-Silicon Validation section: The three pillars of Validation](#))

6.1 An introduction to coverage

6.1.1 What is coverage?

Coverage is a tracking mechanism that records whether or not stimulus has hit its targeted conditions. In its simplest form, coverage provides a hit/miss indication, or perhaps a hit count, for each condition. Coverage can provide more than that. By utilizing coverage within a comprehensive methodology along with testplans, stimulus, and checking, coverage becomes more than just an indicator of whether a condition is hit. It can also be:

- A measurement of stimulus quality
- A measurement of RTL quality and Model Health
- A measurement of Validation progress and completeness

This ‘comprehensive methodology’ makes coverage truly powerful for Validation and the entire project. But what exactly does this comprehensive methodology entail? It requires that stimulus be written in a certain way, depending on the target. It depends on good testplan writing with reviews by experts and stakeholders. And it requires feedback loops from activities such as bug finding. More on this throughout the rest of this chapter.

6.1.2 Why use coverage?

Stimulus, in the form of directed tests, random tests, or formal proofs, does not always do what it is expected to do. Random tests, for example, may simply not hit a condition due to probability, user error, or shortcomings in the test generator. A directed test may be written incorrectly and fail

to exercise the intended condition. Even if written correctly, a directed test may stop hitting the intended condition because of later changes in the DUT or environment. Formal proofs may miss a condition due to improper input assumptions.

Validation has come to realize that **if there is no independent check that stimulus did what it is supposed to do, it probably didn't**. This is true even for the simplest directed tests. Though counterintuitive, this lesson has been reinforced by example after example.

Some reasons a test might fail to exercise its target conditions include:

- **The complexity of the condition:** the greater the number of events that need to align to hit the condition, the harder it is to hit.
- **The distance between input control and the target logic within the DUT:** the farther the targeted logic is from the input, the harder it is to control the target logic using the input.
- **The nuances of the DUT:** the DUT may make it close to impossible to hit certain scenarios though the scenario is considered completely legal.
- **TE/TG bug:** Validation collateral such as the test environment or test generators may be buggy and hence cause it to not exercise some feature.
- **User error:** the user commits an error in test setup or in biasing the test.

There are many reasons why stimulus might not hit a condition; attempting to understand them all and contain them is nearly impossible. Instead, it is more efficient to write a coverage monitor to check whether the condition was properly hit. Only if the coverage monitor reports a miss does Validation investigate where the problem is.

6.1.3 Targeting beyond the enumerated coverage space

Some Validation tasks have a condition space that is finite and *known*, ie, the condition space can be enumerated, targeted with stimulus, and monitored for completion. The stimulus is not required to hit anything outside of the condition space, and thus any form of stimulus can be utilized, ranging from directed to random tests.

Other Validation tasks have near-infinite condition spaces; the condition space cannot be enumerated. In this situation, a subset of the condition space is used as a proxy for the near-infinite condition space. This strategy requires stimulus to hit conditions beyond the enumerated coverage space, thus placing additional responsibility on the stimulus. In this situation, Validation typically relies on formal proofs or random testing as stimulus.

Figure 13 illustrates targeting near-infinite conditions with a small subset of enumerated coverage conditions.

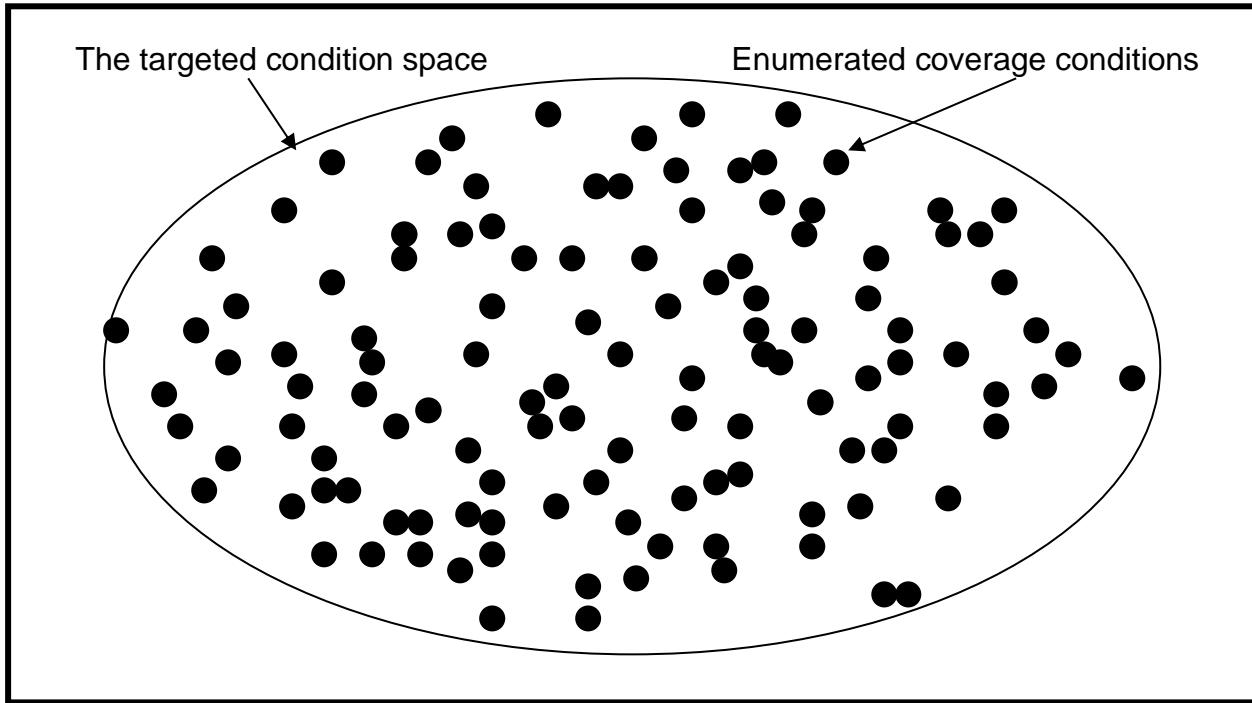


Figure 13

In this situation, stimulus is responsible for hitting more than just the enumerated coverage conditions, but also the *space between the conditions*. To illustrate this, Figure 14 shows a bug that lies between the enumerated coverage conditions. The bug may be logically similar to its neighboring coverage conditions, and has a high likelihood of being more complicated than any of the coverage conditions.

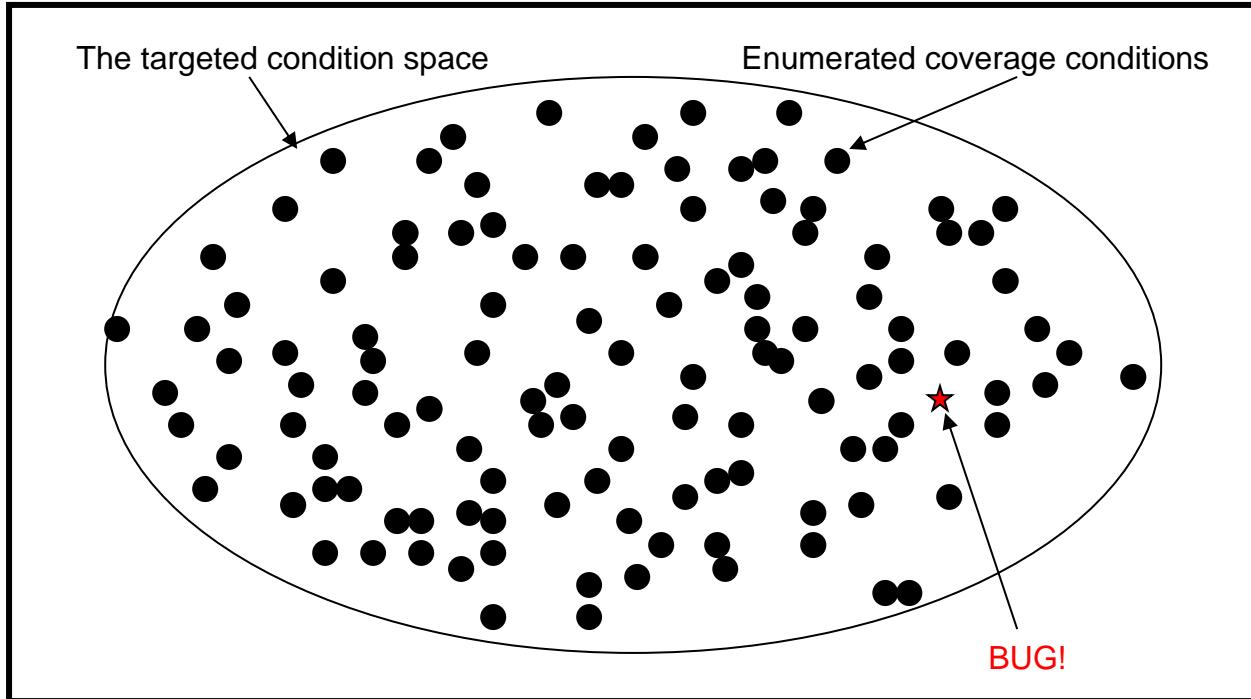


Figure 14

The stimulus is responsible for finding this bug with coverage conditions that are nearby. To do this, Validation uses random testing (see [Stimulus section: Directed Stimulus vs. Random Stimulus](#)) and [Formal Property Verification](#). Both of these methods use coverage but are expected to hit conditions far beyond the coverage space. We make an assumption that if the stimulus hits the neighboring conditions, that it will likely hit some of the conditions in-between them as well. Assuming that Validation hits the Blue, Green, Yellow, Gray, and Brown coverage conditions (Figure 15), it is likely that they will hit the bug as well.

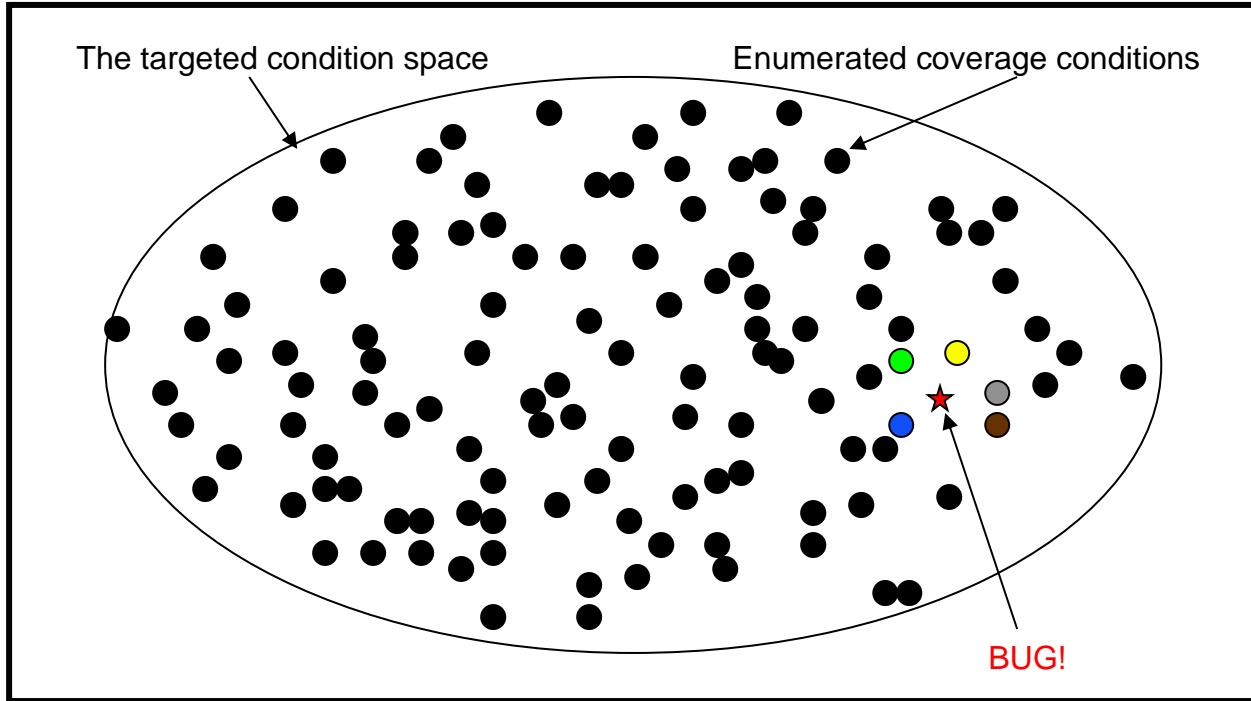


Figure 15

Hitting conditions within the coverage space with two non-overlapping test generators creates a hole that coverage cannot reveal. In Figure 16, the Blue, Green, Yellow, Gray, and Brown conditions were hit – but by different random test generators. Having these neighboring conditions hit by separate test generators makes it less likely that the bug will be hit by the overall Validation effort. Thus, when targeting beyond the enumerated coverage space, the greater the segmentation of the testing effort with multiple test generators the greater the risk of bugs escaping. This does not mean that Validation cannot use multiple test generators. It is simply one more thing to consider when architecting your stimulus and coverage plan. See [Stimulus section: One vs. many stimulus generators](#) for further discussion on this topic.

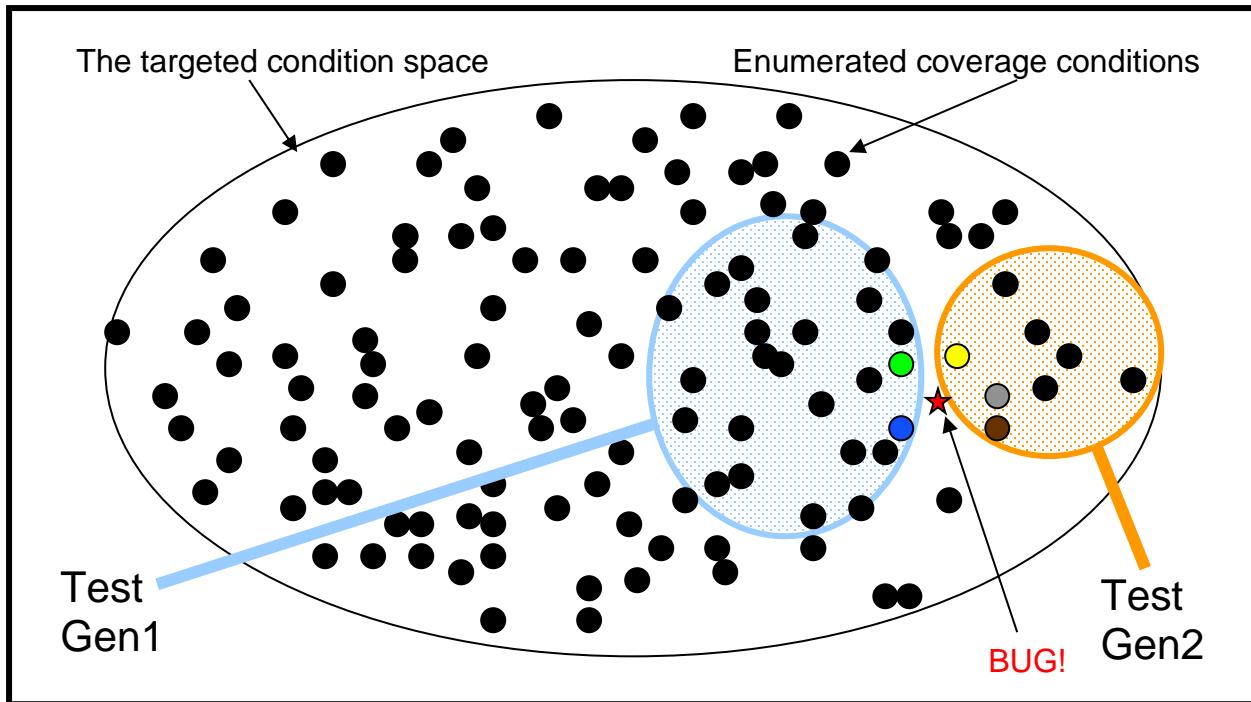


Figure 16

The methodology of targeting beyond the enumerated coverage space requires a properly distributed coverage space else there may be no coverage conditions anywhere close to the bug. This is a balance point for Validation – weighing the cost of creating more (and possibly more complex) coverage points versus the risk of having a bug escape because the cover points are too far apart. Many other factors, such as assumed RTL health, test generator maturity, coverage analysis methodology, and Validation funding also affect this balance and help determine the amount of investment that should be spent on coverage.

6.1.4 The relationship of stimulus, checking, and coverage

From a logical point of view, the coverage effort should not begin in an area until the stimulus creation effort in that area is complete (test environment and test generators coded). Driving coverage in an area where stimulus is not yet done will only tell you something you already know... that stimulus is not yet done. However, this does not mean the stimulus and coverage efforts should not be serialized. Validation may need to parallelize the initial coverage effort with the stimulus effort to minimize the overall schedule.

Checking and coverage efforts are typically orthogonal and done in parallel. However, if forced to choose between one and the other, checking will typically be higher priority because many bugs will not be exposed without a checker even if stimulus hits them. Coverage, however, is not necessary to find any bug; it is a backup step to make sure the stimulus is as good as we expect it should be.

Beyond simple prioritization, there is also effort balancing that can be done between the three pillars to optimize ROI. See [Introduction to Pre-Silicon Validation section: Optimizing ROI among the three pillars](#) for more details.

6.1.5 Condition Complexity and Coverage Depth

A condition within the DUT has a certain level of complexity. Complexity, in this sense, generally constitutes how hard the condition is to hit. Many conditions within the DUT are so complex that they are beyond Pre-Silicon Validation's capability of hitting (within the time, staffing, and compute resource limits of the project). These conditions are therefore left for Post-Silicon Validation to hit, or possibly never to be hit. Pre-Silicon Validation is responsible for hitting the conditions of lower complexity, especially those that might block Post-Silicon Validation.

Figure 17 is a conceptual diagram of condition complexity. It shows a range of condition complexity from the most basic *event coverage* conditions to conditions so complex they are probably never hit during the lifetime of the millions of parts we sell. The figure also attempts to demonstrate the relative number of conditions at each individual area – the dots represent actual data, otherwise the line is drawn to simply show that the condition count is increasing.

Within Figure 17 the first three classes of condition complexity are *event coverage*, *breadth coverage*, and *depth coverage*. They are named for the coverage methodology that utilized them (more on this in section 6.2).

The next condition complexity levels are those reached by random testing in simulation, emulation, and in silicon. The principal difference in the depth of coverage between them is that the added speed benefit going from simulation to emulation and then to silicon gives the test generator radically more cycles in which to hit conditions.

Even with the speed of silicon, we can barely scratch the surface of the number of conditions within the microprocessor. Beyond the complexity level of random testing on silicon is a class known as *errata*. These conditions are so hard to hit that we typically expect that customers will not or cannot hit them, and even if we find a bug in that area, we may decide not to fix it.

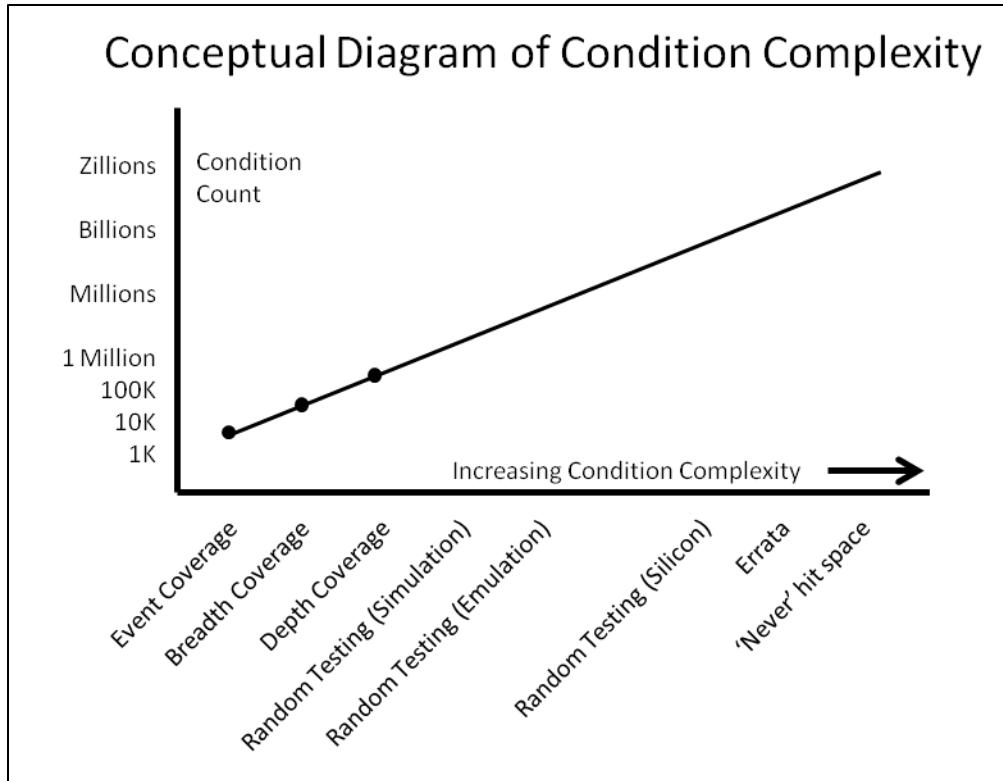


Figure 17

An important lesson from this section is that the condition space that we target in simulation, emulation, and silicon is way bigger than what we can code within coverage monitors. As we don't know the exact size of the targeted condition space, the coverage numbers that we get cannot tell us exactly how well we are hitting our targeted condition space; these numbers should only be considered as some approximation.

6.2 Types of coverage

There are many different types of coverage, generally separated by their methodology or the types of conditions they target. Some of them overlap, but not all do.

6.2.1 Architectural coverage

Architectural coverage is coverage of the high-level specification; ie, it is implementation-agnostic. It can be considered black-box coverage. For instance, if a Validator reads the Software Developers Manual or a C-Spec for one of our devices, they can immediately begin deriving architectural conditions to target, without knowing the details of the implementation.

Because architectural coverage is implementation-agnostic, the coverage monitor does not have to be run on the DUT – it can be run on any suitable simulator that matches the architecture of the

DUT. If the simulator hits the architectural condition, then the DUT will hit the architectural condition as well (of course, this assumes that the two actually match!).

The [Architecture Validation](#) uses architectural coverage which is provided by the same architectural simulator that is used as a checker during DUT simulation. It provides coverage and simultaneously verifies that the DUT is doing the correct architectural thing.

Architectural coverage takes real effort to code, but once coded it can be used on many projects since it does not have to be tied to any specific implementation. Currently, all IA-core IPs within Intel utilize architectural coverage.

6.2.2 Functional coverage (aka microarchitectural coverage)

Functional coverage is coverage of the micro-architectural aspects of a design; ie, the implementation itself. It is considered white-box coverage. Functional coverage is derived from reading the Micro-Architecture Specification (MAS) if one exists, as well as by gathering a deep understanding of the implementation (typically by reading and debugging RTL). See [Becoming the Microarchitecture Expert](#) for more details on learning the microarchitecture and implementation.

Functional coverage is often coded utilizing low level signals within the DUT – items such as valid bits, opcodes, addresses, and events such as FSM transitions, stalls, and clearing conditions. Functional coverage can be coded without utilizing DUT signals directly by relying on an intermediary to connect to the DUT directly and to track complicated protocols. Test environments and debug trackers are example intermediaries that can be used for functional coverage.

Functional coverage comes in many different forms, a few of which are discussed here.

6.2.2.1 Standard functional coverage

‘Standard’ functional coverage is the most basic form of functional coverage. This form utilizes the coverage condition as a Boolean – “did I hit the condition or not”, while also keeping track of how many times the condition has been hit.

The standard functional coverage is subdivided into breadth and depth coverage.

Breadth coverage is a set of simple one or two-dimensional conditions that are evenly distributed across the whole targeted condition space. Breadth coverage attempts to walk through all main events, states and modes covering all basic pieces of logic. The main task of breadth coverage is to make sure that every basic piece of functionality is being exercised and is not missed by the stimulus.

Breadth coverage usually doesn’t include any timing or complex cross coverage conditions, but it can include states that can be relatively difficult to reach. For instance, these are all examples of breadth coverage conditions:

- See Queue A getting full
- See interrupt received when Core X is in a sleep state
- See event B happening when FSM is in the Y state

Depth coverage is a set of coverage conditions that are more complex than breadth coverage. Depth coverage conditions are usually multi-dimensional (i.e. they cross more than two different events, states or modes) and can contain timing expressions. These conditions cover complex microarchitectural scenarios that aim to prove that given logic is deeply exercised with various types of inputs, states and timing dependencies. Depth coverage looks at a design at a much lower level which results in a significantly larger number of depth conditions than the number of breadth conditions.

These are few examples of depth coverage conditions:

- See a MUL instruction followed in 3 cycles by a SUB instruction
- See a cache lookup from CPU thread A targeting set X way Y of the cache, and a pipeline clearing event of type Z received at the same time
- See a sequence of back-to-back {Rd, Wr, Rd, Wr} requests on a memory bus.

Depth coverage conditions are often much harder to hit than breadth conditions, but they can give a better confidence that design is healthy.

Together with breadth coverage, depth coverage had been widely used to evaluate design health on many generations of CPUs. Coding deep coverage conditions and analyzing coverage from these conditions had been taking significant amount of time of projects' schedules. To reduce the amount of time taken to validate a project and improve overall ROI, most teams have gone away from the concept of full breadth and depth coverage-based validation. Instead, many teams use breadth coverage with some limited number of depth conditions with no or almost no timing dependencies. The reduced amount of the coverage data can be balanced by enabling emulation validation at early stages of projects and using more robust random test generators. Note that breadth coverage is still highly important with this approach as it gives a good hope that test generators are able to produce complex multi-dimensional conditions even though these conditions are not explicitly coded as depth coverage conditions.

6.2.2.2 Frequency coverage

Frequency coverage is a form of functional coverage that derives more than just basic hit/miss status from the coverage conditions. Frequency coverage tracks the frequency of the coverage events, typically as measured in events per cycle (or some factor, such as events per 10k cycles). Frequency coverage also tracks distribution, which is the percentage of tests which hit a given condition at least once.

Frequency coverage gives Validation a tool to help fine-tune random testing to maximize its effectiveness. Effective random testing happens when an event occurs often enough to interact with other events but not too often as to begin precluding other events from happening. Validation uses frequency coverage to target the most effective collisions of events; though it may not measure whether tests actually see greater event interaction, Validation can infer that more interaction is happening because they are creating the best possible environment for it to happen.

Another way to look at this is that frequency coverage allows Validation to derive deeper information than the level at which it is coded. Most teams that code frequency coverage do so at the level of *event coverage* (see Figure 17) which creates around 1k-10k conditions per project.

While this is a much shallower depth of coverage than breadth or depth standard functional coverage, the results can be similar because proper frequency coverage helps direct random testing to where it is more likely to naturally hit breadth and depth coverage.

Frequency coverage can also be used as an addition to the standard functional coverage showing which breadth or even depth conditions are not hit frequently enough and flagging that as a potential concern.

6.2.2.3 Sleuth coverage

Sleuth coverage, similar to frequency coverage, attempts to derive deeper coverage information from simpler coverage conditions. Sleuth coverage utilizes *event coverage* (see Figure 17) for low-investment coverage monitor coding, but in addition to monitoring hits of the simple events, it also monitors cross-products of these events. For example, if coverage conditions X and Y are both hit, then the coverage monitors also watch for the cross-product of X^*Y to occur. The cross product can be defined anywhere in the range from “the same test generator template hit both X and Y at some point” to “X and Y were hit in the same cycle”.

Projects can use Sleuth event cross-products at multiple levels to prioritize analysis. A cross-product that is never hit at the *template level* (ie, the same template has hit events X and Y, even if they are in different test instances) should be analyzed prior to analyzing why X and Y weren’t hit in the same cycle. This basically tells the Validator where to start looking. Projects can monitor cross-product coverage at the following levels:

- 1) The template level
- 2) The test-instance level (did X and Y both occur within the same test instance)
- 3) Within a time-window (for instance, did X and Y ever occur within 1000 cycles of one another)

Sleuth coverage attempts to have approximately the same or slightly higher effort to select and code coverage events than frequency coverage, but with smaller risk of missing important event interactions.

The downside of Sleuth is that it places a great deal of importance on writing a superior testplan with *very few conditions* (lest the cross-product space explode). This is very tough to do, and very error and risk prone.

Another downside is that coverage analysis becomes more complex as Sleuth does not only create useful event cross-products, but also cross-products that would never happen in real life, or cross-products that are just not important. With Sleuth methodology, coverage analysis assumes picking useful information from a set of data that may contain many uninteresting data.

Even though Sleuth has certain benefits, there are no Intel projects that currently utilize Sleuth.

6.2.2.4 Formal Property Verification cover points

[Formal Property Verification](#) does not use random testing, but rather uses formal tools to run its checks versus every possible scenario. As such, it would seem coverage is not necessary since the formal tools should hit everything. However, it is possible for the Validator to place incorrect

restrictions on the input to the formal model. These restrictions, also known as *assumptions*, are part of the normal setup of FPV proofs. If the Validator makes a mistake in assuming what neighboring logic may send to the logic under test, it is possible that the logic may not even enter basic states.

One example of this is assuming that neighboring logic cannot send back-to-back operations. If this assumption is not true, there may be important logic that will never be stressed by the formal proof (logic such as hazard checks, stalls, and queue full conditions). Thus, FPV utilizes some basic cover points, somewhat similar to *event coverage* (see Figure 17). These basic cover points show that the FPV proofs are progressing to interesting points and that all important parts of the logic are being stressed.

6.2.3 Software / code coverage

There are a number of software coverage methodologies used in the software world, for instance the use of code coverage (has this code executed) and path coverage (what paths have we taken through the code). For the most part, Pre-Si Validation does not use any of these techniques with the notable exception of the firmware Validation teams such as [Microcode Validation](#) and [Embedded FW Validation](#).

Microcode Validation currently employs a form of path coverage that monitors the executed paths of microcode as it goes through possible redirection points such as branches. Similar to functional coverage, path coverage can be monitored in great depth (imagine all possible paths through SW – an infinite range) to very shallow depth such as tracking paths through only a single branch.

For further information on Microcode coverage, see [Microcode Validation section: Coverage](#).

For RTL validation, it would appear that software validation methods would be applicable since the RTL model is a piece of software. However, unlike typical software, ALL lines of RTL code are run during every clock, so code coverage is not effective. For the same reasons, path coverage is not useful. If we validate a primarily *behavioral* RTL model in the future, code coverage may become a viable option.

6.2.4 Input coverage (aka TE input coverage)

Input coverage is a direct measurement of the input into a DUT, without taking into account anything that occurs within the DUT. Input coverage typically includes items such as opcodes and attributes, modes, events from the outside world such as clearing and stalling conditions, as well as temporal conditions such as back-to-back input operations and other such things.

Input coverage typically requires lower investment than functional coverage – especially in terms of defining the set of coverage conditions. Moreover, its feedback tends to relate back to the test environment (TE) and tests much easier than feedback from functional coverage that originates deep inside a DUT. This facilitates much quicker feedback to the TE and tests.

In small DUTs, or DUTs with straightforward and predictable operations (such as an execution unit like an adder) input coverage can give a coverage depth close to that of functional coverage. In large complex DUTs, where operations are executed far from the control of the input,

microarchitectural features such as re-ordering and pipeline hazards cause the input to have very little correlation with what happens within the DUT. To make matters worse, the DUT can have a large quantity of state buildup that takes time and interesting conditions to create. There is little chance input coverage can offer feedback on such conditions. Thus, in these types of DUTs, input coverage is not a substitute for functional coverage.

6.2.5 TE coverage

Many validation languages provide methods to code coverage events right within validation environments. As many validation environments have access to main RTL interfaces and have checkers that simulate (usually at a higher level) the main functionality within the DUT, building coverage on top of the information available to the test environment can give a clear picture on what is happening within the DUT without looking at the low level RTL code and searching for RTL signals that serve validation needs. Since TE coverage works at a higher level of abstraction and is coded based on TE's view of the DUT without looking at the actual RTL implementation, this code becomes less dependent on RTL code changes that otherwise can break legacy coverage code.

Another benefit of using TE coverage is that coverage events are developed based on internal TE's structures and events that should be much better known to the Validator. A Validator can easily add new coverage code while developing the test environment, and get a quick coverage feedback on the high-level states of the logic covered by the tests.

Besides coverage events that are built to monitor the internal logical states of the DUT, i.e. events that are usually placed within the checkers, coverage events can also be built for any other part of the test environment. Coverage conditions could be useful when they are built on top of the sequences to monitor for basic stimulus produced by the test environment to test the design. This can give quick feedback on the quality of the test environment showing if some events and sequences expected by the TE developers are not being generated because of coding bugs, missing sequences or bugs in randomization. Coverage conditions can also be built within TE monitors to observe the activity on the DUT interfaces at the lower abstraction level.

TE coverage has its own limitations as it usually works at the higher abstraction level and do not view the actual state of the RTL. For instance, many conditions like queue full conditions, allocations to a particular entry within a buffer, internal cache hits etc., can be invisible to the test environment that doesn't simulate RTL at that level of detail. Moreover, many coverage events coded using TE coverage cannot be reused at the higher validation levels, especially when transitioning to emulation and post-silicon (see Coverage Reuse section of this chapter).

6.2.6 Toggle coverage

Toggle coverage is simply coverage on a per-node basis on whether a node has done a low-to-high transition and a high-to-low transition. It generally does not tell whether a design, a microarchitecture, or an architecture has been tested. It does, however, reveal whether or not a node is even being touched by a test. Toggle coverage is sometimes used as a first pass indicator for Fault Grading (see [The Post A-Step World section: Fault Grading](#)).

Toggle coverage is also useful as a basic indicator to design what parts of their RTL have seen ‘basic operation’; this has been used traditionally in some IP teams within Intel. SOC and integration teams also use interface toggle coverage to help verify that they have properly put together and exercised an integrated IP. Knowing that you’ve toggled all signals tells you that you have connected the signals *and* that there is at least a chance the test was doing something with them.

6.3 Implementing coverage-based Validation

The following is an overview of the way we physically implement the concept of coverage.

6.3.1 Developing coverage conditions

Validators develop a list of coverage conditions typically as a part of writing a validation plan (or testplan). The timeframe of developing coverage conditions can vary widely. If the team uses coverage that can be coded with low effort, they should target developing coverage in parallel with initial RTL feature coding and initial feature exercise. This is typically during the IP RTL 0.5 timeframe of the project (see [The Life of a Project section: PLC Milestones for IP Validation](#)). This is a natural time to create an initial set of conditions due to the feature being fresh in the Validator’s mind. If the team is using a deeper, more complex coverage, the coverage effort is likely too great for the IP RTL 0.5 timeframe and should be done during IP RTL 0.8 and IP RTL 1.0 when most coverage conditions are to be coded and filled.

Validators must understand the workings of a feature to write good testplan conditions. This includes the feature’s interaction with other features. At an architectural level, it may be how two IA32 features interact; for instance, how a new instruction operates in a certain mode. At a microarchitectural level it can be how a certain pipeline interacts with the structures, FSMs, and other pipelines around it.

When writing testplans, the conditions within the testplan may map directly or indirectly to coverage conditions. In some disciplines, coverage is automatically (or semi-automatically) generated and thus does not require a testplan. The following sections discuss these methods.

6.3.1.1 Direct testplan-to-condition mapping

Most testplan conditions in the Pre-Silicon Validation world are directly mapped from the initial testplan condition directly into the coverage monitor. This means that the coverage monitor exactly reflects the testplan condition. This is the predominant form of coverage in Pre-Silicon Validation simply because it is possible in Pre-Silicon. Since we have full access to the DUT, any possible condition we can contrive can then be coded.

6.3.1.2 Indirect testplan-to-condition mapping

Testplans can be written in a way that makes it impossible to have a direct testplan to coverage mapping. Post-Silicon Validation often utilizes this method. The testplan may include complicated

conditions that the team wants to target, but there is no way to get those exact coverage conditions from silicon. Because of this, the team relies on using whatever coverage is available on silicon, and typically maps many simple silicon conditions back to the complicated testplan condition. The Validator must then do analysis of the final coverage counts and then infer whether the area was hit well enough.

6.3.1.3 Automatic condition creation

In some Validation disciplines coverage conditions may be automatically generated by special tools that parse code or analyze configuration data, and figure out where coverage is needed. An example of automatic creation of coverage conditions is the PSF Automated Validation tool (PSFVal) that is able to generate coverage conditions for any given configuration of Primary System Fabric (PSF). As PSF RTL itself can be automatically generated for different projects using a set of input configuration parameters, the same parameters are used by the PSFVal tool to generate basic PSF validation collateral that includes coverage.

In other cases coverage conditions can be generated semi-automatically using special tool hints provided by Validators. An example of semi-automatic creation of coverage conditions is the Pathfinder coverage tool used by [Microcode Validation](#). This tool, with some microcode annotation done by Validation, automatically creates the coverage space. Thus, the Microcode Validation team does not write testplan conditions.

6.3.2 Writing the coverage monitor

Once the condition has been decided Validation must code the coverage monitor (in any area where there is no automatic coverage creation). Coverage monitor coding has varied over the course of many projects. Much of functional coverage had been coded in the past with a coverage tool called *Visual Proto*, as well as by ad-hoc methods such as parsing simulation log files for event information. These days the main tools to code functional coverage are *System Verilog Coverage* and *QuickCov*, as well as post-processing coverage written using UTDB and Python.

Architectural conditions are coded in *e-archspec*, a coverage tool that is based on the architectural simulator *Archsim*.

6.3.3 Writing and running tests

When writing tests to hit coverage, it is best to use random tests that span the largest feature list possible to catch all possible cross-product conditions. When running the tests Validators should target sets of features with whatever test generator controls are available while allowing all other features to do everything at some “noise” level – which may be a very tiny quantity!!

As the project progresses, Validation will be running tests and using coverage feedback to better direct their tests into unstressed areas. This may also lead the tests into finding more bugs. See [Stimulus section: Effective use of a random stimulus generator](#) for a discussion on techniques to get the most out of your test generator.

6.3.4 Running coverage monitors with tests

Once a coverage monitor is coded, Validation runs it with random test suites used to test the model. When the tests finish, the coverage monitor will output its coverage, typically to some standardized coverage database.

The Validation team may run thousands of tests within a day, possibly running with many coverage monitors. Typically, a Validator does not analyze coverage from an individual test, but instead analyzes the additive-merge of the coverage from all of the tests. The coverage database often keeps track of which test generators are hitting which conditions as well as tons of other information.

During periods of the project when RTL code is changing, Validation attempts to keep coverage data fresh by ‘discarding’ older coverage data. For instance, if a Validator hit condition X fifteen weeks ago but has never hit this condition again since that time, it makes sense to show condition X as a ‘miss’ because it has not been hit recently (meaning that the most recent RTL code has not been stressed by hitting condition X). The simplest way to keep coverage data fresh is to have the database discard coverage data once it has reached some age limit such as four or eight weeks (both have been used historically). This is called a *rolling coverage window*. The age limit of the rolling coverage window can be a function of how much RTL is changing, but you might also consider the quickness with which a team can get through its test running and coverage process. In teams where test runs take multiple days, it makes sense to have a longer age limit to compensate for this.

6.3.5 Analyzing coverage

Once coverage monitors are coded and run, the next step is to analyze the collected coverage. Coverage analysis can range from being very simple to very complex. At its simplest, analysis involves looking into the coverage database and getting a hit/miss indication. This feedback is then used to update a test, test generator, or proof assumptions to correctly target that condition. On the complex side, coverage analysis includes looking over a broad spectrum of coverage data and attempting to prioritize work given the priorities of the conditions as well as the coverage information.

An example of priority ordering is to pre-prioritize the coverage conditions during initial testplan writing, and then go about analyzing and driving un-hit coverage in that same order.

Another example of priority ordering is to target breadth first. This methodology puts emphasis on hitting the simplest conditions in all areas prior to analyzing more complex conditions in any given area. The more complex the DUT, the more likely that bug finding will be done in layers, which is known as *onion peeling* (Section 5.4) In general, it is best for the project to approach testing, and therefore coverage analysis, in a ‘breadth first’ and then ‘in depth’ method such that the onion peeling begins in all areas early. With a depth-first approach, Validation might completely peel two areas of functionality, then move on to a third area when the project is within a few months of tapeout. This could be trouble if there are simple bugs in that third area. Simple bugs close to tapeout is almost always a bad thing. Of course, on a design with only one or two large changes, it makes sense to jump directly into depth coverage in those areas as fast as possible.

6.3.6 Feedback to the testplan

Validation uses feedback throughout the project to bolster the coverage effort. Feedback typically comes in the form of bugs, especially those that escape from one Validation discipline to another. When a bug escape reveals a weakness in stimulus, Validation should analyze whether the coverage set would have also revealed that weakness. If it would not, Validation should consider adding coverage to close that gap as well as to investigate neighboring areas as well.

A word of warning here: when adding coverage to compensate for an escaped bug, be careful not to add coverage in excess of the normal coverage used to validate that area. If you add numerous deep coverage conditions for each bug escape in an area where you are doing breadth coverage, your efforts will become heavily weighted towards those bug escape areas only, possibly at the expense of good validation of the other areas.

6.3.7 Exit review and the high-water mark

When the Validation effort in an area is complete (or near complete), or when it becomes close to tape-in time, the area should schedule an *exit-review*. An exit review is a process where the Validation team presents its progress against the testplan to stakeholders (Designers, Architects, other Validators, and project managers). The stakeholders help prioritize unfinished validation work, as well as what tasks can be pushed until after tapeout or waived completely. Coverage is closely examined as part of the exit-review process since coverage holes are considered unfinished validation work.

The following items should be considered when deciding whether to spend extra time and resources to hit a coverage hole: the likelihood of a bug being in that area; the likelihood of that bug being disastrous in Post-Silicon; and where could we better spend the Validator's effort. Along with these tactical considerations, the long-term aspect needs to be discussed – while it may not be critical to hit this condition in the short term, it may be more important for the long-term goals of the test generator, so it might still make sense to work on it immediately.

Typically around the exit review or shortly thereafter, the area will reach its *high-water mark* in coverage. This coverage data should be saved as a reference for the future. Validators may ask “I wonder if I hit this condition on A-step”, or may wish to compare coverage on future steppings or projects to this initial datapoint.

6.3.8 Coverage during stepping validation

During some phases of the project, such as [Stepping Validation](#), Validation may run coverage in an area that has undergone some level of change. In some cases, Validation may choose to simply check that the newly hit coverage reaches (or gets close to) the previous high-water mark. This is a simple way to check that the basic level of Validation that you are doing on a stepping is equal to the job you initially did.

6.4 Coverage Reuse

These days coverage can be enabled and efficiently used at different levels of validation such as IP Validation, SOC Integration Validation, SOC Emulation Validation and Post-Silicon Validation. Each of these validation levels has its own specifics, and coverage methodology for these levels has some important differences. Current coverage methodology suggests that most coverage conditions should be developed at the IP Validation level, and higher validation levels should in most (if not all) cases only reuse conditions from the lower levels. Reuse is essential to enable efficient coverage collection at the different validation levels, especially at emulation and post-silicon that do not usually have own coverage conditions.

This chapter covers reuse methodology across different validation levels and briefly describes QuickCov methodology that enables coverage reuse from the lower validation levels up to the Post-Silicon Validation level. It also describes specifics of coverage methodology for each of these levels.

6.4.1 QuickCov Methodology

QuickCov is a methodology for defining common coverage collateral across different test environments used during project development - simulation, emulation, FPGA and silicon. Coverage collateral is initially defined and coded by Pre-Silicon Validation teams that own clusters or IP blocks. This collateral is then used to collect coverage data at different levels of validation environments including post-silicon.

The QuickCov methodology enables reuse by:

- Guaranteeing that coverage code can be synthesizable into actual hardware to work on emulation, FPGA and silicon.
- Giving an option to specify for each coverage condition which validation levels are going to use this condition to collect coverage data. This is important because only a subset of conditions can be reused from a lower level at a higher level because higher levels have harder limitations on the total number of coverage conditions they can support and also because higher validation levels do not need the same level of detail that is necessary at the lower levels.

Figure 18 shows simple reuse map across different validation levels and platforms. Details about these reuse levels will be provided in the following sections.

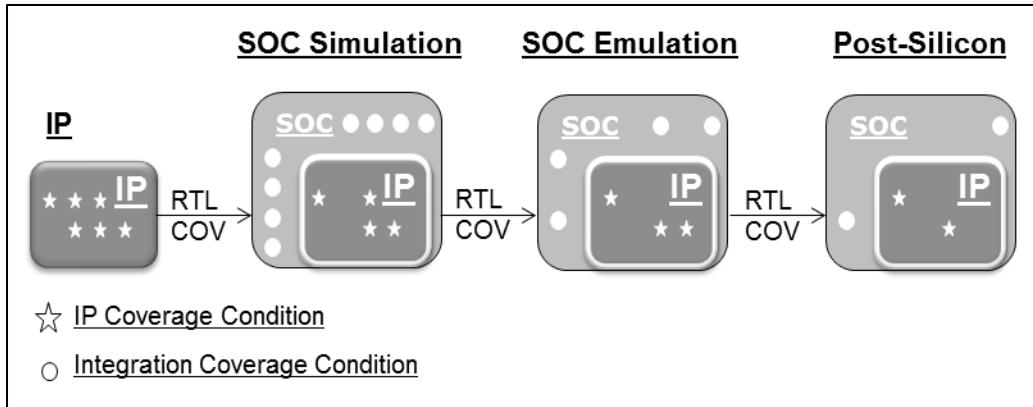


Figure 18. Simple Coverage Reuse Map.

6.4.2 IP Coverage

Coverage development normally starts at the IP Validation level. IP Validation has the most control over the DUT, and Validators at the IP level have most knowledge of the IP's internal functionality. As IP Validator's task is to test all internal logic within the IP, Validators at this level are responsible for developing all breadth and depth coverage conditions that are necessary to evaluate the health of the design and of the test environment in a meaningful way. At this level validation is able to access any RTL signal in the design, and it can code virtually any coverage condition it can think of.

Not all conditions developed at the IP level are to be reused at the higher validation levels. For such conditions that are not expected to be reused, the IP Validation team is flexible to use any tool that can better serve its needs. However, conditions that are to be reused at the higher validation levels have to be coded in compliance with the QuickCov methodology.

When developing coverage conditions at the IP level, it is always important to think if these conditions are useful at the higher validation levels. In addition to that, IP teams can receive feedbacks from other validation teams on the coverage conditions they would like to have at their validation levels, and this feedback should also be incorporated into the coverage code. Coverage conditions that need to be visible at post-silicon have to be coded using existing VISA signals, and if important coverage conditions cannot be developed using the existing VISA signals, then validation team can ask to add these signals to VISA to improve design visibility in silicon.

6.4.3 SOC Simulation Coverage

The SOC Simulation / Integration Validation combines multiple IPs and is responsible for each IP's correct functioning within a subsystem. SOC / Integration Validation is primarily interested in connections between IPs and features that cross multiple IPs, and is not responsible for covering all internal conditions of each IP that is included into the subsystem. Therefore coverage reused at the integration level should be focused on the following types of conditions:

- IP connectivity at its interfaces
- Basic functionality covering interfaces and interactions with other blocks

- Global flows that the IP is involved into
- IP functionality that is of a high risk

Simulation speed at this level is usually much lower than at the IP level which limits the number of coverage conditions that can be hit and analyzed. At this level test generators typically have less control over the IP inputs which also limits the number of concurrent events that can happen within the IP.

It is preferable that most (if not all) coverage conditions that are going to be used at the SOC level are reused from the IP level, but in some cases it becomes necessary to add new conditions that do not exist at the IP level. The methodology for adding new conditions is about the same as for the IP level. The new conditions that are going to be reused at the higher levels have to be coded using the QuickCov methodology, and they have to be tagged properly to indicate at which levels these new conditions are to be used. The conditions that are not going to be reused at the higher levels can be coded using any convenient method.

6.4.4 SOC Emulation Coverage

SOC Emulation Validation has ability to convert coverage conditions coded in accordance with the QuickCov methodology into synthesizable code. During emulation time, coverage data accumulated by the coverage collection and reporting logic is stored in the coverage database and can be analyzed in a similar way as if it was collected on a simulation platform.

Calculation speed is the greatest benefit of SOC Emulation. At this level, validation team can exercise long flows and sequences that are prohibitive in simulation. Tests at this level can be much longer than at the simulation levels, and validation team has ability to direct its focus to volume testing and concurrent stimulus.

Because of the nature of Emulation Validation, coverage that is reused at this level should be targeted to long and complex flows (e.g. reset and power management), most important FSMs, critical queue corner cases (e.g. queue full conditions), global error events and any types of cross-IP and inter-IP conditions that were difficult to hit at the lower levels.

6.4.5 Post-Silicon Coverage

Post-Silicon is where all functionality has to be extensively tested at the highest possible speed. Coverage at this level can be useful to evaluate the quality of post-silicon test collateral and shorten time to PRQ.

Coverage at the Post-Silicon Validation level can be observed using standard observability tools and methods available to view internal signals in silicon (e.g. VISA which stands for Visualization of Internal Signals Architecture). Post-silicon coverage is much more expensive and is more difficult to collect than either simulation or emulation coverage as all coverage events at this level are physically implemented in silicon. Therefore the number of coverage events defined to be used in silicon has to be limited. As mentioned at the IP Coverage section, all coverage events selected to be used by Post-Silicon Validation, have to be connected to the VISA observability network.

Coverage conditions used at the Post-Silicon Validation level are a subset of the coverage conditions used at the SOC emulation. These conditions can include major corner cases, FSM states and some important conditions that are hard to hit at the lower validation levels.

6.4.6 Coverage and PLC

Like other components of the design, coverage has to be developed and filled at proper stages of the project (see [The Life of a Project](#)). There is a list of Product Life Cycle (PLC) requirements related to coverage that have to be fulfilled to satisfy each particular milestone at IP, SOC or Emulation Validation level. These requirements are important to support reuse between different validation levels and ensure a certain level of quality at each milestone.

Figure 19 below shows PLC requirements related to coverage.

Milestone	High Level Cov Goal	IP	SoC	Emulation
Prod	IPDS			
1.0	100% coded 100% filled	100% total cov coded 100% total cov filled (with waivers) Coverage data reviews complete (approve waivers)	100% total cov filled (reusable + SoC)	same as SoC
0.8	90% coded 80% filled	100% reusable events coded and tagged >90% total cov coded >80% total cov filled in volume	100% SoC events coded 100% interconnect/integration cov filled (i.e. PSF/Fabric/Mesh) >50% reusable cov filled	same as SoC
0.5	10% coded 0% filled Cov Plan 100% written	100% reusable events coded for Op5 features (and legacy features) 100% reusable auto-gen interface events 100% cov plans written and reviewed	review and enable re-usable events enable interface events	same as SoC
0.3	SOC cov requests to SIP	no coverage	requests to SIP for reusable coverage. Tied to IP Op5 milestone	
0.0	coverage plan started			

Figure 19. Coverage and PLC.

As can be seen at the picture, there are no coverage deliverables until 0.5 milestone. However, thinking about coverage can start at the earlier stages. For instance, as most coverage conditions are written at the IP level and are just reused at the higher validation levels, SOC VAL 0.3 milestone requires the SOC or subsystem validation team to send requests to the IP teams to add

reusable coverage conditions that satisfy the needs of the SOC validation team. This is necessary for the IP teams to have time to respond to the SOC requests by their 0.5 milestones.

0.5 milestone is where validation testplan has to be written by the IP team and reviewed by its stakeholders that include Validators working at the higher validation levels who would be reusing some of the coverage events. IP has to code all reusable events for validated features that are delivered with this milestone.

For 0.8 milestone, IP teams should code and tag all reusable events, and also code and fill most of all planned coverage events. If SOC team has any coverage conditions that it is not reusing and has to develop, these conditions should be developed by SOC VAL 0.8 milestone so that they can be filled by SOC VAL 1.0. By the same 0.8 milestone, SOC teams should fill all interconnect/integration coverage conditions which would indicate that the IPs were properly integrated.

Finally, 1.0 milestone for both IP and SOC Validation teams imply that all coverage conditions were coded and filled, and if there were any coverage holes detected, all these holes have to be presented and discussed at the exit reviews to conclude if validation team is ready to declare 1.0 milestone (see Exit review and the high-water mark section).

6.5 Coverage goals and important clarifications

Validation has been using the following guideline for deciding if coverage has reached a level that is high enough for tapeout: Coverage is good enough when a Validator can present the missing coverage in an exit review (section 6.3.7) and convince themselves and the stakeholders that it is OK to have not tested those conditions or the areas around those conditions.

There is no set number that indicates “tapeout worthy” coverage. Moreover, if an area taped out with X% coverage on one stepping, it does not mean the X% is appropriate for all steppings due to different risks and requirements. On a given stepping, different functional areas may tape out with different coverage. This is why it is important to have the raw coverage metrics available to all stakeholders so that they can make informed decisions.

In the past, when many projects used depth coverage to evaluate design health, and the number of depth coverage conditions was tremendous, it was almost impossible to hit 100% of coverage. Therefore, such metrics as “We want to hit 85% aggregate coverage” were widely used as a general direction for a project. Nowadays, as most teams, supported by the general coverage methodology, have moved more to the breadth-type coverage, the goal has moved to 100%. As noted above, this doesn’t mean that all coverage conditions have to be necessarily hit, but this means that all coverage holes have to be brought to an exit review and be justified. Some holes can be so serious that they can gate tape-in until they are covered.

There are some clarifications that are important to understand when implementing coverage. The following points are all derived from mistakes we have seen during past projects:

6.5.1 Coverage indicators

There are many different ways to report coverage. The simplest is aggregate coverage – the percentage of the coverage conditions that have been hit, without taking into account any prioritization or weighting. There are many prioritizing and weighting schemes to help make aggregate coverage more meaningful, such as weighting high-priority conditions greater than low-priority conditions, or simply by reporting the low and high priority conditions separately.

Like any indicator, it is important that consumers of the indicator understand the meaning of the data and any caveats. The following sections list some of those caveats.

6.5.1.1 Hitting 100% covered does not mean an area is bug-free

It is possible to hit 100% of the coverage conditions within an area. In that situation stakeholders may assume that the area is fully validated, bug-free, and no longer requires Validation effort. This is not necessarily true.

To begin, coverage is only as good as the coverage specification, ie, the testplan. If the testplan is not complete then it is possible that important conditions are not being stimulated and bugs are being missed.

In addition, even a thorough testplan cannot specify all possible conditions. A bug may exist in some corner case that is not specified in the testplan/coverage and not hit with the stimulus. This is especially true when targeting beyond the enumerated coverage space (see 6.1.3). The coverage data reveals nothing about how well covered the condition space beyond the testplan has been hit. The type of testing used to hit the conditions is important; for instance, directed testing should not be utilized when targeting beyond the enumerated coverage space.

Finally, even with a thorough testplan and comprehensive stimulus, poor checking can still allow bugs to exist when the coverage indicators says “100%”.

6.5.1.2 Coverage Density

An aggregate coverage number does not convey the distribution of the covered conditions within the coverage space. For instance, two areas may have 80% coverage, but it is the area that has more uniform distribution of the 80% that has completed the most validation. Imagine a coverage space that is the set of ALU operations (add, sub, mul, div, xor) crossed with a large set of input data values that creates an overall target of 10,000 conditions. With 80% aggregate coverage, it might seem that rigorous testing of this space has taken place – but what if the 80% coverage comes from hitting 100% of the add, sub, mul, and div conditions, yet not one xor condition has been hit? Not uncovering this testing hole could be catastrophic! Therefore, coverage analysis has to be performed even on areas that show high aggregate coverage.

6.5.2 Prioritizing coverage versus other activities

As mentioned in section 6.1.4, the coverage effort logically follows *after* the stimulus and checking effort of the project, but that some parallelization efforts among the three pillars must take place to minimize the length of the project. This concept is true as well for other activities. For instance,

if there is a serious design-health issue, it generally makes more sense to get failures debugged as fast as possible then to continue pushing for better stimulus through coverage.

Validation must carefully balance its efforts within a project. It should rarely serialize all of its activities, and Validation often works on stimulus, checking, coverage, and debug activities in parallel to minimize the length of the project and to find bugs as soon as possible.

6.5.3 Coverage is not the end result.

Due to the fact that many of the main Validation indicators are based on coverage metrics, both management and individual contributors tend to focus on driving the coverage indicators. Using coverage feedback to fill gaps within test generators is great, but other feedback should not be ignored, such as bug escapes or unhealthy models. Bug escapes are a signal that testplans may need to be updated. Unhealthy models are signs that it might be best to abandon execution activities (such as coverage) for a while and attempt to drive up model health.

Make sure driving coverage does not create bad behavior, such as targeting complex conditions of large size prior to targeting simple conditions. It is vitally important that Validation targets RTL in a breadth-first manner such that all features are first exercised before successive stages of more exhaustive testing.

6.6 Other uses for coverage

6.6.1 Test-suite minimization

Test-suite minimization is a method to find a minimized set of tests that can be used to hit some level of coverage. The method uses coverage information from each test, input from the user, and various heuristics to create the ‘smallest possible’ group of tests to hit the necessary coverage. The input from the user might include:

- A threshold of total coverage to reach
- The minimum number of times a condition should be hit (if possible)
- The maximum number of tests (or cycles)

Test-suite minimization is a great idea from an efficiency standpoint, and it could be considered “complete validation” if the following two things are true:

- The coverage conditions specified are all the conditions needed to test the device (ie, you are not “targeting beyond the enumerated coverage space” (see section 6.1.3)).
- Rerunning the same tests will hit the exact same conditions every time.

These two points could potentially be satisfied in Architecture Validation and Microcode Validation, but even in those areas that have some level of “targeting beyond the enumerated coverage space”. Other disciplines, such as IP Validation – which targets far beyond its coverage space, could never consider a minimized test-suite as complete validation. Instead, test-suite minimization works great for patching testing holes in a regression suite, or attempting to hit a large set of conditions in a turnin-gating regression suite that has size constraints (see [Regressions](#)).

7 Summary

Coverage is one of the three pillars of Validation. Coverage gives Validation feedback on the goodness of its test generators as well as a basic measure of completeness. Almost every Validation discipline utilizes coverage in some form – examples include architectural, functional, and code/path coverage.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 9

The Validation Mindset

By: [Michael Bair](#)

1 Abstract

Effective Validation requires a mindset somewhat different from typical engineering disciplines. Whereas most engineering disciplines have a constructive mindset, often assuming the correctness of what is built, Validation assumes that the design is incorrect in some way and uses a destructive mindset to root out bugs. Other behaviors, traits, and skills differentiate a great Validator from any typical good engineer.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	8/12/2011	Initial Draft	Michael Bair	Matt Kupperman / CCDO Val Staff
1.1	6/27/2016	Update with feedback from Art of Val WG	Michael Bair	

3 Contents

1 Abstract.....	235
2 Chapter Revision History	235
3 Contents.....	236
4 Purpose.....	237
4.1 Why do we need this chapter?.....	237
4.2 What does this chapter cover?	237
4.3 What does this chapter not cover?	237
5 Background Concepts.....	237
6 The Validation Mindset	237
6.1 Aspects of the Validation Mindset.....	237
6.1.1 General Good Engineering	237
6.1.2 Solid Understanding of Validation Fundamentals	241
6.1.3 Focus on what <i>could</i> go wrong	242
6.1.4 Knowing When to Trust.....	243
6.1.5 Going Beyond the Specification.....	244
6.1.6 Comfort with Ambiguity	244
6.1.7 Supporting Design and Architecture	245
6.1.8 Creating and Maintaining a Positive Atmosphere	245
6.1.9 Disciplined Follow-up of Issues and a Focus on Problem Prevention....	245
6.2 Cultural Evolution and the Validation Mindset.....	246
7 Summary.....	246
8 Future Work	247
9 References.....	247

4 Purpose

4.1 Why do we need this chapter?

Validators from many different disciplines have reported that a certain way of thinking, or mindset, is critical to quality validation. This chapter presents the mindset as a set of non-technical fundamentals that produce outstanding Validation engineers.

4.2 What does this chapter cover?

This chapter covers some basic engineering practices that are important to Validation, as well as Validation specific practices and ways of thinking.

4.3 What does this chapter not cover?

This chapter does not cover general management practices or management of a Validation team. That is covered in [Leading a Validation Team](#).

5 Background Concepts

6 The Validation Mindset

The Validation Mindset is a collection of behaviors, actions, and ways of thinking that produce highly effective validators. The Validation Mindset can be thought of as the ideal state for Validator behavior.

6.1 Aspects of the Validation Mindset

The following sections describe various aspects of the Validation Mindset.

6.1.1 General Good Engineering

Some of the general behaviors and skills that make up a good engineer are particularly useful in the field of validation.

6.1.1.1 Conscientiousness

As a Validator, you must care. You are often the last engineer a product sees prior to getting to the customer (unless it is another round of Validation). Moreover, it is not easy for management or outsiders to measure the difference between a Validation job done well versus one done poorly...

or even not done at all. Because of this, a less-conscientious Validator might attempt to do the minimum required to remain off management's radar. Conversely, a conscientious Validator understands that though the odds are low, there is always a chance that an escape could seriously hamper the Post-Si effort and jeopardize the product. This causes good Validators to follow-through, to understand why every failure occurs, to understand exactly why something failed. undebugged failures bother good Validators, worry them, and occasionally keep them up at night.

Conscientious Validators take ownership for their area, work to become an expert and to make sure the job is completed. Your ownership often goes outside of defined boundaries: a conscientious Validator will see problems outside of their own area of responsibility and drive resolution of the problem themselves or make sure that an owner is assigned.

6.1.1.2 Integrity

Validators must be heard, even in uncomfortable situations. You cannot be afraid to speak truth to power, ie, tell those in charge something they might not want to hear. Whether it's "this thing you designed doesn't work" or "our chip is not ready to go", you have the responsibility to raise issues when they are uncovered and to make sure the issues are addressed. You take informed risks, but you also speak up when a risk falls outside of the project's comfort zone.

Another aspect of integrity that is important for Validation is *transparency*. Validation status, indicators, and milestones need to truly convey what Validation has done and not done and what risks are being taken. For instance, a team that is reporting a high test pass-rate would also do well to report if any tests are not yet being run (ie, creating an artificially high pass-rate).

6.1.1.3 Fast Ramp

Validators must ramp at a tremendous rate. The projects on which you work are changing rapidly, not to mention the tools, methods, languages, etc. Moreover, you will likely move from feature to feature, area to area, and discipline to discipline within Validation, and be expected to contribute quickly. The most effective Validators learn quickly on many vectors, and begin contributing even prior to having the complete picture.

6.1.1.4 Strong Problem Solving Skills

The problem space facing Validation is enormous. Modern CPUs and SOCs are extremely complex; as such, debug and other day-to-day engineering work is also extremely complex. In addition, the complexity of the *management* of the project is also high; coordinating the efforts of hundreds of engineers with thousands of tasks is daunting, especially when faced with a fresh set of problems on each subsequent project.

In debug, the space of potential problems is enormous. You must efficiently reduce it to a tractable problem. When debugging a difficult failure, try to eliminate variables (or look for commonalities). You must be tenacious and drive to get to the bottom of things, no matter which path they may end up on, and have the flexibility to change course. You must be adaptable; you may need to adopt others' tools or develop your own to achieve your end goals. Part of this mindset and makeup includes confidence: If a validator is not a very confident person, they may well lack what

it takes to follow a path, make (and admit) mistakes, and re-direct down a different path. The ability to make a decision could be used as one partial measure of confidence. This is not to say the mindset of a Validator is cocky, arrogant, mindlessly deciding things or any other generally unpleasant characteristic.

In all aspects of validation, from debug to project management, Validators must beware assumptions. Assumptions will get you into trouble - not always, but most of the time. You must be curious and always question (see 6.1.4 for a discussion on ‘trust’). Many bugs escape because Validators assume something about a feature or protocol. Many problems surface late in the project because everyone assumes somebody else owns that area. If you need to make assumptions for the sake of efficiency, find a way to have them verified. For instance, on an interface, the consumer’s assumptions become the producer’s assertion checks. A good interface monitor between units can cover both. In formal verification, assumptions must eventually be proved. If they cannot be proved due to complexity or funding issues, the assumptions should be run as a checker with all dynamic simulations.

6.1.1.5 Teamwork and Communication

The number of engineers responsible for a complete product is typically enormous and spread out over the entire planet. The number of these engineers that you work with may start small but will grow as your area of ownership, your expertise, and your time within the team grows. You are responsible for creating a positive working relationship with everyone that you interact with along the way.

To begin with, you must both give and accept help. Nearly all Validation jobs are subject to a continuous stream of help requests. Often your time and expertise is needed to solve a problem that someone else encounters. Debug aid is the most common request made of Validation, but your knowledge of tools, test environments, microarchitecture, software, and even your ability to proofread someone’s paper can be invaluable to someone in need. This will hamper your ability to complete your own deliverables, so you must strike a balance. To begin with, err on the side of enabling others, and then bring in management help as soon as you notice yourself falling a few days behind.

Teamwork also requires giving unrequested help. When you find a more effective way to do the job, it is important to share that with others. Similarly, you need to share hard-learned experiences with others; for instance, tools or methodologies that did not work out and the insight gained from the experience. If you discover an undocumented feature that requires Validation effort, alert other Validation teams so that they do not overlook the feature.

You must also request help when you need it, as well as accept unrequested help. First and foremost, you need others’ technical knowledge to do your job – to debug a failure, write a testplan, or learn a new tool. You must also request help or advice in areas where you don’t *necessarily* need the help, but getting another’s feedback will help further productize your innovation by taking in multiple points of view. You never want to innovate or make decisions in a vacuum. Lastly, the simple act of asking for or accepting help creates a sense of community within the team; it builds relationships and demonstrates that you can rely on each other. If you do this during “normal” periods of the project, it will pay back dividends during the tough periods. This applies to networking outside of your project as well. To become good at accepting help you must keep an

open mind. Initially, other teams' suggestions about tools or methodology might seem of low value because they are different from what you know. If you take the time to understand them you might find something worth adopting and in doing so solidify a working relationship across project teams.

Communication is the cornerstone of solid teamwork, and a necessity in the technical career path. As mentioned previously, you must communicate to give and receive help. You must also communicate status, including sometimes bad news (See also section 6.1.8). As you climb the technical career path, your ability to communicate well will directly affect your ability to influence other teams, to productize your innovations, and to spread your message.

An entire chapter could be devoted to communication, but here are some fundamentals:

- Organize your thoughts prior to speaking
- Make your best judgment of audience's needs regarding level of detail
- Give context to your question or idea... yet be succinct
- Learn to judge your audience's understanding to know when you've lost them
- Keep it interactive
- Ask for feedback

Good communication requires listening to an entire idea without preemptively coloring it with your experience. The world of CPU and SOC engineering is changing dramatically. What was a truth five years ago no longer holds true. Be careful not to color others' ideas with your history prior to hearing the entire idea. While listening to someone, you may already be thinking, "We tried this before and it didn't work" prior to hearing the entire idea. It is usually better to hear someone out entirely instead of dismissing an idea early, even if only within your head. Remember, times have changed, and what did not work in the past might work today. As an example, the idea of moving the majority of fullchip testing to emulation would have been laughable in the past due to a huge number of issues. Today, it is the POR for upcoming products.

Finally, regarding communication, you must work to establish an environment of active, mutual support from day one. This means that you are giving the benefit of the doubt to anybody who comes to you with an idea, and give them the opportunity to be heard without fear of being rebuffed or ridiculed. You will be working with many of the same people for a long time; you never want to create an environment where a co-worker will not request help or share an idea with you because they dislike interacting with you.

Teamwork and communication build trust and relationships; this trust and your relationships will help to successfully complete projects. This is true of those who sit closest to you (other Validators, Designers, Architects, etc), but is *especially* true for those who are farther away: IP suppliers, parallel SOC teams, etc. Something to consider: the farther someone is from you (in distance, or when measured across organizational spans), the *more* effort you should put into teamwork, communication, and general relationship building. It is good for your projects, good for Intel, and it is good for your career long-term.

In an SOC world, communication and relationship building are more important than ever!

6.1.1.6 Flexibility and Responsiveness

Good Validators understand the project schedule, status, and priorities. You need to be flexible: when a high priority request comes in, you need to reorder your work quickly to handle that request. As bugs escape occur, you will need to analyze them and possibly update your plans to target them. The same applies to feedback from other Validators, Architects, Designers, and Post-Si; you should be actively seeking feedback and adjusting your plans and priorities accordingly.

6.1.1.7 Efficiency

Validation is never ‘done’ (see 6.1.2). However, Validation does have to provide its services without employing an endless array of Validators, and it does have to pronounce the design as ‘good enough’ at some point. With this dichotomy of ‘never being done’ and ‘needing to get done’, it is imperative that Validation has a good grasp on the priority of all of its work and is efficient as possible.

Validators must constantly remind themselves to re-evaluate their priorities and search for efficiencies. You need to be in a mindset where you provide the maximum possible confidence of correctness for the minimum possible effort. What is the minimum you can do so that customers have most reasonable doubts and concerns satisfied? What requests from Design and Architecture are reasonable to undertake? There are always more cases to test, but at what point have you done enough?

Bear in mind that Validation sometimes funds tasks that are inefficient in the short term (for instance, the life of the current project) for long-term gains (cross-project inheritance, etc). Validation is ardent on minimizing long-term maintenance cost, as it is usually far higher than initial coding cost. You will find that many quality-related activities, such as coding high-quality Test Environments and Unit Testing of Test Environments, pay off dramatically in the future. Validation might also choose to be less effort-efficient on certain activities in order to hasten progress on the project critical path.

You must also seek efficiency gains by process standardization or automation. If something is worth doing more than once, it is probably worth automating. If something is complex in nature but tends to follow specific steps, write a process for it and have everyone follow it. When innovating a better way to do some task, make sure to follow the entire process to gain the most efficiency from your idea:

Innovate → Productize → Standardize

An innovation by itself rarely brings about efficiency, except maybe for the innovator. When it is productized, then multiple users can start seeing the efficiency benefits from the innovation. The biggest gain though comes from standardization, which results in everyone using the innovation, and then everyone making refinements to the tool or method such that everyone sees increased benefit.

6.1.2 Solid Understanding of Validation Fundamentals

The Validation Mindset begins with understanding some fundamentals of Validation. The following list is a good starting point.

- Validation is an independent auditor of the design

Validation needs to be independent from the design team due to their deliverables and goals being very different. Design wants to finish RTL quickly and move on to schematics and layout; Validation wants to spend a lot of time on the RTL to thoroughly flush out the bugs. See [Leading a Validation Team section: Developing a consistent Validation Mindset](#) and [Dynamic Microarchitecture Validation section: As Design goes, so goes uAV](#) for more details.

- Validation is never ‘done’

Engineers never know the exact number of bugs that are coded into a system, nor do they know when all bugs have been found. As such, the task of Validation is never entirely complete – there are always more boundary cases to look for, there is always more coverage to gain and checkers to write, and there are always more security holes to look for. Instead of attempting to find every bug, Validation decides what critical tasks need to be done prior to tapeout, and what other important tasks should be funded as part of the Validation effort.

- Bugs are social creatures

Bugs are social creatures; when you find one it is likely that the same area will contain more. Many of the reasons for a bug to exist in the first place (complex code, unstable uarch, careless designer) are likely to cause more than one bug in the same area. When a bug is found, it is useful to tune test generators to target that area for a short while to see if more bugs can be flushed from the design. Once you have found a series of similar bugs, or related bugs in a unit, call a design review.

- Bugs often follow historical trends

Areas that are typically buggy on previous projects are likely to be buggy again on future projects, assuming that there is any change in that area. On previous projects, the bugs were introduced for certain reasons (complex code, code that is very timing sensitive, code with lots of band-aids already, limited control for Validation to stress the area, etc). Because that same reasons exist on the next project, it is likely that bugs will be introduced again. The opposite is true also: areas that were particularly clean in previous projects are more likely to remain clean on future projects.

- Validation must do more than just find bugs

Validation must be a partner with Design and Architecture in all manners of dealing with bugs: Prevent, Detect, and Survive. Validation can help prevent bugs with early uarch and MAS reviews as well as code reviews prior to turnin. Validation detects bugs as its everyday job. Validation must use its knowledge of the uarch, the weak points in the design, and the usage models in Post-Si, and help create defeatures and other workaround capabilities that might be useful in Silicon to survive bugs.

6.1.3 Focus on what *could* go wrong

Validation has a destructive mindset (see [Introduction to Pre-Silicon Validation section: Destructive Engineering](#)). You must feel that the design is broken in some way, and that you must find the way to expose that break. That means that when all tests are passing, it simply means you are not testing the areas with the bugs. You should always be thinking about what can go wrong, and you need to understand how the design should work so you can recognize when it does not.

As you focus on what could go wrong, ask yourself “how could it be wrong and I wouldn’t already know about it? How bad would it be if I missed a bug in that area? If there is a bug in this unit, how will it manifest itself? What do I need to be checking? Do I have the right test content and checkers to expose that?” You need to remain creative and restless, after all, Validation has no strict guidelines or menus, and your completion criteria is somewhat artificial given the fundamental theme *Validation is never done*. As you evaluate areas to tackle next, usually the scariest area that is most obscure in your mind is a good place to start. If it frightens you because you do not understand it or it is large and complex, *Bingo!*, you have found a worthy target.

During Tech Readiness and the initial execution phases leading up to the 0.8 milestone (see [The Life of a Project](#)), you will spend much of your time helping the development process along. By the time you reach the late execution phases (for instance, after the 0.8 milestone) you focus most of your time on creating failures. It is always a good day when you see that your overnight regression hit a particular failure exactly once (especially if it has never been hit before). Nothing is better to a validator than that! That means it was hard to hit (after all, it escaped all the way to later in the project) and is probably interesting. It feels rewarding to debug through an interesting failure and help the project by finding a compelling bug. That mindset drives all the other things you do – achieving coverage, improving checking, and improving testing. This is an all-out bug-hunting mindset that is bounded by that moment in time when the failures wane, your coverage goals are reached, and other areas of the design are in desperate need of help.

6.1.4 Knowing When to Trust

Engineers utilize many sources of information. These resources include co-workers and documentation. To begin with, engineers typically trust information provided to them. Over time, engineers start getting a feel for which resources typically lead them down correct paths, and which lead them down fool’s errands. They begin forming an opinion of what level of trust to grant each resource.

Validators go through the same process of forming trusting relationships, but they start from a less-trusting position, and always tend to trust their resources less than other engineers might. This position causes Validators to get *second opinions* when confronted with information that the Validator thinks might be suspect. For example, if the designer on one side of an interface describes some portion of the interface protocol, a good way to double check that information is to run it through a designer from the other side of the interface. If the other designer does not agree with your description, then there is certainly a problem somewhere: either the original designer described the protocol wrong, or the second designer has it wrong, or you did not convey the description well from the first designer to the second.

Another example: a microarchitect may describe how a certain feature works within a project and what the boundary cases may be. To double check this person, and also to get some historical perspective on the feature and what may be problematic for Validation, you can seek out the previous microarchitect owner of the feature (perhaps from a previous project) and get her opinion.

You eventually get enough experience to tell when a piece of information has a high likelihood of being correct; you can judge when you are that point by how often you predict correctly. Until you reach that point, it is best to retain a healthy level of mistrust for any new information brought before you.

See also the [The Post A-Step World section: Serious Trust Issues \(and why you should embrace them\)](#).

6.1.5 Going Beyond the Specification

Validation reaches well outside the expected use-cases of the design to find bugs. Rarely does the business side of the design consider anything but intended usage-models of the design and its features. Specifications mostly deal with the intended usage-model. You need to know (and test) these intended usages, but you also need to test the unintended and unexpected usages as well. You need to ask “what is not specified?”. Boundary conditions often occur when multiple features collide; the design can find itself in an unexpected state when software does not follow the expected usage. These problems require you to consider the design far beyond the original intentions of its features.

Being curious plays a role here as well: many bugs are found just by running small “What if I did X” experiments that are not part of (and should not be part of) any test plan. You should be curious enough to want to know why something does or does not work. You should want to know how and why things are the way they are, and devious in nature to see how you can break those things. You have to use judgment to decide when it is worth your time for these experiments, but it is important to have a need to delve into the areas you do not yet know.

Some boundary cases are well understood; all Validators should know to watch out for corner cases like queues filling, counters depleting, and clocks powering off. You should also be well trained to look for the interactions between features, such as power management interacting with locks and interrupts. After historically-understood cases like this, it will be up to you to figure out where to look next. Be paranoid and pay attention to detail. Don’t just do exactly what you are assigned and think that you are done – make sure you look at the space ‘beyond’ your assignment.

In the world of CPUs and SOCs, not all levels of validation will do this equally. Doing Integration Validation on an interface that is standardized and hardened is far less likely to require validation ‘outside the spec’ than standard IP-level validation.

6.1.6 Comfort with Ambiguity

Validators never work with all of the information they need, let alone nice-to-have information that would make their job easier. You will need to become masterful at dealing with ambiguity, as well as knowing your limits. You will start working before you will have clear definitions of what you are working on. You will have to make progress on validating a unit or feature that you do not yet understand. You will have to make decisions with insufficient data. Some of the decisions will take place daily in your workflow: during debug, you constantly make decisions on what paths to follow and which to ignore. Some decisions are higher-level and outside the workflow: have you ‘done enough’ in this area, and is it time to move on to other pressing problems. You may not have the data necessary to make a decision; finding low-cost ways to quickly generate pertinent data is very valuable.

Validation often runs into ambiguity in the form of competing customer requirements. You might be the first person on the project that realizes that the requirements do not mesh well together and

that a well-defined specification of the interaction has not been created. It is an important skill for validators to be able to internalize many competing user requirements so they can test the most valuable things.

A good Validator needs to understand all these needs, see (and expose) any opposing requirements and inherent ambiguity, and have a basic ability to prioritize those needs. Moreover, if the lack of information is keeping Design or Validation from doing a quality job, it is important to raise that issue.

6.1.7 Supporting Design and Architecture

As mentioned in section 6.1.3, during the later phases of the project Validation is mostly focused on creating failures and finding bugs. Prior to that, during the initial stages of RTL development, you must take on more of an interpersonal, almost managerial mindset. During RTL development, be willing to debug failures in side-models to enable earlier feature completion at the expense of more advanced validation if appropriate. Consider how you can give design what they need very quickly (not necessarily what they want). Think about how you can ensure that Design is following the right methodologies so that you avoid issues such as code being turned in without being exercised.

Providing this type of support to other teams can boost a project. Moreover, this kind of positive interaction leads right into the next section: *Creating and Maintaining a Positive Atmosphere*.

6.1.8 Creating and Maintaining a Positive Atmosphere

Validators must strive to create a positive atmosphere. This is true of any discipline that requires interaction amongst multiple people, especially people with differing day-to-day goals. This is especially true for Validation in that Validation is often the bearer of bad news, such as telling a designer of a bug or an architect of a specification hole. This type of news is personal in that it often points out someone's mistake. Not every recipient handles this well, even when the messenger delivers it delicately. Worse yet, if you carry an attitude that bugs reflect negatively on the capability of designers, it is likely to cause a negative relationship that will harmfully impact all future interaction.

This lesson applies to project flags as well: when raising project level issues such as schedule problems or major technical issues, strive to do so in the most proactive and positive manner that is practical. Moreover, in most discussion forums and emails it does little good to discuss why the problem has arrived. The following simple example attempts to avoid the negative and put focus on the solution: *"The FOO critical path is slipping week-for-week. We are pulling together a team to assess our options and find ways to quickly help"*.

6.1.9 Disciplined Follow-up of Issues and a Focus on Problem Prevention

Validators must go beyond finding and filing bugs. Once an issue is discovered, you should also consider why the issue got created in the first place, why it was found as late as it was found, and

whether other similar issues might also exist. You must follow-up to make sure the bug is fixed correctly and that if the bug reappears that it will be caught appropriately quickly.

Furthermore, especially as you become a senior contributor, you need to grow beyond simply finding bugs to learning other ways to increase the quality of the design or the efficiency of the entire team. Preventing bugs and buggy design are important pieces of the Validation Mindset. Validators must be on the lookout for what ‘bad design’ looks like:

- High latency: pipelines that are (or may) be important for performance that simply take too many pipestages to get basic work done
- Coupling: when disparate pieces of logic are dependent on one another, it creates complexity in the system, and likely bugs due to the different parts of the system not understanding each-others’ boundary cases
- Duplicated logic: any time logic is duplicated it can lead to bugs due to a logic change being done in one of the duplicates but not done in the other. Duplicated logic should be replaced by a single piece of code that can be multiply-instantiated
- Interactions without feedback loops: interfaces that assume everything about the other side (such as resources available) tend to be less flexible but also tend to be buggier

These are just examples. As Validators gain experience they will gain skill in recognizing micro-architecture and RTL code that lends itself to bugs.

6.2 Cultural Evolution and the Validation Mindset

Corporate culture and directives change over time and have significant effect on Validation. For instance, the corporation (or an individual project) may change its risk tolerance over time. Risk taking allows you to have greater return with the same investment, with an increased likelihood of failure. With greater risk tolerance, it may make sense to decrease validation effort in some areas to fund other activities (perhaps add extra features to the project).

Similarly, a project or family of projects may take on an execution directive due to external reasons that allow it to execute in a very different fashion than ‘normal’. For instance, an initial product in a family may take on a ‘done is better than perfect’ directive to get the initial product to a basic level of goodness so that it can be proliferated to other projects earlier (possibly at significant cost to the follow-on projects).

In these situations, it is imperative to continue striving for a strong Validation Mindset in the team, and not let the situation dictate a weaker mindset. To do this, you must keep the aspects of the Validation Mindset fresh in the minds of the Validation team (via training, discussions, staff meetings, awards) and explicitly call out where you are taking backoffs from what might be considered ‘full’ validation.

7 Summary

Some basic behaviors of good engineers are particularly important for Validators. Beyond those, there are a number of behaviors, traits, and skills that are specific to the Validation Mindset that produce proficient and efficient Validators, including:

- Solid Understanding of Validation Fundamentals
- Focus on what could go wrong
- Knowing When to Trust
- Going Beyond the Specification
- Comfort with Ambiguity
- Supporting Design and Architecture
- Creating and Maintaining a Positive Atmosphere
- Disciplined Follow-up of Issues and a Focus on Problem Prevention

These behaviors, traits, and skills turn good engineers into great Validators.

8 Future Work

9 References

The Art of Pre-Si Val: Chapter 10

Validation Platforms

By: [Doug Hergatt](#)

1 Abstract

In this chapter, we will explore the spectrum of modern validation platforms including: Virtual Platform, Simulation, Emulation, FPGA Prototyping as well as hybrid platforms. It becomes a vital decision for every project (large or small) to understand the tradeoffs and limitations of each platform type and select the optimal platform(s) to support project validation. Model build effort, model performance, visibility/debug capabilities, model size/complexity, model costs & model accuracy will dictate which platform(s) to use. In an ideal world, a validation continuum would allow a user to move between various validation platforms seamlessly.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
0.5	12/18/2019	Rough Draft	Nathan Graybeal	Michael Bair
1.0	12/14/2022	Full Release	Doug Hergatt	Michael Bair, Trevor Wieman, Sunitha Prithivi & Emeka Ojeh

3 Contents

1 Abstract.....	249
2 Chapter Revision History	249
3 Contents.....	250
4 Purpose.....	252
4.1 Why do we need this chapter?.....	252
4.2 What does this chapter cover?	252
4.3 What does this chapter not cover?	252
5 Background Concepts.....	252
5.1 Levels of Abstraction	252
5.2 Cycle vs Event Based vs Transactional Processing	254
5.3 Overview of FPGAs	254
5.4 Interfacing to Real Devices	256
5.5 EFFM: Emulation & FPGA Friendly Methodology.....	257
6 Validation Platforms	258
6.1 Types of Pre-Silicon Validation Platforms.....	258
6.1.1 Virtual Platform	259
6.1.2 Simulation.....	259
6.1.3 Emulation.....	260
6.1.4 FPGA Prototype.....	261
6.2 Comparisons and Tradeoffs between Validation Platforms	261
6.2.1 Model Performance	262
6.2.2 Model Build Effort	263
6.2.3 Visibility & Debug Capabilities	265
6.2.4 Model Capacity & Scalability.....	267
6.2.5 Cost per Platform.....	268
6.2.6 Model Accuracy	270
6.3 Hybrid Platforms	271
6.3.1 Hybrid Transactors	272
6.3.2 Examples of Hybrid Models	272
6.4 Ideal Validation Platform Use Cases	273
6.5 Advanced Topics	273

6.5.1	Fast Emulation.....	273
6.5.2	ICE Platform	274
6.5.3	Emulation Technology Under the Hood	275
7	Summary.....	276
8	Future Work.....	276
9	References.....	277

4 Purpose

4.1 Why do we need this chapter?

A detailed discussion of validation platforms is warranted as they are fundamental in establishing “how” project validation will be executed. Though these platforms serve common validation goals, the strengths and limitations between the platforms can be quite stark. In this chapter, we will be focusing on the key differences between each platform type. Selecting the ideal platform(s) to support validation tasks is not always clear and therefore it can become a balancing act to ensure the best return on model performance, cost, model size/complexity, debug visibility, as well as modeling effort. In general, it is likely that a combination of platforms will be utilized on a project to provide coverage of all validation use-cases. Having a clear picture of the validation platform options empowers a validator to select optimal solutions.

4.2 What does this chapter cover?

This chapter will give an overview of the typical pre-silicon validation platforms: Virtual Platform, Simulation, Emulation, FPGA Prototyping, and Hybrid model solutions along with their capabilities and limitations. There will be some discussion on use cases and limited discussion on specific tools and vendors.

4.3 What does this chapter not cover?

This chapter does not give instructions on how to setup or run a validation platform for a project. This chapter will avoid giving deep information on vendor tools used for each validation platform.

There are also executable simulators, such as performance models and architectural models (Keiko, ISIM, Archsim, ZSIM, etc) that we do not cover in this chapter, but may consider in future revisions.

5 Background Concepts

This section will provide an overview of some key background concepts that will be utilized in the discussions of validation platforms.

5.1 Levels of Abstraction

The following design abstraction pyramid shown in Figure 20 describes various levels of design abstraction from Behavioral, RTL, Gate to Transistor levels of detail. Design abstraction levels can be leveraged to improve validation platform model performance (a higher *abstraction* level typically translates into higher model *performance*). This usually involves trading off model detail (*lower fidelity*) as you move up the levels of abstraction. For instance, not being able to model design clocking logic (means lower level of model detail or fidelity) in a behavioral level model (highest level of abstraction). Design abstraction levels can also be utilized to enable early project

validation and FW/SW developers since higher abstraction level models tend to be available sooner.

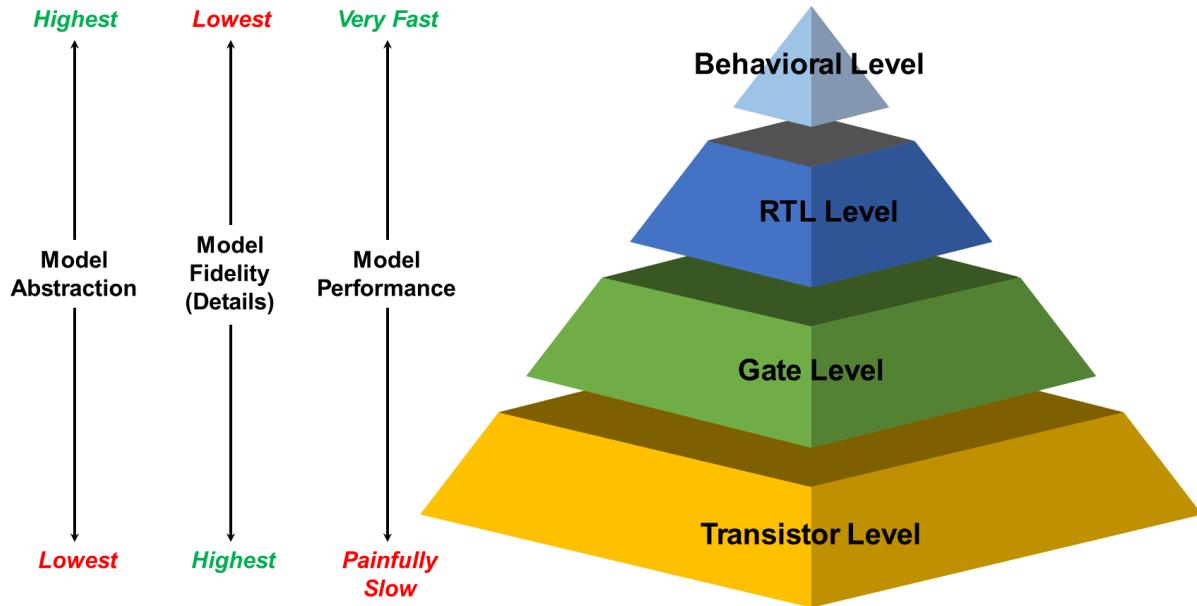


Figure 20: Design Abstraction Levels

Here is an overview of each of the design abstraction levels:

- **Behavioral Level** is typically the highest level of abstraction with the lowest level of model detail or fidelity. These models are transaction-based and are useful for algorithm, data flow and some state machine validation of the design architecture. Behavioral models do not support modeling of the real logic design and are not clock accurate. These models are largely coded using software techniques with languages such as C++ or SystemC (system-level modeling language). Behavioral modeling can also be utilized for testbench coding and system-level modeling. Some analog behavior can also be modeled at this level.
- **RTL Level** refers to register transfer level which is utilized to define the digital portions of a design. RTL designs are typically coded using hardware description languages (HDL) such as Verilog, SystemVerilog or VHDL. This often serves as the golden model in both the design and verification phases of a project. The RTL models the synchronous (clocked) digital circuit in terms of the flow of signals (data) between hardware registers and the logical operations performed on those signals. Care must be taken by project design teams to ensure RTL coding constructs align with coding styles for synthesis and compilation tools using emulation/FPGA friendly methodology (EFFM) guidelines. *For more details, refer to Section 5.5 below.*
- **Gate Level** modeling refers to the design code that has been synthesized into a process technology using primitive gates such NAND, NOR, flops and latches. These models are timing accurate (clocked). Gate models are typically where the bulk of the high-fidelity validation models will be executed. The design at this stage will be in some netlist format in an HDL like structural Verilog.

- **Transistor Level** modeling includes the very highest level of detail and would include SPICE models of key hard-IP's, PHYs (Physical Layer-usually implemented as an analog integrated circuit), etc. Transistor level models will never be run on large designs. This may include mixed-signal simulations as well which will not be covered in this chapter. These models will run *painfully slow*.
- **Mixed abstractions:** we often mix abstraction levels in a validation platform model. For instance, it is possible to have Behavioral-level bus functional code stimulating an RTL-level DUT in a simulation model. Note: there are other permutations of mixed abstraction models that are supported like Behavioral-level with Gate-level, etc.

5.2 Cycle vs Event Based vs Transactional Processing

Since hardware systems are concurrent by nature, it can be very challenging to model a system using software as it operates in sequential fashion. Additionally, the concept of time is also foreign to software systems which processes events as quickly as they are executed. To overcome these differences, cycle and event-based processing techniques were developed. Both techniques have precedence ordering rules (for instance arithmetic operators and relational operators) so that the system behaves in a deterministic fashion and can properly resolve chains of dependencies.

Cycle-based processing refers to the idea that all events are synchronous and occur on the clock cycle. There is no concept of time within this cycle and elements within the design evaluate once per cycle. Cycle-based models typically have higher performance as they are very optimized and do not require large amount of memory to hold timing data. Since there is no concept of timing, this technique cannot detect timing issues such as glitches or edge-related design/implementation issues. Cycle-based simulation is currently not heavily utilized within Intel.

Event-based systems define all actions as an individual event and processes them serially. Basically, if the input stimulus changes that event is queued up, processed, and repeated until the entire system reaches a stable state before moving on to the next cycle. This exhaustive approach provides ultimate granularity in timing (glitches are visible) and low-level functionality but comes at a great cost to performance.

Transactional and System-level simulator techniques are utilized to support virtual platform models. These behavioral software-based models can be used to run unchanged production binaries of the target hardware. These simulators contain both instruction set simulators (ISS) as well as hardware models and can be used to verify large systems.

5.3 Overview of FPGAs

A field-programmable gate array (FPGA) is a special integrated circuit which is designed to be configured by a user after manufacturing-hence *field-programmable*. FPGAs are utilized in very broad-based application spaces including: Telecommunication/Wireless Communications, Embedded Industrial Control, Military, Image Processing, Datacenter/HPC, Broadcast Cable, Automotive as well as Emulation/Prototyping. There are currently 2 primary competitors in the

FPGA market: Xilinx (now part of AMD) and Altera (now an Intel subsidiary) with FPGAs estimated to be an \$8B market in 2022.

An FPGA consists of an array of programmable logic blocks including Logic Modules, DSPs, Memory Blocks, Programmable Routing Switches (fabric) and banks of Reconfigurable I/Os. Refer to the FPGA high-level block diagram shown in Figure 21 below:

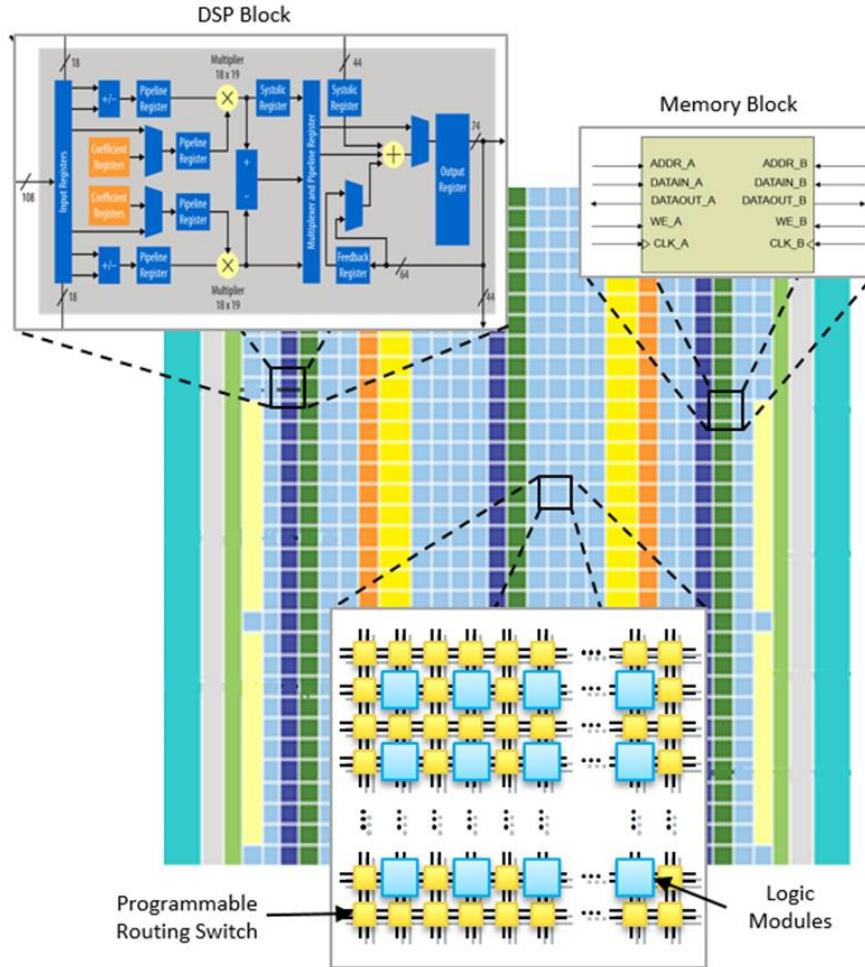


Figure 21: FPGA High-Level Block Diagram

A block diagram of the Intel FPGA (Altera) Stratix 10 Adaptive Logic Module (ALM) is shown below in Figure 22. Logic modules can be configured to perform complex combinational functions using the 8-input Look Up Table (LUT) or act as simple logic gates like NAND, NOR, flops or latches. These 8-input LUTs can also be mapped into two 4-input LUTs as well as many other combinations of X-input LUTs. FPGAs also have a limited number of dedicated clock resources which must be *carefully* utilized when supporting emulation/prototyping models.

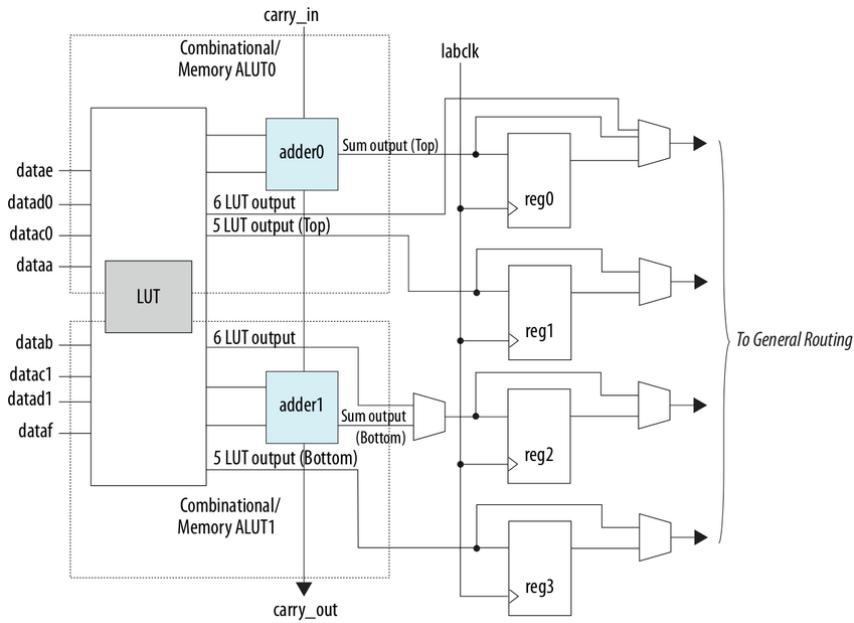


Figure 22: Intel FPGA Stratix 10 Adaptive Logic Module (ALM)

Commercial FPGAs are currently utilized in one Emulation vendor (Synopsys) and play a key role due to the reprogrammable nature of the device as well as the cadence of new technology availability in the market. Custom ASICs (Cadence) and custom FPGAs (Siemens) are utilized by the other two Emulation vendors. All Prototyping vendors (Synopsys, Cadence & Siemens) utilize commercial FPGAs for their solutions. Some of these technology details will be discussed in later sections of this document.

Both Altera and Xilinx provide proprietary software tool stacks (Quartus & Vivado) which enable users to design, analyze, synthesize (compile), place-and-route and program designs as well as view results.

FPGAs have 2 distinct usage models, and these two usage models have very different tool chain requirements:

1. Used in place of ASIC for lower volume application. FPGA-specific RTL code is targeted for this specific application. Tool chain flow likely geared toward single FPGA. *This will likely be an end-product.*
2. Used for prototyping or emulation platform application. SOC RTL code may require minimal changes to target prototyping/emulation. Tool chain flow will require partitioning across multiple FPGAs. *The FPGA will not be the end product, the FPGA is mimicking the end product during development.*

5.4 Interfacing to Real Devices

Interfacing to external or real devices is sometimes necessary or even optimal when creating a pre-silicon validation platform. Real devices may include: HW/SW debuggers, JTAG DFD probes, logic analyzers, test chips, SPI flash, DDRx, UART, USB, PCIe, camera, sensors, audio, etc.

The FPGA Prototyping solution is typically utilized when needing to interface to a real device. Care must be taken to map the real device interface signals properly to the prototype hardware using daughter cards. Daughter cards can be supplied by the vendor or custom developed (Intel utilizes both). Custom developed daughter cards are long-lead items and should be identified in the technical readiness phase of a project. Clocking and interface rates must be respected. Shims or speed bridges can be utilized to interface to devices running >150MHz. Real devices can also cause issues with validation platform models where clocks need to be started and stopped as the interface will likely stop functioning correctly when clocks are stopped. Validation platforms where free-running clocks are utilized (do not need to be stopped) are better suited for interfacing to real devices. See typical daughter card in below Figure 23.



Figure 23: ProFPGA Generic Debug Daughter Card

Emulation solutions can also be interfaced to real devices but almost always require some type of speed bridge to connect to the emulator. This mode of emulation is typically called ICE (In-Circuit Emulation). For more details, refer to advanced topics section 6.5.2.

5.5 EFFM: Emulation & FPGA Friendly Methodology

EFFM stands for Emulation & FPGA Friendly Methodology and is a set of guidelines that can be utilized by IP design teams to create high-quality, integration-friendly SIP (Soft IP) designs, HIP (Hard IP) designs, memory models, UPF (Unified Power Format) code, validation models and other collaterals utilized by SOC integration teams. Intel projects that utilize EFFM guidelines help teams to increase SOC model velocity by catching issues earlier in the design cycle and defining IP delivery rules with enforcement mechanisms to produce clean collaterals.

Non-EFFM compliant collaterals can cause weeks-long delays for validation platform enablement which then causes delays to PSS (pre-silicon solution) HW/FW/SW validation milestones.

Enabling EFFM helps Intel to align on required tasks and drive design rules across IP, design validation and SOC integration teams which will increase model quality and validation velocity. EFM checks can be utilized as turn-in gating checks to make sure collaterals will pass thru emulation and FPGA synthesis or compile flows, similar to a code linting process.

EFFM checks can be utilized to flag:

- Incorrect model definitions (library or RTL macros issues)
- RTL synthesis issues, emulation/FPGA compile tool issues, especially those that LRM (language reference manual) checks didn't catch
- UPF compilation issues
- Memory modeling issues (creating models that can be synthesized by emulation/FPGA)
- Minimize model bring-up issues with incompatible forces, clocking and reset constructs
- Validation collaterals (fix debug or test issues)

For more information: <https://goto/effm>

6 Validation Platforms

6.1 Types of Pre-Silicon Validation Platforms

In this section, we will explore the pre-silicon validation platform options. We will be discussing 4 primary validation platform types in this chapter as shown below in Figure 24.

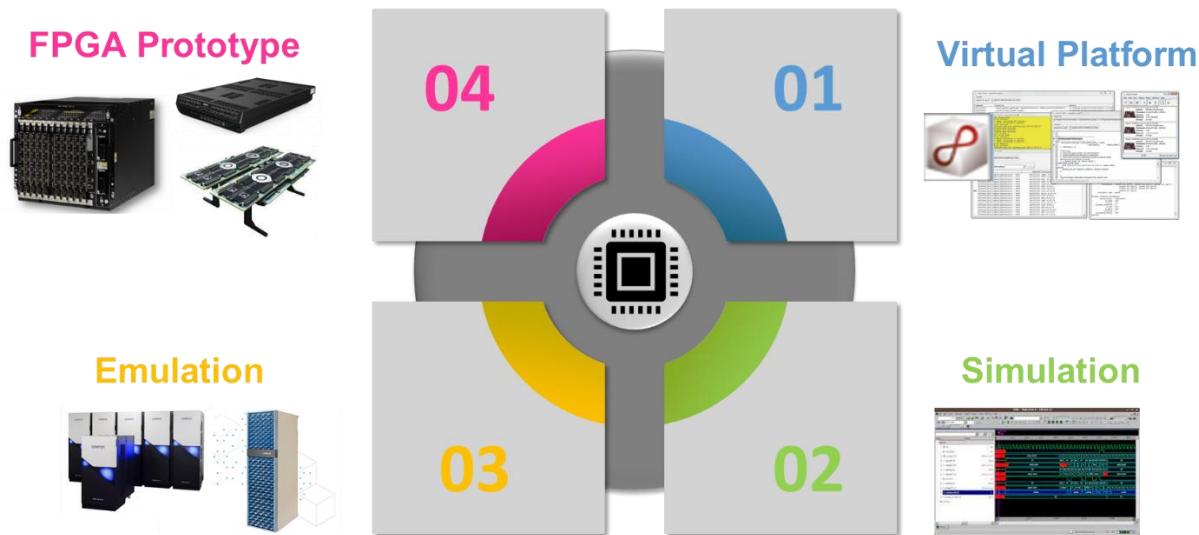


Figure 24: Pre-silicon Validation Platform Types

6.1.1 Virtual Platform

Virtual Platform models are abstract software systems that model architecture, transaction and system flows of the design hardware and not the actual implementation of the design. These models utilize a ***Behavioral level*** of abstraction and typically have no concept of design RTL or clocking. Virtual platforms should be viewed as a digital twin and a separate interpretation of a project spec coded in a language native to the Virtual Platform like C/C++, SystemC or Device Modeling Language (DML). Though these systems are abstract and lack accurate design timing, they can be utilized extensively to develop and boot BIOS and OS images and perform system software level verification.

The virtual platform is heavily utilized during the early development of a project as these models can be constructed very quickly, especially compared to that of the RTL design model. These models enable early development of Firmware, Driver and BIOS ingredients.

Since these models are coded as software systems, they can model very complex and large multi-die projects and can be executed very fast booting a large SOC in minutes. However, because of the behavioral level of abstraction and lack of clocking accuracy, these models are not capable of final sign-off and there are many tests and workloads which cannot be meaningfully run on this platform type.

Note: another class of models which could be considered in this section would be low-level firmware simulators like a Pcode simulator. While not traditionally considered a virtual platform, they are coded with behavioral levels of abstraction and are utilized by FW developers.

Virtual Platform models can be combined with abstract timing and power layers for early evaluation of full multi-core and multi-threaded workloads. These greatly help with architecture pathfinding and early performance per watt projections.

The primary Virtual Platform tool for Intel is Simics (owned by Intel).

6.1.2 Simulation

Simulation models utilize hardware description languages like SystemVerilog to capture the DUT RTL design and leverage testbench constructs (which may be synthesizable) to stimulate and monitor the states of the DUT. Simulation models utilize ***RTL level*** of abstraction (DUT modeling) but may also leverage ***Behavioral level*** for testbench transactors (or bus functional models). These simulation models enable verification of DUT low-level details such as clocking, timing, X-propagation, assertions, error injection, power states and some analog operations. Simulation models support complete signal debug visibility and have very mature compile, run and visualization tools. Simulation models also support UPF (unified power format) constructs for modeling low power aware designs. Since the simulation model leverages an RTL level of abstraction for the DUT, they are typically considered one of the most accurate models. Simulation models can implement large SOCs and are not limited by hardware capacity like Emulation and FPGA prototyping models (we will discuss model performance issues in section 6.2.1). Simulators support multi-state modeling of: 0, 1, X, Z, strong/weak states.

The simulation model can also be extended into the ***Gate level*** of abstraction for supporting timing-accurate gate-level simulations (GLS).

The simulation model is a popular validation platform due to the relatively low model bring up effort required as well as the capabilities described above. However, one of the glaring limitations of the simulation model is the very poor model performance. This platform is not capable of supporting long, cycle-intensive validation workloads including many pre-silicon validation tasks like FW development, BIOS boot, OS boot or PnP (Power & Performance) workloads.

Simulation models can also be used in Formal Verification, as described in [Symbolic Simulation](#).

The primary simulation tools for Intel are Synopsys VCS, Cadence Xcelium, and Siemens Questasim

6.1.3 Emulation

Emulation models utilize special hardware & software to *accelerate* RTL design verification. This specialized emulation hardware is typically implemented with commercial FPGAs or vendor-specific reprogrammable custom ASIC devices. Emulation modeling requires a very complex vendor tool stack to manage technology mapping from an **RTL level** of abstraction to a vendor-specific gate abstraction (which enables model acceleration). The vendor tool stack helps to simplify or partially automate the Emulation model compile, build, run time testbench configuration and enable signal debug visibility of the system. Emulation models can run orders of magnitude faster than RTL simulation models.

Emulation models can be used as a vehicle for both verification and validation due to its balanced capabilities of high model speed and rich debug environment. Debug visibility is considered very high and can have focused visibility scope with options for full visibility scope (may impact model performance). Debug can be enhanced by leveraging protocol-level trackers (see 6.2.3.1) and other monitors during run time.

Since emulation is hardware-based, models are executed with only two logic states, they are not considered useful for analyzing X-state initialization. Additionally, emulation is typically not useful for modeling analog components or accurately modeling precise circuit timing. These systems are very hardware and software intensive (requires specialized SW stack to manage the complex model build flow into expensive, vendor-specific emulation hardware racks) contributing to its high costs to own.

Emulation platforms provide scalability from IP to large SOC systems but at the same time have hardware capacity limitations that must be accounted for when building large SOC models. Model hybridization & IP stubbing are techniques to manage model size.

Emulation model performance is typically throttled by host communication from transactors and the testbench to the emulator hardware. Testbench efficiency (time spent running on the host rather than the hardware) is another issue that impacts model performance.

Intel also utilizes Emulation to support Gate-Level Emulation (GLE) models. The GLE model is an emulation model that leverages the project's **Gate level** of abstraction. This is typically utilized to support HVM (high-volume manufacturing) or scan validation.

Intel has been driving a multi-vendor approach with the Emulation options for Intel as 1) Synopsys ZeBu Server4/5 2) Cadence Palladium Z2 3) Siemens StratoPlus

6.1.4 FPGA Prototype

FPGA Prototype models are also utilized to *accelerate* RTL design validation including FW/SW development tasks. These models require specialized FPGA-based hardware along with complex software stacks to manage technology mapping from an ***RTL level*** of abstraction to a vendor-specific Gate abstraction (which enables model acceleration).

One of the key benefits of FPGA prototype is the model speeds are typically 5-10x better than emulation models. This benefit can be offset by the manually intensive design process and traditionally longer model build turnaround time (TAT). One project strategy to consider is to debug functional issues on emulation models where visibility is high and then transition to FPGA prototype models once the DUT stabilizes.

FPGA prototype models are ideal when projects require interfacing to real devices: HW/SW debuggers, JTAG DFD probes, logic analyzers, test chips, SPI flash, DDRx, UART, USB, PCIe, camera, sensors, audio, etc. Refer to section 5.4 for more details about interfacing to real devices.

FPGA prototyping models have traditionally supported less debug (*limited*) visibility than emulation/simulation models. Prototyping models utilize signal capture techniques similar to built-in logic analyzers with trigger logic, capture logic and signal trace storage logic all requiring FPGA resources to support. This extra debug logic adds resource utilization and congestion on top of the existing DUT design resources. Adding debug signals also requires signal instrumentation (or signal selection) and rerunning portions of the model build flow to add/change debug signals.

One technique that has improved FPGA prototyping visibility & debug has been adding support for protocol-level trackers to capture and view “protocols” rather than “signal-level” capture. These trackers can be added early in the flow and resources accounted for to improve debug turn around. See section 6.2.3.1.

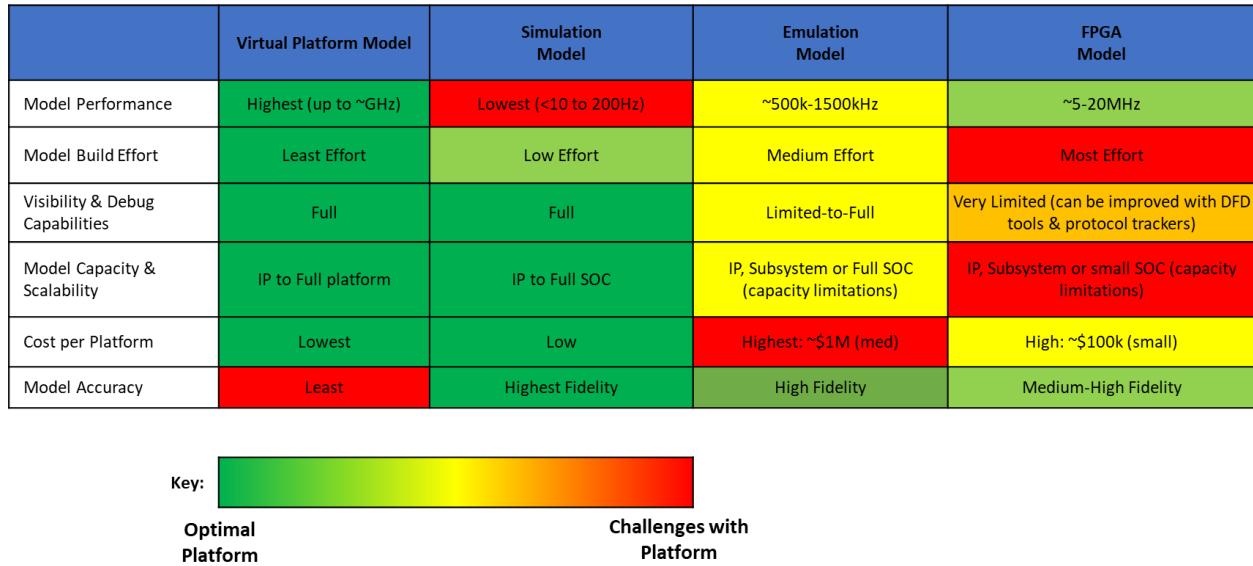
FPGA prototyping models are utilized to support FW/SW developers, DFD HW/SW developers, Production HW/SW/FW interoperability and Interactive SW debug

FPGA prototyping solutions have been historically dominated by Xilinx (AMD) FPGAs but new solutions from Altera (Intel) have been utilized on Intel production projects.

Intel has been driving a multi-vendor approach with the FPGA Prototyping tools for Intel as 1) Synopsys ProtoCompiler/HAPS-100 2) Siemens VPS/ProFPGA-10M

6.2 Comparisons and Tradeoffs between Validation Platforms

This section discusses the critical metrics of each of the Validation Platforms. We will focus on 6 metrics: 1) model build effort 2) model performance 3) visibility & debug capabilities 4) model capacity/scalability 5) model cost 6) model accuracy. Refer to Table 2 below for a summary of these metrics followed by detailed discussions on each metric.

**Table 2: Heatmap Comparison Summary of Validation Platforms**

Technological advances have led to the crossover of platforms into areas of validation that were historically prohibitive, so it's important to continue to assess validation platform capabilities when considering a new validation project. This section aims to give an overview of each platform type, compare platform strengths and limitations, and discuss the why or when each platform is typically used.

6.2.1 Model Performance

Model performance refers to the speed at which a validation platform can be executed or produce results. This is typically measured as cycles per second (Hz) with cycles relating to the highest frequency clock of the model.

- Virtual platform models have the **highest** model performance. This is because virtual platform models utilize behavioral levels of abstraction and do not implement the detailed design behavior (clocking, etc) of the DUT. Since this is a software model running on a host system, the performance can be impacted by the class of host (latest generation of Xeon class server with adequate memory, etc.). Runtime performance degrades only as the *exercised* portions of the model increases. This is different than Simulation models where speed is inversely proportional to size of the model. The Simics VP supports *virtualized execution* which allows regions of simulated target code to be run directly on the host (Simics VMP mode). The closer the host architecture (i.e. ISA) is to the target, the more effective VMP is. Instructions that are not supported on the host need to be interpreted (vs. virtualized) and this will slow the model down. Generally, the latest & greatest host hardware allows near at-speed simulation of next-generation platforms and therefore, model can approach GHz performance.
- Simulation models have the **lowest** model performance. The simulation platform performance is poor because the RTL software must model a very large number of design

events per clock cycle. This also implies that as simulation models get larger, the performance decreases with the model size & complexity. Since this is also a software model running on a host system, the performance can be impacted by the class of host (latest generation of Xeon class server, core count & memory size). Simulation model performance for a large Intel IP is typically 20-200 Hz, and that simulation performance for a large Intel SoC is typically sub 10Hz, with the worst falling below 1 Hz.

- Emulation models have **medium** performance overall but typically have **much better** performance over the equivalent simulation model (this can be several orders of magnitude better in some cases). This is because emulation models leverage acceleration of the RTL design and clocked events can now be implemented with flops/latches and run in parallel in the emulator hardware. Emulation model performance (DriverClock) is typically between 500kHz and 2MHz. Emulation model performance can be limited by transactors (testbench) throttling access to the emulation hardware (DUT). Projects often instrument and track model performance. Emulation model can be painful or not practical for certain use-cases (OS boot, long workloads, etc).
- FPGA prototype models have **high** performance overall but typically **better** performance than traditional emulation models (up to 10x faster). FPGA prototype models which leverage real devices can be 5-10x faster than emulation models. Since typical FPGA prototype models do not leverage transactors, model performance is no longer throttled by host communication and more throttled by levels of logic and other traditional design limitations like clocking and the FPGA technology. FPGA model performance is typically between 5-20MHz, with some real device interfaces support clock rates of 125MHz (like PCIe, etc).

See Figure 25 below for a perspective of the various validation platforms vs Performance

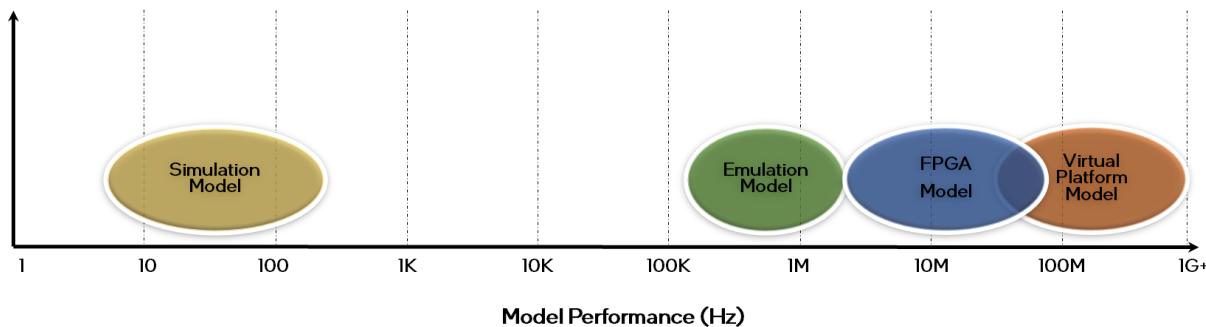


Figure 25: Model Performance

6.2.2 Model Build Effort

The Model build effort can be broken out into two key stages:

1. **Time to First Model Build:** This includes the amount of effort and time required to build the first model to run on the validation platform
 - Virtual Platform models require ***least effort*** to develop as they are coded as system-level software models. They are the fastest models to create. They are usually the first model available for a project, typically very early in the design cycle.
 - The Simulation model is typically the next model to become available after design RTL drops begin. These models require ***more effort*** than the Virtual Platform but still require relatively low effort to create. Model effort includes transactor and testbench development.
 - Emulation models require ***more effort*** than Simulation models to develop because of the complex emulation tool stack which assists the user in mapping the RTL design to the emulator hardware technology. This effort can be negatively impacted by poor quality design RTL inputs or code that does not support emulation/FPGA compile flows (not EFFM compliant). Care must be taken to account for model size and resource limitations of the emulation hardware. Emulation model development is often treated as a subsequent stage after simulation model development, both in terms of building as well as content enabling, but this is not a true dependency.
 - FPGA prototyping models require the ***most effort*** to create. The FPGA prototyping tool stack requires more manual user intervention with less build automation than emulation (partitioning, timing constraints, etc). Interfacing to real devices can also be long-lead development tasks which must be considered up front during the technical requirements phase.
 - *Note: Large changes between RTL drops will cause some thrash which may be similar to the first model bring up efforts on some validation platforms*
2. **Model Build Turn Around Time (TAT):** This includes the amount of effort and time required to update and rebuild an existing model with a small change.
 - Virtual Platforms require ***significantly less effort/time*** to make small updates to the model and rerun. The model compile TAT is very short allowing for ***many iterations*** per day
 - Simulation models also require ***less effort/time*** to make updates to the model and rerun. The model compile TAT is short allowing for a ***few iterations*** per day
 - Emulation models require ***more effort/time*** to make updates to the model and rerun. The model compile TAT is long which may support ***1-2 iterations*** per day (1 iteration for large models)
 - FPGA prototype models require the ***most effort/time*** to make updates to the model and rerun. The model compile TAT is the longest which may support 1 iteration per day. For large models, this may take ***more than 1 day*** to rebuild the model

- Note: for emulation and FPGA prototyping, the backend place/route tasks may account for >50% of the overall model build TAT

Refer to Figure 26 below to view the relationship between Model Effort and Model Performance.

Model Build Effort vs Model Performance

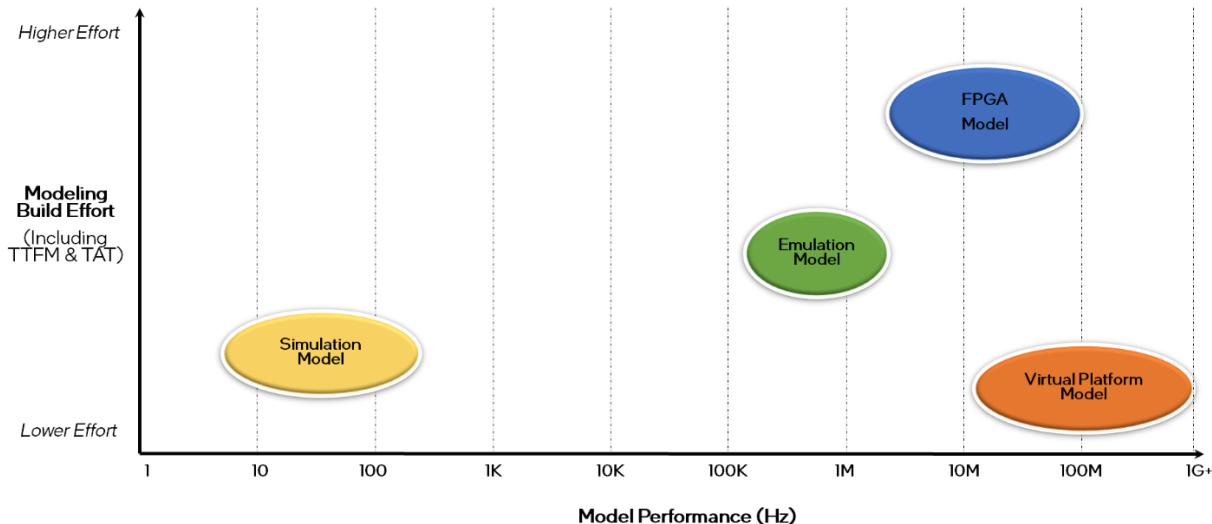


Figure 26: Model Effort vs Model Performance

6.2.3 Visibility & Debug Capabilities

Visibility refers to the level of design detail that is available to perform debug on the validation platform during development (visibility enables debuggability). Debug capabilities refers to features available to isolate and root cause issues. This section reviews some of the visibility and debug capabilities for each validation platform type.

- Virtual Platform models provide **full visibility** of the CPU states, memory states, BIOS/OS execution status, modeled system register states, transactional bus interfaces and overall system status. Virtual platform debug capabilities include setting complex triggers and breakpoints to debug system-level issues. For example, Simics supports debuggers and other SW development tool suites. Virtual platform models are also deterministic allowing for issue reproduction. Trace logging is cheap (unlimited) and easy in Simics (the user can increase log granularity level for a subsystem/IP). This is equivalent to signal-level FSDB generation in RTL.
- Simulation models provide **full visibility** of the RTL design. Simulation tools support full waveform viewing capabilities and the ability to trace signals (to source or destination) in the design using hierarchy browsers. Simulation debug capabilities include support for X-propagation debug, SystemVerilog assertions, ability to set breakpoints, deterministic

behavior to reproduce failures, and delay annotation to debug timing issues. Debug visibility can be disabled to run regressions and then enabled if debug is required to improve model performance.

- Emulation models provide ***limited-to-full visibility*** of the RTL design. Emulation tools support full waveform viewing capabilities and the ability to trace signals (to source or destination) in the design using hierarchy browsers. However, emulation performance will begin to be impacted by enabling full visibility. Simple signal capture for a design does not require a rebuild of the model.
- FPGA prototype models have traditionally provided ***very-limited visibility*** signal capture capabilities. This requires instrumenting the DUT with built-in logic analyzers that support trigger logic, capture logic, and limited trace storage. This visibility logic consumes FPGA resources and can create congestion issues (there is a cost for instrumenting a DUT). Signal instrumentation can be limited to tens of thousands of signals rather than full visibility with these features. Signal capture logic does not require stopping clocks on the model. Additionally, the debug cycle can be long as this technique requires rerunning the model build flow to change signal instrumentation. Next-generation FPGA prototyping solutions support ***limited-full visibility*** readback capabilities. This requires building the model to support readback, hitting a trigger, stopping clocks and then unloading the state of the DUT registers and then reconstructing the waves to view. Once the state is unloaded, clocks can be advanced, and the next DUT state can be unloaded. Readback mode would be utilized on smaller windows around a trigger point. This won't work in models where intrusive clock control cannot be tolerated (those with real devices). Obviously if the DUT supports real devices, users must fully understand the implications of stopping clocks (for instance, stopping clocks on a PCIe or USB device would be catastrophic and require a reboot of the system). Readback works well on DUT models that support clock stopping. Model performance is *significantly* impacted (decreases) when readback is enabled.

6.2.3.1 Using Protocol-level Trackers for visibility

Protocol-level Trackers, also known as Transaction-Level Monitors (TLMs) refers to validation code that allows users to monitor a DUT at a protocol or transaction-level rather than capturing individual signals or buses. Protocol trackers must have a basic understanding of the transactions (or be protocol-aware) that will be captured. Protocol trackers stream the captured data from the platform into storage, typically in binary form. This data can then be written into formatted log files which users can process. Trackers will often have filters or triggers which can be utilized to focus debug and improve validation velocity. Protocol trackers can be utilized on simulation, emulation and FPGA validation platforms. For emulation/FPGA platforms, protocol trackers will need methods to connect to the protocol bus interface-of-interest and the captured data must be packetized via synthesizable collectors and then moved out of the emulator/FPGA for processing.

Adding too many trackers to an emulation platform can begin to degrade model performance, so this is something to consider when instrumenting a model.

6.2.4 Model Capacity & Scalability

This section will explore the platform capacity and ability to scale models from IP, subsystem, die as well as multi-die (full platform) systems. Moore's Law has been pushing next generation Server, Graphics and Client devices into new capacity/size zones which are straining previous generation validation platforms. It is critical that the validation platforms be able to support scaling for next-generation Intel devices.

- Virtual Platform model can **easily scale from IP to full platform**. Since this is a software model, the full platform model can be coded and executed with **little degradation** in performance.
- Simulation model can **easily scale from IP to full platform**. Since this is a software model, the full platform model can be coded. However, because model performance degrades with model size, users should expect **significant performance degradation** for full platform models.
- Emulation model requires **more effort to scale from IP to full platform**. Since this model requires specialized acceleration HW, there are physical model capacity limitations. After the model exceeds a certain number of logic gates, the model will no longer fit in HW. Intel teams often leverage reduced core count (minimum number of cores) or hybrid models to manage model size and run the bulk of the validation testing on the smaller model and then also build a full model to run some cross die or other key use cases to complete coverage. The emulation SW/HW more readily supports model scaling vs FPGA prototyping. We will discuss those limitations in the next bullet. Users should expect **moderate performance degradation** as the size increases. A very large, 128 board server model will also be very expensive to run. Emulation teams often look for ways to optimize the model size.
- FPGA prototyping model **significantly more effort to scale from IP to full platform**. Since this model requires specialized FPGA boards and cabling to connect between FPGAs and boards, there are physical model capacity limitations. After the model exceeds a certain number of logic gates, the model will no longer fit in HW. Intel teams often leverage reduced core count (minimum number of cores) or hybrid models to manage model size and run the bulk of the validation testing on the smaller model and then also build a full model to run some cross die or other key use cases to complete coverage. Because FPGA prototyping requires manual FPGA to FPGA and board to board cabling, scaling to a large full platform models is more difficult. For large models, considerations for connecting from rack to rack may also need to be considered. Users should expect **moderate performance degradation** as the size increases. A very large, 16-24 board (requires 2-3 racks) server model will also be very expensive to run.

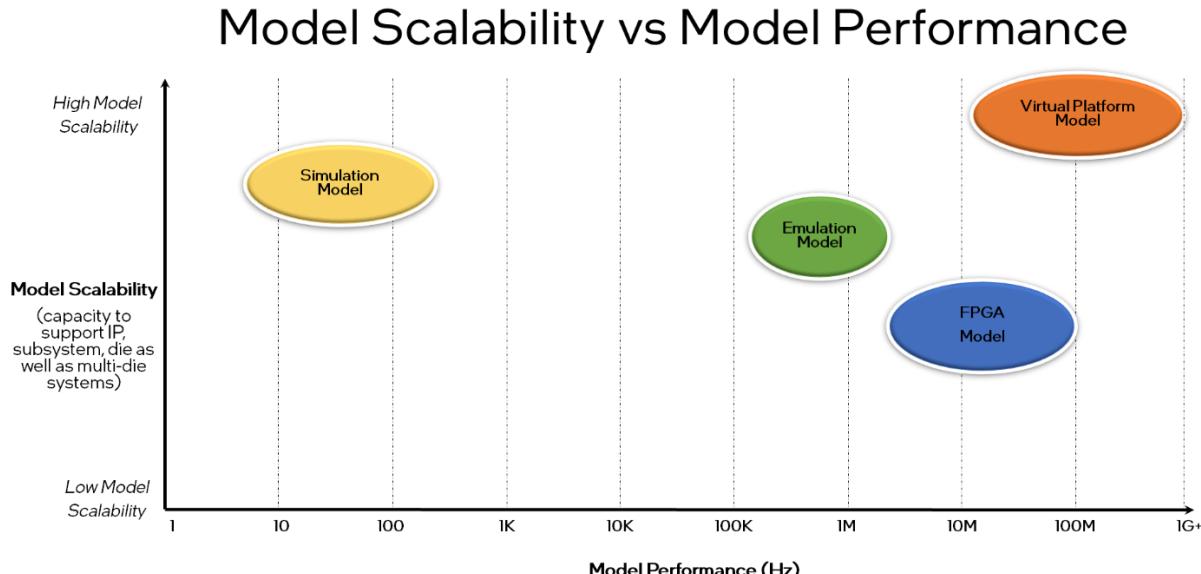


Figure 27: Model Scalability vs Model Performance

6.2.5 Cost per Platform

This section will explore model costs of the various validation platforms. Model cost will play a key role in validation planning decisions. We will need to explore SW, HW, licensing, maintenance and model development cost vectors with a goal to keep things vendor-agnostic. The following discussions assume an *instance* of a validation platform.

- Virtual Platform model is the *lowest cost* solution. Since Simics is an Intel owned tool, this is a significant competitive advantage. Tool licensing and maintenance costs are free. Simics does not require special HW to run (minimal cost of running models on compute cloud resources). The primary model cost for virtual platform will be driven by the resources to build and maintain the model. Use of 3rd party models incurs a licensing cost. Model replication is easily supported by running multiple jobs in parallel with minimal cost impact (running jobs in compute cloud).
- Simulation model is a *low-cost* solution. Intel utilizes EDA simulation tools including Synopsys VCS, Cadence Xcelium, and Mentor Graphics' Questasim to support simulation models. Teams at Intel leverage *enterprise* SW and maintenance licensing agreements with EDA vendors. There is a cost per seat but for our discussion, this cost will be considered minimal. Simulations can be run on any host, so we will also assume minimal HW costs (minimal cost of running model on compute cloud resources). The primary model cost for simulation will be driven by the resources to build and maintain the model. Model replication is easily supported by running multiple jobs in parallel with minimal cost impact (running jobs in compute cloud).

- Emulation model is the **highest cost** solution. The complex SW stack requires special licensing which Intel negotiates as enterprise agreements with key vendors. This typically includes annual on-site vendor AE and maintenance support. Intel typically purchases emulation HW (leasing is rare) and these specialized racks are co-located in pooled data centers in FM and IDC using NetBatch. These data centers require specialized cooling and networking capabilities to support the compute and disk intensive access to the emulation HW. In addition to the high cost of ownership, emulation models also require small teams to be able to build and maintain the models. Since this is a high-value capital asset, Intel utilizes metrics to track and drive emulation efficiencies and closely monitor usage and project board allocation (*Intel owns X number of emulators with a configurable number of boards per emulator, hence the term board allocation for a project*). Model replication is not easily supported for running multiple models in parallel. Emulation require HW allocations to run parallel models, so there is a cost associated. The key emulation cost metric is \$-per-cycle: Very High.
- FPGA prototyping model is typically a **high-cost** solution. The complex SW stack requires special licensing which Intel negotiates as enterprise agreements with key vendors. This typically includes annual on-site vendor AE and maintenance support. Intel typically purchases FPGA prototyping HW and these are co-located in pooled labs or data centers in FM, IDC and PNG using NetBatch. These pools require specialized cooling and networking capabilities to support the compute and disk intensive access to the FPGA prototyping HW. In addition to the high cost of ownership, FPGA prototyping models also require small teams to be able to build and maintain the models. Since FPGA platforms are a high-value capital asset, Intel utilizes metrics to track and drive FPGA efficiencies and closely monitor usage and project FPGA board allocation. Model replication is not easily supported for running multiple models in parallel. FPGA prototyping also requires HW allocations to run parallel models, so there is a cost associated. The key cost metric is \$-per-cycle: High.

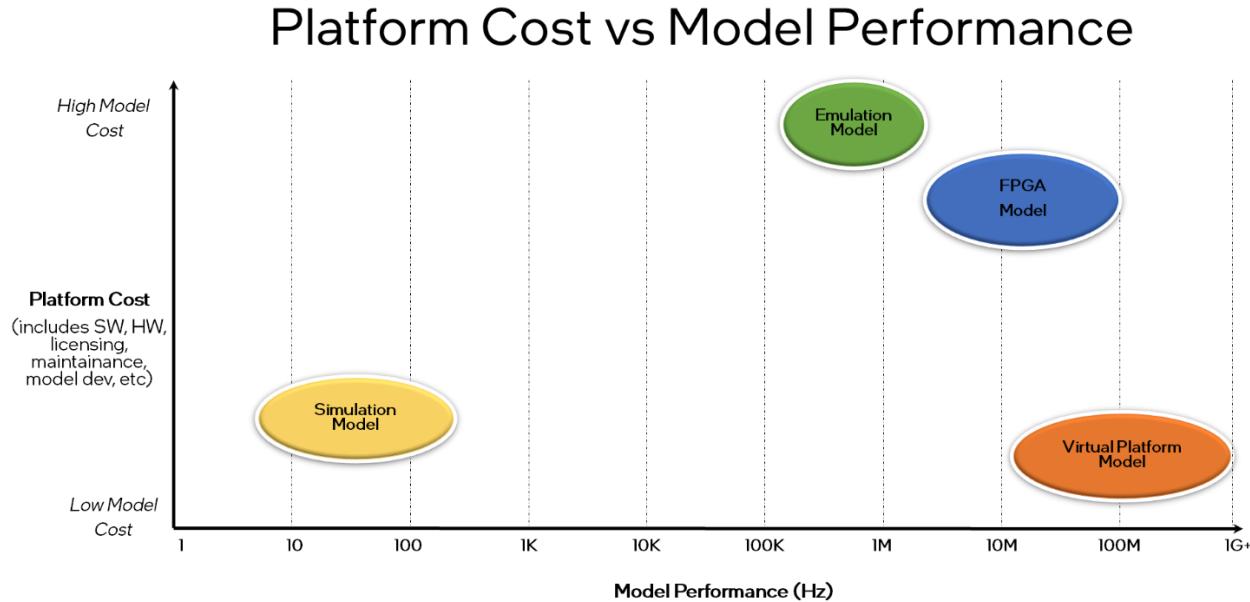


Figure 28: Platform Cost vs Model Performance

6.2.6 Model Accuracy

This section will explore model accuracy, in particular how closely a validation platform represents the behavior of silicon (creating a digital twin). Model accuracy is typically a trade-off between model performance and model fidelity (details).

- Virtual platform model is the **least accurate** solution. Virtual platforms leverage behavioral abstraction level coding of the model and typically lack details of clocking/timing logic.
- Simulation models is the **highest fidelity** solution. Simulation models leverage RTL abstraction level coding and include clocking/timing logic. Simulation models may include components for HIP (PHY's, etc), analog, power and other real-life scenarios.
- Emulation models is a **high-fidelity** solution. Emulation allows acceleration of RTL models including complex clock logic, power gating logic, and memory models. Emulation may not be able to model some components like HIP (PHY's, PLL's, etc) or analog logic which are typically removed. Note: PLL logic which inherently includes some analog behavior, typically needs to force PLL lock during emulation run time. Emulation models (like simulation models) attempt to maintain clock ratios.
- FPGA prototyping model is a **medium-high-fidelity** solution. FPGA prototyping allows acceleration of RTL models including *limited* support for complex clock logic, power gating logic (UPF), and memory models. Notes: 1) proto tools have historically not supported complex clock gate logic conversion [however, next generation proto tools have made some progress here] 2) complex memory models can be more difficult to implement efficiently in FPGA prototyping flows and may require some recoding [next]

generation proto tools are able to better handle SOC memories] 3) FPGA prototyping is not be able to model some components like HIP (PHY's, PLL's, etc) or analog logic which are typically removed. FPGA prototyping models which leverage traditional proto tools may simplify and not maintain SOC clock ratios (align clocks to improve performance or work around tool limitations). Next generation proto tools are able to maintain clock tree ratios.

Model Accuracy vs Model Performance

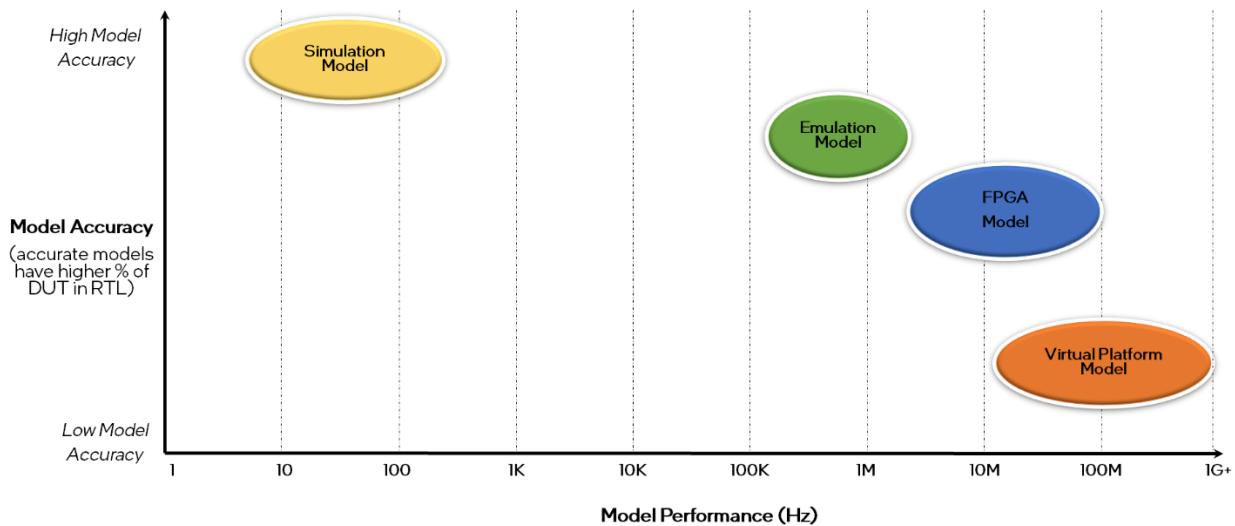


Figure 29: Model Accuracy vs Model Performance

Important Note: Emulation and Simulation and have been graded above as *high* and *highest fidelity* models, but it is important to note that in the pursuit of greater speed and lower latency these models will accept some inaccuracies. Common examples are reset speedups (to skip parts or rush some timed operations) and “*Infinitely Fast Pcode*”, where embedded firmware is run ‘instantaneously’ in a virtual processor instead of the embedded processor.

6.3 Hybrid Platforms

Hybrid models are validation platforms that consist of combinations of any of the previous validation platforms above. In practice, it is almost always a virtual platform connected to a simulator, emulator or FPGA platform. Hybrid platforms typically leverage transactors to interface between two levels of model abstraction. The main objective is to maximize the performance by targeting specific portions of the design onto an optimal platform where it has the greatest benefit. The connection and synchronization of these validation platforms can be very tricky and Intel utilizes transactor libraries to manage this layer.

Hybrid models can also be very useful in managing DUT model size. Refer to Figure 30 below:

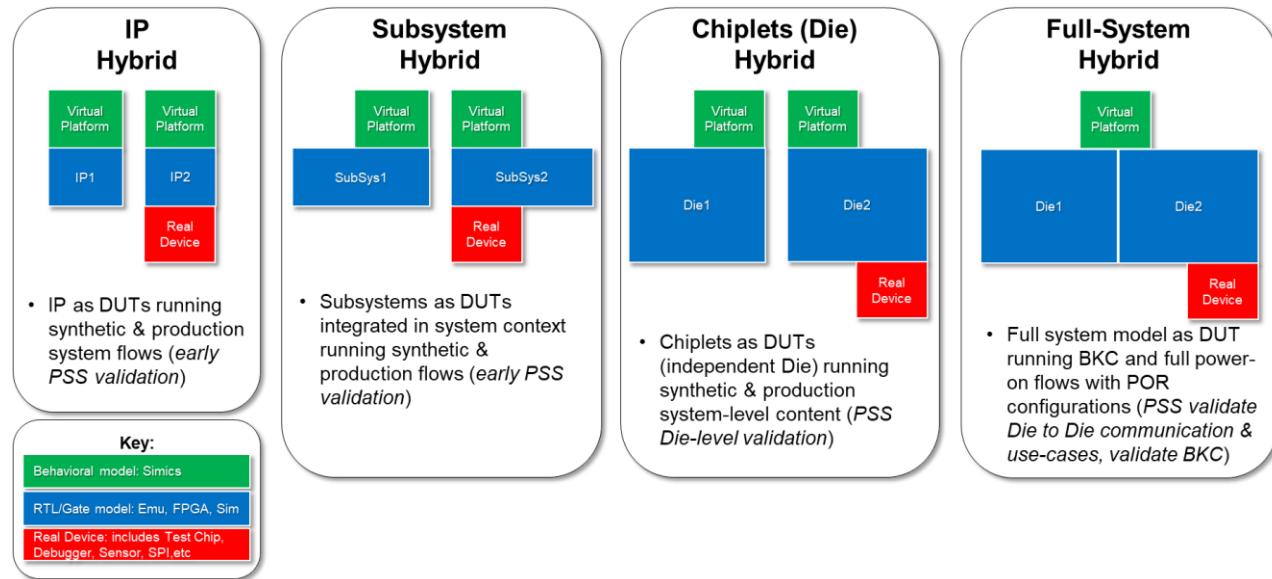


Figure 30: Utilizing Hybrid Models to Manage DUT Size

In addition, Hybrid models enable projects to manage design complexity, design RTL availability as well as full system (multi-die) modeling.

Another benefit of having hybrid models is being able to run test content across various validation platforms (Simics being a common front end to hybrid models).

Intel hybrid environments leverage Simics for the Behavioral abstraction layer and the either Simulation, Emulation or FPGA prototyping validation platform of choice for the RTL abstraction.

6.3.1 Hybrid Transactors

Hybrid transactors provide methods to interface between various levels of model abstractions and allow projects to create complex platforms. Intel Hybrid Transactor libraries include: SPARK (emulation), UHFI (FPGA: unified hybrid FPGA infrastructure) and FGT (FPGA: FPGA generic transactor).

6.3.2 Examples of Hybrid Models

There are many examples of hybrid models including hybrid-Emulation (combining *Virtual Platform + Emulation*), hybrid-FPGA (*Virtual Platform + FPGA Prototyping*) or hybrid-Sim (*Virtual Platform + Simulation*) models. Refer to Figure 31 for examples of hybrid platforms:

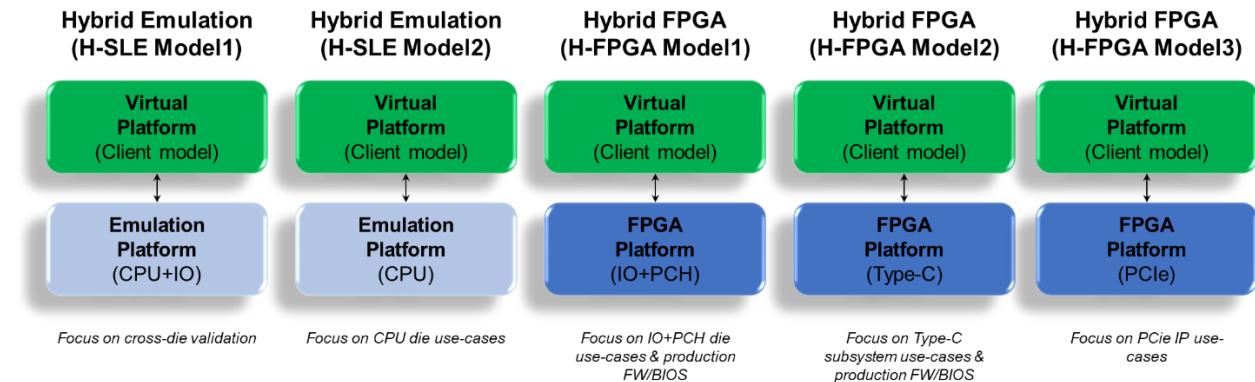


Figure 31: Examples of Client Hybrid Platform Models

6.4 Ideal Validation Platform Use Cases

Deciding which combination of validation platforms to use on a project requires a basic understanding of the metrics presented in section 6.2 above. The model build effort, performance, debug visibility, size, cost, and accuracy all play roles in driving Intel validation platforms. Validation users should attempt to use the right (*ideal*) validation platform at the right time in the project. Here are some final thoughts on *ideal* validation platform use cases

- Virtual Platform models: Ideal platform for early BIOS, OS, driver enablement, and system validation
- Simulation models: Ideal platform for initial hardware exercise and primary bug finding, for reset flow debug, system timing failure issues and short workload test cases. Simulation can run in 4-value mode, where signals can take on value 0, 1, X, or Z, making it especially adept for handling deep power management flows and analog operations.
- Emulation models: Ideal platform for reset flow debug, functional validation, system validation (CPU+PCH+test cards+BIOS), architectural performance validation, power management validation, manufacturing readiness and SV (system validation) tool & debug readiness. Great for medium workload test cases, not ideal platform for OS-based testing
- FPGA prototype models: Ideal platform for BIOS & OS boot flows, production FW development and for long workload test cases

6.5 Advanced Topics

6.5.1 Fast Emulation

Over the last several years, Intel has been working closely to collaborate with a couple of our key EDA vendors on a new capability called “fast emulation”. This technology has also been called “emulation acceleration” or “emulation offloading”. The goal of the collaboration was to combine emulation “*Ease of Use*” with FPGA Prototyping “*Performance*” and leverage the emulation SW stack with FPGA HW. The target for the DriverClock: 3~10 MHz. After several years of development, this technology has reached a level of maturity where it has been deployed on a few

Intel validation projects. The methodology includes building and debugging a model on the emulator where visibility is high and then retargeting to FPGA HW to accelerate the model. Fast emulation models are typically transactor-based.

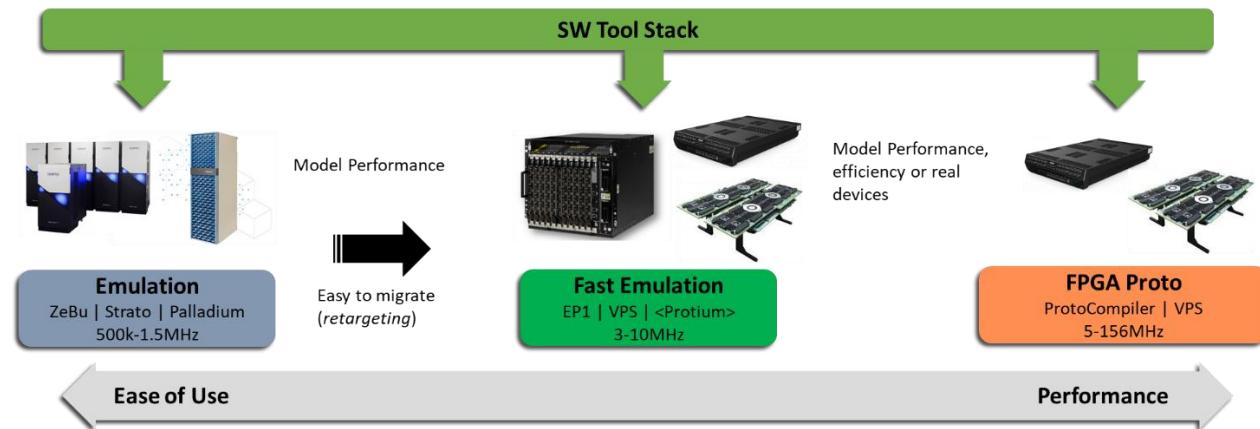


Figure 32: Fast Emulation Technology

6.5.2 ICE Platform

The Emulation platform is the one of the most desirable when it comes to pre-silicon validation of full RTL DUT models. However, our pre-silicon emulation platforms have a gap when users need fidelity, performance and debug in a single model with real world devices. In-Circuit Emulation (ICE) covers this gap and allows for highest fidelity, highest performance and richest debug, creating a post-silicon eco-system in a pre-silicon model. ICE allows RTL in the emulator to communicate with real world silicon devices and hosts, thus enabling post-silicon content that was never possible in our current pre-silicon platforms.

Creating this post-silicon eco-system in a pre-silicon model is important because:

- Allows for a true HW/FW/SW co-validation with the fidelity of silicon
- Allows for highest emulator performance by enabling use-cases like OS boot which allow users to run OS-based content
- Richest debug with deeper waveforms, protocol analyzers for PCIe, HBM and JEM trackers, etc

Figure 33 below shows a generic ICE platform which may include:

- DUT (design RTL) running on emulator
- Embedded Devices which are RTL models of key system devices that are modeled on the emulator to accelerate their execution. In a traditional emulation model (non-ICE), these would be implemented as slower transactors
- Virtual Devices which will be typically modeled as C-based transactors. The ICE model will include a minimal set of virtual devices to improve performance

- The key differentiator for an ICE model is the speed adaptor or speed bridge to connect the emulator to a real device. The speed adaptor is a board developed by Intel (supports voltage level shifting). Speed bridges would be required for higher speed interfaces support buffering. The Real Devices supported to date include RVPs (post-silicon Reference Validation Platform), ITP debugger, test cards and customer devices.

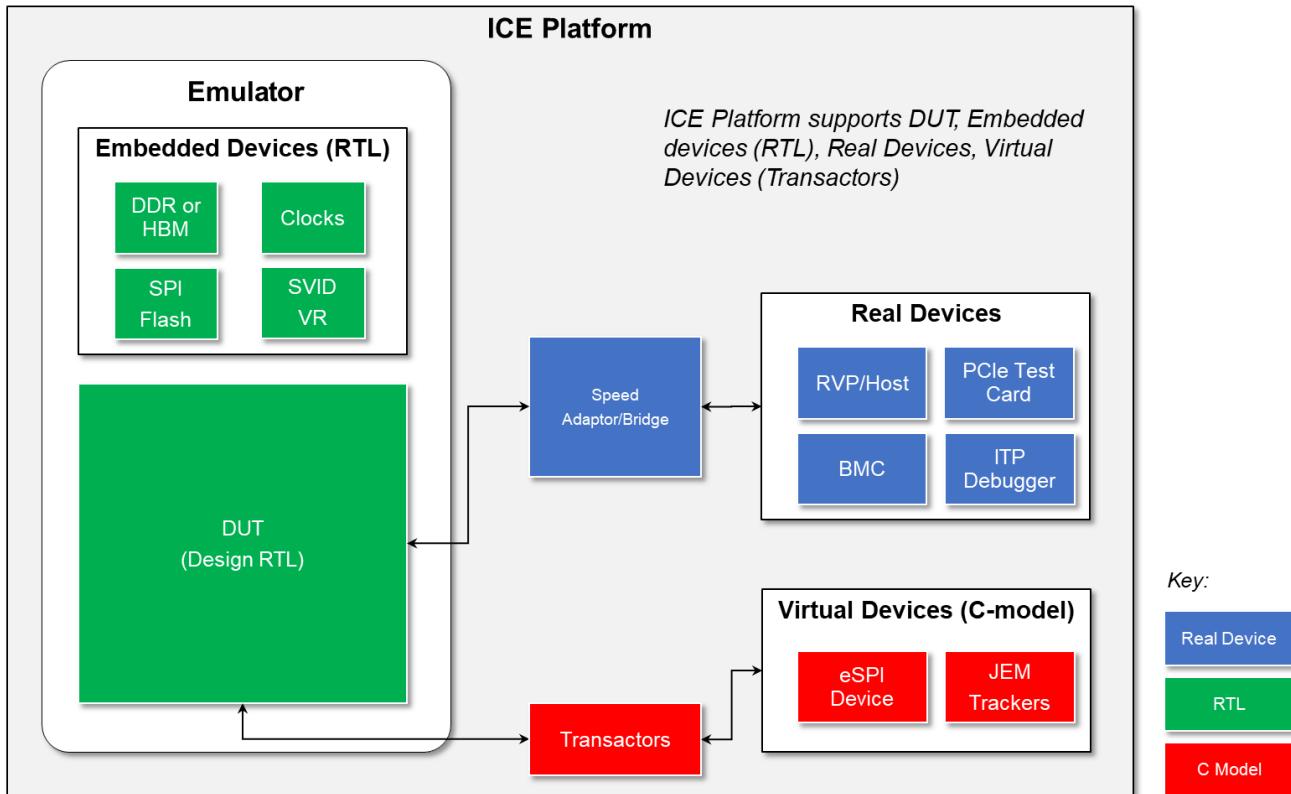


Figure 33: ICE Platform

6.5.3 Emulation Technology Under the Hood

This advanced topic section exposes some of the key technology differences between the current Intel emulator vendor solutions. This includes an under-the-hood peek at some of the technology utilized, data center impact along as well as some of the strengths of each vendor solution. Refer to Figure 34 below.

	Synopsys ZeBu5	Cadence Palladium Z2	Siemens StratoPlus
Technology:	Commercial (Xilinx) FPGAs: only technology scaling with Moore's Law, refreshing every 2-3 years	Custom Logic Processor chips, refreshing every 5-6 years	Custom FPGAs, refreshing every 4-5 years
Model Performance:	Fastest speed	Slower speed	Slowest speed
Datacenter (DC) Impact:	Small rack-mount DC footprint, lower power	Large DC footprint which requires liquid cooling	Large chassis-based DC footprint
Model Build:	More challenging from compile-time perspective	Fast compile times for small & large models	Fast compile times for small models
Debug Visibility	Slower runs when dumping waveforms	No cost of dumping waveforms on runtime/speed; full visibility waves available for offline conversion to FSDB for debug	



Figure 34: Emulation Technology: Under the Hood

7 Summary

Virtual Platform, Simulation, Emulation, FPGA and hybrid platforms are some of the options users can select to support validation tasks. Knowing which platform(s) to select to support a project is critical and will vary based on size and complexity of the design. One platform type is not inherently better than another and it is important to understand the platform metrics presented in this chapter when running validation content.

8 Future Work

The following items should be considered for future addition to this chapter. Some describe some recent development activities which are pushing the state of future validation platforms but were considered beyond the scope of this validation platform basics chapter. Other describe deeper topics and/or use cases that we may or may not want to add to the chapter:

- Utilizing Test Chips to Increase SOC Pre-silicon Validation
- Emulation Save & Restore Capabilities
 - And Emulation-Save-to-Simulation-Restore (eg, PSMI debug tool)
- Content Sharing Between Validation Platforms
- Intel build-your-own solution: FRIO
- Executable performance/architectural models (Keiko, ISIM, Archsim, ZSIM, etc)

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 11

Test Environments

By: [Eddy Ong](#)

1 Abstract

A high-quality Test Environment (TE) is critical to Validation. In the case of IP Validation and the smaller integration DUTs, the TE provides much of the stimulus and checking for dynamic validation. Even in top-level DUTs such as fullchip, the TE is still important for checking and monitoring for debug. Test Environment components are often reused both vertically and horizontally. TE cost can be considerable, but it is often more costly to skip doing proper TEs or to code them poorly or incompletely.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	6/1/2016	Initial Draft	Eddy Ong	John Faistl, Michael Bair
1.1	3/31/2024	Added newer definitions and added TBV	Michael Bair	Jim Huggins, Neriya Bar-Levav, John Faistl

3 Contents

1 Abstract.....	279
2 Chapter Revision History	279
3 Contents.....	280
4 Purpose.....	282
4.1 Why do we need this chapter?.....	282
4.2 What does this chapter cover?	282
4.3 What does this chapter not cover?	282
5 Background Concepts.....	282
5.1 DUT	282
5.2 Abstract Functional Model / Reference Model / Golden Model.....	282
5.3 Bus Functional Model	283
5.4 IDI	283
5.5 IOSF	283
5.6 SOC, IP, IU.....	283
5.7 Validation collateral (VC)	283
5.8 JEM	283
5.9 UTDB.....	284
5.10 Newtonian.....	284
5.11 TxTE	284
6 Test Environments.....	284
6.1 What is a Test Environment?.....	284
6.1.1 Components of a Test Environment.	285
6.2 Test API.....	290
6.2.1 Sequence and Sequence Library.....	291
6.2.2 Test Flow and Test Run Phases.....	292
6.3 Trace-Based Validation (aka Post-Processing)	292
6.3.1 Styles of Post-Processing	293
6.3.2 Post-Processing Toolset	293
6.4 Reuse	294
6.4.1 Vertical Reuse	294
6.4.2 Horizontal Reuse	294

6.4.3	Pros of Reuse	295
6.4.4	Cons of Reuse	295
6.5	Qualities of a Good TE Coder.....	297
7	Summary.....	297
8	Future Work.....	297
8.1	Handling mid simulation reset and power state changes.....	297
8.2	Workarounds	297
8.3	TE Cost.....	297
8.4	TE Documentation	298
8.5	TE Debug-ability	298
8.6	TE simulation performance and memory usage	298
9	References.....	298

4 Purpose

4.1 Why do we need this chapter?

In Dynamic Validation the TE is the main component of validation collateral. Especially at lower level DUTs (IPs and small SOC dut), the TE is often where all of the ‘intention’ that is described in the testplan (stimulus and checks, and sometimes coverage) is actually coded and made real. Moreover, the TE is software, and with any software there are many aspects to ‘getting it right’ and many costs associated with not getting it right. It is critical the Validation understands TE methodology and can strike the right balance in size, complexity, cleanliness, reusability, and other TE vectors.

4.2 What does this chapter cover?

This chapter will describe what Test Environments (TEs) are and why they are critical to Validation. It will describe common components of TEs and their function. The chapter will also discuss different types of reuse of TE components, as well as key tradeoffs to consider during development.

4.3 What does this chapter not cover?

This chapter will not cover any industry standard or Intel standard validation methodology like UVM or Saola.

5 Background Concepts

5.1 DUT

Device Under Test. The part of the validation system that eventually becomes a product, potentially in addition to a small amount of non-synthesizable support RTL to support connecting to test environments. Typically speaking, this is the functionality described by the RTL model. One could envision systems where a testbench is wrapped around a piece of TE collateral; in that case, the DUT is actually the TE.

5.2 Abstract Functional Model / Reference Model / Golden Model

These three terms mean roughly the same thing: any system NOT contained in the DUT whose intent is to simulate the behavior of some part of the DUT specification.

5.3 Bus Functional Model

A BFM is similar to AFM/Refmodel/Golden model above, except that it is a system that simulates the behavior of some portion of the environment that is outside the scope of the DUT. There are other pieces of collateral that are nearly synonymous with BFM. For example, “transactor” is sometimes used to describe a low-complexity BFM whose sole purpose is to send in transactions from the outside world.

5.4 IDI

In-Die Interconnect. This is the standard fabric connection between IA compute cores and GT Graphics IPs to the coherent side of the system fabric.

5.5 IOSF

Intel’s proprietary system fabric. This is often split into two pieces: IOSF Primary, which is the high-bandwidth traffic path for data and messages between agents on the fabric; and IOSF Sideband, which is a low-bandwidth messaging fabric used for handling things such as fuse downloads. IOSF Primary is capable of handling multiple virtual channels. Both IOSF Primary and Sideband have security checking.

5.6 SOC, IP, IU

An SOC is a *System on a Chip*, and industry term for ASICs that contain many different IPs that are put together on a single piece of silicon (or at least within a single package) that can perform much of the operation of an entire computer system.

IP stands for *Intellectual Property*, but in the context of SOCs it specifically means a stand-alone piece of functionality designed by one team and delivered to another for integration into an SOC. An IU is an IP, but is distinct in that it is an IP that is developed by the SOC team itself and thus typically held to a different set of IP standards, milestones, and drop frequency than a standard IP.

5.7 Validation collateral (VC)

Validation collateral is a general term encompassing all things created primarily for use by Validation. Frequently, VCs include passive and active elements; that is, monitoring and checking elements. This term frequently refers to functional checking delivered by a specialized third-party team.

5.8 JEM

JEM is an Intel toolset for creating monitors that can be used within both simulation and emulation. Depending on the needs, this can include interaction with the test environment through a SW

monitor and analysis ports, or it can simply write the monitored data to a JEM trace file. The file can later be accessed via the JEM API.

See <https://goto.intel.com/jem>

5.9 UTDB

UTDB stands for Unified Transaction Database

UTDB toolkit is used for post processing validation.

UTDB toolkit provides a set of Python APIs for:

Upload test data to UTDB database (trace)

Query the data in the database

Detect flows in time series traces

Collected coverage information from the database data

5.10 Newtonian

Newtonian is a tool that uses machine readable architecture and implementation specifications to generate passive validation collateral (monitor, tracker, checking, coverage) that can be integrated into the build and validations flows of an RTL model.

See <https://goto.intel.com/newtonian>

5.11 TxTE

6 Test Environments

6.1 What is a Test Environment?

A test environment is a software layer that sits between the Validator and the DUT (design under test). It plays a large part in implementing the validation strategy used to validate the DUT, namely through checking, stimulus and debug. Some functions of a test environment are:

- Checkers, which runs alongside the DUT, checking the DUT correctness.

- The test environment provides Validators with a window into the DUT by implementing a test interface in which the Validator can write tests to validate the DUT at an abstract high level.
- The test environment contains coverage monitors to track what events and flows have been hit within the DUT.
- When the test runs, the test environment prints multiple log files that log what the DUT and TE checkers did in response to the stimulus. These log files are very useful in creating a theory of what went wrong when checker detected failures.

The test environment is a collection of software, and has all the attributes of software projects written using modern object-oriented programming languages; however, test environment code has unique hardware concepts not found in regular programming languages like C++ or Java:

- A concept of simulation time and clocks. In regular software, the user will write and execute a subroutine, also known as a method or function. In test environment code, there are two types of functions, regular functions that are similar to other programming languages, and time consuming functions. The special thing about a time consuming function is that it will cause the function to wait for simulation time to pass.
- Another way test environment code is special compared to regular software is that TE code has two personalities, a persistent static side and a dynamic side. The dynamic side is similar to objects in common programming languages like C++ or Java, meaning they can be created and destroyed at any time. In the static side (enforced by languages like Specman e or methodologies like UVM), objects are created at time zero before the RTL simulation starts, and they exist until the end of simulation when the test ends. Users cannot create or destroy these static objects outside of these two times.
- The TE code has a unique concept of containing 4 values per bit (1, 0, X and Z), instead of just 1s and 0s

In some ways, the test environment is like the operating system in a personal computer. An OS manages the hardware and provides a simpler abstract interface to the underlying hardware rather than the 1s and 0s that the hardware understands. In that sense, the test environment is like an operating system for the DUT, it provides an abstraction of the main interface into the DUT. Validators use these high-level interfaces instead of low-level signals when driving stimuli into the DUT. The test environment also manages the hardware by configuring it and taking the hardware through reset and getting it ready for the Validator's test.

6.1.1 Components of a Test Environment.

For maintenance and reuse reasons, the TE is broken into multiple smaller components. There are standardized validation methodologies like UVM that dictate how to divide the components and how to connect them. The following sections will describe some common types of persistent TE components; they will not be a complete list of all possible types of components. In many validation methodologies, these components are known as TE agents. As mentioned before, the persistent part of the TE is unique to TE code, these TE components are objects in the software world but what is special about them is that they get instantiated before start of simulation, just before simulation time zero, and are destroyed after simulation ends.

6.1.1.1 BFM (Bus Functional Model)

Full CPU/SOC's are too complicated and too slow to validate in one monolithic simulation. To enable efficient validation of the DUT, the design is divided into smaller pieces. In order to test these smaller blocks, we need something to mimic what used to surround these blocks. That is what BFMs do; they are TE components that replace the missing parts of the design.

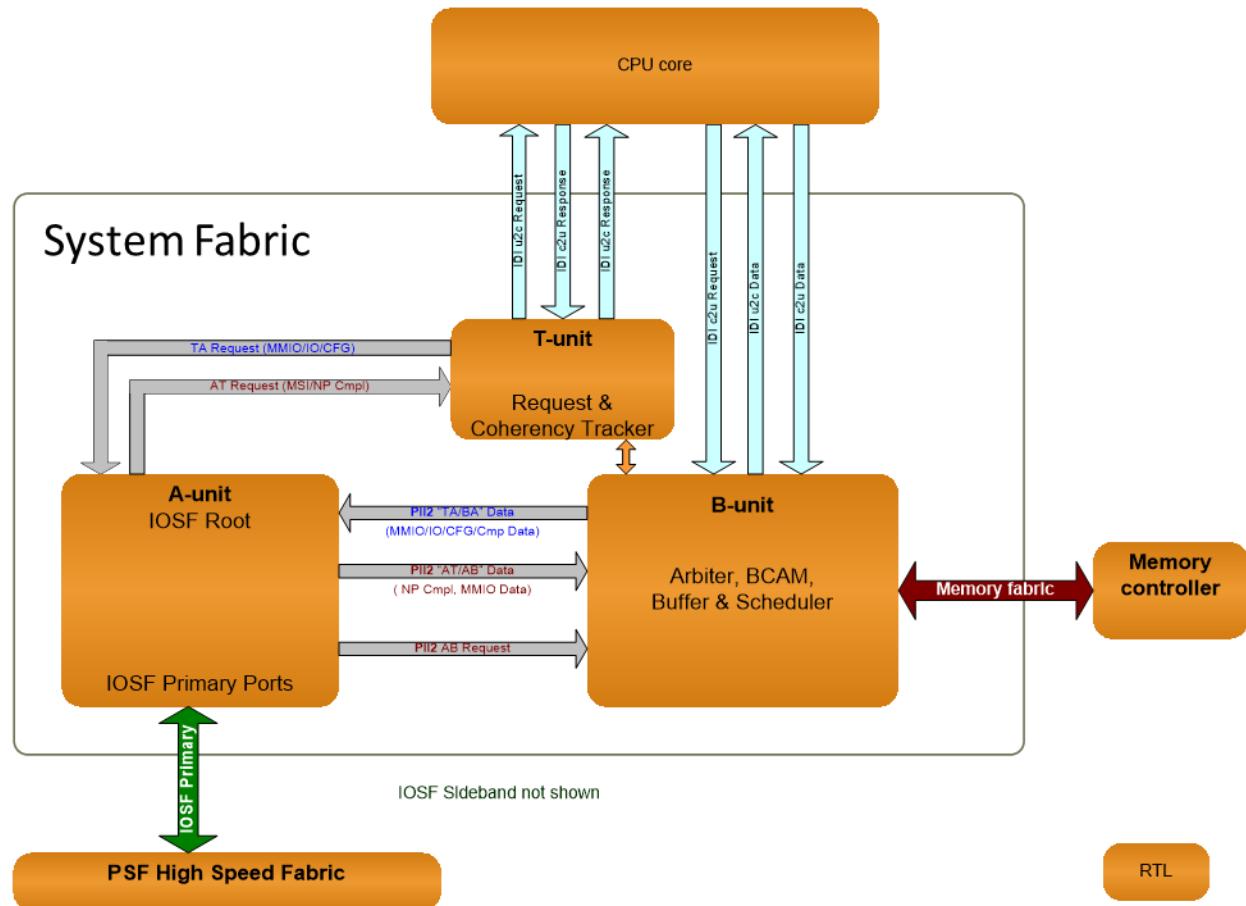


Figure 35 Block diagram of a System Fabric with connection to processor cores, memory and IOSF fabric.

Figure 35 shows a block diagram of an SOC with a System Fabric connected to processor cores, memory and IOSF fabric. In this example, the System Fabric is the design we are testing, also known as a DUT or design under test. There are three main connections to the System Fabric, IDI connections to the cores/graphics on the north, devices on PSF fabric through IOSF connections on the south, and connection to memory controller through some memory fabric. In the System Fabric cluster's test environment, RTL for the cores/devices/controllers on the other side of these three connections are replaced with test environment code in the form of BFMs, as shown in Figure 36.

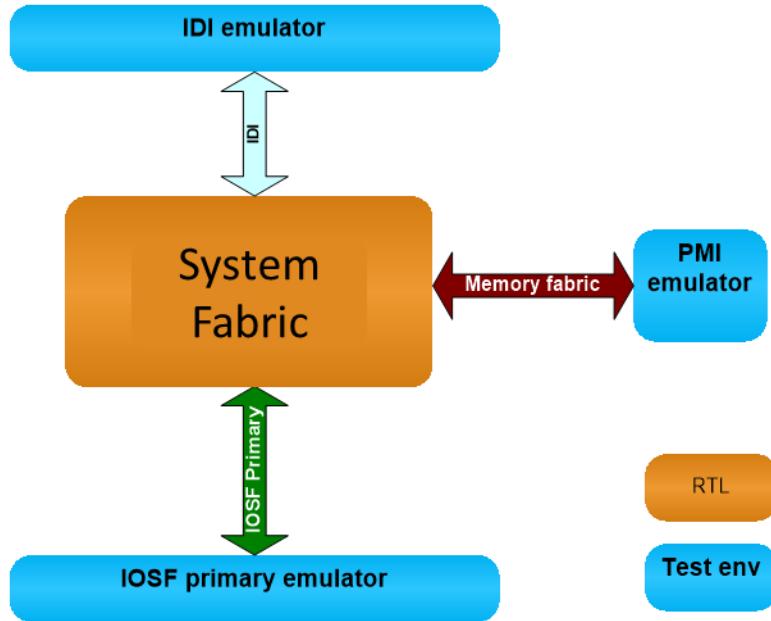


Figure 36 System Fabric cluster test environment block diagram

In an actual validation environment, we need more from the test environment than just emulating the missing RTL blocks. We also need other features like a test interface and checkers. The industry term for the block used to replace the missing RTL is called a BFM or bus functional model. Figure 37 shows the System Fabric test environment with more details of the sub blocks inside of the IDI BFM. Note that this is just an example; it does not mean that all the blocks shown in the IDI BFM must exist in all BFM or TE agents. The following sections will describe each of these blocks in more detail.

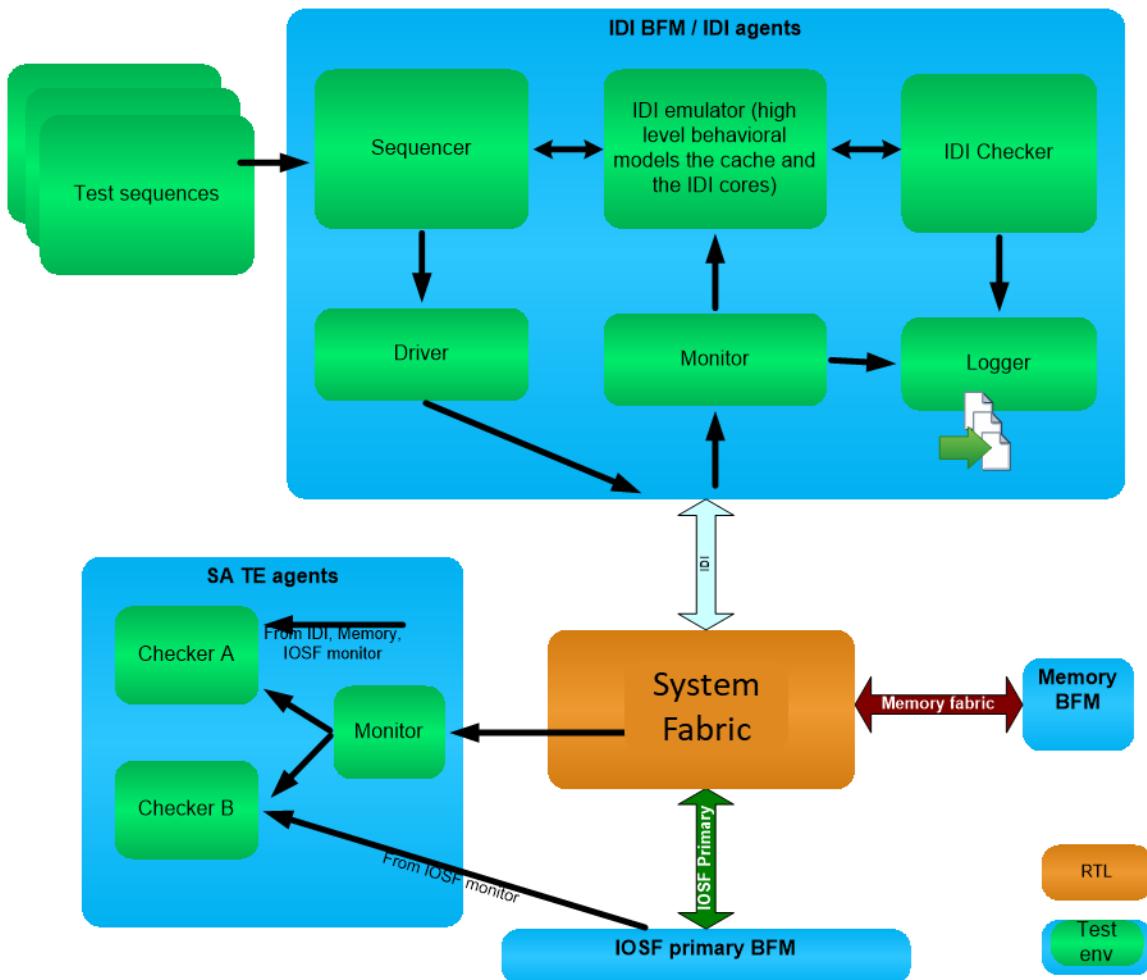


Figure 37 System Fabric cluster test environment block diagram with IDI BFM details and SA internal checkers

6.1.1.2 Sequencers, Drivers

Using the IDI interface from the processor core to System Fabric example, the TE defines the IDI transaction, the base object that TE components at that interface processes. This IDI transaction is similar to an object in object-oriented programming, it is a collection of fields that makes up the IDI transaction, and is the data structure being passed around between the blocks inside of the IDI BFM shown in Figure 37.

At any primary interface of the DUT, the base transaction object forms the base of the test. In the example we are using, the base transaction is the IDI transaction. What this means is that tests at the IDI interface of the System Fabric consist mainly of sending IDI transactions in interesting combinations to test conditions defined in the Test Plan (see [Testplan Writing](#)).

Section 6.2.1 below will describe in more detail how tests are built from sequences and sequence libraries. One common and easy way to write a test is to send many transactions parallel into the TE and let the TE decide what to do with them. This is where the sequencer comes into the picture; it takes transactions requested by tests and decides how to send them into the DUT. The sequencer

knows what the DUT is capable of and sends transactions to the DUT following microarchitecture rules such as:

- DUT can accept 2 transactions in parallel.
- After accepting transaction A with address X, the DUT cannot process another transaction to the same address for 2 cycles.

The sequencer is like a scheduler for instructions to the DUT, once it has picked the next instructions based on the microarchitecture rules coded into the sequencer, it sends the instructions to the driver. To keep the TE modular, the sequencer never talks directly to the DUT. It is the driver's job to convert the abstract transaction object from the sequencer into 1s and 0s that the DUT understands and actually drive the DUT signals in the correct timing.

6.1.1.3 Monitors

For maintenance and reuse reasons, most components of the TE should not interact directly with the DUT. In the IDI BFM example above, the driver and monitors are the only blocks that directly access the RTL signals in the DUT. Monitors disconnect other TE components from the design. The idea is that the other TE components will have higher probability of being reused in other similar systems if they have no dependency on the RTL signals of the DUT.

Monitors read RTL signals and package them into the base object type handled by the portion of the design. For the IDI BFM in the System Fabric TE example, the base object is an IDI transaction. The IDI transaction object should be the sole means of DUT related communication with other TE components. This means that if there are changes in the DUT, only the monitor needs to be changed instead of every component if they had all read RTL signals directly.

Within Intel we have this common definition of a monitor (which is also the UVM definition of a monitor), but we also have more specific classes of monitors called *Hardware Monitors* and *Software Monitors*.

6.1.1.3.1 Hardware Monitors

Intel HW monitors fulfill some of the responsibilities of a standard UVM monitor, but do it in a specific way. Intel HW monitors are JEM based monitor code embedded in RTL typically with a bind statement. The JEM HW Monitors do very little data transformation beyond packetizing data into transactions. These monitors can interact directly with a TE via a SW monitor, and they can also write their data to a binary trace to be accessed via SW Monitors or post-processing agents (like checkers) that access the JEM binary directly.

6.1.1.3.2 Software Monitors

A SW monitor works in conjunction with a HW monitor. The HW Monitor binds directly to the RTL and produces a transactions. The SW monitor takes these transactions and packetizes them in a way that works with SVTB/UVM and puts them on their appropriate UVM analysis ports.

6.1.1.4 Trackers

Trackers take transactions from HW monitor traces, do some amount of data transformation on the transactions, and record them into log files or into a trace database such as UTDB. See [Data Flow within the Val Environment](#) for more information on data transformations.

In some contexts, people will use the term tracker to mean the combination of the monitor and tracker together. People will also use the term *collector* for this as well.

6.1.1.5 Checkers

As their name suggests, checkers in the TE are components that check the correctness of the DUT as the simulation runs. Figure 37 shows separate blocks for BFM^s and checkers, however high level TE checkers usually need to model RTL functions in order to calculate expected RTL behavior to stimulus, so the line between BFM^s and checkers is often blurred or does not even exist. As mentioned in 6.1.1.3 above, checkers should not read RTL signal directly, instead they should get inputs from one or more monitors. This removes dependency between the checkers and the DUT and makes it possible to reuse high-level checkers in different systems. Just as importantly, TE coders should avoid unnecessary dependencies. For example, a lot of the passive collateral such as checkers *could* rely on the IDI BFM... but should not, to help disentangle the overall test environment.

For more information on checkers, see [Checking](#).

6.1.1.6 Loggers

Loggers are TE components that take information from multiple sources and send it to log files. Informative log files allow Validators to have a high-level view of what is happening in the test at the time of failure, come up with theories for the failure, and debug more efficiently. Some of the common sources of log information are:

- Monitors located at the primary interface of the DUT. In the System Fabric example in Figure 37, there are log files created at the IDI, IOSF and memory interface. These monitors log the transaction information going into and out of the System Fabric RTL.
- Another useful type of information to log are traces from checkers, or the information generated by the checkers as they run. These logs show what and how the checker expects RTL to behave to inputs and transactions. This information can be used by Validators to understand the failure and come up with failure theories.

A *tracker* (see 6.1.1.4) is a kind of logger, but trackers also do data transformation and write to both logs and UTDBs.

6.2 Test API

The test API provided by the TE serves as the Validator's window into the test environment and the DUT. The test API provides a transaction level access to the DUT. This transaction level access

hides low-level details like actual signal names and signal timing information from the Validator, and makes it easier to write tests.

In the System Fabric IDI example, driving an IDI transaction into RTL involves wiggling multiple signals over a period of a few clock cycles. If the transaction is a read type transaction that is expected to return data, the data will show up at the System Fabric interface sometime after the transaction is driven into the DUT. The test API hides all these details from the Validator and provides a high-level logical access to the DUT, in our example this includes an IDI object used at the IDI/System Fabric interface used by the checker, monitor and other TE components. In addition, the same object is also used as the main test interface object. The IDI object contains all the fields like opcode, address and other information needed by the TE to drive an IDI transaction into the System Fabric. To validate the DUT, Validators write tests that consist of multiple IDI transactions in interesting combinations. It is the TE's job to know how to send these IDI objects into the DUT.

In addition to the base transaction objects, the test API usually also includes helper functions to simplify tasks that are frequently done in tests. Some examples are:

1. Functions to read/write control registers and fuses.
2. Functions to control DUT power features.
3. Functions to simplify DFX features like tap, to decode serial data into structures.

6.2.1 Sequence and Sequence Library

In its most basic form, a sequence is one instance of the base object sent to the sequencer, which will then drive the sequence into the DUT through the driver as mentioned in 0 above. Using the IDI interface in the System Fabric example, the base sequence at the IDI interface is one IDI transaction. Sequences that are more complicated include multiple IDI transactions sent in interesting combinations. Some examples of sequences at the IDI interface are:

1. Sending 1 random IDI transaction.
2. Sending 2 to 100 IDI transactions serially, keeping one field in all transactions constant, and randomizing all the others.
3. Sending 2 to 100 IDI transactions serially, randomizing one field in all transactions, and keeping all the others constant.
4. Sending the transactions described in 2 or 3 in parallel instead of serial.
5. Sending a write transaction to a random address X, followed by a read to the same address.

To validate the DUT, Validators will write multiple sequences, using the DUT Test Plan as a guide to know what interesting conditions should be covered. These sequences are then added to the TE to form a sequence library. The current best-known method (BKM) in creating a random test generator is simply to pick some number of sequences from the library and run them. The number of sequences to pick is determined by how long an average test is supposed to run. For more information about test and sequences, see [Stimulus](#).

6.2.2 Test Flow and Test Run Phases

Related to the test API is TE's role in dividing the simulation time into multiple phases, these phases are usually defined by validation methodologies like UVM and Saola. The following is a summary of what happens after the Validator has entered test cmd line in terminal. It describes how control is passed from the run tool to the TE to the test. The list below is in time order, step 1-3 are all done at time=0, simulation time moves forward starting at step 4.

1. The run tool executes the simulator (i.e. VCS) which in turn instantiates RTL and TE.
2. TE component objects are created; this includes TE components like monitors, checkers and sequencers.
3. TE's model of the RTL registers are created and values are assigned, some registers may get random values. Internal states of recently created TE components are initialized or randomized.
4. TE runs its reset sequence,
 - 4.1. Power signal asserted, thus powering up RTL.
 - 4.2. Reset signal asserted, clearing out X's from RTL.
 - 4.3. If the clock generator (PLL) is outside of the DUT, the TE will generate the clk.
 - 4.4. Sometime later, reset is de-asserted and the reset sequence is finished.
 - 4.5. TE program RTL registers with values generated in step 3 above.
5. Control is passed back to the test.
6. The test starts generating sequences and sends them to the sequencer.
7. The test sends the last sequence to the sequencer.
8. TE waits for sequencer to drain, and for all sequences to complete.
9. Last cycle of simulation and TE does a final check.
10. TE components are de-allocated and simulation is shut down.

6.3 Trace-Based Validation (aka Post-Processing)

The term "Trace Based Validation" refers to moving from the traditional "online" validation during a test run, which is typically slow, very much dependent on low-level RTL details, harder to maintain and reuse, to fast and efficient post-processing based techniques, in which the validation collateral use log files and other recorded information from the test runs (commonly called "traces") to do the validation logic independently of the RTL test execution. Such post-processing techniques allow super fast turn-around-time of validation collateral execution and fixes, simpler reuse across platforms and integration levels, as well as higher independence of val collateral from low-level RTL details. This, in turn, contributes to significantly higher validator productivity and quality of validation collateral.

Trace-based validation works on *passive* validation collateral. Trackers, checkers, coverage monitors. These items are considered passive because they only consume data when running as part of the TE, they do not produce anything that is used to change the stimulus of the test. Said another way: if these parts of the TE were removed from the TE, a test would still act the same (but errors might not be flagged and coverage might not be tallied).

6.3.1 Styles of Post-Processing

There are two primary styles of trace-based execution: replay post-processing and exclusive post-processing.

6.3.1.1 Replay Post-Processing

This style of trace-based execution leaves the passive components within the TE and executes those components inline as part of a simulation. This means that the trackers, checkers, and coverage monitors are running right along with the simulation and executing cycle-for-cycle with it.

When run within the simulation, the passive components are simply part of the overall simulation. When run in post-processing, the executable can consist of *all* of the passive components of the DUT, or separate executables can be created for each passive collateral. Each has its advantages and disadvantages.

There are some limitations to how validation collateral must be coded for this style. For instance, a checker coded to run via replay post-processing must adhere to the rules of normal inline checkers. For example, it cannot ‘scan the entire test trace’ for something, since the checking must be done as things happen during the test. Checkers of these type typically take items from analysis ports, check the items versus any built up state within the checker, and then update their state.

6.3.1.2 Exclusive Post-Processing

Exclusive post-processing is a design choice where passive components are never built as part of the TE and never run inline as part of a simulation. They are run only against traces created from simulation.

The advantages of exclusive post-processing are the removal (and dis-entanglement) of parts of the TE, and the ability to code the post-processing executable in whatever language and framework the user chooses.

6.3.2 Post-Processing Toolset

When large portions of the TE move to post-processing, it means that a significant infrastructure must exist to find and run all of the components and to collect data from them (for example, collecting errors from checkers). There is an entire stack of tools that is devoted to post-processing:

JEM, UTDB, Opera, PyRal, PyFal, Griffin...

More info can be found at: <https://goto.intel.com/tbvs>

6.4 Reuse

Test environment are usually large, complicated pieces of software and, like all large software projects, we try to make the TE modular to reuse pieces of code in order to reduce overall project effort. In general, there are two ways we reuse TE code.

6.4.1 Vertical Reuse

Vertical reuse is the reuse of a TE component when it is integrated into a larger block. The most obvious example is when an IP gets integrated into the SOC. If part or all of the IP's TE also gets integrated into the SOC, then those TE components are vertically reused in the SOC. The same TE code is used at the IP level and is reused at the SOC level.

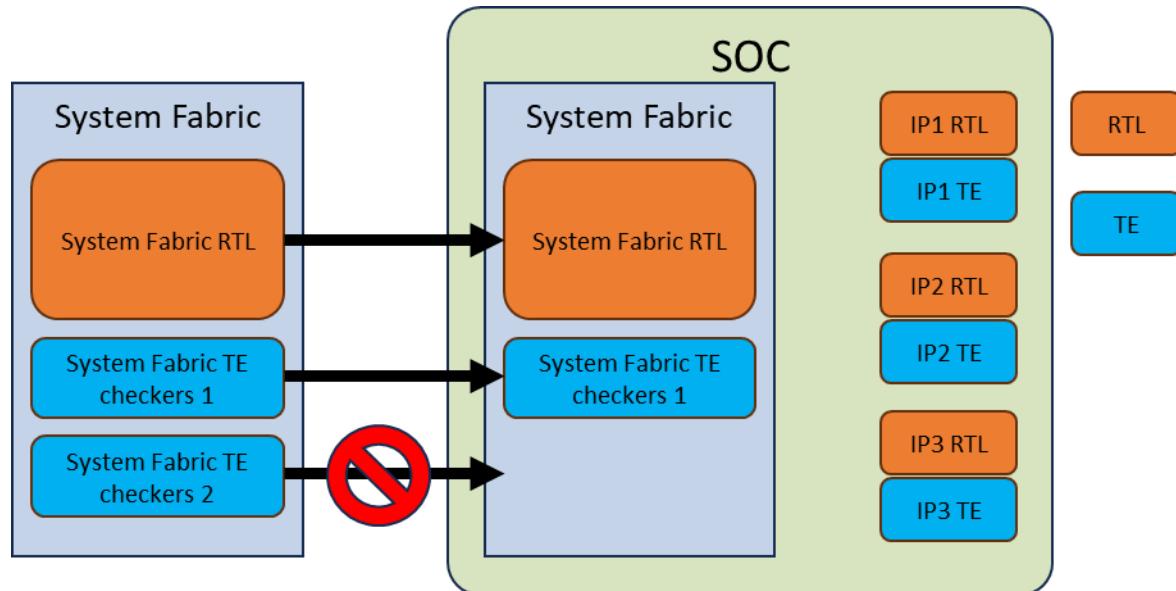


Figure 38 Vertical reuse of part of System Fabric TE from cluster to SOC level.

6.4.2 Horizontal Reuse

Horizontal reuse is reuse of a TE component within multiple independent, non-overlapping blocks. For example, if you created a verification component (VC) for creating table log files and this VC is used in multiple different SOC projects, this is horizontal reuse. Some other examples of horizontal reuse are:

- Framework VCs like UVM and Saola
- Interface/protocol VCs like IOSF primary VC, IOSF sideband VC and IDI VC.
- Utility VCs like table log formatter.

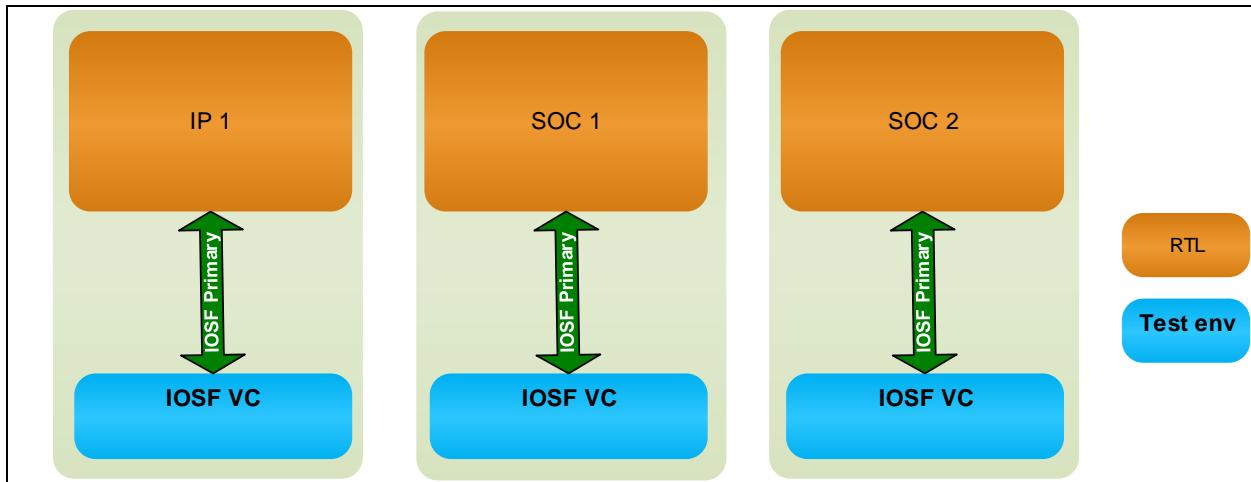


Figure 39 Horizontal reuse of the IOSF VC in different SOC and IP projects.

6.4.3 Pros of Reuse

When a solution is developed once and used multiple times, it reduces the overall effort both for initial development as well as for enhancement and bug fixes. This is the main motivation for reuse, to reduce effort, when code is developed once and reused vertically and/or horizontally.

Reuse can also drive standardization, which indirectly reduces effort. Once such example is the use of framework VCs like UVM and Saola. Aspects of this standardization may not directly reduce total effort when implementing a given module, but rather give a common framework. However, benefit can be seen when standardization indirectly reduces effort by

1. Reducing the reuse overhead. For example, standardization through UVM and Saola makes it cheaper and easier to reuse an IP's test bench collateral in SOC.
2. Making it easier for engineers to move between areas and ramp more quickly.

6.4.4 Cons of Reuse

The pros of reuse can be significant and are often relatively simple to understand. However, there are also cons to reuse that can be more subtle but potentially just as significant. Properly understanding the cons when making a reuse ROI decision takes more careful thought and requires better experience.

Most reuse cons relate to increased complexity and dependencies, and the artifacts of dealing (or not dealing) with them. “Simulation” performance impact is another concern.

This cons section is larger than the pros because it requires more explanation, not because the cons are typically larger than the pros.

6.4.4.1 Cost Due to Complexity of Reuse

At the root of it, code reused across multiple environments is more complex because of the need to work in more situations than if not reused. A big part of handling this comes down to proper encapsulation of the code. Encapsulation helps in two key ways:

- a) Limits expectations the reused component may have on the enclosing environment and vice versa.
- b) Limits details of the reused component exposed to the enclosing environment and vice versa

Too little encapsulation can lead to expectations that cannot be [easily] met, hard to maintain interfaces, and increased chance of conflicts in full environment. However too much encapsulation can increase effort by eliminating simplifying expectations or by limiting usage flexibility too much.

6.4.4.1.1 Expertise Cost

Striking the right encapsulation balance and developing the infrastructure to support that requires real effort by the VC developer, and maybe more importantly the proper expertise. This expertise comes in the form of strong software skills and a strong grasp of the reuse needs and challenges for the particular VC. Methodologies and framework VCs like UVM greatly help here, but they do not solve the problem.

Reuse of a VC without proper enabling effort and expertise from the VC development will cause long-term effort for both the developer and the user of the code. This is something both the developer and consumer of the VC should consider.

6.4.4.1.2 Undesired Reuse Cost

One way to reduce the integration cost of multiple components is to encapsulate them in a larger component/VC which then allows reuse of the connections between these components. However, this can cause the user to reuse components they otherwise would not, which can increase the enabling effort, and hurt performance and stability.

6.4.4.2 Cost due to Increased Dependencies

Reuse causes a component to be used in more environments increasing dependencies on it, and thus between the teams that develop and use those testbench components. These increased dependencies tend to amplify the impact of quality issues in a reused VC and the support cost associated with the VC.

Increased reuse of a component also increases the probability of having to support multiple versions of the component being reused, and having to maintain backward compatibility support for deprecated features, thus increasing maintenance cost.

6.5 Qualities of a Good TE Coder

Test environments are large and complicated software projects, therefore a TE coder needs to have good software engineering skills. However, TE's also interface with the DUT and, in many cases, emulate missing hardware in cluster level models. Therefore, a good TE coder also needs to have strong hardware skills. Below is a list of some of the important quality and skills of a good TE coder:

- Have strong computer science and software engineering skills, like understanding object oriented programming, writing clean code and using techniques like test-driven development.
- Be a microarchitecture expert in the design being tested by the TE, the expertise is needed in order to be able to write checkers.
 - o Understand the microarchitecture, but not the actual RTL implementation details – There is a risk of coding the same bugs in checkers if TE coders look at the RTL too closely.
- Be proficient in the microarchitecture of areas around the design being tested if the TE owner needs to write BFM's to mimic the missing pieces.

7 Summary

Test Environments are typically the greatest cost within Validation outside of debug. They require intelligent planning and good software practices to get right, and can be extremely costly and a hindrance to a project when done wrong. In the world of SOC it is important to plan ahead for TE component reuse and then code in such a way to streamline the reuse effort.

A great TE can make an otherwise-poor validation effort look good; a great team with a poor TE is likely to suffer late milestones and low validation quality.

8 Future Work

8.1 Handling mid simulation reset and power state changes

- Add something about checkers handling reset
- Include power management handling as well as parts of the design losing power and how that is handled by the TE)

8.2 Workarounds

1. Do and do not on how to add workarounds to TE.
2. Risk of workaround not being removed

8.3 TE Cost

- TE code quality

- TDD/unit test, code review
- Another large piece of software to develop and maintain in addition to the RTL.
- TE usually around $\frac{1}{4}$ to $\frac{1}{2}$ the number of lines of code compared to RTL.
- Large and complicated, cite historical TE vs RTL bug ratio.
- How validation team is organized with regards to having a dedicated TE owner vs distributed TE responsibilities.

8.4 TE Documentation

- What to document:
 - Directory structure
 - the test API
 - Sample tests
 - Hints on how to debug the TE
- Where to put the documents?

8.5 TE Debug-ability

TE coders need to think about inevitable failures and make sure that information is easily available to Validators for debug.

- Use error messages that are easy to understand and have enough information for a Validator to take the next step.
- Leave a trail of TE/checker internal state in debug messages that can easily be enabled/disabled.

8.6 TE simulation performance and memory usage

- Periodic CPU usage and memory profile of TE code.
- Gating tests in model to detect sudden change in model performance and memory usage.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 12

Data Flow within the Val Environment

By: [Michael Bair](#)

1 Abstract

This chapter describes data flow and transformations within the validation environment. Data is created/sampled at one point, then transferred from component to component within the tesbench and beyond. The data is also transformed: it becomes encoded, combined, abstracted, and filtered. It may be saved/recorded at multiple points along the way. The purpose of the transformations are to bring the data to the best possible modeling and precision level needed by any consumer.

Data transformations can be complex and can be a source of bugs within the validation environment. The greater the complexity of a given transformation, the tougher it becomes for engineers to ‘debug through’ the transformation.

It is critical to plan how data flows and is transformed within a validation environment. Not transforming to the needs of the components using the data makes for inefficient and complex components. Having too many transformations, or too complex of transformations, leads to a convoluted validation environment that becomes a ramp/training nightmare.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	2/18/2024	Initial Version	Michael Bair	Neriya Bar-Levav; Jim Huggins

3 Contents

1 Abstract.....	299
2 Chapter Revision History	299
3 Contents.....	300
4 Purpose.....	301
4.1 Why do we need this chapter?.....	301
4.2 What does this chapter cover?	301
4.3 What does this chapter not cover?	301
5 Background Concepts	301
5.1 Test Environments (TE), and Test Benches (TB)	301
5.2 DUT	301
6 Data Flow within the Val Environment	302
6.1 What is <i>Data</i>	302
6.2 Data Sources	302
6.2.1 Hardware Monitors	302
6.3 Modeling Level and Precision	303
6.4 Abstraction and Transformation of DUT signals	304
6.4.1 Data Transforms	305
6.4.2 Example Transforms.....	305
6.4.3 When to Transform DUT Data	307
6.4.4 Which VC should do the DUT Response Manipulation?.....	307
6.4.5 Considerations when doing data transformations.	308
6.5 Utilizing logging of intermediate steps	309
7 Summary.....	309
8 Future Work.....	309
9 Bibliography	310

4 Purpose

4.1 Why do we need this chapter?

Data sourcing, transferring, transforming, and consuming are critical aspects of validation environments that are often overlooked when creating/updating our val environments. A validation environment with well-thought-out data flows is a boon to the user. Conversely, one with too many data transforms, or ones that are too complex, or spread out within many components so that engineers cannot trust the meaning of data... well, working in such an environment is much like walking in mud or carrying an anchor: any task will take greater effort and latency.

4.2 What does this chapter cover?

This chapter covers how data flows through validation environments. This includes the sources of data, common transformations, and common usages. It describes common attributes of data; ie, how it might be measured and discussed. It will discuss some of the common tools used in data flows.

4.3 What does this chapter not cover?

This chapter does not cover all the possible data transforms that exist, nor all the usages. It does not cover all data tools.

5 Background Concepts

5.1 Test Environments (TE), and Test Benches (TB)

The Test Environment is all of the validation collateral built into an executable model. Testbench is a generally accepted synonym, although some methodologies draw a distinction between the two. Whereas the Test Environment may include not only the test bench, but also additional validation code embedded in the RTL, as well other code used outside the executable model via post-process. The Test Environment may also be written to run standalone with other Testbenches (so they can test each other and test protocols). See [Test Environments](#).

5.2 DUT

Device Under Test. The part of the validation system that eventually becomes a product, potentially in addition to a small amount of non-synthesizable support RTL to support connecting to test environments. Typically speaking, this is the functionality described by the RTL model. One could envision systems where a testbench is wrapped around a piece of TE collateral; in that case, the DUT is the TE.

6 Data Flow within the Val Environment

6.1 What is Data

Data refers to the information that is generated, processed, and consumed during the testing of Pre-Silicon HW simulation and emulation models. This data can take various forms, such as the input signals fed into the hardware model, the output signals produced by the model, and the internal state of the model during the testing process. It includes all of the information passing between components within the TE as well. The data is used to evaluate the correctness and performance of the hardware model as well as the coverage hit by the stimulus.

6.2 Data Sources

The primary sources of data in a hardware test environment are *hardware monitors* and TE components. Monitors are used to observe and record the behavior of the hardware model during testing, generating data about the DUT's inputs, outputs, and internal state. TE components, on the other hand, are responsible for controlling the testing process, providing the input signals to the model, and interpreting the output signals. Common TE components include emulators, sequencers, drivers, monitors, trackers, checkers, and loggers (see [Test Environments section: Components of a test environment](#)).

6.2.1 Hardware Monitors

Hardware monitors are essential components in a hardware test environment. They are used to observe and record the behavior of the hardware model during testing. This includes tracking the input and output signals, as well as the internal state of the DUT.

Monitors are typically implemented as separate modules that are connected to the DUT's interfaces. They continuously sample the DUT's signals and record the data or transfer the relevant data to downstream components for consumption (or both).

Monitors can be designed to track a wide range of data, depending on the specific requirements of the test environment. For example, they can monitor simple signal values and transfer/record them as-is, or they can transform the signal-level data into complex transactions or high-level behaviors/states of the DUT.

The amount of transformation done within a hardware monitor is a function of multiple factors:

- 1) The requirements of downstream data consumers
 - a. If the downstream consumers need complete signal-level data... then transforming to a higher level of abstraction will not fulfill their needs!
- 2) Performance
 - a. In validation platforms such as emulation, both the compute time and the output of a hardware monitor are constrained, therefore HW monitors used in emulation typically to a basic level of data abstraction and consolidation such that their output is at a transaction level, but not further transforming beyond that.

- b. Recording and storing data at a signal-level and cycle-level granularity requires far more disk and compute than doing so for abstracted data saved on transaction-level granularity

Section 6.4.5 will re-iterate some of the performance concerns when it comes to transformations within HW monitors. See [Test Environments section: Monitors](#) and [Test Environments section: Trackers](#) for more discussion on monitors within the validation environments.

6.3 Modeling Level and Precision

Modeling Level and *Precision* are two categories for measuring components within our testbenches.

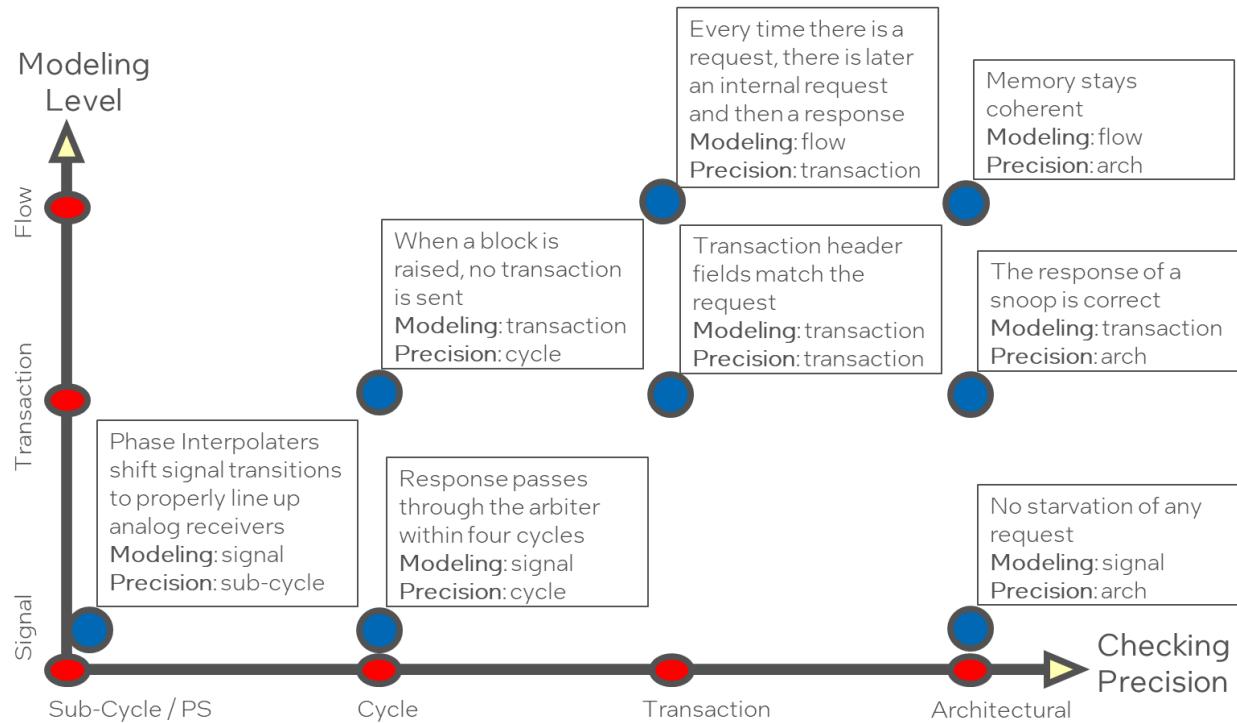
Modeling level defines the abstraction level of the objects being worked on. Typical modeling levels are signal, transaction, and flow levels. These definitions are somewhat loose: signals can range from single-bit signals to objects and contain many fields with enumerated types. Transactions can be single-cycle ‘flits’ or packets that span multiple cycles.

Precision (sometimes called ‘accuracy’) defines the time-granularity of a component such as a checker or coverage. Precision ranges can range from sub-cycle (picoseconds) to architectural level.

The figure below includes various checker examples plotted by their modeling level and their precision. Many checkers have a natural ‘best precision’ level that balances bug finding with complexity and maintenance costs. Driving a checker to finer precision can be valuable in finding lower-level bugs sooner... but can be costly in maintenance and produce more false failures. Components also have a natural ‘best modeling level’; this is often found in the description of the component itself: if the description describes signals, then it probably needs signal-level modeling. If it describes transactions, it is probably best that the component consumes transaction-level inputs. Think of this as a rule:

The data consumed by a TE component such as a checker or coverage monitor should already be transformed to approximately the same modeling level and precision as the definition of the checker rule or coverage.

Components should not cross modeling level or precision boundaries. Trying to check both cycle precision and transaction precision within the same checker is painful. The same is true if you try utilizing signal and transaction level modeling all within the same checker. If you have checking rules that are of two different modeling levels or two different precisions, those rules probably belong in two different checkers.



6.4 Abstraction and Transformation of DUT signals

DUT inputs, outputs, and internal state are typically in the form of RTL *signals*. These are single-bit signals and multi-bit busses, though occasionally the RTL is coded with higher levels of abstraction where an entire transaction record is coded into an object with multiple parts (opcode, address, valid bit, etc) while also encoding some of those parts using enumerated types.

As mentioned in section **Error! Reference source not found.**, some types of checkers work best with signal-level data. A simple Req-Ack protocol would be such a case. For other checks, such as read request data checkers, it is much more convenient to utilize the DUT information *after* it has been transformed to a higher modeling level (aka a higher level of abstraction). While data transformation may have downsides, it is very valuable for checkers to consume data at the same modeling level utilized in the rule definition. This tends to create stable and accurate checkers, prevents checking holes, and lowers the hardship in matching the failure to a specific spec-defined rule. The same is true for coverage monitors: if the coverage conditions are at a signal level, then it makes sense to utilize signal inputs. But if the coverage conditions are defined at the level of a transaction or flow, it is useful to have the data transformed to that level prior to entering the coverage monitor.

Put simply: if a component requires anything more than a minimum amount of data transformation, it is best to do that transformation *outside* of the consuming component so that the transformation of the data and the consumption of the data are kept separate. A new component should be created that does the transformation.

The following sections explain the concepts of abstracting to higher modeling levels. The term ‘data’ will be used to mean any level of information from signals up to flows.

6.4.1 Data Transforms

The data collected from the DUT interfaces may go thru several transformations, such as:

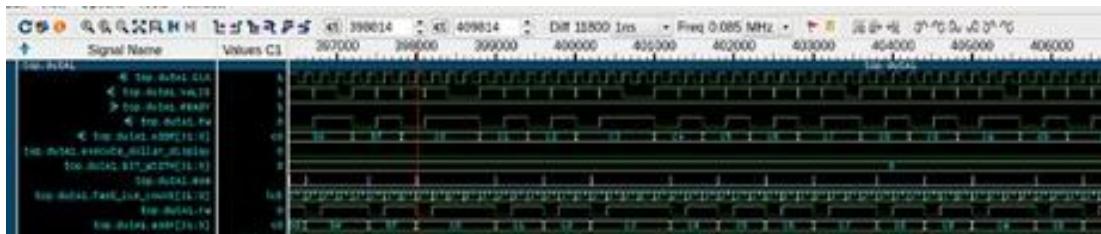
- Refinement, i.e.: splitting DUT response information to fields. Enumerating the values, etc.
 - Elevation: Converting low level information to more abstract information, using configuration information or other inputs. e.g.: decoding address according to the DUT configuration.
 - Re-Packaging, i.e.: converting one or more DUT response to one or more data items. E.g.: connecting response with request or splitting a network packet to all the piggy-backed packets. Linking together numerous transactions that together form a ‘flow’.
 - Filtering, i.e.: ignoring DUT responses. For example, ignoring DUT response when the reset has not finished.
 - Intra-flow cross-interface analysis: Finding relationships between transactions occurring on multiple interfaces where the transactions are really, and feeding all of that information back to a central flow for those transactions
 - Cross-Flow analysis: Finding connections between different flows of transactions; for example, flow A hit a resource conflict with flow B.

The data may go through these transformations in any order and multiple times.

6.4.2 Example Transforms

This section leads you through an example where 3 levels of processing are done to transform signal traces to higher levels of abstraction and to pull data together from multiple places.

- ## 1. Signals: the raw data as seen in the simulator and signal traces



- ## 2. Basic Interface log

The signals have been packed together to form packets, even using data from multiple cycles. Enumerated types have been utilized. Data now appears as a transaction.

Time: 33045000 ps

Packet Type: REO

Opcode: Read32H

Address: 0x45600

Ordered: True

Blocking: False

The transformation from 1 to 2 is typically done in a place called a *tracker* where raw signals are pulled together, opcodes are named, and packed data is pulled apart into meaningful abstract values (like “Ordered” in the list above).

3. Compilation of multiple transactions into a flow

The transaction packet from #2 has been combined with other transaction packets to for a flow of operations on this interface. You can see the initial read request operation merged together with the data returns and complete response. Note that the transforming tool also have this flow a unique identifier (Flow119) for possible use in other tranformations.

Start Time: 33045000 ps

End Time: 33079000 ps

Flow Identifier: Flow119

Flow Type: RequestRspData

Req Opcode: Read16H

Address: 0x45600

Ordered: True

Blocking: False

Ordering Conflict: True

Rsp Opcode: RspData

Data: 08923419 67678172 99722901 48366171 98351836

Data Rsp Times: [33064000 33065000 33066000 33067000]

Completion Rsp Time: 33079000

4. Adding cross-flow information

A transform algorithm runs to find places where different flows are interacting with one another. Note that the ‘Ordering Conflict’ now has a ‘Conflicting Flow’ added to it. The algorithm must have looked across all the flows and determined which flows and conflicts with others.

Start Time: 33045000 ps

End Time: 33079000 ps

Flow Identifier: Flow119

Flow Type: RequestRspData

Req Opcode: Read16H
Address: 0x45600
Ordered: True
Blocking: False
Ordering Conflict: True
Conflicting Flow: Flow116
Rsp Opcode: RspData
Data: 08923419 67678172 99722901 48366171 98351836
Data Rsp Times: [33064000 33065000 33066000 33067000]
Completion Rsp Time: 33079000

6.4.3 When to Transform DUT Data

Data transformation can be done when collecting the data, or when we have a checking event, or many places in-between.

If the transform is not dependent of transient conditions, e.g.: splitting bits to fields, and enumeration of values, it should be done in the sampling point. However, when the transform result may change according to changes in the design inputs and state, it is better to perform the transform as late as possible, thus allowing more collection of information on DUT state and transitions leading up to the checking point.

For example, consider a serial protocol such as IOSF. Taking the interface packets and extracting the opcodes, address and data from the serial interface should be done at the sampling point. but decoding if the access should or should not be blocked (for example, due to SAI violation) should be done at the checking point, when the final values of the related configuration are known.

On the other hand, data manipulation done at the sample point does need to take into account performance impact as well as reuse feasibility. Performance is impacted both by how much work a sample point does but also how much data it sends out and at what frequency. Delaying work until later process might be a good trade off if such processing isn't always needed. Sending too much data, too often, can also degrade performance by too much I/O from the model (which could even stall execution to empty I/O buffers). For a quick “rule of thumb”, a single DPI call in emulation can handle ~350 bits before needing multiple messages to the host machine. In terms of feasibility, if the sample point code is to be shared from simulation to emulation (can often be the case, even for IP teams sharing val code with SoC), such code needs to be restricted to synthesizable verilog language only.

Selecting when to do the transform will affect which VC will do the transform – see the section 6.4.4.

6.4.4 Which VC should do the DUT Response Manipulation?

As a rule, a basic level of manipulation should be done in the monitor/collector VC so that we are not saving/transferring signal-level cycle-level data, which can be compute and disk intensive. After a basic level of abstraction done in the HW monitor where should further data transforms take place? Several considerations may affect the answer.

- When should the manipulation be done, when the response is collected, or at checking event.
Manipulation in the sampling point can be done in the monitor, but manipulation at checking point can only be done in the checkers. According to up-to-date information.
- Who is the consumer of the information? (Single consumer or many)
Many consumers may mean different manipulation per consumer, and perhaps different portion of the data, e.g.: one Checker check that each DUT response has legal CRC. And wish to get each packet as is, while another checker checks the content of packet, and may need to combine several packets together, this apply that the different needs.
In this case, it is better to send the response with only the common manipulation and allow each consumer to perform the desired manipulation.

Usually, a combination of both techniques can achieve the best results.

6.4.5 Considerations when doing data transformations.

Data transformations may lead to several types of TE issues.

- a. The data transformation code itself may be buggy
 - a. This is certainly true if the code is complex; the risk goes up dramatically with the code size and the number of special cases that are coded in.
- b. The more transformation done, the harder it is to debug a data problem within a failing component back through complex data transformation back to its source. Debug is made harder for several reasons:
 - a. For checkers, data transforms may move the checking point far away from the point where the DUT response was erroneous. The debugger may have to struggle through many levels of code where data is ‘put together’
 - b. Data transformation may lead to very high-level data items that can be very different than the originating DUT signals and states. The debugger may be left trying to figure out “*where did this come from?*”
 - c. The transformation may tie together many disparate data flows (for instance, data from two ports) into a higher-level flow. When a component consumes this higher-level flow and detects a problem, it will be hard to pinpoint which data stream was erroneous or whether the combining transform created the problem.
- c. Performance:
 - a. If too little transformation is done during the hardware monitoring stage, the output from the monitor could be as large/bulky as a signal trace. This can slow down both simulation and emulation runs

- b. If too much transformation is done within the hardware monitor, it could also slow down the run, especially in the world of emulation
- c. The best rule of thumb: do a basic level of abstraction in your HW monitor such that you are packing together basic transactions and such that you are not recording transactions every cycle, but don't do anything more complicated than that. Leave further transformation to another processing component.

These problems do not change the fact that data transforms are necessary; as previously mentioned TE agents should consume data at the same modeling level and precision at which they plan to operate. This is simply to awaken you to the fact that data transforms can be a source of problems, and that care should be taken how and where you do them. Moreover, there are many debug capabilities and logging techniques that can solve all the above disadvantages. An important technique is discussed in 6.5.

6.5 Utilizing logging of intermediate steps

As mentioned previously, the more manipulation that is done on a dataset, the more likely there will be bugs in that transformation and the harder it will be to debug the data processing. One helpful methodology is to separate the processing steps and log intermediate stages. This is good coding practice in general (separation of responsibilities inside your processing code), and leaving behind logs of the outcome of each intermediate step helps dramatically in narrowing down *where things went wrong*.

Further, if your data processing is happening in post-processing, it is easy to create your individual processing steps as individual scripts, and easy to modify and test them.

Humans very quickly become comfortable reading standard log files where every line is a new ‘entry’ and columns are fields. Having intermediate printouts of your processed data, in logfile format, can go a long ways towards helping engineers understand and debug your process.

7 Summary

This chapter discusses the concept of data transformation within a hardware test environment, focusing on how data from DUT interfaces undergoes various transformations to enhance its usability for TE components. These transformations include refinement, elevation, re-packing, filtering, intra-flow cross-interface analysis, and cross-flow analysis. When and where these transformations should occur is critical when planning a TE. Components that transform data should remain separate from the components that create or consume data... as much as possible.

Understanding data transforms and getting them right in terms of placement, performance, simplicity, and capability is critical to a performant validation environment.

8 Future Work

This section left intentionally blank.

9 Bibliography

This section left intentionally blank.

The Art of Pre-Si Val: Chapter 13

Testplan Writing

By: [Kalpana Kothapally](#), [Michael Smith](#) and [Michael Bair](#)

1 Abstract

A written testplan codifies the validation Plan of Record by describing the stimulus, checking and coverage used to validate the design. Testplans are composed of two major parts: background information and testplan entries. Background information outlines the validation strategy and establishes the context of validation. Testplan entries describe specific validation objectives in terms of test scenarios, coverage events and checking. A testplan is a living document that acts as a blueprint for all validation activities and is used to measure validation progress during execution. This document provides instructions and insight for testplan writing, maintenance and reuse.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/01/2003	First draft of the document	Kalpana Kothapally, Ghani Kanawati Neeta Ganguly Justin Chien Alon Gluska	Michael S. Bair DPG uAV Methodology WG IAG uAV Methodology WG
1.1	01/15/2004	EITVOX format of the document	Kalpana Kothapally	Michael S. Bair DPG uAV Methodology WG
1.2	02/24/2004	Editing completed	Kalpana Kothapally	Paul Schwabe
1.3	10/21/2004	Removed EITVOX topic to a separate document	Kalpana Kothapally	Steve Cegla
2.0	2/01/2007	Added new material for test case style, identifying test cases, and creating outlines.	Michael Bair	Matt Kupperman DPG uAV Methodology WG
3.0	6/01/2016	Major content changes to reflect shift away from depth and coverage-driven testplan methodology and towards SoC testplan methodologies.	Michael Smith	Michael Bair

3 Contents

1 Abstract.....	311
2 Revision History.....	312
3 Contents.....	313
4 Purpose.....	316
4.1 What is covered in this document?	316
4.2 What is not covered in this document?	316
5 Background Concepts.....	316
6 Testplan Writing	316
6.1 Why Write Testplans?.....	316
6.1.1 Testplans and Execution.....	317
6.1.2 Testplans and Communication	317
6.1.3 Testplans and Project Management	317
6.1.4 Testplans and Learning	317
6.1.5 Testplans as a Reference	318
6.2 Major Components of a Testplan.....	318
6.2.1 Background Information.....	318
6.2.1.1 Scope	318
6.2.1.2 Feature Description	319
6.2.1.3 Validation Strategy	319
6.2.1.4 Test Environment	319
6.2.1.5 Review Notes and Action Items.....	320
6.2.2 Testplan Entries	320
6.2.2.1 Testplan Entry Types.....	320
6.2.2.2 Testplan Entry Fields.....	321
6.2.2.3 Organization of Testplan Entries	325
6.3 Creating Testplan Entries	325
6.3.1 Prioritize Top-Down Requirements	325
6.3.2 Validate the Interfaces of the Design	326
6.3.3 Think in terms of Hierarchy and Scope	326
6.3.4 Use a Structure-Oriented Approach for New Logic.....	326
6.3.5 Use Point-of-View and Setup Analysis to Identify Test Scenarios	327

6.3.6	Include Reset and Power Management	328
6.3.7	Include other Globals	328
6.3.8	Include Configuration	328
6.3.9	Include Stress Entries	328
6.3.10	Include Checking Entries	329
6.3.11	Think, Reflect and Ponder	329
6.3.12	Discipline-based Techniques	329
6.3.13	Using HSD Bugs/ECOs/Issues	329
6.3.14	Learn from Others	329
6.3.15	Learn from Predecessors and the Prior Art	330
6.3.16	General Advice for Writing Testplan Entries	330
6.3.16.1	Get Started	330
6.3.16.2	Follow the Law of Testplan Writing	330
6.3.16.3	Write testplan entries using the same style across the project	331
6.3.16.4	Use the correct type	332
6.3.16.5	Breadth vs. Depth	332
6.3.16.6	Features vs. Implementation	332
6.3.16.7	Do not over-engineer test cases	333
6.3.16.8	Avoid creating too few or too many entries in a testplan	333
6.3.16.9	Avoid referencing signal names	333
6.3.17	Limitations of these methods	333
6.4	Testplan Lifecycle	334
6.4.1	Tasks and Timelines	334
6.4.1.1	Drafting	334
6.4.1.2	10% Review	334
6.4.1.3	Testplan Review	334
6.4.1.4	Testplan Exit Reviews	335
6.4.2	Testplan Maintenance	335
6.4.3	Testplan Sharing and Reuse	336
6.4.3.1	Testplan Sharing	336
6.4.3.2	Testplan Copy Reuse	336
6.4.3.3	Testplan Incremental Reuse	336
6.4.3.4	Testplan Hierarchical Reuse	336

6.4.3.5	Testplan Pre-Silicon / Post-Silicon Reuse	337
6.4.3.6	Testplan Multi-Stepping / Multi-Die Reuse	338
6.4.3.7	Testplan Mixed Reuse.....	338
6.5	Testplan Tools	338
6.5.1	Selecting and Deploying a Testplan Tool.....	338
6.5.2	Lightweight Options	339
6.5.3	Advanced Options	340
7	Summary.....	340
8	Future Work.....	341
9	References.....	341

4 Purpose

The purpose of this chapter is to introduce Validators to testplans and best-known practices of testplan writing.

4.1 What is covered in this document?

Testplan contents and guidelines for drafting a testplan are described in detail. Testplan reviews, maintenance and reuse are also discussed.

4.2 What is not covered in this document?

Specific tools used to generate, record, draft, reuse or share testplans are not covered by this document.

5 Background Concepts

It is assumed that readers are familiar with the planning stages set forth in the [Validation Planning](#) chapter. Additionally, writing testplans requires a solid understanding of validation concepts described in other chapters. Readers are encouraged to refer to other chapters as needed.

6 Testplan Writing

One of the main objectives of [Validation Planning](#) is to create a documented Plan of Record (POR) that outlines validation objectives and directs validation activities during the execution of a project. This documented POR is called a validation testplan. A validation testplan provides guidance for execution throughout a project and establishes a reference point for progress, visibility, review and feedback.

The ultimate objective of Validation is design health, and the process of creating and following a written plan will drive the behaviors and activities that lead to a healthy design. This chapter discusses the motivation and process of creating and following a written testplan.

6.1 Why Write Testplans?

Setting aside time to write testplans takes effort and comes at the cost of time that could be spent working directly to improve the design. Validators may not always recognize the value of capturing validation objectives in documented form. Creating high-quality testplans is well worth the effort and can pay significant dividends even after Pre-Si Validation is done. Several benefits of a written testplan are described here.

6.1.1 Testplans and Execution

Testplans provide the blueprint for all validation activities throughout the project. The testplan outlines all validation objectives and establishes the criteria by which progress is measured. While a testplan cannot describe every single task that needs to be done, it does provide a basis for teams to organize and prioritize their work so that validation objectives can be met in a timely manner.

Without a testplan, there is no reference point for determining the quality of validation. The testplan frames the collective view of stimulus, checking and coverage so that a clear picture of validation quality can emerge. Tracking the progress of an official set of objectives also injects discipline and accountability into validation execution.

6.1.2 Testplans and Communication

A testplan acts as the official Validation POR and serves as the primary means for formally communicating validation objectives and progress to stakeholders. It establishes boundaries of ownership and serves as the focal point for Validation decision-making as Validators, Designers, and Architects collaborate to improve the quality of the design under test. Testplan reviews are held to communicate plans with stakeholders and gather meaningful input and feedback on the test cases to be validated. During project execution, tracking the status of testplan objectives provides visibility and keeps everybody informed about validation progress.

6.1.3 Testplans and Project Management

Testplans provide a baseline for tracking progress not for just each individual validation team, but across all validation domains. Testplans provide visibility that help inform project-level planning and schedule. Projects are managed around global milestones, and planners need to know details that testplan-based indicators can provide, such as:

- Whether major features have been exercised
- Whether Validation has met their tape-in criteria
- Whether certain portions of the design are ahead of schedule, on track or behind in meeting validation objectives.

Deriving indicators from testplan execution status can help provide this information as project managers make decisions and assess risk along the path to tape-in.

6.1.4 Testplans and Learning

The task of drafting a testplan itself acts as a forcing function for the Validator to dive deep into learning design details and validation requirements. Most Validators will ramp quickly on the high-level design and its features, but will not learn the particulars of the design until there is a need. Drafting a testplan will help push them along this learning curve. As a Validator works to identify appropriate test scenarios or coverage items for a testplan, they will be driven to research out design features with which they are less familiar. As the Validator then starts to define testplan conditions in written form with specific details, they will be driven to ask questions that are more precise. This facilitates the ramp of expertise for the Validator, making them more effective.

6.1.5 Testplans as a Reference

An often-overlooked expectation of testplans is that they will serve as historical references for future projects. It is always an asset when Validators can refer to testplans from a previous project that has similarities to their design. This allows them to evaluate a number of things, including:

- The validation scope and objectives of the prior project.
- The validation exit criteria and outcome of the prior project.
- The learnings from the prior project, such as whether or not the validation testplan was sufficient in preventing bug escapes to silicon.

Background information from prior projects testplans is helpful as new projects set the validation strategy, establish scope, ascertain risks and write their testplans. Testplans can even be reused or modified by subsequent projects. The quality of details and reference material in testplans can have an impact on multiple product generations.

6.2 Major Components of a Testplan

Pre-Silicon testplan formats have evolved over time, but nearly all testplans are comprised of parts that can be described by one of two categories: Background Information and Testplan Entries

6.2.1 Background Information

Background information is included in a testplan to establish context and provide the basic knowledge necessary to understand the rest of the document. Background information usually summarizes key points of the scope, requirements and strategy from [Validation Planning](#). It is important to include background information in a testplan because it gives reviewers a high-level understanding of **how** validation will be done.

Background information is usually written in paragraph-style prose and organized into an outline. Background information should be kept brief. Each section should be limited to a few paragraphs, and bullets can be used to highlight key points. Reviewers are expected to have a basic understanding of the design under test and general validation principles, so Validators should avoid re-documenting information that is already available in a HAS or other specifications.

Background information should include the following sections.

6.2.1.1 Scope

This scope section of the testplan describes:

- Which team owns the validation plan.
- Which product or product family the validation plan covers.
- Which product configurations the validation plan covers.
- Which design hierarchy, features or flows the validation plan covers.

- Assumptions about which elements of the design are covered by other validation teams.

6.2.1.2 Feature Description

The feature description section of the testplan provides:

- A brief description of the model or design under test. Images may be used.
- A list of features to be tested with brief (1-2 sentence) descriptions.
- Links to relevant feature documents (HAS, Architectural feature descriptions).

6.2.1.3 Validation Strategy

The validation strategy section of the testplan describes:

- Whether requirements-based validation, specification-based validation or a combination will be used.
- Whether coverage-based conditions, content-based conditions or both will be used.
- Which validation platforms and validation disciplines will be used.
- Which forms of stimulus or stimulus tools will be used. Details might include:
 - Whether directed, directed-random or random tests will be used (or a combination).
 - Which test generators will be used.
 - Which injectors will be used.
 - Which tools will be used to configure the DUT or model.
 - Which tools will be used to run tests or regressions.
 - Which types of regressions will be run (and the volume).
 - Where tests are located.
- Which forms of checking or checking tools will be used. Details might include:
 - Whether tests will be self-checking or whether they require checkers.
 - Whether dynamic checking, static checking or post-processing checks will be used.
 - Whether scoreboard checkers, reference models or compliance monitors are used.
- Which forms of coverage collection or coverage analysis tools will be used.
- Which tools or methods are used for validation using formal proofs, if applicable.
- Which collateral will be inherited and which will be developed.
- Which tools or methods will be used to measure validation progress.
- Which validation methods or areas of focus will be emphasized or prioritized

This section should provide a link to validation strategy documents if applicable.

6.2.1.4 Test Environment

The test environment section of the testplan provides:

- A brief overview of the testbench or test environment. Images may be used.
- A brief description of how the test environment is configured.
- A summary of major interfaces or points of interest for stimulus, checking, monitoring, debug or observability.

- A summary of key components of test environment (behavioral models, emulators, BFM s).
- A summary of key features and/or limitations of the test environment.
- A description of the environment used for formal proof validation, if applicable.

6.2.1.5 Review Notes and Action Items

It is useful to have a dedicated section of the testplan that can be used to capture notes and actions items that are assigned during testplan reviews. Action items may be captured in a table and should have a description and fields for owner, release, milestone and status.

6.2.2 Testplan Entries

The most important section of a testplan is the section that describes *what* will be validated. Testplans capture this information in a list or set of items called the *testplan entries*. Testplan entries usually describe testing and coverage scenarios that need to be validated, and checking that needs to be enabled. Ideally, testplan entries will represent everything that needs to be verified to flush out bugs and demonstrate the health of the design.

6.2.2.1 Testplan Entry Types

Entries for Pre-Silicon validation testplans can be divided into the following categories or types:

Type	Purpose
Functional Test	Used to describe test scenarios involving flows or features that will be validated using directed or directed-random tests.
Functional Stress	Used to describe testing that involves random tests, random injectors or various test modes on full regression suites to stress complex interactions or cross-feature behavior.
Functional Coverage	Used to describe test scenarios that will be validated using coverage feedback or formal coverage.
Functional Checker	Used to describe checking that is to be enabled or performed during validation.
Formal Proof	Used to describe a property to be validated using formal proof validation.

Test, stress and coverage testplan entries each describe testing scenarios or events and are sometimes called *test conditions*, *test cases* or *test case definitions*. Testplan tools might assign specific meaning to these terms, but in this document, they are used interchangeably. In general, these terms do not apply to checker entries.

These types may not be sufficient to describe all validation objectives in a testplan. The following testplan entry types may also be needed to describe items that do not fit into the types above.

Type	Purpose
TE Component	Used to describe components, capabilities or features of the test environment needed for robust validation. For example: <ul style="list-style-type: none"> • Integration, enabling and reuse of TE verification collateral • Algorithms, weights or tools used to randomize configuration or stimulus
Design Component	Used to describe collateral that must be implemented in the design to enable robust validation. For example: <ul style="list-style-type: none"> • Instrumentation code and hardware assertions • Register and fuse definitions

6.2.2.2 Testplan Entry Fields

Testplan entries use a combination of written descriptions and fields to describe a validation objective. The fields contain information that is helpful to Validators and reviewers as they work with the testplan. This section describes each of the building blocks of a testplan entry.

6.2.2.2.1 Title

Each testplan entry has a title, which is a one-line text description of the test entry. The title communicates a basic level of intent and is useful for quickly viewing entries when browsing through a list. The title should be descriptive enough to recognize the condition and differentiate the testplan entry from other similar entries. It should not contain overt details that belong in the description.

6.2.2.2.2 Description

Each testplan entry has a written description. The description should include all information necessary to understand the validation objective. If there is any part of testplan writing that deserves extra attention, this is it! Testplan authors should put in the effort to clearly describe the feature, test scenario, checker or validation objective that a testplan entry represents. As reviewers read the entry, there should be no ambiguity in their minds as to the intent or means of validation. Likewise, if validation is handed off to a new owner, the description should tell them precisely what needs to be done.

The contents of the description will vary, based on the type of the testplan entry:

Type	Description Field
Functional Test	Describe the feature, flow or scenario to be tested. Be explicit about the exact scenarios that will be targeted. Specify which directed or directed-random test(s) will be used to verify the scenario (or provide a link to the test if the testplan tool supports this).

Functional Stress	Describe the random test behavior, test mode, feature crossing, injector or other mechanism that will be used to stress the design. Specify which regression(s) will be run (or provide a link to the regression list if the testplan tool supports this).
Functional Coverage	Describe the functional coverage scenarios that will be targeted. Be explicit about the exact events that will be monitored for coverage. Specify the name of the coverage monitor or formal assertions that will be used collect the coverage (or provide a link to the coverage results if the testplan tool supports this).
Functional Checker	Describe the checker that is to be deployed and a brief summary of the checking it provides.
Formal Proof	Describe the property that will be validated, including assumptions.
TE Component	Describe the TE component, capability or feature that is required.
Design Component	Describe the RTL or design collateral that is required.

6.2.2.2.3 Type

The type field is used to specify the entry types described earlier: functional test, functional stress, functional checker, functional coverage, TE component or RTL component.

6.2.2.2.4 Release

The release field indicates the release for which the entry is valid. Testplans are often maintained through several product releases or steppings, and entries may be added to the testplan after the initial stepping as design changes are introduced. Alternatively, some testplan entries from an initial stepping or release may not apply to a subsequent stepping.

When a testplan is reused across steppings or releases, this field helps differentiate which entries apply to each stepping or release as part of a *Verify All*, *Verify New* or *Verify Select* multi-stepping validation strategy (see [Validation Planning section: Multi-Stepping and Multi-Die Derivative Strategy](#)). In these cases, the field is most useful if it is implemented as a list of each stepping or release for which the entry is valid. Validators are cautioned against using a single testplan entry that specifies multiple releases if the status field can only be used to record the progress for one of the releases. This can lead to ambiguity for determining which release was used in validation.

6.2.2.2.5 Scope

The scope field identifies the scope to which a testplan entry belongs. Projects should decide on appropriate options for populating this field before testplans are written based on the hierarchical and discipline scopes of validation ownership that exist.

Using a scope field instead of a team field is desirable because teams can be vague and change between organizations or steppings, but scope is constant and specific to the validation objective.

Entries within the same testplan usually all have the same scope, because testplans are written by scope. Consequently, this field is only useful in cases where testplan tools contain entries spanning multiple scopes. In these cases, the scope field can be used to create automated searches that look for entries belonging to a specific hierarchy or discipline. The scope field can be used in this way to generate testplan status indicators based on scope and match them up with other indicators from the same scope – such as pass rates or coverage.

Validators are cautioned against sharing testplan entries with other teams or projects where the scope of ownership is not consistent and only one scope can be specified. This can complicate the ability to generate scope-based indicators.

6.2.2.6 Validation Environment

The validation environment field is used to differentiate test entries meant to run on simulation, emulation or silicon. This provides context for the validation objective.

Validators are cautioned against creating a single testplan entry that specifies multiple validation environments if the status field can only be used to record the progress for one of the environments. This can lead to ambiguity for determining which platform was used to make progress.

6.2.2.7 Milestone

Testplan entries can be mapped to project milestones or milestones defined in the Product Lifecycle (PLC). For PLC milestones, the following guidelines apply:

- Entries related to cold boot or reset of the design should be marked as VAL 0.3
- Entries related to basic DUT feature exercise should be marked as VAL 0.5
- Entries related to basic exercise of DUT interfaces are marked as VAL 0.5
- Entries related to random testing and stress scenarios are marked as VAL 0.8
- Entries added during VAL 0.8 in response to bugs, testplan reviews or late-breaking design changes are to be marked as VAL 1.0

The milestone field is useful in helping Validators stage activities and in measuring testplan progress towards project or global milestones.

Validators are cautioned that milestones are not always consistent between projects for the same testplan entries, and this can cause complications if testplan entries are reused between projects.

6.2.2.8 Priority

The priority field is used to assign high, medium or low priority to testplan fields. Not all testplan entries need to be treated equally. Setting priorities is useful for aligning the execution of test cases with various phases of the design cycle. High priority is assigned to entries where testing is urgent because the cost of fixing a bug in the design is high. High and medium priority is assigned to entries that need to be verified before a product can tape in, where the priority suggests the relative order in which the items should be validated. Low priority can be assigned to entries that pose

minimal risk to the product if validation was left undone and a bug were to be found in the design. For example, if basic exercise has been done for a feature that is not customer visible (e.g. debug hooks) it may be appropriate to lower the priority of corner-case testing if there are other high priority, customer-visible features that still need testing. Validators should seek buy-in from the Designers and Architects before assigning a low priority to a test case.

6.2.2.9 Status

The status field indicates the validation progress of a testplan entry. Pre-Silicon testplans generally track status as *complete* or *incomplete* for entries based on the following criteria:

Type	Status Field
Functional Test	Marked complete when the directed or random-directed test that covers the specified scenario have passed and due diligence has been performed to confirm that the scenario has been tested.
Functional Stress	Marked complete when the specified random testing, test mode or injector has been enabled in regressions and due diligence has been performed to confirm that the stress mechanisms are working. Ongoing testing pass rates should be tracked and reported independently.
Functional Coverage	Marked complete when a coverage monitor for the specified events has been coded and enabled in regression testing and it is confirmed that the conditions have been hit at least once. Ongoing coverage results should be tracked and reported independently.
Formal Proof	Marked complete when the formal property has been validated.
Functional Checker	Marked complete when specified checking has been deployed in test runs and regressions.
TE Component	Marked as complete when the test environment supports the specified feature in the production validation test environment.
Design Component	Marked as complete when the change has been turned in and released to the production RTL model (at the relevant hierarchical scope).

Some testplan tools implement status as a *percent complete* for test, stress or coverage entries. This enables pass rates or coverage to be recorded directly within a testplan and can eliminate or reduce the need to track such status independently.

The status field enables Validators to track testplan progress and generate indicators. Recording status is also valuable during testplan reviews, because it shows reviewers which test objectives have been reached and which have not and validation tasks can then be prioritized for completion.

Pre-Silicon does not make any use of a status field to track whether individual testplan entries have been reviewed, since testplan reviews cover the full testplan at once.

This is a simple example of a status field. Status may be tracked in more detailed ways, depending on the project's testplan tool and progress tracking and reporting strategy.

6.2.2.10 Bug or ECO Link

If a testplan entry is created in response to a documented bug fix, issue or an engineering change (ECO), the bug or ECO ID should be linked to the entry in the testplan.

6.2.2.3 Organization of Testplan Entries

Organizing test entries by type first and then by feature (or vice versa) allows reviewers to easily see what is being validated and how it is being validated. Testplan authors may also choose to organize entries in order of increasing complexity or decreasing priority.

6.3 Creating Testplan Entries

The real art behind testplan writing is knowing how to create effective testplan entries. Testplan entries need to represent all requirements *and* establish the standard by which the health of the design is measured. Validators not only need to know what to validate, but how to capture it in an entry.

Fortunately, this skill grows with experience. As Validators actively engage in planning, execution and reviews, they will begin to appreciate the role and value of a well-written testplan. They will begin to see what makes good testplan entry and what does not. Validators of all levels are encouraged to be part of testplan writing and reviews.

The following sections set forth several techniques, tips and advice for creating good testplan entries. This section assumes validators have gathered requirements as described in [Validation Planning](#).

6.3.1 Prioritize Top-Down Requirements

Always include testplan entries that represent the top-down requirements of the design. There is no justification for failing to validate logic or test scenarios directly related to the landing zone definition and product features. After gathering requirements, reading through specifications and consulting with Architects and Designers, Validators should know which features their portion of the design supports. Create testplan entries for these features, with feature-oriented titles and descriptions. Ask questions like:

- How do the landing zone features apply to this portion of the design?
- How do the design or features in this scope fit into the overall system?

6.3.2 Validate the Interfaces of the Design

Many scopes own validation of the interfaces of their design. This is certainly true when the design hierarchy has an external interface that has not already been validated, or when the design hierarchy includes interfaces between blocks that are being connected for the first time. Testplan entries for interfaces should address validation of the interface protocol and verify all supported traffic types from all sources on the interface. Interfaces and interface protocols should also be stressed to the extent possible (e.g. high-bandwidth, traffic combinations, protocol delays etc.). When describing the scenarios to be tested on an interface, be specific about all opcodes and transaction attributes that should be tested. Testplan entries involving interface traffic often use coverage, but functional test entries can also be used as long as the test guarantees to cover the scenarios (and will fail if it does not). Make sure to include traffic types in each direction. Examples include:

- IOSF Sideband Target: See CRRd, CRWR, MemRd, MemWr, IP Ready, Fuse Download Request, CFGRd, CFGWr, PM_RSP on target interface.
- IOSF Sideband Master: See MemRd, MemWr, Fuse Download Complete, PM_REQ on master interface

For integration teams, interfaces should exercise not just all traffic types, but traffic to and from each source and destination agent that communicates on the interface.

6.3.3 Think in terms of Hierarchy and Scope

As described in [Validation Planning](#) Pre-Silicon Validation scopes of ownership are organized hierarchically, so a hierarchical mindset should be used to create testplan entries for common design hierarchies like IP, Subsystems and FC. Validators should think about:

- What portions of the design have been tested in other scopes?
- What portions of the design are new and need to be tested?
- Can more complex testing be done at this level of hierarchy than the next?
- What portions of the design are coming together for the first time and can be tested together? What interfaces, transactions and flows are involved?

6.3.4 Use a Structure-Oriented Approach for New Logic

As features, design changes and bug fixes are implemented in RTL, Validators should verify that new or changed RTL structures work as intended. In these scenarios, a structure-oriented approach can help determine testplan entries. With this bottoms-up approach, Validators look through the design implementation for relevant structures and determine what needs to be exercised. As structures are identified, the Validator asks questions about what it will take to sufficiently validate the logic and adds testplan entries as needed. For example, consider the following structures:

- | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none">• Datapaths• Muxes, Decoders, Arbiters• Flow Control Logic• Arrays, Queues, FIFOs | <ul style="list-style-type: none">- Are all datapaths exercised?- Are all input options and combinations exercised?- Is backpressure exercised? Are idle cases exercised?- Is allocation/deallocation exercised? Pointer wraps? Full? |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- Control Registers - Are register attributes tested?
- FSMs and FSM Arcs - Are all states and paths exercised?
- Clock Crossing Logic - Are all ratio combinations exercised?
- Error Logic - Is this logic exercised?

When using a structure-oriented approach to identify testplan entries, Validators should determine if logic structures can be verified implicitly with tests, or if coverage entries are needed to demonstrate that a targeted scenario occurs during testing. For instance, it is impossible to tell from a functional test without coverage if a design with flow control credits ever consumed all credits causing backpressure. In this case, coverage is needed to verify that the event occurred and there is confidence that the logic works with backpressure.

In many cases, Validators may find that existing testplan entries will sufficiently exercise logic structures you are investigating. Alternatively, Validators may discover logic structures that are not tested by any existing testplan entries and that new entries should be added. MicroArchitectural expertise and implementation knowledge gained while gathering requirements is tremendously helpful in using a structure-oriented approach to develop new test entries.

6.3.5 Use Point-of-View and Setup Analysis to Identify Test Scenarios

Sometimes it is useful to change your point of view as you consider what it takes to exercise a feature. Try looking at things from the point of view of a transaction, a flow or a feature that needs to be validated. Are there interesting paths through the design that can occur? Starting with reset, are there flows or setup conditions that need to be met so that your scenario can even occur which should have their own testplan entries? Consider an upstream PCIe read to memory:

Point of View Example Scenarios:

- The read is retried on the link due to a parity error.
- The read is held back at an arbiter as a burst from a higher priority channel is serviced.
- The read snoops the cores on the way to memory and hits in the cache.
- The read is passed by writes to the same cacheline while waiting in an upstream queue.
- The read is aborted due to insufficient security attributes
- The read completion is blocked by flow control logic waiting for another completion to return the credit.

Setup Analysis:

What prerequisites are needed for an upstream PCIe read to memory?

- PCIe powergood indication, reset deassertion, fuse download, link training, fabric credit initialization, IP ready indication, PCIe device enumeration and configuration
- Memory initialization, PCIe crossbar initialization, etc.

After using point-of-view and setup analysis, the following questions can be asked:

- Are these scenarios already covered by existing testplan entries other validation scopes?
- Is it sufficient to just have a testplan entry for a memory read, or should these scenarios be split out into multiple entries, or multiple cover events in an entry?

6.3.6 Include Reset and Power Management

When applicable, include testplan entries for reset and power management. For example:

- Cold boot, warm reset, function-level reset
- Clock ratio changes
- Power up, power down, device power states, package power states
- Autonomous clock gating scenarios
- Power delivery (UPF) scenarios

6.3.7 Include other Globals

When applicable, include testplan entries for control register accesses, security, fuses, performance, Design for Test (DFT), Design for Debug (DFD) and conditions from other global validation domains.

6.3.8 Include Configuration

Include entries that test the design in all supported configurations. For example:

- Running at all supported clock frequencies or clock ratio combinations
- Running with devices fused off
- Running with interface ports in all supported configurations
- Running with all supported functional modes
- Running with X-Propagation mode enabled

6.3.9 Include Stress Entries

Stressing the design is important to flush out bugs that would not be found by simple exercise of the features. Think about ways that can be used to stress or perturb normal operation and create testplan entries. Examples include:

Stress modes:

- Running all tests with a new or custom mode (i.e. UPF, clock frequencies)
- Running all tests with a new or custom injector (i.e. snoop injector)
- Cross feature testing (i.e. warm reset during package states)

Random Regressions:

- If random regression testing is part of your validation strategy, it should be represented by a testplan entry. Random regressions are a form of stress. Entries should describe the test(s) or list(s) at a high level.

6.3.10 Include Checking Entries

Checking entries should be created to show what checking will be used by validation. Entries should describe the checking at a high level without getting into too much detail. Examples of checker entries include:

- Scoreboard Checker
- Compliance Checking
- Hardware assertions
- Manual waveform reviews

6.3.11 Think, Reflect and Ponder

Ask yourself questions that could lead to the discovery of better testplan entries:

- What are the use cases for this portion of the design?
- What testing will Designers and Architects expect to be done?
- What will the customers at the next stage expect to work?
- What portions of the design are new?
- What portions of the design have changed?
- What portions of the design are being brought together for the first time?
- What portions of the design are poorly understood? Why?
- What could cause the design to lock up or fail to respond to requests?
- What fatal bugs or escapes have been found in this portion of the design?
- Are there configurations that allow or prevent a test scenario from occurring?
- Should the entire specification be validated, or only a portion?

6.3.12 Discipline-based Techniques

There is no simple formula that can be used to identify testplan conditions across all validation disciplines. Readers are encouraged to read the chapters for each discipline, consider the validation objectives, and combine the concepts and learnings with the concepts discussed in this section to create appropriate testplan entries.

6.3.13 Using HSD Bugs/ECOs/Issues

Bugs, ECOs and issues filed in the HSD database are excellent sources for test cases.

6.3.14 Learn from Others

Share your testplan entries with others. Get their feedback. Browse testplan entries written by others. See what types of testplan entries they are writing. Ask whether there is anything about the types, style or quality of the testplan entries you see that is inconsistent with the entries you have in mind. For integration validation, ask the IP providers what they think should be in an integration testplan.

6.3.15 Learn from Predecessors and the Prior Art

Before drafting a testplan, it is worthwhile to reference testplans from prior projects. Some things that may be worth evaluating might include:

- Does the prior testplan have content or testplan entries that you can reuse or leverage for your testplan?
- Did the prior project execute well to the testplan?
- Were there any major gaps from their testplan (in hindsight) that led to bug escapes?
- Were there any testplan activities with questionable return?
- Does the overall organization or style of testplan entries work well?
- Should you follow or deviate from the organization used in prior testplans?
- Do the conditions have sufficient detail?

6.3.16 General Advice for Writing Testplan Entries

Testplan entries form the meat of a testplan. They establish the quality of the testplan and it is important that they be documented in a clear and consistent format so that the validation objectives they represent are not misunderstood. This section contains advice for creating understandable testplan entries. Testplan writers should adhere to the following advice when writing test cases.

6.3.16.1 Get Started

When drafting a testplan, start creating entries as soon as you think of them. Do not worry about overlap, redundancy or organization in the initial stages. Start writing down titles for testplan entries. There is tremendous value to getting your ideas down as you think of them rather than obsessing about details. After brainstorming, you can revisit the testplan to organize the entries more efficiently and pare down entries that do not make sense. Hold off on writing detailed descriptions until you are certain that the entries are the ones you want - otherwise you might end up wasting a lot of time. If there is significant overhead in creating or editing entries in a testplan tool, consider creating your initial drafts and entries in a spreadsheet and transfer your entries to the tool once your testplan starts to stabilize. Additionally, get early feedback from others as the initial set of testplan entries are defined.

6.3.16.2 Follow the Law of Testplan Writing

While drafting testplans, Validators are instructed to always follow the Law of Testplan Writing:

“A testplan condition must be easily understood by anyone with a reasonable background in the design.”

6.3.16.3 Write testplan entries using the same style across the project

When writing testplan entries, it is not appropriate to use any form of shorthand writing or custom syntax that would not be understood by others. It is important to reviewers and future owners of testplans to be able to look at any test condition within a project and immediately be familiar (and comfortable) with the style in which it is written. This helps in reviewing testplans, and is helpful when Validators are moved from unit to unit to mitigate risk as the project proceeds.

While custom styles are discouraged, a few commonly used style conventions have been developed over time that are useful for writing testplan descriptions. These include:

Sets:

Sets represent multiple related scenarios that need to be validated. Curled braces {} are used to indicate the list of set elements. For example, the following testplan entry description represents five different scenarios that need to be tested. Coverage might be used to verify that all five scenarios are hit:

See a {Read, Write, IO Read, IO Write, Completion} on the master interface.

Compound Sets:

Compound sets can also be expressed in testplan entry definitions. For example, the following condition represents $3 \times 2 \times 3 = 18$ scenarios that must be verified:

See a {load, store, nop} receive an {ack, nak} in {stall, normal, burst} mode.

Time Windows:

Time windows are used in testplan conditions to specify temporal relationships between events in the design. Block brackets [] are used to designate time windows. Within the brackets, the testplan author can specify whether the testplan entry targets *every* value in the range or *any* value in the range using the syntax from the following examples:

See a PM request arrive [0..4] qclocks after the power FSM moves to IDLE
This condition targets five scenarios – one for each value in the range (i.e. the scenario must be validated for 0 qclks, 1 qclks, 2 qclks, 3 qclks and 4 qclks)

See a PM request arrive [0, 1-5, 6-10] uclks after the arbiter lock bit is set.
This condition targets 3 scenarios – one for any value in each range (i.e. the scenario must be validated for 0 uclks, any time value between 1 and 5 uclks and any time value between 6 and 10 uclks)

Acronyms are allowable in the description, but again it should follow the rule that they should be easily understood by anyone with a reasonable background in the design.

6.3.16.4 Use the correct type

Testplan entries that describe tests, stimulus or coverage should avoid using the terms *check* or *verify* unless they describe a self-checking test. If the entry really is checking or verifying, consider changing the entry to checker-type.

Likewise, test and stress entries in a testplan should only be used to describe test scenarios that do not use explicit coverage monitors to verify events. If coverage feedback will be used, the type should be coverage.

6.3.16.5 Breadth vs. Depth

Testplan entries are often described in terms of *breadth* vs. *depth*. This is due the fact that testplan entries have a dual purpose: They are used to demonstrate the health of the design by showing what works, but they are also used to describe validation scenarios that will have a high probability of uncovering design flaws. Testplan entries should be broad enough to ensure that all top-down requirements and features of the design have been tested, but deep enough to expose corner-case bugs. Consequently, testplans often include entries on both ends of the breadth and depth spectrum. Entries for breadth usually involve basic testing across a broad range of features (typically validated during PLC VAL 0.5), while entries for depth usually involve stress scenarios (typically validated during PLC VAL 0.8).

6.3.16.6 Features vs. Implementation

One of the dilemmas Validators face when writing a testplan is how to balance validation of an Architectural feature vs. its implementation. Consider the low-power state S0ix power management feature. At the feature level, the validation objective can simply be expressed as “Enter and exit S0ix”. However, at the implementation level, a Validator knows that there are dozens of hardware handshakes, messages and signaling protocols that must be observed to enter and exit S0ix. Both the feature and its hardware implementation must be validated for integration validation to be complete. Validators may feel it is insufficient for the testplan to address only the feature or flow without also comprehending the hardware interactions that are the actual target of the validation – especially when they have to know all of these details to get the flow working. Testplan authors are reminded to not re-document all the low-level details of features already described in specifications, and only highlight enough details to build a picture of their validation process and the items to which they paid particular attention. A testplan entry for S0ix might therefore read:

Title: S0ix Entry and Exit with Save/Restore

Description: Halt all core and IP traffic and see the Punit initiate S0ix entry after sending a PM_REQ to all agents and receiving PM_Response. See critical state saved by the save-restore engine, observe the removal of power, including UPF state corruption, followed by S0ix exit initiated by {X,Y,Z} and critical state restored by the save-restore engine. Test with the s0ix_basic_test template running with the save-restore checker.

6.3.16.7 Do not over-engineer test cases

Engineers make things succinct and compact. This is not necessarily a good trait when writing testplan entries. Trying to fit a description into the smallest space possible will often lead to problems in understanding for reviewers and implementers (and maybe even the testplan writer after some time). ‘Dumbing down’ the writing in the test case can make it much more explicit and understandable. Be repetitive. Be explicit and make everything obvious.

6.3.16.8 Avoid creating too few or too many entries in a testplan

Do not create too many or too few entries. Too many conditions can obscure the big picture and sidetrack Validators and reviewers from the most important validation objectives. Too few conditions can introduce gaps and significant risk. Validators should try to keep a good balance of breadth conditions that ensure basic exercise of all features, and depth conditions that go after corner cases and stress. While it is impossible to provide a guideline that applies to every validation domain, testplan authors should aim to have 20-40 entries in a testplan. Entries themselves should have descriptions that target a limited number of scenarios – fewer than 10 in most cases. This is a manageable, reasonable size for a testplan. There will always be exceptions to this recommendation, but keeping this rule of thumb in mind when creating entries will help Validators as they work to pare down the validation space. If a larger set of entries is justified, consider separating the validation plan into multiple plans based on common features.

6.3.16.9 Avoid referencing signal names

References to signal names or RTL code in testplan entries will result in high maintenance overhead. Therefore, avoid any explicit reference to names of the signals in the RTL or pipelines in your testplan.

Here is an example showing a good and bad way of documenting a coverage test condition:

Good: *Test the presence of a load uop followed by a store uop in back to back clocks*

Bad: *Test the presence of daloadm503h and mostorem503h in back to back clocks*

The second example might be suitable for coding a coverage monitor, it is not a good form to use in a testplan because it is unclear if the test case is particular to the load/store in the given pipe stages or if is applicable to a load or store in any pipe stage.

While signal names is discouraged, it is acceptable to use acronyms and other terms that are commonly used by Architects, Designers and Validators when discussing the design.

6.3.17 Limitations of these methods

The techniques shared here are prescriptive guidelines intended to help Validators as they attempt to come up with testplan entries. Readers should understand that there is no perfect formula for this task and these methods are not meant to serve as a comprehensive set of steps for completing a testplan. Proper engineering judgment must be used. New Validators are encouraged to seek the

expertise and advice of experienced Validators to come up with a set of testplan entries that represent validation objectives with a good balance of breadth and depth.

Stakeholders should understand that it is impossible to come up all possible test cases when the testplan is documented for the first time. Validators will continue looking for applicable test cases to add to the testplan as their expertise and familiarity with the design grows, but feedback from others is also essential. Stakeholders should take advantage of opportunities to review the testplan for completeness and provide feedback.

6.4 Testplan Lifecycle

This section covers the lifecycle of a testplan. The timelines for drafting and reviewing the testplan are provided along with information on how to maintain a testplan over its lifecycle. Testplan sharing and reuse are also discussed.

6.4.1 Tasks and Timelines

The following tasks and timelines are part of a testplan lifecycle.

6.4.1.1 Drafting

Testplan drafting should begin as TR concludes, at the beginning of the PLC VAL 0.0 Milestone. Testplan drafting can easily become an all-consuming task. While it is important to create high-quality testplans, there is a point of diminishing returns and Validators should not go overboard in spending excessive time writing and refining background information or testplan entries. There will be ample time to refine the testplan during the life of the project.

6.4.1.2 10% Review

Before progressing far into testplan drafting, new testplan authors can hold a *10% review* with experienced Validators to ensure that the testplan conditions they are defining are appropriate, well written and that the testplan is in harmony with project guidelines.

6.4.1.3 Testplan Review

Once the initial draft of a testplan is complete, the testplan author should organize a testplan review with stakeholders. Reviews are held near the *start* of the PLC VAL 0.5 milestone. Invite Architects, Designers, managers and Validators from adjacent design units, Post-Silicon stakeholders and other customers of Validation to attend the review. The testplan author should organize and publish the testplan conditions in a format that is easy to review and send copies or a link to reviewers ahead of the review. Stakeholders should be informed that they are expected to read the testplan offline before coming to the review, and that Validation requests their approval of the plan.

The purpose of the review is to communicate the validation scope, strategy and POR plans to stakeholders so that they can provide feedback. Reviews are typically held face-to-face and the testplan author presents the background information and each test condition (one at a time) to the reviewers. All participants endeavor to review assumptions and identify gaps, overlaps or risks in the validation plan. Stakeholders should ask themselves questions such as: ‘Is this test condition good?’ or ‘Is this group of testplan entries sufficient to make sure this feature is validated?’

Notes are taken during the review and the testplan author will add conditions or revise the contents based on the feedback.

6.4.1.4 Testplan Exit Reviews

As part of the PLC VAL 0.8 Milestone, the final version of a testplan is reviewed with stakeholders to discuss the quality of validation and any testplan items that have not been completed. The goal of this review is to identify any uncompleted validation activities that should gate a VAL 1.0 “GO” declaration for taping in the design and sending it out to be manufactured. During the review, Validators, Designers, Architects and managers will apply paranoia based on the learnings and discovery they made during execution and based on experience from prior projects. Reviewers should ask themselves whether they think Validation has demonstrated that design requirements have been met. High priority tasks are identified and documented in the Review section of the testplan as action items to complete as part of VAL 1.0.

At the conclusion of VAL 1.0, the tape-in window will be near and validation teams will be asked to give a “GO / NO-GO” declaration for tape-in. At this point, the testplan is again reviewed to verify that all gating action items were completed, so that the validation team can declare tape-in “GO” for their scope of ownership. Uncompleted items that cannot be completed in a timely manner for the project tape-in window should elicit a “NO-GO” declaration if the items pose a significant risk to the health of the product. Uncompleted items that are not significant enough to halt tape-in should prompt a “GO with RISK” declaration for tape-in, and the team should strive to complete the tasks as soon as possible so that they can change to declaring “GO”.

6.4.2 Testplan Maintenance

After testplans are written, changes can and will occur that require a validation response. Changes come in all varieties: large and small, planned and unplanned significant and insignificant. Validators must track design changes, understand the impact to their validation scope and make sure that the design continues to behave as expected. Often this is accomplished by running a regression of the validation test suite on each release of the design (See [Regressions](#)). In some cases, the change is significant and validation needs to formulate new testplan entries so that the new feature, structure or behavior is validated. In such cases, update the testplan with the new entries. By updating the testplan, the Validator acknowledges and communicates to stakeholders that the validation POR comprehends the change.

The following changes may go into a design during its lifecycle and might require updates to the testplan:

- Product redefinition or new landing zone features added late during development

- Engineering design changes due to delayed implementation, bug fixes, design optimization, critical path timing closure, backend design constraints or silicon feedback
- Closure of Architectural issues, holes or gaps

Tracking all changes to a design can be difficult, especially in cases where no process is in place to detect or report changes. Validators must stay in close contact with stakeholders making decisions about design changes and use testplan reviews to make sure nothing is missed.

In addition to making updates to testplan entries, Validators should also keep the status field of testplan entries up to date.

6.4.3 Testplan Sharing and Reuse

Projects often have a goal of reducing overhead effort associated with testplan writing. To this end, many projects have endeavored to implement some form of testplan sharing or reuse. Testplan sharing and reuse can mean multiple things. Here are some of the different ways testplan reuse can be defined.

6.4.3.1 Testplan Sharing

Testplan sharing is enabling other projects to view your testplan for reference. This is important for helping similar projects understand the history, prior art and context behind the validation of your product. Testplans can be shared directly using secure email, posting to a secure site, or by storing the testplan and its entries in a database.

6.4.3.2 Testplan Copy Reuse

The simplest form of testplan reuse is copying the entire testplan for reuse on another project. The new project is then free to amend, modify or keep the original testplan.

6.4.3.3 Testplan Incremental Reuse

Another form of reuse is incremental reuse, where a new project or stepping refers to the prior testplan, but only creates testplan entries for new conditions. Progress is only tracked on the new conditions and it is assumed that regressions or other mechanisms are in place to keep legacy functionality healthy. Incremental testplan reuse can be used for derivative teams that use a *Verify New* multi-die or multi-stepping derivative strategy (as described in [Validation Planning section: Multi-Stepping and Multi-Die Derivative Strategy](#)). Because incremental reuse only focusses on what is new or changed, it complicates other types of testplan reuse (i.e. multi-stepping/multi-die reuse).

6.4.3.4 Testplan Hierarchical Reuse

Another form of reuse is hierarchical reuse. The theory behind this is that each level of the hierarchy inherits, copies or directly reuses testplan entries from the lower level. For instance, a

subsystem integrating an IP would create integration conditions and then inherit conditions from the IP level for retesting. Depending on the implementation, this type of reuse could face a few challenges. For instance, if testplan entries are reused directly from the lower level (i.e. both levels are using the same entry) there must be a way to differentiate the status and scope.

Pre-silicon Validation generally does not reuse testplan entries across hierarchies. It is not pragmatic to repeat all IP testing at subsystem and fullchip levels in Pre-Silicon, and it is efficient to minimize any unnecessary validation overlap. Consequently, validation objectives and testplan entries tend to be different at each level. For instance IP-level validation will focus on white-box testing of the IP, but the validation emphasis shifts to interactions between the IP and other blocks as it is integrated into a subsystem or Fullchip. Exceptions might include interface transactions or global flows that are important to validate at both levels. Such exceptions are good candidates for hierarchical testplan entry reuse.

6.4.3.5 Testplan Pre-Silicon / Post-Silicon Reuse

Another possible form of reuse is reusing testplans or testplan entries from Pre-Silicon to Post-Silicon. The theory behind this is that all testing done in Pre-Silicon should also be done on actual silicon. Some validation joint team (VJT) efforts have had limited success with this, often at the expense of simplifying testplan entries. The following issues with direct reuse of testplans or testplan entries must be overcome if this type of reuse is to be achieved:

- Pre-Silicon testplans are hierarchically organized (hierarchy first), whereas Post-Silicon organizes with a feature-first orientation. This introduces structural incompatibilities for reuse and Post-Silicon will likely need to glean testplan entries from multiple layers of Pre-Silicon validation plans. For example, the Post-Silicon team owns testing for all of the hardware for an IP or feature. When reusing Pre-Silicon testplans, they may need white-box test conditions that only exist in IP-level testplans and also integration conditions that only exist at a subsystem or Fullchip level. The Post-Silicon team cannot assume that they can reuse the contents of a Fullchip or IP testplan alone for validation (e.g. they cannot assume Pre-Silicon testplans have full hierarchical reuse).
- Pre-Silicon testplans at the IP level target implementation-level structures that are less important to Post-Silicon, which is feature-oriented.
- Sharing testplan entries requires a solution for independently tracking status, setting scope, linking to milestones, indicating the platform and linking to content or coverage. Each of these may be different for Pre-Silicon versus Post-Silicon.
- Post-Silicon can and should have testplan entries that do not exist for Pre-Silicon. Each level of testing should add value above the last, and silicon testing has testing capabilities that go way beyond what Pre-Silicon testing can do.
- If Pre- and Post-Silicon entries are not all the same, there must be a way to independently review testplan entries that belong to Pre-Silicon vs. Post-Silicon.
- Testplan entries must have sufficiently general descriptions such that they can be used for both validation environments. This can lead to inadequate descriptions for a given validation environment.
- Background information and strategy is likely to be different between the two organizations. Testplan tools must support documentation for each party and a way to review the background information with the applicable entries.

6.4.3.6 Testplan Multi-Stepping / Multi-Die Reuse

Another form of testplan reuse involves reusing testplans or testplan entries across multiple steppings or multiple die releases in the same product family. The theory behind this is that products in the same family should have similar validation requirements and testplans can be leveraged. The following issues with direct reuse of testplans or testplan entries on a multi-stepping / multi-die basis are:

- Stepping or die may choose to implement a *Verify All*, *Verify New* or *Verify Select* strategy. The tools must support all options by enabling testplan entries to mark *each* release affected and enable separate status tracking based on unique indicator scopes for each release. It cannot be assumed that testing status on one release counts towards the next as implementation may change for the same feature.
- Not all projects map the same testplan entries to the same milestones. For example the PLC states that new IPs need to have basic exercise done by VAL 0.5 and full regression testing done by VAL 0.8. On a derivative, the IP is considered mature or legacy and many requirements in VAL0.8 shift by one milestone to VAL 0.5.
- Teams must not create release-specific incremental testplans that break the reuse model. All entries for a testplan must go into the same testplan for all products in the family.
- Tools must enable each release to review applicable testplan entries and track status independently

6.4.3.7 Testplan Mixed Reuse

Mixed testplan reuse involves a combination of the reuse methods above. For instance, Pre-Silicon / Post-Silicon reuse and multi-stepping / multi-die reuse can be combined. This creates a challenge for the tools, which must find a way to support the combination based on the constraints listed above, and requires all participant testplan authors to have a solid understanding of the details of reuse.

In the end, it must be remembered that greatest value of the testplan or reuse strategy should go to those who are doing the work of validation.

6.5 Testplan Tools

As mentioned earlier, testplan tools and formats have evolved over time. Every testplan format or tool comes with its own benefits and constraints. This section discusses several options and factors to consider when evaluating or selecting a testplan tool.

6.5.1 Selecting and Deploying a Testplan Tool

Projects should research and decide on a POR validation testplan tool or format during TR. Evaluations should involve hands-on end-to-end trials of the proposed tool usage. When

applicable, testplan templates or outlines should also be developed and deployed so that the various validation teams on the project will be able to produce testplans with similar structure.

As discussed earlier, a testplan serves many purposes, and a high-quality testplan tool will be optimized for the most common activities while also supporting other anticipated uses. The following is a list of capabilities that Validators should look for in a testplan tool:

- **Testplan Entries:** At a minimum, a testplan tool should enable Validators to document *what* will be validated. Testplan authors should be able to create entries for test conditions that represent validation objectives. Tools should support written descriptions for testplan entries, and should allow Validators to mark testplan entries with project-specific fields (ideally the fields described in this chapter). Validators should be able to create testplan entries in ways that map well to execution.
- **Background Information:** Testplan tools should enable Validators to document *how* validation will be done. Testplan authors should be able to provide written background information that sets the context for understanding testplan entries.
- **Ease of Use / Maintenance:** Testplan drafting takes time and may require several iterations of writing. Validators will spend significant time writing and revising background information, creating new testplan conditions, revising testplan conditions and making a variety of edits and adjustments. Tools need to make these common tasks easy.
- **Reviewability:** Validators will share and review their testplans with stakeholders throughout the project. Testplan tools should enable Validators to organize, publish or display relevant testplan contents in an easy-to-review format.
- **Progress Tracking:** Testplan tools should enable Validators to track the status of testplan entries and validation progress. Tools should support appropriate status fields for testplan conditions that map to project milestones. Tools that provide traceability links between testplan conditions and requirements, content or coverage to produce indicators should avoid complexity and make the process of creating links easy to use and intuitive. Tools that extract status to create indicators should be automated and intuitive.
- **Inheritance:** Testplan tools should enable Validators to proliferate testplan conditions for use by similar projects. Refer to the section on Testplan Sharing and Reuse for more details and considerations.

Once a testplan tool is selected, provide training to cover project-specific usage of the tool. The project should give direction on testplan naming conventions and where testplans should be stored or maintained. The project should provide guidance on what sections of a testplan are required or what fields must be used for testplan conditions. Validators should understand how to create a new testplan, how to document background information, how to create testplan entries and how to share the testplan with stakeholders. Drafting due dates, review timelines and other expectations should also be made known to all teams at the end of TR or beginning of VAL 0.0.

6.5.2 Lightweight Options

The most lightweight approach to write testplans is to use established word processor or spreadsheet applications like Microsoft Word or Excel. These applications are efficient for rapidly drafting and maintaining documents, but each has pros and cons.

Word processing applications are good for creating testplans that describe background information and testplan entries in an outline structure with written prose. However, it can be a challenge to identify a format for testplan entries that displays the title, description and fields in a nice page view. Furthermore, formatting and style can be inconsistent across teams, so projects will often provide a template to use. Another drawback of testplans written entirely with word processing applications is that it can be difficult to extract data from a written document to produce indicators.

Spreadsheet applications also have advantages and disadvantages when writing testplans. They work well for capturing lists of testplan entries and can be useful for generating indicators. However, they have inherent limits on formatting content for reviews, since all written text needs to fit into cells – even background information.

6.5.3 Advanced Options

Advanced testplan tools have been built from associative databases, where a database record represents each testplan entry. The database records usually support a written description and various information fields. This approach is useful for enabling testplan-based indicators and provides a framework for linking things like coverage and tests to testplan entries for traceability.

Testplan tools based on a database must provide a way to create, edit, organize and view test condition records by validation scope. This requires custom software applications to work together with the database, and often involves a specific data usage model. This can introduce formatting or structural constraints on a testplan, and may even limit the validation approach in a way that is not consistent with the validation strategy.

Advanced testplan tools may come with unforeseen costs. Teams should ensure that the investment efforts are worth the return. For example, testplan tools with complex data usage models may require significant training. Tools might allow users to create testplan entries with traceability links, but might provide a poor user interface for these tasks, or lack a robust solution for reviewing contents. Pre-Silicon validation on the Broxton project suffered from the effects of a poor testplan tool. Although the tool was a converged solution supporting many features, the user interface was slow, the data usage model was complex and it did not support the anticipated mode of tracking progress for Pre-Silicon validation. Eventually, these issues formed an impassable barrier for Validators and managers. The difficulty of using the tool outstripped its utility, and it was abandoned. Teams reverted to using simple spreadsheets for all practical work and tracking. The next project opted to use lightweight testplans for their POR. The lesson became clear: if the testplan tool is bad, other means will be used.

7 Summary

Written testplans provide a structured roadmap for validation execution. They are also an important tool for communicating validation objectives and measuring progress. The benefits of written testplans extend beyond the current project. Validators of all levels of expertise are encouraged to participate in testplan writing, reviews and execution.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 14

Regressions

By: [Erik W. Berg](#) and [Lance Geiger](#)

1 Abstract

Regression testing is a method of verifying that a code base (such as an RTL model) still behaves as intended after a modification has been made and that the previously attained level of health has not “regressed”. At Intel, the term “regression” is also somewhat misused to describe “stress” or “bulk” runs, which are more extensive test suites, focused on finding bugs in the code base that were not caught by the turnin-gating regressions. This chapter describes both types of regressions and the tradeoffs to consider when constructing, maintaining and optimizing these regressions.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	05/14/2004	First document for review	Erik Berg	Matt Kupperman / DPG-N uAV MWG
1.1	10/11/04	Third document for review	Erik Berg	Matt Plavcan
1.2	12/16/04	Reverted to the original document using the new MWG template	Erik Berg	Steve Cegla
1.3	02/23/05	Incorporated agreed upon revisions	Erik Berg	Steve Cegla
1.4	03/18/05	Finalized document	Erik Berg	Steve Cegla
2.0	04/06/16	Major updates and expanded scope to include “stress” regressions	Lance Geiger	Michael Bair
2.1	04/26/16	Incorporated feedback from Art of Val review sessions	Lance Geiger	

3 Contents

1 Abstract.....	343
2 Revision History.....	343
3 Contents.....	344
4 Purpose.....	346
4.1 Why do we need this chapter?.....	346
4.2 What does this chapter cover?	346
4.3 What does this chapter not cover?	346
5 Background Concepts	346
5.1 Netbatch	346
5.2 CPU time and Wall clock time.....	347
5.3 Compute Cost/Footprint.....	347
5.4 Turnin-Gating Regressions	347
5.5 Stress Regressions.....	347
5.6 Directed vs. Random Tests.....	347
5.7 Seed	348
5.8 GateKeeper	348
5.9 Granite.....	348
6 Turnin-Gating Regressions	348
6.1 Constructing a Turnin-Gating Regression.....	349
6.1.1 Purpose of a Turnin-Gating Regression.....	349
6.1.2 What Should a Turnin-Gating Regression Cover?	349
6.1.3 How Extensive Should a Turnin-Gating Regression Be?.....	349
6.1.4 What Makes a Good Turnin-Gating Regression Test?	350
6.1.5 What Makes a Good Turnin-Gating Regression Testlist?	351
6.1.6 Building a Turnin-Gating Regression Over Time	352
6.1.7 Turnin-Gating Regression Tradeoffs and Gotchas	352
6.2 Organizing a Turnin-Gating Regression	354
6.2.1 DUT-based Partitioning.....	354
6.2.2 Team-based Partitioning.....	354
6.2.3 Feature-based Partitioning.....	355
6.2.4 Regression Levels	355

6.2.5	Documentation.....	356
6.3	Maintaining a Turnin-Gating Regression	356
6.3.1	Handling Failures.....	356
6.3.2	When to Add New Tests	358
6.3.3	When to Remove Old Tests.....	358
6.3.4	Optimizing Compute Usage	359
7	Stress Regressions.....	360
7.1	Constructing a Stress Regression	360
7.1.1	Purpose of a Stress Regression	360
7.1.2	What Should a Stress Regression Cover?.....	360
7.1.3	How Extensive Should a Stress Regression Be?	361
7.1.4	What Makes a Good Stress Regression Test?	361
7.1.5	What Makes a Good Stress Regression Testlist?.....	361
7.1.6	Building a Stress Regression Over Time	362
7.2	Running a Stress Regression	362
7.2.1	Manual vs. Automated	362
7.2.2	Periodic vs. Endless.....	363
7.2.3	Keeping Up With Debug	364
7.2.4	As an Extension of Turnin-Gating Regressions	364
7.2.5	Revalidation	365
7.3	Maintaining a Stress Regression	365
7.3.1	Failure Triage.....	365
7.3.2	When to Add New Tests	365
7.3.3	When to Remove Old Tests.....	365
7.3.4	Optimizing Compute Usage	366
7.3.5	Disk Space Management.....	366
7.3.6	Reruns	367
7.3.7	Reference Runs.....	367
7.3.8	Indicators	367
8	Summary.....	368
9	Future Work.....	368
10	References.....	368

4 Purpose

4.1 Why do we need this chapter?

Turnin-gating regressions are a key component of the RTL model “turnin” process, which is a list of checks and processes every proposed changeset must run and pass before it can be “accepted” into the RTL code repository. These regressions are the last line of defense preventing bugs from making it into the RTL model. Stress regressions, on the other hand, are typically run on released RTL models and are the validation team’s primary mechanism for finding bugs already in the RTL model. As discussed in [Introduction to Pre-Silicon Validation section: Hardware change cost model](#), it is desirable to find bugs as early as possible, so ideally the entire suite of validation tests would be run as part of the turnin process. However, turnin-gating regression tests are often the long pole for RTL turnin latency and overall compute usage, thus the validation team is under constant pressure to reduce the compute footprint and run time of these regressions. Creating and maintaining efficient turnin-gating regressions that strike the right balance between depth and compute usage is an ongoing challenge for all Validators. Further, supplementing the turnin-gating regressions with more extensive stress regressions and achieving high levels of automation is critical to minimizing the effort to run, manage and debug regression failures throughout the life of the project.

4.2 What does this chapter cover?

This chapter covers the tradeoffs in the construction and maintenance of regressions, both as part of turnin-gating checks as well as those run on released RTL models. It also covers guidelines for organizing regressions and automating various aspects of regressions in order to reduce overhead.

4.3 What does this chapter not cover?

This chapter does not cover other aspects of turnin-gating checks such as lint checks, timing and synthesis checks, etc.

5 Background Concepts

5.1 Netbatch

Netbatch³³ is Intel’s proprietary distributed computing platform that allows the submission of many jobs in parallel that can run on various “pools” of hosts around the world. Rather than running each test in a regression suite serially on the same machine, Netbatch enables running all tests in parallel across hundreds or thousands of machines.

³³ <https://intelpedia.intel.com/Netbatch>

5.2 CPU time and Wall clock time

The most common metrics for measuring process execution time are “CPU time” (how long the process spends actively running on a CPU core) and “Wall clock time” (how much time elapses from when the process starts until it completes). For single-threaded processes, CPU time will be less than Wall clock time as some time is spent by the process doing I/O or waiting to become active. For multi-threaded processes, the CPU time can be greater than Wall clock time.

5.3 Compute Cost/Footprint

The primary aspects of a simulation that impact how much it “costs” to run via Netbatch are execution time (both CPU and Wall clock), memory usage (both average and peak) and disk space usage. While execution time is usually the primary focus of regression optimization, Validators should also be aware of and try to minimize memory and disk usage as well.

5.4 Turnin-Gating Regressions

Regressions that are run as part of the GateKeeper turnin process and must pass in order for the turnin to be accepted are considered “gating” regressions. Because every turnin must run these regressions, their compute footprint (especially wall clock time of the longest test) must be minimized.

5.5 Stress Regressions

In addition to the turnin-gating regressions, validation teams often have more extensive (and more random) “stress” or “bulk” regressions, which they run on released models (or on side models prior to turnin). This allows Validators to have more and much longer running tests than what is feasible in a turnin-gating regression. Although not technically “regressions” in the strict sense of the word (as they may be testing new features that were not previously working in the code base), much of Intel refers to these type of runs as “regressions”, as does this chapter.

5.6 Directed vs. Random Tests

Directed tests do the same thing every time they are run. In this respect, they are ideal for turnin-gating regressions as they ensure the same behavior is being checked from model to model, however they may not be representative of the bulk of tests the validation team runs in their more extensive regressions.

Random tests change some aspect of their behavior every time they run. In order to be used in turnin-gating regressions this randomness must be made reproducible using a seed (see next section).

For more info, see [Stimulus section: Directed Stimulus vs. Random Stimulus](#).

5.7 Seed

A seed³⁴ is a number that is used to initialize (or “seed”) a pseudo-random number generator. For example, using an initial value of “5”, a pseudo-random number generator may produce the sequence “5,3,6,1,0,7” which appears “random” but can be reproduced perfectly at a later time by initializing the sequence again with “5”.

In the context of random tests, using the same seed is necessary, but not sufficient to reproduce the same test behavior and configuration across multiple runs. It should be noted that using the same seed across different RTL models can still result in different test behavior and configuration, so tests in turnin-gating regressions should typically take additional measures to ensure consistent behavior, as discussed later in this chapter.

5.8 GateKeeper

GateKeeper³⁵ is an Intel developed tool for performing continuous integration of Intel CPU and SOC projects, including running turnin-gating regressions on incoming changesets (called “turnins”) and building and releasing models on a set schedule. GateKeeper is the primary mechanism by which DDG runs turnin-gating regressions.

5.9 Granite

Granite³⁶ is an Intel developed tool for automating stress regression submission and managing test command lines. It consists of several major components including a database (HSD), a frontend GUI and a dispatch engine and associated command line tools. Granite is the primary mechanism by which DDG manages and runs stress regressions.

6 Turnin-Gating Regressions

Turnin-gating regressions are the last (and sometimes only) line of defense preventing new bugs from being accepted into the RTL repository. They are also often the best way to get quick feedback on whether changes to a side model have broken any existing functionality. While it is not realistic to expect these regressions to catch every possible new bug (or even most bugs), it is critical to ensure that they at least catch all high-impact bugs - those which would cause the code base to take a significant step back in overall health or impede the validation team’s ability to make forward progress. Turnin-gating regressions help the validation team “lock in” model health that would otherwise fluctuate wildly and make achieving “tape-in” quality impossible.

The remainder of this section discusses the various tradeoffs to be considered when creating, organizing and maintaining a project’s turnin-gating regressions.

³⁴ https://en.wikipedia.org/wiki/Random_seed

³⁵ <https://dtspedia.intel.com/Gatekeeper>

³⁶ <https://dtspedia.intel.com/Granite>

6.1 Constructing a Turnin-Gating Regression

6.1.1 Purpose of a Turnin-Gating Regression

The purpose of a turnin-gating regression is to “lock-in” whatever goodness currently exists in the code base and prevent new changes from breaking that existing functionality. This model of work is also known as “always alive” or “continuous integration”. The content of a turnin-gating regression should ensure that subsequent changes to the code base result in a monotonically increasing level of capabilities and health.

6.1.2 What Should a Turnin-Gating Regression Cover?

In the context of pre-silicon validation, the content of a turnin-gating regression is typically centered around ensuring all key RTL features and IPs are covered by one or more simulation and/or emulation tests. However, turnin-gating regressions should not be limited to just RTL features but rather anything in the code base that the validation team relies on. This includes: key testbench functionality, test generators, BFM^s and Verification Components (VCs), FSDB/waveform generation, watch windows, “no force” checks and any other critical environment or simulation/emulation collateral. In recent years, great strides have also been made in enabling FPV proofs as part of the turnin-gating regressions as well as testbench unit tests, so these items should also have some representation in the turnin-gating regression.

6.1.3 How Extensive Should a Turnin-Gating Regression Be?

In a perfect world, anyone making changes to the code base would fully exercise those changes and ensure they did not break any legacy features before submitting a turnin to GateKeeper. In such situations, the turnin-gating regression could be extremely small or possibly even non-existent.

On the other extreme, some engineers may do no testing of their changes (even compilation) before submitting to GateKeeper and rely solely on the turnin-gating regressions for feedback on any typos or bugs that may exist in their changesets.

The reality for most teams is that the size of the code base and amount of churn in the code is too large for even well intentioned engineers to be aware of and test all legacy features before every change. Therefore, some level of turnin-gating regression is always required, but how much is really needed?

Validators should keep in mind that the goal of the turnin-gating regression is to prevent **high-impact** bugs from getting into the code base. Attempting to prevent all (or even most) bugs through the turnin-gating regression is futile and often counterproductive as the increase in GateKeeper turnin duration and additional debug requests from failed turnins can slow the entire team down. At SOC level, it must be assumed that IPs are already validated so testing can be limited to catching high-impact integration bugs (i.e. connectivity and concurrency issues). For these reasons, Validators should typically err on the side of minimizing the turnin-gating regression. Focus

should instead be placed on enabling designers to easily exercise their changes prior to turnin and to maintaining a high quality stress regression that can quickly catch any bugs that do make it into the code base.

6.1.4 What Makes a Good Turnin-Gating Regression Test?

A “good” turnin-gating regression test has many or all of these attributes:

- **Directed & reproducible:** The test should consistently exhibit the same behavior across different RTL models; not only in the transactions it generates but also in the configuration it uses. A fixed seed is typically not sufficient; key configuration knobs should be constrained on the command line (or in the test) and not rely on the seed. Where possible, testbench randomness that is not directly applicable to the feature under test should be disabled. Tests should also never consume files or data outside of the RTL model unless properly versioned to ensure reproducibility.
- **Stable:** In addition to being reproducible, a turnin-gating regression test should be extremely stable as measured by a lack of “false” failures. For example, running 100 seeds of the test on a healthy model should result in a 100% pass rate. Randomness in the test or configuration as well as imprecise self-checking can lead to “flaky” tests, which intermittently fail. Sometimes false failures can be a result of general environment or testbench instability (not specifically in the test). In such cases, the Validator should strive to limit the RTL & testbench configuration used by the test to avoid areas of instability as much as possible.
- **Clear purpose:** The test should have a well-defined purpose, including a clear name (or run mode/dirtag for templates). At SOC level, tests that are primarily targeting a specific IP should make this very clear in their names. Not only does a clear purpose help with debug and identifying what change may be responsible for the failure, but it also helps identify what value would be lost if the test needs to be disabled or removed from the regression. In addition to the test name, assigning every test a unique test ID can help locate specific tests easily in regression failure logs, as well as indicators.
- **Runs fast with minimal memory/disk footprint:** Shorter tests mean quicker turnaround time for turnins and more Netbatch capacity for stress regressions. That said, some flows or features dictate long running tests. In these cases, it is important to identify areas where you can make some concessions, such as reset accelerators or running on a smaller/faster DUT. This may involve having several tests instead of one, but typically minimizing the run time of the longest test in a regression is more valuable than reducing the total compute usage across the entire regression. It is also important to ensure tests do not inadvertently impact run time by using unneeded simulation switches (like `-debug_all`) or generate gigantic log files (due to using `-ok_error` instead of disabling checkers/SVAs).
- **Catches high-impact bugs:** Perhaps the most important (but often overlooked) attribute of a good turnin-gating test is that it routinely catches high-impact bugs. Tests that never uniquely fail a turnin are not particularly valuable. Tests that only catch minor bugs that would have been caught by the stress regressions within a few days and easily fixed are still useful but not as valuable. Prime areas for turnin-gating tests to target are any feature that is often overlooked when just doing basic exercise, such as power flows, security features or performance/latency requirements. Sometimes very simple tests (like a test that

does nothing but ADDs or sends the cores into the weeds with invalid opcodes) can be effective at catching bugs. Validators should put some thought into what they expect each turnin-gating test to catch, and what the impact of potentially allowing a few bugs into those areas would really be.

- **Protects something of value:** Even if a test does not routinely catch high-impact bugs, it may be protecting a feature or area that would cripple validation forward progress if it were to break. In such cases, it is valuable to keep these tests in the regression, even if bugs are not being routinely found in those areas. The “value” can also be debug time. Sometimes a small number of very simple “smoke screen” tests (although not incredibly useful in their own right) can quickly identify when the model is very badly broken and save engineers from diving into debug of the more complex regression tests.
- **Flexible & adaptable:** Just as we add defeatures or chicken bits to hardware features, turnin-gating regression tests should have a set of survivability knobs. These knobs can enable/disable specific portions of the test or control which features/configurations are utilized or which checks are enabled. This type of flexibility can help avoid the “all or nothing” decision when a test starts failing and needs to be disabled.

6.1.5 What Makes a Good Turnin-Gating Regression Testlist?

A “good” turnin-gating regression testlist is of course made up of “good” turnin-gating regression tests (see above), but the sum total of its parts will also exhibit many of these attributes:

- **Clear purpose:** Just like the individual tests, a regression list should have a well-defined purpose. It should be obvious why a test would or would not belong in this list and what types of tests the list contains. The overall pass rate of a list should have some meaning (health of a particular area or feature set). A common example of lack of clear purpose is when the same random test template is added to a regression many times with several arbitrary seeds. A better approach is to identify several key areas/features that need to be targeted and force each instance of the template to target one of those (via command line options or run modes). This also makes it clear to anyone reading the testlist why there are multiple instances of the same test in the list.
- **Wide breadth:** The regression should cover all critical features for its intended area or feature set. Running a list should be sufficient to gain confidence that no major bugs were introduced in that area or feature set.
- **Minimal overlap:** Each test in a regression should have a well-defined and unique purpose. When multiple tests in the regression cover the same area, it becomes difficult to understand the impact of removing/disabling a test and makes it harder to understand what exactly broke when numerous tests fail at the same time. Ideally, most GateKeeper turnin failures in your area should only have a handful (or even one) test failure. If you often see that a majority of your regression tests all fail together, it is a sign that your regression has a lot of overlap and should probably be optimized. Having significant overlap across regression tests can greatly impede debug as many tests will first fail with the same signature and several rounds of onion peeling may be required to identify and fix all issues. Uniquely targeted tests enables more parallel and focused debug.
- **Customized for target audience and concentration of change:** Regression lists should be tailored for each project or team and the historical (or expected) areas of change. For

example, a small IP with only a handful of engineers making changes can get away with a very small and targeted regression list, assuming the team is showing due diligence in exercising changes prior to turnin. A derivative SOC project that is only modifying a few specific features should bias the regression to have more testing of those areas of change, but should also be careful not to eliminate testing of legacy areas as many derivative projects scale back or eliminate validation of these legacy areas.

6.1.6 Building a Turnin-Gating Regression Over Time

A good turnin-gating regression for a mature and healthy code base will look much different than a turnin-gating regression for a code base undergoing extensive change or major new development.

When your code base is undergoing heavy change or starting from an unhealthy state, it can be extremely difficult to identify stable & reproducible tests for the turnin-gating regression. In such cases, the focus should be on establishing “footholds” that lock in some level of health. For example, a big new feature may still be extremely buggy and a test may only pass 70% of the time based on the seed. In such a scenario, a reasonable tradeoff may be to add some of the passing seeds to the turnin-gating regression or disable some checking or assertions, as long as all stakeholders are aware of the situation and agree to help with debug as false failures occur. Not only will this establish a foothold to prevent a complete backslide in model health (i.e. 0% pass rate), but will also act as a forcing function for Design (and Validation) to actively debug failures (as they will now start preventing new turnins).

As a code base matures, Validators should look to stabilize the regression and remove any workarounds that are no longer required. Any duplication or overlap of initial turnin-gating tests should also be scrutinized and cleaned up. For example, a “NOP-HLT” test may have been very useful at the beginning of a project, but is likely completely redundant in later stages. Validators should also look to start adding more complex (and potentially random) tests to the regression. Major validation milestones (i.e. VAL0.5, VAL0.8, and VAL1.0) are good times to audit the regression and improve it.

Another item to consider is documenting planned future additions to a testlist. If you expect a particular feature or area to eventually have several tests (but they are currently not healthy enough to add to the regression), you should add them (commented out) or at least a note to that effect. This helps communicate the intent of the testlist and serves as a reminder and tracking mechanism to ensure those tests do eventually get added.

6.1.7 Turnin-Gating Regression Tradeoffs and Gotchas

As with any engineering discipline, constructing a turnin-gating regression has many tradeoffs and a team can rarely achieve the ideal state. That being said, any concessions or back-offs that are made should be explicitly called out and periodically reviewed. Some of the common tradeoffs and gotchas that the DDG team has encountered over the years are discussed below:

- **Piggybacking:** A tempting optimization to make in your regression lists is to “piggy-back” or utilize the same regression test to cover multiple features or configurations. For example, maybe all of your existing tests pick a random DDR technology but you want to ensure at

least one always picks DDR4. Instead of adding a completely new test to the regression for this purpose, it can save compute if you instead identify an existing test that does the type of transactions you need and update its command line to always force DDR4. The downside to piggy-backing is that tests covering many different features are now much harder to remove or disable without losing substantial regression coverage and debug also becomes less straightforward as the test intent becomes muddled. Also, many commonly piggybacked features can add significant run time to the test (UPF, XPROP, frontdoor, no forces) so care should be taken to select tests that are not already a “long pole” in the regression. Finally, piggybacking many unrelated features on the same test makes it difficult to identify proper stakeholders for when a test needs to be changed or disabled. Validators must always balance optimizing for compute cost with flexibility and the effort to maintain a regression list.

- **Shared arguments:** Sharing command line arguments (via +options/defaults in ifeed format lists) across tests or even across testlists can make command lines much more readable and reduces the occurrence of copy/paste errors. However, shared arguments also have several pitfalls that Validators should be aware of. If shared arguments span a large section of a file (or across multiple files) it may not be obvious to someone adding or modifying a test that these arguments apply. If shared arguments are used in a top level testlist which includes lower level testlists, those lower level tests lists may no longer be runnable standalone (as they are missing the arguments from the top level list). For these reasons, shared arguments are usually best utilized for “headers” or common arguments that are applied to every test (and rarely change) and for small sets of related tests where the shared arguments are easily visible and managed (i.e. can fit in one screen of your text editor). Run modes or OCMs can also be useful for shortening common arguments.
- **Randomness:** In general, having any kind of randomness in the turnin-gating regression is undesirable as it makes failures difficult to reproduce. Randomness can also result in regressions suddenly failing for reasons unrelated to the change being made to the model. For this reason, directed tests (ideally self-checking) make the best candidates for turnin-gating regression tests (especially early on in the life of a project) as they will consistently exhibit the same behavior from model to model. But as a project matures, a validation team will typically move to more random-based testing in their stress regressions, which makes directed tests in the turnin-gating regressions less representative of what the validation team is running day-to-day. At this point, it is usually appropriate to add random tests to the turnin-gating regression, but they must be added with a fixed seed (usually enforced with a lint check), to ensure the “randomness” of the test behaves the same every time the test is run. Even “directed” tests still require a fixed seed as many parts of the environment (such as clock frequencies and memory configuration) are randomized independently of the test and need to be locked down for reproducibility.
- **Reliance on seed:** Validators should also be aware that a fixed seed does not guarantee identical behavior across models. While modern simulators like VCS go to great lengths to ensure consistent randomization across different models with the same seed, some changes (especially use of \$urandom) will still cause a divergence in behavior across models, even with the same seed. For this reason it is important that Validators run a candidate test with many different seeds before adding it to a turnin-gating regression, to ensure that subtle changes in the behavior of a fixed seed do not cause the test to suddenly start failing. These problems can be mitigated by constraining major aspects of the test

configuration or behavior on the command line (or run modes), essentially hardcoding the random behavior. Another approach is to take the output of a random test generator (such as Cafe) and use that as a directed test (with all transactions hardcoded). Finally, ensuring reproducibility is not just limited to the seed. Any collateral that a test uses from outside of the RTL model needs to be properly versioned and the turnin-gating regression test needs to always request an explicit version. This can include test libraries and build tools, pre- and post-processors and the test code itself.

- **Seed diversity:** One pitfall to avoid when using fixed seeds in a regression is to avoid using the same seed for every test. For example, even if you randomize the DDR technology in the test environment, if every test in your turnin-gating regression uses “-seed 1”, they will all end up picking the same DDR technology and you will have no coverage of the others (except for tests which explicitly pick other technologies). It is best to use a unique seed value for every test in your regression to get the most breadth of configuration coverage. If your tests already have unique “test IDs”, one simple trick is to use that test ID as the seed.
- **False passes:** With the focus on ensuring test stability and avoiding false failures, Validators often overlook an even more dangerous issue: false passes. Passing test results are rarely examined and are usually deleted by default. Validators must consider how they can ensure their turnin-gating tests are actually doing what they expect. Can self-checking be added to the test? Can collecting coverage on some key conditions in your turnin-gating regression help? If a test requires manually inspecting log files or waveforms to confirm its behavior, auditing this should be a required item for each major validation milestone (i.e. VAL0.5, VAL0.8, and VAL1.0) and consideration should be given to writing a post-processor, which can automate this analysis.

6.2 Organizing a Turnin-Gating Regression

As the complexity of modern SOCs grows, so does the size and complexity of the turnin-gating regressions needed to protect the model health. Having a well-defined regression organization strategy is essential to managing and maintaining these regressions.

6.2.1 DUT-based Partitioning

At minimum, a project’s regressions typically need to be broken into separate lists per DUT at SOC level (and possibly even at some subsystem or even IP levels). This requirement is usually enforced by the turnin process (such as GateKeeper) which needs to understand dependencies between stages of the model build and regressions. In addition, different DUTs typically have different compute footprints and simulation speeds, so this partitioning aligns nicely with assigning Netbatch resources to each regression list.

6.2.2 Team-based Partitioning

For larger DUTs (such as fullchip or SOC level), it often makes sense to partition the regression lists further by team or IP. This reduces the frequency of merge conflicts from unrelated changes

and makes it easier to identify test ownership. In addition, it helps breaks a large testlist into smaller, easier to understand pieces. It can also make sense to partition regression lists of smaller DUTs and IPs when different teams own certain features/areas of the IP (such as DFX or HVM features). Most testlist formats allow regression lists to “include” other lists, so you can maintain team-based partitioning while still having the convenience of a single top-level list.

6.2.3 Feature-based Partitioning

It is advisable to always group tests that cover a similar area/feature together in the regression list. For some testlist formats (such as “ifeed” format with +defaults/options) this enables sharing command line arguments across multiple tests (reducing the likelihood of typos and copy/paste errors) and also makes maintenance easier as you don’t need to go hunting around to see if you have another test that covers feature XYZ.

6.2.4 Regression Levels

As regression lists grow larger, it can be beneficial to break them into multiple “levels” of increasing compute footprint. For example, in an IP level repo you may be able to run the complete suite of tests as a turnin-gating regression, but when these tests are run in the SOC repo you may only want to run a subset due to the longer simulation times and memory usage.

For the sake of example, the DDG organization has often utilized three levels of regressions as follows:

- Level0 or “L0”: Basic/DOA tests that should be turnin-gating at every level. They also must adhere to the most strict compute limits (max run time of 2 hours, less than 100 total compute hours for the entire regression, as an example).
- Level1 or “L1”: More extensive and/or longer running tests that are often not turnin-gating at the SOC or fullchip level (but are turnin-gating at subsystem or IP level). They typically have more lenient compute requirements.
- Level2 or “L2”: Not an extension of L0/L1 but rather a duplicate of all the L0 and L1 tests with the seeds removed (and usually repeated four or more times). This list is not meant to be turnin-gating but rather to serve as a metric of how “stable” the L0/L1 tests are when the seeds are changed. This serves as an early warning mechanism for failures that will eventually show up in the L0/L1 turnin-gating regressions if the model changes enough to influence seed behavior.
 - NOTE: Some teams have also used “L2” (or even “L3”) as more of a “stress” regression (not necessarily just a repeat of L0/L1) that is not meant to be turnin-gating (or pass 100%) but rather a “kitchen sink” type regression to really stress the RTL. With the advent of Granite, most teams in DDG have relocated these “stress” lists to Granite (rather than residing in the RTL model) so this is now less common.

Having separate regression levels also enables designers to select the appropriate level to run against a side model before attempting turnin. For example, a very simple change would probably only need to run the “Level0” or bare minimum regression. More substantial changes would dictate running “Level1” and possibly “Level2” or even a few rounds of an appropriate “stress” regression.

Regression levels can also be used to optimize Netbatch throughput. One example that has been used by DDG in the past is to create a “level0_fast.list” which contains the longer running tests from Level0 and runs them on a separate and faster Netbatch resource. This can help reduce the overall turnin latency by shortening the run time of these “long pole” tests. Another example is that the DDG emulation team recipe waits for L0 to pass 100% before even launching the L1 to avoid wasted compute when there are L0 failures. For projects where Netbatch (or emulator) capacity is expense and limited, using levels to serialize the regressions like this can be a useful tactic.

6.2.5 Documentation

Just like source code, good comments in the turnin-gating regression can be extremely valuable in documenting test intent, ownership and temporary workarounds. And also just like source code, comments in the turnin-gating regression that are not kept up to date can cause confusion, wasted debug effort and actually be counter-productive. In fact, many engineers will still rely on “git blame” to identify a test owner even when comments are present in the testlist. So what is a Validator to do?

Again, just like source code, the best documentation is actually the code itself (not the comments preceding it). Giving your tests clear names can serve the same purpose as a good comment. In cases where a test template is being used (likely with a generic test name like “dunit_10_random”), Validators should instead use descriptive run modes (for ace format lists) or descriptive –dirtag values (for ifeed format lists) to document what this instance/flavor of the template is intended to cover (i.e. dunit_10_random ... -dirtag DDR4_2400_sOix_stress_xprop).

That being said, having at least some minimal comments in the header of every testlist to explain the overall intent of this particular list and the primary contact is still very valuable. This information usually changes much less frequently than the tests themselves and thus is less prone to becoming stale and out of date. Another strategy is to provide a link to a wiki or some other resource that is maintained more regularly than the testlist comments.

Along the same vein, it is also important that a test’s command line is readable and easy to understand. If methods such as run modes or OCMs are utilized to shorten the command line, those modes/macros should have very clear names that make it obvious what they do without requiring someone to look up their definition. By the same token, if a test forces a particular configuration via constraints or register writes inside the test itself, the test name should reflect this so that anyone using or debugging the test is not surprised by this test behavior.

6.3 Maintaining a Turnin-Gating Regression

6.3.1 Handling Failures

Regressions will eventually fail. When they do fail, Validators need to be prepared to facilitate debug and disposition of these failures. Ideally, the failure will be due to catching a real RTL bug that a designer coded (and must now fix), but regression failures can just as often be due to a test, testbench or general environment issue.

When the bug is due to a test or testbench issue, Validators must often fight pressure from designers to disable the test until the testbench can “catch up”. This is where having a solid understanding (ideally documented in the regression list itself) of the unique value each test adds to a regression becomes vital. In some cases, temporarily disabling a test to enable forward progress is the right decision, but all parties should understand the risks being introduced (including the potential impact to stress regressions) and a bug should always be filed to track re-enabling this test at a later time. In most cases however, the design and validation teams should be working together to exercise and enable features so a turnin attempt is not even made until all existing regression tests (including stress and possibly even newly added tests for this change) are passing.

Validators must also be prepared to push back when a failure does not appear to be related to the test or testbench. A common scenario is a designer has made a change in a completely unrelated portion of the design to the failure and prematurely concludes that the failure has nothing to do with his or her change and wants to push it through. This has the potential to hide a real RTL bug (even if not directly related to the change) and Validators should ensure a bug is always filed in this situation.

A common pitfall to avoid is “seed swizzling”. Ideally, all validation teams would ensure that tests added to a turnin-gating regression are stable (see discussion of “What Makes a Good Turnin-Gating Test?” earlier), but this does not always happen. In these cases, engineers are often tempted to simply change the seed of the failing test to one that happens to pass and move along. Not only is this potentially masking a real hardware issue, but is also just pushing the problem to the next unfortunate soul who runs into the same issue. Changing the seed of a test (especially a random test) should really be viewed as completely removing a test from the regression and replacing it with a potentially different behaving test. This is rarely a good option and in any case should result in a bug being filed to track restoring the regression to its previous state.

Another pitfall to steer clear of is setting the turnin-gating pass rate bar below 100%. If a particular feature is not completely healthy yet and has an average pass rate of 90%, it may be tempting to set the gate at 90% to avoid all the work of constantly debugging the other 10%. The problem with this approach is that you lose control over **which** 10% of the regression is allowed to fail. A particular changeset may break a critical feature that was previously very stable and happened to “get lucky” on the other tests to come in just over the 90% threshold. While there is no perfect solution to this problem, some better approaches include:

- **Temporarily move tests to post-release or stress:** If a particular set of tests is a bit unstable, moving them to a regression that is run after model release may be a decent compromise. These tests will no longer gate turnins, but the team will get quick feedback when they fail. However, this does require some commitment from the design and validation teams to vigilantly monitor and react to these results.
- **Keep in regression and deal with fallout:** In some scenarios (i.e. IP level or repos with a small number of users), it may work better to keep the flaky tests in the regression and have agreement between Design and Validation to support debug requests as they fail. This approach has the benefit of not allowing debug to be put off, but also requires a high level of support from key people to keep the turnin pipe flowing.
- **Disable SVAs/checkers:** If a test is failing due to a particular checker or SVA, temporarily disabling that check may be a reasonable compromise. This is especially true if many tests are capable of hitting the same failure. In any case, be sure to file a bug.

- **Utilize –fail2pass:** If a particular test fails in a certain way that is deemed OK (for now), another option is to write a fail2pass post-processor which inspects the log files to check how a test failed and converts it to a pass if it failed in the expected way. This has the benefit of automatically ignoring known issues, but stilling failing turnins that fail different tests or fail in an unexpected way. As with any temporary change, file a bug to track removal of this workaround once the feature is healthy.

6.3.2 When to Add New Tests

The obvious answer to “When should I add a new test to the regression?” is “whenever a new feature is coded or the existing logic is changed”. However, not all changes or features necessarily justify a new test. Another consideration is whether or not the change is modifying the behavior of the area/feature or just changing the underlying implementation. Changes which only modify the underlying implementation and are not visible outside the block (for example, gating a local clock when a queue is empty) likely don’t justify a new test and are sufficiently covered by existing tests, although each change must be analyzed on a case-by-case basis. New additions to the turnin-gating regressions should be periodically reviewed with the entire team so everyone is aware of the change and can ask questions if anything is unclear.

Another scenario that should cause a Validator to consider adding tests to the regression is the discovery of a bug. Validators must keep in mind that the point of the turnin-gating regression is not to prevent ALL bugs from getting into the model, but bugs that resulted in a significant dip in stress regression pass rate or heavily impacted other teams should be addressed in the turnin-gating regression.

Finally, coverage feedback is another useful method of identifying holes in the turnin-gating regression. If coverage shows that a critical feature or area is not consistently being hit by the tests in the turnin-gating regression, new tests should be added to address this.

6.3.3 When to Remove Old Tests

Validators don’t usually consider removing tests from a regression unless they are either causing failures (see “Handling Failures” above) or are causing the regression to consume too much compute (see “Optimizing Compute Usage” below). But it is just as important to periodically review your regression lists and prune older tests that are no longer adding value or have been superseded by newer tests.

Not only does this effort help avoid wasted compute, it can also improve your day-to-day work as there is one less test to run through Netbatch and consume disk space. When handing off to another team or derivative project, having a regression list bloated with old and less useful tests can cause unneeded confusion and effort as well.

Validators are often afraid to remove tests from a turnin-gating regression, especially when the testlist was inherited and the intent is not completely clear. In such cases using a data driven approach (such as analyzing which tests have uniquely failed GateKeeper turnins) can be useful in identifying candidates for removal. It is also important to try to identify all of the stakeholders

of a test before removing it. This is also another reason why having good documentation is important for regression maintenance and handoffs.

6.3.4 Optimizing Compute Usage

The compute footprint of the turnin-gating regressions should be regularly monitored to look out for both sudden spikes as well as gradual increases in test run time, memory usage and disk space consumption. Even in situations where the regression compute usage is holding steady, there are always optimizations that can be pursued to reduce the long pole time or memory usage of the regressions. Database and visualization tools such as PUMA³⁷ make it easy to create indicators with interactive graphs that allow regression compute usage to be analyzed and dissected. While the reasons for larger-than-desired compute usage can vary based on many factors, there are several common vectors that the validation team should pursue in the never-ending quest to optimize compute usage of the regression:

- **Reduce start-up and reset time:** A simulation can spend anywhere from several minutes to several hours just setting up the simulation environment and taking the RTL model through a reset sequence to prepare it for the actual test content (which is typically the useful part). Identifying ways to speed up this process (such as accelerating reset through testbench injection, using save/restore or “backdoor” control register accesses) has the added benefit of reducing the run time of ALL tests. To avoid missing bugs that are only caught by the “real” reset sequence, a handful of tests (which are not already the long pole) should be selected to run the full reset flow, without accelerators.
- **Prune the tests:** As described in “When to Remove Old Tests” above, Validators should always review the makeup of their regressions and identify tests which are no longer adding enough value to justify their compute usage. One strategy is to mine data from previous turnin attempts to identify which tests have never uniquely caused a turnin to fail over some period of time. These tests may be candidates for removal, although care should be taken to not remove the only test covering a critical area. Also, instead of complete removal, it may be more appropriate to relocate certain tests from the turnin-gating regression to the stress regressions. That way they are still run on a regular basis, but no longer impact the latency of the turnin-gating regressions.
- **Identify proper memory classes:** Regressions that run with too small of a Netbatch memory class may exhibit intermittent and hard to debug failures. In other cases, the tests may not fail but by consuming more than their allotted memory, may cause the host machine to thrash due to constantly paging memory in and out to disk, causing significant increases in wall clock time. Reviewing the actual memory consumption of tests vs. their assigned Netbatch class should be done on a regular basis. Tests which consume significantly more memory than the other tests in the same regression list should be moved to another list, or if the testlist supports it, use a directive to specify a higher memory class (such as the “.nbclass” directive in ifeed format testlists).
- **Run on faster machines:** Some projects and sites may have special Netbatch pools with faster machines (primarily better single threaded performance), although they are usually

³⁷ <https://intelpedia.intel.com/PUMA>

limited in number and reserved for special purposes. Running all the regressions on faster machines is not feasible, but if only a handful of tests are running much longer than the others, it may be possible to reduce the long pole by moving these long tests to the special Netbatch pool. Care must be taken, however, to not limit these jobs to **only** the special pool, as it may become full and the time spent waiting for a free machine may exceed the run time savings of the faster machine.

It has often been debated if model performance and test run time should itself be a gate to turnin. While it may be possible to detect turnins that cause sudden huge spikes (i.e. doubling of average run time) in regression compute, these spikes can also be caused environment issues like license server downtime or fileserver load. Failing incoming turnins based on these criteria could easily cripple a project's ability to make forward progress. Another issue is that most model performance degradation happens gradually or only for certain features/IPs and the overall SOC performance does not spike enough to trigger an alarm. For these reasons, having solid indicators (like PUMA mentioned above) that are frequently monitored and acted upon along with doing periodic VCS profiling to identify low-hanging fruit for optimizations is usually the best method for addressing model performance.

7 Stress Regressions

For the purposes of this section, “stress regressions” refer to any testlists that a validation team runs in addition to the turnin-gating regressions. The rest of this section discusses the unique tradeoffs that must be considered when constructing, running and maintaining stress regressions.

7.1 Constructing a Stress Regression

7.1.1 Purpose of a Stress Regression

Unlike turnin-gating regressions (which are intended to prevent backslide of existing code base health), stress regressions are intended to help measure and drive up the health by finding bugs in the code base as well as driving coverage. The intent of a stress regression should be to catch ALL bugs in the code base (or at least all those targeted by pre-silicon validation).

7.1.2 What Should a Stress Regression Cover?

In short, the stress regression should cover everything that is reasonable to test pre-silicon. In particular, any features or areas not covered (or covered lightly) by the turnin-gating regressions should be a primary focus of the stress regressions. Every aspect of validation including power management, security, performance, negative testing and DFX/HVM features should have representation in the stress regression as well. In addition, stress regressions will typically have a high percentage of random tests and should leverage any available Netbatch capacity to run more seeds of these random tests.

7.1.3 How Extensive Should a Stress Regression Be?

As mentioned above, a stress regression should really strive to cover everything that a Validator can think to test. It is probably more important to identify and document what is NOT tested by the stress regression, such as any features or flows that are assumed to be covered by another validation team, any features or areas that are only covered by turnin-gating regressions and any features or flows that require manual inspection.

Like turnin-gating tests, stress regressions can also be broken into multiple testlists or levels. Some tests make sense to run as often as possible while others may only need to be run periodically, so separating these into different stress testlists makes sense.

Validators should also consider whether to “overstress” the design by testing aspects outside the official POR, or in the case of SOC level, retesting aspects of the IP that should have been validated by the IP team. In general, overstressing can be a good thing as long as tests that go outside the POR are properly tagged (to avoid wasted debug effort). The primary benefit of overstressing is that when the POR inevitably changes (or the quality of IP is found to lacking), you already have covered some aspects of the newly expanded validation space.

7.1.4 What Makes a Good Stress Regression Test?

All of the attributes that make a good turnin-gating test (see “What Makes a Good Turnin-Gating Regression Test?” above) still apply to stress regression tests, although to a lesser degree. Because stress regressions do not have a requirement for a 100% pass rate, there is much less focus on making tests directed and reproducible, but stress tests should still have a clear intent and should focus their testing on the same feature/area from run to run. Just like turnin-gating tests, this focus greatly helps debug and understanding a test’s value to the regression. That being said, it can be valuable to also have some true “kitchen-sink” type tests in the stress regression that are completely unconstrained and free to randomize everything.

While compute usage is much less of a concern for stress regressions, it should still be considered. A test that runs for many hours without doing something useful is taking away Netbatch cycles from other tests, which could be more effectively finding bugs. Long running tests are also harder to debug and take longer to rerun or generate waveforms. Validators should periodically review stress regression indicators to ensure compute usage is reasonable.

7.1.5 What Makes a Good Stress Regression Testlist?

As discussed above, a good stress regression should cover every aspect of a code base that the Validator can think to test. Where appropriate, stress regressions should also be divided into multiple testlists. This can help engineers select an appropriate set of tests to run on a side clone to help exercise their changes.

Unlike turnin-gating regressions, Validators should not try to minimize overlap in the stress regression. Overlap can be good, especially when pass rates are lower and some tests for a particular feature or IP may not be consistently passing from model to model. That said, care should be taken to ensure every test in the stress regression has a clear purpose and is adding real

value. Duplication (especially in command line arguments) can make it harder to apply fixes or tweaks.

7.1.6 Building a Stress Regression Over Time

A stress regression should be created as soon as random test generators are reasonably healthy or there exists more tests than can fit in the turnin-gating regressions. High pass rates (or lack thereof) should not be a primary reason for enabling stress regressions. Even stress regressions with low pass rates can add value by establishing an indicator and showing progress over time.

Validators should also not wait to deploy automation of stress regressions. Getting the automation enabled at the beginning of the project maximizes the effort savings and can help identify any early environment issues so they can be fixed long before the automation becomes more critical to the validation team. It also helps standardize the stress regressions across teams and can provide management a consistent global view of code base health.

Unlike turnin-gating regressions where only a subset of new features or code changes result in adding a new test, for stress regressions, new tests should be routinely added as new features come online, new test generators are deployed or new testbench capability is developed.

In addition, the frequency of stress regressions will likely increase as a project matures. Once random test generators are generally healthy, teams should transition to “endless” (or at least daily) runs to maximize idle Netbatch capacity and help drive coverage.

When should a team stop running stress regressions? The frequency of stress runs will likely peak around the VAL1.0 milestone or just prior to tape-in. After this, stress regressions generally ramp down (perhaps to once a day or once a week), although this is highly dependent on the area (i.e. firmware validation may continue well after tape-in to validate new patches). As the next stepping or project ramps up, the bulk of stress regressions should shift to the new code base, however stress regressions should not be completely disabled on the previous project/stepping, usually until silicon has come back and sufficient confidence has been established in the health of the design. Finding bugs pre-silicon (even after tape-in) is still valuable and much more efficient to root cause and fix/workaround than debugging the same bug in the lab.

7.2 Running a Stress Regression

7.2.1 Manual vs. Automated

Running stress regressions manually (i.e. a Validator submitting his or her area’s weekly regression every Friday afternoon and checking on the results Monday morning) can initially seem like less effort than setting up layers of automation. However, the effort associated with managing and maintaining regression runs starts to quickly add up. Automated regressions are not always the best choice, but they should always be considered (even for simple usages and even when pass rates are still low), especially when the infrastructure is already in place and in use by other validation teams.

Automated regressions can ensure regressions are launched at a regular cadence, even when Validators go on vacation or get swamped and forget to run them. This becomes more important

for random regressions where getting as many runs through Netbatch as possible is desired. A less obvious advantage of automated regressions is that all of the knowledge needed to run that regression is implicitly documented. This allows other Validators and other teams to see how a particular regression is run (how often, which models, which command lines) and makes hand-offs much easier. Automated regressions also provide a mechanism for designers to run stress regressions on their side clones before trying to turnin, without requiring them to know all the intricacies and workarounds that are associated with running a particular regression list.

A downside of automated regressions is that, without manual intervention, they may launch when there is a known environment problem or run on a model with a bug that causes a huge failure rate (and potentially fills up a disk). Mechanisms can be put in place to mitigate these issues (such as nbfeeders automatically stalling submission when a disk fills up and cron jobs which automatically disable tests that are failing too much). However, from another perspective this can be seen as an advantage, as having automated regressions forces Validators to “bake in” all the tribal knowledge needed to effectively run a regression in the current environment into the cron job or regression scheduler itself, where they can be seen by other Validators and teams.

Another aspect to consider is how many Validators on the team will be actively managing the stress regressions. A common approach is the “one man (or woman) show” where one Validator on the team handles all aspects of the stress regression, including adding/removing tests. While this can relieve the other Validators from having to ramp on tools or content of the stress regression, it also creates a disconnect between these Validators and what is actually being tested. It can also diminish some of the value of a tool like Granite (where all tribal knowledge is baked into the command lines in the database). Generally, a better approach is to keep the entire team involved with at least maintaining the stress regressions, perhaps with one Validator being responsible for the initial setup. When that Validator goes on vacation or gets busy, the team is still fully enabled to check on and modify the stress regressions.

7.2.2 Periodic vs. Endless

The most basic form of automated regressions is to simply run a particular testlist on a set schedule (i.e. every Tuesday and Thursday at 8am). Granite refers to this type of automation as “periodic regressions”. The period of submission can also be based on other factors, such as when a new model is released. Periodic submission works well for regressions made of primarily directed tests, where running the same test on the same RTL model multiple times does not add any value.

For regressions containing primarily random tests, the desire is to run as many seeds of each test as is feasible, unless your team’s compute capacity is very limited or the model and/or environment health is so low that infrequent regression runs are already generating more debug than the validation team can handle. When operating in a healthy environment with sufficient compute capacity, “endless” runs are typically superior to periodic scheduling for random tests.

Endless runs operate similar to a grocery store employee stocking shelves at the store. The employee periodically walks down each aisle and checks if any shelves are in need of restocking. Using Granite as an example, the Granite process (known as a “geode”) will wake up periodically (i.e. every 5 minutes) and check the Netbatch pool to see how many jobs are running vs. waiting. If there are fewer waiting jobs than some threshold, Granite will “restock” the shelf by submitting another round of regressions and then go back to sleep. When it wakes up next time, the jobs it

just submitted will likely still be in the Netbatch wait queue, but after a few sleep/wake cycles those jobs will eventually drain into the run queue and Granite will again restock the shelf by submitting more jobs. In this way, Granite is able achieve much more job throughput than any human could be expected to achieve. And by utilizing lower priority Netbatch Qslots, a team can ensure that Granite jobs do no overwhelm the pool and prevent higher priority jobs (like turnin-gating regressions) from running. Tools like Granite are able to “soak up” any excess compute capacity automatically (even at different sites around the world), and usually achieve their peak throughput during nights, weekends and popular vacation times.

7.2.3 Keeping Up With Debug

As mentioned at the beginning of this section, the intent of the stress regression is to catch bugs. But bugs can only be caught if the team is paying attention to and debugging failures. If a team is so backlogged that stress regression failures are being routinely ignored, the frequency of the regressions should be dialed back (but not eliminated). Even in cases where failures are not being actively looked at, it can still be useful to run stress regressions on a regular basis just for the purposes of establishing indicators and monitoring the health of the code base and environment.

Validators should also set goals (typically as part of the validation milestones) to actively debug all of the recent failure buckets and file bugs or make fixes. Project-level indicators can also be a good forcing function to make sure appropriate time is allocated to address any stress regression failures. Where possible, Design and even Architecture should be pulled in to help with debug.

7.2.4 As an Extension of Turnin-Gating Regressions

As discussed in the first half of this chapter, turnin-gating regressions are intentionally limited in scope to balance the depth of the regression versus compute usage. However, stress regressions are not limited to these same constraints and can act as a much more thorough “gate” against potential changes to the model.

Because stress regressions are not automatically run as part of the turnin process, it is up to the designer or Validator making a turnin to manually run the appropriate stress regression on his or her side model before attempting a turnin. This will obviously delay how soon the initial code can be turned in, but can potentially catch a real bug that the turnin-gating regressions would have missed and thus avoid having to make one or more follow up turnins to patch the model.

Because stress regressions do not typically have a 100% pass rate, the designer or Validator must be careful to distinguish between failures that were caused by his or her change and failures that are already being seen on released RTL models. Automation can help here as tools such as “Isti” can compare the failure buckets of a manual regression run on a side clone to the same buckets being worked on in that area by the validation team and denote which buckets are “new” versus “seen”.

7.2.5 Revalidation

Stress regressions can also be a useful vehicle for revalidation - when little to no changes have been made to a particular area for the current stepping or derivative project but the validation team needs to at least do “no harm” validation. Having a fully featured stress regression can automate much of the revalidation process by simply pointing the stress regressions at the new code base. When using stress regressions for revalidation, great care should be taken to identify which areas or flows are NOT covered by the regression. Some aspects of the code base may require specialized directed testing or manual inspection and it should not be assumed that the stress regressions alone would cover all areas.

For the revalidation process to be efficient, it is vital that the previous project properly documented any known failure buckets and coverage holes in an exit review. This information helps determine if failures or coverage holes seen in the revalidation runs are unique to the code base or just repeats of issues seen on the previous model.

7.3 Maintaining a Stress Regression

7.3.1 Failure Triage

In addition to tracking how many tests were run and their pass rate, it is also important to track the failure signatures of each failing test, the root cause of those failures and any bugs that were filed as a result of this debug. Like regressions, automating failure triage is also important, especially as the number of tests being run (and the number of incoming failures) is increased. This topic is described in much more detail in the [Triage](#) chapter.

7.3.2 When to Add New Tests

Similar to turnin-gating regressions, new tests should be added to a stress regression whenever a new feature is added or the code base undergoes a significant change. Unlike turnin-gating regressions, the bar for adding a stress test is much lower as compute cost constraints are much lower for stress regressions and the cost of not catching a bug in the stress regression is much higher.

New tests should also be added (or existing tests modified) to the stress regressions based on coverage feedback. Any features or areas that are not being covered (or not being hit frequently enough) should be routinely addressed in the stress regression.

In addition, whenever a bug is found by a test outside of the stress regression (such as an IP bug escape to SOC), it should be analyzed and stress regressions should be enhanced to close the hole.

7.3.3 When to Remove Old Tests

Stress regressions typically do not have outside pressure to reduce compute usage like turnin-gating regressions, so it is even more important for the validation team to periodically audit the stress regression and identify tests that should be removed or disabled and to audit any disabled tests that may need to be reenabled.

The most common reason for removing (or temporarily disabling) a stress test is that it is failing too much (either due to poor code base health or a test/testbench bug). When disabling a test, it is important to make a note of why it was disabled and make sure some tracking mechanism is in place (either by filing a bug or having a checklist item for the next validation milestone) to ensure that the test is reenabled when appropriate.

The validation team should also periodically scrub the stress list to identify any tests that are no longer adding value to the regression. For example, tests added to the regression during the early stages of the project may have since been completely eclipsed by newer tests or test generators. Removing the baggage of old tests helps keep the stress regression more manageable and reduces the effort and confusion during project handoffs.

7.3.4 Optimizing Compute Usage

Optimizing compute usage of stress regressions is less about trying to fit the regression into some predefined limits and more about looking out for issues causing tests to take more compute than necessary. Although stress regression compute usage does not usually receive the amount of attention from project management that turnin-gating regression usage does, it is still important to optimize as it frees up more Netbatch capacity for you (and other teams) to run even more stress regressions.

Some common items that should be periodically reviewed include:

- **Proper memory classing:** Running your stress regressions in too small of a Netbatch memory class can result in intermittent (and hard to debug) failures and exponentially increase run time (especially for FSDB/VCD reruns). On the other side, using too large of a memory class can cause unnecessary wait times and limit the amount of throughput you can achieve. Memory usage of your stress regressions should be periodically reviewed and adjusted as needed.
- **Are all tests completing?** Most test failures are reported and will show up in a tool like Triage. But some failures (such as command line problems) can cause a test to fail before it has a chance to properly write back a result. Also tests that consume too much memory or run too long may be killed by Netbatch monitors and may not properly write back a failure result. This is why it is important to periodically check and make sure all enabled tests have recent passing results. You may find that some tests have not been passing for some time but nothing showed up in Triage.
- **Test run time:** It is also good to periodically check the average and maximum run time of your stress regression tests. You may find that tests are running much longer than expected, often due to improper memory classing or unneeded simulator switches such as “`-debug_all`”.

7.3.5 Disk Space Management

Disk space in the Intel Engineering Computing environment is expensive, as it must be accessible from any machine on the network, requiring numerous file servers to host the disks. For this reason, Validation must constantly maintain their disk usage, lest their disk fill up, preventing further regressions from running and often resulting in bizarre failures and wasted debug effort.

Maintaining disk space can quickly become a large manual task if automation is not put into place. Because a validation team typically does not have enough disk space to run many weeks' worth of regressions, they must constantly purge older runs, but at the same time be careful not to delete any runs that are still being actively debugged. If an automated regression tool is being used that generates database records for each failure, tools can be used which automatically delete failure directories after a failure has been marked as "debugged" and keep around areas that are marked "wip". Although simple in concept, this type of automation can do a great deal to prevent disks from filling up without requiring constant manual intervention.

7.3.6 Reruns

Another task often related to maintaining regressions is generating trace files (such as FSDB or VCD) for failures or rerunning failures with different command line options (such as increased log verbosity). Managing this manually does not scale beyond more than a handful of reruns per day. Automation here can ensure that failures that occur overnight or during the weekend will have a FSDB ready and waiting the next morning when the Validator comes in.

Putting together automated regression submission, failure triage and disk cleanup along with automated test rerun creates an enormously helpful cycle which frees up Validators to do actual debug work, instead of baby sitting Netbatch jobs or frantically trying to clean up disks that have filled. The DDG team has seen a significant rise in debug capacity and efficiency with all of these measures put into place and highly recommends all teams do the same.

7.3.7 Reference Runs

It can often be useful to refer to a passing "reference" run when debugging a failure to check how it "should have worked". The problem is that most test run tools delete run directories when the test passes (to save disk space) and Validators very rarely have passing FSDBs just lying around. For this reason it is a good idea to purposefully create a "reference run" regression which simply runs a list of healthy tests along with trace files and/or more verbose logs so that Validators always have an area containing a variety of passing tests they can refer to. Again, automation here can eliminate the manual effort both with running and cleaning of the disk space of these runs so that they do not impact other stress regressions.

7.3.8 Indicators

Indicators are an important piece of managing stress regressions as they provide a summary of the current health (pass rate) of your regression tests, and by proxy, a key vector of the overall health of the code base. Indicators also provide historic data to help Validators identify trends and can act as one piece of data into predicting tape-in readiness.

Automated regression tools usually provide some built-in mechanism (usually a database) for tracking the status of current and past regressions and even manual regressions can sometimes leverage this infrastructure as well. Using Granite again as an example, all test runs will generate a database record (known as a "teststat" record) that captures all the useful information about the test run including pass/fail status, date run, model and host it ran on, execution time and command

line. Database queries can then be created to generate any number of indicators including pass rates over time, number of tests run, etc.

Having standardized indicators across all teams on a project helps identify areas that may need more attention and can help highlight larger trends regarding DUT or environment health. In addition, standardized indicators can help teams avoid common pitfalls by leveraging database queries to identify tests that are not being run or are being run but never completing.

8 Summary

Both turnin-gating and stress regressions are critical components of the validation team's day-to-day work and both have unique trade-offs that must be considered. The first half of this chapter covered the key aspects of creating, organizing and maintaining effective yet compute efficient turnin-gating regressions. This remainder of the chapter covered how adding automation to all aspects of stress regressions can greatly reduce manual effort, fully utilize available compute resources and enable Validators to focus more time on debug.

9 Future Work

This section intentionally left blank.

10 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 15

Triage

By: [Darrin Hancock](#)

1 Abstract

Debug is the single largest consumer of Validator time over the course of a project. Much of this time is consumed in Triage, the bucketing of failures and initial stages of debug. There are tools available that aid in triaging failures. These tools aid in bucketing failures, prioritizing debug, and sharing data across the project.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	5/31/2016	Initial Draft	Darrin Hancock	Michael Bair
1.1	8/15/2016	Completed Rev1 of Chapter	Darrin Hancock	Michael Bair

3 Contents

1 Abstract.....	369
2 Chapter Revision History	369
3 Contents.....	370
4 Purpose.....	371
4.1 Why do we need this chapter?.....	371
4.2 What does this chapter cover?	371
4.3 What does this chapter not cover?	371
5 Background Concepts.....	371
5.1 DTS TRIAGE Tool	371
5.2 Triagewb and Tagging Failures	371
5.3 Buckets.....	372
5.4 Characterizers or Bucketing Scripts.....	372
6 Triage	372
6.1 Triaging Failures.....	373
6.2 Creating Buckets of Failures.....	373
6.3 Finding Failures	375
6.4 Assigning Failures	376
6.5 Ownership of Failures.....	376
6.6 Dispositioning Failures.....	377
6.7 Timeliness of Debug	377
6.8 Automation.....	378
7 Summary.....	378
8 Future Work.....	378
9 References.....	378

4 Purpose

4.1 Why do we need this chapter?

During the life of a project, Validators will spend most of their work time debugging failures. Good triaging will more quickly get the failure to the appropriate experts and will help the inexperienced engineer start down the correct path of debug. Good triage practices will greatly save on debug time and avoid the high costs of having multiple engineers unknowingly debug the same failure.

4.2 What does this chapter cover?

This chapter describes methods used to characterize, organize and manage the countless number of failures encountered during the lifespan of a project. This chapter will demonstrate effective ways to triage failures that will minimize the amount of debug time per failure.

4.3 What does this chapter not cover?

This chapter will not cover the details of how to debug failures. Please refer to the [Debug](#) chapter. This chapter will not be an extensive user guide on how to use the DTS Triage tool, but will refer to usages of this tool as the Intel converged solution for bucketing and managing failures.

5 Background Concepts

5.1 DTS TRIAGE Tool

TRIAGE is a validation tool used by project teams to manage and organize their test and build failures. TRIAGE is maintained and developed by DTS. It is released as part of the Venus suite of validation tools. This chapter was not written to describe this tool or to be any type of user guide. This is a chapter written to describe methods and a mindset that enable Validators to be more effective as they debug failures. TRIAGE will be referenced in this chapter, as it is the Intel converged solution for bucketing and managing project failures. When referring to the TRIAGE tool in this document, I will use all caps. For more information on the TRIAGE tool, go to the triage page on DTSpedia.

<https://dtspedia.intel.com/Triage>

5.2 Triagewb and Tagging Failures

Triagewb is a companion script to the TRIAGE tool that is used to insert failures into the failure database. Triagewb is also maintained and developed by DTS. Validators use Triagewb to appropriately tag their failure entries in a way that makes them easy to find when doing database queries.

5.3 Buckets

A bucket is a container for similar failures. Buckets should be given names that are concise yet still describe in human understandable text the failures it contains. Ideally, a bucket should have a single “root caused” issue that is common for all the failures contained in the same bucket.

5.4 Characterizers or Bucketing Scripts

Characterizers or “bucketing scripts” is code written in a scripting language that is used to classify or bucket a failure. To simplify the writing of this chapter, I will refer to these as just characterizers. Often a failure signature is overly detailed and needs a concise classification. Other times, a failure signature is completely vague and code needs to be written to find information on the failure that can be used to more precisely classify the failure. Characterizers are often just code that does regular expression matching and substitutions. Characterizers can also contain sophisticated code that parses log files and does calculations to determine the root cause of failures. These more sophisticated characterizers are typically written for failures that land in generic buckets such as timeouts and hangs.

6 Triage

Triage is a medical term defined as the assignment of degrees of urgency to wounds or illnesses to decide the order of treatment of a large number of patients or casualties. Medical experts will quickly try to assess a patient’s conditions to determine what care they should receive and to what level of expertise is needed to treat such conditions. For example, a medical intern, a nurse, or a physician’s assistant are fully capable of treating certain medical conditions, but for more serious or elusive matters an experienced doctor or a specialized surgeon would be required. In times of disaster or war, the care of serious symptoms might be assigned to individuals who are not trained or certified. Likewise, in times of peace and when a patient’s symptoms are not urgent, inexperienced medical personnel will be given opportunities to treat patients under the supervision of a more experienced doctor.

In a similar manner, we are treating projects as we would a patient that is sick and has many symptoms. We quickly determine which of the many symptoms are serious and may be fatal. Those symptoms are assigned and treated. Our patient slowly gets better as we treat more and more symptoms. We find that there are some types of symptoms that need to be treated first before we can treat other more serious symptoms. Oftentimes, the symptoms we treat first are minor but more visible and widespread. These minor symptoms can make seeing the more serious symptoms difficult. After treating these minor symptoms, we finally can see the more serious issues that were hiding. If we take too long with these minor symptoms, it can push out the discovery of the more serious issues. These more serious issues could push out the timeline of when we expect our patient to be healthy. When time is critical, serious symptoms will be directly assigned to experts to treat as it is of the utmost importance of the timely health of our patient. Throughout the lifetime of a project, Validators with varying levels of experience will treat symptoms both minor and serious. We must take great care with what level of expertise is needed to treat each symptom.

Therefore, in the medical sense, good triaging is critical for saving as many lives as you can with the resources you have. Effective triaging in a design environment is critical for utilizing all the dedicated and shared resources you have to get your project healthy as soon as you can without neglecting the health of other projects or potential projects where resources are also needed.

6.1 Triaging Failures

Validators are constantly running tests to find bugs in the design and consequently in the test bench as well. Tests can fail for a variety of bugs found in RTL, testbenches, tests, cmd lines, and tools. Ideally, you want to be able to identify the cause of the failure as quickly as possible. Good triage tools and methods will achieve this goal. When tests fail, they should be put into a triage database of failures. Triagewb is the converged solution for inserting failing tests into a triage database. In DDG Triagewb has been integrated into the run tools to make this a seamless process. This gives the Validator all the benefits of the TRIAGE tool for each and every failure. Now a Validator can compare failures in their local area to those that are on released models. This also allows data mining of these failures to see if the same kind of failure has been seen before, when it was first seen, what types of tests have hit it, and how prevalent the failure is. When a failure is inserted into a triage database, you want it to go in with all the useful data attached. This will make it easy to organize, characterize, and prioritize. All this information in the triage database creates a great library of failure detail that will help projects with knowledge of health, history of failures, ways to standardize debug, data to create indicators, and much more.

Other type of failures such as those from model builds can also be inserted into the triage database to take advantage the benefits TRIAGE brings. For example, engineers can see if anyone else has hit the same build failure and if there is a known solution for how to fix it. In addition, a cryptic error message may have notes that make it more easy to understand and know how to fix.

By having a database of failures and the tools surrounding it to simplify accessing and modifying the data, teams can now more effectively communicate the state of debug, share failures, collect failure knowledge, and automate common debug tasks.

6.2 Creating Buckets of Failures

Test failures are constantly occurring and the first step of the triage process is to bucket the failure. Failures have many characteristics that help us group them with other like failures. The most useful characteristic used in bucketing a failure is the failure string. Most bucketing can be done using just the failure string alone. In order for this statement to be true, good error messages are essential. Designers and Validators should write error messages in their code where the first line clearly identifies the failure. This first line does not have any data details that could cause bucket explosion, but only details that is needed for good bucketing. The error message should have additional lines that contain all the details that would be useful to help debug that specific failure. This enables characterizers to use the first line of the error message as the bucket name and not require additional code to filter out the details of the error saving time and effort.

Here are some examples of bad error messages:

- Error Occurred – (Error message is too vague more detail is needed)

- Data: 0xf345acdb Mismatched Expected Data: 0xbd45dc31 – (Too much detail. This will cause bucket explosion as failures of the same type can have millions of different values of data. This error message is also vague and doesn't describe where the mismatch occurred)

In these two examples, a characterizer can provide much needed detail and filtering to make a bad error message turn into a good bucket name. When encountering bad error messages you would ideally like the error message to fixed, but in certain project stages where change is not possible or not desired, you can use characterizers to get to the desired bucket name.

Here is an example of a good error message:

```
Data mismatch occurred in XYZ transmit queue  
ReqID of transaction: 0x3F  
Actual Data: 0xABCD Expected Data: 0x1234
```

See that the first line has all the information needed to create a good bucket name with no data present that might need to be filtered. There is information in the 2nd and 3rd lines of the error message that has details that can help the debugger.

In addition to the error message, teams will also append or prepend information to make the bucket name even more useful. For example, you can prepend the bucket name with the type of failure and the area of failure. Here is an example of bucket name using the good error message above.

```
OVM::PCIE::Data mismatch occurred in XYZ transmit queue
```

Now you can see that this error was detected by OVM testbench code. The area where the failure is located is in PCIe. And of course the error message itself.

The best bucket names are those that are concise and very clear as to what is failing. The addition of area of failure and the type of error are very useful when assigning failures. For example, a SVA type of error can easily be assigned to designers who aren't as familiar with the testbench. When running tests at SoC level or Full Chip level the area of the failure will be very helpful for finding area experts. Much automation can be done when using characterizers to help discover details from log files that are in the failure area. These details can be used to determine if a failure is a known issue. As an example certain bugs can manifest themselves in an array of different failures. A characterizer can be used to examine logs to determine if those different failures are actually due to a particular bug and then put them all in the same bucket. Writing characterizers like this do take time, but can save time when dealing with messy bugs like in this example. This will save on redundant debug that will almost always occur with these types of bugs.

Finally, you will want to leave your bucket names pure and not dilute them with extra detail that will make them hard to correlate like failures. For example, you don't want to have 'dut' or stepping in the bucket name. This will make it hard to see failures of the same type in the database without doing some fancy querying. You want to easily share bucket information with anyone that can hit a failure, whether it be at Integration, Full Chip or at the IP level. You want to know the history of the bucket so that information attached to that bucket can be used for debug by anyone and not be lost to a stepping or a dut or something else that pollutes the bucket name.

6.3 Finding Failures

Thousands of test are being run each day to find bugs in the design. A percentage of these test runs will have failed and will need to be debugged. An important part of triaging failures is to make the failures organized and easy to find. Often a person will run a test list and debug the failures straight from the test list run area. This method works for very small teams with small numbers of test runs though it may often be the case that two engineers may debug the same failure at once. When tests are being run via automation with a tool like Granite it will quickly be overwhelming to manage failure debug. You will want to take advantage of a Triage database tool to help you find, organize, and to assign your failures. The common way is to create a database query that will show failures that are owned by your team. Often, these queries will only show failures that are less than 2 weeks old or are actively being debugged. Other times, the query will show the failures that need to be debugged for the current weeks regression. Standard queries should be created for all teams on a project to help them easily find the failure they should own in the database. Using a database for failures makes it easy to see which buckets have the most failures and where the debug resources are most needed.

Here is an example of a TRIAGE standard query used by the DDG PCIe team to get the failures in the database that need to be debugged:

```

WHERE [result].[status] = ( 'fail' )
    AND [result].[debug_status] in ('new', 'wip', 'debugged_keep')
    AND [result].[run_type] = 'production'
    AND (
        DATEDIFF (dd,result.date_run,getdate()) < 15
        OR
        [result].[debug_status] != 'new'
    )
    AND [result].[team] = ( 'iu.pcie' )
    AND [result].[stepping] = ( 'cnls62-p0' )

```

This query will show failures that in the new, wip, or debugged_keep debug state. It will only show failures that are labeled as production runs. It only looks at failure that are less than 2 weeks old unless the debugged state has been change from new and then it will keep them around. It is also looking at just failures that are tagged to be on the pcie team and for a specific stepping.

Another advantage of using a Triage database is that you can see if a failure that you are looking at was hit by others. You can see if it was debugged already and the variety of tests that are hitting this same failure. You can also see if this failure has been linked to any known bugs or if any debug notes have been left by previous debuggers. You can also see the history of the failure and see what model release this failure was first seen.

Another method might be to debug all the failures that were seen on a specific model release. A query can be created to show those failures and then debug can be organized and owners can be assigned to each of the buckets. The database can still be utilized to see the history of these failures as well.

6.4 Assigning Failures

In order to ensure project health, the failures found will need to be debugged. Often it will be overwhelming and impossible to have a single person do all of the debug. Failures should be assigned to spread the workload. Teams often have a designated person assigning and communicating the status of all the failures that are currently being seen. Other teams will take a more proactive approach where debuggers independently assign themselves buckets that they will debug. In team meetings they can review any unassigned buckets to ensure that all buckets get debugged.

When using TRIAGE you have the ability to auto assign failures. This can be done when there is clear ownership of certain types of failures. For example, a new checker is written by a team member. You want all failures being flagged by this new checker to go to that person as it is known that the checker is new and still has issues that they would be the most effective at debugging. Most often, you want the person most familiar with the area of the failure to be the debugger. Other times, failures might be given to the inexperienced to help them learn the area and develop debug skills.

Persons that assign failures to themselves usually are the most capable of debugging that particular failure. There is also an opportunity when assigning failures to yourself of unfamiliar areas that will help you to grow and increase your scope of expertise. This should especially be done when the experts are overwhelmed and need help.

Assigning failures will help your team be organized and effectively debug the mountain of failures before you. Actively assigning failures as a team keeps everyone aware of the variety of failures out there and what could potentially cause havoc in the turnin pipe if not debugged soon. It also will prevent silly silicon escapes that could have easily been found by a person debugging a pre-silicon failure that was sitting in Triage waiting to be debugged. Team meetings should set aside 5-10 minutes to go over the buckets and ensure buckets are assigned and being debugged.

6.5 Ownership of Failures

Characterizers should be used to help show clear ownership of failures. Details surrounding the failure can be used to show the area where the failure occurred. Buckets can also be tagged to show the area of ownership.

For example the PCIe validation team tagged buckets to show the area ownership for that bucket. Transaction Layer failures were given a PCIE_TL flag and Physical Layer were given a PCIE_PL tag as an example. There were other areas that were assigned as well like PCIE_IOSF, PCIE_AFE, PCIE_BFM to show just a few. This enabled the team to easily sort and filter the buckets and show which areas needed the most debug help. It also made it obvious when assigning failures of whom should be debugging that particular bucket.

Once assigned a failure the debugger should make sure the failure is debugged in a timely manner. If they find that they will not be able to debug the failure, they should unassign themselves and let someone else debug that failure or find a new owner for that bucket. When swamped with other

tasks it is important to communicate that you are not debugging the failures assigned to you especially when there are resources available that could be debugging those failures.

There are particular buckets where real bugs are prevalent and ownership of these buckets should be shared. Often these are buckets related to timeouts or system hangs. These oftentimes, require that each and every failure in the bucket be owned.

When owning failures, Validators should document their debug in TRIAGE so that progress can be seen by all who have an interest. Good triage practices requires bucket and failure owner to correctly disposition the failure as covered in the next section.

6.6 Dispositioning Failures

Once failures have been debugged, you will want to disposition the record. When dispositioning a failure you will change the debug status to a debugged state, document the failure, and link any bug or issues that have been filed. When documenting the failure, the Validator should document the failure cause and add bucket notes to explain anything that might help a future debugger or other current debuggers of this same bucket. This will help the new and original debuggers know why this failure occurred and what was done to resolve it. It will give them clues as to how they are seeing the same failure and will give them the information to know who debugged this before. Effective Validators will clearly document the debugged bucket to help future debuggers of the same bucket. The future debugger might even be the original debugger of the failure and yet still not remember the details of the failure or its disposition/solution. Bucket documentation saves countless hours of redundant debug.

Often you will debug issues that either cannot easily be fixed or you are at a stage in a project where only critical bugs can be fixed. These failures can be auto dispositioned to communicate that this is a known issue and no longer needs to be debugged. This can be done via characterizers, scripts, and built in methods in the TRIAGE tool. The name of the bucket can have text in it to clearly state this is a known issue.

6.7 Timeliness of Debug

The triage and debug of all failures can be a daunting task. Teams will create constraints on the failures to be debugged in order to make this task more manageable. For example, DDG uses a 2 week window for failures to be debugged. Standard queries in the TRIAGE tool are setup to only show failures that are less than 2 weeks old. This helps prevent the debug of old failures that have already been directly and indirectly fixed. When using the 2 week window method, there still remains the risk of missed debug but, it is likely that if a failure has been hit once by the current tests running, it will be hit again.

Another method is to run a large regression of tests and then triage and debug all of those failures before running another regression. Triage is essential for the organization and communication of the debug of those failures and easily showing what debug remains.

Teams may also use the priority field in TRIAGE and have SLA's (Service Level Agreement) of when those buckets should be debugged. This may be necessary when project timeliness is of utmost importance.

6.8 Automation

In order to more effectively debug and utilize resources, much automation has been developed by teams that have a long history of using the TRIAGE tool. Here are a few that have been most useful:

- Auto Disk Clean Up – Tool developed to safely delete failure directories. The tool looks at the debug status and age of the failure to determine when it is safe to delete the directory. It will not delete failures that Validators tag to keep around.
- Auto Rerun – Tool developed to rerun failures and generate a signal dump and/or more verbose debug messages. Tool will rerun one failure per bucket by default to help save on disk space and compute resources. The tool will update the record in TRIAGE to make it easy to find the failure that has the wave trace. Now debuggers have everything they need to debug most failures without wasting time to manually rerun tests.
- Auto Debug – Feature developed to automatically debug incoming failures that are known issues. Feature can use bug ID to determine if the bug has been fixed on the model the failure was running on and disposition the failure appropriately. Helps communicate debugged failures and eliminates redundant debug.
- Indicators – Debug indicator can easily be created from TRIAGE data to help projects determine the model health. An indicator feature is now being developed by DTS that will be a part of the Venus suite of tools.

Leveraging TRIAGE data to automate and effectively use compute and disk resources is yet another benefit of a validation team having good triage practices.

7 Summary

Triage is yet another tool, methodology, and process to be utilized by all Validators and all design teams to effectively use resources for the debug and dispositioning of failures. Effective validation teams will make the time to develop and utilize good triage practices.

8 Future Work

Add a section to show some common triage problems and solutions.

Add a section to show the benefits of Triage when on a small team vs a large team.

9 References

DTSpedia Triage Page: <https://dtspedia.intel.com/Triage>

The Art of Pre-Si Val: Chapter 16

Debug

By: [Erik W. Berg and Maria F. Pineda](#)

1 Abstract

“If there is a failure, there is a bug. The only question is where.”

Debug is the single largest consumer of Validator time over the course of a project. Newer Validators learn to debug in a mechanical method, while experts have internalized these methods to gain a certain touch that allows them to do very fast, efficient debug.

This chapter will describe the strategies, attitude and communication skills that will improve a Validator’s impact on a project in the context of specific steps for the debug process.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/3/2009	Initial Draft and First feedback	Erik W. Berg	Michael Bair
1.1	2/12/2016	Overhaul to bring up to date	Maria Pineda	Michael Bair

3 Contents

1 Abstract.....	381
2 Chapter Revision History	381
3 Contents.....	382
4 Purpose.....	384
4.1 Why do we need this chapter?.....	384
4.2 What does this chapter cover?	384
4.3 What does this chapter not cover?	384
5 Background Concepts.....	384
6 Debug – Deconstructing the “Black Art”	384
6.1 Selecting a failure	384
6.1.1 Prioritizing debug	385
6.1.2 Considerations in choosing a failure	385
6.2 Debugging steps.....	386
6.2.1 What are you debugging?.....	386
6.2.2 Avoid duplicate debug efforts.....	387
6.2.3 Reducing the problem.....	387
6.2.4 Choosing observation points to test assumptions	389
6.2.5 Search techniques	390
6.2.6 Critical debug.....	392
6.2.7 The role of experience in Validation.....	393
6.2.8 Recognizing a ‘wrong path’	394
6.3 Debug resources	394
6.3.1 Existing collateral.....	394
6.3.2 Developing your own collateral	396
6.3.3 Taking notes	396
6.4 Attitude and perspective for successful debug.....	397
6.5 Tell a complete story.....	398
6.6 Contacting someone for help	398
6.6.1 Experts.....	398
6.6.2 Novices	398
6.6.3 Somewhere in the middle	399

6.6.4	Ambiguous ownership.....	399
6.7	Asking questions.....	399
6.7.1	Ask open-ended questions	399
6.7.2	Ask questions with a purpose	400
6.7.3	Try to build upon each question.....	400
6.7.4	“Just send me the failure”	400
6.7.5	Throw it over the wall.....	401
6.8	Helping others with debug	401
6.9	Answering questions.....	402
6.10	Reflecting on the failure & debug process	403
6.11	Debug training methodologies	403
6.11.1	Injecting a bug into a clone	403
6.11.2	Interactive debug led by an expert.....	403
6.11.3	Interactive debug with an expert monitor	404
6.11.4	Group debugging sessions	404
6.11.5	Presentations of a root caused failure.....	404
7	Summary.....	404
8	Future Work.....	405
9	References.....	405

Table of Figures

Figure 1 – Debug flow chart	390
Figure 2 – Decision Tree	393

4 Purpose

4.1 Why do we need this chapter?

Through the life of a project, Validators encounter an unending stream of test failures that require debug. Root causing and dispositioning these failures at an ever-increasing rate requires Validators to continuously expand the scope of their knowledge and strengthen cooperation across Validation, Design and Architecture. This cooperation is essential to deliver a high quality project in an increasingly shorter time line.

4.2 What does this chapter cover?

This chapter describes the strategies, tools and mindset that a successful Validator employs over the course of a project.

4.3 What does this chapter not cover?

This chapter does not describe how to debug specific failure signatures or modes.

5 Background Concepts

6 Debug – Deconstructing the “Black Art”

At its heart, the job of any Validator is to test and debug a design. The second part of this job, debug, is often referred to as a “black art”; however, there are behaviors that distinguish great Validators. These behaviors are independent of the individual’s knowledge of the design. Reducing the debug problem in a semi-infinite test space to a set of tractable experiments that will root cause the broken expectation is no black art at all.

This chapter will describe the process of identifying, understanding and root causing a failure. Additionally, the chapter describes the process of guiding others through that process, and sharing the results with the team.

6.1 Selecting a failure

The first step in debugging is finding and claiming a failure. Utilize tools like [Triage](#) to your advantage, and to the efficiency of your team. There is nothing more frustrating than debugging a failure twice; except perhaps, root-causing a failure only to find out your team member did it yesterday.

6.1.1 Prioritizing debug

Our first job as Validators is to drive model health. Where we are in the lifetime of a project will have an impact on how you choose the failures to debug. All failures need to be debugged, the only questions are “by whom?” and “in what order?”

Earlier in a project, you may have the luxury or even be encouraged to debug into unfamiliar areas of the model, so that you can learn the microarchitecture (uarch) or the TE (test environment). This is an excellent opportunity to gain expertise that you can leverage later in the project when critical issues arise, and you are asked to drop everything else to help.

While debugging into unfamiliar logic is important for learning and growth, it is typically slower and may not be as efficient for the team in some situations. There are two occasions when the project may not be able to afford the added latency of debugging an unfamiliar area:

- **Late in the project.**
To achieve maximum throughput, you may have to refer failures to experts in a given area and others will refer failures to you.
- **Debug is blocking others' progress.**
When a failure is blocking forward progress for other Validators, Designers or Architects, this is not the time for you to be learning new uarch when other experts are available.

In these situations, it makes sense to request expert help sooner than normal. Outside of these situations, it is beneficial in the long term to expand your debug expertise and familiarize yourself with other areas’ RTL and environment. Carefully consider the short-term necessity of ‘getting it done’ vs. the long term benefit of expanding your debug proficiency.

6.1.2 Considerations in choosing a failure

Select a failure for debug based on any or all of the following:

- 1) **Unclaimed bucket**
Select a failure from a bucket where none of the failures is marked for active debug by someone else.
- 2) **Largest fallout**
The highest number of failures is potentially having the widest impact on other Validators.
- 3) **Familiar signature (1)**
When possible, select a familiar signature. If you are experienced with a particular type of failure, you will be able to debug it more quickly.
- 4) **Familiar signature (2).**
Later in the project when a critical mass of “Won’t Be Fixed” (WBF) bugs starts to pile up, extra vigilance is required to make sure familiar signatures are not hiding any new bugs.
Anecdote: On the first Core i7 project, at least one escape was hit repeatedly in cluster pre-silicon testing, but it had been ignored because it shared a failure signature with a Won’t-Be-Fixed bug.

5) Unfamiliar signature (1).

If you are the sole owner of your unit, unfamiliar signatures are often an indicator of a new source of bugs in the model.

6) Unfamiliar signature (2).

If schedule allows or if someone needs help, dive in and learn something new.

Inform your team members that you are debugging a given failure. If using TRIAGE, mark the failure ‘WIP.’

6.2 Debugging steps

There is one definite outcome of a complete debug process: you will find a bug. The only question is “Where does it live?” Some possible locations:

- RTL
- Checker
- Assertion
- HAS, C-Spec
- Tool
- Environment
- Coverage Monitors
- Test and Test Generator

Each of these will require different debug knowledge, but the problem solving process is the same. In all cases, a successful Validator will frame a set of questions to reduce the problem space, and allow them to zoom in on the problem. In its essence, debugging is simply the application of the Scientific Method that most of us learned in middle school. The application can be more time consuming and aggravating given the complexity of environment in which we work. However, with time and experience, the debug effort becomes significantly easier.

6.2.1 What are you debugging?

This seems like a simple and obvious statement, but it is critical to the debug process. Getting this step wrong will lead to misguided and wasted efforts.

- **Determine the failure signature**

Find the error message that occurs first in the log files. It is important to note that sometimes multiple errors are flagged, and you cannot always depend on the run tools to figure out which one came first. If you have not determined what the correct error message is, then you are not debugging the failure.

- **Understand the level of abstraction at which the check is failing**

A DUT error message, a checker, an architectural checker, an RTL assertion message and a coverage forbidden condition have different observation points that will guide the debug process.

- **Understand what is happening in the test**

Read the relevant log files to get a sense of the purpose of the test and the behaviors in the RTL that the test is driving.

Finding the correct failure and defining the violated expectation during specific activity in the machine will help avoid wrong paths in the debug process.

6.2.2 Avoid duplicate debug efforts

Avoiding duplicate debug efforts is really a parallel process to identifying “What are you debugging?” in the previous section, but it is a separate process.

- **Avoid multiple people unknowingly debugging the same failure at the same time**
You and your team will need to develop a system to prevent multiple people debugging the same failure at the same time. Triage may be sufficient for this but it is worth discussing within your team. Prevent frustration and hurt feelings by over-communicating.

Anecdote: In HSW, a failure in one of the IPs was blocking an important turn-in in the main SOC repo. It was assigned to the area’s expert, and while they drove home, another team member debugged it, but did not inform the team. The expert spent a few hours debugging the failure overnight, only to find out the next day that it had already been resolved. This caused some hurt feelings, and uncomfortable meetings. Prevent these situations; over-communicate! If you feel you can contribute to a failure, speak up; do not allow your team members to waste their time.

- **Determine if there are open bugs that contain this signature**
Before continuing with debug, determine if there are any open bugs filed that contain this signature. If there are, then look through the current failure to see if it matches the one described in the bug report. If after a reasonable investigation you cannot make that determination, contact the original filer of the bug for more details.
- **Leverage your team’s knowledge**
This is not an invitation to camp in other people’s cubes. Discuss what you are debugging during casual conversations. Perhaps reserve 10-15 minutes in group meetings for people to share what they are debugging. Chances are that someone else has seen something similar that might not be well described in an existing bug report or that was not previously fully root caused. These conversations often jump start a stalled debug effort.
- **Document your findings**
Setting a few minutes after root-causing a failure to document your debug process can be greatly beneficial. Documenting your findings will help you understand and improve your debug process; it will also aid you and your team in future failures. One of the best debuggers in IPG, has a number of OneNote notebooks documenting every failure he has debugged. Computer memory is cheap, your time is not; start writing.

6.2.3 Reducing the problem

Before beginning to debug, take a step back and ask yourself:

- What is the failure?

- What do I know about the expected behavior?
- What am I assuming about the TE/RTL/Checker... etc?

The answers to these questions will lay the foundation for your hypothesis. Postulate a hypothesis that allows you to significantly reduce the validation space.

Without a hypothesis to test, you will needlessly waste time playing a guessing game about the failure without making progress.

Some questions to ask that might guide you to reasonable assumptions:

- The failure is due to a checker. Is the checker correct to flag an error? On the other hand, is the RTL behaving correctly?
 - Choose one to formulate your hypothesis.
- When was the last time the test passed? What has changed since then? The test, the TE, the coverage, the RTL, the checkers?
 - Knowing the most recent deltas in the model are important facts to utilize when formulating a hypothesis.
- If the failure is related to an interface, do the owners of opposite ends of the interface have the same assumptions?
 - Incomplete or contradictory expectations from an interface are common sources of bugs; it is worth considering in your calculations.
- What is unique about this test? Is it driving RTL behavior that other tests are not?

Examples of assumptions that might be reasonable starting points:

- The failure is on a new test; assume it is a test problem.
Part I. When we inherit RTL from previous projects, it is possible that initial failures are due to new tests since the RTL has been tested to some degree.
- The failure is on an established test, but the RTL has been recently changed; assume it is an RTL problem.
Part II. When we inherit tests from previous projects, it is possible that initial failures are due to new RTL changes since the tests were probably stable from the last project.
- During execution, assume a failure is due to an RTL change.
When previously stable tests begin failing, it is possible that the newest RTL change introduced to the model has a bug in it.
- Coverage forbidden-condition fired; assume the forbidden is wrong.
Since the forbidden was likely created last in the TE/RTL/Coverage process, it is possible that the forbidden-condition was not coded correctly, or is actually a legal condition.

- An Architectural checker failed. Assume RTL is correct; consider the checker(s). There may be known issues with a particular transaction that the checker does not handle well.
- A transaction in Tracker-X is hung; assume the completion (CMP) never arrived. There may be a long logical path between a request (REQ) and CMP pair so start debug by focusing on one end of the flow.

Judiciously making and discarding assumptions reduces the debug space. While none of the above assumptions will prove to be foolproof, they will create a starting point for you to collect data, evaluate the hypothesis and either continue on that path or redirect your efforts.

If possible, determine the architectural reason for the checker or forbidden or assertion. Is there a message attached to the failure that gives any insight? Are the comments in the code near the checker that might give you some hints?

6.2.4 Choosing observation points to test assumptions

After making an assumption, a good Validator will test it by choosing data from appropriate observation points. Let us follow this process for one of the previous hypotheses.

- A transaction for a REQ in Tracker-X is hung, assume the CMP never arrived.

We formed this assumption after looking at the possible reasons for a hang. One of the following MUST be true:

1. The REQ, or some portion of its flow, is waiting on a different transaction to complete before it can finish its flow.
2. The REQ, or some portion of its flow, is waiting for a response before it can continue.

If (1) is TRUE, then you are not debugging the failure. The actual failure is dependent on the completion of the “different transaction”. The chosen assumption assumes (2) to be TRUE and initiates the following decision tree:

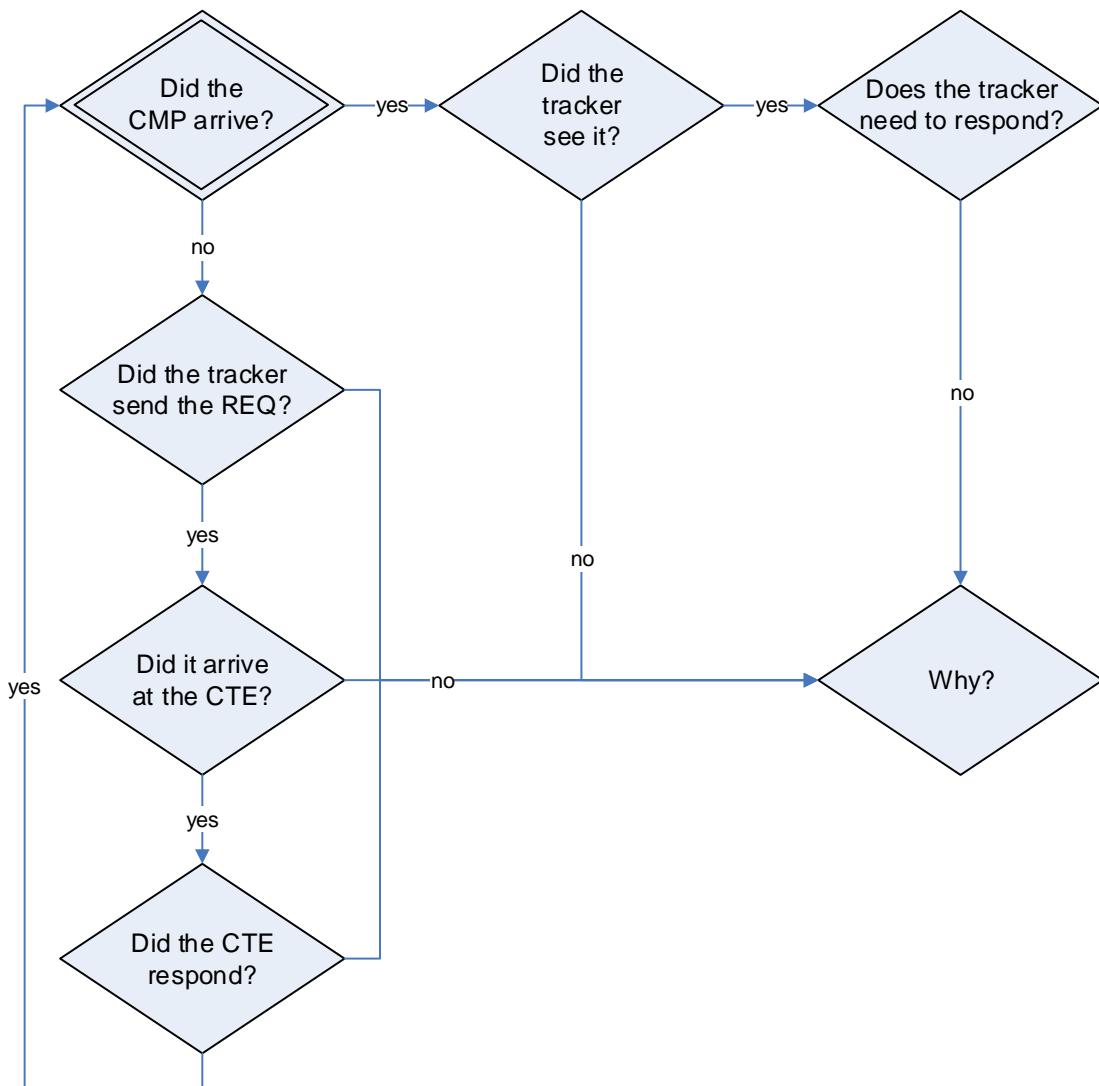


Figure 40 – Debug flow chart

A simple YES or NO can answer each of the questions in the flow chart, and the answers guide the next inquiry and reduce the scope of the problem through a simple binomial regression. If your assumptions lead to questions that are answerable with a MAYBE, then you need to reformulate the assumptions.

6.2.5 Search techniques

A number of search techniques can guide you towards the bug.

6.2.5.1 Playing 20 questions

The 20 questions game is useful when trying to form assumptions or to identify the failure.

1. “Is the checker right or is the RTL right?” – choose checker

2. “The checker relies on two assumptions. Is assumption A right or is assumption B right?” – choose B
3. “Assumption B depends on inputs X and Y. Does input X make sense?” – yes, change direction
4. “Does input Y make sense?” – no, go deeper

and this continues until you either reach the bug or need to switch to another search technique...

6.2.5.2 Traceback

If you are familiar with potential cause of the failure, test or RTL, it may be easy to debug by tracing the bad data/signal/wire back through every gate and evaluate each input and any combinatorial logic to make sure that they make sense. Tracing is not always the most efficient way to start debug. It is often necessary to utilize log files to gain a better picture of the failure cause.

When tracing, we recommend having an “expected/passing behavior” reference waveform capture. We often keep reference captures from older models and even prior projects; ask around, you may be surprised what is available.

One last note, utilize the labels, grouping and coloring features of the waveform viewer tools. It is easy to be overwhelmed by the number signals in a window, and any interruption may cause you to lose track of your debug path. A well-documented waveform file can easily be shared with others when asking for input on the failure.

6.2.5.3 Binary Search

Similar to 20 questions but more applicable when a flow or logic path seems to be broken. This can result in $O(\log n)$ ³⁸ search time. Assume there is a hung tracker entry as in Figure 1 and the expected logical path is:

1. REQ issued
2. REQ arrives at CTE interface
3. CTE issues CMP
4. CMP arrives at tracker
5. Tracker deallocates.

A binary search would go as follows.

1. Did the CMP arrive? – NO.
2. Was the REQ issued? – YES.

The search path is now between the issue of REQ and reception of CMP. Divide the path in two.

³⁸ $O(\log N)$, the O notation of a $\log(n)$ function. In computer science, big O notation is used to classify algorithms by how they respond to changes in input size, such as how the processing time of an algorithm changes as the problem size becomes extremely large. (https://en.wikipedia.org/wiki/Big_O_notation)

3. Did the REQ arrive at the CTE interface? – YES.
First half of the path seems fine.
4. Did the CTE issue the CMP – YES.
Second half of the path seems broken.

You now have two options.

- A. Employ a traceback methodology to follow the path from the CTE interface to the tracker to see where the CMP is stuck or dropped.
- B. Do a binary search between the CTE interface and the tracker to see if there are clear ways to further divide the path.

6.2.5.4 Up-Leveling

Debugging with logs and trackers helps to keep a 10,000 ft view of the problem before digging into the minutiae of RTL signals. It is easier to ask whether your assumptions make sense and to have a sense of what is going on the machine when you can see more than just 1's and 0's. You may be able to navigate to something interesting much earlier than the detected error.

6.2.5.5 Rubber Duck

Logs, specs, waveforms... it is a lot of information to process. Add an email, a couple of IMs and a few more interruptions, and it is possible that you will not be able to see the bug or the next logical step even when it is staring right at you. This is where a rubber duck method can be a lifesaver. Take a deep breath, gather your thoughts and explain the problem to the rubber duck. Do not laugh. As you state the problem, your assumptions and findings, do you see any gaps in your hypothesis? Do you really understand the big picture?

Explaining the problem to a second person usually clears distractions, and allows ideas to reach the surface. This is why collaboration is so important. Nevertheless, your teammates may not appreciate non-stop interruptions, so adopt a rubber duck, and try explaining it the problem first. If you are still stuck, approach a friend.

For more information, visit: https://en.wikipedia.org/wiki/Rubber_duck_debugging

Another version of “rubber duck” that also helps with documentation, is writing an email. Start your email with all the information an expert will need to help you debug the failure. State the area/unit of failure, priority, a concise summary of your findings and assumptions; do not forget to include paths to relevant logs. Many of our top debuggers have found that by the time they have finished the email, they have a better understanding of the problem and have determined next steps for debug. If that is not the case, your next step is simply to press send.

6.2.6 Critical debug

There are stages in a project, where debugging a failure is more important than improving your debugging skills. In these instances, do not hesitate to approach experts, emails, IMs and cube visits. Critical debug is a pass for others to prioritize work on your issue; make sure to not abuse it.

On critical debug, your role is not necessarily to debug the failure, but to own the failure. When you own a failure, you are responsible for finding experts, carrying your failure from cube to cube, calling meetings and making reports. This is an important role, if at times cumbersome. Do not limit your involvement by sending emails; time is of the essence! Enrolling multiple experts is effective speeding up debug and in avoiding, “he said, she said” ambiguity between experts, but it is very disruptive to several people so only evoke it in critical situations.

6.2.7 The role of experience in Validation

Experience does not affect the basic problem-solving algorithm. It may help in making assumptions or knowing what problems a class of failure signature may indicate, or it may even help in knowing where in the debug process to back up to if a particular assumption proves wrong. However, the basic methodology of new and old validators is identical. There is no “black art”.

Consider the decision tree below. A failure signature indicates where a debug process will start. An experienced Validator may have navigated this debug path before, and realize that if they check one assumption they can bypass a number of dead ends, and leap to the source of the problem without checking other assumptions. If their assumption was wrong, the experienced Validator knows to stop, recheck and back out to a new assumption.

Debug experts have perfected the decision-making process, and still follow the same basic flow.

Experienced and inexperienced validators who have no previous knowledge of a failure signature should both follow the same decision tree beginning at Start. However, the experienced Validator is more likely to ask the right questions and understand the answers.

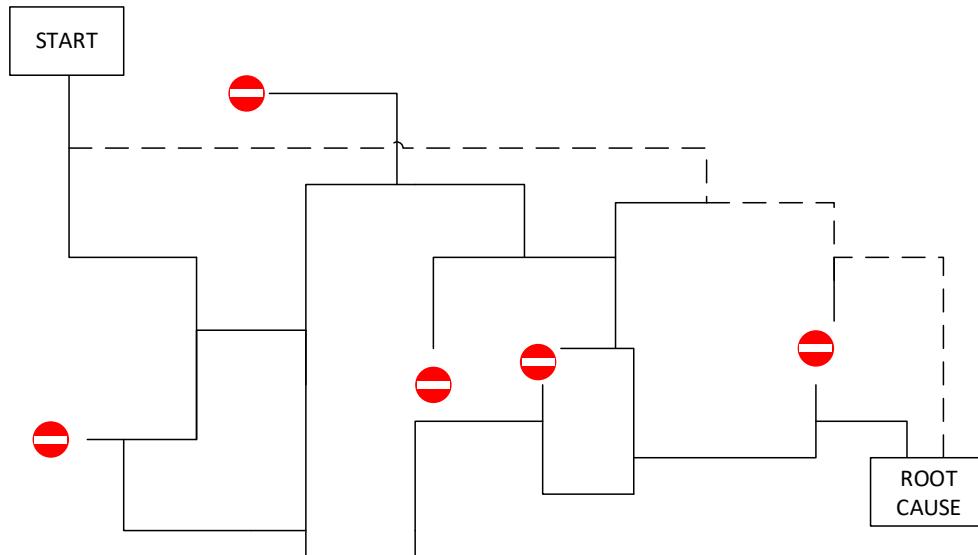


Figure 2 – Decision Tree

6.2.8 Recognizing a ‘wrong path’

Everyone is going to pursue a wrong path in the debug process. Everyone is fooled by red herrings at some point. The trick is minimizing the wasted effort by recognizing wrong paths as early as possible, by comparing anything new you see with the current set of assumptions you are testing. Some things to consider as you root cause a failure:

- Before beginning, list your assumptions or expectations; do they make sense? Discuss them with a rubber-duck/another Validator.
- If you are tracing a signal and leave the fub/unit where an assertion fired, ask yourself if this makes sense given the assumptions you are currently working with.
- If you are tracing a signal and verdi traverses a section of code that does not resolve well, ask yourself if this makes sense.
- Add labels to the collection of signals in verdi. Create a story that you can follow the next day, or that a team member can follow remotely.
- If you see a signal driven in many different places, take note of them all before continuing down the path of one of them. This may reduce the effort needed to back up later.
- Pause every 15-20 minutes and look the story you are constructing with your notes. Does this seem to be working towards the failure or have you gone off the tracks?
- If you are stuck, review your notes on the debug with the rubber duck. If you are still stuck, approach another validator. Sometimes, it is that second set of eyes that can quickly spot the issue.

6.3 Debug resources

Debug collateral is available to assist in your debug efforts. A variety of tools exists in the environment, and each test run generates test-specific information. You may also generate your own collateral (debug scripts and such), and keep it in your local area as you debug in the same area of logic many times. If this collateral proves helpful, share it with the team! You can post it on a wiki, shop talk ([goto/shoptalk](#)) or work with other members to deploy it to the team.

6.3.1 Existing collateral

The DDG Validation environment has several tools available to assist in digging into a failure. Each test run also generates a variety of log files that monitor interfaces or events in the RTL during tests.

6.3.1.1 Waveform and Code interactive viewers

There are a number of waveform/trace viewers available in the industry; one of the most common tools used by DDG is Verdi. These type of tools are very useful for tracing signals through the RTL.

Strengths of these tools include:

- Users can add signals to the wavetrace as needed during the debug session.
- Users can save signal sets into files for use in future debug of the same logic.
 - Verdi uses the *.rc format.
- Signal values can be annotated in the source code making it very easy to find previous transitions.
- Users can trace through RTL quickly by simply double clicking on signal names in the RTL viewer.
- See the Verdi and Debussy quick reference guides from Synopsys for more information.

Limitations include:

- Viewing many signals for many entries in an array can be a bit of an eye chart.
- It is difficult to get a high-level view of what is going on in the machine. It is highly recommended that you have a clear understanding of what you are looking for before starting to trace.

6.3.1.2 IDE – Integrated Development Environment

A large part of our work is the development of test bench collateral, and debug of that code. An IDE can greatly improve your development and simple debug efficiency for syntax errors, structure definitions etc.

A number of teams at Intel are using **DVT Eclipse**. Design and Verification Tools (DVT) is an integrated development environment (IDE) for Specman-e, SystemVerilog, Verilog, and VHDL. Benefits of using DVT-Eclipse include:

- Its “project awareness” so everything in the project is accessible.
- On the fly syntax error reporting.
- Hyperlink navigation, no need to grep to find structure definitions etc.

For more information on DVT-Eclipse: [goto/dvt](#)

When making significant edits in the TB, consider using an IDE. It will save you time, and make you a better coder. Why work like this is still 1984?

6.3.1.3 Log files

The failure directory will have a multitude of log files; in general, the bigger the model, the larger the number of log files. There are log files for specific interfaces, checkers and other loggers in the model. Users can control messages printed to log files and can enhance the messages for future test runs. There are post-processing scripts to annotate logs, especially common for FW logs; a simple combination of **grep** commands can pull meaningful information out of the thousands of lines of log files.

Log files are very helpful for debugging flows through the machine that take a long time to complete, and can offer a very high-level view of RTL traffic. Log files are typically not useful for understanding gate level transactions.

It is crucial that debuggers at all levels take advantage of log files. Small units can often run with trace capture without incurring much of a performance penalty, but at higher clusters like SOC and emulation, simply running with trace capture is not a luxury we can afford. A debugger must reduce the debug in both space (IP) and time (cycle of interest) before pursuing a trace capture. If working in an individual IP, make sure the log files for your IP are relevant and easy to understand. It is also useful to have documentation to go along with the IP-related log files, so that you can easily enable others to effectively debug/understand your area. If working at a higher cluster, ask IP owners for insight into their log files; make sure to keep good notes on their response. You are very likely to run across other issues in the same area; next time, you can go a step further in debug.

6.3.2 Developing your own collateral

Validators will debug through the same logic blocks many times during a project. To facilitate this debug, Validators collect useful .rc files, scripts or sets of notes that help to describe the RTL or TE function. This knowledge is valuable not only to the Validator that created it, but to new and old team members. We often participate in debug of failures many months after finishing a project, or inherit a proliferation of the project a few years later. Documentation is rarely a waste of time; sharing that documentation with others helps the team grow, and frees the Validator to move on to other more interesting activities.

6.3.3 Taking notes

Taking notes is crucial during the debug process, especially when entering new logic areas. A good set of notes will help you to:

1. Backtrack if you have gone down a wrong path.
2. Construct a complete story to present to the potential bug owner.
3. Write a detailed bug report.
4. Ask open-ended questions if you approach another validator for help.
5. Walk away from the debug session, but quickly resume it once you have more time, or Monday morning.

A good set of notes will describe the behavior in the model that led to the failure. Keep notes in chronological order whenever possible so that it is easier to see how a series of transitions in the RTL work together. At a minimum, your notes will include:

1. Cycle times of important transactions or transitions.
2. Specific signal names encountered in the debug path.
3. A blend of messages from log files and signal transitions from Verdi.
4. A description of any assumptions made.
5. Any other observations made that are relevant to understanding your thought process during the debug session.

If you are leaving your debug session to tend to another matter, make sure to record what you think the next steps are to help you remember where you left off once you return.

The best place to take notes is in the body of an email message so that you can easily share them with others; OneNote is another good option.

6.4 Attitude and perspective for successful debug

With any failure, remember that you are lucky to have found it as it may be an exclusive opportunity to learn more about how the machine works. Without the failure, you may never have had the chance to study this part of the architecture and expand your knowledge base. This will allow you in the future to write better tests, checkers, coverage, etc. It gives you standing in discussions with Design/Architecture about decisions they are making.

As your understanding of the model grows, you are more likely to participate in discussions with Design and Architecture for the implementation of new features and bug fixes. Your input will help prevent the introduction of bugs into the model.

Your greatest assets in this process are:

- the knowledge you have built up previously
- your discipline in employing the scientific method to consistently reduce the problem
- your curiosity.

Be comfortable with ambiguity. You do not need to understand everything you see while you debug (complete understanding comes later, if ever) so do not get preoccupied with every little detail. Rather pursue a path of inquiry that seems to make sense.

Remember to come up for air after every 2-3 levels of depth in logic or messages. Look at your notes, and see if your path in inquiry still makes sense. Is a coherent story developing? You will spend plenty of time going down wrong paths, but re-evaluating notes as you progress will prevent this to some degree.

If you have contacted someone for help via email/IM and received no reply, do not hesitate to go find them in their cube or request a face-to-face meeting. Be persistent. It is okay to start with visiting their cube if you have done your due diligence in debug, and have appropriate questions ready.

On occasion, you will be completely lost on a failure, and will be inclined to throw it over to the wall to Design, Architecture or other Validators. The strength of the relationships you build by establishing a record of asking questions or contacting experts the right way, will give you greater latitude in these situations.

Similarly, when someone approaches you for debug help, respond positively. The way you respond will affect your relationship with your co-worker. You *want* them to come back to you again in the future for help.

- 1) If you have time, try to walk your co-worker through the debug process. Encourage them to take notes and ask questions. Your goal is to resolve the current failure and to pass on your knowledge.
- 2) If your time is constrained or if the debug is high priority, ask for the path to the failure, and debug it on your own. After you have finished the process and when time permits, it is always helpful to talk your co-worker through the issue.

6.5 Tell a complete story

The debug process is complete when you describe a coherent story about the failure to the future owner of the fix and they agree. Until you answer all of these questions and the fix owner agrees, the debug process is not complete.

- What failed?
- What are the root causes of why it failed?
- What bad behavior does this cause in the machine?

Occasionally a Designer or Architect will push back and try to attribute the failure to a bad test. In these instances, a Socratic style of inquiry can be very helpful.

“If X and Y happen, what should this piece of hardware do?”

“If X and Y happened to cause A instead of B, would that be a problem?”

On very rare occasions, even with a coherent story, a fix owner may disagree about the cause of the bug. In these cases, it may be necessary to wait until the bug goes to a forum (ex: DCCB, VT meeting) where the collective wisdom of the Arch/Design/Validation teams can examine the case and decide on next steps.

6.6 Contacting someone for help

Choosing to ask for help versus continuing to debug on your own is a balancing act: weighing the need to learn your area and become more skilled in debug, versus the need to get the failure understood and fixed. As you grow you will need less help from others, but there will never come a time when you stop asking for debug help completely.

Consider this a sound principle: when contacting someone for assistance in debug, you need to put in some effort first to debug and understand the failure.

Be cognizant of other Validator’s time, and understand that in most cases you will continue to own the failure, so take notes, take ARs, and always say thank you.

6.6.1 Experts

If you should be an expert in an area, but don’t understand this failure, roll up your sleeves, put up the “Do not disturb” sign and take copious notes as you step through the code line by line to understand it. You do not want to be trying to understand your area of expertise on a high priority failure.

6.6.2 Novices

If you have zero experience in an area that you are debugging, you are still not off the hook. What information can you figure out from the failure? Open the checker or assertion code and understand why it is firing. Maybe you will not understand the chain of events that led to this

failure, but you will be able to have a more substantial conversation with other Validators about it. Consult the HAS or SPEC for the unit. If you can't do anything more, at least you will have shown some effort to whomever you ask for help which will increase the likelihood that they will respond quickly. It will also help you to understand any response.

6.6.3 Somewhere in the middle

You should know this area or want to know the area but do not quite yet. Spend 15-30 minutes on it, and take copious notes. When you are stuck, look back through your notes, and try to put a compelling story together that you can present to an expert. You are trying to do two things here:

1. **Improve your understanding of the code.** This will allow you to get one degree deeper next time. This is good for the team and for your career.
2. **Be prudent with expert's time.** There are only so many experts to go around, and their time can disappear quickly. As you ramp, you might find yourself taking some of the load off the expert!

6.6.4 Ambiguous ownership

It is worth mentioning that some failures do not have clear ownership at first. You may start debugging a failure, but quickly realize another team really should own it. In such cases, it is ok to hand it over without having a complete story regarding the root cause, but you should still have a complete story regarding why you think it is someone else's responsibility.

6.7 Asking questions

The way you ask questions will go a long way towards establishing strong productive relationships with Design, Architecture and the rest of Validation. If your questions are poorly constructed, you will find the answers are as well. If you develop a history of bad questions, you will find that answers of all types will eventually stop coming.

6.7.1 Ask open-ended questions

As a team, we all need to be expanding the scope of our knowledge. Open-ended questions facilitate this by implying a desire to learn more about the failure. Closed-ended questions leave everyone in their little corner of expertise, and limit the complexity that the team can validate.

An example of a closed-ended question: “I saw this failure. Is this a bug?” The person you ask may answer ‘No’ in which case you are no closer to root causing the issue.

An example of an open-ended question: “I’m working on failure signature A. So far, I see B and C causing it. The part that I do not understand is X, Y, and Z. Do you have some time to help me understand what might be going on here?” The listener may or may not know the answer, but at least they know where they might guide you next, and they see that you have put forth effort to understand the failure.

6.7.2 Ask questions with a purpose

Until you know what information you need, you do not know what question to ask.

Example 1. A question without a purpose: “This test is failing, and it might be related to X. What do you think?” The listener has no idea what piece of information you are looking for to work through this debug.

Example 1: A question with a purpose: “I’m seeing failure signature A, and I’ve looked through the code, and it seems like it is because of X, Y and Z. I cannot figure out what the connection is; are you the right person to talk to about this? If so, do you have some time available now so that we can discuss it? If you aren’t the right person, do you know who is?” This kind of question informs the listener what information they might supply that would specifically assist in your debug effort.

Example 2: A question without a purpose: “Should field Y in your sequence X be non-generated?” The listener has no idea what is motivating the question, and no context to understand what problems might be occurring.

Example 2: A question with a purpose: “In my test, I’m trying to constrain field Y in sequence X, but the value is not taking effect. I am running it interactively, and I can see that the field is already populated. I noticed that field Y is auto-generated in the sequence so I’m wondering if I should change it to non-gen. I have not run the experiment yet, but hoped you could comment before I spent the time.” The listener understands what is motivating the question, and might have other ideas to try before the full experiment runs.

6.7.3 Try to build upon each question

Everyone does it from time to time, we ask the same question multiple times. This is something we can try to avoid. Take notes when discussing a failure with an expert. At the end of the discussion, repeat back what the next steps are. If you cannot repeat or understand the issue and next steps then, you certainly will not be able to later. This can also help the person providing the explanation to improve their teaching methods and communication style.

6.7.4 “Just send me the failure”

Occasionally, you will contact someone for help, and they will simply ask for a path to the failure. At first, this sounds great because you have too much debug and too little time. However, this reply is almost as bad as no reply because you will learn nothing from this failure. When someone requests that you “just send the failure”, you should always reply with the path and add a request to either be present while the debug occurs or to have them walk you through the full debug after they have finished. You want to be able to ask questions, learn the code and understand the failure.

Cooperative debug sessions are very helpful for new Validators to ramp on the design. Scheduling 30-60 min to debug a failure with an expert is a popular way to learn debug methodologies in

unfamiliar parts of the design. You would be surprised how many tricks you can learn and teach others.

Another twist on collaborative debug is to share a VNC session, but switch drivers every 5 to 10 min. This way everyone is an active participant, and both debug style and content knowledge are shared.

6.7.5 Throw it over the wall

A caveat to “Just send me the failure,” you may have to judge if the failure is an area you want to develop expertise in, and if you think you will be seeing more of these types of failures in the future. Failures in other Validation domains are often good candidates to throw over the wall after a degree of debug.

6.8 Helping others with debug

Someone who does not know the micro-architecture will ask you for debug help in your area. Even when you are the one providing help instead of requesting it, there are strategies to make the interaction efficient and not redundant.

Identify the correct failure signature

Occasionally multiple error signatures for a given failure occur in different places in the log files. Double check the assumptions of someone requesting help. Co-workers that are not familiar with the uArch, checkers, coverage forbiddens or RTL assertions in your area are occasionally debugging the wrong failure signature.

Understand the debug progress

Ask the person requesting help to explain what they have learned so far from their debug. This is a useful filter against people throwing failures over the wall to you. If someone has not made a valid effort to understand the failure, it may be appropriate to ask them to start the process and ask you more questions later.

As always, there may be times when it is appropriate for them to throw it over the wall to you. You need to make a judgment call as to what is right for you, them, and the project. The right call may be for you to take ownership of the failure. Imagine a failure at tapeout time that is trending to be a horrible bug in your unit - this is an excellent time to grab ownership.

Understand the test setup

Often times, bizarre behavior observed in the RTL by other teams or groups is due to running in modes that they should not be using. Are there any special modes being used? Any defeatures? Is there any background knowledge of the test you need that is not immediately obvious from looking at the fsdb and log files?

Understand the test history

Get a sense for whether the test is reliable or whether it is prone to failure. Is this a new test? What was the historical pass rate? What has caused it to fail in the past? What are they trying to test? Are they implementing a new feature or changing RTL in your area that perhaps you were unaware of or simply forgot?

Understand the boundaries of your knowledge

There will be times when you are the expert, but you do not have a clear answer for why the failure occurred. Wild speculation can send the requestor on wild goose chases that waste their time and damage your reputation as a Validator. It is okay to guess, but you have a responsibility to clearly identify what is a guess and what you know.

Whether you assume final responsibility to finish the debug or not, it is important to help the other person understand the progression you go through when approaching the debug problem. They may never become an expert on your level, but you will help them to get farther into the debug next time. If the root cause ends up as a test or environment issue, this will help them fix other tests that might suffer from the same problem.

6.9 Answering questions

The way you answer questions is as important as how you ask questions, and will go a long way towards establishing strong productive relationships with Design, Architecture and the rest of Validation. If your answers are poorly constructed, you will find yourself answering the same questions repeatedly. If you develop a history of bad answers, you will find that questions of all types will eventually stop coming...which means the other team members have written you off.

Understand the question (helping them)

Strive to ask open-ended questions that help the other person construct a better answer.

When confronted with a garbled email or incoherent questions in your cube, before attempting to answer, ask pointedly what information the other person needs from you in order to make progress. This will help to sift through the jumbled words and get to the point. You will be surprised how often people cannot even identify what information they need. Help them figure it out.

Understand the question (helping you)

Occasionally you will receive a perfectly constructed open-ended question that you do not understand. Admit you do not understand, and do not hesitate to ask them to restate the question. If the question is extremely detailed, do not hesitate to ask them to provide a high level summary of the issue. You do not want to waste time answering a question you do not understand in the hopes of saving face.

Reward effort

When someone has clearly put thought into an email or has taken solid notes in their debug effort, make sure to reward this with equally thoughtful, detailed, prompt replies. Delayed or dismissive replies discourage individuals from working with you, and do not support a collaborative work environment. Your life will be much better and the team will be more efficient if solid work is recognized and treated as such.

Documentation

If you receive the same question from multiple people, consider whether it would be helpful to create documentation that you can post and point people to. If you are already pointing people to documentation or receive follow up questions, consider this a useful form of feedback to improve your collateral, or improve project documentation by adding further descriptions in the HAS or collateral (test bench, tests, etc).

6.10 Reflecting on the failure & debug process

At the end of the debug process, regardless of your grade or your time on the team, YOU are the expert on this failure. As the expert, you have the responsibility to leverage your new knowledge to benefit the team whenever possible.

After you have trudged through several hours (or days) of debug, you have likely identified some things you wished you had known earlier or a lack of tools/documentation that would have made the debug process much easier. It is important to capture these discoveries somewhere, and take action where appropriate. All of the following require initiative from you to save other team members effort in the future.

- Would a TE enhancement be useful?
- Should a DAHelp ticket be filed?
- Could a better checker or assertion have caught this failure earlier?
- Would an RC file be useful?
- Is there a significant hole in the spec?
- Is there a hole in the test plan?
- Are there incorrect assumptions in the HAS or SPEC?

Finally, look to see if any other failures in Triage could be due to the same issue, and disposition those before someone else wastes their time repeating your work.

6.11 Debug training methodologies

Different Validation groups use several strategies to help train and develop newer Validators.

6.11.1 Injecting a bug into a clone

Clone a model and change RTL to create a bug, and then run tests to generate failures for debug.

Pro: Senior Validators know the bug, which saves them from having to debug the failure.

Con: We are not improving model health since these are not bugs that exist in the released model.

6.11.2 Interactive debug led by an expert

Inexperienced Validators can look over the shoulder, and ask questions as experts go through the debug process

Pro: The inexperienced Validator sees a full debug process with the inevitable wrong paths.

Con: Depending on the failure, this may be a long process that eats the time of two Validators instead of one. Also, having a steady stream of questions might impede the progress of the

expert. The new Validator may not be able to follow the process, as she/he is not the one tracing the debug.

6.11.3 Interactive debug with an expert monitor

Inexperienced Validators can lead the debug with the aid of an experienced Validator looking over their shoulder. The experienced person can help them steer close the problem area, while letting the new person explore.

Pro: The inexperienced Validator leads the debug, and sees a full debug process with the inevitable wrong paths.

Con: Depending on the failure, this may be a long process that eats the time of two Validators instead of one.

6.11.4 Group debugging sessions

During a group meeting, choose a failure from triage and debug it as a group.

Pro: Experienced Validators may learn a new tool that is available. Inexperienced Validators will see a full debug process with the inevitable wrong paths and plenty of discussion by the experienced ones.

Con: Depending on the failure, this may be a long process that may not even lead to a root cause in the time allotted. See above.

6.11.5 Presentations of a root caused failure

Choose someone from your group to walk the entire group through the debug of a failure that led to a filed bug.

Pro: The presenter can describe the debug process succinctly.

Con: The wrong paths followed by the Validator are not typically shown in this process. Understanding how to recognize a wrong path and back out is a difficult skill for new Validators to learn.

7 Summary

To newcomers debug may seem like a black art. Given time, experience, and the techniques discussed in this chapter, Validators can become very efficient at debug and learn to teach others debug as well. Be curious, ask questions, document findings, and do not forget the thank you email to those that helped you root cause the failures.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.



The Art of Pre-Si Val: Chapter 17

The Life of a Bug

By: [Matthew Plavcan and Leslie Ong](#)

1 Abstract

This document describes the lifecycle a bug follows from inception, through discovery, documentation, disposition, and possible termination. Each of the stages of a bug are explained, with details of who is involved, and why each step is important. The focus of the document is the interaction of roles across the team in handling the bug, and the possible steps that might need to be taken to validate a bug or improve future bug finding.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	10/05/2004	Initial Revision	Matt Plavcan	uAV/DV Meth. WG
1.1	11/25/2004	Draft Rewrite	Matt Plavcan	uAV/DV Meth. WG
1.2	12/10/2004	L1 Review changes	Matt Plavcan	uAV/DV Meth. WG
1.3	3/6/2004	L2 Review changes	Matt Plavcan	Rick Zucker
1.4	6/15/2005	L3 Review	Matt Plavcan	Paul Schwabe
1.5	2/25/2012	2012 Rewrite	Matt Plavcan	Michael Bair
1.6	3/17/2016	2016 Update	Leslie Ong	Michael Bair

3 Contents

1 Abstract.....	407
2 Revision History.....	407
3 Contents.....	408
4 Purpose.....	409
4.1 Why talk about bugs and bug tracking?	409
4.2 Why document and track bugs?	409
5 Background Concepts.....	410
6 Stages of a Bug	411
6.1 Inception of the Bug.....	411
6.2 Discovery of the Bug.....	412
6.3 Determining the Root Cause.....	413
6.4 Filing Bug Reports	414
6.5 Fate of the Bug	416
6.6 Fixing the Bug.....	418
6.7 Validating the Bug.....	418
6.8 Creating/Changing Validation Infrastructure	419
6.9 Closing the Loop.....	420
6.10 Reintroducing Bugs	420
7 Summary.....	421
8 Future Work.....	421
9 References.....	421

4 Purpose

4.1 Why talk about bugs and bug tracking?

Every engineering product ever designed has had design defects. Intel processors are some of the most complex engineering projects in existence. Even the final shipping versions of a processor will contain bugs. Finding, root-causing, fixing, and validating bugs is a continuous activity throughout the project. This process is not necessarily systematic: There is a variety of means to detect, discover, and debug them, and many of those techniques could be classified as art forms! However, the process of tracking and documenting bugs must be systematic in order to insure an orderly progression towards a healthy product. Unfixed bugs are a danger to the brand value of the company. If they are unknowingly unfixed, they pose a serious risk to the success of the product. Known bugs must be documented for errata purposes, to assist in addressing customer concerns.

4.2 Why document and track bugs?

The team should endeavor to accurately track bugs, so that:

- **The bug can be reviewed by other team members:** A bug report primarily serves as a communications channel for the team to effectively exchange information about the bug and its related failures. Documenting a bug's exact nature allows the team to find a way to address it that is appropriate to the scope of the problem and the point in the project at which it is discovered. A team's collective analysis of a bug exceeds what a single individual can contribute.
- **A disposition can be assigned to the bug:** The description of any proposed fixes for the problem are added to the bug report. All interested parties will have a chance to review them and comment on any problems or issues they might see, before they are introduced into the design. Depending upon the bug's severity and impact, it might or might not be fixed. Bugs that do not pose an immediate problem might have their fixes postponed until a later revision of the design.
- **The bug can be tracked throughout its lifetime:** The team should be able to follow the bug's status and progress of any fixes. As additional information regarding the bug or related bugs is discovered, team members can append or link that data in the report. New failure modes for the bug can be associated with existing bugs. When a change is committed to the design to fix the bug, the tracking system enables the team to understand when and where the fixed code has propagated.
- **Future failures can be analyzed faster:** Even though a bug has been found, it might exist in the design for a long time before it is fixed. A subset of the information associated with a bug report is a searchable database of documented failures. Using bug report failure information, debuggers can quickly and accurately match their failures with known bugs. Where a failure (triage) database exists, the entries for those failures should link to the relevant bug record.
- **Related bugs can be discovered:** Often, multiple bugs with like characteristics will exist in the design. By analyzing how one bug occurs, the Validation team can devise and implement

methods for detecting similar ones. Trends can be analyzed to provide information to the team regarding the types and causes of the design’s bugs. This enables the team to improve the overall health of the design.

- **The team can learn from the bug, to avoid similar problems in future designs:** Processor implementations often share common structures and algorithms, and many projects reuse or base substantial amounts of logic on code or infrastructure from previous designs. When a bug occurs in one design, it is likely to occur in others that implement the same feature. By documenting the bug, future designs can see the pitfalls of the particular implementation and take preemptive steps to avoid repeating those mistakes.

5 Background Concepts

A *bug* is a divergence between the expected behavior of an organized system and the actual behavior of that system. *It causes unexpected or undesired behavior of the design, in terms of architectural results, performance, power consumption, or reliability.*

Bugs are a natural but undesirable part of the design or integration process. They result from a mistake in the design: at some level, a part of the design was either not understood properly or not implemented incorrectly. The specification could be inaccurate or incomplete, or the implementation did not match the specification. Human error comes into play too, when designers integrate an IP and a signal or two are incorrectly hooked up; or temporarily tying off interface signals as a workaround and forgetting to undo this later on.

A *failure* is the symptom of a bug: it is an incorrect behavior of the design that is caught by a checker. This is an important distinction, because the mapping is not one-to-one. A single bug might have several different failure modes. Likewise, if a checking condition is ambiguous or far from the logic that exhibits the bug, a single failure message can map to multiple bugs. Bugs can often setup a latent condition, such that the test might fail long after the original problem occurred.

Various activities during the design and integration process create bugs. During architecture prototyping, high-level bugs sneak into function and protocol definitions. When a design specification is written, bugs are introduced by not addressing some important boundary conditions. The bugs will often lie in the gaps not defined in the collective specifications, or in the ambiguity of the language used to describe the behavior of the logic. As RTL is coded, bugs are introduced into the design, either through misinterpretation of the specification or quirks of editing (copy-paste errors, unclear or ambiguous naming, etc.). As older bugs are fixed or new features added, those logic changes can also introduce new bugs.

Categories of bugs can range from a simple typo in the RTL code to an architectural hole that requires the addition of complex logic to fix. Bugs that require simpler interactions to expose are typically easier to fix, but are also usually found first. (This is reason that the often mention “onion-peeling” phenomenon occurs: The bugs are removed layer by layer, until very few or none remain.) The category of bug often dictates which methods will be most suited to detect it.

Validation is the process of preventing, detecting and enabling the removal of these bugs so that the design will function correctly.

"Finding a bug should be a cause for celebration. Each discovery marks an incremental improvement in the quality of the design." - Doug Clark, Digital Equipment Corp.

The sooner a bug is found, the sooner the team will be able to analyze and fix it. Even if a bug escapes early detection and is discovered so late that it would cause a product recall, it is still beneficial: Intel found the problem and can address it proactively, rather than reacting to mass Internet rumors or angry outcries of customers. Several bugs in the past have escaped to the outside world and were discovered by third parties; it is always an undesirable scenario for someone else to find a problem with our products before we do.

The time in the project when a bug is found affects how the design team might try to fix the bug. As seen in [Introduction to Pre-Silicon Validation section: Hardware change cost model](#), the cost and risk of fixing a bug typically increase as the project moves forward. Early in the project, the team is likely to fix nearly all bugs. At that stage of the project, various options to fix a bug will typically be available, and the effort to fix and risk for change are reasonably tolerable. A project with an imminent tapeout has little tolerance for either, and often foregoes fixing bugs to avoid the high effort or risk. As the project heads towards tape-out, the impact of a bug will have to be significant in order to justify the effort to fix. This is because any change to design at this point will likely push out the tapeout date, which disrupts Intel's commitment to customers and ability to sell the product within the targeted season. Proposed fixes will be limited to what the backend team can absorb, and Validation will have very little time to show that the risk associated with the proposed fix is low.

Intel also needs to push PRQ with lesser steppings. This is necessary for us to be a competitive force in the SoC world. This also means that validation will have to be strategic, so high-impact bugs can be found as early in the project as possible.

The best time to catch a bug is in the early stages of a project, when design has the flexibility to create robust fixes and validation can show that the fix works and does not break anything else. Ideally, bugs are caught as soon as it gets into design. There are a variety of practices that can be used: intelligent development environments, incremental feature development, code reviews, early testing feedback. If the majority of bugs do not survive long enough to infest the design, then the team can discover the remaining bugs more efficiently.

6 Stages of a Bug

6.1 Inception of the Bug

At some point in the design process, the bug comes into existence. Depending on its complexity, the bug might not be obvious at this point; it can remain hidden in the logic, lying in wait. It will not cause a failure unless a test stimulates its failing scenario.

A bug can “evolve” beyond its original form either when it is incorrectly fixed, or if other changes to surrounding logic alter the inputs of the buggy logic. If the logic designer does not understand all of the interactions involved in a protocol, their changes can fix the simple failing case, but part of the original bug remains. New bugs have a tendency to require more complex interactions than the bug they replace. They often require specifically targeted testing and checking to expose them.

6.2 Discovery of the Bug

The Validation team's charter includes removing uncertainty of incorrect functionality by thorough testing and root-causing/debugging failures to find bugs.

"If it isn't tested, it's broken" - Colwell's Law.

To discover a bug requires some form of checking to detect the undesired output. Just as lack of testing allows bugs to thrive, a lack of checking to observe the bug allows a test that triggers an incorrect behavior not to fail.

"If it's tested but not checked, it's still broken" – Bentley's Corollary to Colwell's Law

During the life of the project, the Validation team's testing and checking become increasingly comprehensive, strengthening the chances of finding the bug. Simple (easy to write) checkers typically catch simple failure modes early in the project. However, some bugs have such complex triggering conditions that they will make it past the pre-silicon validation effort.

A variety of validation techniques can find bugs. Some typical tools in the validation toolbox are:

- **Dynamic Testing/Checking:** Functional Validation finds most bugs by testing in a simulation with checkers for functional correctness. The tests are a means of targeting bugs: they can focus on a single condition (directed test), or multiple areas (random testing). The test might be self-checking, or rely on external checkers to fail the test if the simulated behavior is wrong. Checking in this context typically takes the form of either low-level assertions in the RTL, or higher-level abstract checking in the environment.

See [Stimulus](#) and [Checking](#) for more details.

- **Coverage:** Validation uses coverage monitors to detect which areas of the design are healthy. The presence of coverage (with an appropriate checker) can indicate the likelihood that a bug does not exist in that logic. Contrarily, a coverage hole can indirectly indicate the presence of a bug, because that particular area of the design has not yet been tested. However, coverage is an indirect bug finder: the presence or absence of coverage can never *prove* that a bug does or does not exist. Because of these reasons, coverage is often an initial step in finding bugs and relies on the other methods listed here to narrow the scope of inquiry.

See [Coverage](#) for more details.

- **Formal proof validation:** A structured analysis of the code can show that a part of the design exactly matches its specification. By constructing a set of high-level properties that map to the scenarios the logic should correctly handle, they constitute a checker that covers a section of the design. A contradiction between the behaviors of the two models can reveal the presence of a bug.

See [Introduction to Formal Verification](#) for more details.

- **Application Failure:** For post-silicon bugs, physical processors exist, so real programs can execute on them. A failing program might be an indication of a bug that has escaped the pre-silicon validation process. However, because of the number of hardware and software components that are interacting, many causes can exist for a failure other than a processor logic bug. Many logical bugs that are difficult to find in pre-silicon testing environments are quickly revealed in post-silicon tests, although the effort to root-cause them is usually higher.

See [Interaction with Post-Silicon Validation section: Silicon sightings](#) for more details.

- **Inspection:** When a human being manually discovers a bug it is classified under “inspection”. Typically, review of specifications, architecture, code, test plans, or other documentation yields questions regarding specific behaviors. Answering these questions can show a divergence from the expected behavior and expose a bug. An experienced team member can examine code and divine that a bug exists through analysis, without ever writing a test. Validators might envision a problem by understanding the interactions of a system and write a test to hit a specific bug (aka “evil validation”), or someone running a test might inspect the results and discover a bug that no checker caught. Both of these situations are still classified as “by inspection” despite the use of a test to expose them. Because human observation is required for discovery by inspection, it is by far the most expensive way to find bugs.

6.3 Determining the Root Cause

Before a team member can file a bug, they must understand the details surrounding the behavior of that bug. *Root-causing* (aka: debugging) is the process of tracing the observed failure back to the actual bug.

Root-causing a bug might initially seem daunting for someone new to validation techniques, but actually it is very deterministic. It begins with understanding the design and architecture well enough to posit theories of why the failure occurs. Then the debugger deductively tests the logic to discern whether each theory is correct, in a divide-and-conquer process.

“When you have eliminated the impossible, whatever remains, no matter how improbable, must be the truth.” - Sherlock Holmes

For more details on the exact methods for debugging a test and discovering bugs, see the [Debug](#) chapter.

Finding the root cause reveals the type of bugs that is creating the failure:

- **Hardware/Firmware:** The logic does something that is contrary to the specification. The design will require a logic change to implement the specification correctly. Early in the project, some features might not have been implemented, and any tests that target them will fail. Logic bugs are the most common ones encountered during a project, and represent the highest return on testing.
- **Specification:** The specification might not detail the correct behavior for a given set of inputs: it could be wrong or incomplete. Changes to the specification without corresponding changes in the implementation also lead to bugs. Also, multiple specifications can be at odds with each other. Other logic bugs often hide behind specification bugs. Team members should be wary of any specification written based on an existing implementation, as it likely has bugs copied from the logic.
- **Checker:** The checker did not match the specification, so it is failing even though the design is performing a legal behavior. The checker will have to be modified to check the correct behavior. As the checker is a form of executable specification, building a checker based on existing logic is also at risk to duplicate a bug already present in the code under test.

- **Emulated Components:** Often test environments generate conditions that are not part of the prescribed behavior of the RTL, but help test the limits of what the RTL can do. This should be allowed on a case-by-case basis to ensure that a sufficient level of interface testing is being performed. The testing is beneficial, and testers should document any bugs found in these cracks between the possible and the impossible. It often is common to have code that functions “correctly” suddenly develop a bug because dependent input restrictions change. Emulated components also run the risk of being too restrictive, reducing the testing space to less than the possible conditions. Due to both of these issues, and the complexity of writing these emulators, careful review of their capabilities should be part of any Validation effort.
- **Test:** Tests represent a large opportunity for failures, as aggressive tests designed to target boundary conditions sometimes attempt behaviors that are not tolerated by the specification. Again, the testing should be allowed to push the definition of what the RTL can achieve, but only within a reasonable range or variation.

6.4 Filing Bug Reports

Once a debugger understands the bug’s major features, they should file an entry in the bug-tracking system. A bug report serves as a collection point for tracking all information regarding the bug, including: a summary of the bug, failure modes and its impact to affected products, the root-cause, options for fixing, the final fix, and the type of validation used to verify the fix.

A good bug report needs to be specific, comprehensive and reproducible. It should describe the failure modes of tests and exactly what is broken. It also should address all aspects of the bug’s nature and contain detailed information about the cause. Finally, a good bug report should contain the steps to reproduce the bug. By filing a good bug report, anyone affected by the bug will react to it more efficiently. DCCB (Design Change Control Board, which dispositions bugs) can discern the bug’s impact to the project for proper disposition, Designers can come up with the proper fix given the details of the bug, architects can think of other related failure modes to watch out for, and any Validator can reproduce the failure to verify the fix.

Early in the design process, bug reports can be brief, as bugs will be common, and the design is not at a mature level. If these bugs are not fixed correctly or are reintroduced, they are likely to be found by later testing and validation.

As the project progresses, the expectation of the design health rises, and the need for more detailed bug reports increases. Any bugs found during this period are less likely to be detected again, so diligence is required when documenting them. Nearing the end of execution, the bug report needs to be a thorough and complete document for replicating and tracking the bug.

A good bug report includes all of the items required for someone to understand, reproduce, fix the bug and validate the fix:

- **Summary of the bug:** This is an overview of how the bug works. The report should state what the expected behavior of the logic is versus what really occurs. The summary should be brief enough that a person who is not familiar with the logic can understand the basics of how the bug works. It should be detailed enough that an expert in the logic will have a reasonable comprehension of the incorrect behavior after reading it.

- **Scope of the bug:** The bug might lie in one piece of the design but have impact across several units or protocols. The bug report should specify the parts of the design affected by the bug, including micro-architecture, logic, and microcode.
- **Failure mode(s) for the bug:** The report should list the various symptoms that show that the bug's triggering conditions have been hit. There can be a variety of failure modes for a single bug, depending on the level of checking run with a test. A given checker might catch this failure early or in specific circumstances, while other instances of the bug will be caught later by more general checking. Also, different tests might excite the bug in different ways, leading to alternative failure modes.
- **Location of the bug in the design:** The report should describe what logic is broken to cause the bug. Unlike the summary, this is a detailed account and might include specific RTL code, microcode, or other elements quoted directly from the design.
- **Severity of the bug:** The severity of a bug is determined by several factors: How important is it to fix this bug? Does it inhibit major functionality of the design? Does it block testing? Does the bug exist in released parts? The severity is an indication of how much attention the design team needs to pay to analyzing and fixing the bug.
- **Test sequence to re-create the bug:** The report describes how to trigger the bug. This includes a description of what the relevant portion of the failing test does and how the design reacts to the test stimulus at each step. The sequence will typically include signal traces or other descriptions of machine state.
- **Test command line:** For inspection bugs, a test might not exist, but for all other cases, the report must include a way to replicate the failure. If a random test found the original bug, it might be an option to use that instance of the generated test. Alternatively, a Validator may elect to write a simpler directed test to target the bug. The directed nature of the test helps clarify which test elements are causing the bug. As an added benefit, it shortens the time for another person to replicate the bug. The failing test should be provided in a form that can replicate the failure for anyone, without dependencies on private user data or environment. The project typically employs a well-known storage location for this purpose.
- **When/how the bug was introduced:** It is interesting to note the circumstances surrounding the bug's introduction into the design. Bug reports can be linked to other similar bugs for analysis and trend-tracking purposes. This information is not always available, but is worth including if it exists, as it gives a feedback to the team on where they can improve their processes.
- **Possible Fixes/Workarounds:** When filer has a sufficient understanding of the underlying problem, they can add potential fixes to the initial bug report . This is an optional step, but in debugging the bug, they have invested a large amount of time analyzing the bug's nature, and an experienced team member will probably have a good grasp of what a reasonable fix is. Potential solutions for the bug can be discussed and logged in the report. The person making the change should connect the actual logic changes to the bug report when the bug is fixed. The mechanism for doing this can be as simple as posting the code in the report, or it might involve a more complex link between the source-code control system and the bug database.

A bug report that simply states “Test X failed” is woefully insufficient. However, there are some exceptions: Sometimes a brief bug report is filed to act as an early failure filtering database, so that other team members know that a particular failure is being debugged. This should only be done in cases where a standalone triage database does not exist. Sometimes team members may file a bug report before all the information is known, to allow others to collaborate on the debug process. These types of bug reports should be completed with the additional information after the debugging is complete.

An incomplete or incomprehensible bug report is an easy way to let a bug slip through the validation net: It might not be correctly root-caused. It might have a larger scope than first realized. It might not be fixed properly. It might not be validated properly. Care should be taken not to place too much speculation in the bug report. For instance: A mail discussion regarding how a protocol works might be a useful addition to the bug report. However, some of the content of that thread might contain irrelevant, speculative, or erroneous information (often in a different chronological order, and containing copious mail headers!) such that it can confuse people reading the report.

After tape-out, bugs are present in physical silicon and are potentially present in products that have been shipped to customers. At this point, any bug found becomes an *escape*. The procedure for escapes requires more documentation than just a bug report. See the [Post-Silicon Escapes](#) chapter for details.

The debugger is responsible for the bug until it has been root-caused and filed. It is worth noting that the debugger may not necessarily be the engineer that owns final validation of the bug fix. The validator of the bug needs to actively participate in the resolution of the bug, and document relevant discussion in the bug report. The better the documentation is of the bug and of the fix, the higher the confidence that the fix can properly be validated.

6.5 Fate of the Bug

Once the root-cause is known (and a report potentially has been filed), the relevant stakeholders (Architecture, Design, Validation) will discuss how and whether to fix the bug. Depending on the scope of the bug and the phase of the project, the management team might need to approve the changes. At a very late point in the project, corporate strategic decisions can be impacted by the changes, and every new bug will be reviewed to ascertain whether it poses a risk to the viability of the design.

The bug will be assigned one of the following dispositions:

- **Not a Bug:** Because bugs lie on the boundaries of understanding of the design, there are some instances where logic might appear wrong, but is not actually performing incorrect behavior. After careful analysis, these failures are determined to be non-bugs. The checkers or tests that caught them will be amended to avoid the failure in the future.
- **Fix:** The bug will be fixed in the current design model. Early in the project, the default behavior is to fix all bugs, because the fix cost is relatively small, and the design should be as “clean” as possible. Later in the project, the costs and/or risks for fixing bugs will drive other dispositions.

- **Won't Be Fixed:** the bug will not be fixed for this stepping, and it is anticipated that it will never be fixed for this design/product. There are several reasons why a bug would not be fixed (one or more of the following are true):
 - The bug did not cause an externally visible failure. Some bugs are not exposed in architectural state. Minor performance bugs (where the machine functions slower than it should) or non-production features (or defeatures) are good examples.
 - A workaround for the bug exists. A defeature, microcode patch, or software change can be used to avoid the conditions that expose the bug. Because of this, the bug can remain in the design but will not be hit. Careful analysis needs to be made to insure that no cases exist where the workaround does not cover the problem. Workarounds are typically used for late-project or post-silicon bugs, so another stepping is not required for functional correctness.
 - The bug only occurs in extremely rare scenarios. Sometimes these rare failures are deemed tolerable. However, for functional failures, this justification by itself is insufficient to not fix a bug, and an additional reason will likely be required. A clear example why this is true can be seen with the 1995 “FDIV” escape in the PentiumTM Processor, where the exact conditions for the failure were rare, but easily reproducible.
 - The cost to fix the bug is too high. In the later phases of the project, the design is at a mature level, and any change will require rework of the design, and revalidation of the changes. This effort might be greater than the team can incorporate into their schedule without slipping the project deadlines.
 - The risk in fixing the bug this late is too high. The bug might be simple to fix from a logic point of view, but the chances that the change will introduce a new bug are too high.
 - A customer has designed one of their products in a way that is dependent upon the “wrong” behavior. Changing the behavior would cause working software to be incompatible with the new hardware, so the bug cannot be fixed, even though it is technically wrong.
 - The bug does cause a functional problem, but no customer uses that feature. A consensus sometimes can be reached regarding whether customers will be exposed to this bug, but it is difficult to know that no one uses a particular feature; this is a very precarious disposition for a bug.

When designating a functional bug to not be fixed for the entire product lifetime, it results in removing/de-committing a feature. This type of problem potentially has high brand image/customer impact. Instead of doing this, bugs that meet some of the criteria for “no fix” typically are postponed to a later stepping, but prior to product release.

- **Fix in a Future Release:** The bug will not be fixed in the current stepping of the processor, but can be reviewed for being fixed in a later stepping. Many bugs found late in a project usually reach this state, due to either the risk associated with late bug fixes and the cost to fix the bug.

None of these dispositions are necessarily final: If the team discovers that a bug has a wider impact than was initially determined, it should be re-examined for a new disposition.

6.6 Fixing the Bug

The person responsible for fixing the bug should document the fix in the bug report. An exact copy of the changed (and well-documented!) code is an excellent way to document the fix, if feasible. If the bug lies in a test environment, checker or test, the owner of that validation infrastructure will fix it. If the bug lies in RTL, the logic/firmware designers are responsible for fixing it. Architects might be responsible for specifying fixes for bugs that affect multiple units or the high-level micro-architecture of the design. Intel publishes external specifications for many of its products. Bugs found in external specifications require that an update to that specification, and customers be notified of the potential change.

The team should analyze the proposed change and evaluate whether it will correctly fix the bug. Whenever possible, they should examine and test the fix before it is released into a mainline code repository. The Validator for the bug also determines whether it is possible to validate the changes easily. Even the initial code for a feature can be beyond ordinary validation capabilities. If it would take the team too long to validate certain logic, it might be sufficient reason not to implement the feature that way. A Validator should always be on the lookout for code that is too complex to validate successfully, as it represents a haven for bugs. Any team member can always suggest fixes for bugs, and someone more familiar with testing techniques can evaluate which ones might require less effort to validate.

6.7 Validating the Bug

Once a fix for the bug has been coded, the Validator must show that the change now makes the logic work correctly. The scale of effort required to validate a bug varies with the complexity of bug, the area of logic where the bug was found, and the amount of time remaining to validate the product.

For bugs caught in a feature that is being developed on an isolated branch (no other feature code being integrated), running the feature tests and regression suites are usually sufficient to insure that the bug has been fixed. If the bug is particularly complex, it should drive the development of tests around that feature that can adequately exercise the boundary conditions of the feature. In most cases, this branch only affects the team members developing the feature, and lifetime of the bug is extremely short (it's being caught almost as soon as it was introduced). As a result, no bug record is required, unless the bug has some particular characteristic (exceptional complexity, need for long discussion or outside stakeholder input) that merits filing it.

Bugs caught in the released model represent a more difficult problem: They have survived initial feature testing, or were a result of interaction of features that have been integrated together. These bugs require more carefully examination of the validation infrastructure, to see that the testing checking and coverage spans the complexity necessary to find similar bugs. It might be necessary to create specific new validation content. The documentation of these bugs should be higher than a typical feature bug, as it represents the team's learning about the design.

For a complex bug, or one that occurs very late in the project, the Validator needs to have a high confidence that the fix performs as expected. This can be a difficult proposition, for several reasons:

- The fix must not only work for the specific bug case, but for related cases.

Example: A bug in an adder fails to correctly generate a carry when an addition overflows. The original failing test had operands of $2^{15}-1$ and 1. The Validator should show that the adder now carries correctly for cases other than the two operands in the original test. (A formal proof might be useful for a case like this.)

- The Validator should show that complimentary conditions still work correctly. Correcting the logic for a bug could have changed the other related boundaries surrounding the failing case. A Validator should be careful to test those boundaries, including at least one complementary case, before marking a bug validated.

Example: Before the bug, the adder never carried. How does the Validator know that it isn't always generating a carry now? Test a contrary case.

- The bug might have moved in time. Changes to multi-cycle protocols can shift the bug's failure criteria. The logic fix might work for the original conditions, but bug is now in a new location.

Example: A snooping protocol with a five cycle wide time window has a bug where it fails to detect a snoop during the third cycle. A logic change could make the third cycle work correctly, but now one of the other timings does not work.

Early in a project, a Validator might not have to validate a bug exhaustively. The later in the project a bug occurs, the higher the standard of validating it becomes. Successfully validating bugs can require the addition of new tests, injectors, checkers, and coverage monitors to target the affected logic.

6.8 Creating/Changing Validation Infrastructure

The appearance of multiple bugs in the same area is an indication that the area should be a focus for additional testing and coverage. The Validation team is responsible for determining the scope of validation required for a fix and whether new tools need to be employed to target it.

One of the most difficult task a Validator sometimes faces is having a way of directly reproducing a failure. The tests need to produce the bug conditions on both the failing model and the fixed model. But if other changes have been incorporated into either design, test code, or test environment, a random seed which initially found the bug might not reproduce the failure. For this reason, though many bugs are found with random tests, it is sometimes useful to have a directed test to target the exact failure. Injectors can also be used to create boundary conditions that would be too complex or time-dependent to normally direct. These new injectors can also be leveraged for future random testing, though care needs to be taken in creating conditions that could not normally exist in a non-injected test.

The Validator should confirm that the checkers that target this area of logic correctly catch all failing cases for the bug. Bugs that are designated "will not fix" often require workarounds in the tests and checkers to avoid further failures. The changes should be carefully documented (and

tagged with the bug number in the source control logs) to make it clear that this expected behavior is due to a specific bug. Should the bug be fixed later, the workaround will need to be removed. Documenting workarounds in the bug report is highly recommended.

Validating a fix to a checker is often harder, since a correctly working checker will not fail unless there is also failing logic. (The best way to address this problem is negative testing of the checkers independently of the RTL, which requires some additional infrastructure in the test environment.)

New coverage monitors that focus on the area around the bug might also be required. The unit's coverage monitors need to have constructs that target the logic surrounding the bug. Frequently, a logic change will require that the existing coverage for that area be reset, as the conditions that were covered are not the same as the ones now present in the logic. Tests can incorporate self-coverage to insure that they have correctly hit the bug conditions.

Validation infrastructure should be created for long-term design health. The tests and checkers need to be robust enough that they will exercise the bug condition with minimal or no maintenance for the remainder of the project. This makes confirming the absence of previously fixed bugs an easy task later in the project. If a specific test is required to reproduce the failure, it should be added to the regression test suite. These steps automate the ability to detect the failure, reducing team effort and the risks to the project.

When a Validator has expended specific effort to validate a bug (especially if this required crafting infrastructure specific to the failure) they should add the details of how they validated the fix in the bug report. The validation comment can be as simple as a single sentence: "Ran test 'FOO' on model 'BAR', test passed." It can be a complicated description of the testing and coverage confirming that the bug logic has been validated under several scenarios. Leaving out this information will cast doubt on the validity of the fix, especially if a related bug surfaces.

6.9 Closing the Loop

After understanding all the interactions required to trigger a bug or family of bugs, Validators should evaluate their tools: tests, injectors, monitors, and checkers. Which of these were able to catch the bug? Which ones should have caught it, but did not? How could the bug have been found (and debugged) more efficiently? Running the same tests over and over with no new checkers or injectors will cause the bug finding rate to decrease, but does not actually show improvement in the health of the design. Because of the nature of bugs, the tools for hunting them down must incorporate feedback from their successfulness, and improve them in order to continue to find bugs. This is a fundamental philosophy of the validation process.

6.10 Reintroducing Bugs

When the team fixes a bug, another bug could be created by the logic fix. The complex nature of a processor implies that newly changed logic will interact with the logic around it. If the bug fix does not consider these interactions, a series of bugs in the same functional area can develop. In this way, bugs are Darwinian: They follow a pattern of specialization/evolution, with the validation effort playing the selection mechanism. Team members should learn to be bug taxonomists, able to categorize each bug by its characteristics. This enables investigating whether other bugs of that

particular type are present. Bugs with similar triggering conditions, failure modes, or root causes tend to be exposed around the same time, because new testing behaviors that hit a bug can also target the others like it. Similarly, a checker for a feature has a tendency to flush out a number of bugs as it is developed. Few bugs are so specialized and well hidden that they are unique, and have no relatives.

The introduced bugs have a high possibility of being more complex (and harder to find) than the bug they replace. Validators should take care to verify that the bug fix has correctly addressed the failing case without introducing other bugs. Random testing focused on the buggy area run with comprehensive checking or formal proofs guarding the specific logic are the best defense against this problem.

Architecture, logic and firmware also are re-used between many projects. After tape-out, any logic changes to a design require new steppings, which will share any bugs present in the first earlier steppings. When a bug is found, the debugger should evaluate its root-cause to see whether it could possibly affect other projects or steppings. When it does, they must pass the bug report to those projects. This is especially important when a bug occurs in a family of products, some of which might already be in the customer's hands!

7 Summary

Bugs exist in infinite diversity and combinations in modern microprocessors. Bugs are created at various times during a project, and are found through a variety of methods. Finding a bug is the beginning, not the end of the bug-finding process, and the Validator's involvement is required throughout the bug's lifetime. The Validator must fully understand the bug's nature in order to successfully validate it, and doing so might require additional validation tools not present at the time the bug was found. Bugs have similarities to each other that are useful in uncovering like bugs. Understanding how and when bugs occur in the design and how to track and document them are essential Validator skills.

8 Future Work

9 References

The Art of Pre-Si Val: Chapter 18

Indicators

By:[Mark Savoy](#)

1 Abstract

The purpose of the Validation Indicators (combined for now) section is to make the reader aware of indicators that are important to validation. Indicator theory is also explored, including advantages and pitfalls of using specific indicators.

This document is geared towards indicators important to validation. As such, validators, validation managers, and people whose jobs interact with validation will benefit from this document.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/12/2003	Initial Revision	Mark Savoy	Matt Kupperman / DPG-N uAV MWG

3 Contents

1 Abstract.....	423
2 Revision History.....	423
3 Contents.....	424
4 Purpose.....	425
5 Background Concepts.....	425
6 Indicators.....	425
6.1 EITVOX	425
6.1.1 Entry Conditions	426
6.1.2 Inputs and Dependencies	426
6.1.3 Tasks	426
6.1.4 Validation	427
6.1.5 Outputs	427
6.1.6 Exit Conditions.....	428
6.2 Indicator examples.....	428
7 Summary.....	454
8 References.....	454

4 Purpose

What are indicators for? To get started, it is convenient to look at a problem statement about contributors from both an individual contributor and manager standpoints.

Individual contributor:

Any good validator will tell you that while their coverage numbers may look good, coverage alone doesn't tell you the whole story. Other than your direct manager, who really gets to hear the rest of the story? What invariably gets sanitized up the management chain is the all-important indicator of where we are, best expressed as a boolean answer to the question "Are we where we should be?"

Manager:

Any good manager will tell you that while indicators are invaluable for planning purposes, indicators aren't just useful for appeasing pointy-haired obsessives. The process of rolling up indicators should also inspire contributors to improve those indicators by meeting and exceeding commitments that are deemed indicator-worthy. The more this process drives the right behavior, the better.

Taking into account these two conflicting viewpoints on indicators, we can summarize the main purpose of indicators as follows:

Indicators serve three main purposes:

- Provide an accurate measure of progress for ongoing management consumption. This includes everything from focal ranking to determining when resource balancing should occur.
- Drive correct behavior of individual contributors by forcing them to present progress toward their commitments in a public and accountable manner.
- Generate statistical data that may be useful for future consumption (planning, analysis, etc).

5 Background Concepts

Knowledge of basic validation concepts is beneficial, but not necessary for understanding the concepts discussed in this document. The concepts are the important parts. If you aren't clear about specific validation terminology, it is sufficient to make a guess.

Indicators are strongly correlated with a specific set of tasks. Not all tasks should have indicators. If the cost of setting up and maintaining an indicator is greater than its benefit, then no indicator should be used.

6 Indicators

6.1 EITVOX

6.1.1 Entry Conditions

Indicators are a way to *indicate* the status of a task or set of tasks. To that end, the only real entry condition is to have a task or set of tasks defined. Goals do not necessarily need to be known prior to starting the indicators. In fact, the process of creating and using indicators can be a very useful part of setting goals.

6.1.2 Inputs and Dependencies

The major input is going to be what you are measuring. If you are measuring coverage, then the input is the coverage results. If you are measuring milestones, the input is the completion of milestones

Somewhat early in the process, goals should be applied to the indicator. Goals are often strongly influenced by external factors, such as rtl development status, or a proposed project schedule. As a result, those external factors are inputs.

There needs to be a structure for rolling up, displaying, and interpreting the results of the indicator. If this doesn't exist, then the indicator is worthless, for it will not drive individual contributors to do the right thing, nor will it tell management how things are going.

6.1.3 Tasks

- The indicator must be defined. This task should be done as collaboration between management and technical contributors familiar with the task. The key goals here are to come up with a good indicator that maximize usefulness, while minimizing intrusive. Note that this does not necessarily mean making the indicator completely automated. It is often a **GOOD** thing to have your individual contributors roll up their status manually. It forces them to look at the numbers and understand how they match to the goals. This is one of the most powerful ways an indicator can drive correct behavior.
- Once the indicator is defined, a methodology for rolling up and presenting the required data must be put in place. Often, graphing the data trends (which we like to call "pretty pictures") can facilitate a more thorough understanding than just looking at numbers in text. As a result, having data rolled up at regular intervals is very important. It could be daily, weekly, monthly, or any other period that seems appropriate. You don't want to create an unnecessary burden by requiring too frequent roll-ups, but you also don't want it so infrequently that performance to the goal can be ignored for too long a time period. In most cases, a period of one week works well. Another benefit of weekly rollups is that it allows weekly status reports to do double-duty for rolling up indicators. All of this information could be hand-maintained into an Excel spreadsheet, or some more elaborate mechanism could be defined.
- A goal line (or other delineation) usually must be determined and placed on the indicator. This is necessary so that when you talk about the indicator, it can be used to encourage improving to meet the goals. If there isn't a goal line, it becomes hard to justify why more effort needs to be put in to make the indicator look better.
- All of the data must be rolled up on the regularly scheduled intervals. It should be the responsibility of each contributor to roll up their results. If a person's results are not provided in a timely fashion, it is usually acceptable to assume that person's results haven't changed

since the previous roll-up. When rolling up results, each contributor (and manager in the roll-up chain) should look at the data and try to understand what it means. This is where most of an indicator's benefit comes from, because this is what usually drives correct behavior.

- Once all the data is rolled up, the official indicator results need to be published so that all concerned parties can view them. It is a good idea to show these indicators during staff meetings and discuss how things are going. Encouragement to “catch up to the goal line” or to “keep ahead of the goal line” also helps to drive individual contributor behavior.
- If the indicator shows a trend that does not look like the goals will be met in time, then flags need to be raised to inform all concerned parties about the issue. A more formal “smoke alarm” may be employed to show visually where flags would need to be raised. Both managers and individual contributors must work to resolve issues when smoke alarms are triggered. Resource balancing, reprioritization, and re-setting goals are things to consider when addressing the flag. In cases where the indicator doesn't match reality, it may be best to find a more appropriate indicator to use.
- Individual contributors are responsible for modifying their behavior to do the right thing. Hopefully the right thing will also improve the indicator, but this is not always true. If a bad indicator is an obstacle to “doing the right thing”, then the contributor should notify management and do the right thing instead of just following the indicator like a lemming.
- Similarly, individual contributors must analyze the indicator data for possible misconceptions it can create, and flagging this to management. For instance, if you have 99% overall coverage due to one huge construct with 100% coverage, but you still have a whole lot of small constructs with zero coverage, then you aren't anywhere near 99% done. Likewise you could have 100% coverage on all of your constructs except one large one, which brings you down to 1% overall coverage. In neither of these cases is the overall coverage percentage meaningful as an indicator.
- Individual contributors should predict negative or positive indicator trends for the future. This is important because it gives management an early-warning signal that things are changing. For instance, if rtl churn hits you really hard one week, you can tell your manager that next week your coverage numbers are going to go down instead of up. Another example is that just because you achieved 50% coverage in one quarter, it does not mean that you can push it to 100% coverage with another quarter of work. Bugs and boundaries are Darwinian, which means that harder conditions are by definition the last ones hit.

6.1.4 Validation

Indicators don't really have a point when they are “successfully accomplished”. Instead, indicators are simply retired when they have outlived their usefulness. There is no hard line that says one day you have more work to do on the indicator, and the next, there is no work left to do on the indicator (except in the rare cases that you have reached 100% on an indicator that cannot decrease). Usually, work on the tasks that the indicator tracks will continue even after the indicator is retired, and a small amount of new information could be gleaned from its continuation.

6.1.5 Outputs

- Regularly published indicators where all concerned parties can view.

- Drive individual contributor behavior to “do the right thing right”
- Provide management with an understanding of the status trend so that planning decisions can be made.

6.1.6 Exit Conditions

Most indicators eventually become useless. For example, if your indicator shows that results have exceeded goals, and the nature of the indicator does not allow for it to go down, then the indicator is no longer useful. Once the ROI of rolling up and analyzing the results goes to zero, it is time to retire that indicator. There are even circumstances where a goal has not been reached, but indicator retirement is appropriate. For example, in a milestone indicator, it is not necessary to keep the indicator up until all milestones have been reached. Remember, the main purposes of this are to drive correct contributor behavior, and provide management with an understanding of status. If neither of these goals are being facilitated because management considers a job essentially done, it can be acceptable to retire the indicator.

6.2 Indicator examples

There is so much data in the validation arena that you could conceive of an almost unlimited number of indicators. Presented here are some indicators that have been used in the past, as well as indicators that may be useful to implement in the future. Benefits, pitfalls, and theory will be addressed for each one.

Each indicator is given in a table format that contains the following information:

- Indicator title
- Is it currently used?
- Frequency of rollup
- Theory
- Data that it measures
- How it drives individual contributor behavior
- How it drives manager behavior
- Benefits
- Pitfalls

Indicator title	FED Milestones
Currently used	Yes

Frequency of rollup	Weekly
Theory	<p>During FED, there are a myriad of tasks that need to be done to both exercise the rtl model, and get prepared for execution. Each individual task isn't regular enough or big enough to be an indicator unto itself. By calling these tasks milestones, a good aggregate indicator can be created.</p> <p>Note that some of these tasks are ongoing, and not individual data points. The spirit of these ongoing tasks is simply to ensure an appropriate budget of time is allocated to keep those efforts going. If ongoing time has been spent on those tasks, the milestone can be considered complete.</p>
Data measures	<p>it</p> <p>The different FED tasks are divided into milestones that show when significant pieces of work are complete. For instance, in a new design, getting tests written to exercise specific areas of functionality could be milestones. In a proliferation, getting good tests from the previous project running could be a milestone.</p> <p>Some useful milestones (many of which have been used in the past) include:</p> <ul style="list-style-type: none"> • Write exercise testplans for specific design phases (a design phase is just a way of delineating when specific features are coded into the rtl) • Update CTEs and checkers for specific design phases • Tests to exercise specific new functionality written • Tests to exercise specific new functionality passing • Random tests running • Tests ported from previous project • Support FC debug, design pre-release testing, perf analysis • Support debug for other projects • Support model build • Support hiring effort • Complete DV exercise • Begin Execution • Add all new features to existing random templates • Start testplan writing where model stability allows • Start porting prototypes from prior project • Study SMC and lock protocols and develop test plan • Support tool enhancements

	<ul style="list-style-type: none"> • 50% of testplans complete • 50% coverage monitors coded • 15% overall coverage • Periodically running uAV templates • Updated regression list • Testplan started • Testplan 10% review complete • Testplan published
How it drives individual contributor behavior	During the hectic time of FED, a validator's time is often fully consumed in reaction mode. This indicator helps them keep an eye on the big picture, and work to get the milestones done as time allows. Since the milestones usually include test writing/porting, this can help direct them to exercise previously untouched code, and find bugs even faster.
How it drives manager behavior	Slowly moving milestones is an indication to management that the validation team's time is being spent primarily in reaction mode to bugs. This becomes an early indication of when the rtl starts stabilizing enough for validation to spend time pushing for milestone completion. Complete lack of milestone completion should be investigated to make sure that the right trade-offs are being made, because the milestones do need to get done eventually, and they take non-trivial time to complete.
Benefits	The milestones are by definition tasks that need to get done, with non-trivial effort involved. This indicator provides a way to ensure that none of these tasks get forgotten and left undone during the bug frenzy known as FED.
Pitfalls	The number one thing a validator should be doing during FED is enabling the designers to make forward progress by finding and debugging a plethora of bugs. That task does not fit well in a milestone indicator, and as a result, this indicator could be counterproductive if it is the primary way of measuring performance. When the rtl is yielding so many bugs that your validator is 100% occupied with debug, this indicator needs to take second fiddle.

Indicator title	Directed test pass/fail/coded
Currently used	Yes
Frequency of rollup	Weekly
Theory	Directed tests are defined in the testplan for a reason: they want to exercise conditions that are not appropriate for coverage for whatever reason (for

	instance, execute an LEA instruction of various MODRM combinations. In the instruction decoder, you can see the MODRM combinations, and in the back end of the machine, you can see the LEA instruction retiring, but you have no way to match up that decode with that retirement. Directed tests can be written to guarantee those instructions decoded and retired properly).
	As a result, it is important to make sure these directed tests get coded and pass.
Data it measures	Directed tests passing, failing, coded but not run, and not yet coded.
How it drives individual contributor behavior	Causes focus on completing the directed tests and debugging the failures so that bugs in the design can be found.
How it drives manager behavior	
Benefits	Provides a good measure of progress of directed testing status. Very straightforward.
Pitfalls	If too much pressure is applied to increase the pass rate, stupid things could be done, like removing specific tests from the regression (and thus the testplan), disabling checkers, or changing the test to not hit the failing conditions. Remember that passing tests tell you almost nothing. Only the failing tests tell you something about the remaining bugs.

Indicator title	Coverage coded
Currently used	Yes
Frequency of rollup	Weekly
Theory	It takes a long time to code those constructs, and lends itself well to setting goals to drive continuous progress.
Data it measures	<p>This can roll up in a couple of different ways:</p> <ol style="list-style-type: none"> 1. Testplan entries coded 2. Total conditions coded <p>In general, I suggest using the entries coded rather than the conditions coded, because the number of entries is more stable than the total number of conditions.</p>

	The indicator chart should include an indication of the total coded out of the total expected. Note that the total expected will change from week to week, and in general, it is best to not try and compensate for those changes.
How it drives individual contributor behavior	Individuals can see how their contributions are leading to the long-term trend of coding. It allows them to compare their work vs others (setting up a competitive scenario), and also shows pretty clearly when trends are falling behind, and extra attention will be required to get back on track.
How it drives manager behavior	Gives managers a basis for convincing individuals to bring up the coded conditions. Also provides an early-warning mechanism when it looks like this task could slip its goals.
Benefits	Excellent way to drive coverage monitor coding to try and keep to the project schedule.
Pitfalls	<p>Often coverage monitor coding falls behind for good reasons. Perhaps the validator's time is fully consumed in debugging. Perhaps the rtl isn't stable enough to support the coverage monitor. If you push to have coverage monitors coded before the rtl is stable enough, you are just creating a huge amount of rework due to rtl thrash. So, if you are not careful, this indicator could really drive incorrect behavior and thus create significant unnecessary work.</p> <p>It is not uncommon for coded conditions to take a jump in the couple of weeks prior to a commit date. Often this is an indication that there are good reasons NOT to drive condition coding yet (rtl churn), but that the validator stopped working on other appropriate tasks in order to fulfill the committed goals as measured by the indicator.</p>

Indicator title	Conditions hit
Currently used	Yes
Frequency of rollup	Weekly
Theory	<p>How do you know when you have driven out all of the bugs in the design? You don't. However, if your coverage doesn't look good, then you know that there are a lot of identified boundary conditions that haven't been exercised.</p> <p>Collecting high levels of coverage is essential for being ready for tapeout.</p> <p>It is sometimes appropriate to consider directed tests as coverage hit. Specifically, if the testplan called for a directed test, then that test is responsible</p>

	for ensuring that the condition is hit, or else the test should fail. This makes it somewhat easier than keeping a separate indicator.
Data it measures	Conditions hit vs total conditions.
How it drives individual contributor behavior	Drives individuals to write better tests to hit unexercised boundaries in the design. Remember, if you haven't exercised it, you must assume it is broken. As a result, driving coverage will systematically reduce the number of places where bugs could be hiding.
How it drives manager behavior	Allows management to understand the state of coverage and make plans accordingly. Managers should be aware that coverage tends to have roughly an S-curve to them. It starts off with exponential growth, then slows asymptotically. Remember that hitting the last 10% can take 90% of the effort. Also note that coverage will also go down from time to time due to coverage resets or new conditions being coded.
Benefits	Without high coverage, you know that you haven't hit lots of boundary conditions, so it is very important to drive coverage up.
Pitfalls	Coverage doesn't find bugs directly. It points you to deficiencies in your testing. Only by plugging those testing holes do you find bugs. Directed tests can be written to hit many boundary conditions, and is often the most efficient way to hit some of those conditions. However, this can also lead to a false sense of security, because the testplan CAN'T include all boundary conditions where bugs could hide. By writing directed tests and hitting only the narrowly defined conditions from the testplan, you might collect 100% coverage and stop looking in that area for bugs. However, a well written random test will hit boundary conditions around the testplan conditions, and give you higher chance of hitting the really nasty bugs.

Indicator title	Cycles
Currently used	Yes
Frequency of rollup	Weekly or Daily
Theory	In order to forecast how many new machines are required, we must watch use of the existing machines. Also, if machines ever go idle, they are being wasted, and behavior should be modified to use those resources more evenly (we always have more jobs to run, but may not be submitting them optimally).
Data it measures	Number of rtl cycles run, and the amount of time cpus are idle.

How it drives individual contributor behavior	Identifies when extra cycles are available so that jobs can be submitted more optimally. Also indicates if we have a shortage of cycles so that individuals can plan to submit jobs so that high-priority tests aren't held up by lower-priority ones.
How it drives manager behavior	Enables resource planning for purchasing and allocating machines.
Benefits	Critical for balancing money spent on new machines with the need for productivity through increased compute power. This indicator can also point out some gross problems in the pool of machines so that those problems can get appropriate attention.
Pitfalls	None?

Indicator title	Test pass rate
Currently used	Yes (mainly by other groups, like AV)
Frequency of rollup	Weekly
Theory	The more tests that are passing indicates greater rtl health, and runnable tests.
Data it measures	Tests passing, tests failing, tests not written, and tests not run.
How it drives individual contributor behavior	Encourages validators to debug and fix test problems and environment problems, as well as file rtl bugs.
How it drives manager behavior	Raises flags about the model health if the pass rate is lower than expected.
Benefits	If the tests aren't passing at all, then they need to be debugged and the problem needs to be fixed.
Pitfalls	High pass rates could also indicate insufficient tests to hit bugs or insufficient checkers to catch those bugs that get hit. It is possible for the healthiest cluster is the one that has the lowest pas rates.

Indicator title	Bugs filed
-----------------	-------------------

Currently used	Yes
Frequency of rollup	Weekly
Theory	<p>Bugs are good. If a bug exists, we want to find it and file it. If the bug rate is high, then it indicates we are doing our job well. If the bug rate is low, we need to investigate why. Low bug rates could mean many things, such as (roughly in order of likelihood throughout the project):</p> <ul style="list-style-type: none"> • Tests are insufficient to find the unknown bugs. • Unknown bugs are hiding behind already known bugs. (Social nature of bugs) • Tests are hitting unknown bugs, but no checkers are catching the incorrect behavior. • All the major bugs have been found, and validation is trying to find a dwindling number of more and more hard-to hit bugs (Darwinian nature of bugs)
Data it measures	Number of bugs filed. Optionally can keep track of the priority of those bugs.
How it drives individual contributor behavior	Lots of good bugs can show a productive validator hitting new boundary conditions.
How it drives manager behavior	Bug rate is a leading indicator of coverage rate. If the bug rate takes a sudden jump, it is likely that coverage will soon stagnate, or even go down as coverage gets reset. If the bug rate suddenly drops, it can be an indication that forward progress is blocked by a known bug, and will have similar consequences on coverage. This all helps management plan, and also push for design to provide quicker bug-fixes to prevent stagnation.
Benefits	Bugs are good. If they are in there, we want to find them and celebrate their capture.
Pitfalls	<p>This indicator doesn't distinguish between bugs that were easy to find and hard to find. Nor does it take into account which ones will lead to catastrophic failure vs having a minor perf hit.</p> <p>Also note that some units naturally have more bugs than others. It would not be fair to assume that because validator X has found 100 bugs, and validator Y has found 20 bugs that validator X is doing a better job than validator Y. If there are 300 bugs total in X's unit, and only 21 in Y's unit, then you can see that it is not lower productivity by Y that lead to fewer bugs being filed.</p> <p>GOALS FOR THIS INDICATOR SHOULD NEVER BE APPLIED TO A BUG FILING INDICATOR. THAT WOULD DRIVE INCORRECT</p>

	BEHAVIOR! Smoke alarms are ok, but only to drive design to keep the introduced bugs low as we get to then end of the design. Validators should ignore any smoke alarms applied to bug filing indicators, as it could encourage them to not file bugs or to stop looking altogether.
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Indicator title	Bugs and ECOs waiting to be validated
Currently used	Yes
Frequency of rollup	Weekly
Theory	Once designers have released a bug fix, or new code for an ECO, it needs to be validated. The risk of the code being changed incorrectly remains until it gets validated. As a result, timely validation is important for most bugs. Also, the longer it takes before validation is done, the harder it is to reconstruct what changes occurred, and if a bug is found, it becomes more difficult to fix.
Data it measures	Time that unvalidated bugs or ecos have existed in the released state, ready to be validated. Optionally can keep track of priority of those bugs and ecos.
How it drives individual contributor behavior	Drives timely validation of bugs and ecos.
How it drives manager behavior	
Benefits	Decreases time to validate bugs and ecos, and thereby helps to catch introduced bugs quicker, when there is less cost involved in fixing them.
Pitfalls	Some bugs/ecos are rather minor failure mechanisms, and are tough to validate. If there is high confidence that there is no PRQ-gating bug in the changed logic, then it may be appropriate to do other higher priority tasks rather than spend time on validating that bug/eco. This would make the indicator look bad, but might be the right thing to do.

Indicator title	Testplan milestones
Currently used	Yes

Frequency of rollup	Weekly
Theory	You have to get your testplans completed, and reviewed. Then when you are done with validation, exit reviews need to be held. These don't all have to be part of the same indicator.
Data measures it	Testplan Started Testplan 10% review held Testplan published Testplan reviewed Silver exit review held Silver exit review ARs completed Gold exit review held Gold exit review ARs completed.
How it drives individual contributor behavior	Keep focus on the testplan. It is after all the blueprint for successful validation.
How it drives manager behavior	If testplan writing is slipping, that can be an early warning indicator to management that lots of rtl is changing in that area, which is an indication that extra work will be needed to validate that unit/cluster once the testplan does get written..
Benefits	Keeps focus on getting testplans written and reviewed for quality.
Pitfalls	Sometimes rtl is so unstable that finding bugs is like shooting fish in a barrel. At times like this, pushing for testplan development can detract from more urgent priorities.

Indicator title	Health Of Model (HOM) and Health Of Cluster (HOC)
Currently used	Yes
Frequency of rollup	Weekly
Theory	This isn't an indicator about validation, but rather an indicator of the health of the model as seen by validation. If there are a lot of bugs, or a lot of rtl churn breaking coverage monitors, then the model isn't real healthy. If the model isn't

	<p>really healthy, then it is going to have a direct effect on validation work (usually about a 1 week lag between HOM dipping and coverage/bug filing stagnating)</p> <p>Also, HOM needs to be taken into account over several data points. If the HOM jumps really high one week, it usually means that designers went in and explicitly tried to fix things that were causing low HOM (primarily fixing open bugs) As a result, the HOM will shoot up for one week, but then drop down again as validators start onion peeling behind those fixed bugs.</p> <p>The HOM has been found to be a good predictor of when a processor will be ready for tapeout. It has been shown that when the HOM diverges too low from prior designs (when charted vs expected tapeout date), that the tapeout date is unrealistic, and will inevitably change (might as well change the tapeout date early when the HOM consistently tells you are not on track)</p> <p>As it gets closer to tapeout, it becomes harder to get good scores. The number of open bugs required to ding the score decreases. The amount of RTL churn required to ding the score decreases. Etc. Basically, this is a measure of how much validation forward progress is being prevented vs. how much is appropriate at that phase of the project.</p>
Data measures	<p>HOM and HOC are amalgamations of many pieces of data. This includes:</p> <ul style="list-style-type: none"> • Regression score (pass rate) • Forward progress (Is validation held up by rtl bugs/churn? Is validation doing a lot of pre-release testing and not able to do regular validation? Is model release latency unrealistically large?) • Debuggability • RTL stability
How it drives individual contributor behavior	Drives the design team to improve HOM/HOC so that validation can find the bugs in the design quicker.
How it drives manager behavior	Gives managers a big stick to go beat on people to improve HOM/HOC so that their employees can find bugs more quickly. That stick can also be used to tell upper management that despite the top-down goals, we are not on track to meet the tapeout date, and it needs to change before it causes people to do the wrong things.
Benefits	Removes obstacles to finding bugs. Can be expanded to other disciplines as well (One group has implemented a Health Of Timing (HOT) based on HOM in order to indicate timing convergence health.)
Pitfalls	Many people are under the mistaken impression that HOM is a measure of validation. It is really a measure of how healthy the main input to validation (the model) is, and therefore creating roadblocks validation progress.

	Validation has no legitimate mechanism to improve HOM. If pressure is applied to validation to improve the HOM, it could result in bugs not being filed (slip it into the model without calling attention to it), or curtailing of testing. A validator's true role in the HOM is to try to beat it down with a stick as hard as possible for the objective components (aggressively bug hunt, and file all bugs found) and give fair and accurate votes for the subjective ones (forward progress, rtl stability)
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Indicator title	RTL Churn
Currently used	Yes and No
Frequency of rollup	Weekly
Theory	By tracking the rate and location of rtl churn, we can get a leading indicator of where new bugs will be introduced. It is also a leading indicator of validation rework (fixing broken protocytes and coverage reset)
Data measures it	Current indicators measure the total number of lines changed in the rtl or the total number of non-comment lines changed. This really needs to be improved upon going forward. For instance, a comparison of the XLIF files could be done to find the amount of logic churn. Maybe throw some fv in there to weed out the logic that is formally identifiable as equivalent from the churn. All of this could also be used to indicate where change has occurred and give the validator a heads up for bug/eco validation or testplan updates that need to occur due to the change.
How it drives individual contributor behavior	Data to confirm what the validator has been saying about spinning their wheels due to changing rtl. Long-term rtl churn causes a huge amount of rework that can in extreme cases prevent forward progress for an extended time in an otherwise healthy unit.
How it drives manager behavior	Management needs to understand and plan for the way rtl churn affects validation. If rtl is churning away, perhaps a decision to delay writing in-depth protocytes is appropriate, and save a bundle of effort. Management also needs to push on design to keep the rtl churn to a manageable level as we get later in the process. Note that this goes explicitly against the “design wins” of using Cell Based Design (CBD). CBD is envisioned to allow more rtl change later in the project without causing huge rework to backend flows, like layout. As a result, the rtl churn indicator is going to become increasingly more important as this trend continues.
Benefits	Required to minimize un-useful work that will just be thrown out again. Also may be used as justification for slipping goals (hey, validation is extremely dependent on the model)

Pitfalls	Hard to get good data about rtl churn. Current methods include number of lines changed or a subjective vote. Not all changes to rtl are created equal, and if treated as such, can create misleading results.
----------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Indicator title	Design phase completion status
Currently used	Yes
Frequency of rollup	Weekly
Theory	This is a design indicator that is useful for validation. When Design completes coding for a phase, validation needs to exercise that code. Without this indicator, validators would have no good way of knowing what code is fair-game to exercise, and whether or not they are keeping up with design's changes.
Data it measures	Phases completed by design
How it drives individual contributor behavior	
How it drives manager behavior	
Benefits	Improves interaction with design team.
Pitfalls	

Indicator title	Age of Open Bugs
Currently used	Yes
Frequency of rollup	Weekly
Theory	When validation finds a bug, usually it creates a cone of boundary conditions that we cannot exercise, because it would hit this already known failure first. For instance, if you find a bug where a uop gets a stall at a certain place in the machine, then you are unable to exercise the case where the same uop gets the same stall and then gets a clear 1 cycle later, which could have a completely

	different bug associated with it. Therefore, the rate at which bug fixing bug fixing occurs is a good indicator to drive designers to fixing open bugs.
Data it measures	Age and severity of bugs that have not yet been turned in.
How it drives individual contributor behavior	
How it drives manager behavior	Helps managers push for design to fix those bugs, allowing us to find more bugs in the process. Symbiosis works again!
Benefits	Improves design team's attention to fixing bugs in a timely manner.
Pitfalls	<p>Age can be problematic misleading. It is possible for the design team to fix a bug and cause the average age to INCREASE in the process. For instance: Low priority bug #1 has been open for 100 days. High priority bug #2 has been open for 4 days. The average age is 52 days. If the design team fixes high priority bug #2, then it disappears from the indicator, and we are left with an average age of 100 days, almost double what we had before. That could create the incentive not to fix the high priority bug until the low-priority one is fixed, which is the wrong behavior.</p> <p>Also if validation doesn't validate a bug for a long time, and then when they do go to validate it, find a problem and fix-reject the bug, then the bug goes back into the Age category. That is a case where validation behavior has inappropriately caused this design metric to get bad. This could improperly encourage the validator to not fix-reject, and allow the designer to slip the bug fix in silently. This should not be allowed. A reasonable workaround for this case is for the validator to file a new bug instead of fix-rejecting the original one.</p>

Indicator title	Escapes to fullchip
Currently used	Not regularly
Frequency of rollup	Not individually rolled. Did a one-time analysis based on Tibet queries.
Theory	If a bug exists completely within a cluster, but gets caught at fullchip first, it indicates that the CTE or tests or checkers for that cluster are insufficient. Isolated bugs escaping to fullchip are not a cause for large concern, but trends of escapees indicates holes that really need to be plugged.

Data it measures	Bugs caught at fullchip that should have been catchable at cluster.
How it drives individual contributor behavior	Forces validators to go in and analyze why we aren't catching those bugs at cluster, and decide whether there is sufficient ROI in plugging the hole or not.
How it drives manager behavior	Bugs escaping to fullchip are an indication that more targeted attention may need to be paid to the cluster infrastructure or tests. This is often an indication that the validation team is swamped and unable to keep up with the design changes. Possible load balancing required, but it is also possible that the validation team could catch up.
Benefits	Facilitates finding bugs quicker by closing environment holes. By finding a bug before it reaches fullchip, significant cost is saved.
Pitfalls	It would be easy to assume that an escape means a mistake by the validators at cluster. This is usually not true, and should not be treated that way. Given enough time, these escapes would often have been found at cluster as well (features that haven't been fully validated yet)

Indicator title	Model build frequency
Currently used	Yes
Frequency of rollup	Weekly
Theory	When
Data it measures	Frequency of model builds
How it drives individual contributor behavior	Forces the model builder to keep an eye on keeping the frequency up to a good level because that enables more efficient validation.
How it drives manager behavior	
Benefits	Frequent model builds make everything go better. It gives designers more up-to-date models to graft from, and it gives validators more up-to-date models to exercise. It helps minimize the time wasted due to blocking bugs, and decreases the overall onion-peeling process that occurs with many bugs.

	With fewer turnins per model build, this even makes model building itself better, because there are fewer and less nasty merge conflicts.
Pitfalls	<p>If there aren't many turnins, then this indicator can look identical for both excellent and poor model build performance. For instance, if only one turnin occurred in a month, then the model build frequency will indicate one model build for the month, regardless of whether the build occurred immediately after the turnin or waited 20 days before starting.</p> <p>Also, too rapid and too slow model build frequencies can be problematic. A race to have a model build every hour for instance would leave many models unexercised, and would just be wasting resources. Having only one model build every couple of weeks is generally only ok once execution has basically completed, and no more rtl change is going on.</p>

Indicator title	Model build latency
Currently used	Yes
Frequency of rollup	Weekly
Theory	When a designer makes a turnin, we want to minimize the time before it gets into the hands of the validator. This is good for several reasons, as listed in the benefits table entry.
Data it measures	Cumulative age of all turnins currently waiting to be released, or average age.
How it drives individual contributor behavior	Brings attention to improving one of the critical loops of the design/validation cycle.
How it drives manager behavior	
Benefits	<ul style="list-style-type: none"> • Bugs can be found faster • When bugs are found faster, the designer remembers their changes and intent better, facilitating better and faster fixes • When a bug is found, it has the potential to block other testing. The turnaround time for fixing bugs directly impacts the time it takes to onion peel. • Short latency minimizes designers writing new code based on old rtl code. As a result, they are less likely to introduce new bugs due to either merging

	conflicts or incorrect assumptions (because the other code has already changed to break their assumptions)
Pitfalls	None. This is a great indicator.

Indicator title	Number of ECOs/Bug fixes/turnins per model
Currently used	Yes
Frequency of rollup	Weekly
Theory	The more ECOs/Bug fixes/turnins per model, the greater the chance of merge conflict or other introduced bugs. Also the more change, the harder it is to debug introduced bugs. For example, if only one turnin made it into a new model release and all of the sudden tests start failing, it is trivial to do a diff between the old and new code to see exactly what has changed, and therefore get a heads-up on what might be leading to the problem (even if indirect contributor)
Data it measures	Number of ECOs/Bug fixes/turnins per model
How it drives individual contributor behavior	Drives model builder to build more frequent, but smaller models (which helps effort required overall).
How it drives manager behavior	
Benefits	Brings attention to incremental model changes that results in easier validation of newly introduced bugs.
Pitfalls	In an ideal world, this is almost redundant with model-build latency, so the ROI in implementing both is low. In the real world, they do point to different problems that need resolution, but they still try to drive the same behavior.

Indicator title	Bug density
Currently used	Yes
Frequency of rollup	Usually done for post-project analysis.

Theory	In the software world, bug density has shown to be surprisingly consistent over a wide range of projects. Similar trends have been observed in the hardware world, and is good evidence for improving the design flow (possibly by moving to a higher level language)
Data it measures	Bugs per thousand lines of code
How it drives individual contributor behavior	
How it drives manager behavior	<p>Based on the historical trends for similar projects (proliferations vs lead designs, cpus vs chipsets, etc), you can project a range for how many bugs likely exist in this design. If bug filing trends do not map well into this forecast, it could turn into a big red flag for the project. At the very least, it needs to cause discussion and analysis to determine if it is ok. This should NOT be confused with a bug quota, which is counterproductive. Bug quotas cause people to file lots of insignificant bugs and not root cause any of them.</p> <p>This should also cause management to push for bug density reduction by the design team. The more bugs that exist in the design, the higher the likelihood a fatal one will escape to post-si or even production.</p>
Benefits	
Pitfalls	Bug quotas to try and force bug filing to match historical trends should be avoided.

Indicator title	Bug distribution
Currently used	Yes
Frequency of rollup	Usually done for post-project analysis.
Theory	Knowing where most bugs are found pre-silicon gives a heads up as to where bugs will escape to post-si. It also may indicate the relative complexity of units/clusters (although that is a very loose ordering)
Data it measures	How many bugs were found in each unit/cluster. How were they found.
How it drives individual contributor behavior	

How it drives manager behavior	With this data, resource balancing can be planned for before fire-fighting is required.
Benefits	
Pitfalls	Bug density isn't everything when it comes to risk. Some units will catch just about every bug that gets introduced immediately, just because any little bug will cause lots of tests to fail. A hundred bugs found in one of these units is less worrisome than ten bugs in a more complex unit.

Indicator title	Why bugs were introduced
Currently used	Yes
Frequency of rollup	One time rollup at the end of the project
Theory	If you can figure out what is causing all of those bugs to be introduced in the first place, then you can go from a reactionary "catch the bugs" to a more proactive "prevent the bugs" approach.
Data it measures	How/why a bug was introduced
How it drives individual contributor behavior	Helps validators understand the bug swarms better in order to better target their validation efforts.
How it drives manager behavior	
Benefits	<p>The analysis of why wmt bugs were introduced yielded the two largest causes of bug introduction:</p> <ol style="list-style-type: none"> 1. Implementation 2. Complexity <p>As a result, we implemented a Complexity JET and a Logic JET to attack the sources of these bugs. The intent is to slow the 3 to 4x bug growth rate for each new generation of processors, and this data allows us to use Amdahl's law to help.</p>
Pitfalls	You can't prevent all bugs from being coded, so it is important to avoid de-emphasis of bug catching.

Indicator title	10% finished and 10% review
Currently used	No
Frequency of rollup	Weekly
Theory	In order to learn a task well, you have to start somewhere. If you have to start somewhere, you might as well get some useful work out of it. As a result, we often assign a task (write a testplan, write some proto code, write some tests) expecting for learning to occur while achieving needed results. We expect for there to be problems along the way. One way that we can improve this process is to hold 10% reviews. Once a validator has completed about 10% of the task, we should hold a critical review of their work so far. This will provide feedback to show what is going well and what is going poorly so far. The validator then goes back and fixes these things, and continues the rest of the task with a firmer grasp on what is required. As a result, higher quality work is obtained with minimal extra effort (compared to finding poor trends after the task is already complete and ready for review. The rework in those cases can be extreme, and cause demoralization.)
Data it measures	10% finished and the review that follows. This can be applied to any task.
How it drives individual contributor behavior	Gives validators a chance to try things out and figure out the basics. Then a review comes and helps them to improve early in the process so that they can continue from a good solid base and create better work.
How it drives manager behavior	
Benefits	Better testplans, protocoles, tests, or anything else this is applied to. Less rework.
Pitfalls	

Indicator title	Average percentage of coverage
Currently used	No (experimented with it on wmt)
Frequency of rollup	Weekly

Theory	Coverage percent doesn't tell the whole story. For instance, if you have 2 constructs, and one has 1000 conditions, and the other only has 10 conditions, then a validator will feel pressure to push up the 1000 condition construct in order to make the coverage numbers look good. If the smaller construct has covered all 10 conditions, but the larger hasn't hit any, then 10/1010 conditions have been hit, or just under 1%. Conversely, if 1000/1000 of the large one has been hit, but none of the smaller one, then 1000/1010 conditions have been hit, or just over 99%. Average percentage takes the average of the percentages. So, in both of these extreme cases, one was at 100% and the other was at 0%, so $(100 + 0) / 2 = 50\%$ average percentage, and you can see that this is a better indicator of confidence than just coverage percent by itself. By reporting both average percent as well as overall percent, confidence is more accurately represented.
Data it measures	Average percent of coverage = the sum of all of the coverage percents divided by the total number of constructs.
How it drives individual contributor behavior	Keeps the eye on distribution more than just the biggest constructs.
How it drives manager behavior	Keeps management more in-synch on the real status, as opposed to just seeing the really big (and often less interesting) constructs dominate their view.
Benefits	Much less effort than trying to manually choose good weights for every construct. Especially in the case where the largest constructs are an order of magnitude larger than the tier 2 constructs, which are in turn an order of magnitude larger than the tier 3 constructs (which happens more often than you would think)
Pitfalls	

Indicator title	0-hit and 1-3 element holes
Currently used	No
Frequency of rollup	Weekly
Theory	We look at coverage distribution, including zero hit, and up to 3 element holes, before we give the green light for tapeout. Every one requires either filling or buy-off that we can tapeout with the hole. Why not make it part of the indicators that we throughout the project as well?

Data it measures	Number of 0-hit constructs and 1-3 element holes.
How it drives individual contributor behavior	Push individuals to pay attention to holes earlier. This allows more time for filling holes and developing a justification for why some holes don't need to be hit.
How it drives manager behavior	Gives management a concrete indication of how the distribution is going. Usually, this is indicated via disclaimers that "coverage isn't everything" and not backed up with real data like this indicator would.
Benefits	Earlier attention to distribution, and allows pressure to be put on distribution instead of just total coverage. That reallocation of pressure drives better behavior.
Pitfalls	A lot of the little holes are from little somewhat insignificant constructs (ie: lower priority). Too much emphasis on eliminating all of those holes may cause attention to be taken away from higher-priority tasks. This may also encourage larger constructs to be written to avoid small element holes. The size of a construct should depend more on that construct's characteristics than on what makes an indicator look good.

Indicator title	Template Health Indicator
Currently used	No
Frequency of rollup	Weekly
Theory	Coverage numbers and bug rates can look good, even when a lot of dangerous false failures are occurring from the templates. There are also times where too much tweak&run is being used to be aggressive. By definition, this indicator has to have subjective components. This is a HOT topic.
Data it measures	How healthy our templates are
How it drives individual contributor behavior	Applies pressure to keep templates healthy.
How it drives manager behavior	
Benefits	Effort to make healthier templates gets appropriate attention.

Pitfalls	Could pressure fixing of tests by removing functionality and randomness (very bad thing).
----------	-------------------------------------------------------------------------------------------

Indicator title	High/Med/Low breakout of coverage priority
Currently used	No
Frequency of rollup	Weekly
Theory	Not all coverage is equally important, and treating it that way leads to incorrect priorities. Breaking it into High/Med/Low allows
Data measures	For any coverage rollups, this adds the breakout of High/Med/Low priority so that questions can be answered like “How are we doing on the high priority constructs?”
How it drives individual contributor behavior	Applies pressure to get the higher priority constructs healthier faster.
How it drives manager behavior	
Benefits	Much better indicator of health than the base coverage indicator alone.
Pitfalls	A lot of effort to encode the priority for everything.

Indicator title	Complexity (rtl, design, arch, debug)
Currently used	No
Frequency of rollup	Weekly
Theory	There is necessary complexity and unnecessary complexity. We want to eliminate as much unnecessary complexity as possible, because it creates bugs, hides other bugs, makes it harder to debug, and increases risk all unnecessarily. Necessary complexity (for which there is real perf benefit) should be understood to do appropriate validation planning.
Data measures	Complexity of the design. There are no good known methods for calculating complexity. Some ideas include:

	<ul style="list-style-type: none"> • Number of lines of code • Number of logic gates • Arch: high-level interaction complexities (code breakpoint vs uncacheable code) • Design: low-level interconnect complexity (the whole fixes bugs for one unit by routing new signals to another unit) • Design: Spring model (unit complexity is increased by the connectivity with other complex units) • Debug: Temporal locality • Debug: readable code (vec-mapping is BAD)
How it drives individual contributor behavior	Helps target effort where it will be required to overcome complexity (including increased likelihood tough bugs)
How it drives manager behavior	Planning for complexity and drive initiatives to decrease unnecessary complexity.
Benefits	Drive attention to eliminate unnecessary complexity, and improves planning
Pitfalls	No good known method for calculating rtl complexity

Indicator title	Efficiency indicator
Currently used	No
Frequency of rollup	Weekly for escapes to fullchip. Once per project for escapes to silicon.
Theory	The efficiency of validation (per cluster?) would be a good thing to measure. One of our main goals is to drive bug finding earlier in the chain, and we currently have no way to measure our success at doing that.
Data it measures	Number of bug escapes per validator divided by a complexity metric?
How it drives individual contributor behavior	

How it drives manager behavior	Provides early predictor for better planning and load rebalancing.
Benefits	Provides better understanding of bug escapes vs effort expended so that better planning can be made, and more appropriate attention to infrastructure can be applied.
Pitfalls	<p>This should NEVER be used as a performance metric of individuals. There are too many factors involved to be an accurate comparator between individual contributions.</p> <p>Also, there currently is no reasonable measurement of complexity.</p>

Indicator title	Hole Distribution
Currently used	No
Frequency of rollup	Weekly
Theory	When it gets close to tapeout, there are two guiding lights that we use to tell us what needs more attention. Low coverage percentage constructs, and constructs that have poor coverage density. Coverage is an obvious indicator that has been used since the beginning of Coverage Based Validation, but hole analysis has only been applied as a gate to tapeout readiness. Part of the reason for this is that holes often hide behind higher priority low coverage constructs throughout the project, and only become noticeably divergent when the final reviews are ready to be held.
Data it measures	Number or percentage of up to N element holes.
How it drives individual contributor behavior	Forces the individual contributor to pay attention to coverage distribution throughout the project. This is currently done as part of the drive to bring up coverage, but smaller constructs can easily be left out of this analysis. Using this as an indicator and applying goals to it would decrease the number of holes that exist when it comes to tapeout time, and thus decrease overall risk.
How it drives manager behavior	
Benefits	Keeps attention on holes earlier. Encourages an automated way to keep track of holes and push for steady improvement over the long term, rather than leaving it all for the last push for tapeout acceptable coverage.
Pitfalls	Could force attention to be diverted from higher priority constructs that have better density to lower priority constructs that have worse density. This could also encourage larger more complex constructs to be written, because that would

	naturally decrease the lesser element holes by making them part of a larger space. Constructs should be coded without regard to how it would affect this indicator, and instead pay attention to the analyzability and maintenance aspects first, and initial coding issues second. Not all holes are created equal. For instance, there are some specific 1-element holes that involve assertion of the reset pin. Being that reset is a very heavy hammer (and presumably these conditions are all hit by random init), it is acceptable to tapeout without filling that 1-element hole. At the same time, there are some 5-element holes that I would require to be filled before being comfortable with tapeout. Any such information gets lost in a coverage hole indicator, and may drive wrong behavior by giving priority to less interesting conditions.
--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Indicator title	RTL elegance
Currently used	No
Frequency of rollup	Weekly
Theory	The elegance of the rtl directly affects how much time, effort, and risk involved in finding and debugging failures. For instance, CBD on psc drove the rtl down from an already low-level implementation, deeper into vec-mapped code that is human un-readable (Actually, you can read it, and figure out what each statement does individually, but when you have to track what is going on through that CBD code, by the time you follow from one state element to the next, you can no longer remember the entire path you took).
Data it measures	RTL elegance. This is very subjective, and it really just indicates the relative degradation or improvement that is noticed. (Usually it's only degradation, because improvement would cost extra effort that is not funded)
How it drives individual contributor behavior	Applies pressure to rtl designers to write more elegant code. Also applies pressure to tool designers to consider future maintenance and validation of the golden model when creating and deploying tools.
How it drives manager behavior	
Benefits	Better code. Less effort. Less risk. Higher productivity.
Pitfalls	Very subjective measurement.

7 Summary

Indicators serve three main purposes:

- Provide an accurate measure of progress for immediate management consumption. This includes everything from project planning to focal ranking.
- Drive correct behavior of individual contributors by forcing them to present progress toward their commitments in a public and accountable manner.
- Generate statistical data that may be useful for future consumption (planning, analysis, etc).

For the most part, all decisions about what indicators to use and how they should be used is dictated by the above three purposes. There is a cost involved, and it is always an ROI tradeoff.

There are pitfalls with **EVERY** indicator, and those pitfalls should be actively understood so that we don't fall into the trap of thinking that we are done just because the indicator says we are done.

You should never stop doing the “right thing” in order to improve an indicator. You should change the indicator to more accurately indicate the “right thing” or accept the poor indicator performance.

8 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 19

Evil Validation

By: [Michael S. Bair](#) and [Bob Grim](#)

1 Abstract

Evil Validation indicates an extraordinary level of validation beyond the standard validation practices set by the project. Evil Validation needs to be used sparingly only in appropriate situations. Evil validation is not necessary for a successful project, but can be used to reduce risk to ensure a project is successful.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/02/2003	First release, after L2 review	Michael Bair	Matt Kupperman, DPG-N uAV MWG
1.1	1/7/2004	Second release after 1 full round of L3 review.	Michael Bair	Paul Schwabe
1.2	7/31/2005	Small edits plus addition of purpose, background concepts, and summary.	Michael Bair	
2.0	1/20/2016	Initial updates for integration validation prior to broad experienced Validator review	Bob Grim	Michael Bair

3 Contents

1 Abstract.....	455
2 Revision History.....	455
3 Contents.....	456
4 Purpose.....	457
5 Background Concepts	457
6 Evil Validation.....	457
6.1 What is Evil Validation?	457
6.1.1 Examples of Evil Validation.....	457
6.1.2 Evil Validation is Not.....	460
6.2 When and Where to be Evil	461
6.2.1 First, Don't be Evil.....	462
6.2.2 When/Where is it Acceptable to be Evil?	462
6.2.3 Closing the Loop.....	462
6.3 Issues with Evil Validation.....	463
6.3.1 Hard to Measure Success.....	463
6.3.2 It is Tough to Judge When to be Evil	463
6.4 Is Evil Validation Required?	463
7 Summary.....	463
8 Future Work.....	463
9 References.....	464

4 Purpose

There is much talk within validation teams about being evil. Managers ask for it, Designers beg for it, and teams throw the term around as a hot buzzword. Like much of validation, evil is not taught in college. New Validators are not familiar with what it means to be evil. This chapter attempts to classify evil validation and to give examples of how and where to be evil.

5 Background Concepts

Evil Validation is an advanced concept and it is important for the reader to have a firm grasp of many validation concepts and methodologies described within other chapters of this book. [The Validation Mindset](#) describes traits of a good Validator. These traits are the basis of a good *evil* Validator. The three pillars of validation ([Stimulus](#), [Checking](#) and [Coverage](#)) consist of methodologies that have evolved over time and combine to define a high level of validation quality. An evil Validator must have an understanding of these methodologies when deviating from the standards set by the project in question. Understanding multiple [Validation Disciplines](#) will give insight into the strengths and limitations of each type of validation. Finally, expertise is key for the evil Validator to know when and where to apply evil validation (see [Becoming the Microarchitecture Expert](#) or related chapters).

6 Evil Validation

6.1 What is Evil Validation?

The term *evil validation* is an ambiguous and sometimes overused term that can mean many things to different people. For the purposes of this chapter, we will define evil validation as “an extraordinary level of validation.” Over the course of many projects, DDG Pre-Silicon validation has defined a standard level of validation that results in a high quality processor design being sent to the fab. Validation tasks that exceed these standards are expected to have diminishing returns and should be considered unnecessary under most circumstances. As this implies, evil validation lives outside of this standard level of validation. “Evil” doesn’t mean that it is necessarily bad, but should be used with caution.

6.1.1 Examples of Evil Validation

Since this definition is still very ambiguous, a few examples are necessary to try to clarify.

6.1.1.1 Validating to an Old Standard of Validation

The standard level of validation mentioned in the definition of evil validation is constantly being reevaluated, and therefore the line between what is considered standard and evil validation can shift over time. As confusing as this may be, such a change could result in a validation task being categorized differently over the course of many projects. A good example of this happening is

with the level of coverage described in a validation testplan. At one time, it was standard validation practice to create a testplan that included deep, multi-dimensional coverage conditions. A later project evaluated the efficiency of this practice and changed the standard to only include simple breadth coverage and found that it did not negatively affect the overall quality of validation. From that point on, the creation of deep coverage conditions could be considered *evil validation*.

6.1.1.2 Testing Outside the Specification

It is common validation practice to push the design up to (and sometimes a little beyond) its breaking point. This is necessary to hit certain boundary conditions in a reasonable amount of time or to uncover some deeper bugs. The successful Validator will not limit testing to only the expected usage model, but will use any means practical to accomplish this. At the IP level, it is relatively easy to expand testing beyond the specification because the test environment is designed to allow a superset of the traffic expected at the integration levels. As validation progresses further up the integration continuum, it becomes more difficult to generate such stimulus.

Expanding the scope of testing does not come without cost. Going beyond what the Architect and Designer intended the design to handle will find bugs. However, if the testing is too far outside of what is expected, the burden of proving the scenario can happen under normal operation will fall to the Validator, otherwise the Designer could disposition it as “Not a Bug.”

6.1.1.3 Negative Space and Error Testing

Normally included in the specification for a design is a description of how a design will react when something bad happens. This includes things like accesses to undefined memory regions, security violations and error handling. These are all features and should be tested to ensure they work as defined. However, many times encountering these conditions will result in the design entering an unrecoverable state and ending the test. For this reason, these conditions are often tested in isolation. The evil Validator may try to add the testing of these conditions to random tests and crossing them with other traffic to be more efficient. Such testing must be ready to handle unrecoverable states gracefully to really be efficient.

6.1.1.4 Forcing the DUT to do things it rarely does

During the course of a project, a Validator may find that there are certain conditions that are hard to hit due to the structural design of the relevant logic. It might be difficult to fill a queue, use the last buffer in a set, or to stress ways in a cache. The evil Validator would see this as a chance to create an injector to force the RTL into a state that makes rare events more common. Some useful injectors used in past projects are:

- Forcing a cache LRU to always pick the same victim
- Forcing a subset of buffers to be available
- Fill a queue by blocking the output
- Forcing a counter to overflow
- Override arbitration priorities to give more bandwidth to different traffic
- Target internal state with external traffic generators

Again, this is a solution that must be used with some amount of caution. Using an injector to put the DUT into an artificial state may flush out bugs that would be impossible to hit under normal operation. It is up to the Validator to make the case that it is a real bug.

6.1.1.5 Checking at Lower Levels

One way to attack a feature is the use of low-level checkers that watch for unexpected low-level logic aberrations that may never be visible by a higher-level checker. Checking closer to the logic where a bug might occur relieves the responsibility of the test to not only hit the bug but to cause the bug to become visible.

On a recent project, a data checker was added to an intermediate point of the PCIe data path. This uncovered a latent bug where illegal data packets were observed while exiting a low power state. The rogue packets never made it to an externally observable point, and therefore did not cause a fatal error, but the behavior was still undesirable. This bug may have never been caught if it was not for the lower level checker.

See [Checking section: The Value of Implementation Rules](#) for an in-depth discussion on lower-level checks.

6.1.1.6 Speculating on ‘What Might Happen’

Late in the project, after all tests have been written, sufficient coverage has been gathered, and tests are passing with relatively few failures, it may be appropriate to spend a little time thinking about interesting ways of breaking the DUT. We will call this What-If-Analysis.

With What-If-Analysis, you concentrate on a specific piece of hardware and figure out what input might cause the hardware to break. You might try to relate seemingly unrelated things: pieces of logic, actions, features, or any combination of these. This effort ranges from being easy to being extremely difficult depending on the logic you are looking at and your expertise with the hardware. The next step is to conjecture on how the surrounding hardware could set up the input to cause the failing case. This is quite difficult, because it quickly leads out of the hardware you know and into hardware you do not know. At this point, you investigate the ideas by looking at RTL, talking to other Validators, and talking to microarchitecture experts.

How does an *evil validator* identify areas to target with this What-If-Analysis? One possibility is through the course of normal validation activities. Imagine, while debugging a failure, a validator finds some logic that appears to be working, but does not seem quite right. The validator may speculate that the logic in question is not robust. If it was stressed it in a particular way, it might fail. Taking note of this troubling logic is not itself being evil. In fact, all validators should exhibit this behavior, as it is part of the validation mindset. However, acting on this uneasy feeling may be evil if it includes spending an extraordinary amount of effort to write new directed tests, checkers, or injectors that target only at this piece of logic.

Again, What-If-Analysis can be slow and take away from other validation work. It is important to understand when an area requires this kind of evil. It is good to do the work, without knowing a bug will result, especially in buggy algorithms, poorly understood protocols, or areas where a late bug in a feature could require a large microarchitecture change that would be detrimental to the

project. It is the right choice to take the time away from other validation activities to pursue bugs in these areas.

6.1.2 Evil Validation is Not...

The previous section attempted to define what evil validation is. The most unsatisfying part of the definition is that the line between *standard* and *evil validation* is always moving and therefore any concrete example created today may not be valid in the future. Fortunately the converse is a little easier to define. There are many things described as evil or used to describe evil that were not exactly evil. I have heard many times that “being evil is being creative.” Well, not necessarily. Creativity helps as in all design activities, but like all engineering activities, a good share of evil work can be brute force. The following is my list of favorite evil-wannabes.

6.1.2.1 Getting Your Validation Tasks Done First

Finishing your validation tasks quickly is admirable and good for the project (your compute and human cycles can be used elsewhere!), but should not be your guiding priority. A focus on doing quality validation that is reproducible on future steppings and proliferations is more important.

6.1.2.2 Using the Kitchen Sink

A common misconception is that the more you add to a test the more evil it is. This is true, but adding too many features can dilute other features. Making any feature too random might keep the test from hitting anything interesting. The simplest example of this is sending loads and stores to random addresses. If the addresses are random, as in anything between 0 and 0xFFFFFFFF, it is unlikely the store and load will interact at all. If they are random, as in one is random and the other is a clone of the first that is munged using some microarchitecture knowledge, it is likely they will interact. Being evil normally implies some amount of focus on an area of concern.

6.1.2.3 Just Adding More ...

Getting evil does not mean just adding more of some validation task, be that coverage, tests or checking. Doing so is necessary when your test plan is not strong enough in an area, but this is hole filling, not getting evil.

6.1.2.4 A Replacement for other Validation Duties

Getting evil does not replace regular validation activities, such as keeping your testplan up to date, running tests, and analyzing coverage. Your normal activities give the highest confidence of a high-quality tapeout. Getting evil in one area at the expense of the quality of validation in another is not acceptable. Such actions will likely push-out production. Evil work not tied directly to a regular validation target (like writing injectors to help drive coverage in an area) may find some hard to hit bugs that would otherwise make it to silicon, but if it allows a blocking bug to make it

into silicon, it would be the wrong choice to make. The only bugs we want in post-silicon are obscure bugs. The simplest post-silicon tests are very exhaustive, a non-obscure bug is devastating to their ability to get any testing done – and thus requires a quick stepping turn before any real bug onion peeling has occurred.

6.1.2.5 The Same as “Get Healthy” or “Paranoia”

Historically, a “get healthy” effort happens in a phase of a project where the design is so broken that even the simplest of tests cannot run without failing. At this point, all validation activities stop in order to bring the health of the design back up to an acceptable level. Evil validation is something that can only be done on a healthy design.

“Paranoia” is a period prior to tapein where a focused effort is made to identify validation tasks that had not been sufficiently addressed during earlier phases of the project. In short, it is a time to dot all the i’s and cross all the t’s. This may be a time to think about evil validation, but only if all other validation tasks have been covered.

6.1.2.6 Equally Effective Everywhere

The acts of getting evil are not equally effective across the board. If there are areas of the chip that are well understood, well tested, and not undergoing change, there might be higher ROI to have the validation owner for this area help out other areas instead of attempting to squeeze every last bug out of the area. This is not, however, an advertisement to management to put the stop on these types of evil activities.

6.1.2.7 A Job for One Person

Project leadership has at times considered having one or two Validators being “freelance” evil Validators. These Validators would not be tied to one area – their charter instead would be to find areas to be evil and to go after them. This is a nice idea, but it does not work as well in practice. It is not as useful to have one Validator attempting to be 100% evil while the rest of the group concentrates on normal validation. It is more effective to have all Validators be 5% evil.

Why is this true? The majority of the time evil activities stem from normal validation activities. For instance, when trying to hit coverage conditions, a Validator will identify conditions that rarely occur in RTL, and may decide to focus special attention in that area. Another reason is that part of being evil is to become an expert in a particular area. If you only have a few people being evil, even 100% of the time, there is no way they will get as much “evil coverage” across the chip as having all the Validators become somewhat evil in their own areas.

6.2 When and Where to be Evil

Now that evil validation has been defined (both what it is and is not), the budding evil Validator will want to know where to start being evil. This is where things get a little tricky.

6.2.1 Don't be Evil at First

Earlier, this chapter defined evil validation as going beyond the standard level of high quality validation. Being evil should never be the first tool chosen out of a Validator's toolbox. By giving the standard validation practices a chance to work, it is likely that evil validation will not be needed.

6.2.2 When/Where is it Acceptable to be Evil?

There will be the rare occasion where adhering only to standard validation practices is not sufficient, and it may warrant more extreme measures. Evil validation can be used as a way to reduce risk when the project needs it. Keep in mind that none of these situations signifies that it is open season for a Validator to pull out all the stops, but some targeted evil may help. Also note that this is not an exhaustive list.

6.2.2.1 New Features

Just because a feature is new does not immediately make it a candidate for evil validation. Standard validation practices should be feature agnostic and thorough enough to handle new situations. However, there may always be a counter example at some point that requires a little evil validation to help it through the validation process the first time.

6.2.2.2 Escapes

A bug escape is an excellent area to analyze to see if standard validation techniques are still sufficient for high quality validation. The Validator is tasked to ensure that the escape is not due to a simple validation hole before employing evil techniques.

6.2.2.3 Late Changes/Bugs

When a feature or area has bugs found late in the project, there is implicit risk that any change may introduce more instability to the system. Using evil validation may be necessary to give confidence that the required changes are good.

6.2.3 Closing the Loop

The use of evil validation is an indication that standard validation practices may not be sufficient for high quality validation. It is up to the Validator to evaluate the situation and determine how to improve standard validation practices such that evil validation will not be needed in the future. This may even include making the evil practice part of standard validation.

6.3 Issues with Evil Validation

If you have not figured it out already, be warned that the whole concept of evil validation has some major unsolved issues. If you have any ideas, let them be heard!

6.3.1 Hard to Measure Success

Success of evil validation is incredibly hard to measure because of many factors involved. The big picture always needs to be looked at. Yes, you might have validated the heck out of a state machine, but if the rest of the unit is a bug farm in post silicon, it might not have been the right thing to do. Therefore, ROI needs to be addressed. Another factor is the physical outcome of the evilness. Was testing made better? Were bugs found? Was coverage hit? Are Designers more confident in an area? Are there checkers and injectors that can be used as testing collateral in case this area changes in the future? All of these outcomes are examples of successful evilness.

6.3.2 It is Tough to Judge When to be Evil

The decision of when it is appropriate to use evil validation relies on the Validator's judgment. The Validator must weigh the relative risks and rewards of using evil validation on a narrow area of concern versus using the same effort to improve broad validation quality. Either path may be "correct" from a project standpoint, but the decision should not be made in a vacuum. Input from the stakeholders involved needs to be factored in before making a final decision.

6.4 Is Evil Validation Required?

At this point I hope it has been made clear that the standard validation practices used in DDG Pre-Silicon Validation are intended to be sufficient to ensure high quality validation for a project. With that, it can be said that evil validation is not a requirement for a successful project. It also needs to be made clear that the ability to use evil validation, only where needed, is required to reduce risk in some cases.

7 Summary

Evil validation is an excellent tool for expert Validators to have at their disposal, but must be deployed only when required by project needs. A feature using evil validation must strive to return to using standard validation practices as soon as possible. This can be accomplished either by bolstering the standard practice that was lacking or by making the evil validation technique a part of standard validation. Evil validation is not necessary for a successful project, but the ability to use evil validation might be.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 20

IP Validation

By: [Michael Bair and Lance Geiger](#)

1 Abstract

IP Validation is validation of RTL at the IP level, which is typically the smallest DUT where a simulation model is created and validation occurs. Since this is the first level of validation that all (or most) of the RTL code will experience, IP Validation often finds the majority of the bugs within that code. When the RTL code is sent to higher levels of integration (and validation), it is expected that most of the bugs have already been found and fixed.

IP Validation uses many validation disciplines, such as Dynamic Validation, Formal Verification, Mixed Signal Validation, Design for Test Validation, and many more depending on what types of features exist within the IP.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	7/4/2016	Initial Draft	Michael Bair	Lance Geiger, Phil Atkinson
1.1	8/19/2016	Integrated feedback	Lance Geiger	Michael Bair

3 Contents

1 Abstract.....	465
2 Chapter Revision History	465
3 Contents.....	466
4 Purpose.....	467
4.1 Why do we need this chapter?.....	467
4.2 What does this chapter cover?	467
4.3 What does this chapter not cover?	467
5 Background Concepts.....	467
5.1 IP	467
5.2 Intel Reuse Repository (IRR)	468
5.3 Integration Unit (IU)	468
6 IP Validation.....	468
6.1 What to target with IP Validation.....	468
6.2 Picking the Right Disciplines.....	469
6.3 Interface Modeling	470
6.4 Collateral Reuse	471
6.5 Be Prepared to Help Integration	472
6.6 First Line of Defense.....	472
7 Summary.....	472
8 Future Work.....	472
9 References.....	473

4 Purpose

4.1 Why do we need this chapter?

This chapter gives an overview of IP Validation and sets the stage for discussion of all of the other validation disciplines that are used in IP Validation.

4.2 What does this chapter cover?

This chapter covers a high level discussion of what IP Validation is, why it is important, and what validation disciplines it utilizes.

4.3 What does this chapter not cover?

This chapter does not cover details of the different validation disciplines.

5 Background Concepts

5.1 IP

The acronym ‘IP’ stands for “Intellectual Property” but in our context, it generally refers to a block designed and packaged for use in multiple products. IPs are used as building blocks to create larger and more complex designs and help avoid “reinventing the wheel” on every product that Intel designs.

IPs can vary in size from something very small like a FIFO to something very large like an entire IA core. What distinguishes an “IP” from any other block of logic is that IPs are designed and validated in a standalone environment and then packaged with additional collateral (such as documentation and configuration sequences) to enable one or more product teams to easily take the IP and “integrate” it into their product.

IPs are typically designed to be modular by utilizing standardized interfaces and methodologies. IPs may also have parameters than can be tweaked to customize the IP for a particular purpose. Such IPs are known as “soft” IPs as the product team must still synthesize the IP and perform the related backend activities. On the other hand, “hard” IPs are delivered with corresponding backend collateral, so the product team does not need to synthesize the IP, but also loses the flexibility to tweak parameters.

5.2 Intel Reuse Repository (IRR)

IRR³⁹ is an Intel repository used to store both Hard and Soft IPs developed both internally and externally. IRR provides mechanisms for IP developers to upload their IPs to the repo, enforce access mechanisms and for projects to download IPs for integration into their products.

5.3 Integration Unit (IU)

Integration Units or “IUs” are similar to IPs but are developed by the project team as opposed to a separate IP development organization. IUs are typically only delivered to a single project and thus can avoid some of the overhead of IP packaging and uploading to IRR. IU Design and Validation teams work closely with the rest of the product team to ensure a quick and smooth integration process, as well as responding to late feature requests or bug fixes. For the remainder of this chapter, the terms “IP” and “IU” are used interchangeably as the same general validation strategies apply to each.

6 IP Validation

IP Validation is typically the first validation seen by the RTL code contained in the IP. IP Validation will find the vast majority of bugs within the IP, and clear the way for successful integration to higher levels.

6.1 What to target with IP Validation

The cost of finding and fixing a bug increases over time on a project, as discussed in [Introduction to Pre-Silicon Validation section: Validation Cost Models](#). Moreover, the cost of finding bugs in successively higher levels of integration is also more expensive, for the following reasons:

- Debug is more complex. Higher levels of integration mean that operations cross more boundaries and debuggers may need to follow operations across a much bigger design to narrow down the problem. As debug crosses more boundaries, it may also require bringing in more expertise, which also slows the process and creates opportunities for things to fall through the cracks
- Tools and systems are slower. Getting debug traces, tracing through the design, and trying ‘quick fixes’ are slower in higher levels of integration because the design is bigger and it takes longer to compile/execute and load into debuggers.
- Fixes take longer. At higher levels of integration the fixes for bugs may take longer to create (especially when it requires architecting a new protocol on an interface) and longer to get to the validation team (especially if the IP team fixes the bug and requires a full drop for the SOC team to get the fix – full drops can take significant time). In some cases, there may not be any additional IP drops planned and you may have to identify a permanent workaround for a bug.

³⁹ <https://irrprod.cps.intel.com/irr/>

- Finding bugs at Integration/fullchip adds project schedule risk. Not only does code reach those higher integration levels later, but often the test suites come up later too, often due to other problems in integration. Imagine not testing basic power flows at the IP level. If you wait to do that at the SOC level, then that IP RTL code will not be exercised until the code reaches SOC level *and* the SOC level has reached a high-enough level of health (RTL and Test Environment) to have tests execute far enough to stress your IP. That could be many months, *even quarters*, after the RTL code was first delivered at the IP level.

In addition, you want to find bugs as quickly as possible after they are introduced into a design. The longer they linger prior to being found, the harder it is for the designer to remember everything about that particular piece of logic. Longer lag times create costlier and riskier rework.

Thus, IP Validation must strive to find the vast majority of bugs within a design. It is far too expensive (and risky to the project schedule) to have bugs caught at higher levels of integration. Moreover, Validation should not put itself in the situation where it is relying on validation at higher levels of integration to find IP bugs. Not only is it more costly to find bugs at higher levels, but bugs are typically harder to expose at higher levels where stimulus is often less stressful and cycles are more expensive. That being said, some features or flows may be incredibly expensive to validate at the IP level (due to needing to mimic neighboring units or interactions) and may be more efficient to validate at a higher level of integration. In general, this should be the exception to the rule.

To meet these demands, IP Validation must take the goal to try validating *everything*. If IP Validation is utilizing Formal Verification, then it must cover all of the functionality of all of the logic and really push assumptions out to the interfaces (and create assumption checkers that can be run in integration simulations). If IP Validation is utilizing Dynamic Microarchitecture Validation, then random tests must cover as much of the space as possible (all cross products of features and events must be possible) and checking must be set up to catch all types of bugs. If IP Validation is using a mix of disciplines, they must overlap in such a way that bugs do not fall through the cracks between them.

Bottom line: IP Validation should strive to *never* push Validation of any mainline feature to higher levels of integration.

6.2 Picking the Right Disciplines

[Validation Disciplines section](#): [Validation Disciplines](#) provides an overview of the various validation disciplines. Each of these disciplines has unique tradeoffs with respect to IP Validation and an all-encompassing IP Validation strategy will make use of most or all of the disciplines to varying degrees.

[Dynamic Microarchitecture Validation](#) is the often the primary discipline utilized by IPs. Dynamic Microarchitecture Validation is extremely versatile, can be applied to just about any area or feature and can scale from very small to very large designs. Its primary drawback is that significant effort must be spent developing and maintaining a test bench around the RTL design. It also requires significant compute capacity to effectively run random tests and drive coverage to sufficient levels. This discipline is also very reliant on Validators identifying and implementing the stimulus necessary to trigger bugs. While Dynamic Microarchitecture Validation may be capable of finding

nearly every bug in the design given enough test bench/stimulus development and compute cycles, it is not always the most efficient method to do so. For this reason, Dynamic Microarchitecture Validation is rarely sufficient on its own and must be complemented with other disciplines to efficiently validate an IP.

[Formal Property Verification](#) addresses many of the shortcomings of Dynamic Microarchitecture Validation by using mathematics to formally prove aspects of the RTL design. FPV requires significantly less test bench enabling, making it an ideal tool for design exercise (especially for Designers). FPV also applies all possible inputs (except where explicit assumptions say otherwise) so it can identify corner cases that may have never been considered when using the Dynamic Microarchitecture Validation approach. The primary downside to FPV is scalability. FPV is ideal for smaller blocks of logic with standard interfaces and a single clock domain. FPV can definitely be used on larger and more complex designs (even entire IPs) but some tweaks may be required to make performance acceptable, such as artificially shrinking queue sizes or adding input assumptions to simplify the proofs. While some IPs use FPV as their primary validation discipline, typically some level of Dynamic Microarchitecture Validation is still required for tasks such as power-aware validation and to provide simulation collateral reused at higher levels of integration.

Some validation disciplines are applied whenever your IP has particular features. For example, IPs that have analog logic will need to apply [Mixed Signal Validation](#) to those areas. IPs with microcode or firmware will need to apply [Microcode Validation](#) or [Embedded FW Validation](#).

Other disciplines utilize the same techniques as Dynamic Microarchitecture Validation, but are classified separately as they require dedicated focus and are often more critical to the integration process than the internal functional features of the IP. These disciplines include [Reset and Power Management Validation](#), [Design for Test Validation](#), [Design for Debug Validation](#) and [Security Validation](#).

Finally, [Integration Validation](#) is an important discipline, even at the IP level. IPs often must integrate multiple “subIPs” such as IOSF-SB endpoints and TAP controllers and many of the Integration Validation techniques apply to IPs just as much as higher levels. It is also critical that IP Validators understand the product integration process and the issues that can cause problems and delays at the higher integration levels.

6.3 Interface Modeling

When modeling the interfaces to an IP within a test environment, there is natural tension between wanting to model the interface accurately versus wanting the input stimulus on the interface to go ‘far beyond’ what the actual interface is capable of sending. There are good reasons for both directions; when architecting an IP test environment serious thought needs to go into this.

Modeling an interface accurately has multiple *benefits*:

- Reduction in overall validation space – the Test Environment will not try to do things the real interface would not do. The validation space is thus limited to things that can actually happen once the IP is integrated.
 - This lowers the effort of the Validation team
 - This reduces ‘false failures’ – failures where the RTL breaks because it receives an input sequence that the real machine would never do

- Accurate modeling allows validation at the IP level to find real SOC-level bugs, such as architectural deadlock bugs or performance bugs
 - This happens very rarely though

Modeling an interface accurately has the following *drawbacks*:

- If the interface spec is not thoroughly understood, the IP Test Environment's modeling of the interface may not create all of the boundary conditions that the real interface will create, and thus important conditions may not be tested at the IP level.
 - This has been a large portion of integration bugs in the past
 - As interfaces become more standardized (and straightforward), this problem is lessening

Modeling an interface to go well beyond the spec has the following *benefits*:

- The stimulus created on that interface will be much more stressful than the ‘real world’, and thus IP Validation will likely find bugs sooner and hit its overall coverage goals sooner than with less stressful stimulus.
- Something that is not in the spec today may be in the spec tomorrow. By validating outside of the POR range, you can make it easier to quickly validate the inevitable spec changes that come later in the project. Keeping an open mind can also prevent Validators from baking in assumptions too deeply into the testbench, which may become extremely expensive to change later on.

Both Dynamic Microarchitecture Validation and Formal Verification require modeling of interfaces. When doing this modeling, take into account the pluses and minuses of both directions, and see what works for your IP team. It might be a combination – follow the spec, but make sure that the test environment can do every odd variation in the spec (i.e., have the greatest amount of outstanding transactions, let things go out of order, vary timing, etc.).

6.4 Collateral Reuse

Delivery of an IP is not just about the RTL. The verification collateral is also an essential part of the package that helps a product team integrate, use and validate the IP at the higher level. It is important to understand the product team’s needs and validation strategy in order to strike the right balance of validation collateral to reuse. Providing too little collateral can force the integration team to reimplement significant amounts of test bench code or sequences that you already implemented at IP level. Too little collateral can also make debug of your IP difficult at higher levels. On the other hand, providing too much collateral can complicate and delay the integration process and bloat the higher level simulation model with checkers and monitors that provide little ROI to the product team.

It is important to understand how the integration team plans to validate your IP at the higher level and what types of collateral they expect and can most effectively leverage. It is also important to properly document in the integration guide what collateral is included as part of the IP drop and how to integrate it at the higher level. IP Validators should also seek out feedback after their IP is integrated to better understand what issues were encountered and what collateral provided value to the product team.

6.5 Be Prepared to Help Integration

An IP provides no value to the company on its own. The value of an IP is realized when a product team is able to successfully integrate and leverage that IP in a product, saving the time and effort it would have taken the product team to develop that functionality on their own. An IP that is difficult and time consuming to integrate into a product negates much of the value that the functionality and features inside that IP provide.

For these reasons, it is important for IP Validators to be engaged in both the integration process and higher level validation of their IPs. IP Validators should work with the integration team to understand issues and inefficiencies encountered while integrating the IP. These learnings should be incorporated into improved documentation or validation collateral. In addition, IP Validators should review testplans and validation strategies at the higher levels to understand how their IP is being validated and provide feedback. IP Validators should also be proactive in asking for feedback on how validation collateral could be improved or parameterized to allow easier customization for each product.

6.6 First Line of Defense

IP Validation is likely the first team to see an issue with RTL code quality, with schedule, with risk, etc. The team must raise the appropriate project flags if there is an issue. If the IP or IP Validation is late, Integration Validation and FC Feature Val (and RPMV, etc.) will be badly impacted.

7 Summary

IP Validation is the starting point for validation of RTL code. Its job is to find the vast majority of bugs within the IP, and it really strives to find all of them that are possible to find within an IP test environment.

IP Validation uses a number of different disciplines to complete validation of the IP, but then goes beyond this task to also create collateral that is reusable at higher levels of integration, as well as helping out those integration teams.

8 Future Work

Future versions of this chapter should address these topics:

- Packaging an IP for delivery to multiple products
- Parameterizing design and testbench for modularity
- Strategy for validating different product configurations

9 References

This section intentionally left blank.

Dynamic Microarchitecture Validation

By: [Phil Atkinson](#)

1 Abstract

This document describes the role of microarchitecture validation (uAV) within the design project and within pre-silicon validation. uAV validates the microarchitecture or the specific implementation of a processor. uAV uses a white box testing approach. uAV validation occurs at multiple levels within the design ranging from low level cluster uAV to uAV at the super cluster level. Cluster uAV operates on the smallest DUTs, emulating surrounding logic and providing controllability and speed. Supercluster uAV checks the assumptions of cluster uAV and tests cross-cluster protocols. The uAV team has an organization that is similar to the Design team but does not report to the Design team. Interactions with other teams are described as well as the tasks involved in the various levels of uAV.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/31/2004	First release	Phil Atkinson	Matt Kupperman / DPG-N uAV MWG
1.1	4/22/2004	Updates from Level 1 review	Phil Atkinson	Bob Fisch
1.2	6/22/2004	Reworked based on level 2 review	Phil Atkinson	Bob Fisch
1.3	7/6/2004	L3 edits	Phil Atkinson	Paul Schwabe
1.4	3/8/2011	Bringing up to date	Michael Bair	Phil Atkinson
1.5	5/10/2012	Updates to integrate in the Art of Validation	Phil Atkinson	

3 Contents

1 Abstract.....	475
2 Revision History.....	476
3 Contents.....	477
4 Purpose.....	479
4.1 Why do we need this chapter?.....	479
4.2 What does this chapter cover?	479
4.3 What does this chapter not cover?	479
5 Background Concepts.....	479
6 Introduction to Microarchitecture Validation.....	479
6.1 Why uAV?.....	479
6.2 uAV goals	480
6.3 Approach - black box vs. white box	480
6.4 Scope	481
6.4.1 Cluster uAV.....	481
6.4.2 High level DUTs.....	482
6.5 Organization	483
6.5.1 As Design goes, so goes uAV	483
6.5.2 Team size	483
6.6 Interactions	483
6.6.1 Design.....	483
6.6.2 Architecture.....	484
6.6.3 Other Validation Teams	484
6.6.3.1 Architecture Validation.....	484
6.6.3.2 Feature Architecture Validation	484
6.6.3.3 Power Management Validation.....	484
6.6.3.4 Formal Verification.....	485
6.6.3.5 Mixed-Signal Validation	485
6.6.3.6 Design-For-Test Validation.....	485
6.7 uAV tasks.....	485
7 Summary.....	486
8 Future Work.....	486

9 References..... 486

4 Purpose

4.1 Why do we need this chapter?

uAV is an important discipline within Pre-Silicon Validation. In past projects uAV is the first line of bug-finding after Design releases code to Validation. In this role, uAV typically finds the largest quantity of bugs within the Validation team, but often finds itself responsible for the Validation of nearly all of the CPU.

4.2 What does this chapter cover?

This chapter explains the nuances of the uAV team, including its scope, general methodology, tasks, goals, organization, and interactions with other teams.

4.3 What does this chapter not cover?

This chapter does not cover the methodologies of uAV in any depth, nor does it cover the tools used by uAV. Much of this will be covered within other chapters of this book.

5 Background Concepts

6 Introduction to Microarchitecture Validation

6.1 Why uAV?

Within pre-silicon validation, many teams' efforts combine to validate the processor. The role of Microarchitecture Validation (uAV) is to validate the processor by looking at the low microarchitecture details. An IA-32 processor is like a car. The car has a set of things it must do to be a car – drive forward, backwards, turn, brake, etc. The architecture is like the body, shape, and look of the car. The microarchitecture is like the engine and various systems the driver does not see. In the IA-32 world, changing the microarchitecture is like keeping the body of the car but installing a new engine. The IA-32 architecture originated with the 8088 and 8086 and has evolved over time (including going from 16 to 32 to 64 bits support) by adding new modes and new instructions. The instruction set, data types, and modes define the architecture and must be backwards compatible with previous processors. The microarchitecture is the specific implementation of the instruction set, data types, and modes for a given processor.

uAV is essential because bugs exist not only in the architecture but also in the microarchitecture. If you ignore the microarchitecture, you will miss critical bugs. Consider the analogy of validating a car only using architecture knowledge. One test is to make sure the car can go from zero to maximum speed and not break. This testing must be done and will find bugs. However, what if there is a supercharger in the engine that turns on above a certain speed? With this microarchitecture knowledge, you would test the boundaries of the supercharger. Testing would

include when the supercharger turns on and off, how long it stays on and off, and at what temperatures it operates. If the supercharger breaks, the car breaks. It is imperative to do both types of validation or risk the design not working.

6.2 uAV goals

The first goal of uAV is to enable the Design team by finding as many RTL bugs as practical as soon as possible in the design process. Because we work at the implementation level, we are the first validation team to test new RTL code. The Design team relies on uAV early in the project to weed out bugs and later to give confidence there are no bugs in an area. Both types of feedback enable design to continue coding new RTL without rework due to major bugs in the design.

Another goal of uAV is to provide confidence in the design in order to make a decision to tape-out the processor. If it is not tested, it is broken. We employ a variety of tools, indicators, and methodologies to provide confidence in enabling post-silicon debug on first silicon.

Our goals require us to be microarchitecture experts. The expectation is that each Validator is a resource for any questions pertaining to their unit or cluster. This knowledge is especially important in debugging and validating bugs found in post-silicon. Designers, Architects, and other validation teams leverage our knowledge of the microarchitecture. Designers spend much of their time doing non-RTL related activities. Designers also get moved from unit to unit or cluster-to-cluster to address critical paths on the design flow. uAV Validators remain experts in the logic they test. Often, they are the only experts for an area. uAV Validators are also involved in evaluating bug fixes and evaluating the cost of making design changes.

6.3 Approach - black box vs. white box

Each presilicon validation team approaches the problem of finding bugs and providing confidence in the design in different ways. By attacking the validation problem from different angles, the team as a whole succeeds. uAV uses white box testing methods to specify conditions, write tests, and gather coverage.

Consider a design under test to be a box of logic. Black box testing looks at the specification of the inputs and outputs of the box, and tests to those specifications. In the case of IA-32 validation, our [Architecture Validation](#) (AV) team relies primarily on black box testing. They take the *Software Developer's Manual* and test that the processor behaves as specified. Most of their tests are portable from one IA-32 processor to another.

White box testing takes the cover off the box and looks at how things are implemented. uAV relies heavily on white box testing. The uAV Validator looks at internal protocols and looks for bugs that exist in those protocols. The Validator writes tests and checkers that test the implementation of the specification. As a result, the uAV Validator is intimately familiar with the inner workings of the unit or cluster being validated.

Both black box and white box testing need to be done to fully validate a complex design. uAV will find bugs that black box testing will not and vice versa. A simple example is a queue-full boundary condition. Black box testing would not know or care about the scheduling queue used to feed micro-operations to the core. If there is a bug when the queue fills, black box testing would not

target it, but white box testing would. Conversely, if an IA-32 instruction does not behave the way the specification says it should, white box testing might not catch the problem. White box testing derives its test plans from the bottom up of a design. Black box testing starts with the architecture and drives down to the design.

6.4 Scope

The DUTs within the CPU are arranged hierarchically; currently we utilize 3 levels of hierarchy: the lowest level is known as a cluster, the mid-level is a supercluster, and the top level is known as full-chip (see Figure 41). There is another level, emulation, which takes the simulation and puts it on a higher speed hardware emulator. uAV utilizes most of these levels for validation activities, and names the activities after the level (Cluster uAV, or Supercluster uAV). uAV activities at all levels use white box testing as their method, but their scope is different.

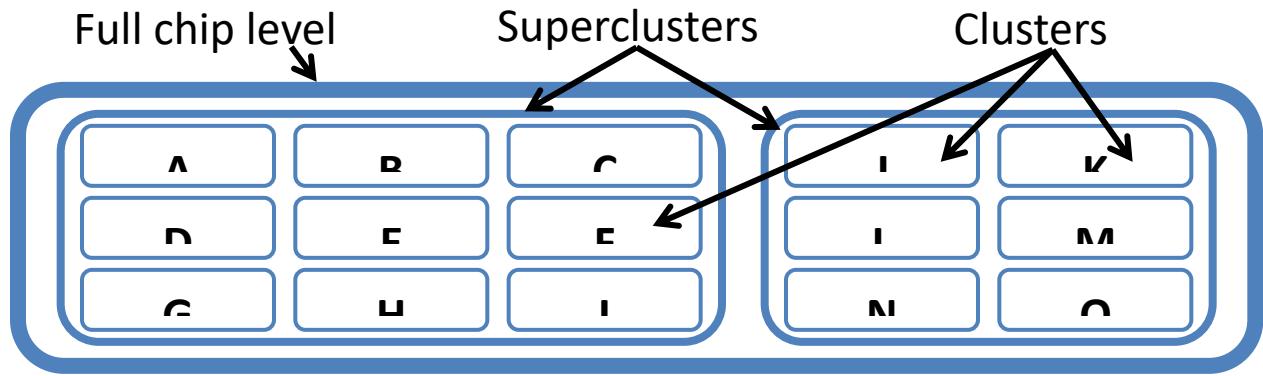


Figure 41

6.4.1 Cluster uAV

Cluster uAV (CuAV) utilizes the divide and conquer methodology by breaking the design into smaller pieces and doing microarchitecture validation on those pieces. We partition related RTL units into clusters and create software drivers and checkers that emulate the logic around the cluster being tested. We call the emulation and checking software the test environment (TE). Writing and maintaining a TE requires a lot of effort. Tests developed for a cluster are not reusable in other clusters and sometimes do not apply to new micro architectures. Some cluster testing is portable to the supercluster level. The tests are highly customized to the specification of the cluster being tested. The investment in a TE is large, but the return is also great. The payoff of cluster testing is greater control, testability, and simulation speed than higher-level DUTs. We find bugs and fix them faster in the design process and stress the design harder and more efficiently than at supercluster or full-chip.

For example, mispredicted branches are a big source of bugs within CPU cores. The processor speculates down predicted branch paths doing lots of work along the way. When the branches eventually resolve, the state of the processor requires cleanup. This is complex and problematic for the out-of-order and memory sections of the machine. To test the out-of-order and memory

clusters at full-chip, it is necessary to create branches that mispredict and have interesting operations down those branches' predicted paths. To do this, you would need to have in-depth knowledge of how branch prediction works in the Instruction Fetch cluster. To test mispredicted branches using TEs, the out-of-order and memory clusters inject bogus conditions directly into the cluster. Now a test can line up bogus operations with lots of interesting conditions quickly and easily.

The cluster uAV team is the biggest customer of cluster models. Because of this, the CuAV teams are the guardians of cluster health. The team actively runs mostly random and some directed tests on the cluster models to uncover bugs. The teams adds tests to the gating regressions to qualify all incoming design changes (see [Regressions](#)). Finding bugs at the higher-level DUTs involves longer debug times and slower simulation speeds than at cluster; thus, the CuAV team must find the vast majority of logic bugs within the CPU to maximize the efficiency of the overall Validation team.

CuAV methodology uses mainly white box testing. The general validation methodology of test plans, testing, debugging, checking, and coverage all apply at cluster. When writing test plans, we look at the RTL and relate it back to what was supposed to be implemented. Microarchitecture specifications (MAS), if they exist, provide a starting point for what to test but do not provide enough detail. We look at the RTL in addition to the MAS in order to write our test plan conditions (see [Testplan Writing](#)).

6.4.2 High level DUTs

Some level of uAV is done at higher level DUTs. Supercluster uAV is the process of doing white-box validation of the processor at DUTs that combine logical clusters. For example, in our current CPUs we have a Core and unCore Supercluster. Full Chip uAV combines super clusters to form the full chip. Emulation takes the full chip DUT and puts it on a hardware accelerator.

If we have such great controllability at cluster, why bother with higher level testing? First, to test algorithms and protocols involving multiple clusters. CuAV is very good at testing individual units and units that communicate within a cluster. CuAV Validators focus on testing their unit and cluster. Higher level testing takes a step back and looks at the bigger picture. For example, self-modifying code involves the uncore, the front end, the memory cluster, the out-of-order cluster, and microcode. Someone focusing on global protocols at full chip finds bugs that cannot be found at cluster. Second, the TE emulates how someone *thinks* the algorithms in neighboring clusters work. An algorithm might not really work that way or might have been intentionally abstract and needs to be tested at higher levels.

Both super cluster and full chip uAV focuses on cross-cluster interfaces and is, therefore, more protocol and feature based than CuAV. In rare cases, uAV makes decisions not to test some things at cluster and instead test them at higher level DUTs. This occurs when the cost to add TE support and test generation capability for a feature outweighs the efficiency gained by testing the feature at the lower level DUT. In hindsight, we have made the mistake to test features at higher level DUTs multiple times. Our experiences have taught us that if you can test it at cluster, you should.

At both supercluster and full chip, we still rely heavily on random testing to validate cluster assumptions. Ideally, super cluster can inherit testing from clusters and that does happen in some areas. We also sometimes coverage at the higher level DUTs to judge the effectiveness of our tests.

We reuse cluster-level coverage monitors for efficiency, though on occasion we will write a monitor specific to supercluster or full-chip. Coverage is now converging to the point where cluster level coverage can be run on any DUT, including emulation and potentially actual silicon.

6.5 Organization

6.5.1 As Design goes, so goes uAV

The uAV Validator must be an expert in the unit and logic being validated. Organizing the team to work closely with Design is the key. Our current CPUs are broken into superclusters, the superclusters are broken into clusters, and the clusters are broken into units. Recent projects had, for example, two superclusters, four or five clusters per supercluster, and three to six units per cluster. Design is organized by physical “sections” which roughly correlate to a logical cluster. Each section/cluster has several units and there is typically one Designer per unit. For uAV, there is typically one Validator per unit. The Validators for units within a cluster report to the cluster uAV manager or team lead. The cluster uAV managers report to the uAV manager. A cluster uAV manager will often manage multiple clusters, which may also include managing a supercluster.

6.5.2 Team size

The uAV team historically has the largest team size compared to other validation teams. The uAV team size is a direct function of the size, the complexity, and the amount of change in the project. On many CPU projects this meant having at least one uAV Validator per unit. Other validation teams are organized and staffed based on functionality. [Architecture Validation](#) staffing is based on the IA-32 feature set. uAV’s staffing is a function of the implementation. On a design with fewer units and less complexity, the uAV team would be smaller. There is no hard and fast rule that uAV must be the largest validation team. One previous CPU proliferation project, Westmere, had a large amount of architectural change (much of it in ucode) and a small amount of uarch change, allowing the uAV team to be much smaller.

6.6 Interactions

6.6.1 Design

uAV Validators are the interface to the Design team. During FED, they work closely with the Design team to test new RTL as it is being developed. uAV Validators (as well as other teams’ Validators), Designers, and Architects are assigned to a *Virtual Team* assigned to a feature. This virtual team owns the implementation, exercise, and turnin of that feature. Validators write TE code and tests, debug failures, and work with Design to identify the best fixes. They also participate in RTL code reviews and review any design specifications. Designers rely on Cluster uAV Validators to show their code is healthy, which enables them to move on to other design tasks.

During Execution, cluster, supercluster and full-chip uAV continue to find bugs and drive confidence in the design via coverage. Because Design is busy with circuits, timing, and layout, designers and Validators spend less time with each other but continue to interact when bugs are

found. Design provides valuable feedback to cluster uAV and full-chip uAV by reviewing test plans and exit checklists.

6.6.2 Architecture

During Technology Readiness, uAV Validators work closely with the Architects to give feedback on the complexity and risk of new microarchitecture features. The uAV team is small and focused on giving feature complexity assessments to architecture and figuring out what tools and methodologies to use in FED and Execution. During FED and Execution, uAV works with Architecture to fix protocol bugs discovered during testing. Architecture also plays a key role in reviewing Cluster uAV and Full-Chip uAV test plans and exit checklists.

6.6.3 Other Validation Teams

uAV also works with and relies on other validation groups to be successful.

6.6.3.1 Architecture Validation

The [Architecture Validation](#) (AV or IAV) team writes test plans and does testing based on the architecture definition of the processor. Sometimes the IAV effort overlaps with uAV, as in the case of opcode coverage in the front-end or paging modes in the backend. uAV is always looking at the architectural cases from a microarchitecture point of view. Together, the two approaches are very effective at rooting out bugs.

The AV team typically works at the Core Supercluster level and has been moving to a more random based testing methodology utilizing post-silicon test generators run on simulation or emulation. In recent projects, the AV team has been responsible for doing Core level uAV activities, which includes validating cluster level assumptions. This has allowed the uAV team to focus on other high priority areas including unCore level Supercluster testing.

6.6.3.2 Feature Architecture Validation

The [Feature Architecture Validation](#) (FAV) team writes test plans and does testing typically at the supercluster, full chip or full chip emulation level. They also have moved to using post-silicon random test generators. On recent projects, they've taken over the role of Full Chip uAV and validate cross cluster and cross super cluster assumptions. The uAV often helps debug failures these failures.

6.6.3.3 Power Management Validation

The Power Management Validation (PMV) team, also known as the [Reset and Power Management Validation](#) (RPMV) team, targets power management conditions within multiple DUTs – working within clusters, superclusters, and full-chip similarly to uAV. Due to the heavy interaction space between power management conditions and uAV conditions, the two teams must work together to

ensure that the cross-product conditions are being tested. In previous projects we have simply enabled the MPV test generators to utilize uAV test sequences without coverage feedback. We are starting to implement basic coverage feedback to show that we have tests that do both power management flows and interesting uAV sequences. We are also actively looking for ways to push power management testing into Cluster uAV, even at the expense of additional TE emulation work.

6.6.3.4 Formal Verification

Compared to uAV, Formal Verification (FV) is a very different validation approach. FV tries to formally prove that the design is correct, encountering bugs along the way. uAV provides confidence that the design is not broken by running tests, running checkers, and using coverage.

FV uses uAV Validators for microarchitectural knowledge. uAV provides feedback to FV on what areas of the design to focus on proving. This requires uAV Validators to have a basic understanding of the strengths and weaknesses of formal verification.

On recent projects, FV techniques have taken over as the primary bug finding vehicle for certain protocols and units. We've experimented with uAV Validators taking on this work and are evaluating ways to move FV techniques into the uAV domain.

See [Formal Property Verification](#) for more details.

6.6.3.5 Mixed-Signal Validation

The [Mixed Signal Validation](#) (MSV) team targets circuits that mix both analog and digital circuitry. To do this efficiently, the MSV team utilizes much of the TE collateral created by the uAV team as well as some of the tests. The MSV team then creates extra TE and test collateral on top of this. The two teams help each other debug failures found in their respective domains.

6.6.3.6 Design for Test Validation

The [Design for Test Validation](#) (DFTV) team validates the operation of the CPU's DFT features, such as debug triggering and observability features. The DFTV teams makes use of the cluster and supercluster test environments provided by the uAV team.

6.7 uAV tasks

The general tasks for uAV are listed below:

- Write the TE - emulation and test interface
- Basic testing
- Basic checking
- Writing test plans
- Test plan reviews

- Writing random test generators
- Running tests
- Writing checkers
- Writing injectors
- Debugging
- Filing and validating bugs
- Writing coverage monitors
- Driving and analyzing coverage
- Revalidation and exit

7 Summary

uAV is an integral part of the pre-silicon validation and Design team. uAV validates the microarchitecture of the processor using white-box techniques. The goal is to find bugs as quickly as practical and provide confidence that no bugs block post-silicon testing. uAV works within multiple levels of DUTs, currently: cluster, supercluster, and full-chip. Cluster uAV takes advantage of faster RTL and more controllability to weed out bugs quickly. Tests written at cluster are not necessarily reusable at the higher-level DUTs. Supercluster and full-chip uAV use random tests to verify cluster assumptions and test cross-cluster protocols. All levels of uAV are needed to find bugs and provide confidence in the design.

Like the Design team, the uAV team organizes itself by cluster. The uAV team is the largest pre-silicon validation team because of the complexity of the microarchitecture. The team interacts with Design, Architecture, and other validation teams throughout the life of the project. uAV has defined processes that need to be done to call the chip validated.

8 Future Work

9 References

The Art of Pre-Si Val: Chapter 22

Introduction to Formal Verification

By: [Erik Reeber](#)

1 Abstract

In this chapter, we survey the validation discipline of formal verification (FV), discuss how to determine whether formal verification should be used in place of or alongside simulation, and provide some general formal verification advice. This chapter is intended to be a high level overview that avoids most of the details required to actually apply these techniques. The following chapters will provide more detail for the particular FV approaches that are most commonly applied within DDG.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	8/20/2016	Initial Draft	Erik Reeber	Amber Telfer

3 Contents

1 Abstract.....	487
2 Chapter Revision History	487
3 Contents.....	488
4 Purpose.....	489
4.1 Why do we need this chapter?.....	489
4.2 What does this chapter cover?	489
4.3 What does this chapter not cover?	489
5 Background Information.....	489
6 Introduction for Formal Verification.....	489
6.1 The Formal Verification Mindset	489
6.2 FV Approaches	491
6.2.1 Formal Property Verification (FPV)	491
6.2.2 Protocol Formal Verification.....	491
6.2.3 Symbolic Simulation.....	491
6.3 When to use Formal Verification.....	492
6.3.1 Hypothetical comparison of different techniques	492
6.3.2 What makes a project well suited to FV	495
6.4 General FV Advice.....	496
7 Summary.....	497
8 Future Work.....	497
9 References.....	497

4 Purpose

4.1 Why do we need this chapter?

FV is an important hardware validation technique that has been used successfully by our team on multiple occasions. For those who have never used FV before, we hope that this chapter will provide a basic understanding of FV and help them know when it should be applied in their work.

4.2 What does this chapter cover?

This chapter provides a brief overview of four of the major FV techniques: theorem proving, model checking, symbolic simulation, and SAT solving. It also discusses what sort of validation problems are most amenable to FV and gives some general advice regarding the use of FV.

4.3 What does this chapter not cover?

The descriptions of the various FV techniques are very brief and are not intended to have sufficient detail to fully utilize them. For more detailed descriptions of the techniques commonly applied at Intel, read the following chapters. This chapter is also not intended to discuss particular domains, such as power management, or core design, and how FV might be applied within them.

5 Background Information

This chapter references IPs and other basic concepts found in the [IP Validation](#) chapter. It also helps to understand the basic concepts of stimulus, checking and coverage as described earlier in *The Art of Pre-Silicon Validation*.

6 Introduction to Formal Verification

6.1 The Formal Verification Mindset

The key difference between simulation and formal verification is that whereas simulation considers one input at a time, formal verification attempts to validate all inputs at once.

For an example, let us imagine that we need to validate a hardware design and our specification says that the following needs to hold:

Output `my_out` is equal to the expression $(A / B) \& D$, where A, B, and D are inputs.

When we look at the RTL though we see the following:

```
assign my_out = A & D
```

Should we file a bug? Well, that depends on the context. For example, if B is known to be 0 at this point, then there is no problem, since then $A \mid B$ is equal to A. In our case, let us say that every bit in D can only be high if A is high. In other words, there exists a C such that

$$D = A \& C$$

One approach now to validating this problem would be to try various values of A, B, C, and D and compare our design to its specification:

A	B	C	D (A & D)	Design (A&D)	Spec ((A B)&D)
0	0	0	0	0	0
1	1	1	1	1	1
0b1011	0b1100	0b0011	0b0011	0b0001	0b0001
0b1001	0b0111	0b1100	0b1000	0b1000	0b1000
0b0011	0b1111	0b1100	0b0000	0b0000	0b0000
0b1111	0b1001	0b1111	0b1111	0b1111	0b1111

This is essentially the simulation approach to validation. Notice that if A, B, and C are 64 bit integers then in order to fully validate the statement you need 2^{192} rows in the table, which is about 10 million times the number of atoms on the Earth. However, we can build a high degree of confidence that our design matches our specification by testing a representative subset of these numbers.

The formal verification approach here, by contrast, would be to find some method or tool that can prove our design matches the specification for all inputs. In this case, any tool that does symbol manipulation will find that the design is correct, since if we plug A & C in for D in our specification we get

$$\begin{aligned}
 (A \mid B) \& D \rightarrow (A \mid B) \& A \& C \rightarrow A \& A \& C \mid B \& A \& C \rightarrow A \& C \mid B \& A \& C \\
 &\rightarrow \\
 (1 \mid B) \& A \& C \rightarrow 1 \& A \& C \rightarrow A \& C
 \end{aligned}$$

And if we plug in A & C in for D in our design we get

$$(A \& D) \rightarrow (A \& A \& C) \rightarrow A \& C$$

Again the core concept here is that instead of treating A, B, C, and D as constants, in formal verification we keep them as variables, considering all cases at once. When FV succeeds it can therefore provide a level of assurance that is impossible to reach through simulation.

6.2 FV Approaches

The methods by which formal verification validates all inputs at once varies based on the tool and the underlying technique. It is often not necessary to fully understand the underlying techniques to use FV tools. Furthermore, many FV tools, such as the ones generally used by DDG, do more than simply validate properties, they also provide counterexamples---a set of inputs for which the design fails in cases where the design cannot fully satisfy its specification.

In this section, we will briefly describe the FV approaches that have been successfully applied by the DDG team.

6.2.1 Formal Property Verification (FPV)

This chapter presents the most common formal technique currently used in DDG. The FPV methodology is based around specifying the correctness of the design with System Verilog Assertions (SVAs) and then constraining the inputs through similar assumptions. A fully automated tool then can then determine whether the assertions hold for the design within the given input space.

We recommend those first considering a formal verification approach. This approach is the most common and most automated formal technique and can often be used as a replacement or supplement to simulation. For a better understanding of the FPV methodology read the [Formal Property Verification](#) chapter.

6.2.2 Protocol Formal Verification

This chapter presents our methodology for planning, developing, and formally verifying a Protocol architectural or micro-architectural model. Unlike the other FV techniques described in this book, the Protocol FV technique is intended to be applied before RTL exists. After developing a high-level formal model and specification of the intended protocol architecture, the model is evaluated for reachability by a model checker. Using this approach, many architecture bugs can be flushed out early in the project life cycle, saving significant time and resources.

After the protocol is validated, the model has added value when used during the architecture development by converted the model's rules into SVA properties and validating them on the RTL implementation. This provides confidence that the implementation does not introduce bugs by not adhering to the protocol's specification. Read the [Protocol Formal Verification](#) chapter learn more details on the flow.

6.2.3 Symbolic Simulation

This chapter presents the basic ideas behind the Symbolic Trajectory Evaluation (STE) technique. The STE technique is used extensively by the core validation team to validate the pipelined execution path. It is also being used to validation some of our microcontrollers and forms the backbone of most of our FEV tools, which validate that the backend design matches the RTL.

Symbolic simulation is a specialized technique that generally requires a higher degree of expertise to apply than FPV, but is extremely powerful when applied to straight-forward pipelined designs. The technique tends to be less valuable, however, for designs with large amounts of control logic or loops, such as finite state machine (FSM) designs. The STE methodology has been a key part of the high quality of the arithmetic units in big core designs. Any Validator who may need to validate a data-driven pipeline or arithmetic logic, we recommend reading more about STE in the [Symbolic Simulation](#) chapter.

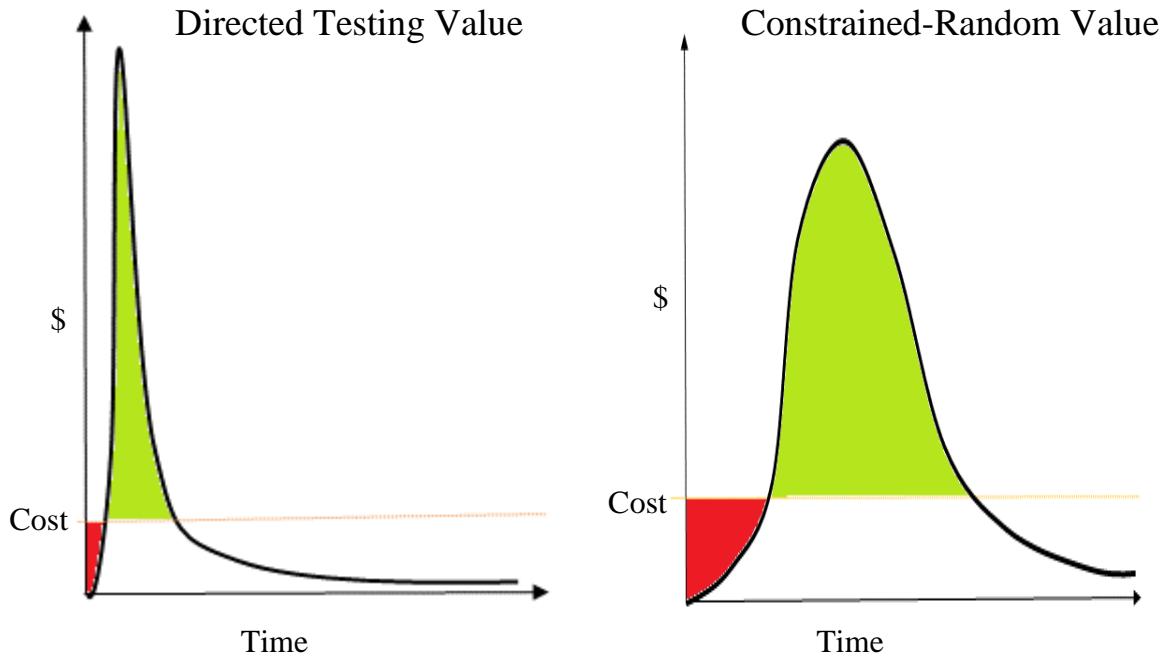
6.3 When to use Formal Verification

There is no hard and fast rule as to when formal verification is the best approach to use on a given hardware validation problem. Some people might argue that formal verification is almost always the best approach, given that it produces a lower risk of Post-Silicon escapes than any other methodology. On the other hand, other people might argue that you should almost never use it because the collateral you inherit usually does not include formal verification and the next project that inherits your code may not include any formal verification experts.

In reality, within DDG we have applied formal verification to some problems and not others, always attempting to use it on problems where it is most suitable. In the following section we will show a comparison of the directed testing, random testing, and formal verification on a hypothetical problem to help illustrate how the techniques tend to shape up against each other. After this hypothetical comparison we will go over some of the traits that make a given problem more or less suitable to formal verification.

6.3.1 Hypothetical comparison of different techniques

On a hypothetical IP validation project, let's start by comparing the directed testing strategy to the constrained-random testing strategy with the following graphs:



Here value (in \$) is defined as something like:

$$\sum P_i * F_i * V_i$$

Where P_i is the probability of a given bug i existing, F_i is the probability of finding it over a given time period, and V_i is the value of finding it (or the cost of not finding it) at this point in the project. Of course, none of these values are easy to determine on a real project, but we can imagine knowing them for this discussion. For example, on a given project there might be a 10% chance of a bug blocking our boot flow and stopping all progress. In this case, both directed and constrained random testing have a very high chance, say 99%, of finding it. Assuming the bug is not hidden from full chip simulation, it would not escape to Silicon, but still there is significant value in finding it at the IP level due to the increased costs and delays associated with full chip simulation. We could imagine that value of finding this bug at the IP level as \$5000, thus the value generated by the potential for this bug is about \$500 and it would be part of the value line in both graphs.

The orange “cost” line represents the cost of validation. This includes the cost of labor, computing, tools, and office space. Whenever the value generated by validation exceeds the “cost” Intel is profiting. Generally, validation has a ramp up time, where investment is being made despite the probability of finding bugs being low, followed by a profitable time where the value of validation exceeds its cost. Eventually, the value starts to drop and the logical place to stop validating is when the value again hits the cost line.

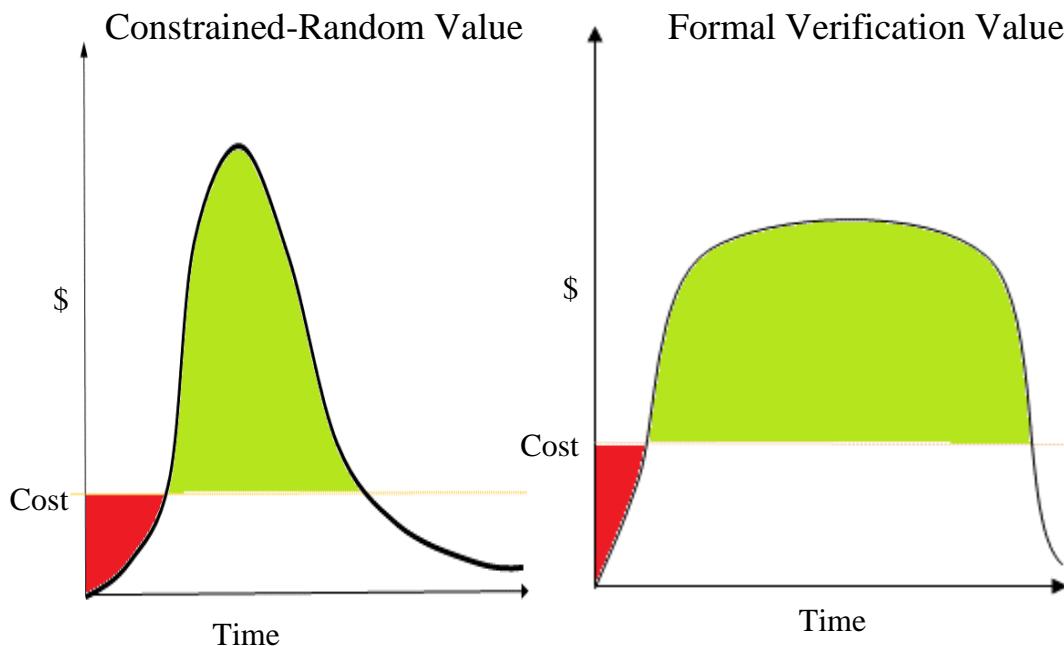
The green area above the cost line represents profit, whereas the red area below the line represents an investment. The best strategy here is therefore the one that produces the largest amount of green space after subtracting the red space.

In our hypothetical scenario, therefore, you can see that constrained-random testing beats directed testing. Directed testing has a relatively small up-front investment, a high peak value, and a slightly

lower cost line, but this is more than made up for by the much larger time that constrained random testing produces high value.

While this is a hypothetical scenario, the graphs here are not far from the truth on many projects. Constrained-random testing does generally take longer to ramp up than directed testing, due to the need to set up a large test bench and checking infrastructure. However, it often produces more value over time because that infrastructure will exercise the system in ways that the Validator never envisioned. On the other hand, directed testing is relatively quick to set up, but requires the user to constantly come up with new stimulus and determine what the result of each new stimulus should be. While this is likely to find high-value bugs early the probability of finding a bug quickly diminishes over time.

Now let's compare constrained-random testing to formal verification:



In our hypothetical example, formal verification produces more value than constrained-random testing. The cost line for formal verification is higher and the peak value is lower, but the ramp-up time is smaller, and most importantly the value stays high for much longer.

Again, while this is just a hypothetical project, these graphs do not seem hard to believe for someone experienced with both formal verification and simulation. Formal verification ramp-up time is noticeably quicker than constrained-random simulation because there is no large test bench infrastructure required. And the value line tends to be much more flat with formal verification than with simulation. This is largely because simulation tends to find the worst bugs first whereas formal verification finds bugs in a seemingly random order. Furthermore, the odds of finding a bug with FV do not drop significantly until you have completely formally verified the project, which takes quite a while. In the end, formal verification almost always takes more validation effort than simulation, but in return you get a much smaller chance of any bugs escaping.

Note that there can be a temptation to end a formal verification project early because it is taking longer than simulation to complete, but doing so is usually a mistake.

6.3.2 What makes a project well suited to FV

In our hypothetical example, FV turned out to be the best approach, but clearly we have not decided to always use FV in DDG. So when should we use it?

As always, we do not have any strict rules, but the following questions are worth considering:

- **How costly is a bug in this area?** Some areas of a design can be easily modified Post-Silicon, whereas others need to be absolutely perfect on the first stepping. The worse bugs are, the more value there is in the overall validation effort. This is equivalent to lowering the “cost” line in our hypothetical graphs in the previous section, because the validation cost is lower relative to its value. As you can see in the graphs, the lower the cost line is, the more FV tends to shine. This is because the other approaches have diminishing returns as you put more effort into finding the last few bugs—FV, on the other hand, just tends to find all the bugs, and there is little point in looking for more after you are done.
- **How likely are corner-case bugs?** Some designs, such as boot flows are executed pretty much the same every time. In this case, directed testing often works great. On the other hand, some designs, such as floating point dividers, tend to lend themselves to nasty corner-case bugs. Corner cases can be very challenging to catch with simulation, but FV is relatively good at handling them. When you have a design where the bugs are likely only happening for one in a trillion inputs, FV becomes a tempting choice.
- **How much persistent state is in this design?** The best designs to apply FV to, such as the EXE cluster in a core, tend to be those with very little long term state. On the other end of the spectrum, are designs with microcontrollers embedded in them, where the entire instruction and data RAM count as persistent state. If a design has very little state then model checkers may not be necessary, and you can apply symbolic simulation or SAT solving on the design directly. On the other hand, if a design has a ton of state, then no model checker will be able to validate anything non-trivial about it. This is why FV is generally applied to smaller portions of designs, avoiding any areas with heavy state. The final SOC inevitably has too much state for a model checker to handle.
- **How well defined is the specification?** FV tends to require a complete specification of the desired design behavior, whereas simulation can often simply avoid odd stimulus or checking ambiguous cases. If the design is not well specified then FV inevitably includes the effort of creating a good spec. On the other hand if a design is well specified up front, for example, if it must follow an IEEE standard, then this effort is already complete making FV a little easier.
- **How complex is the specification relative to the design?** Generally, FV does best when the specification is relatively simple and the design is relatively complex. This is largely because complex designs tend to have corner-case bugs and simpler specifications are easier to write (and FV requires a fully written specification).
- **How experienced is your validation team with FV?** Obviously it is easier to use FV if the validation team slated for the project is already experienced in FV. If a design is well suited to FV it may be worthwhile to build the experience even if it is not currently present in the team, but for borderline cases it is best to use the techniques you already know.
- **What validation collateral is the project inheriting?** If your team is inheriting a large amount of simulation-based collateral then it is harder to justify switching to FV rather

than expanding on the existing collateral. Similarly, if your team is inheriting a large amount of FV collateral, then it is harder to justify switching to simulation. Even if you hit an unacceptable restriction with model checking—where your state space is too large—it may be worthwhile working around that with some other FV technique, such as theorem proving, rather than moving away from FV.

Generally speaking, FV makes the most sense at the smallest level of validation—the IP or unit level rather than at the integration level. It is probably currently being underutilized at this level due to lack of experience and the momentum generated by having large amounts of existing simulation-based collateral. While, in the short term it often makes sense to choose simulation, in the long run we need to build up our FV experience as well. Over time FV will likely become even more compelling as it is likely that advances in FV will outpace advances in simulation-based methodologies.

6.4 General FV Advice

To complete our introduction to FV we would like to conclude with some general advice that we have gained through our experience. Any particular technique will have its own methodology and more detailed advice on how to use it, but some truths seem to pervade across all of FV:

- **Pick a problem that your technique can handle.** A common mistake made by people new to FV is to attempt to tackle a problem that is very challenging even for an advanced person and may even be impossible with the technique you have planned. To avoid falling into this trap, consult someone with FV experience while determining whether to use FV on your particular problem.
- **Expect FV to take longer than simulation.** It is not necessarily true that FV will always require a longer validation schedule than simulation, but you should be prepared for this possibility. As we showed in our hypothetical comparison of techniques, it is quite plausible for FV to be the best choice for a problem but still require a longer schedule than simulation.
- **Carefully review your specification.** Validation plans in FV tend to look a little different than in simulation. Simulation validation plans often revolve around what stimulus you plan to create, whereas FV defaults to all possible stimulus. With FV you should instead review your specification and assumptions with the design and architecture teams. Make sure they understand what you are planning to validate, and **what you are not going to validate**. Incomplete specifications are easy to create, and you must be extra careful to validate everything that is required for the design to function.
- **Validate progress.** A particularly common incompleteness in FV specifications is to not validate progress. It is easy to focus on safety properties, those that show that the design does not do anything wrong, but it is usually equally important to show that the design does not hang. FV should validate both.
- **Validate your assumptions.** FV efforts always produce assumptions about the input space to avoid illegal conditions. These assumptions should be validated (possibly with simulation) at a higher level of the design. Validating them is an important sanity check to ensure you are validating the actual design space and not missing important legal inputs. It also can often find integration bugs, where your design is being used improperly.

- **Use “bogus” assumptions early on, and then remove them.** It is often a good idea early on to include “bogus” assumptions which restrict your input space to a subset of the legal space. For example, you might assume that your design’s clock always toggles even though in reality it may be sometimes gated. These “bogus” assumptions help you start debugging failures earlier and make early debug faster. Once proofs are all passing with the “bogus” assumptions though, these assumptions will usually need to be removed to validate the remainder of the legal input space.
- **Fall back on theorem proving.** Because theorem proving scales well, it is often a good idea to fall back on theorem proving techniques when other FV techniques hit a wall. For example, induction and generalization can be used to create properties that can be validated by symbolic simulation. Case splitting often turns a large difficult property into a set of smaller easier properties. And ordinals can turn a progress property that is challenging for a model checker into a bunch of safety properties that the model checker has no difficulty proving.

7 Summary

FV is a powerful set of techniques that when applied to hardware verification problems can often result in a higher quality design than is possible with simulation alone. In this chapter, we discussed the FV mindset, gave an overview of four major FV techniques, provided guidance on the types of designs most amenable to FV, and provided some general FV advice.

FV is a tool that every person doing hardware validation should know when to apply. In the subsequent chapters we will discuss some of the particular FV techniques that have been used by our team in more detail.

8 Future Work

This section intentionally left blank.

9 References

1. Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir A. Frolov, Erik Reeber, Armaghan Naik: Replacing Testing with Formal Verification in Intel Core™ i7 Processor Execution Engine Validation. *CAV 2009*: 414-429
2. M. Kaufmann and J S. Moore. ACL2: A Computational Logic for Applicative Common Lisp. See URL: <http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html>
3. T. Nipkow, L. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant for Higher Order Logics, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, London, UK, 2002
4. G. Dowek, A. Felty, G. Huet, C. Paulin, and B. Werner. The Coq Proof Assistant User Guide Version 5.6. Technical Report TR 134, INRIA, Dec. 1991.

5. S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE 1992)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748—752, London, UK, June 1992. Springer-Verlag
6. D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148—200, 1998.
7. E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs*, Workshop, May 1981, volume 131 of *Lecture Notes in Computer Science*, pages 52—71, 1982, Springer-Verlag.
8. The Latest Version of the HOL Automated Proof System for Higher Order Logic. See URL: <http://hol.sourceforge.net>
9. W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on System Verification. *Journal of Automated Reasoning*, 5(4):409—530, 1989.
10. J S. Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. *10th Anniversary Colloquium of UNU/IIST 2002*: 161—172.
11. Anna Slobodová, Jared Davis, Sol Swords, Warren A. Hunt Jr.: A flexible formal verification framework for industrial scale validation. *MEMOCODE 2011*:89-97.

The Art of Pre-Si Val: Chapter 23

Formal Property Verification

By: Prakash Math and [Celia Wall](#)

1 Abstract

Formal Property Verification (FPV) has been successfully applied to Intel IA-32/EM64T CPU designs over a wide range of pre-silicon validation problems. However, the application of Formal Methods has long been considered more of an art form than an engineering practice. There is a myth that it can be used only by Formal Verification experts. FPV practice and methods have evolved from being an art form to a standardized practice that can be applied by any design engineer. The FPV method is presented in this chapter as a step-by-step process that can be applied by any engineer familiar with hardware design. There remain some challenges to apply the FPV methodology successfully. These challenges are explained along with heuristics on how to overcome them. FPV methods can be a powerful tool in the hands of an RTL coder or a Validator. They have many advantages over traditional validation methodologies. FPV's correct use can result in a higher quality RTL, at a much earlier stage in the design process, and with much less effort when compared to traditional simulation-based validation methods.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	5/30/2012	Initial Complete Version	Prakash Math	Michael Bair / CCDO Val Staff
1.1	8/20/2016	Major Update	Celia Wall	Amber Telfer

3 Contents

1 Abstract.....	499
2 Chapter Revision History	499
3 Contents.....	500
4 Purpose.....	502
4.1 Why do we need this chapter?.....	502
4.2 What does this chapter cover?	503
4.3 What does this chapter not cover?	503
5 Background Concepts.....	503
5.1 Bounded Model Checking.....	503
5.2 Initial State of the FV model.....	503
5.3 Formal Specification: Liveness and Safety	504
5.4 Design Under Test.....	504
5.5 Model size reductions	504
5.5.1 Free	504
5.5.2 Blackbox	504
5.5.3 Set	505
5.5.4 Parameter reductions.....	505
5.6 Proof Wiggling	505
6 FPV Methodology.....	505
6.1 Feasibility Analysis	506
6.1.1 High-level Verification Plan	507
6.1.2 Identifying the DUT	507
6.1.3 Preliminary Analysis of the DUT	508
6.1.4 Decomposition Strategy.....	508
6.1.5 Effort Estimate	509
6.2 Verification Plan.....	509
6.2.1 Verification Strategy.....	509
6.2.2 Review of verification plan	511
6.2.3 Dynamic Plan.....	511
6.3 FV environment development.....	512
6.3.1 Restricted Environment.....	512

6.3.2	RTL Abstraction	513
6.4	RTL Verification	514
6.4.1	Coding Specifications	514
6.4.2	Dealing with RTL bugs in FV	515
6.5	Reducing FV complexity	515
6.6	Quality Checks.....	517
6.6.1	Assumption Checking:	517
6.6.2	Cover Points	518
6.6.3	Trigger and Vacuity Checks.....	518
6.7	Proof Review	519
6.8	Maintenance	519
7	Full IP/IU Validation by FPV.....	519
8	Design Exercise using FPV	520
9	Advantages of FPV over dynamic validation.....	521
10	Summary.....	522
11	Future Work.....	522
11.1	FPV Coding Methodology	522
11.2	Porting Proofs from Previous Projects	522
11.3	FPV in Mixed-Signal Validation.....	523
11.4	FPV in Post-silicon Debug Analysis.....	523
12	References.....	523

4 Purpose

4.1 Why do we need this chapter?

Formal Property Verification (FPV) is particularly adept at finding deep bugs in RTL designs and providing high confidence in complicated and hard to validate functional units.

We have refined FPV over the course of the past decade while verifying functionality in several Intel IA32/EMT64T microprocessor designs, IP and integration units. With the objective to push verification earlier in the design process and as the design evolves, our methodology strives to provide early feedback as features develop and to ensure that changes do not introduce new bugs. FPV has been an efficient strategy to complement or replace simulation-based validation activities in many design areas.

The FPV methodology described here has key advantages over other validation methodologies when applied correctly. Traditional simulation-based validation methodologies rely on onion layer peeling of simple RTL bugs early in the development phase, and successively finding harder-to-hit corner case bugs to get RTL to a healthy state. The simulation-based validation can have disadvantages compared to formal validation strategy.

- It takes more time (or effort) to get RTL to tape-in quality;
- It also leads to a lot of RTL churn throughout the whole design cycle and more bugs introduced because of the churn. Bug fixes closer to the end of the design cycle tend to be less robust.

In contrast, FPV methodology exposes all possible failure scenarios (and hence bugs) at the same time, early in the design cycle. Thus, using FPV to validate RTL has significant advantages:

- The RTL is higher quality as measured by bugs present in the model at RTL release time. FPV reduces the onion layer peeling to get the RTL healthier over a period;
- The RTL model is more robust as measured by tendency for bugs to creep in after model release. This is because the RTL coder has the luxury of time earlier in the design cycle to explore all failure scenarios and implement correct fixes even if doing so means significant RTL changes;
- FPV provides much quicker feedback on the correctness of RTL changes made during the course of the project; and
- The design cycle is condensed, as measured by time required to get RTL model to tape-out quality.

FPV has other unique benefits compared to simulation-based validation methodologies.

- FPV collateral can be used for post-silicon debug analysis.
- FPV provides a medium for early design exercise, before TE environments are ready.

In this chapter, we describe the application of FPV as a standardized step-by-step process. This should be useful to any RTL coder or validation engineer who is looking to explore the use of FPV for the purposes of establishing design correctness.

4.2 What does this chapter cover?

In this chapter, we describe the FPV methodology as a step-by-step process. We describe in detail how each step is performed. We also address some of the challenges and barriers faced during the process and how to deal with them.

We contrast the FPV methodology with simulation based methodologies wherever applicable and necessary. FPV opens the door for unit-based validation and hence provides a powerful tool in the hands of RTL coders and Validators.

FPV also opens the door for several new usage scenarios. For example, it can aid the back-end design to implement perfect timing fixes. We cover a few of these new scenarios in the document.

4.3 What does this chapter not cover?

The FPV methodology described in this chapter is independent of the formal tools or the specification language used to develop the formal infrastructure. This chapter does not cover the use of formal tools and the various hooks, bells and whistles offered by tools to enable FPV. Similarly, we do not cover the specification languages used to write properties. Application of formal methods requires at least a rudimentary understanding of formal concepts. This chapter does not cover the theory underlying the use of formal methods.

5 Background Concepts

There are many formal concepts that may be unfamiliar to Validators new to FPV. Here we introduce some general FV concepts referenced throughout this document.

5.1 Bounded Model Checking

There are 4 main types of Formal Verification. This document focuses on bounded model checking, a type of FPV (Formal Property Verification). Bounded Model Checking is a method by which we use time to limit our search of the reachable state space. For a discussion on Model checking and other types of Formal Theory, see the Art of Validation: FV Theory Fundamentals.

5.2 Initial State of the FV model

Whether a Validator is using Formal methods or simulation, it is important to begin testing functionality of a circuit from a legal state. In FPV, the verification of a property is composed of an initialization phase followed by a verification phase. The verification phase is reliant on first getting the circuit into a sound and legal initial state. To achieve this, interface reset should be asserted for the duration of the initialization phase, to get state elements to their reset value for the beginning of the verification phase. Typically, additional interface signals must hold a constant value or sequence to bring the circuit to a sound initial state. When simple assertions can run past the first few clocks of the verification phase, a good initial state is reached.

5.3 Formal Specification: Liveness and Safety

Liveness and Safety properties are two types of properties employed in Formal Verification. Liveness properties assert that a logical statement must eventually happen. For example, a request must eventually be granted. Safety properties assert that a logical statement is true in a defined time window. For example, a request must be granted within 3 cycles. Liveness properties are compute intensive and should be avoided if there is an alternative safety property.

5.4 Design Under Test

Design Under Test (DUT) is a common hardware verification term that refers to the design block that is being tested. In FPV, we take a different approach to choosing our DUT. Since FPV tools explore all possible behaviors of the design, there is a fundamental limitation on the amount of logic they can meaningfully consume. This process of exploring all possible behaviors is computationally expensive, in terms of both time to compute and memory use. We cannot feed the formal tools arbitrarily large design blocks to test, as we are limited by tool capacity. One option is to select an RTL DUT with simple or standard interfaces, which is either naturally small enough, or can have its structure sizes reduced through RTL defines or parameter changes and make that our FPV_DUT. Another option is to carve out logic from the design block, to create the ‘DUT’. Here, by ‘carve out’ it means we remove the logic we aren’t targeting with our validation. Often times we carve out the data paths and state arrays, and only the control logic is included in the DUT.

5.5 Model size reductions

There are several methods used to carve out the DUT in targeted validation. These methods can also limit or restrict the state space. The most commonly methods are Free (stopat), Blackbox, and Set.

5.5.1 Free

Free, or stopat, is the equivalent of removing the fan-in of a signal and converting it to a primary input.

5.5.2 Blackbox

Blackbox is a way to remove modules from your DUT. It effectively removes the module and turns its interface into a primary input. It is used to remove logic from our DUT that we don’t wish to validate. One common usage for Blackbox is to remove data structures from the DUT in targeted validation.

5.5.3 Set

Set is performed on a primary input. It is used to assign a primary input to a logical 1 or 0.

5.5.4 Parameter reductions

Parameter reductions are a way to reduce the size of structures in the RTL in order to reach a deeper state space. We use RTL defines to reduce the structures only for our verification environment. It is important to do periodic validation on the non-reduced structure sizes as well, in case there are RTL bugs that cannot be hit with the reduced sizes.

5.6 Proof Wiggling

Proof wiggling happens as a result of doing early FPV environment setup. We can say we have performed proof wiggling when the following tasks are completed

- Clocks are running
- DUT is reset properly
- Initialization sequence is complete
- Outputs of DUT can toggle
- Can use simple assertions to begin to build FV environment

6 FPV Methodology

In this section, we describe the FPV methodology. It has been successfully applied on several Intel IA32/EMT64T microprocessor designs [1,2,3,4,5, and 6], most notably in the Cache controller logic in the Core (MOB, DCU, PMH, MLC, and APIC). It can accommodate validation of large and complicated features that span cluster and unit boundaries. More recently, it has been applied as the sole validation approach for SoC units and IPs: Cunit, IOMMU, AXI2IDI, MLM.

The methodology divides FPV activity into two tasks. The first task develops the FV environment. FV environment development requires a strong understanding of formal concepts such as proof decomposition, verification planning, and application of abstraction techniques. Someone familiar with formal methods and tool idiosyncrasies typically develops the FV environment. Similar to the dynamic methodology: the Test Environment (TE) expert develops the TE, and the validation task uses the TE to develop test content and test RTL. The development of the FV environment hides most of the formal complexity from the Validator. It also insulates the Validator from hitting failures due to missing environment constraints. The second task is the actual verification of micro-architecture features. This involves developing formal specifications based on some verification plan, running the FPV tools, and debugging failures. This activity is ideally suited to someone who knows the micro-architecture and RTL intimately. This strategy of delineating the FPV activity into two activities – one that leverages formal expertise to build the FV environment and the other that exploits the design knowledge to develop formal specs and validate the DUT, leads to strong and efficient verification.

As mentioned earlier, the key component of this methodology is development of a robust FV environment, similar to a cluster dynamic test environment. However, a fundamental difference allows FV environments to be ready much earlier than test environments. Test-Environments need considerable effort to specify the set of possible input stimulus. Formal tools assume all inputs are possible. Thus, formal tools require no new work to define new stimulus to test new features. The formal environment only adds restrictions that exclude behaviors that result in false counterexamples. Our experience has shown this can be a faster task to complete. Additionally, formal environments can simplify the problem by over approximating the set of legal inputs, resulting in more cases checked, but without violating the targeted assertions [1].

The FPV methodology can be further broken down into feasibility analysis, verification plan, FV environment development, RTL verification, reducing FV complexity, quality checks, and proof review/regression. The figure below provides a roadmap for the methodology flow that we describe next.

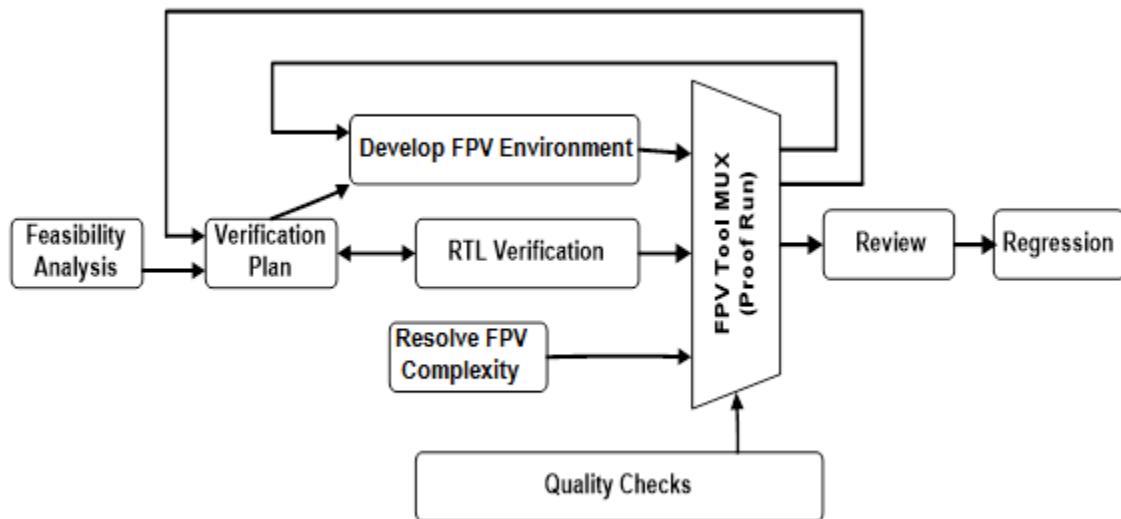


Figure 42 FPV Methodology

6.1 Feasibility Analysis

FPV tools encode all possible system inputs. However, like all exhaustive validation techniques, they have capacity limitations. Feasibility analysis involves understanding the scope of the validation challenge and determining if the validation goal is tractable using FPV given the project schedule and the resource constraints.

Feasibility Analysis attempts to

- Understand the high-level verification goal;

- Identify the DUT size such that; it has a well-defined boundary so that a meaningful verification plan can be developed; and the FPV tool is able to provide feedback - compile, run verification and report failures in a reasonable time;
- Compile the DUT and run a sample FPV specification to make sure that the tool response time is reasonable;
- Develop a high-level decomposition strategy and a high-level verification plan in consultation with architects and designers; and
- Develop preliminary effort-estimate for completing wiggling stage and other verification milestones.

6.1.1 High-level Verification Plan

An important aspect of starting any verification activity is to understand the verification objective. It is a simple statement about the correctness of a unit, or a feature or a protocol. Often times, the high-level verification goal is obtained by chatting with the designer or the architect. For example, the high-level verification goal for a cache controller could be to ensure that data coherency is maintained or that the MESI protocol is preserved. The verification goal can also be much smaller in scope. For example, it could be as simple as ensuring that a given logic never sends out two requests at the same time. It is this high-level verification goal that determines all the following steps in the FPV process: choosing the DUT, creating a decomposition strategy, and estimating the amount of effort involved in completing the task.

There is one more key advantage of FPV methodology: the effort involved in completing the task is directly proportional to the complexity of the behaviors validated. For example, consider that the verification goal is to ensure that a complex snoop-protocol is implemented correctly. The snooping behavior is complex with lots of interaction and cross-unit communication. It takes significant effort to set up an FPV environment that models the DUT interaction with surrounding logic. Hence, it takes a lot more effort to validate the snoop protocol. Conversely, consider the case where the verification goal is to ensure that two signals are always mutex. This, in all likelihood, is much easier to validate. Most of the guaranteeing logic that ensures the mutex property is usually contained in small cone of logic that the tool can easily handle.

The high-level verification plan should never be more than half a page. It is something one should be able to describe in a few minutes. The details on how this high-level verification goal is implemented are discussed in the [Verification Plan](#) sub-section.

6.1.2 Identifying the DUT

In the past, a lot of effort has gone into carving out logic to target with FPV. Nowadays, it is much more common to find standard or simple interfaces at unit boundaries. As a result, we likely select a DUT as the unit/IP/module level, so we can make use of any reusable validation collateral that accompanies standard interfaces. Still, with FPV, we have the ability to carve out any piece of logic and validate its functionality. This means that a designer can write and validate a simple IP/module, using FPV.

An important step at the start of the proof development stage is to determine the DUT that the user intends to carve out to prove the high-level verification goal. Ideally, the user works with a DUT that has the following attributes.

- Well understood input-output relationship;
- Documented specification; and
- Stable DUT boundary – interface not prone to change;

These aspects of the DUT enable the user to capture the functional requirements of the DUT and develop a sound verification plan.

6.1.3 Preliminary Analysis of the DUT

The goal of preliminary analysis is to make sure that the formal tool is able to compile the DUT, run the proofs, and provide feedback in a reasonable amount of time. The preliminary analysis helps to ensure that the early proof development, also called proof wiggling, proceeds smoothly.

In formal proofs, the inputs to the DUT can take all possible values. In the initial phase of proof development, unconstrained behavior of the inputs to the DUT results in many failures whose root-cause is an illegal or impossible micro-architecture condition. The wiggling stage involves iteratively querying the FPV tool (running simple assertions), identifying illegal behaviors by debugging failures and adding necessary constraints to the proof environment. During the wiggling stage, it is important that the user is able to compile the RTL model, run the proof and produce failures in a reasonable amount of time. This allows the user to iterate through the early wiggling stage quickly to produce a robust proof environment. The wiggling stage in the verification process has to proceed at a fast pace. If the user spends too much time in the wiggling stage then the ROI of FPV can become questionable.

The user has to determine the right balance between choosing a DUT large enough to have well defined boundaries and the response time in which the FPV tool provides for the chosen DUT. Here are the general rules of thumb that help determine if the FPV tool feedback is quick enough. None of these rules need to be explicitly adhered to, but they are intended to give the user a rough idea how to tell when their DUT is properly sized.

- The compilation time is less than 5 minutes
- The length of time before failures at low bounds (range 20 to 40) is reported in less than $\frac{1}{2}$ hour
- Less than 60,000 state elements (flops/latches)

6.1.4 Decomposition Strategy

The high level verification goal should be clear at the start of the feasibility analysis. The implementation of micro-architectural flows that need to be analyzed to complete the verification task may be encompassed within a single design block. If this design unit is small enough to be digested by the FPV tool then there is no decomposition strategy necessary. The design unit becomes the DUT. However, if the logic cone that implements the feature (also called guaranteeing logic) is contained in several units or a large design block that cannot be easily handled by the FPV tool then we need to come up with a strategy to divide the large design block in smaller sub-blocks.

The high-level verification goal for the large block is decomposed into separate smaller verification goals for each of the sub-blocks, a divide and conquer strategy. We often rely on this strategy to validate large and complex design blocks that cannot be handled by the tool because of capacity limitation. The division of a large design block into smaller sub-blocks should be such that when we put together the separate verification plans for each of the sub-blocks, they imply the high-level verification goal with which we started. The decomposition strategy is sometimes obvious from implementation of the micro-architecture flows, especially if the implementation of the feature is naturally divided into multiple design blocks. In some cases, the decomposition is not straightforward, especially when the guaranteeing logic cannot be separated into different logical blocks.

It is important to take steps to convince ourselves of the soundness of the decomposition strategy. We typically analyze our strategy via pen-and-paper analysis, which is prone to error just like any other manual process. Hence, it is important to review the decomposition strategy with architects and other formal verification experts to catch errors or holes as early as possible. We can also check interface assumptions supporting our decomposition strategy by connecting them to the simulation environment.

6.1.5 Effort Estimate

It is difficult to come up with an effort estimate for developing FPV proofs because completion of FPV proofs is a binary outcome. Either the proofs are passing or they are failing. If they are passing, the work is done pending quality checks. If the proofs fail it is difficult to determine many failures the user has to wade through before the proof is complete. However, the user should be able to develop a rough estimate based on

- Scope of the verification goal;
- Complexity of the micro-architectural flows under analysis;
- Decomposition strategy adopted; and
- Preliminary analysis previously performed.

6.2 Verification Plan

The Verification plan is a document that captures statements of correctness the DUT is expected to meet. In other words, it captures what it means for the DUT to implement the intended functionality correctly.

6.2.1 Verification Strategy

What it means for a given DUT to be implemented correctly can be answered a number of different ways. Depending on how the question is answered, it leads to different verification strategies. It can be answered as a list of correctness requirements (or properties) which the DUT has to satisfy. The DUT is said to be implemented correctly if it satisfies the list of properties for all input combinations to the DUT. Sometimes the correctness requirement of a given DUT can be captured in the form of a finite state machine (FSM). In such instances, the correctness requirement can be

coded as a reference design. The verification strategy in this instance is to ensure the RTL implementation satisfies the specification coded as an abstract reference design. In general, the logic behavior dictates the verification strategy.

Consider the example of validating the cache controller logic. The function of a cache in a microprocessor is to store data for quick access and thereby increase performance. In addition to the data, the cache may store attributes related to the data. The verification goal for the cache controller is to ensure, firstly, that the data is correctly written into the cache and secondly, that the data is read correctly from the cache. There are different verification strategies to validate such a cache.

- **Enumerated Properties:** All possible transitions of the cache attributes are captured in the form of a table or as a list. Each row in the table or line in the list is encoded as a formal assertion and the cache controller is validated when all the assertions are passing.
- **Reference design:** Build an abstract model of the cache. The abstract model contains only those attributes of the cache that need to be validated. Formal assertions ensure that attributes in the abstract model are the same as those in the RTL model for all entries in the cache.

The high level verification plan discussed in [Feasibility Analysis](#) talks about the overall objective of the FPV activity. The actual verification plan adds details on how this objective is achieved. The verification plan should consider the following factors:

- The level of verification confidence that is required (e.g. full or bounded proof). In most cases, a bounded proof will provide a sufficiently high degree of confidence in the correctness of the design. The bound indicates the depth (in clock phases) our verification reached without violating the property. We are confident in our bound when we can see interesting events occurring within that bound.
- Collaboration with other validation teams. This is to avoid duplicate work and more importantly, to ensure that there is no validation hole. It also establishes a communication channel with other validation teams.
- Decomposition strategy
- Verification Strategy
- RTL development Schedule. It should provide important milestones that provide the design and validation teams feedback on the health of the RTL model through the course of RTL development.

Development of the verification plan is the most important step in the FPV methodology as it lays the foundation for all verification that happens later. An incorrectly laid plan can lead to missed bugs and verification overhead down the road. Hence, it is important to spend quality time in developing a robust verification plan.

The verification plan is usually no more than a few pages. If it takes the user dozens of pages to describe the correctness requirements of the DUT, then the verification plan is getting complex and prone to errors. Let all the complexity reside in the RTL implementation. The user should be able to abstract away the implementation details and relate to the input-output relationship of the DUT to create a sound verification plan.

Unfortunately developing a good verification plan is more an art form than an established methodology. Why is developing a good verification plan difficult? It is difficult because, as mentioned earlier, the DUT boundaries and decomposition strategies require careful consideration of how the verification plans for each DUT will fit together. Identifying the verification plan for each DUT is like solving a jigsaw puzzle; it may require some amount of ingenuity from the user. If the verification plans of the different blocks do not fit together well then there are chances of bug escapes. We can gain confidence in our verification plan by reviewing with design and architecture.

6.2.2 Review of verification plan

As mentioned earlier, development of the verification plan is the most important step in the FPV methodology. Hence, it is important to review the plan as early as possible. There are two reviews that should be done.

- Self Review: The user studies all the bugs found in the area in the current or previous projects and convinces oneself that the verification plan would have captured the previous flaws. Also, inspired from these previous bugs the user should be able to conjure new failure scenarios and evaluate how the verification plan would catch the imagined bugs.
- External Review: The RTL coders and architects may have some assumptions built into their notions of how the feature works, why it works, and when it may not work. Leveraging their knowledge to review the verification plan is crucial. It should be easy for both architects and RTL coders to read the verification plan to understand the intent and provide a quick feedback.

6.2.3 Dynamic Plan

The verification plan is dynamic in that it may need to change through the course of the project. It needs to be revised based on the feedback received during the verification process (as shown in [Fig 1](#)) because developing the verification plan is not intuitive and relies on Validator ingenuity. There is potential for the user to overlook some interactions in the DUT. Hence, the user should constantly look for feedback from several different sources:

- The failure traces during proof development reveal many possible behaviors. The user has to be vigilant and understand how the verification plan would account for the behaviors or interactions exposed in the failures.
- The user should be aware of the RTL bugs from previous generation projects in the same and related DUTs. These bugs may provide sanity checks on the verification plan to understand if and how the same bugs would be caught on the current project. The bug description also provides insight into complex areas in the design that are prone to bugs.
- The user should monitor bugs found in the same or related area by other validation methodologies. Analysis of such bugs not only act as a sanity check on the verification plan, but also exposes cases left uncovered by the plan.
- The user should keep up to date on any late feature or specification changes and update the plan accordingly.

6.3 FV environment development

The main goal of developing the FV environment is to apply constraints on the FV model and prevent illegal behaviors. As Intel increases its use of standard interfaces (IDI/AXI/IOSF), an opportunity arises for a simplified FV environment. Validation collateral is developed hand in hand with standard interfaces to check specification compliance. If our DUT includes one of these standard interfaces, we can quickly instantiate the pre-existing collateral to create the formal environment.

The following are the most common tasks performed while developing the environment:

- Create a good initialization sequence to initialize the FV model. A good initial state prevents illegal initial states from causing failures. However, it must also be random enough that it is easy for the tool to reach states deep in the micro-architectural flows within a small bound.
- Constrain inputs to the DUT to prevent illegal behaviors in the FV model without excluding legal behaviors. This should largely eliminate false failures while implementing the verification plan.
- Abstract the behaviors of surrounding logic. This also helps in dealing with complexity issues. This aspect of environment development usually involves abstract modeling of surrounding logic.

A well-developed environment should mostly be independent of the actual design implementation of the DUT, because the environment resides on the boundary of the DUT. It references inputs to the DUT, restricts the behavior of these inputs to legal cases, and abstracts the logic surrounding the DUT. Such an environment should enable validation of any feature or logical entity within the DUT.

FV environment development tends to be an iterative process. To begin, certain representative assertions (part of the verification plan) are chosen. These assertions check against the RTL implementation. In this early stage, the assertions fail mostly because the FV model is not properly initialized or because of illegal inputs to the DUT. We start with a DUT that has unrestricted freedom of inputs and through the generation of failures, we gradually restrict the inputs to valid micro-architecture behaviors by running the proofs, debugging the failures, and then adding restrictions to the environment. The development of the environment runs from the beginning of the proof development stage all the way to proof maintenance mode.

6.3.1 Restricted Environment

When the proof development work starts from scratch, all inputs to the DUT are completely unrestricted and the initial state of the DUT is random. Unconstrained inputs and a random initial state lead to many false failures or illegal micro-architecture states. The root cause of these early failures are fairly trivial, such as, clocks not toggling, reset sequence not flushing out X values, clocks not enabled, Array-Freeze, Iscan, DFT logic corrupting the micro-architecture states, etc. Even though the root-cause of these failures is trivial, they are difficult to debug because there is noise in early failures. The term noise refers to signal transitions that are not contributing to the failure. The noise in the early failure traces could be because:

- The early FV model is unconstrained, i.e. all inputs to the DUT are free to take any possible value. It is difficult to figure out what combination of illegal inputs is the root cause of the current failure. There could be more than one illegal input causing the failure.
- The initial state of the FV model is random. This initial state contains lots of illegal micro-architecture states. It is difficult to weed out the ones causing the current failure

It is common that when the user starts to build the FV environment from scratch she is also learning the micro-architecture, making initial failure debug that much more difficult.

At this stage, the user has to go through careful iterative steps of eliminating these false failures and start building the FV environment. A heavily restricted environment helps the user navigate through all the noise and make quick forward progress. Restricting the environment involves constraining the inputs of the DUT to a known, small subset of all possible legal behaviors. For example, this could imply

- All clocks are powered.
- Disable all DFT, Array-Freeze, Iscan, defeature logic.
- Disallow certain inputs to the DUT by tying relevant input signal to the DUT to a fixed value. For example, we may disallow all snoops by tying the input snoop signal to 0.
- Restricting the inputs of the DUT to allow a small subset of allowed legal behaviors. For example, allowing only one type of request to come into the DUT, even though the default behavior may allow several dozens of different requests.

Starting with a restricted environment allows the user to look at simple failures that have less noise. The user is able to debug them quickly since she knows that the root cause of the failure is limited to a small subset of input conditions. This removes the distraction of having to consider all possible behaviors to root-cause the failure. Moreover, this also helps in learning the RTL by studying each aspect of the implementation, one at a time.

Once the user has passing properties, restrictions can be removed one by one to continue building the environment. Here the focus is developing a robust proof environment by slowly allowing more behaviors in the FV model.

After the FV model begins to produce nontrivial failures, it is time to add more assertions to the proof run. At this stage, the user continues to add necessary restrictions to the proof environment, but the focus starts to shift to finding RTL bugs. The FV environment continues to develop through the life of the project.

6.3.2 RTL Abstraction

When the FPV DUT is a control-heavy piece of logic carved out from a bigger RTL DUT, we are effectively creating a new interface. We must develop the FV environment on this new interface, properly abstracting the behavior of the surrounding logic. Depending on the complexity of the interaction between our new DUT and the surrounding logic, this abstraction can take several forms. In most cases, the surrounding logic can be abstracted by a collection of assumptions. In cases where the interaction between the DUT and surrounding logic is complex, an abstracted reference model of the surrounding logic may be required.

So when should we create a reference model and when would just a collection of assumptions be sufficient? The failure scenarios exposed during the verification process guide the creation of an RTL abstraction of the surrounding logic, because it is almost impossible to know what collection of constraints is sufficient for a given DUT. The necessary constraints depend not only on the interaction between the DUT and surrounding logic, but also on the target specifications. If adding constraints to the model requires a lot of bookkeeping (for instance, to keep track of what is going into the DUT and what is being sent out) it is best to develop a reference model that captures this abstraction in a simple way. In situations that do not require a lot of bookkeeping and information tracking from past events, a collection of constraints is often sufficient. In the early stages of developing the FV environment, it is good to start with simple constraints on the inputs, which can then evolve into an abstracted reference model of the surrounding logic at later stages if required. One additional advantage of property-based assumptions, rather than reference models, is that they are much more orthogonal (Changing one property doesn't affect the others).

6.4 RTL Verification

RTL Verification is the step in the FPV methodology where we validate the DUT. The correctness requirements identified in the verification plan transform into assertions, and depending on the strategy in the verification plan, may involve development of a significant amount of modeling code. The formal tool checks the assertions. If there are failures, these could be because of

- Illegal behavior. If illegal behavior is because of illegal input to the DUT, then update the environment with an appropriate assumption. If the illegal behavior is because of illegal initial state, then update the initialization sequence.
- The formal specification is wrong or is incomplete, in which case we correct it.
- If the failure exposes a design flaw or bug, we correct the RTL implementation.

This iterative process continues until the proof passes.

6.4.1 Coding Specifications

The verification plan captures the intent and the strategy to achieve the verification goal. The user codes the verification plan into assertions that the formal tool understands. The assertions capture the functional correctness requirements as documented in the verification plan. Since the assertions describe correctness of RTL behavior, their definition requires sampling of RTL signals.

Sampling RTL signals embedded deep in the implementation to create the assertions would defeat the purpose of verifying the RTL implementation, because if RTL signals sampled are incorrectly implemented the assertion will fail to capture these bugs. Ideally, we sample the inputs or outputs of the DUT and use them to create the required instrumentation logic to capture the verification intent documented in the verification plan. The instrumentation logic created by the user should be as abstract as possible. Adding too many details into it will create a complex code – thus making the validation code prone to bugs.

The implementation of instrumentation logic should be as removed from the RTL implementation as possible. Try to avoid the temptation to cut and paste code from the RTL. When sampling RTL signals to create the instrumentation code, make sure that either the sampled RTL signals are on

the DUT boundary, or the signals are not closely related to the behaviors that are being validated. In some cases, efficiency and expediency may require the user to use RTL signals embedded deep in the DUT while developing the assertions. These areas should be well documented and reviewed to ensure quality.

6.4.2 Dealing with RTL bugs in FV

Most of the time the root causes of failures reported by the tool are not RTL implementation flaws. If the root cause of a failure requires either updating the environment or changing the specification, it is usually straightforward, and since these files are maintained by the user, they can be updated easily. However, if the root cause of the failure is because of a genuine RTL bug, then resolving it can be more time consuming, as the design team owns the RTL and it may take a while to identify and implement the correct fix. Ideally, we would like to continue with the verification process and make forward progress. However, because of the nature of FPV, many assertions may run into the same failure and prevent forward progress in the verification process. In such scenarios, to enable forward progress we have to choose a workaround for a short duration of time until the RTL flaw has been resolved and the fix implemented. We do this by any of the following options

- Changing the FPV spec to avoid the failure scenario
- Make a false assumption to prevent the failure scenario from being reported.
- Add necessary restrictions to prevent the failure scenarios. For example, if one of the causes of the bug is ECC errors, then ECC errors can be disabled by setting appropriate input signals related to ECC errors to ‘0’
- Running on a local clone with an RTL fix

6.5 Reducing FV complexity

The feasibility analysis phase of the verification process provides feedback on the ability of the tool to digest the chosen DUT. During the course of implementing the verification plan, it is likely that the user runs into the dreaded FV complexity wall. We hit the FV complexity wall when the tool takes an inordinately long time to reach a reasonable bound or is unable to reach the required bound, because either the memory required to maintain the state space during verification becomes very large or the computation time required to calculate the next state becomes very large, or both. The likelihood of hitting the FV complexity wall increases as we continue to make forward progress since we continue to reach deeper bounds to find the next layer of failures. This is also exacerbated by the fact that we continue to add more complexity to the FV model in the form of additional instrumentation logic, assumptions and assertions.

There are several ways to deal with FV complexity. The following sections explore some of these techniques.

6.5.1.1 User Guided Abstraction

The RTL can reduce in complexity by restricting the reachable state space. One of the techniques is data abstraction where we reduce the size of logical structures (e.g. reducing the number of

buffer entries, or restricting data space to only a few values). Structural abstraction allows the tool to reach interesting micro-architecture points in the flow at much lower bounds than it otherwise could. For example, if there are several corner cases in the micro-architectural flows when the queues are full, then reducing the size of the buffers from 40 to 4 allows the tool to explore behaviors around a full queue condition at a lower bound.

Since RTL models change frequently, compiling an abstracted FV model from the RTL must be automatic. It is a good idea to work with the RTL coders to parameterize the sizes of various buffers and arrays. The reduced sizes of the buffers should be nested in an *ifdef FPV_RESTRICT* block in the RTL, allowing the FPV user to take advantage of the parameterized values to work with an abstracted RTL model.

FPV tools also support User Guided Abstraction through by providing methods to carve out logic and perform targeted initialization. These methods allow the user to add restrictions or abstraction to the FV model. There are also advances in FPV tools that carry out automatic abstraction simplifications.

The aforementioned FV model abstractions cannot be checked or verified in the simulation environment. They require review with the designer, other Validators and architects. Hence, it is important to keep the abstraction set as minimal as possible.

6.5.1.2 Preloading

In a DUT with high complexity or extended flows, it can be prudent to employ a method called preloading or initial value abstraction. In this method, we avoid resetting certain structures in the DUT so we can begin the verification in an arbitrary state. We then constrain the environment to allow the structures to begin the verification in any legal state. For example, in a design with several flows that must happen in sequence, we can preload the state such that we begin verification as if some of the flows have already happened, enabling the tool to reach much deeper bounds than if we started all structures in the reset state.

6.5.1.3 Semi-Formal Verification

During proof runs, the FPV tool starts checking the specification starting from an initial state that is an arbitrary legal state. This initial state is derived based on the initialization sequence provided by the user. The tool capacity limitations may prevent us from reaching reasonable bounds. One way to reach deeper bounds at the cost of not checking all possible reachable states is to use semi-Formal.

The semi-formal flow may allow analysis at a deeper bounds than those reachable via true formal analysis. The semi-formal flow starts just like a normal proof run: starting the search from the initial state derived by using the initialization sequence provided by the user. The flow relies on the existence of a set of user provided cover points, or logical expressions, that represent micro-architectural events. The flow then uses a sequence of cover points to guide verification along specific paths. After it reaches the first cover point provided in the sequence during the normal proof run it stops. It then creates a new initial state that satisfies the first-cover point and starts a new proof run using this new initial state, continuing the state-space exploration until it reaches the second cover point provided in the sequence. This process repeats until it traverses the entire

sequence of cover points provided to the flow or until the tool finds a state that violates the specification.

With semi-formal verification, it is possible to guide the tool along a path that is an interesting sequence of events in the micro-architectural flow. It may not have been possible to reach these deep states during a normal proof run. For example, if the DUT being validated has a long queue, it may require the tool to reach a very deep bound to explore behaviors after the queue is full. If capacity limitations prevent the tool from exploring these deep states, then semi-formal verification can specify a sequence of interesting states guiding it to the queue-full state and then allowing it to explore more behavior starting from a full queue.

In the past, Semi-Formal has been a flow that required a lot of manual intervention, involving analysis of properties and identifying interesting cover points to use in the deep state-space searching. In our current formal tool, this approach is largely automated.

6.6 Quality Checks

A passing proof does not necessarily mean that the DUT satisfies the property under all possible input scenarios. Proof development is a manual effort, prone to human errors. It involves the rigorous process of debugging many failures, adding constraints to the proof environment, constructing an abstract model of behavior, writing assertions to validate the DUT and guiding the process toward the final verification goal. During this rigorous manual effort, it is inevitable that errors creep in. It is crucial that we have hooks in place to catch these errors or proof limitations as quickly as possible.

6.6.1 Assumption Checking:

Validation quality is only as good as the FV environment created and the formal assertions verified. If the environment is excessively restrictive, it may prevent certain legal behaviors from being checked, resulting in false positives. Hence, validating the environment is an important step in the FPV methodology. Usually we convert restrictions into checkers in the dynamic simulation environment or in the emulation environment; and review the environment with designers, Validators and architects.

Ideally, we want to find FV environment errors as early in the validation process as possible. Finding errors late in the project can lead to finding RTL bugs late in the design, and may have the effect of lengthening the design cycle. The other bad effect of finding errors late is that it may lead to rework in the environment. The assumptions in the FV environment fit together as a part of a puzzle preventing illegal inputs to the DUT. Once we displace one of the assumption pieces, it may require significant rework in other pieces to make them all fit together again. The best way to avoid late rework is to check the assumptions in the simulation environment even as we create them. One way to do this is to store a collection of simulation traces and check any new assumption added against them, giving the user early feedback on the correctness of the assumption. Additionally, a more exhaustive check of all assumptions against a much larger collection of simulation tests provides stronger quality checking.

The following data is useful in determining the quality of the assumptions in the FV environment:

- All the assumptions in the FV environment have triggered in the simulation runs. For example, if there is an assumption that says that the request coming into the DUT should have a valid request type associated with it then it is important to see that simulation traces have the request coming to the DUT.
- Analyze the number of times the assumptions trigger in the simulation runs. There should be sufficient coverage of the assumption in the simulation traces so that it flags any invalid assumptions in the simulation traces. In the example above, we expect to see hundreds of instances of this request coming into the DUT, so that we have sufficient confidence that we have captured all valid request types in the assumption.

6.6.2 Cover Points

Cover points are logical expressions or sequences, containing one or more RTL or validation instrumentation signals. Validators write cover points on interesting signals or microarchitectural events. They provide feedback to the user as to whether the cover point is in the reachable state space of the proof run. A micro-architectural event may be absent from the reachable state space for a number of reasons.

- A micro-architectural event may require a long execution trace and the tool may not be able reach that state before hitting the complexity wall;
- The assumptions in the proof environment are too restrictive or incorrect, preventing some legal behaviors from being explored by the tool;
- Two assumptions conflict, resulting in a vacuous proof; or
- There is a bug in the RTL or the micro-architecture that make a certain state not reachable

In all these cases, cover points provide feedback to the user as to whether a certain micro-architecture event is covered by the proof or not. If the cover point cannot be reached, then the user must analyze why that particular cover point is not in the state space being explored. If it is due to an error in the FV environment, it can be fixed. However, if the cover point is very deep and the tool is not able to reach it, it shows the limitation of the proof. The user can then take remedial actions, as discussed previously.

A representative suite of cover points ensures you are not silently overconstraining your design. Some areas to have cover points include: all FSM states, all types of requests, writes and reads to caches, all expected outputs from DUT.

6.6.3 Trigger and Vacuity Checks

Trigger and Vacuity checks are similar to cover points in that they provide feedback on the coverage of the state-space explored. However, the formal tool automatically does these without the user having to provide explicit cover points. The trigger of a property indicates what event causes the property to be checked. For example, take a commonly used property: “A implies B.” This means that the property is checked whenever ‘A’ is true. In this case, occurrence of ‘A’ is the trigger. The formal tool reports the triggering event in a similar way to how it reports coverage of a cover point.

Trigger check information for an assertion provides a valuable piece of information. It tells the user at what bound, or clock tick, the property was covered. If the property is never triggered, this means that proof is passing vacuously (the property is trivially true). If the property is triggered, the range from the bound at which the property is triggered and the final bound of the proof is when the “real” checking of the proof is done. Based on the micro-architectural understanding the user can determine if the final bound is a reasonable bound to give a good confidence on the correctness of design with respect to the property. It is also possible to write “A implies B” in other forms. We can write it as a property without a trigger: “not A or B”. In this case, if A were never to happen, this property will trivially pass, with no indication to the user that the property is vacuous. For this reason, it is preferable to use trigger-based properties whenever possible.

6.7 Proof Review

Reviewing the verification plan and the implementation of the verification plan is necessary. The objective of the proof review is to catch human errors as early as possible. Errors may have crept into the verification process during any of the previous steps mentioned. Catching them early mitigates the risk to the project and enables corrective actions. The reviews are a means of communication to all the stakeholders. Reviews establish what proofs do and do not cover. They also identify additional verification goals, concerns and potentially increase of scope of the proofs.

6.8 Maintenance

RTL models change throughout the project for timing, circuit, or for functional reasons. Regular regressions, preferably on a weekly basis, depending on the amount of RTL churn, should continue through the rest of the project. It is a good idea for a subset of the proofs to be part of the design turnin-gating regression, especially in designs that depend predominantly on formal proofs to ensure functional correctness. Adding formal turnin-gating regressions provides several benefits including ensuring the FPV model build works, maintaining model ealth, and automatic checking of RTL embedded assertions.

7 Full IP/IU Validation by FPV

With FPV as the primary validation vehicle in some units, comes the responsibility to consider other related validation areas: x-propagation, upf, pgcb, power flows, dfx, and control register validation. We have developed formal methodologies for validating x-propagation and control registers. Many of the other areas can be validated with FPV but that is not always the best choice. So for any areas associated with the full validation of a unit we don’t tackle with formal, we need to have a plan to either have simple dynamic validation support to do the coverage, or uplevel to SoC.

8 Design Exercise using FPV

FPV can be a great choice to do early RTL exercise. In this usage model, it is not necessary to develop a complete FV environment. It is only necessary to achieve the steps to FPV Wiggling, as outlined in [Section 5.6](#), so a light, targeted environment may suffice.

Guidelines for successful early design exercise:

- Build the FV environment at well-defined interface or logical entities to keep it simple and stable;
- Develop a minimal FV environment by only adding constraints as needed;
- Develop a design exercise plan to help predict convergence;
- Leverage formal experts and reusable collateral to help develop FV environment; and
- Get to know your formal tool waveform viewing features to query your design.

Reasons to choose FPV over dynamic for design exercise:

- **FPV uses design languages:** The TE is a complex piece of software. Moreover, it is written in a specification language (Specman/SVTB) that is often unfamiliar to designers. Debug of dynamic traces requires a reasonable understanding of the TE to make efficient forward progress, especially if incoming features require many TE changes and global TE checkers. Lack of TE knowledge on the part of designers can be a significant detriment to exercising new features. Since we develop the FV environment using standard design languages, it is abstract, easy to understand, and localized to a single unit, it is easier to comprehend than the TE.
- **FPV can aid the designer in RTL coding:** Often times, bugs are coded into the design because it is humanly impossible to comprehend all possible scenarios for which certain micro-architecture condition is true or false. Bugs are created when the RTL code does not account for some of these scenarios. FPV can be a crucial tool in the hands of a designer to help him determine the behavior scenarios while coding RTL.
- **FPV is available before TE:** Using dynamic validation methodology is dependent on the availability of TE. It requires significant expertise to develop and usually is created by dedicated engineers within the validation team. Limited TE resources tend to become critical paths of the validation schedule, especially in a virtual team framework, where parallel feature development heavily stresses the limited TE resources. A designer choosing FPV can enable his own design exercise by developing and maintaining a minimal FV environment.
- **FPV design exercise simply queries the design instead of writing tests:** Even if the TE is available and designer has TE knowledge, using dynamic requires directed test writing skills. This is a difficult skill to master, especially if the logic being exercised has several asynchronous events and cross-feature interactions. FPV, on the other hand, involves the designer querying the RTL logic to determine whether it meets the desired specification. This is equivalent to the designer coding embedded assertions or writing coverpoints and is not an additional overhead.

- **FPV design exercise done at unit level:** Dynamic validation is usually done at a cluster level granularity. When new features are tested, we exercise the logic at a cluster granularity, even if the RTL changes are embedded deep within the cluster. Hence, cluster-level micro-architecture knowledge is crucial to using dynamic. Since FPV can be applied at a much smaller granularity, abstract understanding of the surrounding logic is sufficient.

9 Advantages of FPV over dynamic validation

In this section, we discuss the reasons a Validator might want to choose FPV over dynamic validation:

- **FPV is good for Targeted Validation:** FPV enables one to carve out and target the functionality of a specific logic cone and exhaustively analyze all behaviors. Carving out an arbitrary piece of logic presents some challenges discussed earlier in the chapter, but other validation methodologies do not afford this kind of flexibility and quality in early validation results. It is ideal for initial validation of new features and requires the minimal subset of micro-architecture knowledge. An FPV-based approach can guarantee nearly complete functionality within a significantly reduced timeframe compared to a strictly dynamic approach.
- **FPV leads to high quality validation:** By definition, FPV explores all possible behaviors of a given DUT. It does not distinguish between a trivial and a complex bug. As long as a Validator adds checks for all specification requirements, she can find most of the design flaws at early states of RTL implementation. The bug fixes and other RTL changes required for timing and circuit layout can be validated under all possible input scenarios leading to fast turnaround time and robust and healthy RTL at an early stage in the coding process, which is exactly the goal for which the virtual team model strives. FPV can improve initial design robustness and reduce cross-project inheritance issues by shifting the timeframe for discovery away from the tape-out deadline. The dynamic methodology does the onion layer peeling of finding the trivial bugs first and the more complex bugs later, causing RTL churn over the course of the project.
- **FPV Compile and Verification times are faster:** The debug – fix – compile - verify loop is significantly faster in FPV. This leads to many more FPV debug iterations in the time it would take for a single dynamic iteration.
- **Debug is more efficient in FPV:** Debug of failures is one of the most time consuming steps during validation.
 - **FPV failures are easier to root cause:** As early FPV uses a collection of local assertions similar to embedded assertions, it is easier to root cause failures. Dynamic validation mostly relies on global checkers that are implemented at a cluster level, hence, it takes substantial effort to determine the root cause.
 - **FPV failure traces are short:** FPV failure traces are typically of the order of 10 to 20 cycles. FPV usually gives the shortest path to a failure scenario. Dynamic failure traces often involve analyses spanning tens of thousands of clock cycles. The RTL

coder must trace events backward in time until the genuine error is uncovered, a process that traditionally indicates a direct correlation between debug productivity and unit/cluster-level architectural knowledge.

- **Implementing FPV verification plan is simple:** Implementing a dynamic validation testplan involves identifying important test conditions or micro-architecture events that have to be covered to provide confidence in the RTL health. Directed or randomized tests have to be written to hit most of these conditions and coverage must be analyzed to find out if there is sufficient coverage. All the three steps of identifying test conditions, test writing and coverage analysis require non-trivial effort. However, the FPV verification plan consists of functional correctness requirements that are well documented in feature HASes and MASes.

10 Summary

FPV has been widely acknowledged to be effective in finding extreme corner-case bugs. It has been successful in providing high confidence in complicated and hard to validate functional units. FPV has evolved over the last decade and has made significant inroads in influencing pre-silicon validation. It is now capable of significantly reducing traditional simulation based validation or completely replacing it. An FPV methodology is presented in this chapter as a standardized process. This approach can be a powerful tool in the hands of any validation engineer and can be applied to validate any functional block. Used correctly, it can lead to high quality RTL as measured by bugs present in the model, robust RTL as measured by the new bugs being introduced when the RTL undergoes changes and to reduced design cycle time. Moreover, FPV can be a useful tool in the hands of RTL when applied to early stages of RTL development. FPV has benefits in other domains not traditionally part of pre-silicon functional validation such as correct timing fixes, or aid in post-silicon debug analysis.

11 Future Work

11.1 FPV Coding Methodology

11.2 Porting Proofs from Previous Projects

- Things to Avoid
- Porting
 - Signal Remapping
 - Check Proofs in Simulation
 - Assumption checking
 - Assertion checking
 - Check Proofs in Formal Tool
 - Quality Checks
- Verification Plan and beyond

11.3 FPV in Mixed-Signal Validation

11.4 FPV in Post-silicon Debug Analysis

12 References

1. Verification of Pentium® 4 BUS Recycle Logic using Symbolic Simulation and Induction. Khurram Sajid and Roope Kaivola. DTTC 2003
2. High Level Formal Verification of Next-Generation Microprocessors. Tom Schubert DAC 2003.
3. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. Roope Kaivola et al. CAV 2009
4. FPV Bug Hunting – The Sandy Bridge Experience. Dan Carmi et al. DTTC 2008
5. Combining Formal and Dynamic Validation on the Hagensville Uncore. Flemming Andersen et al. DTTC 2009
6. Assertion Based Design and Validation. Alon Flasher, Arel Nathanson, Alon Gluska, Miriam Brusilovsky, DTTC 2007

The Art of Pre-Si Val: Chapter 24

Protocol Formal Verification

By: [Annette Upton](#)

1 Abstract

Intel sells parts because our engineers understand the designs, not because tests and proofs pass. Increasing the velocity at which the company releases new chips, then, requires increasing the velocity at which its engineers understand their designs. Protocol Formal Verification is one method by which to enable and speed this learning curve, pulling questions and concerns that otherwise might appear only in late stages of design and execution into the earlier stages of architecture and microarchitecture creation. Furthermore, codifying the understanding, the answers to the questions, and the resolutions to the concerns Protocol Formal Verification provides a communication medium by which to enable and speed communication among engineers participating in various stages of the design project.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/1/2011	Initial Draft	Annette Upton	Michael Bair / CCDO Val Staff
1.1	5/30/2016	Small updates	Annette Upton	Michael Bair

3 Contents

1 Abstract.....	525
2 Chapter Revision History	525
3 Contents.....	526
4 Purpose.....	528
4.1 Why do we need this chapter?.....	528
4.2 What does this chapter cover?	528
4.3 What does this chapter not cover?	528
5 Background Concepts.....	529
5.1 Abstraction.....	529
5.1.1 Structural Abstraction.....	529
5.1.2 Behavioral Abstraction.....	530
5.1.3 Data Abstraction	530
5.1.4 Temporal Abstraction.....	530
5.2 Model Checking	530
5.3 IO Equivalence	531
5.4 The Commutative Diagram	532
6 Protocol Formal Verification	533
6.1 The Purpose of Protocol Formal Verification	533
6.2 Good Protocol Formal Verification Targets	536
6.3 Types of Bugs.....	536
6.3.1 New flows	536
6.3.2 Ambiguous states	536
6.3.3 Fundamental limitations	537
6.3.4 Dangerous Race Conditions	537
6.4 Planning a Protocol Formal Verification Effort	537
6.4.1 The Objective of the Verification Effort.....	537
6.4.2 The Key Concepts and Flows	538
6.4.3 The Model Structure and Module Hierarchy	538
6.4.4 The Key State Elements	539
6.4.5 The Key Communication Channels	539
6.4.6 The Key Abstractions.....	539

6.4.7	Liveness Verification.....	540
6.5	Protocol Modeling and Verification Method	541
6.5.1	Understand the design.....	541
6.5.2	Create and Verify the Protocol FV Model.....	541
6.5.3	If in Doubt, Leave it out.....	542
6.5.4	Don't Panic Over Major Rewrites	542
6.5.5	Measuring Progress.....	542
6.6	Comparing the Protocol FV Model to the RTL	543
7	Summary.....	543
8	Future Work	543
9	References.....	544

4 Purpose

4.1 Why do we need this chapter?

Contrary to common belief, formal verification (FV) does not require a mature register transfer level (RTL) design in order to contribute to the quality of the finished part. In fact, it contributes significantly to the end quality of the design when applied before RTL coding begins, while the architecture and microarchitecture are in conceptual development. In fact, in its ideal state FV is an inception-to-fabrication design aid, rather than a validation-only activity.

In addition to setting the record straight on the value of Protocol Verification, this chapter details the foundational concepts and methods by which FV makes its greatest Protocol contribution. It considers the structures for which Protocol verification is best suited and enumerates the types of concerns that Protocol verification identifies. Further, the chapter discusses planning and executing a Protocol FV project. It then addresses the possibility of showing the relationship between the Protocol FV model and the RTL either formally or informally. Finally, it sums up and offers suggestions for future directions in which this work might grow to quicken further our validation pace.

4.2 What does this chapter cover?

This chapter treats the specifics of planning, developing and formally verifying a Protocol architectural or microarchitectural model. It begins by discussing the theoretical topics fundamental to the work, including abstraction, IO Equivalence and the commutative diagram. Then, it continues by outlining the particulars of performing Protocol verification in a project timeline, including refining selected portions of the architectural model with more concrete models and showing that the concretization is sound.

This chapter also covers creating a formal connection between the abstract, Protocol model and the more concrete RTL model, though lightly in this revision.

4.3 What does this chapter not cover?

This chapter does not consider formal logics, nor their various advantages or applications, nor any languages founded upon them. It does not engage in the holy wars of formal tool vs. formal tool. It does not examine the theory or the algorithms by which tools perform their proofs, nor does it consider the soundness of such algorithms.

Abstract model and verification methods also contribute to the sound development of standards and are independent of their realizations in particular implementations; however, this chapter foregoes discussing that application of Protocol verification, focusing, instead, on applying the technique to the microprocessor design cycle, specifically.

5 Background Concepts

A complete introduction to the formal verification concepts used in performing Protocol Verification requires a semester course, which is obviously beyond the scope of this chapter. This section, however, briefly introduces the central ideas, with the intent to make the reader conversant in the relevant fundamentals.

One of the most central concepts to all of formal verification is that of abstraction and verification engineers categorize the abstractions they make into four buckets: structural, behavioral, data, and temporal abstractions. As Protocol models mature the details that validators add to them need partitioning, otherwise the model grows too large to run feasibly. Section 5.2 introduces model checkers, the formal verification tool most commonly used to perform protocol FV. IO Equivalence, the third fundamental concept this section considers, makes that partitioning sound. Finally, in making any comparisons between the Protocol model and the RTL, we make use of the commutative diagram.

5.1 Abstraction

Abstraction is so important to Protocol Formal Verification that we sometimes call it abstract modeling and verification⁴⁰. The astute reader notices a distinct theme throughout this chapter.

At its most general, abstraction is simply the removal or hiding of irrelevant details, as taught in most early computer programming courses. Early work within the academic FV community considers abstraction along four axes, structural, behavioral, data, and temporal. While experienced FV practitioners rarely categorize the abstractions they use daily in these classes, they serve a valuable function to the novice learning to choose, make, and model hardware systems. This section discusses each of the four types of abstraction, briefly, as a reference point for the remainder of the chapter.

5.1.1 Structural Abstraction

Structural abstraction is generally the easiest of the four to understand. It relates to the omission of details of the circuit under consideration. For instance, rather than model each individual gate separately and connect them to form a circuit that performs a certain calculation, we might use the mathematical expression representing the circuit's calculation as defined by our modeling language. Another example of structural abstraction would be using a list data structure to represent a FIFO, rather than modeling the read, write pointers, the read, write enables, and the individual data slots.

⁴⁰ No one name completely summarizes the nature of Protocol FV. Others used in some circumstances include Pre-RTL FV, Abstract FV, high level modeling, but each one fails to capture some instance to which the techniques apply or some subtlety of performing Protocol FV. Practitioners use all these terms somewhat interchangeably, generally trying to fit the right term to the context.

5.1.2 Behavioral Abstraction

Behavioral abstraction, on the other hand, is generally the most difficult to understand as it typically occurs implicitly in any abstract modeling or verification effort. By choosing only certain parts of the system to model or certain behavior of the circuit to specify, the validator performs behavioral abstraction. For instance, we may choose to verify that an arithmetic unit adds correctly in a given proof. That arithmetic unit probably performs subtraction, multiplication, and division, as well, but we abstract those behaviors away by only specifying and proving the correctness of the addition behavior. In a subsequent proof, we might verify multiplication, eliminating addition, subtraction, and division via behavioral abstraction.

5.1.3 Data Abstraction

Data abstraction usually involves encoding information in some structured format, optimizing for human consumption, instead of writing out the bit stream or vector. For instance, we might represent a number as an integer rather than a 32-bit register or a data packet as a structure rather than a bit stream or a set of bus wires.

5.1.4 Temporal Abstraction

In temporal abstraction, we discretize the notion of time, so that the values of state elements are only visible at distinct moments. The distance between adjacent moments, or time steps, determines the amount of temporal abstraction at work. Larger distances between ticks make highly abstract models. In a temporally concrete representation, for instance, the verifier might detail each stage of a cache lookup pipeline separately, while in another, the entire cache lookup computation might take only a single time step. Representing a data packet sent at one agent and received at another makes a more abstract model than representing the packet's passage from sender to receiver, stopping at each router in between.

5.2 Model Checking

Conceptually, a model checker, the verification tool generally used in this sort of work, takes the place of a test harness. It provides the inputs to the design under test (DUT) and watches the outputs of the DUT. The main difference between a model checker and a test harness suitable for dynamic validation is that the model checker systematically presents the DUT with all possible inputs at every reachable state, whereas a given test presents the DUT with a single series of inputs.

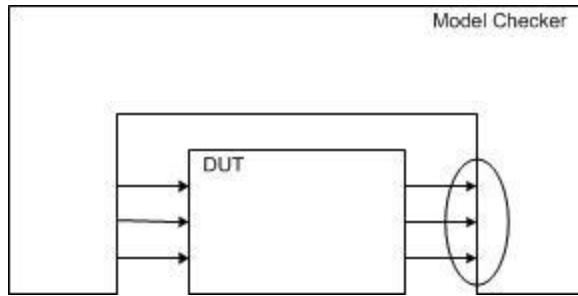


Figure 43 A model checker is a verification tool that acts like an exhaustive test harness.

Instead of installing checkers in the test harness to ensure test runs never violate important conditions or properties, the model checker explicitly checks that a given property holds at every execution state for all possible inputs. Ideally, verification engineers write their properties such that they only check the outputs of the DUT (as indicated in Figure 43), as dynamic validation engineers do when writing checkers for black box testing. Unfortunately, due to the exhaustive nature of the model checker it can generally only operate feasibly on relatively small portions of a design. As a result, it sometimes becomes necessary, to write white box properties, those that check signals internal to the DUT, rather than at its outputs.

Furthermore, as most DUTs verified using model checkers represent pieces carefully carved of a larger design, some inputs that the model checker wants to try may not be legal. For instance, the input signals may be mutually exclusive, but the model checker naively tries all input combinations, including those in which the signals are not mutually exclusive. Such illegal inputs often lead to property violations, called false negatives or false failures, which the model checker writes out for the FV engineer to study. To rectify false failures, the verification engineer constrains the inputs the model checker presents to the DUT.

FV engineers typically iterate through the constrain-model check-investigate failure loop many times in the course of verifying a single property. Though time consuming, each pass through the loop provides insight into the design’s functionality. Over the course of verifying several related properties, the FV engineer often becomes an expert on the DUT, having seen much of its behavior.

In addition to writing white box properties the exhaustive enumeration model checkers perform often require other control techniques. An adequate treatment of all the techniques used to control exhaustive enumeration problems is beyond the scope of this chapter. The engineering wizardry often associate with FV is all in controlling the exhaustive state space and doing so such that the resulting proof is sound.

5.3 IO Equivalence

The formal verification literature considers several correctness criteria or ways in which we might formally compare two systems or state machines and determine whether they are “the same” or not. Discussion of most of the available correctness criteria is beyond the scope of this chapter, but Protocol formal verification depends critically upon one criterion: input/output equivalence (IO equivalence). Simply stated, two views of a block are IO equivalent, if no environment can distinguish the two via IO signals alone.

In the example in Figure 44, the solid-line view of Block 1 is IO equivalent to the dashed-line, more detailed view of Block 1 if there is no way for Blocks 2 and 3 to distinguish between them based solely on their interfaces with either abstraction or Block 1. In classical IO Equivalence, we must show that there is absolutely no environment that both correctly implements the interfaces to Block 1 that distinguishes which implementation of it we have in place. For our purposes in this work, however, it is enough to show that the environment in which the block is designed to operate cannot distinguish between the two implementations under any circumstances.

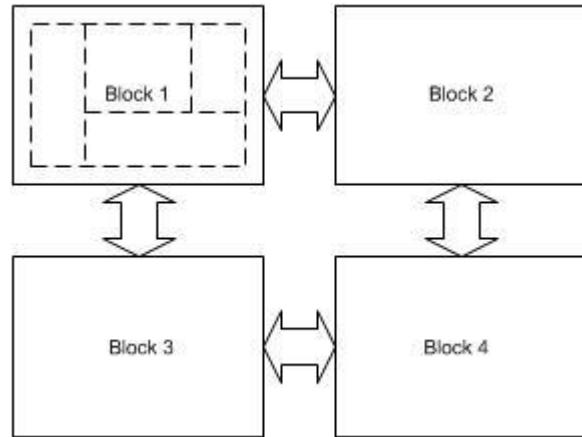


Figure 44 The environment cannot distinguish two IO Equivalent blocks via their interfaces alone.

This correctness criterion is typically somewhat counterintuitive to the evil validator who asks, “But what if the dashed Block 1 does some internal computation differently than the solid black Block 1 does it?” To which the answer is, “It doesn’t matter, as long as no block in the environment can tell the difference solely via IO.”

A full treatment of the various forms of IO Equivalence, the proof obligations that indicate systems meet them, and the subtle differences in their meanings are beyond the scope of this chapter. The interested reader is referred to Section 9, to learn more about correctness criteria and process algebras (the field that created the criteria). The Dill work and the Nowick work treat IO Equivalence directly, while Hennessy, Milner and van Glabbeek consider a full range of process algebra topics.

5.4 The Commutative Diagram

In formal verification, we often find ourselves making comparisons between two models, one abstract and one concrete. The terms “abstract” and “concrete” are only relative to one another when speaking in this context. The concrete model may be RTL or it may be some other model, perhaps more abstract than RTL. The only characteristic that matters is that it is more concrete than the abstract model. The commutative diagram is the basis for comparing abstract and concrete models.

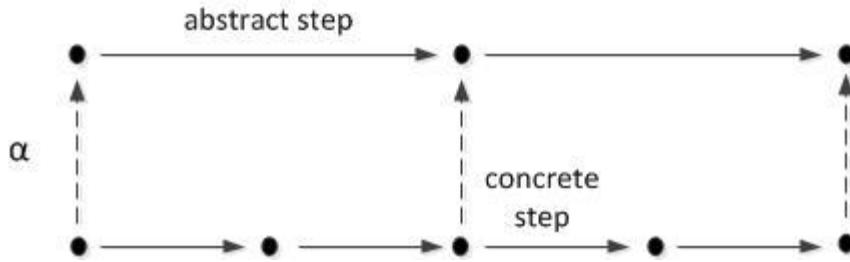


Figure 45 The commutative diagram shows that abstracting and then taking an abstract step produces the same result as taking some number of concrete steps and then abstracting.

Conceptually, the commutative diagram is simple, as shown in Figure 45. The α symbol in the diagram represents the abstraction function used to relate the two models. It may be arbitrarily complex and probably never actually finds its way to paper in any meaningful form, but, like all functions, when applied to a state of the concrete model (represented by dots in the lower execution trace) it must map to exactly one state in the abstract model (dots in the upper execution trace). It may take the concrete model several steps—likely more than the two shown in this commutative diagram, in fact—in order to keep up with all the computation that happens in a single step in the abstract model, but at some point, α must be able to map a concrete state to an abstract state again.

6 Protocol Formal Verification

Increasing the velocity at which Intel Corporation releases new chips requires increasing the velocity at which its engineers understand their designs. Protocol Formal Verification is one method by which to enable and speed this learning curve, pulling questions and concerns that otherwise might appear only in late stages of design and execution into the earlier stages of architecture and microarchitecture creation. This section considers first the purpose of performing Protocol Formal Verification, then the types of design elements that make good targets for this type of formal verification as well as the types of issues it tends to identify.

6.1 The Purpose of Protocol Formal Verification

The purpose of performing Protocol formal verification is to enable increased understanding of the architecture and microarchitecture of interest in the early stages of design. By using the model checker to explore the implications of design decisions, we find cases and interactions that humans—no matter how careful or experienced—cannot foresee. Repeated interaction between the architects and their design implications, as well as continuing analysis of “what if” scenarios, deepens their overall understanding of the design both at the macro level and at the micro level. The deepened and completed understanding, then, enables sound decisions, and safe issue resolutions and robust bug fixes throughout the design cycle.

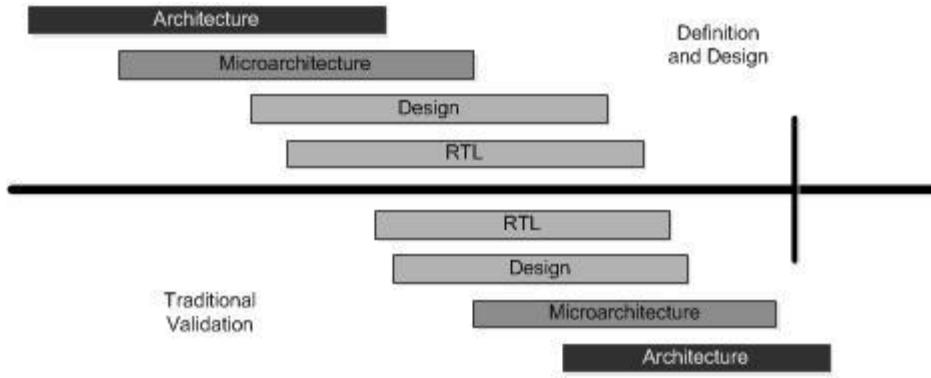


Figure 46 Dynamic simulation validation finds bugs in roughly the reverse order of their introduction into the design, possibly leaving the costliest bugs until late in the program.

A second, ancillary purpose of abstract modeling and verification is to aid the early stages of architecture and design in getting the big picture correct so that errors do not accumulate as the design progresses toward completion. Some think of the processes as depicted in the following two diagrams. Figure 46 shows how dynamic validation means identify architectural and microarchitectural concerns. First, an architectural mistake occurs, possibly due to an overlooked case, and then the microarchitecture perpetuates it into the design and thence to the RTL. The designer also may encounter difficulties in understanding the related details, hence potentially causing microarchitectural or design level problems nearby.

Dynamic validation finds the problems in the reverse order. First, we find the simple coding problems, like writing to the wrong control register address or shifting one too few bits over a serial interface. Only when the compilation and test get past the mundane problems, can we find design bugs such as a missing control register or a message sent over the wrong interface. Typically, validators and designers find and fix a significant number of design issues before many microarchitecture concerns surface. Unfortunately, exercising the far reaching, architectural problems often require many lower level details to work, which in turn can require a great deal of effort and many design iterations to achieve. Furthermore, the time and effort required to peel the onion down to the architectural concerns means they are generally the last to be addressed in the design cycle, when they are the most difficult to rectify due to the vastness of the disruption caused by such changes.

Figure 47 shows how Protocol formal verification can explore the design space and identify concerns in the order in which they occur. The FV engineer finds cases and states that the architects have not thought of by virtue of being the first engineer to implement the architecture. The model checker finds cases and orderings that no one thought of by virtue of being a model checker. FV engineers can raise and resolve such architectural concerns easily since they, the architects, designers, and other validators are all working and thinking in architectural context and since RTL implementation has not yet begun. Architectures can undergo massive revision, since relatively little effort has gone into them and unknown dependencies on the features in question cannot yet exist.

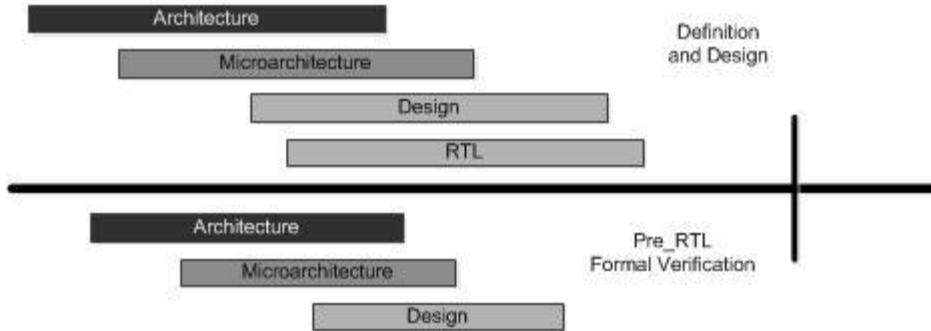


Figure 47 Protocol Formal Verification finds bugs in roughly the order of their introduction to the design, far-reaching bugs early in the program when designers can comprehend their implications easily.

The statement bears repetition: the amount of time and effort spent creating and verifying the FV model is small in comparison to the potential effort expended down an RTL-only development path. Certainly, major revisions in the architecture mean major revisions in the Protocol FV model. Section 6.5.4 considers the human implications of managing rewrites.

Once the architectural holes and concerns are resolved, the FV engineer can move on to microarchitectural concerns along with the architects and designers. By replacing architectural models of select blocks with more detailed microarchitectural blocks, and reverifying, we can determine whether a given microarchitecture implements the architecture it replaces or not. This method directly reflects the proof obligations established to show IO Equivalence between the architectural block and the microarchitectural version. Learning at this stage that two given state machines cannot provide the functionality required to satisfy the environment is much better than waiting a year to learn that. Also, while verifying the architectural block numerous issues in the architecture are sorted out and solidified, which makes the microarchitecture design task much simpler since those first-order concerns are no longer there to muddy the waters.

Similarly, having the microarchitectural issues properly dispatched, working through the final design and implementation is a far easier task than it might otherwise be. Furthermore, if we choose, we can use the Protocol FV model as a specification for the RTL and prove that the RTL implements the specification properly via more traditional FV techniques. Alternately, we can generate RTL assertions from the Protocol FV model that run in simulation to ensure that the transitions that occur in the Protocol FV model (the specification in this instance) also occur in the RTL model.

A number of things contribute to the high cost of correcting bugs found late in a project (see [Introduction to Pre-Silicon Validation section: Hardware change cost model](#)). One of the greatest contributors to bug cost is the amount of logic changed by the bug fix and the physical constraints on making those schematic and layout changes. As a result, any old bug can be expensive, but architectural bugs usually require substantial change and as a result are often costly.

6.2 Good Protocol Formal Verification Targets

In general, FV prefers structural locality and tight integration to distribution. That means that verifying the correctness of a single module is usually easier than verifying the correctness of some long flow that touches many parts of the machine, a memory read that looks up multiple levels of cache, for instance. While the same holds for Protocol verification, abstraction enables Protocol verification to spread out to cover distributed entities better than most other FV approaches.

Protocol verification produces best results when applied to systems of coordinating state machines. Communication and transaction processing systems, such as caching agents or other memory system components present ideal opportunities for Protocol verification to impact the design cycle. While applying Protocol verification to power management systems faces the challenge of distribution, we have also seen success in verifying power management in this fashion.

6.3 Types of Bugs

In this section, we discuss some of the types of discoveries we make when modeling and verifying protocol ideas in their infancy and during development. Modeling a protocol and having an FV tool execute it puts the protocol under a microscope. Using the FV tool, we are able to discover new flows to consider, identify incomplete definitions or states with ambiguous decisions, expose fundamental limitations of the ideas in satisfying the specifications, determine resource constraints of the protocol, highlight complex or lengthy algorithms, and find other manner of “bugs.” The following subsections present some examples.

6.3.1 New flows

Protocol verification can reveal the need for new flows or transactions. For example, the exhaustive search can expose combinations of states thought to be impossible. After architects analyze the flows, they update their legal state tables to include the newly identified state combinations, and provide input on how the cache hierarchy should handle the added flows for continued FV.

6.3.2 Ambiguous states

Because the model checker tries to follow all enabled paths, if two transitions exit a single state under the same conditions (making the choice between the two transition nondeterministic), the model checker tries both and exposes the results. Data buffer control analysis discovered two flows involving a buffer receiving a read request, where one flow required the buffer to deallocate and the other required it to wait for a write, yet neither the data buffer nor the read sender had sufficient information to disambiguate the cases. Analysis of the flows (and others related to them) pointed to a third agent that had the needed information and made an earlier request in all cases.

6.3.3 Fundamental limitations

Protocols involve cooperating state machines that pass information among them and make decisions using only partial state information about the system. The protocol developer strives to ensure the necessary information is passed for accurate decisions and that each state machine has available the state information it needs to make correct decisions in all cases. Essentially, the developer is striving to create invariants about how the agents work. Sometimes, these invariants are entirely unsound or they hold for only a majority of cases. Model checking the protocol exposes such limitations.

6.3.4 Dangerous Race Conditions

The exhaustive nature of model checking produces all possible orderings of next state transitions. If two or more transitions are enabled from a given state, the model checker attempts to make them in all possible orders, transition A followed by transition B and then transition B followed by A, for example. Unsafe event orderings cause the model checker to fail, providing a trace showing the event ordering as well as events leading to it. Unsafe event orderings can be constrained out of the model one at a time, allowing the verification engineer to see each separately to maximize understanding of both the unsafe orderings and the safe orderings, as required.

6.4 Planning a Protocol Formal Verification Effort

Given that Protocol Verification occurs in the early stages of architecture and design, only so much planning prior to beginning the effort is possible. Things change and they often change a lot early in the design cycle. Protocol FV work often begins before product definition solidifies, architects and management reject features for various reasons, and roadmaps dump some projects and sprout others. All of these changes affect the Protocol verification plan, causing changes both large and small in the Protocol Verification plan. Changes to the verification plan and model, however, constitute progress toward a final product (see Section 6.5.4).

That said, though, the FV engineer should consider a few things at project outset, if possible. The objective of the verification determines the approach taken and the concepts and flows coded. The concepts and flows included in the verification then determine which state elements and communication channels should be included. The state elements and communication channels, in turn, give rise to decisions surrounding required abstractions.

We discuss these considerations in order of the likelihood that adequate information exists at the beginning of a Protocol verification effort and then offer a final word of advice regarding Protocol planning in this section.

6.4.1 The Objective of the Verification Effort

The eventual goal of the Protocol FV effort influences every subsequent decision made in the modeling and verification work. Even in the absence of a clear objective at the outset of a project, considering the ultimate goal of the effort from time to time provides insight into the work, the design, and may save the effort of wandering down a red herring path for longer than necessary.

Questions to consider may include the following

- **What are you trying to verify?** Knowing exactly what the design under verification is directs your work to model and verify the right things. Verifying cache coherence, for example, does not need to include a model of the power control unit. Also, identifying the key characteristics of the design early, help guide abstraction choices.
- **At what level of abstraction will you model?** Not all blocks in a model need to contain equal detail. In addition to keeping stub blocks very abstract, you will likely find that certain portions of some blocks need more concretization than others need, as you work.
- **Will you replace abstract blocks with concrete blocks?** Think carefully about interfaces—just as you would in any other IP swapping effort—if concretizing some blocks of the model is on the agenda. Of course, your interfaces should model the design and ensuring design faithfulness aids in coding robust interfaces.
- **Will you eventually connect your Protocol verification model with RTL?** If you plan to connect the Protocol FV model to the RTL, you may need the most abstract model to be very abstract, or you may even need to specialize several different overlapping models, each concentrating on one portion of the design under verification, in order to concretize a particular block enough.
- **By what techniques might the abstract-to-RTL connection occur?** If the goal is to connect the Protocol FV model to the RTL somehow, coding style may be determined in part, by which verification method performs that connecting verification.

6.4.2 The Key Concepts and Flows

No protocol verification effort can model the entire chip. First, no model checker can handle that load between the inception of the chip and its end-of-product-life. Second, if you could model all production features in the FV model, there would be no need for the RTL.

The choice of concepts and flows to include in the Protocol model depends on the objective of the modeling and verification effort. If, for instance the purpose of the Protocol verification work is to ensure that a particular block is implementable, then, all transactions that flow through that particular block should be modeled. If, on the other hand, the verification target is something as broad and far-reaching as ensuring that power management flows do not deadlock the machine, a knowledge of which flows interact with the power management flows is required early in the effort. Furthermore, the verification engineer needs to know where and how the other flows interact with those of interest, as abstracting behaviors of the flows that do not create interesting interactions may be required to reduce the reachable state space of the model enough to enable the model checker to check the properties of interest.

6.4.3 The Model Structure and Module Hierarchy

Model structure, for the most part, follows the structural hierarchy of the RTL. Abstractions, of course, may occur, but in order to be faithful to the real design, some representation of the actual structure must be present in the FV model.

6.4.4 The Key State Elements

Though it may seem obvious, identifying the key state elements that should be included in the model is an important early step. Candidates for early consideration include the central state of any state machines and any data structures on which the state machines operate. Err on the side of including too little information in early versions of the state element models. You will always find out later, thanks to the model checker, if you need to add something. It is more difficult, however, to determine that you have something you do not need—the model checker does not help you make that determination. Furthermore, it may be that by not including it in the model you learn that some state element planned for the RTL is unnecessary.

6.4.5 The Key Communication Channels

The kinds and number of communication channels can vary widely, depending on what you model and verify. While the FV engineer might abstract the internal workings of the communication channels, abstractions that place messages on channels on which they do not travel in the design provide verification holes in which problems can easily hide.

6.4.6 The Key Abstractions

Any successful modeling and verification effort requires all four classes of abstraction, but some play larger roles than others in a given effort.

It is certain that the verification engineer employs behavioral abstraction, for the reasons discussed above, namely, that it is infeasible to model the entire chip in the Protocol model. If the possibility existed, it leaves the designers without much work.

The verification engineer will almost certainly use data abstraction in the early, abstract phases of the Protocol verification effort, but when concrete blocks replace abstract blocks as the work matures, bit vectors may replace integers and pipeline stages may replace atomic computations.

Structural abstraction plays a larger role in Protocol FV work that stretches across the chip than it does in work confined to a small set of blocks. While the work in the local structures omits certain portions of blocks, the broader work probably only includes certain portions of carefully chosen blocks.

Verification engineers rarely write cycle accurate models and in so doing they use temporal abstraction to their advantage almost without thought. Certainly, some portions of a model can perform more computation in a model timestep than other portions. In fact, temporally adjacent model time steps may not even perform the same amount of computation, and as a result be modeled at the same level of temporal abstraction.

The abstractions employed need not be symmetric in any sense. An FV engineer may need to raise one communication channel to a very abstract level, while another channel remains more concrete, for instance. The only way to know how those skewed abstractions should be handled and should (or should not) relate to one another is to understand the model, the design under verification, and the purpose of the work. If, in the big picture, another validation effort covers a state element or communication channel adequately, the likelihood that an abstraction is adequate in Protocol work

is high. If, on the other hand, the Protocol Verification effort focuses heavily on a set of state elements, they likely require concretization.

6.4.7 Liveness Verification

Liveness verification of a protocol focuses on ensuring that every transaction or request eventually completes. When verifying a protocol, the primary concern is checking that the protocol satisfies its purpose (e.g., a cache coherence protocol successfully maintains data consistency) and that its state machines and communication mechanism are complete (i.e., can handle all cases the protocol can reach) and unambiguous. Doing these safety proofs—proofs that guarantee that bad things never happen—is our first concern in protocol verification, and they account for 90%, or more, of the protocol verification effort.

Only after demonstrating that the protocol is sound do we consider liveness verification. One reason for doing liveness proofs only after completing safety proofs has to do with the difficulty of mechanically checking liveness properties: It is much harder for the tools than is verifying safety properties. Safety properties require only tracking all visited states. Liveness properties also require tracking paths between visited states and traversing the graphs.

Another reason for doing liveness verification after the protocol’s safety proof is because verifying that a system is fair requires knowing all of the states a system can reach. That is, the fairness of a system builds upon the state machines and communication mechanisms that create the entire reachable state space of the system. In FV vernacular, how the protocol fundamentally works is an assumption for the fairness mechanisms. Because fairness verification requires a complete understanding of the state machines and communication mechanisms to ensure its validity, we do the safety proof first to acquire that complete understanding.

Assuming the presence of a completed safety proof for a protocol, verifying the fairness of a protocol involves decomposing the problem into verifying that every state transition, or every set of state transitions, occurs fairly. (If some state transition was not fair then it would be possible for some request to reach a point at which that transition must occur yet does not, violating the fairness property.) What does it mean to verify that a state transition is fair? From the perspective of the protocol specification, that question is nonsensical. The specification simply says what transitions need to occur and it expects implementations to ensure those transitions occur fairly.

Talking about protocol liveness verification is a misnomer. What we are really talking about is verifying an implementation’s fairness of the protocol’s transitions. For example, suppose a cache-coherence protocol specification requires a cache receiving a snoop to process the snoop and update the cache according to the transitions described in a table. From a liveness perspective, what we need to verify for that transition is that an implementation receiving a snoop eventually processes it and updates its cache accordingly. We need to verify that the microarchitecture’s protocols are fair for all of the cache-coherence protocol transitions.

In short, liveness verification falls mostly in the domain of microarchitectural protocol verification. We first model the microarchitecture to verify it supports the architectural protocols, the safety proofs, and then we decompose down to the microarchitecture’s state transitions and verify they are fair with respect to the architectural specification’s transitions. For those transitions where it is not obvious that they occur fairly, we write targeted models at the implementation of those transitions (usually involving schedulers and arbiters and the few state elements they make

decisions upon) and verify fairness within them. The models we write for liveness use the same techniques described throughout this chapter but also include a fairness property that states that some transition must eventually occur, and a number of fairness assumptions guaranteeing that the outside world eventually returns acknowledgements.

6.5 Protocol Modeling and Verification Method

While planning is great, even experienced engineers can let a blank text editor screen intimidate them into over planning. Experience suggests that choosing a place to start is largely an arbitrary decision. Unless a particular feature or flow needs to take priority, starting somewhere is more important than starting in the “right” place. This section offers some advice regarding actually creating the Protocol FV model.

6.5.1 Understand the design

“Understand the design,” sounds obvious, but it is easy to lose sight of the real intent of the design under verification. Accurately reflecting the architects’ ideas as well as any existing design elements is a constant Protocol Verification challenge. Asking, “What really happens in the design?” can help the FV engineer stay focused on writing faithful Protocol Verification models.

The only differences between your model and the functionality going into the RTL should be your abstractions. Thinking back to the commutative diagram, the Protocol model represents the abstract model, while the RTL represents the concrete model. The abstraction function relates the two models at every observable step. Concentrating on making the commutative diagram map as directly and literally to the Protocol Verification model creation can also aid the verifier in creating the right model.

6.5.2 Create and Verify the Protocol FV Model

Protocol FV practitioners have devised a modeling method that relies on model checker results to introduce features in an ordered manner:

1. Determine a feature-introduction order. Because the goal is to model all the important features, the order chosen may be somewhat arbitrary; however, introducing features that are most relevant to the verification objective first provides the best means for learning and scrutinizing the implementation.
2. Add the first feature and its initial state definition to model.
3. Run the model checker.
4. When the model checker fails with incomplete model or other issue, expand model or resolve the concern to handle failing state and return to step 3.
5. When the verification passes, implying model is complete for feature, disable it, add the next feature on list, and return to step 3. When two or more features pass model checking in isolation, then enable them simultaneously and return to step 3.

6.5.3 If in Doubt, Leave it out

There is no way to know everything needed before beginning the modeling part of the work. You will leave stuff out and need to go back and insert it later. You will wonder early on if this or that is needed. When in doubt regarding its necessity to the work, do not create a given state element or communication channel or even flow fragment or concept. If you do actually need any of those things in order to complete the work, the model checker will show you. It will show you that your model is incomplete and then you can determine whether that is a shortcoming in your abstraction or in the original concept.

This advice applies to experienced FV engineers as well as first-time verifiers. Even when written by experienced engineers, the model progresses more quickly when model checking results drive all model changes and additions. Getting fancy or overly aggressive causes more problems than it solves in the end.

6.5.4 Don't Panic Over Major Rewrites

Successful completion of any Protocol FV project, or indeed any FV project, requires multiple rewrites. Verification engineers rewrite their environmental modeling. They rewrite the module under verification. They fix bugs. Some of these rewrites are small and simple, but often the rework required to get back on track is extensive.

The first few rewrite situations, especially when the required rewrite is extensive, generally cause new FV engineers some distress. They often induce even experienced FV engineers to utter a few choice words. Though easier said than done, remembering that needing a rewrite represents progress toward the end goal makes a better choice than succumbing to frustration. Finding the necessity to make a major change is good on several fronts:

1. The realization represents a significant step in your understanding of the design under verification, its surrounding environment, and the implications of the design decisions made in both. Understanding the design, after all, is the point of doing Protocol verification.
2. The changes make your model more faithful to the design the architects have in mind, which strengthens the verification results; and
3. With each successive major change, the FV engineer usually finds the ability to back up, make the changes, and arrive at the level of model maturity at which you realized you needed to make the changes substantially faster. Within a few iterations rewriting the code that originally consumed a couple of weeks to generate takes only a few hours to rework. The shortened rework cycle phenomenon indicates deepened understanding, and progress toward the fixed point representing the ultimate goal as well.

6.5.5 Measuring Progress

Given the fact that Protocol FV happens early in the design cycle and that it can require many rewrites, measuring progress is a challenging prospect. Ultimately, the purpose of Protocol FV is to aid in understanding the architecture and design in as much depth as possible as early as possible in the design cycle. That very fact makes measuring progress challenging, something with which management is never comfortable.

Some possible metrics:

- The number of flows or features completed vs. the number committed;
- The number of issues, concerns, bugs found;
- The number of questions asked as a result of verification work;
- The number of questions answered;
- The amount of time since the last major revision;
- The amount of time the most recent major revision required;
- The number of concepts or flows involved in the most recent revision; and
- The reason for the most recent revision.

6.6 Comparing the Protocol FV Model to the RTL

Ideally, the Protocol FV model and the ensuing RTL coincide in some meaningful way at the end of the project. In future versions of this chapter, it will consider several methods for comparing the Protocol FV model with RTL.

7 Summary

When used as an end-to-end architecture and design aid, Protocol FV can help architecture and design understanding and convergence, which in turn enables robust, clean repository code. It explores design space and raises questions early in the design cycle when engineers can address them most easily. Protocol FV can find cases in which new flows need to be created, states in which the available information does not fully determine the next computation, fundamental limitations in design approaches, and problematic race conditions. Using a model checker-driven approach ensures that the FV model contains only necessary elements, focusing on the fundamentals that ensure a shortened design pathway to a correct microprocessor.

8 Future Work

Protocol formal verification is maturing as a methodology, and yet there are still many opportunities for further exploratory work.

First, evaluating the possible value of checking the Protocol Verification models into the RTL repository so that everyone—not just the FV wizard has access to them. If the FV languages limit the value of broad distribution, perhaps that is impetus to consider different languages or broader training in FV languages.

Second, TuRTL, a tool that generates simulation checkers from the FV models should continue development.

Third, once the Protocol FV model checks into the RTL repository, then TuRTL generates the connection checkers as a part of the model build, ensuring that other validators are always running checkers current to the RTL under inspection and forcing FV engineers to be accountable to the

overall RTL effort and staging plan. Again, value of having checkers always running should be evaluated carefully.

Fourth, much academic research centers on the use of various specification languages for this and other types of formal verification work. Some of those languages include message passing diagrams and other diagrammatic notations. Since such diagrams commonly find use in discussing message passing in protocols, it seems reasonable to use them as input languages for Protocol Verification work. Their intuitive nature indicates that they may also overcome language barriers inhibiting non-FV experts from reaping benefits from this type of FV.

9 References

- R. Beers. Protocol formal verification: an Intel experience. In *Proceedings of the 45th Annual Design Automation Conference* (Anaheim, California, June 08 - 13, 2008). DAC '08. ACM, New York, NY
- A. Bunker. *Applying a Visual Specification Language to Hardware Verification*. PhD thesis, University of Utah, August 2003.
- D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD thesis, Carnegie Mellon University, 1988.
- G. Gopalakrishnan, E. Brunvand, N. Michell, and S. M. Nowick. A Correctness Criterion for Asynchronous Circuit Validation and Optimization. *IEEE Trans. on Computer Aided Design*, 13(11):1309–1318, November 1994.
- M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.
- T. F. Melham. Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic, PhD dissertation, University of Cambridge. 1990.
- R. Milner, *Communication and Concurrency*, Prentice-Hall, 1995.
- R. J. van Glabbeek, The Linear Time-Branching Time Spectrum (Extended Abstract), CONCUR 1990, Theories of Concurrency, Amsterdam, The Netherlands, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, 278-297.
- R. J. van Glabbeek, The Linear Time-Branching Time Spectrum II, E. Best (ed.), 4th International Conference on Concurrency Theory, CONCUR 1993, Lecture Notes in Computer Science 715, Springer-Verlag, Hildesheim, Germany, 66-81.

The Art of Pre-Si Val: Chapter 25

Symbolic Simulation

By: [Carl Seger](#)

1 Abstract

Symbolic simulation is a technique that marries formal technology with the more familiar simulation paradigm. Instead of simulating 0's and 1's, one can also send in variables or even expressions, thus simulating the circuit for a large number of different input values in one single simulation run. Intuitively, symbolic simulation uses "short and fat" patterns, traditional simulation uses "long and skinny" input patterns. By using a three-valued simulator as a basis for the symbolic simulator, one can achieve even greater efficiency, since signals that "should not matter" can be set to the value X. In this chapter, we go from the simple concepts underlying symbolic simulation, to extensions that allow symbolic simulation to verify the correctness of data-path intensive circuits. We also discuss the type of circuits that have been successfully verified using this technology and methodology.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	2/24/2012	Initial Draft	Carl Seger	
1.1	5/18/2012	Revised version from feedback	Carl Seger	Michael Bair / CCDO Val Staff

3 Contents

1 Abstract.....	545
2 Chapter Revision History	545
3 Contents.....	546
4 Purpose.....	547
4.1 Why do we need this chapter?.....	547
4.2 What does this chapter cover?	547
4.3 What does this chapter not cover?	547
5 Background Concepts.....	547
5.1 Three valued simulation for verification	547
5.2 Symbolic simulation	549
5.3 Symbolic Trajectory Evaluation (STE)	550
5.4 Parametric Representation	552
5.5 Dynamic Weakening	554
5.6 Relational STE.....	555
5.7 Parallel Execution	556
6 Forte System	556
6.1 Specifications in forte.....	556
6.2 Verification Script.....	557
6.3 Complexity Management	558
6.4 Debugging	558
6.5 Regression and Re-use	559
7 Verification Methodology	560
8 Types of circuits for which STE/rSTE works well	562
9 Summary.....	562
10 References.....	563

4 Purpose

4.1 Why do we need this chapter?

Symbolic simulation based on STE is a powerful formal verification technique that is alone in that it can handle complete cluster level designs and, in particular, large data paths. For large compute intensive clusters, STE should be considered a serious validation alternative to traditional dynamic stimulus. In particular, since it is notoriously difficult to achieve good coverage with dynamic stimulus for wide compute intensive clusters, using a technique that provides 100% coverage is very attractive.

4.2 What does this chapter cover?

We review the basic ideas behind STE by going through a number of relatively simple examples. We then discuss how more complex verification efforts can be constructed from these basic principles and bring in many of the engineering aspects that must be considered when trying to use STE as the sole validation tool on live RTL.

4.3 What does this chapter not cover?

Although we discuss STE in some depth, we do not consider some of the underlying technologies that make STE practical. In particular, we assume the reader has some knowledge of Ordered Binary Decision Diagrams (OBDDS) as well as Satisfaction solving (SAT). Briefly, we use both techniques to represent Boolean expressions and efficiently determine whether two expressions denote the same Boolean function or not. For more details, we refer the reader to [1] and [2].

5 Background Concepts

5.1 Three valued simulation for verification

It is common to extend binary simulators with a third "X" value. The value "X" represents un-driven or unknown values. If the simulator handles the X as a true unknown value, one can take advantage of the X value to cover many input patterns at the same time. When we say that the simulator handles X as truly unknown, we require the simulator to have the property that if we replace the X with a 0 or 1, we could not change any result of the simulator from 1 or 0 to X. Basically, if we gain information, the simulator cannot compute anything that has less information⁴¹.

⁴¹ It is worth pointing out that correct Verilog simulators do not obey this requirement, since one can actually test for the X value and thus change the behavior of the simulator so that more X's are produced when an input is changed from X to 0 or 1. In practice though, this problem can easily be mitigated by simply avoiding a few constructs in Verilog.

To illustrate how X values reduce the number of test vectors needed, consider the trivial example of a 7-input AND gate. In Figure 1, we give the (trivial) specification and Figure 2 gives one possible implementation.

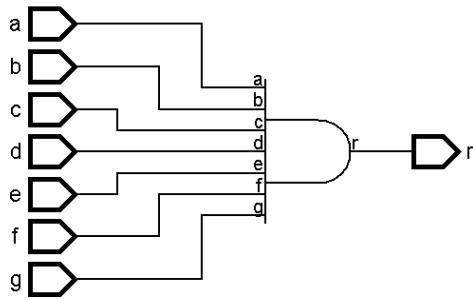


Figure 1: 7-input AND gate specification.

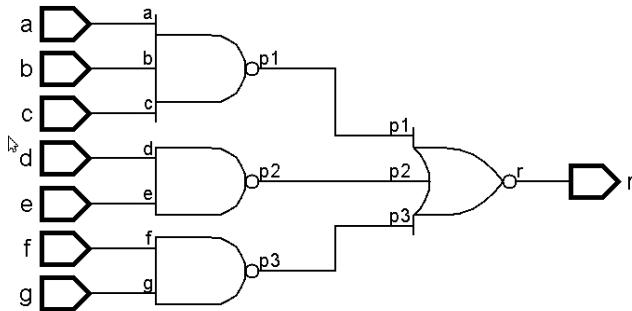


Figure 2: First implementation of 7-input AND gate.

If we want to make sure the implementation is correct, we would need to run through all 128 ($=2^7$) patterns, from 0000000 to 1111111. However, if we have a truly unknown X value available, a more efficient process would be to use the 8 patterns, 0XXXXXX, X0XXXXX, XX0XXXX, XXX0XXX, XXXX0XX, XXXXX0X, XXXXXX0, and finally 1111111. The reason for why this is equivalent to the 128 patterns is that according to the specification, once one of the inputs is set to 0, all the other inputs should not matter. Note that the specification completely determines the number of patterns needed. Thus, one can verify a number of different implementations using the same set of patterns.

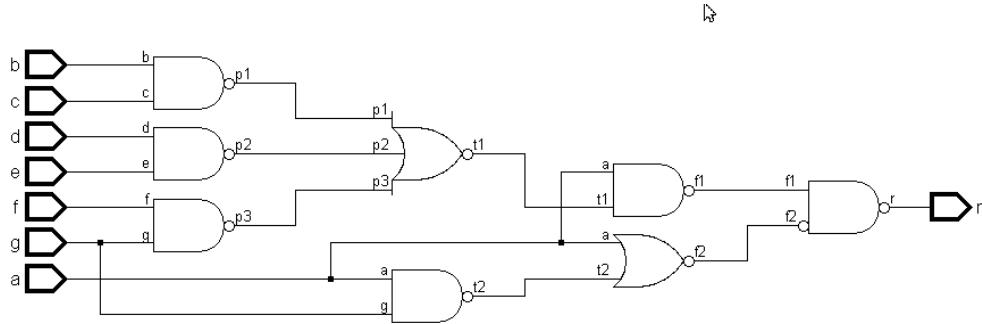


Figure 3: Second implementation of 7-input AND gate.

Although Xs can be extremely efficient in reducing the number of patterns needed, the inherent pessimism with which one has to treat Xs, can cause problems. Consider the (convoluted) proposed implementation for the 7-input AND gate in Figure 3. It is easy to convince oneself that for all 128 binary input patterns, this implementation computes the same result as the specification. However, it does not necessarily compute the same result for every ternary input vector. For example, consider the second pattern above, i.e., X0XXXXX. In Figure 4, we show what values a three-valued simulator computes for this input pattern.

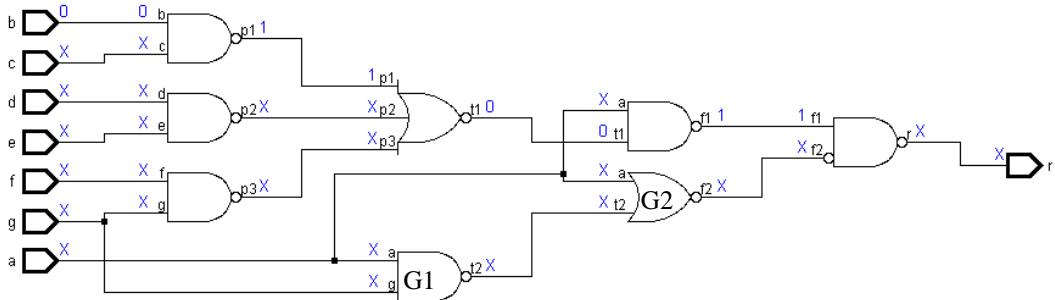


Figure 4: Example of X pessimism.

Here we do not get the expected 0 on the output, but rather an X value. The reason for this unexpected result lies in the pessimism the simulator must use in treating the gates G1 and G2. It is easy to see that for $a = 0$, the output of G2 should be 0 since the output of G1 will be 1 which in turn forces the output of G2 to 0. On the other hand, if $a=1$, then the output of G2 should also be 0 since $a=1$ forces G2 low by itself. Thus, if we analyze G1 and G2 together, we see that the output of G2 is always 0. However, during ternary simulation, G1 and G2 are simulated independently and thus the X on a and g propagate and eventually produce an unknown output value.

Although this type of phenomena does occur in typical designs, it is infrequent. Unfortunately, it can cause significant debugging effort.

5.2 Symbolic simulation

An alternative to scalar (0, 1 and possibly X) simulation is to use symbolic simulation. The basic idea is quite simple; let the simulator propagate expressions representing all possible behaviors

instead of specific values. For combinational circuits this is straightforward. For sequential circuits, a subtle fix-point iterative algorithm is required. For our purposes, it suffices to say that one can in fact create a simulator that computes expressions denoting all possible behaviors of the circuit. In Figure 5 we show an example of what a symbolic simulator might compute for the circuit in Figure 3 when we let every input be a symbolic variable named the same as the input node.

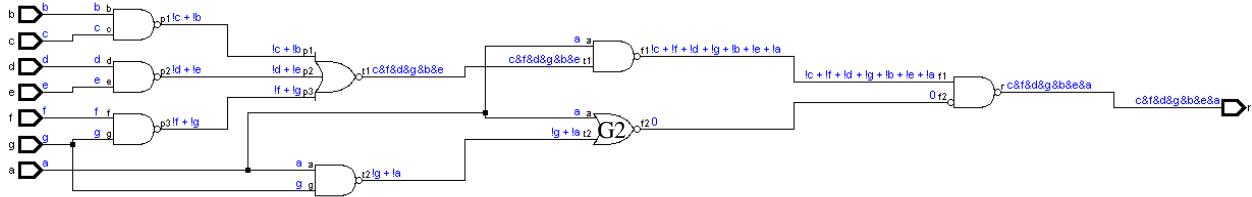


Figure 5: Symbolic simulation of second implementation.

In Figure 5 we use a symbolic simulator that also simplifies the expressions as it carries out the simulation. As a result, the simulator determines that the output of G2 is always 0 since the OR of "a" and "!a" is always 1.

In order to create a symbolic simulator some basic design tradeoffs must be made. Three major issues need to be addressed:

1. How the expressions should be represented;
2. When and how much should expressions be simplified; and
3. How to compare the resulting expressions with the desired results.

Over the years, two basic technologies have emerged as practical alternatives to symbolic simulation: OBDDs and SAT solvers. The first keeps the expressions in a unique representation and carries out a complete simplification of the expressions at every step of the simulation. Effectively, this means that the final comparison is trivial. However, performing the simulation can be very time consuming and/or may never finish due to excessive time or memory requirements. The other approach does not make the expressions unique and only applies some relatively weak simplification routines during the simulation. As a result, the simulation is relatively inexpensive but this approach pays a high computation cost at the end when the expressions are compared for equality. Until around 2002, only OBDD based symbolic simulation was practical. However, there has been a enormous body of applied research done on practical SAT solving and as a result, today SAT is rapidly becoming the de facto standard. OBDDs still have a significant advantage for arithmetic circuits, only. Finally, the SAT approach is almost entirely automated whereas the OBDD approach often need manual guidance in form of an order of all the variables used in the expressions. This by itself explains the attractiveness of the SAT approach.

5.3 Symbolic Trajectory Evaluation (STE)

Although symbolic simulation is powerful for verification, it is not free. Whether the computational requirement in terms of memory and time occur during the simulation or at the end

during the comparison, symbolic evaluation is a compute intensive activity. For OBDDs, the number of variables is often of critical importance and simulations requiring more than a few hundred variables are often not feasible. For SAT solving based symbolic simulation, the dependency on the number of variables is less direct and one can often use many hundreds or even thousands of variables. However, here the complexity of the computation the variables are involved in is much more important. For example, if the circuit performs bit-vector arithmetic, SAT based symbolic simulation is often infeasible. Thus, there is a need for greater efficiency than what plain symbolic simulation can provide.

Similar to the idea of using Xs in a binary simulator, one can introduce Xs in a symbolic simulator as well. Symbolic Trajectory Evaluation (STE) does exactly that. Of course, the symbolic simulator now needs to compute ternary (three-valued) expressions rather than binary expressions and the comparisons must handle three-valued comparisons. One simple approach to extend a binary symbolic simulator to a ternary symbolic simulator is to encode the three values as a pair of Boolean values. That way one can use already existing Boolean manipulation packages. Traditionally, the encoding used is that 0 is encoded as (0,1), 1 is encoded as (1,0) and X is encoded as (1,1). Now the symbolic simulator propagates pairs of expressions through the circuit. For example Figure 6, shows the dual-rail expressions for an inverter and a 2-input AND gates. It is easy to convince oneself that this encoding treats X (1,1) as a truly undefined value and that it agrees with a binary simulator on binary inputs.

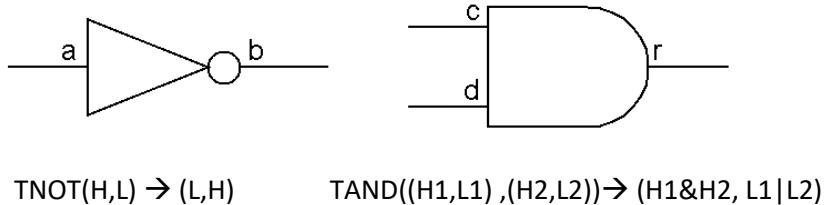


Figure 6: Symbolic dual-rail encoding of inverter and AND gate.

In addition to combining the symbolic simulation with the three-valued simulation paradigm, STE also adds a simple language for describing desired behaviors of a circuit. In particular, it adds a simple way of describing sequences of inputs that produce some sequences of output values. Contrary to many more powerful specification languages, the STE specification language is simple. Only fixed length sequences can be described yielding fixed length responses. Although this limitation makes it difficult, or even impossible, to state some specifications, the simplicity of the language leads to an efficient verification algorithm.

To illustrate a very simple STE verification, consider the circuit in Figure 7. A pictorial representation of a specification for this circuit is given in Figure 8. Basically, the six 32-bit input vectors should be added together and then compared with the input constant k. If they equal, then the ok output should be high two clock cycles later.

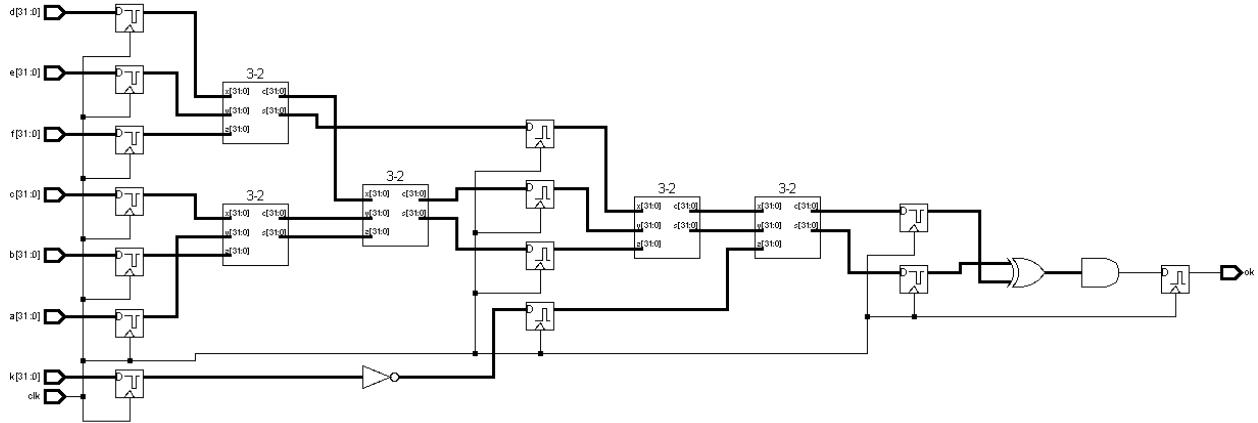


Figure 7: Simple circuit to illustrate STE

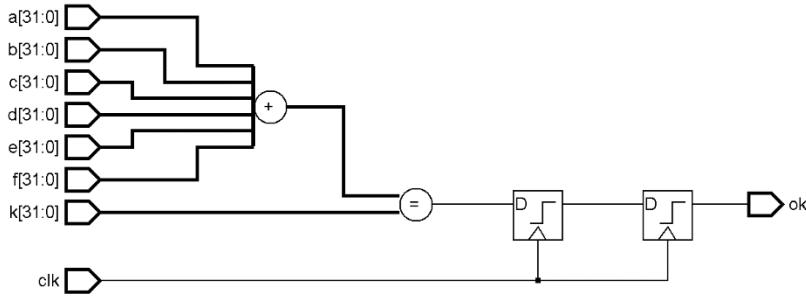


Figure 8: Pictorial specification for circuit in Figure 7.

An STE verification of this design essentially contains three components. First, we would have to describe the clk waveform. Here we would assume that clk is low from 0 to 1, high from 1 to 2, low from 2 to 3, and high from 3 to 4. Second, we would assume that each vector input (a, b, c, d, e, f , and k) have vectors of arbitrary Boolean variables on them from time 0 to 1 (say A, B, C, D, E, F , and K). Finally, we would check that the output ok has the value $(A+B+C+D+E+F=K)$ from time 3 to 4. In Section 6.1, we return to this example and show how the actual verification works in the Forte system.

5.4 Parametric Representation

Symbolic simulation and symbolic STE in particular can cover a large number of inputs in one single verification run. However, in many, if not most, cases we are not interested in verifying the behavior for all possible input combinations, but rather for a subset of those vectors. For example, maybe some of the input vectors are mutually exclusive, or if the valid bit is high in cycle n , then the wakeup signal should have been 1 in cycle $n-1$, etc. One possible way to incorporate such assumptions is to perform the STE run without the assumptions, and then verify that the failure expressions from the STE run denote input vectors that are outside the space implied by the input assumptions. However, doing the symbolic simulation without these input constraints often makes the STE verification run into memory and/or time complexity issues because we simulate the circuit under operating conditions it never experiences in real use and likely is not designed to

handle. Fortunately, there is a technique, called parametric representation, that allows us to incorporate input constraints in the expressions we use for the STE simulation.

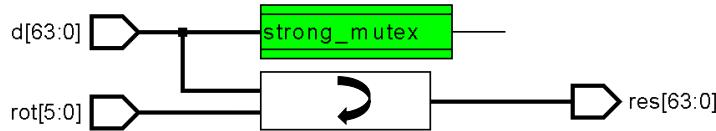


Figure 9: Specification of a rotator of a strongly mutex input vector.

Consider for example, the specification circuit shown in Figures 9. The circuit takes a strongly mutex (one-and-only-one signal is high) input d and rotates it by an amount determined by the input rot . In Figure 10, we provide one possible implementation that uses a pair of shifters, one shifting left and one shifting right. In Figure 11, we provide an alternative implementation that uses a encoder-adder-decoder. Note that the implementation in Figure 10 does not rely on the strong mutex property, whereas the implementation in Figure 11 only works as intended when the input vector d is indeed strongly mutex.

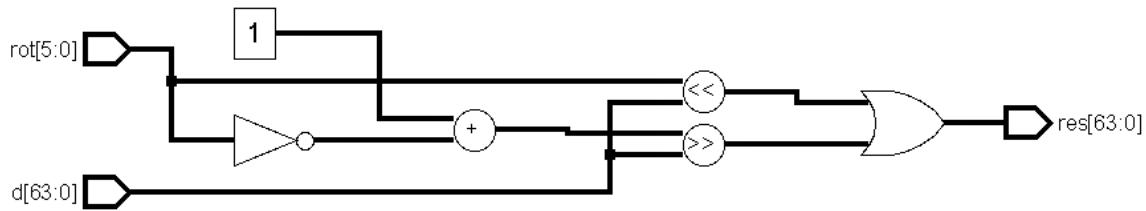


Figure 10: Rotator built from shifters.

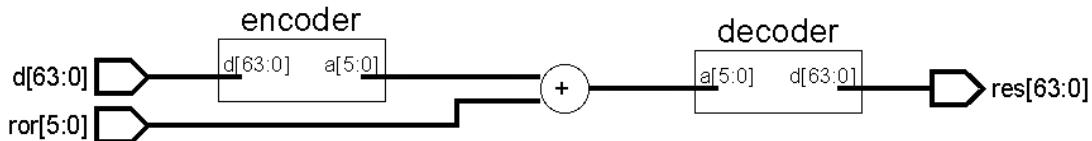


Figure 11: Rotator built from encoder/decoder.

To illustrate the idea of a parametric representation, consider symbolically simulating the circuits with a bitvector $R[5:0]$ on the rot input and the expressions in Table 1 for all the d input.

Table 1. One possible parametric representation for strong mutex condition on d[63:0]

Bit	Expression
d[63]:	D[63]
d[62]:	$\sim D[63] \& D[62]$
d[61]:	$\sim D[63] \& \sim D[62] \& D[61]$
...	...
d[1]:	$\sim D[63] \& \sim D[62] \& \dots \& \sim D[2] \& D[1]$
d[0]:	$\sim D[63] \& \sim D[62] \& \dots \& \sim D[2] \& \sim D[1]$

Note that the expressions are designed so that the vector has exactly one signal high for every assignment of values to the D[63:0] variables and that every possible strongly mutex input vector is covered. Although tedious to confirm manually, one can convince oneself that symbolically simulating this input vectors on the circuits in Figures 10 and 11 yields identical results. With access to a symbolic simulator it is also easy to convince oneself that the strong mutex property is a necessary condition for both circuits to be correct.

Finally, it is worth pointing out that parametric representation is an extremely effective method of controlling the complexity of some symbolic simulation. In fact, it is common to divide the input space in large (100s) cases and use parametric representation to reduce the symbolic simulation complexity. Of course, one then trades off complexity of running one symbolic simulation with more complex expressions against running multiple simulation runs with simpler expressions. In practice, experience has shown that the latter is usually a better approach. In addition, it allows the use of multiple machines/processes thus drastically reducing the total symbolic simulation time.

5.5 Dynamic Weakening

A common hardware design style computes some alternatives speculatively and then chooses the right one at the last moment. This technique allows long running computations to start early and thus reduce their perceived latency. As an example of this type of circuit, consider the circuit in Figure 12.

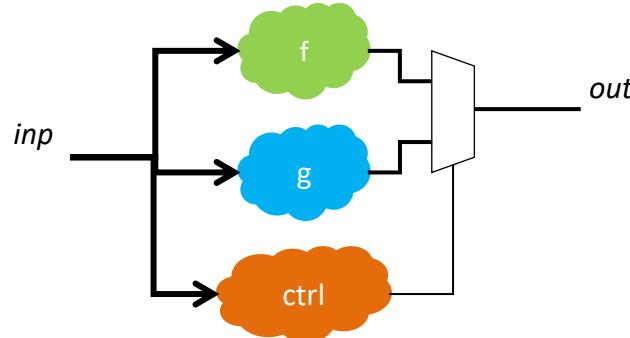


Figure 12: Speculative computation circuit

When verifying such circuits, it is common to do the formal verification by case splits; first verify the f computation, then the g computation. However, rather than cutting the circuit and verifying f, g, and the MUX logic separately, it is usually more efficient to encode the case of interest by adding an assumption and use parametric representation to simulate only the case of interest. Unfortunately, when verifying the "f case", the symbolic simulator must compute the behavior of the g computation as well as the f computation, since it does not know yet that the output of g is ignored for this case. Unfortunately, it is quite possible, and in fact quite common, that this computation takes a very long time and/or use excessive amounts of memory.

Dynamic weakening is a technology built into the symbolic simulator that can help in situations like the one in Figure 12. Intuitively, if the memory consumption is growing over some threshold, the simulator will weaken the signal by making it X⁴². In the example of Figure 12, during the symbolic simulation for the "f" case, if the expressions inside g become too large, they are replaced with Xs. As a result, the output of g may contain Xs as well. However, while we are doing the f case, the parametric representation for the inputs will guarantee that the control signal to the MUX is set so that we will pick up the output of f and ignore the Xs on g.

In practical STE verification, case splits using parametric representation coupled with dynamic weakening are critical and are used on the majority of verification runs.

5.6 Relational STE

Although STE as introduced above is a powerful and effective verification method, it has some limitations. One difficulty is that the specifications must be close to functional, meaning that the user must specify exactly what the result should be. It is possible to deal with don't-cares by using the when clauses in STE, but other specifications can be difficult to state functionally. Consider for example the carry-save adder shown in Figure 13. It is one of the most commonly

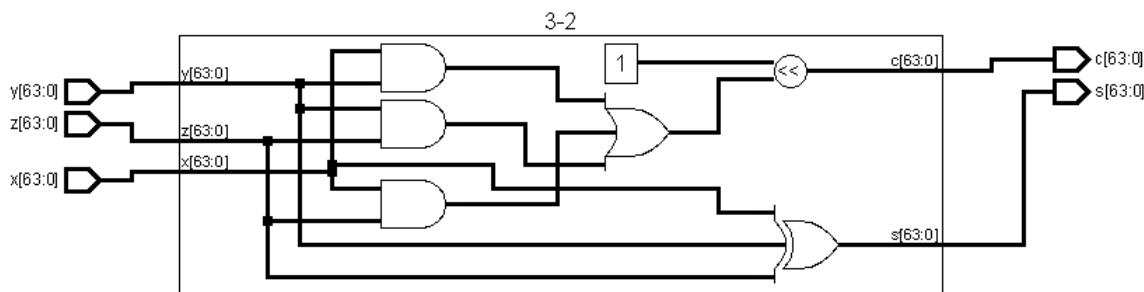


Figure 13: Carry-save adder.

used building blocks in high-performance arithmetic data paths. Now consider how to write a specification for the desired behavior. It is, of course, possible to state that the s output should be the bitwise-XOR or the three input vectors and that c should be the bitwise-majority function shifted left by one, but now the specification looks like the circuit. A much more natural and, in

⁴² The actual code is slightly more subtle in that it does not set the node specifically to X but rather makes the symbolic expression "closer" to X as to maintain as much information as possible but still guarantee reduction in size.

fact the property the design relies on, would be to prove that the sum of the three input vectors should equal the sum of the two output vectors. Unfortunately, in basic STE, such specifications cannot be formulated, let alone verified.

Relational STE, or rSTE, is an extension of STE to allow such relational specifications. In rSTE, input constraints, output constraints, and input-output relations are handled the same way allowing much more concise and simple specifications to be written. In addition, rSTE trades off this extended capability with only a modest performance penalty. Thus, in practice, one can almost universally use rSTE instead of the more primitive STE even when the latter would suffice.

5.7 Parallel Execution

An important aspect of STE/rSTE is that by partitioning the input data space using parametric representation, dynamic weakening, etc., a collection of verification runs are created. Furthermore, these individual verification runs are completely independent. As a result, they are perfectly parallelizable. Thus, by employing a large number of cores and/or machines, many formal verification runs that sequentially would be prohibitively time consuming, can be done in a reasonable time; albeit using a large pool of machines. In fact, it is common to run STE/rSTE jobs that spawn off several hundreds or even thousands of subtasks. Providing an infrastructure that spawns off the jobs and tracks all the results is a non-trivial engineering task but critical to ensure the formal verification effort is complete.

6 Forte System

Inside Intel, STE verification work is carried out in the Forte verification framework, originally built on top of the Voss system[3]. Most of the verification code is written in reFLect: specifications, whether they are functional specifications or relational constraints, verification facilities, analysis routines etc. The execution of a verification task in the framework amounts to the evaluation of a reFLect program.

6.1 Specifications in forte

The backbone of any verification or design debugging effort is a formal specification of required behavior—or, more loosely, some group of properties that the design is expected to satisfy, expressed in a formal specification language. Forte’s design is based on a foundational approach: specifications are expressed in a formalism with only a few simple temporal logic primitives, but which is also embedded in a full-featured functional programming language. This approach gives a generic open framework in which to engineer tailored solutions to individual verification problems. For each verification effort or project, the user creates just the right specification constructs for the problem domain. In practice, one can reuse much of this over a whole class of verifications. For many verifications, suitable libraries may already exist, so the tailoring effort can be cost effective.

To illustrate this idea with a simple example, consider the STE verification of the circuit in Figure 7 again. In Forte, the primitive temporal expression is a 5-tuple (w, n, v, f, t) , that denotes the

temporal formula: "when the condition w holds, node n should take the value v from time f to time t ". A larger temporal formula in STE is simply a list of such 5-tuples. Thus, to constrain the clock, we could write:

```
let cant = [(T, "clk", F, 0, 1), (T, "clk", T, 1, 2), (T, "clk", F, 2, 3), (T, "clk", T, 3, 4)];
```

Effectively, we say that the signal clk goes 0101 over the first 4 phases. However, this primitive notation is not very practical for anything but trivial specifications. Instead, we create a function, that when evaluated creates the desired pattern. For example:

```
let cycle i = [(T, "clk", F, i, i+1), (T, "clk", T, i+1, i+2)];
let cant = cycle(0) @ cycle(1);
```

In fact, since clocks are such a common type of signal, a library of clock functions has been created so that we could simply write:

```
let cant = mk_clock "clk" "01" 2;
```

For the functional or relational specifications, a similar approach is used. For example, one possible functional specification for the circuit in Figure 7 could look like:

```
letrec bvadd cin (a:as) (b:bs) =
  (cin XOR a XOR b):(bvadd (cin AND (a OR b) OR a AND b) as bs)
  /\ bvadd cin [] [] = []
;

let add_vecs av bv cv v ev fv =
  bvadd F av (bvadd F bv (bvadd F cv (bvadd F dv (bvadd F ev fv))));

let func_spec av bv cv dv ev fv kv = (add_vecs av bv cv dv ev fv) = kv;
```

The `bvadd` function is an example of a recursive function defined using pattern matching. It computes the bitvector addition of two lists of Boolean expressions the same way as a ripple-carry adder would do it, one bit at a time. The `add_vecs` function then uses five invocations of the `bvadd` function to add the 6 bitvectors together. Finally, `func_spec` computes the condition under which the sum of the six bitvectors equals the constant `kv` vector.

In practice, a user of Forte would never write specifications at such a low level. Instead, he/she would use a library of "building-block" functions. To illustrate this, the same functional specification could be written as:

```
let func_spec av bv cv dv ev fv kv =
  (av '+' bv '+' cv '+' dv '+' ev '+' fv) '=' kv;
```

6.2 Verification Script

After creating the specification functions, reFLect drives the verification engine. The RTL code is imported into Forte as a compiled model. In addition to a function to carry out STE verification on the RTL model, reFLect also provides a rich library of functions that circuit models. Virtually every aspect of the verification process is controlled through some low-level primitives over which a rich environment has been built by defining functions in reFLect. In practical STE verification,

it is quite rare to use the primitive interfaces. However, their availability to the user provides flexibility to tailor the verification process to the problem at hand.

6.3 Complexity Management

Once immediate problems in a verification script have been corrected, e.g., missing control signals, wrong polarity or timing on signals, etc., the user often encounter capacity issues with the formal verification engine. If the verification is performed using a SAT solver, this usually translates into excessive run time requirements. If OBDDs are used, the limiting resource is usually memory rather than time. In either case, the user faces the challenging task of finding a way to mitigate these capacity limitations. Sometimes, this can be as easy as finding a better variable ordering for the OBDDs. At other times, breaking down the top-level verification into many smaller tasks is required. The programmability of the verification system is a key here allowing custom generated solutions to be developed with fairly modest effort.

It should be pointed out that complexity management techniques that do not depend on the actual circuit under verification, are highly desirable. Consequently, input case splits are very commonly applied. In fact, it is not unusual to break the input space into several hundred or even thousand individual runs. By running all these jobs in parallel, complete verification times measured in a few hours are regularly achieved even for extremely complex verification tasks.

6.4 Debugging

The bulk of any verification effort is debugging, so it is crucial to optimize the verification environment for proof failure, not success. An effective verification environment must inform the user quickly when verification fails, and it must provide focused feedback to help pinpoint the cause of failure, which means a tight debug loop: simulate the circuit, analyze and debug the counterexample, modify the specification or circuit, and resimulate.

In practice, most bugs are in the verification process itself rather than the device under verification. Early on, there are many bugs in the specification. Later in the verification effort, verification bugs become more subtle and harder to distinguish from genuine circuit bugs.

Experience shows that automation and visualization play important roles in providing effective debugging support. It is often very useful to be able to both execute specifications and simulate circuits for specific input values to investigate disagreements. Executable specifications are naturally expressible in reFLect, and simulation is fundamental to STE. The Forte environment also provides automatic counterexample generation for failed model-checking runs, with special care taken to translate internally generated counterexamples into the user's terms. Additionally, counterexample analysis in Forte is integrated tightly with tools for visualizing circuits and waveforms. In Figure 14 we show a typical example of circuit visualization and waveform viewing is used to help the user to debug some counterexample.

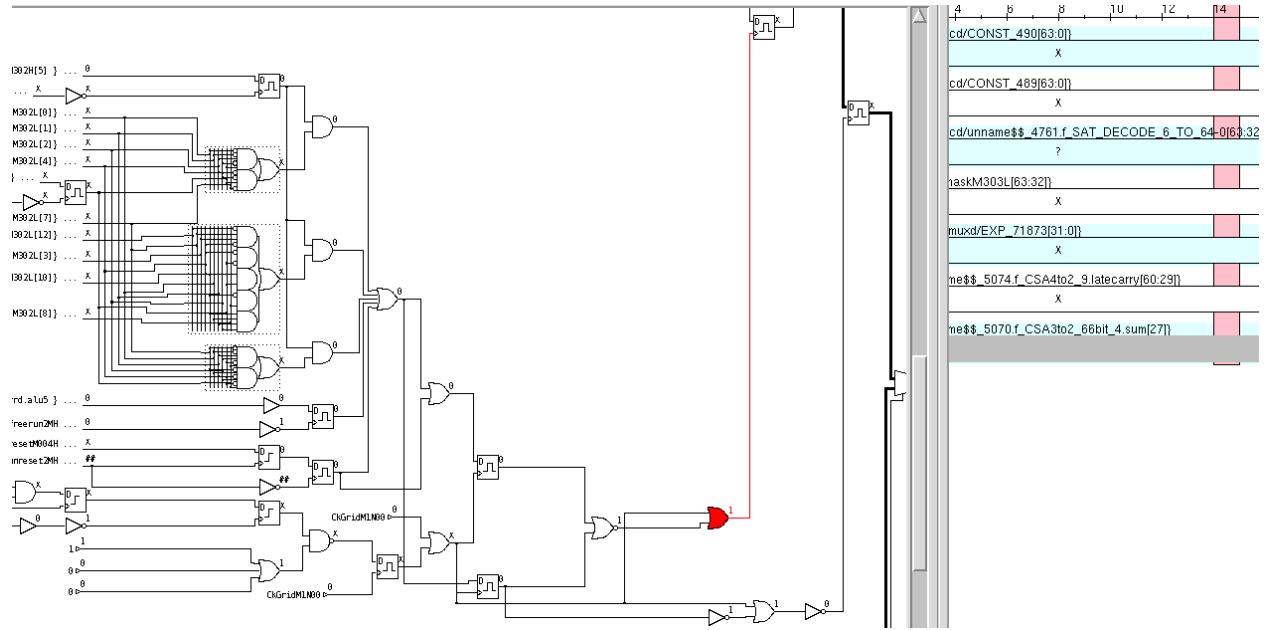


Figure 14: Example of visualization and counter example simulation.

Using a concrete counterexample to isolate the source of a problem is helpful, but the user must understand the behavior of internal signals and do a mental comparison between the circuit and the correct behavior to identify where the circuit fails to meet the specification. It can be much more effective if the user can explore the entire failure domain, or at least intellectually recognizable subsets of it. For this purpose, Forte provides “what-if analysis” to help the user understand the failed proofs. The STE model-checking algorithm computes a data structure that completely characterizes the difference between the circuit and specification; the user can then invoke reFLect programs, either drawn from a library or tailor-made for the problem at hand, that probe this data structure in informative ways.

This facility allows the user to focus on interesting and easily understood counterexamples, instead of being limited to ones chosen arbitrarily by the system. For example, it is often helpful to see a counterexample with as few signals asserted as possible. This can easily be achieved by calling an reFLect library function that generates a concrete counterexample with this property from the failure domain computed by STE. With a little bespoke reFLect programming, more domain-specific analyses are also possible. For example, knowing that an arithmetic circuit processes odd numbers correctly but not even ones could focus attention on the least significant bit, rather than straying into areas of the circuit that compute the sign. One can obtain this information by writing appropriate reFLect functions to probe the failure domain—in this case discovering that it contains only even numbers.

6.5 Regression and Re-use

Verification is an expensive and human-intensive activity. It should therefore support the reuse of proof efforts to amortize verification cost over the lifetime of a changing design, or even over multiple design projects. Two particularly good targets for reuse are specifications and high-level problem decomposition strategies. These often do not vary greatly from implementation to

implementation, and with the right technical machinery, they can be insulated from the messy details of individual circuits.

Reuse of specifications depends on having the capacity to treat large-grained functionality, so that specifications can be made as circuit independent as possible. It must also be possible to structure specifications in a way that separates the circuit-dependent parts and functional parts. In Forte, program structuring in reFLect is the technology that makes this possible. In practice, of course, achieving a well-structured specification and interface to the circuit requires thoughtful and skilled design.

Reuse of problem-decomposition strategies depends on making the strategies circuit independent. A proof based on structural decomposition, or some other implementation feature, is not likely to be reusable for future designs. In particular, structural decomposition required for model checking large circuits makes it unlikely that one will ever see two components whose proofs can be the same.

A better strategy is to base verification methodologies on patterns or common structures in specifications, rather than on patterns in implementations. Specifications are cleaner and generally suffer from fewer idiosyncrasies than circuits. For example, case-splitting strategies derived from analysis of an algorithmically formulated specification often can be used across many different circuit implementations.

One particular important mechanism for isolating the verification effort from the changing RTL code is to isolate node names and specific timing from the general specifications. In Forte, we accomplish this by creating a "circuit-API" consisting of reFLect functions that may allow us to say for example that the `src1` vector is high at time t when in fact the source vector may be split into two time slots and may even be negated. A reFLect function is used to provide this type of isolation.

Although Forte provides executable code at the end of a successful verification effort, how often this code should be updated and run to ensure the RTL remains healthy is a non-trivial tradeoff. Too much regression and all effort should be spent on debugging failures. Too little regression and every regression run start looking like a fresh verification effort. If formal verification using STE is the primary validation mechanism, this tradeoff is easier. Continuous regression is necessary and in fact expected.

7 Verification Methodology

An effective methodology imposes structure on the overall verification effort. This not only helps new users learn but also increases the productivity of experienced users.

At the top-most level, formal verification based on STE has three major focus areas:

1. Data path correctness
2. Control and bypass correctness
3. State correctness

The data path correctness stage focuses on formally specifying the intended behaviors of a number of opcodes and then carrying out an STE/rSTE verification for these specifications. During these verification runs, control, state, and bypass logics are assumed to work correctly. These

assumptions are captured as reFLect functions/programs and are later verified during the subsequent verification stages.

One of the most challenging aspects during the data path verification is how to properly deal with don't-care conditions. It is easy to over-specify some functionality, which may work for some time, but rears its ugly head later when changes are introduced and/or the same specification is used to verify a new compatible part.

The STE/rSTE verification of the data paths is done using either a SAT based solver or OBDDs. In addition, case splits partition the input space into simpler groups that can be verified independently. Few operations require structural decomposition, i.e., that the verification is broken down into verifying individual hardware blocks with interfaces between the blocks exposed and mentioned explicitly in the verification scripts. Examples of these types of problems are wide multipliers, dividers and square root circuits.

Control verification is a more circuit specific activity. It usually involves finding and verifying invariant conditions that hold when assuming some conditions on the interfaces to the design under verification. In modern designs, the aggressive use of clock gating and other power saving features, means that the number of invariants needed is often large, which leads to quite tedious manual efforts that are both time consuming and mechanical. The local use of other types of model checking algorithms, e.g., local reachability, can sometimes reduce this effort. In the Forte framework, these types of algorithms can be written and specialized to particular contexts, allowing an experienced user to increase productivity significantly.

The assumptions made about the inputs to the unit cannot be verified since they rely on logic and/or invariants of blocks outside the unit under verification. Unless those blocks are also verified formally, these interface assumptions must be validated through traditional dynamic validation.

Finally, most designs contain significant state holding arrays, register files or CAMs. For these units, the "something good" verification usually is done as part of the data path verification. However, the "nothing bad" verification is done separately, and as a final verification task. This verification can also be used to verify that timing requirements between writing some data and before anyone can access the data, are guaranteed by the environment and the control logic of the unit.

In the three focus areas above, the actual STE/rSTE verification moves through a series of phases, each of which have a specific aim and produces well defined reFLect code artifacts. Briefly, the phases are as follows:

1. Wiggling--understanding circuit behavior;
2. Targeted scalar verification;
3. Symbolic model checking; and
4. Theorem proving, if needed.

Having a clear understanding of this sequence gives guiding structure to the work of Forte users. Each phase also produces well-defined code artifacts that help to structure verification code. In addition, by saving these artifacts as code segments, much pre-work is available when regression runs later fail and debugging commences.

An important, and often neglected, aspect of large formal verification efforts, and ones based on STE/rSTE and Forte in particular, is good software practices in the development of the verification

scripts. In fact, a large verification effort is very similar to a large coding effort and the same techniques and discipline is required.

8 Types of circuits for which STE/rSTE works well

As already alluded to above, it is clear that verifying data computations that take place in wide data paths is ideally suited to STE/rSTE verification. Thus, floating point and integer execution units are immediate candidates as well as media and graphics pipelines. In addition, in modern micro-architectures a significant amount of control logic processes data. For example, there are linked-list structures implementing mappings and for which a garbage collecting mechanism is employed. Such "data-as-control" units are also well suited for STE/rSTE verification.

Units that consists of large numbers of finite state machines that communicate with each other are, on the other hand, circuits for which STE/rSTE are not particularly well suited. The reason for this is simply that STE/rSTE does not provide a means for computing the set of reachable states, which usually is a prerequisite for verification of this type of circuit. Of course, if the user can state an invariant capturing the relation among the various state machines, then STE/rSTE can be used to verify the invariant and thus the correctness of such circuits. However, the manual effort and expertise such invariant proof requires, usually makes it impractical and other techniques may be more appropriate.

9 Summary

In this chapter we introduced symbolic trajectory evaluation (STE) and extensions to it as a very powerful and effective formal verification technique for verifying large data paths and data-as-control circuits. STE is the only formal verification technology that routinely can work on cluster level designs and properties. As a result, STE, as implemented in the Forte verification system, has become the mainstay of all formal verification of data paths in Intel. By extending the basic STE engines with some enhancements, today's Forte system has allowed a complete formal verification of the HSW EXE cluster including not only the data path (implementing over 3,500 uops), but also all the control, bypass, and state registers. By structuring the verification effort following good software engineering practices, the formal verification effort have been so successful that it has replaced uAV as the validation method on the HSW EXE cluster.

Although the execution cluster is the most visible success of STE verification, other areas are also entirely verified using STE. For example, the microcontroller in the power control unit is entirely verified using STE. In addition, STE has become the main validation tool in graphics cores.

Looking forward, more clusters probably will be validated entirely with formal verification using STE. For some of this to take place, integration of some existing technology will likely be needed, mostly to automate tedious tasks reducing the manual effort, as well as develop re-usable specifications for the behavior of such new clusters.

10 References

- [1] Bryant, R.E.; Graph-Based Algorithms for Boolean Function Manipulation, In: IEEE Transactions on Computers,, 1986, vol.C-35, no.8, pp.677-691.
- [2] Moskewicz, M., Madigan, C. F., Zhao Y., Zhang L., and Malik, S.. Chaff: engineering an efficient SAT solver. In Proceedings of the 38th annual Design Automation Conference (DAC '01). ACM, New York, NY, USA, pp. 530-535.
- [3] Hazelhurst, S., Seger, C.-J.H.: Symbolic trajectory evaluation. In: Kropf, T., (ed.), Formal Hardware Verification. Springer, Berlin Heidelberg New York, 1997, pp. 3-78.
- [4] Paulson, L.: ML for the Working Programmer. Cambridge, Cambridge, 1996.
- [5] Seger, C.-J. H. Jones, R. B. O Leary, J. W. Melham, T. Aagaard, M. D. Barrett, C. Syme, D.: An Industrially Effective Environment for Formal Hardware Verification, In: IEEE Transactions on Computer Aided Designs of Integrated Circuits and Systems, 2005, Vol. 24; Number 9, pp. 1381-1405.
- [6] Kaivola, R., Kohatsu, K.; Proof engineering in the large: formal verification of Pentium®4 floating-point divider, In: International Journal on Software Tools for Technology Transfer, 2004, Vol. 4, Issue 3, pp. 323-334.
- [7] Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A., : Replacing Testing with Formal Verification in Intel Core™ i7 Processor Execution Engine Validation, In: Computer Aided Verification, Lecture Notes in Computer Science 2009, Vol. 5643, pp. 414-429.

The Art of Pre-Si Val: Chapter 26

FV Theory Fundamentals

By: [Erik Reeber](#)

1 Abstract

This chapter reviews some of the background theory behind formal verification (FV). We begin by a discussion of automated theorem proving, followed by model checking, symbolic simulation with Boolean Decision Diagrams (BDDs), and finally Satisfiability (SAT) Solving. For each of these FV topics we give a brief overview of how the technique works with an example. We also compare the strengths and weaknesses of each approach and how it has been applied to hardware validation.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	6/17/2016	Initial Draft	Erik Reeber	Michael Bair

3 Contents

1 Abstract.....	565
2 Chapter Revision History	565
3 Contents.....	566
4 Purpose.....	567
4.1 Why do we need this chapter?.....	567
4.2 What does this chapter cover?	567
4.3 What does this chapter not cover?	567
5 Automated Theorem Proving	567
5.1.1 Rewriting.....	567
5.1.2 Induction	568
5.1.3 Progress Proofs with Ordinals	570
5.1.4 Strengths and Weaknesses of Automated Theorem Proving	571
5.1.5 Applications of Theorem Proving	571
6 Model Checking.....	572
6.1.1 Model checking example	572
6.1.2 Validating Progress.....	574
6.1.3 Strengths and Weaknesses of Model Checking	576
6.1.4 Applications of Model Checking	576
6.1.5 Bounded Model Checking.....	576
7 Symbolic Simulation.....	577
7.1.1 BDD Example	577
7.1.2 Strengths and Weaknesses of Symbolic Simulation.....	579
7.1.3 Applications of Symbolic Simulation	580
8 Satisfiability (SAT) Solving.....	580
8.1.1 SAT Solving Example	581
8.1.2 Strengths and Weaknesses of SAT Solving	583
8.1.3 Applications of SAT Solving.....	583
8.1.4 Satisfiability Modulo Theory (SMT) Solving	584
9 Summary.....	584
10 Future Work	585
11 References.....	585

4 Purpose

4.1 Why do we need this chapter?

FV is an important hardware validation technique that has been applied successfully within Intel and externally on a wide spectrum of design applications. For those who have never used FV, or who have used it without understanding how it works, we hope this chapter will provide some insight. This insight will then allow a validator to apply it more effectively and more fully understand its limitations.

4.2 What does this chapter cover?

This chapter provides an overview with examples of four major FV techniques: theorem proving, model checking, symbolic simulation, and SAT solving. These are all techniques that have been applied successfully on industrial designs.

4.3 What does this chapter not cover?

The descriptions of the various FV techniques are relatively brief. For more detailed descriptions, we suggest consulting a textbook on formal verification, such as the one written by our colleagues at Intel [12]. This chapter is also not intended to discuss how FV may be applied within particular domains, such as power management or core design.

5 Automated Theorem Proving

Automated theorem proving was the first FV technique to be applied to hardware and software designs and continues to be applied to both within certain contexts. In this approach, traditional mathematical proof procedures, or rules of inference, form the basis of the validation effort.

In automated theorem proving, a tool both guides and confirms the use of rules of inference to validate that a design satisfies its specification. Most of the commonly used tools here are developed at research institutions and are publically available, including ACL2, HOL, Isabelle, Coq, and PVS [2,3,4,5,8]. While any rule of inference can be used, the techniques employed by these tools usually fall into the categories of rewriting, induction, and generalization.

5.1.1 Rewriting

The most common rule invoked by most theorem provers is rewriting. For example, let's imagine we want to prove that $A*B*C = C*B*A$. One way to do so would be to rewrite $(C*B)*A$ through a series of trusted rules:

$$(C*A)*B = (A*C)*B = A*(C*B) = A*(B*C) = (A*B)*C$$

Here we start by using the commutative property of multiplication to transform $(C^*A)^*B$ into $(A^*C)^*B$. We then use the associative property to transform it to $A^*(C^*B)$ and then use the commutative and associative property again to go to $A^*(B^*C)$ and $(A^*B)^*C$ respectively.

In this way, rewriting simply takes one expression and changes it into an equivalent expression. It often is guided by a desire to make the expression in question closer to some normal form. In the above example, we are being guided by the idea that variables should be expressed in alphabetical order. Once in alphabetical order it is easier to determine if two expressions are equal.

Most axioms or theorems can be expressed as rewrite rules and either left to continually operate (such as those making multiplicative variables in alphabetical order) or used only as directed by a proof. For example, let's suppose we define the following function:

$$f(x)=4*x, g(x)=x/2$$

Both of these definitions make good rewrite rules. Whenever we see an expression $f(x)$, where x is anything, we can replace it with $4*x$ and whenever we see the expression $g(x)$ we can similarly replace it with $x/2$. For example, say we want to show that:

$$g(f(x))=f(g(x))$$

Using the definitions as rewrite rules produces the following proof.

$$g(f(x))=f(g(x)) \Rightarrow g(4*x)=f(x/2) \Rightarrow (4*x)/2=4*(x/2) \Rightarrow 2*x=2*x$$

Now we can also use “ $g(f(x))=f(g(x))$ ” as a rewrite rule as well, if we want to stop using the definitions. For example, we could use it as follows to simplify an expression involving just calls to f and g :

$$g(f(g(f(x)))) \Rightarrow g(f(f(g(x)))) \Rightarrow f(g(f(g(x)))) \Rightarrow f(f(g(g(x))))$$

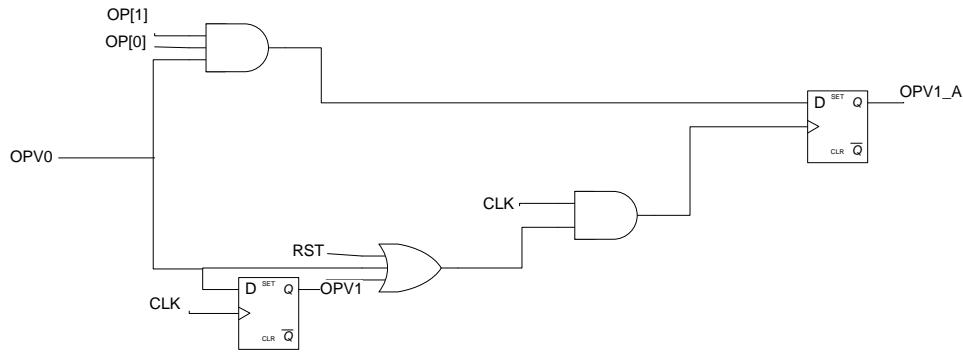
The savvy reader might notice that all of our rewriting above is being done inside-out rather than outside-in, $g(f(g(f(x))))$ rewrites to $g(f(f(g(x))))$ rather than $f(g(g(f(x))))$. Rewriting inside out is a common choice among automated theorem provers, but is certainly not a requirement.

5.1.2 Induction

Induction is a fundamental rule of inference that can extend rewriting to validate almost any hardware or software design. The basic concept of induction is that we prove an expression $f(n)$ is true for $f(0)$ and then show that by assuming it is true for $f(n-1)$ we can show it is true for $f(n)$. From this we can conclude $f(n)$ for all n . We refer the assumption of $f(n-1)$ as the *inductive hypothesis*, and the proof that this implies $f(n)$ is known as the *inductive step*.

5.1.2.1 Induction Hardware Validation Example

For an example from hardware validation consider the following circuit:



The above circuit is actually similar to the many circuits one commonly finds within a core data path logic. The idea here is that $OP[1:0]$ is an opcode and $OPV0$ represents whether we have a valid operation at pipe stage 0. We are then producing a signal $OPV1_A$ that represents whether we have operation “A” ($OP[1:0] == 3$) in pipestage 1.

Our specification is that the $OPV1_A$ wire needs to be asserted if and only if we have an OP_A going through the system in pipestage 1.

The reason why we need induction to prove this property is that our flop is using a gated clock. We need to show not only that the specification holds when the clock is running, but also that the specification holds when the clock is gated as well. Note that our circuit is purposefully designed such that it only gates when there is no valid operation---in that case it is safe to leave the flop output stuck at 0.

The inductive proof looks as follows:

Define $OPV1_A(t)$ as the value of wire $OPV1_A$ at time t .

Base Step. First $OPV1_A(1)=0$ since RST is high at time 0, forcing the clock to toggle (and no OP_A is in the system at reset).

Inductive Step. If the clock toggles, then $OPV1_A(t)$ is $OP[1:0](t-1)==3 \&& OPV0(t-1)$, which is what we want. So we only need to concern ourselves with the case where it does not toggle. When it does not toggle, we know that $OPV0(t-1)$ is 0 and $OPV0(t-2)$ is 0. But since $OPV0(t-2)$ is 0 by the induction hypothesis we have $OPV1_A(t-1) = 0$, which now implies $OPV1_A(t)=0$ since the clock is not toggling. Since $OPV0(t-1)$ is 0, we have exactly what we want $OPV1_A(t) = OPV0(t-1) = 0$.

5.1.2.2 Generalization

It should be noted that generalization is often required prior to using induction. With generalization we are proving a stronger property than what we want, which is necessary for the inductive hypothesis to be strong enough in the inductive step.

For example, in our previous scenario we might only want to prove the following

$$OPV1_A(t) \text{ implies } OP[1:0](t-1)==3$$

However, if we attempt to prove this statement directly by induction the proof will fail. For the inductive hypothesis to be strong enough we need to prove OPV1_A(t) implies OPV0(t-1) as well.

5.1.3 Progress Proofs with Ordinals

An important part of any validation effort is showing that the machine makes progress---i.e., does not hang. The proof technique usually employed in automated theorem proving for showing progress is to map the states of the machine to *ordinals* and then show that unless the machine has reached its destination the next state has a smaller ordinal than the previous state.

At its simplest an ordinal can be a natural number. Clearly if a natural number is always decreasing we will eventually hit the lowest natural number, 0, which is generally defined as the finishing state where more progress is no longer needed.

Ordinals get a little more complex though when we add the ordinal ω (omega). The ordinal ω is a number that is larger than every natural number. If we map a state to ω it may seem like that defeats progress because there are an infinite number of possible states below it. However, all of those smaller states are natural numbers, so we will still have to pick a natural number next, and then we have a finite number of those before we hit 0. Therefore, if we map all the states of the machine to natural numbers or ω and we show that the next state is always a smaller ordinal, then we are guaranteed to be making progress towards state 0.

On top of ω we also can include all the arithmetic (+, *, and \wedge) with ω and natural numbers and these are still ordinals. For example, $\omega+\omega$, or ω^*2 is an ordinal that is larger than all natural numbers and larger than all natural numbers plus ω . Then ω^*3 is larger still and $\omega^*\omega$ or ω^2 is even bigger, with ω^ω even bigger. All of these still share the property that if we map our states to them and show that the next state is a smaller ordinal then we are guaranteed to be making progress toward state 0 (where progress is no longer needed).

5.1.3.1 Hardware example

Imagine that a chip has a boot FSM that must be transitioned through as part of the boot process. This machine might call other sub-machines and wait on them in turn. The design may further make a call to another part of the machine and then wait on an acknowledgment. How would all of this state be mapped to ordinals?

The ordinal here would likely look something like the following:

$$\omega^2 * \text{BOOT_NUM}(\text{BOOT_STATE}) + \omega * \text{SUB_NUM}(\text{SUB_STATE}) + \text{MAX_WAIT}(I)$$

Here `BOOT_NUM` is a function that converts the boot FSM into natural numbers and `BOOT_STATE` is the current boot FSM's state. Similarly `SUB_NUM` is a function that converts the sub-machine's state into natural numbers and `SUB_STATE` is the current sub-machine state. Finally, `MAX_WAIT` is the maximum amount of time that the other part of the machine is allowed to take before acknowledging given the current and past values of the interface `I` to it.

Now the proof problem becomes showing that either the BOOT FSM is progressing to a smaller numbered state, or the sub machine is, or we are currently waiting for the acknowledgement (in which case the maximum possible time decreases by one each cycle).

5.1.4 Strengths and Weaknesses of Automated Theorem Proving

The key strength, relative to the other formal verification techniques, of automated theorem proving is that it is scalable. Every other technique described here hits a wall at some point where it no longer can continue to be used once a design reaches a certain level (or kind) of complexity. On the other hand, automated theorem proving can work regardless of how complex the design is. As an academic project an entire processor was proven correct using automated theorem proving, along with its compiler, its operating system, and some application software [9,10]. There was no point at which automated theorem proving could not be successfully applied.

Why does theorem proving always seem to work? It is because fundamentally we do not build things unless the designers and architects have some underlining justification for it. In the end, theorem proving is about teasing out that justification, and giving it to a computer for double-checking.

On the other hand, despite the obvious benefits of theorem proving it is rarely used at Intel. The reason for that is that while it is scalable, it is not easy. There is a reason why designers do not write full proofs that their designs are valid. While the proof is in their head, putting it on paper can be quite challenging. Creating a proof that a computer can follow is even more challenging.

Furthermore, proofs are relatively brittle. A small change in design can have a major impact on its correctness justification. Thus, maintaining proofs is a non-trivial task. And it becomes near impossible if the proof needs to be handed off to a non-expert on the following project.

5.1.5 Applications of Theorem Proving

Automated theorem proving has been used successfully by some of Intel's competitors. For example, AMD used automated theorem proving to validate the Athlon floating point design [6]. Also Centaur Technology, another x86 design company, has used automated theorem proving to validate a non-trivial portion of their core design [11].

On the other hand, automated theorem proving have always been used sparingly at Intel. There have been some limited use of theorem provers for the EXE cluster within the core design, but these were never put into the main validation environment and we are unaware of any major automated theorem proving effort at Intel.

Despite our infrequent use of this technique though, we feel that a strong knowledge of theorem proving can be a huge help to anyone engaged in formal verification. When other formal verification techniques are not up to handling the full complexity of a design, the techniques from theorem proving can be used to stich smaller validation efforts together into one cohesive unit. Within the EXE cluster in the Haswell core design, for example, we used induction, generalization, and case splitting to handle problems that were beyond the scope of symbolic simulation. Furthermore, on BXT at times when our model checking tools could not fully validate progress properties, we found an ordinal-based approach was able to get to a full proof.

6 Model Checking

Model checking is an alternative approach to formal verification, focused on the validation of finite state machines. The basic approach was first developed in the early 1980's [7], and the creators have since been awarded the Turing Award for it. Model checking has been used far more at Intel than theorem proving and is a widely accepted validation methodology for hardware across the industry. Model checking generally forms the foundation of Formal Property Verification (FPV), which is described in more detail in the [Formal Property Verification](#) chapter.

A key insight to model checking is that finite state machines can be represented as a graph with the machine states as nodes. Most validation problems can then be reduced to the problem of determining whether a set of illegal states are reachable from the start state(s).

6.1.1 Model checking example

For a concrete example, consider the following simple Verilog code:

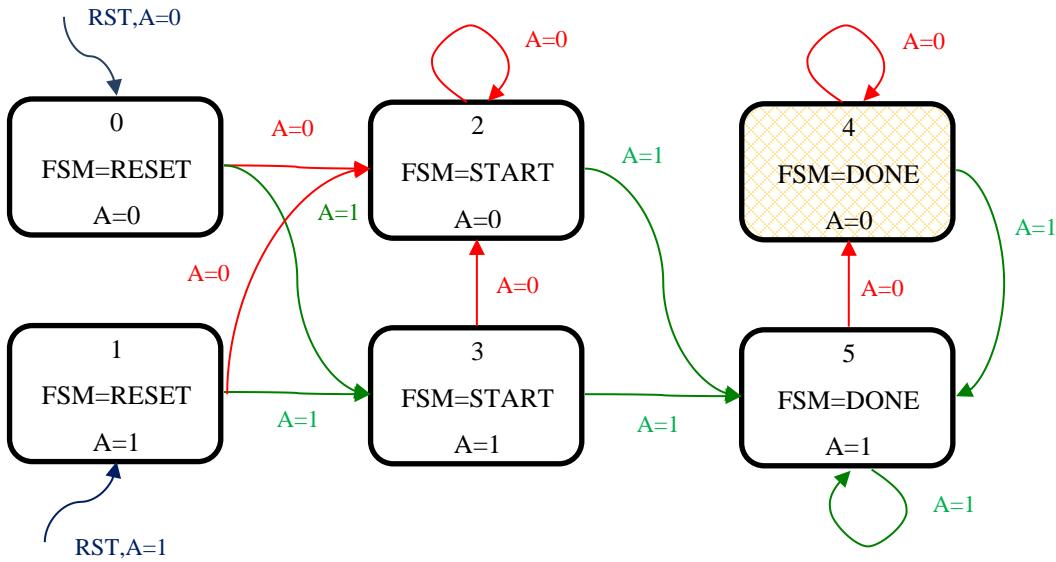
This code represents a simple finite state machine that might be involved in the boot process for a larger machine.

```
unique casez (FSM)
    RESET: FSM_Nxt = START;
    START: FSM_Nxt = A ? DONE : FSM;
    default: FSM_Nxt = FSM;
endcase
always_ff @(posedge clk) begin
    FSM <= RST ? RESET : FSM_Nxt;
end
```

This finite state machine starts in the RESET state, then transitions to the START state, where it waits for the signal A to go high. Once A is high, it transitions to the DONE state.

This example is inspired by real circuits we have seen in the boot logic for Intel chips. Often a boot FSM will enter a START state after reset and as a result send out a request to another part of the chip, such as to lock the clock frequencies. We then wait for an acknowledgement, in this case signal A, before moving on to the next state in the FSM, after which it may be safe to boot another part of the chip.

We can represent this FSM using the following graph:



The numbers here just label the six states for reference. Note that we have doubled the number of states in the machine to help us represent the current value of our input, A.

A simple property that we might want to prove is:

If the FSM is in the DONE state, then A is high.

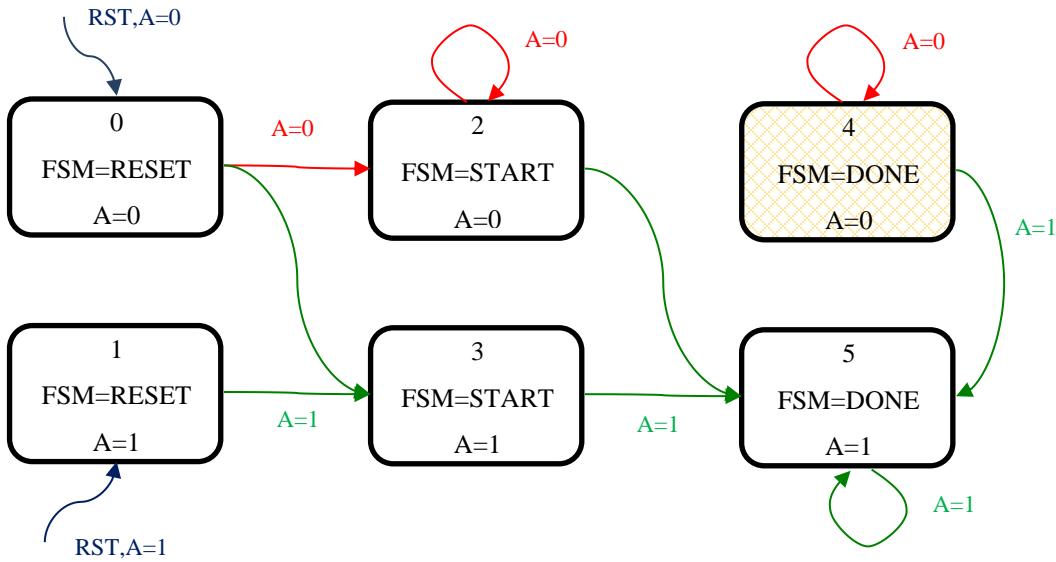
The way we can go about proving, or refuting, this property is by noting that state 4 is illegal. The question then becomes whether we can reach this state. Of course, the answer in this example is yes, and a model checker will produce a counterexample trace showing how to get to state 4. For example, we can transition from 0 to 2 to 5 to 4 by keeping A low for a couple cycles, raising it to reach the DONE state, and then dropping it again to get to the illegal state 4.

The validator may then immediately decide that this counterexample is not real, and add a new assumption to the system:

The signal A never falls

Remember that this FSM was inspired by our real boot FSMs. If signal A represents, for example, an acknowledgement that clocks are ready for use, we do not expect such an acknowledgement to ever fall.

The effect of this assumption on our analysis is to disallow several of the transitions, producing the following new graph:



Now the model checker will return that our property holds because state 4 is not reachable (we start in either state 0 or state 1 and we cannot get to state 4 from those states).

6.1.2 Validating Progress

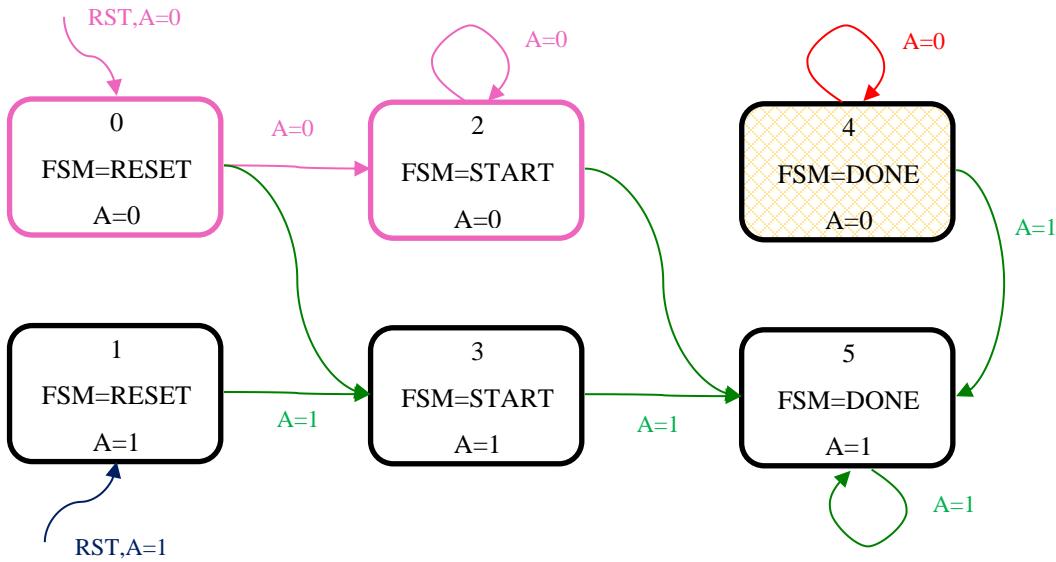
The astute reader may note that basic reachability analysis can validate that we never get into an illegal state, but a lot of real world bugs involve machines hanging.

Fortunately, model checking has a solution to this problem. Progress properties can be validated by requiring that certain states always be reached. The proof fails if and only if the graph contains a loop without the required states.

For example, let's attempt to prove the following:

Eventually the FSM reaches the DONE state

Looking at the graph, a model checker will produce the following counterexample (shown in pink):



The counterexample here is that the machine starts in state 0, then loops in state 2 forever, never receiving a high A and thus never moving to DONE.

The validator may decide that this counterexample is not a problem because we are guaranteed to see A go high at some point. Going back to our real world inspiration, we would have sent a request to the clocking unit once we entered the START state and the clocking unit needs to eventually acknowledge our request. So let us say that there is a REQ signal coming out of our machine that is high whenever the FSM is in the START state. Once REQ goes high it is required that the machine will respond by raising the A signal. We can encode this idea through the following assumption:

$$\text{Eventually either } \text{REQ} \text{ is low or } A \text{ is high}$$

This assumption will disallow any loop that does not include either a high A signal or a low REQ signal. The loop here just includes state 2 indefinitely, which has A low and REQ high (since the FSM is in the START state) so it no longer constitutes a counterexample. Since this loop is the only one without the FSM getting to DONE, the model checker will conclude that our property holds—the FSM must always reach the DONE state.

A careful reader may note that we have been vague about the reset signal here and we could construct a loop that goes from state 1 to state 3 and then back to state 1 through a reset. We are assuming here that the reset is high at the beginning of time and never again. Without that assumption there would be another loop in our graph that a model checker would find as a counterexample to our progress property.

6.1.3 Strengths and Weaknesses of Model Checking

The strength of model checking, relative to theorem proving, is that it has a much higher degree of automation. With theorem proving we generally have to guide the prover toward the correct proof, but with model checking the proof is found entirely by the tool.

In fact, model checking is arguably easier than many testing-based methodologies. With constrained random testing, for example, one needs to define a legal input state and write checkers to validate that the design is correct within that input state. Model checking simply requires the same—we need to constrain the input space through assumptions and define checkers through assertions. However, with constrained random testing we need to also define coverage to ensure that we are hitting the interesting cases and may have to guide the random inputs towards interesting cases. With model checking this effort is largely unnecessary because we are hitting the entire space all the time. Coverage may still be defined in a model checker, but it is only there to double check that we have not accidentally reduced the input space too much. Furthermore, constrained random testing requires long waits for new failing tests to be generated and run—model checking by contrast tends to find the bad inputs much faster.

Relative to theorem proving, model checking also has a huge advantage in that failures result in counterexamples. Theorem provers are searching for a proof. When they fail to find one it may suggest a problem, but it does not necessarily mean that a proof does not exist. On the other hand, model checkers generally show exactly why an assertion does not hold.

The weakness of model checking is that as design complexity rises, and especially as the number of reachable states in the machine rises, the memory and/or time required tends to rise exponentially. In effect, for small problems model checking works great, but eventually model checkers hit a wall and struggle to give any useful information. This problem is why model checking has not been as effective in software verification, where the state space is very large, and potentially infinite.

6.1.4 Applications of Model Checking

Model checking has been used effectively many times within Intel, as it forms the basis for the FPV and protocol formal verification technique. For example, on the Broxton project FPV was used as the primary validation technique for the IOMMU unit. It was also used extensively to validate the System Agent and to validate targeted portions of the PMC and P-unit.

Model checking has also been used elsewhere outside of Intel in the larger hardware validation community. Every senior validator should be aware of the technique and use it when appropriate.

6.1.5 Bounded Model Checking

Bounded model checking is one way to work around the limits of model checking. In bounded model checking rather than attempting to search the entire state space the space is limited to a bounded number of transitions.

Bounded model checking often comes naturally from full model checking since a failed attempt to reach any illegal states usually produces a statement from the tool that all states within x steps of reset have been traversed without finding a counterexample. In this way bounded model

checking often becomes the validation technique of choice when model checking fails to yield a full proof.

Bounded model checking is often sufficient since counterexamples rarely require large numbers of transitions. One can also start from something other than the reset state, such as a state produced in the middle of an SOC test and then use bounded model checking to show that no counterexamples exist within a bounded number of transitions from that state. In this way, while bounded model checking does not produce complete assurance that an assertion holds it can provide very strong evidence that it does.

Bounded model checking also is a useful approach for gating regressions. A full model checking proof may take a lot of memory and/or time to run, but a bounded proof will find many of the same counterexamples in much less time. This means that it is often worthwhile to use bounded model checking as a check on any new RTL and then have a periodic regression that runs the full proof.

7 Symbolic Simulation

Symbolic simulation is another highly automated formal verification technique that has had a great amount of success when applied to certain hardware validation problems. It is described in detail in [Symbolic Simulation](#) chapter, but here we give a more introductory description.

Consider again the following simple statement:

$$A*B*C = C*A*B$$

Our methodology for solving this problem using theorem proving was essentially to use the distributive and commutative properties guided by the goal of putting variables in alphabetical order. In essence, we were using a canonical form for multiplication of variables (alphabetical order) such that determining whether two expressions were equivalent amounts to simply asking whether the two expressions are syntactically the same (asking whether $A*B*C$ equals $A*B*C$ is trivial, as is asking whether $A*B*B$ equals $A*B*C$).

The key insight to symbolic simulation is the same basic idea—if we put two expressions into a canonical form then determining whether they are equal is trivial.

In symbolic simulation the variables generally take on Boolean values and the canonical form generally used is known as Boolean Decision Diagrams (BDDs). The difficulty with a canonical form is that often the canonical form is not a compact way of expressing the formula in question, but BDDs are designed to exploit structure sharing in order to keep the structure from becoming overly large.

7.1.1 BDD Example

For an example, let's say that we are attempting to compute $q[1:0]$ where $q[1:0]$ is the sum of two, two bit inputs a and b :

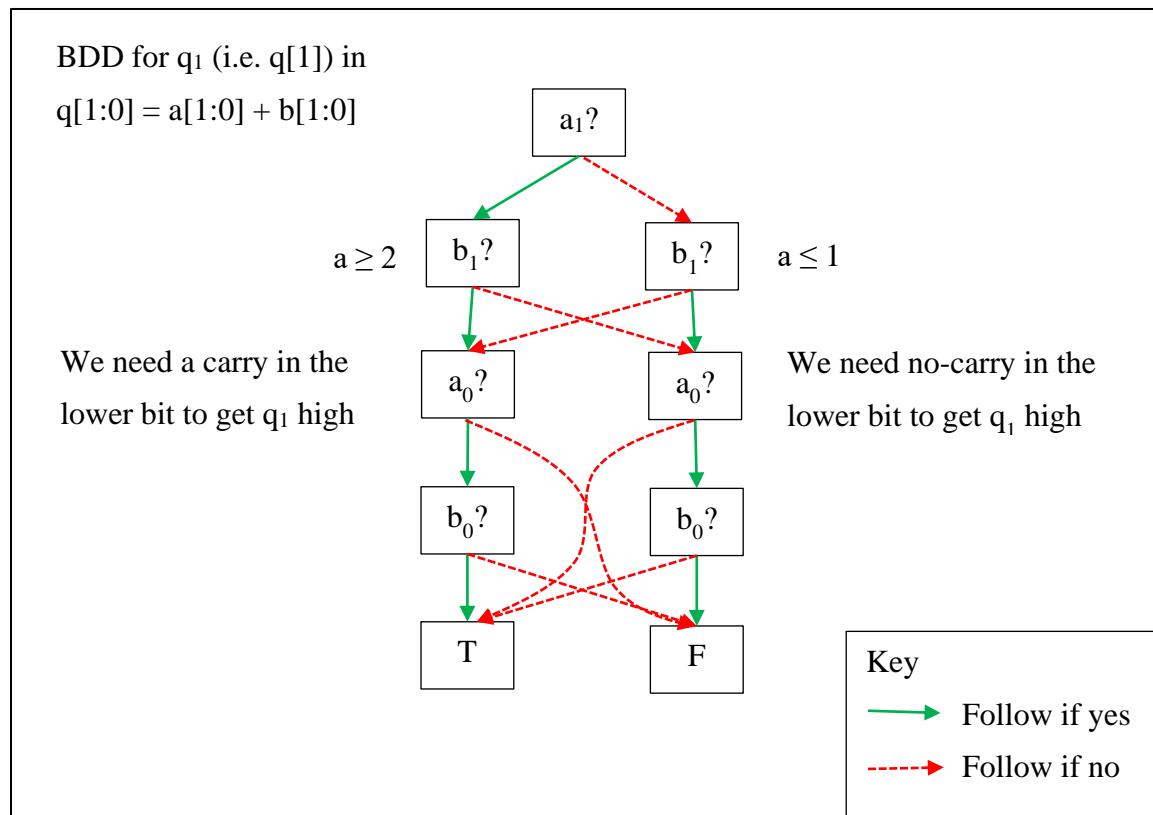
$$q[1:0] = a[1:0]+b[1:0]$$

Now, let's focus on the upper bit $q[1]$, which we'll denote q_1 . If we want to express q_1 in simple terms of AND ($\&\&$), OR ($\|$), and NOT ($!$) gates in terms of the inputs a and b , we can express q_1 as:

$$\begin{aligned} q_1 = & \quad a_1 \&\& a_0 \&\& b_1 \&\& b_0 \quad \| \quad a_1 \&\& a_0 \&\& !b_1 \&\& !b_0 \quad \| \quad !a_1 \&\& !a_0 \&\& b_1 \&\& b_0 \quad \| \\ & a_1 \&\& !a_0 \&\& !b_1 \quad \| \quad !a_1 \&\& b_1 \&\& !b_0 \quad \| \quad !a_1 \&\& a_0 \&\& !b_1 \&\& b_0 \end{aligned}$$

Note that this expression is already getting pretty large, and it is not obvious if that it is a canonical form. Clearly if we were going to talk about the upper bit in a 32 bit summation the expression could get massive.

Here is what an actual BDD for this expression looks like:

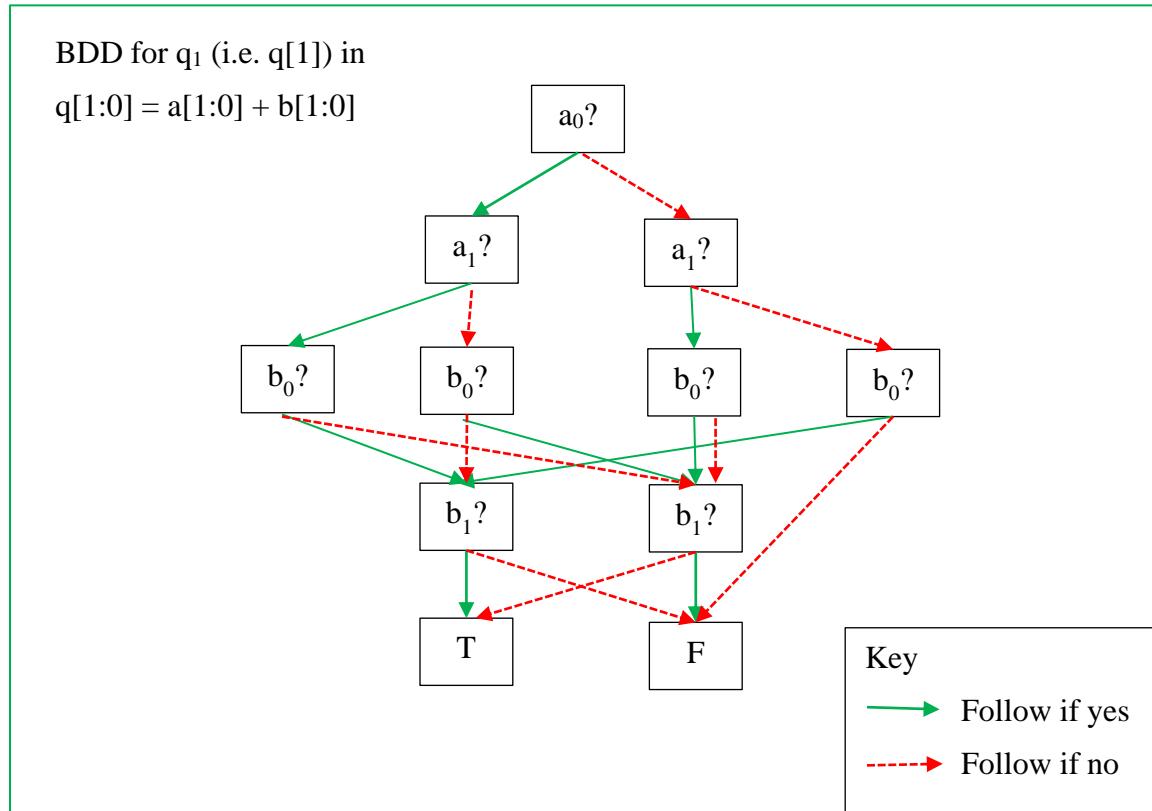


The BDD is essentially a decision tree, with each node representing whether a given Boolean input is true or false. The tree ends with a final node representing the answer, either True or False. For example, if we follow just the “yes” transitions in the figure, then we are led to True, which represents that q_1 is 1 when $q=3+3$. Alternatively, if we follow the “yes” transitions until we get to the final $b_0?$, but then follow a no then we are led to False, which represent q_1 is 0 when $q=3+2$.

The key to the success of BDDs lies in the structure sharing. A full binary tree for q_1 would contain 31 nodes, but in the above figure we only have 9. The low number of nodes is because there is no need to duplicate nodes that have the same structure. For example, if we need a carry in the lower bit, it does not matter whether it is because a_1 and b_1 are high or whether a_1 and b_1 are both low.

Note that there is no guarantee that structure sharing will occur. In the worst case a canonical expression can grow exponential in the number of input bits. However, in practice it is usually possible to ensure that BDDs stay within a reasonable size for most problems (multiplication and division being notably difficult).

Also, for the BDD to be canonical a particular variable ordering must be used. The variable ordering is simply the order in which we ask questions in the decision tree. In our example, we used the ordering a_1, b_1, a_0, b_0 but we could have used any ordering and each would produce a different BDD. For example, the following shows the BDD produced by the a_0, a_1, b_0, b_1 ordering:



Note that the structure sharing here is noticeably worse than with the previous ordering. This problem will only get worse if we considered larger summations. The interleaved ordering is in fact the best ordering for addition. Developing a good ordering is a bit of an art, so it is often best to consult an expert if you suspect a better variable ordering may help.

7.1.2 Strengths and Weaknesses of Symbolic Simulation

Like model checking, symbolic simulation has a high level of automation. Some effort is required to develop a good variable ordering, but for the most part symbolic simulation is a fully automated methodology. Furthermore, if two BDDs need to be equal but are not it is possible to construct a counterexample to show why they are not.

Symbolic simulation often scales better than model checking, but not as well as theorem proving. Essentially symbolic simulation works well as long as there exists a good variable ordering for a particular validation problem. Often as the complexity rises the chance that a variable will not fit

well into the existing ordering increases and eventually we hit a point at which BDDs require too much space to be effective. At this point, we generally need to break up the problem into smaller pieces, possibly using automated theorem proving or external reviews to double check that our pieces together add up to a cohesive validation strategy.

A greater weakness of symbolic simulation is that it requires a finite number of input variables. If we look back at our model checking example, even the simple example we showed actually had an infinite number of input variables—one for each time step. Symbolic simulation is an effective strategy for validating straight-lined logic, where we can consider a single time step in isolation but cannot by itself validate a finite state machine that changes over time. Instead, symbolic simulation must be combined with induction or model checking to validate this sort of hardware.

7.1.3 Applications of Symbolic Simulation

Symbolic simulation is widely used within Intel and the industry. It is primarily used by FEV to determine that the RTL matches the back-end design. This technique is usually mandated for all of our designs prior to tape out.

Symbolic simulation has also replaced testing as the primary validation methodology used to validate pipelined circuitry in the execution cluster within our core designs [1]. It has also been used heavily in the validation of Intel-based micro controllers commonly used by IPs, such as the power controller unit (P-unit) within our SOC projects.

There has recently been a trend to replace symbolic simulation with the satisfiability (SAT) solving technique, described in the following section. However, symbolic simulation outperforms solving on some tasks and it is rarely worth replacing working symbolic simulation collateral with new SAT-based collateral.

8 Satisfiability (SAT) Solving

SAT solving is a technique that started coming into common use around 2002 and applies to much of the same domain as symbolic simulation.

The idea behind SAT solving is to convert a proof problem into a common format that is friendly to automation, and then design automated tools that target that format. The format here is conjunctive normal form (CNF), which includes just Boolean variables with the AND, OR, and NOT logic operations. A CNF formula looks something like the following:

$$(a \parallel b) \&& (!b \parallel c \parallel d) \&& (b \parallel !c \parallel d)$$

Here a, b, and c are Boolean variables. A variable or its negation is known as a *literal*. Literals are ORed together to form *clauses* and clauses are ANDed together to create the complete formula. Thus in the above example “!b” is a literal, “!b \parallel c \parallel d” is a clause, and the whole thing is a formula.

Unlike theorem proving, rather than asking whether the formula is valid, in SAT solving the question is whether the formula **can** be satisfied. The question we generally ask a SAT solver in hardware validation is whether a violation exists to a given assertion---such a violation must satisfy all the properties we give it. If a violation is found, then it is presented to the user as a failure. If

the SAT solver finds that there is no satisfying assignment then we know there is no violation and therefore the assertion holds.

SAT solvers have been built to solve CNF problems as fast as possible, with competitions being held to show which SAT solvers (and hence which underlying solving techniques) are fastest and most effective. The result has been that impressively large hardware validation problems are now solvable through SAT solving.

It should be noted that more so than any of the other techniques SAT solving can be largely applied as a black box, without understanding how the tools work. The most important thing to know is that they produce specific counter examples for failing assertions and, like most FV techniques, cannot generally scale past a certain level of complexity.

8.1.1 SAT Solving Example

As an example, let's return to our previous summation:

$$q[1:0] = a[1:0] + b[1:0]$$

Unlike with BDDs, there is no requirement that the problem be written in terms of the input variables. Removing this limitation greatly reduces the size of the expression, largely avoiding the need for the large amounts of memory required by symbolic simulation.

In this case, we can add a new variable c_0 that represents the carry out after adding a_0 and b_0 --in practice this carry out would likely appear as a wire in the circuit implementing q_1 and the FV tool would automatically create it as part of the conversion to SAT. Here c_0 is defined as:

$$c_0 = a_0 \&\& b_0$$

In CNF the above formula becomes:

1. $\neg a_0 \parallel \neg b_0 \parallel c_0$
2. $a_0 \parallel \neg c_0$
3. $b_0 \parallel \neg c_0$

Here we are numbering each of the clauses 1, 2, and 3 for reference. The CNF formula is created by ANDing these together.

It can be helpful to think of each clause as an implication leading to the last literal. For example “ $\neg a_0 \parallel \neg b_0 \parallel c_0$ ” is the same as “ $a_0 \&\& b_0 \rightarrow c_0$ ”. The three clauses together fully define the same if and only if as our definition of c_0 .

With this definition of c_0 , we can now write q_1 as:

$$q_1 = c_0 \&\& a_1 \&\& b_1 \parallel c_0 \&\& \neg a_1 \&\& \neg b_1 \parallel \neg c_0 \&\& a_1 \&\& \neg b_1 \parallel \neg c_0 \&\& \neg a_1 \&\& b_1$$

In CNF it looks like:

4. $\neg c_0 \parallel \neg a_1 \parallel \neg b_1 \parallel q_1$
5. $c_0 \parallel a_1 \parallel b_1 \parallel \neg q_1$
6. $\neg c_0 \parallel \neg a_1 \parallel b_1 \parallel \neg q_1$
7. $\neg c_0 \parallel a_1 \parallel \neg b_1 \parallel \neg q_1$
8. $c_0 \parallel \neg a_1 \parallel \neg b_1 \parallel \neg q_1$
9. $c_0 \parallel a_1 \parallel \neg b_1 \parallel q_1$
10. $c_0 \parallel \neg a_1 \parallel b_1 \parallel q_1$
11. $\neg c_0 \parallel a_1 \parallel b_1 \parallel q_1$

Now let us propose the following assertion:

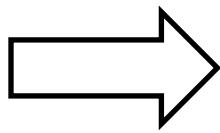
If a_0 and b_0 are both high, then q_1 is high

This assertion is not valid, but let us see how a SAT solver would find a counterexample. To search for a counterexample we need to invert the assertion, since we are looking for a violation to the property. In CNF our assertion then looks like:

12. a_0
13. b_0
14. $\neg q_1$

Or putting the whole problem together we get the following:

1. $\neg a_0 \parallel \neg b_0 \parallel c_0$
2. $a_0 \parallel \neg c_0$
3. $b_0 \parallel \neg c_0$
4. $\neg c_0 \parallel \neg a_1 \parallel \neg b_1 \parallel q_1$
5. $c_0 \parallel a_1 \parallel b_1 \parallel \neg q_1$
6. $\neg c_0 \parallel \neg a_1 \parallel b_1 \parallel \neg q_1$
7. $\neg c_0 \parallel a_1 \parallel \neg b_1 \parallel \neg q_1$
8. $c_0 \parallel \neg a_1 \parallel \neg b_1 \parallel \neg q_1$
9. $c_0 \parallel a_1 \parallel \neg b_1 \parallel q_1$
10. $c_0 \parallel \neg a_1 \parallel b_1 \parallel q_1$
11. $\neg c_0 \parallel a_1 \parallel b_1 \parallel q_1$
12. a_0
13. b_0
14. $\neg q_1$



1. c_0
2. T
3. T
4. $\neg a_1 \parallel \neg b_1$
5. T
6. T
7. T
8. T
9. T
10. T
11. $a_1 \parallel b_1$
12. a_0
13. b_0
14. $\neg q_1$

On the left we see the full problem, presented as a CNF formula with 14 clauses. On the right, we can see the formula after the initial round of simplification by a SAT solver.

Once a SAT solver is presented with the problem it will first note that the singleton literals a_0 , b_0 , and $\neg q_1$ must hold, so any instances of these literals elsewhere can be removed. Clauses with these literals in them can be reduced to True and clauses with the negation of these literals can have the literal deleted. Once, a_0 , b_0 , and q_1 have been confined to clauses 12, 13, and 14 a SAT solver would find that clause 1 is a singleton and carry on the simplification for c_0 .

What we are left with is two non-trivial clauses, numbers 4 and 11. At this point a SAT solver would likely simply pick a value for one of the remaining variables and see if the clauses all resolve to true. If it picks true for a_1 for example, then clause 11 resolves to T and clause 4 resolves to $\neg b_1$. The satisfying assignment is then:

$$a_0=1, a_1=1, b_0=1, b_1=0, c_0=1, q_1=0$$

What this assignment is essentially saying is that if $a=3$ and $b=2$ then $q=1$ (after the overflow), which violates our original assertion:

If a_0 and b_0 are both high, then q_1 is high

8.1.2 Strengths and Weaknesses of SAT Solving

The strengths and weaknesses of SAT solving are largely the same as symbolic simulation. Like symbolic simulation, SAT solving is largely an automatic procedure that only applies to domains with a finite number of input variables. It usually must be combined with model checking or induction to validate properties that cover all time.

Relative to symbolic simulation SAT solving has the strength that no variable ordering is required, which increases the automation since there is no requirement for the user to think about this issue. Also, some problems that have no good variable ordering can be solved by SAT solvers. However, like symbolic simulation, as problems grow in complexity they eventually become insurmountable. In symbolic simulation too much complexity usually presents itself as the machine running out of memory as the BDDs grow in size, but in SAT solving this complexity usually presents itself as the SAT solver not finding any answer after a long search.

One nice feature about SAT solvers is that they tend to find counterexamples well beyond the complexity limits of a full proof. It is common for SAT solvers to come back with a counterexample in less than a second, for example, when a full proof (a statement that the problem is unsatisfiable) requires minutes.

8.1.3 Applications of SAT Solving

SAT solving has been widely applied within hardware validation across industry. It is heavily used within model checkers and has also been replacing symbolic simulation for many FV and datapath problems.

Within Intel we use SAT solvers indirectly when using FPV as the model checkers are using SAT as part of their verification strategy. We have also started using SAT solvers for many of the EXE datapath problems where symbolic simulation used to be the strategy of choice.

8.1.4 Satisfiability Modulo Theory (SMT) Solving

Within the formal verification community there has been an attempt to build on the success of SAT solving to address higher-level validation problems. For example, rather than restricting the problem to Boolean variables, we can consider integer variables and include linear arithmetic equations. In this manner, there have been multiple extensions to the basic SAT solving theory proposed and competitions are held between different solvers that accept these extended theories.

Perhaps the most interesting SMT for hardware verification is the one that includes bit vectors and bit vector arithmetic. This theory directly maps to many hardware verification problems. The advantage to using such a theory is that the solver can make use of higher-level properties, such as the commutative properties of addition and multiplication to simplify problems. Also, the compactness of the problem passed to the solver reduces the memory footprint. These ideas show promise but to our knowledge have not been widely applied within Intel (aside from perhaps being used by model checkers “under the hood”).

9 Summary

In this chapter, we gave an overview of four major FV techniques and provided some guidance as to when they might be applied.

Automated theorem proving is the oldest FV technique and requires a large amount of human effort, but can be scaled to very large hardware designs. While theorem proving is rarely used at Intel the ideas behind this technique are useful to understand and can often be applied when other formal techniques hit capacity constraints.

Model checking by contrast has found much wider applicability within hardware validation. With model checking one can validate assertions or get failing examples automatically. Model checking, however, cannot generally be applied to large designs, such as entire SOCs, due to capacity constraints. For more information on how to apply model checking, read the [Formal Property Verification](#) and [Protocol Formal Verification](#) chapters.

Symbolic simulation is another fully automated technique that has found important applications within hardware validation at Intel. Symbolic simulation can often handle larger designs than model checking, but supports a more limited range of specifications which generally makes it applicable only to datapath-oriented designs without a great deal of feedback. For more information on applying this technique, see the [Symbolic Simulation](#) chapter.

Satisfiability solving is finally another fully automated technique. It can be applied to mostly the same domains as symbolic simulation and has been replacing symbolic simulation in many places as well as generally supporting model checking improvements over the last decade.

In general FV is a powerful set of techniques that when applied to hardware verification problems can often result in a higher quality design than is possible with simulation alone. We feel that understanding some of the theory fundamental to these techniques can be very helpful when deciding when to apply them and in finding solutions for confronting challenges that may arise.

10 Future Work

This section intentionally left blank.

11 References

12. Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir A. Frolov, Erik Reeber, Armaghan Naik: Replacing Testing with Formal Verification in Intel Core™ i7 Processor Execution Engine Validation. *CAV 2009*: 414-429
13. M. Kaufmann and J S. Moore. ACL2: A Computational Logic for Applicative Common Lisp. See URL: <http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html>
14. T. Nipkow, L. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant for Higher Order Logics, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, London, UK, 2002
15. G. Dowek, A. Felty, G. Huet, C. Paulin, and B. Werner. The Coq Proof Assistant User Guide Version 5.6. Technical Report TR 134, INRIA, Dec. 1991.
16. S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE 1992)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748—752, London, UK, June 1992. Springer-Verlag
17. D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148—200, 1998.
18. E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs*, Workshop, May 1981, volume 131 of *Lecture Notes in Computer Science*, pages 52—71, 1982, Springer-Verlag.
19. The Latest Version of the HOL Automated Proof System for Higher Order Logic. See URL: <http://hol.sourceforge.net>
20. W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on System Verification. *Journal of Automated Reasoning*, 5(4):409—530, 1989.
21. J S. Moore. A Grand Challenge Proposal for Formal Methods: A Verified Stack. *10th Anniversary Colloquium of UNU/IIST 2002*: 161—172.
22. Anna Slobodová, Jared Davis, Sol Swords, Warren A. Hunt Jr.: A flexible formal verification framework for industrial scale validation. *MEMOCODE 2011*:89-97.
23. E. Seligman, T. Schubert, and M. V. A. K. Kumar, *Formal Verification*, Morgan Kaufmann Publishers, 2015

The Art of Pre-Si Val: Chapter 27

Mixed Signal Validation

By: [Kalpana Kothapally](#)

1 Abstract

Mixed Signal Validation (MSV) verifies the functional interactions between analog and digital parts of the hardware. Much of this effort centers around different IO interfaces to the chip, PLLs, sensors, voltage regulators etc. within the hardware.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	12/13/2011	Initial Draft	Kalpana Kothapally	Michael Bair

3 Contents

1 Abstract.....	587
2 Chapter Revision History	587
3 Contents.....	588
4 Purpose.....	590
4.1 Why do we need this chapter?.....	590
4.2 What does this chapter cover?	590
4.3 What does this chapter not cover?	590
5 Background Concepts/Terminology	590
6 Mixed Signal Validation	591
6.1 Introduction	591
6.2 Why do we need MSV?	591
6.3 How is validation of analog/mixed signal different from digital domain?	592
6.4 Validation Methodology	592
6.4.1 Guiding Principles.....	593
6.4.2 Simulation Tools	593
6.4.3 Test Environments.....	596
6.4.4 Tests/Checkers/Coverage	597
6.4.5 Documentation and Reviews	598
6.5 Focus during different project phases.....	600
6.5.1 Tech Readiness.....	600
6.5.2 FED.....	600
6.5.3 Execution	600
6.5.4 Post-Silicon.....	601
6.6 Interactions with other teams	601
6.6.1 Architecture Team	601
6.6.2 Analog circuit architects	601
6.6.3 Analog design engineers	601
6.6.4 RTL design engineers.....	602
6.6.5 Other Validation Teams	602
6.6.6 DTS/DA Team	602
6.6.7 Post-Si Teams	603

6.7 Challenges.....	603
6.7.1 Insufficient feature definition	603
6.7.2 Minimal documentation	603
6.7.3 Dependency on design milestones	604
6.7.4 Design churn.....	605
6.7.5 Mixed signal simulation limitations	605
6.7.6 Analog modeling complexities	605
6.7.7 Conflicting requirements	606
6.7.8 Clock domain and power domain crossings.....	607
6.7.9 Difficult/impossible to find some bugs in Pre-Si	607
6.7.10 Skillset challenges	608
6.7.11 Prioritization and Risk Taking	609
7 Summary.....	609
8 Future Work.....	609
9 References.....	610

4 Purpose

4.1 Why do we need this chapter?

Mixed Signal Validation is a relatively new discipline both at Intel and in the larger semiconductor industry. Within Intel and specifically in the CPU world, it has gained tremendous importance since 2005 due to the exponential growth in analog content compared to previous projects. Due to the newness of the activity and the involvement of a much smaller subset of the Design/Validation population in analog/mixed signal, MSV discipline is not well known or understood by the larger Validation and design audience. The primary purpose of this document is to demystify MSV in a language that a non-analog/MSV person can understand. It can also serve as a good introduction to any engineer new to MSV or to a project team that is building MSV effort from ground up.

4.2 What does this chapter cover?

This chapter will introduce the concepts of MSV, highlight the similarities and differences from other Validation disciplines and explain the role of MSV in the overall design development cycle. Most of what is documented is heavily biased towards the methods and practices used in the major CPU projects. Chipset, SOC and other IP development teams at Intel use similar, but not exactly the same methods.

4.3 What does this chapter not cover?

This chapter will not go into any technical details of different analog circuits or describe any specific tools in use by MSV. This is also not a tutorial on how to do MSV.

5 Background Concepts/Terminology

- Analog – any part of the chip that does not operate like a simple on/off transistor.
- FEV – Formal Equivalence Verification.
- Behavioral models – Representation of a circuit using a hardware description language like Verilog, System Verilog, VHDL. Verilog-AMS and VHDL-AMS are equivalent languages for describing analog circuit behavior.
- Digital validator – Validation engineer who is responsible for validation of purely digital logic (anybody working in all other disciplines of Pre-Si Validation except MSV)
- Mixed Signal validator – Validation engineer who is responsible for validation of analog and mixed signal portions of the design

6 Mixed Signal Validation

6.1 Introduction

Mixed Signal Validation focuses on Pre-Silicon validation of interactions between analog and digital components. The primary goal is to find logical/functional bugs in the implementation of mixed signal features. As a result, most of the bugs found by this discipline are in digital logic that is closely interacting with analog portions of the machine. In the process, some bugs maybe found in the analog circuits, but this is usually a small percentage of the total bugs found.

Typically, this validation is concentrated in the IO units (aka PADs or AFE or electrical layer), clocking/PLLs, sensors, voltage regulators and other such units in the machine. This includes validation of DFx hooks designed for debug and survivability of analog and mixed signal features. All these units/features are part of the ‘unCore’ portion of the design.

The actual scope of what all is covered under the MSV umbrella may vary to some extent across product teams at Intel. Most of the differences stem from the organizational differences between Design and Validation. For example, in some teams, designers might own MSV as an extension of analog validation or RTL design. In other teams, often in the larger CPU teams, there may be a team focused explicitly on MSV. Even in the latter case, the ownership line between what is covered by MSV team versus the rest of validation teams in the project may vary due to staffing/skillset reasons. As long as the ownership lines are well established and understood by all the key players in the activity, the likelihood of bugs falling through the cracks is small.

6.2 Why do we need MSV?

During the past 5-6 years, specifically since the Nehalem family products, the number and complexity of on-die analog circuits in CPUs has increased rapidly in order to enable Intel to produce products that support higher IO speeds and deliver much more uarch capability at significantly lower power. Analog/mixed signal interactions also continue to grow as we integrate more and more chipset/platform functionality onto a single die or package to reduce platform and development costs. CPU designs from Intel are often on the leading edge of process technology – while this is a tremendous advantage for Intel, it also poses significant challenges in the design of analog circuits since they are very susceptible to process changes. This further increases the interactions between analog circuits and digital logic in the machine due to the addition of new state machines to compensate for process, voltage, temperature (PVT) variations.

Often, bugs in these areas tend to be significant limiters for Post-Si Validation. Nearly all mixed signal features in the machine play a very important role in initializing the machine (i.e., during reset sequence). Bugs in these features can prevent the machine from even fetching a single instruction from memory thus preventing the machine from even booting. They are often very difficult to debug and reproduce from part to part due to the varying characteristics of the system platforms and due to impact of PVT variations on analog circuits. Bugs that can prevent boot are usually difficult to work-around using microcode or pcode patches. All of this has a direct impact on time to PRQ. With such a significant risk to PRQ, we need a systematic Pre-Si Validation approach to flush out design/architecture bugs and make sure these portions of the CPU work

properly on silicon. MSV team serves precisely this purpose. Apart from this role, the team is also capable of supporting Post-Si debug activities involving debug and resolution of silicon sightings.

6.3 How is validation of analog/mixed signal different from digital domain?

In order to understand and appreciate the nuances of the rest of this document, it is important to understand a few key differences between purely digital vs analog designs and how they impact the design and validation process.

- **Golden reference:** In digital design, RTL is the golden reference. In the analog domain, circuits are often the golden reference. Typically, schematics are drawn, simulated using a spice simulator and then RTL may be created using the schematics as a reference.
- **Formal Equivalence Verification:** In digital domain, all transistors are created from the RTL. Validation is performed on the RTL. FEV tools compare the RTL against the auto generated/synthesized or hand drawn schematic netlists and prove that the transistor level implementation matches the RTL level representation. Validators rarely think about whether or not the RTL is matching schematics. However, in the analog world, there are no tools to FEV the RTL against analog circuit.
- **Black boxes:** Nearly, all verification tools treat analog circuits as ‘black boxes’ – i.e., they do not understand the nuances associated with the internals of analog circuits and often stop at verifying only at the interfaces. As a result, there are many ways to miss problems in the analog domain. This can also happen for some special digital circuits, but, for the most part, almost all digital circuits have full coverage in nearly all domains (logic, timing, schematics, layout etc.). Even gate level simulation tools like LVR black boxes because the tools have not yet scaled to handle complex analog circuits.
- **Science vs Art:** Digital design generally follows a repeatable set of processes to enable development of quick, large scale designs that scale well with process. In contrast, most analog design is still done as an art – ie., there are few set rules of what is and isn’t allowed giving a great deal of freedom to the circuit designer to create a circuit topology that best meets the often competing, stringent power, area, performance goals. This freedom also leads to many exceptions to rules in the analog domain needing custom or/and manual verification solutions which tend to be effort intensive and error prone.

These differences lead to differences in MSV methodology compared to other validation methods and highlight the need for a slightly different mindset for a mixed signal validator compared to other validation domains. In addition, these differences make bugs in the analog domain easily escape to silicon.

6.4 Validation Methodology

Learnings from multiple projects have established that using methodical and repeatable validation processes plays a key role in consistently delivering high quality results. At Intel, this is the main

reason validation has evolved as its own discipline and separated from the design/development process and, in many product development teams, it is its own separate organization. This is true for MSV as well. The differences in analog vs digital design have made it difficult to realize consistent high quality results. In the recent years (specifically, since 2005), MSV has evolved to leverage all the goodness of established digital validation practices with customization to meet the unique needs of validating mixed signal features.

In many respects, mixed signal validation is like uAV (see [Dynamic Microarchitecture Validation](#)). Both teams focus on a combination of white box and black box testing to find functional design bugs. They need in-depth understanding of the micro-architecture and need to understand the details of feature implementation in design in order to exhaustively validate it and flush out implementation bugs.

The following sub-sections briefly explain the philosophy, methods and techniques used by MSV. Any validator familiar with uAV methods will notice where there are similarities and differences in the two approaches.

6.4.1 Guiding Principles

Here are the basic principles that lay the foundation for MSV.

- Use methodical approach to ensure rigorous validation
 - Detailed test plans, focused tests, checkers, test plan reviews, exit reviews
- Expose analog/mixed signal content as much as possible to all other validation/verification domains to increase Pre-Si coverage
 - Expose all digital design embedded around analog blocks to FEV
 - Model as much analog circuit behavior as possible in RTL
 - Reuse digital validation TEs, tests, checkers as much as possible
- Make sure validation process is repeatable and regress-able
 - Automate and deploy weekly regression runs to ensure newly introduced problems don't go unnoticed
 - Make sure pass/fail conditions are coded for every test case, including those for analog circuit behavior
 - Identify and document behaviors which can be checked only using visual inspection (i.e., cannot be automated) in test plans
 - Automate/eliminate any manual tasks to reduce human errors
- Learn to pick the right priorities to maximize ROI
- Many reviews with experts and Post-Si stakeholders to minimize escapes
 - Analog model reviews, checker reviews, simulation reviews

6.4.2 Simulation Tools

Unlike the rest of Pre-Si validation, MSV uses a combination of digital, analog and mixed signal simulation techniques.

6.4.2.1 Pure digital simulation (aka RTL based validation):

The entire design, including all analog components in the machine, is modeled in RTL in Verilog/System Verilog/VHDL and simulated using VCS and other digital simulation tools like the rest of validation. Mixed signal validators and analog RTL coders work together closely to fully model the behavior of the analog circuits in RTL. It takes practice, good understanding of analog circuits, creativity and a strong validator mindset to come up with effective ways to represent the circuit behavior digitally. In the rest of this document, validation of mixed signal features using pure digital simulation as often referred to as RTL based validation.

There are many advantages of this approach:

- Helps find bugs in A \Leftrightarrow D interactions before schematics are ready and FEV clean
- Can get validation coverage for mixed signal features through other validation team's test runs
- Enable digital/RTL validation of features that are impractical to validate with schematics
- Enable exhaustive validation of features owned by others which have interactions with IO/CLK
- Prevent RTL \Leftrightarrow silicon mismatches on tester
- Enable finding bugs in test cases that MSV did not think of, but other teams may be targeting through their tests
- Can inject non-ideal behaviors of the devices, chip input/output signals (like jitter, duty cycle distortion, attenuation) and flush out bugs faster

The pitfall with using this simulator is since there is no FEV between the analog RTL model and schematics, there is no guarantee that the model being used to validate the design is faithfully representing the analog circuit behavior. Validation can miss some design bugs due to this.

There is also at least one significant side effect of applying this approach. Simulations can slow down for all users of the digital model due to the low level modeling of analog circuits, especially those related to anything that is sensitive to high frequency clocks.

6.4.2.2 Mixed Signal Simulation:

Digital logic is modeled in RTL in Verilog/System Verilog/VHDL. Analog circuits may be represented as transistors (using schematics) or Verilog-AMS or VHDL-AMS model. These two different pieces can be simulated together under a mixed signal simulation engine like Mynx (developed by DTS) or an external vendor tool like Discovery AMS or Cadence AMS. Mixed signal simulation tools integrate a digital simulation engine with Spice engine. There is flexibility in these tools that allows different analog circuits to be represented as a combination of RTL, schematics and V-AMS models. It is important to choose a mixed signal simulator that supports the same digital simulator as the rest of the validation and the same analog simulator as the circuit design team in order to minimize differences in simulation results due to tool nuances.

Since mixed signal simulator allows us to simulate digital logic with the golden representation of analog circuit, it helps eliminate any misses due to mis-modeling of the analog circuits in RTL or Verilog-AMS/VHDL-AMS. It can also be treated as a stop-gap solution for lack of FEV for analog circuits. However, the coverage provided by this is not as exhaustive as FEV is for digital circuits.

However, there are many difficulties with using this simulator:

- Special build tools are required to build the simulation DUTs that can run on the mixed signal simulator.
- RTL needs to satisfy some requirements to build a mixed signal simulation DUT.
 - Interfaces must match between schematic and RTL for drop-in replacement
 - RTL and schematics for the surrounding digital design must be FEV clean
- This requires building multiple DUTs to cover different test cases due to simulation performance issues involved with replacing all analog circuits of interest with Spice or Verilog-AMS equivalent models.
- Simulations through a mixed signal simulator often run many times (10-100x) slower than an equivalent pure digital representation. Any onion peeling in these simulations takes up significantly longer iterations than in other domains. As a result, validation coverage that we can obtain through this simulator is limited.
- Debugging mixed signal simulations can be twice as hard as debugging a standalone digital or analog circuit simulation due to the interactions between the 2 simulation engines.
- Special regression tools are required to automate the runs on multiple projects/steppings.
- Analog circuit simulation results are difficult to check with automation.
- Tools are still quite immature. Validation activities often expose tool issues during execution that take long time to address which is a significant impediment to completion of validation on schedule. In addition, workarounds for tool issues often require multiple experimental runs that are time consuming and wasting precious engineering time.

6.4.2.3 Analog/Circuit Simulation:

Analog circuits are simulated with transistor level representation using a spice engine like Lynx (developed by DTS) or Nanosim (developed by Synopsys) or Spectre (developed by Cadence). This is generally in use to validate functionality and performance of each analog circuit in isolation. It can also be used to simulate multiple circuits (analog and digital together), but simulation speeds tend to be very slow limiting the application of this method for such cases. Analog circuit designers are the primary customers of such simulation tool.

Recently, external tool vendors have invested in tools like XA (developed by Synopsys) that trade off simulation accuracy for speed. Such tools are useful to validate small scale mixed signal circuits. MSV can collaborate with the analog design team or run simulations using these tools where appropriate. This can serve as a good alternative to the effort intensive mixed signal simulations in some cases.

6.4.2.4 Summary

In summary, due to the significant advantages described above, MSV engineers prefer to run as many test cases as possible using RTL based validation techniques – i.e., they prefer running simulations on a fast digital simulator over the slow mixed signal simulator. However, this will not be sufficient in all cases. Each validation engineer will need to choose the simulation engine that is most appropriate for each test case. This requires a good understanding of what is being

covered through the RTL simulations and ability to identify what is not modeled in the RTL for analog circuits, what behaviors are not captured, the interactions of these behaviors with digital logic and the impact of not testing these interactions using mixed signal simulator.

6.4.3 Test Environments

MSV teams across Intel use the lowest level environments possible that are sufficient to thoroughly validate a feature. The actual environments may however vary for various reasons. This section attempts to list all the known methods in practice and the pros and cons of each method.

- **Reuse digital validation environments:** MSV use the same cluster and fullchip test environments as the rest of validation. These environments can be in any language – Specman/E or System Verilog or Perl/Mace etc. For example, MSV can use PCI-E cluster test environment (with the BFM playing the role of the PCI-E card/device) to validate all the mixed signal features and DFx hooks associated with IO/AFE/electrical layer of PCI-E. Or, MSV can use uncore or fullchip test environment to validate PLLs. Additional hooks/capabilities are included in the TEs to support any mixed signal specific test case validation. Some examples of these mixed signal specific test capabilities include injecting duty cycle variation or jitter on the input clock and data pins, introducing a slope on supply/voltage signals, generating clocks with real silicon frequencies, and having knobs to run simulations with different representations of analog models (schematics vs RTL). Often, there are knobs in the TE to enable these capabilities only for MSV runs. The biggest advantage of reusing digital validation environments is it enables MSV team to take full advantage of RTL based validation techniques by getting coverage for mixed signal features through the rest of the validation teams' test runs. This can have significant effort savings for MSV and helps find bugs that MSV may miss due to unforeseen interactions with the rest of the machine.
- **Custom test environments:** We can develop test environments around the mixed signal portions of the design. For example, we can create a SVTB or Specman/E environment just for the PCI-E IO/AFE or a SVTB to test the PLL modes of operation. This is analogous to creating a unit level test environment for every single unit testing in the digital domain. We can create them quickly, allow fine grain control for stimuli and provide fast turnaround times for validation. Validation can progress even if the surrounding units are not healthy. But, they also have pitfalls - the more custom environments we create, the more effort it takes to create and maintain them. More importantly, we need to validate the assumptions made by such environments.
- **Extract test vectors:** Another approach is to extract the test vectors from the digital validation runs and feed them as stimuli to circuit simulation or mixed signal simulation tools. This is sort of a poor man's validation strategy and commonly used by small design teams where the designers are responsible for validation. The advantage of this approach in such a circumstance is the designer doesn't need to learn about what can sometimes be complex validation environment. But, there are many pitfalls with this approach. The whole process tends to be manual, error prone and not easy to regress. It is difficult to debug any issues and there is a higher risk of misdiagnosing problems due to miscommunication of issues between the 2 people involved.

All the above three environments can be used for both RTL based validation using digital simulator and mixed signal simulations. SVTB based custom test environments are generally easier to get working in mixed signal simulation without running into many tool issues. Cluster/Fullchip TE based environments also work fairly well for mixed signal sims, but they also tend to impose constraints that take more effort to work-around and enable in a regression friendly flow because we need to make sure changes we make for mixed signal sims don't break the rest of the validation teams' runs.

With all these choices, there is one major risk that is unique for mixed signal feature validation. In digital domain, for most test cases that are run on the cluster/unit test environments, we also get at least breadth coverage for them at the higher levels through fullchip simulations or test runs on emulation model. There are established checkers like archsim that can expose failures if something is missed by a lower level checker or TE. Also, if the unit validator misses a bug, there is a chance that some other validator might find it because well written random tests stress all parts of the machine. That safety net is not automatically present for mixed signal features. Often, the analog circuit modeling support that is needed in the RTL and hooks in the TEs for validating mixed signal features can increase test run times and slow down the simulation significantly. For various reasons, most motivated by the drive for higher simulation performance, most digital validation engineers do not want to pay this penalty in their runs. Emulation models also tend to remove anything related to analog because most analog RTL models are not synthesizable. As a result, if MSV misses a bug in the environment they are using, that bug will most likely just escape to silicon.

6.4.4 Tests/Checkers/Coverage

In terms of exciting digital signals, tests created by MSV are not different from any other digital test content. When it comes to exciting analog signals, for both RTL based validation and mixed signal simulations, in order to see voltage changes between 0 and 1, real number support in Specman/E or System Verilog can be used. For mixed signal simulations, if there is a need to modify the slope/slew rate of a signal, specifying the stimuli in Verilog-AMS or Spice is more common.

Similar considerations are required for checkers as well. If the signals are visible in digital simulator, then, there is nothing unique in mixed signal domain. However, if we need to check the correctness of an analog signal in mixed signal simulation, we can digitize the value and apply a digital language checker or use Verilog-AMS/Spice based checkers. Creating analog value checkers however is difficult to get right without causing many false failures. So, just like creating analog models for RTL signals, this takes experience and good understanding of what tradeoffs are reasonable to make.

Coverage space for mixed signal features is often not very big. Randomness required in the test content is unrelated to the micro-architectural event combinations that we often want to see in functional/pure logic validation. Instead, it is more related to the various knobs that can impact the operation of some FSMs or different operation modes of some circuits. As a result, most mixed signal feature related stimuli tend to be directed or directed random. Coverage based validation methods are therefore not worth the ROI. However, whether or not a coverage feedback tool is

required depends on the complexity of the digital logic related to mixed signal features increases in the future. Even if we used coverage tools, it would be limited to getting feedback on digital stimuli. There are no tools to support coverage for analog signals and there is no need either for such a capability.

6.4.5 Documentation and Reviews

As mentioned previously in this document, if MSV misses finding a bug, the probability of finding the bug accidentally by other validation efforts is very small unless explicit steps are taken to maintain that safety net. MSV team can miss a bug for various reasons – incomplete understanding of the specs, missing or incomplete feature specification, missing test cases, incorrect modeling of the analog circuits, incorrect/missing checks for verifying the proper behavior and insufficient mixed signal simulation coverage to list a few. As a result, technical content reviews of MSV collateral play a very important role in minimizing silicon escapes.

6.4.5.1 Test plan documentation/Review

To ensure minimal escapes due to miscommunication between Validation and Design/Architecture/Post-Si, mixed signal validators should pay special attention to clearly documenting test plans. It is highly recommended to include the following information in the test plans.

- Scope of the document to highlight what is covered under the MSV umbrella vs other teams
- Introduction to validation strategy and test environments used by the test plan
- For each feature covered by the plan, list the following
 - Description of the test case
 - Simulation environment – RTL vs mixed signal simulation
 - Simulation DUT
 - Criteria for pass/fail – i.e., how will the expected behavior be checked?
 - If a checker cannot be coded/automated, it is important to highlight that the correctness of the behavior will be verified by inspection. There may be more than a handful of such cases in MSV and therefore important to highlight in the test plan.
 - For DFx related test cases, it is useful to list the usage model assumptions.
- Analog model validation – this section should list how validation will make sure that any RTL or other behavioral models used in place of analog circuits during validation will be verified against schematics.
- Document any limitations with the coverage provided by the plan

A detailed walk through of the entire test plan with all the stakeholders is an important step in ensuring validation is adequate. Reviewers must include all the experts and stakeholders for the features – typically, the review list includes circuit architects, analog designers, RTL coders, validators of neighboring units even if they are outside MSV and Post-Si representatives. Reviewers should analyze the document to ensure the list of test cases is complete. Validator should solicit feedback on the validation strategy for each test case and clearly highlight any

limitations with that approach. They should also make sure the test plan captures the correct checks for expected behavior for each test case to ensure no bugs escape due to incorrect or incomplete checking. At the end of the review, all the reviewers should have a clear understanding of exactly what is and is not going to be covered by the validation team.

Test plans should be developed as soon as the validator has completed some basic testing of the features to develop enough understanding of the design and the features. Reviews should be held early enough in the project to identify and correct any issues related to expertise gaps or staffing issues and also to ensure validator has enough time in the schedule to respond to any surprising increases in validation scope.

6.4.5.2 Analog model review

Since there is no automatic method to compare behavioral models for analog circuits against schematics, the quality of MSV result may suffer if the models are inadequate and not faithfully representing behavior of analog circuit. Even though mixed signal simulations may be run to precisely cover this gap, they may not be exhaustive and might be run very late in the project leading to late bugs in the design. Thus, an important step in minimizing such a risk is to hold detailed reviews of each model with the analog circuit designer, validator, circuit architect and the RTL developer for the model and surrounding digital logic. It can be very useful to document a review checklist that can be used as a guide to go through each model in the design and make sure it meets all requirements. MSV team should take ownership of making sure such reviews happen before any major tapein. Such reviews are also important when teams reuse analog IP from other projects to make sure the analog models are still sufficient to meet the validation requirements of the features on the consuming project. These reviews are also a good spot to make sure that no digital logic is embedded inside an analog RTL model to eliminate escapes due to lack of FEV on the digital logic. And, if there are exceptions, review can help ensure these exceptions get special attention through exhaustive circuit simulations. It is important to take the time for these reviews after initial development and also after the analog circuit design is frozen for tapein.

6.4.5.3 Simulations review

There are many benefits to taking the time to review simulation results for some critical test cases with the design/arch experts. Such reviews are particularly useful for mixed signal simulations where the validators may not be fully aware of the in-depth operation of the circuits and may miss any weird behaviors that might get noticed only if the waveforms are visually inspected with a fine comb. Often, some of these nuances do not cause simulation failures because the bad behavior doesn't propagate or get exposed to a failure point. It is also useful to review waveforms from digital simulations at least for the basic power up and power down sequences to ensure the simulations are being run with proper test flow. This helps to ensure that there are no undesirable simulation speed up hacks and that the correct usage sequence is used to test DFx hooks. This also helps to verify that there are no bugs escaping due to incorrect checkers. Such reviews are most useful when the simulations are run for the first time and on the final tapein model for any major product.

6.4.5.4 Exit review

This is the last opportunity in the project cycle before tapein to ensure correctness and completeness of MSV. The goal of this review is to make sure that all tapein gating validation activities have been completed and anything left unfinished is either not important for this tapein or has low risk of leading to a fatal bug on silicon. All the stakeholders who participated in the test plan review are expected to weigh in during these reviews. Ideally, this review should be held when validation team believes all tapein gating validation has been completed. However, there are advantages of holding this review when around 95% of the validation is complete to help the design team have an insight into the risk to their database closure schedule created by the remaining validation work.

6.5 Focus during different project phases

The following section describes the focus of MSV team during the key phases of product development (see [The Life of a Project](#)). This is generally most relevant for a lead product development rather than the shorter development cycle of proliferations.

6.5.1 Tech Readiness

Similar to other validation teams, most of the focus of MSV team during this time is on defining the validation improvements for the next project. They also work with the analog circuit architects to evaluate new features and where possible, define the micro-architectural implementation of some of the features.

6.5.2 FED

While the RTL team focuses on developing RTL for the 1st time during FED, MSV team develops RTL models for analog circuits and validation collateral to support RTL based validation of mixed signal features. Goal of the team is to flush out basic bugs in the design as quickly as possible. At the same time, team starts creating execution test plans that will form the blue print for all execution validation activities. Team also pioneers mixed signal simulation flow with at least a couple of test cases to ensure all the required tools are in place for applying this technique on a large scale later.

6.5.3 Execution

During this phase, focus shifts from basic bug finding to exhaustively validating every piece of the design. Test plans are reviewed with all the Pre-Si and Post-Si stakeholders. Completion of most of the RTL based validation activities is the primary priority. Once the design team completes the 1st FEV milestone with the fully developed RTL (usually defined as completion of CDR1 in most projects), MSV team is enabled to start all the mixed signal simulation activity. Both mixed signal simulation and digital validation activities need to be completed by the time physical design is ready for tapein.

6.5.4 Post-Silicon

After 1st silicon tapein, validation engineers move on to working on different steppings/proliferations of the project while also supporting any Post-Si debug requests. When it comes to Post-Si debug opportunities, MSV engineers have an advantage over most of their design counterparts. They might be the only engineers on the project who have a good high level understanding of the operation of the features along with the knowledge of lowest level details. It is important that the MSV team be able to run all the content they developed on the A0 stepping of the product on all successive steppings to make sure there are no bugs introduced due to changes. It is for this reason that all simulations – digital and mixed signal, must be developed to be regression friendly with as little manual effort as possible.

6.6 Interactions with other teams

Due to the special nuances associated with validation of mixed signal features compared to digital logic, MSV engineers need to interact with several different teams in the product development process.

6.6.1 Architecture Team

Unlike most other validation domains, there is actually very little interaction between the logic architecture team and MSV team. The only time they may interact is to understand the architectural implications of a bug or a design change in the analog domain.

6.6.2 Analog circuit architects

Experienced analog design engineers are generally the architects for mixed signal features on our products. They usually take a high level system spec (like DDR3 spec from JEDEC), translate that into arch/uarch/circuit requirements and establish/document the micro-architecture spec for mixed signal features. MSV engineers closely interact with these experts on anything related to how a mixed signal feature is supposed to work.

6.6.3 Analog design engineers

These engineers define the topology of analog circuits used in mixed signal features and implement the schematics for them. They own making sure that the analog circuit operates in isolation to meet functional and performance/robustness requirements using Spice simulators. MSV engineers engage the designers in reviewing the RTL models for analog circuits, taking the schematic netlists from them and simulating them with other analog/digital circuits using mixed signal simulator and debugging any simulation problems. There may be cases where the validation and circuit engineers work together to simulate analog circuits by extracting stimuli from the RTL instead of running a mixed signal simulation.

6.6.4 RTL design engineers

Similar to any other validation domain, MSV interacts the most with RTL engineers. They work with them almost on a daily basis especially during the FED stage of the project to exercise mixed signal features. MSV engineers partner with the RTL coders to model analog circuit behavior in RTL often taking ownership of the development of these models and helping them maintain these models when the analog circuits change during the project execution.

6.6.5 Other Validation Teams

6.6.5.1 uAV

Since MSV strives to reuse the test environments and tests/checkers created by uAV team to get coverage for mixed signal features, they interact heavily with uAV engineers. The uAV team helps the MSV team put in the hooks needed for MSV while the MSV engineers help uAV with debug of failures related to mixed signal features.

6.6.5.2 PMV

Similar to uAV, MSV team prefers to reuse power management test content to validate interactions of mixed signal features in power management flows. Thus, the two teams interact with each other to share the content and debug expertise.

6.6.5.3 DFTV

Interactions between MSV and DFTV are limited to ensuring full coverage of TAP controls for mixed signal features by sharing the validation effort.

6.6.5.4 Reset Validation/TE developers

Since most mixed signal feature validation is focused on what happens during reset phase of the machine operation, MSV needs support from the reset test environment developers to add in necessary hooks to test mixed signal features. MSV also works with the reset validation team to ensure that the high level reset flow being emulated in the TE matches the actual spec expected by the mixed signal design.

6.6.6 DTS/DA Team

Mixed signal simulations require special tools for DUT builds, test runs and regression runs. DA team provides this support. MSV team works closely with the DA team to debug and resolve blocking simulation tool issues and to pioneer/develop new tools for the future.

6.6.7 Post-Si Teams

6.6.7.1 Pre-Si Engagement

Engineers from Post-Si, specifically the HVM/DV/EV teams, have a stake in participating in Pre-Si design and validation activities in order to develop the knowledge required to create Post-Si test content and debug content/silicon issues. MSV team helps the Post-Si engineers with developing the required collateral in Pre-Si domain so they can validate their usage models and also extract test vectors for use on the tester. Sometimes, they prefer to develop this expertise by having key engineers do a tour of duty to Pre-Si team. It is mutually beneficial to support such a request and take advantage of a ‘free’ resource in the MSV team.

6.6.7.2 Post-Si Debug Support

Involvement from the MSV team in Post-Si debug activities in most teams is currently limited to supporting sighting debug by theorizing failure root causes or by trying to reproduce behaviors observed in Post-Si in Pre-Si simulations.

6.7 Challenges

All Pre-Silicon validation teams deal with many challenges during project execution. Experience shows that MSV team deals with several unique challenges. As a result, MSV team deals with significantly higher schedule pressure and validation completion becomes one of the long poles to tapein.

6.7.1 Insufficient feature definition

Often, mixed signal features are defined at a very high level during TR stage of the project. There is usually not enough detail baked in yet to start RTL implementation of the feature using that definition. In the best case, feature details are fully defined and documented by the experts before RTL coding starts. More often, such details are hashed out in the process of RTL coding and testing. As a result, RTL development and testing take much longer than originally planned squeezing both physical design and validation schedule.

6.7.2 Minimal documentation

There is universal agreement that written specs like HAS/MAS can greatly accelerate validation progress. In spite of that, most product teams at Intel attach little importance to creating reliable and the correct detail of documentation irrespective of how complex or new the feature/design changes are. In the MSV domain, the impact of poor documentation is immense and can lead to significant validation quality issues especially if the validator is new or inexperienced in this domain.

The following are examples of poor documentation impacting MSV:

- The PCI-E external spec specifies that receiver equalization may be required to meet the electrical performance targets. They intentionally leave the implementation of this feature to the product team. Unless there is a micro-architectural document that describes the mixed signal control loop and underlying algorithm, it is an uphill task to do proper validation of this feature. Problems in such an algorithm will not naturally show up as data corruption failures – so, unless the validator puts in explicit checkers in the environment to verify correctness of the algorithm, design/algorithm bugs will escape to silicon. In the absence of a documented spec, validators are left with no alternative, but to use the RTL/analog design as the spec. The pitfall with this approach is unless you have a validator with the right expertise and can envision how something will work on silicon, validation content may have the exact same bugs as design and they will go undetected till silicon.
- DFx feature often lack a description of the usage model of the feature. They are implemented in the design to enable margining of analog circuits or support other DFx test methods like debug hooks, burnin etc. The complete way to validate these is to simulate the post-silicon usage model of these hooks. If these usage models are not documented, the validator will have to make assumptions of how something might be used which may not be accurate and lead to a broken DFx feature on silicon.

To make matters worse, there is a strong desire across Intel to reuse analog/mixed signal IP to eliminate PRQ delays due to reinventing the wheel. As a result, the impact of lack of specs can be seen over multiple generations of the IP reuse if the IP reuse teams misunderstand any assumptions about the IP. This has a multiplicative negative effect.

6.7.3 Dependency on design milestones

Validation of mixed signal features requires both RTL (for digital logic) and analog circuit schematics (for RTL modeling of analog circuits and for mixed signal simulations). The 1st dependency, relying on RTL models, is the same for all validation teams. But, the dependency on schematics adds an extra layer of potential delays to validation execution.

- For example, analog circuit schematics are often in flux until the end of CDR0 phase of the design which often lines up about 3 quarters into the FED phase of the project. This means reliable models of analog circuits cannot be fully developed until well into FED. Due to the same reason, low level details of mixed signal features don't get finalized until this point. This means most features cannot be tested until towards the end of FED.
- Another dependency is with the FEV milestone which lines up with design's CDR1 completion. In order to run mixed signal simulations where schematics are simulated with RTL, design must be FEV clean at least to the extent of making sure the interfaces of the analog circuits and the schematic behavior of the surrounding logic matches with the RTL. Otherwise, effort will be wasted in debugging issues that will go away once RTL and schematics are lined up.

In recent projects, the window between CDR1 completion and tapein is getting rapidly reduced to 3-6 months window. This directly eats into the schedule for mixed signal simulations completion. Unless RTL based validation is completed early, it is difficult to complete both activities before tapein.

6.7.4 Design churn

While there is a strong desire among all involved to identify design changes as early as possible in the product development cycle (namely during FED), analog design tends to go through a significantly higher rate of change compared to other parts of the design, throughout the life cycle of the project. In most cases, they do not change the micro-architecture significantly. However, they end up being several small changes in analog circuits due to process learnings or improvements based on circuit simulations, discovery due to incorrect assumptions made for global features, insufficient DFT/HVM hooks, enhancements for post-si debug capabilities etc.. Unless there is a proper communication/scope management process in the project, validation might end up seeing these changes only when the project FEV milestones are completed which is too late to respond/adapt. One way to minimize surprises is by requesting design team to document any changes in change tracking database like HSD as early as possible in the project timeline and by imposing stricter change control as tapein approaches.

6.7.5 Mixed signal simulation limitations

Apart from the difficulties associated with mixed signal simulations mentioned in section 6.4.2.2, there are other limitations with the usage of this tool.

- Since these simulations need special DUTs to be built and take a very long time to simulate, it is impossible to make mixed signal simulations as part of the gating regression list that must be run before any design changes are committed to the central RTL repository. As a result, it is often difficult to prevent changes in the design that can break mixed signal simulations. This implies much wasted effort for validation in maintaining mixed signal simulation regressions even for simple problems that should not have been accepted into the repository in the first place.
- Mixed signal simulation performance is likely not going to scale to the level required to simulate very long sequences like package state power management flows or the full reset sequence of a machine where analog circuits play an important role. The only solution is to break down the problem into smaller simulations and get at least some coverage. But, often, even this is not practical to achieve since the shortest simulations tend to run for at least several days.
- There is no mixed signal simulation tool in the industry that satisfactorily meets all the needs of MSV. A tool that provides significant performance may not support some other key capability. If there is indeed a tool that meets most mixed signal requirements, it may not use the same digital simulator as the rest of validation or the same spice engine as the analog spice engine. In such scenarios, it is impossible to make the larger digital validation team or analog design team move to a different simulator primarily to benefit MSV since this may mean loss of other important capabilities for these other teams.

6.7.6 Analog modeling complexities

When developing RTL for digital design, the primary concern of the engineer is to ensure the functionality of the design is correct. An expert engineer may consider how the simulation performance is impacted by a particular coding style, but this is generally limited to few design

blocks like caches or arrays. Automated tools like lint enforce model conformance to clearly defined rules. In contrast, RTL for analog circuits is more hand crafted and automated checks are coming online slowly. Several factors need to be considered while developing a robust and exhaustive model.

- Which behaviors of the circuit are worth representing in RTL?
- What is the best possible way to model these behaviors?
- Will it be sufficient to find bugs in interactions with other circuits?
- How can one expose failures in simulation if a specific behavior is incorrect?
- What is the performance impact of the particular modeling approach? If it slows down significantly, how can we model it differently to avoid the penalty without significantly losing validation capability?
- Is the model friendly for emulation use? If not, is there a simpler/different model that is synthesizable on hardware emulation for use by the rest of validation?
- Is the model compatible with mixed signal simulation tools?
- Should any assertions be coded to verify assumptions of the interfaces or of the model itself when it is integrated with the rest of the design/other analog circuits?

Since the quality of RTL based mixed signal validation is largely dependent on the completeness of the analog models, it is critical to pay close attention to all these requirements. The challenge however, is most of this insight and expertise is acquired over time through hands on experience. It is also often very time consuming process since it may take several experiments before the best model is developed. In addition to that, the skill-set required (combination of analog circuit knowledge, validation concepts, RTL modeling languages, signal processing/mathematical analysis) is also unique which only a few engineers possess in an organization.

6.7.7 Conflicting requirements

Goals and capabilities required by MSV are often in contradiction with what other validation teams want to see. For example, MSV prefers to expose more analog/mixed signal content to other validation teams to get coverage. But, other validation teams do not want this if it causes an increase in test run times or slower simulation performance. At times, the only way to satisfy other team's requests may be to fully compile out portions of the design which then increases the risk of escapes. Also, the simulation sequences that MSV is interested in are sped up in Pre-Silicon simulation to reduce the time spent in what is seen as generally uninteresting to most other validation efforts. These speed up hacks also eliminate any source of coverage gains from others' runs and shift the responsibility of ensuring hack free simulations back in MSV team's domain. With digital validation efforts also suffering due to model slowness issues due to the rapidly growing size of the CPU models these days, it is increasingly becoming a challenge to develop one model that satisfies the needs of the MSV team and the rest of validation.

The MSV team needs to account for the additional effort required to maintain at least two variations of models. Also, special attention must be paid to ensure that MSV test content gets run on the right model to eliminate the risk of giving a false indication of the health of the design due to incomplete modeling of the analog portions.

6.7.8 Clock domain and power domain crossings

Compared to digital logic in the rest of the machine, there are many more clock domain and power domain crossings in the mixed signal domain. With several efforts in the recent times to improve the verification of multi cycle overrides and model variability associated with handling clock domain crossings using custom flops instead of proper synchronizer, there are many new assertions coded in RTL. Since these assertions are introduced in the RTL in the later phases of the project, issues with tests/checkers not being compatible with such design checks are not uncovered until late in the project. As a result, MSV team may end up getting distracted in debugging and responding to these failures taking time away from other equally important tapein gating activity.

Currently available multiple power plane simulation techniques do not handle any modules that operate on more than 1 power supply domain. This limitation introduces a big hole for power domain crossings coverage in most analog circuits where this is a common case scenario. In order to minimize the impact of this hole, special care must be taken in RTL modeling of analog circuits to make the model sensitive to different power supplies to flush out whatever issues are possible to simulate using RTL or mixed signal simulation methods. This is a perfect example of case where MSV scope is different from other validation domains.

6.7.9 Difficult/impossible to find some bugs in Pre-Si

There are specific classes of bugs in the mixed signal domain that are difficult to find through regular validation activities. Once found on silicon, some of them may look too simple to have escaped to silicon. But it is often not as obvious in Pre-Si simulations or in some cases, just impossible to find them in Pre-Silicon. Below are some examples:

- Glitches on signals may not always propagate to downstream logic in a way that can lead to a simulation failure, but can cause real and fatal problems in real silicon. The impact is especially bad when this happens on signals involved in clock domain crossings. The only way such issues may be found is accidentally when a validator/designer notices glitches in waveforms and follows up on them.
- Some glitches may not even appear in simulation due to the unit time delay through all digital logic gates. Usually, we don't care about such cases. But, if the glitchy signal is accidentally being used directly as an input to an analog circuit instead of going through a sampler/metaflop, the circuit can produce different outputs which may break the machine behavior. Neither RTL nor mixed signal simulation methods can catch such problems. Other tools like LVR may also not find this problem since analog circuits are black boxed there.
- For some simulations, it is not easy to define a robust pass/fail check unless you are an expert in the feature or/and its usage on silicon. Many mixed signal DFx features fall in this category.
- Some silicon failures occur due to miscorrelations between the process file with which all simulations were run and the actual transistor behaviors on silicon. These can manifest themselves as digital logic that controls analog circuit operation unable to compensate for the process variations. Mixed signal simulations are typically run only at 1 process corner. Even if simulations were run at different corners, it is impossible to find such issues Pre-Silicon.
- Noise on the external pins can lead to incorrect and potentially fatal operation of the machine if there is no proper glitch filtering on the pins. If design forgets to add a filter, it will never show up as a failure in simulations unless you think of the test case upfront and put in explicit

effort to either visually inspect that there are filters or put in effort to inject some sort of noise in simulations.

In general, analyzing Pre-Silicon validation escapes from all projects from which we can gather such information is a very good way to understand the vulnerabilities in validation and design processes. Using this knowledge, validation can try to pay special attention to these and try to minimize the gap through all creative means at their disposal.

6.7.10 Skillset challenges

One of the biggest challenges for the MSV team is to find engineers who have the right skillset, mindset and interest in participating in these validation activities. Not only does each individual need to possess a specific combination of technical skillset, it is important for the team to be comprised of engineers who bring in unique value to the team helping the team benefit from the diverse skillset. This enables the team to cover a wide range of mixed signal features that need different technical backgrounds and work together to address new problems.

The following combination of skills is highly desirable in any MSV engineer.

- Digital/logic design
- Analog circuit basics
- Basic computer architecture
- Experience with spice simulation and logic simulation
- Proficient in either Verilog or VHDL
- Exposure to Verilog/VHDL-AMS
- Evil validator mindset
- Deal with ambiguity
- Creative thinking
- Strong problem solving

In addition, organizations can benefit from having at least one or two engineers in the team who have expertise in one or more of the following:, application of Matlab, expertise in RF/high frequency analog/mixed signal circuits, good foundation in math and signal processing, understanding of signal integrity issues, knowledge of power delivery techniques.

Most engineers choose either a digital or analog design path early in their career and stick to that. For the few who actually take interest in both the domains, only a few of them possess the above combination of skill set and more often, they prefer a design engineer career over validation. With analog design being equally in high demand as MSV, it is difficult to attract good talent to this domain.

Once onboard, unlike analog design engineers who spend most of their careers becoming/operating as experts in 1 area, MSV team members have to develop the ability to cover all the diverse areas that range from PLLs to IOs to power delivery methods and whatever else comes their way on the products. On one hand, it is important to develop in-depth expertise in each area to be better prepared to solve future challenges/complexities in that area. On the other hand, they should also have the ability to move around in the team picking up ownership of different areas if necessary to build bench strength. Validators should learn how to develop enough depth of expertise to do the job while focusing on increasing their knowledge and understanding over time.

6.7.11 Prioritization and Risk Taking

Most engineering teams have to make difficult choices at some point or the other during the product development phase. Mixed signal validation is no different in this regard. In the recent years, on most projects across organizations, MSV has become one of the critical paths to tapein. In a resource constrained environment, the only way the team can stay on schedule while still minimizing risk to product quality is by making tradeoffs and funding the highest ROI activities. This requires careful prioritization of all the validation work through a well-defined prioritization process. However, what makes the process more challenging in this domain is the need to coalesce/interpret the information provided by experts from several different domains (RTL, analog circuit designers, feature architects, validators, post-si HVM/tester experts, post-si logic debug/validation) who may not all speak the same language and whittle it down to what it means to validation. It becomes more daunting because most mixed signal features must work on first silicon making and there is little to no safety net for many features beyond MSV and analog design team – this can easily make most engineers risk averse. Finding the right balance between what needs to be validated before tapein vs what can be ZBBed will continue to be a challenge until the MSV teams develop enough experience in making such decisions and build credibility with their customers by delivering the right quality of design on silicon.

7 Summary

With the increase in analog content in Intel products, MSV must be an integral part of all Pre-Silicon validation activities in ensuring the quality of silicon design before tapein. In many ways, MSV methods and philosophy are very similar to those of other validation disciplines and especially similar to uAV. There are also noticeable differences in the actual execution and the challenges faced by the MSV team compared to other validation domains and analog design. It is important to recognize that MSV methods are still immature and evolving. It is more of an art than science compared to other validation disciplines. Standardization is very important and should be enforced in this domain also wherever appropriate. But, both managers and validation engineers should also recognize that there may not be one solution that fits all needs. It is important to internalize the basic principles and learn to apply them effectively to the problem at hand while following the recommended review processes to make sure there are no gaping holes in the solutions applied. And, with every challenge that exists in this domain, it is an opportunity for innovation that is waiting to be tapped with significant positive impact to Intel.

8 Future Work

With MSV established as a discipline in most teams only during the past 5-6 years, there is much opportunity and need to improve execution of MSV and accelerate progress to keep this activity out of the project tapein and PRQ critical paths. Also, there is significant emphasis on reuse of analog IP across Intel in order to minimize reinventing the wheel and benefit from silicon process learnings. In future revisions of this document, expect to see more details on specific topics like the art of analog RTL modeling, creating MSV IP for reuse and others.

9 References

- DTTC 2008 Publication: Challenges and Learnings from Nehalem Mixed Signal Validation
<http://magic.intel.com/DisplayFile.aspx?FileUploadId=56157&Version=5&now=1639173944138>
- Mynx DTS web page: <http://dts.intel.com/product/Mynx/Pages/Default.aspx>
- Raven: A tool for automatic generation of analog behavioral models
<http://magic.intel.com/DisplayFile.aspx?FileUploadId=46804&Version=4&now=1639174218843>

The Art of Pre-Si Val: Chapter 28

Becoming the Microarchitecture Expert

By: Bob Fisch [and Sailaja Madduri](#)

1 Abstract

An IP Validator needs to learn the microarchitecture and become the microarchitecture expert for their IP not only to meet their deliverables but also because they frequently become the main source company-wide for answering questions about the IP. Learning the microarchitecture involves developing expertise in Functional Awareness, Design Knowledge, Implementation Familiarity, Interface Knowledge, Interaction Knowledge, and Historical Familiarity. In developing expertise in these areas, an IP Validator first becomes Oriented, then in succession becomes Capable, becomes Proficient, and finally becomes the Expert. In the course of advancing through these four stages, an IP Validator uses receptive, participatory, and creative activities to gain microarchitecture expertise.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	05/25/2004	Initial draft	Bob Fisch	Matt Kupperman / DPG-N uAV MWG
1.1	06/18/2004	Initial set of revisions from Level 1 review	Bob Fisch	Matt Kupperman / DPG-N uAV MWG
1.2	06/22/2004	Final set of revisions from Level 1 review	Bob Fisch	
2.0	06/22/2004	Initial released version	Bob Fisch	Paul Schwabe
2.2	09/15/2004	L3 Review	Bob Fisch	Steve Cegla
2.3	9/23/2004	Revision based on L3 review comments	Bob Fisch	Steve Cegla
2.4	03/09/2016	Revision to include IP terminology, remove EITVOX, and bring up to date	Sailaja Madduri	Michael Bair

3 Contents

1 Abstract.....	611
2 Revision History.....	612
3 Contents.....	613
4 Purpose.....	615
4.1 Why Learn the Microarchitecture?	615
4.2 Why Become the Microarchitecture Expert?.....	615
4.3 What are the Obstacles?	616
4.4 Does Every Validator Need to Become a Microarchitecture Expert?	617
5 Background Concepts	617
5.1 Microarchitecture Knowledge and Validation	617
5.2 Levels of Expertise	618
5.3 Aspects of Microarchitecture Knowledge	619
5.4 What a Microarchitecture Expert Knows.....	619
5.4.1 Functionality Awareness	622
5.4.2 Design Knowledge.....	622
5.4.3 Implementation Familiarity	622
5.4.4 Interface Knowledge	622
5.4.5 Interaction Knowledge	623
5.4.6 Historical Familiarity.....	623
5.5 Activities that Increase Microarchitecture Expertise.....	623
5.5.1 Receptive Activities.....	623
5.5.2 Participatory Activities.....	624
5.5.3 Creative Activities	624
5.6 Behavioral Expectations of an Expert	625
6 Learning the Microarchitecture.....	626
6.1 Becoming Oriented	626
6.2 Becoming Capable	627
6.3 Becoming Proficient.....	627
6.4 Becoming the Expert	629
7 Summary.....	630
8 Future Work.....	630

9 References.....	630
--------------------------	------------

4 Purpose

4.1 Why Learn the Microarchitecture?

IP Validation fundamentally involves exploring the microarchitecture (uArch) in order to identify areas requiring thorough scrutiny. This is true both at an early project stage, when testplans are being formulated, and at a late project stage, when coverage goals have been achieved yet certain areas seem to merit further attention for reasons possibly grounded in nothing more than an intuitive hunch.

IP Validation takes a white-box approach to validation, where the testplans are strongly driven by the implementation of the design. Indeed, exploring the limits of the design implementation is just as important as exploring the limits of the uArch itself. Thus, a major part of learning the uArch is learning its implementation. This is critical for identifying risky scenarios that could expose the design to failure and that should appear explicitly in a testplan.

An IP Validator becomes more effective with increased uArch knowledge in her area of responsibility as well as in other areas. Broader uArch knowledge translates into the ability to independently and efficiently root cause a wider variety of test failure modes. Such breadth also enhances the ability to realize connections among different units or clusters, which is valuable when examining a component of the design or evaluating a bug fix. Even the mere ability to ask the right question of the right person may depend on basic knowledge of uArch across boundaries.

Microarchitecture expertise in one area comes with transferable skills that can be ported over for building uArch expertise in other areas. So being an expert in one area puts a Validator on a faster track to becoming an expert in a new area.

4.2 Why Become the Microarchitecture Expert?

In the beginnings of a project, the architects are the primary uArch experts, since it is their vision that defines the project. This vision is passed on to designers who set about implementing this vision. As design constraints take their toll, the resulting uArch becomes more complex and designers become the primary source of uArch expertise. They understand the tradeoffs available and the rationales for the choices made. However, in our vertical design organization, the role of design transitions from logic design to circuit design. The focus over the course of a project shifts from writing RTL to creating schematics to analyzing and fixing speedpaths. The power of this approach is that fixing speedpaths merely requires finding a different approach to creating equivalent logic without necessarily needing to examine the uArch this logic is implementing. However, because of this focus away from the uArch, the Designer's role as uArch expert is diluted.

In parallel with the design process, IP Validation is active validating the uArch. While Design's focus strays from the logic design, Validation is learning more and more about it. When questions about the uArch arise, whether from within the Pre-Silicon team, from others associated with the project, or from other projects, these questions are sometimes directed at the Validator responsible for that area. As time proceeds, the Validator fields a larger share of such inquiries. Beyond tapeout, designers typically move to new responsibilities or to new projects before Validators make similar moves. A Validator may become the sole source on a project capable of answering detailed questions about her area.

Because of these shifts in roles as a project matures, it is vital for an IP Validator to become the uArch expert in her area. During periods when other sources for the same expertise are less readily available, the questions asked are often more difficult and more critical. It is in these latter stages of the project when arcane boundary cases cause concern, when hypothetical behaviors based on indirectly related uArch changes are explored, and when bugs are found which may also exist on previous steppings in production. At times like this, it is not enough to be merely knowledgeable about the uArch. The Validator will be expected to become the uArch expert in her area.

Let us clarify that the expertise described throughout this document refers to uArch expertise, as opposed to validation expertise. This latter expertise involves knowledge of methodologies, tools, computing environments, and other mechanics enabling a Validator to do her job. There will be high correlation between those who are validation experts and those who are uArch experts. This document focuses on how to develop uArch expertise and only peripherally touches upon validation expertise. The term *expert*, without qualification, is used throughout to mean a uArch expert.

4.3 What are the Obstacles?

A Validator's ability to learn the uArch and become the uArch expert in her area is greatly affected by many circumstances beyond her immediate control. These circumstances can affect the speed at which expertise can be developed. This section touches upon several such circumstances in order to convey that the process of learning the uArch and becoming the expert in a timely fashion requires more than a mere will to do so.

Often, the need to develop expertise exists in an environment where few if any experts in the area already exist. Such lack of expertise may be limited to IP Validation or the general Validation community, or it may be project wide—not atypical for a proliferation. In fact, it may be the lack of expertise in an area that necessitated the development of an expert in the first place. Without other experts immediately available to ask difficult questions, someone new to an area will have added difficulty in learning that uArch.

A lack of quality documentation will slow the process of learning the uArch. In our experience, the quality of available documentation for ramping Validators on a new area is often poor.

The complexity of the uArch area will also affect the ability to develop expertise. This will have the most acute effect in tandem with other hindrances to learning the uArch, since a complex area will more frequently spur difficult questions whose answers may not be ascertained quickly.

The experience level of the Validator plays an enormous role when learning a new area. Someone with prior uArch expertise, even in an unrelated area, will be better prepared to take on the challenges of developing new expertise than someone with no uArch expertise. Even mere familiarity with the validation environment is one less impediment to making progress.

The activity level of the project will have major impact on learning new a uArch. A project in early execution will provide ample opportunity for a Validator to develop strategies for gaining expertise. Bugs involving only moderately complex interactions are common during these times allowing a gradual ramp for a Validator new to the area. On the other hand, a post-A0 tapeout project in the midst of a series of steppings and the accompanying revalidations will provide limited opportunities for substantive debug, and those opportunities will generally involve

extremely intricate interactions. This is not an easy environment in which to become the expert. The importance of the project activity level in a new Validator's ability to learn the uArch should not be underestimated.

4.4 Does Every Validator Need to Become a Microarchitecture Expert?

A manager wants all her Validators to be uArch experts. There are many reasons for this, but perhaps the reason with broadest scope is that a team is most valuable when its collective knowledge is greatest. Furthermore, each Validator wants to learn more about the uArch, so each will act to gain expertise.

However, it is not necessary for everyone to be a uArch expert in order for an IP Validation team to succeed. In fact, a team can be successful without any uArch experts. Proliferation projects are often examples of this. For this reason, a manager needs to keep in mind that their goal is project success. Development of uArch experts is a path towards that goal, but it is not the only path. Expectations for individuals need to be realistic, and partial success in learning the uArch may be a quality goal for an IP Validator. Requiring that everyone becomes a uArch expert may be counterproductive.

Having multiple uArch experts in a single uArch area is good. This creates flexibility when a critical issue arises, such as a post-silicon sighting in an area requiring investigation by an expert. Having multiple experts provides a choice of Validators to assign to such a task, lessening the possibility that someone must abandon a critical task on a different project in order to attend to a more critical task from a company perspective.

Unfortunately, it may be difficult to develop more than one expert in an area. The personal experiences of one Validator are rarely those of another. If one expert already exists, a project would literally have to prevent them from performing validation activities in their area of expertise in order to give someone with less expertise the chance to become an expert. This could probably only happen through reassignment of responsibilities where the expert in one area becomes responsible for a new area where their expertise is significantly lower. One natural opportunity for such reassignment is when moving Validators to a new project.

5 Background Concepts

5.1 Microarchitecture Knowledge and Validation

The organization of our IP Validation teams has been largely based on IP ownership. An IP Validator will own one or more IPs, or share ownership of a larger IP, and will focus her efforts on validating that IP. The Integration Validation team is responsible for validating the interactions of that IP with other IPs. Validating the implementation demonstrates that the RTL does what the design set out to implement. Validating the interactions shows that the design does what the uArch needs it to do for global correctness. The process of validating an IP's implementation and interactions results in the development of uArch knowledge related to the IP. When we speak of learning the uArch, we usually mean developing such expertise. This chapter specifically focuses

on uArch expertise in the context of IP validation. Expertise in the context of Integration Validation will be covered in the [Becoming the Architecture Expert](#) chapter

Of course, there is uArch that is not based in a particular IP. Chassis protocols are shared across multiple IPs. An example of this in BXT is the IOSF protocol. Other protocols may be implemented with pieces scattered across the chip having no direct cooperation among most of these pieces. In such a case, it would be inappropriate to claim that the protocol is shared. An example of this in BXT is the S0ix flow. Such uArch areas often have owners whose focus is not IP based. These aspects of uArch expertise are covered in [Becoming the Architecture Expert](#) chapter

Throughout the remainder of this document, we will focus on the scenario where learning the uArch means understanding an IP's implementation and interactions. This will provide the most concrete approach to identifying what needs to be done to learn the uArch and become the expert, which will in turn help discuss how to gain such expertise. For someone interested in gaining uArch expertise in an area that is not IP based, the ideas presented here will translate readily into that context. The admitted oversimplification of equating learning uArch with developing IP expertise is more than offset by the clarity this simplification allows and the fact that this simplification describes the common case.

5.2 Levels of Expertise

In any given uArch area, a Validator lies somewhere in the spectrum spanning from no knowledge whatsoever to complete knowledge of all detail. For the purposes of the present discussion, it is useful to define some broad categories in order to distinguish individuals far apart on this spectrum as well as to track the progress of any one individual along this spectrum. In this section, we introduce terms denoting such categories. In later sections, more detailed descriptions of what can be expected from someone at each level of expertise will be presented. The terms presented here are not necessarily broadly used, but the IP Validation team generally understands the levels of expertise conveyed by them.

Someone new to an IP will initially strive to become *oriented*. At this level, one has a general idea of what goes on but has yet to learn enough to make effective progress for validation.

After orientation, a Validator will progress towards becoming *capable*. This reflects the ability to make effective progress on validation tasks, although a great deal of help from others is still necessary. Being capable enables efficient learning, because new ideas have a context in which to take root and make sense.

Through the accumulated knowledge acquired during the capable phase, a Validator eventually becomes *proficient*. This reflects a level of independence in working through issues in order to make progress. Although help from others is still necessary, the levels of such discussions are significantly more in depth than at the capable level and often lead to investigations where quality bugs are found. Furthermore, the proficient person will readily be able to answer most questions about her IP.

Beyond proficiency, continued examination of an IP will lead to the creation of an *expert*. At this level, the Validator can quickly provide answers to very intricate questions about her IP. This includes assessments of what would happen if a change to the uArch were implemented. If

someone on the project has a question about an IP, the expert is the person she will go to and expect a good answer, regardless of how general or specific the question.

The four terms *Oriented*, *Capable*, *Proficient*, and *Expert* are used throughout this document to indicate levels of expertise consistent with the overview given here.

5.3 Aspects of Microarchitecture Knowledge

Knowledge of a uArch area is multifaceted. Not only does the general mechanism of the uArch need to be understood, but an awareness of its implementation is also important. Peripheral aspects involving related uArch areas should also be learned. Familiarity with how the uArch evolved to its current state can be valuable.

In this section, we define several terms to capture these different aspects of uArch knowledge. In later sections, these will be used to describe what is typically known by Validators at various levels of expertise.

The six terms introduced here to encompass the various aspects of uArch knowledge are *Functional Awareness*, *Design Knowledge*, *Implementation Familiarity*, *Interface Knowledge*, *Interaction Knowledge*, and *Historical Familiarity*.

Functional Awareness is the identification of the functional components of a unit. *Design Knowledge* is knowledge of the algorithms used to create the functional components. *Implementation Familiarity* is awareness of how these algorithms are translated into logic design. *Interface Knowledge* is knowledge of the input and output signals for an IP and the information each represents. *Interaction Knowledge* is the understanding of how a unit's uArch is affected by other units. *Historical Familiarity* is awareness of how the IP evolved over time and what tradeoffs were made in the design and implementation.

Although the focus here is on uArch, developing knowledge of the user-visible architecture that the uArch is implementing can be useful. Use case knowledge is helpful in effectively collaborating with SoC/Firmware/Software teams, but IP validation should not be limited to just the use cases that a SoC is targeting.

5.4 What a Microarchitecture Expert Knows

Table 1 defines the levels of expertise introduced in section 5.2 in terms of expertise achieved on each of the aspects of uArch knowledge introduced in section 5.3. In this section, we will elaborate on how a Validator progresses in each aspect of uArch knowledge as her expertise grows in order to clarify the content of this table.

The intent of defining uArch expertise is to create a framework used in section 6 to describe approaches to learning the uArch and developing expertise. Methods effective for a Validator at one level of expertise will often be of little use to a Validator at another, and this framework gives a context for making such distinctions. In creating Table 1, there is no intent to create a rating system for judging the knowledge any particular Validator. Furthermore, a Validator will display varying levels of expertise in different areas.

In the table, each aspect of uArch knowledge is measured on a scale from 1 to 5. Descriptions of intermediate levels of expertise are made as is useful. The levels of expertise correspond to this

numerical system as follows: Oriented is Level 1, Capable is Level 2, Proficient is Level 4, and Expert is Level 5. No label has been given to Level 3. This reflects the significant learning required to progress from Capable to Proficient, coupled with the lack of utility in creating a label for any intermediate level of expertise.

A Validator is unlikely to progress in a uniform manner through all aspects of uArch knowledge as she gains expertise. Indeed, knowledge on one axis may proceed rapidly to higher levels while knowledge on another lags at lower levels. Furthermore, a Validator can be considered a true expert without necessarily maxing out on all the scales. Section 6 will contain additional discussion of what a Validator at a particular expertise level knows relative to the aspects of uArch knowledge presented in Table 1. Table 1 should be considered as a set of rough guidelines describing how an individual tends to gain expertise.

Table 1: What is uArch Expertise?

	Level 1	Level 2	Level 3	Level 4	Level 5
Expertise Level	Oriented	Capable	Proficient	Expert	
Functionality Awareness	Awareness of high-level functionality	Awareness of primary functional components	Awareness of secondary functional components	Awareness of all functional components	
Design Knowledge	General understanding of algorithms used and important structures	Solid understanding of common case algorithms	Understanding of “obvious” boundary cases	Understanding of feasibly encountered boundary cases	Understanding of the most arcane boundary cases
Implementation Familiarity	Familiarity with RTL organization and mapping to important structures	General familiarity with RTL, including key RTL signals		Detailed familiarity with RTL implementing primary functionality	Detailed familiarity with RTL implementing all functionality
Interface Knowledge	Familiarity with key datapath and control interfaces and the primary functional areas with which they are associated	Understanding of both primary and secondary uses of the key inputs to the IP as well as the mechanics for producing the key outputs from the IP	Understanding of how key interfaces are used and produced, including secondary uses and unusual interactions	Familiarity with other interfaces and the functional areas with which they are associated	Understanding of all interfaces and how they are used/produced
Interaction Knowledge	Awareness of interacting blocks and the functionality they utilize or contribute to		General understanding of mechanics of other blocks which are relevant to interaction		Detailed understanding of how other blocks interact: algorithms, RTL, boundary cases
Historical Familiarity	Awareness of current implementation	Familiarity with notable recent bugs/issues/ECOs		Knowledge of bugs, issues, ECOs encountered over the lifetime of current project	Knowledge of bugs, issues, ECOs encountered over multiple generations leading up to current project
				General awareness of tradeoffs considered when design/uArch was developed	Solid understanding of tradeoffs considered when design/uArch was developed

Figure 48

5.4.1 Functionality Awareness

A Validator needs to be aware of what functionality exists in her IP. Initially, she will gain a high-level view of this functionality. Next, the primary functional components of the IP will become apparent. These are the key functional components for which the IP exists in the first place. Later, the Validator will learn of the secondary functional components in the unit. These are auxiliary calculations that support the primary functionality but which address side issues instead of the main focus. Finally, there may be other functionality that has nothing to do with the primary functionality of the IP but is situated in the IP out of convenience. As a Validator gains expertise, she will progressively become aware of these functional components.

To clarify this description an example from BXT will be given using the System Agent. At a high level, the System Agent provides a path for communication between the cores, IO and memory. Primary functionality includes providing coherence, global memory ordering, arbitration and scheduling of requests. Secondary functionality includes security enforcement and priority support for requests. Other functionality that is neither primary nor secondary includes the C- Unit (Configuration management Unit) that provides a bridge between IOSF Primary and Sideband agents.

5.4.2 Design Knowledge

Once a Validator is aware of functionality in her IP, she can learn the approach used to create that functionality. Initially, important data structures such as arrays or buffers are identified, and the general algorithms used to manipulate these are learned. After this, the details of the calculations in the common cases are understood. Beyond this, the Validator identifies how boundary cases are handled, starting with the simplest such instances and progressing to very elaborate scenarios requiring many disparate ingredients.

5.4.3 Implementation Familiarity

Intimate familiarity with the RTL that implements the design is valuable. A Validator new to an IP would start by identifying the organization of RTL for that IP, with particular attention to identifying where important data structures are defined. Early in her progress, the Validator would gain general familiarity with the RTL and would become aware of key signals. As she gains more expertise, the Validator would develop more detailed familiarity with the RTL, first in the portions implementing primary functionality and then expanding to the entire body of code.

5.4.4 Interface Knowledge

A Validator should develop familiarity with all the inputs to and outputs from her IP. Early on, she should gain awareness of the key datapath and control interfaces and the primary functional areas with which they are associated. With time, secondary uses of such inputs will be learned and the roles of other interfaces will be understood. Eventually, the Validator will know all the places where each input to her IP is used as well as how each output from her IP is produced.

5.4.5 Interaction Knowledge

To fully understand a unit, a Validator needs to gain knowledge of other units that interact with her unit. Identification of these interacting units and the functionality to which they contribute is a first step. Beyond this, it is useful to attain a general understanding of how these other units contribute to the calculation through consumption and production of the interface signals. In the end, a Validator needs to have a thorough understanding of the relevant portions of the interacting units, including boundary cases. Note that this falls well short of becoming an expert in these other units, but this effort will nevertheless raise a Validator's level of uArch expertise in these units. More information is available on this in the [Integration Validation](#) chapter.

5.4.6 Historical Familiarity

A Validator new to an IP is unlikely to be aware of any past decisions that influenced the current implementation. As she gains experience, she will become aware of notable bugs, issues, and ECOs filed against her IP—both new and those previously filed. As time proceeds, this knowledge will expand to filings against this IP from past generations of products, either as a result of her database searches or from direct experience after working on multiple projects. In parallel, as expertise grows, an understanding will develop of the design tradeoffs available and why particular choices were made.

5.5 Activities that Increase Microarchitecture Expertise

Many activities a Validator can undertake will increase her uArch expertise. Some of these are more receptive in nature and are better suited to someone with low expertise. Others are more creative in nature and aid a somewhat experienced Validator to become even more knowledgeable. Still others are participatory, falling somewhere between receptive and creative. In this section we touch upon these activities. In section 6 references will be made to these activities and how Validators of different levels of expertise use them.

5.5.1 Receptive Activities

Receptive activities generally refer to reading or watching available sources of information. Of course, a prerequisite to using such sources is to have them available; this tautology merits expression because of the dismal state of documentation for most projects. We list here many of these sources of information.

A fundamental source for learning uArch is the RTL. It can be used for tracing signals or for reading comments. A Validator should be skeptical of comments since they can be out-of-date or just plain wrong, but even a wrong comment can suggest to a Validator where to look for more information about the uArch.

Documentation generated by design or architecture is another source for learning uArch. Possibly the best sources would be from a complete and up-to-date High level Architecture Spec (HAS) and MicroArchitecture Spec (MAS). However, other less formal documentation such as e-mail communication or written notes can also be quite helpful.

Reading previous exercise plans, testplans, exit review checklists, or other documentation created by Validation can also help with learning the uArch. These are more likely to be available only late in a project or on a proliferation project.

The Programmer's Reference Manual (PRM) is a source for learning detailed architecture that may be helpful in learning uArch. At a minimum, this will specify what the uArch has to support.

Attending presentations given by experts, watching recordings of such presentations, or merely reading foils from such presentations can be useful. The presenter may be a Validator, Designer, or Architect.

5.5.2 Participatory Activities

Participatory activities refer to activities that have some dependence on others to create content but in which a Validator new to a unit can still play an active role.

The mere act of designating a person as a Validation IP owner is a critical step to developing uArch knowledge. This act implies that questions about this IP should be asked of that Validator and that she has a responsibility to identify the answers. Expectations should be realistic, but it is still important that the Validator learn the questions of others and then seek to find the answers, either through her own explorations or by asking other experts. Such questions will indicate what some of the interesting areas are, and the investigation or discussion used to find the answers will be fruitful.

Asking uArch questions of Designers, Architects, and other Validators will create good discussion for developing expertise. Team members need to understand the value of helping less experienced Validators gain uArch knowledge and the importance of their roles in this effort.

A mentor or buddy system is another approach to getting an inexperienced Validator engaged in discussions about her unit.

Participating in focus groups where uArch is discussed is a good way to develop IP expertise. The level of discussion may frequently exceed the knowledge of a Validator new to her IP, but this provides a level of exposure difficult to get otherwise, and an inexperienced Validator will learn good things out of such discussions.

Taking classes such as those offered by Intel University on iA32 assembly coding and protected mode architecture can reinforce architectural details that may be helpful in learning uArch.

5.5.3 Creative Activities

Creative activities refer to activities where the Validator creates the primary content. Some substantive expertise is needed before a Validator can undertake most of these activities.

Running and debugging tests is perhaps the most efficient of all approaches for learning uArch. This activity encompasses many tasks: watching tests run with DVE and visual traces to learn typical behavior; pouring through RTL to identify how key signals are generated; talking with designers, architects, and Validators to question odd behavior and to identify and evaluate bug fixes; writing bug reports to document the eventual discoveries and propose solutions. Each of these tasks is useful in developing expertise.

For many of the deliverables required of a Validator, the struggle to generate them will result in increased knowledge about her unit. The activities undertaken as part of this struggle include writing exercise plans and testplans, writing self-checking directed tests, coding coverage conditions, coding the test environment, coding test generators, and coding checkers and injectors. Furthermore, driving coverage to acceptable levels requires detailed coverage analysis including identification of forbidden conditions. Such coverage efforts may culminate in the creation of exit review checklists. These activities, the output of which is denoted in later sections as *validation collateral*, pose numerous challenges. Expertise is a byproduct of successfully meeting these challenges.

Higher levels of expertise can be achieved by tracking interesting bugs, issues, and ECOs. Analyzing escapes, both in terms of debugging the failures as well as understanding why past testing did not find them, will also generate avenues for learning new things about the uArch.

Disseminating uArch to others, either through presentations or other documentation, is a valuable learning technique. The act of writing down or organizing one's knowledge is a good way to identify areas of incomplete understanding and to fill in these gaps in knowledge. The feedback from giving a presentation or disseminating documentation will also provide opportunities to increase one's expertise.

5.6 Behavioral Expectations of an Expert

The primary expectation of an Expert is the ability to provide information about her area of expertise. However, an Expert will also be expected to exhibit certain qualities unrelated to uArch knowledge. In this section, we touch upon some of these expectations. Validators moving along the spectrum towards becoming the Expert will demonstrate shades of these qualities at levels corresponding to their level of expertise. Managers should encourage development of these qualities among their team members.

Perhaps the most important expectation of an Expert is to be a role model for others on the team. An Expert is expected to be able to set her own goals, plan her work towards these goals, set her schedules and milestones, and get management approval for her plans. An important aspect of this is that an Expert will have developed a good sense for the amount of time required to accomplish tasks ranging from the simple to the complex.

A related expectation is that once an Expert's goals and schedule are in place, she will find a way to either meet these goals or explain why falling short can be tolerated. Experts understand that goals are set with incomplete information and that the effort towards hitting these goals will reveal new information that might better convey what is really necessary and what does not have enough relative merit to pursue. An Expert can look at the big picture and figure out how to do only that which is necessary and leave that which is unnecessary undone.

An Expert is expected to continually track her progress, realize when her overall schedule is in peril, replan her efforts, and communicate such plan changes to her management. This ability to plan, track, and replan is highly valued, since it provides management the best opportunity to find help or otherwise address a situation before it reaches the critical level.

There are often critical needs that the team cannot deal with because of its frantic activity on other fronts. An Expert is expected to identify such tasks and take ownership without being asked to do

so. An Expert is able to assess the importance of such a task and weigh it against her current activities and schedule constraints. She is able to decide whether she has the bandwidth to take on additional responsibilities and whether her other activities can tolerate a schedule slip due to the relative importance of the new task. If the Expert sees the importance of the task but cannot take it on herself, she will find a different owner and thereby drive the task to completion.

6 Learning the Microarchitecture

In this section, we describe the separate processes of becoming Oriented, becoming Capable, becoming Proficient, and becoming the Expert. These levels of expertise were defined in section 5.2. Each of these processes is addressed in a separate subsection below.

Most of the tasks referenced in the following processes were elaborated upon in section 5.5. Reference section 5.5 for greater detail about such tasks.

6.1 Becoming Oriented

Anyone new to an IP area starts at this point regardless of her background. The primary input to becoming oriented is to have a uArch to learn. Other inputs that are good to have are materials that specify or describe the uArch. These include items mentioned in section 5.5.1. RTL is needed in order to make significant progress in learning the implementation, but is otherwise not required. The efforts made to become Oriented will generally center around the receptive activities mentioned in section 5.5.1. Some participatory activities from section 5.5.2 will also be undertaken.

Usually, another Validator will first give an informal orientation to the uArch that is being learned. The background of the person being Oriented dictates the level of this initial presentation. An experienced person might only need a short one-on-one to identify where her focus should be, whereas a new hire might need a series of presentations giving overviews of the entire design and what the IP is responsible for before focusing on the details of the IP being learned.

Beyond this, the Validator scans the written documentation available, watches any available live or recorded presentations, and looks through the RTL. On this first pass, the Validator will likely glean only the surface of the body of knowledge she is trying to learn. She asks questions of other Validators, Designers, and Architects in order to solidify her general understanding of what the IP is supposed to do and how it fits into the larger scheme.

If a mentor or buddy system is to be used, then such mentoring or buddying is initiated during this orientation. This provides another source for asking questions.

Responsibility for the IP should clearly be given to the Validator becoming Oriented. She should also attend any technical forums that her team members normally attend to contribute their uArch expertise. Making sense out of the uArch that is discussed during these activities will be difficult, but as long as expectations are set appropriately, such efforts will prove valuable.

The Validator may use the test environment to run tests and watch how her IP operates. This can involve tracing key signals, interfaces, and internal structures. Limited debug may be undertaken, but she will have severe difficulty making progress on any nontrivial debug situation at this time.

For each of the aspects of uArch knowledge listed in Table 1, when a Validator has achieved a level of expertise listed in the Level 1 column, she can consider herself Oriented in that aspect.

6.2 Becoming Capable

A Validator should already be Oriented before she starts working towards becoming Capable. One difference from becoming Oriented is that availability to RTL is more important in striving to become Capable. Although one can gain more expertise even if RTL is not available, such effort will be severely hampered in this case.

The efforts made to become Capable will generally center around the receptive activities mentioned in section 5.5.1 and the participatory activities mentioned in section 5.5.2. Some creative activities from section 5.5.3 will also be attempted.

Because the Validator is already Oriented, she has a context in which to start accumulating knowledge about her IP. She will continue to examine documentation, attend presentations, watch recordings, participate in technical forums, and ask questions of others. Her level of involvement in each of these activities will be increased. Documentation will be read with significant comprehension instead of merely skimmed. At presentations and forums, she will be able to ask good focused questions whose answers will contribute to her knowledge.

She will start to recognize when items under consideration pertain specifically to her IP. Mechanics of how her IP functions will start to make sense. Many details will still be elusive, and some functional areas will remain a mystery, but she will be developing a good understanding of the primary functional areas of her unit. When questions are asked of her about her IP, she will begin to know some of the answers and know where to look or who to ask to find many of the rest.

The IP Validator will start working more heavily with the RTL, both by examining signals while running tests as well as just looking through the code to learn about specific logic implementation. Her debug efforts will become more successful but will still be challenging. Depending on the stage of the project, she may have opportunities to file some bugs, requiring her to write descriptions of functional behavior.

Project activity may require the Validator becoming Capable to work on creating validation collateral. Such efforts will provide many learning experiences but at this stage she will be inefficient at making progress.

For each of the aspects of uArch knowledge listed in Table 1, when a Validator has achieved a level of expertise listed in the Level 2 column she can consider herself Capable in that aspect.

6.3 Becoming Proficient

An IP Validator should already be Capable before she starts working towards becoming Proficient. Documentation used while becoming Oriented and Capable can still be useful but is less critical. The most useful of such documentation is that which describes the detail of implementation or other complexity about the uArch. Such documents will most often be used as references rather than as documents to read.

The efforts made to become Proficient will generally center around the participatory activities mentioned in section 5.5.2 and the creative activities of section 5.5.3. The receptive activities of section 5.5.1 do not play as large a role, since documentation becomes a reference to support other activities.

Because the Validator is already Capable, she is now able to undertake more complex tasks and figure out how to complete them with reasonable efficiency. She will become adept at developing any validation collateral for which she is responsible. She will be able to run and debug tests independently, getting help from others when the logic cone being examined strays far from her current range of accumulated expertise. The conversations about topics outside her range of expertise are learning experiences themselves for developing more uArch knowledge.

The learning process while becoming Proficient increases in efficiency. As each new piece of information is encountered, it is usually clear how it fits into the larger uArch puzzle. Gaps in the Validator's knowledge are more evident—she knows what she doesn't know—permitting her to seek sources for filling in the holes. The Validator can check newly learned information for consistency against her accumulated knowledge. This is a key skill for confirming correctness of her overall understanding and for alerting her to boundary cases not previously realized. This process of identifying boundary cases through inspection can lead to identifying bugs not yet encountered through testing and checking.

Experience with bugs, issues, and ECOs will grow significantly while becoming Proficient. The Validator will file some of these and will be asked to validate or otherwise provide input to others. Database searches while performing debug will reveal past bugs, issues, and ECOs of note.

At forums that discuss uArch, the Validator will consistently contribute useful information to the discussion. She will be able to learn from discussions outside her current range of knowledge and use these new learnings in future tasks.

As questions come to her regarding her IP, the Validator becomes increasingly able to provide an immediate answer or to investigate the issue independently. These inquiries will at times lead to conversations with Designers, Architects, and other Validators when they involve sufficiently complex and interesting behavior not yet considered by the Validator.

The Validator may be asked to deliver presentations to groups of Validators or others needing to learn the uArch. Often, others may have given such presentations in the past and created accompanying foils, so the Validator may not need to organize the presentation but merely learn the content and present it. This situation provides a gradual ramp for delivering such presentations.

Through all of these tasks, the Validator will greatly broaden her base of uArch knowledge.

In the process of becoming Proficient, the Validator is likely to generate much validation collateral associated with her IP depending on the stage of the project. In addition, the Validator may generate some additional written documentation that will live on to communicate information about her IP. This could be informal such as e-mail, or it could be prepared documents distributed to the team or posted to a central repository. She may organize a presentation with accompanying foils. The presentation may be given several times, possibly by other team members, and a recording may be made of one such presentation for future viewing. One should not set expectations too high regarding the amount of documentation the Validator becoming Proficient might generate, but there is likely to be some produced.

For each of the aspects of uArch knowledge listed in Table 1, when a Validator has achieved a level of expertise listed in the Level 4 column, she can consider herself Proficient in that aspect.

6.4 Becoming the Expert

A Validator should already be Proficient before she starts working towards becoming the Expert. The Validator becoming the Expert relies heavily on what she learns from other people, and any material available serves as a possible reference along the path towards gaining expertise. The RTL has been completely consumed, so it merely serves as another reference for information. Even the uArch has been scrutinized exhaustively and much activity revolves around hypothesizing about what would happen if the uArch were changed.

The efforts made to become the Expert will generally center on the creative activities of section 5.5.3. The Validator will also contribute to the participatory activities of section 5.5.2, but her role is more likely to be that of a leader or a source of information from which others learn.

Because the Validator is already Proficient, she can be relied on to complete complex tasks, such as generation of Validator collateral, with high quality in a reasonable amount of time. She will be able to independently debug failures involving a broad area of logic whose boundaries are well beyond that of her IP. When unfamiliar logic is encountered, she can communicate efficiently with an expert in that area to acquire whatever information is necessary. Effectively, there are no boundaries to the complexity of uArch she can undertake in order to understand how the logic related to her IP behaves.

The discussions that a Validator becoming the Expert participates in are broad in scope. She is likely to contribute to forums about how the uArch should evolve for future projects. She may collaborate with other experts about how to validate a processor more efficiently. She will field company-wide questions in a variety of contexts where resolution requires detailed knowledge about the workings of her unit.

Through this activity, the Validator will gain increased perspective on her IP in relation to the project as a whole. This applies not only to the present state of affairs, but also in a historical context with regard to past projects. This will influence decisions made by project staff about future directions.

The Validator becoming the Expert disseminates a substantial amount of information about her IP. Some may be written, and some may be in the form of presented material. Although the validation collateral contains some of this information, she often creates documentation beyond the contents of the validation collateral.

In the process of becoming Expert, the Validator is likely to generate much collateral associated with her IP depending on the stage of the project. Furthermore, she will create materials documenting her IP. Besides written documentation, she is likely to give presentations about the uArch in her IP. Foils may be prepared that others may use for future reference or for recreating the presentation to a new audience. Presentations may be recorded for future viewing by those learning the uArch.

For each of the aspects of uArch knowledge listed in Table 1, when a Validator has achieved a level of expertise listed in the Level 5 column, she can consider herself an Expert in that aspect. In reality, though, she is never truly finished gaining expertise. The process of learning the uArch and becoming the Expert is an unending process.

7 Summary

In order to describe how a Validator learns uArch and becomes the expert, we broke down these rather nebulous concepts into concrete pieces or stages. We defined learning the uArch as developing expertise in Functional Awareness, Design Knowledge, Implementation Familiarity, Interface Knowledge, Interaction Knowledge, and Historical Familiarity. We partitioned the process of becoming the expert into four stages: becoming Oriented, becoming Capable, becoming Proficient, and becoming the Expert. Table 1 provides a road map that roughly suggests how much expertise is acquired in each of these areas of uArch knowledge as one progresses through the four stages of overall uArch expertise. We also categorized the activities that a Validator uses to develop expertise into three categories: receptive, participatory, and creative. By making this categorization of activities, we are able to convey that two Validators at different levels of expertise will participate in different types of tasks, as opposed to participating in the same tasks at different levels of involvement. The distinction here is important for managers to understand in order to create a good environment and maintain reasonable expectations for those developing expertise.

Obviously, the entire endeavor of learning uArch and becoming the expert are genuinely human tasks of enormous complexity requiring unbounded creativity. The reader should not expect to be able to reduce these tasks to algorithms that can be applied to robotic Validators. Although this document essentially creates such a framework, this is done as a vehicle to effectively communicate many ingredients to learning the uArch and becoming the expert in an effort to share what we have found to be effective. The intent is not to dehumanize the process. Hopefully, the reader will understand this and will use this document in the spirit with which it is intended.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 29

Design for Test Validation

By: [Chun Tan](#) and [Erik Samuelson](#)

1 Abstract

The primary goal of Design for Test (DFT) Validation is to find the functional/logic bugs in DFT features and validate the usage model flows HVM (High Volume Manufacturing) will use.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	1/8/2011	Initial Draft	Chun Tan	Michael Bair / CCDO Val Staff
1.1	1/25/2016	Update/Modernization	Erik Samuelson	Daniel Fisher, Clint Whiteside, Jae Ko, Michael Bair

3 Contents

1 Abstract.....	631
2 Chapter Revision History	631
3 Contents.....	632
4 Purpose.....	634
4.1 Why do we need this chapter?.....	634
4.2 What does this chapter cover?	634
4.3 What does this chapter not cover?	634
5 Background Concepts.....	634
5.1 HVM (High Volume Manufacturing)	634
5.2 Sort Testing	635
5.3 Burn-in	635
5.4 Class Testing	635
5.5 PPV	635
5.6 Tester	635
5.7 Pattern Generation	635
5.8 Types of HVM Content	636
5.8.1 Structural Test	636
5.8.2 Functional Test	636
6 DFT Validation	636
6.1 Introduction to DFT Validation	636
6.2 Why do we need DFT Validation?	637
6.3 The Basic Characteristics of DFT features	638
6.3.1 Typical DFT System.....	638
6.3.2 DFT Modes/Hooks.....	639
6.3.3 DFT Usage Models.....	640
6.4 The Challenges in the DFT Validation world.....	640
6.4.1 IP vs. Integration Validation responsibilities	640
6.4.2 DFT logic is everywhere	640
6.4.3 False functional failures	641
6.4.4 Some DFT Logic is directly implemented in backend flow	641
6.4.5 High Bug Volume.....	641

6.4.6	Validation environment does not resemble tester environment	641
6.4.7	Ownership Management.....	641
6.4.8	Random testing is virtually impossible	642
6.4.9	Unclear usage models	642
6.5	The mindset of a good DFT Validator	642
6.5.1	Know your feature and its usage models well.....	642
6.5.2	Actively engage in usage models definition	643
6.5.3	Acquire good breath of knowledge on the functional feature	643
6.5.4	Prepare to deal with ambiguity and take advantage of others' expertise	643
6.5.5	Develop Flexible and Scalable DFT environments and Tests.....	643
6.5.6	Define clear Validation scope	644
6.5.7	Find bug workarounds for forward progress	644
6.6	The Role of DFT Validation across the project phases.....	644
6.6.1	Tech Readiness.....	644
6.6.2	VAL0.0->VAL0.5	644
6.6.3	VAL0.8->VAL1.0	644
6.6.4	Post-silicon	645
6.7	Interaction with other teams.....	645
6.7.1	High Volume Manufacturing.....	645
6.7.2	IP Integration Teams.....	645
6.7.3	Other Validation teams	645
7	Summary.....	645
8	Future Work.....	646
8.1	Defining VJT Model with HVM Team	646
8.2	Roles and responsibility between Horizontal DFX and Product DFX Teams	646
9	References.....	646

4 Purpose

4.1 Why do we need this chapter?

DFT features enable Intel's aggressive High-Volume Manufacturing testing goal, i.e. delivering high-quality parts to the customer with the lowest testing cost possible. DFT Validation is responsible for validating the functionality of these DFT features. This chapter will share the DFT Validation experience accumulated through several projects. It can serve as a good introduction of DFT Validation for engineers new to DFT Validation or to a project team that is building a DFT Validation effort for the first time. For project teams who have done DFT Validation before, the DFT Validation challenges highlighted in this chapter should resonate with them and the principles documented in the chapter to approach common problems should serve as a good guide.

4.2 What does this chapter cover?

This chapter will introduce DFT technologies, explain the need for a dedicated DFT Validation focus, review the common challenges for DFT validation, explain the needed mindset of the DFT Validator, review the role of DFT Validation across the project stages, and finally discuss interaction of the DFT Validation team with other teams. The content is based on several generations of Client CPU and SoC DFT Validation experience, but the common principles described should apply to DFT Validation in other product teams as well.

4.3 What does this chapter not cover?

Many elements of DFT Validation are similar to other Validation disciplines, such as the need for quality test plans, efficient debug, etc. Therefore, this chapter will only single out validation principles that are unique to DFT Validation. Besides that, this chapter will not describe individual DFT features in detail. For readers who are interested in system debug features (on-die Trace/Trigger, Array Freeze and Dump, PSMI, etc) validation, the Design For Debug (DFD) chapter (see [Design for Debug Validation](#)) will cover it as a different Validation discipline.

5 Background Concepts

5.1 HVM (High Volume Manufacturing)

HVM is the process of efficiently manufacturing silicon products in millions of units. One balancing act within HVM is to keep outgoing silicon quality high (so customers do not receive bad parts) while keeping manufacturing costs low. The amount of time a part has to go through screening has a direct impact to the bottom line. Good DFT enables effective screening with low test time. Bad test quality and large test time can multiply to shave millions of dollars off a product's profit to Intel.

5.2 Sort Testing

Sort testing is the wafer-level testing to screen out defective dies. Only good dies cut out from the wafer will be packaged for the next level of testing. If bad parts are identified at Sort, we save the effort of later steps in the manufacturing process – such as packaging. Sort Tests include Structural tests (SBFT, SCAN, PBIST/MBIST).

5.3 Burn-in

After die are packaged they are put through Burn-in. Burn-in offers one of the most effective methods of reliability screening at the component level by running individual parts under high temperature and voltage to induce failure (infant mortality) of marginally working parts.

5.4 Class Testing

Class testing is the package-level testing after the good die is assembled into the package and run through Burn-In. Class Testing is used to “bin” parts based on frequency, or other criteria (working cores, or cache). Class Tests include Structural tests, Functional Tests, and IO Tests.

5.5 PPV

Product Platform Validation is used as a final screen to understand the quality of the overall test program. PPV is in-system testing, and can be used to identify defects not identified in Class test. PPV is expensive, so there is a strong desire to minimize its use. As the quality of other screens improves over the program’s lifetime – the goal is that PPV becomes limited to a monitor for DPM (Defect per Million) rates, and we can avoid the cost of using it on every part.

5.6 Tester

Tester is the platform for applying testing stimulus to the parts and checking the response. There are several types of tester platforms: functional tester, structural tester, debug tester etc. The Tester is a Stored Response Engine, where responses are all fixed. We know in advance at any given cycle what value we will drive in as stimulus. A real system by contrast has something connected to the other side of our part that understands the protocol and behaves dynamically.

5.7 Pattern Generation

Pattern Generation is the process of generating content and expected responses for HVM tests.

5.8 Types of HVM Content

5.8.1 Structural Test

Structural tests focus on SoC structures, such as Arrays or a set of combinational logic. The methods and tests used to debug arrays are collectively called “ArrayDFT” whereas “Scan” provides coverage of combinational logic.

5.8.2 Functional Test

The functional test methods used in HVM are similar to non-DFT tests. They utilize functional fabrics and caches and their operation is similar to non-DFT tests. Some examples are SBFT and TAM (more on that later).

6 DFT Validation

6.1 Introduction to DFT Validation

DFT Validation focuses on features used in HVM (High Volume Manufacturing) for enabling part testing and screening for defects. These test features include but are not limited to the following list:

Test Ports	TAP (Test Access Port)	Serial test access port. Standard IEEE protocol for debug and test access to silicon parts.
	STF (Structural Test Fabric)	Parallel test access port. Chassis standard that is popular for SoC designs. Parallel test fabrics allow us to reduce amount of time required for test by delivering patterns more quickly vs. TAP.
	MCI (Multi Cores Interface)	Parallel test access port. Popular for high-speed CPU designs. Parallel test fabrics allow us to reduce amount of time required for test.
Structural Test	Scan	<p>Scan is an industry standard method to test combinational logic between sequential elements. Scan insertion allows the control of clock, reset, and data input/output of flops before and after some combinational logic. Scan coverage levels are typically described in two ways: Stuck At (Stuck@) which is used to identify when logic is stuck at certain values, and At Speed (@Speed) to screen speed paths.</p> <p>Two common methods of Scan are used: Mux-D in SoC designs is lower in area cost but impacts timing, and LSSD where additional flops are added outside the critical path (popular in high frequency designs), which obviously translates to higher area and hence die cost but with the benefit of minimizing timing impact.</p> <p>ATPG (Automatic Test Pattern Generation) is an industry tool to support generating test patterns for the logic targeted by Scan.</p>

	ArrayDFT	Methods to target arrays and register files. One such method is by using BIST (Built In Self-Test) engines. These apply algorithmic array test patterns to detect faults and help with redundancy programming. Array DFT insertion tools used are sometimes in-house such as EPBIST, or from external vendors such as MBIST from Mentor. LYA (Low Yield Analysis) gives the ability to get current measurements of every transistor in a particular cell. This would lead to the type of defect a memory cell has, without doing any Physical Failure Analysis. For example - Open transistor, Missing contact, or shorted transistors.
Functional Test	SBFT (Structural-based Functional Test)	The functional test is pre-loaded into product's internal caches, and all cache requests are serviced by the pre-loaded data.
	TAM (Test Access Method)	TAM is a standardized IP that sits on the IOSF Primary Fabric. It is able to initiate requests and responses that mimic other IPs connected to the IOSF fabric.
HVM	HVM Reset	The HVM reset flow is a modification of normal platform reset to meet testing requirements (test time requirement and tester limitations).
	IOV	An acronym of acronyms: IDV – In Die Variation ODI – On Die Inductor VDM – Voltage Droop Monitor These are specialized circuits used for measuring and debugging the behavior of Silicon across the die.
	Burn-In	Burnin features will generate high signal activity, aimed at reaching a certain level of toggle coverage that is needed to induce failures in weak parts.
Other	BScan (Boundary Scan)	BScan allows direct IO controllability. One usage is for debug/test of board level connectivity of components.

6.2 Why do we need DFT Validation?

DFT features, like any other functional feature are required for any product to be successful, and critical to Intel's profitability, thus having a focus on high quality validation within the Pre-Silicon Validation team is vital to success. It is impossible to ship real products without high quality DFT. Without high quality DFT, we can have insufficient coverage to hit required DPM (Defects per Million) rates, we can add significant cost to screening parts, or have unsatisfactory rate of customer returns. Without DFT we could not understand and characterize our manufacturing process to enable high volume, and we would be unable to increase yield and frequency.

Having DFT Validation within the Pre-Silicon validation team aligns the methods used for DFT Validation with those best practices of the rest of the Pre-Silicon Validation community with regard to tools, flows, and methods as well as with schedule.

Having a dedicated DFT focus also enables more effective engagement with the HVM team and allows for continued efficiency and productivity improvements.

6.3 The Basic Characteristics of DFT features

In order to understand the challenges in DFT Validation, it is essential to understand first the characteristics of the features. This section provides a high-level introduction to DFT features without going into great detail. It explains a typical system, the DFT modes, and DFT usage models.

6.3.1 Typical DFT System

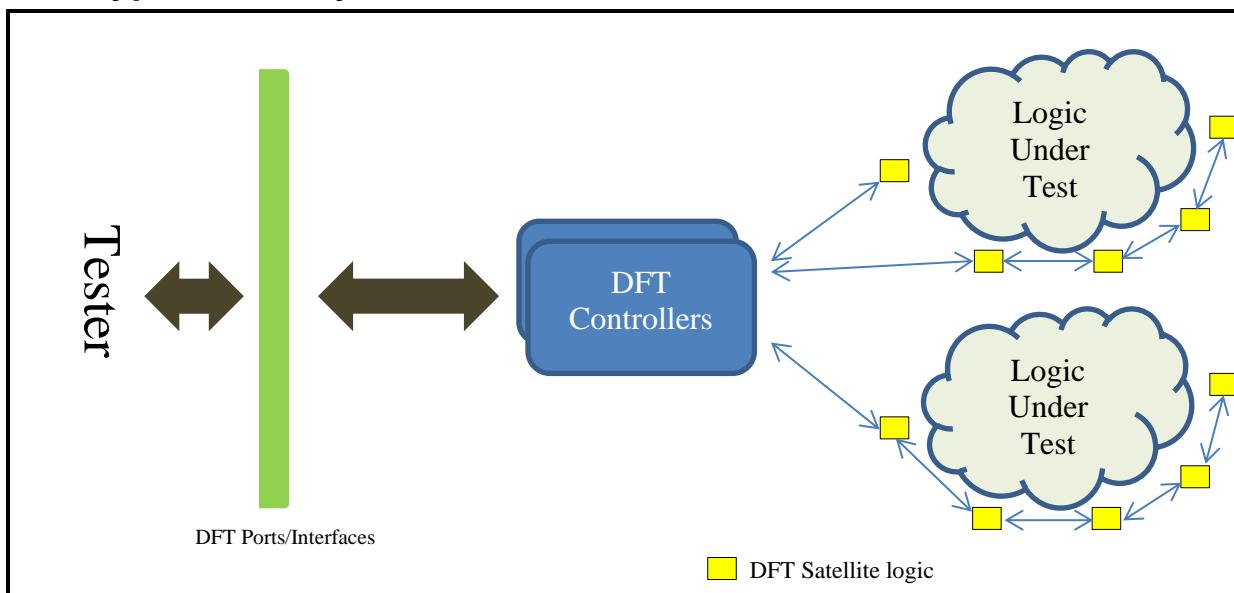


Figure 49: Typical DFT System

A typical DFT system (Figure 49) consists of the following components:

- DFT ports/interfaces
- DFT Controller
- DFT satellite logic

The following sections discuss these in more detail.

6.3.1.1 DFT ports/interfaces

The DFT ports/interfaces are the pin interfaces used by the tester to drive in the stimulus or receive the responding data. The TAP (Test Access Port) is usually the main test interface, but the TAP only allows data sent in/out serially at slow Tap-Clock speed. The parallel test port is used to increase data bandwidth (see STF and MCI in the DFT feature table above). In most projects, existing functional pins are redefined to serve as the parallel test port. For example, the Nehalem project reused CSI pins as the parallel test ports, while the Haswell, and Cannonlake projects reused the DDR interface as the parallel test ports.

6.3.1.2 DFT Controller

The DFT controller is the central data hub that plays the role of coordinating and transmitting data between the DFT interfaces and the internal logic. It usually implements the state machine for coordinating data in/out, buffering the data to accommodate incoming and outgoing data rate, or performing data translation from external format to internal data based on user configurations. The DFT controller can be as simple as a TAP FSM to coordinate data shifting to the targeted TAP data registers. More complex DFT systems can have more than one DFT controllers; each can have different features or form a separate DFT system.

6.3.1.3 DFT satellite logic

The DFT satellite logic, or local controller logic, is put near the logic under test for delivering the testing stimulus to the targeted logic and sending the response data out for checking. This local DFT logic is usually simple with a small footprint, allowing multiple instantiations in the Design, and multiple instances of the logic can be connected together to form one or more chains. The logic chains are then connected to the DFT controller so that data can be delivered to the local controller logic in a cost effective manner.

6.3.2 DFT Modes/Hooks

Besides the DFT system for transmitting the test data and controlling test data application, various DFT modes/hooks are also added to the design for various reasons. To list a few:

- Enable testing of the functional logic: a DFT mode is added to allow testing stimulus delivered to the logic under test and have the data response sent out (by intercepting the output data path). The surrounding logic (logic not under test) is also required to not disrupt the testing during DFT mode. For example, an array may be required to drive out static values during Scan test application.
- Reuse the functional logic infrastructure for testing purpose: To keep the design cost low, DFT features often reuse the functional logic infrastructure for testing purposes instead of creating dedicated logic. For example, the LLC and MLC caches are reused to store test stimulus and DDR and the memory controller datapath are reused to send testing data in and out. A DFT mode is added to allow the DFT usage of this functional logic.
- Reduce test time and tester cost: Since test time is one of the major parameters determining the cost of HVM testing, DFT hooks/modes can be added to reduce the overall test application

time. For example, DFT modes are added to suppress PLL relocking and shorten the pcode warm reset flow to enable faster warm reset during HVM testing.

These DFT hooks are usually simple logic. When the DFT modes are set, the behavior of the functional logic is altered for DFT usages.

6.3.3 DFT Usage Models

DFT is designed around how to minimize DPM, with higher Logic, Array and IO coverage. The usage sequence is defined to make use of the different DFT logic/hooks/modes in the DFT system to accomplish a specific testing or debug purpose. The usage sequence is generally a series of TAP or Parallel DFT operations for configuring the logic in DFT mode, sending in the testing data, and reading out the test result. As part of the usage definition, it may also require the sequence to be applied when the CPU or the logic is at a specific state. For example, the sequence may require the ability to perform the TAP unlock right after the power is good in order to be able to perform the fuse overrides before early fuse download has begun. The usage sequence should replicate the Post-Silicon usage situation. For instance, use-case DFT Validation should model the unconnected PCIe/DMI pins in tester to ensure the reset flow is making progress without them. The DFT Validator needs to have a detailed understanding of HVM Use cases to come up with tests that model HVM Behavior.

6.4 The Challenges in the DFT Validation world

Many challenges in DFT Validation are related to the nature of DFT features. This section attempts to describe some of the common issues and challenges.

6.4.1 IP vs. Integration Validation responsibilities

IPs are required and expected to validate all features, including DFx ones at IP level, and early in their design process. The PLC (Product Life Cycle - see [The Life of a Project](#)) has strict requirements on expectations for DFx features, and they begin early in the development process given the high impact to project backend. Validating IP specific features at integration level is impractical in terms of bug turnaround time and expertise of the specifics of that IP. The SoC DFT Validator is interested in validating the integration to the SoC's other DFT networks/components as well as top level connection fabrics.

If IPs are not produced with sufficient quality of RTL and associated collateral like RDL descriptions, it creates extra work for the SoC Integration team, to debug failures, and to utilize workarounds to make forward progress.

6.4.2 DFT logic is everywhere

To meet our DPM (Defect per Million) Targets, we need to cover a very high percentage of our chip's logic. Therefore DFT logic is present almost without exception in every IP, and even in the majority of MSFF's in the design. This requires the DFT Validator to interact with numerous

stakeholders and suppliers, and to be comfortable in the face of ambiguity – able to quickly understand different parts of the design.

6.4.3 False functional failures

A common pain point for DFT Validation is false failures from functional checkers. Oftentimes checkers are coded without regard to HVM/DFT usage. Deciding when to enable functional checkers and when not to, is a careful balancing act; false checker failures create wasted effort. The ROI of enabling these checkers needs to be carefully considered.

6.4.4 Some DFT Logic is directly implemented in backend flow

Scan presents a unique challenge, because the majority of the Scan logic is not present in the RTL model but instead inserted into the netlist. This requires unique approaches of using dummy scan chains for validation, and extra paranoia/evil validation is required with this gap in modelling in mind.

6.4.5 High Bug Volume

Given the high amount of logic impacted by DFT, bug rates are typically higher than other areas. Lack of familiarity with DFT can be another factor if logic is implemented or changed by those who are not DFT experts. A lack of familiarity with Tester limitations and real usage models can also be a source of bugs.

6.4.6 Validation environment does not resemble tester environment

The Pre-Silicon Validation environment is created to emulate the customer platform, but the tester environment can be very different from the customer platform. For example, for the HSW product, the PCIe and DMI interfaces were not driven on the tester due to a tester card limitation; therefore, a special test environment mode had to be created to emulate having no PCIe/DMI driver.

Moreover, with the increasing number of products integrating multiple-chips on one package, the need to emulate the specific tester platform has also increased. For example, the CRW project had EDRAM integrated with the CPU on one package, so the CPU and EDRAM were tested as separate die at SORT. However, the Pre-Silicon test environment contained only a model with both components combined in one simulation DUT. For DFT validation, special environment changes were made to find unique bugs where the CPU and EDRAM die were separate.

6.4.7 Ownership Management

Given the large number of teams the DFT Validation team interacts with, one frequent challenge is clearly understanding ownership breakdown between all the players: IP Validation, SoC DFX Integration, Integration Validation, and HVM teams. The scope of Validation and ownership

needs to be clearly defined upfront to avoid bugs slipping through. See [Validation Planning section: Establishing Scope](#) for further details).

6.4.8 Random testing is virtually impossible

For most DFT features, full random testing is not applicable. This is because DFT features are designed for a specific usage, and most feature users are internal engineers who can be directed to follow a specific usage sequence. The ROI is low for developing environments and checkers that can support fully random usage sequences. It is important to ensure usage models are clearly defined to minimize the effort in writing directed tests to cover different possible usages.

6.4.9 Unclear usage models

DFT feature definition typically focuses more on the design costs (area, timing, power) rather than the feature usage models. Not having a solid usage model can lead to discoveries during validation or in Post-Silicon. Even with a solid definition, it can still be difficult for Validators to relate to the actual usages and the purpose without actually being the real users on the tester. For example, a Validator may understand what vector of data a BIST engine can generate with certain register configuration, but it can be hard to relate to the type of IO problem (crosstalk, etc) that the data pattern is trying to expose. Without a full understanding, it can be a challenge to see the real problem during debug. Moreover, the defined usage models often leave too much flexibility to users, leading to exploding numbers of possible test cases.

6.5 The mindset of a good DFT Validator

This section lists the mindset a good DFT Validator should have. Although they are not listed in this section, all Validators are expected to also follow the general Validation principles, such as knowing their features well, creating a thorough testplan, developing good debug skills, and interacting well with their design partners in resolving bugs or issues. See [The Validation Mindset](#) for an in-depth look at the basic skills and mindset every Validator needs.

6.5.1 Know your feature and its usage models well

All Validators must learn the feature well before they can perform good Validation. Validators might not always own a feature from the beginning as they might inherit a feature from another product team or from their fellow Validator in the middle of the project. Even so, investing sufficient time to learn their feature well is still the most important step for the feature owner. For DFT Validators, learning the feature includes learning its usages – the usage sequences and the usage conditions, such as tester setting, ratios, SoC state when the usage sequence is applied, etc. The test cases developed should closely resemble the actual usage models of the feature. A measure of a good DFT Validator is their understanding of the reason behind the usage flow definition. This information can benefit the Validator during debug in separating a real hardware bug versus usage discoveries. The Validator can also use the knowledge to suggest alternative flows to accomplish the same purpose or find workarounds to bugs.

6.5.2 Actively engage in usage models definition

DFT Validators should ensure that their feature has solid usage models defined by actively participating in the definition process. Effort should be made to reduce the number of possible usage models – if there are multiple sequences utilized for a specific testing (or debug) purpose, only one way should be supported. The supported usage model should be documented well to avoid confusion from the users. During test plan reviews, the Validator should make it clear what usage flow will be validated and what will not be validated.

6.5.3 Acquire good breath of knowledge on the functional feature

Apart from learning their DFT feature, DFT Validators should acquire a good breath of knowledge on the functional feature under test. For example, array DFT Validators should not only understand the DFT access mechanism to the array, efforts should also be made to understand the functionality of the array during normal mode, the functional transactions to the array, the input and output datapath, and how DFT data intercepts the datapath. This knowledge will help accelerate debug; moreover, the Validator can also work more efficiently with the design in communicating bugs, suggesting fixes, and finding workarounds.

6.5.4 Prepare to deal with ambiguity and take advantage of others' expertise

Despite attempts to learn the functional feature, DFT Validators have to be prepared and be comfortable working in areas without knowing all the details. To effectively deal with ambiguity, Validators must make assumptions on the unknown area based on their high-level knowledge of the functional feature; continue making progress in their validation area, and use data from simulation to confirm their assumptions.

In addition, it is important to take advantage of others' expertise. Prior to approaching an area expert, Validators should do enough preparation and know exactly what questions need answers. In addition, it is more effective to communicate clearly the issues in the language that the area expert can understand as most of them are not experts in testing or DFT features.

6.5.5 Develop Flexible and Scalable DFT environments and Tests

Additional DFT logic can be added and existing DFT logic can change in the middle of the project as designs converge. Hence, the DFT test environment and tests have to be highly flexible and scalable to accommodate any additional testing logic without a major restructuring or rewriting. Similarly, test environments and tests should be scalable to work in different simulation DUTs. Validators should consider any possible opportunity to automate the validation process to reduce on-going maintenance efforts. This is especially important to enable supporting multiple derivatives and SKUs.

6.5.6 Define clear Validation scope

For DFT features that involve multiple Validation owners, it is essential to define and document the DFT Validation scope clearly. The DFT Validator should ensure that other owners understand the DFT usage and their Validation scope. It is a good idea to have enough overlap between different Validation owners to make sure no bugs can squeak through the Validation gaps.

6.5.7 Find bug workarounds for forward progress

Like other functional bugs, fixes for late-found DFT bugs can be postponed to future product steppings. This is especially true for non-fatal DFT bugs; for example, bugs that only result in loss of test coverage. Therefore, Validators should be prepared to find alternative flows that might workaround the bugs for forward progress.

6.6 The Role of DFT Validation across the project phases

6.6.1 Tech Readiness

During Tech Readiness, DFT Validation is involved in the architecture definition by helping to assess the Validation impact for the proposed features. Validators play the check-and-balance role to avoid the introduction of features with no clear benefits. While feature definition usually focuses more on logic functionality and design cost (area, timing, power), Validators need to ensure that the usage models are also well defined. Similar to other Validation teams, DFT Validation team members also work on defining Validation methodology and tool improvements for the upcoming project.

6.6.2 VAL0.0->VAL0.5

By RTL0.5, DFT coding should largely be complete. The DFT Validator should be prepared to have completed basic exercise by VAL0.5. The goal is to flush out basic bugs in the design as quickly as possible. Besides basic exercise, the team also creates test plans for all Validation execution activities and developing the test environment needed to support the rest of validation. Validators should also be ensuring high quality regression tests are in the model turnin flow.

6.6.3 VAL0.8->VAL1.0

During this phase, focus shifts from basic bug finding to exhaustively validating every piece of the design. In most areas, the main test cases during this phase are usage model test cases and corner cases. Upon completion of VAL0.8, an exit review will be done with the stakeholders to ensure that the feature has achieved necessary tape-in quality.

6.6.4 Post-silicon

Similar to other Validation domains, DFT Validation team will move on to working on different steppings and proliferations of the project while also supporting any Post-Si debug requests after 1st silicon tapein. Even where there are no changes in DFT features, the functional feature changes can still create DFT bugs. Therefore, feature health needs to be maintained for every product stepping.

6.7 Interaction with other teams

6.7.1 High Volume Manufacturing

HVM is one of the main customers of DFT Validation as they will be directly impacted by the validation quality. The test content group within the HVM team relies on the DFT features to develop the HVM test content, such as scan, array test, SBFT, Burnin, etc. The DFT Validation team validates the DFT features according to their usage models. Knowledge sharing of usage models, of workarounds and limitations, and of flows is important for success.

6.7.2 IP Integration Teams

IP Integration Validation teams often own targeted DFT Validation of their IP, such as TAP Basic Access testing. This allows the integrator to utilize their connections to the IPs with regard to IP specific questions and to perform the appropriate testing as IP drops occur.

6.7.3 Other Validation teams

DFT Validation is based on the methodology developed by the functional validation teams. In addition, DFT Validators will take advantage of the functional feature knowledge and test environment knowledge of the functional feature Validator when debugging failures requiring deeper functional feature knowledge.

7 Summary

DFT features are important for a product to achieve their testing and debug goals. Having a dedicated DFT Validation team is necessary to ensure sufficient Validation attention is given to DFT features. In many ways, DFT Validation methodology is very similar to those of other Validation teams. The unique challenges of DFT Validation are usually due to the nature of the DFT features and interaction with unique customers (HVM).

8 Future Work

8.1 Defining VJT Model with HVM Team

8.2 Roles and responsibility between Horizontal DFX and Product DFX Teams

9 References

This section intentionally left blank.

The Art of Validation: Chapter 30

Design for Debug Validation

By: [Erik Samuelson](#)

1 Abstract

Like much of Validation, Design for Debug Validation (DFD) is an art. In this chapter, you will learn about what makes DFD unique as a Pre-Silicon Validation discipline, the history of Intel Debug Features, the role of DFD across a project's life, and the principles of a successful DFD Validation program.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	10/4/2011	Initial Draft	Erik Samuelson	
1.1	12/6/2011	First Reviewed Doc	Erik Samuelson	Leslie Ong, Michael StClair, Ofir Chen
1.2	12/15/2011	Second Reviewed Doc	Erik Samuelson	Michael Bair
1.3	1/25/2016	Updates/Modernization	Erik Samuelson	Clint Whiteside, Michael Bair

3 Contents

1 Abstract.....	647
2 Chapter Revision History	647
3 Contents.....	648
4 Purpose.....	650
4.1 Why do we need this chapter?.....	650
4.2 What does this chapter cover?	650
4.3 What does this chapter not cover?	650
5 Background Concepts.....	650
5.1 Sighting.....	650
6 DFD Validation.....	650
6.1 The History of Intel Debug Features	651
6.1.1 Early Debug Methods	651
6.1.2 Golden Age of Debug	652
6.1.3 The Fall of the Logic Analyzer	652
6.1.4 A Probeless World	652
6.1.5 SoC transition	652
6.1.6 Future Directions	653
6.2 DFD Features	653
6.3 DFD Role across the project timeline	654
6.3.1 Tech Readiness.....	654
6.3.2 VAL0.0->VAL0.5	654
6.3.3 VAL0.8->VAL1.0	655
6.3.4 Post-Silicon.....	655
6.4 DFD Principles.....	655
6.4.1 Principle: Keep Debug Features Simple and Cheap.....	655
6.4.2 Principle: Do things the “Post-Silicon Way”.....	655
6.4.3 Principle: Steal and leverage Functional test content to execute with Debug Features	656
6.4.4 Principle: Maintain focus on DFD Validation	657
6.4.5 Principle: Be prepared to deal with late or non-engagement from Post-Silicon	657

6.4.6	Principle: Beware of the over-use of Loaners	657
6.4.7	Principle: Don't get distracted by Post-Silicon support.....	658
6.4.8	Principle: Not all Debug Features are created equal	658
7	Summary.....	658
8	Future Work.....	658
9	References.....	659

4 Purpose

4.1 Why do we need this chapter?

Design for Debug (DFD) features are critical to the Post-Si portion of the project. Validation, platform, and software teams, as well as external customers, utilize them to debug failures and platform issues. DFD must ensure high quality DFD features to meet these needs.

4.2 What does this chapter cover?

This chapter will seek to describe the aspects of DFD that are different from other Validation disciplines.

4.3 What does this chapter not cover?

Many elements of DFD are similar to other Validation disciplines, such as the need for quality test plan writing, efficient debug, how to execute tests, etc. For the most part, DFD techniques used are based on those of other Validation teams, unless where otherwise noted. As a starting point – the reader should become familiar with general validation topics.

This chapter does not cover interactions with Post-Si Validation in depth – please see the [Interaction with Post-Silicon Validation](#) and [The Post A-Step World](#) chapters for that.

5 Background Concepts

5.1 Sighting

A sighting is a Post-Silicon record of a failure that has a better-than-fair probability of resulting from a silicon bug.

6 DFD Validation

DFD Validation is focused on logic validation of features targeted towards enabling Post-Silicon debug and validation capabilities. In contrast, DFT Validation ([Design for Test Validation](#)), is focused on features used for high volume manufacturing to test the correct manufacturing of a given part. There is often re-use of features between DFT Validation and DFD Validation, such as those used in snapshot debug (Array Freeze & Dump or AFD). Debug features used for visibility and debug of analog circuit behavior are typically owned by MSV ([Mixed Signal Validation](#)), and are not covered in this chapter.

Silicon DFD features are critical to resolving sightings efficiently and quickly. Without quality DFD features, we would be left to do black box debug (low visibility), severely lengthening the back-end (Post-Silicon) schedule. Debug features are also increasingly a feature for OEM customers to do system level debug and software tuning. Quality DFD features can mean the difference between resolving issues in hours versus days or weeks. DFD therefore plays a critical role in the time to market of our products and those of our customers.

As a rule, features whose correct functionality is required on silicon must be validated. Debug features are no exception. In terms of priority, DFD features need to rank high on a project's priority list. We know we will have problems to debug on silicon – quality DFD features have a major impact on our ability to quickly close issues and meet our project schedules.

Maintaining a specific focus on DFD ensures that DFD feature validation is not postponed to the end of the project life cycle, an inevitable outcome if DFD is treated as one of many items to be covered by the primary functional validation teams. Because DFD features are often not customer-visible features (although that is increasingly changing, more on that later), it is often impossible for the main Validation team to prioritize it above mainline functional validation. If a project progresses through execution without a specific DFD focus, DFD features will be tested late in execution. In this situation DFD feature quality will be compromised, compromising Post-Silicon debug capability in the process.

6.1 The History of Intel Debug Features

The DFD features and methods used have changed dramatically over the last few decades. The following provides a glimpse into the enormity of those changes; this in turn gives perspective into DFD's growing responsibility.

6.1.1 Early Debug Methods

Early Debug methods relied on two principle technologies, Array Freeze & Dump (AFD), now dubbed “snapshot debug”, and Logic Analyzers. Snapshot debug is a mechanism that provides visibility into many of the arrays and scan nodes on the chip. It provides a static snapshot of what is going on in the CPU. This technique provides high visibility for a single point of time.

Logic Analyzers (LA) use external probes to observe chip pins, and present a trace of behavior to the debugger. For many projects, this meant the Front Side Bus (FSB), an interface between the CPU and Chipset. LAs offer a continuous picture, but of a single interface.

The usage of these two techniques is often referred to as manual debug. The key advantages were that the Silicon area cost (and up-front architecture cost) of these features is extremely low, given the fact that AFD relies on DFT mechanisms that are used to test arrays like caches during the manufacturing process. The downside of manual debug is the lack of visibility within the SoC, either in time or in state space. With a lack of visibility, debug time is typically a lot longer due to its iterative nature.

6.1.2 Golden Age of Debug

With the introduction of PSMI (Periodic System Management Interrupt) on Pentium Pro, and particularly with its productization on Pentium 4, a golden age of debug was born. PSMI is a technique used to port a behavior from Silicon onto a simulator, with cycle-for-cycle accuracy. PSMI provided full visibility into the CPU for a long duration of time. For many CPU generations PSMI was the preferred debug tool, and resolved the majority of Post-Silicon sightings. PSMI turned Post-Silicon debug into a Pre-Silicon debug challenge, quickening the path to sighting resolution.

6.1.3 The Fall of the Logic Analyzer

Starting with the Nehalem generation, the ability to capture traces of all interfaces with high accuracy began to greatly suffer. The introduction of new high speed external interfaces born out of the transition away from FSB to integrated Memory Controllers, and high speed serial buses (QPI, PCIe) in multi-processor systems, were responsible for this loss of accuracy. PSMI, with its reliance on accurate probing, became challenging and error prone. Nehalem saw a large falloff in the rate of usage of PSMI. The elimination of the FSB also posed a large challenge to the ability of OEM customers to debug their systems. The information they were accustomed to getting was now hidden in buried interfaces between the Core and Uncore. It was clear that our dependency on Logic Analyzers had become a major liability.

6.1.4 A Probeless World

To meet the debug challenge of buried internal interfaces, and the lack of accuracy and high expense of LA technology, on-chip tracing technologies were developed. This included OCLA (On-Chip Logic Analyzer) introduced on the Beckton server program and GDXC (Generic Debug eXternal Connection) introduced on the SandyBridge program. These techniques allow tracing of internal signals with capture either to a portion of the cache, to external DRAM, or to a dedicated debug port captured via Logic Analyzer. These probeless and semi-probeless techniques were also used to solve some of the challenges PSMI faced on the Eagleton and Haswell programs by providing the set of inputs needed for PSMI Replay.

Techniques like NOA (Node Observability Architecture) and VISA (Visualization of Internal State Architecture) were also introduced to allow continuous observability of a small number of internal signals chosen from a larger set of available signals on a sighting-specific basis.

6.1.5 SoC transition

The SoC transition has led to the introduction of DFx Chassis in an effort to standardize DFx interfaces and technologies, including DFD. VISA is now a standard for visibility of internal signals, and most IPs include this important interface. Some that which to delivered protocol or packetized trace are connected to a DTF (Debug Trace Fabric). Common re-usable elements (IPs) have been introduced for trace output on DTF (Generic Sniffer). Beyond fabrics, the SoC transition also introduced common re-usable IPs for Trace Aggregation (Northpeak) and for

Closed Chassis Debug (EXI). In a Closed Chassis, access to debug hooks/interfaces is restricted, because the system under debug is a customer solution or FFRD (Form Factor Reference Design).

6.1.6 Future Directions

To address the challenge our OEM customers face with debugability (high cost, low visibility LAs), the need to productize DFD features in a secure way is critical. Customers are demanding inexpensive, easy-to-use, mature debug techniques, and we must deliver them to remain competitive. The VCU, or Validation Control Unit, DCI, or Direct Connect Interface, and Debug Butler, are all methods for allowing secure access to debug hooks.

As Intel CPU design becomes more SoC-like (System On a Chip), convergence in debug methods across different Intellectual Property (IP) providers is critical. Defining standard interfaces and methods remains an ongoing challenge.

VISA remains complex for synthesis to meet timing for. An alternative approach likely to gain traction is to packetize VISA and send over the DTF fabric.

6.2 DFD Features

DFD Validation focuses on features used in System Validation for platform, IP, and component debug. These debug features include but are not limited to the following list:

Debug Access	TAP (Test Access Port)	Serial test access port. Standard IEEE protocol for debug and test access to silicon parts.
	EXI	Embedded DFX Interface, allows closed chassis access of debug features and trace.
	VCU	Validation Control Unit is a microcontroller for secure access to debug features and for survivability of bugs.
Trace	VISA	Visualization of Internal Signals Architecture, is a pervasive method to provide visibility of internal signals in the design. VISA is composed of a series of Muxes that allow for signal selection across the design.
	DTF	Debug Trace Fabric is a fabric for the delivery of trace messages from IPs to a trace aggregator.
	FTH	Fabric Trace Hooks – allow observation of fabric messages, such as IOSF Primary by mirroring packets to the Trace Aggregator
Triggering	RTB	Regional Trigger Block – is a hierarchical network for taking triggers and delivering them to different places such as the trace aggregator. Older technologies include CTB – Cluster Trigger Block, which served a similar

		purpose. Complex triggers can be created which look for complex events from multiple sources.
	uBreakPoints	Common method for Cores and Graphics engines to match certain events and create a trigger.
Aggregation	Northpeak	Northpeak is a trace Aggregator. It is able to collect traces from several sources (software messages delivered over functional fabrics) with hardware messages delivered over DTF or VISA, and output to different sources such as Memory, Display Port, or Pins. Included as well is capabilities for identifying, triggering, and counting events.
Run Control	Run Control	Run Control, or Probe Mode, is the ability to halt and restart core or microcontroller operation to allow inspection of logic. Some projects have a module called the RCM (Run Control Module) for coordination of break across multiple cores and firmware agents.

The primary goal of DFD Validation is to find the functional/logic bugs in DFD features and validate the usage model flows System Validation will use.

6.3 DFD Role across the project timeline

6.3.1 Tech Readiness

During Tech Readiness, DFD is actively involved in architecture definition. Often DFD must bridge the gap between Post-Silicon (end customer) requests and the Pre-Silicon world (feature implementation). In addition, DFD impact must typically be assessed for every proposed feature. How we will debug any given project feature, and how the feature interacts with our DFD features are central concerns. For example, the addition of a new power management mode may impact our ability to re-use a trace storage solution, or a new temperature sensor may impact our ability to use PSMI Replay to debug.

6.3.2 VAL0.0->VAL0.5

By RTL0.5, DFD coding should largely be complete. The DFD Validator should be prepared to have completed basic exercise by VAL0.5. The goal is to flush out basic bugs in the design as quickly as possible. Besides basic exercise, the team also creates test plans for all Validation execution activities and developing the test environment needed to support the rest of Validation. Validators should also be ensuring high quality regression tests are in the model turnin flow.

An important risk to combat is a lack of Post-Silicon engagement. Often Post-Silicon engineers are fully engaged in an existing silicon product, yet the train of future projects continues unabated. Seeking Post-Silicon support for solutions and re-using Post-Silicon software all require early engagement from Post-Silicon engineers. In some cases, *DFD may need to move forward without Post-Silicon support – by investing in alternative software solutions or validation strategies* – but obtaining synergy with Post-Silicon users is the preferable path.

6.3.3 VAL0.8->VAL1.0

During this phase, focus shifts from basic bug finding to exhaustively validating every piece of the design. In most areas, the main test cases during this phase are usage model test cases and corner cases. Upon completion of VAL0.8, an exit review will be done with the stakeholders to ensure that the feature has achieved necessary tape-in quality.

6.3.4 Post-Silicon

After Silicon, DFD team members typically become at the minimum peripherally engaged in Post-Silicon execution, and at the extreme wholly consumed by Post-Silicon work. Issues with DFD features are certain to arise, and the experts will need to be called in. During this time, however, new steppings and proliferation projects will also need attention, and it is critical that eyes remain on DFD feature validation for these future projects. Establishing a Post-Silicon owner for debug tools and methods is important to allow the team to continue to focus on its Pre-Silicon deliverables. A past successful method to enable a cleaner handoff is through the use of tour-of-duties (TOD). These TOD participants develop a rich in-depth feature knowledge, that they can then take with them to the Post-Silicon world. However, an over-reliance on Post-Silicon TOD participants can also be a liability when the resources return to Post-Silicon (see 6.4.6). Remember the train of proliferations and future steppings continues, and that will require Pre-Silicon attention too.

6.4 DFD Principles

6.4.1 Principle: Keep Debug Features Simple and Cheap

Debug features need to remain cheap, in terms of area and power, as they will carry a footprint on each part Intel sells. While some DFD features are productized for OEM partners, many remain only for use by Intel. Keeping features “cheap” also forces us to re-use functional paths, for example re-using memory for trace storage. While that re-use does come with some cost, for example, intrusiveness to functional behavior that can impact failure reproduction, it also means we leverage the validation and design effort of those functional paths. This increases the odds of a correct implementation and reduces validation cost.

Often times, as the bridge between Pre-Silicon and Post-Silicon, DFD must reign in the scope and complexity of DFD feature requests to the realities of available resources. Fitting DFD features into existing functional features is an excellent strategy to achieve this.

6.4.2 Principle: Do things the “Post-Silicon Way”

Wherever possible – doing things the Post-Silicon “way,” i.e. modeling the actual debug usage method, is the preferred strategy. For example, for on die trace validation the preference would be to use the real trace reconstruction software algorithms and real trace setup software to recreate the captured interface and check that against the actual interface. Proceeding in this manner can

identify holes in the software algorithm, and may point to additional actions the hardware needs to take. Sometimes this principle will allow us to fully re-use existing software, or software that needs to be created anyway. This can reduce development time, and also add confidence in the Post-Silicon software.

Doing things the Post-Silicon way often means that:

- DFD is in the position of doing the first level of integration between software and hardware, and testing the complete usage model of the feature.
- DFD identifies architectural issues earlier in the design process.
- Validated tools and collateral can be shared between Pre-Silicon and Post-Silicon.
- DFD must be prepared to effectively communicate software requirements – such as with trace reconstruction or array snapshot sequences.

Wherever possible, the DFD Validation team, should do things the “Post-Silicon” way.

6.4.3 Principle: Steal and leverage Functional test content to execute with Debug Features

We desire to use our DFD features during any kind of functional traffic. It follows, then, that for validation of many DFD features we typically wish to test them along with a variety of functional traffic. For much of DFD the desired functional coverage space is often largely similar to that of functional teams. Largely the coverage space is a representative sample of functional test content, with DFD features enabled. Consequently, for many types of DFD tests we want to run tests that are representative of what the entire Validation team is doing. Doing so in a lightweight and inexpensive manner is a challenge. Duplicating coverage and verifying that duplication can be expensive.

Fitting our testing within the framework of other functional validation teams is therefore important, as is leveraging the test generators and tests of other teams. When creating test plans and strategies, how DFD will interact and incorporate functional test content is a paramount concern. In an ideal world, all Validators add their flows to central test generators and common areas that allow other teams to easily incorporate that test content. The reality is that often that content is not healthy enough to distribute globally, or that Validators are cautious about a global distribution, or that certain testing cannot be packaged easily for the consumption of others.

Another pitfall would be to rely solely on turnin regression lists for test content. While that can be a good starting point, there are often significant content holes there, particularly with complex or late features, or in areas requiring long tests. The DFD Validator therefore needs to keep an eye on what functional conditions are must-haves, and of high importance. Typically we are less concerned with every corner case uArch condition, but more interested in whether micro-architectural features are enabled and exercised at all along with DFD features. One method to achieve this would be simple breadth level coverage of interesting uArch features.

While achieving good breadth level coverage of uArch features is important for many DFD areas, there are aspects of DFD that are worthy of coverage in and of themselves. For example, with on die Tracing, do we reach overflow or backpressure conditions? Do we capture every kind of packet, including special packets like timing packets? (Full disclosure: this is an area DFD needs to improve in, as we have not actually employed any specific DFD coverage mechanism.)

To achieve good functional coverage, the DFD Validation team must not be afraid to heavily leverage (aka steal) the validation work of other teams.

6.4.4 Principle: Maintain focus on DFD Validation

Given the fact that DFD tests leverage functional validation tests, functional failures will be encountered. Caution is advised here. The DFD team's charter is to focus on validating DFD features, and it is important that that focus be maintained. Dealing then with functional failures is a challenge. Often DFD features can induce functional failures, so functional failures need to be examined, particularly if the failure signature is unique to DFD tests. However if a failure looks unrelated to DFD – it is best to find an owner from the functional validation team, or at a minimum to confirm that there are at least open bugs that can cause the failure signature.

A dedicated and continuous focus must remain on DFD Validation by the DFD team; hand off functional failures to other appropriate Validators.

6.4.5 Principle: Be prepared to deal with late or non-engagement from Post-Silicon

The Post-Silicon Validation team does not always have the same schedule or priority that Pre-Silicon Validation does. Post-Silicon Validation is working with priorities driven by PRQ schedules, where the Pre-Silicon Validation team is driven by Tapeout schedules. Gaps in Post-Silicon engagement can therefore be a common occurrence. This can introduce challenges to strategies of sharing tools and methods and reaching acceptable architectural solutions. Non-engagement may mean a best effort is needed in considering Post-Silicon requirements. It may mean that in order to make forward progress, Pre-Silicon software tools may need to be created.

The DFD team must be prepared to deal with late or non-engagement in architecture, design, and validation of Post-Silicon Debug Features.

6.4.6 Principle: Beware of the over-use of Loaners

The Post-Silicon Validation team is often very interested in offering loaners to assist with the validation of DFD features. This engagement helps fund validation, and is often used to fill critical funding gaps from the design team. It also enables the transfer of valuable skills to future Post-Silicon debuggers on the available debug hooks. However, the Pre-Silicon Validation team must be prepared to assume ownership of these features once silicon arrives – as the Post-Silicon Validator will quickly be consumed by silicon debug. The Pre-Silicon team must keep a close watch over these activities to ensure it can pick ownership back up. It also must ensure the quality of validation is done according to its standards.

An overreliance on loaners can be particularly difficult as the project switches to a post silicon focus. This is compounded by the fact that many on the DFD team get involved in Post-Silicon debug.

6.4.7 Principle: Don't get distracted by Post-Silicon support

As already noted, Post-Silicon and Pre-Silicon Validation teams are working with different priorities. As a corporation we tend to prioritize Silicon work ahead of Pre-Silicon work because of TTM (Time to Money) concerns. This will lead to conflicting priorities. It is very easy for DFD team members to be completely consumed with Post-Silicon support, and in many cases this is critical support that needs to be provided. However, without an eye to future steppings and projects, we put great risk on the debug-ability of our future projects. Balance is therefore required, as is developing Post-Silicon owners for Post-Silicon support. This can be achieved through training, Pre-Silicon Tour of Duties, and in constructing a division of labor in tools and support.

DFD Validation must maintain a focus on Pre-Silicon activities, lest quality suffer on future steppings and proliferations.

6.4.8 Principle: Not all Debug Features are created equal

To the DFD newbie, it will quickly become apparent how much demand there is to add this or that widget, a special mode here, a new capture source there. Paranoia about how to survive or debug this or that possible situation can quickly expand the list of DFD features. Just as quickly, the scope of validating all features with the highest quality can become unsustainable. DFD must play a role in constraining the feature list – or the team can very quickly become defocused from the features that really matter. On the other hand, a debug hook can often be added with very low cost and risk, with a high potential payout if it works and is needed (excellent examples are defeatures). In such situations it often makes sense to do a very barebones exercise – enough to give some confidence in the basics of the feature – but no more, less we risk being distracted from higher value DFD features. It is important to communicate to stakeholders that some risk is being taken with the correct implementation of that feature.

DFD Validation works in a world of constrained resources, with many desired debug hooks – focus must be prioritized on the highest ROI features.

7 Summary

A DFD Validation focus is critical to the success of our silicon projects. The quality of DFD features has a direct correlation on how quickly Post-Silicon debug can occur, translating directly to reducing TTM. DFD plays a critical role in the success of our CPU projects by taking part in defining the right DFD features, ensuring the correct operation of those features, and providing support to the Post-Silicon debugger.

8 Future Work

9 References

- [1] Chinna Prudvi, Sanjay Salem, Mike StClair, Erik Samuelson, Ray Ramadorai, “Revolutionizing System debug: The Haswell approach”, DTTC 2010,
http://dttc.intel.com/secured/2010/Presented_Papers/14732.pdf,
<http://dttc.intel.com/secured/2010/Presentations/14732.pdf>
- [2] Erik Samuelson, “PSMI 2.0”, DTTC 2011,
http://dttc.intel.com/secured/2011/Presented_Papers/16323.pdf,
<http://dttc.intel.com/secured/2011/Presentations/16323.pdf>
- [3] Michael St. Clair, “Array Descriptors: A Method For Mapping Array Properties To A Common Abstraction”, http://dttc.intel.com/secured/2006/Presented_Papers/4399.pdf

The Art of Pre-Si Val: Chapter 31

Microcode Validation

By: [Colin Wood](#)

1 Abstract

Microcode, or *uCode*, Validation is one aspect of firmware validation, verifying that uCode carries out the requirements of the IA32 architecture. This chapter discusses the methodologies and tools of the uCodeV team, as well as its relationships with other teams.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	9/15/2011	Initial Draft	Colin Wood	Michael Bair / CCDO Val Staff

3 Contents

1 Abstract.....	661
2 Chapter Revision History	661
3 Contents.....	662
4 Purpose.....	663
4.1 Why do we need this chapter?.....	663
4.2 What does this chapter cover?	663
4.3 What does this chapter not cover?	663
5 Background Concepts.....	663
5.1 Microcode	663
5.2 Microcode Patch	663
6 Microcode Validation.....	664
6.1 uCodeV Philosophy	664
6.2 uCodeV Fundamentals	664
6.2.1 Tests.....	664
6.2.2 Simulation	665
6.2.3 Checking.....	666
6.2.4 Coverage	667
6.3 Interaction with Other Teams.....	669
6.3.1 Architecture and uCode Design	669
6.3.2 Core IA and Feature Architecture Validation.....	669
6.3.3 uCodeV Teams across Intel.....	669
6.3.4 Post-Silicon Debug	669
6.4 Challenges.....	670
7 Summary.....	670
8 Future Work	670
9 References.....	670

4 Purpose

4.1 Why do we need this chapter?

Microcode is as critical a part of an IA32 processor as the hardware itself, and thus it needs to be validated to produce a usable product. While Microcode Validation is similar in many respects to other Pre-Silicon Validation disciplines, there are several significant differences, and understanding those requires a grasp of uCodeV philosophy and methodology.

4.2 What does this chapter cover?

The chapter covers the basics of uCodeV philosophy and the fundamental activities composing uCodeV methodology along with a high-level description of the tools used to implement these activities.

4.3 What does this chapter not cover?

This chapter does not cover in-depth methodology specific to any given microcode flow, nor any in-depth look at the tools used by uCodeV.

5 Background Concepts

5.1 Microcode

Microcode is the processor firmware found in the core of a CPU that implements IA32 architectural instructions and features. The flexibility of microcode allows creating architectural features which would be difficult or even impossible to implement directly in hardware. Microcode also implements various exception handlers and micro-architectural facilities such as inter-thread/core/package communication. Each microcode flow consists of a sequence of one or more micro-operations (uops) and is said to implement an algorithm. Microcode flows are categorized into various areas of related functionality such as segmentation, virtualization, or floating-point arithmetic.

5.2 Microcode Patch

Since the P6 (Pentium Pro) processor generation, IA32 CPUs have had the capability to provide post-tapeout changes to processor firmware, commonly referred to as a “microcode patch”. Patching is a powerful feature that enables Intel to work around many bugs (both hardware and firmware, and not necessarily in the CPU) on a real silicon platform. Since patches can be shipped as part of production systems, patching enables Intel to reduce time-to-market (TTM) by avoiding further steppings to fix the bugs in the physical design. Patches also contribute to survivability of bugs (a.k.a. errata) found after production release of processor platform, avoiding the need to recall shipping products.

6 Microcode Validation

6.1 uCodeV Philosophy

The primary task of any Validation team is to deliver a product of sufficient quality that subsequent consumers of the product can make use of it. For uCodeV, this means ensuring that the microcode algorithms themselves are correct and that uCode bugs have a minimal impact on the work of other Validation teams, including the Post-Silicon (Post-Si) teams. Of course, uCode never reaches this goal, but it is worth striving for, nonetheless. Unlike many hardware areas, bugs in uCode are rarely fatal to tape-in or PRQ due to the ability to patch uCode, and thus it is not absolutely necessary to ensure complete correctness. Trade-offs can be made between earlier and later effort, but constraints such as the availability of patch resources must be taken into account.

In pursuit of its goal, the uCodeV team has several advantages not afforded to other Validation disciplines. First, testplans are unnecessary since a list of testcases are easily generated from the coverage space which itself is automatically generated from the uCode source. Second, the uCode test environment is such that a test builds in a short period of time (seconds to a few minutes). Third, the uCode simulation environment is fast enough that a typical test run requires well less than a minute. Taken together, this means that a uCode Validator can spend more of his or her time focusing on the highest ROI activities such as test writing, coverage analysis, and debug instead of being weighed down by overhead activities such as testplan writing or waiting for test runs to complete.

Other Validation disciplines are generally responsible for ensuring the correctness of the uCode algorithms in combination with the actual hardware such as [Architecture Validation](#) and [Feature Architecture Validation](#) disciplines. Members of the uCodeV team frequently attend testplan reviews held by these other teams in order to provide input on various interesting conditions that need to be covered.

6.2 uCodeV Fundamentals

Like most Validation disciplines, microcode validation relies upon three fundamentals: stimulus, checking, and coverage. Here we break out stimulus into its two sub-components: tests and simulation.

6.2.1 Tests

As microcode implements the IA32 processor architecture, most uCodeV tests are architectural in nature and thus are fundamentally assembly tests similar to those used for core and fullchip validation. The uCodeV team employs a number of different types of tests from directed self-checking tests to fully random instruction test (RIT) generation tools such as Café or vtrand. Due to the ever-expanding size of the IA32 architecture and the never-expanding size of the uCodeV team, it is critical that quality tests be reusable from processor stepping to stepping and processor

generation to generation. This allows the uCodeV team to focus on validating new features with a minimum of time spent regressing inherited features.

6.2.1.1 Directed Tests

The uCodeV team uses directed self-checking tests for initial exercise of new uCode features, basing them on the External Architectural Specification (EAS) or Micro-Architectural Specification (MAS) for each feature. The advantages of using directed self-checking tests are that the tests are more likely to be usable for future generations and are less likely to break due to interaction with bugs in other features due to their focused nature. The uCodeV team typically employs the Testgen library of assembly macros (written in a perl-based macro-processing language called Max) when writing these tests. The uCodeV team will often re-use existing Testgen templates or create new variations of them in order to exercise new features. An additional advantage of using Testgen is that the tests are relatively quick to build as opposed to other test-generation tools where constraint-solving may incur significantly more overhead than test run time. Developers of the architectural checker (archsim) use these tests as a reference when implementing support for new features.

6.2.1.2 Random Test Generators

The uCodeV team employs several random test generators primarily for gaining code coverage. The Café RIT owned by the Post-Silicon Validation team sees extensive use by the uCodeV team. As Café is frequently late in providing support for new features and is not always focused on exercising microcode flows, the uCodeV team maintains several test generators directed at exercising specific areas (e.g. VT-x, TXT, or tasking).

6.2.1.3 Injectors

The uCodeV team also maintains a number of tools used to inject microarchitectural events into the simulator for coverage, for example error-correcting code (ECC) or power-management events. These injectors allow emulation of the interaction with other CPU agents, power-management controllers, cosmic-rays, etc., which are difficult to setup within a single-threaded test.

6.2.2 Simulation

A key component of uCodeV methodology is the microcode simulator, Microsim.

6.2.2.1 Functionality

Microsim simulates a sufficient portion of the CPU to allow execution of almost all microcode flows with a reasonable degree of accuracy. Microsim simulates instruction fetch and decode into micro-operations (uops), resource allocation and scheduling, out-of-order uop execution, in-order uop retirement, and any necessary event processing.

6.2.2.2 Speed

Microsim is relatively fast, capable of executing between 4000 and 10000 uops/s depending on the “branchiness” of the test code and the depth of speculative execution.

6.2.2.3 Limitations

Microsim has limited support for the Advanced Programmable Interrupt Controller (APIC) and translation look-aside buffers (TLBs). Microsim’s support for multiple-threads per core is still a work in progress. Microsim does not currently support simulating multiple cores. Microsim does not support processor data or instruction caches. Microsim is not cycle-accurate as compared to RTL simulation. Microsim does not support emulation of performance-monitoring hardware. It is possible to work around some of these limitations via injection, but exercise of some uCode flows must be performed on a higher-level RTL or emulation model. This does result in a small number of bug escapes from the uCode cluster to higher-level clusters, but these escapes are generally considered acceptable to the uCodeV team as they are rarely blocking to other validation teams.

6.2.2.4 MOOSE

The Atom uCodeV teams use a different microcode simulator called MOOSE. MOOSE has similar speed and functionality to microsim.

6.2.3 Checking

It is critical that uCodeV tests be checked for correctness. Writing self-checking tests can be quite burdensome, particularly for random test generators. For this reason, the uCodeV environment incorporates a number of automatic checking mechanisms. The two primary checking mechanisms are archsim and uCode rules.

6.2.3.1 Archsim

Archsim is the architectural simulator. Archsim can verify that each microcode flow matches the IA32 spec. The chekhov tool is used to compare the architectural state maintained by microsim with that calculated by archsim at the end of each microcode flow. If a mismatch is detected, chekhov flags an error and the test fails.

6.2.3.2 Microcode Rules

Microcode rules (also referred to as uCode restrictions) are a body of rules governing the interaction between a uCode flow and hardware or between one uCode flow and another. Some rules are unwaivable, meaning that a violation of the rule will result in incorrect operation of the machine. Other rules are more like good coding practices and can be waived provided the coder

knows what he or she is doing. The list of rules is kept in HSD and shared between projects. Each project inherits all of the parent project's rules and is free to modify or mark them as obsolete. Rules are checked by one of four mechanisms: the build tools, uc_lint, pathcheck, or microsim. The microassembler converts the uCode source files into machine code and the linker fixes up code references and places the machine code in uCode memory. Both can detect simple uCode rule violations. The uc_lint tool is a static analysis tool that parses the assembled microcode and looks for further violations of uCode rules not caught by the microassembler or linker. The pathcheck tool builds a list of possible paths through uCode and applies rule checks to each path. The advantage of this tool is that it enables detecting a number of bugs statically at build time that would normally require dynamic simulation in order to catch, thus pushing bug-finding ‘upstream’. Microsim dynamically applies a large list of rule checks when running a test. Validators code these rule checks such that erroneous uCode can be detected merely by exercising the offending flow as opposed to a traditional test where a functional failure is needed to explicitly expose the flaw. The majority of uCode rule checks are implemented in microsim. The uCode assembly syntax supports a waiver mechanism whereby a microcoder can ignore a given rule on a given line of microcode.

6.2.4 Coverage

Like many Validation teams, uCodeV relies heavily on coverage to determine the quality of its testing. There are several forms of code coverage used: path, statement, branch, and n-branch. Regardless of the coverage type, the pathfinder tool suite is used to create coverage databases, gather coverage statistics, and analyze them. Coverage databases are generated via a two-step process in which the ulst_convert tool first parses the assembled uCode into an intermediate constraint-language description of each flow. The pathogen tool converts this description into a SQL database for the various coverage spaces.

6.2.4.1 Path Coverage

A path-coverage space is the set of all possible paths through the various microcode flows. A path begins at the beginning of an instruction or event handler and ends at the end of an instruction, an event handler, or an event taken during an instruction or event handler. The default path-generation algorithm generates a new path set from each possible branch target. The algorithm is constrained via annotation added directly to the uCode source. The minimal annotation required is a description of indirect branch targets along with handling for loops. In order to eliminate impossible paths from the coverage space, additional annotation forbids following certain sequences of branch/target pairs. Path-pruning annotation is used to collapse similar paths together and prevent “path explosion” where the size of the path-coverage space is too big to build or analyze or else to simply eliminate paths that are not considered interesting. Through annotation, we identify the coverage conditions that we intend to target, and the process of writing annotation is - in some ways - analogous to the coding of AV or uAV testplans. For this reason, the uCodeV team generally maintains path annotation to a level sufficient for 2-branch coverage and only does analysis of full path coverage in specific, high-risk areas.

6.2.4.2 Statement and Branch Coverage

The process of full path-coverage analysis can be quite time-consuming, and experience and analysis have shown that the results of achieving extremely high levels (e.g. > 90%) of path coverage are often not worth the time invested in writing analyzing annotation. To this end, the uCodeV began pursuing the Optimal Coverage and Checking Automation for Microcode (OCCAM) methodology to automate more of the coverage generation process. OCCAM v1.0 focuses on generating the most basic forms of code coverage: statement and branch coverage. Statement coverage treats each uop in the assembled uCode as a coverage term. Several different sequences are considered to be branch coverage terms. The most basic case is a conditional or indirect branch uop followed by its target uop. Since conditional move uops serve as a convenient ‘if-then-else’ idiom for uCode, whether the condition of these uops is satisfied or not is also considered to be a branch. Sequences where an exceptional condition may occur in the middle of a uCode flow create two additional branch coverage terms: events and restarts. The former represents the transition from the original uCode flow to the event-handling flow. The latter represents the return back from the event handler to the original flow if the event type requires restarting the original flow. Code is considered to be fully exercised once statement coverage reaches approximately 98% and non-event branch coverage exceeds 95%.

6.2.4.3 N-Branch Coverage

N -branch analysis of path coverage conditions is an extension of branch coverage where the coverage terms consist of sequences of n branch/target pairs in the same microcode flow. The uCodeV team currently generates at most 2-branch coverage statistics as it was determined that 3-branch and higher coverage has extremely low return on investment. All n -branch coverage (where $n > 1$) requires that the subsequent branch/target pairs exist along the same path through microcode. The current tools require that a fairly complete path coverage database be built to make this determination. A simple area of uCode such as integer arithmetic may have few to no 2-branch sequences and is fully validated once statement and branch coverage targets are met. A complex area of uCode like VT-x is considered to be validated once 2-branch coverage reaches approximately 90%.

6.2.4.4 Coverage Limitations

The uCodeV coverage methodology does take into account interactions between uCode flows and various exceptional conditions. For example, coverage can determine that all memory operations in a flow can handle taking a segmentation violation fault. However, the methodology does not include coverage for other inter-flow interactions unless code for handling those interactions exists in a flow. The uCodeV coverage methodology also does not include data-space coverage. As a hedge against these limitations, the team attempts to gain the majority of its coverage from random or directed-random test generators.

6.3 Interaction with Other Teams

The uCodeV team must maintain interactions with a number of other teams at Intel.

6.3.1 Architecture and uCode Design

The uCodeV team should interact closely with the architecture and uCode design teams, especially during the Tech Readiness (TR) and Frontend Development (FED) phases of a project where most features are defined and initially implemented. The uCodeV team's insight into the complexity of features and knowledge of uCode interactions provides valuable feedback to the architects when developing new features. The feedback allows for creation of simpler features that are easier to validate now and in the future. At times, testing will reveal deficits in the initial architecture, and the uCodeV team will relate those issues and ensure that the features are redesigned correctly.

6.3.2 Core IA and Feature Architecture Validation

The uCodeV team must work closely with the Core IAV ([Architecture Validation](#)) and FAV ([Feature Architecture Validation](#)) teams as those teams are the direct consumers of uCodeV work. In addition, uCodeV must ensure that either Core IAV or FAV picks up validation of those uCode features that are not easily implemented in microsim, e.g. complex cross-core or timing-sensitive interactions. The Core IAV team also often relies on the archsim enabling efforts of the uCodeV team.

6.3.3 uCodeV Teams across Intel

As with any Validation discipline, it is important to share methodologies between teams across projects at Intel. The uCodeV team must strive to promulgate its various improvements not only to the next derivative but also to other non-derived projects.

6.3.4 Post-Silicon Debug

As experts in uCode debug, uCode Validators may often find themselves called upon to help debug Post-Si sightings. This help may be as simple as giving opinions on the possible cause of problems, but may also include looking at traces generated via PSMI or writing debug uCode patches to help expose critical state.

In addition, the Post-Si validation team gathers path coverage statistics via a Design for Validation (DFV) hook in the processor hardware that can calculate path signatures. The team uses the same Pathfinder tools to generate and analyze path coverage databases as the uCodeV team although the coverage spaces are different due to limitations in the DFV hardware's signature generation capabilities. The uCodeV team provides support for these efforts.

6.4 Challenges

One of the biggest challenges faced by uCodeV relates directly to the mutable nature of uCode: because it can be easily changed, it often will be. Much of the uCode collateral can be modified until a few weeks before tape-in, and it is possible to add some features entirely in a patch. Knowledge of this capability results in many late uCode-only feature requests, and the uCodeV team must be able to respond accordingly. It is crucial that uCodeV leadership be able to evaluate the complexity of these late requests and push back against them if necessary, even if “it’s only microcode”.

7 Summary

Microcode Validation is critical to the success of our silicon projects. The quality of uCode has a direct correlation on how quickly Post-Silicon debug can occur, translating directly to reducing TTM. Microcode Validation plays a critical role in the success of our CPU projects by taking part in defining the right features, ensuring the correct operation of those features, and providing support to the Post-Silicon debugger.

8 Future Work

The uCodeV team is pursuing further automation of coverage-space generation via OCCAM v2.0. The idea behind OCCAM v2.0 is to use various graph algorithms to derive the coverage space rather than relying heavily on path annotation. Those same algorithms will be used to provide better visualization of the coverage space.

9 References

The Art of Pre-Si Val: Chapter 32

Embedded FW Validation

By: [Colin Wood](#)

1 Abstract

The Firmware Validation (FiV) team owns validation of various forms of embedded firmware. This chapter discusses the methodologies and tools of the FiV team, as well as its relationships with other teams.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	2/1/2016	Initial Draft	Colin Wood	Michael Bair
1.1	8/10/2016	Release Version	Colin Wood	Michael Bair

3 Contents

1 Abstract.....	671
2 Chapter Revision History	671
3 Contents.....	672
4 Purpose.....	673
4.1 Why do we need this chapter?.....	673
4.2 What does this chapter cover?	673
4.3 What does this chapter not cover?	673
5 Background Concepts.....	673
5.1 Embedded Firmware	673
5.2 Patchable Firmware.....	673
6 Embedded FW Validation	674
6.1 FiV Philosophy.....	674
6.2 FiV Fundamentals.....	675
6.2.1 Simulation.....	675
6.2.2 Tests.....	676
6.2.3 Checking.....	677
6.2.4 Coverage	677
6.3 Interaction with Other Teams.....	678
6.3.1 Architecture and FW Development.....	678
6.3.2 SoC Integration Validation	678
6.3.3 Post-Si Validation	679
6.3.4 Other FiV Teams across Intel	679
6.4 Challenges.....	679
7 Summary.....	679
8 Future Work	679
9 References.....	679

4 Purpose

4.1 Why do we need this chapter?

Embedded FW is a critical part of modern computing components, whether a discrete CPU or chipset or a fully integrated SoC, and thus it must be validated in order to produce a working product. Firmware Validation is similar in many respects to other Pre-Silicon Validation disciplines, but there are several significant differences. Understanding the differences requires a grasp of FiV philosophy and methodology.

4.2 What does this chapter cover?

The chapter covers the basics of FiV philosophy and the fundamental activities composing FiV methodology along with a high-level description of the tools used to implement these activities.

4.3 What does this chapter not cover?

This chapter does not cover in-depth methodology specific to any given type of embedded FW, nor any in-depth look at the tools used by FiV.

5 Background Concepts

5.1 Embedded Firmware

Firmware is a form of software that provides low-level control of a device. Embedded firmware is that firmware which runs directly on microcontrollers in an SoC and is frequently contained within a ROM in the device, thus differentiating it from other platform-level FW which typically runs on the primary compute cores (e.g. BIOS). Most FW is written in C (albeit with proprietary extensions per microcontroller), but some is written in assembly (e.g. pCode written for the Foxton microcontroller).

5.2 Patchable Firmware

While most microcontrollers in an SoC have firmware embedded in a ROM, for PRQ acceleration and survivability purposes they also include some form of RAM to enable patching or extending the functionality available in ROM. Patching is a powerful feature that enables Intel to work around many bugs (both hardware and firmware, and not necessarily in the CPU) on a real silicon platform. Since patches can be shipped as part of production systems, patching enables Intel to reduce time-to-market (TTM) by avoiding further steppings to fix the bugs in the physical design. Patches also contribute to survivability of bugs (a.k.a. errata) found after production release of a processor platform, avoiding the need to recall products in the field.

Some firmwares support replacing the entirety of ROM with new code in RAM, others provide hooks to enable patching out individual routines or even individual lines of code.

Even for FW that is otherwise patchable, there may be areas that are not. Examples of these include early portions of the reset or boot sequences where FW must execute prior to patch availability. Extra scrutiny must be applied to these flows, as the risk of production-gating bugs is much higher.

6 Embedded FW Validation

6.1 FiV Philosophy

The primary task of any Validation team is to deliver a product of sufficient quality that subsequent consumers of the product can make use of it. For FiV, this means ensuring that the firmware algorithms themselves are correct and that FW bugs have a minimal impact on the work of other Validation teams, including the Post-Si teams. Of course, FiV never reaches this goal, but it is worth striving for, nonetheless. Unlike many hardware areas, bugs in FW are rarely fatal to tape-in or PRQ due to the ability to patch FW, and thus it is not absolutely necessary to ensure complete correctness. Trade-offs can be made between earlier and later effort, but constraints such as the availability of patch resources or the impact on validation forward progress must be taken into account.

In pursuit of its goal, the FiV team has several advantages not afforded to other Validation disciplines:

- Use of testplans is de-emphasized since a list of the most critical testcases is easily generated from the coverage space which itself is automatically generated from the FW source.
- The FW test environment is such that tests are interpreted and thus do not require a build stage.
- The FW simulation environment is fast enough that a typical test run requires well under a minute [1].
- The high level of abstraction in FW tests and the test environment contribute to significant re-use across derivatives and project generations.

Taken together, this means that a FW Validator can spend more of his or her time focusing on activities with the highest return on investment (ROI) such as test writing, coverage analysis, and debug instead of being slowed down by overhead activities such as testplan writing, waiting for test runs to complete, or revalidating legacy features.

Other Validation disciplines are generally responsible for ensuring the correctness of the FW algorithms in combination with the actual hardware such as [Reset and Power Management Validation](#) disciplines. Members of the FiV team frequently attend testplan reviews held by these other teams in order to provide input on various interesting conditions that need to be covered

Some teams at Intel attempt to validate FW at the corresponding RTL unit level; e.g. validating PMC FW in the PMC IP's RTL validation environment. At first glance, this might appear to be a good idea. After all, the hardware with which the FW directly interacts is all modeled in the corresponding unit's RTL, right? Unfortunately, FW operates on a higher level of abstraction than

the hardware upon which it executes and frequently needs to communicate with entities outside of its own unit. Validation at the RTL unit level frequently leads to one of two compromises: either add a tremendous amount of detailed modeling for the sake of validating FW, thus complicating the validation of the RTL itself, or else have very little modeling and risk insufficiently validating the FW.

In either case, RTL simulation is frequently too slow to test effectively some of the more complex and longer latency firmware interactions. Some teams at Intel attempt to work around this issue by using IP-level FPGA solutions. Unfortunately, an FPGA solution requires relatively complete and healthy RTL to be available before it is useful for running FW, and this can lead to significant schedule delays for FW development and validation.

6.2 FiV Fundamentals

Like many Validation disciplines, Firmware Validation relies upon the three pillars of [Stimulus](#), [Checking](#), and [Coverage](#). Here we break out stimulus into its two sub-components: simulation and tests.

6.2.1 Simulation

A fast firmware-only simulation environment is a key component of FiV methodology [1]. RTL validation teams typically rely upon simulation or emulation environments provided for them by external vendors. For FiV, this is usually not the case. FiV teams frequently develop and maintain their own simulation environments. There are several key points to keep in mind when developing such an environment.

6.2.1.1 Functionality

A FW Validator must be able to observe and control FW interactions at the level of abstraction expected by the FW. Typically this means being able to determine what FW flows are running at any given point in time, to trace FW accesses to memory-mapped IO registers, and to drive responses to FW via these same registers. Depending on the microcontroller architecture, precise interrupt injection may also be required in order to validate FW corner cases. A simple execution trace in a log file may provide sufficient visibility for assembly-based FW, but visibility for FW written in higher-level languages may involve the use of a source-level debugger if available.

6.2.1.2 Speed

...is the key! Typical microcontrollers can execute instructions anywhere from tens of thousands to billions of instructions per second. A practical FW simulator should strive to be within these orders of magnitude; e.g. the P-unit FW simulator used for the Broxton (BXT) SoC family is capable of up to 3MIPS. Achieving this level of simulator performance allows for tests of sufficient length to exercise complex FW interactions to complete within seconds of real time.

6.2.1.3 Limitations

Obviously, the primary limitation with this approach is the lack of true hardware modeling. Testing of FW interactions with hardware are thus only as good as the hardware modeling in the simulator and/or test environment. Fortunately, developers write FW at a level of abstraction that does not require low-level modeling of the hardware. Transaction-level modeling suffices for most cases.

6.2.2 Tests

As the scope of FW functionality tends to grow over time, and the size of the FiV team less so, it is critical that quality tests be reusable from product stepping to stepping, derivative to derivative, and generation to generation. This allows the FiV team to focus on validating new features with a minimum of time spent regressing inherited features.

6.2.2.1 Directed Tests

The FiV team uses directed self-checking tests for initial exercise of new FW features, basing them on the implementation specification for each feature. In addition to the usual High-level and Micro-architectural Specification (HAS and MAS, sometimes called a C-spec and A-spec), the FW Validator will frequently encounter a Firmware Architectural Specification (FAS). The advantages of using directed self-checking tests are that the tests are more likely to be re-usable for future generations and are less likely to break due to interaction with bugs in other features due to their focused nature.

While FiV is a fundamentally white-box approach, tests written from a black-box perspective are more re-usable. Tests that rely on low-level knowledge of the FW structure (variables, functions, etc.) are the ones that will have to be re-written as that structure undergoes change. Stimulus should focus on using higher-level interfaces and concepts that transcend the current code implementation instead.

In order to serve as proper regression tests, it is also critical that directed exercise tests avoid randomness as much as possible. This includes randomness in terms of configuration as well as the test stimulus itself.

Tests should also attempt to verify both that stimulus occurred as expected as well as that the FW generated the correct response. Failure to do both can lead to “false positive” tests in which the test passes because it failed to generate the stimulus necessary to detect the failure.

6.2.2.2 Automated Testing

Whenever possible, the FiV team relies on automated generation of test collateral. Many FW interfaces have machine-consumable specifications such as an XML description of functions and parameters. Some FW code structures may be regular enough that the source code itself is parsable to produce a set of test cases. In either case, it is advantageous to write tools to auto-generate these test cases when possible rather than to rely on hand-maintained tests.

6.2.2.3 Random Test Generators

FiV employs random test generators (RTGs) primarily for gaining code coverage. The primary RTG used by the DDG FiV team is called kiwi. Kiwi reuses existing TE library functionality in an attempt to cross-product different features and stimulus.

6.2.2.4 OS Traces

Some FW interactions are visible at the OS level. To provide useful real-world stimulus, the FiV team records traces of these interactions on an actual physical system and then plays these traces back in the FW test environment; e.g. BXT power-management traces were taken on a BayTrail SoC-based tablet [2].

6.2.3 Checking

The FiV team prefers a behavioral-level model of FW as a checker wherever possible. For example, the Foxton-based P-unit FW validation teams rely heavily on a tool called *coconut* to provide transaction-level modeling of pCode behavior. However, such a model is not always available, and it may be impractical to develop and maintain one. As such, many FW checks are completely ad hoc.

FW Validators write checks in many ways. For example, some checking is best handled in the TE, while others are most practically implemented in the simulator. Runtime assertions are even coded directly into the FW source code (although they are executed and checked in the simulator).

FW Validators write checks at as high a level of abstraction as possible to avoid coupling to the FW code. Failure to do so is a maintenance nightmare, as every change to FW requires a corresponding change to checker code.

6.2.4 Coverage

Of the many forms of code coverage available, FiV focuses on statement (or line) and condition (or branch) coverage. Statement/line coverage measures whether each individual source code statement or line of assembly has been executed. Condition/branch coverage indicates what portion of all possible outcomes of a conditional code construct or branch have been satisfied. As condition/branch coverage typically subsumes statement/line coverage (i.e. one typically finds uncovered statements/lines in the shadow of an uncovered condition/branch), condition/branch metrics are the more important of the two and receive the most focus. Various tools exist to generate these coverage conditions automatically by parsing the FW source code.

FiV employs code coverage as both a substitute for hand-maintained testplans as well as a quality check for stimulus. FiV prioritizes coverage conditions to maximize ROI.

6.2.4.1 In Lieu of Testplans

The highest ROI test conditions for FiV are typically those described by code coverage. Since these conditions can be auto-generated from the FW source, there is no need to hand-maintain a testplan of such conditions, and thus the FiV does not do so.

6.2.4.2 Ensuring Stimulus Quality

In the RTL validation realm, measuring coverage requires enumerating coverage conditions and coding the coverage constructs by hand. Due to this, RTL validators typically measure and drive coverage late in the validation cycle, often only to help direct their random testing efforts. In contrast, firmware coverage is very low cost and measuring it can begin as soon as firmware and stimulus exist. The FiV team takes advantage of this to help drive test development, including directed test conditions. Coverage measurement provides the Validator with immediate feedback on whether or not his or her tests are hitting the expected conditions.

6.2.4.3 Coverage Prioritization

It is useful to prioritize coverage conditions (e.g. High, Medium, Low) to simplify coverage analysis and reporting. It is rarely practical to achieve 100% full code coverage, but 100% of high priority conditions may have significantly greater ROI.

6.3 Interaction with Other Teams

The FiV team must maintain interactions with various other teams at Intel.

6.3.1 Architecture and FW Development

The FiV team should interact closely with the Architecture and FW development teams, especially during the early phases of a project where most features are defined and initially implemented. The FiV team's insight into the complexity of features and knowledge of FW interactions provides valuable feedback to the architects when developing new features. The feedback allows for creation of simpler features that are easier to validate now and in the future. At times, testing will reveal deficits in the initial architecture, and the FiV team will relate those issues and ensure that architects and designers redesign the features correctly.

6.3.2 SoC Integration Validation

The FiV team must work closely with the SoC Integration Validation team ([Integration Validation](#)) as it is the direct consumer of FiV work. In addition, FiV must ensure that Integration Validation picks up validation of those FW features that are difficult to implement in the FiV test environment, e.g. low-level FW \leftrightarrow HW interactions.

6.3.3 Post-Si Validation

As experts in FW debug, FW Validators may often find themselves called upon to help debug Post-Si issues. This help may be as simple as giving opinions on the possible cause of problems, but may also include looking at traces generated from a platform, writing debug FW patches to help expose critical state, or even participating in first boot activities to help bring up the platform.

6.3.4 Other FiV Teams across Intel

As with any Validation discipline, it is important to share methodologies between teams across projects at Intel. The FiV team must strive to promulgate its various improvements not only to the next derivative but also to other non-derived projects.

6.4 Challenges

One of the biggest challenges faced by FiV relates directly to the mutable nature of FW: because it can be changed easily, it often will be. Much of the FW collateral is modifiable until a few weeks before tape-in, and it is possible to add some features entirely in a patch. Knowledge of this capability results in many late FW-only feature requests, and the FiV team must be able to respond accordingly. It is crucial that FiV leadership be able to evaluate the complexity of these late requests and push back against them if necessary, even if “it’s only firmware”.

7 Summary

Firmware Validation is critical to the success of our silicon projects. The quality of FW has a direct correlation on how quickly Post-Silicon debug can occur, translating directly to reducing TTM. Firmware Validation plays a crucial role in the success of our CPU projects by taking part in defining the right features, ensuring the correct operation of those features, and providing support to the Post-Silicon debugger.

8 Future Work

This section intentionally left blank.

9 References

1. *Improving Validation of Foxton-based Firmware*. Colin Wood. DTTC 2013.
2. *Running Windows OS Power-Management Traces in Pre-Silicon using FORE*. Colin Wood and Rick Turner. DTTC 2015.

The Art of Pre-Si Val: Chapter 33

Integration Validation (In Prog)

By: [Matthew C Chidester](#)

1 Abstract

As the number of transistors available on a single die increases, the undeniable trend is towards incorporating more and more platform components on-die. It is no longer possible for a single team to design, implement, and validate all of these individual components. Some components may be owned by the project team, while others may be taken as-is from a previous project, or designed by different group within Intel, or even purchased from an external vendor. Regardless of the source of the components, it falls upon the project team to validate that the particular configuration and combination of components will function correctly together. This activity is known as Integration Validation.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	7/1/2016	Initial bullet-point version	Matt Chidester	Michael Bair

3 Contents

1 Abstract.....	681
2 Chapter Revision History	681
3 Contents.....	682
4 Purpose.....	683
4.1 Why do we need this chapter?.....	683
4.2 What does this chapter cover?	683
4.3 What does this chapter not cover?	683
5 Background Concepts.....	683
5.1 Glossary of Integration Validation Terms.....	683
5.2 What does it mean to do an “Integration”?.....	683
6 Integration Validation.....	684
6.1 Integration vs. IP validation.....	684
6.2 Types of Integration Validation	684
6.3 Frequency of Integration.....	684
6.4 Collateral Reuse	685
6.5 Sources of bugs.....	685
6.6 Integration tests	685
6.7 Integration coverage	686
6.8 Interactions with other teams	687
6.8.1 IP team	687
6.8.2 RTL integration	687
6.8.3 Other SoC disciplines	687
6.8.4 Post-Si	688
7 Summary.....	688
8 Future Work.....	688
9 References.....	689

4 Purpose

4.1 Why do we need this chapter?

- Almost all designs are integrations.
- IP validation is imperfect.
- Prevent (blocking) bugs from reaching Si.

4.2 What does this chapter cover?

- A validation effort focused on a single IP and all of the interactions it can have with the rest of the system. E.g. “Core integration validation” or “USB integration validation”

4.3 What does this chapter not cover?

- Interactions involving multiple IPs acting independently (e.g. core and USB traffic simultaneously to memory) → Covered in [Fullchip Validation](#) chapter.
- Firmware integration → Covered in [Embedded FW Validation](#) and [Fullchip Validation](#) chapters.
- Not specifically covering subsystem validation (or IPs within IPs)

5 Background Concepts

5.1 Glossary of Integration Validation Terms

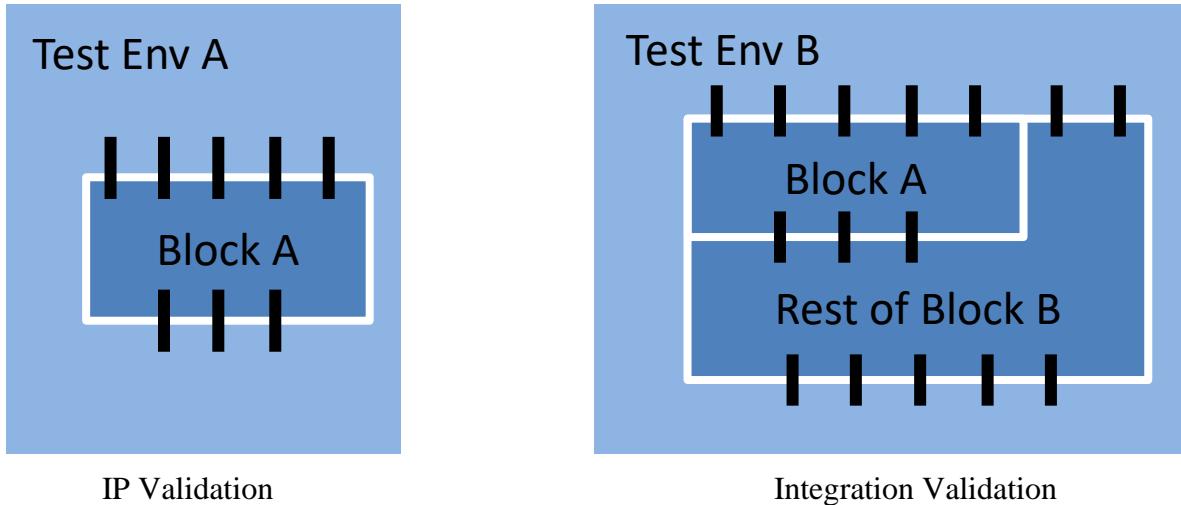
- IP
- Subsystems
- SoC
- HIP
- SIP

5.2 What does it mean to do an “Integration”?

- Taking an IP block from a different team and attaching it to a chassis/fabric.
- Also including additional IPs which interact directly with the target IP.
- Includes both RTL and validation collaterals
- Will need to be repeated multiple times during the course of a project

6 Integration Validation

6.1 Integration vs. IP validation



- IP validation: Depth validation, all corner cases, typically random testing, small/fast simulation models or inexpensive FPGA platforms, validate to full RTL HAS
- Integration validation: Focus on interface signals/flows, cover representative subsets, typically more directed testing, larger/slower simulation models or expensive emulator platforms, validate to “Integration HAS”
- White vs. Black: IP validation fully whitebox; integration validation is whitebox around the edges, blacker box towards the middle.

6.2 Types of Integration Validation

- Subsystem Integration vs. Re-Integration
 - Subsystem Integration: IP team validates IP + phy + PLL, and delivers to SoC team as a single block → Desired work mode
 - Re-Integration: Instead of single block, IP + phy + PLL are delivered individually to SoC team → Need to re-validate all the interfaces between 3 components. Focus can be on connectivity and less on interactions between the internal blocks
- Tradeoffs on DUTs
 - How many components to model?
 - Lots of small DUTs vs. fewer big DUTs

6.3 Frequency of Integration

- Multiple integrations needed due to IP development, bug fixes.

- Conventional wisdom: “Integrate early and often”
- Reality: Integration can range from trivial to huge effort
- Rule of thumb:
 - IPs with relatively little change or high integration cost → fewer integrations
 - IPs with large amount of change or low integration cost → more integrations
- Invest in making IPs with large amount of changes easy to integrate (analyze source of issues and push fixes back to IP provider, automate with scripts, etc).

6.4 Collateral Reuse

- Tests
- Configuration
- Trackers
- Checkers
- Coverage

6.5 Sources of bugs

To understand the scope of integration validation, it is useful to consider what types of bugs are typically expected to be found by this activity:

Author's note: possibly break following list into subsections with more descriptions/examples

- Connectivity Bugs: The process of integrating the IP into the design is prone to errors. Signals may be left dangling, connected to the wrong place, or mistakenly tied off high or low. These bugs are introduced at the integration level, and therefore cannot be found anywhere else. If IP validation were completely infallible, there would be no other type of bug to find! However...
- IP Validation Escapes: The IP team never has a 100% accurate model of the external system. Often this is intentionally done to simplify the IP testbench, but there also can be mistakes leading to incorrect modelling.
- Architectural Misinterpretation: Sometimes an IP may have an incomplete specification for the types of transactions it is required to handle from another IP. It is also not uncommon to find cases where two different teams, reading the same specification, can come to opposite conclusions on what a particular detail means. In either case, each IP team believes they have validated the correct behavior, and the mismatch can only be found when the two IPs are actually hooked up to each other.
- Concurrent Interactions: (Is this really the same as “Architectural Misinterpretation”?)
Link to Feature Architecture Validation or FC Validation

6.6 Integration tests

- Take basic testcases from IP level and re-run at integration level
 - Focus on flows that involve IP-to-IP interaction
 - Ex: PCI Integration validation:

- PCIE reads to memory controller IP
 - PCIE writes to memory controller IP
 - PCIE controller is put into low power state by Punit IP
 - PCIE internal registers are accessed by TAP
- SAOLA sequences
 - Intel methodology for reusing IP content @ SoC
 - Typically provided directly from the IP pre-Si validation team and therefore is most up-to-date with current capabilities of the IP RTL.
 - Many pitfalls to beware of
 - Assumptions on the configuration
 - Sequences containing flows that will be done by real IPs @ SoC
 - Translating low-level sequence designed to work directly at the IP interface pins to a more appropriate location elsewhere in the SoC fabric
 - Not reusable at emulation or in silicon
- Software sequences
 - Setup IP testcases using IA32 code that runs on a core
 - Ex: Maestro code written in C
 - Typically self-checking
 - Needed to exercise IP in hardware emulation or silicon
 - Typically provided from IP post-Si validation team (if one exists!) and can often lag RTL availability, or even assume functionality greater than current RTL drop.

6.7 Integration coverage

- Often primary source is test coverage
 - IP and integration teams work together to define most important testcases to cover all integration flows
 - When all testcases are passing, flows are deemed complete
 - Leads to escapes when an integration flow is missed or test did not do what it was expected to do
- Toggle coverage
 - Low-effort way to ensure that all interface pins of an IP have seen some activity.
 - Best way to catch accidental dangle/tie-offs
 - Can show when test content is missing or misbehaving
 - Examples: TAP pins never toggled, bidirectional pin was never put into “output” mode, 2nd output port never used
- Interface coverage
 - Medium-effort way to check protocol-level interaction between IPs
 - Example: IDI or IOSF opcode coverage
- Event coverage
 - High-effort way to see if important internal events have been triggered
 - Only feasible if IP team can provide cover points to integration team → even then may result in maintenance costs
 - Example: Credit counter depleted, low-power state reached, snoop hit valid data in IP

- Portability:
 - Standard tools like QuickCov or VISA events should be agreed upon and used. See [Coverage](#) chapter

6.8 Interactions with other teams

- Number one most important thing for successful integration validation is communication!
- Building good relationships is also key → can lead to “keeping you in the loop”, getting more honest feedback, cooperatively solving difficult problems

6.8.1 IP team

- Have regular meetings with IP validation representative(s) to discuss progress/issues
- Review IP testplan(s), pass rates, coverage rates, etc. to judge readiness of IP for integration validation on a feature-by-feature basis
- Have IP team review integration testplan(s), pass rates, coverage rates, etc. to increase chance that important details aren't missed.
- Ideally IP team should help with integrations, debug, or even test running in the SoC environment
 - For some IPs this is impossible (external vendor or internal IP w/ defunct team) → know the support model in such cases!
- Know how to file bugs and flag issues.

6.8.2 RTL integration

- SoC team should provide RTL owner(s) to work with IP RTL representative(s) to integrate and compile IP in SoC environment
 - RTL owner(s) typically become expert at level of interface pins and then proceed into backend domain (clocking, timing, circuit marginalities, layout/pinrings, etc).
- Integration validator will be expected to help debug failures during these integrations
 - Validation owner(s) typically become expert at level of interface pins and then proceed into functional domain (protocols, handshakes, data flows, etc).
- Need regular meetings to discuss integration progress/issues between SoC and IP teams. Can be combined with validation meeting or separate.

6.8.3 Other SoC disciplines

- Will reuse flows from integration validation domains to combine into more complex test scenarios
- Need feedback from integration validation on:
 - When basic IP features are healthy enough to add to fullchip scenarios
 - How to configure the IP
- Examples:

- DFX validation team may want integration validation to demonstrate basic TAP/VISA accesses to IP before they enable the IP as a target for more exhaustive testcases.
- Power Management validation may want integration validation to run basic test to put IP in and out of its sleep states before they enable global package-level sleep states
- FC validation team will want to see IP traffic flows working individually by integration validation team before mixing it with traffic from other IPs
- Test content can flow in both directions:
 - DFX validator might provide template for basic TAP access to be used by integration validators
 - Integration validator might provide sequence to put IP into sleep state to be used by PM validator when writing package-level scenario

6.8.4 Post-Si

- Testcases (especially software-based sequences) for integration validation may come from IP post-Si validation team.
- Testplans/testcases/coverage may be shared with post-Si SoC validation team
- Integration validators often develop valuable expertise for debugging silicon failures in their IP
- Post-Si team often seeks IP expertise prior to silicon arrival in the form of training, tour-of-duty, JVT efforts, etc.
- See the [Interaction with Post-Silicon Validation](#) chapter.

7 Summary

The summary.

8 Future Work

Address following topics somewhere above (new or existing sections?):

- Configuration challenge(s) in integration validation
 - Fuses, BARs, memory map, enable bits, defeature modes, etc.
 - Random vs. directed
 - Simulation vs. emulation (vs. silicon)
- Simulation vs. Emulation
 - When to use and for what
 - Pitfalls (forces, phys, checking, debug visibility)
- Standards
 - Different types:
 - Interface: IDI, IOSF, CMI, etc
 - Chassis blocks: PMA, fuse pullers, SBR, etc.

- Methodology: RDL/RAL, SAOLA, UPF
- Impact on integration validation

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 34

Architecture Validation

By: [Wei Kang Teh](#)

1 Abstract

Architecture Validation (AV) verifies that the CPU meets the external published specifications of the Intel Architecture, including instruction and register set, addressing modes, etc. In this chapter, we introduce how we do AV for our CPU projects today, including some historical developments to set the appropriate context and to explore the role AV plays in today's validation of a CPU design.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	2/22/2012	Initial Draft	Wei Kang Teh	Michael Bair / CCDO Val Staff

3 Contents

1 Abstract.....	691
2 Chapter Revision History	691
3 Contents.....	692
4 Purpose.....	693
4.1 Why do we need this chapter?.....	693
4.2 What does this chapter cover?	693
4.3 What does this chapter not cover?	693
5 Background Concepts.....	693
6 Architecture Validation.....	693
6.1 Overview of Intel Architecture Validation (IAV)	693
6.2 IAV Test Plans	694
6.3 IAV Test Writing.....	695
6.4 Directed vs. Random Testing.....	696
6.5 Coverage-based IA Validation	697
6.6 IAV Checking	698
6.6.1 Archsim, RTL Simulation Interface and Chekhov	698
6.6.2 Special Checkers and Self-Check	698
6.6.3 Other Types of Checking	698
6.7 IAV Regressions	699
6.7.1 Legacy Periodic Multi-Level Regressions	699
6.7.2 Coverage Driven Regressions	699
6.8 Other Developments.....	699
6.8.1 Hardware Emulation for IAV	699
7 Summary.....	700
8 Future Work.....	700
9 References.....	700

4 Purpose

4.1 Why do we need this chapter?

A well-defined and validated architecture that meets the published specifications of any CPU design enables software programmers and end-users to write useful and productive programs that function as expected when run on the target CPU. The goal of Architecture Validation (AV) is to ensure that when the various design components like hardware and firmware are put together, they collectively serve to enable this purpose. AV has been and continues to be an integral validation discipline for each design project. This chapter serves to introduce how we do AV today and to describe key AV components and methodologies.

4.2 What does this chapter cover?

This chapter takes a largely CPU-centric view of AV. It covers an overview of AV and explains key concepts and methodologies used today. It also provides some historical context to how AV has evolved over the years to enable readers to better understand the state of current tools and methodologies.

4.3 What does this chapter not cover?

It is not the intent of this chapter to describe in details the tools or specific techniques deployed today in AV. However, some examples are used to better illustrate the concepts and principles.

5 Background Concepts

6 Architecture Validation

6.1 Overview of Intel Architecture Validation (IAV)

The goal of Intel Architecture Validation (IAV) is to ensure that we have an Intel Architecture (IA) compliant machine as described in Intel's software developer manuals (SDMs). This encompasses both aspects of architecture and integration validation. IAV validates the collection of features of a microprocessor as are seen by the operating system (OS) and software programmers. This includes the IA instruction set, registers, addressing modes and methods. IAV divides the architecture into sub-features or areas called taxonomies. Examples of typical IAV taxonomies are Paging, Segmentation, Interrupt & Exception and Integer.

IAV is done typically at the supercluster and/or full chip level. For every architectural feature, there is a requirement that at least some portion of the validation is done at the level that encapsulates the entire feature. This level of validation addresses the limitations of cluster or lower-level validation by focusing on cluster-to-cluster and firmware-hardware interactions. Mismatches in these interactions are very possible in a feature's implementation and may still exist despite validation of each cluster to its micro-architecture specifications. In other words, IAV aims

to ensure that when the various lower-level and often micro-architectural components of a feature are brought together, they collectively function as expected to deliver the IA specifications. From the view of software testing, this is often referred to as black-box testing where IAV focuses on the overall functionality of an architectural feature rather than its inner structures or workings. Even when implementing changes or bug fixes that should not alter the architecture, IAV regressions are often broadly run as part of no-harm validation to ensure nothing breaks. Examples of these include micro-architecture changes and firmware patch validation.

There are other considerations why this level of architecture and integration validation is required. Validation at the lower level typically translates to modeling of a test environment (TE) and use of emulators. Errors in the TE modeling and emulation may mask bugs that otherwise would not have been found without integration validation. Also with this TE modeling and emulation, it is possible and often true, that we do not exercise the actual timings of the cross-cluster logic interactions thereby leaving the possibility of bug escapes. Lastly, a good amount of validation overlap at the higher-level DUTs is required for higher validation confidence.

Besides validating architectural features to their specifications, IAV works closely with architecture, design and other validation teams to ensure that new features are well architected and defined. This is especially important as any new architectural feature that is released with our CPUs will immediately become an IA legacy to be supported in future CPUs. This is true even for CPU behaviors that are either undocumented or architected as undefined but with a consistent predictable behavior as software may come to rely on them.

Within the validation community, IAV is closely related to the Feature Architecture Validation (FAV) and Microcode Validation (uCodeV) teams. IAV shares many of the same tools and methodologies as FAV. See FAV chapter for more details. Since significant parts of the architecture are implemented in microcode, IAV works closely with uCodeV to validate each architectural feature. See chapters [Feature Architecture Validation](#) and [Microcode Validation](#) for additional information.

6.2 IAV Test Plans

Along the lines of IAV partitioning of architectural features, IAV test plans describe the overall validation strategy, assumptions, tools and actual test conditions for each taxonomy. An IAV validator typically creates the validation plan from Processor Architecture and High-Level Architecture Specifications (PAS/HAS), Intel SDMs, HSD feature issues/ECOs and from exchanges with the architecture and design teams. Once the initial validation test plan draft has been created, it is then officially reviewed with representatives and stakeholders from architecture, design and validation.

A more recent addition to IAV test plans has been the creation of a section containing information of interest to Post-Silicon validation. Areas of concern from Pre-Si IAV are listed in this section along with assumptions and in some cases, division of responsibilities between Pre- and Post-Silicon validation. In addition to having this section, the corresponding Post-Silicon validator is invited to the IAV test plan review with architects, designers and other Pre-Silicon validation teams to provide inputs and feedback.

Today, there is an on-going initiative to move one step further to create a holistic combined Pre- and Post-Si test plan. Currently trialed in one taxonomy, this initiative brings together both validators from Pre-Si IAV and Post-Si System Validation (SV) to jointly develop the test plan from the beginning. Firstly, all test plan conditions for a feature are defined together. These are subsequently mapped to each team's respective validation coverage in Pre- and Post-Si. Several factors are taken into account when determining the validation mapping. Traditional considerations like the goal of Pre-Si IAV to ensure sufficient feature health to enable Post-Si SV execution while relying on Post-Si for added randomness and cross-product validation still applies. On top of this, both teams aim to reduce validation redundancy while still taking enough precaution to ensure we have sufficient validation overlap in place. Test generators used by both teams are also taken into account since they have inherent strengths and limitations. As this initiative to jointly develop a holistic test plan is underway, we expect further changes and refinement to the approach as both teams learn more together.

Another fairly recent change to how IAV test plans look is the inclusion of security validation as another consideration. Early on in the project architectural features with security implications and concerns are identified by Pre-Si IAV, Pre-Si security validation experts and representatives from the Security Center of Excellence (SeCOE). Example features include System Management Mode (SMM) and random number generation. These features are then subject to additional test plan reviews with security stakeholders to ensure the security aspects are well covered.

As with validation test plans in general, IAV strives to maintain a live and up-to-date document with each successive CPU project. IAV validators are expected to revise the test plans to include learning from the current project. This includes validation trends like bugs found and escapes to Post-Si. Subsequent changes that affect each validation area like new ECOs are also documented.

6.3 IAV Test Writing

An AV test is written to primarily validate the architectural aspects and behavior of an IA-compliant CPU. However, as part of integration validation and typical of validating at higher DUT levels, some level of micro-architectural testing is also attained through these AV tests. In some cases, AV tests are explicitly written to target specific micro-architectural conditions at the supercluster and/or full chip. AV tests are usually written in assembly language. AV test writing has evolved over time. In the past, the first AV tests were almost entirely written in assembly code (ASM). This was followed by the development of Max and Testgen tests that use Perl with embedded assembly. Directed AV tests are today mostly written in Testgen though variants of legacy ASM and Max tests still exist in the IAV test suite. These tests have been retained as legacy because architectural features are designed into our CPUs and retained for backward IA compatibility for each generation.

Directed AV Testgen tests are written to hit specific conditions as described in the validation test plan. Using Testgen with its set of setup and control test libraries, an AV test contains the entire system for testing on its own. There is no BIOS or OS kernel within an AV test. Instead, these test libraries are used to set up the test with everything it needs to run standalone from reset. An AV test writer usually just needs to write specific portions of assembly code that are needed per the test condition and let the test libraries manage the rest of setup and non-essential randomization. This greatly simplifies AV test writing by keeping the actual unique test code concise and hides

non-essential test writing details from the writer. It is possible for an AV test writer to write specific tests without an immediate need to know every aspect of the architecture. Test maintenance is also made easier and more scalable as general changes can be applied to the test libraries instead of individual tests. It is part of an IAV validator's responsibility to develop Testgen libraries appropriate to each architectural feature. This is typically done during the Front-End Development (FED) phase before the larger test writing effort takes off.

A directed AV test is written to target one or more specific conditions but typically allows everything non-essential to the condition(s) to be randomized. For example, to validate a general-purpose arithmetic instruction, we may randomize the processor execution and paging modes. Directed AV tests are usually short and contain only several hundred instructions at most.

Besides directed AV test writing, test generators that generate random assembly code like Stargate and Cafe are in use today. Stargate, which used to be known as Agate until its merge with the System Test Generator (STG) component, takes user test templates written in the Test Specification Language (TSL) format and expands them using internal system and architecture description models to generate architectural tests. This higher-level abstraction has several benefits. Firstly, test base maintenance is easier. In the past, an IAV validator is required to include all the necessary conditions into his Max/Testgen test. In some cases, large numbers of tests were created with just minimal changes between them for added coverage. This created a huge maintenance problem when test fixes or changes need to be applied across every variant of these tests. Secondly, the use of Stargate reduces the fixed simulation cost of running a large number of almost similar tests. This leads to more efficient and effective use of IAV validation compute cycles. More on directed vs. random testing in Section 6.4 below.

AV tests can usually be run on various platforms such as an architecture simulator, against RTL and on actual silicon.

In the past, writing silicon-friendly AV tests was also an important consideration so that we may reuse these tests to generate production traces for functional testing. Considerations like writing efficient self-checking tests that maximize coverage vs. test run time benefit both Pre-Si IAV and are good candidates for production test traces. However, in some cases, this does limit test writing flexibility and/or incur additional overhead for pre-silicon AV test writing. For example, event injectors and signal pounders that are used to simplify tests cannot be used with production test traces; using silicon-friendly replacements for these injectors often increases test complexity and consumes simulation cycles.

6.4 Directed vs. Random Testing

Historically, directed testing has been the main test methodology for AV validation. For architecture validation at the higher-level DUTs, directed testing has many advantages. Firstly directed testing capabilities allow us to create specific integration scenarios of interest to AV, often much earlier and easier than we would otherwise have with random testing. This is particularly useful and important for early validation and exercise to prove basic feature functionality. However, directed testing also has its limitations. With this approach, we are confined only to the conditions that are listed in the validation test plan. In other words, IAV is limited to the available specifications of a feature and to how much a validator knows about it.

This is where random testing comes into the picture. While it may take longer to develop architectural coverage monitors and checkers to extract useful feedback metrics and subsequently direct validation to detect bugs, random testing brings its own set of advantages. Increased randomness allows IAV to explore validation space beyond what could possibly be thought of by a validator. To a certain extent depending on factors such as test generator effectiveness, random testing allows close to an infinite validation space.

The general rule that IAV uses as a guideline today is to have a mixture of directed and random testing for each architectural feature. What this implies is that IAV is gradually evolving from being purely directed to include some portions of random testing. This translates to requiring a random AV test generator, greater use of architecture coverage monitors and rewriting of IAV test plans to better suit random testing. IAV is continually pushing its test generation tools to fit the mold of ‘good randomization while easily writing directed content’.

For an in-depth comparison of directed versus random testing, please see [Stimulus section: Directed Stimulus vs. Random Stimulus](#).

6.5 Coverage-based IA Validation

With IAV evolving towards random testing, architectural coverage monitors and checkers are now an important component of our validation. This requires several changes to the way we do traditional directed IA validation. Firstly, overall IAV strategy and test plans are adapted to the new methodology. Detailed directed test conditions are abstracted to higher-level “breadth” coverage space. Random testing using an IA test generator is then applied selectively to target key areas of the architecture. Architectural coverage monitors collect coverage metrics and provide feedback data of what has been tested and what has not to assess the validation quality of the design. Based on these results, the testing is then redirected as needed to ensure that key uncovered parts of the architecture are proven to be working as expected.

Architectural coverage development and the subsequent analyses and drive usually takes place during the execution (EXEC) phase of a main CPU project. For follow-on steppings or projects with minimal changes, IAV may decide against re-collecting architectural coverage. Coverage is today measured using a multi-week rolling methodology with target goals for each taxonomy. Exit reviews are also held to review coverage holes to ensure all the necessary closure is in place before tape-in.

Even before the move towards random testing, architecture coverage has been a component of IAV. Coverage is typically collected for new architectural features or a selected target area of concern to provide a one-time feedback on the effectiveness of our directed test suite. While issues such as test bugs and illegal test conditions have been uncovered by this approach, it is generally of less ROI compared to the benefits of random coverage-based architecture validation. There is also another drawback with this approach. The quality of architectural coverage monitors often deteriorate over time due to lack of use and subsequent maintenance.

6.6 IAV Checking

6.6.1 Archsim, RTL Simulation Interface and Chekhov

Today, IAV uses Architecture Simulator (Archsim), RTL Simulation Interface (RTLSI) and Chekhov as our primary validation checking infrastructure. Archsim models the functionality of our target CPU at the architecture level. RTLSI provides test environment support to keep RTL in-sync with Archsim. Lastly, Chekhov is used to compare and check between Archsim and RTL statea during runtime for correctness. As a result, a key goal of IAV can be achieved: making sure that the CPU works according to the Intel Architecture specifications.

Architectural checking using Archsim is done at the macro-instruction boundary and checks the range of architectural visible state. This includes both explicit and implicit changes to the state as defined in Intel Architecture specifications. For most parts of IAV, Archsim is sufficient as the only checker used. The exception to this is when Archsim has limitations to model certain aspects or features of a target CPU implementation. These are often micro-architectural, for example, Translation Look-aside Buffers (TLBs), etc.

On a side note, Archsim also has several other uses beyond checking. Since Archsim models the functionality at the higher architectural-level, a key benefit is that it runs much faster standalone than RTL simulation. Archsim is often used by validators to quickly verify that their new tests are working as expected and is commonly used for debug where a validator can rerun the assembly code to reproduce or analyze failures. Archsim is also used as a key simulator in our random test generators to create IA traces that can then be run on RTL and silicon.

6.6.2 Special Checkers and Self-Check

To address the limitations and shortfalls of Archsim for checking, IAV uses specialized checkers and manual self-checking. Lagrange Technology/Cache as RAM (LT/CRAM) protocol and TLB checkers are such examples. These checkers are written in Specman e and generally involves greater understanding of lower-level micro-architectural implementation details. They extract current machine states and check for correctness given those states.

For directed tests in which Archsim could not reliably be expected to provide checking, manual self-check is used whereby a validator embeds his understanding of the expected architectural behavior into his tests. Archsim is then either disabled from running along with the test or as is often the case, allowed to run but with checking disabled. In general, regardless of whether Archsim support for an aspect of the architecture is available or not, the rule is for a validator to write self-checking tests whenever possible. The Testgen libraries and templates already provide basic self-check capabilities by default today.

6.6.3 Other Types of Checking

In some areas we use software validation to cross validate an architectural feature with external specifications and usage models. For example, in cryptography, we validated Advanced Encryption Standard (AES) against OpenSSL open source software that contains basic cryptographic functions, including the FIP197 standard implemented by AES.

Early on in the development of architectural features, visual inspection may be used to check for correctness when relevant checkers are not ready or when coverage infrastructure is not yet developed.

6.7 IAV Regressions

IAV regressions are typically rolled up as a key project indicator to provide an assessment into the overall health of the CPU design.

6.7.1 Legacy Periodic Multi-Level Regressions

There are several types of regressions in IAV. Traditionally IAV runs periodic multi-level regressions that contain mostly directed IAV test content. In this model, IAV test content is divided according to their measure of architectural coverage, from the most basic conditions in Level 0 regressions to the most random and widest coverage space in Level 3 or 4. Each level of regression typically has representative content that spans the breadth of the Intel Architecture. Each level is then run according to the pre-determined frequency, usually starting with Level 0 as the most frequent followed by Level 1, Level 2 and so on. As the number of tests increases with each regression level, higher level regressions typically only run a few times for each project.

Several considerations determine where a test lives in regression. These includes the criticality level of a test to feature health and functionality indication (basic vs. corner cases, new features vs. legacy), test length and runtime, regression frequencies and available compute resources.

6.7.2 Coverage Driven Regressions

With the move towards random coverage-based validation methodology in IAV, there is a new regression that runs random test content continuously to meet architectural coverage goals. Many of the considerations described above for traditional IAV regressions also apply to this new regression.

6.8 Other Developments

6.8.1 Hardware Emulation for IAV

IAV on hardware emulation as a validation platform is a recent development. Using this platform, many of the benefits of emulation in terms of speed and maintenance can be realized by IAV. Examples include “real-time” architectural coverage that can be achieved today on hardware emulation, ability to run longer tests that increases the probability of hitting corner case conditions sooner and makes possible the validation of long architectural flows such WBINVD/INVD instructions that have begun to demonstrate increasingly prohibitive costs on simulation.

The use of hardware emulation also brings both Pre- and Post-Si validation closer to a common platform that could open up new possibilities of collaboration between both teams in validation content, tools and methodologies.

However, there are inherent challenges as well. For example, today's methodology on hardware emulation relies a lot on self and end-of-test checking vs. runtime lockstep architectural checking on simulation. Since we do not self-check every architectural state at every macro-instruction boundary, it is arguable if such an approach can fully meet the IAV goal of ensuring the CPU meets IA specifications. Along with the promising benefits that hardware emulation bring to the table, debates and discussions over various challenges like this are on-going today to determine the way forward.

7 Summary

IAV has been and continues to be an integral validation discipline in the design of modern CPUs. A well-defined and validated architecture that meets the published specifications of any CPU design enables software programmers and end-users to write useful and productive programs. IAV also plays an important part to ensure our CPUs are backward compatible, a key advantage for programmers to continue their software development on Intel Architecture. IAV tools and methodologies will have to continue to evolve to meet current and future challenges.

8 Future Work

9 References

The Art of Pre-Si Val: Chapter 35

Feature Architecture Validation

By: [Steven Saar](#)

1 Abstract

This chapter describes the Feature Architecture Validation (FAV) team's role and methodologies in validating a microprocessor. FAV validates features that generally have an architecture that is not published external to Intel and that have significant interactions between hardware clusters and between hardware and firmware. FAV has historically used mostly directed testing on simulation, though the use of emulation is currently being explored.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	2/13/2012	Initial Draft	Steven Saar	Steve Petersen / FAV team / Michael Bair / CCDO Val Staff

3 Contents

1 Abstract.....	701
2 Chapter Revision History	701
3 Contents.....	702
4 Purpose.....	704
4.1 Why do we need this chapter?.....	704
4.2 What does this chapter cover?	704
4.3 What does this chapter not cover?	704
5 Background Concepts.....	704
6 Feature Architecture Validation	704
6.1 What is FAV?.....	704
6.1.1 Overview of FAV	704
6.1.2 Typical FAV Taxonomies	704
6.1.3 FAV and Other Validation Teams	705
6.2 FAV's Role During Each Project Stage.....	706
6.2.1 Tech Readiness.....	706
6.2.2 Front End Design.....	706
6.2.3 Execution	707
6.2.4 Post-Si	707
6.2.5 Dealing with multiple projects/steppings	707
6.3 FAV testplans	707
6.3.1 What does an FAV testplan look like?	707
6.3.2 Writing an FAV Testplan.....	708
6.3.3 Testplan reviews.....	708
6.4 Random vs. Directed testing in FAV	709
6.5 Tools.....	709
6.5.1 Test Generators.....	709
6.5.2 Injectors	710
6.5.3 Checking.....	710
6.6 Test environment.....	711
6.6.1 Simulation.....	711
6.6.2 Emulation.....	711

6.7 FAV regressions	711
6.7.1 Periodic regressions	711
6.7.2 Gating regressions.....	712
7 Summary.....	712
8 Future Work.....	712
9 References.....	712

4 Purpose

4.1 Why do we need this chapter?

Feature Architecture Validation plays an important role in processor validation. Many of the features validated by FAV are critical debug features that need to be functional to enable post-si progress once silicon returns from the fab. This chapter serves to document FAV's role within the pre-silicon validation team and current FAV methodologies.

4.2 What does this chapter cover?

The chapter defines the areas typically covered by FAV, FAV's role over the course of a processor development project, and the interactions between FAV and other teams. It also describes various aspects of FAV methodology including testplans, tests, tools, regressions, and environment.

4.3 What does this chapter not cover?

This chapter does not cover the details of specific tools used on a specific processor project; it instead describes the characteristics of tools that are well suited to FAV's methodologies.

5 Background Concepts

6 Feature Architecture Validation

6.1 What is FAV?

6.1.1 Overview of FAV

Feature Architecture Validation (FAV) is a pre-silicon validation discipline that focuses on the validation of a collection of features that have both architectural and microarchitectural characteristics. These features typically are not fully architecturally defined in the IA Software Developer's Manual; the detailed architecture of each feature is usually internal to Intel and can vary processor to processor. The features often contain significant interactions between various hardware clusters and firmware. Many FAV features play a role in helping to debug processor silicon and platforms.

6.1.2 Typical FAV Taxonomies

Some typical FAV taxonomies are shown in the table below.

Taxonomy	Description
MCA	MCA (Machine Check Architecture) logs errors in various parts of the processor and signals error events when appropriate.
Patching	Patching is used to fix or work around bugs found in silicon and can also be used to help debug and root-cause post-si sightings.
Perfmon	Perfmon consists of a collection of counters that track the occurrence of various microarchitectural events. These counters can be used for debug and performance tuning.
PFAT	PFAT (Platform Firmware Armoring Technology) is a security feature that can help protect the BIOS from attacks.
Probe Mode	Probe Mode allows post-si validators to access and control internal processor state through the TAP.
Reset	Reset consists of the sequence of operations needed to initialize the processor.
Trusted Paths	Trusted Paths is a security feature that restricts access to sensitive hardware state to certain trusted microcode flows.

6.1.3 FAV and Other Validation Teams

Because many FAV features contain significant interactions with firmware (uicode, pcode, etc.), the Firmware Validation team validates many of the same features as FAV (see [Microcode Validation](#) and [Embedded FW Validation](#) for more information on Firmware Validation). The main difference here is that the firmware validation team works at a cluster level; they validate the real firmware with simulated hardware. While this enables them to validate the firmware in isolation, FAV is needed to validate the interactions between the firmware and the hardware.

Likewise, cluster uAV (micro-architecture validation) validates many of the individual components of FAV features but cannot cover interactions between hardware clusters or between hardware and firmware.

IAV (Intel Architecture Validation) is in some ways closely related to FAV. Both validate features with inter-cluster interactions at a core or fullchip level. The primary difference is that IAV's features are architectural; they are generally fully defined in the IA Software Developer's Manual. Legacy behavior generally carries forward to future processors (though enhancements may be added). In contrast, FAV areas are not publically architecturally defined and do not necessarily maintain forward and backward compatibility.

FAV features carry several challenges that make them difficult to validate. Because the feature specifications can change significantly from project-to-project it can be difficult to leverage validation collateral (testplans, tests, checkers, etc.) from previous projects for a new project. Details of FAV features also tend to change over the course of a project. Changes are often made close to tapeout or on later steppings and features can sometimes be added or removed at the last minute.

Another challenge can be the lack of good documentation. Because the specifications of FAV features are not published externally like those of architectural features, validators must rely on internal documentation. Often documentation is focused on one particular component of the feature (for example, only the firmware) leaving the validator without good global documentation. In some cases, the validator must take responsibility to drive the creation of the necessary documentation.

Because FAV features usually touch multiple clusters there are often multiple different designers involved with the feature implementation. In some cases this results in an inconsistent schedule where one piece may be implemented much sooner than another. By the time the remaining components are implemented the details of the design are not as fresh in the mind of the designer of the first piece. It sometimes falls to the validator to help coordinate the implementation of the necessary feature components. Because multiple designers are often involved assumptions made about cross-cluster interfaces can be incorrect, leading to bugs.

6.2 FAV's Role During Each Project Stage

6.2.1 Tech Readiness

Feature Architecture Validators typically become involved with a new project during the Tech Readiness (TR) phase. During TR they work with the architects and designers to provide feedback on new feature proposals and start to plan their validation strategies and methodologies for the project.

As architects and designers start filing new features in HSD, Feature Architecture Validators track features relevant to areas they will own and provide feedback to the feature owners. In many cases this feedback will focus on ways to clarify the spec and/or make validation of the feature easier. Validators will sometimes complete a formal complexity analysis process that can be used by project planners and management to properly understand and manage the overall scope of the project as they decide which features to make POR.

Feature Architecture Validators will also work to figure out how to best validate their areas on the new project. They will evaluate the effectiveness of tools and methodologies used on previous projects and investigate new tools and methodologies. HSD features will be filed and used to track these investigations.

6.2.2 Front End Design

During Front End Design (FED), Feature Architecture Validators will begin exercising portions of new or modified features they own as the changes are implemented. Validators will write exercise testplans consisting of the sets of basic conditions they plan to validate during FED. They will often be a part of virtual teams, beginning their exercise in side clones provided by designers. In some cases they will fix bugs or make RTL changes themselves in order to make forward progress. As tests are written and features are turned in to released models, tests will be added into regressions that will be run periodically to make sure other changes do not break already-working functionality. Execution testplans are also typically completed during FED.

6.2.3 Execution

Once the execution testplan is completed and reviewed with stakeholders, Feature Architecture Validators' work focuses on writing execution tests, driving these tests to pass by debugging failures, finding and filing RTL and TE bugs, and fixing test problems. Checkers, injectors, and other tools are developed as necessary. In the weeks leading up to tapein the owner of each area will make a weekly Go/No-Go call based on the completion of the planned execution work for the area and the regression pass rate.

6.2.4 Post-Si

After tapein of the first stepping the FAV team quickly shifts focus to the next stepping. In some cases some portion of the testplan is deferred from one stepping to the next. In other cases feature changes or new features are planned for a later stepping. Because several FAV features are used for post-si debug, validators will often be “on call” to help the post-si team work through any problems with these features once silicon returns.

6.2.5 Dealing with multiple projects/steppings

Depending on the roadmap for a particular project there is often a time when multiple steppings (often for more than one project) are in flight at the same time. During this time regressions are run on each stepping and failures are debugged. Some tests may need to be updated based on differences between different steppings and some tests may apply to one stepping but not another. Tests should be written to be as flexible as possible to be able to deal with a wide range of project configurations. For example, a derivative project might have more or fewer cores than the original. Tests should be written so that they can run (and cover the full intent of the testcase) regardless of the number of cores present in a project.

6.3 FAV testplans

6.3.1 What does an FAV testplan look like?

An FAV testplan consists of a description of the feature, a description of the validation approach, and a list of test conditions that the validator plans to cover. The testcases are often broken down into sections to better organize the testplan. Each testcase includes a description of the condition being targeted, a list of any special injectors or checkers needed to cover the testcase, and the number of tests expected to be used to cover the testcase. As tests are written it is important that there is some link (test name, path, etc.) connecting the testplan testcase to the test(s) written for that testcase, so that someone looking through the testplan later on can easily map each testplan condition to the actual test(s) covering that condition.

6.3.2 Writing an FAV Testplan

When a validator is starting to write an FAV testplan their first step should be to make sure they have a solid understanding of the feature they will be validating. They should read all available documentation (HAS's/MAS's, HSD features, code comments, etc.) and discuss any unclear areas with architects, designers, or other validators knowledgeable about the feature. If the feature is not new to the project it is a good idea to start with a testplan from a previous project. This can then be extended or modified as necessary based on the additions and changes to the feature on the current project.

To come up with test conditions based on changes or new features the validator should first think through the functionality of each component of the feature. For example, does an error in one specific unit cause the expected MCA error code in the correct bank? How about an error in a different unit? In many cases individual components of a feature can be put together into a longer flow. These “full flow” testcases verify the interactions between different feature components.

Many FAV features contain certain checks that must pass for the feature to operate. For example, a patch will not load if various patch header fields do not contain the expected values. In most cases the validator would include testcases that cover each of these checks to make sure they are properly implemented.

FAV features often have interactions with other FAV features and with features owned by IAV, PMV (Power Management Validation), and other groups. Though in some cases these testcases will be owned by other pre-si validation teams, the interactions should still be included in the FAV testplan with a note indicating which team owns the testcase.

Several FAV features have important security-related considerations. Patching is a good example as there are several security mechanisms in place to ensure a malicious user cannot load their own patch. There are also potential security concerns that might not be immediately obvious. Validators should consider taking one of the available security validation classes to learn more about security interactions. It is also useful to meet with security experts in SeCOE (Security Center of Excellence) while writing the testplan.

6.3.3 Testplan reviews

An important part of developing a testplan is getting feedback from stakeholders in a testplan review. If a validator is relatively new to FAV it is a good idea for them to review an early copy (possibly a single section) of their testplan with a teammate to make sure they are on the right track. Once an entire testplan draft has been completed it should be reviewed with all stakeholders. First, the validator should identify who should review the testplan. For FAV features this typically includes architects, designers, pre-silicon validators in other teams, post-silicon validators, validators working on other projects, and security experts.

The validator should schedule a testplan review meeting at a time that works for as many stakeholders as possible and send out their testplan draft with enough notice (at least a couple weeks) so that the reviewers have time to perform a thorough review. If it is clear during the meeting that the reviewers did not review the testplan ahead of time it can be more effective to reschedule the meeting to allow them more time to review, rather than having them read the testplan for the first time during the meeting. A teammate should attend to take notes and assist

the area owner as needed. Bringing donuts to the testplan review meeting can be an effective way to encourage attendance and participation.

If the changes to a feature are relatively small, an offline testplan review may be appropriate. For an offline review the validator should email their testplan to the reviewers and request feedback by a certain date. This approach often requires more follow-up than an in-person meeting.

Upon completion of the review (whether performed in person or offline), each stakeholder should give their “Stamp of Approval” to the testplan.

6.4 Random vs. Directed testing in FAV

Though random testing is widely used in many other pre-si validation disciplines (see [Stimulus](#)), most FAV features are covered primarily with directed tests. FAV features often require a specific sequence of operations to set up a condition and these operations are either not easy to randomize or not interesting to randomize. That does not mean there is no randomization; code and data address ranges, execution modes, and paging modes are among the parameters that are randomized in most FAV tests.

Coverage is typically used with random testing to ensure that the conditions of interest are being hit. With directed testing the set of directed tests act as coverage. When the set of directed tests are run, the conditions of interest (defined in the testplan) are covered. The FAV directed test files are stored in a central database and revision-controlled. Each test file will have one or more test records (with varying command line arguments defined), each covering a different portion of the validation space.

6.5 Tools

6.5.1 Test Generators

FAV tests generally consist of x86 assembly instructions, also known as macrocode, that are executed by a CPU core, either in a core or fullchip configuration. x86 requires various data structures (descriptor tables, page tables, etc.) to be created as well as setup code to get a test to the point where interesting user code can be executed. An ideal FAV test generator takes care of as much of this generic setup as possible, allowing the test writer to focus on the specific test components unique to the particular testcase. This makes tests easier to write and maintain and makes the test file itself more readable. Some tests in certain FAV taxonomies consist primarily of test environment code or custom firmware patches.

It is important to be able to easily set up the various conditions that the test writer may be interested in. For example, a validator might want to write a test where a page fault is triggered by a specific instruction; a good test generator would make setting this up relatively easy. This means that tests would have a higher-level component in addition to specified x86 macrocode. This higher-level component would allow randomization of parameters, computation of values based on the specific model the test is being run on, and customization based on the project or stepping on which the test is being run.

An ideal test generator would have a quick build time to allow for quick iteration during test development. It is also important to have good error messages if something fails during the test build.

6.5.2 Injectors

In order to set up certain test conditions it is useful to be able to inject values into an RTL simulation. For example, control registers are commonly injected to set defeatures, enable debug modes, or overwrite default values with something specific to the test case being written. It is also important to be able to inject various architectural and microarchitectural events (like interrupts and traps) in order to validate the interaction of these events with a FAV feature, or to use an event to enter or exit a particular processor mode. In some cases it can be useful to be able to pound individual RTL signals as well.

In addition to providing flexibility of what can be injected, an injector well-suited for FAV should also provide flexibility of when the injections occur. It is often useful to be able to trigger an injection based on the completion of an x86 macroinstruction, the completion of a particular firmware instruction, or a certain amount of time following a previous injection. It is also useful to be able to control whether an injection should occur every time the triggering condition occurs or only some of the time.

Validators need to be careful not to rely too heavily on injections in FAV tests. Injections can bypass certain hardware or firmware flows, masking bugs in these flows. Testing full feature flows at the fullchip level with as few injections as possible provides the most confidence these features will work as intended on silicon.

6.5.3 Checking

Many core and fullchip tests in areas outside of FAV rely on an architectural checker that compares the architectural state following each macroinstruction with the state in an architectural simulator. If the architectural simulator does not fully model the behavior of a FAV feature (as is typically the case) it can falsely flag errors during FAV tests.

Some form of checking is required to determine whether a FAV test passes or fails. Many FAV tests rely on “self-checking.” This means that the x86 macrocode in the test itself checks an architectural or microarchitectural condition that indicates whether the feature being tested behaved as expected.

Some aspects of FAV features are difficult to self-check. For these conditions a custom checker can be written and added into the test environment. This checker can look directly at signal values and tie in with other portions of the test environment as needed.

Many checkers and monitors written and maintained by other teams are run by default during FAV tests. Some of these print output into various log files and in some cases a Perl postprocessor can parse these log files to determine whether a particular portion of a feature behaved as expected.

6.6 Test environment

6.6.1 Simulation

Historically FAV tests have been run on simulation. Fullchip simulation tests run very slowly, on the order of 10 Hz, limiting how long a test can be. Though some full flow tests are quite long, many FAV test conditions can be covered with relatively short tests. Simulation gives a lot of flexibility to checkers and injectors, allowing access to read or write any RTL signal. Simulation models can be easily cloned and altered for side runs; this is especially useful during FED as new features are coded, allowing validators to quickly fix bugs and iterate through problems. In simulation the test environment takes care of a lot of initialization. User tests are run immediately following a (shortened) reset flow. This short setup code makes it less likely for a single bug to cause all tests to fail.

6.6.2 Emulation

Tests run on emulation (specialized hardware based on FPGAs) typically run more than three orders of magnitude faster than those run on simulation resulting in tests completing much more quickly than when run on simulation. This enables much longer tests to be run on emulation than can be run on simulation. It also allows tests to run through the full reset flow rather than the shortened/pounded flow used in simulation. Checking on emulation relies more on self-checking tests and post-processing than on-the-fly checking. Injections are much less flexible on emulation than on simulation. Most existing cluster checkers and monitors do not run on emulation, reducing checking but also avoiding having to deal with checker and test environment bugs. There is a longer initialization sequence in emulation before a test is run. This makes it easier for a single bug to block all tests on a particular model. Overall, emulation shows promise for FAV in areas such as reset and for long authentication flows. As project configurations become more complex (more cores, more integrated IP blocks) simulation run times can increase to the point where emulation is the only feasible approach to validate certain flows. The full details of the FAV emulation methodology are still a work-in-progress.

6.7 FAV regressions

6.7.1 Periodic regressions

Once FAV tests are written and passing in side runs they should be added into regression testlists. These regressions will be run regularly as the project progresses to help find newly introduced RTL and firmware bugs. The optimal frequency of these periodic regressions varies based on the scope of change in a project, the project phase, and the team's debug bandwidth. If there is a lot of change it is useful to run regressions more frequently. If there is less RTL change, it is less likely for new RTL bugs to be introduced and regressions can either be run less frequently, or a subset of tests can be rerun during each regression.

6.7.2 Gating regressions

Once an FAV feature has a basic level of health in the released model it is useful to include one or two tests into the gating regression lists. These gating regressions are run every time anyone attempts a turnin to a code repository and all of the gating tests must pass before a turnin is accepted. Because FAV features have many cross-cluster interactions it can be especially easy for a designer in one area to make a change without realizing the impact on a broader feature. Including FAV tests in these regressions helps protect against this problem.

7 Summary

FAV validates a set of important features that typically have many cross-cluster interactions, have an architecture that is not published external to Intel, and play a role in enabling effective post-silicon debug. These characteristics of FAV features result in a number of challenges Feature Architecture Validators must overcome to effectively perform their job. FAV testplans enumerate specific test cases which are typically covered using a collection of directed tests. Test generators, injectors, and checkers used in FAV support this directed testing approach. FAV tests have historically been run on simulation though emulation has the potential to enhance the validation of some FAV areas.

8 Future Work

9 References

The Art of Pre-Si Val: Chapter 36

Reset and Power Management Validation

By: [Garret Staus](#), [Kavitha Ramasamy](#), and the DDG RPMV Team

1 Abstract

The charter of reset and PM validation is to target flows that involve removing or adding power to the system. This presents unique technical and methodological challenges that many other validation teams do not have to face. Validators have to make special considerations regarding how to model power loss, how to validate the multiple types of firmware typically involved in these flows, and how to validate global flows that inherently involve most or all IPs in the system. This chapter will explore these challenges and the solution space for them.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	7/5/2016	Initial Draft	Garret Staus	Michael Bair

3 Contents

1 Abstract.....	713
2 Chapter Revision History	713
3 Contents.....	714
4 Purpose.....	716
4.1 Why do we need this chapter?.....	716
5 Background Concepts	716
5.1 PMU & Pcode	716
5.2 UPF	716
5.3 X Propagation.....	717
5.4 Power Gating	717
5.5 Clock Gating	717
5.6 Power States	717
5.7 State Retention.....	717
5.8 Save & Restore.....	718
5.9 Isolation	718
6 Reset and Power Management Validation	718
6.1 Commonalities between Power Management and Reset	718
6.2 Importance of UPF and X-Propagation.....	719
6.3 Validation Obstacles Unique to RPM.....	719
6.4 Hierarchical Validation	720
6.4.1 IP RPM Validation.....	720
6.4.2 IP Integration RPM Validation.....	721
6.4.3 Global PM Validation	721
6.5 RPMV Test Environment	722
6.5.1 RPMV Requirements for TE Components	722
6.5.2 Special RPM TE	722
6.6 Working with Firmware	724
6.6.1 Firmware in the Design	724
6.6.2 Validation with Firmware.....	724
6.6.3 Challenges of Validation with Firmware	725
6.6.4 Firmware Related RPMV Validation Skills	726

6.7	Power and Performance Validation	727
7	Summary.....	728
8	Future Work.....	728
9	References.....	728

4 Purpose

4.1 Why do we need this chapter?

The usage model of Intel's products is evolving at a fast pace, making reset and power management increasingly important and complex. Battery life and wake time are primary considerations when marketing a product as opposed to raw processing power, and validating these flows has become a significant undertaking.

Reset and Power Management (RPM) require special consideration for Validators for several reasons. Both flows are global in nature, with interactions from all IPs and fabrics, so an open relationship with other validation teams is critical to ensure that the entire design is participating in RPM flows as expected. In addition, accurately modeling power delivery and power loss is a critical factor in RPM validation. This presents challenges in the test environment and creates unique dependencies on the design. Another impactful consideration is that the scope and length of RPM flows is larger than most validation flows, and the scope of coverage and checking is slightly different compared to other validation areas.

This chapter will explore these obstacles and potential solutions in detail. It will not tackle validation approaches for specific types of power or reset flows, but will approach these flows generically to identify their common problem and solution space.

5 Background Concepts

Validating reset and power management flows involves a large number of terms that do not frequently get used in other validation areas. These terms describe design logic related to power as well as the tools Validators use to effectively model power and its effects.

5.1 PMU & Pcode

In almost any Intel design, a central IP is responsible for coordinating the system during RPM flows. This IP typically involves hardware FSMs and a microprocessor that runs firmware, and depending on the project, may have one of several names. Common names include P-Unit, PCU, PMU, and PMC. This chapter will refer to this IP as the “PMU” and the firmware that runs on it “Pcode”, for consistency.

5.2 UPF

Unified Power Format (UPF) is a language used to describe the power intent, or power strategy, of a logic design. UPF is IEEE Standard P1801. A UPF file defines power domains and their associated supply nets, power switches, isolation, level shifters, retention, power states and voltages.

From a validation perspective, UPF is critical because it allows Validators to test system flows that involve power loss and delivery. Domains that have no power will be injected as unknown values in a simulation (“X”), instead of known values (“0” or “1”).

5.3 X Propagation

Typical RTL simulations use an overly optimistic approach to propagating X’s through the design, particularly through certain types of logic blocks. One solution for this problem is the “VCS X-propagation simulator”, which allows X-propagation problems to be exposed by standard RTL simulation.

5.4 Power Gating

Power gating is a flow wherein a domain of the chip may have its power locally turned off to reduce power consumption and leakage. This domain is referred to as a “gated” domain, and requires physical power gate (PG) cells to ramp voltage based on control signals. Usually a UPF or a Verilog file models the PG cells.

5.5 Clock Gating

Clock gating is a technique used to save power during low power states by turning off the clocks to sub-domains of a hierarchy. The voltage domains that a clock belongs to, as well as the current power state, are both factors considered for clock gating. Trunk gating is another technique wherein a set of IPs sharing a clock trunk can shut down the corresponding clock trunk when the entire set of IPs is not requesting the clock to be active.

5.6 Power States

A power state is a term to describe the current state of the system in relation to what power rails are ramped and what parts of the system are power and/or clock gated. The OS or some other software initiates some power states, while the hardware autonomously initiates others based on inactivity within the system.

There are many different levels of sleep states and shutdown. Everything from full-shutdown to full-on can be represented by a particular power state.

5.7 State Retention

When a domain powers down, all its sequential elements are corrupted. Retaining (or saving) the state of some sequential elements is advantageous because it avoids the need to do a full reset, which consumes a large amount of time and power.

Local state retention retains specific registers and sequential elements and restores them back, obviating the need to reset them. However, local state retention has an area and leakage overhead on the backend, which designers must consider. When power is fully removed from a logic block, register states may be retained in a dedicated RAM.

Another solution to retaining state is Save/Restore, which is discussed in the following section.

5.8 Save & Restore

During entry and exit from sleep states, IPs in the SOC need to externally save their context before they lose power and then restore it when exiting the deep power state, to meet aggressive entry and exits latencies. Hardware, firmware, and drivers are all responsible for save/restore in different times in the flow, depending on the needs of the IP. The PMU plays a crucial role in coordinating save & restore.

5.9 Isolation

Powered-off domains and IPs do not drive their outputs during low power states, leading to floating output nodes. This might be a problem when some other active module uses those floating nodes as an input. This unexpected signal value might cause high current consumption because of an inappropriate logic level, or it might cause improper logic behavior of the active module because of the “gibberish” inputs coming from the turned off domain. Isolation cells placed between power domains solves this problem by driving constant supply (i.e. driving 0/1) or by latching the old value when the domain is down. Isolation cells assure predefined or latched values to be available to the other active domains.

6 Reset and Power Management Validation

Reset is a generic term used to describe a flow where the system is waking from an approximately “full off” power state to an approximately “full on” power state. Typically, this refers to cold boot, which is a flow that happens when pressing the power button on your laptop or desktop after it you have shut it down. It may also refer to the flow of re-booting after you have restarted your machine, or have woken it up from an OS-initiated sleep state.

The idea behind power management is simple: deliver performance when needed, save power when not. PM flows are usually hardware or firmware initiated flows that are invisible to the operating system, and should be as seamless to the user as possible. They typically require additional configuration to allow for power states where only some power is removed, but not all, or when only certain parts of the design are awake (and others are asleep).

6.1 Commonalities between Power Management and Reset

Historically, DDG has treated Reset and Power Management as two different disciplines. More recently, DDG has combined the two into one team due to their similarities, both in philosophy

and execution. While Reset and Power Management target different flows and test content, both disciplines are concerned with many of the same aspects of the design and validation environment.

Both Reset and PM involve the bringing up and bringing down of voltage rails, and the consequences of doing so. During reset flows, rails will frequently be taken away and brought back up (in addition to reset being asserted); in PM flows, rails are typically dropped as part of an autonomous sleep state in which system state must be saved and restored. Removing voltage has a huge range of consequences, and it is necessary to obtain an accurate way to model power loss and its effects on the digital logic. See the UPF/Xprop section of this chapter for more information on the details of this modeling and its importance. Most other teams do not test power removal, so it typically falls to the RPM team to find issues with isolation or retention at the SOC level. Additionally, test environments and RTL assertions that were coded without power loss in mind leads to many false errors that can hamper the debug effort and affect the overall project.

6.2 Importance of UPF and X-Propagation

The key trait of power loss within a domain is that all logic on that domain will no longer be driven. The result on silicon is effectively an unknown value on any wire that would normally be a 0 or a 1. Without UPF, simulation tools would not contain the required information to accurately model this effect (instead, the value is typically retained on the wire). RPMV flows frequently cycle power and it is crucial that system state is accurately modeled so that bugs do not escape to silicon.

When power is dropped, there is a chance of X's propagating out of power gated domains to those which are not power gated. X-Prop assists in aggressively modeling the propagation of X's through logic blocks (such as a mux) so that any potential issues are discovered before silicon. In some cases, this may lead to overly pessimistic modeling and wasted validation time on x-propagation scenarios that will not actually occur on silicon. Validators and validation managers must be aware of this trade-off when extensively using X-Prop on simulation.

6.3 Validation Obstacles Unique to RPM

The nature of RPMV leads to many obstacles that are unique, or at least more prevalent, than in other validation disciplines.

The global nature of RPMV flows makes it very difficult to code detailed and low-level checkers and monitors for RPMV flows. Additionally, the architecture of sleep entry and exit flows are often subject to change as the needs of firmware and hardware change throughout a project. Due to this, RPMV tests are frequently lightweight in the amount of checking they perform. Instead, the philosophy is that most RPMV bugs will cause a functional failure in the flow itself, and so self-checking tests are sufficient to catch most bugs. To catch other, non-functional bugs, detailed waveform reviews are held at multiple stages of a project to sanity-check important interfaces and areas of the design that are suspect to bugs. There are some exceptions to this rule, such as the save/restore checker.

Designers and Validators alike frequently code assertions and checkers without power loss in mind, and so RPMV spends a large amount of time debugging and understanding RTL assertions. Most often, these assertions are firing due to a bug in the assertion itself, rather than an actual RTL

issue. It is a common occurrence to see hundreds or thousands of assertions fire when enabling an RPMV flow for the first time, and few ever lead to real bugs. However, one must debug and understand each assertion to ensure quality health of the design.

During the entry and exit of many RPMV flows, clocks need to be requested, acknowledged, and unrequested according to specific protocols to ensure proper functionality. A violation of protocol may not be caught by a generic test as a functional failure, making waveform reviews even more important. While debugging a failure, RPM Validators need a detailed understanding of clocking and the handshakes required to be able to fully root case many bugs.

Lastly, many IPs are integrated into the design with a default configuration, and a different “low power” configuration, which is typically used by customers to help save battery life in their systems. RPMV must test all low power configurations together and perform the suite of RPMV flow testing to ensure that these flows still work with the different configuration. This typically requires extensive coordination with IP integrators.

6.4 Hierarchical Validation

Power Management Validation is done at multiple levels:

- *Firmware Validation*: Firmware tested with no RTL modeling.
- *PMU Validation*: PMU-only with real firmware
- *IP PM Validation*: Done at the IP level where PMU and firmware are emulated
- *IP Integration PM Validation*: Done at the SoC level where the real PMU and firmware are used.
- *Global PM Validation*: Done with all IP's, concentrating on PM interactions between different IP's.

The goal is to validate features at the lower levels as much as possible and catch bugs. This is very important, as the cost of discovering basic power management bugs in larger environments is higher due to slower runtimes and test cases that are more complex. Integration PM Validation relies on extensive, localized IP testing done at the IP level. Global validation relies on Integration and IP teams to do extensive testing of the IP so that the Global team can concentrate on cross-feature, concurrency and use case validation.

This section discusses the minimum amount of validation required at different levels. This is not an exhaustive list of all the conditions tested at the IP level and is not a substitute for a good, detailed IP-level PM testplan. It highlights some of the conditions that the integration team believes need to be tested at the IP level before integration and describes how the integration team expects those conditions are tested. For more details on FW Validation please refer to the [Embedded FW Validation](#) chapter.

6.4.1 IP RPM Validation

IP power management features are validated at the IP level with PMU and Pcode emulators that mimic the behavior of real RTL for the required handshakes between IP and PMU. As the

validation environment contains only the IP of interest, it takes far less time to run a test compared to higher levels.

Some of the basic features that need covering at the IP level (not limited to this list) are:

- All local power conditions (power actions that do not require interaction with a PMU)
- All interactions with PMU (with PMU responses being emulated)
- Any power related interaction with other IP's
- Local power gating conditions
- Power corner case testing
- Delivery and validation of IP RDL for save restore
- Basic entry/exit for deep package states
- Traffic before and after deep package states
- Wake events and abort conditions with package states.
- Device power down states

The IP team also provides the checkers and sequences that can be re-used at higher levels (Integration and Global).

6.4.2 IP Integration RPM Validation

Once SOC design and validation teams have integrated the IP into the SOC, the integration team validates the IP power management features with real PMU RTL and firmware. The integration team works with the IP team to re-use most of the validation collateral that was developed by the IP team. All the IP PM features are enabled at this level, using either simulation or emulation platforms based on the test conditions and IP. Validation may disable IPs for this testing, depending on the requirements.

Some of the items that the integration team usually owns include:

- Detailed pre-drop review with the IP team
- Basic connectivity and basic protocol testing of all interfaces to the IP (between IP and PMU)
- Deep power states
- Wake events and abort conditions with deep power states
- Heavy IP traffic before and after deep power states

6.4.3 Global PM Validation

Global power management validation brings all the IPs together and validates the features of the design against them. Typically, this occurs at two levels; uncore and full chip. The uncore platform uses emulated cores to mimic the IA cores. The goal at this level is to validate PMU at a global level. Full chip targets validation with the full RTL, including cores and graphics.

Below are some aspects to focus on when doing global PM validation.

- Global flows (reset and PM states which affect all IPs)
- Cross-feature testing where different PM features are run at the same time to hit corner case scenarios.

- Cross-IP PM flows where PM flows from different IPs are enabled at the same time. This includes traffic from different IPs where appropriate with PM flows.
- Making sure deep power states are working with all IPs involved.
- Some concurrency traffic from different IPs.
- Wake up events and abort conditions with deep sleep states.

6.5 RPMV Test Environment

Even though RPM features are exercised at the IP level, validation of global RPM flows requires the RPMV team to operate at integration level (typically SoC). Due to the size and complexity of SoC models, simulation capacity is very limited and simulating even a tiny portion of some PM flows can take several days. While there are many factors involved in simulation speed, the TE (Test Environment) is a big part of it. In addition, exercise of certain RPM features can often expose conditions that TE programmers forget (e.g. losing power).

In this section, we list additional requirements that RPMV enforces on all TE components working at the RPM level. We also talk about special TE components designed to validate or support testing of different RPM features.

6.5.1 RPMV Requirements for TE Components

When coding a new piece of RPMV TE, the TE coder should ask them self these questions:

- Can my TE components, especially checkers and assertions, handle different RPM flows, including all power states? Will they all work correctly when the power to the corresponding design block is turned off? This is probably the most important PM requirement to all TE components – the ability to handle these flows correctly without creating false failures and stopping simulation.
- Does the TE require anything extra to model realistic SOC behavior? For example, if a BFM drives an interface that may lose power and start driving Xs to the design, does the TE have a way of modelling this behavior?
- Does the TE use correct data types for storing RTL values and perform the correct operations on those values? This is especially important when considering 2-state or 4-state variables and operators.

6.5.2 Special RPM TE

In general, test environments designed to support validation of RPM features should follow the same rules and coding practices as any other TE component. The following are specific TE components which are frequently owned or maintained by the RPMV team itself.

6.5.2.1 Emulators/BFMs/VCs/Responders

If a design block that should participate in global PM flows is not present in a DUT, there must be some TE component that emulates this block's PM functionality. The complexity of such emulators varies greatly depending on the emulated functionality, the intelligence and configurability of the emulator, and other factors. There may already be a BFM for the design block, which RPM Validators can configure or extend. In other cases, a Validator may need to implement a new TE agent. In some cases, the design block (e.g., PMU) is so complex that coding and supporting an emulator may take several months of full-time work. It is very important to understand the complexity of the emulators when making decisions on what you stub out of your DUT.

While every project is different and there is no common guideline for coding such emulators, it is good to keep the following in mind before you start the actual coding.

$$e = mc^2$$

Do not confuse it with the famous Einstein's equation. The actual meaning is

$$\text{Errors} = (\text{more code})^2.$$

In other words, the more accurate you want your emulator to be, the more code you must write to handle different corner case scenarios, and therefore the harder it is to maintain and debug. Always try to design your emulator as high-level as possible and do not overcomplicate the implementation.

6.5.2.2 Save/Restore Checking

Historically, the Save/Restore (S/R) checker is one of the most important RPM checkers, and Validators should consider it necessary for any power-state tests where save/restore occurs. The main idea of the S/R checker is very simple: find all registers with S/R attributes, take a snapshot of every register before the save, take another snapshot after the restore, compare the data, and report an error if there is a mismatch. In most implementations, “backdoor” control register (CR) reads make the snapshot. The checker needs to ensure that power loss corrupts CR values between save and restore, otherwise it may not catch interesting design bugs.

Modern designs keep all registers and their attributes in a database. This database should provide a way for the checker to get a list of all registers, marked with S/R attribute. The checker could get the same information directly from firmware instead, but if there is a bug in firmware, the checker will not be able to catch it.

The checker may be very simple or complex depending on when it chooses to sample the CRs. Ideally, the checker should take the “save” snapshot for each register at the same time when firmware reads the CR, and take the “restore” snapshot immediately after the data is written to the CR in the restored design block. If either save or restore sampling time is misaligned between firmware and the S/R checker, and a CR happened to change value during that time, the checker would report a false failure. In reality, making this “ideal” checker is not typically a good idea, as it would require the checker to watch for activity on lots of interfaces, decode all traffic, and would require intimate knowledge of the firmware.

Through a different approach, the S/R checker can be very simple. It could take a snapshot of all S/R registers once for save and once for restore, much later than firmware actually does this. To

avoid the false failures caused by misaligned sampling time, the test could help the checker by quiescing the traffic in the system. The issue with quiescence, however, is that after a period of inactivity, a design unit may save its CR to some internal memory and enter an auto-PG state. When the checker finally decides to sample the data, it may get an X-value.

Each of these approaches has its pros and cons. It is good to follow the “truth is somewhere in the middle” rule and implement multiple sampling points, but probably not on a per-register basis.

Since the checker is working at the SOC level, it should understand which design blocks are not present to exclude those from checking. In addition, the checker should provide a way to exclude checks for specified registers from TE and from a command line.

6.6 Working with Firmware

6.6.1 Firmware in the Design

Modern SOC and CPU designs typically include some form of FW. Whenever a sub-block or IP contains its own microprocessor, chances are it also has its own FW. IA client products typically have at least two: one for the Core, called microcode, and one for the PMU. An SOC will most likely have more depending on how many IP's have a microprocessor. The BXT family, for example, had six different types of firmware in six different blocks. Having this additional element increases the complexity of the validation space.

6.6.2 Validation with Firmware

Firmware affects Power Management and Reset in particular, as it controls large portions of the functionality related to those areas. The sequencing and various phases of reset, the long sequence of putting the Package or the SOC into one of a dozen power saving states, and handling thermal events are just a small sample of the RPMV functions controlled or affected by FW. This is the case not just for the PMU, but for the IPs as well. In most cases, an IP that has its own microprocessor will control the reset and power management for that IP. This sets up a natural path of interaction between the PMU FW and the IP FW, and with it, a large amount of potential problems.

It is very important that the final version of the FW (production FW) be available very early in the project validation cycle so that it can be included in all pre-silicon testing. In addition, all the test environments and platforms should be able to include the microcontroller and the respective FW. This way, the intended behavior of the RTL and the interactions of various FW with each other can be exposed and tested and all the various FW assumptions verified. Running with actual FW has the big potential of exposing RTL bugs that may not be visible otherwise, and of course, FW bugs will be uncovered. This goes a long way in improving the quality of the product making Si bring-up and PRQ faster.

6.6.3 Challenges of Validation with Firmware

6.6.3.1 Firmware Availability

The realities of working with firmware often necessitate pragmatic solutions. Production FW, or even the engineering version of the FW, is not typically available early in the project. For projects that are the first in the family line, it is likely that teams are developing FW at the same time as the RTL. For derivative products, usually an already existing code base is available that is close enough to start with. An internal team typically develops the PMU FW, allowing RPMV to work with them more closely in terms of schedule and feature requests. IP FW, however, usually comes from completely different organizations, making it more challenging due to a difference in priorities and schedules.

How to handle late FW availability depends on whether external or internal groups are developing the FW. If it is developed within the same organization, validation should attempt to try to get the code as early as possible. If that does not work, then validation should establish a viable schedule with the FW. There needs to be a discussion on what the priorities are for feature development and testing, as well as the various schedule pressures each group is experiencing. Validation and FW need to agree on the timing and sequence of feature delivery, and then map out the schedule to see where things line up and what features are the pain points. In other words, RPMV needs to have close coordination with the FW team and have clear expectations from each other.

Sometimes, even with this close coordination, some of the features will not be in time for the validation goals. Occasionally, the planned FW structure is not enough to hit timelines, even with efficient validation. The next step, which can be a costly one, is to write custom validation or engineering FW as a stopgap. Validators will need to make a judgment call, in consultation with managers and experts, on whether or not to invest effort in developing validation FW. Sometimes, the FW team may be able to provide a skeleton FW or framework that will make it easier to add additional code to make validation FW. Validation should request this from the FW team as early as possible in the project. If creating engineering FW is not practical, simply waiting for the actual FW to become available is an option. Validators and managers need to comprehend this into the schedule and properly assess the risk.

In past projects, silicon bugs have appeared in IPs that validation could have found pre-silicon, if the proper production FW had been run. If an external team is delivering the IP FW, then it is a little more challenging. The first item to investigate is if the FW team can deliver the FW on time, and if not, validation should understand the risk and impact to schedule. When working with external FW teams, the probability increases that engineering FW will need to be developed, and so it is prudent to include these costs early in the planning stage. It is likely that the IP Integration Validator will write the engineering FW, and it is important to make sure he or she adds RPMV tasks to their planning costs and tries to get an early agreement on this plan.

6.6.3.2 Computational Cost of Running with Firmware

When the FW is available, there are cases where the simulation or emulation cost of running with the FW maybe too high or prohibitive. RPM Validators need to understand the expected effect on speed of simulation or emulation runs when the actual FW is included and plan accordingly. Firmware should be used in simulation and emulation if the performance cost is determined to be

acceptably low. As the performance cost increases, it becomes more important to be selective over which tests run with firmware. Each project will need to decide how much cost is acceptable.

For the primary PMU firmware, however, all effort should be taken to run with the latest released FW code with all pre-Silicon content. This has been a repeated learning from both the client and SOC families to the point that not running with the real FW pre-Si would be considered a drastic and potentially unacceptable risk in RPMV. One should consider all input from stakeholders and managers when deciding to run firmware during pre-silicon validation. So far, there has not been an issue concerning computational resources on either client or SOC.

For IP FW the goals should be similar to PMU firmware. However, with some IPs the introduced delays are prohibitive. On some SOC designs, running with the actual FW for some IPs can cause reset alone to take weeks to run. This is sometimes due to the sheer size of the FW itself, or sometimes it is part of the current technological limits of the simulation and emulation tools. When the cost of running with FW is prohibitive, there is no alternative but to create validation FW. It is important for this decision to be made early so that planning can start early in the project.

6.6.3.3 Validation Firmware: Some Tips

If writing Validation FW is inevitable, here are some things to keep in mind:

- Evaluate the risk of not creating validation FW and just either wait for the FW to arrive or test the feature in silicon instead. These cases may be rare but it might be acceptable.
- In the absence of actual FW, have architects and FW coders come up with a clear description of the flow for which the validation FW is being written
- Try to see if the FW team can provide a basic skeleton or framework, upon which one can build the validation FW.
- Structure the validation FW code in a way that will allow additional features to be added or supported later on.
- Have a good branching and revision control system. If Val FW is needed in one stepping, it will most likely be needed on others and planning for how to keep track of changes and branch versions for each stepping in advance will save a lot of time. Val FW usually starts out in someone's personal directory; however, it eventually becomes project critical and therefore planning should attempt to include this early on.

6.6.4 Firmware Related RPMV Validation Skills

Since RPMV features interact a lot with FW, RPMV Validators need to develop additional skills to help in this area. First, RPMV Validators should develop good working relationships with the FW coders and FW Validators in addition to those with design and architecture. This is especially true for the PMU FW. Having good communication will minimize misunderstandings and save time.

Secondly, dealing a lot with FW also necessitates that RPMV Validators need to develop the additional skill of being able to read and understand various types of FW. This could be in more than one language. RPMV Validators are well-served by being able to open the actual FW, read, and understand what it is doing. As there are numerous questions and bugs in this code, debug is

much more efficient if Validators and FW coders speak the same language and can read the same code. When the debug path leads to the FW, the Validator should not shy away from diving in, but should instead be able to do a cursory inspection of the code, get a general understanding of what is going on, and try to see what the problem is. This is similar to being able to read RTL code.

Validators also need to develop a working understanding of the general architecture of the PMU firmware. This helps in debugging failures more quickly and bringing additional insight into the big picture of RPMV.

Lastly, RPM Validators need to be familiar with the FW related debug tools available as well as the various log files and list files that are created in the process of building models and running tests.

6.7 Power and Performance Validation

Power and Performance (PnP) Validation looks to verify and improve the power usage of the CPU/SOC in various system power states. For each power state, architecture provides a specification of the expected state of voltage rails, clocks, etc. A validation test is run to get the system in a certain power state. Validators then check the power state in that test run against this specification. Some things that are checked include the state of LDOs, SRAMs, register files, clock gates, and power gates. This requires IPs to behave as they would in a production system. Therefore, the test environment needs to mimic this configuration carefully. Once Validation has a trace that matches the specification for each power state, this trace is sent to the design team who will run the trace through their tools to estimate power. Architecture, design, and validation groups analyze the results to make sure that the power targets are met.

The team also identifies areas where additional power can be saved by analyzing test run logs, simulation traces, and the power estimates from design. Some examples of power saving methods are the following:

- Reducing hardware latency
- Changes to firmware to reduce latency by removing unnecessary code or by re-ordering flows to hide long latency tasks
- Looking for clocks that are toggling that could be gated
- Looking for opportunistically running clocks that could be gated more often
- Looking for HW that could/should be power gated
- Doing Proof-of-Concept implementation by prototyping HW changes in pre-silicon, and FW changes in pre-silicon and post-silicon.

When issues are found or optimizations are approved, they can be addressed via a hardware or firmware change. The validation team works closely with Architecture and Design to attempt to produce optimal solutions to save power.

7 Summary

Reset and Power Management flows are a critical feature of today's Intel chips, and by extension, the validation of those flows has grown increasingly important and complex. RPM flows are especially unique because they attempt to model and validate unique features of the chip, such as power loss and the ability to recover from power loss. As long as power consumption is a top consideration in today's designs, the learnings and approaches for RPM Validation will remain equally important.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 37

Performance Validation (TBD)

By:

1 Abstract

Performance Validation addresses validation of various performance vectors such as memory and IO bandwidth and latency.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers

The Art of Pre-Si Val: Chapter 38

Power Performance (PnP) Validation (TBD)

By:

1 Abstract

Power Performance (PnP) Validation targets whether a product's clock domains and power domains lower or shut down as needed to minimize power and hit the right level of performance per watt.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers

Security Validation

By: [Soon Seng Loh and Jean-Philippe Martin](#)

1 Abstract

Validating security is important because it ensures that a product or system has been designed and implemented with security in mind. It helps identify and resolve potential vulnerabilities before they can be exploited by attackers. Security issues in products can have disastrous consequences: data breaches, service disruption, reputation damage, financial and legal penalties. Neglecting security validation at the initial stages can lead to increased development costs in the long run, as engineers will have to focus on fixing vulnerabilities in previous products, instead of spending their bandwidth working on new projects.

Security intent validation is the process of confirming that a product or system has been developed and implemented with security considerations in mind. It includes:

- 1) Feature validation: verifying that security features and controls are implemented correctly, functioning correctly, and configured properly.
- 2) Security assurance: ensuring that there are no defects or issues that could potentially create vulnerabilities and that the IP or system is capable to respond adequately to attacks

Security intent validation in the pre-silicon phase may not be a common task for validation engineers. Therefore, this chapter will offer an overview of the currently available tools, methodologies, and best practices.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
0.1	Sep 2023	Initial Draft	Soon Seng Loh and Jean-Philippe Martin	Michael Bair
0.21	09/25/2023	Revised Initial Draft	Soon Seng Loh and Jean-Philippe Martin	Michael Bair
0.22	09/25/2023	Revised Initial Draft – Open for reviews	Soon Seng Loh and Jean-Philippe Martin	Michael Bair
0.23	02/01/2024	Revised including comments and suggestions from the following individuals: <ul style="list-style-type: none"> • Martin-Langerwerf, Javier • Carreno, Julien • Morales Gonzales, Jacob • Barkur Sadshiva, Sweta • Morfin Mendizabal, Erick • Liew, Vui Yong • Yitbarek, Salessawi Ferede • Levin, Alexander S • Ariel, Itay • Weng, Lichen • Bair, Michael 	Soon Seng Loh and Jean-Philippe Martin	Michael Bair
0.24	3/31/2024	Initial Publication Revision	Soon Seng Loh and Jean-Philippe Martin	Michael Bair

3 Contents

1 Abstract.....	733
2 Chapter Revision History	734
3 Contents.....	735
4 Purpose.....	737
4.1 Why do we need this chapter?.....	737
4.2 What does this chapter cover?	737
4.3 What does this chapter not cover?	737
5 Background Security Concepts.....	737
5.1 Security Mindset	737
5.1.1 Security awareness	737
5.1.2 Security continuous learning.....	738
5.1.3 Security processes and activities.....	738
5.1.4 Managing security.....	738
5.2 Security Concepts.....	739
5.3 Examples of security functions	740
5.4 Examples of secure flows in a system	742
6 Security Validation.....	743
6.1 Security – functional validation	743
6.1.1 Cross-domain validation and security	743
6.1.2 Validating crypto blocks, RNGs, PUFs.....	743
6.1.3 Validating access control	743
6.1.4 Validating control register access control on IOSF	746
6.1.5 Validating memory protection	749
6.2 Client SoC secure flows.....	752
6.2.1 Validating secret download and random number read.....	753
6.2.2 Validating FW loading/patching	753
6.2.3 Validating memory encryption.....	753
6.2.4 Validating storage encryption.....	754
6.2.5 Validating Secure Boot flows	754
6.2.6 Validating BIOS Guard (Platform FW Armoring Technology)	756
6.2.7 Validating Intel Trusted Execution Environment (TEE)	756

6.2.8	Validating Software Guard Extensions (SGX).....	757
6.2.9	Validating Trusted Domain Execution (TDX) based on MKTME	758
6.3	Negative testing	758
6.3.1	Negative Space classification for validation	759
6.3.2	Examples of Positive and Negative Testing.....	760
6.4	Security assurance	760
6.4.1	Security considerations for IP Integration	761
6.4.2	Threat modeling and attack reproduction.....	761
6.4.3	Static configuration input sanitization.....	761
6.4.4	HW Fuzzing	762
6.4.5	Structural hazards and logical side-channels.....	763
6.4.6	Protocol bridges.....	766
6.4.7	Formal Verification and Secure-Path Verification	767
6.4.8	Information flow and TProp.....	768
6.5	Trade-offs	768
6.5.1	Verification Scoping, Risk assessment And Prioritization	769
6.5.2	Discovery vs Assurance.....	769
7	Summary.....	770
8	Future Work.....	770
9	References.....	771

4 Purpose

4.1 Why do we need this chapter?

This chapter on security validation is important for establishing a strong foundation in secure development. By understanding the importance of security testing and validation, you'll gain awareness of how to build secure systems from the ground up. The chapter explores high-level concepts to equip you with the knowledge to identify vulnerabilities, validate implemented security measures, and ultimately create trustworthy and resilient products.

4.2 What does this chapter cover?

This chapter covers background security concepts, it then covers high-level concept of security validation which span over validating security functions, validating secure flows, negative and fuzz testing and finally security assurance.

4.3 What does this chapter not cover?

This chapter assume that the reader has understanding of standard validation methodologies and techniques cover in other chapters of the book. It will not cover validating specialized logic or advanced techniques. Finally, the chapter does not pretend to cover all aspects of security which requires breadth and depth of knowledge and experience.

5 Background Security Concepts

5.1 Security Mindset

Security issues in designs have had a significant negative impact on our company's brand and reputation in the past and could again in the future. Security issues lead to a loss of customer trust and potential legal and financial implications. Fixing security issues consumes substantial engineering resources. Moreover, patches implemented to fix flaws may affect system performance adversely and cause inconvenience to customers.

Growing and adopting a security mindset is crucial for design and validation teams. This proactive approach involves anticipating, identifying, and mitigating potential security threats throughout the architecture, design and validation process.

5.1.1 Security awareness

Awareness of the importance of security in designs ensures that potential vulnerabilities are addressed before they can be exploited, thereby safeguarding the company's reputation and optimizing engineering resources. This can help engineers understand the need for security measures and become more vigilant in identifying potential security issues.

Learning is complementary to awareness, and it may spark curiosity, which in turn leads to investigation and discovery.

5.1.2 Security continuous learning

Education is another key aspect of a security mindset. It involves continuous learning and staying updated on the latest security trends and threats.

Learning helps understand past security issues and potential problems in similar systems. It can give opportunities to discover ways to architect or design an IP or system to be more robust. Discussions with other validation teams can help find new ways to stress the logic and to test mitigations.

Some educational resources include:

DDG LEVEL UP	goto/ddg.levelup.security
IPAS Security Belt Certification	goto/IPASbelt
Intel Security design principles	goto/securityprinciples
IPAS Tech Sharing Forum	goto/ipastechsharing
PRT Early Learning Forum	goto/PRTEarlyLearning
Physical Attacks Strategy, Policy, and Guidelines	goto/physicalattacks

5.1.3 Security processes and activities

Knowledge alone is ineffective. Applying and learning by doing is most effective to solidify the behaviors associated with building a strong security mindset.

Reviews, Security Development Lifecycle (SDL) tasks, and hackathons help the security of products and solidifies the team's security mindset. These activities provide practical experience and insights into potential security issues and their solutions. Hackathons provide a good way to step into an adversary's shoes and find ways to break the IP or system, which could find an unforeseen issue, ways to improve the design or the validation in the future.

5.1.4 Managing security

Management buy-in and support for security is critical to foster an effective security mindset. This includes allocating resources, encouraging and recognizing efforts, and enforcing requirements for teams that are resisting to adopting a security mindset.

Lastly, managing security requires you to learn and understand the balance or trade-off between too little and too much security. Too little security can leave the hardware vulnerable to attacks, while too much security can lead to unnecessary complexity, potential usability issues or engineering bandwidth waste. Therefore, a security mindset involves understanding the right balance to ensure robust quality designs and on-time delivery.

5.2 Security Concepts

This chapter covers a range of security concepts that are essential to understand to design and implement secure systems.

- **Asset:** a valuable resource (information or function) that needs to be protected in a system.
- **Attackers/Adversaries:** individual or group that seeks to exploit vulnerabilities to compromise the integrity, availability, or confidentiality of a system. Intel's IPAS Security Architecture Forum (SAFE) defines a specific classification of adversaries.
- **Attack point:** specific vulnerability or weakness in a system that can be exploited by an attacker to gain unauthorized access or cause damage.
- **Attack vector:** entry point to a vulnerable system that an attacker may exploit.
- **Threat modeling:** a structured approach to identifying, evaluating, and prioritizing potential security threats to a system.
- **Mitigations:** techniques or measures that are used to reduce the potential impact or severity of a security vulnerability or threat on a hardware device or system. Some examples include encryption, access controls, segmentation/partitioning, secure boot, secure update mechanism, integrity protection, side-channel protection.
- **Cryptography:** the use of cyphers and other cryptographic techniques to protect sensitive data and communications.
- **Access Control:** the processes and logic used to regulate who can access specific resources in the system.
- **Integrity detection and protection:** protecting the integrity of data and systems from unauthorized modifications.
- **Classes of attacks and impact:** different types of security threats, such as physical attack, firmware replacement, memory attacks, access control bypass, side-channel, fault injection.
- **Information flow:** the movement of signal value (data and information) through a system, including how it is stored, processed, and transmitted.
- **Security Objectives:** specific goals and targets that aim to maintain the confidentiality, integrity, and availability of a system's functionality or data.
- **Security intent validation:** process of ensuring that the security features and controls are correctly identified, defined, and implemented. It involves validating the functional correctness of the controls, ensuring the system is free from defects that could lead to vulnerabilities, and ensuring the system's ability to withstand expected attacks within the defined scope.
- **Hackathons :** collaborative event where participants simulate attacks on an IP or system to identify vulnerabilities. Traditionally done on live systems, pre-silicon hackathons are actually tabletop exercises. Findings from hackathons help devise testing plan or guide the development of mitigating solutions.
- **Red Teaming:** proactive approach where engineers mimic the behavior of potential attackers to test the effectiveness of a system's security measures. In pre-silicon validation, this is done by creating test scenarios that reproduce potential attacks or create unexpected events such as fault injection or interfaces fuzzing.
- **Security Development Lifecycle (SDL or SDLC):** a set of security practices and guidelines that should be followed throughout the product development process to ensure that security is integrated into the design and implementation of a system. It also provides guidelines on

reviews and necessary testing for security assurance. Intel provides a framework, a set of tools and processes that are defining SDL (goto/sdle)

- **Weakness:** flaw or bug in a system's definition, design, implementation, or operations that could be exploited to compromise the system's security.
- **Vulnerability:** a weakness that can be exploited by an attacker to violate a system's intended security objectives
- **Exploit:** method or technique, piece of software, data, or sequence of commands that takes advantage of a vulnerability in order to cause unintended behavior or gain unauthorized access to a system.
- **CWE:** CWEs refer to the Common Weakness Enumeration of potential security vulnerabilities and issues specifically related to the design and architecture of hardware or software components (<https://cwe.mitre.org/>).
- **CVE:** CVEs refer to the Common Vulnerabilities and Exposures identified in hardware or software components that could potentially be exploited to compromise the security of a system (<https://cve.mitre.org/>).
- **RAVE:** The Rational Analysis of Vulnerabilities and Exposures (RAVE) is a methodology to assist Intel in classifying security vulnerabilities in Intel supported pre and post release products, solutions, or offerings (<https://goto/rave>)
- **PSIRT:** Product Security Incident Response Team (PSIRT) is a group within an organization responsible for managing the response to security vulnerabilities identified in the organization's products. (<https://goto/psirt>)
- **RAS:** RAS stands for Reliability, Availability, and Serviceability, which are system design attributes that measure the ability of a system to operate correctly, remain accessible for use, and be repaired or updated without significant downtime. RAS and security are closely related. A system that is reliable, available, and serviceable is likely to be more secure. Conversely, a system that lacks in these areas may be more vulnerable to security threats.
- **Trusted Computing Base:** (aka TCB) set of all hardware, firmware, and software components in a system that are essential to its security. TCB is responsible for enforcing system-wide security policies. Intel platform TCB comprises the components that are critical to meeting Intel's platform security objectives. Platform TCB components include the secure engines, processor hardware, processor microcode, system firmware, BIOS settings, and platform software (PSW).

5.3 Examples of security functions

Security functions are a combination of Hardware, Software, Firmware and Physical package attributes responsible of enforcing security policies and supporting the isolation of resources in a system (data, code, functions). Hardware security primitives are specialized functional blocks that play an important role in ensuring trust, integrity, and authenticity of the system.

Features that have for goal to protect information and regulate access include:

- **Crypto primitives:** smallest functional blocks that implement fundamental cryptographic operations, such as encryption, decryption, hashing, digital signatures and Message Authentication Codes (MACs)

- **Crypto accelerators** and Hardware Security Module (HSM): dedicated cryptographic processors that provides secure key storage, key management, and cryptographic operations.
- **Trusted Platform Module** (TPM): a microcontroller that provides hardware-based cryptographic functions and secure storage of keys and other sensitive data. Security Engines
- **Authenticated Code Module** (ACM): An Intel signed executable code, which serves a particular - usually security-related - purpose. ACMs execute in an isolated environment for only a short time, performing a specific security-sensitive function and then exit. These modules execute within an area protected by the CPU in a known good configuration. Intel digitally signs ACMs and the CPU rejects any module not appropriately signed. The verification key used by the CPU is contained or protected by the hardware and is considered part of Intel TCB.
- **Secure enclaves:** a separate processor or thread within a system responsible for secure boot, secure storage, and cryptographic operations. Example: Intel SGX
- **Random number generators (RNG):** hardware blocks that generate (pseudo or truly) random numbers by measuring physical phenomena, such as thermal noise or radioactive decay. RNGs can be used to generate cryptographic keys that are unpredictable and resistant to attacks. A **DRNG** (Deterministic Random Number Generator) produces a predictable sequence of numbers, while a **TRNG** (True Random Number Generator) generates numbers from a truly random physical process.
- **Physically unclonable functions (PUFs):** hardware feature that leverages physical or electrical process variation characteristics of a design to create a unique identifier per part. A PUF can be used in security applications to uniquely identify individual devices or dies and help authentication, encryption and general binding of off-chip assets specific to that part.
- **Key generation and derivation blocks:** Key generation (KG) refers to a function creating new keys. Key derivation refers to creating new keys from existing ones
- **Key management unit:** (KMU) component responsible for handling and managing cryptographic keys in a secure manner within a system.
- **Access control:** function that ensures that only authorized requests on an interface are granted access. Access control blocks may be configurable or controllable to allow access policies to change over time.
- **Identity downgrade:** function that diminishes the access level or rights of a process or an individual making a request.
- **Memory protection:** features manage accesses to memory by dividing the memory into segments and assigning access permissions to each segment. Some examples of implementations include Memory Management Units (MMU) and Memory Protection Units (MPU). Another memory protection technique is called address space layout randomization (ASLR)
- **Integrity functions:** There are various integrity primitives: checksums, simple form of redundancy, Cyclic Redundancy Checks (CRCs), a more advanced form of redundancy check that can detect more types of data corruption, and HMAC (Hash-based Message Authentication Code), a specific type of message authentication code involving a cryptographic hash function and a secret cryptographic key.

- **Clock and Power monitors:** clock and voltage monitors are used to detect and prevent physical attacks on clock and power of a system.
- **MAC:** A Message Authentication Code (MAC) is a cryptographically generated secret code attached to a message, ensuring both the sender's identity and the message's integrity.

Some safety and reliability mechanisms can contribute to security by ensuring data integrity and preventing corruption:

- **Redundancy:** duplication of critical components or functions to increase system reliability. In hardware can be implemented through Dual or Triple Modular Redundancy (DMR/TMR) where identical components perform the same operation. It can also be achieved through RAID (Redundant Array of Independent Disks) for data storage, or through multiple power supplies or network paths to ensure system availability.
- **ECC:** (Error Correction Code) method used to detect and correct errors in data, ensuring data integrity and reliability. Usually implemented on transmission buses and in memory arrays.
- **Parity:** Error detection technique where an extra bit is added to each data byte

5.4 Examples of secure flows in a system

- Secret metal keys read by Firmware (FW)
- Secret keys download from Fuse to FW/SRAM/Register
- Random number read by FW/SW
- Memory encryption key setup
- Memory encryption flow
- Storage encryption key setup
- Storage encryption flow
- Key exchange protocol (including die-to-die)
- Secure tunneling (including die-to-die)
- Protected media replay
- Microcode patching
- Secure FW download and authentication
- Memory Zeroing
- Secure Debug
- TCB verification
- Remote attestation
- Software Guard eXtensions (SGX) flows
- Trust Domain eXtensions (TDX) flows

6 Security Validation

6.1 Security – functional validation

Functional validation of security involves ensuring that the security architecture requirements of the system are implemented correctly. This means that functions supporting security objectives are bug-free according to the specifications and configured correctly.

This section describes the validation of security functions, security features and the validation of secure flows in a system. Threats and Mitigation defined during architectural reviews and documents need to be exercised towards end-to-end security goal.

6.1.1 Cross-domain validation and security

Security validation needs to take into consideration the various operational modes of the system and security properties need to be upheld:

- At each stage of boot and reset
- Throughout power saving flows
- Across the various supported DFX flows
- Before production and during HVM
- During and after system updates and patching

This chapter will not cover all scenarios and domains, however some examples might mention cross-domain validation.

6.1.2 Validating crypto blocks, RNGs, PUFs

Due to the criticality and the complex validation process of some security functions this document will not cover such requirements. SAFE and SDL have specific implementation and validation methodologies for cryptographic functions, random number generators and physically unclonable functions.

The independent development of cryptographic algorithms should be strictly avoided. Established cryptographic algorithms demonstrably provide superior security assurances, reduce implementation complexities, and minimize long-term maintenance burdens.

6.1.3 Validating access control

6.1.3.1 Generic overview of access control

Access control are ways of limiting access to a system or subsystem, to a function, to logical/virtual resources or to physical resources. Access control usually consists of three stages:

- **Identification:** act of indicating a person or component's identity. Producing unique distinguishable credentials. The identification may contain information about the privilege of the requestor. Example of identification: IOSF SAI (see next chapter), portID, AMBA

AxID or AxPROT, process ID. Some control signal may not be designated as access control in the specification; however they might be used to allow or deny access.

- **Authentication:** process of verifying and validating identity provided by the requestor and ensuring that the requestor is the one that produced the credentials. This is not very common in standard hardware interface protocol and tend to be at a higher level of abstraction.
- **Authorization (or access):** function of 1) Specifying access rights and privileges to resources (policy definition). Policies can be static or dynamic (administration) 2) Granting or denying access to resources based on identity (policy enforcement). Access may be granted only after identification and authentication.

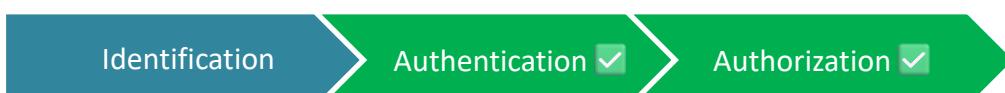


For more details on access control, please refer to the Level 2 Access Control Module in the Security Level Up.

6.1.3.2 Validating access control function

Validating access control consist of creating the right stimuli to assess that access is granted only with identity specified by the access right policy. If authentication is part of the access control, then it needs to be conclusive that the identity is valid before moving the authorization phase.

Positive Testing (Access Granted): Positive testing ensures that with a valid identity or privilege in the access right policy, access is granted.



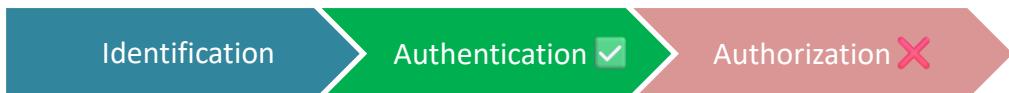
Example of access control: In a system with multiple cores (8) and other IPs, the destination IP can only be accessed by a secure root process load or store access initiated by a core. Cores are identified with AxID value set from 0 to 7. The system architecture does not allow non-core IPs to send root/privilege request: only a core can be identified as root. The destination IP does not permit processes to execute from its memory space (fetching/instruction requests), does not allow non-secure process requests, nor non-root access (user/app request). (AxPROT identifies privilege level. See section A4.7 “Access permissions” of ACE/AXI specifications.)

Example of access granted: Core 0 initiate a store operation from a secure root process:

- **Identification:** AWPROT[2:0] is set to 3'b001 (privilege, secure, data)
- **Authentication:** AWPROT[2]=1 is sent by a core AWID < 8'h08
- **Authorization:** control logic at receiving IP checks that incoming AWPROT==3'b001

The positive test should encompass the various combination of granted authorization: AWPROT[2:0]=3'b001 with various AWID between 0 and 7. Unless some internal error, the test should also check that the destination IP responded indicated there was no issue with the access with BRESP=2'b00. Similar testing should be performed on the read channel (ARPROT=3'b001 and ARID<8 with resulting RRESP=2'b00

Negative Testing (Access Denied – Authorization failed): Negative testing ensures that access is denied if an identity or privilege is not in the access right policy.



Example of access denied at authorization phase: Core is trying to load/store from a non-secure state.

- **Identification:** AWPROT[2:0] is set to 3'b011 (privilege, non-secure, data)
- **Authentication:** AWPROT[2]=1 is sent by a core AWID < 8'h08 ✓
- **Authorization:** control logic at receiving IP checks that incoming AWPROT==3'b001 ✗

The negative test should encompass the various combination of failing authorization: AWPROT[2:0] set to 3'b?1?, 3'b1??, 3'b??0. It may randomize the AWID between 0 and 7. The test should also check that the destination IP responded indicated that access was denied with BRESP=2`b10. Similar testing should be performed on the read channel.

If the policy can be changed dynamically, it is important to validate that updating the policy works correctly. For example, the IP might start at reset with a laxist policy where any AxPROT get access. It is important to test the various combinations of policy values and incoming identity.

Negative Testing (Access Denied – Authentication failed): Negative testing ensures that access is denied if an identity or privilege is invalid or if the id cannot be validated as a valid credential.



Example of access denied: IP can only be accessed by a secure root process data access (no fetching, no non-secure request, no user request). See section A4.7 “Access permissions” of ACE/AXI specifications.

- **Identification:** AWPROT[2:0] is set to 3'b011 (privilege, non-secure, data)
- **Authentication:** AWPROT[2]=1 is sent by no sent by a core AWID > 8'h07 ✗
- **Authorization:** with failed authentication, control logic denies access without checking AWPROT ✗

The negative test should encompass the various combination of failing authentication: any random AWID>7. AxPROT[2:0] can be set to any value – especially 3'b001. The test should also check that the destination IP responded indicated that access was denied with BRESP=2`b10. Similar testing should be performed on the read channel.

6.1.3.3 Reproducing attacks relevant to access control

The following attacks can be created to validate access control:

- **Identification**

- Identity forging (spoofing). An IP manages to generate an ID or privilege it is not supposed to.
- Privilege escalation: exploiting a vulnerability to change to a higher privilege level.
- Routing requests through an intermediary (confused deputy): An IP manages to trick another one to send a request on its behalf leveraging the tricked IP's privilege.
- **Authentication**
 - Replay attack: capture of transmitted authentication or access control information and its subsequent retransmission with the intent of producing an unauthorized effect or gaining unauthorized access
 - Brute force: uses trial-and-error to guess valid credentials
- **Authorization/Access**
 - Policy override: changing the policy to loosen access control so it allows access that otherwise would be denied
 - Brute force: uses trial-and-error to guess credentials in allow list

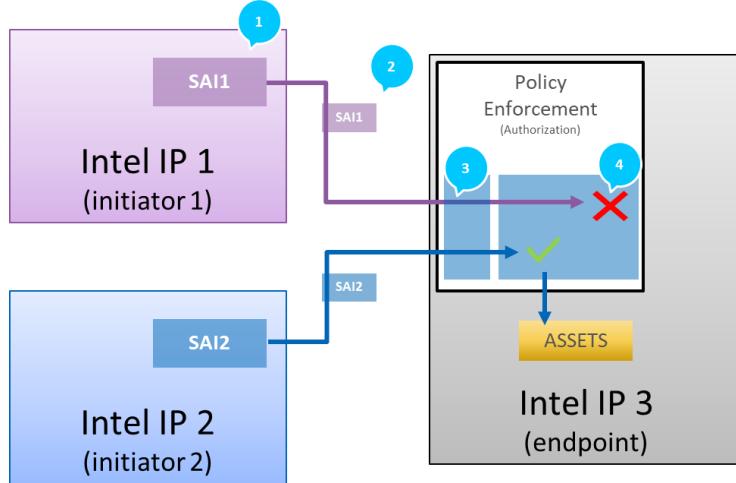
6.1.4 Validating control register access control on IOSF

This section outlines the functional validation of control registers functional blocks implementing SAI-based access control.

Intel On-chip System Fabric (IOSF) is the converged interface architecture defined to enable broad Intellectual Property (IP) reuse across Intel. IOSF supports SAI-based access control. SAI (Security Attributes of Initiator) is a unique security attribute carried in all transactions to enforce access control for protected resources in a system. SAI-based access control architecture consists of three main components:

- a. SAI generator - typically built inside the request initiator. The generated security attribute uniquely identifies the initiator.
- b. SoC's standard interconnection networks that carry and protect the integrity of the SAI attribute of the initiator, even across different interconnection domains. When crossing dielets, sometimes the SAI attribute may be downgraded for trust domain disaggregation.
- c. A policy enforcer at the target that assesses the initiator's credentials by referring its SAI value to the access control policy of the targeted asset.

[Please refer to the IOSF specifications for details.]



For more details on access control, please refer to the Level 2 SAI Access Control Module in the Security Level Up.

Intel's CREST (Converged REgister Specification Test) utility is a SystemVerilog test framework for control register validation. Under the Register Improvement Program (RIP), CREST utility provides a list of standard control register test algorithms in OVM/UVM sequences based on the RIP recommendations. [Please refer to the CREST utility page for details]

6.1.4.1 Validating SAI (Security Attribute of Initiators)

SAI is an immutable property of an access request initiator used for access control enforcement. This attribute must be generated by hardware often configured by RTL parameter and must be attached to every outgoing transaction. Unlike the transaction's source ID, SAI must not get transformed at bridges or routers separating interconnection domains, the SAI value persists while travelling from source to destination. There are cases where SAI may be downgraded, for example when crossing die-to-die boundary. The downgrading reduces the privilege of incoming requests so that they are not able to spoof a critical IP on the local die.

All fabric endpoints can only use the project assigned SAIs in constructing out-going transactions. At SoC-level, a checker monitors all transactions and verify the HW generated SAIs against its sources (initiators). This ensures correctness of the attributes in the design.

At IP-level, a simple self-check in UVM or any form of checker can be used to verify the hardware generated SAI values. To validate that an IP generates only expected SAI, validator can perform RTL code reviews or use formal. At system-level, SAINT-ReX can be used to automate the backtrace of opcodes and SAIs that can be generated by each instantiated endpoint. (<https://github.com/intel-innersource/applications.automation.client-hw.common.saint-rex>)

6.1.4.2 Validating control register access control policy

An access control policy determines which IP is trusted IP to read or write local control registers (CRs). Access control policies are also called SAI policies. Typically, the SAI policies are implemented in sets of configurable control/read/write registers. The policy enforcing logic uses the read and write policy registers to enforce the read and write permissions to the assets. The access control is self-referential. This means the control policy enforces access control on its own policy register. The SAI from the incoming transaction is used to index into the control/read/write policy registers and the value of the policy registers determine the access permission to the protected assets. A value of “1” indicates permission allowed and a “0” indicates permission denied. Access control policies may be sub-divided into separate access control for read (RAC), write (WAC) and updating policies (CP).

At IP-level, the policy registers and its associated access control logic need to be fully validated, this includes and any logic capable of bypassing access control.

Intel's Converged Register Specification Test (CREST) is a SystemVerilog test sequence the Saola RAL (Register Abstraction Layer) to perform basic control register validation. The CREST template provides comprehensive access control testing for both positive and negative conditions with all permitted, not permitted, and illegal SAIs.

Policies are set by parameters that may be overridden at integration. At SOC-level, validation teams need to ensure the IP’s access control policy default values are properly configured by hardware in the fully elaborated design.

6.1.4.3 Validating dynamic policy update (register locking by FW)

Besides the legacy use of lock-bits, FW may decide to update the CP/WAC/RAC policy registers to exclude some agents so they cannot reach certain registers or memory ranges. As an example, this method of locking is used to prevent some IPs from writing to BIOS-configurable registers once BIOS has completed. Integration validation should ensure FW has properly updated all necessary policy registers of the integrated IPs throughout different boot phases. In addition, if policy register values are reset during power state transitions, integration validation need to ensure those registers are saved and restored. Otherwise, the policy registers will be reset to their default values and not to the runtime values after power state exit.

6.1.4.4 Most common SAI bugs

This section lists the known bugs in the SAI access control:

- **SAI parameter misconfiguration:** SAI are critical as they define the role (hence the privilege) of a requestor. A misconfigured SAI can be due to poor specification or to an implementation bug. Improper setup of SAI may potentially allow unauthorized IPs to access sensitive information or functions in the system. It may also lead to functional issue when an IP is unable to access a resource it needs for proper functioning.
- **Incorrect policy specification:** security rules are defined erroneously during architecture, leaving gaps attackers can exploit.

- **Missing policy definition:** This is a specific case of incorrect policy specification when no specific instructions on how to handle a transaction or request, creating a security blind spot
- **Policy reset value misconfiguration:** A misconfigured SAI reset policy can be due to poor integration or to an implementation bug. If default reset values of a policy are set incorrectly at reset, it may potentially leave some parts of the system vulnerable. If access is needed but blocked due to a bad default value at reset, it may limit functionality and potential lead to the system to be bricked.
- **Missing policy registers:** Implementation might be optimized to skip the use of some policy registers. This may lead to limited configurability and flexibility
- **Master and shadow copy mismatches:** In some IPs and systems, hardware might implement shadow registers. Such shadow registers are not directly accessible, and their function is to ensure that various part of the logic get the same access control rules. Hardware function automatically updates the value in the shadow when the master value changes. Sometimes, a bug leads to master and shadow copy not matching. This could potentially lead to limited functionality, security risks, or crash.
- **Policy value misconfiguration from Save/Restore flow:** During IP power saving flows, values of the policy registers might be saved. When the IP gets restarted, the save/restore logic should reconfigure the SAI policy registers with the saved values. If the value before and after the power saving cycle are different – due to a functional issue or attack, the new policies might be wrong and lead to security risk or limit functionality.

The list is not exhaustive and represents commonly seen security issues. Most functional bugs in access control logic are not covered here.

6.1.5 Validating memory protection

6.1.5.1 MMU, MPU and ASLR validation

MMU: (Memory Management Unit) in a CPU or system is a hardware component that translates virtual addresses and safeguards memory access.

MPU: (Memory Protection Unit) is a hardware component that enforces granular permissions on memory access.

ASLR: (Address Space Layout Randomization) shuffles a program's memory layout to thwart attacks exploiting predictable memory locations.

MMU, MPU and ASLR validation are not covered in this document.

6.1.5.2 Validating memory access control policy

Access control to memory space is only applicable to stolen or isolated memory spaces, such as RootSpace-3 memory accesses. These memory ranges are not visible to the OS and allocated by BIOS and CSME before the “BIOS_DONE” stage. Accesses to these protected regions use physical addresses, must bypass IOMMU and rely on SAI value of the requesters to enforce access control. Only transactions with permitted SAIs are allowed to access the protected ranges. If a read

request is denied, the access control block returns a completion with Undefined Response (UR) to the requester. If a write request is denied, the request will be silently dropped.

IMR (Isolated Memory Range) architecture provides a generic and flexible implementation of stolen memory ranges that can be allocated to any agent. IMRs have a number of usages, e.g. secure storage of FW to be loaded by IPs and secure storage of IP local data.

At IP-level, the IMR base/mask and policy registers and its associated access control logic have to be fully validated. The test plan should cover all SAI values (allowed, denied and unexpected values).

At SOC-level, all integration teams need to ensure the IP's access control policy default values are properly configured by hardware. Need to exercise IMR configuration flow by BIOS/ESE/CSE. MCHECK enabling team need to ensure MCHECK able to access and verify all memory configuration includes IMRs.

6.1.5.3 Validating legacy stolen memory region

Some legacy stolen memory ranges still exist in client and are allocated by BIOS. For example: PAM, GSM, DSM, PRMRR, DPR, TPR, P2P ranges etc. Access control of these legacy ranges is implemented in a different way compared to IMR. They have no unified access control mechanism. For examples, only DISPLAY_SAI is granted accesses to GSM/DSM to prevent other agents access to media content. Only VTd_SAI is allowed to access PMR region where device page tables reside. On decoding of Peer-to-Peer regions, IOC (or MS2IOSF/HIOP on server) verifies Bus:Device:Function (B/D/F) numbers of the accesses. At IP-level, the configuration of these legacy stolen regions and their associated access control logics must be fully validated. The test plan should cover all positive and negative conditions (allowed, disallowed and unexpected SAI values, PortID, BDF, Root Space, etc).

These legacy stolen memory regions may belong to a different trust-boundary (TCB). Most of these protected regions are configured and locked by BIOS, hence they are within BIOS's TCB. One of the exceptions is PRMRR region which is higher than BIOS TCB. The PRMRR range is used to protect Intel core's XuCode in system memory from unauthorized accesses. Any BIOS and device access to this range is prohibited. The range is defined by a pair of PRMRR BASE/MASK registers. Microcode and Intel-authenticated MCHECK code module own the configuration and lock of the region. Hence, hardware and MCHECK need to ensure the integrity of the protected regions. At SOC-level, system boot with PRMRR setup with MCHECK patching is enabled and exercised.

6.1.5.4 Protecting the system from range overlaps

A system may have multiple protected memory ranges. Some are fixed like C6-SRAM range, some are illegal like the non-canonical range, some are configurable like APIC, SMRR and PRMRR.

A common attack vector of memory range overlap originates in the definition of ranges to define access privileges. For example, only SMM SW can read\write from SMM range. When memory ranges overlap, Access with a given privilege to access one of the memory ranges may manage to access the second range, that should have been protected according to the definition.

Another attack vector is to change the victim's behavior by overlapping a memory range with its memory. For example, a VMM attacker can attempt to overlap the VAPIC with the enclave SW memory causing the uCode to respond to the victim's activities.

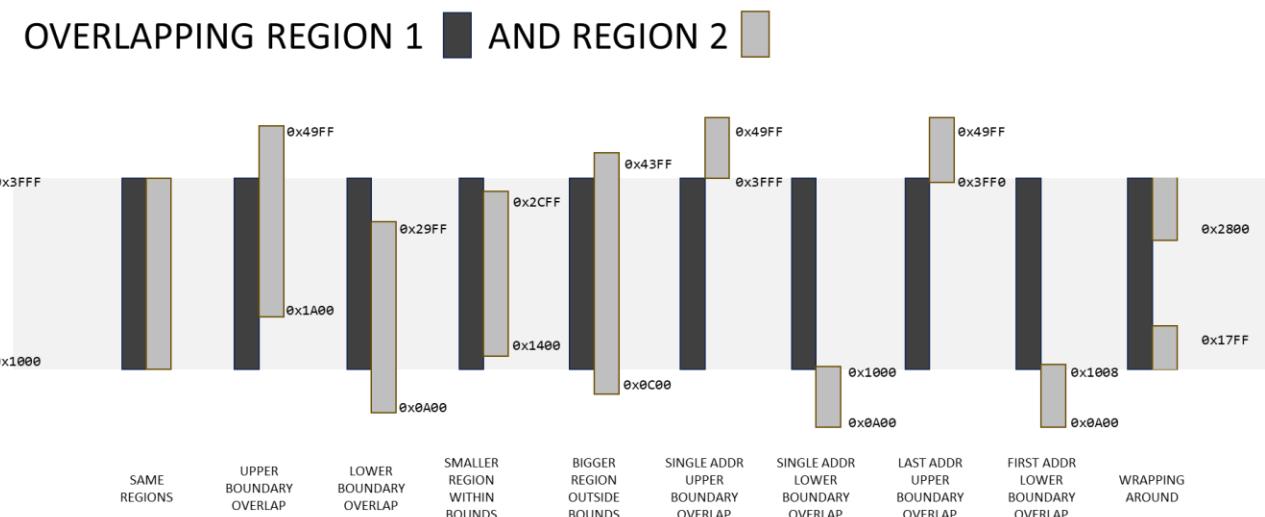
Another example is that on read from a protected range, SW will often read 0xFFFFFFFF instead of real data. The attacker can overlap a protected range with the victim's memory, causing it to read an injected value (0xFFFFFFFF) instead of the expected value from memory. The value can then poison the process that should have read a functional value and lead to privilege escalation, functional issues or crash.

These are examples of vulnerabilities that were found in our products.

There are two popular mitigation methods for range overlap:

- Have a function detect the overlap on configuration and reject the configuration.
 - Mind that the protection needs to be installed on both range configurations
 - Mind that the legacy protection code needs to be revised when a new range is introduced
- Detect the overlap on access (range registers in HW)
 - Pay close attention to state transitions. Identify the exact point where access privileges should be changed.

The figure illustrates the various combination of range overlap or memory access region overlap:



	REGION_1_BASE_ADDRESS	REGION_1_TOP_ADDRESS	REGION_1_RANGE
	> TOTAL_MEMBASEADDRESS + 1	< TOTAL_MEMTOPADDRESS -1	> 0
	REGION_2_BASE_ADDRESS	REGION_2_TOP_ADDRESS	REGION_2_RANGE
SAME REGIONS	= REGION_1_BASE_ADDRESS	= REGION_1_TOP_ADDRESS	= REGION_2_RANGE
UPPER BOUNDARY OVERLAP	> REGION_1_BASE_ADDRESS && < REGION_1_TOP_ADDRESS	> REGION_1_TOP_ADDRESS && ≤ TOTAL_MEM_TOP_ADDRESS	> REGION_1_TOP_ADDRESS - REGION_2_BASE_ADDRESS
LOWER BOUNDARY OVERLAP	< REGION_1_BASE_ADDRESS && ≥ TOTAL_MEM_BASE_ADDRESS	< REGION_1_TOP_ADDRESS	< REGION_1_TOP_ADDRESS - REGION_2_BASE_ADDRESS && > REGION_1_BASE_ADDRESS - REGION_2_BASE_ADDRESS
SMALLER REGION WITHIN BOUNDS	> REGION_1_BASE_ADDRESS	< REGION_1_TOP_ADDRESS	< REGION_1_TOP_ADDRESS - REGION_2_BASE_ADDRESS
BIGGER REGION OUTSIDE BOUNDS	< REGION_1_BASE_ADDRESS && ≥ TOTAL_MEM_BASE_ADDRESS	> REGION_1_TOP_ADDRESS && ≤ TOTAL_MEM_TOP_ADDRESS	> REGION_1_TOP_ADDRESS - REGION_2_BASE_ADDRESS
SINGLE ADDRESS UPPER BOUND OVERLAP	= REGION_1_TOP_ADDRESS	> REGION_1_TOP_ADDRESS && ≤ TOTAL_MEM_TOP_ADDRESS	≥ 1 && ≤ TOTAL_MEM_TOP_ADDRESS - REGION_1_TOP_ADDRESS
SINGLE ADDRESS LOWER BOUND OVERLAP	< REGION_1_BASE_ADDRESS && ≥ TOTAL_MEM_BASE_ADDRESS	= REGION_1_BASE_ADDRESS	≥ 1 && ≤ REGION_1_BASE_ADDRESS - TOTAL_MEM_BASE_ADDRESS
LAST ADDR UPPER BOUND OVERLAP	= REGION_1_TOP_ADDRESS - 1	> REGION_1_TOP_ADDRESS && ≤ TOTAL_MEM_TOP_ADDRESS	> 1 && < TOTAL_MEM_TOP_ADDRESS - REGION_1_TOP_ADDRESS
FIRST ADDRESS LOWER BOUND OVERLAP	< REGION_1_BASE_ADDRESS && ≥ TOTAL_MEM_BASE_ADDRESS	= REGION_1_BASE_ADDRESS + 1	&& < REGION_1_BASE_ADDRESS - TOTAL_MEM_BASE_ADDRESS
WRAPPING AROUND	> REGION_1_BASE_ADDRESS && < REGION_1_TOP_ADDRESS && > REGION_2_TOP_ADDRESS	> REGION_1_BASE_ADDRESS && < REGION_1_TOP_ADDRESS && < REGION_2_BASE_ADDRESS	< TOTAL_MEM_TOP - TOTAL_MEM_BASE_ADDRESS +1 && > TOTAL_MEM_TOP - REGION_2_BASE_ADDRESS + REGION_1_BASE_ADDRESS

6.1.5.5 Validating memory lock after system reset

If Trusted Execution Technology (TXT) is enabled, then following reset, there could be secrets in memory that must be erased before any unsecure entity can access them. Therefore, hardware should lock down memory paths and blocked all memory accesses by default. Microcode will not unlock memory via LTCTRLSTS register. Later, BIOS executes SCLEAN ACM to unlock core memory path, erase the memory content and unlock the device memory path. At SOC-level, system boot with TXT enabled, memory lock is exercised and verified.

6.1.5.6 Validating address decoding priority

Address decoding priority is necessary to ensure the integrity of special memory ranges such as fixed LT/BIOS/CRAB address spaces, stolen memory, IMR, MCHBAR, lockable and unlockable MMIO BARs. For an example, when core microcode reads secure boot policy from CSE during secure boot, the LT access can only be routed to CSE but not to other destination. All routing agents involved should decode the address properly and forward to the right destination in order to ensure the integrity of information. Since the enforcement of address decoding and routing is distributed in multiple IPs throughout the path, most of these conditions cannot be fully validated at IU/IP-level. At SoC-level, all sort of memory redirection attacks need to be checked.

6.2 Client SoC secure flows

This section covers a few secure flows in client SoC systems.

6.2.1 Validating secret download and random number read

Some platform security features such as microcode patching, MKTME, TSE, SGX, SPIRAL need to fetch related secret keys from other non-volatile entities such as fuses and metal keys in PLAs during boot phase or feature configuration. For an example, SGX and SPIRAL secrets are pulled and stored in DFX Aggregator before debug authentication window is closed and debug policy is determined. These keys are zeroed in DFx Aggregator when the debug policy is determined to be NOT “Security Locked”.

Early boot IPs such as DFx Aggregator reads the random numbers from fuse-DRNG (hardware random number generator) to generate ephemeral key for debug session. Meanwhile, Core microcode, system BIOS and OS may also fetch the random numbers from hardware random number generator (DRNG) for ephemeral key generation.

At SOC-level, all logical paths taken by secret (download and reading) require be fully validated for its data confidentiality and integrity.

6.2.2 Validating FW loading/patching

There are many firmware-based subsystems in client SOC such as Core microcode, ACP - Acode, Punit - Pcode and DMU - Dcode. Firmware executed from ROM needs ROM code patching for bug fixing and firmware runs on RAM required firmware loading before reset.

Hardware and microcode firmware of BigCore and Atom families supports a few ways of patching for microcode ROM and registers. They are Patch-In-Fuse to fix core reset microcode sequences, FIT patching before reset vector and BIOS wrmsr patching for MCHECK execution and Xcode loading.

CPU power management controllers’ firmware such as Pcode, Dcode and Acode which are executed from local SRAM have to be loaded before the microcontroller comes out of reset. In client, ESE supports the FW authentication and download of the Pcode/Dcode/Acode images.

At SOC-level, all flavors of microcode patching and firmware loading have to be fully validated for its data confidentiality and integrity.

6.2.3 Validating memory encryption

System memory (DRAM) encryption aims at mitigating the cold boot attack where the adversary freezes DRAM DIMMs and move to another system under control of the adversary to extract secrets from the frozen memory. With cold boot attack, the security researchers were able to extract the Bit locker key from the frozen memory and decrypt the hard drive of the computer under attack.

Total Memory Encryption (TME) provides a capability to encrypt entire system memory. Once BIOS enabled and locked TME, it encrypts all the data on external memory buses using AES-XTS algorithm. The encryption keys used for TME which generated with random numbers from DRNG are not accessible by software nor exposed on any external interfaces.

Multi-Key Total-Memory-Encryption (MKTME) builds on TME provides kernel or hypervisor the ability to assign encryption key at a page granularity. The MKTME supports a fixed number of encryption keys and system software can use each of the available keys for encrypting any page

of the memory. When MKTME enabled, the Core Paging or VTd MMU translated physical addresses consists of 4 bit KeyID that needs to be used for encryption/decryption at the CCE (memory encryption engine). KeyID is programmed in Page Tables by system software.

Note: In server, the number of KeyID is configurable (from 2 to 1024) and use from 1 to 10 bits. The KeyID is overloaded into the MSB of the system address space. In server, the memory security engine (MSE) plays a similar role to client's CCE and is responsible for memory encryption.

The main SOC components of MKTME feature are Core paging, IOMMU, KeyID in the address bus of fabrics, memory subsystem configuration and routing, MKTME KeyID MASK and Check in all routing agents, KeyID violations and handling, Core RD/WRMSR and new instructions to configure and activate MKTME. All these features need to be fully validated at its respective IP and Integration level.

At global level, the memory encryption configuration by system BIOS in conjunction with all flavors of memory configurations have to be fully validated.

6.2.4 Validating storage encryption

Storage or Non-Volatile Memory (NVM) encryption is a technology which protects information by converting it cryptographically before storing which cannot be deciphered easily by unauthorized personnels. The technology aims to mitigate physical attacks such as stolen computer where attacker can take out the storage device and plug them in another system to get whatever personal data in them.

Total Storage Encryption (TSE) is a new feature which encrypt external storage devices (NVMe) connected through PCIe. Intel is using XTS-AES encryption/decryption algorithm for its NVMe storage encryption which is commonly used by disk encryption. The encryption/decryption is implemented in IOCCE (IO Converged Crypto Engine).

While in DRAM encryption, memory physical address is used as initial value for AES-XTS tweak generation. NVMe uses Logical Block Address (LBA) which never exposes on PCIe TLP. Therefore, TSE uses a TSE Table memory structure to map between PCIe address and NVMe LBA. Unused TLP MSB address bits are repurposed to carry pointers to TSE Table in memory.

The main SOC components of TSE feature are IOCCE, IOC, TSE IMR, integrity of PCIe TLP address, TSE Table, memory subsystem configuration, IOCCE violations and handling, Core RD/WRMSR and new instructions to configure and activate TSE. All these features need to be fully validated at its respective IP and Integration level.

At global level, the storage encryption configuration and TSE Table setup in TSE IMR by system BIOS in conjunction with all flavors of memory configurations have to be fully validated.

6.2.5 Validating Secure Boot flows

There are many terms in the industry used in describing platform secure boot. The term Secure Boot is often used to refer to what is called as verified boot. In fact, secure boot can be achieved using either measured or verified boot.

6.2.5.1 Unsecured Boot

Platform boot executes the boot code and continues executing code to configure the platform and boot to the OS. There is no validation of the codes if they are correct or from an authorized source. There is no history, recording or audit log of the boot process after the platform boots. The users of these platforms make an assumption that the executed boot code is what the Platform Manufacturer intended, and the loaded OS is what the user intended to use.

6.2.5.2 Measured Boot

Similar to unsecured boot, measured boot executes code which boots the platform without verifying whether the code is correct or from an authorized source. However, measured boot records the boot sequence into a trusted entity which is resistant to tampering by any untrusted component during boot, typically this is accomplished with TPM services (CSE). The measurements start at the Root of Trust for Measurement (RTM) which is the beginning of the transitive chain of trust. If the root is untrusted, the entire chain is untrustworthy. In measured boot, there is a provable record that the platform booted correctly.

6.2.5.3 Verified Boot

Verified boot measures and verifies every step in the boot process to ensure the boot code is correct and from an authorized source prior to executing it. This determination starts at the Root of Trust for Verification (RTV). Like the RTM, if the RTV is untrusted, the entire chain is untrustworthy. In Verified boot there is no record of the boot process and no intrinsic proof that the platform booted correctly.

6.2.5.4 Trusted Boot (Measured + Verified)

Trusted boot combines the benefits of both Verified boot and Measured boot. While Verified boot focuses on ensuring the integrity of the boot process by verifying digital signatures of the boot codes, and Measured boot measures and logs the boot process integrity in the TPM.

To ensure a secure state throughout the entire platform operation, Trusted boot expands the chain of trust from the initial boot process through the entire system runtime where it extends the measurements to other critical operating system components, such as drivers, kernel modules, and system services, as they load and execute during runtime.

6.2.5.5 Validating Intel Boot Guard (aka Anchor Cove)

Boot Guard (Anchor Cove) technology is a hardware-based boot integrity protection technology which extends the Intel chain of trust via the Intel CPU microcode to include BIOS as the root of trust to any type of secure boots. Anchor Cove acts as a static RTM in Measured boot which performs the initial measurement of the platform's Initial Boot Block (IBB) into the TPM. Anchor Cove provides the RTV for Verified boot which cryptographically verifies the Initial Boot Block (IBB) of BIOS using the platform Boot Policy Key.

In legacy boot at the end CPU reset sequence, CPU microcode unconditionally fetches the reset vector and begins executing BIOS boot block. When Anchor Cove enabled, CPU microcode locates the Anchor Cove ACM via the BIOS FIT (FW Interface Table) and loads the ACM into cache. Then CPU microcode authenticates the ACM before starting to execute it.

The Anchor Cove ACM locates and verifies the Boot Policy Manifest which contains information such as pointer to BIOS IBB, Boot types (verified or measured or both) and instructions to set up VT-d protection. Anchor Cove ACM then loads IBB into the LLC and performs verification or/and measurement before it jumps to the IBB entry point in the LLC.

With Boot Guard technology, platform manufacturers can now define and enforce platform boot policies. Any invocation of unauthorized or tampered boot block will trigger the platform protection and stall the system boot.

Ingredients: CSME Fuse and FW, Boot policy, LT cycle and routing, Pcode/Dcode FW

6.2.6 Validating BIOS Guard (Platform FW Armoring Technology)

SPI flash memory on a platform is partitioned for several different flash consumers within a system. Typically, these include BIOS, Intel Manageability Engine, and Intel Gigabit Ethernet controller, etc. Some mobile systems used a mobile embedded controller (EC) with a discrete flash part connected via a private SPI bus. The integrity, and sometime confidentiality, of the flash memory has to be ensured in order to prevent malicious overwritten and causing a persistent tampering.

A protection mechanism exists in the PCH that prevents any write or erase cycles from being sent to the SPI flash memory unless the part is opened for writes via a CPU special cycle in response to software writing an associated MSR. Historically, this MSR has been writeable from SMM-mode only. With Bios Guard Technology (PFAT), the MSR used to generate the flash open/close special cycles is only writeable from NP-PPPE mode which can only be invoked in SMM-mode. This has further shrunk the possible attack surface and only allows the flash operations in a more defensible environment. The PFAT feature encompasses the following platform components:

- CPU CRAM mode execution (Non-persistent PPPE)
- Secure logical CPU rendezvous, quiesce and wakeup in SMM
- Flash Open/Close special cycles WRMSR only allowed in protected CRAM mode

6.2.7 Validating Intel Trusted Execution Environment (TEE)

Trusted Execution Environment (TEE) is a secure and isolated execution environment within a computer system that provides a higher level of security to preserve confidentiality and integrity of sensitive operations and data. Integrity requirement prevents code and data in the TEE from being replaced or modified by unauthorized entities. While confidentiality prevents sensitive data (assets) in the TEE from being exposed to unauthorized viewers. TEE is often implemented using a combination of hardware and software components to ensure that certain processes can run in a trusted and protected manner, isolated from the rest of the system's software and hardware components.

6.2.7.1 Intel Trusted Execution Technology (TXT) with Safer Mode Extension (SMX)

TXT is a set of hardware extensions to Intel processors and chipsets which enables a trusted environment where applications can run within their own space without interference from other software or devices in the system, hence it can protect against software-based attacks and preserve the confidentiality and integrity of execution and runtime data in the system.

As mentioned in the previous topic, Static Root of Trust for Measurement (SRTM) executes on each platform reset creates a chain of trust from reset to the measured environment. However, maintaining a chain of trust for a lengthy time or entire operational flow may be challenging as the system may need to support unknown software entities. To address this issue, TXT provides the Dynamic Root of Trust for Measurement (DRTM) where the launch of a Measured Launched Environment (MLE) can occur at any time without resorting to a platform reset.

To establish an MLE, the system uses the SMX GETSEC[SENTER] instruction to invoke an ACM. GETSEC[SENTER] broadcasts messages to the chipset and other logical processors in the platform. In response, the logical processors perform basic cleanup and wait to join the MLE. After all logical processors are in the wait state, the initiating processor loads, authenticates and executes the ACM. The ACM checks the platform chipset and processor configurations and then measures and launches the MLE. The MLE initializes a secure system configuration and then issues a new SMX instruction that wakes up the other logical processors and brings them into the measured environment. At some point later, it is possible for the MLE to exit and then be launched again, without issuing a system reset.

Ingredients: SMX, Microcode, Pcode FW, LT cycle, TPM configuration and routing.

6.2.8 Validating Software Guard Extensions (SGX)

Intel's Software Guard Extensions (SGX) is an extension to the Intel PPPE architecture which aims to provide integrity and confidentiality assurance to any secure computation performed on a computer. These extensions allow software to instantiate a protected software container and executes in an enclave. An enclave is a hardware-based encrypted memory area which provides confidentiality and integrity of the execution and data. It shares the same space as Intel PPPE code modules. Attempted accesses to the enclave memory area from software not resident in the enclave are prevented even from privileged software such as BIOS, VMM, hypervisor or operating systems.

With SGX services, an enclave can prove its identity to a remote party and be securely provisioned with keys and credentials. The enclave can also request a platform specific key that it can use to protect keys and data that it wishes to store outside of the enclave. SGX is designed to be useful for implementing secure remote computation, secure web browsing, and digital rights management (DRM).

SGX TCB consists of SOC hardware, CPU microcode, Punit-firmware, CCE, MCHECK, Xucode.

At SOC-level, SGX platform ingredients such as BIOS/MCHECK/Xucode modules have to be fully enabled and validated for its confidentiality and integrity.

6.2.9 Validating Trusted Domain Execution (TDX) based on MKTME

Trust Domain Extensions (TDX) extends the CPU Virtual Machine Extension (VMX) architecture to enable a new hardware-isolated Virtual Machines (VMs) called Trust Domains (TDs). The extended VMX architecture introduced a new Secure Arbitration Mode (SEAM). SEAM VMX root operation is designed to host a CPU-attested TDX module to manage TD VMs who builds, tears down and starts execution of TD VMs. TD VMs are launched by TDX module and operate in SEAM VMX non-root mode.

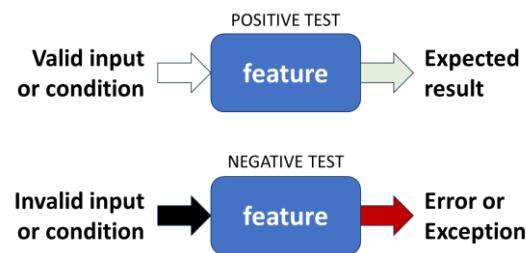
TDX module and TD VMs are isolated from the Virtual-Machine Manager (VMM), hypervisor and other non-TD software on the host platform. Only the TDX module executing inside the SEAM-memory range are allowed to access to SEAM-memory range. All other software accesses and direct-memory access (DMA) from devices to this memory range are prohibited.

Meanwhile, the TD VM's integrity protection is based on an extension of MKTME support. MKTME key indexes can be partitioned into a set of private KeyIDs and a set of Shared KeyID(s). Shared KeyIDs are managed by VMM or hypervisor, but only the TDX module in SEAM mode can use CPU PCONFIG instruction to program and manage private KeyIDs. Only TDX module and TD VMs are allowed to access memory using private KeyIDs.

Intel TDX aims to enhance confidential computing by helping protect TD VMs from a broad range of software attacks and helps to reduce the TD Trusted Computing Base (TCB). TDX also protects against physical attacks such as DRAM cold-boot attacks and DRAM interface analysis.

6.3 Negative testing

There are two main strategies when validating a feature: positive testing and negative testing. Positive testing determines that the logic works as expected. If an error or the wrong behavior is encountered during positive testing the test fails. Negative testing ensures that the logic can gracefully handle an incorrect input or an unexpected behavior on the inputs. The purpose of negative testing is to ensure such situation are detected so that there is no crash or hang. Expected behavior is defined in the specification. Some behaviors that are not defined in specification and could be uncovered during negative testing.



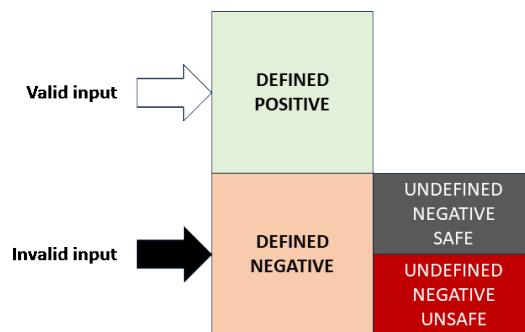
For example, if "When A and B Then C" is a positive testing scenario, then a negative testing scenario could be "When not A and/or not B Then not C".

Positive conditions refer to testing scenarios where valid inputs are provided to the system, and the expected behavior is for the system to function correctly. Negative conditions refer to testing scenarios where invalid inputs are provided to the system, and the expected behavior is for the system to handle these inputs gracefully and without errors.

Specified behavior for positive conditions refers to the expected behavior of the system when valid inputs are provided, while specified behavior for negative conditions refers to the expected behavior of the system when invalid inputs are provided. Negative conditions may be defined in the specifications to be handled. For example, with the IOSF protocol, it sometimes means either ignoring/dropping the request, or responding with an UR opcode.

Unspecified negative conditions with unsafe outcomes refer to testing scenarios where invalid inputs are provided that could cause harm to the system or data. These scenarios may not be explicitly defined in the testing requirements, but they are still important to test to ensure the safety and security of the system.

Unspecified negative conditions with safe outcomes refer to testing scenarios where invalid inputs are provided that do not cause harm to the system or data. These scenarios may not be explicitly defined in the testing requirements, but they can still provide valuable insights into the behavior of the system under unexpected conditions.



Sometimes, negative conditions are not covered in the specifications due to an overlook or because such conditions are not expected to reach or propagate to the feature or interface. Adversaries tend to search in specifications or use trial and error to discover such exploitable misses.

6.3.1 Negative Space classification for validation

Categories	Definition
Positive	<p>Intended design behaviors to serve certain purposes or functionalities.</p> <p>Verify the correctness of functionalities of a design. Perform appropriate actions using valid operation and operands (opcode/data/packet/signals)</p> <p>Defined in Specifications (HAS/MAS/FAS/etc.)</p>
Grade 1 Negative	<p>Intended design behaviors for expected invalid/erroneous conditions occur within a design. Design acts gracefully for correction and prevention of any functional/security breakdowns.</p> <p>Validation performs appropriate actions using expected invalid/erroneous conditions (invalid opcode/data/packet/signals) and verify the handling of the incidents.</p> <p>Errors and handlings defined in Specifications (HAS/MAS/FAS/etc.)</p>

Grade 2 Negative	<p>Conditions which are out of spec (Undefined conditions/behaviors, don't care, Specification misses/gaps/bugs, etc.)</p> <p>Can be defined with mutations and extensions of positive and grade 1 negative space.</p> <p>Perform to break a design using invalid, undefined, illegal or unacceptable commands (operations, operands, actions...).</p> <p>Techniques: Directed tests, Fuzzing, FPV, etc.</p>
Grade 3 Negative	<p>Conditions which are way out of spec and involve more than one negative conditions.</p> <p>Perform to penetrate a design with threat models (assets and attack vectors).</p> <p>Or extend grade 2 negative findings to prove a potential exploit.</p> <p>Or reproduce PRT-grade issues.</p> <p>Techniques: Directed tests, Fuzzing, FPV, SPV, etc.</p>

6.3.2 Examples of Positive and Negative Testing

Positive	Negative
Control Register attribute validation (Lock-bit, Read/Write-only, Read/Write-once, Block-on-Debug, etc.)	Invalid control register accesses (Invalid opcodes, address, attributes)
Primary-to-Sideband Forwarding (opcodes, addresses, attributes, SAI, etc.)	Unsupported Primary-to-Sideband routing (opcodes, addresses, attributes, whitelist etc.)
SRAM MBIST and Zeroing	SRAM MBIST and ECC error handling
Cryptographic primitives and configuration (Encryption, HASH, MAC, key exchange)	Handling of Cryptographic errors

6.4 Security assurance

Another part of product security validation is gaining confidence that the security features, the architecture, and the inherent properties of the system are sufficient to be resilient against security failures and attacks.

This section describes the various activities outside of functional validation that ensure that security functions are sufficient. The adversary in scope will provide some guidelines on what mitigations measures should be in place. It is good to keep in mind that an IP could be over-engineered.

6.4.1 Security considerations for IP Integration

Most systems are built by putting together a set of IPs. It is important to understand that a vulnerability in an IP could undermine the overall system security. Such vulnerability could come from 1) a security bug in the IP 2) a misconfiguration of the IP during integration 3) a misconfiguration of the IP during logical operations.

To reduce the risk of security bug in an integrated IP, the security architect ensures that the IP has gone through SDL. For Intel IP/subsystem that integrate commercial off-the-shelf (COTS) third party IPs (3PIP), the team should ensure that the 3PIP has an associated security risk assessment (SRA). IPAS team keeps track of 3PIP SRAs.

6.4.2 Threat modeling and attack reproduction

Threat modeling is a set of proactive activities to help identify the potential attack vectors that may cause harm to a system or its assets. During the threat modeling process, potential threats such as structural vulnerabilities, functional deficiencies or the absence of appropriate safeguards can be identified, enumerated, and risk graded. Identified threats are evaluated against existing countermeasures and mitigations. Such mitigations should be in the functional specifications.

Reproducing past known attacks will provide confidence that the system is not vulnerable. A list of past security PRT bugs should be readily available within Intel (<https://goto/prtdashboard>) and periodically reviewed to avoid repeating mistakes. It also is good practice to periodically review CVEs and news to understand vulnerabilities in our competition's products.

Attack re-creation can uncover new problems. Creating scenarios that reproduce potential attacks can help uncover the lack of correct mitigations. To help decide on scenarios to create, a validator will decide on:

- 1) The asset under attack
- 2) Which property to compromise (confidentiality, integrity, availability, max persistence ...)
- 3) The vector of attack (logical interface, physical error injection, ...). Using these three elements, the validator can create property, an assertion, a simulation test, a checker, or a combination of.

Types of attacks to create:

- Interface attacks: replay attack, man in the middle, dropped packets, spoofing, scanning,
...
- Physical attacks: fault injection, random reset, ...
- Misconfiguration: this include static misconfiguration (RTL define, parameters,...) and dynamic misconfiguration (control registers).

6.4.3 Static configuration input sanitization

Some IP blocks' RTL can be configured with `define or parameters. It is important to understand how incorrectly setting the static configurations of the IP could lead to the logic to be elaborated in a way that can lead to a security weakness or vulnerability.

A parameter or a `define could decide if access control is part of the design or not. Often a parameter becomes the default value of an access control policy register, therefore the value at integration is critical to ensure proper access quarantine during operations at system level. Such a parameter should be tested during IP level validation by having the testbench change the value.

It is sometimes difficult to know if a combination of defines and parameters could create a weakness. For example: an elegant design would have a single parameter deciding on the size of a vector, other parameters such as MSB or address encoding could be derived from the first one. However, the same IP could be coded in a way that requires the three parameters to be set at integration. If one of the parameters is set incorrectly the design might still elaborate but there's no way to know how the IP will operate. While this could fall under functional validation, it is important for the validation team to understand that `defines and parameters may have negative impact on security.

6.4.4 HW Fuzzing

Fuzzing, also known as fuzz testing, is an automated testing method that provides invalid, malformed, or unexpected input into an IP or system. Fuzzing is very useful in finding conditions or scenario that were not thought during the specification process.

Fuzzing is usually more common in software testing. It is done during development it can be perform at program level or API level. In real products fuzzing can be done from one process to another one and from one machine to another using software protocols. Software fuzzing can find hardware issues however limitations exist.

Hardware fuzzing is done from a hardware interface. It can target software or hardware. It might require custom-built hardware to connect to the desired port.

Fuzzing in simulation can be complicated to set up and the ROI might be limited. In general, it is recommended to do fuzzing only on interfaces that are at risk; such interfaces are:

- 1) Software or firmware driven interfaces: ports that can be driven by FW of a microcontroller. This would mean register accesses, mailbox communications or core accesses to the system.
- 2) External facing interfaces: digital portions of PHYs, analog PHYs, communication from external pins to internal IPs
- 3) HW to HW interfaces: critical interfaces or interfaces to security-critical blocks (security functions, core inputs, power management IPs), or bridge IPs between protocols

Because there is a lot of different types of interfaces and protocols it is important to have a common framework to understand how to fuzz hardware. In general protocol signals are logically divided into control and data planes. Fuzzing can be done at various dimensions such as: 1) content fuzzing (control fields or the data fields), and 2) protocol shape fuzzing (packet shapes and length, signal ordering, extra packets or signaling).

Fuzzing can uncover functional issues and security issues:

- Potential functional findings (Uncover logical and physical defects)
 - o Detect race conditions and deadlocks in logic
 - o Detect FSM or control encoding/decoding errors

- Detect unspecified or undocumented logic blocks or functions
- Detect unmapped memory or memory aliasing
- Potential security findings (Identify attack vector and quantify robustness of logic)
 - Lead to hang or crash (DoS), or to Access Control Bypass (Elevation of Privilege / Data Exfiltration/Corruption)
 - Identify other stability, availability, or DoS-resilience issues
 - Assess risk of reverse-engineering or logic fingerprinting
- HW fuzzing can find SW bugs

Before deciding on doing fuzzing, it is important to do scoping and have the right strategy. Decide which interface to drive, at what entry point. Decide on a generator and how it will connect to the test environment.

Since fuzzing will lead to unexpected behavior, functional checkers and assertions should be disabled. This makes analyzing results difficult. You must have a clear understanding on what to monitor and how to analyze results.

6.4.5 Structural hazards and logical side-channels

A pipelined microprocessor design is a concept in computer architecture where the execution of instructions is broken down into several stages such as instruction fetch, decode, execution, memory access and write back. Each stage operates concurrently on different instructions. This technique aims to improve the overall performance of the processor by increasing its throughput and allowing multiple instructions to be in various stages of execution simultaneously. However, the parallelism in compact pipelining may cause unexpected resource conflicts.

There are three main types of hazards in pipelined processors. **Data Hazards** occur when there is a dependency between instructions that causes them to be executed out of order or stalled in the pipeline. **Control Hazards** arise when the pipeline must stall or make incorrect decisions due to a change in the control flow, such as a branch instruction. This can lead to wasted cycles while the correct path of execution is determined. **Structural Hazards** occur when there is a physical limitation in the hardware resources that prevents the simultaneous execution of certain instructions. For example, if two instructions require access to an arithmetic or memory unit at the same pipeline stage, a structural hazard arises because only one instruction can use that resource at a time. Structural hazards can lead to many design issues such as pipeline stall or deadlock, performance and quality degradation, unpredicted behaviors. Most importantly, structural hazards can potentially lead to security vulnerabilities. For example, an attacker could potentially manipulate the timing and resource access patterns to gain unauthorized access to sensitive data or execute malicious code.

Logical side-channels are a type of side-channel attack that exploits the logical behavior of a system, rather than its physical properties. In contrast to physical side-channel attacks, which rely on measuring physical properties such as power consumption, electromagnetic emissions, or timing variations, logical side-channel attacks rely on observing the system's logical behavior, such as its response to certain inputs, the sequence of instructions executed, or the state of its memory.

Logical side-channels can arise from various sources, including:

1. Functional behavior: The system's functional behavior can reveal information about the data being processed or the operations being performed. For example, the time it takes for a cryptographic algorithm to encrypt or decrypt data can depend on the key used, leading to a timing side-channel attack.
2. Control flow: The control flow of a program can reveal information about the data being processed. For example, the sequence of instructions executed in a conditional branch can depend on the value of a secret variable, leading to a control-flow side-channel attack.
3. Data flow: The data flow of a program can reveal information about the data being processed. For example, the values stored in memory can depend on the values of secret variables, leading to a data-flow side-channel attack.
4. Memory access patterns: The pattern of memory accesses made by a program can reveal information about the data being processed. For example, the sequence of memory addresses accessed can depend on the value of a secret key, leading to a cache side-channel attack.

Logical side-channel attacks can be used to extract sensitive information from a system, such as cryptographic keys, passwords, or other confidential data. Therefore, it is important to consider logical side-channels when designing and implementing secure systems, and to use techniques such as constant-time programming, data blinding, and memory access obfuscation to mitigate the risk of logical side-channel attacks.

6.4.5.1 Speculative execution vulnerabilities

Speculative execution is a performance optimization technique used in modern computer processors where the processor executes instructions that may not be needed, based on predictions of future instructions. The goal is to reduce the time the processor would have to wait for data or instructions to be available, improving overall performance.

Hardware techniques to implement speculation in cores and systems include:

1. Branch Prediction: The processor predicts the outcome of conditional branch instructions and starts executing instructions along the predicted path before the actual outcome is known.
2. Data Prefetching: The processor fetches data from memory into the cache before it is consumed by the core, based on the prediction of future memory access patterns.
3. Out-of-Order Execution: The processor executes instructions in an order different from the order they appear in the instruction stream, to avoid stalls due to data dependencies.
4. Speculative Load: The processor speculatively loads data from memory before it is known whether the load is valid or not, to hide memory latency.

Known speculative execution vulnerabilities include:

1. Spectre: This vulnerability exploits branch prediction to trick the processor into leaking sensitive data from one process to another.
2. Meltdown: This vulnerability exploits out-of-order execution to leak sensitive data from kernel memory to user processes.

3. Foreshadow: This vulnerability exploits speculative execution to leak data from Intel's Software Guard Extensions (SGX) secure enclaves.
4. ZombieLoad: This vulnerability exploits speculative loads to leak data from previously executed processes.

To understand where speculative execution hazards could lie, one can think of speculative execution as a form of "optimistic" computation, where the processor makes predictions about future instructions and data, and executes instructions based on those predictions. If the predictions turn out to be incorrect, the processor must roll back the speculative execution and start over, potentially wasting computational resources and opening up opportunities for security vulnerabilities. Therefore, any situation where the processor makes predictions based on incomplete or inaccurate information could potentially lead to speculative execution hazards.

Methodologies for discovery and validation of speculative execution vulnerabilities include formal methods, Xprop and Tprop. The details of such methodologies are not described in this chapter.

6.4.5.2 Cache side-channel attack

A cache side-channel attack is a type of security vulnerability that exploits the behavior of cache memory in modern processors to leak sensitive information. It takes advantage of the fact that accessing data in cache memory is faster than accessing data from main memory. By carefully observing the timing or access patterns of cache memory, attackers can deduce certain information about the data being processed by a target system, even when they don't have direct access to that data.

Here's how a cache side-channel attack works:

Cache Timing: Cache memory is organized into multiple levels, with higher levels being larger but slower. When a processor fetches data from memory, it often loads more data than immediately needed into the cache, anticipating that the nearby data might also be accessed. If an attacker can observe the timing of cache accesses, they might detect whether specific data was present in the cache or not.

Access Patterns: An attacker can manipulate the cache by causing certain data to be loaded into it. By observing changes in access patterns, the attacker can deduce information about the target's memory access behavior. For example, if a particular data item is accessed faster than others, it might indicate that this data was recently accessed and is present in the cache.

Timing Attacks: The attacker can execute carefully designed code or operations that result in different cache access times depending on the data present in the cache. By measuring these timing differences, they can infer information about sensitive data being processed, such as encryption keys or passwords.

Flush and Reload Attacks: One common cache side-channel attack is the "Flush and Reload" attack. The attacker flushes (evicts) a particular cache line, waits for the target to access memory, and then measures the time taken to reload that cache line. Shorter reload times imply that the accessed memory location was likely in the cache, revealing potentially sensitive information.

Prime and Probe Attacks: In a "Prime and Probe" attack, the attacker primes the cache by loading specific data into it. Later, they probe the cache by accessing the same memory locations, and the timing differences reveal whether the data was already in the cache or not.

Cache side-channel attacks have been used to exploit vulnerabilities in cryptographic systems, where sensitive data like encryption keys might be inferred through cache timing. These attacks can be challenging to defend against because they don't directly target the cryptographic algorithms but rather exploit the hardware's inherent behavior.

Mitigation strategies against cache side-channel attacks include techniques such as cache partitioning, cache access control, constant-time algorithms, etc.

Methodologies for discovery and validation of cache side-channels include: formal methods, Xprop and Tprop. The details of such methodologies are not described in this chapter.

6.4.5.3 Stale data and data exfiltration

In a computer or system-on-chip (SoC), data is highly mobilized in operations such as generation, processing, analysis, storing, etc. Secure systems may require cryptographic computation when retrieving, processing, or storing sensitive data. Some of these data carry significant commercial value or impact when they get compromised or exposed. For an example, leakage of the licensed digital media content due to the vulnerabilities in a SOC design could result in legal liability and loss of reputation.

Stale-data is the residue of data-movement in an SOC or cross-contamination of data between transactions which share the same data structures or paths in a system-on-chip. When data moves within a SOC during operations, data is copied and retained in all microarchitectural structures, such as buffers and registers, throughout the data path even long after the movement has completed. Often, the data protection is well defined at the source and destination of a data movement with mechanism such as access control, block or dummy data during debug, RAM scrubbing, memory lockdown, etc. However, the data residue retained throughout the intermediate structures is typically neglected or unknown. These structures are not functionally visible like registers/memory. They are serving the request and data movement in the design hence these internal structures could hold a lot of machine secrets.

As the data exfiltration techniques from hardware are getting mature and more widely available, data movement and handling has become one of important security aspects in secure SoC design. The main objective of data security is to ensure the confidentiality, integrity, and availability (CIA) of data in a system.

Methodologies for discovery and validation of stale data include formal methods, Xprop and Tprop. The details of such methodologies are not described in this chapter.

6.4.6 Protocol bridges

Systems-on-chip rely on multiple communication protocols to interconnect IP blocks and communicate with the external world. These protocols facilitate communication from IP to IP or from IP to I/O (external world), where packetization converts IP transactions (e.g., data read, data write, commands) into packets using a proprietary internal packet-based protocol. The protocol is

designed to be flexible, allowing a variety of packet shapes. Each packet comprises the transaction payload, a header with packet routing and control information, and, optionally, a trailer that provides additional controls and a checksum.

Specific problems come when misconstrued packet control portions are misinterpreted by the receiving decode logic. This sometimes leads to a security concern where bugs in the decode logic may allow for the bypass of protection measures.

Protocol bridges refer to IPs or functional units which provide a mechanism to convert or/and route transactions across different fabrics in different protocols. The protocol conversion can be unidirectional or bidirectional. The conversion implementation can be as simple as for connectivity of different fabrics in SOC design or bridging the communication between agents across fabrics. For example, P2SB allows the host CPUs access to resources residing on the IOSF-SB network and allows IOSF-SB devices with no Primary interface to initiate MSI interrupts on IOSF-Primary.

For connectivity bridges who perform direct protocol conversion such as AXI2CFI, IOSF2CFI, ICXL2CFI, DNI2CFI, they also implement some level of security requirements such as address translation and range access control (IMR), SAI generation or conversion from input protocol to output protocol, etc.

For more complex fabric routers, they can impose more restrictions such as white-list for destination Port IDs, hardware or software configurable opcode and address range mapping, etc. It can also implement hardware mailboxes to allow agents of one fabric to initiate transactions on another fabric. All configurations to HW mailbox registers are required to use the exact same permitted SAI value, else the request to HW mailbox to be terminated without launching of transaction on target fabric.

6.4.7 Formal Verification and Secure-Path Verification

Formal verification (FV) uses formal methods based on mathematical logic and formal reasoning to prove that a design meets its specified requirements or properties.

The method starts by creating a precise mathematical model of a design which captures the behavior, structure and constraints of the design in a formal and unambiguous manner. Then, the method exercises all possible design states and transitions which allows the detection of all corner conditions and subtle errors that might be missed by traditional testing methods. The design under test must satisfy all its functional or safety properties in order to prove its functional correctness and robustness in handling all possible conditions and inputs.

However, a 100% FV-proven design does not mean it is a highly secure design. There are many reasons to this.

- FV property definition may not include security requirements, or the property wasn't security intended but more on functional and safety.
- FV can prove basic security requirements and mitigations which are clearly defined in spec and implemented in the design. But it cannot cover all subtle security requirements or complex attack scenarios. Some dedicated efforts required in translating any type of in-scope threats into specific properties.
- Some security threats or issues are hard to be translated into FV properties.

- Security scope is broad and security verification is a risk-based approach, hence it is impossible to translate all potential threats into FV properties.

Secure-Path Verification is a field of FV which focuses on secure-data propagation in the design in searching for potential data leakage and cross-contamination.

6.4.8 Information flow and TProp

Taint propagation is not a new concept: in the late 19th century, European hydrogeologists started using a special dye to color the waters of underground rivers to find where they resurface. T-prop is a similar idea for RTL design flows. Similar to coloring and tracing the flow of water, T-prop allows tagging (or tainting) selected signals in the RTL, enabling dynamic tracking of the flow of the tainted signals to observe which design locations they affect.

Tprop can be used for the following use-cases of information flow tracking and analysis:

- **Data confidentiality analysis:** May allow the analysis of how data travels within a design to identify potential leaks, where sensitive data may inadvertently reach unintended parts or boundaries of the design.
- **Data integrity analysis:** Enables the analysis of data flowing through the design, to identify potential sources of data corruption or unauthorized modifications. Such analysis could also be used to qualify the effectiveness of fault detection or fault correction.
- **Data persistence quantification:** Used to quantify how long and where sensitive data persists in the design. Sensitive data may remain in caches, buffers or registers for an unspecified duration thereby representing a risk of potential vulnerabilities.
- **Control flow analysis:** May help in identifying vulnerabilities that are due to incorrect control flow, such as unintended execution paths that might lead to unauthorized access or code injection.
- **Design debug and discovery with transaction tracking:** May provide a way to quickly identify a transaction and its path since Verdi displays tainted signals in a different color. This approach saves time compared to the manual addition and tracing of signals in block traversed by a transaction.

Other similar methods exist:

- **CELLIFT** by ETH Zurich (<https://intel.sharepoint.com/sites/ipastechsharing/SitePages/2022-06-29.aspx>)
- **FPV/SPV:** While formal path verification (FPV) and secure path verification (SPV) formal methods can help validate the existence of logical paths between two logic nodes (a source and a destination) in IPs and systems, it has some limitations with scalability and for software configured logic. FPV and SPV are complementary to taint propagation and information flow analysis.

6.5 Trade-offs

This section describes the impact/ROI of doing security validation.

6.5.1 Verification Scoping, Risk assessment And Prioritization

The common practice in secure development involves activities such as defining security objectives, identifying security requirements, threat modeling, architectural and implementation review, risk assessment and verification. Ultimately, these processes aim to reduce the likelihood of security breaches, data leaks, and other security incidents by addressing vulnerabilities and threats early in the development process.

Risk assessment and prioritization is a critical step of the security verification process. It involves evaluating the potential risks associated with a design and determining how to prioritize and address those risks. The goal is to allocate resources effectively and focus on mitigating the most significant security threats first.

The explanation of the risk assessment areas and the factors to consider for prioritization as below:

1. **Critical Assets:** Identifying critical security assets that need protection, such as sensitive data (e.g., cryptographic keys and operations, configuration, fuses), hardware components, firmware images, etc. Prioritize risks associated with critical assets that are essential for the CIA (Confidentiality, Integrity, Availability) requirements of the design.
2. **Potential vulnerabilities and likelihood of exploitation:** Enumerate potential threats to those assets. Threats can come from various sources, such as malicious hackers, insider threats, natural disasters, or system failures. Estimate the likelihood of each threat exploiting a vulnerability in terms of probability, cost and known exploits available.
3. **Known threats and vulnerabilities:** Address risks related to known, active threats of a design. This considers historical data, security mitigations in place and some threat intelligence.
4. **High-Impact Risks:** Assess the potential impact of each threat exploiting a vulnerability. Consider the consequences in terms of financial loss, damage to reputation, regulatory penalties, and disruption to business operations. Give priority to risks with a high impact on the organization's business, reputation, or customers.
5. **Regulatory Compliance:** Address risks that are related to regulatory requirements or industry standards first, as non-compliance can lead to legal consequences.
6. **Resource Availability:** Consider the availability of resources (e.g., budget, headcounts) for addressing specific risks. Allocate resources to address the most critical risks first.

By conducting a thorough risk assessment and prioritizing based on these factors, organizations can effectively allocate their resources and efforts to address the most significant security risks, thereby enhancing the overall security posture of their systems and operations.

6.5.2 Discovery vs Assurance

While discovery is about finding vulnerabilities, assurance is about ensuring that those vulnerabilities are being appropriately addressed. Both phases are crucial for comprehensive security validation.

- **Discovery:** This phase is primarily about identifying vulnerabilities, threats, and risks in a system. It involves various security testing techniques such as penetration testing, vulnerability scanning, and risk assessment. The goal is to discover any potential weaknesses that could be exploited by an adversary (in scope). This phase is often more

focused on the technical aspects of the system and is usually performed by security experts or ethical hackers. In pre-silicon, this includes advance fuzzing, creating variations of known attack scenarios, creating variations of known vulnerability validation tests, porting attacks or vulnerability test to new protocol or IP.

- **Assurance:** This phase is about providing confidence that the identified risks are being managed effectively. It involves evaluating and verifying the effectiveness of the mitigations in place (controls, countermeasures, and safeguards). This could include hackathons, code review, as well as testing the effectiveness of security controls and mitigation. The goal is to assure that the system is secure and that any residual risk is acceptable. This phase often involves both technical and non-technical aspects, such as SDLe completion, compliance with regulations and standards. In pre-silicon validation specialized coverage, target testing (including negative testing) and key/rnd persistence quantification are some of the required validation collaterals used to provide some confidence that security mitigations have been sufficiently tested.

Both discovery and assurance are important for quality security validation.

7 Summary

Validating security is important because it ensures that a product or system has been designed and implemented with security in mind. It helps identify and resolve potential vulnerabilities before they can be exploited by attackers. Security issues in products can have disastrous consequences: data breaches, service disruption, reputation damage, financial and legal penalties.

Security Validation is a vast field; it covers functional validation of security-related features, validating secure flows end-to-end, negative testing, and security assurance. This chapter is an introduction to get the reader grounded on what Security Validation is, why we do it, and the areas to begin planning. The scope is, in some sense, enormous, and validation teams need to understand the whole picture to set the best strategy for each project.

8 Future Work

1. References to RNG, Crypto and PUF validation
2. Threat modeling and attack reproduction: How to do threat modeling – where to include or provide reference? How to create pre-silicon test plan – where to include or provide reference?
3. Structural hazards and logical side-channels: Speculative execution discovery and validation
4. Structural hazards and logical side-channels: Cache side-channel discovery and validation
5. Structural hazards and logical side-channels: Stale data discovery and validation
6. Negative Testing: access control
7. Negative Testing: address decode
8. Negative Testing: control decode
9. Negative Testing: event ordering
10. Add section on negative testing of each phases of access control
11. Add section on use of Formal

12. Trade-offs: Testing and ROI:

1. Where and when to test what. When to stop
2. IP-level vs System-level validation
3. E2E validation pre-Si vs post-Si

13. Trade-offs: Classifying bugs

1. Classifying bugs (spec, functional, security, ...), where the bug lies (functional bug with security impact, security function bug, security bug, ...)
2. How to use risk assessment to help decide if a bug needs fixing or not. (future)
3. PSIRT and PRT (future)
4. Public disclosure (future)

14. Metrics

9 References

https://read.nxtbook.com/ieee/spectrum/spectrum_na_march_2019/how_the_spectre_and_meltdown_.html

Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA
<https://www.sciencedirect.com/science/article/pii/S0306437920300338>

Intel security technologies: : <https://ircs.intel.com/WebUI/Documents/Download/65155>

The Art of Pre-Si Val: Chapter 40

Fullchip Validation

By: [Mike F. Miller](#)

1 Abstract

In modern designs, large, deep hierarchies with multiple integration steps are used to construct chips. The “last” level of the integration hierarchy is usually an entire chip or package to be delivered to customers. Feature and Cross-Feature validation done at this level is usually called Fullchip Validation (FCV). Because this is the “highest” point in the hierarchy that has Pre-Silicon models, the customers and goals of validating this last level are somewhat different than other levels in the design. This chapter first discusses the differences in goals, customers and guidelines that result. Then examples of specific FCV activities are discussed. Not all activities are appropriate for all projects, but should provide ideas and guidance.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	07/25/2016	First complete draft	Mike F. Miller	Michael Bair, Cameron Wilde

3 Contents

1 Abstract.....	773
2 Revision History.....	773
3 Contents.....	774
4 Purpose.....	776
4.1 Why do we need this chapter?.....	776
4.2 What does this chapter cover?	776
4.3 What does this chapter not cover?	776
5 Background Information.....	776
6 Fullchip Validation (FCV).....	777
6.1 FCV goals.....	777
6.2 FCV Customers	777
6.2.1 Post-Silicon Validation	777
6.2.2 Other Fullchip and IV teams	778
6.2.3 Model build teams.....	778
6.2.4 External Validation teams	778
6.3 FCV guidelines	778
6.3.1 Platform fidelity over model performance.....	778
6.3.2 Realistic content over synthetic content.....	779
6.3.3 Breadth of content over depth of content.....	779
6.3.4 Silicon configurations over random configurations.....	780
6.3.5 Reused content over building new.....	780
6.3.6 Enabling others over increasing depth.....	780
6.4 Specific activities	780
6.4.1 Defining fullchip environments, tools methodologies and flows	781
6.4.2 Fullchip features and flows (FV)	781
6.4.3 Cross Feature Validation (CFV).....	781
6.4.4 Configuration and Defeatures	782
6.4.5 Non-tapein component integration.....	783
6.4.6 Firmware and Software integration	785
7 Summary.....	787
8 Future Work.....	787

8.1.1	Quality over quantity	787
8.1.2	Performance Validation.....	787
9	References.....	788

4 Purpose

This chapter discusses Fullchip Validation as an activity that is distinct from other validation activities. The goal is to introduce the reader to both “traditional” Fullchip Validation activities that tend to be common to all projects and also to some of the extended activities that don’t apply to all projects. The reader should then be able to have a broad enough understanding to know what to apply to their project.

4.1 Why do we need this chapter?

Because the exact activities of Fullchip Validation varies from project-to-project based on both different project validation requirements and varying organizational choices, it can be very challenging to get a good grip on what exactly “Fullchip Validation” is. Fullchip validation as a more abstract activity (as opposed to a specific team) has some properties that are different from validating at other levels of the integration hierarchy and requires some distinctly different priorities. Failure to recognize these differences when doing Fullchip Validation will result in high costs and escapes of critical bugs to Post-Silicon.

4.2 What does this chapter cover?

This chapter takes an exceptionally broad view of Fullchip Validation. It covers what in some projects is called Platform Validation. Platform Validation includes not only the chip that the specific Design/Validation team has responsibility for but also the broader platform integration (other components in the system).

4.3 What does this chapter not cover?

Team structure is omitted from this chapter. There are multiple appropriate solutions possible for organizing the work and readers should consider the discussion in [Validation Disciplines](#) and [Leading a Validation Team](#) to help guide their team composition decisions. In DDG the FCV team has a broad range of responsibilities that extend beyond the elements included here, but in other product teams they organize the work differently.

This chapter does not discuss model build activities, although they are sometimes included in the Fullchip Validation team for management reasons.

5 Background Information

This chapter assumes you have read the [Validation Disciplines](#) chapter and also have a good grounding in the basic concepts of stimulus, checking and coverage as described earlier in *The Art of Pre-Silicon Validation*.

6 Fullchip Validation (FCV)

6.1 FCV goals

Despite the best efforts and intentions of IP Validation and Integration Validation bugs do escape their testing. Some bugs are only discoverable when multiple components are integrated together because of modeling issues, conflicting specifications or other factors have masked issues. Because of this, we need to address this class of issues using higher fidelity models. The primary customer of FCV is the Post-Silicon Validation team. Their environment is the target of the FCV team's efforts. The goals of fullchip validation are:

- Discover Pre-Si bugs that are best exposed by multiple system components, including firmware and software, interacting together or missed by other val methods. (i.e. improve the design)
- To demonstrate that the silicon will behave correctly in the environment it will be configured and placed into for bringup and volume validation. (i.e. “prove” the design)

An interesting note on goals is that FCV is not trying to eliminate all bugs and rarely uses bugs found as a metric of success. In general, because of the nature of bugs and the percentage that should be escaping other levels of validation, very few bugs should be making it to FCV. Moreover, the FCV team should be actively working with IP integration validation and IP validation teams to catch more bugs at those levels, further reducing the percentage of bugs that the FCV team finds.

6.2 FCV Customers

6.2.1 Post-Silicon Validation

Post-Silicon Validation is the primary customer of FCV activities. Not all bugs have the same impact to the project, and different validation techniques can emphasize the finding of particular classes of bugs. For the Post-Silicon Validation team some bugs are dramatically more impactful than others and the FCV Validators focus on activities with the following priority:

1. Eliminate bugs that block validation of features (so volume validation can start)
2. Reduce the volume of bugs that reach the Post-Silicon team
3. Eliminate simple but hard-to-debug bugs caused by micro-architectural interactions

While all of these are good things to find, because of resource issues, you may need to prioritize. For example, the team might prioritize a full reset test with the chipset and production firmware over running long random instruction tests because the full reset would uncover bugs that would more likely block the Post-Silicon team and running long random instruction tests would more likely find hard-to-debug micro-architectural issues.

When silicon does arrive, the FCV team should support the Post-Silicon team by aiding in initial lab debug and reproducing failures found on silicon in higher-visibility environments (emulation and simulation for example)

6.2.2 Other Fullchip and IV teams

The FCV Validators are responsible for setting and clearly communicating the models, tools, methodologies and flows (TMFs) for the other teams using fullchip models such as the IP Integration Validation teams. (see section 6.4.1).

6.2.3 Model build teams

The model build teams (particularly the emulation and FPGA build teams) have great expertise in compiling RTL. They do not have detailed knowledge of specific projects, tests or debugging logic issues. The FCV team needs to support them with breadth-oriented regression test content and debug support. The test content is usually a reuse of existing breadth content that the FCV team already maintains for their own validation purposes. This allows the model build teams to remain horizontally aligned to technologies and rely on FCV for project specific (vertically aligned) support.

6.2.4 External Validation teams

For chipsets (or other external chips), firmware and software, the FCV team needs to provide models, TFMs and debug support. This varies widely between different projects, but the underlying principle is that FCV provides support for the fullchip models so that Validators outside the project can productively engage with the project. In practice this frequently results as interactions with non-tapein components (section 6.4.5) and FW/SW teams (section 6.4.6).

6.3 FCV guidelines

The two FCV goals result in FCV Validators making tradeoffs that are significantly different than the Integration Validation or IP Validation teams make. This does not imply that other teams are making the wrong decisions, but rather that FCV is complementary, not redundant, to their efforts. While these tradeoffs are not absolute, they are helpful as guidance to understand how FCV is unique.

6.3.1 Platform fidelity over model performance

Because FCV Validators are looking for interaction problems between multiple components, the activities tend to utilize more complete and higher fidelity models than other Pre-Si activities. The goal is to address potential deficiencies in other validation environments that might be masking bugs. For example, by having all the components on the chip modeled in a “fullchip” model, you eliminate the risk that BFM-modeled IPs are modeled incorrectly. Another example is doing “co-simulation” where a CPU complex and the associated chipset are modeled together to eliminate risk that the BFMs used in the respective environments were not correctly modeling the real behavior of the devices. Similarly, running the production firmware for an IP instead of test firmware can expose new behaviors and interactions that are problematic. There is a significant cost to building and working in these large environments. Frequently the runtime of the tests can

be a significant limitation. The team looks at project specific risks and costs to make appropriate tradeoffs for models and test content. These tradeoffs will vary significantly between projects, even to the point of some activities discussed in this chapter skipped for a given project.

6.3.2 Realistic content over synthetic content

The desire for fidelity also impacts other decisions like test content, checkers and coverage. Where possible, the team will want to reuse collateral or create collateral for reuse by the Post-Silicon Validation teams. This is a very challenging goal because of the significant differences in performance and visibility between the Pre-Silicon environments and the silicon environment. The [Interaction with Post-Silicon Validation](#) chapter describes some of these differences. While complete convergence is not currently possible, there are places where some convergence is possible at reasonable cost and is highly beneficial. For example using the same coverage collection mechanism as Post-Silicon enables FCV Validators to exercise the mechanism by using it.

Unfortunately, the costs associated with running realistic content are frequently prohibitive to Pre-Silicon Validation platforms, so there are many cases where synthetic or silicon-unfriendly techniques are appropriate. For example on Client designs in the mid-2010s, the Post-Silicon teams were using OS Based Validation (OSbV) heavily, but the Pre-Silicon platforms could not boot the OS fast enough to make it a viable bug discovery technique. FCV did not use that content, despite a strong preference to use it. Experience and good engineering judgment are necessary.

In practice this also leads the team to use more directed content like use cases rather than highly random content. Much of this is focused on “proving” that the design will work correctly for certain high-value use cases like secure boot, or doing a software-involved power-state transition. These activities ensure that there are not critical bugs that would completely block the Post-Silicon team’s progress from validating a broad section of functionality.

6.3.3 Breadth of content over depth of content

In general, covering a wider breadth of interactions is preferred over depth into a particular area. We can assume that the IV and IP teams have already done substantial depth testing for their IP. If FCV is finding significant bugs in a particular IP that could have been found in a smaller environment, it is feedback that the smaller environment is not being used appropriately or that there is a bug preventing the exercise or detection of a certain feature and corrective action should be taken. Because part of the goal of FCV is to find unanticipated or missed bugs, FCV needs to not make too many assumptions about where the bugs are initially, but be data driven off a broad set of content that lightly covers all areas and use failures to direct the team toward problematic areas. Because the FCV teams environments are expensive, constructing and running high-depth content everywhere is simply too expensive and overlaps excessively with the IV and IP team’s activities.

This also extends to our Validator skillsets. Because FC Validators are expected to work across the entire chip (and sometimes entire platform), their knowledge and debug skills are more focused on understanding interfaces and global flows, not on microarchitectural details of specific IPs.

FCV team members develop deeper knowledge of specific IPs as they engage with the design and other Validators to debug and resolve issues, but the proactive trend is to develop breadth first.

6.3.4 Silicon configurations over random configurations

Because Post-Silicon Validation is the primary customer of FCV, using the same configurations as they do is highly beneficial to eliminating blocking bugs and should be the team's first priority. Other configurations should also be used for other validation benefits such as model performance, more easily hitting corner cases and evaluating performance targets.

6.3.5 Reused content over building new

The team is highly motivated to use to reuse either IP Validation or Post-Silicon content instead of building its own. It is too expensive to be experts in all the IPs necessary in a SOC-era design. Because of this dependency on the IP validation teams, the FCV team needs to drive standards for reuse (see section 6.4.1).

Most FCV teams have deep experience in one or more IPs for historical reasons (e.g. teams were previously an IP validation team). While this can be an asset, the team needs to recognize that their knowledge of that IP will degrade over time and they need to use the collateral from the new IP owners instead of developing their own.

6.3.6 Enabling others over increasing depth

The FCV team is usually interacting with other groups outside their immediate team (other feature and integration validation teams, Firmware/Software developers, other platform component teams, Post-Silicon teams) and they should prioritize enabling those other teams to do work on Pre-Silicon platforms over increasing their own depth for two significant reasons:

1. The more people who can use fullchip Pre-Silicon platforms means more total content can be run, triaged and debugged.
2. Other teams bring new test content, fidelity, experience and ideas – this is good because the team values getting new, unique content to expose latent assumptions and issues in the design hidden by the existing environment.

Because of these benefits, as well as many others, in many areas the FCV team will work to enable others to make forward progress using common models and validation collateral.

6.4 Specific activities

There are specific activities that the FCV Validators engage in to support the customers and goals. While the FCV Validators are not limited to these, and for tactical reasons they might be organized in other ‘teams’ they are used here to provide concrete examples both from current and historical projects.

6.4.1 Defining fullchip environments, tools methodologies and flows

The FCV Validators are responsible for collecting requirements and setting tools, methodologies and flows (TMFs) for the other teams using fullchip models and doing integration validation work. They are responsible for working with the model build teams to ensure that the models created and TFM are globally optimal for the project and aligned with the company strategic directions. This is critical, as the FCV Validators must reuse collateral from the IP Validation teams (see section 6.3.5). In practice there is a very large amount of project-to-project reuse of validation content, particularly for large, cross-SOC flows. Where possible, reusing, or getting updates from the IP Validation and IP Integration Validation teams is preferred to get all the optimizations and improvements associated with the new version of the IP.

6.4.2 Fullchip features and flows (Feature Validation)

Fullchip features and flows are any features or flows that require multiple IPs to support. This is a Feature Validation approach (as described in chapter [Validation Disciplines](#)). Examples include:

- Cache coherency (Core, GT, Uncore and memory cluster are all involved)
- Memory ordering (nearly all IPs)
- Reset/Boot (security engine, storage cluster, power management controller, fabrics, USB, firmwares)
- Interrupts (most PCIe-like devices, uncore and core)
- Power Management (global, but some IPs are more involved than others)
- Machine Check Architecture – MCA (events come from many IPs and must be aggregated and sent to the cores)
- DFX features (all IPs)

Some fullchip features have dedicated teams because of the unique complexity of those features or historical organizational reasons. For example, cache coherency and memory ordering on the Pentium 4 projects were in a separate MultiProcessor Validation (MPV) team that focused exclusively on those issues, but in Core Client projects that work is done by the FCV team.

In this portion of work, more attention is paid to depth of testing than other areas because these areas could not have been exercised well at the IP Validation or Integration Validation levels (or there is modeling of the missing components that may be deficient).

6.4.3 Cross Feature Validation (CFV)

The Cross Feature Validation approach as previously described in [Validation Disciplines](#) is used heavily by FCV Validators. Combining functionality in ways that expose critical issues is a significant value add that is specific to this team. This means that the team needs to work closely with other integration validation and feature validation teams to ensure that the test content is reusable with the FC models and can be combined in useful ways. This generally means that the FCV team needs to own Fullchip test generator and modeling methodology to ensure that all the groups can build test content that is consistent and compatible. This does not mean that the FCV team needs to own all the tools, but rather work together with providers to ensure consistent and globally optimal solutions.

Fullchip models tend to be very expensive to run, and test content from different integration or feature validation areas do not always work well together. Because of this, engineering judgment must be used to limit the scope of combinations to cost-effective areas. There are multiple approaches to doing this analysis and building a testplan, below are two:

- Look at all the feature validation and integration validation areas and identify places where common resources are arbitrated and used. The most obvious is contention in fabrics and memory that could result in deadlocks, livelocks or unacceptable latencies⁴³. There are other interactions of features that are less obvious such as power management flows with other non-atomic flows, security features and non-default configurations. Each project has unique properties and interactions of features in the hardware that need to be considered.
- Look for high level use cases that require multiple blocks to be functioning together at near-peak performance. Use cases are particularly helpful for products where the user requirements are well defined. Good use cases help focus validation into high value areas and are particularly good at removing Post-Silicon blocking bugs, which is one of FCV's primary goals.

The creation and review of Cross-Feature driven testplans requires very careful balancing of resources and risks that are project and technology dependent.

A side effect of doing Cross Feature Validation is that the team doing it may discover a feature that is insufficiently validated in isolation. The normal behavior of the FCV team is to work with the Feature Validation team to understand the gaps and get them addressed. This may mean that the FC Validators will engage heavily on Feature Validation work to get the area cleaned up so that Cross-Feature Validation can proceed with good ROI. This is an important aspect of the team in that they actively contribute to identifying and eliminating holes in other validation areas that arise for a variety of reasons.

6.4.4 Configuration and Defeatures

The Post-Silicon Validation team has some severe restrictions in working with Silicon. One of these is that they cannot easily change the hardware. This means that when a bug (or persistent manufacturing defect) is found, the team has limited options for fixing or working around the bug so forward progress can be made. Designers include many configuration options and defeatures to increase the survivability of the design. There is only limited testing of the majority of these defeatures as the primary focus of the Pre-Silicon Validation team is eliminating bugs in the default configurations of the machine.

To support the Post-Silicon Validation team, the FCV team uses their large SOC-wide regressions along with enabling uncommon configurations and defeatures that should not impact functionality. This is usually done with automation to randomly select configurations and defeatures as there are a lot of items and combinations. With automation and a high pass rate on the SOC-wide regressions, this generally results in a cost-effective way to identify problems. This breadth testing tends to find a number of broken items, which are documented and occasionally fixed. Even if not

⁴³ Some devices have realtime output requirements (e.g. Display, Audio, USB and PCIe) and in the system we make guarantees on request latencies to ensure that those devices don't "stutter" or "flicker" which would result in a bad user experience.

fixed, documentation for the Post-Silicon Validation team is invaluable to prevent them from wasting time trying to use that particular configuration or defeature.

6.4.5 Non-tapein component integration

There are situations where there is significant risk in BFM modeling that needs to be addressed by including high-fidelity models of devices not part of the “chip”. Simulating (or emulating) multiple chips is called “co-simulation” or “co-emulation” (we will use the term “cosim” to generically refer to both) to indicate that multiple chips are being simulated or emulated together. This is not always necessary. Most chip-to-chip interfaces are rigidly defined by external specs that are well understood and have trustworthy BFM (like PCIe, USB, SATA). But for cases where they are not, or the interface is new, having a Pre-Silicon model of both ends of the interface running together is a great way to find bugs in both the implementation and specification.

For any component integration activity, the first and most important activity is to establish a good working relationship between the individuals from the two teams doing work. This means starting early, having frequent, open communications, face to face if possible. Doing this will set up the activity for success.

Figure 50: Example of abstraction

While large models are very powerful for increasing confidence in basic functionality, they tend to be exceptionally expensive to build, maintain and run. Moreover, most blocks perform some form of abstraction between interfaces. For example, the uncore provides a very good abstraction between the Core and the Chipset as shown in Figure 1. This significantly reduces the probability of bugs that could only be exposed by having the Core and Chipset RTL in the same model. In this case, it may be much cheaper to have a model of just the uncore and the chipset as there are not direct interactions between the core and the chipset. If you are also validating IAFW running on the cores and accessing the chipset as shown in Figure 2, then it becomes more interesting to have the cores included. Doing so, allows you to test the interactions of the IAFW with the core, uncore and chipset. This is interesting because the IAFW interacts with the core microcode to create special transactions, and with the uncore setting up routing registers, interrupt mapping and other configuration that impacts how the core and uncore hardware interacts with each other and the PCH.

In general, the FCV team must take care to select models that ensure good coverage of interacting components at low cost. Use of hybrid models and sub-models like an uncore-chipset model, or even a cluster-to-cluster model can provide models with sufficient performance to run a lot of content. Smaller and hybrid models can have much lower overhead and runtime latency than larger



Figure 51: Example of coupling

model with minimal risk of escapes. See the [Validation Platforms](#) chapter for more discussion on these topics.

6.4.5.1 Traditional Cosim

In traditional co-simulation, a shim is added to both the ‘fullchip’ simulations of the two chips where their interfaces would connect to each other and the shims talk across a RPC link to each other. This has the benefit of leaving the two independent environments nearly independent of each other, but limits communication between the two testbenches. As an example, the first QuickPath Interconnect (QPI) platform was one or two Core client CPU parts with a chipset. Because it was a new connection mechanism designed completely from scratch, the team did substantial co-simulation of the CPU and chipset. The effort found over 50 bugs, many of which would have been fatal to the platform coming up at high speed and likely would have delayed the project by a stepping. By contrast, most of the DMI or OPI-based platforms do not usually use cosim because the interface is reused and the BFM has proven sufficient for testing the components individually. Also, most Client platform validation teams also utilize SLE, which also exercises the Processor-Chipset link sufficiently well.

One of the critical issues with cosim is that the majority of the work is not getting the two RTL models to work together, but in getting the two testbenches to work with each other. The two RTL models are designed to work together, but the testbenches are usually not. When scoping a cosim activity, be sure to look closely at how the testbenches and other validation collateral from each chip will need to be modified (or removed) to work together and what implications that has on your stimulus, checking and coverage.

6.4.5.2 System Level Emulation

System Level Emulation (SLE) is an emulation platform that usually includes all of the major components and testcards in a system into a single (very large) emulation model. This is sometimes owned by Post-Silicon Validation teams as well as Pre-Silicon teams but is a very valuable tool for both teams as it provides near binary-compatibility for FW/SW running on the platform.

6.4.5.3 Direct inclusion

There are other situations where integrating the RTL of an additional chip into the “fullchip model” is appropriate. These are usually when the RTL of the second chip is small, or is a completely custom interface for the primary chip and needs many validation cycles. For example, in the Core Iris Pro platforms, there is a CPU chip and an eDRAM⁴⁴ chip that is custom built for the CPU. These are simulated together as part of the “fullchip” model because running the eDRAM RTL is not substantially more expensive than a BFM would be. This approach eliminates the costs and risks of building a BFM. Similarly, in some SOC designs, the discrete Power Management IC

⁴⁴ eDRAM is “embedded DRAM”, an Intel-fabricated DRAM module that is exclusively for enabling high-performance integrated graphics. It is a separate chip from the CPU/GPU for fabrication cost reasons.

(PMIC) that is responsible for power delivery to the SOC is included in the SOC model because it is easier than building a separate BFM and has negligible simulation cost.

6.4.6 Firmware and Software integration

For platform firmware and software that is not ‘preintegrated’ to the RTL model, there is a significant gap in our traditional validation approaches, mostly due to organizational choices. Some firmware is integrated directly into the RTL model and utilizes the same revision control and integration mechanisms as the RTL. Some examples include ucode, pcode and PMC FW which are managed almost identically to an RTL-based IP during the Pre-Silicon timeframe. Chapter [Embedded FW Validation](#) describes this situation.

Firmware developed outside the integrated model has a different set of PLC requirements and needs to be integrated at a minimum as part of the PSS processes as discussed in [The Life of a Project](#). Pre-Silicon Validation has some additional goals beyond the PSS activities. These additional goals are to run a few high-value use cases on higher fidelity models (see section 6.3.1) to ensure Post-Silicon Validation is not blocked in key reset and power management flows. This is usually a joint discussion with the PSS team to determine the right crossover point between needing FCV hardware debug expertise to work on a high fidelity model versus relying on software debug skills in the PSS teams.

There are challenging issues that require attention when doing FW/SW integration on Pre-Silicon models. Many of these challenges arise from differences between the underlying cost models and organizational choices currently in place. These aspects do not apply equally to all groups, but are generalizations to consider when working in this challenging environment.

6.4.6.1 Development Style

The fundamental cost model differences between HW and FW/SW as described in the [Introduction to Pre-Silicon Validation](#) significantly impact the timelines and development style. Because of the exceptionally high cost of change after RTL freeze, the HW team looks very much like a “big upfront design” style. In stark contrast, the cost of change for the FW/SW teams does not increase significantly until well into the Post-Si timeframe⁴⁵. This results in the FW/SW teams generally using more iterative or agile strategies and accepting a much higher rate of change very “late” in the project relative to the HW teams where late change entails high risk.

6.4.6.2 Staffing Availability

Early in the project, it is very difficult to get Firmware and PSS support because their engineers are working extensively post-tapein on earlier projects. Because of this, it is highly advantageous to have FCV staff fund early exploration of using models (usually emulation models) to run and debug boot firmware and software. They can work with the model delivery team to get sufficient

⁴⁵ This is not true for FW that is put into ROMs, but this is a small fraction of the overall FW/SW stack and usually extensive patching mechanisms exist to keep the cost of change relatively low.

run and debug support in place (e.g. building and loading flash images, fuse settings and appropriate tracker files to enable debugging firmware issues).

6.4.6.3 Organizational Mismatch

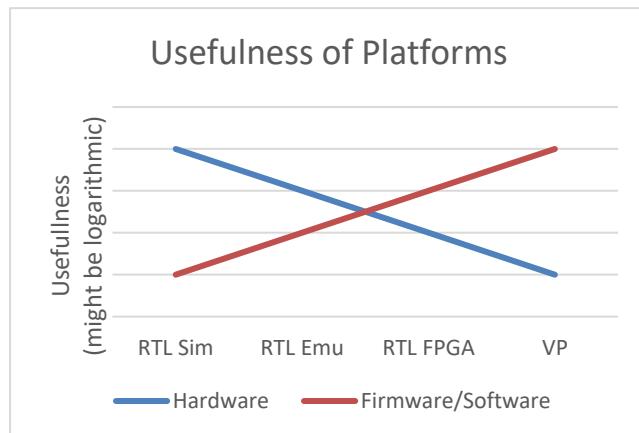
While hardware teams have loose associations with business groups, they will frequently supply multiple business groups out of the same family. Hardware groups also frequently have a “ping-pong” between two teams where a product family is supported in alternating generations between two teams. In contrast, the FW/SW teams are usually inside the business groups and support a specific swim-lane of customers. This mismatch in ownership means that each team must deal with multiple teams on the “other side” and more relationships must be cultivated as well as working through roles and responsibilities, timelines and methods of works with many stakeholders.

6.4.6.4 Workflow/Debug Style

For HW teams the build-run-debug latency is usually measured in hours to days. In contrast FW/SW tends to be much faster, in the minutes to hours range. Because of this, the HW teams tend to do debug as a post-run activity on logs or traces and SW/FW teams debug interactively on a platform (virtual or real) using debuggers. Because the debug approaches start from a different framework, it is very challenging for Validators to switch between the two styles as the entire debug toolstack makes assumptions about interactive versus non-interactive⁴⁶. This becomes additionally problematic when discussing the effectiveness of particular Pre-Silicon platforms.

6.4.6.5 Pre-Silicon Platform Choices

One of the more contentious issues between HW and FW/SW teams is the choice of platforms. This is largely due to the fact that the teams value different attributes of the platforms. The HW teams prefer the RTL-fidelity and internal visibility of RTL simulation and emulation models, but the FW/SW teams do not require that level of visibility into the HW and instead need the higher performance of FPGA and VP models to enable their workflow and debug tools.



It is important for both teams to recognize that the different platforms provide different value propositions and that they value different things. If the HW team is supplying models to a FW/SW team they need to not assume that they know “what is best” for that team. In addition, any kind of “convergence” will likely require both teams to accept some inefficiencies.

⁴⁶ As a historical note, hardware teams did debug interactively on simulation on the P4P family of processors up until the very early 2000s when simulation performance degraded to the point it was more productive to do debugging as a post-processing activity.

See the [Validation Platforms](#) chapter for a deeper dive comparisons of all the Pre-Si platforms.

7 Summary

Fullchip Validation is a mixture of multiple approaches that emphasizes breadth of validation to ensure that the Post-Silicon team can maximize the learning from each piece of silicon. FCV activities are complimentary to IP and Integration Validation as well as supporting them with guidance to global optimizations and feedback to weak areas. Bringing together key platform ingredients increases the quality of validation by eliminating assumptions made in other validation efforts about the behavior of those ingredients and thereby reducing risk to a successful poweron of silicon.

8 Future Work

There are some areas that FCV is starting to get engaged in but are not entirely ready for inclusion in this chapter. They are listed below with some early discussion of the topics.

Make sure to discuss “UX” issues and relationship with PSS.

8.1.1 Quality over quantity

Need to think more about this as a strategy/value statement.

8.1.2 Performance Validation

The FCV team owns Pre-Silicon end-to-end performance validation of the system. Current modeling technology limits what can be done and many tradeoffs must be made.

The first step is working with the architecture team to define what the performance requirements are at the platform level and what the measureable performance requirements are for individual IPs or subsystems. The FCV team’s goal is to ensure that the multiple components that participate in delivering that performance deliver the end-to-end performance.

8.1.2.1 Workload Performance

Run perf-like workloads on a fullchip model(most test content needs to come from the IP/IV teams)

Things like:

- CPU, GT, PCIe max bandwidth to DRAM tests
- Max pixel count display tests
- Multi-IP workloads (South image capture, CPU, GT, Display, Audio, GMM for recognition and PCIe for storage)

Ideally, production software use cases from the Post-Silicon environment would be best to capture all the behaviors together (see section 6.3.2). In practice, synthetic content from IP teams may be a better choice for easier bringup and evaluating if the performance criteria are being met.

8.1.2.2 End to End Interconnect Performance

Verifying end to end interconnect performance can identify issues with interactions of IPs and fabrics and interactions of traffic from multiple IPs. There are a couple of specific activities that the FCV team can use to identify problems:

Add checkers on interfaces for latency and isoch failures. Ideally, enable Post-Silicon teams by getting some of the checkers into visible places like MCA or VISA so they are not having to listen for audio glitches. Having the checkers enables the team to find the bugs faster by catching micro-architectural failures, not architectural failures which are more rare. The challenge is to define the micro-architectural rules so that there are not false positives.

8.1.2.3 IP performance

IP performance validation should be done by the IP team. They have the detailed knowledge of the IP necessary to understand the requirements and environment to evaluate if the design is meeting those requirements.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 41

Becoming the Architecture Expert (TBD)

By:

1 Abstract

It is important for Validators working within Integration Validation and Feature Validation to become *Architecture Experts*.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers

The Art of Pre-Si Val: Chapter 42

The Post A-Step World

By:[Michael St. Clair](#)

1 Abstract

During the course of a microprocessor project, the Pre-Silicon team develops critical skill sets that are invaluable to Post-Silicon tasks. The success of the project depends on loaning team members out to help the Post-Silicon teams. The team enters the *Post A-step* world.

This chapter explains the various roles that the Pre-Silicon Validators play in Post-Silicon, including the nebulous phase between tapeout and silicon arrival. Details are provided on the areas where Pre-Silicon Validators are most involved, including the new focus of the team that is still working on Pre-Silicon Validation.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
0.1	11/11/2004	First release of document	Michael St. Clair	L1 draft reviewed by Michael Bair.
0.2	3/28/2005	L3 Review	Michael St. Clair	Steve Cegla
0.3	3/28/2005	Incorporated L3 review comments	Michael St. Clair	Steve Cegla
1.0	3/28/2005	Finalized L3 review comments – Released document	Michael St. Clair	Steve Cegla

3 Contents

1 Abstract.....	791
2 Revision History.....	791
3 Contents.....	792
4 Purpose.....	793
5 Background Concepts	793
5.1 Post-Si versus Post A-Step: The Splintering of the Pre-Si Validation Team..	793
6 Post A-Step Validation.....	794
6.1 After Tapeout and Prior to Silicon Arrival.....	794
6.2 Training.....	794
6.3 Post-Si Tools Readiness	795
6.4 A Change of Mindset	795
6.4.1 Expectations	796
6.5 System Debug	796
6.5.1 SV Debug	796
6.5.2 CV Debug	797
6.5.3 Serious Trust Issues (and why you should embrace them)	797
6.5.4 Useful Skill Sets.....	798
6.6 Pattern Debug.....	799
6.7 Speedpath Debug (and why we're not good at it).....	799
6.8 Low Yield Analysis (LYA).....	800
6.9 Fault Grading	800
6.9.1 Useful Skill Sets.....	800
6.10 Stepping Validation.....	801
6.11 Fire Fighting.....	801
6.12 War Stories.....	802
6.12.1 The Prescott A0 Test Chip.....	802
6.12.2 The WMT ITLB Global Bit Problem – A Cautionary Tale	802
7 Summary.....	804
8 Future Work.....	804
9 References.....	804

4 Purpose

In order to help meet project goals, portions of the Pre-Silicon team are often loaned out to assist the Post-Silicon teams after tapeout. Pre-silicon team members can be on loan for a considerable time and can have a significant impact on the validation effort. The resulting impact to the Pre-Silicon team is reduced headcount and increased workload—everyone on the team gets a bigger piece of the pie.

When the Pre-Silicon team members enter the Post A-step world, everyone gets involved in either teaching or learning. Before Validation begins, the tools for the validation effort need to be in place. There is a new mindset: things run faster, there is less observability, and changes are harder to make. The areas of validation and debug in which Pre-Silicon Validators assist include SV debug, CV debug, speedpath debug, pattern debug, low yield analysis and fault grading. Each area has its own set of problems and required skill sets.

5 Background Concepts

5.1 Post-Si versus Post A-Step: The Splintering of the Pre-Si Validation Team

Before bothering with the details of what goes on after that tiny, tiny chip that you've been slaving so hard to validate for months arrives, it's important to set some expectations. There is a subtle but important difference between *Post-Si* and *Post A-step*. In an ideal world, there would be a clean transition of roles and we could all happily debug to our hearts content. Once our hearts were content, we would move on to the next project. However, as you have probably figured out by now, Pre-Silicon Validation is far, far from an ideal world.

The arrival of silicon does not mean that the entire Pre-Silicon team pulls up stakes and heads off to help the Post-Silicon team in the Post-Si world. Once the designers have had about a week to breathe for the first time in months, they have a bad habit of going back to their usual role of creating bugs to help us keep our jobs. If the Pre-Silicon Validation team left them to their own devices, the next stepping of the chip might look worse than the first.

However, the Pre-Silicon team has developed a number of critical skill sets that are invaluable to Post-Silicon tasks. As a result, the success of the project depends on being able to loan team members out, often for long periods, to help the Post-Silicon teams get a grasp on the incredible complexity of our micro-architecture.

Instead of entering the Post-Si world, then, the team instead enters the Post A-step world, where Post-Silicon tasks **and** Pre-Silicon tasks for the next stepping run concurrently and the effort often spikes alarmingly.

Much of the team is scattered to the Post-Silicon winds that often blow from all directions at once. Whoever remains is left with the job of validating changes for future steppings in a model we call *reduced headcount validation*; everyone gets a considerably bigger chunk of the pie to watch over. Even senior Validators are not able to be experts about all of the areas they are validating—a scary proposition.

There usually are not many changes but since any change can have subtle and far-reaching implications, there is a large task in front of everyone once the team enters the Post A-step world. Many small changes can become numbing after a while, but it is extremely important to be vigilant with each of them.

And even once you have settled into a steady, if nervous, rhythm, there's a good chance you'll be immediately yanked off of what you're doing to help with whatever has climbed to the top of the project priority list.

Welcome to the Post A-step world.

OK, enough of the scare tactics. The work does wind down over time, and eventually, the team gets back together for the next Pre-Silicon push, a little wiser to the ways of the world.

The rest of this chapter explains the various roles that the Pre-Silicon Validators play in Post-Silicon, including the nebulous phase between tapeout and silicon arrival. The following sections go into detail on what to expect for some of the areas in which Pre-Silicon Validators are most involved, including the new focus of the portion of the team still working on Pre-Silicon Validation.

6 Post A-Step Validation

6.1 After Tapeout and Prior to Silicon Arrival

In every project, there is a bit of a lull between tapeout and silicon arrival while the fab gets their chance to shine. Some of this time is usually used for a well-deserved breather. However, there are also a number of important activities and deliverables for the Pre-Silicon team.

6.2 Training

The most common activity in which just about everyone gets involved—either as a teacher or as a student—is training.

For Validators who are entering their first Post-Si experience, there is usually an overwhelming schedule of classes on the tools of the debug trade: array dumps, micro-breakpoints, ITP, logic analyzers, even kernel debuggers for those feeling brave. There are separate tracks available for pattern and speedpath debug, and there are micro-architecture classes for the Post-Silicon team to catch up. In general, conference room scheduling is blitzed during this time.

In addition to taking all these classes, the Validation team is also responsible for creating or helping with some of the classes. The architects own the training sessions for the micro-architecture, but we often pitch in. The Validation team contains the preeminent experts on test writing, so we usually give classes (sometimes informal) to help out for fault grading. Validation tests are usually used for the *functional* (chip execution, as opposed to just array/circuit testing) portions of the initial tester patterns. Finally, those team members loaned out need to train whoever is replacing them on how to run their tests and on details of their portion of the machine.

6.3 Post-Si Tools Readiness

Hopefully, this activity has long since started. One of the keys to tool health on first silicon is early engagement between the Post-Silicon tools team and the Pre-Silicon Validation team. There is a lot of development involved for both teams to support all of the tools used on an Intel microprocessor. There are a number of advantages to close coordination and an early start:

- The earlier it gets started, the more likely you are to find the inevitable RTL bug buried under a mountain of tool bugs while there's still some chance it can be fixed.
- Creating the Post-Silicon tools early can allow these tools to be used for Pre-Silicon Validation of the DFT/DFD features. This avoids duplicating the effort (done once by the Pre-Silicon team, then abandoned and done again the way the Post-Silicon team needs it), and also greatly increases the health of the Post-Silicon tools, since all assumptions have been checked and the tools in general are more mature.
- The team with the strength in the area the tool addresses can develop tools. For example, Pre-Silicon Validators spend a lot of time dealing with signal paths and the model, so owning the array dump libraries makes sense. However, they have much less experience writing clean/compatible PSMI handler asm code or software DLLs, so the Post-Silicon team should assume those roles.

Even with a healthy head start, there are usually some ongoing activities for tool readiness during this time. The Post-Silicon tools to which the Validation team contributes include:

- PSMI (in cores)
- VISA
- Northpeak
- Array freeze and dump
- Micro-breakpoints
- Miscellaneous parser scripts

All of these are discussed in depth in the [Interaction with Post-Silicon Validation](#) chapter.

6.4 A Change of Mindset

Along with all of the training and tool readiness, it is important for the team to adopt a bit of a new philosophy on life. Anyone involved with debug will be going from a world where it takes an hour to run 1000 bus clocks to a world where that same amount of work is done in 1/200,000 of a second. However, you can see almost nothing about what is actually going on while it does run. It runs, it completed, it broke, and all you have is a test complaint, a hung part, or a system crash to go on. This takes some getting used to. The specific skills to develop are discussed later in this chapter.

Any changes that are going into the design at this point are generally fought tooth and nail by the control boards and treated with an extreme level of paranoia. Since we are about to start selling

the chip at this point, the experienced part of the team is hesitant to do anything to the chip unless necessary.

Where fixes are unavoidable, the creativity of the team is tested. What is the smallest change that can possibly be made? Can the fix be implemented using a change in just a metal layer, which is considerably quicker and less costly to turn around? Can a microcode patch or a functional defeature be used to work around the bug? Microcode and pcode patches (see [Microcode Validation section: Microcode Patch](#)) can be surprisingly powerful to work around real hardware problems. In general, the solutions to bugs take much different forms than they do in Pre-Silicon, and are often remarkable in their ingenuity, if not elegance.

6.4.1 Expectations

As part of the mindset change, get ready for bugs in your unit. It is best to embrace them as ‘opportunities for learning.’ At the very least, do not get despondent when they do show up, because they will be there, and you will be needed to help debug them.

Also, expect your new Post-A step life to consist of a lot of fire fighting. There will be more on this later.

6.5 System Debug

Since this is the group where Pre-Silicon is most heavily involved (*although fault grading has been making a run at the title for the last couple of projects*), we will start here.

Validators often get their first genuine ‘black box’ experience in system debug, and it can be a bit of a shock. Even with all of the tools described in section 6.3, it can still be next to impossible to figure out what the heck is going on inside that devilish little piece of silicon. You can consider yourself incredibly lucky if the only issues you encounter are actual logic bugs. These are generally buried beneath so many test problems, software problems, circuit problems and tool problems that the less intrepid debugger could get very depressed and cynical. Actually, the career debuggers are almost universally cynical, but we will get to that when we talk about trust in section 6.5.3.

6.5.1 SV Debug

System Validation is separated into two primary groups: SV and CV. SV is a team that is similar to the Pre-Silicon Validation team in that they generate their own test content for the express purpose of beating the tar out of the chip to go find bugs. The tests are generally short to medium length (Cafe tests complete in fractions of a second; longer algorithmic tests typically take a couple of minutes), and all of the code is known and accessible. This a key distinction discussed more in [Interaction with Post-Silicon Validation](#) chapter.

SV debug typically progresses in a series of steps once a failure is observed:

- Initial triage
- Gathering a PSMI trace, array dumps, or VISA traces

- Replay on emulation, if possible, and dump/trace analysis
- Bug filing, workarounds, signature analysis

This process is discussed in depth in the [Interaction with Post-Silicon Validation section: Silicon sightings, hardware bugs, workarounds, FIBs, and escape analysis](#) section.

6.5.2 CV Debug

Compatibility Validation (CV) consists of running the new lamp against every software program they can find to test that it behaves like the old lamps and hopefully faster.

CV debug is typically done within an unknown code space (for some reason, most software developers get a little touchy when you ask for their source code). Because the code cannot be analyzed easily to help debug the problem, a wider range of debug hooks are used. These hooks include ITP, array dumps, logic analyzers, and creative micro-breakpoints and patches.

6.5.3 Serious Trust Issues (and why you should embrace them)

As mentioned previously, experienced debug engineers are universally cynical—the unofficial motto for the debug tools team has become “how did this ever work?” This is actually a healthy state of mind relative to Post-Silicon debug. You cannot trust anyone else’s observations; you cannot trust your debug tools; and you cannot even trust yourself from before lunch to when you come back after lunch. Trust me, for now. Once you have done this long enough, you will not trust me, either.

The problem is that system debug often involves debugging almost everything at once. The software or test could be wrong; the debug tools could be wrong; the chipset could be wrong; and the processor could be wrong. Even with a high threshold set by the SV team to open a *sighting*—a Post-Silicon record of a failure that has a better-than-fair probability of resulting from a silicon bug—most sightings resolve to issues outside of the processor. Over the course of the original Pentium 4 Post-Silicon debug, only 21% of the sightings were ultimately resolved to silicon bugs and an eighth of those were already caught by Pre-Silicon.

Ask yourself a lot of questions, for example:

- Is the tool that showed me a value that I am basing my theory on reliable?
- Is there some other way to confirm that value?
- Can I reproduce the failure reliably?
- Does more than one part exhibit the problem?
- Do previous steppings or previous processors also fail with this test/software?
- Does the speed the part is running at affect the problem?

And on, and on, and on... paranoia pays.

Challenge everything you see. Record all of your observations so you can verify or reject them with confidence later, and never run too fast to a conclusion. You will be wrong more than you are right.

6.5.4 Useful Skill Sets

Along with paranoia and cynicism, there are number of useful skills for system debug that dovetail with Validation strengths:

- Micro-architecture knowledge is king (all that slavish devotion to the RTL put to use)
- Knowledge of the bus protocol
- Logic analyzer experience
- TAP knowledge
- ITP experience
- Perl/Python/ITP script efficiency
- Platform/chipset understanding
- Microcode, Pcode, and patch writing proficiency
- VISA knowledge
- EMON/ubreakpoint micro-architecture understanding
- Creativity to help with ‘onion peeling’ and triage
- Ability to learn quickly
- Discipline
- Data logging skills
- Ability to create a hypothesis and apply the scientific method (not always critical Pre-Silicon...)
- Patience—man, you must be patient sometimes
- Ability to work well in teams—latency, not efficiency, is often the key in Post-Silicon, so sometimes large teams or even multiple teams may be attacking the same problem
- Empathy to cope with stressed-out co-workers—when the pressure is on and everyone is working nights and weekends, people are less friendly than normal. Do not take it personally.
- Finally, a long list of contacts—you won’t be the expert on everything; you will need to call in the right people to help

6.6 Pattern Debug

The next three topics, pattern debug, speedpath debug, and low yield analysis, all deal with debug done on testers. Testers are very expensive tools specially designed to exercise the chip. The Validation team spends virtually all of their tester time on the functional testers. Functional testers can drive all of the pins on the chip and emulate functional chipset behavior.

Pattern debug refers to debugging frequency-independent failures in the patterns that are generated for the testers. The patterns are derived from Pre-Silicon tests so the entire test and cycle-by-cycle expected behavior is known in advance. When the silicon chip does not match the expected behavior, at any frequency, period, the Validation team doing pattern debug can help identify the problem thanks to their micro-architectural knowledge and test writing experience.

In theory, an assignment to do pattern debug is only for a short term. There usually are not that many hard failures in patterns to be debugged. Recently, the definition of *pattern debug* has started to encompass pattern creation, test porting, and test maintenance, increasing the effort more than you would guess. The later Pentium 4 experience included babysitting trace generation and tool development that should have been completed before tapeout. This is an example where the utilization of loaned Validation resources needs careful management.

6.7 Speedpath Debug (and why we're not good at it)

Speedpath debug is primarily focused on circuits and Validation members can be useful in this process during the initial triage. They can group different scan signatures to one likely root-cause. They can help trace back a problem that was only observed in the scan chain long after the cycle which is identified to be the problem by the shroo tool.

There are a few rare birds in Validation that are good at this (they get a kick out of it, and shouldn't be discouraged from pursuing their passion). However, for the most part, as a team, **we are not that good at this and we probably shouldn't be doing much of it.**

There are few reasons for this.

First, our strength is not thinking about speedpaths. We have not done the Pre-Silicon analysis to find speedpaths and fix them. We are not familiar with the timing databases or how to use them. We usually do not have the baseline to talk very intelligently about things like timing paths through transparent latches and domino logic. Ahhh, domino logic.

Second, we are not all that good with schematics, which usually are relevant for speedpath debug. We do not know the tools, the environment, or the eight stroke mouse incantations that make navigating the schematics so easy.

Finally, maybe in large part because of the previous two reasons, we tend to get... well... disinterested in the whole process. Otherwise excellent Validation team members have been observed with glazed expressions during their tours of duty with speedpath debug.

This recommendation is consistent with observations from speedpath debug management, by the way. They do not generally ask for a whole lot of Pre-Silicon Validators when staffing projects due to these historical reasons, for better or for worse.

6.8 Low Yield Analysis (LYA)

At some point (almost immediately after silicon arrives, actually), the folks in the fab become keenly interested in this thing called *yield*. It turns out there is an awful lot of money riding on how many good die are produced per wafer.

Low yield analysis is sometimes a hybrid of speedpath debug and pattern debug. Often, yield problems cause faults that show up like hard pattern failures, similar to those from pattern debug. Sometimes, these problems cause voltage ceilings or floors in certain patterns that differ from the usual shmoo edge or other dodgy behaviors.

Pre-silicon Validators can help debug these problems in the same way they help debug patterns and speedpaths. More important, they can help to write tests to screen for the problem, or fix the tests causing the problem if the problem is not an actual silicon issue.

6.9 Fault Grading

Along with yield, the fab team also tracks DPM (Defects Per Million) very carefully, waiting for the day when the content of the tests run on the testers is sufficient in and of itself to declare a part good. The ability to use the HVM testers to screen parts, instead of needing to plug each part into a system to test it, saves millions of dollars (yes, actual money). In some cases, fault grading is the critical path to supplying the volume of parts that Intel delivers.

Over time, the Pre-Silicon Validation team has become more involved with fault grading, for a good reason: the key skills in getting high fault coverage are full-chip test writing abilities and knowing how to target micro-architectural conditions. Both of these skills happen to be exactly our areas of expertise. Although fault grading has not been the highest priority for us, the Validation team will continue to be critical to the fault grading effort, and we need to budget accordingly.

The test content the Validation team helps develop typically runs in one of two environments: the functional tester or in a special mode called SBFT (Structural Based Functional Testing). The functional tester is essentially equivalent to a chipset in its abilities. SBFT is limited to pre-loading the caches using the reduced pin set of the structural testers, and has a number of rules that must be satisfied.

The tests developed for fault grading naturally must be *silicon-friendly*. Some of the test tricks (pounders and accelerators) you are familiar with will not apply to the fault grading effort. It is important to keep this in mind during template development by trying to build in the ability to configure a template to run in a silicon-friendly mode.

6.9.1 Useful Skill Sets

Some useful skills for the fault-grading effort include:

- Full-chip test writing skills
- Micro-architectural knowledge, especially how to target specific corners of the architecture
- Tool writing and tool creation experience

- Patience
- And maybe most important, an understanding of the fault grading process and what it's really all about; DPM is the end, fault grading is just the means

6.10 Stepping Validation

After a good six pages of reading about all the Post-Silicon work the team helps with, you might have forgotten the lecture about the Post A-step world. Just in case, here is a quick reminder:

This team is still first-and-foremost responsible for Pre-Silicon Validation.

As mentioned earlier, designers and architects do not lose their knack for creating bugs once the part tapes out. They are still very good at it given the chance. As soon as they make more changes to the RTL, they will create more bugs.

Since all of the other Post-Silicon tasks have drained the Validation team, whoever is left is required to do the work that everyone else left behind—a much smaller team spread very, very thin. Whether this is the best model is definitely open for debate; but with a frozen headcount and experience with the micro-architecture concentrated in just a few teams, it's almost impossible to avoid.

The target changes for the remaining team. They focus more narrowly on the changes that are occurring, which should always be accompanied by an ECO. Team members need to adjust to be able to acquire a broader understanding of the DUT rather than the depth that was critical to Pre-Silicon. They must be nimble to respond to logic escapes and work on multiple steppings in parallel. They must have the intuition to call back an expert from their Post-Silicon work when needed and only when needed—a tricky line to define.

This subject is discussed in greater detail in the [Stepping Validation](#) and [ECO Validation](#) chapters.

6.11 Fire Fighting

As much as anything, once the chip has taped out, it pays to be flexible. Consider yourself permanently on-call and ready to respond to the next flare-up. Regardless of what you get assigned to initially, depending on how the silicon actually behaves, assume that you might be needed almost anywhere. The actions that the Pre-Silicon Validation team takes during this time can have a critical near-term impact on the company; so be ready to contribute in any way you can.

While you are off fighting fires, keep one foot in the Pre-Silicon Validation world. Keep track of changes going into the model in units you helped to validate, so you can assess their impact and give hints and warnings to the new owners. Keep running at least a few tests so that your environment and tests do not rot out from underneath you. Be sure to communicate to your manager how your time is being used and what is being asked of you. Try to avoid getting lost and utilized poorly while you're on loan as this is a common problem.

Expect there to be Post-Silicon bugs in the same areas where the Pre-Silicon bugs were most dense or where they were found late in the execution phase. Remember, there will always be some surprises, and they are seldom pleasant.

6.12 War Stories

The remainder of this chapter contains a few illustrative examples; time for the grizzled veterans to throw their wooden legs on the chair in front of them, scratch the scar on their chin, mutter “Rain’s coming again...”, and give tales of days gone by. Read them at your leisure, but skip them at your peril.

6.12.1 The Prescott A0 Test Chip

Initial tapeouts for new processors on new processes are always exciting. Adrenaline flowing, large contingents of people including some relatively highly placed managers hover around the silicon labs while the FUD network waits to run full-tilt based on the results.

Prescott (PSC), one of the last Pentium 4 processors, was no different. Initial reports of some trouble in sort testing did nothing to dissuade the hordes of debuggers, testers and onlookers from descending on the first initial peanut parts with the hope of cranking the part through to a successful OS boot (for better or for worse, always the headline maker).

However, PSC was stubborn—it certainly didn’t run through BIOS in any kind of hurry. Actually, the part appeared to go straight to IERR assertion without getting much else done.

As the night wore on, and more and more people decided to wait for the next day (which was Saturday—there are rules in the FAB that dictate that parts must be delivered on Friday afternoon), it became more and more obvious that something was pretty terribly wrong. There was no evidence of any activity on the bus. Array dumps showed that the part did not make progress past the first few uops of reset. Finally, although the scan dumps weren’t working yet, a DFT feature confirmed that the part had retired exactly two uops—no more, no less.

Eventually, it was discovered that an oversight in the formal verification process had led to a number of schematic errors which caused the retirement logic to fail to progress past any uops which weren’t initially allocated. Since the PSC Microcode Sequencer only issued at most two uops, that was basically the hard cap. For all sane intents and purposes, the chip was dead.

All of which left the team in a bit of a bind. In order to make schedules, there was no way to simply wait for the next metal stepping to fix the bug; too much time would be lost, and if there was another critical, blocking issue lurking, the processor would slip, possibly missing its window entirely. The only other option was to try to do everything imaginable to exercise the rest of the chip in spite of the schematic issues.

What this implied is that every conceivable concoction of DFT hacks was tried at some point during the next three weeks.

6.12.2 The WMT ITLB Global Bit Problem – A Cautionary Tale

A few quarters into Post-Silicon on the Willamette (WMT) project (the original Pentium 4), there was still one sighting that was causing doubt as to whether WMT could go into production in Single Threaded (ST) mode. The sighting was extremely hard to reproduce, and had only been hit

in Multi-Thread (MT) mode so far, but that certainly did not exclude it from ST, since bugs are usually hit at a much faster pace in MT than ST.

The outcome of the failures was generally an unexpected INT3, but there were other “unable to duplicate” failures within the system lab that had the feel of “corrupted code”. As the sighting grew older and older, more and more pressure was placed on root causing the sighting to pave the way for production (PRQ).

After months of gnawing at bits and pieces of information, and trying all conceivable methods of capturing a failure, finally a failure was captured by PSMI that reproduced successfully on emulation and on Pre-Silicon RTL. (One of the maddening reasons this took so long is that the smoking gun – a bad paging attribute inside the ITLB – was not dumpable in Post-Silicon since the ITLB had not properly implemented array dump).

The trace was debugged on RTL. In looking at ITLB array dumps at various save points, one translation stood out dramatically in the otherwise standard styles of operating system and application translations. This translation had an address that was obviously set up for an application, and it was marked as a “user” page, another mark of being an application page. However, it also had its Global bit set, which is never true for application pages.

With a quick search to find the source of this aberration, it was found that the following logic bug existed. The ITLB kept a counter for one specific function: if, while waiting to write translation data into the ITLB, the write was stalled for 64 consecutive clocks by stores snooping the ITLB for SMC, then the counter would tell the ITLB to do the write, ignore the next snoop, and simply tell the snoop “hit”. When this case, mixed with the other thread doing a certain operation within the ITLB at the same time, the replacement would use the Global bit from the other thread. Oops.

While this bug was somewhat rare in Post-Silicon Validation, this was in fact a relatively easy case to hit in Pre-Silicon with cluster testing. Given the amount of testing done over the course of the project, it was, in all likelihood, hit. Hitting the bug, however, did not cause a failure, since there would have to be some sort of check that the Global bit was not what it was supposed to be. Such a checker, or a method for a test to expose the failure (such as in Post-Si), did not exist. Moreover, while the testplan for this unit did have conditions within this area to guide random tests towards this type of timing boundary, the testplan called for *directed tests* to hit the conditions (a failing common to early Pentium 4 Validation). What should have happened here is that a checker should have been written to catch the bug, and the random test suite should have been run to see if it found the bug. If not, the test suite should have been augmented.

None of this happened. The bug was fixed on the next stepping, and a newly written directed cluster test verified that the fix worked.

One month later, a new, even rarer bug started popping up on the new stepping. After much more gnashing of teeth and rending of garments, another trace was captured and the bug was reproduced on RTL. Sure enough, the case was almost exactly the same as the first, except that it took one extra boundary to hit the condition – a stall signal had to assert for just a cycle or two right at the time the original bug occurred – and this version of the bug “bypassed” the fix to the original bug.

This time, be assured, a full checker was written for the ITLB array, and the random tests were checked to make sure they hit the bug.

And, hopefully, that Validator learned from those particular mistakes.

7 Summary

The arrival of silicon does not mean that the entire Pre-Silicon team pulls up stakes and heads off to help the Post-Silicon team in the Post-Si world. The Pre-Silicon team has developed a number of critical skill sets that are invaluable to Post-Silicon tasks. As a result, the success of the project depends on loaning team members out to help the Post-Silicon teams. The Pre-Silicon Validation team can provide invaluable expertise in system debug, pattern debug, speedpath debug, low-yield analysis, fault grading, and stepping validation. The Pre-Silicon Validation team must be ready and willing to assist in these efforts while being careful to not be misused or burnt out in the process.

8 Future Work

9 References

The Art of Pre-Si Val: Chapter 43

Interaction with Post-Silicon Validation

By: [Michael St. Clair and David Rogers](#)

1 Abstract

The Pre-Silicon Validation team must actively work with the Post-Silicon Validation team throughout the life of a successful project. If this partnership is not well understood or neglected, then opportunities to share critical knowledge and collateral will be missed, to the detriment of Silicon quality and PRQ schedule.

This chapter details the interactions that occur between Pre-Silicon Validation and the primary Post-Silicon Validation team Pre-Silicon Validation works with – System Validation (SV). It describes the basics of Pre-Silicon Validation deliverables, shared collateral, what tasks the Pre-Silicon and Post-Silicon team should cooperate and coordinate, and what role Pre-Silicon Validation plays during Post-Silicon testing and debug.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	7/20/2005	Initial Revision	Michael St. Clair	Matt Kupperman / DPG-N uAV MWG
2.0	12/22/2005	Completed Level 3 Review.	Michael St. Clair	Paul Schwabe
3.0	4/20/2016	Revised to reflect latest SoC and Client methodologies and modes of partnering with Post-Silicon Validation	David Rogers	Michael Bair, Maria Pineda, Erik Samuelson

3 Contents

1 Abstract.....	805
2 Chapter Revision History	805
3 Contents.....	806
4 Purpose.....	808
4.1 Why do we need this chapter?.....	808
4.2 What does this chapter cover?	808
4.3 What does this chapter not cover?	808
5 The Pre-Silicon and Post-Silicon partnership	808
5.1 Post-Silicon partners.....	808
5.2 Pre-Silicon Validation deliverables.....	809
5.3 Mode of work	809
5.4 Engage early.....	809
5.5 Tours of Duty	810
6 Areas of collaboration	810
6.1 Test plans	810
6.2 Communicating and addressing areas of risk.....	811
6.3 Test content	811
6.4 Debug features	812
6.4.1 VISA.....	813
6.4.2 Micro-breakpoints and triggers	813
6.4.3 Array freeze and dump	813
6.4.4 Defeatures	814
6.4.5 Firmware patches	815
6.4.6 Miscellaneous debug scripts.....	815
7 Silicon Validation	816
7.1 Stages of Silicon Execution	816
7.1.1 Silicon readiness.....	816
7.1.2 Silicon Power-on.....	816
7.1.3 Execution	817
7.2 Silicon sightings, hardware bugs, workarounds, FIBs, and escape analysis .	817
7.2.1 Silicon sightings	818

7.2.2	Silicon hardware bugs.....	818
7.2.3	Bug workarounds.....	819
7.2.4	Silicon FIBs.....	820
7.2.5	Silicon escape analysis.....	820
8	Summary.....	820
9	Future Work.....	821
10	References.....	821

4 Purpose

4.1 Why do we need this chapter?

Validation is not complete when a project tapes in. It enters a new, even more challenging phase once Silicon arrives. Successful power-on and Silicon execution relies on establishing a strong partnership with your Post-Silicon Validation counterpart well before tape-in. Enabling Silicon validation to execute effectively and quickly find the next level of bugs is key to fixing any known issues and minimizing the number of steppings required to reach PRQ.

4.2 What does this chapter cover?

This chapter describes the activities that are part of an effective partnership between Pre-Silicon Validation and Post-Silicon Validation throughout the life of a project.

4.3 What does this chapter not cover?

This chapter does not cover the details of Silicon debug features, the many disciplines that exist Post-Silicon, or how to debug Silicon failures. See the [Design for Debug Validation](#) and [The Post A-Step World](#) chapters for related information.

5 The Pre-Silicon and Post-Silicon partnership

As mentioned in [The Post A-Step World](#), if the Pre-Silicon team does not engage with the Post-Silicon team well in advance of first tape-in, the lack of knowledge sharing and collaboration will permit bugs to survive long into Post-Silicon execution. At a minimum, both teams risk a loss in efficiency, and most likely a slip in schedule. In the worst-case scenario, late Silicon learnings may lead to additional steppings.

It is very important for the Pre-Silicon Validation team to proactively take responsibility for working with the Post-Silicon team on everything from writing test plans, to sharing expertise, to developing validation collateral, to validating debug hooks, to debugging Silicon sightings.

The Post-Silicon team has a lot to offer as well in the way of validation assistance in the Pre-Silicon timeframe, as well as providing knowledge of real-world platforms and usage models.

5.1 Post-Silicon partners

The Post-Silicon team is comprised of many different organizations, each having very specialized roles and responsibilities needed to take microprocessor Silicon and its encompassing platform to PRQ. [The Post A-Step World](#) provides a fuller overview of Post-Silicon teams.

This chapter deals primarily with interactions with the System Validation (SV) Post-Silicon team. The SV team plays a role and employs methodologies on Silicon that most closely resemble the role and methodologies of Pre-Silicon Validation. The SV team runs a combination of synthetic

tests and OS-based content. The SV team is responsible for executing a Post-Silicon Validation test plan and is a driver of resolving Silicon sightings by debugging failures to their root-cause – test issues, OS, software drivers, IA FW (BIOS), IP microcontroller firmware, platform, and microprocessor bugs.

5.2 Pre-Silicon Validation deliverables

Regardless of the degree of collaboration, Pre-Silicon Validation is responsible for delivering the following fundamental items to Post-Silicon before Silicon arrives:

- Pre-Silicon Validation test plan
- Healthy, fully functional RTL
- Validated debug hooks and features
- Communication of risk areas, known bugs, and Pre-Silicon gaps that need greater attention on Silicon

5.3 Mode of work

Collaborative work models have historically ranged from limited periodic interactions to co-ownership of Pre-Silicon and Post-Silicon activities via Validation Joint Teams (VJTs). The challenges and merits of each work model are beyond this document, but in general, a more tightly coupled, highly collaborative mode of work has tremendous benefits to both organizations when done effectively.

Frequent communication and knowledge sharing in both directions is key to establishing a strong partnership between Pre-Silicon and Post-Silicon Validation. A mindset of continuous collaboration yields much greater benefits than periodic milestone-based “hand-offs”. A virtuous cycle results from an interactive, collaborative work model. Frequent communication and BKM sharing leads to a greater understanding of common problems, which leads to identifying common solutions, which leads to shared collateral, which leads to even greater interaction, greater expertise, and so on. When Pre-Silicon and Post-Silicon Validators understand each other’s domains, they are able to optimize for the whole, avoid needless duplication of work, and more proactively resolve issues before they become problems that may lead to another stepping.

5.4 Engage early

Pre-Silicon Validation activities typically begin much earlier than Post-Silicon activities. Accordingly, it is the Pre-Silicon Validator’s responsibility to engage with Post-Silicon Validation and other platform partners such as BIOS, FW, and OS teams as early as practical. Because some teams may not start working on a project until just before tape-in, Pre-Silicon Validators often need to be an active driving force to bring all parties together early enough in execution so that everything will complete in time for first Silicon.

When interactions are early and frequent, Pre-Silicon and Post-Silicon Validators have an opportunity to work together on many items that will improve the quality of validation in both domains.

5.5 Tours of Duty

One of the best ways to share knowledge and expertise is to have Pre-Silicon and Post-Silicon Validators embed a number of Validators in each other's teams for an extended rotation of 2-4 quarters. No amount of communication, chalk talks, or reviews comes close to the knowledge gained from diving in head-first and experiencing the job firsthand. This BKM has been a long-standing practice for helping to ramp the Post-Silicon team on new architectures, features and IPs. Conversely, several quarters spent in the lab debugging failures in a real system running a production OS provides a Pre-Silicon Validator with a new perspective on their work and insight into how decisions made early in Pre-Silicon can have unexpected impact downstream.

6 Areas of collaboration

6.1 Test plans

As outlined in [Testplan Writing](#), test plans should be written and reviewed in a way that the Pre-Silicon test plan and Post-Silicon test plan complement each other and cover feature validation end-to-end. The test plan does not need to be “unified” and share the same test conditions between teams since that adds logistical and methodological complexities. It is helpful, however, for Pre- and Post-Silicon teams to use the same test plan database and organize their testplans in a similar way to permit ease of comparing test conditions across each plan and better discussions of condition-specific details.

Pre-Silicon Validators should include their Post-Silicon counterparts in test plan reviews, and vice-versa. When reviewing test plans, Validators should look for conditions that are lacking consistent validation across Pre- and Post-Silicon, missing from one plan or the others, or have unnecessary duplication of validation. Pre- and Post-Silicon test plans should be complementary; each plan should mitigate risks that may exist in the other.

Post-Silicon Validation test plans contain conditions derived from higher business unit functional requirements that come from system level or software level use cases. These conditions are, from a Pre-Silicon perspective, high-level and often vague since they are defined without firsthand knowledge of the underlying processor implementation. In contrast, Pre-Silicon Validation test plans are derived from architecture and design specs and a detailed knowledge of the processor implementation, but often lack a connection to how features may (or may not) be used by various OSes and SW stacks. As a result, it can be difficult to translate use cases into feature or function-specific test cases. However, there is great value in being able to do so, since it helps to ensure both validation teams are correctly targeting functionality important to the end-user. The process of driving platform use cases down to executable Post-Silicon test conditions may highlight validation gaps, identify implementation misunderstandings, or help define testing required on Silicon to hit key project objectives. It may require several rounds of Q&A and clarification of the platform level POR until all parties have a consistent picture and agree upon what should be validated.

6.2 Communicating and addressing areas of risk

As processors have increased in complexity, added features, incorporated new or highly modified IPs, and began operating on very short execution schedules it has become increasingly difficult to identify everything that must be done for changing areas and develop effective strategies on how to test it. This is where Pre-Silicon Validation provides beneficial insight to Post-Silicon Validation.

Certain micro-architecture enhancements, such as adding extra buffers to the memory unit or enhancing the branch prediction algorithms, tend to be easy for Post-Silicon teams to test. Existing tests will cover most if not all of the new boundary conditions introduced, and the Post-Silicon Validation team only needs to be provided with a general notion of what to target.

On the other hand, enhancements to the instruction set or completely new IP blocks require early engagement by the SV team to develop test content and mechanisms to test the functionality in the platform. Pre-Silicon should engage with Post-Silicon early in the project in order to influence the development of test content and technologies that are capable of validating and debugging the new features. The Pre-Silicon team should schedule dedicated reviews with the Post-Silicon team (along with the architecture and design teams) to discuss usage models, validation strategy, debug features, and testing limitations.

Pre-Silicon Validation should also provide input that helps the SV team to meet the challenges of validating the new processor. For example, Pre-Silicon should communicate the complexity of each functional area and provide guidance as to which areas will require greater effort and which areas can get by with less.

Perhaps the most important aspect of communicating risk is to have ongoing conversations at every stage of the project. It starts with understanding and reconciling each team's testplans, and continues throughout execution culminating with Validation 1.0 exit reviews that clearly document what was done or not done Pre-Silicon, highlight buggy areas, and contain recommendations as to what must be covered Post-Silicon. Conversely, Pre-Silicon Validators should review Post-Silicon power-on plans and discuss which features must be checked out as part of SoC power-on exit.

6.3 Test content

There are benefits to sharing validation collateral across Pre-Silicon and Post-Silicon environments. Beyond the development effort savings that comes with common test collateral – one test generator with shared libraries should cost less to develop than two independent generators – there are also quality and knowledge sharing benefits. Having two teams of Validators build upon each other's work can lead to a more comprehensive functional base and greater richness in testing for both domains. Moreover, using the same tools provides a common frame of reference and increases the degree of interaction between teams.

Common test collateral is a win for the Post-Silicon team because it allows them start with test content that is already healthy and tuned to work with the specific quirks of the processor they are about to test. Equally important, running Pre-Silicon can provide an early and more accurate sample of the coverage that the Post-Silicon tests will be able to obtain. Traditionally it is very difficult to measure coverage during Post-Silicon testing. Running and tuning Post-Silicon tests

during the Pre-Silicon timeframe allows the Post-Silicon test writers to learn their test engine strengths and weaknesses and do early tuning of the engine to target poorly covered areas.

Sharing collateral, however, is easier said than done due to differences in the validation environments, methodologies and goals. For example, there are a number of conditions around IO IPs that cannot be well tested on emulation and rely on different types of collateral (transactors on emulation vs. test cards on Silicon). The Pre-Silicon environment favors test content that is shorter, more directed, and more easily regressed on simulation and emulation. Post-Silicon may be able to reuse some Pre-Silicon content for initial SoC power-on checkout, but generally needs to quickly move on to running longer, more stressful, and more random synthetic content as well as OS-based content with real-world workloads. Sharing of collateral should be done where there are clear benefits to both teams, and should not be forced when validation usage models or schedules required different solutions optimized to the specific needs of each group.

The SV team runs a combination of synthetic test content and OS-based content. Synthetic test content, often referred to as “bare-metal”, is typically used to target specific IPs, features, or global flows in a direct or semi-directed way. Synthetic tests are typically short and complete within a few seconds. OS-based content consists of applications that run on top of a full software stack. Applications range from targeted code snippets to full real-world workloads. Most OS-based content is run on production or pre-production versions of commercially available OSes such as Windows, Android, and various flavors of Linux. All SV content, both synthetic and OS-based, is run on top of a contoured Integrated FW Image, or IFWI, consisting of BIOS, drivers, and IP firmware.

6.4 Debug features

Pre-Silicon Validation plays an important role in the definition of features and functionality that aid Silicon debug. Validators should become familiar with the debug hooks for their area and work with Silicon debug experts to ensure they are sufficient for effective Silicon validation. For example, tracing hooks create messages containing feature-specific information (such as Power Control Unit actions or Fabric messages) that are routed to aggregators within the processor that are accessible by tools on the platform for debug use.

Just as with any other functionality, debug features are likely broken unless a Pre-Silicon Validator has demonstrated otherwise. Debug features should be an integral part of any area’s validation plan and be validated along with all other features. There is a pitfall of validating debug features last, after all other functionality has been demonstrated to work. Do not fall into this trap – complete debug feature validation during Val 0.8 so that fundamental bugs may be fixed in time for tape-in. Bottom line: Ensure the primary signals you need to debug are available via debug hooks, get to know your Silicon debug experts, and ensure your debug features are healthy – unless you wish to spend your nights and weekends staring at a logic analyzer in the lab.

For a more comprehensive overview of debug features beyond the items listed below, see [Design for Debug Validation](#).

6.4.1 VISA

At the time of this writing, VISA (Visualization of Internal Signals Architecture) is a major debug feature that exists throughout much of the design. VISA allows the state of a pre-defined subset of internal signals to be captured and, though a series of muxes, be sent over a standard debug interface for external debug use. It is very important for Pre-Silicon Validators to work with the design team to identify critical signals to be made accessible for VISA in the RTL 0.8 timeframe.

As an exercise during your next Pre-Silicon debug session, try to root-cause the failure using only the signals available to VISA and see how far you get. Is there anything critical missing? If so, get it added. Having access to the right signals could be the difference between being able to root-cause a Silicon sighting in a day vs. spending a month in the lab running experiments.

6.4.2 Micro-breakpoints and triggers

Micro-breakpoints are special actions that can be programmed to take place based on micro-architectural events measured by the EMON (Event MONitoring) registers. When triggered, micro-breakpoints stop execution within the machine to allow further debug of the state at this suspended point in time or allow further actions to automatically be carried out in parallel with test execution. The inserted actions can be anything from taking a scan snapshot, to accessing a control register, to asserting external BPM (BreakPoint Monitor) pins. The events are pre-defined conditions in the EMON registers; some of the more commonly used events include an LIP match in the instruction decode pipeline, a UIP match in ucode, or the triggering of the ultimate livelock detector.

When defining EMON events early in the project, it is important to consider their usefulness to Post-Silicon debug as well as their role as performance counters. EMON events that are the most useful Post-Silicon, typically involve a unique event – for example, a uip match to a microcode event handler, or a livelock detector triggering. Pre-Silicon Validators should review the proposed events and suggest additional useful events. While validating EMON events, Pre-Silicon Validators should keep in mind how the events will be used Post-Silicon, not just how it was defined Pre-Silicon. Failing to consider Post-Silicon usage is a common pitfall that can lead to some serious forehead smacking once Silicon arrives.

Whereas some IPs such as cores and graphics have micro-breakpoints (Core, GT), others have triggers connected to the RTB (Regional Trigger Block). The mechanisms of micro-breakpoints and triggers are conceptually similar, although each IP has its own functionally-specific details.

Since the whole mechanism can be quite unwieldy to put together and get right, it is important to have a script that sets default values correctly and abstracts control register details. This script is typically owned by the Post-Silicon team. A useful Pre-Silicon exercise is to use the script to set up various trigger events and then verify results by running tests on simulation or emulation.

6.4.3 Array freeze and dump

Array freeze and dump is a mechanism used to access the state of all of the arrays on the chip to help determine what was going on during, or unfortunately, more typically after, a failure. Array freeze and dump is particularly effective in cases where the system hangs. It uses the same DFT

mechanisms that are used to test the arrays on the tester, but uses them only for observation of the unknown state values in the system. This mechanism is often employed to debug core sightings since the cores employ many different arrays that provide breadcrumbs during the debug process.

The mechanism used during array freeze is to assert a signal which is distributed globally on the chip that tells all arrays to stop updating their contents (the “freeze”), and then after the freeze has occurred, use DFT hooks to read out the current state of the arrays (the “dump”). The freeze provides a consistent snapshot in time across all arrays, which is particularly important in the context of renamed state. For the arrays which are part of reconstructing the renamed state, exact freeze timings are given and pointers are also frozen so that a set of uops are consistently allocated, retired, or invalid across all of the arrays.

The states recovered using array freeze and dump are combined with capture of the scanout nodes and reads of all control registers as input to Post-Silicon debug visualization tools. This may be employed to triage a Post-Silicon failure, determine what is going on, whether we have seen it before, and what could be used for a workaround if it looks like a processor bug. In some cases, freeze and dump in combination with ITP may be all that is necessary to debug a problem. For example, by repeating scan captures with an exact delay from an event like a livelock breaker firing, you can in some cases see all of the elements of a livelock bug. On some bugs where it is difficult to capture a PSMI trace (another core-centric Silicon debug feature), it may be the only way to debug, and there is no shortage of grizzled veterans who can tell you how much “fun” that can be.

The Pre-Silicon DFT team does most of the heavy lifting necessary for the array freeze and dump mechanisms to work. They validate that the DAT mechanism for each array works, and that the DAT controller itself works.

Each IP or IU Validation team is responsible for validating that:

- All arrays freeze their content under array freeze.
- The freeze timings are accurate. In order to be able to reconstruct the Post-Silicon state, the set of arrays that track allocation, retirement and renaming have to freeze at precise times to have consistent contents.
- The dump can successfully occur while freeze is asserted and in different functional modes (e.g. during multi-threaded execution, which is not typically used in DFT array tests).

Prior to Silicon arrival, the mappings from DFT command results to RTL node names, in the form of a Perl library for P4, have to be tested and deployed. Preferably, these are used and verified during the Pre-Silicon timeframe. These scripts should be owned by the IP or IU Validation steam, who can most effectively hunt through the RTL for the mappings of the array signals.

6.4.4 Defeatures

Although they are often given a lower priority by Pre-Silicon Validation, defeatures take on a much higher importance during Post-Silicon Validation. When developing a test plan and prioritizing what is tested during execution, it is very import to understand the risk of various new features to determine which defeatures must be validated as well as the feature itself. This could be the difference between an additional stepping or a “diving save”.

Defeatures are used in a number of ways:

- Defeatures can help to accelerate failures. If you know that a failure is exacerbated by throttling a resource, you can use a defeature to cause that throttle on a much more frequent basis. For example, if a failure revolves around a writeback conflict, you could use a defeature to reduce the number of ways in the cache in order to force writebacks to occur much more often.
- Defeatures can help to troubleshoot and prove debug theories. For example, if you suspect that a bug requires a branch prediction to be present, turning off the Branch Prediction Unit (BPU) can give you a quick verification that you are on the right track.
- In a pinch, defeatures can be used in very clever ways to work around Post-Silicon bugs. The use of defeatures to survive otherwise fatal problems may save a stepping or a post-PRQ recall. The downside is usually lost performance or increased power.
- Post-A0 bug HW fixes often contain specific defeatures designed to ensure that if there are issues with a bug fix on a stepping, an incomplete fix can be turned off so that the situation is no worse than prior Silicon and back to a well-understood functional state.

6.4.5 Firmware patches

Firmware patches, which allow you to replace or expand uCode, pCode, and PMC FW for any given instruction, exception routine, or feature, are an incredibly powerful tool for both debugging failures and working around bugs in Silicon. Some usage models include:

- Stowing away fault context relevant to the failure that would otherwise be obliterated during the long path to the blue screen of death
- Forcing serializations where they don't normally occur to test theories about bugs
- Conditionally storing information or taking a probe-mode interrupt based on the current context of the machine
- Recording context-specific information about the code flow; for example, the pattern of accesses to a particular MSR
- Finally, of course, FW patches are the primary vehicle for working around bugs for either further onion peeling or for PRQ. It has been a very long time since Intel sold a processor that did not require a production patch for a bug workaround.

6.4.6 Miscellaneous debug scripts

Debug scripts deserve an honorable mention here. Scripts created to make control register results readable or parse machine check registers will suddenly take on a whole new importance Post-Silicon. Trust me that you will suddenly gain a new appreciation of the machine check architecture, at least to figure out what exactly was logged just before the whole system crashed.

So gather up anything that operates off of obscure state and makes sense of it, write new scripts to fill in any gaps that are identified, make the scripts user-friendly, and then archive them. When working with your Post-Silicon counterpart, see if they fill a need in the Post-Silicon world. Many

of them will lead a very long and productive life through the current project and beyond to proliferations.

7 Silicon Validation

7.1 Stages of Silicon Execution

Silicon Validation proceeds through several phases of execution for each stepping. As would be expected, activities are most intense and focused on first Silicon when functionality, and different organizations, need to come together for the first time.

7.1.1 Silicon readiness

Silicon readiness overlaps with much of the Pre-Silicon Validation timeframe from early design execution to Silicon arrival. This is when the Post-Silicon Validation team writes test plans, develops test content, and participates in Pre-Silicon Validation tours of duty. It culminates with preparations for first Silicon arrival which include crafting power-on execution plans, holding focused training sessions in advance of lab work, and bringing together a myriad of stakeholders (Pre-Silicon, Post-Silicon, OS, business unit, etc.) from across the company to form Platform Virtual Teams (PVTs) that focus on specific aspects of the Silicon and platform. For example, PVTs are formed to cover each major IP such as core, graphics or memory; debug features such as TAP; and global flows such as reset or power management. Pre-Silicon Validators should work closely with Post-Silicon Validation under the PVT umbrella to define Power-on and PRQ exit criteria on a feature-by-feature, IP-by-IP basis. Validators should sanity check requirements and assumptions with all stakeholders to ensure the creation of robust and complete execution plans.

7.1.2 Silicon Power-on

Power-on kicks off with the excitement of first boot and continues until all major Silicon and platform features are checked out and verified to work as expected. It is among the most, if not the most, intense and highly focused stage of the entire project. It consists of three sub-phases: first boot, SoC power-on, and platform power-on. Power-on execution teams consist of experts from all major disciplines – Architecture, Design, Pre-Silicon Validation, Post-Silicon Validation, OS, BIOS, firmware, software – who work together as “one team” to achieve power-on goals as quickly as possible.

During first boot, a focused team of reset and debug experts bring Silicon to life and work around the clock to complete the boot flow for each POR operating system. This requires working through a host of process and part-specific issues, as well as software stack issues. Initial boot of each OS is often accomplished through the use of defeatures and creative workarounds via debug patches and scripts. Once a boot recipe is achieved, the requisite “Hello World” email is sent from the operating platform, and the boot team is allowed to get a few hours of sleep.

SoC power-on is owned by the Post-Silicon Validation team and consists of checking out all major SoC features and functionality. As parts become available, they are distributed to each of the PVTs

to execute their power-on checkout plan. This typically consists of several dozen fundamental features and use cases that demonstrate basic functionality. SoC power-on is performed primarily with synthetic content, but may include some targeted OS-based content when OS drivers are required to check out Silicon functionality. In addition to checking out features, each team works to establish a configuration and workaround recipe for the wider team to use. SoC power-on exit is declared once all PVTs have enabled expected functionality or have identified blocking bugs that prevent further checkout.

Platform power-on is owned by each OS platform team. Although the focus and ownership changes from Post-Silicon Validation to OS platform teams, the same diverse execution team remains involved, merely shifting between driver and support roles. Platform power-on builds upon the basic functionality demonstrated to work for the SoC, and proceeds to check out the entire platform, including BIOS, firmware, drivers, OS, and associated platform hardware. As can be imagined, pulling together an entire software stack to enable new IPs and features for the first time requires considerable onion peeling and collaboration between everyone involved. Platform power-on exit is achieved once all major IPs, features, and use cases are demonstrated to work for each OS or issues are identified that block further checkout.

7.1.3 Execution

Once platform power-on exit is declared, Post-Silicon enters a phase of general execution where validation teams complete their broader test plan execution and platform teams complete their checkout of all identified use cases. Activities tend to revolve around resolving Silicon sightings generated by the validation and platform teams. Silicon moves beyond the lab to many internal customers who require healthy and extremely stable platforms. Silicon yield and functional health coming out of sort (a manufacturing step which runs functional tests on each die while still on a wafer that identifies healthy parts and provides process feedback) are driven to a point where engineering samples (ES1, ES2) may be shipped to external customers, which usually generates another round of learnings and identifies additional areas of validation focus.

In advance of PRQ, qual samples (QS) are shipped to customers for final testing and feedback. In this timeframe, task forces are formed to resolve sightings that gate QS or PRQ. As with power-on activities, the expertise and insight of Pre-Silicon Validation is often instrumental in helping to quickly troubleshoot and resolve the failures each task force goes after.

Once all test plan conditions and use cases are covered, PRQ-gating sightings have been resolved, and task forces have fully understood all critical issues, the Post-Silicon Validation team, along with all other organizations involved give a “Go” for PRQ.

7.2 Silicon sightings, hardware bugs, workarounds, FIBs, and escape analysis

Just as with the life of a Pre-Silicon bug, there is a general flow for debugging, fixing, validating, and learning from bugs that exist in Silicon.

7.2.1 Silicon sightings

A sighting is a database record that documents the details of a particular Silicon failure as it is being debugged. Sightings aid clear communication and accelerate debug by providing a single location to document actions being taken to resolve a Silicon failure. Experts can often debug straight-forward problems within as little as a day, but complex or difficult-to-reproduce failures may take several weeks to resolve. Resolving difficult sightings may require the input of many experts from around the world, ranging from the Validators in the lab to HW design, Architecture, SW, and Platform experts. Sightings are closely tracked and reviewed regularly in debug forums established for various domains. Processor bugs are tracked in an SoC sightings database, whereas software and platform bugs are tracked in a Platforms sightings database.

The Post-Silicon team takes a number of steps to determine if a failure is real and unique. For example, a failure should be reproduced on multiple platforms before a sighting is filed, in order to filter out platform-specific and part-specific failures. One of the big hurdles that frequently complicates the initial triage process is reproducibility. Failures in Post-Silicon are often achingly hard to get to happen in a repeatable fashion or within a short enough timeframe. Determinism can help to solve this problem, so mechanisms that aid determinism are frequently designed into the processor and validated Pre-Silicon. Determinism is never a guarantee, however, and becomes much more difficult with high-speed serial interconnect topologies.

Most failures wind up as test, configuration, or SW (platform) bugs. A small fraction of initial Silicon failures require detailed analysis and become sightings. Of these, an even smaller fraction are dispositioned as processor hardware bugs. There is a tremendous amount of work that is done by the Post-Silicon team before the Pre-Silicon team typically hears about a failure, unless they are actively partnering in the lab.

Pre-Silicon Validators are most useful during the sighting resolution phase by helping to accelerate the debug process. Pre-Silicon Validators can suggest experiments, rule known bugs in or out, help analyze failure signatures, attempt to reproduce a failure Pre-Silicon, and of course pull in additional Pre-Silicon experts as needed.

7.2.2 Silicon hardware bugs

When a sighting is resolved to a new hardware bug, Pre-Silicon Validation must become more actively involved in follow-up steps. The aftermath of finding a bug can be more work than finding the bug in the first place. A lot of that work is discussed in [Post-Silicon Escapes](#), but there is a lot of work before it has even gone that far.

The first thing that needs to be done after a sighting is root-caused to a new processor hardware bug is to file a very detailed bug report and make sure all major stakeholders are aware of both the problem and the planned response. The Pre-Silicon team often owns filing the bugs discovered during Post-Silicon debug. Whether or not a Validator was involved in resolving the initial sightings, Validators are frequently the best candidates to describe the bug in context and in careful detail, or to update the bug if the relevant details are lacking. Standards for Silicon bug write-ups are very high. Many people will be keenly interested in the problem and the breadth of its impact. So be precise; document everything that is known about the bug and how it manifests.

The next key activity, especially for bugs that cannot be worked around, is to identify a precise signature of the bug. Doing this well is extremely important, and the Pre-Silicon team can often play a critical role – especially where the signature is clearest in an array dump.

If the signature of the bug is not developed, then the team spends lots of time debugging the same bug... over... and over... until the stepping with the fix finally rolls around. On the other side of the knife-edge, if the bug signature is defined too broadly, then the shadow it casts might very well include another bug's signature. It is more than a little disheartening to discover on the next stepping that not all of those livelocks were the same bug. This happens, and not just with livelocks. So be careful – help craft the best bug signature, or signatures, to break through all the noise.

Once the team has confidence in the bug signature, it is useful in some cases to write a script that will identify the bug based on the dump state. You will get tired of staring at dumps for the same ol' bug which the Post-Silicon team will suddenly have no trouble hitting. Just be sure for these scripts that they are somewhat conservative; better to let some cases of the bug through, which adds a small amount of work for the Pre-Silicon Validator, than to misidentify a bug as its cousin and have to spin yet another stepping of the processor.

7.2.3 Bug workarounds

Post-Silicon teams that are impacted by a hardware bug will immediately request a workaround for the bug to enable progress well before a hardware fix may be available via the next stepping of the chip. The Pre-Silicon team can usually deliver one – with varying levels of impact. In fact, a workaround may be preferable to a hardware fix when the risk, impact, and completeness of a “temporary workaround” is deemed acceptable as a permanent solution. The Pre-Silicon Validation team is well qualified to suggest workarounds and assess their risk.

Often, a workaround involves a defeature. If the bug can be shown to only occur in a specific context (for example, when locks are executed on both threads in parallel), then it is very useful to be able to work around the bug with the defeature in order to onion peel or differentiate this bug from a different bug. The problem, though, is to evaluate the impact of the defeature carefully. If the defeature is “turn off all caching”, then it is probably a bridge too far as a bug workaround; the testing of the part is significantly compromised with such a large defeature enabled.

Another primary type of workaround is a firmware patch. Patches can of course fix most ucode and pcode ROM bugs; but they can also be remarkably powerful as workarounds for bugs in other areas of the chip. Adding fences in precise spots, swapping uop types, an extra flush or two, or reordering a global reset or power management flow can all help to avoid logic failures that would otherwise be impossible to work around with only defeatures.

The job of validating a hardware bug workaround is shared by both Pre-Silicon and Post-Silicon validation. In particular, the adoption of a proposed workaround as a permanent fix should usually motivate quite a bit of targeted validation from the Pre-Silicon team. In addition, it is important to immediately verify workarounds in the Post-Silicon. For a bug which is worked around by a defeature, it should be deployed in a patch as quickly and widely as possible. Defeatures very often have their own bugs associated with them. The sooner these bugs are discovered and assessed, the better.

7.2.4 Silicon FIBs

When a patch or defeature workaround is not available, a fix may be identified by the design team that directly edits the internal gates of an interconnect path on a small number of parts. This type of fix is called a FIB and is named after the Focused Ion Beam that is used to modify the Silicon gates by cutting a metal line or depositing a metal to hook up a gate differently and change its function.

FIBs are used to test out theories for program-blocking sightings as well as to test in Silicon proposed changes in the logic for a future stepping. For a logic bug that has a relatively simple fix, where “simple” is defined as a small number of gates, an option that is often pursued in Post-Silicon is to define and try out a FIB edit to test the logic fix well before a new stepping is available.

It is a remarkable technology, and quite tricky. The FIB team is actually quite good at their jobs, but you should still expect that not all FIBs will be successful, and they can also be quite fragile. Where they are successful, though, they can be instrumental in exposing problems with a proposed fix or peeling the onion past the current blocking bug.

However, as mentioned previously, a perfect layout FIB is not necessarily a perfect bug fix. Pre-Silicon Validation must run validation cycles on an RTL model with the logic equivalent of the FIB edit to ensure that the bug fix is robust and nothing else is broken by the proposed change.

7.2.5 Silicon escape analysis

Whenever a new bug is identified in Silicon, the Pre-Silicon Validation team must quickly characterize the bug, assess why the bug was not caught Pre-Silicon, and identify any actions required to prevent similar escapes in the future. Project managers are keenly aware of issues that limit Post-Silicon validation progress, risk functionality needed for customer samples, or may require an additional stepping to fix. It is very important to perform a thorough analysis of the circumstances surrounding each and every escape, and ensure the bug report escape analysis fields are updated in a timely manner.

As part of the process, it is important that Pre-Silicon and Post-Silicon Validators compare notes on individual escapes and periodically compare notes on any trends and patterns that may emerge.

Further information is discussed in the [Post-Silicon Escapes](#) chapter.

8 Summary

The Pre-Silicon Validation team must work closely with the Post-Silicon Validation team throughout the life of the project for both teams – and the project itself – to be successful. Teams that establish a strong partnership and collaborate regularly on multiple fronts will develop greater expertise, better validation collateral, and be able to respond much more effectively to any issues that arise.

Prior to Silicon arrival, the Pre-Silicon team should at a minimum develop a common end-to-end validation test plan, deliver fully functional RTL, validate debug hooks and features, and ensure that risk areas needing greater validation on Silicon are understood and addressed. Furthermore,

each Pre-Silicon Validator should seek to truly understand Post-Silicon requirements to make better decisions about Pre-Silicon Validation scope.

Once Silicon arrives, Pre-Silicon Validation should stay engaged with Post-Silicon Validation by helping to debug sightings, analyze failures, propose workarounds, and thoroughly test all bug fixes and permanent workarounds. Learnings from Post-Silicon escapes should be incorporated into validation improvements on subsequent steppings and projects.

9 Future Work

This section intentionally left blank.

10 References

This section intentionally left blank.

Post-Silicon Escapes

By: [Matt Plavcan](#)

1 Abstract

When a processor has taped-out and physical parts exist, there are still bugs present in the design. These bugs have a name: *escapes*. Escapes have made it past all methods and processes used to find bugs in pre-silicon testing. Post-silicon escape validation is the process of tracking escapes, finding their root-causes, validating their fixes or workarounds, and closing the gaps in the pre-silicon validation methods to catch these and similar bugs in future steppings.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	1/31/2004	Initial Revision	Matt Plavcan	uAV/DV Meth. WG
1.1	2/24/2004	Revision after preliminary team review	Matt Plavcan	uAV/DV Meth. WG
1.2	5/11/2004	L2 Review revisions	Matt Plavcan	Joe Matarazzo
1.3	10/27/2004	L3 Review revisions	Matt Plavcan	Steve Cegla

3 Contents

1 Abstract.....	823
2 Revision History.....	823
3 Contents.....	824
4 Purpose of Post-Silicon Escape Validation	825
5 Background Concepts	825
6 Stages of Post-Silicon Escape Validation.....	827
6.1 Transitioning from a “Bug” to an “Escape” (Entry Conditions).....	827
6.2 Debugging a Sighting, Filing the Bug (Inputs and Dependencies).....	828
6.3 Tasks	829
6.3.1 Identify and Document the Root-cause of the Escape	829
6.3.2 Identify the Fixes and Workarounds for the Escape	831
6.3.3 Implement the Validation Infrastructure	832
6.3.4 Pre-Silicon Stepping Validation.....	833
6.4 Is the Escape Really Contained? (Validation).....	833
6.5 Closing the Validation Loop (Outputs)	833
6.6 Escape Contained (Exit Conditions)	834
7 Summary.....	834
8 References.....	834

4 Purpose of Post-Silicon Escape Validation

Although a processor stepping has taped-out, bugs still exist in the design. A bug that exists after tapeout has a name: *escape*. The severity of these bugs can range from obscure corners of infrequently used features to user-visible differences in behavior of the instruction set. The pre-silicon validation team is integral in finding, root-causing, and validating bugs found in physical parts. Once an escape is found, the next stepping might require a logic change, or the current stepping might need a workaround applied to function properly.

Successful validation of bug fixes is essential. The time and money required to fabricate a stepping is prohibitively high. Failure to validate an escape properly will likely result in a similar bug in the next stepping. Depending on its nature, a bug can prevent a stepping being qualified for production, increasing the time to market (revenue shipments). Worse still is the possibility of a processor bug evading detection by the validation team, shipping to customers, and then found later. When this happens, damage can be brought to Intel's brand name and financial position by requiring a recall of the affected parts.

Escape validation has two key goals. The primary goal is to validate fixes for silicon bugs to a high enough confidence level to enable tape-out of the next stepping. The second goal of the escape validation process is to refine the validation team's ability to catch potential escapes earlier. Validators try to identify potential gaps in validation strategy or methods that allowed the bugs to escape detection during pre-silicon testing.

5 Background Concepts

Pre-silicon validation for a stepping usually winds down following a tape-out (although, validation might be in progress for future steppings). The idle period between pre-silicon testing and post-silicon escape validation lasts until first silicon returns. After the first systems are assembled, the post-silicon teams begin their debug cycle. When a silicon platform does not match expected behavior, the post-silicon teams file sightings.

Sightings describe specific failure modes on a system platform. A sighting might be a regular, identifiable problem, an intermittent system crash, or a one-time glitch in the functions of a program. Not all sightings are processor bugs: they can represent a software bug, an operating system problem, a chipset bug, or a variety of other issues outside of the silicon. For the remainder of this document, *sightings* refers only to sightings that have a logic bug in processor as a cause. Non-logic processor bugs include schematic mismatches, circuit speed-path/marginalities, or low-yield failures. The process in this document **does not apply to non-logic bugs**. Possible sources of sightings include:

- **System Validation (SV):** Runs randomized, aggressive tests across many architectural and micro-architectural directions. Their validation methods are similar to the methods employed in pre-silicon testing. SV test behavior is more aggressive than that of most typical applications. The strength of SV lies in the sheer number of machine cycles run. The cycles are orders of magnitude higher than all pre-silicon simulation runs for the project combined. Like all post-silicon testing, however, SV tests suffer from a lack of direct observation of most of the conditions they generate. The SV tests excel at finding micro-architectural collisions that require too many conditions to align to hit.

Example: One bug in Willamette required 19 specific steps to set it up. This specific scenario was highly unlikely to occur in pre-silicon testing. SV found it in the first few weeks of post-silicon testing.

- **Compatibility Validation (CV):** Runs every commercial application they can find, under a real operating system. Most often, CV finds legacy compatibility bugs where the current design diverges from architectural behavior of a previous version of a processor. CV also can discover micro-architectural bugs that reflect real usage models that are not easily created in the pre-silicon environment. CV testing often involves repeated stress testing of a particular application or operating system. CV runs are an excellent representation of the real-world environment in which the processor will be running.

Example: On Prescott, a cross-thread livelock condition involving cache locks was discovered only after tape-out, despite exhaustive testing pre-silicon by multiple validation teams.

- **Performance Validation:** Runs benchmarks and common applications to determine relative/competitive performance. Performance bugs do not cause an architectural mismatch so they are often missed in pre-silicon. Since checkers typically do not target performance bugs, they never actually fail a simulation. These bugs can be as crippling to a product as a logic failure, since they limit the ability of the processor relative to competing products.
- **Customer:** First silicon that has been determined to be relatively “healthy” is delivered to customers to qualify the processors for their platforms. Any bugs that are subtle enough to slip past SV/CV in the first few weeks of testing are likely to make it into a customer’s hands. Because these bugs are hard to catch, they can linger for a long time before being discovered and/or root-caused.

Example 1: The Pentium® 4 processor micro-architecture had a cross-thread CR2 corruption bug. The bug only failed under a specific customer’s OS, but worked in general for Windows and Linux. The debug teams from Intel had extreme difficulty in reproducing the bug. It took nearly two years to replicate the bug successfully. By the time the problem was root-caused over 100 million Pentium® 4-family products had been shipped! Luckily, a patch was able to work around this devious bug in the released products.

Example 2: The German version of Windows found multi-cycle speedpath bug (not a logic bug, but still a sighting.)

- **Pre-Silicon Validation:** Though technically not sightings, pre-silicon tests can catch the same bug a test on a system would find. After a tape-out, pre-silicon validation might still be running tests or pursuing conditions identified in their exit reviews. It is possible to find last-minute bugs in these areas that affect whether a part ships to customers.

Example: During Prescott A0 tape-out, a memory-cluster checker, written after an AR from an exit review, found a critical time-bomb-fault bug just in time to intercept the tape-out.

Projects can also find bugs that apply to multiple processors with the same base micro-architecture.

Example: The Tejas validation team found a micro-architectural collision case that corrupted a segment register. This bug escaped Prescott validation.

Once validation identifies and can reproduce a sighting, it can be debugged to find its root-cause. If the root-cause is a logic bug, the escape validation process begins.

Typically, escape validation involves only a few members of the uAV/DV team at one time. The number of validators depends on the number of open sightings, and those sightings' location in the processor. These validators act as focal points for escape validation activity. The root-cause the failures will include other validation team members as needed. The following skills are essential for a sighting validator:

- **A deep understanding of micro-architecture of the processor:** This knowledge enables them to analyze sightings and understand the conditions that expose a bug. They need to have a good comprehension not just of their own unit, but also across a breadth of protocols and functions for multiple units and clusters. Because of the time it takes to accrue this knowledge, escape validation tasks are usually assigned to senior validators. A valuable part of the validator's experience is knowledge of the history of previous bugs in the design, and understanding how failure modes point to specific design features.
- **A working knowledge of the debug techniques of [The Post A-Step World](#):** The debug of the initial sighting might not directly involve the validator. However, once a logic bug is suspected, the validator usually performs some debugging on a validation platform, similar to System Validation. An understanding of how the overall system works (chipset, other system devices) is useful. A validator who is good at system debug can leverage the knowledge of others on these topics to assist in their debug.
- **Familiarity with the pre-silicon validation process:** To root-cause the failure, the validator will attempt replicate the failure on a simulation run. The validator utilizes good test writing skills ([Stimulus](#)) to target the failure mode. Working knowledge of [Testplan Writing](#) and [Checking](#) is also important. These skills are essential not only to root-cause the sighting, but to refine any of the pre-silicon validation tasks that were implicated as sources of the escape.

An escape validator is expected to challenge assumptions regarding the design and of the validation performed on the processor. If the pre-silicon validation process previously failed to find a bug, the validator needs to pay attention to detail and documentation to insure that this bug and its causes are adequately tracked.

6 Stages of Post-Silicon Escape Validation

6.1 Transitioning from a “Bug” to an “Escape” (Entry Conditions)

An escape is any logic bug for a validated feature that exists in a physical (taped-out) part. This classification is independent of whether the bug is found in a pre-silicon model or in a post-silicon platform.

Bugs in features that are not part of the POR for the stepping and have not gone through the validation process are not considered escapes. However, they will follow a debug process similar

to escapes and will point out weaknesses in the validation of production features. A change to the plan of record (POR) of a feature after silicon is produced can cause a feature to become broken. This case is not an escape, if the logic matches the original (pre-silicon) POR.

Post-silicon escape validation begins with the confirmation of a logic bug in physical silicon.

6.2 Debugging a Sighting, Filing the Bug (Inputs and Dependencies)

The key goal in debugging a sighting is to find the root-cause(s) of the bug. A secondary goal, and side effect of the process, is to derive a failing test for the pre-silicon environment. This test will aid in debugging and eventually in validating the fix. Logic sightings are root-caused using a combination of one or more of the following methods:

Note: The specific details about debugging a post-silicon failure are beyond the scope of this document.

- **Inspection/Induction:** Sometimes the particular signature of a bug gives hints about where the root-cause lies. In these cases, the validator can examine the logic. If feasible, the validator can construct a pre-silicon directed test to target the failure quickly. Even if a pre-silicon test cannot be constructed, often a validator can describe a “what-if” case to test on the platform and narrow down the possible bug sources. These are the ultimate cases of the validator understanding the micro-architecture of their unit. Debug by inspection represents the least effort to root-cause a bug. Escapes found using this debug method often point to high-level architectural design flaws and conceptual checking holes.

Example: In Prescott, both a DAC and a SAAT bug were root-caused simply by analyzing the failure modes of the sightings.

- **PSMI (Periodic System Management Interrupt):** When a test exists to reproduce a bug on a platform, PSMI can be used to convert the silicon-platform test into one that can be run on a simulator. Bringing a failure from a post-silicon environment into the pre-silicon environment allows absolute observation of all states in the design. As a side benefit, it removes the complex interactions associated with the chipset and system platform from the debug equation. This is the preferred general-purpose mechanism for resolving a sighting. However, since PSMI is itself a logic feature that relies on software and hardware tools, it might not work. PSMI might not resolve sightings where the cause lies outside the processor, for example in the chipset. PSMI tampers with the micro-architecture while saving and restoring machine state, which can set up the conditions that create a bug. Likewise, the actions of PSMI can prevent the required conditions for a bug from occurring.

Example: In Prescott, two bugs existed that would only cause problems after PSMI had restored the processor state. Without PSMI, they would be neither triggered nor observable.

- **Defeature modes, Patches, Micro-breakpoints, EMON, Scanout, etc:** The processor boasts a variety of features for observation of micro-architectural states and for workarounds to a failure mode. These features can be extremely valuable. In some cases, they might be the only way to root-cause a bug. These methods are often useful when PSMI debugging is not functioning correctly or when PSMI is a possible source of the bug.

Note: some of these (defeatures, patches) are employed as workarounds for bugs in a production part, to avoid another stepping. Consider how these features may affect the bug, even if other means are being used for debugging.

- **Other:** New methods are always being invented; use them as needed. This document is a set of guidelines to serve as a starting point for approaching the problem, not hard and fast rules.

Once a simulation run exists, a pre-silicon validator can start to root-cause the bug. (The validator might not be the validator for the unit where the bug exists.) Any architects, designers, or validators who have relevant knowledge can be consulted. When a bug reaches this state, it is nearly identical to a pre-silicon bug, and can be debugged as such. However, it is occasionally necessary to iterate over the process. Any of the above methods might have yielded an incomplete result

Example 1: PSMI captured a trace but debug showed that the failure occurred much earlier than the PSMI segment.

Example 2: The test could show that a bug identified by “inspection” is not the true cause.

6.3 Tasks

Once the bug is conclusively root-caused, the validator files a bug in the tracking database and starts an [Escape Validation Checklist](#). The checklist contains the following:

- An analysis of why the bug occurred
- A description of how the bug got through pre-silicon validation
- A list of fixes for each effective stepping
- A description of how the escape was validated
- Steps being taken to prevent similar bugs from escaping in the future

6.3.1 Identify and Document the Root-cause of the Escape

Successful escape validation is detail oriented. Clear and precise documentation is an important part of the process. (Remember, the bug already escaped a well-documented pre-silicon validation effort.)

The validator files a bug documenting the exact conditions that caused the bug, and the conditions that could cause a similar bug. Another useful item to include is a list of similar conditions that do not cause the bug (and identify why they do not). This process helps define the boundaries of the failure mode and might point to why the bug escaped.

The validator immediately begins to fill out an Escape Validation Checklist. This document tracks the entire escape validation process. The document provides a framework for the validator to show the work done to validate the bug and close the validation hole. This includes information for the stepping where the bug was found and for future steppings as well.

Once the root-cause of the bug is known, the validator identifies the root-cause of the escape. At first, it might not be clear what the difference is between the cause of the escape and the root-case of the bug, since we've previously defined post-silicon bugs as escapes. However, the escape is more than just the bug. It includes the reasons why the bug made it past the multi-tiered validation

screening and into the physical part. This is part of normal bug validation, too, but here it takes on a higher importance, due to the escape having survived through the continuous and escalating enhancements in checking, coverage, and testing during the pre-silicon validation process. Some possible causes are:

- **Feature (micro-architecture boundary) was never tested:** For some reason, this feature was not part of a test plan, was not included in coverage, or no test was written to hit the condition. Perhaps a test existed, but was never run. The validator needs to take steps to fill these gaps.

(*Colwell's Law: "If it isn't tested, it's broken"*)

- **Emulation for feature/protocol was incomplete/incorrect:** The bug lay in an inter-cluster protocol, and assumptions or documentation about how another cluster behaved were incorrect. Improving the emulation will help close the hole. Allowing for more aggressive randomness in a protocol than actually occurs in the fullchip design can flush out similar bugs. Sometimes certain interactions are not tested for a given environment, because they are better suited to be tested elsewhere (for example, microcode is not present in many cluster environments, and is handled at fullchip).
- **Checking hole:** A checker that was required to catch the failure was not implemented, did not address the specific case, or the checker itself had a bug. Writing good checkers prevents coding the same bugs into the checker that is present in the RTL design.

(*Bentley's Corollary to Colwell's Law: "If it isn't checked, it's broken"*)

- **Escape maps to a low-coverage area:** The bug lay in a feature area that had a test plan section, was tested and checked, but high coverage was not collected. During the validation exit review, the area was deemed lower risk, and no further action was taken. The assessment was wrong: there was a bug. The assumptions used to justify this and other “low-risk” decisions need to be re-examined.
- **Design Complexity:** The bug required a large number of conditions to align. These types of failures can be caught in a pre-silicon environment, but tend to be “lucky hits,” even with good random testing. Bugs like this either require a huge variety of tests to be run to find the single combination, which results in a failure, or the bug requires a complex accumulation of machine state to set up the bug. The number of cycles required to hit these states is more than pre-silicon validation will run. This testing is easily achievable in post-silicon testing.
- **Validation Complexity:** The multiple feature interactions necessary to create this bug were too complex or disparate to be addressed well during pre-silicon validation. There are limits on the type of pre-silicon testing that can be performed. Sometimes key knowledge about how various features interact might not have been available or apparent to the validator.

Several escapes in Prescott revolved around an interaction of loading segment registers with null selectors in 64-bit mode. The same logic had been validated and worked correctly for 32-bit modes, but due to several micro-architectural collisions, the 64-bit case took multiple steppings to work correctly. Detailed examination of this logic by several knowledgeable

designers and validators failed to root out all the possible problems, even after the first bugs in this area were found.

- **Other:** Reasons exist beyond those listed above, but they need further scrutiny to be justified as a source of an escape.

Keep in mind that escapes have a tendency to fall in between definitions, so it is likely that more than one of the causes listed above may apply.

The validator must keep in mind whether the causes for the bug are reasonable. Was the pre-silicon effort in this area sufficient to address the type of bugs that were being found?

A sound evaluation of how a bug escaped is the starting point to closing holes in the validation process. This evaluation will allow pre-silicon validation to catch not only this bug but also other related bugs.

6.3.2 Identify the Fixes and Workarounds for the Escape

At this point, an architect or designer will describe possible ways to fix the bug. This usually breaks down into one of the following categories:

- **Logic change:** An RTL change is made. This change can be a single wire or gate change (often the case for simple bugs on metal-only steppings), or an entire feature could be re-architected. Sometimes, if the logic is relatively simple, a few silicon parts can be modified via a FIB to implement the changes. This allows risky fixes to be evaluated before they are committed to the next stepping.
- **Microcode change:** When a bug lies in microcode or the timing of uops in the machine exposes a logic problem, the fix might require a change to microcode to remove the bug. The microcode can be modified in two ways: either the microcode in the ROM can be changed (currently, this requires a full layer stepping), or a microcode patch is applied. Patches do not require any change to the silicon. Patches are very frequently used for production-level workarounds or fixes to bugs.
- **Defeature modes:** Some units contain control register bits that can turn off certain behaviors, potentially barring the offending conditions from occurring. These bits can be set by tools on silicon debug platforms or enabled by a microcode patch. The *defeature* might not have been fully validated previously, if it was not considered a mainline feature or risk area. Because of this, defeatures might require additional validation effort. However, defeatures are valuable as workarounds to the bug to allow forward progress to be made, even if they are not used for production.
- **Acceptable Failure Mode:** Sometimes a sighting is resolved to a real logic bug, but real-world systems will never create the bug in any rational way, or the failure is in a feature area that is not heavily used by software. An erratum for the processor will be documented, but no action will be taken to correct the bug on this stepping. The bug could be fixed in future steppings (or not). The decision of whether or not a bug can impact real-world systems needs to be made by a consensus of architecture, software, and platform experts. The decision whether or not to fix a bug requires approval by senior project management. The decision not to fix a bug potentially exposes it to customers, and should never be taken lightly.

- **Combination of the above:** Some bugs are complex enough to require more than one of the above elements to work around them.

Regardless of the type of fix, the validator needs to work with the person specifying the fix to understand the implications of the change and to provide feedback on what is feasible to implement. Each stepping might have a different fix or workaround depending on the scope of the failure, the risk of the fix, or the timeline for producing the next stepping and/or qualifying a stepping for production. The validator needs to assure themselves that they can adequately test the fix in the time allotted.

The design team implements a fix and releases it in a new model. It is the validator's responsibility to determine the scope of these changes, and how to validate them properly. This is an onerous burden, since the viability of a stepping and perhaps an entire product might be predicated on correctly fixing the bug. The validator should understand the nature of the changes. The validator should examine any implications of the changes, not only in the area where the fix is located, but also in any multi-unit interactions.

6.3.3 Implement the Validation Infrastructure

The validator needs to implement any new checking, coverage, or testing needed to target the area surrounding the bug. This effort can be viewed as two overlapping phases: immediate validation of the bug and long-term closure of the escape.

Validating a post-silicon bug is similar to validation of a pre-silicon one:

- The validator must have tests that target the failure modes.
- The validator must insure that proper checking and coverage is in place to track whether the bug conditions occurred.
- The validator must determine whether the correct behavior resulted.

For a silicon escape, it is likely that one or more of these are missing or incomplete. The validator must create a level of testing, checking, and coverage appropriate to validate the bug for the next stepping.

Deciding what portion of the infrastructure to implement near-term is based on the risk posed by the escape, the amount of change incurred to fix the bug, and the likelihood of catching more bugs of the same type or in the same area. The higher the probability that similar bugs lie in wait, the more effort is required on developing ways to catch them.

Long-term closure of the validation holes might require further development of random testing, emulation, checking, and coverage constructs. This effort is not necessary for immediate validation of the bug, but contributes to finding and preventing future bugs in the same area.

However, when planning closure of a long-term validation hole, the validator should not constrain the possible infrastructure enhancements based on the amount available time. Keep in mind that while some of these improvements might not be implemented immediately, they are still valuable. The priority of implementing these improvements might also change based on new information gained from other escapes.

6.3.4 Pre-Silicon Stepping Validation

Once a new model exists with the appropriate logic fixes, defeatures, or patches, the pre-silicon team validates the model to a high level of quality. The scope of the validation effort varies based on the magnitude of the change. Multiple validators may be involved in validating the fix if it spans several functional areas or protocols. The team uses their new infrastructure to validate the bug and documents the details of how the bug was validated in the Escape Checklist.

The process of validating fixes to the silicon is described in the [Stepping Validation](#) chapter.

6.4 Is the Escape Really Contained? (Validation)

At some point, the validator confirms that the bug is fixed and validated in the released model. The enhanced testing, checking, and coverage created by the validation team will be in place to catch similar bugs.

At this point in time, with no critical open bugs, another stepping will be produced, or a production microcode patch will be deployed to fix the bug for the current stepping.

SV and CV have a key role in validating a fix, because of their ability to run a massive number of cycles. The sighting was originally found on silicon, so the same conditions can be applied to the new part or patch to see that the failure does not recur. After the fixes or workarounds are tested exhaustively in systems, the confidence in a given fix needs to be high enough to declare the bug *validated*.

If the original bug resurfaces, repeat the above steps until the reasons why the bug escaped *again* are fully understood. If this condition occurs, multiple validation team members may need to work together to find the escape causes. Repeat escapes should be a rare occurrence.

Bugs that are similar to the escape might still exist in the design. A test targeting a bug fix (short-term validation) might not extend far enough to cover related areas nor address the surrounding logic sufficiently to catch related bugs. Similar bugs found in silicon might indicate a lack of improvement in the validation of that area and deserves a higher level of scrutiny.

6.5 Closing the Validation Loop (Outputs)

Direct products of the escape validation process are:

- The primary goal of the process is to ensure a set of fixes or workarounds exist for all applicable steppings and projects
- A description for why the bug occurred has been created and filed in the bug database
- An analysis of how the bug slipped past validation is documented and available
- One or more tests that create the failing conditions have been created and are available

In addition, as required:

- New conditions for the failing area are in the unit/cluster's protocyte
- New checkers or enhanced checking around the bug conditions have been created

- New cluster emulation has been created to more aggressively test the bug boundaries
- New injectors have been created to stimulate the failing behavior more often
- New methods for validation that target this category of bug have been created and distributed to the pre-silicon team

If another bug or category of bugs is found through the debug of this escape, the same steps listed above are to be applied to *that* escape. Other validation teams might have different ways of finding this bug, some of which might be more suited to finding similar bugs.

Example: The Formal Validation team has the ability to prove that a logic change is equivalent to another. For instance, a newly added defeature for a risky fix, when enabled, does revert the logic to its original behavior. FV has also shown impressive proofs of parts of some complex data space units (IPD, FMXD) and higher-level protocols (RRRR marble proof).

6.6 Escape Contained (Exit Conditions)

For a single escape, the validation process ends when:

- The changes required to fix the bugs have been validated to a level at least equivalent, if not higher than, the rest of the design.
- The database bug has reached a terminal state: *Validated, Future Fix, Won't Be Fixed*.
- An Escape Checklist for this bug has been completed, linked, and checked in to the central escape document repository. The database bug is updated to link to the escape document.
- If the bug is exposed to customers in a production stepping, an erratum has been filed for this escape. Any other affected products will also need to have their errata documents updated.
- Any bug trends that imply other escapes need to be pursued and exhausted. This is the most ambiguous step, as it relies on the validator to research the bugs, discern any common elements to bugs, and pursue any leads to find related bugs. This effort is potentially open-ended, so the validator must decide what constitutes sufficient investigation.

The escape validation cycle will begin again with the discovery of the next logic sighting.

7 Summary

The validation process is not perfect: it requires feedback on successes and failures in order to improve. Some categories of bugs will always escape detection and make their way into silicon. However, by carefully examining the paths that these bugs took to get into the design, the net can be made finer and tighter, preventing similar problems. The ever-increasing complexity of modern designs makes this refinement a requisite part of the validation cycle. If applied correctly, future designs will continue to tape-out with a minimum of logic problems that could delay qualification and shipment of the parts to the customers, or cause costly recalls.

8 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 45

ECO Validation

By:[Leo AL-Aqrabawi](#)

1 Abstract

This document covers the methodology DDG Validation uses to validate ECOs for CPU projects. ECO validation is a key part of the project lifecycle. The ECO process controls and monitors all the changes introduced into a stable design. Validating an ECO is our first line-of-defense to ensure RTL changes do not cause more problems. The validator has different tasks to perform during the different stages of the process: approving the ECO, risk assessment, verifying that the RTL follows the specifications, and validating that the RTL runs as expected, and sign-off.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	9/19/2003	First release of document	Leo AL-Aqrabawi	Matt Kupperman / DPG-N uAV MWG
1.1	10/21/2003	Applied EITVOX format and incorporated feedback after online review	Leo AL-Aqrabawi	Matt Kupperman / DPG-N uAV MWG
1.2	11/17/2003	Used the methodology template	Leo AL-Aqrabawi	Mark Savoy
1.3	12/12/2003	Incorporated MSavoy's feedback	Leo AL-Aqrabawi	Mark Savoy
1.4	6/15/2004	Grammar and readability	Leo AL-Aqrabawi	Paul Schwabe

3 Contents

1 Abstract.....	837
2 Revision History.....	837
3 Contents.....	838
4 Purpose.....	840
5 Background Concepts	840
6 ECO Validation Process	840
6.1 Entry Conditions	840
6.2 Inputs and Dependencies	840
6.3 Tasks	841
6.3.1 Before an ECO is filed	841
6.3.2 Approving an ECO.....	841
6.3.2.1 Risk of late bugs and escapes.....	842
6.3.2.2 Magnitude and scale of stepping.....	843
6.3.2.3 Your experience	843
6.3.2.4 Intent of the ECO.....	844
6.3.2.5 ECO interaction	844
6.3.2.6 Other options.....	844
6.3.3 Preparing for ECO validation	845
6.3.3.1 ECO scope	845
6.3.3.2 Ramping up on new logic areas	845
6.3.3.3 Validation Plan	846
6.3.4 Validating the ECO	847
6.3.4.1 Inspecting changes.....	848
6.3.4.2 Updating prototypes.....	848
6.3.4.3 Resetting coverage.....	848
6.3.4.4 New checkers.....	849
6.3.4.5 New injectors.....	849
6.3.4.6 Running tests	850
6.4 Validation.....	850
6.5 Outputs	850
6.6 Exit Conditions.....	851

7 Summary.....	851
8 References.....	851

4 Purpose

As a CPU design matures and becomes stable, Engineering Change Orders (ECOs) govern all changes in the RTL. The ECO process involves tracking, documenting, and managing the changes made to the RTL. Although ECO control kicks in at the late stages of the A0 stepping, all major changes to the RTL require ECOs at any time in the project. The ECO process clarifies and synchronizes the design and validation efforts. The ECO process is most important during the validation of steppings.

Teams can use ECOs to track changes in validation tools. Changes to iASM, Archsim, Proto, or the CTE require ECO control. Tool ECOs are beyond the scope of this document.

5 Background Concepts

Design, validation, and management drive the ECO process. The process includes managing and documenting the work and the team interaction that is part of the change. The design team files ECOs to fix bugs, fix speedpaths, or add new microarchitecture features. All potentially affected parties indicate the benefits, effort, and risk estimates of the ECO. Based on this information and the return on investment (ROI), management makes the decision to approve or disapprove the ECO. The ECO should include enough technical detail, so anyone familiar with the area can understand the impact of the ECO. After the ECO is approved, the design team implements the ECO. Finally, the validation team tests, verifies, and validates the ECO. The validator files any bugs the ECO introduces or signs off the ECO as “validated.”

6 ECO Validation Process

6.1 Entry Conditions

The ECO starts when the designers have a problem in a stepping that requires a change in the design. The problem might be a speedpath problem, a bug fix, or new microarchitecture features. The ECO officially starts when the design team files the ECO. Validators should be involved early with the design team during planning for the ECO, because validators can often see bugs before they manifest. Being involved with ECOs are major changes to the microarchitecture is particularly important. For simpler ECOs, validators can wait until the ECO is filed or approved.

6.2 Inputs and Dependencies

The validator cannot validate an ECO until the design team finalizes their proposal and files the ECO. The ECO should include a clear description of the ECO. The description should clearly state the problem the ECO solves. The description should describe how the RTL looks before and after the team completes work on the ECO.

The main input validation receives is an RTL release or a graft model. The validation team can begin validation with a release of the RTL changes in either a fullchip or cluster model. Validation eventually needs both models to complete testing and validation.

Design teams are not always diligent in documenting their work. Many times, they change their plans and the RTL without documenting the changes. Be aware of all ECO changes, and push the designers to document these changes. The closer to tape out the more designers get pressure to meet deadlines. This is when designers tend to neglect documenting their work. During this stage, be extra tenacious and make sure the designer does their job well.

6.3 Tasks

The role of the validator includes more than finding the bugs an ECO causes. The validator is responsible for preventing design from coding bugs. Before design files the ECO, the validator interacts with designers to finds out their plans for the current stepping. The validator provides input when the design change control board (DCCB) is approving an ECO. The validator prepares for the validation effort after design files the ECO and before design turns in the RTL. After design turns in the RTL, the validator starts the task of finding bugs. Before approving the ECO effort, the validator makes sure the ECO is well documented.

6.3.1 Before an ECO is filed

The validator's role starts before design files an ECO. The validator finds out what changes the designers are planning. This activity enables the validator to be proactive in the approval process. Being involved early can prevent validation from being the bottleneck of the tapeout. To be effective, the validator makes sure the design team hears and considers their input. Note that the design team has no obligation to take the validator's input at this stage. This stage is over once the design decides to file or not to file the ECO. Staying in touch with designers helps you in approving the ECO, and helps you make informed decisions on ECOs in the DCCB.

Staying in the loop with designers is always a good idea, even after they file the ECO. It helps to determine when to start validation and if they plan additional changes. In some cases, designers might make additional changes after the validator signs off on the ECO. These additional changes can be extremely dangerous. Be especially aware of these changes. Should this happen, design should document the ECO with the new changes to RTL. The ECO should include the new validation ARs.

After design files the ECO, assign at least one validator to the ECO. That ECO validator is usually the owner of the units the ECO is affecting. If multiple units are affected, more than one validator needs to own the validation ARs. If the ECO is a bug fix, the validation owner should be the bug filer.

6.3.2 Approving an ECO

When an ECO is filed, the ECO does not necessarily go into the design. It is merely a start of a new phase: the approval phase. The approval phase starts by having all technical parties affected by the ECO indicate their effort and concerns, and sign off their approval. Then, the design change control board (DCCB) decides on the fate of the ECO. In the DCCB, experts and stakeholders

decide on approving the ECO or finding alternate options to stay on the path to tape-out based on the best ROI. Usually architects, designers, and managers use their own metrics to weigh their decision. The validators should have their say in the approval. They should contribute to judging the feasibility and risk if the change gets into the design. Involving validation is important at this stage, because validators know the microarchitecture best and are the best judges of the risk of coding in new bugs. Validation should provide information on the effort needed to validate the change and the availability of resources before the tapeout date.

Before the DCCB process completes, the validator goes through four important metrics before approving an ECO:

- The risk of introducing a fatal bug going undetected.
- The magnitude of the current stepping.
- The validator's level of experience in that area of logic.
- The risk of interference with other pending ECOs.

Note that the work done before the ECO goes to the DCCB cannot assure the ECO will not create new bugs. Until the ECO details are concrete, the ECO is filed, and RTL is changed, certain bugs cannot be discovered.

The DCCB should approve or reject an ECO and suggest other options to fix the problem. The validator's role at this stage is successful at this stage by having only feasible ECOs (from the validator's perspective) approved. When a consensus is reached, the DCCB role is over for this ECO.

6.3.2.1 Risk of late bugs and escapes

Validators — from their experience with microarchitecture validation — recognize some areas of the design are “buggy.” These areas are vulnerable to bugs because of a history of bugs or escapes, or the design is weak, not easily scalable, and relies on too many assumptions to function correctly. For a validator, finding a bug is good, but the earlier the bug is found the better. The validator's job is to not only find bugs, but also to help prevent them.

This does not suggest that any ECO proposed in a vulnerable area should be rejected, but consider the following metrics before committing to validating an ECO.

- What amount of validation time, with respect to the tape-out schedule, will it take a bug to appear in the ECO? How many cycles will it take to validate the ECO? We have a limited number of cycles we can run during pre-silicon validation, and this can have a high impact depending on how close tape-out is. The closer we are to tape-out, the less time we have to exercise the logic by random testing. Hitting such bugs in silicon might not be as hard. An example of such conditions is the rare modes of operation that we might not have enough pre- and post-silicon tests to validate.
- Is the change in an area that is not heavily exercised during our regular testing and validation? This is a factor in pushing bugs to post-silicon. Livelock detectors are areas that

are not invoked easily; we design the chip intending not to use that logic. Only hard corner conditions can exercise the bug, and you might need extremely rare conditions to find bug in such areas pre-silicon. The bright side is that livelock breakers are not as fatal an area as - say - the FP divider that could cause data corruption. This effort is another factor to consider.

- If the worst happens and a bug is found in silicon, how fatal would the bug be to tape-out? If the bug escapes silicon, will an errata or recall be required? This is a big question. If there is no defeature for that ECO, no metal fix, and no patch to work around it, only a new stepping might fix that bug. If the bug could cause data corruption, it would require a recall if shipped to customers. This decision is most important and a decision where the validator should be most cautious before approving an ECO.

6.3.2.2 Magnitude and scale of stepping

A major ECO proposed in a small stepping increases the risk of bugs. A risky ECO in a small stepping can create bugs that validation will not catch until tape-out or after. Small steppings have smaller validation teams, shorter deadlines, and closer tape-outs. Team members usually own larger areas of the chip with which they might not be familiar.

Spreading small changes across different steppings takes significantly more validation effort and time than adding several changes to the same stepping. Design can move a new feature or enhancement to the next stepping more easily if the steppings tapeout closer to each other. If design introduces a new ECO is in a *No-RTL-Change* stepping, the change requires no validation effort. However, an ECO for one minor RTL-change requires resetting and recollecting coverage, which can take a few weeks. If design introduces another ECO in the same cluster and stepping, the additional validation work is insignificant. Recollecting the coverage will cost the same whether we have one or ten ECOS.

6.3.2.3 Your experience

During stepping validation, the validation team works in smaller teams and owns greater areas of logic. It is common to own an area with which you are unfamiliar. Consider your experience in an area when approving a bold change in the RTL. This is more important during later stages of the stepping. Usually right after A0 tape-out, these considerations are not a big problem. Consider how many ECOS you own and your other responsibilities. When the ECO is a high risk such that you might not handle it well, call in previous owners of the units to help validate the ECO. Loaners can help validate the ECO and help assess the effort required to validate the ECO. Taking their advice can make sure you do not miss anything major you might have seen if you were more familiar with the area. Your ability to justify pulling these validators from their work and the importance of the stepping and the ECO are factors to consider when deciding to reject the ECO or calling in help. Estimate the validation effort for the ECO, provide the information to the DCCB, and document it in the ECO. This will give designers a heads up as to how much time they can spend before turning in the new RTL.

6.3.2.4 Intent of the ECO

There might be a situation where an ECO is proposed in a short stepping, for example an A1 stepping. The change can introduce a bug you might not find until close to or after tape-out. If the ECO is a fix for a silicon bug found in A0, you might have to accept the risk. If the ECO is an enhancement or a speedpath fix, you can push back and stop the ECO approval.

6.3.2.5 ECO interaction

Sometimes you have multiple ECOs in the progress at the same time. Some ECOs might be pending approval, while others are not filed yet. What is really scary is having two or more ECOs affecting each other in-flight. Especially, if one ECO can break an assumption used as a basis for approving and validating others.

A good example from the PSC project occurred when two bug fixes created a new bug. The first bug was in an interaction between software pre-fetch and a page walk. The bug caused an unbreakable livelock. The DAC RQueue livelock detection mechanism did not recognize the pre-fetch as a mechanism that could livelock. The fix was to add the pre-fetch request type to the livelock detection mechanism.

In the second bug, the DAC RQueue had two mechanisms to detect livelocks (via incrementing a counter). The first mechanism detects livelocks when RQueue does not grant a request to the UL1. The second mechanism detects livelocks when RQueue grants a UL1 request, but the BSU recycles it. The first mechanism was a standard behavior, but the second mechanism is controlled by a defeature bit. In this bug, there was an interaction between an RFO and a request already in the BSQ. Instead of fixing the problem in the logic, the solution enabled the second livelock detection mechanism such that recycled requests increment the livelock counter.

The first bug included pre-fetched in the livelock detection scheme, and the second bug extended the livelock detection scheme to include recycled requests. What we did not realize was that pre-fetch requests that get recycled are dropped. Therefore, it was possible to increment the livelock detector via a pre-fetch that was recycled, which makes the livelock counter overflow. The overflow of the livelock counter set a blocking bit for a queue that no longer has a request. This unexpected interaction resulted in a third livelock.

Consider such situations during risk assessment and in the DCCB. If a pending ECO interacts with another ECO in the DCCB, the ECO you approve might be much more complex than initially thought.

6.3.2.6 Other options

If the stakes are too high for approving an ECO and the civil wars between design and validation erupt, the DCCB arbitrates the conflict. The DCCB considers the alternatives to filing the ECO. Living without an enhancement or pushing it to later stepping is often a good idea. Designers might find circuit solutions to their speedpaths. Sometimes a change to a more stable area of the chip can achieve the same results as an ECO on a buggy area. As a last resort, slipping tapeout might be an

acceptable option. Make the alternatives clear during the DCCB to avoid validation being a project bottleneck. Remember a healthy, slow design is better than a fast broken design. Providing healthy products to our customers keeps us competitive and saves our reputation and trademark.

If the DCCB approves the ECO against the recommendation of validation, your last option might be to disagree and commit. There are times that we compromise as part of our jobs as validators.

6.3.3 Preparing for ECO validation

After the ECO is approved, start preparing to validate it. The work begins before design turns in the RTL. The time you spend preparing depends on the scale and importance of the ECO, your familiarity with the logic area, and your current responsibilities.

First, start learning the logic area the ECO is targeting. Lay out a clear plan of what to look for in validating the ECO. Efficient use of your “idle time” before the ECO is implemented can improve your productivity and the time to validate the ECO. This work could keep the validation from becoming the bottleneck of the stepping.

6.3.3.1 ECO scope

Some ECOs only have local impact on the chip; the impact is not visible outside cluster boundaries. These ECOs could be bug fixes, speedpath fixes, or performance enhancements. The interface outside the cluster might remain unchanged. In these cases, validate the ECO at the cluster level only. If an ECO does not affect the RTL, verify the ECO does not change the RTL. You can then waive the validation AR.

Some ECOs, such as ECOs adding new features, create changes visible across the chip. Such ECOs must include validation at the uAV level. Validators from neighboring clusters must be aware of the change, and there must be a validation AR from every affected area.

If the ECO changes assumptions about the cluster boundaries or how the whole cluster looks from the outside, update other CTEs accordingly. If a change in assumptions affects the protocytes of the cluster, other clusters need to restart their coverage. Ask experts from other clusters about how dangerous the impact is when the interface changes between the clusters. Ask about any concerns they might have. They might have assumptions in their clusters that can break if your cluster changes. Add such conditions as forbiddens in your protocyte to make sure the conditions never occur. In addition, push the designers to add checking directly into the RTL to check these assumptions even when your protocytes are not installed.

6.3.3.2 Ramping up on new logic areas

Validating an ECO often requires learning new areas of the logic. This is especially true for new validators and stepping validation. Stepping validators usually own more areas of the logic after the A0 stepping. They tend to have less thorough knowledge in the areas they own. Stepping

validators tend to own logic in a cluster they are familiar with, giving them a head start over new validators on the high-level micro-architecture. Ask an expert for any information you need to ramp up on unfamiliar areas.

Learning the microarchitecture and the design implementations of an area are equally important. Reading the MAS gets you started on the microarchitecture (assuming it is up to date). Scanning through the RTL helps learn implementation details of a certain fub or superfub. Drawing pipelines and state machines is possible by reverse engineering the RTL and helps visualize the flow. Since you usually have a stable RTL model before the ECO is filed, run sample tests that exercise the area and to see how different parts of the design interact. Bringing up watch windows and following transactions⁴⁷ across the machine can be helpful.

Like the chicken and egg problem, a test plan is the child of understanding the microarchitecture as well as a good source for understanding the microarchitecture. Looking at the existing test plans for the unit you are validating can help you understand important and interesting corner conditions. Locate an expert for every unit or fub in the chip who can answer questions or provide chalk-talk protocols of the area of change. A good validator can learn and incorporate information about unfamiliar areas, and find all places the area affects to validate it.

6.3.3.3 Validation Plan

The unit you are validating has an existing test plan from the early validation of the A0 stepping. Because you own the logic in the unit, you are responsible for updating the test plan. The new plan should cover all new corner conditions introduced by the ECO and the different forbiddens that need to be changed in the protocyte after the ECO is filed. Do not document all forbiddens in the test plan. Only include forbiddens the test plan explicitly relies on as checkers. Do not document conditions that are forbidden, because the design is unable to hit these conditions and are added to shrink the coverage space.

It is important to update the existing test plan over writing an ECO specific test plan. Updating the existing plan keeps that plan up-to-date and complete, and provides a better feeling of the coverage. For example, if you have two plans and get 90% coverage on one and 50% on the other, it is difficult to understand how much the overall coverage is. If you do that, include the weight of each test plan with any coverage reports to clarify the coverage. If every validator creates their own test plan for every major ECO in the area, the test plan for any following steppings will be scattered all over, making the plan hard to find and maintain.

Sometimes an ECO introduces a feature specific for one stepping that does not apply to all other steppings. In other cases, new features might still be under evaluation or not solidly defined. Be careful when adding changes for new features to the test plan. Add the features in a clearly marked addendum (font, color, etc.). Specify which features apply to which steppings. After features tape-out and become permanent for all following projects, add them permanently to the unit test plan.

⁴⁷ A transaction could be a uop in RRRR context, a load or store in MEU/BUS context, a macro instruction or a branch in MITE context, or a trace in TDE context.

If a feature is removed before it is implemented, remove it from the test plan. However, if design removes the feature after implementing it, add new conditions to the test plan to verify that adding and removing the feature did not create new boundary conditions. There is a possibility that adding and removing the feature introduces changes to the design. Clearly state this in the testplan. Carefully documenting new features is especially helpful when another team inherits the test plan for subsequent steppings or projects.

The test plan is not the only planning you do when validating an ECO. Determine what you need to do and give an estimate on the time you need to validate the ECO. The next section discusses the suggested tasks to validate an ECO. Depending on the scale of the ECO, choose which tasks are needed or worth the effort. Document the tasks in the ECO with an estimate of the time required. Such planning helps determine when you will be done validating the ECO.

6.3.4 Validating the ECO

After design turns in the first RTL for the ECO, make your judgment if the time is good to start validating the ECO. When to start validation is an important judgment call. If you wait too long to validate the ECO, the designers might forget many of the details and intentions when they filed the ECO. This is especially important if design did not document the ECO well. If you start validation late, there might be enough time for finding bugs, filing bug reports, fixing bugs, and validating the fixes. Starting late might cause tapeout to slip. Remember, the time to tape out was one factor for deciding on the ECO approval in the DCCB. Starting validation on time is as important as filing the ECO on time.

On the other hand, starting validation too early while design is turning in more RTL changes might waste your time and effort. Staying in the loop with the designers minimizes this risk. However, you might have deadlines to meet for this or other ECOs. If the ECO requires a significant amount of work and has a short deadline, you can justify starting your validation with the first RTL release. There is plenty of work to do that does not depend on later RTL changes. If you write your own checkers and injectors for the ECO, you can update them in line with the new RTL changes. The designer should not make any further changes to the RTL without revisiting the RTL-change ARs and validation-ARs (given the validation ARs are already signed off). Some tasks you can start with the first RTL turn in, while some others might have to wait. Sometimes it is worth the effort to start a task, and reset the task if you have to later. This effort can work if the risk of change is less, and the time line is tighter.

Once you are ready to start validating the ECO, get a clear message from the designers on their final plans. Read the ECO proposal again, and confirm that no assumptions or plans have changed since the DCCB. However, the only way to see the actual implementation of the ECO is to have a cluster RTL release. You cannot validate, verify, or test against an ECO without an RTL model. If you did your homework and design delays the RTL release, push on the designers to turn in their changes. This action will enable you to write checkers, injectors, and prototypes.

Designers might try to sneak in more changes than initially proposed and approved. Therefore, inspect the RTL. Be familiar with how the RTL looked before the ECO, through either previous experience or your preliminary preparation. You might need to add new ECO-specific checkers

and injectors, and update the protocyte by adding new skews and forbiddens. Some ECOs might require resetting coverage for your clusters and even adjacent clusters.

6.3.4.1 Inspecting changes

As validator, you are responsible to make sure the ECO does not introduce bugs into the design. You can accomplish this goal with two tasks:

- Validate that the proposed changes fit naturally in the existing microarchitecture without breaking any of the existing protocols or assumptions.
- Verify that the designer has implemented exactly what was proposed by the ECO.

If the proposed changes make sense and seem reasonable, but the implementation is different, there could be big problems. A difference between the proposed design and the implementation is common for even the simplest ECOs. Even if such differences do not cause failures, flag the differences and make sure the designer updates and reevaluates the ECO specification. If you think the discrepancy can cause failures, file a bug. Many bugs are found by inspection only; that is why one of the “How found” fields in the TIBET⁴⁸ bugs is “inspection.”

6.3.4.2 Updating protocytes

Inspecting the design changes is your starting point to updating the protocytes. Because protocytes are the realization of the test plan, update the protocytes to be in line with the test plan. Note that some of the existing skews and temporal flows might fail due to signal name changes. Other skews and flows might need to be updated due to pipe-stage changes. You can add new skews to cover for new interesting features and corner conditions. Add new forbiddens for areas that concern you or might break some of the new assumptions. If you are suspicious about an ECO that might cause bugs if existing assumptions in the cluster are changed, add these assumptions as forbiddens to your proto code and document the additions in the test plan. If a later ECO relaxes these assumptions, it will create a bug (see ECO interaction) and your proto should catch it.

6.3.4.3 Resetting coverage

If the ECO has any changes in a logic area of a fub, reset the coverage of that fub. These changes could be as subtle as pipe stage changes or changes in clock gating (gated to free running or different gating logic).

It might be hard to decide which changes affect fubs from the same unit. Since the interface is not very crisp within a unit, it is safer to reset coverage for the whole unit. You might need to reset coverage for other units if they contain driving or receiving logic from that changed unit.

⁴⁸ TIBET: Test, Issue, Bug, and ECO Tracking tool.

You might have to make judgment calls when evaluating the return of resetting any level of coverage. As a rule of thumb, when in doubt, reset coverage. If you are not sure a change affects the coverage for an area, it is safer to reset the coverage.

6.3.4.4 New checkers

Any new checker created specifically for an ECO is like a microscope that can show any violation to assumptions before the assumptions can cause visible failures. These checkers shorten the time required to find bugs. In an existing CTE, uAV and AV checkers are looking for violations in protocols at their respective level of abstraction. To catch a bug, the bug has to propagate and be observable at the level where these checkers run. Many bugs take longer to appear because they are masked. You can easily hit the same bug repeatedly, but never get a failure. Remember the fourth commandment of validation: “Even if it’s tested, if it isn’t checked it still doesn’t work!”

For example, suppose an FP divider can get the wrong answer for a specific input. If you have a test that feeds that rare input to the adder at a replaying uop, the results will not be committed to the architectural register unless that uop commits. If the pattern is too hard to hit (think of the FP space), you will lose your chance of catching the bug if the uop does replay and get different input pattern when it commits. Adding a checker around the adder when it was designed increases your chances of hitting the bug.

Some checkers might be so important that they get integrated into the CTEs. A checker is any mechanism running with the test to cause a failure if any small part of the chip does something wrong. You can write checkers as part of a mace self-checking test, an API, or temporal flows and forbiddens in the protocyte.

6.3.4.5 New injectors

You can add injectors to initialize or change intermediate states of the chip to do something weird. You can expose bugs that usually appear in silicon by using injectors to stress the logic in a way only possible with billions of random cycles. You might suspect a corner case will occur if an array is loaded with certain data, and a certain uop reads the data. If the array is difficult to initialize using regular cluster or fullchip test, write an injector to force the values of that array to what you want. If you want to see what happens when several events occur on the same uop and are not able to do it in cluster or fullchip, force it by an injector.

If an ECO-specific checker is the microscope that enables you to see bugs before they surface, an injector is the micro-hand that lets you manipulate things inside your little universe. For more details on evil injector, see the Getting Evil chapter. Using focused checkers and injectors for a small area of logic is analogous to creating a unit or sub level CTE (UTE/ FTE). It is better not to use injectors when the conditions are possible to hit with natural tests. Injectors do not exercise the setup logic to create the condition. If the setup conditions are a gate to observing a failure, the injectors might not detect the failure.

6.3.4.6 Running tests

Run enough tests on most ECOs to make sure the design does what it is supposed to. There is no point in creating checkers and injectors if you don not run tests against them. There is no point in resetting the coverage if you do not regain coverage. Always run enough tests to, at least, regain your coverage (if reset), and hit the newly created conditions. Regaining coverage might not be a condition for signing off your AR, but it can be a condition for tape-out.

6.4 Validation

Not all of the validation tasks described are necessary to validate every ECO. The tasks depend on the scope the ECO. Some ECOs require less work, others more work. For example, if the ECO is for a speedpath fix only, you can validate the ECO by inspecting the RTL or by running random regression tests with the appropriate checkers. If an ECO is a bug fix, it is sufficient to validate the bug and sign off the ECO. At the other extreme, if you find you can do more to validate an ECO and you have the time, you are more than welcome to do more validation. Innovation is always welcome in validation!

During the initial preparation, you marked the tasks to validate the ECO. Use the task list as a checklist. If you have done the validation work needed and regained all reset coverage to acceptable goals, you might be in good shape to stop the validation work. Even when you stop work on the ECO, your role is not over. You are not done until you sign off.

6.5 Outputs

As validator, your major deliverable is an ECO with bug free RTL. However, you own more. Make sure the ECO work is complete before you sign off your AR. Track all changes that went into the design via the ECO and test plans. Hence, verify that your test plan is up-to-date, and the ECO is well documented with all changes.

At the end of the ECO work, verify that the test plan is up-to-date and contains all new conditions added by the ECO. Clearly mark new conditions if they are temporary, or add them to the original plan if permanent. These new conditions should clearly point to the ECO that introduced them.

Make sure the ECO contains all the information needed to understand the plan in the future. If the designer has not documented their work, make sure they complete the documentation or do it yourself. The ECO description should include the following information:

- The problem the ECO is trying to solve.
- The rationale for filing the ECO.
- Why other options are not suitable for fixing that problem.
- How the RTL looked before and after the ECO, and what changed.
- The cluster model where the changes went in, and file revisions that went in.
- Any additional turn-ins along with the reasons for making each turn-in.

Specify your planned actions and what you did to validate the ECO. The information might come in handy for continuous improvement should the ECO introduce a bug later that you missed. The information is also useful if the ECO is revisited later in debugging a failure. When revisited, the level of validation done for an ECO helps the person debugging the problem identify if the ECO could be a suspect for the bug. Any related bugs or ECOs should have links in the current ECO. After you complete all the necessary work to validate the ECO, sign off your validation AR.

6.6 Exit Conditions

After you complete your ARs and are confident the ECO is thoroughly tested, the risk of bug is almost zero, and the ECO and test plan are well documented, you are ready to sign off your validation AR. Unfortunately, signing off your AR might not be the end for your work on the ECO. Stay on top of the ECO to make sure design introduces no new changes. If you run into failures later in the stepping or in silicon that might be related to that ECO, revisit the ECO and keep your fingers crossed!

7 Summary

ECO validation is an extremely important process that requires a great deal of interaction between design and validation. The validator should be proactive at all phases of the process and be in close communication with the designers. The validation team should be a stakeholder for approving every ECO. The validator is responsible for making sure everything is in order and well documented before signing off the ECO. Most important, the validator should take all possible actions to make sure no bugs escape the validation process.

8 Future Work

This section intentionally left blank.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 46

Stepping Validation

By: [Erik Berg](#)

1 Abstract

After A0 tapeout, the goals of validation, to deliver a fully functional and bug-free part, do not change, but the methodology employed and the scope do change. Issues include lower staffing, skill of the validator, validator loaning, and how to rate the severity of changes to the RTL.

2 Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.1	02/17/2004	First release of completed and reviewed document	Erik Berg	Matt Plavcan
1.2	03/16/2004	Edited version.	Erik Berg	Matt Plavcan
1.3	05/14/2004	Second L3 review completed.	Erik Berg	Matt Plavcan
1.4	05/18/04	Final edits of L3 review completed	Erik Berg	Matt Plavcan

3 Contents

1 Abstract.....	853
2 Revision History.....	853
3 Contents.....	854
4 Purpose.....	855
5 Background Concepts	855
5.1 What type of RTL changes go into steppings?	855
5.2 What is the validation mindset change?	855
5.3 What skill sets are required?.....	855
6 Stepping Validation.....	856
6.1 Staffing after tapeout	856
6.1.1 Loaning	856
6.2 Stepping Validation.....	856
6.2.1 Entry Conditions	856
6.2.2 Inputs and Dependencies	857
6.2.3 Tasks	857
6.2.3.1 No change since last review.....	857
6.2.3.2 Some changes since last exit review.....	857
6.2.3.3 Changes in surrounding units/clusters that could affect this unit	858
6.2.3.4 Historical trends in changes	858
6.2.4 Validation	858
6.2.5 Outputs	859
6.2.6 Exit Conditions	859
7 Common pitfalls associated with stepping validation.....	860
8 Summary.....	860
9 References.....	860

4 Purpose

Stepping validation is a separate animal from the validation effort before A0 tapeout. Stepping validation begins when new issues that were not necessarily issues before the A0 tapeout arise. Examples include staffing issues and whether to implement an engineering change order (ECO). In stepping validation, the pace of validation increases and fewer resources are available. Validators who can multitask and have a high-level of understanding of the interactions between units are valued highly. Many questions remain about the best way to perform stepping validation, including how to loan validators, how to reduce the workload on the remaining validators, and how to rate the severity of changes to the RTL. This document discusses these questions and how the validation team can resolve issues in these areas going forward.

5 Background Concepts

5.1 What type of RTL changes go into steppings?

RTL changes can result from changes due to performance, timing, or bug fixes. Before implementing changes, the design and validation teams consult and coordinate on the implementation.

There are two kinds of steppings: dash steppings and full layer steppings. Dash steppings involve metal only layer changes, for example, a change from stepping G0 to G1. Full layer steppings involve changes to all silicon layers, for example, a change from G1 to H0.

Full layer steppings are typically changes that designers cannot implement by a metal-only layer change. For validation, this means adding new features, performance improvements, bug fixes, and timing fixes needed for frequency pushes. Metal-only layer changes are for bug fixes or speedpaths found in silicon that can be fixed without a full layer stepping.

5.2 What is the validation mindset change?

Validators are more risk-averse in the later steppings, because the scope of exiting the stepping is not as exhaustive as the A0 stepping. With steppings, we do not need exhaustive testing because we are starting with a quality part. Still, the potential increases for higher-level bugs to slip through to a later stepping, because the validation of changes might not be far reaching enough. Our goals are to prevent backsliding in overall quality and to test rigorously any specific changes. These goals are similar to the goals of a revalidation effort.

We cannot depend on the designers to know the risk of a change. In our validation role, we are gatekeepers for the changes instead of simply checkers. The gatekeeper mindset and potential methods for doing this are discussed more in later sections. One method, described in the 6.1.1 section, is a scoring system that rates how much the changes will affect surrounding units. The system is not yet developed but would be an objective method for rating the risk of a change.

5.3 What skill sets are required?

There are several critical skills required for stepping validation including:

- To know the details of the process.
- To know the details of the microarchitecture.

- To know where to get an answer.

A sharp eye for detail and an ability to multitask are common validation skills required before the A0 stepping. It is crucial that nothing is dropped in the validation process. Any changes not carefully shepherded through the process could be a fertile ground for bugs not found.

Understanding the microarchitecture at a high level helps the validator identify the effects across the chip that any changes might cause.

There are knowledge gaps even for experienced validators. Knowing where to get an answer is an important skill to have. The skill includes contacting the right people who have more expertise in the area being validated.

6 Stepping Validation

6.1 Staffing after tapeout

Full staffing is usually maintained through the last major tapeout stepping. After the last major tapeout, validation teams transition to pushbutton validation with light staffing. Bug and ECO trends can be used to modify this staffing plan and ensure that potential problem areas have adequate people resources.

6.1.1 Loaning

The practice of loaning validators to other parts of a project has advantages and disadvantages. The validation team has experienced and talented validators who can provide help in different validation areas to get a processor to PRQ. The practice is generally recognized and promoted with the stipulation that owners of the design validation (DV) sections need to be able to move back quickly if an emergency develops. Limiting the amount of loaning from any cluster to 50% is a general guideline though not a hard rule.

One method is to loan validators from areas that are rated to need the least help, perhaps using a scoring method (this is an idea that has been floated but the implementation details are very unsettled) to determine the amount of resources that need to stay with the DV area. Once a validator has been loaned, another benefit of the scoring system would be to help draw validators back, depending on the severity of the score for the change.

Some activities, such as system validation (SV), are regarded as better fits for validators, because validators tend to improve their knowledge of the chip through the work. Other activities, such as fault grading, are painful for validators but are still valuable experiences (at least in the author's experience).

6.2 Stepping Validation

6.2.1 Entry Conditions

The validator performs several activities before commencing stepping validation. During the A0 stepping, there are many people per cluster. During later steppings, many validators move to different areas due to loaning. To make the tasks more manageable with fewer people during stepping validation, we spend more time and effort automating common tasks and improving the infrastructure after the A0 stepping. Therefore, fewer people can perform the work more

efficiently. We often use brute force during steppings by throwing more validators at problem areas as they arise. This can be avoided through careful preparation. Tasks associated with improved infrastructure include:

- Updating test lists so they have expressed purposes and can be counted on to target specific areas of micro-architecture.
- Documenting all known bugs.
- Explaining all false failures or identifying/developing APIs that can detect the failure signature.

These tasks prevent unnecessary debug or cleanup work that could cloud the view of the validator and prevent full attention being given to changes.

6.2.2 Inputs and Dependencies

The only input in stepping validation is a released model for the new stepping.

6.2.3 Tasks

The task of the validator is always to root out bugs that exist in the current stepping and get the model to tapeout quality. Depending on the complexity of the changes since the last stepping, the scope of the validation and the amount of effort necessary can change. To complete sections 5.2 and 5.3, the validator performs the following smaller tasks:

- Changes the test plan to reflect any new conditions to be tested.
- Generates new test templates (either original or tweaked versions of existing templates).
- Updates the infrastructure to include new or expanded coverage spaces and new checkers.

6.2.3.1 No change since last review

If nothing has changed within a cluster, including the cluster test environment, there is no reason to revalidate at the cluster level. If a validator has an idea of how to be evil at design validation (see [Evil Validation](#)), they should continue to test the logic. While there is little reason to run tests to revalidate the stepping, there is value in continuing to run tests to maintain the tool chain and test templates. The health of the tools and tests is critical. Later changes might require rapid checking and not allow for repair of tools that stopped working for some magical, undiscovered reason.

Validating at fullchip is different from cluster validation, because something always changes on the chip requiring some tests to be run. If a change has been made, the validator determines the level of validation based on the level of interaction with the changed logic, measured by a scoring methodology.

6.2.3.2 Some changes since last exit review

In a stepping, a change does not happen if no ECO is filed. No ECO will be approved by the DCCB without the expressed consent of validation. There is no difference between these new items arising from the changes and the original items in the test plan. Any new items must be driven to a silver quality exit criteria. The issues arise over how to validate these new items and the surrounding

logic. If the logic does not have implications outside the functional unit block (FUB), we can limit the validation to the FUB. If it is a feature, we need to validate everything it touches.

6.2.3.3 Changes in surrounding units/clusters that could affect this unit

There is a different mindset for validating bug fixes and ECOs. Before the A0 stepping, we are unconcerned with the interactions with the surrounding logic, because everything will be exited later. After the A0 stepping, the estimate of the effort changes because of the potential effects on the surrounding logic that will not be exited again. Having a high-level understanding of the protocols and interactions a unit or cluster has with the surrounding logic is the reason why experienced validators are highly valued in the stepping validation effort.

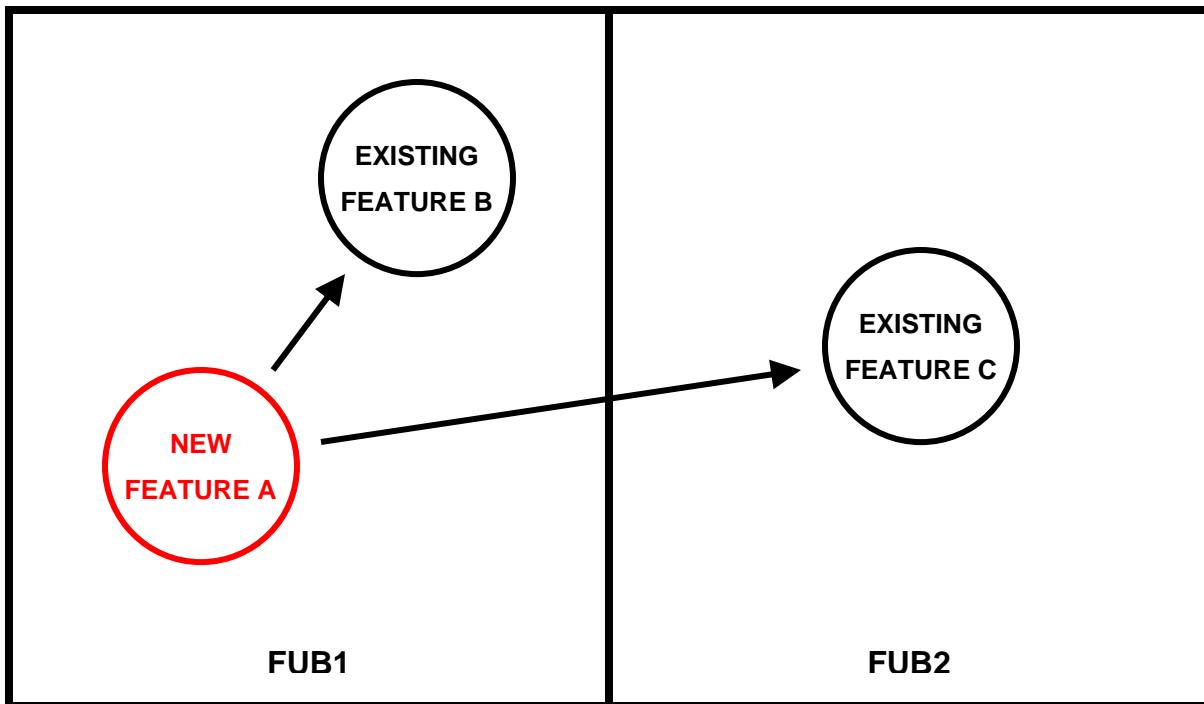
6.2.3.4 Historical trends in changes

Historical trends of where bugs were filed before tapeout can point to potential problem areas (due to timing or performance issues or the complexity of the MEU) that will result in more changes after tapeout. Resources can be shifted to cover these areas better and any potential changes could be rated riskier or even blocked if the validation resources cannot cover the changes.

6.2.4 Validation

Validation of a *new feature* is complete when the following questions are answered:

- Does Feature A work? The validator must drive any new RTL to the A0 stepping quality exit criteria.
- Has the interaction between Feature A and Feature B within FUB1 been validated? The validator must test the interactions between a new feature and other features within the FUB.
- Has the interaction between Feature A in FUB1 and Feature C in FUB2 been validated? The validator validates any changes in logic that interact with logic outside the FUB. The following picture shows this impact.



Validation of a stepping is complete when:

- All new features have been validated.
- All updates to the test plan, templates, and validation infrastructure have been completed.

6.2.5 Outputs

Several items are output from this process:

- A tapeout quality stepping.
- Updated test plan.
- New test templates.
- Updated validation infrastructure (coverage spaces and checkers).
- Exit Conditions
- Staffing management questions have been answered.

6.2.6 Exit Conditions

Simple: Do not stop until tapeout quality has been reached and ***all validation bullets have been completed***. One may be tempted to cease validation efforts once a stepping has been taped out, however, the stepping validation effort is not considered complete until the above condition has been met.

7 Common pitfalls associated with stepping validation

The most common pitfall of stepping validation is understaffing. An erroneous assumption has been repeatedly made that anyone can be plugged in to help with validation. The problem is that the critical part of stepping validation lies in gauging the effect of the changes on the surrounding logic. New validators do not easily or quickly understand these interactions. Running tests to target the changes is one thing. Choosing or writing the correct tests to target the interactions between the changes and the existing logic is another. The validators that own this process need to be experts.

Another issue related to validation of steppings is the decay of tools and tests from diminished use. A nasty cycle can develop after the last major stepping as validators are loaned out and fewer tests are run. Cycles need to be run to maintain tools and test suites so that when the inevitable emergency arrives, the validation can proceed quickly with minimal impact from tool and test health.

To avoid common pitfalls, the validator needs to do the following:

- Set up the validation environment as soon as the stepping is ready to go, instead of waiting until tests are ready to run.
- Automate tasks that will be performed multiple times to make up for the reduced headcount.

8 Summary

Stepping validation involves different responsibilities for the validator than before the tapeout of the A0 stepping. Validators need to be involved in decisions about whether design changes should be implemented and often have to consider interactions with other clusters that are not being changed. Experience plays a large role in determining the level of validation for logic surrounding RTL changes, though a scoring system has been proposed (there are no current plans to implement it). In stepping validation, fewer validators are typically used and more automation is used to accomplish the tasks.

9 References

This section intentionally left blank.

The Art of Pre-Si Val: Chapter 47

Glossary

By:[Matthew Kupperman](#)

1 Abstract

Welcome to our Glossary!

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0				

3 Contents

1 Abstract.....	861
2 Chapter Revision History	861
3 Contents.....	862
4 Glossary.....	864
4.1 Array Freeze and Dump	864
4.2 COBALT	864
4.3 Coverage	864
4.4 CV.....	864
4.5 DFD	864
4.6 DFT.....	864
4.7 DPM.....	864
4.8 DUT	864
4.9 EMON.....	865
4.10 Escape.....	865
4.11 FSB.....	865
4.12 FTS.....	865
4.13 ITP	865
4.14 Microbreakpoints	865
4.15 ODCS	866
4.16 Onion Peeling	866
4.17 Pipe-X.....	866
4.18 PRM.....	866
4.19 PRQ.....	866
4.20 PSMI.....	866
4.21 Random Test (Pure vs Directed)	866
4.22 SBFT	867
4.23 Scan/Scanout	867
4.24 Schmoo	867
4.25 Shroo	867
4.26 Sighting.....	867
4.27 Silicon-friendly	867

4.28 Stepping	867
4.29 SV	868
4.30 TAP	868
4.31 Tapeout	868
4.32 TLA	868
4.33 Watch Windows	868

4 Glossary

See also the Intel Acronyms Dictionary, here: https://intelpedia.intel.com/Intel_Acronyms

4.1 AR

Action Required. A well-defined task that needs to be completed by a deadline.

4.2 Array Freeze and Dump

The name given to the process of interrupting the processor during its execution post-silicon and capturing all of the data in the arrays on the chip using the DFT mechanism. See chapter 34 for more detail.

4.3 BKM

Best Known Method. BKM is intended to document proven methods so others may use them. They provide a means of transmitting knowledge from one group or individual to another.

4.4 COBALT

Hardware accelerator used to accelerate the PSMI replay process.

4.5 Coverage

A tool to provide feedback on whether tests are hitting specific conditions

4.6 CV

Compatibility Validation. This is a post-silicon team which runs a wide range of existing software on the new processor in order to ensure that the processor behaves in the same manner as previous processors and/or to the PRM specifications.

4.7 DFD

Design For Debug

4.8 DFT

Design For Testability. Also generically used to refer to the mechanisms used to test the arrays on the chip post-silicon for faults, which are also used to read the arrays in post-silicon debug.

4.9 DPM

Defects Per Million. A measure of how many defective parts are escaping to customers, and the key benchmark of the fault grading effort.

4.10 DUT

Device Under Test

4.11 EMON

Event MONitors. EMON refers to the counters on the chip that can be configured to count microarchitectural events, either for performance analysis or to generate microbreakpoints for debug.

4.12 Escape

a problem that you did not catch - said to have escaped to some higher (more expensive) level; examples: a bug that escapes cluster testing to be caught at fullchip, or that escapes pre-silicon validation to be caught by post-silicon testing, or worst of all, a bug which escapes Intel validation to be found in the field by customers

4.13 EXE

4.14 EXEC

4.15 FSB

Front Side Bus. The primary bus used to connect Pentium™ 4 and earlier processors to the rest of the system, typically to the north bridge or Memory controller hub of the chipset, and optionally to other CPUs.

4.16 FTS

Focused Test Suite. Test suite used by the SV team which contains specially designed tests directed a particular area of functionality.

4.17 FUB

Functional Unit Block. The smallest hierarchical unit of the design. Typically one RTL file.

4.18 ITP

In-Target Probe. This is a software tool developed by an Intel group (XTG) which is used as the primary debug interface in systems with the JTAG port of the processor. This tool is used extensively during post-silicon debug.

4.19 Microbreakpoints

The name given to the mechanism used to take one or multiple pre-defined "actions" based on a microarchitectural event. The mechanism consists of two parts; the EMON architecture which is used to count microarchitectural events, and the microbreakpoint controllers which specify the action to be taken. One common example would be to program the controller to cause a scan snapshot and array freeze when a particular microcode routine was executed to help debug a post-silicon problem.

4.20 MITE

Macro Instruction Translation Engine. One of the clusters in the Pentium™ 4. The MITE was responsible for fetching IA32 instructions, decoding them into uops, and delivering them to the TDE cluster.

4.21 MEU

Memory Execution Cluster. One of the clusters in the Pentium™ 4.

4.22 ODCS

On Die Clock Shrink. A mechanism designed onto the chip to cause a targeted cycle in the test to "shrink" - the clock is compressed to less than the normal operating period, effectively increasing the perceived frequency for that single clock.

4.23 Onion Peeling

The situation where multiple bugs exist in a feature, and each bug makes it impossible to find the other bugs until that one bug is fixed. Thus, bug finding becomes serialized where bugs are found in "layers", uncovered one at a time only after the previous layer is removed. Onion peeling on silicon is the worst case scenario, since the turnaround time to fix each layer is very high.

4.24 Pipe-X

A post-silicon test suite involving relatively short random instruction streams used by the SV team.

4.25 PRM

Programmer Reference Manuals. The manuals Intel publishes that describe the functional behavior of IA32 processors and how to program them.

4.26 PRQ

Product Release Qualification. The final approval to release parts to a customer for sale. Note that we often ship products to customers in anticipation of PRQ but are not allowed to sell them until we grant PRQ.

4.27 PSMI

Stands for "Periodic System Management Interrupt". This is the primary post-silicon debug tool used by the Wmt, Nwd and Psc teams. It consists of periodically interrupting the processor during the flow of execution; saving the current state or forcing it to a known state; capturing the bus trace during this handler after a special synchronization point; and replaying the result on the interactive CSIM model. For more detail, see chapter 34.

4.28 Random Test (Pure vs Directed)

A test whose content is determined through making a series of random choices. A directed random test is one where the test creator provides guidance (weights) in order to make certain desired events more likely to happen than others. A pure random test is one where the test creator has little to no control over the test content.

4.29 RRRR

4.30 SBFT

Structural Based Functional Testing. This refers to a set of tests which are designed for the limited pin connections in the structural tester. The tests are preloaded into the UL1 using the DAT controller; the tests execute and write back results to the UL1. No FSB transactions are allowed within an SBFT test.

4.31 Scan/Scanout

A mechanism used in post-silicon to capture the current state of a set of state nodes defined in pre-silicon for debug.

4.32 Schmoo

Plots created, usually on testers, to show the behavior of the part across different operating conditions. The classic shmoo is a plot with two axes, frequency and voltage, which shows a curve describing the fastest speed where the processor operates as the RTL model predicted at each voltage.

4.33 Shroo

The tool used in speedpath debug which uses ODCS to shrink individual clocks and find the clock or clocks which make a given bus/scan failing signature worse when shrunk. This is used to narrow down a speedpath to a specific cycle of failure in the test.

4.34 Sighting

A post-silicon record of a failure that has a better-than-fair probability of resulting from a silicon bug. The record includes configuration of the platform, the number of threads executing during the failure, and all observations made about the type of failure.

4.35 Silicon-friendly

Refers to pre-silicon tests which do not have any behavior that cannot be replicated by the post-silicon testers; in other words, all input is done by way of the pins, and no part of the test forces any of the RTL state through "pounders".

4.36 Stepping

A revision of the silicon. These are divided into two types; a "full layer" stepping, in which all of the masks used to create the silicon can change, and a "metal layer" stepping, in which the only masks which are regenerated are the metal layers, allowing for fewer possible changes. Each stepping gets a different name; the usual system is to change the letter of a stepping on each full layer stepping and the number of a stepping on each metal layer stepping. For example, the first metal layer stepping from A0 is A1; and the first full layer stepping from A0 is typically labeled B0.

4.37 SV

System Validation. This is a post-silicon team which creates and runs post-silicon tests which are designed specifically to find bugs on the silicon.

4.38 TAP

Test Access Port. This is the JTAG-compliant controller which uses a small set of pins not used for processor functionality to allow a wide range of debug commands to be executed post-silicon.

4.39 Tapeout

The process of creating the masks which will be used to create the actual silicon. Tapeout is the end of the pre-silicon process for any particular stepping, and is one of the key lines of demarcation for the pre-silicon team.

4.40 TDE

Trace Delivery Engine. Cluster which caches traces of uops decoded by the MITE, and sends them to the RRRR cluster to be scheduled.

4.41 TLA

Trace Logic Analyzer. Used to capture the activity on the pins of the processor for analysis.

Three Letter Acronym.

4.42 Watch Windows

Watch windows are used as an enhancement to the debug process, allowing the user to visualize the behavior of the chip at a higher level of abstraction than the RTL signals. Perl script files with a ".w" extension are used to process the raw RTL signals based on expert user translations. For example, the "ROB" watch window is used to show the current state of retirement and allocation in the machine, especially what uops are in the machine and which is the next to retire.

4.43 ZBB

Zero Based Budget. Planning tool used to list out priorities, cost those projects, and identify which projects will receive funding and which will not.

The Art of Pre-Si Val: Chapter 48

The Art of Val Methodology Forum

By: [Phil Atkinson](#)

1 Abstract

This chapter describes how to organize and effectively run an Art of Val methodology forum. The forum is a weekly discussion group made of up of Validators new to the concepts in [The Art of Pre-Silicon Validation](#). Each week, the group discusses and debates roughly one chapter of [The Art of Pre-Silicon Validation](#). The goal of the forum is to have the participants understand the “why” of Validation. This chapter describes the format of the forum, roles of people in the forum, a suggested schedule, discussion questions, and other backup material.

2 Chapter Revision History

Rev.	Date	Description	Document Owner	Primary Reviewers and/or Approvers
1.0	8/18/2006	Initial draft	Phil Atkinson	DEG uAV Meth WG
1.5	10/13/2006	Updated from Level 1 feedback	Phil Atkinson	DEG uAV Meth WG
2.0	5/16/2012	Updated with Art of Val feedback relevant to all groups	Allan Rudwick	Michael Bair

3 Contents

1 Abstract.....	869
2 Chapter Revision History	869
3 Contents.....	870
4 Purpose.....	872
5 Background concepts	872
5.1 Art of Pre-Silicon Validation	872
6 Forum overview.....	872
7 Organizing the forum.....	872
7.1 Roles	872
7.1.1 Organizer	872
7.1.2 Facilitator	874
7.1.3 Content expert	874
7.1.4 Participants	875
7.2 Schedule.....	875
8 Leading the forum.....	876
8.1 Ground rules	876
8.2 Leading the discussion	876
9 Support material.....	877
9.1 Sample Emails.....	877
9.1.1 Welcome message to participants	877
9.1.2 Wrap-up and next agenda message	878
9.1.3 Reminder message to facilitator	878
9.1.4 Reminder message to participants	879
9.1.5 Invitation to content expert.....	879
9.2 Discussion questions	879
9.2.1 “Introduction to Pre-Silicon Validation”	880
9.2.2 “The Life of a Project”	880
9.2.3 “The Validation Mindset”.....	880
9.2.4 “Stimulus”.....	881
9.2.5 “Checkers”	881
9.2.6 “Coverage”	881

9.2.7	“Debug”	882
9.2.8	“Becoming the Microarchitecture Expert”	882
9.2.9	“The Life of a Bug”	882
9.2.10	“The Post A-Step World”	883
9.2.11	“Execution Testplan Methodology”	883
9.2.12	“Evil Validation”	883
10	Summary	884
11	Future Work	884
12	Acknowledgments	884

4 Purpose

The purpose of writing The Art of Pre-Silicon Validation was to teach new Validators the methodology needed to do their job. In doing so they are able to make the right Validation trade-offs in their day to day activities, increasing the quality of the logical health of the design. While reading the book would lead to some level of comprehension, it is this team's experience that discussing the topics with peers and experts leads to a greater understanding of the material and its application. To this end, the Art of Val methodology forum was created.

5 Background concepts

5.1 Art of Pre-Silicon Validation

The Art of Pre-Silicon Validation is the collection of chapters describing the what and why of Pre-Silicon Validation. It is the book your reading right now.

6 Forum overview

The Art of Val methodology forum is a series of weekly, hour long discussions, lasting several months, covering various chapters in The Art of Pre-Silicon Validation. On average, the participants discuss one chapter a week. The participants in the forum should be on-site and in the same room. All participants are expected to actively discuss the topics and to be there every week. It does not work to have people call into the forum.

7 Organizing the forum

7.1 Roles

There are four major roles in organizing and running an Art of Val methodology forum: the organizer, the facilitator, the content expert, and the participants. The organizer owns the forum and is responsible for making sure the forum goes smoothly. The facilitator is chosen from among the participants on a rotating basis. The content expert is a rotating guest invited to come and give their insights on the chapter being discussed. A content expert is not mandatory for each chapter, but is highly recommended. The participants are Validators new to the Validation group. They are expected to attend the forum each week.

7.1.1 Organizer

The organizer manages and owns the methodology session. He or she is responsible for the logistics of the session and is also responsible for how the forum progresses. If something isn't right, it's the organizer's job to notice and to do something about it. It is not the organizer's role to teach the content to the participants. The organizer should be familiar with the material, but does not need to be an expert. He or she should be at every meeting. Here's the list of responsibilities:

Setting up the first forum:

- Schedule the room and time for the forum. The forum should be held for an hour once a week. If at all possible, it should be at the same time and same place each week.
- Get buy-in from participant managers
- Get hard copies of *The Art of Pre-Silicon Validation* and hand out to participants at least one week before the first session.
- Plan which chapters will be covered and in what order (see Schedule section below)
- Send out the welcome email (draft copy in Support Material)
- Invite and schedule a content expert for the first topic (draft copy in Support Material)
- Be the facilitator at the first session
- During the first meeting, discuss the forum roles and responsibilities
 - Facilitator, content expert, or the organizer will not teach the material

During each forum:

- Be the gatekeeper to keep the discussion on track
- Be the time keeper.
- Enforce a “no laptop” policy each meeting
- If the discussion is quiet, ask questions to the group to further the discussion. The organizer can use the discussion questions listed at the end of this chapter.
- Make sure all people are participating by asking specific people questions during the session
- Make sure all important content is covered.
- Decide if the topic needs more time. If so, allow the discussion to continue at the next meeting.
- Pick a new facilitator in round-robin order before adjourning. Every participant should be facilitator at least once.
- Handle any ARs that come up

After each forum:

- Send out a wrap-up email along with an agenda for the next week’s session. (draft copy in Support Material). These are not minutes. No one needs to take minutes during the forum.

Before the next forum:

- Check in with the facilitator to make sure he/she will be prepared
- Invite and schedule the next content expert
- Reschedule sessions if too many participants will be out

7.1.2 Facilitator

The organizer chooses the next facilitator at the end of each session. The facilitator's job is to lead the discussion for his or her assigned chapter. He or she does this by reading the chapter before the next session, comprehending the material and writing down pertinent questions to ask the group about the material. It is not the facilitator's job to teach the material, only to facilitate discussion. Each participant should be a facilitator at least once.

Before the assigned forum:

- Read the assigned chapter
- Write down several open-ended questions that cover the breadth of the chapter
 - Attempt to hit all key points in the chapter
 - Do not summarize the chapter or go over the chapter section by section

During the forum:

- Ask the questions to participants
- Ask follow-up questions when needed

7.1.3 Content expert

The content expert should be just that, an expert in the chapter being covered.

It is up to the organizer to decide if a content expert is required for a topic. The organizer may act as the content expert if he or she has sufficient experience in a topic. Or the organizer may decide that a content expert isn't needed because the topic is straight forward, for example the "Life of a Bug" chapter.

During the forum:

- Ask questions of the participants to further the discussion
- Relate stories or their opinion when appropriate
- Relate discussion to current project

Before the forum:

If the content expert is familiar with the Art of Val forum and the content of the chapter then he or she is not expected to prepare any material for the forum, and is not expected to teach anything. During the forum, the expert can ask the participants questions to encourage discussion on the topic. The expert can also relate past experiences and "war stories" to highlight the material. The content expert should be careful to not monopolize the discussion, instead allowing the participants to discover and internalize concepts through their own discussions.

However, if the content expert has not sat through an Art of Val Forum before, the facilitator should explain what is expected of a content expert and coach them through the session. It is important that the content expert not answer simple questions and help guide the discussion. Failing to do this will lead to frustration for the organizer and a less productive session. An organizer should help guide a novice content expert so that discussion progresses productively.

If the expert was not involved in the writing or reviewing of the chapter, the organizer will ask the content expert to read the chapter before the session. They should be prepared to discuss things they disagree with the chapter on and give the reasons for those disagreements, not simply blindly teach the chapter. They should not ignore their real world experience either.

7.1.4 Participants

The participants of the forum are the most important members. Each forum should be made up of 5-8 Validators new to the concepts in [The Art of Pre-Silicon Validation](#). Less than 5 participants doesn't yield enough discussion. More than 8 participants leads to a few people not being able to actively participate each week. Each participant is expected to participate in each meeting by answering and asking questions. For RCGs, it's generally better to wait a few months before they join a forum in order to get comfortable with the basics of Validation. Each participant should be a facilitator at least once.

Before the forum:

- Read the assigned chapter and be prepared to ask and answer questions

During the forum:

- Show up, even if they haven't read the chapter
- Participate in the discussion

7.2 Schedule

Below is a schedule used towards the end of the Haswell project family. At the time this schedule was first used, the project was in the Execution phase. Topics should be chosen that are relevant to the participants, so the team-focused chapters might be added for some groups and not others. Another option is to break up the schedule into two methodology forums: one that covers Front End Development and Execution tasks and methodologies, and another that deals with getting to Production and beyond. Not all chapters need to be covered.

The time of day the forum is scheduled is important. Since this forum requires participation, an 8am meeting most likely would not work. Before lunch and at the end of the day have been found to be effective times.

It is up to the organizer to decide if a topic needs more or less time depending on how the discussion goes in the forum. If the discussion is lively, the participants are engaged, and the points within a chapter are being addressed, the organizer can elect to spend more time on the chapter during the next session. In that case, the organizer should plan on starting the next chapter after the first chapter concludes sometime during the next session and should prepare accordingly.

This was a list for interns who had 10 weeks of availability to participate in the forum:

1. Introduction to Pre-Silicon Validation
2. Life of a Project

3. The Validation Disciplines
4. The Validation Mindset
5. Stimulus
6. Checkers
7. Coverage
8. Debug
9. Life of a Bug
10. Post A-step World

This is the extended list to be added new hires who did not have a time constraint:

11. Becoming an Expert
12. Execution Testplan Methodology

One overview chapter from each section:

13. Introduction to Microarchitecture Validation
14. Introduction to FPV
15. Mixed Signal Validation
16. Design for Test Validation
17. Microcode Validation
18. Architecture Validation

2 final concepts:

19. Turnin Gating Regressions
20. Evil Validation

8 Leading the forum

8.1 Ground rules

During the forum, there should be no laptops. This is not negotiable. The forum is only valuable to people if they are involved in the discussion. Laptops are too much of a distraction. If participants wish to take notes, they should bring a pen.

8.2 Leading the discussion

At the beginning of the session, the organizer should welcome people back and remind the participants of the chapter they will be covering. He or she should also introduce the content expert, if present. The content expert should then give a brief introduction of how long they've been in Validation and what they have done. The organizer should then turn the discussion over to the facilitator to lead.

The facilitator should ask questions that guide the discussion through the chapter, making sure to hit on key points. It's not necessary to cover all the material in the chapter. The organizer can steer the discussion to a particular sub-topic by asking questions if needed. The content expert, if present, can also ask and answer questions or relate past experiences.

If the participants aren't being very talkative, the facilitator and the organizer should ask specific people to answer questions. There aren't necessarily right or wrong answers, so participants should feel free to express their opinion. The organizer may also talk to the participants one on one outside of the forum to prod them into participating.

If the facilitator is marching through the chapter the organizer should feel free to steer the facilitator in the right direction. The purpose is to cover the chapter at a high level; the organization of the chapter should not frame the order topics are discussed.

9 Support material

9.1 Sample Emails

9.1.1 Welcome message to participants

Send at least a week before the first session to all participants and the participant's managers.

Subject: Announcing the Art of Val forum

With new people joining the group, it is time to start another edition of the Art of Val forum. The goal of the forum is to have the participants understand the “why” of Validation. In our experience it’s not enough to just know what to do. To be an expert Validator, you need to know why. We will do this by meeting weekly in a small group and discussing various chapters and topics in [The Art of Pre-Silicon Validation](#). The more effort you put into these discussions, the more you will get out of the forum.

When writing [The Art of Pre-Silicon Validation](#), the authors and contributors spent hours and hours discussing what we have done in the past and why we did what we did. This forum will generate similar discussions. We may not always agree with the chapters and that is fine. This forum is not a lecture. There won’t be a teacher and there won’t be tests. Instead, the forum is organized to encourage discussion.

Logistics:

I will serve as the organizer of the forum and identify a chapter (or a portion of a chapter) to read each week. Participants are expected to read the assigned chapter and be prepared to answer questions on the material. We have enough topics to run the forum for at least <X> months.

One participant on a rotating basis is the facilitator responsible for identifying several discussion questions or topics which pertain to the main ideas from that week's reading. Depending on the topic, we may also have a special guest content expert. The content expert's job is to lend his or her experiences to the discussion. It is not to teach the material.

At the weekly 1-hour meeting, the facilitator leads the discussion focusing on the questions they generated. Participants will give their opinions, suggest alternative approaches, provide rationales for preferences, ask for clarifications, etc. There are no right or wrong answers - what is important is that ideas are shared. I will help focus discussion if necessary and provide context based on my past experience. The point is to get folks thinking about the methodology, challenging why we did things the way we did, determining whether it could be done better, etc.

I realize there may be weeks where finding time to read is impossible. Please come to the forum anyway and contribute to the discussion. You will get more out of the discussion if you do read the material in advance. One suggestion is to schedule time in your own Outlook each week for that week's reading.

One very important rule – **no laptops in the forum.** The forum only works if everyone is focused and contributing. I will be very strict about this rule.

The first chapter to read is the Introduction to Pre-Silicon Validation. Our special guest expert will be <content expert>. I will serve as facilitator for the first meeting.

Let me know if you have any questions. I'll see you <date and time> for our first meeting. The meeting should be on your Outlook. Let me know if it isn't.

<Organizer>

9.1.2 Wrap-up and next agenda message

This email message should be sent shortly after each session.

Subject: Art of Val – next session

Hey all,

Thanks for the great discussion today and thanks to <last facilitator> for leading us through “<chapter name>.” Next week we'll start “<next chapter name>”. <next facilitator> has agreed to facilitate the discussion. <Content expert> will be our guest expert. The next session is at <time, date, and location of next session>.

Your homework is to read “<next chapter name>.” As always, if you don't get a chance to read it, please come anyway.

<Organizer name>

Date:	Next chapters:	Facilitator
01/12/23	The Validation disciplines	<Person A>
01/19/23	The Post A-step world	<Person B>
01/26/23	Interactions with Post-Si Validation	<Person C>

9.1.3 Reminder message to facilitator

Send this message at least two days before the next session or just stop by their office to remind them.

Subject: Art of Val

Hi <Facilitator name>,

I'm just checking in to remind you that you're facilitating the Art of Val this week.

Thanks,

<Organizer name>

9.1.4 Reminder message to participants

Reply to “Wrap-up and next agenda message” and add the following:

Subject: Re: Art of Val – next session

Hey all,

Just a quick reminder that our next session is <tomorrow>.

<Organizer name>

9.1.5 Invitation to content expert

If they are familiar with the chapter and the forum

Subject: Art of Val forum – content expert –

Hi <expert>,

I'm currently leading an Art of Val forum with participants from <X and Y> groups. Next week, we'll be discussing <chapter X> and I'd like to invite you as a content expert.

No pre-work is needed on your part. You won't be teaching or presenting anything. <The facilitator> and I will be leading the discussion. Your role is to ask questions of the participants to further the discussion and/or chime in with your own experiences. The forum works best when the participants figure things out on their own so **be aware not to monopolize the discussion**.

The forum is a delicate balance between the content expert, the facilitator and the participants. If you think you might be talking too much, feel free to defer to me about guiding the conversation. I may ask the participants to what they think or I may ask you for an example. Don't simply answer the questions from the participants. It can be good to single out a participant that is being shy. Things work best when everyone in the room is involved in the discussion.

The forum is at <time and date>. Does that work for you?

Thanks,

<organizer>

If they are not familiar with the chapter, change “No prework...” to

Your AR is to read the chapter before the session and think about examples that agree with or contradict the chapter.

9.2 Discussion questions

Below are discussion questions gathered from past Art of Val methodology sessions. The discussion questions may be used by the Organizer to further the discussion. It is recommended

that the Facilitator read a chapter and come up with his or her own questions before referring to these questions. It is not necessary to ask each of these questions during a session.

9.2.1 “Introduction to Pre-Silicon Validation”

- What is the goal of Pre-Silicon Validation? How do you measure that goal?
- How do you define success in Pre-Silicon Validation?
- Why is it better to find a bug early?
- Why is software validation different than RTL hardware validation? After all, RTL is just software.
- What’s the difference between testing and validation?
- What is high quality validation? Is it always needed?
- Is it OK to not do high quality validation on part of the design? Why?
- Why is overlap good in Pre-Silicon Validation?
- What are the three pillars of Validation?
- When has Pre-Silicon Validation “done enough” for tape-in?
- Why is it dangerous to only track passing tests?
- Is there a technical career path for Validators? Do you agree?

9.2.2 “The Life of a Project”

- How long are the project phases? What makes a phase longer or shorter?
- What is the tick/tock model?
- How thorough should you test in Front End Development and Execution?
- What are the goals of Validation in each project phase?
- Does Validation push out tape-in? Why or why not?
- How do you set coverage goals?

9.2.3 “The Validation Mindset”

- What are the goals of Validation?
- How should Validators interact with design and architecture teams? How should they interact with other Validation Teams?
- When should a Validator decide to focus on one area of the design?
- Can Validators trust anything? What should they trust?

- What would a ‘perfect Validator’ do? What does perfection mean in the real world?
- How can being a great communicator help a Validator?

9.2.4 “Stimulus”

- Why do we use directed and random tests?
- Should a Validation team write 1 test or 100 tests? Why?
- How can you focus testing on a specific feature with weights? What are some pitfalls?
- What would you do differently if you have features that are very close to the input stimulus versus having a lot of logic in between the stimulus and the logic you are trying to exercise?
- When should you be worried about interactions between features?
- How can you stress different DUT configurations?
- Are injectors that sense the state of the machine and then act good or bad? Why?

9.2.5 “Checkers”

- Why do we need checkers? What do we need to check?
- How long does checking code last? How might the code be reused?
- Who can specify what checking is needed? What constraints go into that decision?
- What are the benefits of assertions versus Standard TE checkers versus AFM-based checkers?
- What can Validators use to help plan checkers for proposed new features?
- What are false-failures? Why are they a problem?
- How can Formal Validation affect what checkers are written?
- How does emulation factor into checking decisions?

9.2.6 “Coverage”

- What is coverage?
- Why use coverage?
- Can’t a Validator code a coverage construct wrong? Doesn’t that make coverage less valuable?
- Is the goal of coverage to find bugs?
- What is frequency coverage?
- What are other forms of coverage?

- When should Validation start doing coverage?
- Should you start coding coverage monitors before the test plan is done?
- How do you define a coverage space?
- Does it matter if you hit all the conditions? Why?
- Is it OK to write directed tests to hit coverage? Why or why not?
- What do you use for feedback if there aren't bugs in an area?

9.2.7 “Debug”

- What does a failure mean?
- How do you decide which failures are important?
- Are certain types of bugs more important than others? How will this guide your debug?
- What are pitfalls in duplicating debug effort?
- Where should you place your faith when debugging?
- How will experience inform and guide your debug efforts?
- Is it important to keep track of your debug process? Why?
- When should you get help on debug? How readily should you help others?
- How will you find owners to help debug complicated failures? What information will you bring them?
- What should you do after finding a bug?

9.2.8 “Becoming the Microarchitecture Expert”

- Why should you become a microarchitecture (uArch) expert?
- Do you agree with the expert chart? Why or why not?
- Why is historical familiarity important to being an expert?
- Does everyone in Validation need to be an expert? Why?
- What are the obstacles to being an expert?
- Should every unit have an expert? Why?
- How do you keep experts?

9.2.9 “The Life of a Bug”

- Why is it important to file a good bug report?

- Who is the audience of a bug report?
- How are bugs found?
- Who's job is it to root-cause a bug?
- What are the components of a good bug report?
- What's involved in validating a bug?
- If you debug in an area you don't know, should you debug it or an expert?

9.2.10 “The Post A-Step World”

- What's the difference between Pre-Silicon Validation and post A-step Validation?
- What do Pre-Silicon Validator's do post A-step?
- What are the considerations in fixing a bug post A-step?
- What kind of validation needs to be done on RTL stepping changes?
- How does a Validation keep himself/herself utilized post A-step?
- What is ITP? CV? SV?

9.2.11 “Execution Testplan Methodology”

- What is a test plan?
- Why does uAV need test plans?
- What are the consequences of not having a test plan?
- How do you know you've written a good test plan?
- What's the most important part of a test plan?
- How does a Validator differentiate between what is “interesting” from what's “not interesting?”
- What are the pitfalls of white box testing?
- When does a Validator need to update the test plan?
- Should a Validator update the test plan due to bugs?
- How does a Validator decide on the priority of test plan entries?
- Why are test plans written in English and not a programming language?

9.2.12 “Evil Validation”

- What is evil validation?

- When should a Validator do evil validation?
- Is it necessary to do evil validation to tape-in a product?
- What are your experiences doing evil validation?
- What is the mindset of an evil Validator?
- Is it required to be a uArch expert to be evil?
- Who are good references on where or how to be evil?
- What are different ways to be evil?
- Is evil validation data driven?
- How do you trade off normal Validation work vs. evil validation?

10 Summary

The Art of Val forum provides an interactive way to introduce [The Art of Pre-Silicon Validation](#). This chapter outlined roles and responsibilities to run a successful forum. By following these guidelines, Validators are set up to proliferate [The Art of Pre-Silicon Validation](#) methodologies in their jobs, increasing the quality of validation they do.

11 Future Work

Further discussion questions may be added for other chapters.

12 Acknowledgments

The Art of Val methodology forum was originally designed by Uber-Validator Bob Fisch. Thanks to Bob for his dedication in teaching uAV to the masses.