An experience to finish code refinement earlier at behavioral level

*Dae-Han Youn, Sik Kim, Byeong Min, Kyu-Myung Choi
Samsung Electronics Co. Ltd.
San#24 Nongseo-Dong, Giheung-Gu,
Yongin-City, Gyeonggi-Do, Korea 446-711
*(82)-31-209-4263
*dhan.youn@samsung.com

ABSTRACT
Use of behavioral description and HLS (high level synthesis) flow has allowed designers to shorten design TAT (turn-around- time) with its better performance in hardware description, functional simulation and RTL generation when it compared to the design flow with RTL designs. However, code refinement in HLS flow would be performed in RT (register-transfer) level due to the lack of proper methodology in HLS flow. Code refinement in RT level is time-consuming due to simulation time overhead and lack of readability of synthesized RTL designs. This paper describes our experience to move the code refinement flow from RT level to behavioral level. We analyzed code coverage gaps during and after the logic simulation of behavioral level and RT level designs, and proposed behavioral level design guides to get the same level of coverage value, which helps to finish the code refinement work at the behavioral level design stage. Our experiments showed that we can get fairly good quality of code refinement result with this proposal as well as over 70% reduced code refinement time.

Categories and Subject Descriptors
[ESL Design and Verification]: Experience using ESL and/or TLM for system-level design and verification.

General Terms
Code refinement, Code coverage, HLS flow

Keywords
Behavioral Code Coverage, High-Level Synthesis.

1. Introduction

HLS has been widely adopted in SoC designs for its high design productivity. Once a behavioral level design is prepared initially, behavioral code refinement follows to get the design that best fits to a given specification [1]. In HLS flow, *Code Refinement* means rewriting or optimizing the behavioral source code to meet the size and timing requirements or to achieve function and code coverage goals. Code refinement to meet size and timing requirement can be performed based on the HLS report without exploring RTL design. However, code refinement to achieve function and code coverage goals makes a long feedback loop to include RTL exploration for debugging or RTL simulation. This paper limits the meaning of code refinement within the scope of coverage closure of RTL and behavioral designs.

Traditional high-level synthesis flow is shown in the left side of figure 1. To get the design which meets the requirements in specification, a series of feedbacks is needed. The internal loop is for behavioral refinement with behavioral simulation. The external loop is mainly for corrective work after debugging and measuring coverage. Though behavioral simulation of the internal loop is quick and simple, we have used RTL simulation of the external loop to measure coverage, because there hasn't been much activity to verify behavioral design using coverage metrics due to the lack of proper tools.

Simulation with the synthesized RTL design includes a closure of code/function coverage metrics, where the simulation is time-consuming due to simulation time overhead and lack of readability of the synthesized RTL designs, which results in debugging overhead. Once a design bug is found in this stage, the design activity is returned to the behavioral coding stage, which forms a relatively long feedback loop and it is time-consuming work.

In HLS flow, reducing code refinement time is inevitable to meet the TAT requirement of modern SoC designs. A new idea is to move this code refinement work to the earlier stage at behavioral level as shown in the right side of figure 1. This means that verification at behavioral level should be able to cover the verification activities of measuring coverage in RTL. Therefore, we have applied JEDAcc tool to measure coverage at behavioral level and set up new verification flow like the right side of figure 1. Successful code refinement at behavioral level makes the feedback loop short as shown in figure1.

This paper focuses on the analysis of code coverage metrics in behavior level and RT level to see if the code coverage measurement can be migrated. Sanguinetti and Zhang showed their behavioral level code coverage definition is equivalent to RTL code coverage and, as the number of test increases the weighted average of the behavioral level code coverage result tends to converge to the RTL code coverage result [2]. However, we found the RTL code coverage trend leaves the gap after the simulation with all testbench, which is the reason that the design refinement at RT level is still needed. This paper shows the analysis result of this gap in coverage trends. It includes the analysis of the trends of code coverage closure, high-level design guides to minimize the coverage gap and a design flow to meet the design quality early at behavioral level design stage.

RTL synthesis flow has been successfully established, where gate-level design is no longer explored to meet functionality requirements. They finish code refinement in RTL design and throw it to a logic synthesizer tool. Formal equivalence checker tool will check whether the transform from RTL to gate is successful or not. Similarly, in an HLS flow, RTL code refinement can be waived as long as the HLS flow supports behavioral level code refinement methodology and formal equivalence checker [3] between behavioral level and RTL designs.

Chapter 2 shows the detail of code coverage analysis. Chapter 3 introduces high-level design guide to minimize coverage gap between behavioral design and RTL design. Chapter 4 shows experimental results, and it exploits the correlation between the code coverage results in two different levels. Finally, Chapter 5 covers a conclusion.

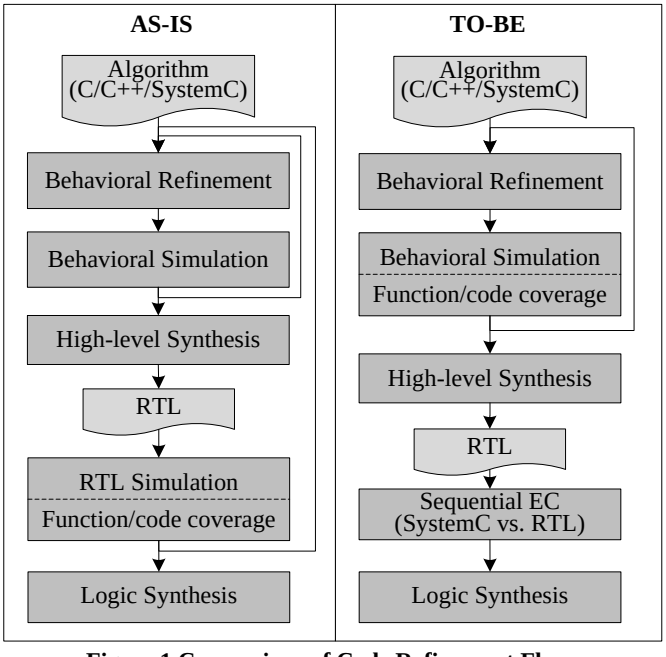

Figure 1 Comparison of Code Refinement Flows

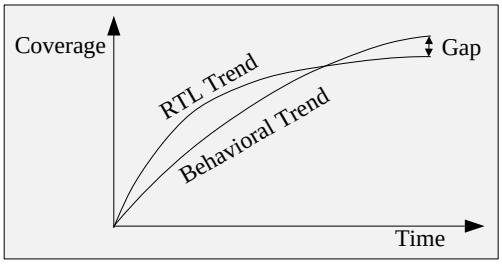

Figure 2 Gap in Coverage Trends

2. Analysis of Code Coverage Measurement

In this section, we analyzed characteristics of behavioral and RTL code coverage metrics. In our experiments, we applied Cadence ICC solution for RTL coverage measurement and also applied JEDAcc for behavioral coverage measurement. Because the two coverage metrics are implemented in different tools, it is very important to compare the two metrics to replace one with another. There are three typical coverage metrics in RTL domain, such as block, expression and toggle [4]. We mainly use the block and expression coverage metric to measure the quality of the test inputs and the design. Block coverage is a basic code coverage type that identifies which block of the code has been executed and which has not. This characteristic is identical to that of line coverage metric at behavioral level. Expression coverage factorizes logical expressions and monitors them during simulation run. It measures how thoroughly the testbench exercises the logical expressions in assignment statements and procedural control constructs (if/case conditions). This characteristic is identical to the sum of characteristics of decision, condition and multi-condition metric at behavioral level [5].

Figure 3 shows the relation between behavioral and RT level code coverage metrics. In figure 3, when the if-statement is hit, the behavioral level line coverage increases, which is identical to the block coverage of RTL. Sum of the decision, condition, and multi-condition is identical to the expression coverage of RTL. Behavioral level decision coverage monitors the results of whether the if-statement is true or false. Multi-condition monitors the combination of each condition.

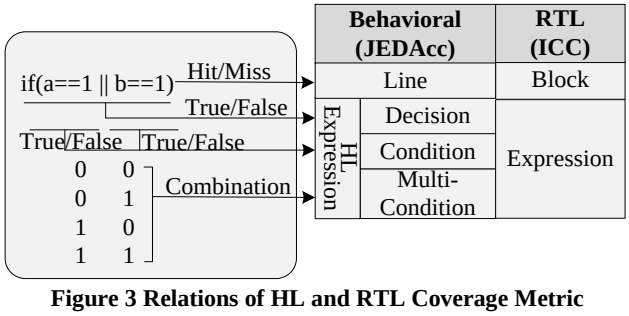

Figure 3 Relations of HL and RTL Coverage Metric

We can intuitively understand that these coverage metrics are closely correlated since a line of behavioral code will produce potentially many lines in RTL design. However, because of this reason, the coverage trend of each design can show different shape as simulation time grows, as shown in figure 2. Figure 4 explains how this gap happens. The two expressions are translated into multiple lines of RTL, and according to the order of excitations, coverage results in each side can be different. When first half of the statement is excited in behavioral level coverage, coverage ratio is 1/2, and RTL code coverage is 3/4. If second statement is excited first, the RTL coverage ratio is 1/4, while behavioral level code coverage ratio is still 1/2.

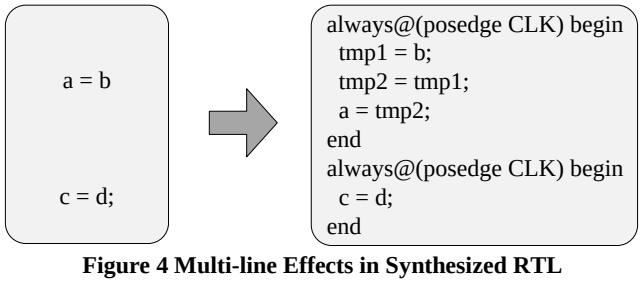

Figure 4 Multi-line Effects in Synthesized RTL

The coverage result reaches a certain saturation point as most of the statements in the design are excited with good quality of testbench. In other words, once fair amount of coverage items are covered, the coverage result is expected to reach a certain point. However, in our experiments, we found that the saturation points of behavioral level and RTL designs are different, shown as the "gap" in figure 2. RTL code coverage value is always the same as or less than behavioral level code coverage value after a certain saturation point. Unless the RTL coverage result meets certain coverage goals, RTL code refinement is still needed until the reasons of low coverage result are all identified. An RTL design with low coverage result might have unreachable codes, which incurs untestable logic blocks in synthesis process. Therefore, we analyzed root causes of the coverage gap and provided high-level design guides to minimize the gap. The detail of high-level design guide will be presented in the next chapter.

3. High Level Design Guide

This chapter shows high-level design guides from our experiences in finding the root cause of the coverage difference, and it helps to minimize the gap so that we do not need to perform further code refinement in RTL designs.

3.1 High Level Synthesis Constraints

We usually use high-level synthesis constraints during high level synthesis, but some of them may affect not only functionality but code coverage result. Therefore, a user needs to consider the effect on functionality and code coverage before using them. For example, wait statement is used to insert a delay of 1 cycle which can affect the timing and functionality of the given example. In the behavioral source code at the top left of figure 5, there is a for-loop which iterate two times, and there are 3 wait statements in the loop body. The combination of the given loop count number and the number of wait statement causes unreachable RTL code generation and lower code coverage result. In the timing diagram at the bottom left of figure 5, the period of cycle states and drain is 2, because the loop count for the for-loop is set to 2. This shows that the drain and the cycle2_state cannot be high simultaneously. Therefore, the unreachable codes in the RTL design has been generated as illustrated at the right side of the figure 5. Unreachable codes can be generated by inserting wait statements more than loop count for the for-loop. The designers should be aware of these corner case results during their design exploration and it is recommended to avoid these corner cases in general.

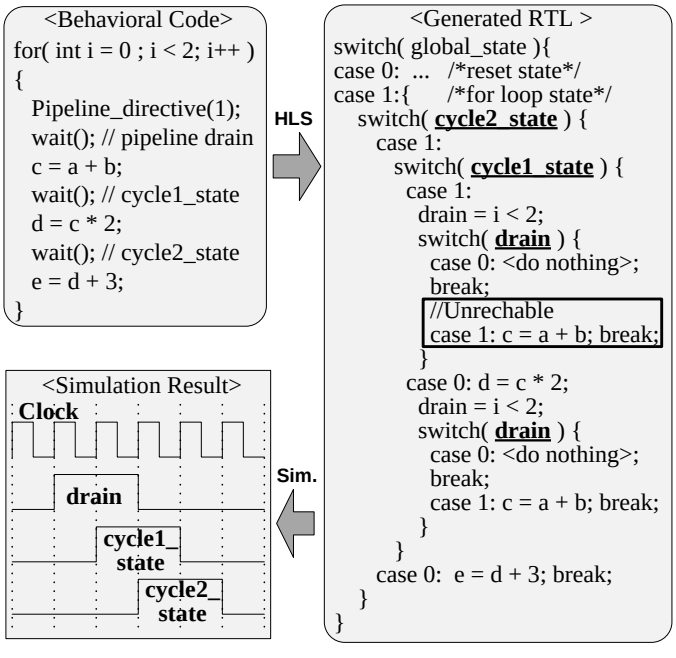

Figure 5 No Case of "drain ==1 && cycle2 ==1"

3.2 Style of RTL Output

Good RTL coding style of generated RTL designs can give better RTL code coverage result. The "if statement" which has no execution body in RTL code as shown in the left side of figure 6 can be the cause of low branch coverage result. If there is no constraint in high level synthesis process, the synthesizer would generate the RTL code shown at the left side of figure 6. High level synthesizer should be guided with the option for the type of output style:

-- output_style_starc = +S2.8.1.4

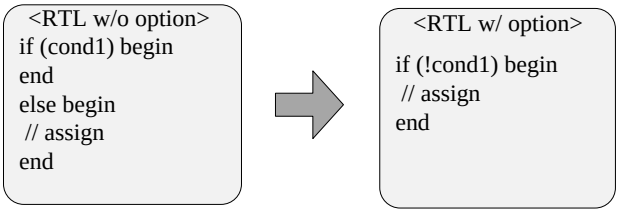

Figure 6 RTL Generation According to Right Lint Rule

3.3 Optimization Options

The unreachable codes in RTL which a designer did not intend to make may be generated in optimization process. Figure 7 shows an example case related to the optimization of switch statement. During the implementation and optimization process of the switch statements, the unreachable codes can be generated as shown at the right side of figure 7. If we assume that the condition "cs1==0" is exclusive with "cs2==0", all cases can be executed in the codes at the left side of figure 7, but not in the code at right side. This occurrence should be prevented by disabling the option for switch optimization:

-output_style_merge_case=off

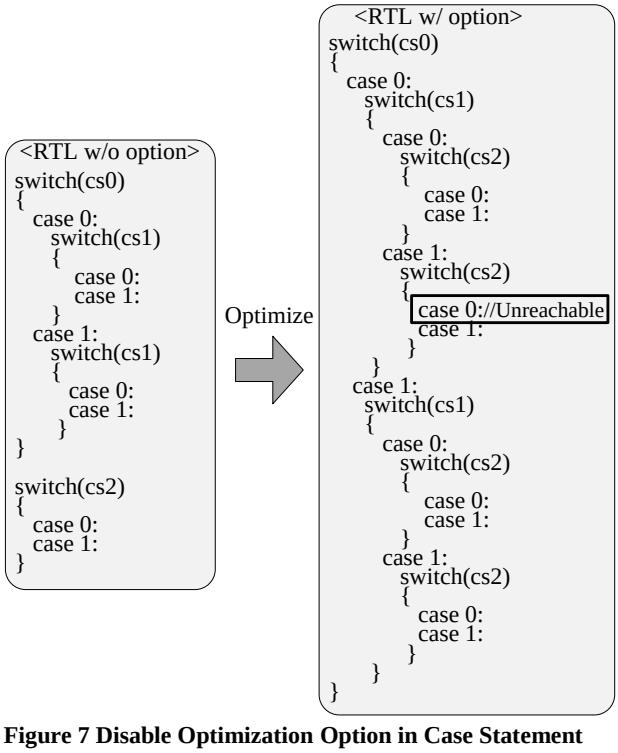

Figure 7 Disable Optimization Option in Case Statement

Therefore, the high level design guides to minimize the coverage gap are summarized as follows

- Guide1: Recommend to keep the number of wait statements not to exceed the loop count number to avoid unreachable code generation
- Guide2: Use the STARC option to change the RTL output code style for better code coverage result
- Guide3: Remove the switch optimization option to avoid undesirable merge of case statements

Theoretically, there might be no substantial area/timing overhead because the given coding guideline does not change the number of registers and the number of data-path elements of given design.

However, there is a possibility of area overhead due to the guide3 because it disables possible code optimization chances for better logic synthesis results [1]. In our experiments, the area overhead was under 1%.

4. Case Study: Development of Scaler IP

As a device-under-test, a Scaler IP was used. Figure 8 shows a block diagram of the IP. A scaler consists of scaling functions, control block, and memories [6]. Operation speed constraint is 200MHz.

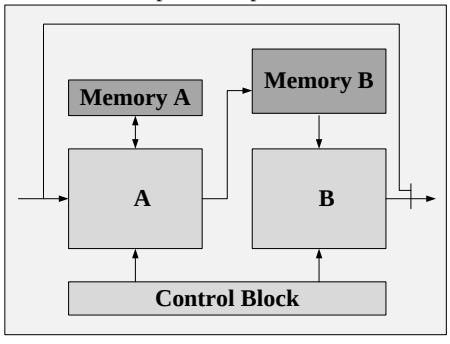

Figure 8 Block Diagram of the DUT

Chapter 3 has shown three high level design guides which can effectively control the RTL generation process to minimize the coverage gap, which is the difference of saturated code coverage values at behavioral level and RTL designs. As we applied these three design guides one by one, the coverage value showed gradual increase as shown in table 2. This means that the design guides are very effective in decreasing hardware redundancy in generated RTL.

Table 1 Coverage Enhancement According to HL Guide

HL Design Guide	RTL Code Coverage			
	Block		After	
	Before	After	Before	After
HL Constraints	99% (4159/4188)	99% (4160/4188)	96% (1549/1614)	96% (1551/1614)
RTL Output Style	99% (4160/4188)	99% (4166/4188)	96% (1551/1614)	98% (1582/1614)
Optimization Options	99% (4166/4572)	100% (4188/4188)	98% (1582/1614)	99% (1598/1614)

We tested behavioral and RTL designs with common testbench and DUT that all the design guides have applied to see the correlation of RTL and behavioral code coverage behavior. Table 2 shows the behavioral and RTL code coverage snap-shots as the number of input image frames increases. Behavioral code coverage value is lower than that of RTL design in the beginning, but it increases in proportion to the increase of stimulus inputs and finally behavioral code coverage result converged to the same value as shown in figure 9. This result shows that with fairly good testbench, we can get the same coverage results in both behavioral level and RTL designs.

Table 2 Analysis of Behavioral and RT Level Code Coverage

Test Input	HL code coverage		RTL code coverage	
	Line	Expression	Block	Expression
1 frame	61%	41%	82%	89%
5 frame	93%	87%	100%	98%
10 frame	97%	92%	100%	99%
15 frame	100%	96%	100%	99%
20 frame	100%	99%	100%	99%

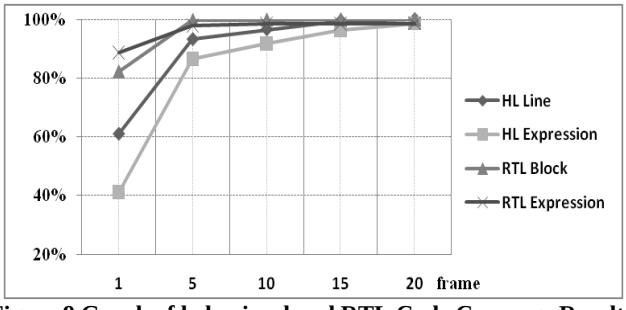

Figure 9 Graph of behavioral and RTL Code Coverage Results

Even though we have to perform sequential equivalence check in the proposed code refinement flow, overall verification time can be reduced since RTL simulation, the most time consuming part, is omitted. Figure 10 shows the difference of elapsed time for each abstraction level. Code refinement time is a sum of behavioral functional verification and sequential equivalence checking, and it is smaller than that of RTL verification by over 70%.

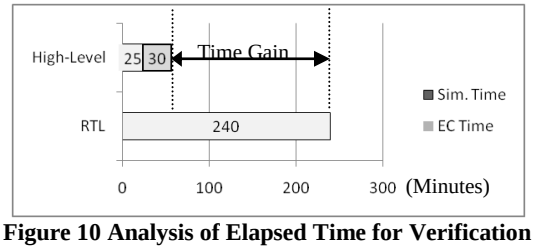

Figure 10 Analysis of Elapsed Time for Verification

5. Conclusion

In this work, we described our experience to move code refinement from RTL to behavioral level. The coverage gap of the saturated code coverage value between behavioral level and RTL designs can be minimized by the high level design guides which we developed through our experience, which means that we can successfully finish code refinement early at behavioral level design. We also showed a full work flow for behavioral level design which includes sequential equivalence checker to back up hardware compatibility. Our experiments showed that it is possible to reduce code refinement time by over 70% since we do not have to perform iterative high-level synthesis and RT-level simulation flow.

6. References

[1] Forte, "Cynthesizer User's Guide", Product Version 4.0
[2] John Sanguinetti et al, "The Relationship of Code Coverage Metrics on High-level and RTL Code", IEEE International High-level Design Validation and Test Workshop, 2010.
[3] Calypto, "Sequential Logical Equivalence Check(SLEC) User Manual", Product Version 5.0
[4] JEDA, "JEDAcc User Manual", Product Version 1.3
[5] Cadence, "ICC User Guide", Product Version 9.2
[6] Chun-Ho Kim et al, "Winscale: An Image-Scaling Algorithm Using an Area Pixel Model", IEEE Transactions on Circuits and Systems for Video Technology, VOL. 13, NO. 6, 2003