RTL Transformation Methods to Achieve Order of Magnitude TAT Improvement in VLSI Design

Ashfaq Khan, Shuhui Lin, Dan Standring, Adam Cajiao Campos, Soowan Suh, Satish Venkatesan
Intel Folsom Campus, 1900 Prairie City Rd, Folsom, CA, USA
ashfaq.khan@intel.com; shuhui.lin@intel.com; daniel.m.standring@intel.com; adam.cajiao.campos@intel.com

***Abstract* - A typical VLSI implementation starts from a logical or architectural view of the design in RTL, where multiple building blocks are connected without considering physical constraints. This view then goes through a series of modifications required to meet structural design (SD) and manufacturability needs. While some of these modifications enjoy automation through industry standard EDA tools (e.g., DFT insertion), majority of them are currently implemented manually by designers and are considered regular design activity. To make matters worse, some of these steps often need to be repeated based on the feedback from the downstream consumers of the design. In this paper we demonstrate that, with the recent advancements of the EDA industry, many of these activities can now be automated in the form of RTL Transformation. Such transformations include, but are not limited to, Physical Hierarchy restructuring, Feedthrough/Tie-off/Repeater insertion, Clock/Reset re-distribution, Fuse isolation etc. Our methods combine user spec and home-grown code with industry standard EDA tools to achieve RTL transformations on the fly as part of the RTL generation flow. This is a paradigm shift in RTL development, resulting in order of magnitude improvement in design turn-around time (TAT). We describe several applications/transformations from a production use case where multiple weeks of design work have been either fully eliminated or reduced to a fraction of the original TAT, depending on the type of the transformation. We also present various best-known methods (BKMs) and gotchas that need to be considered while performing RTL transformations to achieve the best results, given the current state of the EDA tools.**

I. Introduction

An RTL design goes through various updates before an architectural view of it can be ready for consumption for structural design processing, where it is synthesized into a netlist, placed/routed, and combined with other collaterals to form the final GDS database for manufacturing. These updates include Hierarchy restructuring, Feedthrough/Tie-off/Repeater insertion, Clock/Reset re-distribution, Fuse isolation etc., and are typically considered as regular design activity. Some of these activities are often repeated due to incremental feedback from the downstream consumers of the design and contribute to several weeks of TAT in the entire design cycle [1, 2].

In this paper, we demonstrate that with the latest advancements in the EDA tools, **it is now possible to automate these activities in the form of RTL transformation**. Figure 1a shows the key concept - how the user spec and home-grown high-level APIs are used in conjunction with a vendor tool and its low-level APIs to perform effective RTL transformations automatically. Figure 1b shows how RTL transformation fits in the overall design cycle. We start with an architectural view of the RTL, which goes through various transformations until it's ready to be handed off for SD activities. SD team can perform additional transformations without any coordination with the RTL team to generate the final RTL view that goes through the rest of the design process to produce GDS for manufacturing.

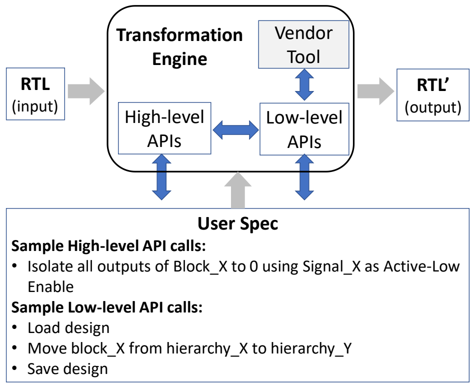

Transformation Engine
Vendor Tool
RTL (input)
RTL' (output)
High-level APIs
Low-level APIs
User Spec
Sample High-level API calls:
• Isolate all outputs of Block_X to 0 using Signal_X as Active-Low Enable
Sample Low-level API calls:
• Load design
• Move block_X from hierarchy_X to hierarchy_Y
• Save design

Figure 1a: Spec-based automated RTL Transformation

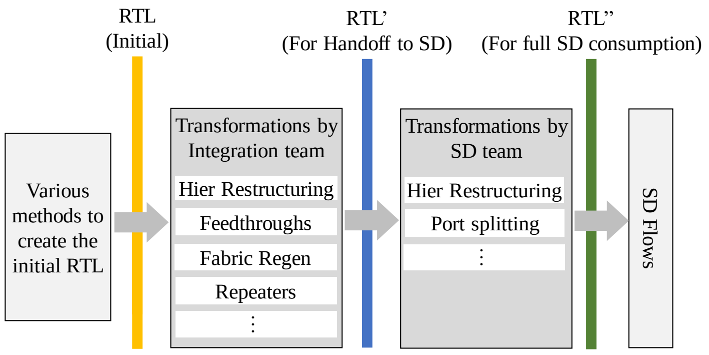

RTL (Initial)
RTL' (For Handoff to SD)
RTL'' (For full SD consumption)
Various methods to create the initial RTL
Transformations by Integration team
Hier Restructuring
Feedthroughs
Fabric Regen
Repeaters
⋮
Transformations by SD team
Hier Restructuring
Port splitting
⋮
SD Flows

Figure 1b: RTL Transformation in the overall design cycle (Automated, correct-by-construction, performed in sequence)

II. AUTOMATED RTL TRANSFORMATION

A. Upfront considerations

Repetitive work with simple spec: Any update to RTL that could be defined in a machine-readable spec is a candidate for automated RTL transformation. However, the best choices are generally those where such a spec is significantly simpler to write than to manually update the RTL itself, and where the updates may need to be applied more than once.

Maturity of the underlying tool: The maturity of the underlying EDA tool is a major factor to consider. It not only needs to be capable of reliably handling the transformations, but also needs to be able to parse different possible variations of the input RTL and to generate RTL that will be consumable by all downstream tools.

In-house development of high-level APIs: Another major issue to consider is the need to develop in-house APIs to augment vendor tool APIs. We found it necessary to do so to facilitate concise user spec and, in some cases, to post-process the tool-generated RTL to make it consumable by various tools (e.g., by adding ifdef statements into the generated RTL).

Validation of tool-generated RTL: Next major thing to consider is how to validate the generated RTL, which is supposed to be correct by construction. We used both simulation and FEV methods.

B. Major applications

Hierarchy Restructuring: Modifying the hierarchy of a design is required in the development flow to achieve specific goals at different areas such as validation and structural design. The most common one being to create hierarchies that satisfy specific flow requirements. Structural design hierarchies are typically grouped according to the physical location that they are intended to have, this way processes like timing convergence and floorplanning can be done at a higher level than the hardening target used for synthesis, place and route. In contrast, validation flows are typically executed on single modules or groups of modules that are associated by their functionality instead of their physical location. Transformation flows prove to be particularly effective taking a base input RTL and generating different outputs targeted for hardening and validation that use different arrangement of units for the same design. Figure 2 shows an example of this scenario for a design that contains non-synthesizable modules that are included only on validation targets.

In structural design, restructuring hierarchy is primarily done to generate designs that contain the ideal number of units to balance runtime and quality of results obtained from design automation tools. The gate count associated to modules present in a given design has a direct impact in the resources required by design automation tools to perform their tasks. Having small designs can lead to faster throughput times but may underutilize the potential of the tools. Overly large design may take advantage of tools capacity at the expense of larger runtimes. The flexibility to group designs and move modules between hierarchies through transformation flows enables exploration of arrangements that can achieve an optimal balance between runtime and benefits obtained from the DA tools. Transformations can be automated to perform movement of module instances between hierarchies with a single line of code, as represented in Figure 3, where a simple spec can be processed by the flow to move one unit between hierarchies and reduce the number of physical ports and wires required between units.

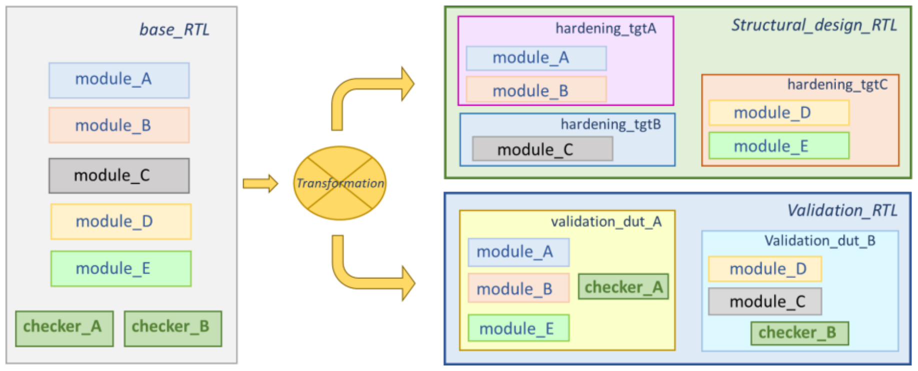
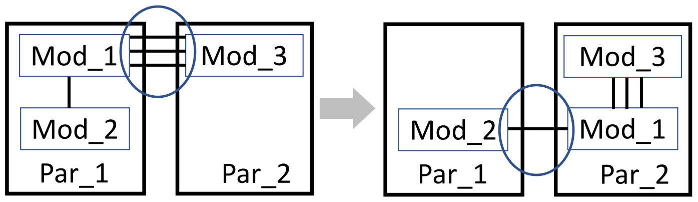


Figure 2 Hierarchy restructuring to generate RTL models consumed by structural design and validation flows



Pseudo spec:
Move Mod_1 from Par_1 to Par_2

Figure 3: Hierarchy restructuring to reduce routing congestion

Hierarchy restructuring without macro	Hierarchy restructuring with macro
Level module_name instance_name	*Level module_name instance_name*
2 par par1	2 par par1
3 localunit localunit0	3 localunit localunit0
3 localunit localunit1	3 localunit localunit1
3 localunit localunit2	3 localunit localunit2
3 localunitx localunitx1	3 localunitx localunitx1
2 par2 par2	2 par par2
3 localunit localunit3	3 localunit localunit0
3 localunit localunit4	3 localunit localunit1
3 localunit localunit5	3 localunit localunit2
3 localunitx localunitx2	3 localunitx localunitx1

Pseudo spec:
Update reference of par2 to be par

Figure 4: Hierarchy restructuring to create MI design

After the RTL is handed off to the backend team, transformation flow can be used in a limited way to perform additional hierarchy restructuring. One example of this usage model is when the design contains multiple top-level blocks delivered by multiple IP teams using different RTL design environments. Each top-level block contains SD partitioned hierarchies created by the transformation flow, but further movement of these partitioned hierarchies across different RTL design environments is not possible. SD team can optimize the floorplan further by using the transformation flow to move partitioned hierarchies from different IP's top-level blocks and create the final SD RTL for implementation without coordination with RTL teams. This kind of SD transformation will not have simulation validation coverage and are limited to those that can be fully covered by Low Power Formal Equivalence Verification.

The creation of multiple-instantiated (MI) blocks and uniquification of MI instances for exceptions handling is also achieved through transformation flows to meet special SD requirements without requiring design team effort. During early phases of a project, SD team can use the transformation flow to do floorplan explorations. For example, when hierarchy restructuring results in two or more physical hardening targets with the same content, the transformation flow can update the reference modules to use a single MI macro design while retaining the IO connections to all the instances, which drastically reduces the physical implementation effort by optimizing the number of hardening targets in a project. Figure 4 shows an example of macro hierarchy creation used to eliminate a hardening target in a design where two hardening targets have the same unit instances.

The reverse transformation of uniquifying existing macros can also be done if the floorplan requires different shapes for different instances of the macro. The finalized SD macro transformations are sent back to the design team and combined with other transformations to create the final signoff RTL to SD. Traditionally, the design team is required to invest significant effort in creating RTL to support early floorplan trials. Currently, the transformation flow enabled the SD team to do multiple iterations of floorplan explorations on its own, providing operational flexibility in the organization and significantly reducing the design team effort.

Feedthrough and Tie-off Insertion: Feedthrough (FT) connections are needed in the RTL used by the hardening tools to reflect the reality of signals that that need to be routed through intermediate hierarchies/blocks without going through any functional changes to reach their destination. Design automation tools are capable of creating FT connections across instances of unique hierarchies; but doing so across instances multiply instantiated (MI) modules remains a challenge. Such cases require special handling in RTL to create ports and connections across multiple modules and ensure the inputs of all MI instances are properly driven, typically involving multiple people and taking turnaround time in the order of days to complete each time that changes are required. Transformation flows are able to operate on an architectural view of the design and generate the required ports, connections and tieoffs to be used on a physical view of the same design without requiring any modification to the input RTL, eliminating the overhead on the design team and reducing the turnaround time for RTL updates to the order of hours taken by the transformation developer to implement changes. Figure 5 shows an example of feedthrough insertion implemented through a specification-based transformation.

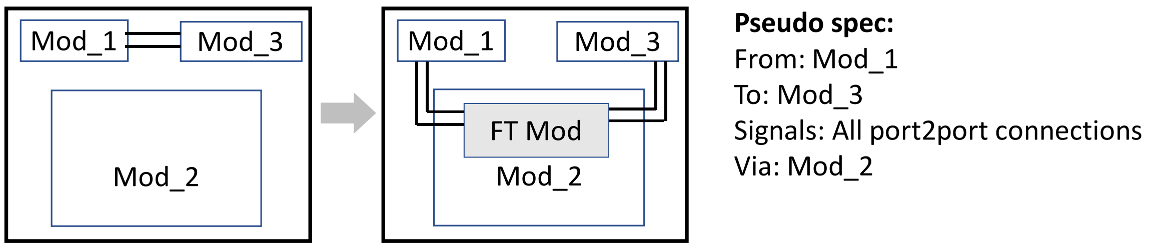


Figure 5: Feedthrough insertion to match physical placement requirement

The use of transformation flows to implement FT connections is also leveraged to include repeater flops in the paths created on the RTL in order to meet timing requirements across long distances. To achieve this, the transformations create RTL that uses a special set of parameters to control the number of repeaters added, communication protocols and type of repeater modules to use for each connection depending on the sender and receiver designs. In order to facilitate timing convergence on paths across FTs, these can be created as serialized set of nodes between origin and destination for point-to-point connection, or as a tree of nodes for broadcasted signals. With such topology of nodes available, the structural design team is able to provide a specification file that instructs the transformation flow the amount and placement of repeater flops between nodes required to meet timing. Repeater insertion through transformation flows eliminates the need of feedback loops between SD and design teams, drastically reducing TAT of RTL generation for timing convergence activities.

Clock/Reset Re-distribution: Following hierarchy restructuring, clock and reset distribution requires expansion and re-alignment for the hardened floorplan. Each additional partition requires replicated clock or reset driver components as well as the associated local partition connectivity cleanup. New design content is not required in this phase as architectural view RTL is replicated for each new instantiation.

Topology Restructuring: Transformations also include complex activities like topology restructuring. With the hierarchy restructuring complete, wholly new floorplan specific RTL is integrated, for example design for test (DFT) content. Furthermore, some floorplan-coupled fabric or topological design content is no longer valid. As an example, topology dependent fabric requires expansion in terms of physical router additions and associated internal routing table updates. In this case, new high-level APIs are used to scrub the deprecated RTL from the design, and then instantiate, create, and connect the new or additional floorplan specific RTL collateral.

Fuse Isolation Insertion: Another beneficial use case of RTL transformation is adding logical isolation to fused outputs. We created high-level APIs that can consume a machine-readable spec that specifies the hierarchy/boundary that should be used to identify output ports, the hierarchies where the isolation modules should be inserted, what enable signal to use, and an exclusion list (outputs that should not be isolated). If the hierarchy to identify the output ports and the hierarchy to insert the isolation module are the same, it would imply a simple case where outputs of a single module are isolated. Figure 6 shows a spec and corresponding transformation. The automation takes care of identifying the output signals of Partition_1 that are also the output signals of SoC. These signals are then isolated by inserting an isolation module in Partition_1. Note that SoC-internal signals are not isolated. The excluded out0 signal is also not isolated. Since the spec only lists Partition_1, isolation is only inserted for the outputs that originate from Partition_1.

Others: There are more transformations possible than the ones listed above, e.g., thermal diode insertion, parameterized logic reduction, validation collateral additions, port splitting etc.

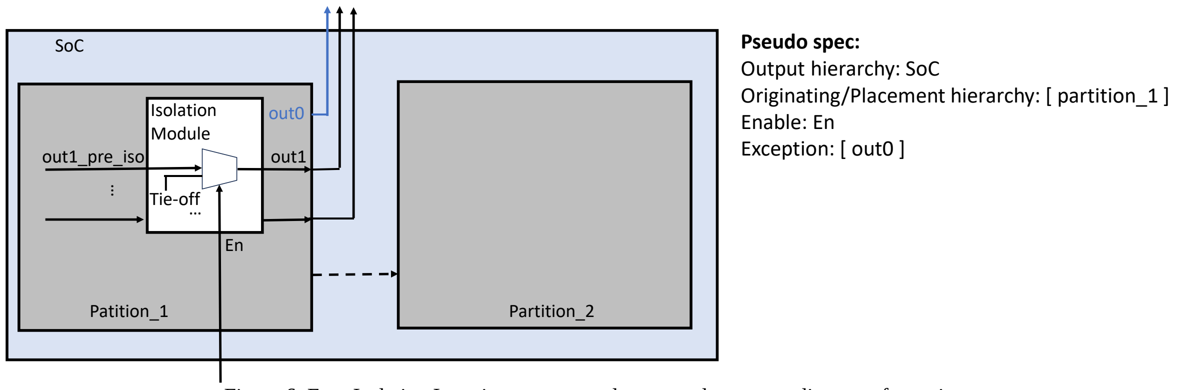


Figure 6: Fuse Isolation Insertion – an example spec and corresponding transformation

C. Validation

As RTL goes through multiple transformations, real errors and bugs can be unintentionally injected from tool/flow issues or from incorrect transformation specification. A solid validation methodology is critical in detecting unexpected changes from these transformations early and prevent costly late detections. Simulation based validation will provide the final transformation RTL signoff but requires validation expertise and has high turnaround time preventing quick feedback to transformation developers. Formal Equivalence Verification (FEV) is the clear choice to validate that two designs are functionally equivalent with shorter runtime but is not effective if some of the transformations are not equivalent, such as topology restructuring and repeater insertion. Such transformations can cause thousands of FEV failures making the debug impossible. A validation approach using different FEV techniques and constraints to mask expected differences was developed to detect real transformation issues. The use of FEV for RTL transformation validation instead of simulation provides a 5-10X runtime efficiency improvement in issue detection and debug resulting in major productivity gains throughout the transformation development cycle. The following sections describe the FEV challenges and techniques deployed to validate different types of transformations.

FEV Session Setup: The transformation FEV validation flow uses a simple RTL vs. RTL FEV session that accepts paths to the pre and post transformed RTL as well as a list of functional blocks which are not modified by the transformation flow. FEV tool directive is applied to create stubbed views of those blocks such that the FEV session only models the functional block interfaces and their connectivity.

Validation for Feedthrough and Tie-off Insertion Transformations: Feedthrough insertion is a FEV compliant transformation that routes functional blocks through intermediate physical hierarchies by the automated creation of new feedthrough modules (FT_mod shown on Figure 5). FEV session uses the full logic views of these newly created modules to model the feedthrough wires and ensure the original end-to-end connections between functional blocks are equivalent. Similarly, FEV uses the full logic view of new tieoff modules added by the transformation flow to model and check tieoff values.

Validation for Repeater Transformations: Repeater transformation represents a simpler category of nonequivalent transforms. The original functional block A to block B connectivity is broken with the insertion of repeater module by the transformation flow. Unlike feedthroughs, repeater modules contain flops and would cause nonequivalence if the full logic RTL views are used in FEV. But if stubbed views of repeater modules are used, FEV cannot “see” the original block A as the driver and will declare failure on block B, preventing detection of real transformation connectivity issues. One method of addressing this expected failure scenario is by applying FEV virtual wire constraints on the stubbed repeater module’s input and output ports (Figure 7a). With these constraints, FEV will pass if the repeater connectivity from block A to block B through the repeater module is correct and will fail if the connectivity is incorrect. Another method of handling repeaters in FEV is to code the repeater block with assignment statements from outputs to inputs based on a special `ifdef compiler directive (Figure 7b). Using this method, FEV reads the RTL view of the repeater module by applying the special directive and models the repeater block with assign feedthrough wires, allowing real connectivity issues to the repeater module to be detected. These techniques mask the repeaters from FEV; simulation will provide the final validation coverage for the repeater functionality.

Validation of Fused Isolation: This is a nonequivalent transformation involving insertion of logic isolation blocks on existing signals. FEV reads in the full logic view of these newly added isolation blocks, creates a cut point on the isolation enable signal and applies a constraint value to disable the isolation logic to mask the expected failures. This way, if the connectivity to the isolation blocks is incorrect, FEV would be able to detect the error.

Validation of Complex Hierarchy Restructuring: Hierarchy manipulation is one of the main transformations required to convert architectural RTL view to SD RTL. It is generally a FEV compliant transformation but still poses challenges. During this transformation, new SD hierarchies such as partitions (groups of functional blocks), macros (multiply instantiated hierarchies), and other physical hierarchies are created, and functional blocks are moved from the original architectural hierarchies into newly created SD hierarchies. FEV relies heavily on the ability to map compare objects by name. In the context of RTL transformation FEV, it expects the instance paths to the stubbed functional blocks to be the same. But these hierarchy movements change the hierarchical instance paths drastically, causing FEV tool to rely on functional mapping based on structural similarities instead. This is a much more compute intensive process which can take several hours on large designs and is prone to incorrect mapping, causing FEV to be ineffective as a fast bug detection tool. To improve FEV mapping accuracy and reduce FEV mapping time from hours to minutes, the hierarchy transformation flow can generate a FEV guidance file with mapping constraints between the pre and post transform hierarchical instance names of the functional blocks, as shown in Figure 8.

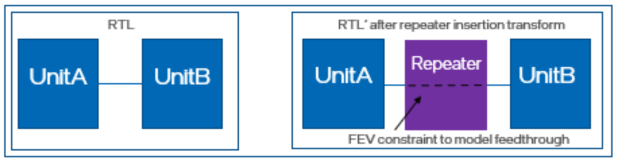

RTL
UnitA
UnitB
RTL' after repeater insertion transform
UnitA
Repeater
UnitB
FEV constraint to model feedthrough

Figure 7a: FEV constraint to model repeater as feedthrough

```
`ifdef REPEATER_TRANSFORM_FEV
   always_comb out = in;
`else
   <repeater functional implementation>
`endif
```

Figure 7b: Verilog `ifdef to model repeater as feedthrough for FEV

Original Hierarchy	New Hierarchy
Level module_name instance_name	*Level module_name instance_name*
1 arch1 arch1	1 section1 sec1
2 unit1 unit1	2 par1 par1
2 unit2 unit2	3 unit1 unit1
1 arch2 arch2	3 unit4 unit4
2 unit3 unit3	2 par2 par2
3 unit4 unit4	3 unit2 unit2
	3 unit3 unit3

FEV mapping:
FEV_map arch1/unit1 sec1/par1/unit1
FEV_map arch1/unit2 sec1/par2/unit2
FEV_map arch2/unit3 sec1/par2/unit3
FEV_map arch2/unit4 sec1/par1/unit4

Figure 8: Hierarchy transformations and associated FEV mappings

Validation of Clock and Reset Distribution Transformations: This transformation involves changing the number of instances driving clocks and resets between architectural view and SD RTL view. For example, in architectural design, two functionally equivalent clock/reset distribution blocks C1..C2 can service 20 functional blocks. In structural design, four functionally equivalent clock/reset blocks C1..C4 can service the same 20 functional blocks, depending on physical design needs. This is a category of equivalent transformation but requires special FEV constraints to convey logic block replication. FEV tools have support for many-to-many mapping between two designs allowing user to specify that C1..C2 on the pre transformed design are equivalent to C1..C4 on the post transformed design. FEV tool would then use this assumption constraint for verifying all the downstream client blocks as well as verifying that the replicated clock/reset blocks themselves are equivalent.

Validation of Complex Topology Restructuring: Topology restructuring transformations, such as changing fabric router's architectural view by adding more routers and changing their client connections, represent a very challenging category of nonequivalent transformations requiring more complex handling in FEV. Figure 9 shows an example of pre and post transformed router topology; the purple/yellow/green router instances are mapped together in FEV, as are other functional blocks of the same name, and the red circles indicate blocks which fail FEV, even if the topology changes and interconnects are transformed properly as intended. Multiple options were considered to handle this transformation in FEV, including ignoring the failures or attempting to model the router modules as wire feedthroughs. Ultimately, an approach was developed to provide as much FEV coverage as possible based on the transformation developer's intent. The blocks with failures are driven by different routers and FEV cut points are applied to the output ports of those driver routers, then mapped together to allow FEV to model them as if they are assumed to be equivalent drivers. With this approach, the FEV constraints act as a comparison spec so that FEV will pass if the transformation flow correctly connects the intended topology and will fail if the connections are not correct.

Improving Transformation FEV Efficiency with Custom Constraint Procedures: Transformation FEV requires application of many constraints to do special modeling or mask expected failures, create cut points and mappings, model feedthroughs, create replication equivalences, all of which would be tedious to manually apply and maintain. To simplify the FEV constraint process, a library of custom constraint procedures was developed using FEV tool API to identify modules, instances, pins and fanouts so that frequently used constraints could be applied easily by the end-user with single line procedure calls.

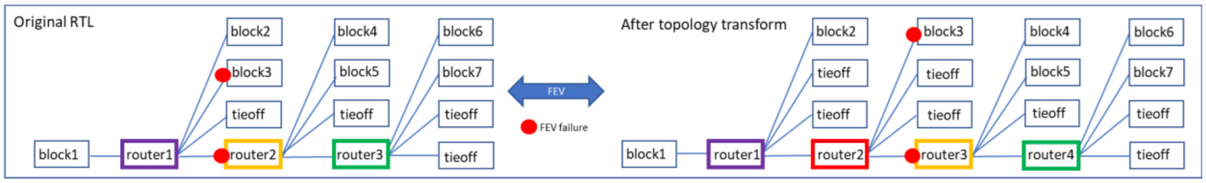


Figure 9: Topology differences before and after transformation causing FEV failures

Table 1: Examples of TAT reduction due to automated RTL transformation in a production VLSI design

RTL Transformation	**TAT Reduction**
Hierarchy Restructuring	4-6 Weeks
Feedthrough and Tie-off Insertion	5-7 Weeks
Fuse Isolation Insertion	2-3 Weeks

The FEV validation methodology was able to detect several issues in the transformed RTL such as input pins tied off to incorrect values, incorrect clock/reset signals, missing or incorrect connectivity as well as unexpected topology changes. These issues could be detected in FEV 5-10X faster compared to simulation. Debug of connectivity issues using FEV is also 10X+ simpler and faster and does not require validation expertise compared to simulation. FEV tool can pinpoint failures directly on the input pins of the stubbed functional blocks so that transformation developers could easily root cause the issue by tracing the driver nets. Although FEV validation methodology cannot fully cover all nonequivalent transformations, it provides fast coverage of a large percentage of transformation bugs and increases confidence in the use of automated RTL transformations.

III. Results

The RTL transformation methods described in this paper were deployed for production use in the development of the latest two generations of Visual Computing segment products of our company. These designs are the main components of some of the industry leading graphics products. Table 1 summarizes the TAT improvement we observed following the deployment of these methods. For most of the activities, the **TAT reduction was generally over 10x**. There was also corresponding reduction in NRE cost, which are not reported here.

As with any new technology, there is a cost to ramping a new tool and methodology, including developing in-house APIs. This cost is expected to be amortized over multiple projects, hence not reported here.

IV. Conclusion and Future work

RTL transformation methods described in this paper resulted in a paradigm shift in the entire design process, resulting in order of magnitude TAT improvements. We believe that the EDA industry is at a point where these methods can be scaled to many designs to yield similar improvements. In terms of future work, one of the major remaining challenges is handling latch-on content, collaterals that accompany RTL and a need to remain up to date as the RTL changes and matures, e.g., UPF. Consuming RTL content from various sources and generating add-ons to RTL (e.g., post-processing needed to add any ifdef statement in RTL) is also a challenge that sometimes require ad hoc efforts on the users' part. We plan to work with the EDA industry to resolve these issues and scale RTL transformation methods further.

Acknowledgment

We thank multiple Graphics teams at Intel that pioneered various RTL transformation technologies described in this paper.

References

[1] "RTL Restructuring Issues", Ed Spering, July 18th,2023. https://semiengineering.com/rtl-restructuring-issues [Website visited on Aug 18, 2023]

[2] "System to catch Implementation gotchas in the RTL Restructuring process", DVCon 2016