

Achieving Faster Code Coverage Closure using High-Level Synthesis

Surendhar Thudukuchi Chandrapandiyani, Preetham Lakshmikanthan, Ashwani Aggarwal
 Samsung Semiconductor India R&D, Bangalore, India
surendhar.tc@samsung.com, preetham.lk@samsung.com, ashwani.a@samsung.com

Youngchan Lee, Youngsik Kim, Seonil Brian Choi
 Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, Korea
yc0325.lee@samsung.com, ys31.kim@samsung.com, seonilb.choi@samsung.com

Abstract—Currently High-Level Synthesis (HLS) is widely being used to design IPs in both the industry as well as in academia. This is because Electronic Design Automation (EDA) synthesis tools are more robust than before and system-level design languages (like SystemC) help in coding algorithms or designs at a higher level of abstraction. However, closing code coverage as part of the HLS flow is extremely difficult and there is no standard flow or methodology to achieve code coverage closure earlier in the HLS design cycle. Our research attempts to provide a set of complete and comprehensive guidelines for designers to achieve 100% code coverage in a structured manner before RTL sign-off. These guidelines are agnostic to EDA tool vendors and can easily be adopted in any HLS design flow. Experimental results show the effectiveness of our approach on a ~1.3M gate Samsung multimedia IP design, where code coverage closure was achieved 22.2% faster when compared to the traditional HLS RTL sign-off flow.

Keywords—High-Level Synthesis; SystemC; Code Coverage; Methodology; Flow Guidelines; RTL sign-off

I. INTRODUCTION

In the last few years, there has been significant traction in using High-Level Synthesis (HLS) [1][2] to design IPs in both the industry and academia. Increasing complexity in system architecture and faster time-to-market constraints are forcing design houses to move to higher levels of abstraction. HLS Electronic Design Automation (EDA) synthesis tools [5][6] are more robust than before and Electronic System-Level (ESL) design using SystemC help in coding algorithms/designs at a higher level of abstraction than traditional Register-Transfer Level (RTL). SystemC [7] combines the best features of the object-oriented C++ language with hardware description constructs, making it well-suited to describe designs at a higher level of abstraction, i.e., a system-level modeling language.

HLS is the process of transformation [3] of a design at a higher level of abstraction (written in C, C++, SystemC, etc.) to RTL (Verilog/VHDL). Figure 1 shows the typical HLS flow. The SystemC design along with a process node technology component library and constraints/directives for the synthesis tool are inputs to the HLS engine. The output of the HLS engine is a synthesized Verilog/VHDL design that best satisfies all the specified constraints.

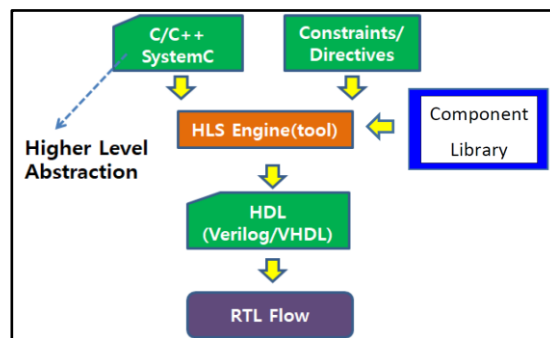


Figure 1. High-Level Synthesis Flow

HLS helps in quick turnaround of design changes and allows for design space exploration and trade-off among various configurations in regards to area, latency and power constraints. Design re-spins using HLS take much lesser time and engineers can focus on the design work while tools handle the implementation specifics. However, HLS tools are still not fully automated and require the designer's inputs to guide the tool [4] in arriving at an optimal solution.

Closing code coverage as part of the HLS flow is extremely difficult. A literature survey search [8][9][10][11] yielded no good results on a clear cut procedure to follow such that code coverage closure is achieved as part of the HLS flow before RTL sign-off. Our research work attempts to provide a set of guidelines for designers to achieve code coverage closure in a structured manner before RTL sign-off. It also helps reach code coverage closure faster than the traditional methods followed. These guidelines are EDA tool vendor agnostic and have been tested out on a Samsung multimedia IP that synthesized to a ~1.3M gate design.

The organization of the rest of the paper is as follows. Section II provides information about some commonly used code coverage terminology. The crux of this research, i.e., the methodology for closing code coverage in HLS, is explained in Section III. Section IV provides experimental results to validate the code coverage closure methodology described. Finally, conclusions drawn from this work are detailed in Section V.

II. CODE COVERAGE MEASUREMENT TERMINOLOGY

Code coverage is a measurement of the percentage of the design source code executed by running tests (regressions) and applying various input vector combinations. There are various measurement metrics that constitute code coverage. Those terminologies are explained below:

- (1) **Line Coverage:** This is a measure of all possible executable lines of code in the source code having been executed at least once. It ensures that each line in the design source is covered at least once by the tests run.
- (2) **Function Coverage:** This measures the extent to which functions present in the source code are covered during testing. Depending on the combination of input values used, a function may or may not be called. The purpose of this metric is to ensure that all the functions in the design source have been called.
- (3) **Condition/Expression Coverage:** Wherever there is a condition in the source code, its evaluation would result in a Boolean value of either true or false. Conditional coverage looks at all Boolean expressions and counts the number of times it was true or false.
- (4) **Block Coverage:** This is a measure of which blocks in the code have executed and which have not.
- (5) **Branch Coverage:** This looks at all the conditional branching statements ('if-then-else' and 'case' branches) and counts each branch taken.
- (6) **Statement Coverage:** This measure provides information whether or not all statements within a block have executed.
- (7) **Toggle Coverage:** This is a measure of the activity (value change either from '0' to '1' or from '1' to '0') of various signals in a design and provides information on un-toggled signals or signals that remain constant during a simulation run.
- (8) **Finite State Machine (FSM) Coverage:** This is a measure of what states were visited in the FSM and which transitions were taken (state coverage, transition coverage and arc coverage).

III. METHODOLOGY FOR CLOSING CODE COVERAGE IN HLS

Closing code coverage earlier in the HLS design cycle would lead to reduction in regression runs/testing and thereby consequently reducing CPU usage and simulation tool licenses consumed. However, closing code coverage [12][13] earlier in the HLS flow is extremely difficult and there isn't any well-defined procedure to achieve this before RTL sign-off.

It is impossible to completely close code coverage at the SystemC level. If one closes coverage at the SystemC level, there's no support for things like toggle coverage, FSM coverage, etc. Completely closing coverage at the RTL level would increase the Turn Around Time (TAT) in rapidly re-spinning designs using HLS. Hence, we devised an approach which is an optimal mix of code coverage measurement that spans across both SystemC and

RTL levels to achieve coverage closure. Figure 2 is a flow diagram of the proposed HLS code coverage methodology.

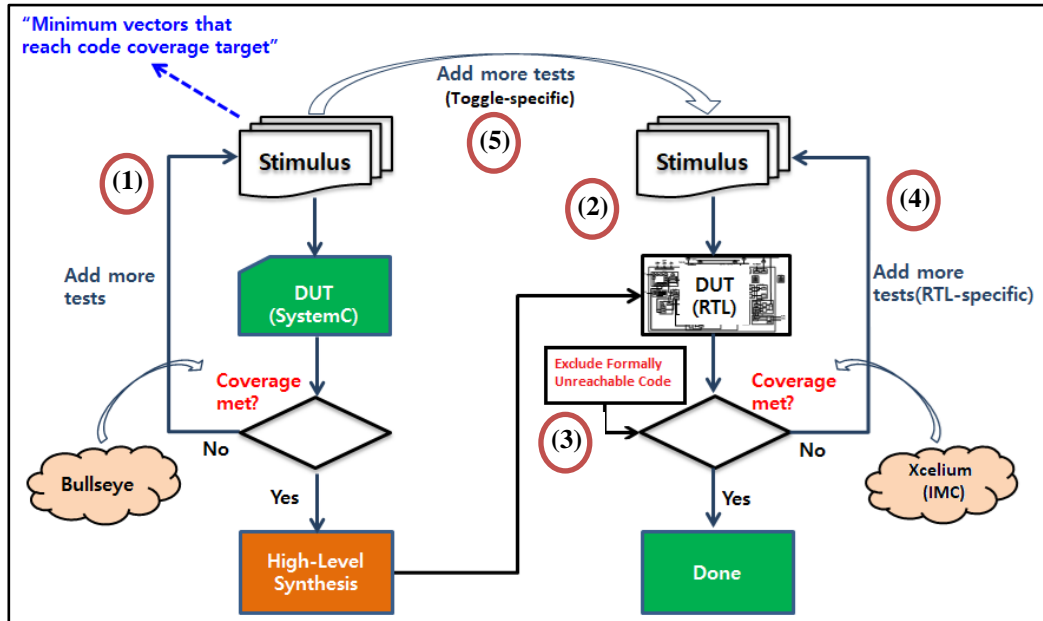


Figure 2. HLS Code Coverage Closure Flow

The flow diagram is explained below as a set of guidelines/5 steps that designers can use to achieve 100% code coverage in a structured way before RTL sign-off. This methodology helps a designer reach code coverage closure faster than the traditional methods followed.

- (1) Make use of any code coverage tool (that also supports condition/decision coverage) to achieve 100% code coverage on the SystemC design. Use the minimal set of tests from your regression suite to achieve this 100% code coverage. This will ensure lesser testing/coverage measurement at the RTL level. Our experiments have shown Bullseye [14] to cover code better than GNU's Gcov/Lcov tools [15]. Lcov is the graphical front-end built over GCC's Gcov coverage testing tool. Gcov covers only line and function coverage, with no support to measure condition/decision evaluation. It also does not provide any option to exclude pieces of code from measurement. Bullseye alleviates the prior problems listed with Gcov/Lcov.

Tip 1: There is a possibility that SystemC code coverage tools might have their limitations even though they report 100% code coverage. Our experiments using a SystemC code coverage tool showed that 100% code coverage was achieved even though,

- a) It didn't cover bitwise, logical not and relational operators in Right Hand Side (RHS) expression evaluation.
For e.g.: `out1 = (!in1) & (in2);`
The '!' and '&' in the RHS are not covered.
- b) It didn't cover all array elements on the RHS of an expression.
For e.g.: `out2 = mem_arr[indx];`
Just one index value is enough and the coverage tool shows this statement as 100% covered. An important point to note is that not all array indices have been tested.

This SystemC code coverage tool reliability factor is yet another important reason that code coverage closure has to be finished at the RTL level. Table 1 on the next page gives a summary of the code coverage and debug features present in SystemC vs. RTL code coverage tools.

Table 1. Feature Support in SystemC vs. RTL Code Coverage Tools

Features	C++/SystemC		RTL
	Gcov/Lcov [15]	Bullseye [14]	Questa [16], VCS [17], Xcelium [18]
Statement Coverage	✓	✓	✓
Branch Coverage	X	✓	✓
Condition/Expression Coverage	X	✓	✓
Instance (Object) Based Reporting	X	X	✓
Visualization & Debugger	X	✓	✓
Waive/Exclusion Feature	X	✓	✓
Toggle Coverage	N.A	N.A	✓
FSM Coverage	N.A	N.A	✓

(2) Run the same minimal set of tests (as described in step III(1)) on the synthesized RTL code. Measure RTL code coverage using your favorite code coverage tool. Exclude measurement of toggle coverage here and the code coverage achieved will still be lesser than 100%. This is because the synthesis tool adds extra RTL components for:

- Pipelining and stall architecture
- Implicit “else” or “default” branch statements
- Unrolled loops (Parallel copies are made in hardware)
- FSMs
- External memory structures
- First-In First-Out (FIFO) buffers, math functions and other structures from tool vendor libraries
- Other synthesis tool directives

All these new RTL code constructs will have to be covered at the RTL level only. It is possible for the synthesis tool to generate a few efficient RTL structures by specifying synthesis coding directives/options, but majority of the RTL code generated cannot be easily correlated back to the SystemC design. Figure 3 gives an example where direct correlation cannot be made between the SystemC code and its synthesized RTL.

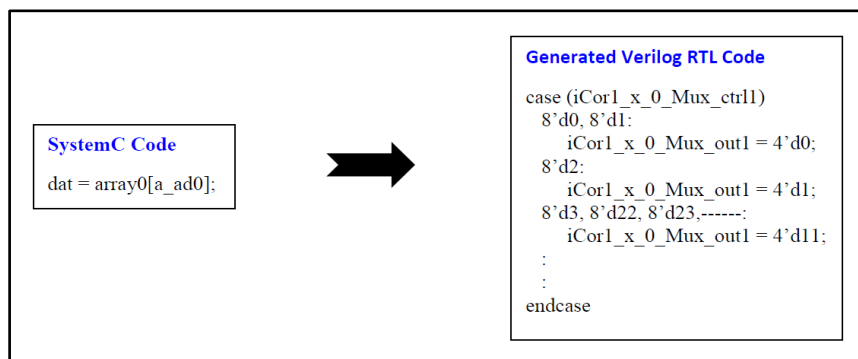


Figure 3. Correlation between SystemC code and its Synthesized RTL

(3) It is possible that some of these new RTL constructs are just dead code and might not be reachable/sensitized from the inputs. Use your favorite formal verification tool and run the “unreachability” analysis on this synthesized RTL code. Our experiments have shown lots of unreachable code and states in the RTL design. The “formally” unreachable code segments and states need to be excluded from code coverage analysis.

Tip 2: Increase the “prover time” to a much larger value than the default in the formal tool for running unreachable analysis. This is because there are chances that the tool might return many “undetermined” cases due to insufficient default run time provided.

(4) Having excluded the unreachable code, add tests to cover the remaining new RTL code constructs present. This will complete code coverage for the RTL design with the exception of toggle coverage.

- (5) Finally, add tests to cover the toggle coverage holes that remain. Once this is done, code coverage will be at 100% and the RTL design can be signed-off for the back-end design flow.

IV. EXPERIMENTAL RESULTS

A Samsung multimedia IP design was developed using HLS. The original SystemC design had 6980 lines of code, while the synthesized RTL design contained 356959 lines of code. Finally, this translated to a ~1.3M gate sized design. The methodology guidelines specified in Section III were applied on this multimedia IP and 100% code coverage was achieved as follows for each step:

- (1) The Bullseye tool was used to measure code coverage of the SystemC design. 69 tests (minimal number) were used to achieve 100% code coverage at the SystemC level. Figure 4 shows the coverage aspects of the design.

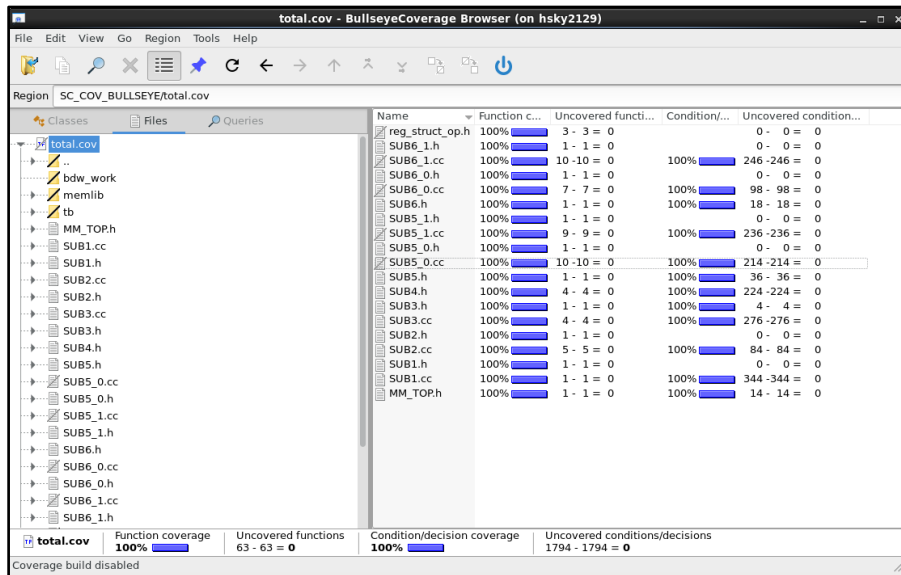


Figure 4. Code Coverage of the SystemC Design

- (2) The same 69 tests (minimal number) run at the SystemC level, were run on the synthesized RTL design and code coverage measured. Toggle coverage was excluded in this step and Figure 5 shows that 96.77% code coverage was achieved on the RTL design.

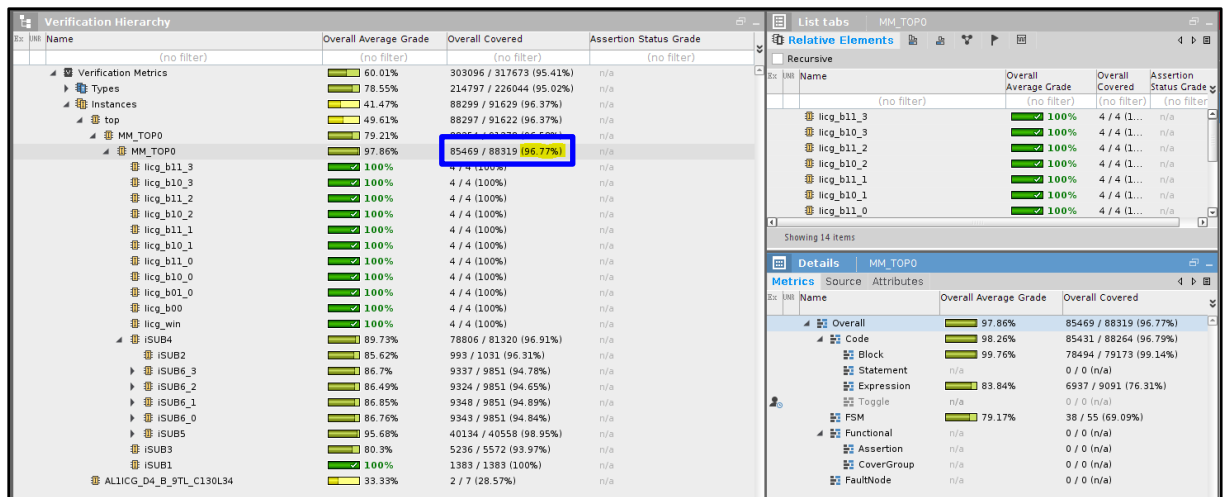


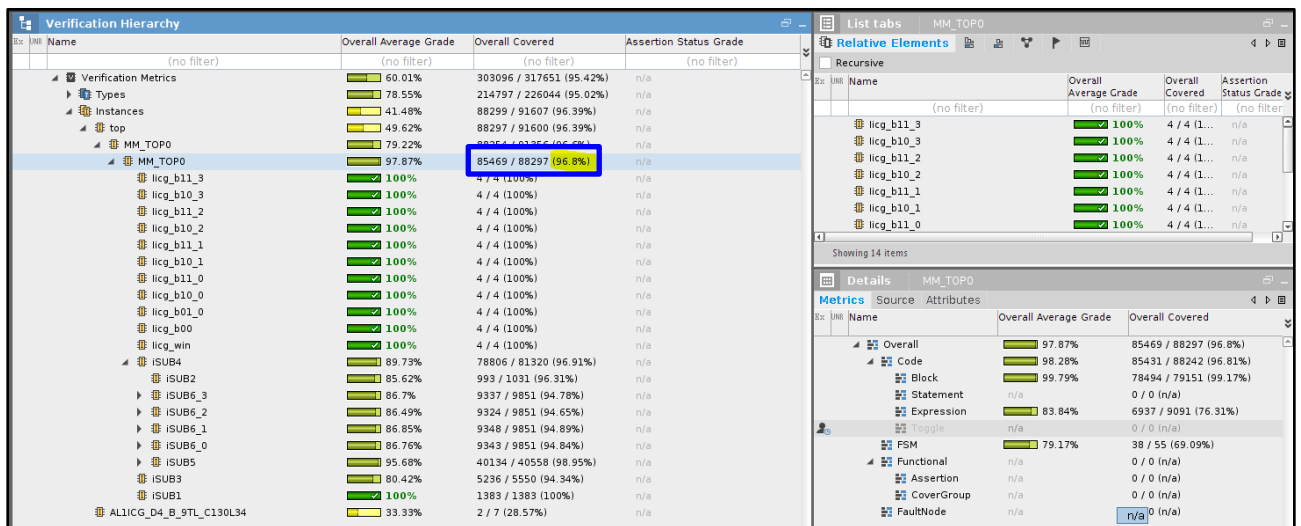
Figure 5. RTL Code Coverage (Excluding Toggle)

- (3) Formal “unreachability” analysis was run on the RTL design for a period of 36 hours (the “default” prover time is 8 hours). Figure 6 is the run log file that shows 112 code blocks are unreachable. These unreachable code blocks were excluded from the RTL code coverage measured. Figure 7 shows that 96.80% code coverage was achieved on the RTL design (excluding the 112 unreachable code blocks).

```

=====
JasperGold Verification Results
=====
2020.09p002 64 bits for Linux64 3.10.0-1062.1.1.el7.x86_64
Host Name: srvr1.samsung.com
Working Directory: /user/hls004/JASPER-RUN
=====
DESIGN INFO
=====
Top Level Module      : mm_top0
=====
SUMMARY
=====
Total Tasks           : 4
Total Properties      : 15248
  assumptions          : 0
    - approved        : 0
    - temporary       : 0
  assertions          : 0
    - proven          : 0
    - marked_proven   : 0
    - cex             : 0
    - ar_cex          : 0
    - undetermined    : 0
    - unprocessed     : 0
    - error           : 0
  covers              : 15248
    - unreachable     : 112                ( 0.7% )
    - covered         : 15136            ( 99.3% )
    - ar_covered      : 0                ( 0.0% )
    - undetermined    : 0                ( 0.0% )
    - unprocessed     : 0                ( 0.0% )
    - error           : 0                ( 0.0% )
=====
  
```

Figure 6. RTL Design “Unreachability” Analysis Results



The screenshot shows the Verification Hierarchy window on the left and the Metrics window on the right. The Verification Hierarchy window displays a tree view of the design with columns for Name, Overall Average Grade, Overall Covered, and Assertion Status Grade. The MM_TOPO module is expanded, showing a list of code blocks with their respective coverage percentages. The Overall Covered column for MM_TOPO is highlighted in yellow, showing 85469 / 88297 (96.8%).

Name	Overall Average Grade	Overall Covered	Assertion Status Grade
Verification Metrics	60.01%	303096 / 317651 (95.42%)	n/a
Types	78.55%	214797 / 226044 (95.02%)	n/a
Instances	41.48%	88299 / 91607 (96.39%)	n/a
top	49.62%	88297 / 91600 (96.39%)	n/a
MM_TOPO	79.22%	85469 / 88297 (96.8%)	n/a
MM_TOPO	97.87%	85469 / 88297 (96.8%)	n/a
licg_b11_3	100%	4 / 4 (100%)	n/a
licg_b10_3	100%	4 / 4 (100%)	n/a
licg_b11_2	100%	4 / 4 (100%)	n/a
licg_b10_2	100%	4 / 4 (100%)	n/a
licg_b11_1	100%	4 / 4 (100%)	n/a
licg_b10_1	100%	4 / 4 (100%)	n/a
licg_b11_0	100%	4 / 4 (100%)	n/a
licg_b10_0	100%	4 / 4 (100%)	n/a
licg_b01_0	100%	4 / 4 (100%)	n/a
licg_b00	100%	4 / 4 (100%)	n/a
licg_win	100%	4 / 4 (100%)	n/a
ISUB4	89.73%	78806 / 81320 (96.91%)	n/a
ISUB2	85.62%	993 / 1031 (96.31%)	n/a
ISUB6_3	86.7%	9337 / 9851 (94.78%)	n/a
ISUB6_2	86.49%	9324 / 9851 (94.65%)	n/a
ISUB6_1	86.85%	9348 / 9851 (94.89%)	n/a
ISUB6_0	86.76%	9343 / 9851 (94.84%)	n/a
ISUB5	95.68%	40134 / 40558 (98.95%)	n/a
ISUB3	80.42%	5236 / 5550 (94.34%)	n/a
ISUB1	100%	1383 / 1383 (100%)	n/a
AL1ICG_D4_B_9TL_C130L34	33.33%	2 / 7 (28.57%)	n/a

The Metrics window shows a detailed view of the MM_TOPO module metrics. The Overall Average Grade is 97.87% and the Overall Covered is 85469 / 88297 (96.8%). The metrics are broken down by source and attributes, showing that the Code source has the highest coverage at 98.28% (85431 / 88242).

Source	Attributes	Overall Average Grade	Overall Covered
Overall		97.87%	85469 / 88297 (96.8%)
Code		98.28%	85431 / 88242 (96.81%)
Block		99.79%	78494 / 79151 (99.17%)
Statement		n/a	0 / 0 (n/a)
Expression		83.84%	6937 / 9091 (76.31%)
Toggle		n/a	0 / 0 (n/a)
FSM		79.17%	38 / 55 (69.09%)
Functional		n/a	0 / 0 (n/a)
Assertion		n/a	0 / 0 (n/a)
CoverGroup		n/a	0 / 0 (n/a)
FaultNode		n/a	0 / 0 (n/a)

Figure 7. RTL Code Coverage (Excluding Uncovered Code Blocks)

- (4) After careful analysis, 47 more tests were added to cover the remaining new RTL code constructs. Figure 8 on the next page shows that 100% code coverage (excluding toggle) was achieved on the RTL design after adding these new tests. However, the overall code coverage including toggle seen at this point is still only 94.71%.

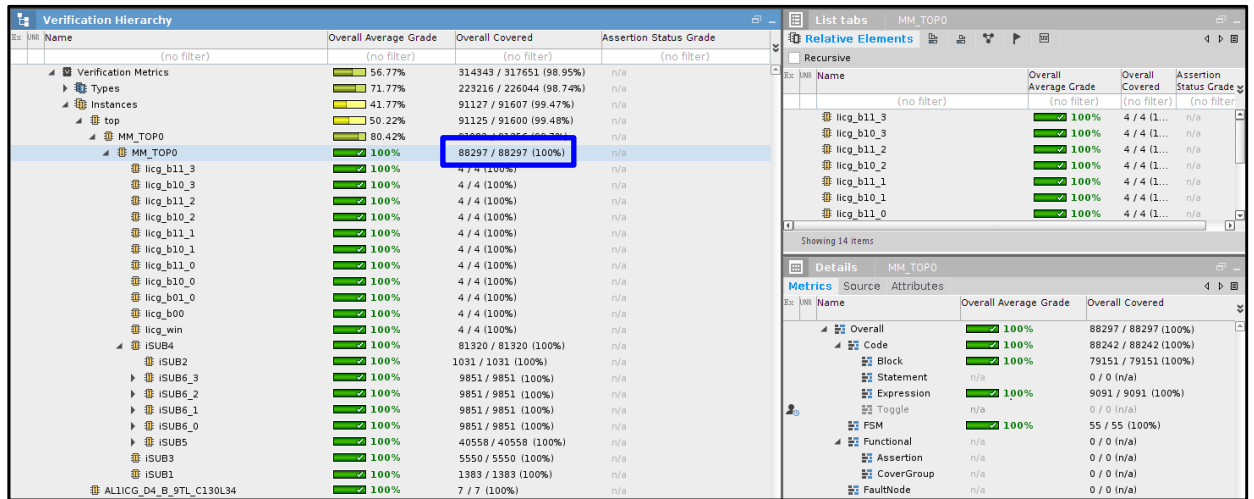


Figure 8. Code Coverage Including New RTL Constructs (Excluding Toggle)

- (5) Finally, added 13 tests to cover all the toggle points for this design. Figure 9 shows that 100% code coverage was achieved on the RTL design after adding these new tests and all the toggle bins were hit.

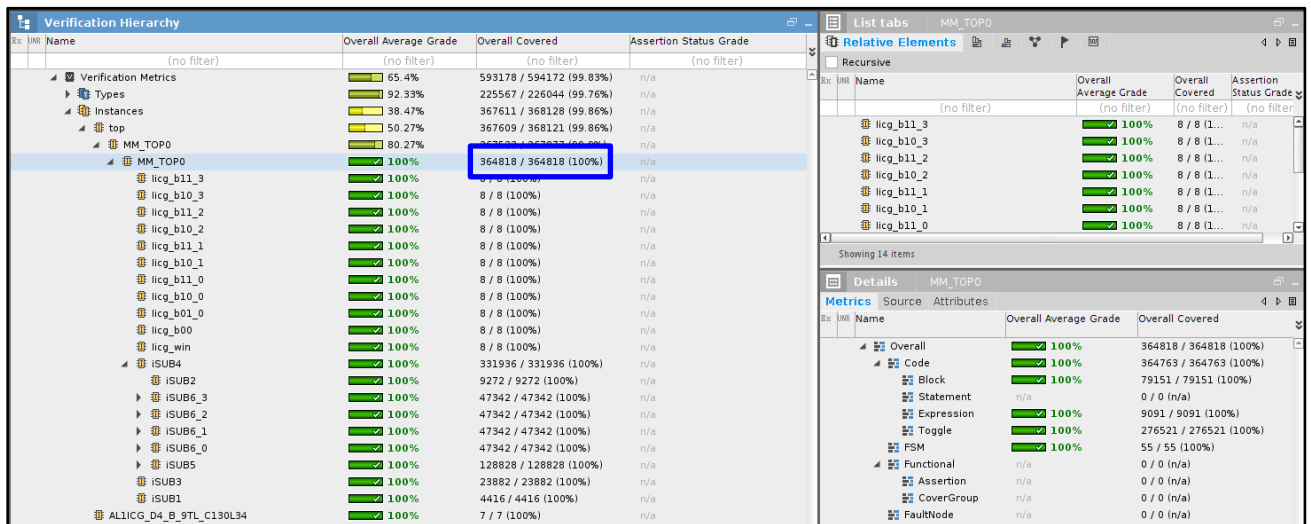


Figure 9. RTL Code Coverage (Including Toggle)

Code coverage closure was achieved **22.2%** faster on this ~1.3M gate multimedia design with these guidelines when compared to the traditional HLS RTL sign-off flow.

V. CONCLUSION

We have defined a complete and comprehensive methodology agnostic to EDA tool vendors, which achieves code coverage closure earlier in the HLS design cycle. Experimental results clearly show the effectiveness of our approach on a ~1.3M gate multimedia design, where much faster code coverage closure was achieved. Learnings from our experiments with various tools during this exercise have been provided as useful tips and information pointers (as part of this paper write-up) for practical use during HLS design. The HLS code coverage methodology guidelines described here are ready to be deployed across various design teams at Samsung.

ACKNOWLEDGMENT

The authors would like to sincerely thank the Cadence HLS team – Mr. Vijay Prakash, Mr. Mike Meredith, Mr. Xingri Li, Mr. Lokesh Jigalur and Mr. Jongchul Kim for supporting the Samsung IDT-DM team with the Stratus toolset and in answering our numerous queries on various HLS topics.

REFERENCES

- [1] G.D. Micheli, “Synthesis and Optimization of Digital Circuits”, McGraw Hill Publication, 1994.
- [2] S. Baranov, “High Level Synthesis of Digital Systems: For Data Path and Control Dominated Systems”, ISBN Canada Publication, 2018.
- [3] M. Fingeroff, “High Level Synthesis Blue Book”, Xlibris US Publication, 2010.
- [4] P. Coussy, D.D. Gajski, M. Meredith and A. Takach, “An Introduction to High-Level Synthesis”, in IEEE Design & Test of Computers, vol. 26, no. 4, pp 8-17, July 2009.
- [5] Cadence -- Stratus High-Level Synthesis. {Online}. Available: https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html
- [6] Siemens -- Catapult High-Level Synthesis. {Online}. Available: <https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/>
- [7] “IEEE Standard for Standard SystemC Language Reference Manual”, in IEEE Std. 1666-2011 (Revision of IEEE Std. 1666-2005), IEEE Publication, 2012. {Online}. Available: <https://ieeexplore.ieee.org/servlet/opac?punumber=6134617>
- [8] F. Sijstermans and J. Li, “Working Smarter, Not Harder: NVIDIA Closes Design Complexity Gap with High-Level Synthesis”. {Online}. Available: https://s3.amazonaws.com/s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_96098.pdf
- [9] G. Singh, “Closing Coverage in a High-Level Synthesis Flow”. {Online}. Available: https://s3.amazonaws.com/s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_103675.pdf
- [10] B. Bowyer, “Closing Functional and Structural Coverage on RTL generated by High-Level Synthesis”. {Online}. Available: https://s3.amazonaws.com/s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_94094.pdf
- [11] T. Kawabe, “Konica Minolta Proves C++ Level Signoff Possibilities using Catapult HLS Platform”. {Online}. Available: https://s3.amazonaws.com/s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_105113.pdf
- [12] A. Tan, “Closing Coverage in HLS”. {Online}. Available: <https://semiwiki.com/eda/7757-closing-coverage-in-hls/>
- [13] J. Sanguinetti and E. Zhang, "The relationship of code coverage metrics on high-level and RTL code," 2010 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2010, pp. 138-141.
- [14] Bullseye Coverage Measurement. {Online}. Available: <https://bullseye.com/measurementTechnique.html>
- [15] Gcov – A Test Coverage Program. {Online}. Available: <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>
- [16] Siemens – Questa Advanced Simulator. {Online}. Available: <https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/>
- [17] Synopsys – VCS. {Online}. Available: <https://www.synopsys.com/verification/simulation/vcs.html>
- [18] Cadence – Xcelium Logic Simulation. {Online}. Available: https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html