Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 174 (2007) 9–18

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Error Diagnosis in Equivalence Checking of High Performance Microprocessors

Alper Sen[1]

Verification Tools Research and Development, Design Technology Organization
Freescale Semiconductor Inc.,
Austin, TX, USA

Abstract

We describe techniques for diagnosing errors in formal equivalence checking of RTL and transistor level models of high performance microprocessors at Freescale Semiconductor Inc. We use Symbolic Trajectory based Evalaution (STE) for combinational equivalence checking. STE accurately captures transistor level behaviors. We use simulation based error diagnosis techniques and present a seamless integration of them in our current verification environments.

Keywords: equivalence checking, symbolic trajectory evaluation, error diagnosis, simulation

1 Introduction

Verification and debugging process consume more than half of the design cycle of a microprocessor. This process involves various formal and non-formal techniques such as formal property checking, assertion based verification, runtime verification, equivalence checking, and simulation. Most techniques automatically generate a counter-example when an error exists in the design but do not locate (diagnose) the error. Manual error diagnosis is a time consuming effort and with the ever increasing complexity of designs, diagnosis is becoming more and more challenging. Several automatic error diagnosis techniques have been developed for formal property verification, equivalence checking and manufacturing fault detection.

We describe techniques for diagnosing errors in formal equivalence checking of RTL and transistor level models of high performance microprocessors at Freescale Semiconductor Inc. Most of the previous approaches in error diagnosis of microprocessors use gate level models rather than transistor level models. However, for timing, power and other performance issues, some circuits are custom built, that is,

[1] Email: alper.sen@freescale.com

doi:10.1016/j.entcs.2006.12.026

they are manually implemented at transistor level using such artifacts as self-timed components, static and dynamic logics. These artifacts and others prevent using boolean model extraction tools on transistor level models and producing correct gate level models. Our transistor level model captures transistor level dynamic behavior. It takes into account bidirectional transistors, charge sharing and different transistor strengths. Due to the low level implementation details of transistor level model, error diagnosis becomes even more challenging since now signal timing and transistor strengths may also cause errors.

Our equivalence checker uses Symbolic Trajectory Evaluation (STE) for checking equivalence between RTL and transistor level models [3]. STE [5] is a powerful verification technique based on symbolic ternary simulation using 0, 1, and unknown value "X". The verification properties in STE are simple assertions in the form of "antecedent implies consequent", where antecedent and consequent are tuples denoting the value of a circuit element at a specific time. Our in-house checker is unique in that it can automatically generate all assertions for equivalence checking from RTL model. Hence, it eliminates potential errors that may result from manually generating such assertions. Verification proceeds by symbolically simulating the transistor level model using the values in the antecedent and then checking the value of a given comparison point with its expected value in the consequent.

Whenever an assertion fails, the tool automatically generates a counter-example from error BDD in the form of a VCD file, which includes circuit elements in the cone of influence of the comparison point. These failures can be categorized as weak and strong fails. A weak fail occurs due to the abstraction of parts of the circuit using "X" values, which then get propagated to the comparison point. A strong fail occurs when binary value of the comparison point in the consequent is different from binary value of it in the simulation. These types of fails are more difficult to diagnose than weak fails since there could be many more reasons for the error resulting from errors in comparison point mapping, wire connection, design element, inversion parity, control logic, and transistor strength.

We have used several techniques for analyzing each type of failure. These techniques include a combination of error vector simulation, backward and forward propagation of values. We do not assume a specific error model as in previous simulation based error diagnosis techniques, hence our techniques are applicable regardless of the error types. Experimental results on latest microprocessor designs demonstrate the effectiveness of our approach. In the following sections, we present a background on Equivalence Checking, Symbolic Trajectory Evaluation and then present error diagnosis algorithms for weak and strong fails.

2 Error Diagnosis in Equivalence Checking

Equivalence Checking (EC) formally proves that two representations of a design (specification and implementation) exhibit exactly the same behavior. Due to the hierarchical nature of hardware development, the same design has representations at different abstraction levels. Designers are used to checking equivalence of different representations using simulation. Hence, equivalence checking easily fits in current verification methodologies and requires little input from the designer.

Our goal is to apply Error Diagnosis in small steps. Hence, we chose to work with Equivalence Checking first, among other formal methods. We check equivalence of RTL (specification) and Transistor Level (implementation) models using a Combinational Equivalence Checker (CEC). In CEC, combinational blocks separated by sequential nodes are compared and a map between sequential nodes (comparison points) in specification and implementation is given by the user. When there is a mismatch between models, a counterexample is generated automatically. So, why do we have counterexamples during equivalence checking? Some of the reasons are lsited below.

- Implementation errors. There could be extra or missing circuit elements, wire connections, timing related errors, transistor errors in the implementation or logic blocks may be different from the specification.
- Abstraction. Since designs are complex, we generally use abstraction techniques during EC. It is possible to abstract the implementation model more than necessary such that we obtain false negatives.
- Mapping. User given maps for CEC may be erroneous.
- Constraints are routinely used to restrict the environment of designs for comparison. For example, functional and scan clocks should not be on. It is possible to have wrong or missing constraints.
- Specification may be erroneous and hence may need to be changed.

Our goal is to find possible causes or locations of the EC error by analysis of the counterexamples. This is the so-called “Simulation Based Error Diagnosis”. This is different than Error Correction, where the goal is to diagnose and then repair the implementation so that the equivalence is established. Also, there are other approaches for error diagnosis such as SAT-based, structural based, and trace based.

Simulation based error diagnosis provide a scalable and fast solution, and it can seamlessly be integrated into the current verification frameworks. This is because it works on counterexamples rather than modifications or static analysis of the designs.

However, Simulation based error diagnosis has traditionally been used for gate level designs with zero delay models rather than transistor level models as in our case. Our custom built transistor level implementations have self-timed, precharged, dynamic logic with complex clocking. Figure 1 displays a simple array column with complex signal timing. In such complex models, transistor strengths also impact the functionality of designs.

Static analysis of such custom-built transistor level implementations is not accurate. This is because for structures such as self-timed logic, e.g. write-enable signal that is used to control access to the array column, there is no equivalent boolean function. Consider the example in Figure 2, where the boolean function at *out* is zero, whereas, in fact, *out* is a pulsed waveform. Such self-timed behavior

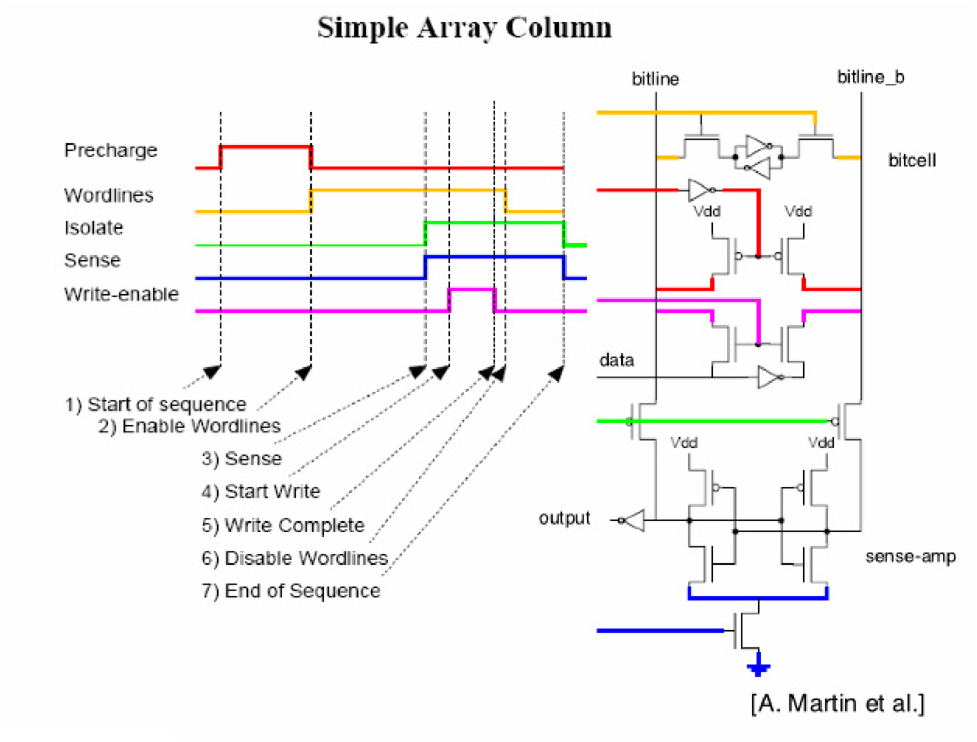


Fig. 1. Simple Array Column

can be captured using non-zero delay simulation. Our EC tool uses a unit-delay switch level model that handles such complex transistor level circuits. Note that upon using a non-zero delay model, counterexamples of combinational equivalence checking have sequences of signal values rather than a single set of signal values. This is similar to counterexamples in sequential equivalence checking.

3 Symbolic Trajectory Evaluation

We can check equivalence between a specification and an implementation using simulation. We present three types of simulation: scalar, ternary and symbolic simulation. For example, we can check equivalence of an n input NAND gate using assertions (input vectors) as in Figure 3. Scalar simulation uses binary values 0 and 1 during simulation and we need 2^n assertions. Ternary simulation uses 0, 1, and X during simulation, where X denotes an undefined value. For example, when one of the inputs is 0, the output of NAND gate will be 0 regardless of other inputs. Hence, we need $n+1$ assertions. Symbolic simulation uses symbolic variables during

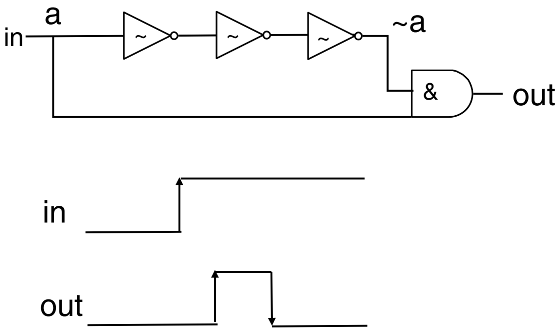
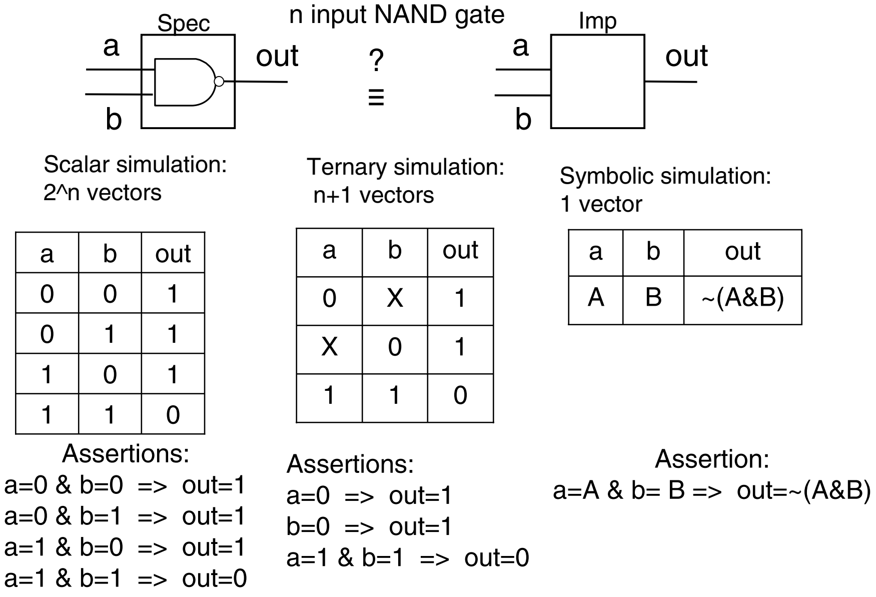


Fig. 2. Self-timed Custom Logic

simulation. We need a single symbolic assertion to check for equivalence.

Symbolic Trajectory Evaluation (STE) is a formal verification technique that combines symbolic simulation with ternary simulation. STE uses 0, 1, and X for simulation, where a partial order relation is defined between these symbols such that X is weaker than 0 and 1, but 0 and 1 are not comparable. Specifications in STE are simple assertions *antecedent* $\Rightarrow$ *consequent*, where antecedent and consequent are boolean expressions on circuit nodes and the temporal operator is next-time. An antecedent defines how to initialize the implementation nodes and the consequent defines the expected values for comparison points. STE uses a powerful yet simple abstraction by using X for unknown variables. This effectively reduces the state space of the design to the variables interesting for current equivalence purpose. Another powerful abstraction technique in STE is symbolic indexing, where instead of using n symbolic variables, we can use $log\ n$ symbolic variables for certain cases.

STE is useful for custom logic structure analysis since it uses simulation. However, we need to be careful in using abstraction and symbolic variables in STE, otherwise we may obtain false negatives or suffer from state explosion, respectively. Our particular Equivalence Checker uses STE and is unique in that assertions are generated automatically from RTL models [6] and then these assertions are mapped to the implementation nodes. We also have a semi-automatic mapping of specification and implementation nodes [1]. An STE algorithm consists of 3 steps:



Scalar simulation:
2^n vectors

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

Assertions:
a=0 & b=0 => out=1
a=0 & b=1 => out=1
a=1 & b=0 => out=1
a=1 & b=1 => out=0

Ternary simulation:
n+1 vectors

a	b	out
0	X	1
X	0	1
1	1	0

Assertions:
a=0 => out=1
b=0 => out=1
a=1 & b=1 => out=0

Symbolic simulation:
1 vector

a	b	out
A	B	~(A&B)

Assertion:
a=A & b= B => out=~(A&B)

Fig. 3. Equivalence Checking using Simulation

(i) Initialize the circuit nodes using the values from the *Antecedent*.

(ii) Simulate the circuit to obtain *Trajectory*.

(iii) Check if the *Consequent* is weaker than the Trajectory value of the comparison node, if not, return a counterexample.

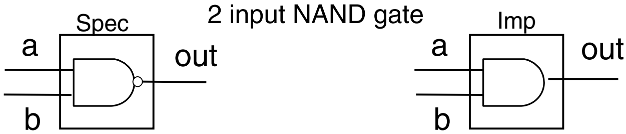
Figure 4 demonstrates STE algorithm on a simple example where the specification is a 2-input NAND gate and the implementation is a 2-input AND gate. In this case, there is a counterexample when $a = 0$ and $b = 1$ at Time 2. Note that the time steps for variables are denoted in brackets in the assertions.

There are two types of counter examples in an STE-based equivalence checker.

(i) Weak Fail: This occurs when Simulation (Trajectory) value of the comparison node is X, whereas the Consequent value is 0 or 1.

(ii) Strong Fail: This occurs when Simulation (Trajectory) value of the comparison node is 0, whereas the Consequent value is 1 and vice versa.

Next, we discuss techniques for Weak and Strong Fail Diagnosis.



Assertion:

a[1]=A & b[1]= B → out[2]= ~(A&B)

	Time 1 (a, b, out)	Time 2 (a, b, out)
Antecedent	(A, B, X)	(X, X, X)
Trajectory	(A, B, X)	(X, X, (A&B))
Consequent	(X, X, X)	(X, X, ~(A&B))

Fig. 4. STE Equivalence Checking

4 Weak Fail Diagnosis

We now list some of the reasons for this type of error and how to diagnose and also correct those.

- Some of the input nodes in the cone of logic for the comparison point may have been abstracted to X, hence we need abstraction refinement. We accomplish this by traversing the fanin cone of the X node and driving (giving symbolic variables) to nodes on the path that are of interest.
- Some of the input nodes in the cone of logic for the comparison point may not have been driven at correct times.
- When more than one path is enabled on a bus node, where one path is pulling it to low and the other to high, results in X at the bus node. This is called bus contention. We ask the user to anlayze the situation and possibly add constraints to eliminate bus contention.
- We may need to fix timing of some of the nodes in the circuit such as the sense amp enable signal and the write enable signals should arrive in particular order, otherwise an X may propagate into the bitcells.
- We may need to fix transistor strengths where two strong transistors try to drive the same node, where in reality one of those transistors should have been a weak

transistor.

5 Strong Fail Diagnosis

We use a combination of Simulation-based Error Diagnosis techniques to diagnose nodes. It is important that our approach have the following characteristics:

(i) Small number of error locations (diagnosis nodes).

(ii) Fast.

(iii) Contains actual error locations.

Next, we describe techniques satisfying the above requirements. These are Path Backtrace [4] and Complementation [2] techniques.

5.1 Path Backtrace

This technique traverses the design backwards using controlling variables of a logic function. We say that a support node a of a logic function F is *controlling* if complementing the value of a changes the value of F. Note that since we are using a unit delay model, we need to use the values of nodes at appropriate times. We next give an outline of the Path Backtrace algorithm.

```
PathBacktrace(nd):
 Input: Comparison node, counter example, implementation
 Output: Diagnosis nodes in Controlling list

 If nd is input or is already processed then return
 Compute excitation function F for nd from transistor level
 for every node x in support of F
  if x is controlling then add x to Controlling list L,
    if not already added
 end for
 if L=[] then PathBacktrace(y) for all nodes y in support of F
 else choose a node y in L and PathBacktrace(y)
```

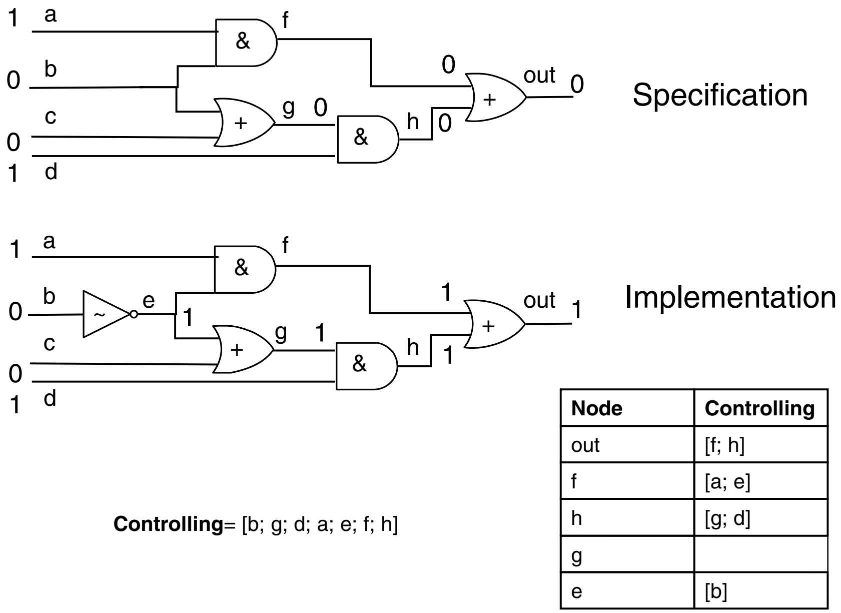
Figure 5 shows a specification and its implementation. The implementation has an extra inverter gate. For the sake of simplicity, we use a counterexample with a single input vector (rather than a sequence), namely, $a = 1, b = 0, c = 0, d = 1$. After running the algorithm, the list of controlling nodes is $L = [b; g; d; a; e; f; h]$.

However, we can do better by using multiple counterexamples. For example, if we also used counterexample $a = 1, b = 1, c = 0, d = 1$, we would get $L = [b; c; g; e; f; h]$. Furthermore, since there is a single error, we can take the intersection of all lists and obtain $L = [b; e; g; h; f]$, which contains the error location, $[b; e]$.

5.2 Complementation

This technique uses scalar forward simulation of design. It uses the list of nodes generated by Path Backtrace, then for each node checks if changing (complementing) the value of the node is observable at the output. We next give an outline of the Complementation algorithm.

```
Complementation(L):
 Input: List of nodes, counter example, implementation
 Output: Diagnosis nodes in Complementation list
```



Node	Controlling
out	[f; h]
f	[a; e]
h	[g; d]
g	
e	[b]

Controlling= [b; g; d; a; e; f; h]

Fig. 5. Application of Path Backtrace Algorithm

```
for each node y in list L
  Complement the value of y;
  Simulate the implementation;
  Add y to Complementation list, C, if the output value changes
```

If Complementation algorithm is applied to the implementation in Figure 5 with list $L = [b; e; g; h; f]$ then we obtain the list $C = [b; e]$. We can further improve this algorithm by exploiting the *topological dominance* between nodes in every fanout free region of the design. We say that a node x is dominated by another node y, if every path that starts from x and ends in the output node goes through a node y. If y does not belong to the complementation list, that is, complementing y does not change the output value, then x does not belong to the complementation list either. For example, g is dominated by h in Figure 5, hence we obtain a faster algorithm.

In summary, for Strong Fail Diagnosis, we use Path Backtrace with Multiple Counterexamples, followed by Complementation with Dominance relation. Our experimental results demonstrate that both Weak Fail and Strong Fail diagnosis algorithms are easy to use in current verfication methodologies. Furthermore, they help in reducing the debugging time.

6 Conclusion

We presented a domain specific application of error diagnosis techniques. We diagnose errors generated during equivalence checking of RTL and custom-built transistor level models of high performance microprocessors. Our techniques are applied using a simulation based formal verification technique called Symbolic Trajectory Evaluation, which accurately captures transistor level behavior. We incorporate Simulation based Error Diagnosis approaches which are seamlessly and efficiently integrated in our current verification environment. In particular, our techniques are considerably fast, the number of error locations is small and contain actual error locations. Overall, user feedbacks indicate that our error diagnosis approach plays an important role on reducing the verification and debugging time, ultimately increasing quality, robustness and performance of microprocessors. We believe that error diagnosis has a lot of potential to be successful in an industrial setting due to its immediate impact. As a future work, we plan to apply error diagnosis techniques for functional verification.

References

[1] H. Anand, J. Bhadra, A. Sen, M. S. Abadir, and K. G. Davis. Establishing Latch Correspondence for Embedded Circuits of PowerPC Microprocessors. In *Proceedings of IEEE High-Level Design Validation and Test Workshop (HLDVT)*, pages 37–44, 2005.

[2] S. Huang, K-C. Chen, and K-T. Cheng. Error Correction Based on Verification Techniques. In *Proceedings of Design Automation Conference (DAC)*, pages 258–261, 1996.

[3] N. Krishnamurthy, A. K. Martin, M. S. Abadir, and J. A. Abraham. Validating powerpc microprocessor custom memories. *IEEE Design and Test of Computers*, 17(4):61–76, 2000.

[4] A. Kuehlmann, D. I. Cheng, A. Srinivasan, and D. P. LaPotin. Error Diagnosis for Transistor-level Verification. In *Proceedings of Design Automation Conference (DAC)*, pages 218–224, 1994.

[5] C-J. H. Seger and R. E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[6] L-C. Wang, M. S. Abadir, and N. Krishnamurthy. Automatic Generation of Assertions for Formal Verification of PowerPC Microprocessor Arrays Using Symbolic Trajectory Evaluation. In *Proceedings of Design Automation Conference (DAC)*, pages 534–537, 1998.