JULY 9–13, 2023

MOSCONE WEST CENTER
SAN FRANCISCO, CA, USA

Shift Left Detection of Logic Equivalence Abort Points via RTL Linter

Motivation

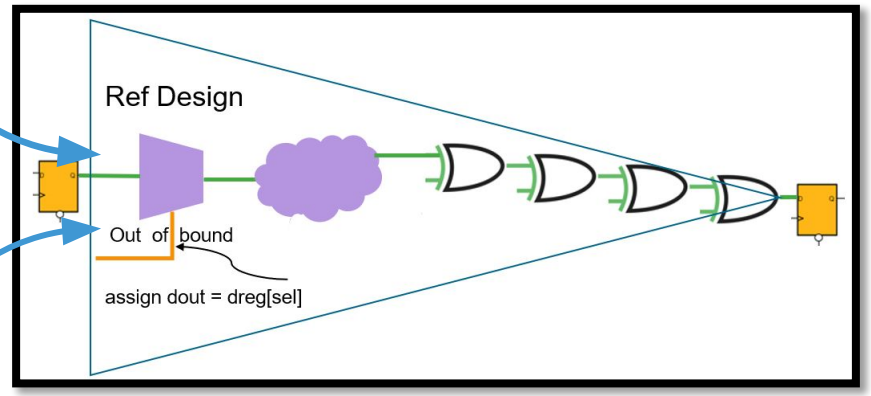
- Issues with logic equivalence aborts?
  - Equivalence checking is NP-complete problem, and it may happen that verification on some points may not converge and logic equivalence tools may abort with respect to certain end points
  - The primary reason for this is the lack of similarity between the RTL and the optimized Netlist
  - Resolving these hard points can be a challenge, and in some cases, re-synthesis by disabling some class of synthesis optimizations can help verification
  - Detecting hard points during verification can be too-late as RTL may be frozen at that design-stage
  - Customers are looking for “early” detection (during RTL Linting) of hard to verify points, thus creating a process where RTL-changes and synthesis constraints can be identified at linting stage to ensure reliable [synthesis + verification] results

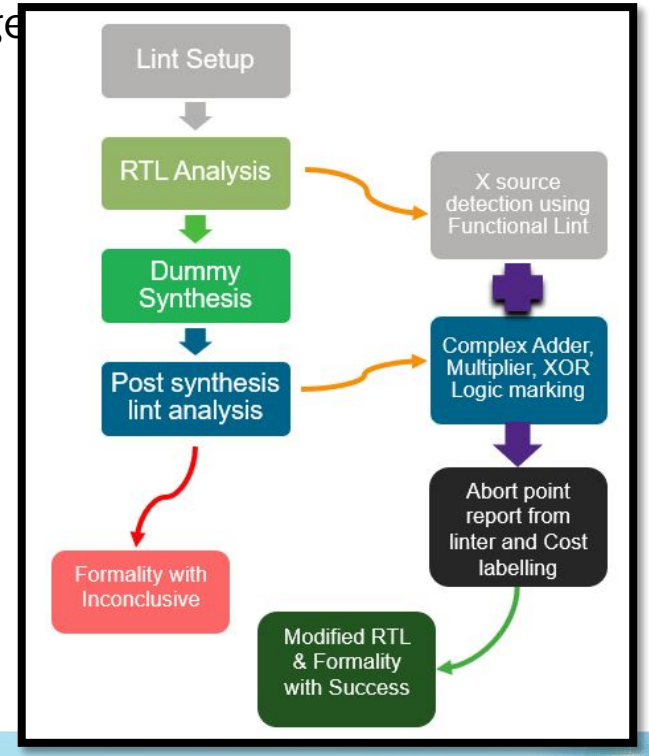
- Lack of synergy between front-end static linters, equivalence checkers and implementation tools
  - **Missing** analysis of complex data paths with don’t cares at an earlier stage. No handshake with implementation tools w.r.t synthesis QoR
  - Need to improve the regular lint flow **by smartly catching and reporting accurate aborts points** from logic equivalence early at the RTL level
  - **Need an innovative way to generate and pass** synthesis constraints that will prevent aggressive optimizations on certain cones

Main Idea –Detect Hard-To-Verify Abort Points

- Shift left approach to identity a sequence of RTL constructs that cause hard verification
  - RTL linting tool to identify RTL constructs that can provide an early-warning to designers about hard-verifications
  - Designers can take steps during the linting stage to re-code that piece of RTL or in some cases, setup synthesis with constraints that will prevent aggressive optimizations on such cones
  - [illegible]so matters. They need to be combined with multiple large

Out of bound indexing	input [3:0] idx; wire [10:0] A; wire Z1; assign Z1 = A[idx]; idx variable can attain a value which can cause out of bound indexing for A
Explicit X assign	assign b = sel ? a : 'bx;
X via casex statements	casex (A) 2'b?1: begin R = B[0]; end default: begin R = 'x; end endcase
Case statements not fully specified	case (A[1:0]) 2'b11: begin R = B[0]; end 2'b10: begin R = B[1]; end endcase R can be undefined when A attains other possible values





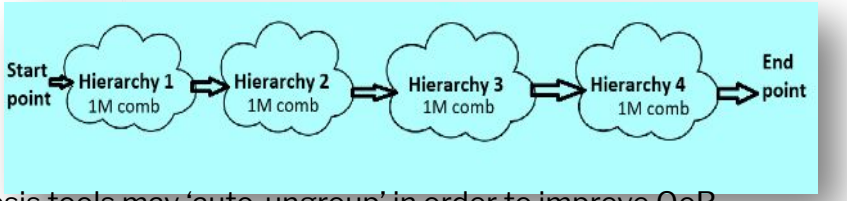
Guiding Synthesis Tool To Improve Equivalence Convergence

Generation of Synthesis Constraints

Standalone long XOR chains

- CRC logic have huge XOR-chains. These chains may get ungrouped and get combined with neighboring hierarchies containing additional XOR chains
- Once ungrouped, they may become hard to verify
- Identification of “long” XOR chains and providing guidance/restrictions from static to implementation tools to preserve these XOR hierarchies during synthesis

Huge no of combo gates in the cone of endpoint



- Synthesis tools may ‘auto-ungroup’ in order to improve QoR.
- With auto-ungrouping, equivalence checkers may face verification challenges due to huge no of ungrouped gates.
- This solution shall allow user to choose a threshold-value w.r.t ‘max no of ungrouped gates’ in the fan-in cone.
- Tool will analyze the fan-in cone and determine if ungrouping certain set of cascaded hierarchies can lead to huge no of ungrouped gates(>threshold limit).
- Incase tool detects the possibility of the same, it shall generate a constraint to guide the synthesis tool user to not ‘ungroup’ certain hierarchies

Attach custom weights and determine score

- Cost is attached to each logic gate. Higher cost means higher is the probability of getting stuck during equivalence checking
- Solution provides customizability and user can modify and attach custom weights to determine the cost of each path
- To indicate the size of an operator, specify the variable '$size' in the equation.
- To indicate the back-to-back count of the operators, specify the variable '$count' in the equation

Tag Name	Cell Type	Default Equation
MULT	Multiplier	'(log($size)/log(2))*3'
ADDER	Adder or Subtractor	'(log($size)/log(2))*2'
DIV	Division Operator	'(log($size)/log(2))*4'
MOD	Modules Operator	'(log($size)/log(2))*3'
XOR	XOR/XNOR	'(log($size)/log(2))*1'
COMP	Comparator	'(log($size)/log(2))*1'
MUX	Multiplexer	'(log($size)/log(2))*1'
BUF	Buffer	0
INV	Inverter	0
OTHER	Other Combinational Cells	'(log($size)/log(2))*1'

Evidence

- **TABLE 1** shows the comparison data where a static checker can warn upfront about the potential abort points in the equivalence checker. Matching %age is calculated after comparing names of the potential abort points reported by both the tools. There can be naming differences for which a deterministic script-based name transformation engine is written
- **TABLE 2** shows the performance improvement in the equivalence checker after RTL was modified based upon the suggestions from the static checker
- This is extremely promising, and a value add for the industry

Design	Equivalence checker abort/unverified points (total count)	Detection of potential faulty end points (total count)	Match %age
D1	1454	1380	~95%
D2	99	98	~99%

TABLE 1

Design	Original runtime (Formality)	Runtime after RTL modification
D1	~97 hours	Few mins
D2	~20 hours	~0.3 hour

TABLE 2

Summary

- Shift left approach to identify a sequence of RTL constructs that cause hard verification for logic equivalence checkers and getting more predictable sign-off on the netlist
- Handshake between static and the synthesis tools facilitating the verification flow to be left shifted
- Ability to attach custom weights and determine complexity score
- Detection of Hard-To-Verify Problems cuts down the logic equivalence time significantly
- Future Direction
  - Improve the runtime for X-sources detection using Formal engines
  - Introduce Machine Learning to detect hard-to-verify pattern in logic equivalence