Testpoint Synthesis Using Symbolic Simulation

Kai-Hui Chang, Yen-Ting Liu and Chris Browy
Avery Design Systems, Inc., Tewksbury, MA, USA
978-851-3627
changkh@avery-design.com, pheonix@avery-design.com, cbrowy@avery-design.com

Abstract

Testpoints used in Register Transfer Level (RTL) verification may be optimized away during logic synthesis, preventing testbenches from functioning properly in gate-level simulation. In this paper we propose a testpoint synthesis technique using symbolic simulation that produces more efficient driving equations compared with methods based on logic synthesis. Additionally, we propose a methodology that allows RTL testbenches to be used directly for gate-level simulation.

I. Introduction

Testpoints are design variables accessed by the testbench to provide important information for verification. Testpoints can be either registers or wires in the RTL code. After logic synthesis, registers are mostly preserved, but wires can be optimized away. The loss of testpoints at the gate level impacts testbench effectiveness and makes debugging more difficult.

Even though synthesis tools can preserve wires that are testpoints, doing so prevents full logic optimization and produces sub-optimal netlists. Given that most testpoints are used only for verification, this consequence is undesirable. To address this problem, one solution is to run the RTL version of the block along with the gate-level netlist and update RTL registers based on gate-level Flip-Flop (FF) values to reconstruct testpoint values at the RTL. However, connecting all the ports and updating RTL registers correctly require considerable effort, and timing issues between RTL and the netlist can cause racing conditions that are difficult to resolve. Another solution is to run logic synthesis on the same block with testpoints preserved to produce a new block and then connect inputs from the original block to the new block. The new block is for testpoints only and is not part of the design. The disadvantage of this method is that registers in the new block may have different values compared with the original block due to physical optimizations or tool errors, which may allow bugs to escape and make debugging more difficult. In addition, simulating another netlist creates considerable simulation overhead, which can slow down gate-level simulation.

In this paper we propose a new technique that uses symbolic simulation to recreate testpoints at the gate level and a new methodology that makes RTL testbenches easily portable to the gate level. Symbolic simulation generates word-level equations for reconstructing testpoints, which can significantly reduce simulation overhead compared with testpoint reconstruction methods based on logic synthesis. The methodology also allows RTL testbench to be ported into gate-level with minimal effort. By enabling the same testbench for both RTL and gate-level, debugging effort can be greatly reduced.

The rest of the paper is organized as follows. In Section II we provide necessary background for understanding this work and review related work. Section III describes our proposed solution and Section IV provides discussions on issues we encountered due to synthesis optimizations. In Section V we provide some case studies on how our methodology is applied to solve real problems, and Section VI concludes this paper.

II. Background and Related Work

A. Symbolic Simulation

Symbolic simulation is a technique that simulates symbols instead of scalar values [4, 5, 8]. A symbol represents all possible values that a variable can have, thus symbolic traces generated from symbolic simulation can comprehensively represent full design behavior.

For example, for a logic expression "$a + b$", symbolic simulation produces a symbolic trace "$a_s + b_s$" (subscript s denotes the symbol used to represent the value in a variable). In contrast, logic simulation produces a scalar value based on the values in registers *a* and *b*. For example, if *a* is 1 and *b* is 3, then 4 is produced. Symbolic simulation preserves word-level operations that are suitable for testpoint reconstruction.

B. Related Work

In [2] the authors proposed several techniques for identifying correlations between RTL and gate-level variables. They use naming similarity, structural similarity and functional similarity to achieve this goal. Their empirical evaluation showed good results on several designs. However, their techniques do not create signals that are optimized away in netlists. Therefore, their solution is more suitable for manual debugging than automated testpoint reconstruction for testbenches.

There are tools and research that perform signal reconstruction based on limited information, and such techniques are related to this work. In academia, dominators have been shown to be useful for reconstructing logic values [3]. In industry, Siloti from Synopsys [9] can reconstruct signal values from a minimal set of recorded signals and correlate gate-level results to the RTL source code. However, it does not regenerate signals that no longer exist in the gate-level netlist as testpoints. On the other hand, the techniques proposed in this paper may benefit from such tools to further reduce the complexity of generated driving equations.

Even if a signal is optimized away, related signal can be used to reconstruct the desired signal with additional logic. Finding minimal additional logic to reconstruct a signal is highly related to the Engineering Change Order (ECO) problem. For example, Chang *et al.* proposed Distinguishing-Power Search (DPS) and Goal-Directed Search (GDS) techniques to generate a new signal [1]. Yang *et al.* proposed the use of approximate Sets of Pairs of Functions to be Distinguished (SPFDs) to generate such new signals [7]. Smita *et al.* also proposed a solution called DeltaSyn that considers both logical and physical information [6]. Such ECO techniques can reconstruct testpoints with minimal additional logic and are also suitable for testbench reconstruction. However, ECO problems are difficult and can be time-consuming to solve, which may be over-killing for the purpose of generating testpoints for verification.

III. PROPOSED SOLUTION

In our proposed solution we strive to handle testpoint reconnection for gate-level simulation with minor changes to the testbench. To achieve this goal, we propose a new methodology. In this section we describe our proposed methodology in detail and provide some discussions.

A. Prerequisites

The prerequisites of the methodology are:

- All global accesses must be written as ‘macro.variable so that the hierarchy can be easily changed at the gate level.
- A table is provided that contains the following information: a macro name, an RTL module name whose boundary is preserved at the gate level, an additional hierarchy under module, a gate-level hierarchy and a gate-level module name. We need this information because certain hierarchies may be flattened at the gate level and knowing which hierarchy is still preserved is important for finding RTL signals that still exist at the gate-level.
- A list of testpoint variables that need to be reconstructed from the gate-level netlist.

To identify testpoint variables that need to be reconstructed, originally we analyze compilation errors from simulators when compiling the testbench with the gate-level netlist. However, we found that certain variables may still exist in the netlist, such as inputs to boundary-preserved blocks. In this case, no errors will be produced for those variables, and their corresponding testpoints will not be reconstructed in the reconnection modules. Therefore, we now always analyze Verilog source files to identify hierarchical references as testpoints.

B. Proposed Methodology

Given the above information, our methodology works as follows:

1. For each macro, create a reconnection module and declare testpoints in the module.
2. Read the RTL block that the macro points to and synthesize testpoints based on registers and primary inputs of the closest hierarchy-preserved block using symbolic simulation.
3. Read the netlist and map RTL primary inputs as well as registers driving the symbolic traces to signals in the netlist.
4. In the reconnection modules, write the symbolic traces and use them to drive their corresponding testpoints.

Step 2 is performed by injecting a symbol to each clocked register and each input port to represent registers and ports. And then symbolic simulation is performed for one cycle to produce symbolic traces for the testpoints. The symbolic traces for the testpoints are the driving equations of the testpoints. Note that type “reg” in Verilog may be synthesized into wires and they are considered “wires” instead of “registers” in this work.

One issue for Step 2 is that there can be behavioral models used in a design such as memories or phase lock loops that are difficult for symbolic simulation to handle. Given that the same models are typically used for both RTL and gate-level netlists, those models can be blackboxed for the purpose of testpoint reconstruction because their ports are preserved in netlists. To handle blackboxed modules, symbols should be injected to the outputs of those modules similar to how the primary inputs of the design are handled. Symbolic simulation can then correctly reconstruct testpoints using values from outputs of the blackboxed modules.

The advantage of using symbolic simulation instead of logic synthesis for testpoint synthesis is that word-level expressions are preserved in symbolic simulation, creating much simpler reconnection equations that can considerably reduce simulation overhead. An example of testpoint synthesis is shown in Figure 1. Note that in the example, symbolic simulation preserved the use of addition in the reconnection module. If logic synthesis were to be used to generate the reconnection module, a two-bit adder composed of several gates would be required.

Testbench:	$display(‘hier1.a);		
(a)	**Original RTL code**	(b)	**Gate-level testpoint reconnection module**
	module bl; (at dut.b1)		module tbconn_hier1;
	reg [1:0] a, b, c;		wire [1:0] a;
			assign a = {dut.b1.b_1.q, dut.b1.b_0.q} + {dut.b1.c_1.q,
	always @*		dut.b1.c_0.q};
	a = b + c;		endmodule
	endmodule		
Macro	‘define hier1 dut.b1		‘define hier1 tbconn_hier1

Fig. 1. Testpoint synthesis example. Assume *‘hier1.a* is accessed from the testbench. Variable *b* and *c* are registers, and *a* is a wire after synthesis. (a) shows the original RTL code, and (b) shows the reconnection module generated by testpoint synthesis. By changing macro *hier1* from *dut.b1* to *tbconn_hier1*, an RTL testbench can be easily ported to the gate-level.

The overall flow for our methodology is shown in Figure 2. Given design RTL, we perform testpoint reconstruction using symbolic simulation to obtain driving equations of testpoints. For the gate-level netlist, we extract its FFs and primary inputs as potential signal drivers and save them into an object database. A mapping Perl script is then used to connect testpoint driving signals to gate-level signal drivers. Reconnection modules are then produced as output. If any driving signal cannot be found, mapping errors are produced and manual reconstruction is required for those signals.

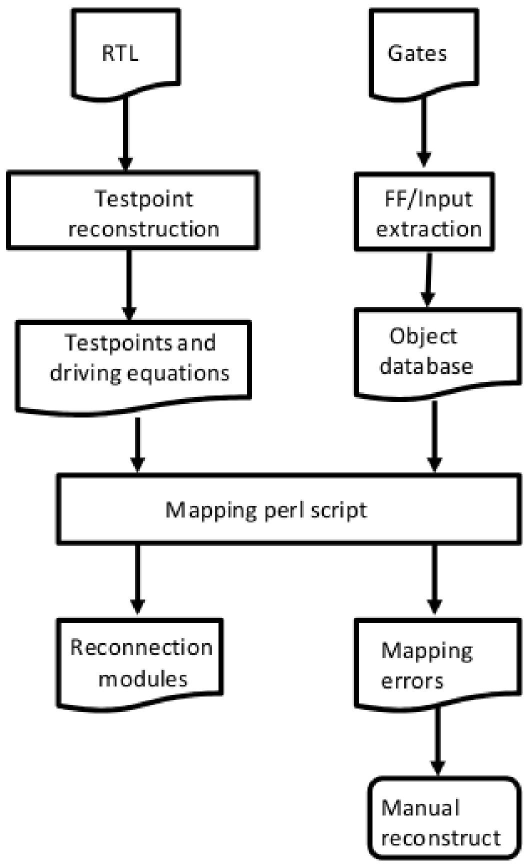

Fig. 2. Overall testpoint reconstruction flow.

C. Discussions

Currently, the mapping Perl script is implemented by manually analyzing synthesized netlists to identify naming conventions that allow us to map RTL variables to their corresponding gate-level FFs. For example, "top.dut.a.b.c" may become "top.dut.a_b.c_reg.q", and the rule for such mapping can be: (1) regular expression matching where "." can be replaced with "_", (2) add "_reg" to the RTL variable, and then (3) add ".q" to the variable. These rules are then encoded in the Perl script to perform the mapping. Since the rules may change when synthesis optimizations or options change, the rules may need to be revised for each new design. However, after several iterations on multiple designs most conventions will be identified and the script will be able to handle most new designs without additional changes.

Even though the generated testpoints at the gate level should be functionally equivalent to those at the RTL, there can be timing or event order differences. For zero-delay simulation, testpoints at the RTL and the gate level should match perfectly unless there is racing condition due to event ordering. For the case of TARMAC, the result is that information may be shown in a different order, but all the shown messages will still appear at the same time. For unit-delay or SDF-annotated simulation, sampling the gate-level testpoints at incorrect time can produce incorrect results. To address this problem, sampling of testpoint values should be aligned to the clocks so that only stable values are sampled.

Occasionally there are cases where the testbench has to include part of the design hierarchy when accessing a design variable. For example, a variable may be accessed using "'macro.hier1.hier2.var" instead of the recommended "'macro.var". To address this problem, when creating reconnection modules we add additional hierarchies so that the testpoint can be found. Figure 3 shows an example.

```
Macro: ‘macro.hier1.hier2.var
module tbconn_macro;
wire hier1_hier2_var= driving equations;
assign hier1.hier2.var= hier1_hier2_var;
dummy1 hier1();
endmodule
module dummy1;
dummy2 hier2();
endmodule
module dummy2;
wire var;
endmodule
```

Fig. 3. An example to show how additional hierarchies are added in the reconnection modules.

IV. Handling Synthesis Optimizations

Our proposed methodology decomposes testpoints based on FFs and primary inputs of boundary preserved blocks. This method works well if FFs and primary inputs are completely preserved after logic synthesis. In practice, however, synthesis tools can perform optimizations that change FFs or primary inputs. For example, FFs may be inverted to remove an inverter at its input or output, and ports may become unconnected with real signals going through a different port. Such problems are especially serious when physical synthesis tools are applied to improve physical design quality: changes made by physical design tools can be irregular and hard to predict. During our collaboration with our industrial partners we encountered several issues caused by synthesis optimizations. In this section we describe what the issues were and how we solved the problems.

A. Inverted FFs

Inverted FFs are quite common because they can reduce the use of inverters, thus improving circuit timing and area. We found that occasionally ”_inv” is added to FFs to indicate that the FF has been inverted. However, this is not always the case. To identify FFs that are inverted, our solution is to compare RTL and gate-level waveforms generated from running the same test at reset deassertion time. If the polarities of values are different for any FF, we know the FF is inverted. When we generate testpoint reconnection logic for inverted FFs, we add a negation operation to the FF’s output to correct its polarity.

B. Unconnected Inputs

Occasionally an input may be optimized away and the signal will propagate using a different port. This can be detected by checking for “Z” values at ports using gate-level waveforms. To reconstruct unconnected ports, one way is to decompose the block at a higher-level hierarchy where the driver of the port is preserved as either a FF or a port. Another method is to search for ports of the block with similar names and then compare RTL and gate-level waveforms to make sure they toggle the same way at similar time. Most of the time the port that replaces the original input can be identified this way.

C. Discussions

It is important to note that the methods described in this section are heuristics that work well in practice but are not formal proofs. However, without additional input from synthesis tools, it is extremely difficult to reconstruct the complete testpoint logic without any heuristic. If a signal driver still cannot be found despite all the heuristic efforts (such as missing FFs), we drive the signal with X. In this way, if the FF is indeed used by the testpoint, then the X will propagate to the testpoint and the problem can be detected immediately. To this end, because we write driving equations of reconstructed testpoints using continuous assignments, X-optimism can never occur and the X will affect testpoints only if the signal is used. Therefore, if simulation runs fine without any X being sampled at the testpoints, the missing signal drivers can be ignored.

If Xs are detected at testpoints, manual effort may be required to identify the missing signals and the reconnection may need to be performed manually. In our experience, our proposed flow can handle most industrial designs without problem. For a small number of designs where Xs are still being detected, only one or two signals need to be manually reconstructed. By automating most of the testpoint reconstruction task, engineers’ time can be greatly saved.

V. Case Studies

We applied the methodology to ARM's TARMAC trace plug-in environment on three different wireless communication blocks so that TARMAC can also be used in gate-level simulation. TARMAC accesses variables within RTL blocks and several variables cannot be found in the gate-level netlist. Setting up the tool and flow took approximately a day, mostly spent on building the filelist and constructing the macro table. For each design minor changes were needed to revise the mapping Perl script. For example, for the last design multi-bit FFs were used, and the script needs to be changed so that they can be properly handled. Runtime of the flow is approximately an hour. Symbolic simulation took about one third of the time and the mapping Perl script took the rest of the time due to the large number of FFs that need to be searched for a matching. The generated reconnection modules allow TARMAC to be used in gate-level simulation as well, which significantly facilitates gate-level debugging.

To show the advantage of testpoint synthesis using symbolic simulation, we generated reconnection modules for one of the three blocks using symbolic simulation and logic synthesis, and then we compared their complexity. The former has approximately 10K word-level operations and the latter has 600K Boolean operations, suggesting much better performance when simulation is performed using results from symbolic simulation instead of logic synthesis for testpoint reconstruction.

In one of the cases the testpoints are reconstructed at block-level and the results are used in both block-level and system-level testing. One complication for reusing the results for system-level testing is that the same core may be instantiated more than once. This is not a serious issue for our methodology because the cores are identical. Therefore, the generated reconnection modules can be reused directly. And as long as the macros are defined using relative path instead of absolute path, logic simulators should also be able to find the correct signal sources for the reconnection modules.

VI. Conclusions

In this work we presented a methodology that uses symbolic simulation to reconstruct testpoints from gate-level netlists so that the same testbench can be ported from the RTL to gate level with minimal changes. Symbolic simulation preserves word-level operations. Therefore, driving equations of the reconstructed testpoints are considerably simpler than methods based on logic synthesis and can significantly reduce simulation overhead. Our case studies on TARMAC show how the methodology can be successfully applied to industrial designs. The same methodology can be applied to any testpoint that probes RTL signals, not limited to the TARMAC interface.

References

[1] K.-H. Chang, I. L. Markov, V. Bertacco, "Fixing Design Errors with Counterexamples and Resynthesis," *IEEE Trans. on Computer-Aided Design*, Jan. 2008, pp. 184-188.
[2] E. Chueng, X. Chen, F. Tsai, Y.-C. Hsu and H. Hsieh, "Bridging RTL and Gate: Correlating Different Levels of Abstraction for Design Debugging", *HLDVT'07*, pp. 73-80
[3] N. Jha and S. Gupta, "Testing of Digital Systems", Cambridge University Press, 2003.
[4] A. Kolbl, J. Kukula and R. Damiano, "Symbolic RTL Simulation," *DAC'01*, pp. 47-52.
[5] A. Kolbl, J. Kukula, K. Antreich and R. Damiano, "Handling Special Constructs in Symbolic Simulation," *DAC'02*, pp. 105-110.
[6] S. Krishnaswamy, H. Ren, N. Modi and R. Puri, "DeltaSyn: an Efficient Logic Difference Optimizer for ECO Synthesis," *ICCAD'09*, p. 789-796.
[7] Y.S.Yang, S.Sinha, A.Veneris and R.K.Brayton, "Automating Logic Transformations with Approximate SPFDs," *IEEE Trans. on Computer-Aided Design*, May 2011, pp. 651-664.
[8] Avery Design Systems Inc., http://www.avery-design.com
[9] Synopsys Inc., http://www.synopsys.com