A Universal Macro Block Mapping Scheme for Arithmetic Circuits

Xing Wei
The Chinese University of Hong Kong, Hong Kong
xwei@cse.cuhk.edu.hk

Yi Diao
The Chinese University of Hong Kong, Hong Kong
ydiao@cse.cuhk.edu.hk

Tak-Kei Lam
The Chinese University of Hong Kong, Hong Kong
tklam@cse.cuhk.edu.hk

Yu-Liang Wu
Easy-Logic Technology Limited
ylw@easylogic.hk

ABSTRACT

A macro block is a functional unit that can be re-used in circuit designs. The problem of general macro block mapping is to identify such embedded parts, whose I/O signals are unknown, from the netlist that may have been optimized in various ways. The mapping results can then be used to ease the functional verification process or for replacement by more advanced intellectual property (IP) macros. In the past literatures, the mapping problem is mostly limited to the identification of a single adder or multiplier with I/O signals given, which is already NP-hard. However, in today's typical arithmetic circuits (like digital signal processing (DSP) applications), it is not unusual to have combinations of arithmetic operators implemented as macro blocks for performance gain. To solve this new practical mapping problem, we propose a flow to identify and build a forest of one-bit-adder trees using structural information and formal verification techniques, followed by algorithms that locate macro boundaries and I/O signal orders. Experimental results show that our algorithm is highly practical and scalable. It is capable of identifying any combinations of arbitrary adders and multipliers such as $(a + b) \times c$ and $a \times b + c \times d + e \times f$, where each operand is a multi-bit constant or variable. Most of the benchmarks in ICCAD 2013 CAD Contest [1] can be well handled by our algorithm.

Keywords

Adder, Multiplier, Technology mapping, Arithmetic logic

1. INTRODUCTION

Macro blocks are the building components in integrated circuits (ICs). They implement specific functions that range from basic logic to complex arithmetic operations. Standard cells in application-specific integrated circuits (ASICs) can be regarded as a kind of macro blocks which implement primitive functions. The paradigm of macro blocks is commonly applied in field-programmable gate arrays (FPGAs) as well as in ASICs. For example, some FPGA contains embedded multipliers or digital signal processing (DSP) units.

The use of more complex macro blocks is getting popular because the scale of integrated circuits keeps on growing continuously and rapidly. From an engineer's point of view, dividing a circuit into modules and then macro blocks is a more manageable practice to encourage the reuse of resource and knowledge. Since a macro block implements a specific function, it can be heavily and strategically optimized for timing, area and power consumption.

Engineers may want to synthesize designs using macro blocks that implement complex functions. They may also want to verify the synthesized designs. At one extreme, they may even want to replace a certain portion of a synthesized, placed and routed netlist using new macro blocks built with more advanced technologies. These tasks are not straightforward. The main difficulty is to extract the boundaries of sub-circuits. Since the scales of circuit design and macro blocks are now very large, mapping macro blocks is difficult and has become an infeasible tasks for engineers to perform manually. Therefore, it is valuable to develop a robust methodology that can utilize macro blocks automatically and efficiently.

Arithmetic operators are essential to all computations. In this paper, we focus on the mapping problem of identifying adder-based arithmetic logics, which can be used to construct various combinations of adders and multipliers in Boolean combinational netlists. Our contributions to this problem include:

1. an efficient method to identify a forest of one-bit adder trees of the given arithmetic structure;
2. a novel method to locate the inputs and outputs together with their orderings of multiple-bit adders and multipliers;
3. a scalable equivalence checking scheme based on SAT and polynomial algebras;
4. the capability to identify arbitrary combination of basic arithmetic functions such as $(a + b) \times c$ or $a \times b + c \times d + e \times f + ...y \times z$.

978-3-9815370-4-8/DATE15/©2015 EDAA

Our approach has been tested against the benchmarks released for the ICCAD Contest 2013 by Cadence, which include macros with complex combinations of additions and multiplications. Comparing with the prior mapping works which were limited to identifying a single adder or multiplier under known I/O pins, our work provides a further step to solve the increasingly complicated mapping problem.

1.1 Previous Works

Macro block mapping or general technology mapping cannot be achieved without efficient verification methods. Equivalence checks are necessary to match a part in the given netlist against a database of macro blocks. In some cases, equivalence checking can be accomplished by structural traversals in gate-level. In general, however, the netlist has already been highly optimized and the implementation details of macro blocks are not available, hence applying structural analysis alone may not be sufficient for macro mapping.

Gate-level simulation techniques were developed and applied as a general verification method. Since they do not rely on structural information, they provide higher coverage than pure structural analysis techniques. The major short coming of simulation-based techniques is that the run time is reasonable only for small circuits. Formal verification methods can provide higher coverage than simulation-based techniques within the same amount of time. Therefore, they are widely adopted for equivalence checking in the industry.

One of the very first formal verification methods was reported in [2]. Binary decision diagrams (BDDs), which are canonical form of logic functions, were later proposed in [3] and were found to be powerful for equivalence checking. Using BDDs in equivalence checking is to represent the netlist function and the specification using BDDs and to compare the respective BDDs. BDDs are also used as a tool for logical reasoning only. For instance, logical reasoning was done using BDDs in the first general method that could verify large multipliers with different architectures [4]. Techniques for solving Boolean satisfiability (SAT) problems have been greatly advanced in this decade . Many high quality SAT-solvers such as MiniSat [5] are now available. It was proven in [6] that SAT-based methods are robust and flexible enough to be used in equivalence checking, and are more scalable than BDD-based methods.

Nevertheless, it has been known that general formal verification methods are unsuitable for arithmetic logics. In the recent work [7], the experimental results show that the performance of general SAT-based methods in verifying multipliers was not as good as the authors' algebraic approach. Hence, formal verification methods dedicated to arithmetic logics (especially multipliers), which rely on mathematical models, structural analysis and characteristics of the arithmetic operations have been developed [4, 7–14].

Equivalence checking is a part of the macro mapping process. The latter is more difficult because what should be verified is not known in advance. In the classic approach [15, 16], arithmetic blocks are detected from the high-level description (register-transfer level (RTL)) of the circuit. Then, the mapped arithmetic blocks are expanded into their Boolean networks. These Boolean sub-networks are labelled. They may be restored back to their corresponding arithmetic macro blocks by using the labels depending on the optimization result of the Boolean network of the whole circuit. The more general and advanced mapping approaches are those that apply the mentioned verification methods. There is a trend that polynomial algebras is becoming increasingly popular in macro mapping and arithmetic logic synthesis [17–19].

1.2 Problem Definition

The problem of macro mapping is defined as follows. Given a combinational gate-level design D and a library of macro blocks $B = \{B1, B2, B3, ...\}$ specified in RTL. The objective is to utilize as many macro blocks as possible such that the number of primitive gates left unmapped is minimized in the mapped design D. The problem in general is to

$$\text{minimize} \sum_{g \notin G} \text{gate size of } g \tag{1}$$

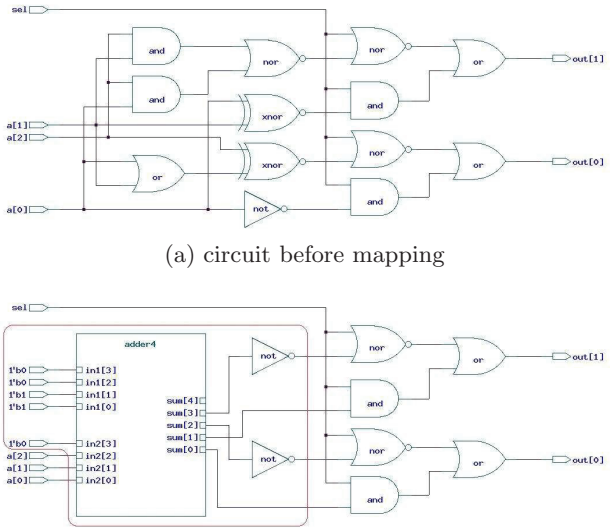
$$\text{subject to } (D' = D - G + M) \equiv D,$$

G is the set of primitive gates mapped

$M \in B$ is the set of macro blocks utilized

D is the given design

D' is the mapped design



(a) circuit before mapping

(b) circuit after mapping

Figure 1: An example of macro mapping process

Figure 1 is a simple example given by ICCAD contest [1]. Assume that Figure 1(a) is the original circuit and we have a 4-bit ADDER2 macro. After the mapping process as shown in Figure 1(b), we can see that 6 functional gates are mapped into the macro and well separated from other random logics.

2. ALGORITHM FLOW

An overview of our approach of identifying adders and multipliers is presented first with the details of each step explained subsequently. Due to the page limit, in this paper we only give typical adder and multiplier constructions (e.g. Carry Save Adder based) to illustrate our ideas, which without

the loss of generality can be extended to solve other kinds of multipliers such as Booth-encoded and constant multipliers.

In Algorithm 1, the first step is to recognize logic gates implementing an exclusive-or (XOR) function, which can be formed by AND/OR primitives or by a stand-alone XOR primitive. A forest of XOR trees can thus be formed with each XOR locating a one-bit half adder and each XOR tree representing an addition logic of the same bit-weight, and the forest in turn can represent either an adder or multiplier or even an arbitrary complicated arithmetic structure built by a combination of both adders and multipliers.

Algorithm 1: Procedure *Identify_adder_multiplier*

input : a combinational gate-level circuit D,
RTL description of an adder or multiplier macro b
output: whether the macro b is identified

```
1 begin
2     Build XOR trees;
3     Find the connections among XOR trees;
4     boundary ← Determine candidate macro boundary;
5     if boundary is incomplete then
6         FindCompleteBoundary(boundary);
7     if SubCircuit(boundary) ≡ b then
8         return identified;
9     return not identified;
```

Each XOR tree corresponds to the addition of digits in the same bit-weight position, and can be considered as a column of additions of several numbers for an adder or for a multiplier. The carry out signals of each XOR tree can also be the inputs of the XOR tree of the next bit-weight. The arithmetic structure and bit-weights of the forest of XOR trees are then recognized. Based on the type of the macro, whether it is an adder or a multiplier, the XOR forest is expanded accordingly. If it is a multiplier, the inputs of the XOR forests are the bit-products of the multiplier. Once the implementation of these bit-products have been determined, the boundary of the candidate macro can then be solved.

The orders of the input and output pins of the candidate macro are determined using the information obtained during the construction of the XOR forest. In our method, a system of integer linear equations is set up using the bit-weight indices of the XOR trees. This system of integer linear equations is then solved to determine the pin orders

In case the boundary found is not complete, local searches for the missing outputs using verification methods will be applied. The sub-circuit circled by the boundary may need to be verified if its function cannot be ascertained based on the information reflected by the XOR trees.

2.1 XOR Trees Construction

Figure 2 illustrates a construction of XOR trees, where the cut enumeration method [20, 21] can be used to provide a comprehensive coverage in identifying XOR function. In this method, speed is not compromised because the enumeration process is local and the cut size required is only two.

2.2 XOR Forest Construction

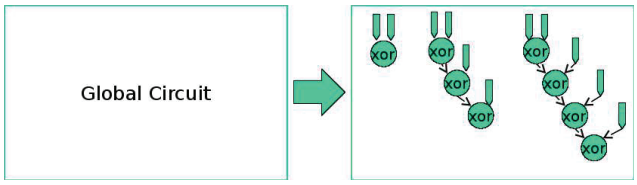


Figure 2: Construction of XOR trees

Having found the separated XOR trees is not enough to identify adders and multipliers. The connection hierarchy between the XOR trees and the carry out signals generated by every XOR tree would need to be located and analyzed, and then an XOR forest can be constructed as shown in Figure 3.

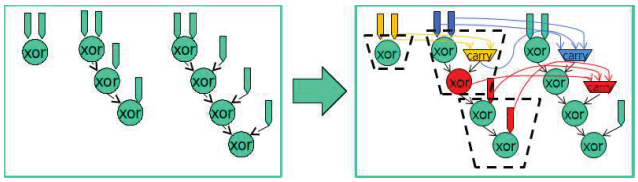


Figure 3: Construction of an XOR forest

The identification of carry out signals is an essential process to connect the XOR trees. With reference to Figure 4, we may have three tries at each XOR function since the function may be a complete one-bit half adder or just a part of an one-bit full adder. In each try, if the carry out signal can be found somewhere in the netlist and is just the input of another XOR tree, the connection between these two XOR trees can be built and whether the tested function is a half adder or part of a full adder is also determined.

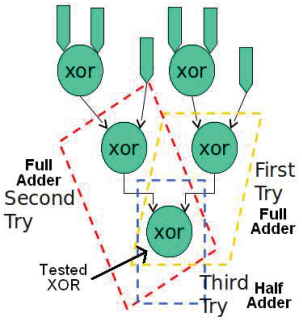


Figure 4: Identification of carry out signals

In Figure 5, the different bit-weight indices of XOR trees are illustrated. The tree at bit-weight index n is fed with the carry out signals from bit-weight index $n-1$. If an XOR tree whose inputs are not the carries from other XOR trees, it is at index 1 for a multiplier or at 0 for an adder. This information of bit-weight indices is crucial for the later calculation deciding the input and output pin orders of the candidate macro block.

2.3 Input Identification

As illustrated in Figure 6, the boundary locating for a multiplier macro is relatively more complicated. Since the inputs

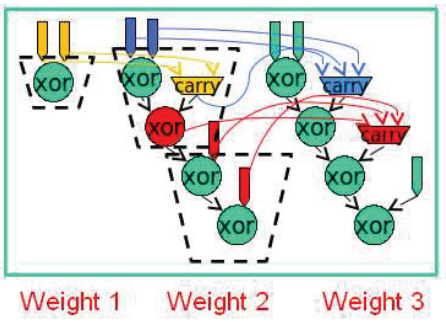


Figure 5: Levelization of XOR trees

to the XOR forest are the bit-products, the signals of the multiplicand and multiplier are not yet known. For example, in the Figure 6 multiplier, the two yellowish and blue AND gates need to be located to determine the set of input signals $\{x0, x1, x2, x3\}$ of the multiplier.

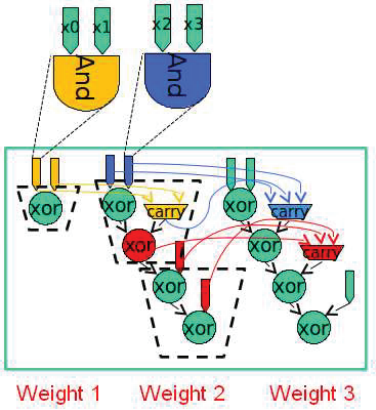


Figure 6: Input identification for multipliers

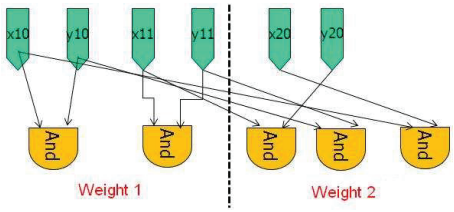


Figure 7: Determination of input indices

Considering the example in Figure 7, each bit-product has a pair of input pins. Based on our observation of the general multiplication mechanism, we derived a method to order the input pins of the multiplier.

LEMMA 1. *The sum of the bit-weights of a bit-product's input pins is the same as the bit-weight index of the XOR tree which the bit-product is associated with.*

For examples, the sum of the indices of pins x_{10} and y_{10} in Figure 7 is 1, and that of pins x_{20} and y_{20} is 2. For a XOR forest with n bit-weights where the indices of the jth input signal pairs at bit-weight index i are $\{x_{ij}, y_{ij}\}$, the following system of linear equations can then be formulated:

$$\begin{aligned} x_{10} + y_{10} &= 1 \\ x_{11} + y_{11} &= 1 \\ x_{20} + y_{20} &= 2 \\ &\dots \\ x_{nj} + y_{nj} &= n \end{aligned} \tag{2}$$

The equations have a special property that for each equation, there are exactly two variables with coefficients being always equal to 1. The complexity of solving this kind of system of equations has been proven to be $O(n)$. Hence, we have the following lemma:

LEMMA 2. *Given an XOR forest representing a multiplier function, the complexity of solving the indices of all multiplier inputs is $O(n)$.*

PROOF. *The problem of solving indices of multiplier inputs can be converted into solving a system of equations which has special properties as stated above. Thus the complexity of solving the indices is just $O(n)$ which is equal to that of solving a system of above equations.* □

After calculating the bit index of each input pin, they can be easily classified into the two buses of input signals A and B of the multiplier. Figure 8 shows a simple example. The number of each of the AND gates' input pins represent the bit index of the pin. Suppose the leftmost pin is classified into bus B. Then, another input to the AND gate must be a signal in bus A. This reasoning process is performed on each of the input pin pairs to decide the bus group of each pin.

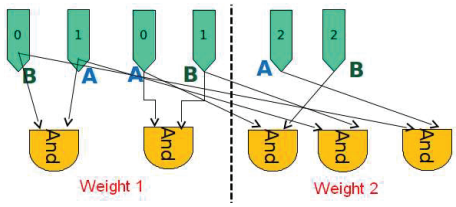


Figure 8: Determination of input buses

2.4 Output Identification

An XOR forest is defined to be a "**complete** forest" for an N-bit multiplication function if:

1. the maximum bit-weight index of the XOR forest is $2N - 1$;
2. all bit-products can be identified for all bit-weights;
3. all carry out signals are part of the input signals of the next bit-weight XOR tree except the XOR tree of the highest bit-weight.

The outputs of the **complete** XOR forest with or without additional inverters are the output pins of the macro block. By tracing the structure of the XOR trees, whether the outputs have to be negated can be deduced. It was in fact shown in our experiments that over 50% output pins of the candidate macros can be identified in this way and need not be verified.

LEMMA 3. *Given an N-bit multiplication macro to be mapped, if its complete XOR forest can be identified, formal verification of the mapped result can be waived.*

PROOF. *For convenience we suppose* $out[2n-1:0] = a[n-1:0] \times b[n-1:0]$. *The output function at bit-weight index 1 of a complete XOR forest is* $a_0b_1 + a_1b_0$ *which is obviously equivalent to* $out[1]$. *Assume that the output function at bit-weight index* k *of a complete XOR forest is equivalent to* $out[k]$, *the function at* $k+1$ *is calculated as the sum of all carry out signals from* k *and all bit-products at* $k+1$, *which is just the function of* $out[k+1]$. *By mathematical induction, formal verification of the mapped result can be waived if the complete XOR forest is identified.* □

There are circumstances in which the XOR forest is so incomplete that the function implemented cannot be verified. Formal verification methods such as SAT are then required to identify the missing output pins of the target macro. Our experiments have shown that only a small portion of the output pins cannot be identified, and the missing output pins can be located by SAT within negligible time.

Table 1: An example of long multiplication and addition

			a_1	a_0
$\times$			b_1	b_0
			a_1b_0	a_0b_0
$(+)$		a_1b_1	a_0b_1	
		a_1b_1	$a_1b_0 + a_0b_1$	a_0b_0
$+$			c_1	c_0
	ϕ	a_1b_1	$a_1b_0 + a_0b_1 + c_1$	$a_0b_0 + c_0$

In our verification method, the formulae of the output pins of the target macro are calculated. For example, suppose the operands of a multiplier-adder $(a \times b + c)$ in bit vector form are a_1a_0, b_1b_0 and c_1c_0, where the bit index increases from the least to the most significant bit. The last row in Table 1 lists the formulae for each bit of the 4-bit result.

For each missing output pin of the target macro whose formula is not ϕ, its identical signal can be efficiently found by formula matching. The formulae of the candidate signals are computed by modelling the Boolean network as a network of one-bit half adders. Suppose an one-bit half adder has inputs a and b. Let the sum and the carry out be s and c respectively. The inputs and the outputs of an one-bit half adder has the following relationship (in base two):

$$c = a \times b \qquad (3)$$
$$s = a + b$$

By applying Equation 3 recursively, the polynomial formulae of the candidate signals can be deduced.

The above method cannot be applied to those missing output pins whose formulae are ϕ. This is because the ϕ represents an empty formula and is not unique. For example, the formula of a ground signal in the circuit is also ϕ by definition. Nevertheless, this fast polynomial algebraic method can still be employed in that case to filter candidates which will be verified by SAT-based methods. Since the macro outputs with formula ϕ are always close to or at the most significant bit positions, the corresponding SAT problem instances are relatively less complex. Thus, the integrated algebraic and SAT-based approach is still much superior to pure SAT-based approach.

3. COMPLEX MACRO IDENTIFICATION

Our algorithm can be extended naturally to handle complex arithmetic macros such as the arithmetic sum of products: $a \times b + c \times d$, $a \times b + c \times d + e \times f$. For these kind of macros, the XOR forest can be built in the same way as that of an $a \times b$ macro. The only difference is that there are more inputs (bit-products) for each bit-weight.

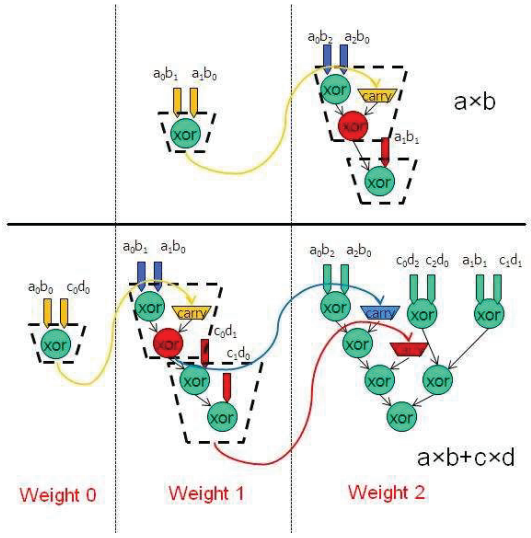


Figure 9: Difference between $a \times b$ and $a \times b + c \times d$ macros

Figure 9 is an example showing the differences between macros $a \times b$ and $a \times b + c \times d$. An $a \times b$ macro at bit-weight index **1** has 2 inputs (bit-products) a_0b_1 and a_1b_0, while an $a \times b + c \times d$ macro has 4 inputs a_0b_1, a_1b_0, c_0d_1, and c_1d_0.

The remaining flow is similar to what we have described in the previous section. After constructing an XOR forest, a system of integer linear equations is built to decide the index number of each input, which is followed by the process of classifying all inputs into the appropriate buses.

4. EXPERIMENTAL RESULTS

The benchmark set released by Cadence's logic verification team [1] for the ICCAD 2013 contest was adapted to evaluate our approach. Each of these benchmarks is a gate-level circuit design to be mapped to a set of common macro blocks including adders, multipliers, adder-multipliers, multiplier-adders, multiplexers and multi-bit primitive gates. (In the contest, each of the macro blocks can be used only once, but our algorithm can do mapping without such constraint.) The operands to the arithmetic macros may be either signed or unsigned variables or just constants. In our experiments, we targeted to identify only the former four types of adder-based macros so as to measure the effectiveness of our arithematic mapping framework more accurately. Table 2 shows the details of the benchmarks. Our framework was written in C++. The experiments were performed on a laptop (duo 3GHz and 3GB RAM). The results were verified by Cadence's verification tool "LEC".

Table 2: Benchmark Information [1]

Case*	Design size	Contained arithmetic macros
ut1	33	$a+b$
ut3	430	$a+b$
ut5	1282	$a \times b$
ut7	851	$a \times b$
ut8	685	$a+b+c+d$
ut9	3599	$a \times b$
ut10	1026	$a+b$
mt1	3535	$a \times b, (a+b) \times c$
mt2	3313	$a \times b + c \times d$
mt4	65115	$a+b$
mt5	137805	$a \times b$
ht1	15047	$a+b, a \times b, a \times b + c \times d, a \times b + c \times d + e \times f$
ht2	130946	$a \times b$
ht3	10809	$a+b, a+b+c, a \times b$
ht4	17526	$a \times b, a \times b + c \times d, a \times b + c \times d + e \times f$
ht5	28061	$a+b, a+b+c, a \times b, a \times b + c \times d$
total	420063	

*Cases without arithmetic macros are excluded.

Table 3: Mapping Results

	Our algorithm			1st*	
Case	#map macros	#map gates	CPU time(s)	#map macros	#map gates
ut1	1	20	1	1	20
ut3	1	127	1	1	127
ut5	1	1121	1	1	1121
ut7	1	481	1	1	481
ut8	2	594	1	2	594
ut9	1	667	4	1	667
ut10	3	806	1	3	806
mt1	2	3279	101	1	1735
mt2	2	1416	6	2	1416
mt4	16	721	75	16	721
mt5	0	0	302	0	0
ht1	6	4922	118	2	4713
ht2	0	0	80	0	0
ht3	6	3774	20	6	3774
ht4	5	9394	41	5	9394
ht5	4	1221	304	0	0
total	51	28543	1057	42	25569
ratio	**1.21x**			**1**	

*1st: the first place winner of ICCAD Contest 2013

We compare our results with those of the first place winner of the ICCAD contest. The results are listed in Table 3. Columns **#map(ped) macros** list the number of macros identified by our algorithms and the first place winner (***1st***) respectively. For each case, our algorithm gains no less than what ***1st*** achieved. Our algorithm can find 21% more macros than ***1st*** in total. Note that our algorithm is capable of finding multiple macros within one case.

Columns **#map(ped) gates** indicate the size of macros mapped. Obviously, if we can identify a macro with more inputs (or larger size), more gates can be mapped into the macro (e.g. the number of the gates mapped into a 24-input addition macro is almost double than that mapped into a 12-input addition macro). Column **CPU time** shows the runtime for each benchmark. The CPU time depends on the design size as well as the number and size of the macros. Our algorithm can finish in around 100 seconds for most cases and in 300 seconds at most.

5. CONCLUSION

Although there were works trying to handle the mapping problem of arithmetic functions, the difficult process of locating macro boundary was often not discussed in details. In this paper, we propose a universal scheme to identify major macro blocks for arithmetic logics. In order to allow a normalised comparison by future works, our algorithms were tested against the benchmarks prepared by ICCAD Contest 2013. The results show that our approach can even outperform the first place winner of the contest.

6. REFERENCES

[1] ICCAD Contest, "Technology mapping for macro blocks." `http://cad_contest.cs.nctu.edu.tw/CAD-contest-at-ICCAD2013/problem_a/`, 2013.

[2] G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean comparison of hardware and flowcharts," *IBM Journal of Research and Development*, vol. 26, no. 1, pp. 106–116, 1982.

[3] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, 1986.

[4] T. Stanion, "Implicit verification of structurally dissimilar arithmetic circuits," in *Computer Design, 1999. (ICCD '99) International Conference on*, pp. 46–50, 1999.

[5] N. Een and N. Sorensson, "An extensible sat-solver," in *Proc. SAT*, pp. 502–518, 2003.

[6] E. Goldberg, M. Prasad, and R. Brayton, "Using sat for combinational equivalence checking," in *DATE'01*, pp. 114–121, 2001.

[7] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski, "Algebraic approach to arithmetic design verification," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pp. 67–71, 2011.

[8] Y.-A. C. Randal E. Bryant, "Verification of arithmetic circuits with binary moment diagrams," in *DAC'95*, pp. 535–541, 1995.

[9] J.-C. Chen and Y.-A. Chen, "Equivalence checking of integer multipliers," in *ASP-DAC'01*, pp. 169–174, 2001.

[10] Y.-T. Chang and K.-T. Cheng, "Induction-based gate-level verification of multipliers," in *ICCAD'01*, pp. 190–193, 2001.

[11] D. Stoffel and W. Kunz, "Verification of integer multipliers on the arithmetic bit level," in *ICCAD'01*, pp. 183–189, 2001.

[12] O. Sarbishei, B. Alizadeh, and M. Fujita, "Arithmetic circuits verification without looking for internal equivalences," in *Formal Methods and Models for Co-Design, 2008. 6th ACM/IEEE International Conference on*, pp. 7–16, 2008.

[13] C.-Y. Lai, S.-L. Huang, and K.-Y. Khoo, "Improving constant-coefficient multiplier verification by partial product identification," in *DATE'08*, pp. 813–818, 2008.

[14] J. Lv, P. Kalla, and F. Enescu, "Efficient gröbner basis reductions for formal verification of galois field arithmetic circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 9, pp. 1409–1420, 2013.

[15] M. Quayle and C.-L. Huang, "Complex operator synthesis," in *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pp. 514–517, 1994.

[16] H. Savoj, D.-J. Wang, D. Hoang, and C.-L. Hiang, "Optimal complex operator mapping," in *ASIC Conference and Exhibit, 1997. Proceedings., Tenth Annual IEEE International*, pp. 215–218, 1997.

[17] J. Smith and G. De Micheli, "Polynomial circuit models for component matching in high-level synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 6, pp. 783–800, 2001.

[18] A. Peymandoust and G. De Micheli, "Application of symbolic computer algebra in high-level data-flow synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 22, no. 9, pp. 1154–1165, 2003.

[19] S. Ghandali, B. Alizadeh, Z. Navabi, and M. Fujita, "Polynomial datapath synthesis and optimization based on vanishing polynomial over z2m and algebraic techniques," in *Formal Methods and Models for Codesign 2012 10th IEEE/ACM International Conference on*, pp. 65–74, 2012.

[20] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. Field-Programmable Gate Arrays*, 1999.

[21] N. Een, "Cut sweeping," *Cadence Technical Report*, 2007.