Solver Technology for System-level to RTL Equivalence Checking

Alfred Koelbl
koelbl@synopsys.com

Reily Jacoby
reilyj@synopsys.com

Himanshu Jain
hjain@synopsys.com

Carl Pixley
cpixley@synopsys.com

Verification Group
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124

***Abstract*—Checking the equivalence of a system-level model against an RTL design is a major challenge. The reason is that usually the system-level model is written by a system architect, whereas the RTL implementation is created by a hardware designer. This approach leads to two models that are significantly different. Checking the equivalence of real-life designs requires strong solver technology. The challenges can only be overcome with a combination of bit-level and word-level reasoning techniques, combined with the right orchestration. In this paper, we discuss solver technology that has shown to be effective on many real-life equivalence checking problems.**

I. Introduction

With the rising complexity of modern IC designs, a growing trend towards designing hardware at higher levels of abstraction has emerged. Many companies have adopted a methodology that starts out with a design specification in a system-level programming language like C++/SystemC or SystemVerilog. A higher level of abstraction enables easier design exploration and more thorough validation due to higher simulation speed. By adding more and more implementation details, the system-level specification is eventually refined to a Register-Transfer-Level (RTL) description. This process is either performed manually by a hardware designer or automatically by a high-level synthesis tool. In such a flow, one of the most important verification tasks is to check whether the RTL implementation is indeed equivalent to the system-level model. Today, the prevalent technique for checking if the resulting RTL complies with the system-level specification is simulation. As exhaustive simulation is not feasible for industrial-sized designs, corner-case bugs frequently remain undetected. This problem gives rise to the use of formal techniques.

Whereas formal equivalence checking for RTL to RTL comparisons has been in production use for many years now, system-level to RTL checking is an area of active research. The huge semantic gap between the system-level model and the RTL requires sophisticated solvers that have the capacity to deal with the real-life designs. In this paper, we discuss our experience with solver technology specifically for system-level to RTL equivalence checking. As shown in Fig. 1, the equivalence checking flow starts with two models, a system-level model specified in a system-level language like C++, SystemC or SystemVerilog and an RTL implementation.

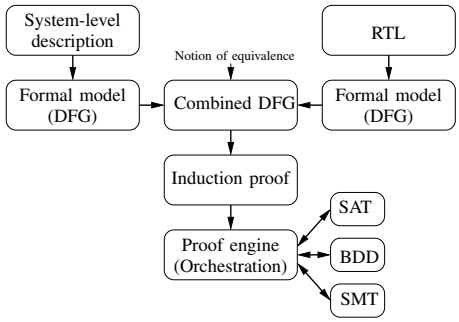

Fig. 1. Equivalence checking flow.

The first step is to convert both models into a formal representation. In our case, this representation is a word-level data-flow-graph (DFG) [1]. The DFG of a system-level model represents its input/output transition relation whereas the DFG constructed for the RTL is basically a word-level netlist. An example program and its DFG is shown in Fig. 2. The inputs (shown at the top) are the global variables a, b and c. The expressions feeding the outputs a', b' and c' (at the bottom) represent the symbolic expressions of the global variables a, b and c after executing function fct.

In the next step, the two DFGs are combined according to a given notion of equivalence. For instance, the system-level model can be untimed whereas the RTL is pipelined. For the proof engine, the two DFGs must be combined such that timing differences are eliminated ([2]), thereby reducing the problem to a cycle-accurate equivalence check.

```
int a, b, c;

void fct()
{
    b++;
    if (a > 0)
        c = a + b;
    else
        c = a * b;
    a = c;
}
```

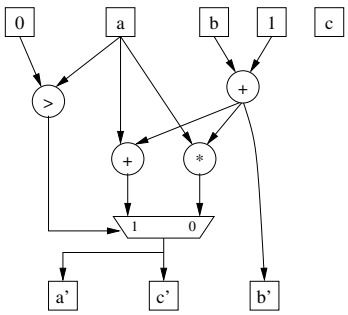

Fig. 2. Data-flow-graph example.

978-3-9810801-5-5/DATE09 © 2009 EDAA

The resulting DFG could be checked by standard reachability techniques. Unfortunately, reachability computation only works well for control dominated designs but does not scale for data-paths. A technique that scales much better is k-induction. In k-induction, the transition relation (T) in form of a DFG is unrolled $k-1$ times. A property P is added, stating that the outputs are equivalent in these $k-1$ unrollings. The inductive step then proves that the outputs are also equivalent in the kth unrolling. The induction base checks whether the outputs are indeed equivalent in the first $k-1$ unrollings, if the system is started in the initial state I:

$$I(s_0) \wedge T(s_0, s_1) \wedge ... \wedge T(s_{k-2}, s_{k-1}) \Rightarrow P(s_0) \wedge ... \wedge P(s_{k-1})$$

$$P(s_0) \wedge T(s_0, s_1) \wedge ... \wedge P(s_{k-1}) \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k)$$

The result of formulating the induction proof is a large formula represented as a DFG that must be checked for satisfiability. If the formula is unsatisfiable, the outputs of the two designs are equivalent, otherwise a counter-example is produced revealing the discrepancy. Details on how the DFG is constructed for the induction proof is beyond the scope of this paper ([3]). The focus of this paper is to explain the techniques used for checking the satisfiability of the induction DFG. Fig. 3 shows the DFG structure of a typical equivalence checking problem.

The corresponding outputs of the system-level description o_{spec} and the RTL description o_{impl} are compared. Additional outputs may be created to test side conditions to help ensure the integrity of the formal model. Side conditions arise from undefined behaviors in the C++ (or possibly RTL) program. For example, in the ANSI standard the result of out-of-bounds array accesses, negative shifts or division by zero is undefined.

Generating a formal model for a language with undefined behavior is problematic because the model always needs to take into account all possible cases. In practice it is not feasible to model all possible behavior but we can restrict ourselves to only the fully defined behavior. The conditions for which the behavior becomes undefined are explicitly entered into the DFG in form of a satisfiability check. If any of the side conditions are violated, the C++ program behaves in an undefined way and we need to flag this to the user.

In addition to the specification and implementation DFGs that need to be checked for equivalence, a constraint DFG is used to model input constraints, output don't-cares and register mappings. The corresponding outputs of the two DFGs only need to be equivalent if all constraints are fulfilled.

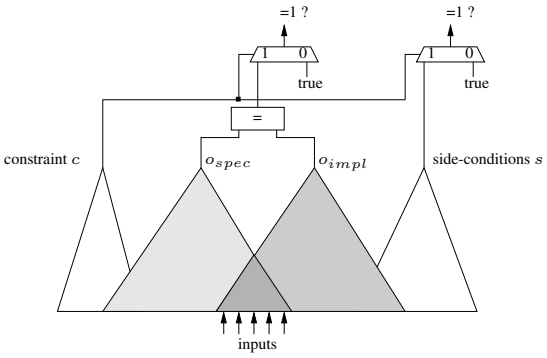

Fig. 3. DFG structure of equivalence checking problem

In Section II we describe the high level flow of the solver system. In Section III, we discuss our bit-level leaf engines. Section IV provides a summary of our word-level leaf engine technology. In Section V, we discuss many of our rewrite engines that have proven successful in solving hard problems. In Section VI, we describe our engine sequencing environment, which controls the order and duration of the various engines. The paper concludes with results in Section VII.

II. Overview

The most straight-forward approach to solving the equivalence problem of Fig. 3 would be to simply convert the word-level DFG into a bit-blasted representation and then use any of a number of bit-level solvers (e.g. SAT, BDD, ATPG) to complete the proof. While this is an effective strategy for some designs, it has proven to be inadequate over a broad sampling of industrial examples. Large designs and designs that contain arithmetic operators are especially problematic for this approach.

Word-level representations, such as a DFG, have a number of attractive qualities that the equivalent bit-blasted representations do not have. Word-level representations take up far less memory than a bit-level representation and therefore allow very large designs to be compared. Since word-level graphs reflect a higher level of abstraction than bit-level graphs, more powerful kinds of reasoning can be employed (e.g. easily recognizing that A(B+C) is the same as A*B + A*C). Moreover, since word level graphs more closely reflect the source from which they are compiled, better user feedback about the source of discrepancies is possible.

We have therefore designed the solver system motivated by the belief that maintaining a word level representation throughout the proof process, while still exploiting the power of bit-level solvers, would result in the most robust and efficient equivalence checking system. The high level structure of this system is similar to the architecture of state-of-the-art bit-level equivalence checkers in use today. The emphasis is to identify structural similarities between the two designs and use this information to simplify the proof of the outputs.

When the DFG of figure 3 is given to the solvers, the following operations are performed:

1) Form potentially equivalent pairs (PEPs)
2) Random pattern simulation
3) Redundancy removal
4) Graph re-writing
5) Solve the PEPs
6) Solve the outputs

Not all of these steps are used on each design and some steps, such as graph re-writing, can be repeated multiple times. The last four steps generally involve specifying and invoking an engine sequence (described in Section VI) to prove or disprove the problems of that step. This flow is implemented in a very flexible manner - operations can be skipped or permuted depending on the specific problem.

- Form potentially equivalent pairs
  One random pattern is simulated and any pair of nodes that have the same simulation value are considered to be a PEP. In bit-level equivalence checking it is sufficient to consider whether two nodes have the same binary value for all random patterns applied. In word-level systems multiple notions of what it means for two nodes to be equivalent must be employed. This is due to the fact that system-level models are often written in a programming language with a restricted set of data types. For example, a 4-bit data item in the RTL may be stored in an integer (32-bit) in a C model. To increase the likelihood of finding internal equivalences, we use different notions of what it means for two nodes to be equivalent.
  - Same bitwidth. Two nodes have the same bit width and the same value.
  - Zero pad. Two nodes have different bit widths. The two nodes are considered equivalent if by padding the most significant bits of the shorter word with zeros causes the two nodes to have the same value.
  - Sign extension. Two nodes have different bit widths. The two nodes are considered equivalent if by performing sign extension on the shorter word causes the two nodes to have the same value.
  - Extraction. Two nodes have different bit widths. The two nodes are considered equivalent if there is a bit-range extraction of the larger node that has an identical value as the shorter word.
- Random pattern simulation
  The purpose of this step is to reduce the number of PEPs that must be sent to the engine sequence. Anytime a random pattern causes the two nodes of a PEP to differ (with respect to the notion of equivalence being employed), that PEP can be discarded. Note that random simulation and proofs take into account the constraint logic, i.e., the condition for two intermediate nodes being equivalent is $c \rightarrow (p_1 == p_2)$, were c denotes the constraint logic and p_1 and p_2 are the potential intermediate equivalent nodes. We use a constraint solver to ensure that the random patterns generated satisfy all the constraints.
- Redundancy removal
  If a node has had the same value for every random pattern applied, it is formally compared to that value in this step. If the node is found to be constant valued, it is replaced with a constant and the simplification is propagated.
- Graph re-writing
  A very flexible graph re-write capability has been implemented in this system and is described in section V. Multiple sets of re-write rules have been developed. Which sets are employed is controlled by a set of heuristics.
- Solve the PEPs
  The proof engines are invoked for each of the PEPs and if the engine comes back with a proof, the two nodes of the PEP are merged. However, in some cases the merging is not performed and equivalence of the two nodes is captured as a learned relationship and added to the set of constraints. Merging is attractive since it always reduces the size of the graph for subsequent PEP proofs. Storing the learned relationship is a more powerful mechanism and is useful on some of the more difficult problems. If the two nodes are shown to be not equivalent, then the counter example trace is used to prune the set of PEPs.
- Solve the outputs
  Finally, the proof engines are invoked for each of the output comparisons.

III. Bit-level engines

Our bit-level engines consist of Boolean satisfiability (SAT) solvers, ATPG and BDD based solvers. The word-level DFG is converted into a bit-level netlist. The standard technique for checking the satisfiability of a bit-level formula is to convert it into CNF and invoke an off-the-shelf SAT solver. This technique is very successful due to significant improvements in the capacity of SAT solvers over the past decade. Most SAT solvers use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. In state-of-the-art SAT solvers the basic DPLL algorithm is enhanced using various techniques such as two-watched literal scheme for efficient Boolean Constraint Propagation, conflict driven learning, non-chronological backtracking, rapid restarts, conflict clause minimization, variable activity based decision, pre-processing, and phase saving [4], [5], [6], [7], [8].

We found that current state-of-the-art SAT solvers perform well and are comparable in performance on our problems. SAT solvers form our primary bit-level engines. However, we have also encountered problems where a BDD engine provided superior results. This is especially the case for large XOR trees that occur in cryptographic designs. In addition to SAT and BDDs we use a bit-level redundancy removal engine, a structural ATPG engine that takes into account equivalences and several bit-level optimization engines that try to simplify the problem as much as possible.

In summary, bit-level engines work well on bit-level problems and SAT solvers are good for finding counter-examples. The main disadvantage is that the high-level structure present in a word-level formula gets lost. Reasoning about the bit-level encodings of operators such as multiplication/division/mod is difficult for bit-level engines. As the bitwidth of operators increases the corresponding bit-level problems become harder.

IV. Word-level engines

Word-level engines seek to remove the weakness of bit-level solvers on arithmetic designs. This is usually done by combining multiple theories such as linear arithmetic, theory of arrays, bit-vectors in a single prover framework called Satisfiability Modulo Theories (SMT) framework. Currently there are three main approaches to SMT solving:

1) *Eager* SMT solvers first try to solve or simplify the word-level problem using pre-processing, rewrites and abstraction. If the rewrites are not sufficient for a solution, the word-level formula is bit-blasted and handed

to a SAT solver. Some eager SMT solvers are BAT [9], STP[10], Boolector[11], Beaver [12].

2) *Lazy* SMT solvers combine a fast Boolean SAT solver with dedicated *theory solvers*. A theory solver is only required to determine the satisfiability of a conjunction of atomic terms of a particular logical theory. A given formula ϕ is abstracted to a Boolean formula ϕ_b, by replacing all atomic theory predicates in ϕ by Boolean variables. The satisfiability of ϕ_b is checked using a SAT solver. If ϕ_b is unsatisfiable, then ϕ is also unsatisfiable. There are many variations and optimizations possible on the basic technique (see [13] for a survey). Some lazy SMT solvers are Z3, Yices, CVClite, CVC3.
3) Word Level Decision Diagrams: Most hardware and software designs perform operations over words of data. The functionality of such word-level designs is not represented efficiently at the bit-level using OBDDs. *Word level decision diagrams* (WLDDs) generalize the idea of OBDDs to represent functions that map a vector of Boolean variables (or words) to integers, reals, or rational numbers. WLDDs provide a more natural way of representing functions that operate over words of data. Some examples of WLDDs are Multi-terminal Binary Decision Diagrams (MTBDDs), Binary Moment Diagrams (BMDs), Multiplicative Binary Moment Diagrams (*BMDs, K*BMDs), Taylor expansion diagrams. WLDDs can represent various operations such as addition, multiplication and exponentiation efficiently.

Our experience with SMT solvers shows that only a few theories are useful for system-level equivalence checking. As we are always dealing with fixed width operators, theories with unbounded precision are not applicable. Many discrepancies are caused by overflow and wrap-around effects, which means that we need to model bit-vectors accurately. The theories that we have found useful are the theory of bit-vectors and the theory of arrays. Array operations are implemented by read and write nodes in the DFG. The theory of arrays allows efficient treatment of array operations without explicitly representing every single array location. Furthermore, our problems involve heavy mixing of modular arithmetic and bitwise operators, which poses significant problems for the solvers of category 3. We had the most success with solvers of category 1. Rewrites for bit-vector arithmetic are applied on the word-level first. If the problem can not be solved by the pre-processing/rewrites, the resulting problem is converted to CNF and handed over to a SAT solver. We use our own word-level SMT solver Combat that implements the theory of bit-vectors on top of a SAT solver.

The success of word-level engines is strongly correlated with the quality of the rewrites. There is still lots of opportunity for research as the rewrites used in existing solvers are not particularly powerful. Major shortcoming of current SMT solvers for our purposes is that they mostly target (software) model checking problems and do not exploit the similarities present in the formulas arising in equivalence checking.

V. Rewrite engines

In dealing with modular arithmetic and word-level validity checking, the most promising approach to date are rewrites. A rewrite rule transforms a subgraph of a DFG into another structurally different but functionally equivalent one. In our equivalence checker, the rewrite engines are our core word-level decision procedures. We distinguish between several different types of rewrites:

1) Local rewrites
   These rewrites always result in a simpler DFG. This category comprises constant propagation, e.g., $a \wedge 0 = 0$ and rewrites that directly simplify an operator without traversing its transitive fanin, e.g., $a \wedge a = a$.
2) Simplifying rewrites
   These are more comples rewrites that potentially involve several DFG nodes and require some matching in the transitive fanin of the processed node. For example, the following rule simplifies two consecutive multiplexers with the same selector input:

   $$mux(s1, mux(s1, v1, v2), v3) \rightarrow mux(s1, v1, v3).$$

   We only allow rewrites that do not break sharing between DFG nodes. This is necessary to guarantee that the resulting DFG is always smaller than the original DFG. Simplifying rewrites (and local rewrites) are always enabled. All other rewrite categories can potentially increase the size of the DFG and are only applied selectively.
3) Bitwidth/interval analysis rewrites
   These rewrites require an analysis pass. For example, the bitwidth analysis performs a very simple symbolic simulation on the bit-level with values 0, 1, x and $\overline{x}$. The analysis is used to determine the minimum bitwidth necessary for operators. Furthermore, it can be used to detect simple redundancies and equivalences that can not be easily detected by simplifying rewrites.
   Interval analysis [14] computes a range interval for each word-level DFG node. The range is used to shrink the bitwidth of operators, e.g., if range analysis determines that the result of a 32-bit unsigned addition is guaranteed to be in the interval [0, 255], the addition can be reduced to a bitwidth of 8 bits.
4) Normalizing rewrites
   These rewrites do not necessarily simplify the DFG. Instead, they try to reduce a subgraph to a normal form [15]. The hope is that if both designs get converted to a normal form, their graphs might become isomorphic. For example, these rewrites are quite successful in matching structurally different adder trees. A normal form is not necessarily canonical, i.e., it is rather unlikely that two big DFGs when converted to a normal form become isomorphic. Thus, these rewrites are more successful in proving intermediate equivalences. A drawback of normalizing rewrites is that they might increase the size of the DFG and thus can be counter-productive.

5) Implications between the two designs
   These rewrites add constraints that describe non-trivial relationships between both designs. Consider a multiplier m_s in the specification and a multiplier m_i in the implementation:

$$m_s = a \cdot b \qquad m_i = (a << v) \cdot b,$$

   where a, b and v represent expression in the DFG. The first operands of the multipliers do not match and will not be detected as intermediate equivalences. However, a rewrite rule can detect that there actually exists a relationship between the two multipliers and add a constraint c relating the outputs of the two multipliers:

$$c = (m_i == (m_s << v))$$

   These rewrites can also be used to detect non-trivial relationships within a design, e.g., if signals are detected that are one-hot encoded it could be beneficial to add an explicit one-hot constraint for them.
6) Adding redundant logic
   These rewrites add redundant logic. The hope is that a subsequent intermediate equivalence detection pass will be able to find a relationship between the design and the redundant logic, thereby increasing sharing between the two designs. For example, in many designs, integer division and remainder computation go hand in hand. Thus, whenever we detect an implication between two integer division nodes and that implication also leads to an implication between their remainders, we also add a remainder node and the remainder implication. If the design indeed computes the remainder, the hope is that the intermediate equivalence pass detects equivalence between the existing remainder and the newly introduced redundant remainder.
7) Rewrites requiring proofs
   These are rewrites that are only valid under some very specific conditions. Checking whether these conditions are fulfilled might require additional proofs. These rewrites pose an interesting implementation problem because they lead to recursive calls of the proof engine.

Rewrites are very powerful but it is very important that they are checked for correctness. Furthermore, it is important that all rewrites that do not monotonically decrease the size of the DFG are not invoked cyclically.

VI. Engine sequencing

The ability to invoke the right engines in the right order is essential for the success of an overall proof strategy. Our engine sequencing strategy uses a specification mechanism that allowes the user great flexibility over the order in which engines are applied to each problem. The application of the various bit-level and word-level engines to a problem is controlled by a prover specification. A prover specification describes the order in which the various engines are applied to a proof and the level of effort for each engine.

A prover specification consists of zero or more meta engines and one or more of the leaf engines discussed in Sections III and IV. Only leaf engines are capable of deciding whether a given problem is satisfiable or unsatisfiable.

The meta engines control the level of effort expended by the leaf engines in its scope. They also control when the leaf engines should no longer be called. And finally, meta engines can modify the existing problem into a set of (hopefully) easier sub-problems for the leaf engines to work on. Meta engines always have one (or more) child engine(s) specified. Leaf engines do not have any child engines.

A prover specification can be visualized as a tree graph in which the interior nodes are meta engines and the leaf nodes are associated with leaf engines. Given a proof problem, a pre-order traversal of the prover specification tree is performed applying each engine visited on the traversal until the problem is solved or until the traversal ends without an answer.
Some of the leaf engines are:

- sat - bit-level SAT engine
- bdd - bit-level BDD solver
- opt - bit-level optimization engine
- atpg - bit-level test pattern generation engine
- combat - word-level solver

The meta-engines are:

- seq. The seq engine takes 1 or more child engines as an argument. It simply passes the problem it is given to each child engine in the sequence until the problem is solved or until there are no more engines.
- parallel. The parallel engine takes 1 or more child engines as an argument. The problem is passed to all child engines at the same time and the first engine to complete causes all the other engines to be killed.
- limit. The limit engine takes 1 child engine as an argument. The limit engine monitors the time spent in its child engine and stops working on any future problems after the resource limits are exceeded. There are a number of resources other than time that the limit engine tracks.
- cut. The cut engine takes 1 child engine as an argument. The cut engine attempts to find cut sets of already merged nodes in the fanin cone of the current problem. The cut points are replaced by newly created primary inputs and the resulting problem is passed to the child engine. Options control how many cut sets are formed.
- case. The case engine takes 1 child engine as an argument. Options allow the specification of a set of signals that will be case split. All possible values for these signals are enumerated and each sub-problem is sent to the child engine. Many problems become significantly easier by performing a case-split on the right input variable or internal condition. The hope is that the individual cases are much easier to solve than the original problem.

Below is an example of a prover specification.

```
{seq {limit -t 10 {sat}}
     {limit -t 20 {seq {bdd} {cut -n 2 {sat}}}}
     {limit -t 1000 {seq {opt} {atpg}}} }
```

When a PEP is given to the prover that is generated from this specification, the seq engine simply passes the problem to the first engine in the sequence. The limit engine checks to see if the cpu time of all calls to its child engine (sat) have exceeded 10 seconds. If is has, it returns back to the seq engine which then passes the problem to the next engine in the sequence. However, if the time limit is not exceeded, the limit engine will send the problem to the sat engine. If the sat engine is able to solve the problem, the result is returned and no further engines are called.

Fig. 4 illustrates the efficacy of having multiple engines integrated by the flexible prover specification architecture. This is a DSP intensive design containing many adders and multipliers. In the first attempt, utilizing only the bit-level engines, 52 of the 68 outputs remained unsolved. Once combat, the word-level solver, was introduced into the prover specification, only 18 outputs remained unsolved. Bringing the logic optimization engine into the engine flow decreased the number of unsolved outputs down to 5. And finally, by turning on the graph rewriting rules, we were able to solve all of the outputs.

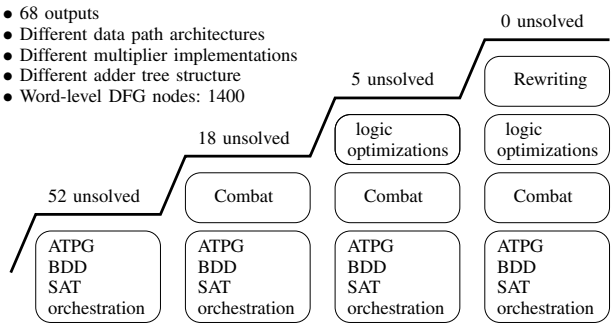

Fig. 4. Effectiveness comes from many techniques

VII. Experimental results

The presented solver technology is part of our system-level to RTL equivalence checking tool "Hector". As a case study, we picked the formal equivalence check of the Synopsys DesignWare floating point division component against a software implementation of the IEEE floating point standard in C. The C implementation is the publicly available "softfloat" library. This library has been around for many years and no bugs have been reported since 1998 ([16]). The proof makes heavy use of all the techniques mentioned above. Both the C/Verilog implementations of the floating point division take two 32-bit floating point numbers (dividend and divisor) as input and return a 32-bit floating point number as result. We check the equivalence for four different IEEE rounding modes: round to zero, round to nearest (even), round to positive infinity, and round to negative infinity. Our first attempt to solve this problem was to bit-blast the DFG into a bit-level Boolean formula and check the equivalence using state-of-the-art SAT solvers. Current SAT solvers did not finish even after running for days. This is not surprising because the miter DFG contains two unsigned integer division operations of different width, one 64-bit integer multiplication operation, and one 49-bit unsigned integer modulo (remainder) operation. Current bit-level solvers do not perform well on the Boolean encoding of division, multiplication, and modulo operations.

Exploiting intermediate equivalence points had some impact in reducing the size of the miter DFG but they were still not sufficient to solve the problem. Differences in the way the input operands are normalized in both implementations resulted in only few internal equivalences.

The rewrite engines, however, proved very effective for this design. The idea is to automatically discover implicit word-level relationships between the results of division, multiplication, and modulo operations and make them explicit in form of additional constraints. With the new constraints, the bit-level engines can avoid reasoning about the hard operations.

The key, however, was the ability to orchestrate the different techniques appropriately. The result of the rewrite engines increased the sharing and made the implicit word-level relationships explicit but it was still not enough to prove the outputs equivalent. Another pass of finding intermediate equivalences and merging them detected enough equivalences to make the final SAT solver call tractable. Using all of the engines mentioned above, the total time for the equivalence check was under five minutes.

References

[1] A. Koelbl and C. Pixley, "Constructing efficient formal models from high-level descriptions using symbolic simulation," *Int. J. Parallel Program.*, vol. 33, no. 6, pp. 645–666, 2005.

[2] A. Koelbl, Y. Lu, and A. Mathur, "Embedded tutorial: formal equivalence checking between system-level models and rtl," in *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*. IEEE Computer Society, 2005, pp. 965–971.

[3] A. Koelbl, J. R. Burch, and C. Pixley, "Memory modeling in esl-rtl equivalence checking," in *DAC '07: Proceedings of the 44th annual conference on Design automation*. ACM, 2007, pp. 205–209.

[4] "Minisat sat solver," http://minisat.se/.

[5] J. P. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, November 1996, pp. 220–227.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference (DAC'01)*, June 2001, pp. 530–535.

[7] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *SAT*, 2007, pp. 294–299.

[8] N. Eén and A. Biere, "Effective Preprocessing in SAT Through Variable and Clause Elimination." in *SAT*, 2005, pp. 61–75.

[9] P. Manolios, S. K. Srinivasan, and D. Vroon, "Automatic memory reductions for rtl model verification," in *ICCAD 2006: ACM-IEEE International Conference on Computer Aided Design*, 2006.

[10] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification (CAV '07)*. Berlin, Germany: Springer-Verlag, July 2007.

[11] "Boolector , http://fmv.jku.at/boolector/."

[12] "Beaver smt solver," http://uclid.eecs.berkeley.edu/Beaver/.

[13] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[14] W. Backes, "A new interval approximation supporting bit operations and byte access," 2006.

[15] C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker," in *16th International Conference on Computer Aided Verification, CAV'04*, 2004.

[16] "Softfloat library," http://jhauser.us/arithmetic/SoftFloat.html.