on Debug and coloring the simulator

Or my new pet project

The bottom line

The code that goes into Asic or Fpga is synthesizable RTL.
For a long time we simulate mostly RTL (and derived GLV). For historic reasons, All current simulators more or less ignore this important fact. In current simulators, when change of signal's value occurs, only limited amount of info is passed to the destination rtl, namely new value and arrival time. Strength - (weak, strong,..) has no significance in RTL. My idea is to add "metadata" to all changes, or what we call "color". Rest of this article is about what can the metadata include and why it is useful.

Reasoning

This is about debug hardships and how they relate to Verification. And one idea how alleviate this problem.

It is often heard that the cost of verification skyrockets. My feeling that it is the cost of debug that actually contributes significantly to that. When regression is run, nothing is more frustrating than several tests catching bugs. Fixing those causes new bugs to surface and so on the cycle continues. Verification attracts much attention and efforts to improve it. While the step-sibling "debug" is left behind. Not all verification industry standards i endorse (see my opinion on the ugly cousins - SV & UVM), but at least there is an awareness. Debug on the other hand is left basically with wave viewers. Yes, Verdi and Vtool may be nice but are not the answer.

DFT stands for design for test. DFV (design for verification) is heard much less, but it is as important as DFT for the success of silicon project. It means that architecture of the chip should make it is easy to verify it. Divide and conquer is one useful strategy. But it doesn't end there. DBG (design for debuggability) can also be a valuable partner. DBG is much more trickier, and architecture plays a significant role here too. Several chips were designed by team of two people (one of them is me). In this tiny team, whenever timing, debug or other hurdle arises, we could go and demand better solutions from the chip architects, which were still us. This is not common in a regular chip design companies. It is even worse when the designers have little say in system architecture.

Claim #1 : Simulators are stupid.

Simulators are stupid. Why should i say that? For a long time we write synthesizable RTL. The design is in synthesizable RTL (which is a subset of a full language). This creates rigid constraints on the code. Yet all modern simulators ignore this fact. All backend and frontend tools have this knowledge. In RTL it is forbidden for two outputs to drive a single input. Any synthesis tool wil fail. Any timing tool will fail. Only simulator is ok with that. More? Any other tool knows what is clock, what is reset (async) and what is supply (UPF?). Only simulator completely ignores it.

Making simulation / emulation twice as fast, only divides the harvest from each run. ***The more Sims you can kick-off the less you think about them...***

If we let simulation be aware of context, perhaps debug will be smoother.

Claim #2: Verilog standard encourages this ignorance

Any tool knows what are flip-flops, latches, ram modules, rom modules and combinatorial blobs.   Who is completely ignorant? You get it. Verilog standard, in effort to mimic VHDL (language used mainly for building stupidly expensive bombs that in a good scenario blow to pieces barren rocks and in bad case friendlies) goes on a path that deliberately obscures this knowledge.

You may say, that all tools "infer" flops and wires, based on always type (always_ff, always_comb) or on clock edge or on bunch of other silly syntax tricks.My take: **Barking on the wrong tree (or pissing thereof).** Much more robust and error free is specify explicitly the hardware.

Claim #3: If the language had clear notion of flops, supplies, latches, combies, clocks, resets, scans - it could have enhanced the debug experience.

Verification effort looks at the design as mostly black box. If verification needs to peek deep inside the design to verify it - it only means the architecture is shaky. When verification catches a bug, the effort becomes debugging. Here the design must be humanly comprehensible to facilitate the debug process.

Observation #1: CDC, RDC, LINT, SCAN* and UPF (add property checking)

This is a minimal list of aspects we have to cover to make our RTL code useful. For each one of them, several vendors provide different tools that demand different files. Creation of these files, running the tools, interpreting log-files is a tedious task. And usually for each new vendor the task begins practically from zero.
We should be able to embed all what we know about the design into one file - the RTL code. If we are able to add metadata to the code itself, specifying in a natural way and within the language itself (not after comments) different power domains, RTL or TestBench, who is clock or async reset, flops, latches, "control" signals vs "data-path" signals and more - we could change the situation and ask each vendor to create his own input files. In the way **They** like it and return the results in a unified way.
I know, for large outfits, where there is a special person per each special task it is less of a concern. For small teams this usually involves endless debates and discussions where to put the effort and when.
But even for large outfits, having "vertical" designers not deeply familiar with the whole system, it reduces effectiveness and consumes precious time.

- SCAN here relates only to to enable scan mode. Like excluded modules and SCAN ingress egress ports.

Debugging

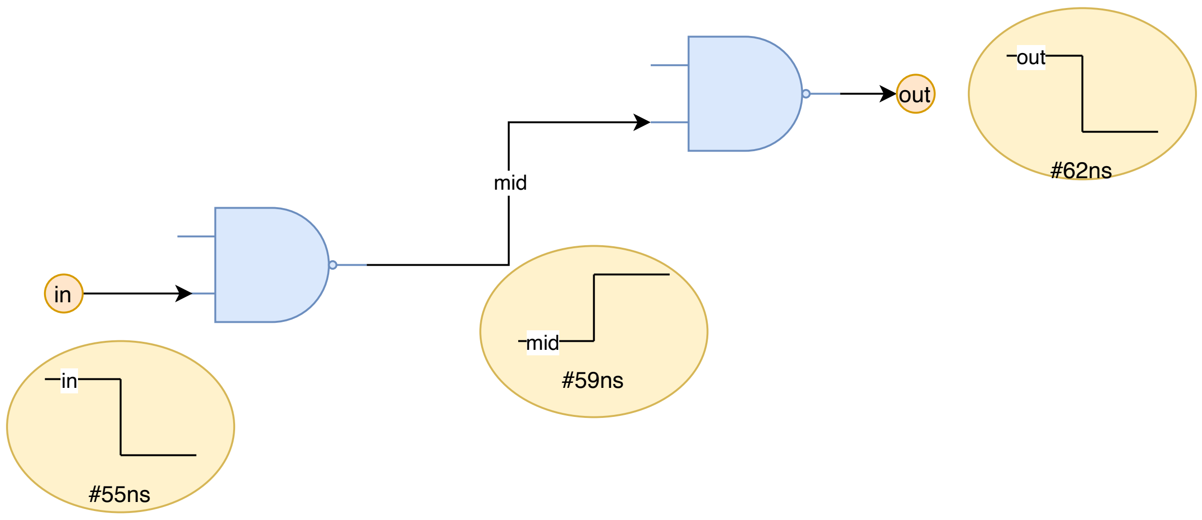
Simulators are getting faster. Partly because servers are faster, partly because emulation and parallel simulation.
But this is not a solution to debug - "Having CPU twice as fast, will lead only to running the infinite loops twice the amount".
Verilog Simulators ignore a lot knowledge that Synthesis and P&R have. This knowledge - who is clock, who is reset, who is flipflop or latch - used by all tools, except the simulator.
On the other hand, simulators produce internally data, that could help debugging, but vanishes inside them.
Some simulators will tell You about un-driven nets. Most wont. But in RTL, there is no reason to have un-driven net.
None of the simulators tell You about over-driven nets. Once again, in RTL, this is not acceptable.

Clock or reset becoming unknown in the middle of simulation (if the simulator is aware of who is clock and who isn't) is also catastrophic event.

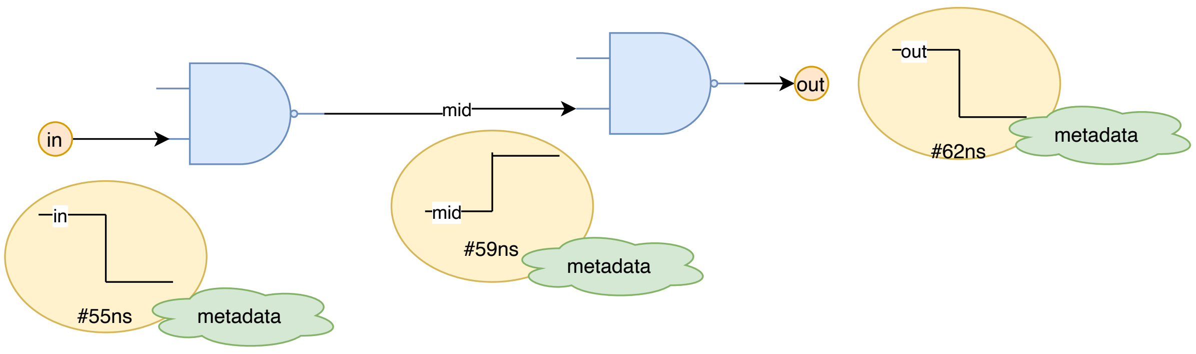
Idea

- Make a simulator that is aware of all knowledge we have (and should have) about RTL.
- Allow metadata about net (origins, clocking ,supply) value to be propogated.

  Regular simulator, on event of net value change, sends to the destination of that change, just the affected net, new value and time the change is relevant. It will not tell us, what created that change,



Is it a flop or combi logic, or in what hierarchy it was produced and more.
The **idea** is to add "**metadata**" to each change event, or what we call "**color**".
This data can include, but not limited to: clocking, supply, source info and individual colors. It will look something like this:



To keep the performance reasonable, It should be clear that we are talking about new simulator. Not upgrade of Xcellium/Vcs/Modelsim, but a completely new product.

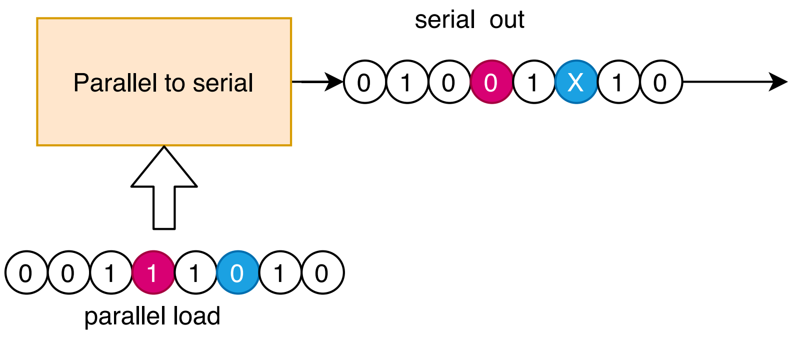
The metadata is independent of actual net values. It either stays the same or modified according to to metadata rules.

For current exploration] version, these ideas are piggy-bagged on regular verilog simulators.

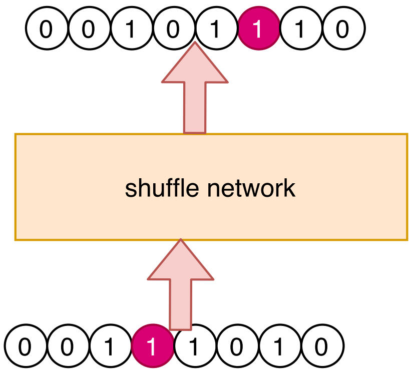
Where coloring might help

Basic example would be when net or flop becomes unknown are wrong, The simulator may present the knowledge about the support set and who caused the value to be wrong.

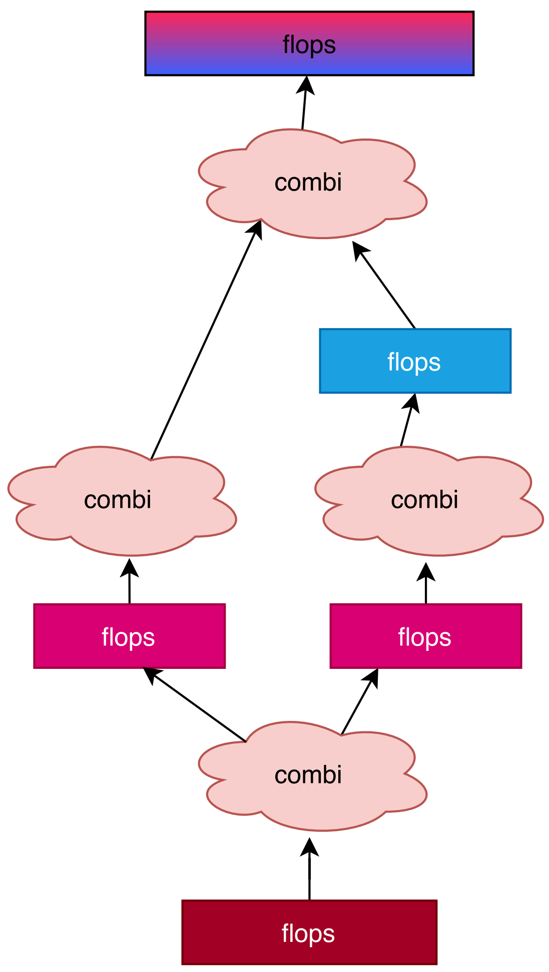
For example we have a parallel to serial converter design. I want to "color" certain serial bit and then observe, where it comes out on the parallel side. Couple of possible bugs are highlighted by changing the blue bit to unknown and red bit flipping value. Here "color" is a property of changing value that stays constant.



Or we have a configurable shuffle design. where input wide bus is shuffled to output wide bus. Debug entails checking what input bit goes to what output bit. Just marking each bit in different "color" makes this process trivial.



Or we have deep pipeline, where stages diverge and merge. Marking each input cycle data with different color, we may observe that no two different colors arrive at computing element at the same time.



Another example: Support set

Suppose Flop becomes X or just an unexpected value. Tracing the reasons for that are painful in RTL and even more so in GateLevel or absolute nightmare in SDF simulation. In my vision, metadata creates traceable support set of changes that led to this Flop. Metadata may include, source flops and when their last change happened. Ideally, simulator smarter part can produce intelligent reasons for the unfortunate Flop event.

Keep in mind that each net marked "clock" counts it's cycles, so each change of clocked flop is marked with that cycle and it arrives to the next flop, where it can be checked.

Think of SDF simulation. Just by marking clock edges in arriving data, will save hours of debug time. Re-convergent X propagation by counting inversions along the path.

For now, preprocessing of the code, produces tables (in python) of source nets affecting destination nets. Ideally it should be built-in into simulator. These tables are used by python-driven-simulation to mimic these functionalities.

Sideways note

Years ago, symbolic simulators were trying to enter the mainstream. There You mark some values as "symbols" - variables. A bit could become 0,1,'x' or "a". "a" is a variable. If it goes through inverter - the output becomes "!a".   Using formal techniques, these "symbols" were propagated as "boolean functions". The complexity was managed through careful introduction of only few "symbols".
For some reason this never caught on. By the way, if all nets get to be variables, it becomes formal verification of property checking. The advantage is that user can be anywhere on the spectrum - from regular logical simulation to partial formal to full formal - and thus adopt the runtime to the needs.

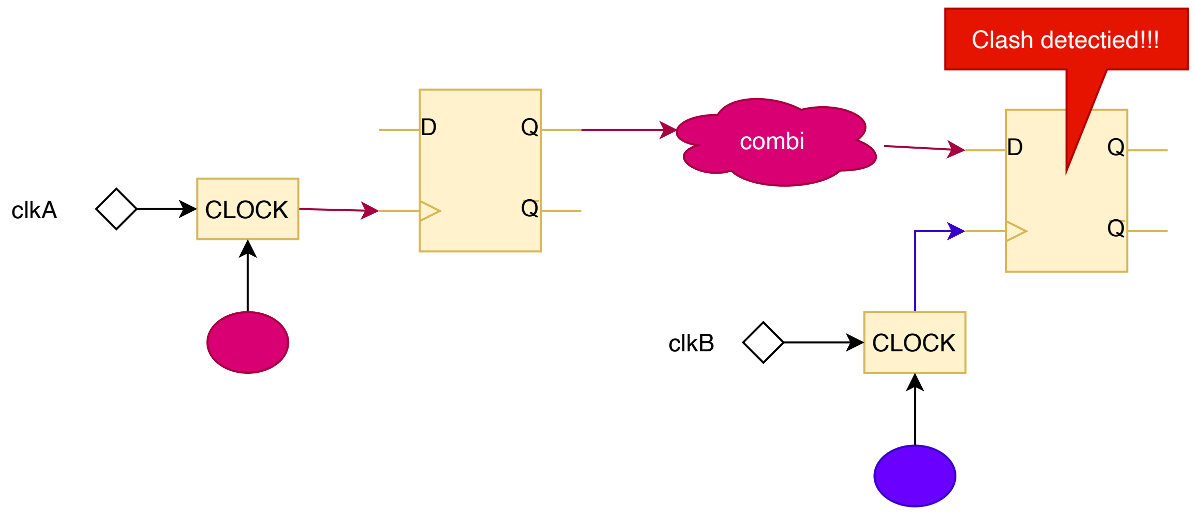
Some thoughts on updating Verilog RTL

Few upgrades can make verilog RTL much more powerful and suitable to simulation and backend. *It is not a showstopper not to do it, but will help.* For current trials i just add **(* pragmas *)** that allow to add any meta-data deemed useful.

1. insert formal primitives for: clock, supply and reset and more.

Special instances "color" the signal passing through them. it can be clock name, supply name, reset or generic color. Supply instance works differently. It colors all instances and outputs in the module. It is not connected to specific instance in module. If You have trouble with that, probably your UPF has problems too.

2. Rules of propogation of all of the above



Box labeled CLOCK colors (adds metadata) the regular net clkA to be a clock. FlipFlops pass clock input metadata to Q outputs. This "color" through combinatorial logic. Upon arrival at the next flop, colors of clock and data input are compared. Special nets, like clocks and resets also make the simulator sensitive to transitions, especially to unknown value. Simulator can also report what flops don't have colored clock inputs.

Similar checks can be applied on nets arriving from different modules and carriyng a different supply name. Or that supply is turned off. Or that supply has different voltage level.

Another check is that clock colored signal (original clock, not Q through flop) enter combi logic or D input of a flop. Same applies to async reset signals.

3. Allow coloring the net values with arbitrary colors and add rules to propogate those.

This enables debug of complex logic structures. Altough the rules are not strong enough.

4. allow marking code as RTL vs vs MODELS.

this will allow placing arbitrary code in non-design areas and prevent checkers to flag errors.

5. allow marking in RTL hierarchy the supplies, voltages and their status

Covers checks of isolation markers, creation of UPF files for other tools, Simulating supply changes natively and so on.

Some more points

The above points and ideas above are just the beginning. Every time i meet a debug problem, i can think of more powerful features to add. Example?

- Check correct synchronization (no combi cones before, no re-convergence)
- Create timing constraints and UPF directly from RTL
- Catch wrong .
- Catch path-throughs, long traveiling across hierarchies paths, in and out paths.
- Catch combi loops.

Dynamic simulation may seem an overkill for some of these examples, but static tools tend to be over pessimistic too. Simulator can be run in several levels of ignoring the meta-data to balance runtime over checking.

next chapter: on pilot implementation