Automatic Word-level Abstraction of Datapath

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

***Abstract*—Abstracting word information from gate-level designs is essential for formal verification, technology mapping and hardware security applications. In this paper, we present a novel method to abstract the word-level information from arithmetic gate-level circuits using a computer algebraic approach. The proposed technique translates the gate-level circuit into algebraic domain and applies algebraic rewriting to extract the arithmetic function. During the iterative rewriting, intermediate Pseudo-Boolean expressions are examined to identify word-level candidates. The proposed algorithm is able to abstract the word components from candidates and to reason about the word operation from the internal expressions. Successful experiments were performed on gate-level datapaths, including multipliers of up to 128-bit widths.**

Keywords*— *Word-level abstraction; Reverse engineering; Formal verification; Arithmetic Datapaths.

I. Introduction

One of the most challenging problems encountered in hardware design is functional verification of arithmetic datapaths. Many high-level verification techniques have been developed for high-level descriptions such as *Register transfer level* (RTL), or system-level, *SystemC* or *SystemVerilog*. However, hardware verification is still typically performed on low-level design representations, with gate-level netlists. The reason for this is that most high-level components, such as word declaration, modularization, function selection, etc., are flatten into netlists of Boolean gates by logic synthesis and technology mapping. Boolean logic techniques based on Binary Decision Diagrams (BDDs) or Binary Moment Diagrams (BMDs) and satisfiability (SAT) solvers cannot handle complex arithmetic designs on such a low level. For this reason, the gate-level datapath verification problem is challenging and remains open.

A straight-forward approach to formal verification problem is *Boolean satisfiability* (SAT). Several SAT solvers have been developed to solve the Boolean decision problem. Most of them are implemented using the computationally expensive DPLL decision procedure [1], which makes solving non-linear decision problems expensive [2]. Abstracting the word-level information from gate-level netlists, while maintaining the useful information about the control logic is a well known method that enable the high-level formulation. This approach, called *reverse engineering* [3], can provide significant improvement in performance and scalability. However, few works discuss the abstraction of gate-level designs.

In this paper, we present a novel method to abstract the word-level information from gate-level datapaths using a Computer Algebra approach. The proposed method is able to abstract high-level components from a set of gate-level netlists and generate corresponding word-level descriptions. At the same time, it is combined with a verification technique that enables scalable datapath verification. Our approach solves the problem in three steps: 1) Identify the word candidates during the function extraction process; 2) Classify word candidates as linear and non-linear expressions. 3) Abstract the word information and generate the correspondence between the words and gate-level netlist, and reason about the word operation.

II. Related Work

One of the most successful abstraction techniques is Counterexample-Guided Abstraction Refinement (CEGAR) [4]. It has been shown to be an effective paradigm in a variety of hardware and software verification scenarios [5][4][6]. Clarke et. al. [4] successfully demonstrated how to automate abstraction and refinement in the context of model checking for safety properties of hardware and software systems. In [7], CEGAR-based correspondence checking is applied to microprocessor datapaths. An automation tool UCLID [8] is able to abstract words into uninterpreted entities, and prove properties on the rest of the circuit. The authors of [9][10] introduced effective verification of out-of-order microprocessor based on UCLID. Moreover, *term-level* abstraction has been found to be especially useful in microprocessor design verification, using techniques such as term-level bounded model checking, correspondence checking, refinement verification, and predicate abstraction [11][12][13]. However, these techniques only apply to bit-vector behavioral RTL description.

The work proposed in [14][3] is the most relevant to ours. These present a variety of techniques to identify high-level components, such as adders and subtractors, which is applicable to arithmetic datapaths. The authors proposed two techniques to identify candidate words: *shape hashing* and *bitslice*. *Shape hashing* represents the backward reachable gates in feasible depths from a given wire, while the *bitslice* technique additionally finds similar wires by functional matching. They also addressed the problem of reverse engineering with extensive logic sharing using Quantified Boolean Formula (QBF). However, this technique is not efficient for large non-linear arithmetic operations since it requires *bit-blasting*. Additionally, for long cascaded word operation such as Multiply-Accumulator (MAC), it requires many *word propagate* iterations. Our work aims at overcoming these limitations.

978-1-4799-5341-7/16/$31.00 ©2016 IEEE

III. Word-level Abstraction

This section describes the implementation of our approach. First, we briefly review the *function extraction* technique. We then introduce an algorithm to identify word candidates and finalize the abstraction.

A. Function Extraction

The *function extraction* method computes a unique bit-level polynomial function implemented by the circuit directly from its gate-level implementation. This is done by rewriting the polynomial representing encoding of the primary outputs (the *output signature*) into a polynomial expressed in terms of the primary inputs (the *input signature*), using algebraic model of the internal gates. The method, described in our earlier work [2], uses an algebraic model of the circuit, with logic gates represented by algebraic expressions, while treating the circuit signals as strictly *Boolean* variables. The following algebraic model is used to represent basic Boolean gates.

$$\begin{aligned} \neg a &= 1 - a \\ a \wedge b &= a \cdot b \\ a \vee b &= a + b - a \cdot b \\ a \oplus b &= a + b - 2a \cdot b \end{aligned} \tag{1}$$

Functional correctness of the circuit is proved by successively rewriting the output signature, Sig_{out}, into a signature at the primary inputs (PI) and comparing it with the expected input signature, Sig_{in}. The rewriting process successively applies Eq. (1), followed by an algebraic simplification of polynomial terms to arrive at a unique algebraic expression. At each step of the procedure, an intermediate polynomial generated by the rewriting corresponds to a *cut* in the circuit, a set of signals separating primary inputs from primary outputs. Each intermediate polynomial is a pseudo-Boolean expression, a multi-variate polynomials with Boolean variables. Specifically, it represents an integer in Z_{2^n} with variables in Z_2.

This paper proposes a mechanism to identify a *word expression* from the intermediate expression during the algebraic rewriting. This rewriting is performed in reverse-topological order: once a given variable (output of a gate) is substituted by an algebraic expression of the gate inputs, it will be eliminated from the current cut expression and will never be considered again. That is, a variable is substituted for only after substituting all signals in its logical cone. Hence, we apply *Pre-ordering* before the abstraction, with the gate-level netlist sorted in reverse-topological order.

B. Word-level Abstraction

The outline of this technique is shown in Figure 1. The function of identifying the word-candidate expression is implemented by checking each intermediate expression during the algebraic rewriting. We define the *cut* as a $word\ cut$ if the intermediate expression is a $word\ expression.$

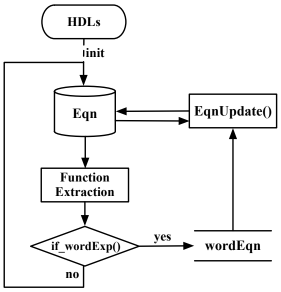

Fig. 1. Overview of function extraction based abstraction procedure.

Definition 1 *Word expression:* Assume a pseudo-Boolean expression $\mathcal{E}(c_1\mathcal{M}_1, c_2\mathcal{M}_2, ..., c_p\mathcal{M}_p)$ where $\mathcal{M}_i$ ($i = 1, 2, ..., k$) is a monomial and c_i is a coefficient. If there exists a subexpression $\mathcal{E}_{sub}(c_{i_1}\mathcal{M}_{i_1}, ...c_{i_n}\mathcal{M}i_n)$ such that it forms a binary word with $n \geq 2$, then this expression is a *word expression.*

The procedure starts at the initial expression for Sig_{out}. In Figure 1, the function *if_wordExp()* returns *true* if the expression is a candidate word. In this case the program will mark this *cut* as a *word cut*. Function *EqnUpdate()* finalizes the word-abstraction; it returns the word equation that corresponds to the gate expressions generated by rewriting between the $word\ cut$ and the starting *cut* at the current rewriting iteration. The next expression starts with the updated $word\ cut$, i.e. the abstracted word expression. This way, the correspondence between the gate-level equations and the abstracted word equations is established automatically.

In this work we assume that we have no knowledge of the type of arithmetic operators present in the design. The type of arithmetic operations we consider here includes the *finite-precision* integer arithmetic operators. It can be classified into linear or non-linear depending on the type of arithmetic operation. Additionally, we classify the algebraic expressions into *single-word* and *multiple-word* expression, based on the number of possible words. For example, *shifter* is a linear *single-word* expression; *adder* is a linear *multiple-word* expression; and *multiplier* is a non-linear *multiple-word* expression. The method for identifying the *word cut* is shown in Algorithm 1.

- **AssociateGraph** ($line$ 2)**:** It builds a graph $\mathcal{G}(V, A)$ of dependencies for each signal in the netlist. Each node in $\mathcal{G}$ records the extended logic cone and a list of direct fanouts for a given gate.
- **Rewriting** ($line$ 4)**:** The $Sig_{out} - Sig_{in}$ rewriting technique has been described in Section III-A and published in [2].
- **Algebraic Decomposition** ($line$ 5)**:** It searches for, and removes variables that are shared by every term of the expression (e.g. $(ab + ac) \rightarrow (b + c)$). The goal of this step is to make it easier to recognize the word signal that have control signals. The reason for doing this is that if a word w is controlled by a Boolean signal s, each term of the pseudo-Boolean expression that is always multiplied by s.

Algorithm 1 Identify Word Cut

```
Input: Algebraic gate-level equations
Output: Word equations with word cut mark
 1: $n \leftarrow \#.eqns$
 2: $G(V, A) \leftarrow AssociateGraph()$
 3: while $n \neq 0$ do
 4:     Record intermediate expression as $\mathcal{E}$
 5:     Extract common variables in $\mathcal{E}$
 6:     Classify linear (non-linear) sub-expression in $\mathcal{E}$ into $\mathcal{E}_l(\mathcal{E}_{nl})$
 7:     if $\mathcal{E}_l$ can be abstracted as linear word then
 8:         Mark this iteration as word-cut
 9:         and generate word equation $\mathcal{W}l_{i1}$
10:     if $\mathcal{E}_{nl}$ can be abstracted as non-linear word then
11:         Mark this iteration as word-cut
12:         and generate word equation $\mathcal{W}nl_{i2}$
13:     $n \leftarrow n - 1$
14: return $\mathcal{W}l_{1,2...,i_1}, \mathcal{W}nl_{1,2...,i_2}$
```

The non-decomposable expression is also checked by $step$ 8 or $step$ 11, since the expression may also contain extractable word(s).

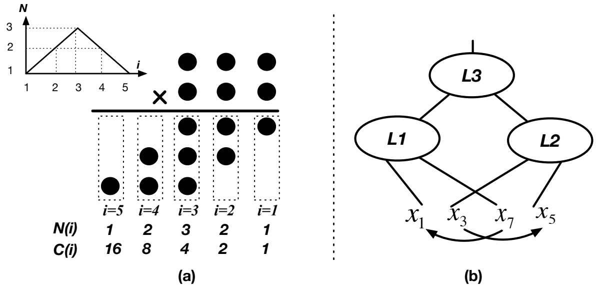

Fig. 2. (a) Coefficients distribution of 3-bit multiplier. (b) Variables pairing for expression $\{\cdots + 4x_1x_5 + 2x_1x_3 + 2x_5x_7 + x_3x_7 + \cdots\}$

• **Finding word candidates** ($line$ $8, 11$)**:** The expression is partitioned into a set of linear terms ($\mathcal{E}_l$) and a set of non-linear terms ($\mathcal{E}_{nl}$). First consider an expression, $\mathcal{E}_l$ containing p linear terms. Let $\mathcal{C} = \{c_1, c_2, ..., c_p\}$ be a set of coefficients associated with the terms e_i of the expression. It is easy to see that if the coefficients of some sub-expression $\mathcal{E}_{sub}$ of $\mathcal{E}$ form a series of increasing powers of two, i.e., if for some ordering of the elements of C, the coefficients satisfy the condition $c_k = 2c_{k-1}$, then the corresponding terms of $\mathcal{E}_{sub}$ can potentially form a word. That is, such a subexpression is a *word candidate*. In the next section we explain how to identify and extract the actual word from the expression in case when there are multiple terms with the same coefficient c_k.

The situation with non-linear expressions $\mathcal{E}_{nl}$ (polynomials containing nonlinear terms) requires more in-depth analysis of the distribution of the coefficient values. We explain it here with an example of an integer $m \times m$-bit multiplier, see Figure 2 (a). Let i be a bit position of the result, where $i = 1, ..., 2m-1$. Let $c_i = 2^{i-1}$ be the coefficient associated with column i of the result, shown in Figure 2 (a) as a black dot. Note, that depending on the bit position i there will be several coefficients with same value c_i. For the 3×3 multiplier we have: $C = \{1, 2, 2, 4, 4, 4, 8, 8, 16\}$, each corresponding to a black dot.

We now define the term N_i as the number of coefficients with value $c_i = 2^{i-1}$ for the bit position i. The value of N_i for $i = 1, ..., n$, where $n = 2m - 1$ is odd, can be computed by:

$$N_i = \begin{cases} i+1 & \text{if } \ i \ \leq \ (n-1)/2 \\ n-i & \text{if } \ i \ > \ (n-1)/2 \end{cases}$$

In the case of the 3-bit multiplier, with $n = 5$ result bits, we have: $N = \{1, 2, 3, 2, 1\}$, shown in Figure 2. Any expression that matches this forms is a candidate expressions; two candidate expressions might contain the same term. It can be shown that this analysis applies to any integer multiplier, regardless of its internal structure. Similar formulas can be derived for other nonlinear datapath operators, but (unlike for a linear ones) they all have similar "uneven distribution" behavior, where there is a larger number of the same coefficients in the middle bits than in the boundary bits.

• **Finalization** ($step$ $9, 12$)**:** At this point our algorithm will generate a valid signature for the word being abstracted. Continuing with a case of a general multiplier, consider a subexpression $\{\cdots + 4x_1x_5 + 2x_1x_3 + 2x_5x_7 + x_3x_7 + \cdots\}$ (which contains product of two words, $(2x_1 + x_7)(2x_5 + x_3)$). To determine if the sub-expression contains a word, we examine the term with the largest coefficient, $4x_1x_5$, and try to find the terms that together with $4x_1x_5$ would form a nonlinear expression, a product of some words. Specifically, for the term $4x_1x_5$ to form a word we need to find two terms with coefficients 2 (in this case $2x_1x_3$ and $2x_5x_7$), and one term with coefficient 1 (x_3x_7). To do that, have to match the remaining bits (here x_3, x_7) with the variables with largest coefficient we have just discovered (x_1, x_5). This is done by analyzing the topology of the network represented by the *AssociateGraph*, G. In this case we pair x_7 with x_1, and x_3 with x_5, and declare that $4x_1x_5 + 2x_1x_3 + 2x_5x_7 + x_3x_7$ contains two words: $(2x_1 + x_7)$ and $(2x_5 + x_3)$. This matching is shown in Figure 2 (b). The same process is applied to linear expressions as well.

C. Illustrative Example

We illustrate the concept of our abstraction using a simple ALU design (Figure 3). The inputs A, B, C are 4-bit wide and P is 8-bit wide, and f_C is a majority function. $M1, M2$ are 4-bit multipliers and ADD is an 8-bit adder. The output of two multipliers are selected by the f_C. Note that all components in the ALU are gate-level. The output signature is $F = \sum_{i=0}^{8} f_i 2^i$.

The first identified $word$ cut is $\mathcal{C}_1$. Using Algorithm 1, $\mathcal{C}_1$ is identified as a *multi-word* linear expression. The word of $word$ cut $\mathcal{C}_1$ is associated with $M1, M2$, f_C, and primary input P. The signals associated with P are all primary inputs and the signals associated with $\mathcal{W}_1$ include $M1, M2$ and f_C. Hence, the word P and $\mathcal{W}_1$ can be easily abstracted in *cut* $\mathcal{C}_1$. The next $word$ cut is identified as $\mathcal{C}_2$. This $word$ cut is different than $\mathcal{C}_1$ since there is a MUX operation. The function of cut $\mathcal{C}_2$ is $(1 - f_C)\mathcal{W}_2 + f_C\mathcal{W}_3$. Note that the

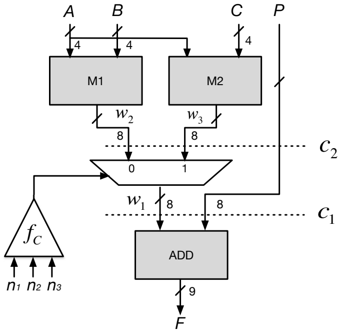

Fig. 3. ALU design with multiplier M_1, M_2, adder ADD, and $f_C = MAJ(n1, n2, n3)$

expression is $flattened$ to maximize the cancellations. Hence, the expression of c_2 in our process is $\mathcal{W}_2 - \mathcal{W}_2 f_C + \mathcal{W}_3 f_C$. The $step$ 5 in Algorithm 1 recognizes the decomposable part $-\mathcal{W}_2 f_C + \mathcal{W}_3 f_C$, and returns $-\mathcal{W}_2 + \mathcal{W}_3$. Then, the returned expression is identified in $step$ 11 as a non-linear *multi-word*. The word $\mathcal{W}_2$ and $\mathcal{W}_3$ come from two different multipliers. Therefore, the associated graph is able to classify these two words from the expression. As we mentioned in $step$ 5, the non-decomposable expression, in case $\mathcal{W}_2$, is a word expression. This is the reason why our algorithm checks both decomposable and non-decomposable sub-expressions. The process stops when the rewriting reaches the PIs.

IV. Experimental Results

The technique proposed in this paper was implemented in C++. The input to the program is a gate-level netlist with a known output signature (the binary encoding of the output bits) [2]. The experiments were conducted on a PC with Intel Processor Core i5-3470 CPU 3.20GHz x4 and 15.6 GB memory. To evaluate our word-abstraction technique, we tested the multiply-accumulate (MAC) design with bit-width ranging from 8 to 128, with an *enable* control. The results are shown in Table I. The *enable* function is the same as f_C in Figure 3.

The function of MAC can be written on a word level as: $\mathcal{F} = (A \times B) \cdot f_C + P$. It contains two word-level operations, addition and multiplication. The first column in Table I shows the bit-width of the design, and second column shows the number of gates. The column labeled *pre-ordering* includes the CPU time for ordering the netlist and parsing the netlist into expressions. Columns *addition* and *multiplication* show the CPU time of abstracting these two operations. Figure 4 shows the CPU time as a function of the number of gates. The complexity of the proposed algorithm is $O(n)$ for addition (linear $word\ cut$) and $O(n^2)$ for multiplication (non-linear $word\ cut$).

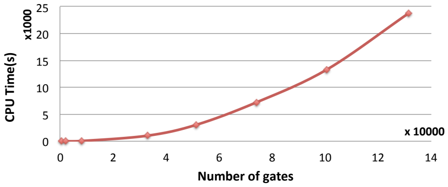

Fig. 4. MAC abstraction CPU time

size k	*#. gates*	*pre-ordering*	*Addition*	*Multiplication*	*Total*
8	529	0.01 s	0.01 s	0.22 s	0.24 s
16	2089	0.01 s	0.03 s	2.71 s	2.75 s
32	8281	0.03 s	0.11 s	50.7 s	51.00 s
64	32953	0.07 s	0.47 s	1028.9 s	1029 s
80	51432	0.12 s	0.76 s	3049.5 s	3050 s
96	74008	0.15 s	1.27 s	2.0 hrs	2.0 hrs
112	100681	0.21 s	1.62 s	3.7 hrs	3.7 hrs
128	131465	0.27 s	2.23 s	6.6 hrs	6.6 hrs

TABLE I. Word-abstraction evaluation using Multiply-accumulator s = seconds, hrs = hours

V. Conclusion and Future work

In this paper we presented a novel computer algebraic approach for abstracting the word-level information from gate-level netlist. In contrast to [3][14], we address the problem of abstracting words from a large non-linear gate-level arithmetic circuit (MAC). This approach is currently not applicable to designs with a non-arithmetic combinational logic attached to the output, as it would be difficult to reason about word association for such "noisy" signals. Additionally, we observe that identifying candidate expression is very time-consuming and it is not necessary that it be called in each iteration. In the future, we will focus on abstraction with a "noisy" combinational logic and on improving the extraction performance.

References

[1] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL (t)," *JACM*, vol. 53, no. 6, pp. 937–977, 2006.

[2] M. Ciesielski, C. Yu, W. Brown, D. Liu, and A. Rossi, "Verification of Gate-level Arithmetic Circuits by Function Extraction," in *DAC 2015*.

[3] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, S. Seshia *et al.*, "Wordrev: Finding Word-level Structures in a Sea of Bit-level Gates," in *HOST 2013*. IEEE, pp. 67–74.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided Abstraction Refinement," in *CAV 2000*. Springer, pp. 154–169.

[5] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *ACM SIGPLAN Notices*, vol. 37, no. 1, 2002, pp. 1–3.

[6] D. Kroening, A. Groce, and E. Clarke, "Counterexample Guided Abstraction Refinement via Program Execution," in *Formal Methods and Software Engineering*. Springer, 2004, pp. 224–238.

[7] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Refinement Strategies for Verification methods based on Datapath abstraction," in *ASP-DAC 2006*. IEEE Press, 2006, pp. 19–24.

[8] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," in *CAV 2002*.

[9] S. K. Lahiri, S. A. Seshia, and R. E. Bryant, "Modeling and Verification of Out-of-Order Microprocessors in UCLID," in *FMCAD 2002*.

[10] e. a. Mneimneh, Maher, "Scalable hybrid verification of complex microprocessors," in *38th DAC*. ACM, 2001, pp. 41–46.

[11] P. Manolios and S. K. Srinivasan, "Refinement Maps for Efficient Verification of Processor Models," in *DATE 2005*, pp. 1304–1309.

[12] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word level Predicate Abstraction and Refinement for Verifying RTL verilog," in *42nd DAC*. ACM, 2005, pp. 445–450.

[13] e. a. Jain, Himanshu, "Word-level Predicate-abstraction and Refinement Techniques for Verifying RTL verilog," *TCAD*, 2008.

[14] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse Engineering Digital Circuits Using Functional Analysis," in *DATE 2013*, pp. 1277–1280.