INTRODUCTION
This is an overview of best practices for FPGA or ASIC design, assuming a traditional waterfall development process. There are four development phases: PLAN, EXECUTE, VERIFY and SUPPORT. A review step is recommended between each phase, as prescribed by DO-254. These concepts can be used in alternative methodologies, like Agile.

This article is divided into descriptions of each phase shown in Figure 1 Best Practices Flow. Each phase is broken up into Overview, Techniques and Deliverables sections. Information in the Overview is always specific to each phase. Information in the other sections may apply to other phases but will not be repeated in each phase. This article will only introduce a topic where it is first used. Subsequent phases will provide a reference to the section where the topic is introduced. For example, the PLAN phase Techniques section calls out issue tracking. A project should track issues throughout the whole development process and not just in the PLAN phase. This document only introduces issue tracking in the PLAN phase. Subsequent phases will provide a reference to the PLAN phase where the topic is introduced.

Many of the techniques discussed in this paper are detailed on the Verification Academy website, https://verificationacademy.com/. This article is meant as a quick overview of the recommended techniques.

PLAN PHASE
Overview
Most designs start with system requirements. System requirements are decomposed to lower levels of the design, i.e. sub-systems, such as an FPGA. Requirements decomposition can be done by Systems Engineering or the functional design team responsible for the sub-system level design. Legacy requirements can often be leveraged for the new design to minimize the time spent in this phase. Good, not perfect, requirements are vital to the success of any project. This is often a balancing act to find what is “good enough” to move the design forward. It is a mistake to start sub-
system level designs without requirements and generally leads to conflict between functional teams and longer overall design cycles.

Language selection is a key milestone of this phase. VHDL or Verilog are for design and SystemVerilog for verification is typical among Mentor’s customer base. Verification and design code re-use is also decided at this phase. Leveraging legacy requirements will help identify code for re-use. During this phase, selecting third party IP can help complete this design faster. This will also open the door to using standard verification IP to streamline the verification activity.

The FPGA requirements provide the foundation for decisions on how the design will be partitioned and how it is tested. This is when the decision is made to use constrained random and UVM techniques. While all designs can benefit from these techniques, sometimes a big formal structure is not worth it. In all cases a Test Plan should be documented and with more advanced techniques a Verification Architecture Document (VAD) should be developed. As a rule of thumb in gauging simulation effort, industry studies find that every month spent in simulation, produces a savings of two to three months of lab testing.

**Techniques**

**System Engineering**
Managing requirements can be difficult. There are tools in the market that can make sure you have clear requirements as well as manage them throughout the project.

Hardware Architecture – Proper partitioning of the design can optimize design re-use, complexity, power, quality, reliability, etc.

**Coding Guidelines**
This makes sure that code looks somewhat uniform especially if multiple designers are involved.

**Issue Tracking**
There is a tradeoff between the overhead of issue tracking and the value gained for issue tracking. Some teams wait until the product is out in the field to manage bugs and issues. Tracking from the beginning of the project provides visibility and insight into decisions made throughout the development cycle. As a rule of thumb, issue tracking should be used as soon as the design asset is delivered to other team members.

**Revision Control**
Tools such as CVS, SVN or GIT provide revision control of the design assets. This is vital to manage design configurations being used by other team members. As a rule of thumb, revision control should be used as soon as the design asset is delivered to other team members.

**Deliverables**
There are three deliverables in this phase that become inputs to the next phase. These tend to be fluid documents and will change throughout all the phases as the design matures.

**Design Description Document (D3)**
This should have the structure of your design. There can be multiple D3s for a design. The over arching D3 details the FPGA at a top level. Furthermore, the design should be broken up into manageable design units with the over arching document that links them all together. At a minimum, each of these documents diagram state machines, IO, pipelines and registers. It should include any third party IP and what configurations are entailed with using them.

**Test Plan**
Often this is a list of requirements and the plan to test them. There are details in the D3s which are not necessarily requirements, but will possible need testing. The person who is verifying this design should make sure these details are documented in the Test Plan as well.

**Verification Architecture Document (VAD)**
This is for advanced verification methodologies. This is basically the D3 for the verification environment. It details what components are needed to be created, what can be used from another project and how they interconnect. It also gives a day in the life of (DITL) for the Device Under Test (DUT). This also documents any verification IP that would be required. A template of what this looks like is available on the Verification Academy website or upon request.
EXECUTE PHASE

Overview
This is the coding phase of the design. From the D3, test plan and VAD, both the designs and the testing environment are created. This should be the shortest phase if the PLAN phase was as complete as possible.

The EXECUTE and the VERIFY phases at times blur together. But there should still be a review between the phases the first time you move from one to the next and periodically between the two.

Techniques

VHDL & Verilog for RTL Design
These are the two main languages used to develop RTL. VHDL is more structured but more involved in writing and Verilog is less structured and can cause unforeseen behavior in the code.

SystemVerilog and UVM for Verification
A more complete solution to creating a testbench; SystemVerilog gives the tools to create the environment and UVM provides a structure to use when developing a test bench. It makes the code more unified, understandable and re-usable.

Assertion Based Verification (ABV)
Product Specification Language (PSL) & SystemVerilog Assertions (SVA) are the two languages utilized for ABV. When an assert fires, it’s pointing at a potential problem in the design. Using asserts quickens debug time and can potentially point at problems that the environment would miss. Both the verification engineer and the design engineer (to assure the correctness of the design) should write these.

Constrained Random Verification
Creates meaningful random simulation stimulus for a design. Minimizes the need to stress the design through directed test patterns. It also minimizes the effort to create test patterns and reduces the size and effort of managing the test suite.

UVM
Universal Verification Methodology (UVM) is an Accellera standard with support from multiple vendors. It is a standardized methodology for verifying integrated circuit designs. The UVM class library provides automation of the SystemVerilog language, i.e sequences and data automation features, etc.

Coding Guidelines
Guidelines adopted to enforce coding uniformities within a project or across the entire organization. Promotes design quality and re-use.

Issue Tracking
See the PLAN phase, Techniques for details of Issue Tracking.

Revision Control
See the PLAN phase, Techniques for details of Revision Control.

Deliverables
At a minimum there are two deliverables from this phase, the RTL design and the test bench. Often additional deliverables are necessary to support the design and debugging in the lab. For example, there could be micro code that goes into a processor that’s attached to or embedded inside the FPGA.

RTL Design
Depending on what the final device is going to be, it could be RTL files or a burn file for the FPGA. If the deliverable is IP then delivery would be the design files for another iteration through the EXECUTE phase (this is not shown in the diagram).

Test Bench
This is the SystemVerilog code necessary to produce the test bench for the asset being delivered. If you are just developing IP then you would deliver the test bench assets for the IP for integration into the overall design test bench. Test bench IP assets should be added to the re-use library.

VERIFY PHASE

Overview
Spending the proper amount of time in this phase, to establish the design’s correctness, is the foundation of these best practices. Many teams want to rush into the lab because software wants something that they can use to debug code. While this is important, staying in the lab for hardware verification often delays the final hardware release up to 3 times longer than necessary. The best
practices flow has the VERIFY phase divided into two distinct stages. The first phase is Functional Verify followed by the Lab Verify phase. This feature provides leverage in a couple ways. First, using simulation as the primary form of debug provides a much shorter (Agile/Scrum) loop to fix the inevitable shortcomings of the upstream deliverables – which manifest themselves in simulation problems. Finding these problems in simulation rather than the lab has shown, through repeated industry benchmarks, a verification reduction of up to 3x. Second, this flow allows the project to schedule staged hardware builds for the software organization. This allows the hardware design team to verify specific features via simulation before delivering hardware to the lab.

The reason for this simulation centric flow is twofold. First, it is much easier to debug a design in simulation than the lab. Second, it reduces the overall churn because software is testing on verified hardware and hardware can focus on hardware verification rather than responding to problems found during software testing.

The question that is always asked is how does a project know it’s done with simulation? The most common metrics are Code Coverage and Functional Coverage. Code Coverage tells whether a line of code is executed or not and is a built-in feature of simulators. Functional Coverage uses user-defined points in the test bench code that are mapped to either requirements or design details from the D3 instead of lines of code. In conjunction with the two metrics above, many companies use inflection in the issue-tracking curve to determine how robust the design is. Use this metric with caution because it’s tied with how sophisticated the verification environment is.

A thorough verification environment and test suite provides significant value to the process. Most value comes from the ability to run regressions and make sure a bug fix or the addition of a design feature doesn’t break anything else in the design. Pivotal to this capability is the use of self-checking methods.

Of course lab work is always necessary in the design process. However, putting the bulk of the effort into simulation makes the lab work faster and more efficient. Industry has found that for every month spent in simulation, you save about two to three months of testing in the lab. Also, with a sophisticated simulation environment if a bug is found in the lab, it’s much easier to replicate in the simulation environment, which leads to faster debug times.

When longer hardware lead times are expected, e.g. found with an ASIC development, other methods are required to provide early software/firmware development and debug. System level modeling in conjunction with hardware emulation lets software development and test begin much earlier in the development cycle – yielding a significant development cycle reduction for the entire system.

**Techniques**

**Code Coverage**
This capability is built into all the Questa™ simulators. While it is easy to turn on, it still requires analysis to determine why sections of code are not being executed or if they are even reachable. Holes that aren’t being tested will require directed test cases or modification of existing test cases or exclusions.

**Functional Coverage**
This is a user defined metric. It shows how things interact in a way that code coverage does not.

**Assertions**
Add more of these as bugs are found to make it easier to debug.

**Formal Property Checking**
This is recommended for DO-254. This makes sure the design does what it’s supposed to do and no more than that.

**Formal Equivalence Checking**
This is a requirement for DO-254. You must have the ability to make sure that throughout the process of transforming code to gates, it remains the same.

**Regression Runs**
This is the process of checking to see if anything failed when changes are made to a baselined design. Any change to the baseline has the potential for unforeseen downstream effects.

**Deliverables**

**Baseline Design**
A version of the design that is working properly based on a specific configuration of design requirements and
design assets. Using design baselines allows downstream development to begin based on a specific configuration while upstream design continues toward completion.

**SUPPORT PHASE**

**Overview**

This phase is required for maintaining design configurations for interim baseline releases throughout a project, maintaining libraries of reusable design assets for use within the current project and derived projects with compatible requirements and features, and maintaining design configurations that can be leveraged as a starting point for future designs.

Configuration controls are vital in this phase. Maintaining revision linkage between requirements, design documents, test documents, the design code and test bench code is necessary to assure quality releases of these design assets as a coherent package for project, design and verification re-use. Implied in this statement is the need to keep documentation up to date as requirements change throughout the design process.

**Techniques**

**Configuration Management**

Configuration management is a process for maintaining consistency of a product’s requirements with its design assets, performance, functional and physical attributes. A strong systems engineering discipline should be leveraged when setting up configuration management practices.

**Deliverables**

**Project Baseline Configurations**

A project baseline configuration is a snapshot in time containing all the design assets necessary for handoff to other functional groups that will use the assets to continue their design process. The final design deliverable is an end case of a baseline configuration where all the assets that compose the design are complete and ready for product release.

**Re-Use Library**

The re-use library will contain design building blocks that can be re-used in multiple locations within a given project or re-used on future projects. These building blocks are in reality a package of all the parts that compose a particular design asset, i.e. requirements, design documents, design code, test bench documentation, testbench code, test results, revision history, bug history, etc. Being able to leverage pieces of a design that is already completed and verified is key to speeding up schedules down the road.

**CONCLUSION**

At times, people don’t know what they don’t know, making it easy to try to get right to the coding phase of the design process without creating a plan on what to do. Or if they have a plan, they write it and don’t review it from time to time since they aren’t in the planning phase anymore. When things were a single one-designer-does-it-all, not all these steps were taken simply because there wasn’t anyone else involved in the process. That isn’t the case anymore, from either help with board design or software people writing code to be executed on your block, no one designs in a vacuum anymore.

Bruce Lee once said, “Adapt what is useful, reject what is useless, and add what is specifically your own.” I recommend you take what you can from this article and take some of the thoughts into your own design process.
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

20 Video Courses Available Covering
- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Basic and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- Power Aware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 4400 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com