On

The Origin of Bugs,

Or

Understanding hardware bugs and how to avoid them

Bryan Dickman, Valytic Consulting Limited

Abstract

Where do bugs come from? What are the common ways that bugs are introduced into designs and what can design engineers and verification engineers jointly do to avoid them?

1 Introduction

Note that I am writing this whitepaper from the perspective of several decades in the semiconductor industry, working for a highly successful IP company (whose roots are firmly in CPU design), and working in the field of Design Verification, which means I have spent a lot of my career thinking about and analyzing bugs! This whitepaper does not set out to publish a new or novel approach to verification or bug discovery, but instead to provide an overview of bugs in a hardware design context, with some insights into what causes bugs and some strategies that can be used to avoid them. Some of these strategies are already familiar to software developers and may be equally applicable to hardware developers. Many hardware developers are already using some or all of them anyhow. Finally, as data science becomes increasingly accessible as a tool to solve many problems in many fields, including engineering, I take a very high-level view on how analytics can be applied to the bug analytics problem.

Note that the data visualizations shown in this paper are hypothetical drawings and not graphs of real data.

1.1 Why do we still have bugs (and jobs as verification engineers)?

How do you design a complex hardware IP such as a processor that is fully verified? Hardware is getting more complex and bugs are getting correspondingly more complex. It is sometimes mind-bending to fully understand the convoluted conditions that lead to bugs being activated. The chances of these conditions arising may seem remote when analyzing the bug using the verification environment. However, what you have to consider is that complex IP products such as CPUs, GPUs, IoT, ML and other System IPs are being deployed into billions of devices, within many unique environments, and executing vast masses of OS and application software at GHz speeds. This means that those bugs observed in your verification environments, which you believe to be extreme and rare, might adversely affect entire product lines of deployed technology with many unforeseen consequences. The stakes for hardware IP developers have never been higher.

Building functionally correct products is the shared responsibility of both the design team and the verification team. It's a false assumption that complex designs like processors can be exhaustively verified, and that the final design can be free of all bugs.

I assert that:

All complex designs contain bugs.

Verification can never achieve completeness and designs must be built in a way that fully accounts for verification achievability.

My second assertion is that:

*Verification is a resource limited **'quest'** to find as many bugs as possible before shipping.*

It is common practice to separate these two roles out and identify them as distinct disciplines. Verification Engineers should employ strategies that increase the chances to find all bugs. Design Engineers should employ strategies that minimize the risk of bugs occurring.

These two disciplines require many shared skills and some unique ones as shown in the following Venn diagram.

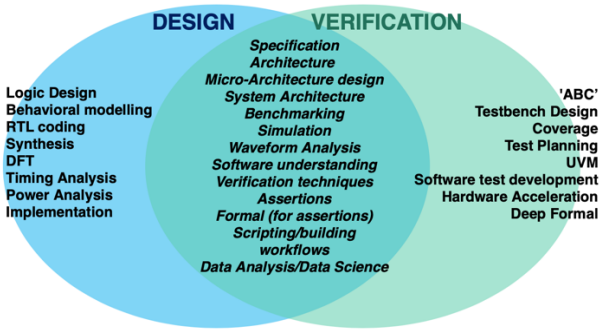

Figure 1 Design and Verification Skills Venn Diagram

1.1.1 Hardware Designer Engineers

Design teams focus on achieving both functional and physical goals such as power, performance and area (PPA). These goals define the product as much as the functionality and make the difference between a successful product and an unsuccessful one. Designers are highly skilled in logic design and understand how the HDL[1] translates into actual gates and registers as it passes through front-end design tools such as synthesis and timing-analysis. They understand complex architectures and micro-architectures and how to exploit them to optimize functional performance. They must understand system level architectures and how their product works in a system environment and how software will use it. They know how to balance functional performance with physical performance. For example, how to optimize 'work-output per unit of energy' for a processor. There are many front-end EDA tools to master and they need a good understanding of implementation workflows and how their RTL code transforms to physical circuits on silicon. Good designers are constantly aware of the power consumption of their logic, and design everything with power-efficiency and low-power in mind.

Design is a discipline of craftsmanship where good design emerges from bringing together decades of experience, knowhow and training, skills and talent, ingenuity and innovation, and attention to detail.

1.1.2 Hardware Verification Engineers

Designers are perfectly capable of performing the verification of their own designs, but it is accepted good practice, where resources are available, that the verification tasks are undertaken by dedicated verification engineers who have a slightly different skillset and with the added benefit of an independent interpretation of the design specification. This independent understanding of the specification can often expose specification ambiguities or misinterpretations in the early test planning phase, before expensive verification testing cycles have been run.

Verification Engineers (sometimes referred to as Design Verification Engineers or DV Engineers) need a broad set of disciplines which may be more biased towards software development these days, but also with a head for understanding hardware specifications, architectures and RTL code. There are many verification tools and languages that have emerged over recent decades that DV Engineers must master such as SystemVerilog, Emulators, FPGAs, Formal Verification etc. They will also will be constructing much of the stimulus with software programs written in various languages from Assembler to C, and they will be masters of scripting and building bespoke tools and infrastructures to automate and optimize testing to reach bugs more quickly, extend the search areas more deeply, and optimize the available infrastructures upon which verification is executed. They must also be masters of data analysis, since verification work generates much data. Sifting and analyzing data to understand verification progress while the design and verification environments are constantly changing, is also a major challenge. Coverage is a key concept for turning verification into a measurable process.

Verification engineers have a few specific personality traits. After all they appear get their kicks from demonstrating that something is broken!

I like to describe this as the **ABC** verification philosophy.

*Assume nothing; **B**elieve no-one; **C**heck everything!*

However, they have the same objectives as the design team. They want to deliver a bug-free high-quality product, and they want to share the glory for that as much as the design team. When serious bugs are missed and

[1] Hardware Description Language (Verilog, VHDL, SystemVerilog)

make it through to the end product, the responsibility is equally shared. Teams should adopt strong practices of blame-free Root Cause Analysis (RCA[2]) to understand if the escape was due to an omission of verification or a weakness of design, or a combination of both. They can then apply appropriate corrective actions to one or both areas to ensure that the learning is captured and implemented for the current product, sibling products, future products and maybe even looking back at legacy products. It is vital that good practices of knowledge sharing ensure that bug learnings are shared openly within the organization through RCA. When bugs affect shipped products, there must be credible evidence of organizational learning that will rebuild customer confidence. This philosophy of continuous marginal improvements will aggregate to significant and measurable overall improvements over time.

The nature of the task does foster some healthy competition between the disciplines to build clean code and to wring out all of the bugs, no matter how obscure they might be. Both roles are intellectually demanding and rewarding.

2 When is a bug a bug?

All complex designs contain bugs, without exception. No design can ever be bug-free. Verification is an NP-Hard[3] problem, i.e. there is no perfect solution. Verification is a time and resource-limited quest to find as many bugs as possible, caveated by evidencing no known impactful bugs (known-knowns[4]), and also to ensure that coverage goals and stability targets have been achieved (known-unknowns), at the point of product delivery. Of course, this leaves the remaining bugs that will likely be present in the shipped products (the unknown-unknowns), and we hope that should they emerge, they are not impactful and/or can be satisfactorily worked around in the field.

So, what makes a bug into a bug? Let's talk about functional bugs for now. I will omit the usual tale of the moth that got trapped in a relay[5].

2.1 Observable

Generally, bugs are recognized as erroneous behaviors that lead to an observable error. This error might be observed under normal operating conditions, and therefore should be easy to detect. However, it might require a combination and/or sequence of events to trigger the condition that makes it rare. We need to be careful about 'rare' because something that takes a lot of effort to find in a verification environment, might trigger with alarming regularity in the final device running real software payloads. Errors that manifest as some sort of deadlock state are easier to detect in a real system, and it may be possible to work around with a reset from a higher privileged system. Errors that manifest as 'denial-of-service' may require additional detection mechanisms to be built into the verification environments that will catch these conditions with time-outs. Errors that manifest as a data corruptions can be 'silent errors' that could be more damaging to the user application – imagine a data corruption in your banking application! The infamous Pentium FDIV[6] bug was a data corruption, albeit one that was never observed in real systems and discovered by academics.

2.2 Non-Observable

An unreachable bug might exist in the code where the full set of conditions required to trigger the bug cannot occur and the bug can never propagate to an observable error. Maybe some or all-but-one of the conditions can occur, yet there is one key condition that prevents the error from occurring. Is this a 'bug' or a 'feature'? Such problems (or non-problems) might be exposed through code review, waveform analysis or formal verification. Is it wise to fix this class of coding error? That depends on where you are in the development cycle. Changing the code might introduce a risk of inadvertently turning an unobservable error into an observable error if the code is not 100% understood. Leaving the code as is, might risk that a future change could be disruptive if the code is non-intuitive, or some other dependency elsewhere affects the observability of the error.

Non-reachable bugs may be 'time-bombs' that cause a problem later on if the design is changed.

They negatively impact code readability and maintainability. Should design teams clean up all such issues up with code re-factoring or do design teams typically leave such code well alone for fear of breaking the design? If an unsafe coding practice has been used, then there is risk that changing it could introduce new bugs, especially if "it's not my code", or "I wrote it a long time ago" and "it's complicated and I'm not fully sure how it works any more".

This is where designer and verification engineer's experience and judgement come into play.

[2] https://en.wikipedia.org/wiki/Root_cause_analysis

[3] https://en.wikipedia.org/wiki/NP-hardness

[4] https://en.wikipedia.org/wiki/There_are_known_knowns

[5] https://www.computerhistory.org/tdih/september/9/

[6] https://en.wikipedia.org/wiki/Pentium_FDIV_bug

2.3 Vulnerabilities

There is a further category of bugs that can be better described as vulnerabilities. With security being at the forefront of concerns for modern computer technology, and being architected and built into most systems today, developers always need to be concerned about leaving unintended vulnerabilities in the design that could be exploited by malicious code, for example to access or leak secure data by running non-secure software. A recent and well publicized example of this is the security vulnerability named as Spectre[7]/Meltdown[8], which affected a number or processors across the industry. Again, these bugs were not found in real systems but were exposed by the Google Project Zero team in an effort of penetration testing or white-hat hacking[9].

Other vulnerabilities fall under the category of 'denial-of-service' attacks, where malicious code has the potential to put the device into a locked state (e.g. live-lock or dead-lock) so that the device is hindered from making forward progress or responding to inputs.

This class of bug is oftentimes discovered through explorative soak testing using constrained-random generation methods. It can be the case that regular software such as Kernels or Applications, or other code that has been compiled, will never observe the vulnerability, and so it exists only for the case of contrived code. But of course, as a vulnerability, there is a possibility that malicious code might exploit it for ill intent, and because of this these sort of bugs are often considered to be critical.

Finding such errors using the usual verification environments can be challenging and efforts must be taken to model threats and pro-actively verify them. Security verification is a major consideration for most hardware and software developers.

2.4 Auxiliary functions

Finally, there is a class of bug that may be considered to be less impactful if it only impacts secondary or auxiliary functionality of the system such as debug for example. So, a processer for example, works correctly in normal execution modes, but an error might manifest when using the debug capabilities which might be considered an inconvenience rather than rendering the device unusable. In some cases, it may be possible to work around this error, in others it might render the debug function unusable in which case this is a more serious product usability issue, especially for software developers.

Similar to debug functions, other supporting functions such as performance or event counters might be considered to be non-critical to the operation of the target application but are helpful for software developers to benchmark and optimize their code. Bugs in these functionalities might manifest as accuracy errors that would lead to false assumptions for the software developer.

2.5 Safety-critical and Reliability

With products like processors going into more and more safety-critical applications, safety and reliability is an increasing concern for product designers. Products are adding features to account for the reliability of the silicon in terms of detection and correction of soft-errors (or single event upsets – SEU), where one or more bits can be flipped (typically in DRAM) by the presence of alpha particles (from impurities in the packaging) or background radiation. Designs typically employ ECC[10] functionality to detect and correct single-bit failures and to detect multi-bit failures (SECDED) for data stored in memory. More sophisticated systems might employ lock-step schemes where duplicated functionality and comparison logic is used to detect behavioral errors from SEUs affecting logic circuits as well as memory. As with all functionality, this is also prone to functional bugs. The severity of such bugs depends on the application. For safety critical systems such bugs may be intolerable. For non-safety-critical applications, they may be of lesser concern. For example, if a bug requires a sequence of soft-errors to occur with a certain timing, the probability may be the product of two rare[11] events, and the outcome may still be a fatal error, but perhaps with incorrect error logging. In a similar way to vulnerability verification, detection of this class of bug requires some different verification strategies such as injecting faults to trigger logic that is otherwise quiescent in normal operation.

2.5.1 Performance Bugs

Sometimes bugs in functionality do not manifest as functional errors but can impact the delivered performance of the product. If this is deemed to be in contradiction of the specification, it might be considered to be a bug, if not by the developer, at least by the customer, depending on the severity of the performance loss. So, in this scenario we are saying an error in coding does not result in incorrect function, there is no corruption, deadlock or incorrect computation, but the coding error has degraded performance in some way. A

[7] So-named because the vulnerability arises from processors that implement speculative execution and that the problem was considered hard to fix and would therefore "haunt us for a long time".

[8] https://meltdownattack.com

[9] https://en.wikipedia.org/wiki/White_hat_(computer_security)

[10] https://en.wikipedia.org/wiki/ECC_memory

[11] Studies have shown error rates ranging from 10−10 error/bit·h (roughly one bit error per hour per gigabyte of memory) to 10−17 error/bit·h.

typical example of this class of bug might be a transient 'starvation' scenario, where under certain conditions, which might be rare, a request is starved of a response for significantly longer than expected, and this can manifest as a hiccup in throughput that may not be noticed immediately.

This class of bug is problematic for functional verification. They may be detected from analysis of benchmark results or performance verification suites. They are often detected from code reviews or simply observed when eye-balling waveforms and realizing that something is not behaving as intended.

2.5.2 Clocking and Reset Bugs

Finally, there is a class of bug related to clock domains crossing and asynchronous events such as resets. There are specific tools available to check these cases, but we will not discuss this class of bug here today.

3 When are bugs found?

The impact and severity of bugs very much depends on when they are found. As a rule, bugs found earlier in the development cycle are much less impactful and easier to fix, although in extreme cases they may lead to some level of re-architecting or be indicative of the need to refactor large areas of code. However, this is much better done in the early phases and may be too costly or not possible in the later stages of development.

As a working hypothesis, the rate of finding bugs over the development cycle may resemble a Gaussian distribution with the mid-point being around or ideally before the beta quality point but before the first release point. More than 90% of bugs are found within this verification window where the effort tends to be in developing and bringing up both the RTL and the verification environment in parallel and achieving substantial levels of basic coverage. However, the remaining ~10% of bugs will be found at the later stages, where the effort is more focused on bug-hunting through deep soak testing, stress testing, and coverage-closure to hit the remaining harder-to-reach coverage points. The machine and human effort applied to finding these bugs is disproportionately high, but the significance of these bugs is also much higher, since their impact is potentially much greater and more costly.

As a hypothesis, >70% of the total verification effort (or cost) may be consumed in wringing out the last <10% of the bugs, so these later bugs are high effort/expensive bugs to find!

Any productivity improvements or efficiency gains realized for this phase of testing can deliver a significant saving on the overall development cost for the product. Here's my whiteboard view of what this looks like in theory (clearly not real data!).

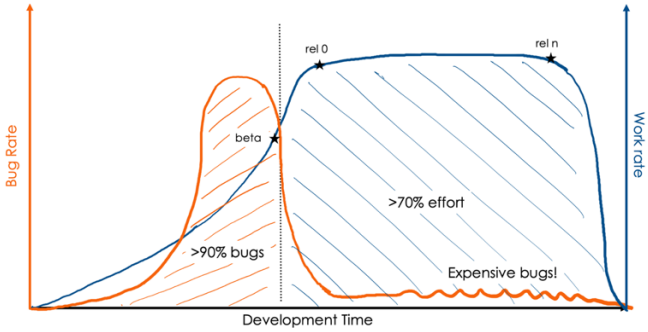

Figure 2: Bug discovery correlated with verification work

4 Recognizing where bugs come from

4.1 Common Root Causes

It's useful for designers and verification engineers to have a good understanding of where bugs come from. Root Cause Analysis as mentioned earlier is a great way to build up learning retrospectively after bugs have happened. Product teams can always review the post-mortems or retrospectives[12] from other teams to check that they do not repeat the same mistakes, but how do designers avoid bugs in the first place and recognize situations where the risk of bugs is high?

Bugs can occur either while creating new code, or when changing code. For example, bug-fixing or performance tuning bugs generally occur while changing code. Specification or interface bugs tend to occur when creating new code but can still also occur due to code changes.

The following table summarizes the typical root causes for bugs. As with all data shown in this whitepaper the data is not real and is for illustration purposes only.

[12] If using an Agile (https://en.wikipedia.org/wiki/Agile_software_development) approach.

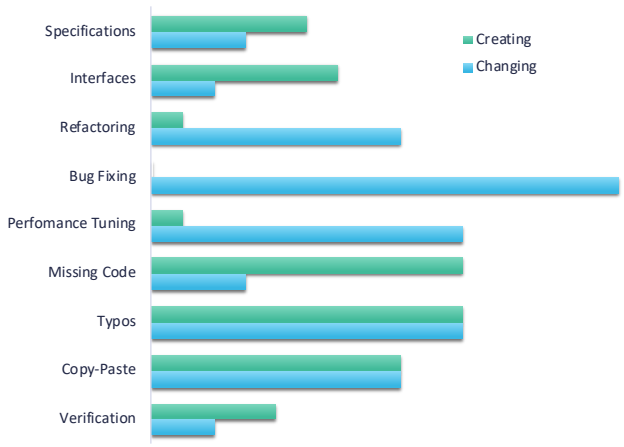

Figure 3 Bug Root Causes

4.1.1 Specifications

Specifications are a common root cause of bugs. Errors in the spec are likely to propagate into the RTL coding unless detected by spec reviewing. However, it may not be an error, and more of an ambiguity, that leads the spec reader, the designer or the verification engineer, to make a false assumption. With luck, the designer and the verification engineer will make different assumptions about what is the correct behavior, in which case there is good chance that the verification environment will expose the difference, and then the team has to arbitrate with the spec owner as to which is correct and hopefully amend and clarify it. This is a spec bug. However, it is also possible that both arrive at the same, potentially incorrect understanding, in which case the verification environment cannot detect the error and we have a 'common-mode-failure'. Unless specifications are executable, there is a manual and intellectual step to realize the spec into the implemented design and the corresponding verification environment.

Further, specs have been known to change (don't laugh!). Spec changes can be very costly and disruptive when following a standard waterfall approach for product development (historically typical for hardware development projects), but they are expected, anticipated and managed for those projects following an Agile development process. However, late spec changes can mean that functionality needs to be 'bolted-on' in some cases, which can lead to non-elegant coding, unless the designer chooses to refactor the code to take account of the new requirements. The latter is the ideal, but in practice not always achieved due to schedule pressure. This obviously leads us back to poor code health and the risk of the sort of bugs that arise from poor coding style.

4.1.2 Structure and Interfaces

Sophisticated hardware products these days are usually too complicated for any one individual to maintain a complete understanding of all functionality at any one time. So, the problems have to be broken down into logical units for both design and verification, and the interfaces between these logical units also define the interfaces between the people in the teams. Good interfaces are pivotal in building a good hierarchical design.

Most complex designs employ more than one designer. Wherever the work needs to be partitioned between humans, the boundaries need to be clearly defined with well-defined interfaces. Most designs are partitioned to multiple depths of hierarchy to ensure that functionality can be encapsulated into humanly understandable chunks. Each partition and sub-partition must have a rigorously defined interface that both sides (usually different developers) fully understand. Not surprisingly, interface misunderstandings are a common source of bugs. If an interface definition is at all ambiguous, it is highly probable that different developers will make different assumptions. They will develop and test their unit or module in isolation finding no problems, but when integrated together, they may fail. The gross miss-understandings should become apparent very quickly and are generally easily fixed, although in some cases might lead to significant rewrite once the interface ambiguities are resolved. More subtle errors might not be discovered until much later or need complex conditions to trigger them.

4.1.3 Code Churn

In common with software development, RTL code development tends to be a somewhat iterative process. It's not usual to capture perfect code that never needs to be further updated. Code can churn for various reasons as follows. Each time the code is updated there is a risk of deteriorating code health and introduction of new bugs. There are four main causes of code churn: -

- **Missing features** – the full set of features may not be captured in the first cast of the code, but incrementally added, and this can affect the elegance of the initial code if it wasn't fully considered from the start.
- **PPA tuning** – designers need to design for function and PPA, but sometimes function takes priority and code is subsequently iterated to achieve PPA targets. These performance changes may be disruptive and require a rethink of the micro-architecture or some re-structuring of the design, and they can often increase code complexity as 'clever tricks' are applied that might be non-intuitive on first inspection. For example, re-pipelining optimizations where logic is moved from one side of a register to another in order to meet timing requirements or save logic. Sometimes multiple iterations occur to achieve the target, each time potentially deteriorating code health and increasing bug risk.
- **Bug fixing** – some bugs may be simple to fix, without impacting code health. More complicated bug fixes

may require multiple lines of code or multiple modules to be updated which increases the risk or making a further mistake. Sometimes bug-fixes are temporary patches to the code, with the intention to revisit[13] and clean up or refactor the code at a later date. Complex bug fixes can take several goes to get right and we find ourselves in a cycle of bug-fix upon bug-fix in an attempt to get a clean pass of verification. When this happens, you know that code health is deteriorating and that might cause a problem later on. In some cases, it is prudent to take stock and reassess the code and consider if refactoring or rewriting is going to be a better course in the long run.

- **Refactoring** – is normally done as a bug-mitigation strategy. Cleaning up the code makes it more readable and maintainable and so should reduce the risk of bugs. Of course, sizable code re-writes will also increase the risk of introducing new bugs through lack of understanding of the intended behavior maybe, especially if the refactoring is being done by a different author.

4.1.4 Typos and copy-paste

For software and hardware developers, typos are the curse of code authoring. Of course most of the glaring typos are picked up by the HDL compiler, but it is so easy to miss a term in an expression through a lapse in concentration, or an interruption, or to miss-type a signal name by a single character (yet for it to still be a valid signal name so that code compiles despite this). It is also historically more common in hardware coding than software coding (where good practice is to reduce code duplication with functions) to copy-paste large chunks of code to replicate structures in the design. This is especially true when replicating logic for performance tuning, or when there is a behavioral structure that is similar but functionally independent from the original. These replicated structures can quickly get out of step if the linkage between them is not tracked. A code update for one structure could be accidently missed in its copy-pasted cousins. These errors can sometimes be difficult to pick up from eye-balling the code.

These are human errors, but we hope that good peer reviewing, continuous integration (CI) testing, and basic verification will quickly pick these errors up. A good Integrated Development Environment (IDE) can help to minimize these simple errors during code authoring, but some will be missed. Of course, some escape only to be very troublesome much further down the line.

4.1.5 Verification Bugs

It would be remiss not to discuss bugs in the verification environment since it is also a complex piece of development, oftentimes similar in complexity to the RTL Design. It also models the behaviors of the design in order to perform independent checking. Since the verification environment is also developed from an understanding or interpretation of the design specification, it is prone to similar errors which can lead to one of the following situations: -

- False Positive: the test passes but fails to pick up an erroneous result or behavior in the design.
- False Negative: the test fails but the design behavior is correct.

The ‘false-negative’ is easier to detect since the test reports a failure and debug needs to occur. However, there is a risk that the wrong conclusion is reached. The verification environment may be assumed to be correct, and the design is then incorrectly adapted to make the (incorrect) test pass. A new bug in introduced.

The ‘false-positive’ is more worrying as an error in the design behavior has occurred but not been detected, so this could be a ‘bug-escape’. This can be down to several reasons. Either there is missing test coverage from the verification environment (and this is not seen by analysis of coverage data, e.g. there may be missing functional coverage), or the designers and the verification engineers have made a ‘common-mode’ assumption from their misinterpretation of the spec, i.e. they both got it wrong in the same direction. Hence the design contains a bug, and the verification environment contains the same bug.

Because of this risk, it is important that the same level of scrutiny (independent if possible) is applied to the verification codebase as the RTL design code base.

4.2 A word about coding style and readability

Coding style and code quality can be a major factor that increases the risk of bugs. If code does not conform to organizational coding rules, it can quickly become hard to read and functionality becomes obfuscated. This is where knowledge and understanding can be lost. The designer himself may struggle to retain intimate knowledge of how the code works with the passage of time, let alone others being able to understand it should they be in the unfortunate position of having to fix someone else’s code at a later date. Some developers, in software and hardware development, find commenting and documenting source code to be tedious and an overhead that hinders momentum and productivity. It also becomes an overhead for maintenance. Out of date and incorrect comments are

[13] Hopefully the author will leave a searchable “REVISIT” or similar string in the code.

likely worse than no comments as they are misleading, e.g. when an update is made to the code (to fix a bug or to implement a performance optimization) but not to the comments in the interest of expediency. Again, this is part of the design engineering craft. Good designers know how to use comments and documentation appropriately and how to write code that is clean, well-structured and where the behavior is implicit and easy to understand.

Naming conventions aid code readability. Projects should at least follow the agreed project coding rules and naming standards, if not the organizational ones, especially if the project team spans the organization or needs flexibility to do so in the future.

Coding rules need to be obeyed and checking needs to be automated using an HDL linting technology. Linting is a well-established software methodology to improve code quality, but also well established for HDL for many years now. Many of these tools provide out-of-the-box rulesets, or teams or organizations may develop their own custom rulesets.

If a project team was to lose a key designer before the product is complete, their code might be considered to be abandoned. Another designer will need to pick up ownership of the code and be confident in understanding it so that they can change or develop the code further to fix bugs, add functions or optimizations. Oftentimes when code is hard to read, the adoption risk can be higher than a code re-write. A code re-write can seem to be expensive and disruptive, although in the long run it will likely save time and cost.

Further, if a project is reusing a lot of code from a previous project, they may have a similar issue to the one above where the inherited code is less well understood or conforms to a different set of coding rules and styles. The original author may or may not be available to help. If said code needs to be modified and cannot be treated as an entirely reusable block (i.e. no changes at all are required), there is risk of introducing bugs through lack of perfect understanding of the original intent. Design teams need to assess the risk when reusing code in this way, and oftentimes although seemingly costly and inefficient, it is better to rewrite the functionality using the inherited code as a reference. This way the team knows that they can support the rewritten code and understand it to the same level as other units in the design.

4.3 A word about Complexity

Complexity can arise in RTL code for very good reasons. The architecture that is being implemented may be inherently complex, or the chosen micro-architecture might be necessarily complex in order to meet challenging PPA targets. After all, these things are the attributes that will differentiate the product from the competition. However, complexity in the RTL code needs to be managed. Different RTL designers may have different personal styles of writing RTL code, some being very comfortable with a high degree of complexity in their code, others may be less so, and all could still be operating well within the project or organizational coding standards. Not all units, blocks or modules are equal. Some are purely structural and mainly consist of wiring to connect up other sub-blocks. These modules can be very repetitive and generally void of logic. Others might implement complex state-machines or algorithms. Some designers prefer to control the synthesis tools very tightly with explicit gate-level coding. Others prefer to code at a more behavioral level and leave the synthesis tools to arrive at the optimal solution. In general, the behavioral code is easier to read than the gate-level coding style, but sometimes designers need to code at this low level in order to constrain the implementation to meet their exact requirements.

Additionally, clock-gating and low-power mechanisms can add significant complexity to designs and may be non-trivial to understand when reviewing the code.

There is no commonly agreed standard way to measure complexity in RTL code. Software developers have been using lines-of-code, McCabe cyclomatic-complexity[14] or code indentation-complexity[15] to get some measure of code complexity, but these do not always work so well for HDLs. A search of the internet will find a handful of tools that claim to offer HDL complexity analysis, often based on McCabe cyclomatic-complexity. If you think about complexity for logic circuits, you might also consider how deep and wide the logic paths are between registers (the logic-cone-of-influence) i.e. how many terms are involved in determining the input to a register.

Designers typically rely on experience and engineering judgement to estimate the complexity of RTL code, and may refactor code to reduce risk when uneasy about their ability to understand and maintain the code now or in the future.

Complex RTL code can (but does not always) lead to complex verification environments, and the propensity to make coding errors then exists in both the RTL code and the verification code.

Clearly, complexity increases the risk of bugs, and the bugs themselves might be subtle or complex.

They might not be triggered immediately, and then be difficult to debug, especially if the outcome from the bug does not become observable until a long time after the trigger point.

[14] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[15] https://github.com/adamtornhill/indent-complexity-proxy

5 Strategies to avoid bugs

How can design teams minimize the number of bugs that get coded into the design? As we have already asserted earlier, there is no such thing as a bug-free design, but there are classes of bugs that are extremely hard to find – the 'unknown-unknowns'. We can't account for them in test planning, because we don't know what they are. We don't have coverage goals to reassure ourselves that these cases have been both stimulated and checked, because we don't know what they are. We 'hope' that comprehensive random verification environments will eventually flush them out. We can check to ensure that our random constraints do not over-constrain the stimulus and that sufficient[16] 'assurance-cycles' have been run that the code appears to be stable. We can review and re-review everything in the verification environment and brainstorm the question "what else can we do?". We can adopt an approach of continuous improvement. When a bug is found, no matter how it is found, we need to ask the question "why was this not found earlier?". We need to review the testing around this space to see if it can be enhanced to increase the probability of triggering this bug sooner and with higher frequency. We also want to check for the presence of any sibling bugs that may be lurking and use this hindsight to consider if other areas of verification can be improved.

It's not enough to endlessly improve the verification environment (but we are going to do that regardless), and we need to look at how the design can be codified in a way that minimizes bugs in the first place, on the premise that not all bugs can be found.

5.1 Reviewing

Good design reviewing practices can stop a lot of bugs from ever even making it to the first simulation cycle. It's not fully reliable, but it has been shown to catch serious bugs so that they never 'pupate' into 'flying' bugs. They are eliminated at birth. This often means that these coding errors are not recorded or recognized as bugs, it's just a part of the code-writing process. Code writers can review their own code, and most will do so of course. Peer reviewing is more powerful as a second set of eyes will often pick up things that the code writer is blind to. A good hardware development workflow will always include a rigorous code reviewing process.

5.2 Design Risk Mitigation Strategies

As we discussed earlier, oftentimes complexity is necessary and unavoidable in order to achieve the performance and functional targets of the design. Sometimes significantly complex behaviors can be added to achieve marginal performance gains. The aggregation of all such gains can make the difference between hitting the targets or not, but it may be possible to disable individual optimizations in software, without crippling the overall performance of the device. Designers often practice this by adding hidden[17] feature downgrade configuration bits, which can be enabled should a crippling bug arise in the optimization logic. These programmer bits can be 'get-out-of-jail-free' bits should this scenario arise as it allows the device to still function thanks to a simple software patch, or at least buy some time until the revised and fixed hardware is available. The trick is to recognize when it is wise to add these bits. Clearly if it was necessary to enable all such bits, the resultant performance might no longer be within acceptable limits. Beware however, that adding logic to enable or disable behaviors extends the verification space and introduces some marginal risk that the added logic itself contains bugs. Further, be careful of any interdependencies between these bits that might manifest as unanticipated behaviors.

5.3 Designer Assertions

Use of assertions (typically SVA[18]) for hardware coding is now common practice in the same way it has been for software to catch unintended behaviors at the point of failure, rather than later once (or if) the error has propagated into an observable failure. Designers are well placed to codify assertions as they codify the RTL code. They are built-in checkers that are there only for the purposes of verification. In general assertions are not added as synthesizable code that end up being present in final silicon (though in some circumstances they could be, or it may be desirable to include them in the FPGA verification environment for example).

The value of assertions written by the designers is that they can be used to capture 'intent'.

Actually, the very act of writing assertions, which is codifying behavioral assumptions that might otherwise have remained in the designer's head, can catch bugs before they ever really come into existence.

[16] You have to analytically decide what sufficient means!

[17] hidden from the programmer's spec

[18] https://www.verificationguide.com/p/systemverilog-assertions.html

It's that additional intellectual step, where the designer might think "Oh, hang on, that doesn't quite work does it!", leading to better code quality and less bugs before we start. The assertions then also act as a really concise documentation of how the function is intended to work, and they are forever there for all verification cycles as guardian checkers that will pick up behavioral discrepancies and lead the development team straight to the root of the problem with minimal debug effort. Of course, as is the case with the verification environment in general, the error could equally be in the assertion and not the RTL code. But we would hope that those functional differences between code and assertion will become observable and can then be debugged and corrected. The risk of 'common-mode' failure still exists, however.

Further value can be gained from assertions when formal verification tools are used. If the designer assertions are essentially describing properties of the design, it may be that these properties can be formally explored and even proven. Applying formal in this way acting on smaller units of design can be very successful. Proven properties can give a much higher level of assurance that the function is correct under all possible circumstances (or at least hold true for a convincing number of cycles – a 'bounded proof'), and if the RTL is built up on proven sub-structures, we essentially have a 'correct-by-construction' approach to building the full design.

5.4 DevOps approaches from the software development world

DevOps was first coined in 2009 and is born out of the earlier movements of Lean and Agile. There has been a largescale adoption of the 'DevOps' approach for software development and deployment in recent years, and pretty much any grown-up software company that is provisioning a software platform to users, will be doing so with a DevOps approach. If you want a good introduction to DevOps try Ref [1], though this is a much-documented area. Any major social media platform you can think of, Cloud services, on-line banking platforms, etc. etc. will be following the DevOps methods out of necessity, since they are developing, delivering and operating business-critical platforms. They cannot afford to make mistakes when deploying new features to the platform, that might result in a service outage for hours, or days, or longer while their developers are working to issue a patch and restore the service. Through DevOps, product owners, developers, QA, IT operations, and security specialists work together, not only to help each other, but to ensure that the overall organization succeeds.

They enable a fast flow of planned work into production, performing tens, hundreds or thousands of code-deploys per day, while achieving world-class stability, reliability, availability and security. This is why DevOps is mission critical.

Hardware development teams might consider that they are all about 'Dev' and less so about 'Ops'. However, there are elements of DevOps that can be applied in the hardware development space. I won't attempt to explore all ideas here but some examples from Ref [1] include: -

- Continuous Integration
- Pair Programming
- Blame-free Retrospectives
- Trunk-based development
- Code Refactoring built into the workflow
- Swarming on defects
- Test-driven code development
- Telemetry (operational analytics)

5.4.1 Continuous Integration

Central to DevOps is the concept of CICD (Continuous Integration/Continuous Delivery (or Deployment). This is where development teams build and operate a CICD pipeline that starts with the commit of changes, which are then automatically tested by continuous integration suites, and then automatically built and deployed to the target platform usually through a pipeline of deployment environments such as development, test, staging and production.

Continuous Integration (CI) is a methodology borrowed from software development and commonly adopted for hardware design and verification teams. It says that we continuously build, test, and integrate our code and environments, increasing the frequency of integrating and testing from periodic to continuous. This helps to reduce the amount of broken code that gets checked-in and can be thought of as an 'trunk is always-working' model. The trick is how to decide what tests from the entirety of the testing environment should be selected for CI testing to give sufficient coverage that the code is essentially working, leaving the deeper ongoing exploratory testing to the main regression and soak testing environments.

There are opensource (e.g. Jenkins[19]) and commercially supported CI systems available. CI is often a component of more comprehensive CICD workflows such as Gitlab[20], or Bitbucket[21].

[19] https://jenkins.io

[20] https://about.gitlab.com

[21] https://bitbucket.org/product/

5.4.2 Automated Code Reviewing (Pair-programming)

Many software development teams are adopting the Agile concept of pair-programming[22] practices and automating this process using one of the available code review solutions such as Gerrit[23] or Gitlab. Gerrit is a GIT[24] server that provides code review and access controls on the GIT repository. With Gerrit, when a developer makes a change, it is sent to this store of pending changes, where other developers can review, discuss and approve the change. After enough reviewers grant their approval, the change becomes an official part of the codebase. In addition to this store of pending changes, Gerrit captures notes and comments about each change. These features allow developers to review changes at their convenience, or when conversations about a change can't happen face to face. They also help to create a record of the conversation around a given change, which can provide a history of when a change was made and why.

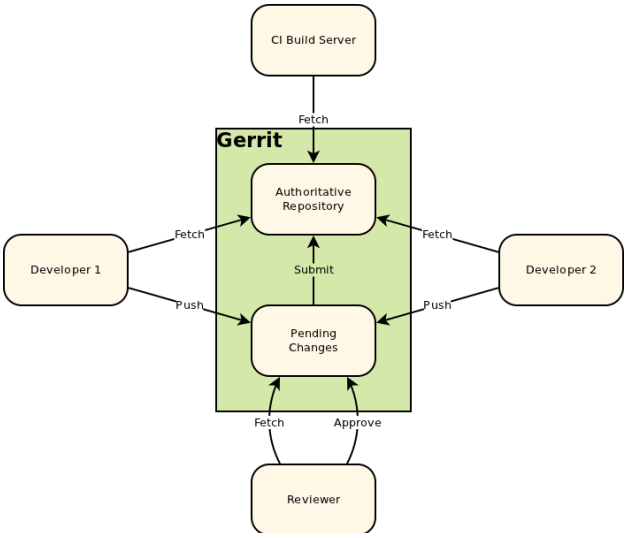

Figure 4 Gerrit workflow[25]

This workflow enforces a much higher level of scrutiny and ensures that knowledge about the code and how it works is shared with at least one other person, preferably more. It also ensures that all coders are aligning to a common coding standard and quality by enforcing this cross-checking. Of course, there is an overhead to this, in the additional time and rigor required to commit code, and this may be in conflict with the need for expediency and delivery pressure. At the end of the day, it's a trade-off between more time taken on high quality coding, against potential time lost and delay from complex debugging and bug-fixing later on, and ultimately on final product quality.

5.4.3 Code Refactoring and Technical Debt

Technical debt[26] (also known as design debt or code debt) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.

Technical debt can also easily accrue in hardware development, as quick fixes or sub-optimal coding may be applied to make fast progress towards critical project milestones, with a view that this can be cleaned up at a later date. Oftentimes it is done knowingly with a "revisit" comment in the code that can be parsed for with scripting later.

I've taken the following good summary of refactoring from Wikipedia[27]:-

"Code refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behavior. Refactoring is intended to improve nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity; these can improve source-code maintainability and create a more expressive internal architecture or object model to improve extensibility.

If done well, code refactoring may help software developers discover and fix hidden or dormant bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity. If done poorly it may fail the requirement that external functionality not be changed, introduce new bugs, or both."

Refactoring is just as applicable to hardware development as software development and is a best practice that design teams should adopt, despite this sometimes being in conflict with schedule pressures, since it may pay off significantly in the longer term, due to higher quality and better readability and maintainability of the codebase. Refactoring needs to be planned into the development schedule, whether waterfall or agile. According to Ref [1], software development teams are recommended to plan in at least 20% of time for code refactoring. There is no reason why this guideline would not be fully applicable to hardware code development.

[22] https://en.wikipedia.org/wiki/Pair_programming

[23] https://www.gerritcodereview.com/

[24] https://git-scm.com

[25] https://www.gerritcodereview.com/intro-how-gerrit-works.html

[26] https://en.wikipedia.org/wiki/Technical_debt

[27] https://en.wikipedia.org/wiki/Code_refactoring

6 How data and analytics can help?

6.1 Coverage Analytics

Code Coverage and Functional Coverage give indications of progress for verification efforts. Both are required and both need to be tracked with regular analysis and review of coverage gaps, leading to ongoing refinements in the verification environment. Branch and line code coverage simply tells us that lines of Verilog have been visited during testing, whereas expression or FSM coverage can give more detailed insights into how expressions have been triggered. This usually requires some manual review to understand what is reachable (and therefore should be hit) versus what is unreachable and will never be hit. In this sense there is a notion of completeness about code coverage in that every line of RTL code is there for a purpose, and should be executed at least once, unless the RTL code is in fact redundant. If reviewing confirms that this is redundant code, then it should be removed, as it has no purpose. Functional coverage does not have the same notion of completeness. Full coverage of the implemented coverage points can be achieved, but how do we know that our functional coverage model is complete? We don't! It's a best effort based on a process of brainstorming, reviewing, feedback and iterative refinement.

However, coverage is typically the set of metrics that gets the most attention in terms of assessing verification progress and signing off completion.

Finally, it is most important to remember the following: -

`Covered != Verified`

Sadly, meeting coverage goals alone does not guarantee an absence of bugs.

6.2 Bug Analytics

Bug tracking and analytics is a good way to understand verification progress. It requires some rigor in the consistent capture of bug data, which is sometimes not the highest priority when making a rapid fix and making progress is imperative. But bug data is a rich measure of the state of the design and the value of the verification work that is being done. It is important to be aware that a lack of finding bugs with the current environment, does not necessarily indicate victory, even though coverage targets may have been met. It may be simply that the current verification environment has saturated, it is no longer capable of finding further bugs, and we do not know if there remain unexplored sequences where further bugs may lurk. So, saturation is really a checkpoint where the engineering team need to scrub the verification environment to consider if it can be further extended. At the end of the day, engineering experience and judgement tell us if all conceivable cases are being explored or not, and if some are not, what the risk or likelihood is that critical bugs are being missed.

This plateauing of the cumulative bug discovery curve over time gives a visual indication of these verification saturation points.

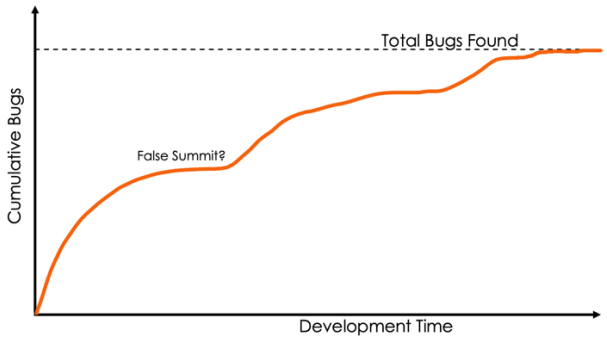

Figure 5 Progression of the cumulative bug curve

Note that the cumulative bug curve might not be a smooth ideal progression. There may several false saturation points on the way where the curve appears to be flattening off, but then further changes to the RTL codebase, or transitions to other verification environments might trigger further phases of active bug discovery.

Better still is to correlate the verification effort (as cycles, or CPU hours for example), with the bug discovery rate. This will help us to identify situations where bugs are no longer being discovered despite significant ongoing verification effort such as continued soak testing. Further to this, if we correlate source-code commits for both the RTL and the verification environment, we can reason about why we might be running (and consuming resources) when bugs are not being found, and the RTL and verification code is static. We are in a "saturation zone" where further verification effort is no longer yielding bugs, and we have to decide when to stop. In a world of constrained-random testing methodologies, we can continue to run marginally different cycles infinitely. This poses some questions for the hardware development team.

1. What is the magic target for bug-free and change-free verification that we are happy to sign-off against when we achieve it?
2. If a bug is subsequently discovered deep into this saturation zone, will we need to revise the target (and if so by how much?) and reset the release testing clock?

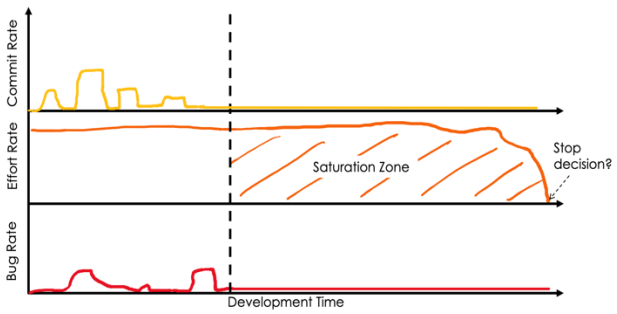

Figure 6 Bug discovery correlated with effort and code change

6.3 Bug Prediction

How great would it be if we knew precisely how many bugs our design contains and can measure our bug discovery progress against this?

Imagine: "I've got 23 bugs left to find and when I have found them, we are all done!".

What would an imaginary bug prediction curve look like and how would we reason about any gaps between predicted and actual?

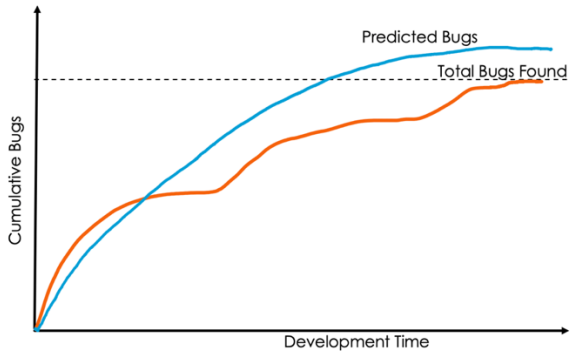

Figure 7 Imaginary Bug Prediction Curve

However, there have been some efforts, perhaps not that widely known, to make sensible predictions of how many bugs a design will contain. Again, these studies are typically within the software development domain. Some ideas[28] have been presented several years ago that show good results for hardware bug curve predictions that use the Rayleigh Distribution Model. Can prediction models help us to at least approximate our bug projections to give some guidance on how we are performing in a data-driven way? I'm not saying there are perfect solutions here, but it may warrant some further research, especially in a world where Machine Learning is proliferating into all sorts of complex prediction problems. Given a clean bugs-dataset for a comparable design, can we experiment with ML algorithms (such as Decision Trees, Naïve Bayes, Artificial Neural Networks (ANNs) for example) and some basic feature engineering to determine which bug and design metrics are most important and give the best prediction results? I'm not recommending a specific solution here, as that would be a whole whitepaper in itself, but it seems like a rich area for data science. The success or failure of this will be very dependent on how much data is being consistently collected for bugs. A good bug schema is needed that will ensure data collection goes beyond simply describing the bug itself. Bug classification details, impact and root causes will be needed. Also, data related to code churn, code size, complexity measures etc. will be important factors. A quick web search will reveal several research studies of this, again mostly in the software development domain. But there is no reason why these techniques could not map to hardware development with a little effort.

6.4 Codebase Analytics

There is a rich dataset that is less often analyzed for RTL design projects and this is the revision control system data, GIT for example. Version control practices can vary from team to team but if used in a consistent way by the development team, the GIT repository can give insights into the health of our codebase, e.g. whereabouts are the problem areas in the design or the verification environment. This would be indicated either by size and/or complexity of the code or commit rates that indicate code churn hotspots. Understanding where those hotspots are might be beneficial when considering where to focus verification efforts or when to consider any code refactoring. This is another example of where methods used by software developers could be used for hardware developers.

There is an excellent book that covers this topic, Ref [2], with supporting website and analytics tooling available. The title points to the use of forensic techniques to understand defects and bad design in programs. The Author (Adam Tornhill) introduces the idea of 'hotspots' that represent complex parts of the code base that have changed quickly because frequent changes to complex code usually indicate declining quality. The richness of the GIT data is that it contains the evolution of the codebase over time. In the book and the website, he shows some powerful interactive visualizations of hot-spots, that make it easy to identify areas of concern and analyze how they change over time. A complexity metric is required to do this (recall the discussion on complexity earlier), and for software developers there are several options such as simply counting lines of code (LOCs - comments and blank lines), or McCabe cyclomatic complexity.

[28] https://www.testandverification.com/DVClub/24_Jan_2011/Greg_Smith.pdf

However, Tornhill recommends the use of a simpler code indentation metric which works well when source code is properly structured and in general the level of indents is a good indicator of how the code is constructed and the number of decision levels that exist. For Verilog or VHDL, this complexity metric may not be appropriate however, especially given the variation between simple structural modules, that are mostly wiring for example, versus more complex modules that might encode algorithms or state machines, and also behavioural coding styles versus instantiated gates coding styles. Nonetheless, simple LOC measures correlated with commit data is still likely to be a highly useful indicator of areas of concern. If you have a viable complexity metric you can then track and visualise how that complexity changes over time, identifying situations where complexity is growing to a level of concern, that might suggest some code refactoring is needed, and then how that looks after the refactoring. Further, it may be possible to identify unknown linkages or couplings in the codebase. Commits that tend to occur in groups where the coupling between files is not necessarily obvious is giving us further insights into how the codebase is structured that will help us to understand where bugs and other defects might occur.

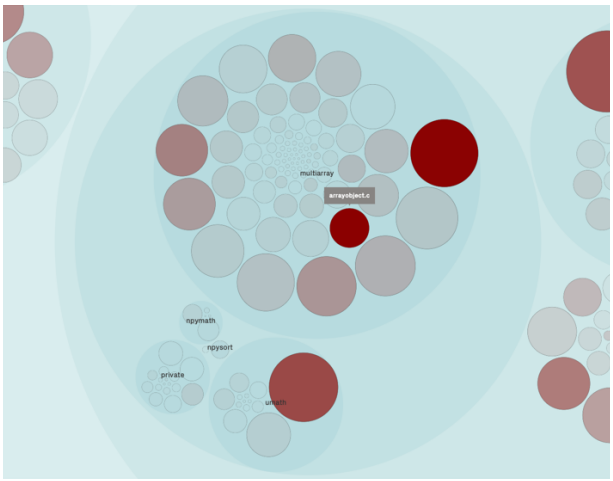

Figure 8 Screenshot of Hotspot visualization[29] taken from codescene

The above screenshot is taken from Tornhill's website[30] and shows an interactive visualisation of hotspot analysis. This view shows the relative complexity of each code module (in this case C) by size, and the commit rate by colour – the darker cells indicating highest commit rate. The viz can also be flipped to show code age, defects and refactoring targets. The darker cells above are clearly the obvious targets to consider for refactoring.

Tornhill goes on to examine other aspects of the GIT data that can be of interest and relevance to hardware development. He refers to social aspects of code. For example, the GIT repository contains interesting data on who has made commits, this being the development team in general. It's useful to understand what code is owned by which developers, how this changed over time, or to identify 'abandoned code' where the main developer has left the team or the organisation. In this case the team need to consider how the code will be re-adopted and then if the code is in a state where it can be done so with confidence or needs to be recoded in order to be fully understood.

Tornhill's work also considers the codebase from an architecture and a project management point of view. See Tornhill's blog[31] for further details.

7 Conclusion

While hardware developers think differently about how they develop their product to software developers, there are many overlaps. After all, both are developing code to implement their products. Hardware developers being confined to Verilog, SystemVerilog or VHDL in the main, while software developers have many rich software languages to choose from these days. The stakes for both are equally high. A critical hardware bug might incur significant hardware re-work costs. If impactful bugs make it into the field, products may be degraded in function or performance due to impactful workarounds, or even recalled in extreme circumstances. Building silicon and building hardware products is an expensive business. However, developing and operating complex software platforms with vast numbers of users is equally expensive and damaging when the platform is unavailable or financial damage has been wrought by a critical security issue for example. The software development world has embraced the principles of Agile and DevOps to ensure that such systems can be operated with great reliability and can be updated and deployed to users quickly and silently, and rapidly rolled-back if things go wrong. Some of this learning from the software development world can be applied to the hardware development world (excluding rapid roll-back maybe!).

It is also important to understand the nature of bugs and the scenarios that can lead to them. What are the origins of bugs and what strategies can be applied to minimize them?

Data analytics is a useful toolbox that is available to us when dealing with complex hardware or software product developments. Getting to grips with the data so that it can be used effectively can be a significant effort. Datasets needs to be clean and complete. Good visualizations are incredibly powerful for telling the story of the data and communicating insights quickly. You should never

[29] https://codescene.io/projects/171/jobs/15343/results/code/hotspots/system-map

[30] Permission kindly granted by Adam Tornhill

[31] https://empear.com/blog/codescene-prioritize-technical-debt-in-react/

underestimate the power of the human brain for pattern recognition when data is presented in a visual form rather than rows and columns of numbers. Machine Learning is becoming a powerful tool for building useful predictions from that data and using this learning to reduce development efforts and increase productivity.

8 References

1. The DevOps Handbook: Kim, Humble, DeBois, Willis
2. Your Code as a Crime Scene: Adam Tornhill (https://www.adamtornhill.com/articles/crimescene/codeascrimescene.htm)