Verification of Arithmetic Datapath Designs using Word-level Approach - A Case Study

Cunxi Yu, Walter Brown, Maciej Ciesielski
ECE Department, University of Massachusetts, Amherst, USA
ycunxi@umass.edu, webrown@umass.edu, ciesiel@ecs.umass.edu

***Abstract*— The paper describes an efficient method to prove equivalence between two integer arithmetic datapath designs specified at the register transfer level. The method is illustrated with an industrial ALU design. As reported in literature, solving it using a commercial equivalence checking tool required case-splitting, which limits its applicability to larger designs. We show how such a task can be solved as a simpler verification problem without case-splitting. We demonstrate both the word-level and bit-level approach to this problem and show that the method is scalable to large combinational datapath circuits. Experimental results demonstrate the application of the method to large combinational arithmetic circuits.**

Keywords*— *functional verification; arithmetic circuits; RTL transformations.

I. Introduction

One of the most challenging problems encountered in hardware design is the functional verification of arithmetic data paths. Boolean logic techniques based on Binary Decision Diagrams (BDDs) and satisfiability (SAT) solvers cannot handle complex arithmetic designs because they require "bit-blasting", i.e., flattening of the design into bit-level netlists. Exhaustive simulation is infeasible as equivalence checking between system-level and RTL or between different RTL models using simulation is prohibitively long. Despite the recent developments in formal verification tools, proving the correctness of RTL synthesis of large datapath designs is still beyond the capabilities of traditional formal tools.

In this paper we present a method to prove the correctness of RTL datapath design w.r.t. its original RTL specification using an integer ALU design taken from [1] as case-study. The verification of this 16-bit integer ALU design was reported to be solved using case-splitting and a commercial formal equivalence checker from Synopsys, Hector. While the CPU time of 8 sec for a 16-bit datapath circuit is sufficiently short, it is not clear how it would scale to larger designs. Here, we show how our method can solve the problem with virtually no limit on bit-width size.

II. Related Work

Previous work in this field is mostly related to symbolic simulation and canonical diagram representations. Different canonical representations have been proposed to check equivalence of designs on the same (or comparable) abstraction levels, including Binary Decision Diagrams (BDDs), Binary Moment Diagrams (BMDs), Taylor Expansion Diagrams (TED) [2], and other hybrid diagrams [3]. The application of BDDs is limited mostly to bit-level logic circuits because building a BDD for a complex arithmetic circuit often requires an excessive amount of memory. BMDs and TEDs offer a better space complexity and can be useful if word-level information is available.

Most of the work in register transfer level (RTL) verification concentrates on verifying translation from high level specification (such as C) to RTL [4]; some use data flow graph (DFG) as a formal model for high-level specification [5]. Others use RTL to TLM (transaction level model) abstraction for redesign and verification of RTL IPs [6]. Assertion-based verification techniques (ABV) are also used for system-level designs [7]. Industrial work in RTL verification typically addresses verification of RTL protocol implementation against its specification and uses temporal specification languages, such as TLA, and TLC model checking [8].

RTL vs gate-level verification is typically solved by translating the RTL design into some representative gate-level design and performing gate-level equivalence checking. These methods use a host of verification techniques, including SAT, SMT, and ATPG, some of them relying on comparing "structurally similar points", which may not exist between the two designs. For this reason, these tools are not scalable for large arithmetic circuits. In general, RTL to gate-level verification is typically formulated and solved as a SAT problem and is mostly based on structural analysis of the design [9]. Other tools, such as ABC, apply SAT to a "product design" constructed by creating a miter of two designs and trying to prove it to be unSAT. Gate-level verification of bit-level arithmetic circuits has been also approached using computationally expensive computer algebra methods [10] [11].

In this paper we concentrate on RTL-to-RTL verification of integer arithmetic circuits, using an industrial integer ALU design as a case-study, and apply a combination of word-level canonical representation (TED) and symbolic rewriting. Term rewriting techniques that have been used by some researchers [12] are incomplete, as they rely on simple rewriting rules (distributivity, commutativity, and associativity) and use non-canonical representations.

III. RTL Verification of Datapaths

Consider an integer arithmetic logic unit (ALU), shown in Figure 1, taken from [1]. This architecture is used often in implementing integer operations for standard graphics APIs. The design consists of three word-level n-bit inputs, $A,\ B,\ C$, representing unsigned integers. Each of the operands can be optionally negated under the control of single-bit signals, neg_A, neg_B, neg_C. These bits, together with other configuration bits (en_{ab}, en_c and a negation bit neg_y of one of the local outputs), provide control for various arithmetic functions: $A \cdot B,\ -A \cdot B,\ A \cdot B + C,\ A \cdot B - C$, etc.

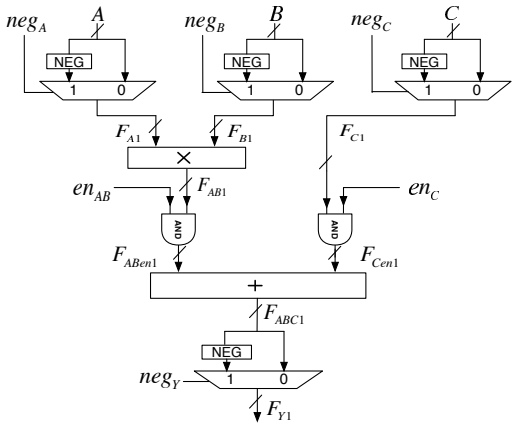


Fig. 1. Integer ALU - initial RTL design

In [1] the description of the integer ALU design was subjected to a number of algebraic and Boolean transformations resulting in the modified design shown in Figure 2. While the applied transformations can be shown to be mathematically correct, it is important to formally verify if the resulting RTL *hardware* implementation is indeed equivalent to the original one. This must be done to ensure that unexpected bugs, typically related to finite bit-widths, sign extension, or two's complement implementation of subtraction, did not creep into the final implementation. In [1] this problem was solved for bit-width $n = 16$ using *Hector*, a formal equivalence checking tool from Synopsys. The approach taken there required case-splitting and separately solving a number of individual cases, determined by the combination of the control signals.

We approach this problem differently and perform verification using symbolic representations for both RTL designs to check if they are equivalent. Two versions of the proof are considered here: 1) In the first method the symbolic equations are derived for each design and a canonical TED representation [2] is used to show that the two RTL implementations are equivalent for *arbitrary* bit-width, while still considering two's complement representation for negative numbers; and 2) A more convincing method considers a bit-level composition of the RTL structure and shows that the equivalence can be proven for large operand bit-widths, at least up to 256. Note: The *bit-level* RTL structure should not be confused with a *gate-level model*, since the arithmetic and logic operators are still defined

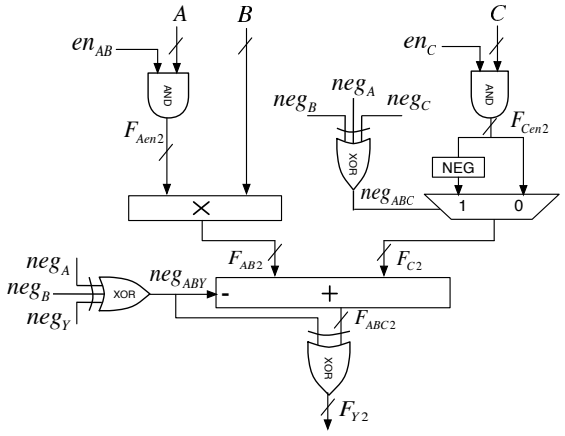


Fig. 2. Integer ALU - final RTL design

at the register transfer level. In a separate model (beyond the scope of this paper) we can also demonstrate the correctness of our approach to gate-level designs.

A. Word-level Verification

In this model, the unsigned word-level operand X is represented simply as variable X (positive number) or as $-X$ (negative number), regardless of the number of bits (assuming no overflow).

Original Design (F_{Y1})

The first level includes three identical modules, each composed of a *negator* (NEG) and a multiplexer (MUX) to select the operand in a positive or in a negated form. The output of the first MUX, associated with operand A, is

$$F_{A1} = (1 - neg_A)A + neg_A(-A) = A \cdot (1 - 2 \cdot neg_A)$$

Note that for $neg_A = 0$, $F_{A1} = A$; and for $neg_A = 1$, $F_{A1} = -A$, as required. Similar expressions are derived for modules with inputs B and C, and outputs F_{B1}, F_{C1}, respectively. Next design level includes a multiplier followed by an enable signal en_{AB}, producing $F_{ABen1} = en_{AB}(F_{A1} \cdot F_{B1})$ and $F_{Cen1} = en_C \cdot F_{C1}$.

The next level has an adder with inputs $F_{AB1},\ F_{C1}$

$$F_{ABC1} = F_{AB1} + F_{C1}$$

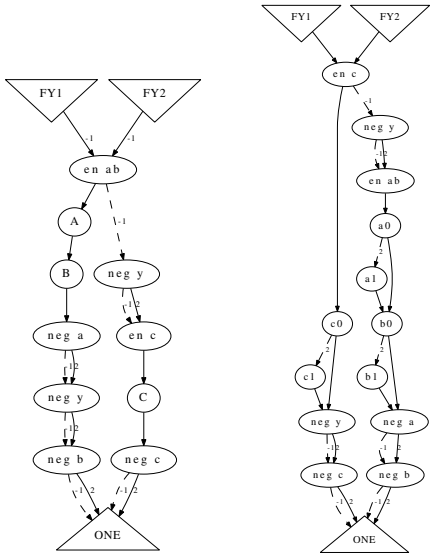
The lowest level has a negator gate for F_{ABC1}, controlled by neg_Y

$$F_{Y1} = F_{ABC1}(1 - 2 \cdot neg_Y)$$

The entire set of such equations is written into the TDS system [13][1] and represented by a canonical, word-level diagram,

[1]TDS is a system for behavioral transformation of designs specified at behavioral or RTL level. It transforms the initial design specifications into an optimal DFG prior to high-level synthesis. It is used here to represent the design in canonical form using Taylor Expansion Diagram (TED).

TED [2]. The diagram automatically represents the function in normal factored form in terms of the primary inputs, as shown in Figure 3(a).



(a) (b)

Fig. 3. TED representation of the integer ALU design: (a) word-level model; (b) bit-level model

Final Design (F_{Y2})

The transformed design is shown in Figure 2, where

$$neg_{ABC} = neg_A \oplus neg_B \oplus neg_C$$

$$neg_{ABY} = neg_A \oplus neg_B \oplus neg_Y$$

Translation of the Boolean operator $\oplus$ (XOR) into an algebraic expression can be done using the following relation:

$$x \oplus y = x + y - 2 \cdot x \cdot y \tag{1}$$

By applying this formula to the above equations, we obtain:

$$neg_{AB} = neg_A + neg_B - 2 \cdot neg_A \cdot neg_B$$

$$neg_{ABC} = neg_{AB} + neg_C - 2 \cdot neg_{AB} \cdot neg_C$$

$$neg_{ABY} = neg_{AB} + neg_Y - 2 \cdot neg_{AB} \cdot neg_Y$$

With this, the remaining part of the design can be described by the following set of expressions:

$$F_{Aen2} = en_{AB} \cdot A$$

$$F_{AB2} = F_{Aen2} \cdot B$$

$$F_{Cen2} = en_C \cdot C$$

$$F_{C2} = F_{Cen2} \cdot (1 - 2 \cdot neg_{ABC})$$

$$F_{ABC2} = F_{AB2} + F_{C2} - neg_{ABY}$$

where neg_{ABY} is a *binary* variable. The same signal is then applied to an XOR to conditionally flip the bits of the word-level signal F_{ABC2} computed by the add/sub module. The algebraic model for XOR shown in (1) does not apply to such bit-wise operations on a word-level signal, and needs to be suitably modified. Specifically, it can be modeled as a MUX, shown in Figure 4. When $neg_{ABY} = 0$, the output of the adder ($F_{AB2} + F_{C2} - 0$) is passed to F_{Y2}; and when $neg_{ABY} = 1$, the adder's output, ($F_{AB2} + F_{C2} - 1$), is bit-wise complemented by an XOR. To model this, we use the standard relation between the bit-wise complement and a word-level complement/negation, $-X = \overline{X} + 1$. This, as already shown in [1], can be rewritten as $-(X-1) = \overline{X-1} + 1$, which, in turn, implies that

$$-X = \overline{X-1} \tag{2}$$

We can now model the XOR and a MUX with inputs X and $-X$, where $X = F_{AB2} + F_{C2}$, as follows:

$$F_{Y2} = (1 - neg_{ABY})X + neg_{ABY}(-X) = X(1 - 2 \cdot neg_{ABY})$$

which is similar to the negator developed earlier.

Substituting $X = F_{AB2} + F_{C2}$ in the above equation gives the following model for the resulting MUX (c.f. Figure 4).

$$F_{Y2} = (F_{AB2} + F_{C2})(1 - 2 \cdot neg_{ABY})$$

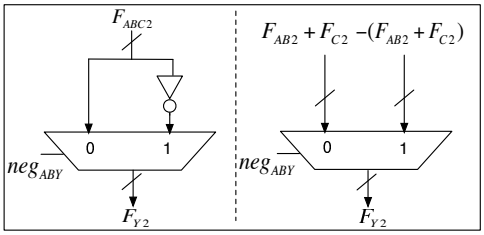


Fig. 4. Modeling the word-level XOR as MUX.

B. Bit-level Verification

To perform RTL verification at the bit-level, we must consider the bit composition of each of the word-level signals. This is done by expressing each n-bit unsigned number X by its binary encoding: $X = \sum_{i=0}^{n-1} 2^i x_i$. The negative number ($-X$) is represented using two's complement model as $-X = 2^n - X$. Specifically, we express the word-level input A using binary encoding $A = 2^{n-1}a_{n-1} + \cdots + 2a_1 + a_0$, and similarly for inputs B and C. The system of equations derived in Section III-A, together with the binary-encoded inputs (and intermediate signals, as needed) is then used to generate the final canonical TED representation.

Figure 3(b) illustrates this approach for a simple case of 2-bit operands, and demonstrates that both designs represent the same function, hence are equivalent. The 2-bit case is used here for illustration only, since the generated TED diagrams for larger cases would be too big for paper reproduction. However, the results shown in Section IV clearly demonstrate that this approach can be used to solve the equivalence verification problem for this design at least up to 256-bit operands.

IV. Results

In this paper, the TED representation was used simply to illustrate the concept of symbolic RTL verification rather than as a robust method to solve the equivalence verification problem. Nevertheless, TED, in addition to providing the word-level symbolic solution, can easily handle the Integer ALU design with up to 26-bit operands (beyond which the internal memory management is not efficient). The CPU runtime for such solutions is shown in Table I. As we can see, the solution can be obtained within fractions of a second. The experiments were run on a PC with an Intel Processor Core i5-3470 CPU 3.20GHz x4 and 15.6 GB of memory.

An alternative, and a more efficient solution is based on an approach that computes (i.e., extracts) the function performed by the design by rewriting the symbolic expressions of a design from the primary outputs to primary inputs. Such an approach has been used in our earlier work [14] in the context of arithmetic bit-level (ABL) networks, but applies verbatim here. In this approach, the specification polynomial (called *input signature*) of the given design is computed from the word-level outputs (called *output signature*). It is expressed in terms of the primary input variables: the operands A, B, C, control signals neg_i, and other configuration signals. Such a computed signature is then compared to the input signature obtained in a similar manner from the other design. This is done by running both designs on two cores of the same processor and checking if $F_{Y1} - F_{Y2} = 0$. As shown in the table (column *Function-Extract)*, this approach is scalable: it can solve the bit-level Integer ALU for operands with at least 256 bits in a matter of seconds.

In comparison, in [1] a commercial combinational RTL equivalence tool, Hector, was used to formally verify equivalence of a 16-bit instance of this ALU design. Solving this problem with Hector required 16-way case splitting (performed by hand) and solving the 16 simpler problems corresponding to some combinations of the configuration bits. The CPU time of 8 seconds reported in [1] cannot be used to compare to our results since the parameters of the computing platform were not given. Larger design were not attempted in [1], claiming an increased difficulty experienced by the solver.

Operand size	TDS [13]		Function-Extract	
	CPU (sec)	Mem (MB)	CPU (sec)	Mem(MB)
4	0.01	3.4	0.01	2.4
8	0.03	4.5	0.03	4.0
16	0.06	8.9	0.11	9.6
26	0.19	18.2	0.32	22.0
32	-	-	0.48	32.3
64	-	-	1.93	124.8
128	-	-	8.07	494.3
256	-	-	34.66	1984.5

TABLE I. CPU time and memory usage for solving integer ALU

V. Conclusions

The equivalence verification method presented here relies on computing symbolic expressions for the circuit outputs. It is applicable to solving equivalence checking problems on different abstraction levels, from word-level to bit-level. Although conceptually simple, this method has proven to be efficient in solving problems that can be expressed as symbolic polynomials with word-level or bit-level variables. Our current work extends this concept to gate-level designs.

VI. Acknowledgment

This research is supported by a grant from the National Science Foundation, Award No. CCF-1319496.

References

[1] Drane Theo and Jain Himanshu, "Formal Verification and Validation of High-level Optimizations of Arithmetic Datapath Blocks," in *SNUG Awards 2011*. Synopsys, 2011.

[2] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs," *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.

[3] D.K. Pradhan and ed. I.G. Harris, *Practical Design Verification*, Cambridge University Press, 2009.

[4] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver Technology for System-level to RTL Equivalence Checking," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09*. IEEE, 2009, pp. 196–201.

[5] Alfred Koelbl and Carl Pixley, "Constructing Efficient Formal Models from High-level Descriptions using Symbolic Simulation," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 645–666, 2005.

[6] N. Bombieri, F. Fummi, V. Guarnieri, G. Pravadelli, and S. Vinco, "Redesign and verification of rtl ips through RTL-to-TLM abstraction and tlm synthesis," in *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on*. IEEE, 2012, pp. 76–81.

[7] H. Sohofi and Z. Navabi, "Assertion-based Verification for System-level Designs," in *15th International IEEE Symposium on Quality Electronic Design (ISQED),* , 2014, pp. 582–588.

[8] Robert Beers, "Pre-RTL Formal Verification: an Intel Experience," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. IEEE, 2008, pp. 806–811.

[9] H. Mangassarian, B. Le, and A. Veneris, "Debugging RTL using Structural Dominance," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 1, pp. 153–166, 2014.

[10] N. Shekhar, P. Kalla, and F. Enescu, "Equivalence Verification of Polynomial Data-Paths Using Ideal Membership Testing," *IEEE Trans. on Computer-Aided Design*, vol. 26, no. 7, pp. 1320–1330, July 2007.

[11] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G.M. Greuel, "Stable: A new QF-BV SMT Solver for Hard Verification Problems combining Boolean Reasoning with Computer Algebra," in *DATE*, 2011, pp. 155–160.

[12] S. Vasudevan, V. Viswanath, R. W. Sumners, and J. A. Abraham, "Automatic Verification of Arithmetic Circuits in RTL using Stepwise Refinement of Term Rewriting Systems," *IEEE Trans. on Computers*, vol. 56, no. 10, pp. 1401–1414, 2007.

[13] M. Ciesielski, D. Gomez-Prado, Q. Ren, J. Guillot, and E. Boutillon, "Optimization of Data-Flow Computation using Canonical TED Representation," *IEEE Trans. on Computers*, Sept. 2009, pp. 1321–1333. TDS system online: www.ecs.umass.edu/ece/labs/vlsicad/cadlab/tds/TDS.html

[14] M. Ciesielski, W. Brown, D. Liu, and A. Rossi, "Function Extraction from Arithmetic Bit-level Circuits," in *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*. IEEE, 2014, pp. 356–361.