How to attack your chip from the top? Why is it so difficult to put together a good top level verification plan? Here are a few ideas.

Top Level Verification is fundamentally different from Module Level Verification. As long as you''re in the standalone module realm life is quite good - in most cases you''ll get a decent specification document, the functionality will typically be focused around a limited number of elements, and you''ll get the hang of it quite quickly. Also, what needs to be verified is quite self-explanatory when you verify a standalone module and usually you''ll have a limited number of external interfaces which makes life a lot easier. But how do you go about verifying a system that has many modules, each of which has been verified as a standalone module? What are the things you should verify looking at your chip from the top?

Common verification methodologies have been focusing on making your module level verification components reusable for the top level verification, but not much focus has been put on the principles of verifying a sub-system or an entire chip. Obviously you don''t want to do the same work twice - that means you don''t want to run the entire module regression suite again on the top level DUT. Sometimes it''s almost impossible to reach all internal states of the top level DUT just by driving signals on the boundaries.

So while putting together a good verification plan for a standalone module is relatively easy, compiling an efficient yet thorough top level verification plan requires a different approach and is more like an art. So once again, as always with verification, experience plays a key role. Let''s go over the fundamentals of building a good top level verification plan.

The first thing you want to verify is that all modules have been *properly integrated* into the system. You want to make sure that all the interconnections are correct and no functional wires have been left unconnected. Linting tools could do the job of finding unconnected ports (you can also do this visually, if you dare) but in terms of verification plan the idea is to move chunks of data across the system, covering all functional blocks along the way. Our assumption here is that all wrong interconnections will show up during the process, taking the form of data integrity errors or similar errors. One particular approach is the
"Verification Flows"
approach which is a very efficient way to verify a System-on-Chip, but may be applied to other types of designs too. The "Flows" approach makes use of the
superposition
principle - your DUT (top level, remember?) is considered as a set of various independent

end-to-end data paths. Their sum gives you the overall functionality so you''re doing ok there, and the big advantage is that each flow can be verified independently, essentially reducing the number of modules (and verification environments) being simulated at the same time.

Yet, in many applications it''s the "all modules playing at the same time" feature that you''ll want to focus on right after integration checks. This phase is sometimes referred to as *Concurrency Tests* . Clearly the "Verification Flows" approach would be inappropriate here and what you''ll want to do is actually turn on all your drivers/generators and attack the poor chip on all fronts. A good top level coverage plan comes to mind here, as this phase really takes the whole Random/Coverage Driven methodology to the extreme. Typical bugs that will show up at this stage are related to system architecture, for example - a problem that occurs when several modules are accessing a shared resource.

Oh, If you have a processor on board, you''ll probably want to run a bunch of tests with the actual CPU in place and run some "software" tests. This is often referred to as *Co-Simulation* and it''s usually another thing you''ll have to cover at the top level. In fact, if you want to verify your device drivers in simulation today''s leading tools will offer you some great solutions. Call you local EDA vendor and ask about
HW/SW Co-Verification
(which is the next step after Co-Simulation).

Leaving software aside, let''s go back to our top level RTL verification. So what''s left to verify after we''ve covered integration and concurrency features? Well, you will need to verify the *reset* functionality of your device. You have to make sure that all modules behave nicely when the various reset conditions occur. And the same for different clock frequencies and ratios - top level simulations are the right place to play with different
clock settings
to make sure your chip works well under various clock conditions (only the supported ones, of course). Once again, random generation with a solid coverage model in your top level verification plan should get the job done.

What else? if your device has *multiple functions* multiplexed on its pins you might want to make sure that each individual function works well and that there is no interference. While you''re there, remember that pins, or more accurately *I/O pads* , sometimes have special functionality and therefore require functional verification too. Consult your chip architect for more details. A typical bug here could be a disconnected control wire that

activates an internal pull-up resistor, for instance.

The top level simulation is also a place where you can run *performance tests*, e.g. measure average bandwidth on selected interfaces. This is more difficult with heavy designs, but it could work.

Various *system level scenarios* may also find their way to the top level verification plan. These come in two flavors. The first is called *"acceptance tests"* and usually involves very basic scenarios such as "boot from ROM", "activate DMA", "load SW from external memory" and stuff like that. The intention here is to make sure that we've covered the most critical aspects of the chip''s functionality (or - "if we have a bug in any of these features, we''re doomed", if you will). The other flavor is all about *"corner cases"*. Here we want to make sure that our product is robust and can handle extreme situations such as buffer underflow or overflow, various error conditions, etc. Every system scenario should get its own entry in the top level verification plan.

Lastly, if we still have time, it would be nice to run a bunch of *sanity tests* (or even more than that) to test the internal modules at the top level. Even though this might sound redundant to some extent, it still gives us a final safety net before we reach the point of no return. After all, we''re all human beings - we might have missed a bug at the module level, or maybe we have a newer (modified) version of some internal block and we don''t even know it. Better safe than sorry, right?

And one last word of advice - writing a solid top level verification plan is not less an art than anything else. Take these guidelines into consideration and add on top of them what you and your colleagues have learned from previous experience. Remember, a solid verification plan is the key to your success.