# Equivalence Checking a Floating-point Unit against a High-level C Model

Rajdeep Mukherjee[1], Saurabh Joshi[2], Andreas Griesmayer[3], Daniel Kroening[1], and Tom Melham[1]

[1] University of Oxford, UK
[2] IIT Hyderabad, India
[3] ARM Limited
{rajdeep.mukherjee,kroening,tom.melham}@cs.ox.ac.uk,
sbjoshi@iith.ac.in, andreas.griesmayer@arm.com

**Abstract.** Semiconductor companies have increasingly adopted a methodology that starts with a system-level design specification in C/C++/SystemC. This model is extensively simulated to ensure correct functionality and performance. Later, a Register Transfer Level (RTL) implementation is created in Verilog, either manually by a designer or automatically by a high-level synthesis tool. It is essential to check that the C and Verilog programs are consistent. In this paper, we present a two-step approach, embodied in two equivalence checking tools, VERIFOX and HW-CBMC, to validate designs at the software and RTL levels, respectively. VERIFOX is used for equivalence checking of an untimed software model in C against a high-level reference model in C. HW-CBMC verifies the equivalence of a Verilog RTL implementation against an untimed software model in C. To evaluate our tools, we applied them to a commercial floating-point arithmetic unit (FPU) from ARM and an open-source dual-path floating-point adder.

## 1 Introduction

One of the most important tasks in Electronic Design Automation (EDA) is to check whether the low-level implementation (RTL or gate-level) complies with the system-level specification. Figure 1 illustrates the role of equivalence checking (EC) in the design process. In this paper, we present a new EC tool, VERIFOX, that is used for equivalence checking of an untimed software (SW) model against a high-level reference model. Later, a Register Transfer Level (RTL) model is implemented, either manually by a hardware designer or automatically by a synthesis tool. To guarantee that the RTL is consistent with the SW model, we use an existing tool, HW-CBMC [15], to check the correctness of the synthesized hardware RTL against a SW model.

In this paper, we address the most general and thus most difficult variant of EC: the case where the high-level and the low-level design are substantially different. State-of-the-art tools, such as Hector [14] from Synopsys and SLEC from Calypto,[4] rely on *equivalence points* [18], and hence they are ineffective in this scenario. We present

[4] http://calypto.com/en/products/slec/

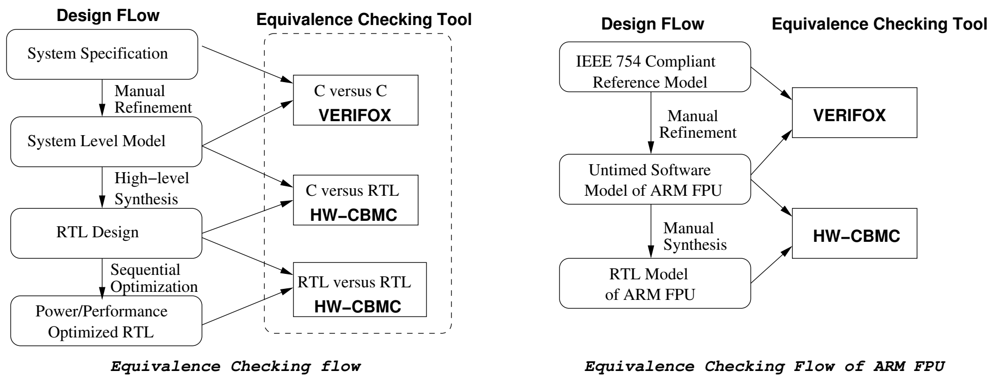

**Fig. 1.** Electronic Design Automation Flow

an approach based on bounded analysis, embodied in the tools VERIFOX and HW-CBMC, that can handle arbitrary designs. EC is broadly classified into two separate categories: combinational equivalence checking (CEC) and sequential equivalence checking (SEC). CEC is used for a pair of models that are cycle accurate and have the same state-holding elements. SEC is used when the high-level model is not cycle accurate or has a substantially different set of state-holding elements [1, 11]. It is well-known that EC of floating-point designs is difficult [12, 19]. So there is a need for automatic tools that formally validate floating-point designs at various stages of the synthesis flow, as illustrated by right side flow of Figure 1.

An extended version of this paper, showing worked examples and giving further technical details, is available at [17].

## 2 VERIFOX: A tool for equivalence checking of C programs

VERIFOX is a path-based symbolic execution tool for equivalence checking of C programs. The tool architecture is shown on the left side of Figure 2. VERIFOX supports the C89 and C99 standards. The key feature is symbolic reasoning about equivalence between FP operations. To this end, VERIFOX implements a model of the core IEEE 754 arithmetic operations—single- and double-precision addition, subtraction, multiplication, and division—which can be used as reference designs for equivalence checking. So VERIFOX does not require external reference models for equivalence checking of floating-point designs. This significantly simplifies the users effort to do equivalence checking at software level. The reference model in VERIFOX is equivalent to the Softfloat model.[5] VERIFOX also supports SAT and SMT backends for constraint solving.

Given a reference model, an implementation model in C and a set of partition constraints, VERIFOX performs depth-first exploration of program paths with certain optimizations, such as eager infeasible path pruning and incremental constraint solving. This enables automatic decomposition of the verification state-space into subproblems,

[5] http://www.jhauser.us/arithmetic/SoftFloat.html

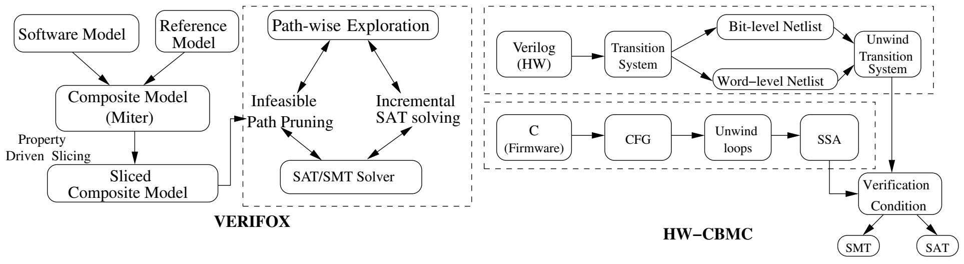

**Fig. 2.** VERIFOX and HW-CBMC Tool Architecture

by input-space and/or state-space decomposition. The decomposition is done in tandem in both models, exploiting the structure present in the high-level model. The approach generates many but simpler SAT/SMT queries, similar to the technique followed in KLEE [4]. Figure 3 shows three feasible path constraints corresponding to the three paths in the program on the left. In contrast, the last column of Figure 3 shows the monolithic path-constraint generated by HW-CBMC. We distribute the pre-compiled static binary of VERIFOX at `http://www.cprover.org/verifox`.

***Incremental solving in* VERIFOX.** VERIFOX can be run in two different modes: partial incremental and full incremental. In partial incremental mode, only one solver instance is maintained while going down a single path. So when making a feasibility check from one branch $b_1$ to another branch $b_2$ along a single path, only the program segment from $b_1$ to $b_2$ is encoded as a constraint and added to the existing solver instance. Internal solver states and the information that the solver gathers during the search remain valid as long as all the queries that are posed to the solver in succession are monotonically stronger. If the solver solves a formula $\phi$, then posing $\phi \wedge \psi$ as a query to the same solver instance allows one to reuse solver knowledge it has already acquired, because any assignment that falsifies $\phi$ also falsifies $\phi \wedge \psi$. Thus the solver need not revisit the assignments that it has already ruled out. This results in speeding up the feasibility check of the symbolic state at $b_2$, as the feasibility check at $b_1$ was *true*. A new solver instance is used to explore a different path, after the current path is detected as infeasible.

In full incremental mode, only one solver instance is maintained throughout the whole symbolic execution. Let $\phi_{b_1 b_2}$ denote the encoding of the path fragment from $b_1$

| Program | Path Constraint 1 | Path Constraint 2 | Path Constraint 3 | Monolithic Path Constraint |
|---|---|---|---|---|
| `void top(){`<br>` if(reset) {`<br>`   x=0;`<br>`   y=0; }`<br>` else {`<br>`  if(a > b)`<br>`   x=a+b;`<br>`  else`<br>`   y=(a & 3)<<b; }}` | $C_1 \equiv$ $reset_1 \neq 0 \wedge$ $x_2 = 0 \wedge$ $y_2 = 0$ | $C_2 \equiv$ $reset_1 = 0 \wedge$ $b_1 \not\geq a_1 \wedge$ $x_3 = a_1 + b_1$ | $C_3 \equiv$ $reset_1 = 0 \wedge$ $b_1 \geq a_1 \wedge$ $y_3 = (a_1 \& 3)$ $<< b_1$ | $C \iff ((guard_1 = \neg(reset_1 = 0)) \wedge$ $(x_2 = 0) \wedge (y_2 = 0) \wedge$ $(x_3 = x_1) \wedge (y_3 = y_1) \wedge$ $(guard_2 = \neg(b_1 >= a_1)) \wedge$ $(x_4 = a_1 + b_1) \wedge (x_5 = x_3) \wedge$ $(y_4 = (a_1 \& 3) << b_1) \wedge$ $(x_6 = ite(guard_2, x_4, x_5)) \wedge$ $(y_5 = ite(guard_2, y_3, y_4)) \wedge$ $(x_7 = ite(guard_1, 0, x_6)) \wedge$ $(y_6 = ite(guard_1, 0, y_5)))$ |

**Fig. 3.** Single-path and Monolithic Symbolic Execution

to $b_2$. It is added in the solver as $B_{b_1 b_2} \Rightarrow \phi_{b_1 b_2}$. Then, $B_{b_1 b_2}$ is added as a *blocking variable*[6] to enforce constraints specified by $\phi_{b_1 b_2}$. Blocking variables are treated specially inside the solvers: unlike regular variables or clauses, the blocking can be removed in subsequent queries without invalidating the solver instance. When one wants to backtrack the symbolic execution, the blocking $B_{b_1 b_2}$ is removed and a unit clause $\neg B_{b_1 b_2}$ is added to the solver, thus effectively removing $\phi_{b_1 b_2}$.

## 3 HW-CBMC: A tool for equivalence checking of C and RTL

HW-CBMC is used for bounded equivalence checking of C and Verilog RTL. The tool architecture is shown on the right side of Figure 2. HW-CBMC supports IEEE 1364-2005 System Verilog standards and the C89, C99 standards. HW-CBMC maintains two separate flows for hardware and software. The top flow in Figure 2 uses synthesis to obtain either a bit-level or a word-level netlist from Verilog RTL. The bottom flow illustrates the translation of the C program into static single assignment (SSA) form [9]. These two flows meet only at the solver. Thus, HW-CBMC generates a monolithic formula from the C and RTL description, which is then checked with SAT/SMT solvers. HW-CBMC provides specific handshake primitives such as $next\_timeframe()$ and $set\_inputs()$ that direct the tool to set the inputs to the hardware signals and advance the clock, respectively. The details of HW-CBMC are available online.[7]

## 4 Experimental Results

In this section, we report experimental results for equivalence checking of difficult floating-point designs. All our experiments were performed on an Intel® Xeon® machine with 3.07 GHz clock speed and 48 GB RAM. All times reported are in seconds. MiniSAT-2.2.0 [10] was used as underlying SAT solver with VERIFOX 0.1 and HW-CBMC 5.4. The timeout for all our experiments was set to 2 hours.

***Proprietary Floating-point Arithmetic Core:*** We verified parts of a floating-point arithmetic unit (FPU) of a next generation ARM® GPU. The FP core is primarily composed of single- and double-precision *ADD*, *SUB*, *FMA* and *TBL* functional units, the register files, and interface logic. The pipelined computation unit implements FP operations on a 128-bit data-path. In this paper, we verified the single-precision addition (*FP-ADD*), rounding (*FP-ROUND*), minimum (*FP-MIN*) and maximum (*FP-MAX*) operations. The FP-ADD unit can perform two operations in parallel by using two 64-bit adders over multiple pipeline stages. Each 64-bit unit can also perform operations with smaller bit widths. The FPU decodes the incoming instruction, applies the input modifiers and provides properly modified input data to the respective sub-unit. The implementation is around 38000 LOC, generating tens of thousands of gates. We obtained the SW model (in C) and the Verilog RTL model of the FPU core from ARM. (Due to proprietary nature of the FPU design, we can not share the commercial ARM IP.)

[6] The SAT community uses the term *assumption variables* or *assumptions*, but we will use the term blocking variable to avoid ambiguity with assumptions in the program.

[7] http://www.cprover.org/hardware/sequential-equivalence/

***Open-source Dual-path Floating-point Adder:*** We have developed both a C and a Verilog implementation of an IEEE-754 32-bit single-precision dual-path floating point adder/subtractor. This floating-point design includes various modules for packing, unpacking, normalizing, rounding and handling of infinite, normal, subnormal, zero and NaN (Not-a-Number) cases. We distribute the C and RTL implementation of the dual-path FP adder at `http://www.cprover.org/verifox`.

***Reference Model:*** The IEEE 754 compliant floating-point implementations in VERIFOX are used as the golden reference model for equivalence checking at the software level. For equivalence checking at the RTL phase, we used the untimed software model from ARM as the reference model, as shown on the right side of Figure 1.

***Miters for Equivalence Checking:*** A miter circuit [3] is built from two given circuits *A* and *B* as follows: identical inputs are fed into *A* and *B*, and the outputs of *A* and *B* are compared using a comparator. For equivalence checking at software level, one of the circuits is a SW program and the other is a high-level reference model. For the RTL phase, one of the circuits is a SW program treated as reference model and the other is an RTL implementation.

***Case-splitting for Equivalence Checking:*** Case-splitting is a common practice to scale up formal verification [12,14,19] and is often performed by user-specified assumptions. The `CPROVER_assume(c)` statement instructs HW-CBMC and VERIFOX to restrict the analysis to only those paths satisfying a given condition `c`. For example, we can limit the analysis to those paths that are exercised by inputs where the rounding mode is nearest-even (RNE) and both input numbers are NaNs by adding the following line:

```
CPROVER_assume(roundingMode==RNE && uf_nan && ug_nan);
```

***Discussion of Results:*** Table 1 reports the run times for equivalence checking of the ARM FPU and the dual-path FP adder. Column 1 gives the name of FP design and columns 2–6 show the runtimes for partition modes INF, ZERO, NaN, SUBNORMAL, and NORMAL respectively. For example, the partition constraint 'INF' means addition of two infinite numbers. Column 7 reports the total time for equivalence checking without any partitioning.

VERIFOX successfully proved the equivalence of all FP operations in the SW implementation of ARM FPU against the built-in reference model. However, a bug in FP-MIN and FP-MAX (reported as ERROR in Table 1) is detected by HW-CBMC in the RTL implementation of ARM FPU when checked against the SW model of ARM FPU for the case when both the input numbers are NaN. This happens mostly due to bugs in the high-level synthesis tool or during manual translation of SW model to RTL. Further, we investigate the reason for higher verification times for subnormal numbers compared to normal, infinity, NaN's and zero's. This is attributed to higher number of paths in subnormal case compared to INF, NaN's and zero's. Closest to our floating-point symbolic execution technique in VERIFOX is the tool KLEE-FP [8]. We could not, however, run KLEE-FP on the software models because the front-end of KLEE-FP failed to parse the ARM models.

| Design | Case-splitting | | | | | No-partition |
|---|---|---|---|---|---|---|
| | INF | ZERO | NaN | SUBNORMAL | NORMAL | Total |
| Equivalence checking at Software Level (VERIFOX) | | | | | | |
| FP-ADD | 9.56 | 11.54 | 9.95 | 1124.18 | 77.74 | 1566.72 |
| FP-ROUND | 1.24 | 1.36 | 1.32 | 3.78 | 1.63 | 4.71 |
| FP-MIN | 9.76 | 9.85 | 9.78 | 28.67 | 9.86 | 48.70 |
| FP-MAX | 9.80 | 9.88 | 9.97 | 28.70 | 9.90 | 35.81 |
| DUAL-PATH ADDER | 3.15 | 3.11 | 2.14 | 88.12 | 55.28 | 497.67 |
| Equivalence checking at RTL (HW-CBMC) | | | | | | |
| FP-ADD | 18.12 | 18.02 | 17.87 | 18.73 | 39.60 | 40.72 |
| FP-ROUND | 11.87 | 12.73 | 13.44 | 13.67 | 14.03 | 14.11 |
| FP-MIN | 13.72 | 13.62 | ERROR | 14.10 | 14.08 | 14.15 |
| FP-MAX | 13.70 | 13.58 | ERROR | 14.09 | 14.06 | 14.05 |
| DUAL-PATH ADDER | 0.88 | 0.87 | 0.99 | 169.49 | 22.42 | 668.61 |

**Table 1.** Equivalence checking of ARM FPU and DUAL-PATH Adder (All time in seconds)

## 5 Related work

The concept of symbolic execution [4, 7, 13] is prevalent in the software domain for automated test generation as well as bug finding. Tools such as Dart [13], Klee [4], EXE [5], Cloud9 [16] employ such a technique for efficient test case generation and bug finding. By contrast, we used path-wise symbolic execution for equivalence checking of software models against a reference model. A user-provided assumption specifies certain testability criteria that render majority of the design logic irrelevant [12, 14, 19], thus giving rise to large number of infeasible paths in the design. Conventional SAT-based bounded model checking [2, 6, 15] can not exploit this infeasibility because these techniques create a monolithic formula by unrolling the entire transition system up to a given bound, which is then passed to SAT/SMT solver. These tools perform case-splitting at the level of solver through the effect of constant propagation. Optimizations such as eager path pruning combined with incremental encoding enable VERIFOX to address this limitation.

## 6 Concluding Remarks

In this paper we presented VERIFOX, our path-based symbolic execution tool, which is used for equivalence checking of arbitrary software models in C. The key feature of VERIFOX is symbolic reasoning on the equivalence between floating-point operations. To this end, VERIFOX implements a model of the core IEEE 754 arithmetic operations, which can be used for reference models. Further, to validate the synthesis of RTL from software model, we used our existing tool, HW-CBMC, for equivalence checking of RTL designs against the software model used as reference. We successfully demonstrated the utility of our equivalence checking tool chain, VERIFOX and HW-CBMC, on a large commercial FPU core from ARM and a dual-path FP adder. Experience suggests that the synthesis of software models to RTL is often error prone—this emphasizes the need for automated equivalence checking tools at various stages of EDA flow. In the future, we plan to investigate various path exploration strategies and path-merging techniques in VERIFOX to further scale equivalence checking to complex data and control intensive designs.

## Acknowledgements

Part of the presented work was conducted during an internship at ARM. The authors want to thank in particular Luka Dejanovic, Joe Tapply, and Ian Clifford for their help with setting up the experiments.

## References

1. Baumgartner, J., Mony, H., Paruthi, V., Kanzelman, R., Janssen, G.: Scalable sequential equivalence checking across arbitrary design transformations. In: ICCD. pp. 259–266. IEEE (2006)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
3. Brand, D.: Verification of large synthesized designs. In: ICCAD. pp. 534–537 (1993)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224. USENIX (2008)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12(2) (2008)
6. Clarke, E., Kroening, D.: Hardware verification using ANSI-C programs as a reference. In: Proceedings of the 2003 Asia and South Pacific Design Automation Conference. pp. 308–311. ASP-DAC, ACM (2003)
7. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. 2(3), 215–222 (1976)
8. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic crosschecking of floating-point and SIMD code. In: EuroSys. pp. 315–328 (2011)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: POPL. pp. 25–35. ACM (1989)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: SAT. pp. 61–75 (2005)
11. van Eijk, C.A.J.: Sequential equivalence checking without state space traversal. In: DATE. pp. 618–623. IEEE (1998)
12. Fujita, M.: Verification of arithmetic circuits by comparing two similar circuits. In: CAV. vol. 1102, pp. 159–168. Springer (1996)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. pp. 213–223 (2005)
14. Kölbl, A., Jacoby, R., Jain, H., Pixley, C.: Solver technology for system-level to RTL equivalence checking. In: DATE. pp. 196–201. IEEE (2009)
15. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC. pp. 368–371 (2003)
16. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: PLDI. pp. 193–204 (2012)
17. Mukherjee, R., Joshi, S., Griesmayer, A., Kroening, D., Melham, T.: Equivalence checking a floating-point unit against a high-level C model: Extended version. arXiv Computing Research Repository arXiv:1609.00169 [cs.SE] (September 2016)
18. Wu, W., Hsiao, M.S.: Mining global constraints for improving bounded sequential equivalence checking. In: DAC. pp. 743–748. ACM (2006)
19. Xue, B., Chatterjee, P., Shukla, S.K.: Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models. In: ASP-DAC. pp. 723–728. IEEE (2013)