WolFEx: Word-Level Function Extraction and Simplification from Gate-Level Arithmetic Circuits

Kuo-Wei Ho[1], Shao-Ting Chung[1], Tian-Fu Chen[2], Yu-Wei Fan[1], Che Cheng[1], Cheng-Han Liu[2], Jie-Hong R. Jiang[1,2,3]
[1]*Graduate Institute of Electronics Engineering, National Taiwan University*, Taipei, Taiwan
[2]*Graduate School of Advanced Technology, National Taiwan University*, Taipei, Taiwan
[3]*Department of Electrical Engineering, National Taiwan University*, Taipei, Taiwan

Abstract—**Extracting word-level functions from gate-level circuits is challenging and crucial in security, synthesis, and verification applications. State-of-the-art approaches identify subcircuits to match against a predefined library of components. However, they fail for highly-optimized arithmetic circuits due to the absence of intermediate word structures and the high complexity of verifying arithmetic functions. The challenge of learning arithmetic operations from gate-level netlists is posed in the 2022 ICCAD CAD Contest. This work tackles the challenge by devising and combining algebraic, statistical, and structural techniques into an operational flow for function extraction and simplification. Beyond the contest setting, our method also deals with circuits without their input- and output-pin information. Experiments on the contest benchmarks show that our method outperforms the winning teams in the contest in both the number of solved cases and the compactness of the extracted word-level expressions. Moreover, our method can effectively extract most word-level functions within 10 minutes.**

I. Introduction

Extracting high-level information from low-level implementation can be beneficial and sometimes indispensable in the design flow of integrated circuits. E.g., the Layout Versus Schematic (LVS) problem [1] is essential in physical design verification. Also, extracting word-level functions from gate-level netlists can be important in applications such as hardware Trojan detection [2], intellectual property (IP) infringement verification [3], multiplier circuit verification [4], [5], engineering change order (ECO) [6], design debugging [7], circuit optimization [7], among others. While generating gate-level netlists from physical layout is well-studied [8], retrieving word-level functions from gate-level netlists remains challenging.

The importance and immaturity of word-level function extraction trigger the challenge posed by the 2022 ICCAD CAD Contest, Problem A: *Learning arithmetic operations from gate-level circuit* [6]. The task is to express the functionality of highly-optimized gate-level circuits using word-level arithmetic operators. Although there are relevant techniques in the literature, no existing algorithm can solve the problem directly. This work aims to tackle this challenge.

The related efforts can be roughly classified into two categories. In one category, prior efforts, e.g., [9], [10], focus on circuit partitioning, which discovers word structures, datapaths, or inner function blocks. In the other category, prior methods, e.g., [11], [12], focus on matching a partitioned subcircuit against a library of predefined components to identify its high-level function. Recent work [13] combines methods in the above two categories. It identifies intermediate word structures and partitions the circuit by enumerating cuts from the primary output and matching adders or multipliers during the process. These matching-based approaches are limited by the given library and cannot handle complex function blocks. Moreover, aggressive optimization by synthesis tools often destroys datapath and word structures, making the partition task more complicated or even impossible. In such a case, the partition-then-match flow is hardly applicable.

To overcome the limitations of prior work, we engineer a computation flow to extract word-level functions from gate-level circuits. It combines and extends techniques from polynomial rewriting [14], symbolic regression [15], structural analysis, and SAT solving. Polynomial rewriting is a computer algebra technique primarily adopted in arithmetic circuit verification, e.g., [5], [14]. It is also applied in [13] to extract datapaths. On the other hand, symbolic regression is a mathematical regression technique for data analysis [15]. To the best of our knowledge, it has not been exploited in word-level function extraction.

This work was supported by the National Science and Technology Council of Taiwan under grants 111-2221-E-002-182 and 111-2923-E-002-013-MY3.

The main results of this work include the following. First, the proposed polynomial-rewriting-based function extraction technique is enhanced to extract complex word-level functions, such as words with constant exponent, exponential functions, and various comparisons. Unlike [13], our method does not require the existence of intermediate word structures in the underlying netlist. We further enhance the method to function extraction without using the input- and output-pin information. Second, for the first time, we adopt symbolic regression in word-level function extraction by interpreting bit-vectors as binary-coded integers. While symbolic regression searches for an expression that fits real-valued sampled patterns, the proposed algorithm combines it with SAT-based technique to iteratively search and refine the expression with counterexamples. Third, a *linear coefficient fitting* algorithm is devised to complement the polynomial-rewriting and symbolic-regression based methods when the number of input words is large and some control logic is involved. Although it only explores expressions in a restricted form, it supports more operations (especially comparators) than the proposed polynomial-rewriting-based method and its efficiency is not as sensitive to the number of words as in symbolic regression. Fourth, we propose a function extraction flow that integrates the above procedures to recover and simplify high-level functionality from highly-optimized combinational arithmetic circuits without assumptions on the structural information. Experimental results on the CAD Contest benchmarks demonstrate that our approach outperforms the contest-winning teams in both the number of solved cases and solution quality. Moreover, our approach can efficiently extract high-level functions within 10 minutes in most cases.

The rest of this paper is organized as follows. We start with preliminaries in Section II and the problem statement in Section III. In Section IV, we introduce the proposed algorithms and supplement them with key implementation details in Section V. Finally, Section VI shows the experimental results, and Section VII concludes the paper.

II. Preliminaries

Each signal in a circuit is associated with a Boolean variable $x \in \{0, 1\}$. A bit-vector of $\ell \geq 1$ variables $W = (w_{\ell-1}, \ldots, w_1, w_0)$ associated with some arithmetic data is referred to as a *word*, where w_i is the i^{th} bit of W. In this work, the capital letter W denotes an arbitrary word, and capital letters X and Y denote a word at the primary inputs (PIs) and primary outputs (POs) of a circuit, respectively.

We refer to an expression composed of words and RTL operations as a *word-level expression* (*expression* for short), denoted by the capital letter E. An *RTL description* consists of the declaration of PIs, POs, and internal words, followed by the assignment of each internal and PO word with an expression.

A. Algebraic Model

A *pseudo-Boolean monomial* (*monomial* for short) $M = \prod_{i=1}^{n} x_i^{k_i}$ is a product of variables x_i's, denoted by the capital letter M. A *term* cM is a monomial M multiplied by a constant $c \in \mathbb{Z}$. A *pseudo-Boolean polynomial* (*polynomial* for short) $P = \sum_{i=1}^{n} c_i M_i$ is a sum of the terms $c_i M_i$, denoted by the capital letter P. Since $x \cdot x = x$ for Boolean variable x, the monomial x^k reduces to x, and all monomials with the same set of variables can be merged into a single term. That is, any polynomial P can be represented in the form

$$P = c_0 M_0 + c_1 M_1 + c_2 M_2 + \cdots + c_k M_k \,, \tag{1}$$

where each M_i is a monomial with variables of order at most 1, and each $c_i \in \mathbb{Z}$ is a constant coefficient.

In this work, Boolean connectives are associated with algebraic operations. In particular, the algebraic models of the considered basic Boolean gates are as follows.

$$\begin{array}{rcl} \neg a & = & 1 - a \\ a \wedge b & = & ab \\ a \vee b & = & a + b - ab \\ a \oplus b & = & a + b - 2ab \end{array} \tag{2}$$

A word W with ℓ bits represents an integer, and its functionality can be expressed by a polynomial P_W. Specifically, the *unsigned* integer value of W can be expressed using the polynomial

$$\sum_{i=0}^{\ell-1} 2^i w_i, \tag{3}$$

and the *signed* integer value, in the form of 2's complement, can be expressed using the polynomial

$$-2^{\ell-1} w_{\ell-1} + \sum_{i=0}^{\ell-2} 2^i w_i \,. \tag{4}$$

B. Polynomial Rewriting

Polynomial rewriting is an algebraic technique that represents each PO word as a unique polynomial over the PI variables from the gate-level implementation. The technique can be applied for functional analysis [13] and formal verification of arithmetic circuits [14].

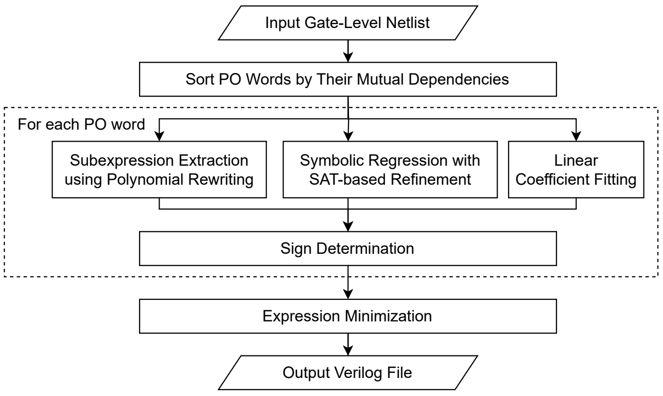
The rewriting process is as follows. For each PO word Y, first let the polynomial P be the word polynomial of Y shown in Eq. (3). Then, for each gate g in the transitive fanin cone of Y, following the reverse topological order from the PO to PIs, we substitute the variable of the fanout signal of g in P by the variables of its fanin signals using the arithmetic model defined in Eq. (2). E.g., if g is an OR gate specifying $z = a \vee b$, then each occurrence of z in P is substituted by $(a + b - ab)$ in the polynomial. When the process is done, P contains only PI variables, and represents the function of the word Y.

C. Symbolic Regression

Symbolic regression [15] is a regression analysis technique for deriving a symbolic expression that best fits a given dataset in terms of accuracy and simplicity. It explores different combinations of mathematical operators, constants, and variables. More formally, let the dataset $D = \{(\vec{d_1}, t_1), \ldots, (\vec{d_n}, t_n)\}$, for $\vec{d_i} \in \mathbb{R}^k$ and $t_i \in \mathbb{R}$, be a sample set of assignments to the input variables $\vec{x} = (x_1, \ldots, x_k) \in \mathbb{R}^k$ and the corresponding values of the output variable $y \in \mathbb{R}$. The symbolic regression problem on D aims to find an expression $E(\vec{x})$ in terms of some pre-specified operators, variables in $\vec{x}$, and real constants, such that a given loss function

$$L(t_1, \ldots, t_n, E(\vec{d_1}), \ldots, E(\vec{d_n}))$$

and the expression cost of $E(\vec{x})$ w.r.t. a specified cost metric are minimized.



Input Gate-Level Netlist
Sort PO Words by Their Mutual Dependencies
For each PO word
Subexpression Extraction using Polynomial Rewriting
Symbolic Regression with SAT-based Refinement
Linear Coefficient Fitting
Sign Determination
Expression Minimization
Output Verilog File

Fig. 1. The proposed function extraction flow

The state-of-the-art methods for symbolic regression include genetic-programming-based methods [16] and evolutionary polynomial regression [17]. In this work, we adopt and extend the symbolic regression package `PySR` [18], which is based on genetic programming, in our word-level function extraction flow.

D. RTL Expression Minimization with Equivalence Graph

RTL expression minimization using *equivalence graphs* (e-graphs) [19] has been studied in [20]. While the reader is referred to [20] for detailed exposition, we give a brief background. An e-graph is a 3-tuple (V, C, H), where V is a set of *e-nodes*, $C \subseteq 2^V$ is a set of *e-classes* that partitions V, and $H \subseteq V \times C$ is a set of directed edges. Each e-node is associated with an operator and characterizes an expression. Each e-class consists of a set of e-nodes whose characterized expressions are functionally equivalent. An edge (v, c) for $v \in V$ and $c \in C$ signifying the operator of v takes the expression characterized by an e-node in c as an operand. Given an e-graph and a set of *rewriting rules* [20], in the form of $R_1 \to R_2$ to convert expression pattern R_1 to R_2, the e-graph is iteratively transformed to create new e-nodes and merge equivalent e-classes until *saturation*, namely, no more changes can be made. Then, a cost-minimized expression can be extracted from the saturated e-graph. The minimization essentially requires exploration of common subexpression sharing and is formulated with integer linear programming (ILP) in [20].

III. Problem Formulation

We define the word-level function extraction problem as follows.

Problem Statement (Word-Level Function Extraction)**.** Let U be a set of word-level operations, each associated with a nonnegative cost. Let C be an unknown word-level expression realizing some controlled arithmetic function(s) composed of the operations in U. Let N_C be a gate-level netlist synthesized from C, where the word-level operations are realized by primitive gates, including AND2, OR2, NAND2, NOR2, XOR2, OR2, BUF, and NOT. The inputs and outputs of C and N_C are either single-bit inputs or multi-bit words with bit-order specified. Then, given U and N_C, the objective is to find a word-level expression C' composed of operations in U with a minimum cost that is functionally equivalent to N_C.

In this work, we tackle Problem A of the 2022 ICCAD CAD Contest [6] (referred to as the CAD Contest in the sequel), where U contains arithmetic operations, e.g., addition $(+)$, subtraction $(-)$, multiplication $(\times)$, etc., as well as logical operators, e.g., Condition $(?:)$, Equality $(==)$, etc. A comprehensive list of operations and their associated costs can be found in [6].

IV. Methods

Fig. 1 shows the proposed word-level function extraction flow, WolFEx. Given the input gate-level netlist, we first sort the PO words by their functional dependencies before solving each one. If any bit of

TABLE I
THE PO WORDS OF CASE *text17* IN THE CONTEST BENCHMARKS.

PO	Functionality	Bitwidth	Sign
out1	*in2* $-$ *in1* $- 2$	33	Signed
out4	*in3* $-$ *in1* $- 2$	33	Signed
out2	*out1* > 7	1	Unsigned
out3	*out1* < -4	1	Unsigned
out5	*out4* > 7	1	Unsigned
out6	*out4* < -4	1	Unsigned

a PO word Y_1 is in the transitive fanin cone of another PO word Y_2, then Y_1 is solved before Y_2. Next, we extract word-level expressions from each PO word Y using three different techniques: *polynomial rewriting*, *symbolic regression*, and *linear coefficient fitting*. Finally, we perform *combinational equivalence checking* (CEC) to verify the correctness of these expressions, and decide whether each word is signed or unsigned using the obtained expressions.

Each technique aims for different types of functions. Applying different techniques to a PO may result in different expressions. To obtain better solution quality, in our implementation we sequentially apply all the techniques to each PO with a specific timeout for each technique, although they can also be applied in parallel. Our flow collects all equivalent expressions obtained by these different techniques and exploits them in the final expression optimization stage. Alternatively, for better efficiency, we can skip to the next PO word once a word-level expression is obtained. These two options will be evaluated in Section VI as the *default* and *fast* settings.

After the expressions of all PO words are obtained, we minimize the total cost by exploring equivalent expressions and sharing common subexpressions using e-graph. If, for some output words, all of the techniques failed to find a word-level expression with a small enough cost, then our extraction procedure fails for these output words, and their original gate-level implementations are adopted in the output Verilog file.

A. An Illustrative Example of the Computation Flow

Before going into the details, we illustrate the proposed flow on the contest benchmark *test17*, which contains three PI words and six PO words. PI *in1* is a 31-bit unsigned integer. PI *in2* and *in3* are 32-bit unsigned integers. Table I shows the sorted PO words. For simplicity, we first apply the polynomial rewriting technique, and then the symbolic regression technique. Once an expression is obtained for one PO, we move on to the next PO.

First, by assuming *out1* to be an integer and performing polynomial rewriting, we can obtain a polynomial

$$-(a_0 + 2a_1 + 4a_2 + ...) + (b_0 + 2b_1 + 4b_2 + ...) - 2,$$

where a_i and b_i are the i^{th} bit of PI words *in1* and *in2*, respectively. Then, the word-level expression $(-1) \times$ *in1* $+$ *in2* $- 2$ can be obtained by substituting the polynomials representing the integer values of *in1* and *in2* with the word *in1* and *in2*, respectivley. Another multi-bit PO word *out4* can be solved similarly.

The polynomial rewriting technique cannot solve the following four single-bit PO words. To apply the symbolic regression technique to *out2*, some patterns are randomly sampled from the design. Suppose the sampled patterns of (*out1*, *out2*) are $\{(4,0),(6,0),(8,1)\}$, the symbolic regression on these patterns may result in an incorrect expression *out1* > 6. Performing CEC finds and adds a counterexample $(7,0)$ to the sampled patterns. Thus, the correct expression *out1* > 7 will be found in the next iteration. The rest of the PO words are then solved similarly.

Finally, the total cost of the resulting six expressions is 10. Note that the negative sign of negative numbers has no cost, since the numbers can be written in a binary form without the negative sign. The optimization step then finds exactly the same expressions shown in Table I. Furthermore, the common subexpression (*in1* $- 2$) is shared, and the resulting total cost is 7.

Algorithm 1 Extract word-level subexpressions from polynomial P

```
 1: procedure POLY-EXTRACT($P$)
 2:     if $P$ is a constant then
 3:         return $P$
 4:     for all PI word $X$ do
 5:         if $P/x_0 \neq 0$ then
 6:             $(P_Q, P_R, E_D) \leftarrow$ EXTRACT-WORD($P, X$)
 7:             if $P_Q \neq 0$ then
 8:                 $E_Q \leftarrow$ POLY-EXTRACT($P_Q$)
 9:                 $E_R \leftarrow$ POLY-EXTRACT($P_R$)
10:                 return $E_Q \times E_D + E_R$
11:     return $P$
```

B. Subexpression Extraction using Polynomial Rewriting

For each PO word Y with ℓ bits, we first apply polynomial rewriting to derive the polynomial P_Y that expresses the functionality of Y in terms of PIs, and then use our word-level subexpression extraction procedure to derive a word-level expression of P_Y.

The main idea of the procedure is as follows. Given the polynomial P_Y, we want to find a *single-word subexpression* E_D, i.e., an expression composed of only one PI word, with polynomial representation P_D such that $P = P_Q \cdot P_D + P_R$ for some polynomials P_Q and P_R. After recursively performing the extraction on P_Q and P_R, the word-level expression $E_Q \times E_D + E_R$ that represents the word-level functionality of Y can be obtained.[1]

Algorithm 1 shows the procedure of subexpression extraction. Given a polynomial P, we first choose a PI word X. If P contains the variables x_0 of the LSB of X (Line 5), we try to find a factored form $P_Q \cdot P_D + P_R$ using the subprocedure EXTRACT-WORD(P, X). Algorithm 1 is then applied recursively on P_Q and P_R (Lines 8 and 9) if EXTRACT-WORD succeeds. On the other hand, if the procedure EXTRACT-WORD fails and returns $(0, P, \emptyset)$, we attempt with the next input word (Line 4). The recursion stops when the target polynomial becomes a constant (Line 3) or when the extraction fails for all input words (Line 11).

EXTRACT-WORD attempts to find three different types of single-word subexpressions: X^k, k^X, and $(X{==}k)$, where k is some integer constant that can be computed according to P, and $(X{==}k)$ denotes the equality operator between X and k, which is evaluated to 1 if x is equal to k, and 0 otherwise.

If an expression contains the multiplication or addition of a word X, the expression can be expressed as $E_Q \times X + E_R$. Thus, using X as a divisor, EXTRACT-WORD is able to identify two polynomials E_Q and E_R. By trying different input words recursively on E_Q and E_R, the addition and multiplication of words can be fully extracted.

However, using X as a divisor, the extraction procedure can fail when X^k, for $k > 1$, exists in the expression. It is because binary variables with different exponents cannot be distinguished. For example, let $X = (x_1, x_0)$ be a two-bit word. The polynomial of X^2 is

$$P = (2x_1 + x_0)^2 = 4x_1x_0 + 4x_1 + x_0.$$

When EXTRACT-WORD try to find a form $P_Q \times P_D + P_R$ with $P_D = (2x_1 + x_0)$, the polynomial of X, it yields

$$P = (4x_1 + 1) \cdot (2x_1 + x_0) + (-6)x_1,$$

where both $P_Q = 4x_1 + 1$ and $P_R = (-6)x_1$ cannot be further extracted at the word level. To overcome this issue, EXTRACT-WORD extracts X^k with the largest possible k directly, instead of extracting X for k times.

Moreover, we consider another two often encountered subexpressions k^X and $(X{==}k)$, which have simple polynomial representations that can be computed and recognized easily. Multiplying an

[1]Note that even if the procedure fails to extract subexpressions from some polynomials during the recursion, we can still generate an expression by replacing the variables in those polynomials with their actual PI signals.

expression by k^X for $k = 2$ corresponds to the left-shift (by X) operation. On the other hand, $(X{=}{=}k)$ mostly appears in control logic as the selection signal of multiplexers. The polynomials of the three considered subexpressions are shown in Eq. (5), where $X = (x_{\ell-1}, \ldots, x_0)$ and k_i denotes the i^{th} bit of the integer k in its binary representation.

$$\begin{aligned} X^k &= \left(\sum_{i=0}^{\ell-1} 2^i x_i\right)^k \\ k^X &= \prod_{i=0}^{\ell-1} \left(1 + (k^{2^i} - 1)x_i\right) \\ (X{=}{=}k) &= \prod_{i=0}^{\ell-1} (k_i x_i + (1 - k_i)(1 - x_i)) \end{aligned} \tag{5}$$

For X^k, in the case that X represents a signed integer, the polynomial yielded from rewriting is matched against X^k with signed X as shown in Eq. (4). For k^X, since a negative exponent introduces a fraction, we assume it to be unsigned. Finally, for $X{=}{=}k$, the polynomial is the same as when X is unsigned.

The main weakness of algorithm EXTRACT-WORD is its limited types of subexpressions. It can only extract additions and multiplications of the aforementioned three types of subexpressions. Furthermore, when a design contains operators like right-shift, bit-select, or relational predicate other than equality, polynomial rewriting may yield a polynomial that grows exponentially with respect to the bit-width of operands. As a result, meaningful polynomials cannot be derived or recognized in a reasonable time. This is the main reason that no single-bit PO words can be solved by the proposed algorithm.

On the other hand, as the polynomial representation exactly describes the functionality of output words, the extracted word-level expression is guaranteed to be correct. Unlike the other two methods, namely, symbolic regression and linear coefficient fitting, that extract functionality from simulation patterns, the method of polynomial rewriting does not require further verification and refinement. However, its performance is highly affected by the complexity and bit-width of the input design. To overcome this issue, we propose a heuristic in Section V-A by applying polynomial rewriting on only part of the design.

C. Polynomial Rewriting without Input/Output-Pin Information

Beyond the problem formulated in Section III, we extend the methods described in Section IV-B to extract function even in the absence of input- and output-pin information. The challenge lies in distinguishing the signals in different PI and PO words and determining their ordering, in addition to extracting the word-level function.

This extended flow as shown in Fig. 2 builds upon the method presented in Section IV-B, thus excluding single-bit POs from consideration. It is important to note that the other two techniques, *symbolic regression* and *linear coefficient fitting*, cannot be applied without input- and output-pin information. This is because they rely on the interpretation of PI and PO words as integers before deriving the function.

The main idea is to first identify the lowest few bits of PO words using the circuit structural information. Then, for each PO word, the polynomial that represents the functionality of the lowest bits of the PO word is obtained using the partial rewriting technique to be described in Section V-A. Afterward, a modified version (see Section IV-C2) of Algorithm 1 is performed to extract word-level subexpressions and identify the lowest bits of PI words at the same time. Finally, by assuming that the whole design has the same functionality except for having a larger bit-width, the rest of the PI and PO bits are matched to their corresponding words. The expression minimization afterward is the same as in the original flow.

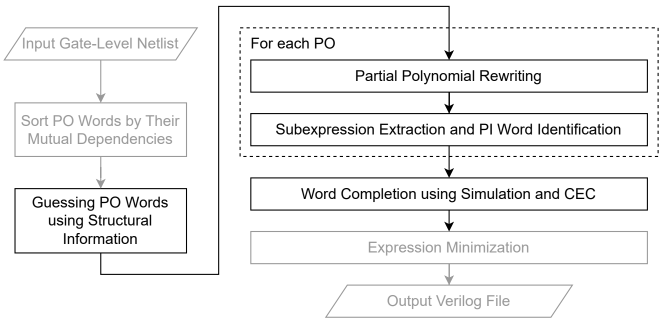

Input Gate-Level Netlist
Sort PO Words by Their Mutual Dependencies
Guessing PO Words using Structural Information
For each PO
Partial Polynomial Rewriting
Subexpression Extraction and PI Word Identification
Word Completion using Simulation and CEC
Expression Minimization
Output Verilog File

Fig. 2. The proposed function extraction flow without input- and output-pin information. The gray steps are the same as those in the original flow.

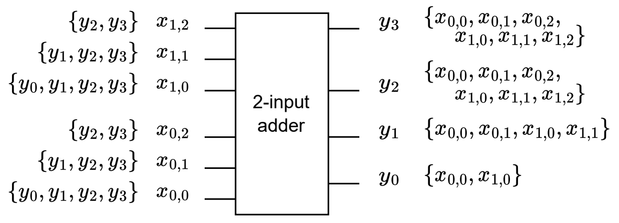

$\{y_2, y_3\}$ $x_{1,2}$
$\{y_1, y_2, y_3\}$ $x_{1,1}$
$\{y_0, y_1, y_2, y_3\}$ $x_{1,0}$
$\{y_2, y_3\}$ $x_{0,2}$
$\{y_1, y_2, y_3\}$ $x_{0,1}$
$\{y_0, y_1, y_2, y_3\}$ $x_{0,0}$
2-input adder
y_3 $\{x_{0,0}, x_{0,1}, x_{0,2}, x_{1,0}, x_{1,1}, x_{1,2}\}$
y_2 $\{x_{0,0}, x_{0,1}, x_{0,2}, x_{1,0}, x_{1,1}, x_{1,2}\}$
y_1 $\{x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}\}$
y_0 $\{x_{0,0}, x_{1,0}\}$

Fig. 3. The PI (resp. PO) signals in the transitive fanin (resp. fanout) cone of each PO (resp. PI) signal in a 2-input 3-bit adder, where $x_{i,j}$ is the j^{th} bit of the PI word X_i and y_i is the i^{th} bit of the PO word Y.

1) Guessing PO Words using Structural Information: The partial rewriting technique discussed in Section V-A can be performed on the lowest few bits of a PO word. Therefore, the main objective of this step is to identify and determine the ordering of the lowest few bits of each PO. To do so, we analyze the structural information of PO signals.

Let $S_{\text{PI}}(y)$ denote the set of PI signals in the transitive fanin cone of signal y. considering an adder of two ℓ-bit words X_0, X_1 with $(\ell+1)$-bit output word Y. Observe that the lowest $\ell - 1$ bits satisfy the property that $S_{\text{PI}}(y_i) \subsetneq S_{\text{PI}}(y_j)$ for $i < j$, where y_i refers to the i^{th} bit of Y (as can be seen from Fig. 3). This property also holds for the lowest $\ell - 1$ output bits in a multiplier. In fact, often in arithmetic designs consisting of word-level additions, multiplications, and possibly some control logic, the lowest bits of PO words exhibit this support relation property. Although it is not always the case, at least in the contest benchmarks, all of the PO words that can be solved using the method in Section IV-B have enough lowest bits that exhibit this property.

However, using the above property is not enough to distinguish signals between PO words. Thus, we further use the support relation and the topological distance between PO signals to construct a PO word Y. E.g., we assume that the transitive fanin cone of a signal in Y does not contain any other signals in Y. Also, we consider the closest signals that satisfies the above properties to be the neighboring bits in Y.

2) Subexpression Extraction and PI Words Identification: We modify Algorithm 1 to extract subexpressions and identify PI words at the same time. In the main loop (Line 4), instead of iterating through all PI words, a signal is chosen to serve as the LSB x_0 of some unknown PI word X. By Eq. (3), observe that the variable of the LSB has the smallest coefficient in the polynomial representation of its word. Thus, in a design consisting of additions, multiplications, and control logic, the monomial with the smallest coefficient must contain the LSB of some PI words. Then, in this monomial, the variable that appears in the least number of monomials is selected first as it is less likely to be a control signal.[2]

[2]According to our observation, a control signal involved in a word-level operation often appears in all the monomials derived from each individual bit in the related words.

Algorithm 2 Search a word-level expression for PO word Y on netlist N using symbolic regression

```
1:  procedure SR-REFINE($Y$)
2:      $N \leftarrow$ the netlist of the transitive fanin cone of $Y$
3:      $S \leftarrow$ SAMPLE($N$)
4:      loop
5:          $(E, loss) \leftarrow$ SYMBOLIC-REGRESSION($S$)
6:          if $loss \neq 0$ then                    ▷ Symbolic regression failed
7:              return $\emptyset$
8:          $I \leftarrow$ implementation of $E$
9:          if CONSTRAINED-CEC($N, I$) = TRUE then
10:             return $E$                           ▷ Correct expression found
11:         else
12:             add counterexamples to $S$
```

In this modified algorithm, function EXTRACT-WORD only looks for the single-word subexpression X^1. Observed that in the polynomial of X^1 (in the form of Eq. (3)), the coefficient of the i^{th} bit x_i is 2^i. Thus, the i^{th} bit x_i can be identified by checking if $P/x_i = 2^i \times P/x_0$.

Furthermore, let $S_{\text{PO}}(x)$ denote the set of PO signals in the transitive fanout cone of a signal x. As the PO words satisfy the support relation properties mentioned in Section IV-C1, the lowest bits of PI words also satisfy $S_{\text{PO}}(x_j) \subsetneq S_{\text{PO}}(x_i)$ for $i < j$, as can be seen from Fig. 3. Thus, the candidate signals of x_i can be chosen by this heuristic.

3) Word Completion using Simulation and CEC: After the lower part of all PI and PO words and the word-level expressions of all PO words are obtained, a design is synthesized with the same functionality but larger bit-widths. The identified parts of the PO words and the LSBs of identified PI words are matched to the corresponding input and output signals in the synthesized design.

The correctness of the matched signals can be verified by setting the unmatched PI signals to zero and performing CEC on the matched PO signals. Furthermore, when the lowest ℓ bits are already matched in a word, we can find a signal as the $\ell+1^{\text{th}}$ bit by either simulation or SAT solving (followed by CEC). For example, consider the 2-input adder in Fig. 3, given that $x_{0,0}$, $x_{1,0}$, and y_0 are matched, we can identify y_1 by the pattern $(X_0, X_1, Y) = (1_2, 1_2, 10_2)$ since all the bits except y_1 are zero in this case. Similarly, we can then identify $x_{1,1}$ by the pattern $(X_0, X_1, Y) = (1_2, 10_2, 11_2)$. Finally, all the unmatched signals in the design can be matched one by one and the correct bit order can be obtained.

D. Symbolic Regression and SAT-based Refinement

To extract a word-level expression equivalent to the given implementation, we randomly simulate the circuit to collect a set of patterns S, and use symbolic regression to find expressions that exactly fit these patterns. However, since the expressions should fit all possible input patterns in addition to the patterns in S, we further refine the expression using patterns that do not fit the expression. These patterns are referred to as *counterexamples* and are found by performing CEC on the original circuit and the one synthesized from the obtained expression. The process is repeated until no counterexample can be found.

Algorithm 2 shows the expression finding procedure using symbolic regression. We perform random simulation and collect a set of patterns S (Line 3). Then, we repeatedly perform symbolic regression on the collected samples (Line 5), verify the correctness of the found expression (Line 9), and update S accordingly (Line 12), until either a correct one is found (Line 10) or the symbolic regression fails (Line 7).

The procedure CONSTRAINED-CEC(N, I) performs equivalence checking under the constraints that the sign bits are fixed to 0 for all PI words with unknown signs. It returns TRUE if N is equivalent to I. The procedure SYMBOLIC-REGRESSION returns an expression E and a value $loss$ indicating the difference between the target output values and the evaluation of E on the sampled patterns. An expression E is feasible only when $loss = 0$. If multiple feasible expressions are found, it returns the one with the lowest complexity in terms of the number of symbols and operators used.

Additionally, efforts should be made to determine the sign of each word. For PO words, we perform symbolic regression using both signed and unsigned interpretations. For PI words, we only use the patterns and counterexamples in which the sign-bits are 0 and postpone the determination of their signs to the procedure detailed in Section V-C.

The variety of supported operations makes the algorithm powerful in handling designs with relatively few input words. In addition, computing only with sampled patterns makes the symbolic regression algorithm work insensitive to the bit-width and structure of a design. However, it scales poorly as the number of variables increases due to the large search space, even if the function has a simple structure. Also, it can hardly handle control logic in the circuit. To alleviate these issues, we propose linear coefficient fitting in Section IV-E and function extraction with an enumeration of control input assignments in Section V-D.

E. Linear Coefficient Fitting

To improve the scalability of word-level function extraction, we develop the linear-coefficient-fitting method to target simple functions with large numbers of variables and possibly some control logic. In linear coefficient fitting, four types of functions are considered, which are linear, multiplication, exponential, and polynomial functions. These four types of functions are selected because they can be written as linear combinations of terms with unknown constant coefficients. Thus we can use a system of linear equations to solve the unknown coefficients. Moreover, as these four types of functions can express some complex functions (e.g., logarithmic, trigonometric) by, e.g., Taylor expansion, they are basic and with good generality.

Formally, if Y is the PO word and $\vec{X} = (X_1, \ldots, X_n)$ are the n PI words, then the four types of functions are of the following forms:

$$\begin{aligned}
&\text{linear:} && Y = a_0 + \sum_{i=1}^{n} a_i X_i \\
&\text{multiplication:} && Y = a_0 \cdot \prod_{i=1}^{n} {X_i}^{a_i} \\
&\text{exponential:} && Y = a_0 \cdot \prod_{i=1}^{n} {a_i}^{X_i} \\
&\text{polynomial:} && Y = \sum_{0 \le t_i \le k} \left(a_{(t_1,\ldots,t_n)} \prod_{i=1}^{n} {X_i}^{t_i}\right),
\end{aligned}$$

where a_i's are integer constants to be fitted, and k is a predefined integer indicating the maximum power in the polynomial. We note that although polynomial functions subsume linear and multiplication functions, guessing polynomial functions incurs higher complexity. Therefore, we tend to try linear and multiplication functions first due to their relatively little effort but handling well common linear and multiplication relations.

For linear functions, there are $n+1$ unknown a_i's. We sample at least $n+1$ value assignements to $\vec{X}$ and their corresponding value of Y and generate a system of linear equations. Solving the system of equations gives the coefficients $a_0, \ldots, a_n$.

For multiplication and exponential functions, although a_i's are not linear coefficients, we can first convert them to linear functions by taking logarithms, and transform them back after the linear system is solved. Although precision loss may occur during this process, we can simply round them back to integers during the transformation phase as long as the precision is high enough.

For polynomial functions, it is easy to see that it has the same structure as linear functions, except that there are $(k+1)^n$

possible terms, meaning $(k+1)^n$ samplings to $(\vec{X}, Y)$ are needed. Fortunately, often we find that most a_i's are zero.

Our method begins with taking one of the input words, say X_1, as our *principal input word*. We then rewrite the equation as

$$Y = \sum_{0 \le t_1 \le k} a_{t_1}(X_2, X_3, \ldots, X_n) {X_1}^{t_1} ,$$

where a_{t_1} is a function of $X_2, X_3, \ldots, X_n$. Then, we randomly assign values to $(X_2, \ldots, X_n)$ and use the previous method to find a_i's under this condition. We repeat this procedure several times. If a certain a_{t_1} is the same under all different assignments of $(X_2, \ldots, X_n)$, it is supposed to be independent of $X_2, \ldots, X_n$. We collect these a_{t_1}'s into a set I_1. Next, we include one more variable, X_2, and rewrite the equation as

$$Y = \sum_{\substack{0 \le t_1, t_2 \le k, \\ t_1 \notin I_1}} a_{t_1,t_2}(X_3, \ldots, X_n) {X_1}^{t_1} {X_2}^{t_2} + \sum_{t_1 \in I_1} a_{t_1} {X_1}^{t_1} .$$

Note that a_{t_1}'s in the second summation term are decided, so there are only $(k+1)(k+1-|I_1|)$ coefficients to compute, where $|I_1|$ is the size of I_1. Similarly, we randomly choose a combination of $(X_3, \ldots, X_n)$ and find a_i's under different assignments to decide whether each a_{t_1,t_2} is going to be fixed. We repeat this procedure until all input words are included as principal input words.

The above method covers the case that the PO Y is an integer. We extend the method to the situation where Y is a single bit representing the result of some comparison. For comparator ">," we select an input word as the principle input word, say X_1, and assume that the inequality can be rewritten as $Y = (X_1 > f(X_2, \ldots, X_n))$, where f is an unknown function. We then randomly choose a combination of $\vec{X}(X_1) = [X_2, \ldots, X_n]$, and use binary search to find the minimal value, denoted $\mathit{boundary}(X_1)$, making $Y = 1$. As $\mathit{boundary}(X_1)$ is a function of $X_2, \ldots, X_n$, the aforementioned approach can be applied to find $f(X_2, \ldots, X_n)$. For comparator "<," the extraction can be done similarly. Also, as X_1 is an integer, this approach naturally extends to ">=" and "<=."

If the value of X_1 making $Y = 1$ (resp. $Y = 0$) is difficult to find, then the inequality may be of the type "==" (resp. "!="). In other words, the inequality may be either $Y = (X_1 {==} f(X_2, \ldots, X_n))$ or $Y = (X_1 != f(X_2, \ldots, X_n))$. In this case, we use an SAT solver to find the value of X_1 that makes $Y = 1$ (resp. $Y = 0$), denoted $\mathit{key}(X_1)$. As $\mathit{key}(X_1)$ is also a function of $X_2, \ldots, X_n$, we can find $f(X_2, \ldots, X_n)$ using the same approach.

This algorithm addresses some issues in the previous two algorithms. When compared to the polynomial-rewriting-based method, it can recognize expressions involving more operations. On the other hand, instead of solving the whole function, it searches for an expression with some input words fixed. This strategy overcomes the complexity issue in the symbolic-regression-based method. However, the search space is restricted by the limited types of operators and the forms of the target expression. As it is hard to predefine all possible equations, some cases can only be solved using the aforementioned two algorithms. For example, the equation $Y = a_0 X_0 + a_1^{X_1}$ cannot be represented as linear combinations of some fixed terms with unknown constant coefficients.

F. Expression Minimization

In this work, each Verilog operator is associated with a nonnegative constant cost. The cost of an expression is calculated as the sum of the costs of operator appearances in the output Verilog description. We adopt further the method proposed in [20] to minimize the cost of expressions using e-graphs.

The algorithm of expression minimization is described as follows. Given the expressions of all PO words obtained from the three function extraction procedures, because a PO may have multiple equivalent expressions extracted via different methods, the algorithm constructs an e-graph, where equivalent (sub)expressions are grouped into the same e-class. The defined rewriting rules are then applied to the e-graph until saturation as described in Section II-D. Finally, an ILP-based formulation is used to extract an expression of minimum cost from the saturated e-graph.

In our work, based on the rewriting rules in [20], we customize the rules by adding and removing certain rules to better suit the operators under our consideration. Furthermore, for certain rewriting rules $R_1 \to R_2$, their inverse rules $R_2 \to R_1$ are included to explore more equivalent (sub)expressions. As mentioned in [20], sometimes it is necessary to make a transformation that produces more costly expressions in order to achieve better optimization. For example, below are the Verilog snippets obtained with polynomial rewriting of our solution to *test19* before expression optimization.

```
assign out1 = (in6?in5:0)+in2+24*in8+in1;
assign out2 = (in9?in5:0)+in2+24*in8+in1.
```

After the subexpression sharing, they are simplified to

```
assign t0 = in1 + in2;
assign t1 = t0 + 24 * in8;
assign t2 = in5 + t1;
assign out1 = in6 ? t2 : t1;
assign out2 = in9 ? t2 : t1.
```

(The cost is reduced from 10 to 6 under the contest setting.) This optimal result can be obtained only by first moving the additions into the second and third operands of the if-then-else operator, even though the transformation initially leads to a larger cost. As our method allows bi-directional transformations, which allow "up-hill" moves as above, it is less likely to be trapped in a local optimum.

V. Implementation Details

A. Polynomial Rewriting over Modular Arithmetic

Polynomial rewriting over an arithmetic function under truncation can be problematic. The truncation of a word in polynomial rewriting introduces additional monomials, which grow exponentially with respect to the bit-width of the operands of an arithmetic operator. For example, consider the addition of two 2-bit words A and B. If the 3-bit sum is to be truncated to its lowest two bits, the polynomial rewriting yields the polynomial

$$P = 2a_1 + a_0 + 2b_1 + b_0 - 4a_1b_1 - 4a_1a_0b_0 - 4a_0b_1b_0 + 8a_1a_0b_1b_0,$$

in contrast to

$$2a_1 + a_0 + 2b_1 + b_0,$$

the result without truncation. As a result, the truncation highly increases the complexity of rewriting and the difficulty in common subexpression extraction.

Observe that a function truncated to its lowest ℓ bits can be viewed as the same function modulo 2^ℓ. Prior work [21] elaborates the effects of modular arithmetic on the polynomial rewriting for modular multipliers, and proposes a coefficient correction technique to eliminate the extra monomials. In this work, we show that the technique can also be applied to any truncated function. That is, the polynomial without truncation can be rederived by applying the modulo operation on the coefficients. It boosts the efficiency of rewriting due to the reduction in the size of polynomials.

Furthermore, we exploit truncation as an abstraction-refinement strategy for arithmetic function identification. We note that the function of a truncated PO may still be recognized even under an aggressive truncation of the PO to its lowest $k < \ell$ bits before performing polynomial rewriting. While the obtained expression may not correctly represent the original function modulo 2^ℓ, this technique serves as a heuristic with low computation effort to prevent costly rewriting of the whole design. By our empirical experience, using no more than 10 bits is enough to obtain the correct functionality in most cases.

B. Support Information Utilization

Although our method is mainly function-based, we also take structural information into account for performance improvement. For an output word Y, we define its *support words* as the PI and PO words in its transitive fanin cone. Note that a PI support word can be excluded if other support words already form a feasible cut [22]. We limit the solution space to expressions that consist of only the support words. We note that utilizing the support information can significantly improve the performance of symbolic regression.

C. Sign Determination

After extracting an expression E from a PO word Y, the signs of some PI words and Y may be left undecided. We first determine the signs of PI words one at a time and fix the sign-bits of other PI words with unknown signs to 0. The following procedure is repeated for each PI support word X of Y. Let N be the transitive-fanin-cone circuit of Y, and N_s (resp. N_u) be the implementation of E with X assumed to be signed (resp. unsigned). To determine the sign of X, we check whether N_s and N_u are equivalent to N. If only N_s (resp. N_u) is equivalent to N, we let X be a signed (resp. unsigned) word. If both are equivalent to N, we leave the sign of X undecided.[3] If both N_s and N_u are not equivalent to N, E is incorrect when the sign-bit of X is 1. In this case, we assume that X is signed and find a new expression using symbolic regression. The process of sign determination continues after a new expression is found.

Finally, we search for an input assignment that makes the sign-bit of Y become 1. Y is signed if and only if the valuation of E is negative under the assignment.

D. Case Enumeration of Control Inputs

While the proposed methods focus mainly on arithmetic operations,

it tends to fail under the presence of control logic such as multiplexers since the corresponding expression can be very complex. To overcome this issue, if the extraction procedure fails, we break down the functionality of the circuit by enumerating all possible assignments on single-bit inputs. We then perform the extraction procedure under each assignment.

Also, we observe that multiplexers are commonly nested in certain designs. E.g., PO *out3* in case *test15* can be described in part by the Verilog expression

```
in4?out2:(in5?out1:(in6?(in9+in1*in10):in7))
```

where *in4*, *in5*, *in6* are single-bit control inputs of multiplexers. It can be seen that when *in4* is set to 1, the values of *in5* and *in6* do not affect the functionality. These redundant control inputs can be easily found using simulation or CEC. The number of control input assignments can thus be effectively reduced.

VI. Experimental Results

The proposed method, WolFEx, is implemented using C++, Python, and Rust. We use `PySR` [18] as our symbolic regression engine and `egg` [19] for e-graph manipulation and ILP-based expression extraction. In the implementation, a runtime limit is imposed on the subprocedures, including CEC, ILP solving, polynomial rewriting, and symbolic regression. In particular, the CEC procedure returns TRUE if no counterexample is found in time, and the ILP solver returns a sub-optimal solution if the timeout is reached. All experiments were conducted on a Linux machine with 2.2 GHz Intel Xeon CPU and 128 GB RAM.

The CAD Contest benchmarks, including 20 public cases (*test01* to *test20*) and 10 hidden cases (*test21* to *test30*), were taken to evaluate the proposed method. The benchmarks are gate-level netlists that implement some controlled arithmetic functions consisting of word-level operations: Addition (A), Subtraction (S), Multiplication (M), Left/Right Shift (SF), Less than (LT), Less than or Equal to (LE), Greater than (GT), Greater than or Equal to (GE), Equal to (EQ), Conditioning (C),[4] and Bit Selection (BS).[5] According to [6], the costs of the above operations along with the eight types of primitive gates mentioned in Section III are all set to 1.[6] The detail of each case is shown in Table II, where columns "#Gates," "#Bits" "Max #Bits," "Operations," and "#PO Words" report the gate count, the total number of PI/PO signals, the maximum bit width of PI and PO words, the operations used, and the number of single/multi-bit PO words, respectively.

As discussed in Section I, there is no suitable previous work for comparison. However, we compare our method to those of the contest-winning teams. Note that the contest evaluated the final submissions only on 20 out of the 30 cases, including the public cases *test10*, *test12* to *test20*, and the ten hidden ones. Among these 20 cases, four cannot be solved by all teams.

Following the CAD Contest rules, an 8-hour runtime limit for each instance was imposed in our experiment. We note that although the contest evaluation was conducted on a machine different from ours, our program finished in 2 hours for each solved case. WolFEx was evaluated in two different settings: *default* (-d) and *fast* (-f). The *default* setting, which tries all different techniques, is tailored for the best solution quality within the runtime limit. The *fast* setting aims to obtain a *feasible* solution as fast as possible by moving on to the next PO word once a word-level expression is obtained for the current PO word without trying all techniques. Following the definition of the CAD Contest, a solution is called *feasible* if the resulting cost is smaller than 30 % of the cost of the gate-level netlist.

Moreover, another setting *no-pin* (-n) aims to obtain a solution by the modified flow described in Section IV-C without input- and output-pin information. We evaluate its effectiveness by ignoring the input- and output-pin information in the benchmarks. As mentioned in Section IV-C, the *no-pin* setting cannot solve single-bit PO words due to its reliance on the polynomial rewriting technique. Therefore, we assess its performance in its ability to solve multi-bit PO words.

The resulting costs are shown in Table II, where columns "Orig," "Ct Best," "W-f," and "W-d," report the cost of the original (unknown) word-level expression before synthesis, the lowest cost achieved by the participating teams, the cost of our *fast* setting, and the cost of our *default* setting, respectively. Column "W-f Time" reports the runtime in seconds of the *fast* setting, and column "#W-n" reports the number of PO words solved with the *no-pin* setting. Unavailable data are denoted by "N/A," and those of unsolved cases are denoted by "-." The runtime of the unsolved cases is denoted by "TO."

By comparing columns "Ct Best" and "W-d" in Table II, it is clearly seen that W-d outperforms the winning methods of the contest. W-d successfully solved all 20 public and 6 hidden cases. All cases solved in the contest are also solved by our method. Notably, W-d achieved the best costs in all the cases except *test29*. The reason for the bad quality of *test29* may be due to it larger control input bits, which are solved by enumerating multiple control input assignments and expressed using multiple if-then-else operators. Also, the optimization process timed out before the optimal solution was found. Thus, the control logic was not thoroughly optimized in our solution. By comparing columns "Orig" and "W-d," it is seen that our method achieved lower costs than the original word-level expression in cases *test10* and *test19* due to common subexpression sharing as mentioned in Section IV-F.

According to the evaluation criteria [6] of the CAD Contest, the score of each team on each case is evaluated as follows:

$$\text{score} = \frac{\text{minimum (best) cost among all teams}}{\text{cost of the team}}.$$

[3]While the sign of X does not change the valuation of E, it could still affect other POs.

[4]The operator `?:`.

[5]The operator `w[a:b]` or `w[a]`.

[6]Other Verilog keyword statements, such as the declaration of signals, keywords `signed`, `unsigned`, and `assign`, are of cost 0.

TABLE II
RESULTS OF WORD-LEVEL FUNCTION EXTRACTION ON THE CAD CONTEST BENCHMARKS.

Case	#Gates	#Bits		Max #Bits	Operations	Cost				W-f Time	#PO Words		#W-n
		PI	PO			Orig	Ct Best	W-f	W-d		Single	Multi	
test01	47	12	4	4	N/A	N/A	N/A	**2**	**2**	1.52	0	1	1
test02	79	12	4	4	N/A	N/A	N/A	**2**	**2**	1.85	0	1	1
test03	435	57	20	20	N/A	N/A	N/A	**3**	**3**	3.28	0	1	1
test04	9995	80	68	34	N/A	N/A	N/A	**6**	**6**	48.28	0	3	3
test05	1746	64	34	34	N/A	N/A	N/A	**6**	**6**	45.47	0	1	1
test06	6182	96	65	65	N/A	N/A	N/A	**3**	**3**	46.78	0	1	0
test07	3628	50	32	32	N/A	N/A	N/A	10	**8**	47.20	0	1	1
test08	3110	108	64	64	N/A	N/A	N/A	**4**	**4**	68.77	0	1	0
test09	9818	96	41	41	N/A	N/A	N/A	**3**	**3**	152.95	0	1	0
test10	3029	48	64	32	A:3, S:1, M:2	6	**4**	**4**	**4**	39.89	0	2	2
test11	2181	24	40	24	N/A	N/A	N/A	**2**	**2**	22.07	0	2	1
test12	13290	264	26	26	A:11, M:12	23	**23**	**23**	**23**	385.16	0	1	1
test13	1067	48	5	14	S:4, C:6, LT: 5	15	18	**17**	**17**	423.41	5	0	0
test14	107	16	1	9	S:1, LT:1	2	**2**	**2**	**2**	27.83	1	0	0
test15	4231	152	97	33	A:4, M:2, C:3	9	12	**9**	**9**	471.63	0	3	2
test16	999	139	4	36	S:1, LT:1, LE: 1, EQ: 2	5	**5**	**5**	**5**	62.36	4	0	0
test17	1365	95	70	33	A:1, S:2, LT:4	7	**7**	9	**7**	62.60	4	2	2
test18	174	20	17	5	A:3, S:1, LT:2	6	12	8	**7**	4.66	2	3	3
test19	2379	96	285	7	A:45, M:2, BS:1, C:43	91	91	**50**	**50**	8.18	0	43	43
test20	5140	121	17	17	A:6, M:6, C:6	18	**18**	**18**	**18**	162.80	0	1	1
test21	2074	205	13	13	A:37, S:2, C:4, SF:2	45	55	**49**	**49**	42.97	0	1	1
test22	107253	830	69	64	A:12, M:13	25	**25**	**25**	**25**	2442.03	0	1	1
test23	504	55	52	26	A:2, C:1	3	4	**3**	**3**	22.73	0	2	2
test24	18230	977	46	27	A:60, S:1, M:28, SF:3, BS:1	93	N/A	-	-	TO	0	2	0
test25	357	12	27	7	S:1, GE:10, LE:10	21	90	**20**	**20**	15.81	20	1	1
test26	17521	261	32	24	A:13, S:7, M:12, C:2, BS:12	46	N/A	-	-	TO	0	2	0
test27	3539	141	26	13	A:18, S:5, C:25	48	N/A	-	-	TO	0	2	0
test28	8970	132	6	24	A:9, M:12, LT:6	27	N/A	-	-	TO	6	0	0
test29	843	86	26	26	A:8, S:1, M:1, C:7	17	**53**	115	69	1004.93	0	1	0
test30	16525	220	32	32	A:6, M:6, C:6	18	**18**	19	**18**	445.20	0	1	1

The total score is the sum of the scores of the individual cases. Note that for a cost c with the score s on a case, the best cost of the case equals $s \times c$. While the 1st, 2nd, and 3rd places in the CAD Contest received total scores of 11.17, 10.68, and 9.63, respectively, In contrast, W-d attained a total score of 15.74. Evidently, our method outperforms the winning teams in the contest.

For efficiency evaluation, we compared W-f to W-d to investigate how fast a feasible solution can be obtained and how the quality of the feasible solution is. As observed from column "W-f time," although the timeout was set to 8 hours, all public cases and three hidden ones can be solved within 10 minutes. While W-f focuses on efficiency, it yields a higher cost than W-d in only 5 out of the 26 solved cases.

For the no-pin setting, we see that #W-n completely (resp. partially) solved the muti-bit PO words for 17 (resp. 2) cases. (There are 4 cases with no multi-bit PO words.) In these cases, we can obtain the same expressions solved by the original polynomial rewriting technique with the input- and output-pin information, exception for *test06*, *test08*, and *test11*, where there are subexpressions like 2^X or X^k $(k > 1)$ for some PI word X, which are difficult to be recognized without knowing the bit ordering of PI words.

To compare the effectiveness of different function extraction techniques, we applied each technique individually to the benchmarks. The obtained expressions were then minimized using the expression optimization technique mentioned in Section IV-F. As mentioned in Section IV, if the technique fails to find the expressions of some POs, their original gate-level implementations are adopted in the output Verilog file. The experimental results are shown in Table III, where the original cost, the achieved costs, and the runtimes of methods polynomial rewriting (PR), symbolic regression (SR), and linear coefficient fitting (LCF) are shown. In the table, an entry marked by "-" indicates the corresponding method fails to simplify a circuit. The table omits the cases in which all three methods fail. As one can see, PR and SR account for the most optimal solutions, but no one dominates the other as they give better solutions in different cases. On the other hand, LCF provides the best solution for case 21. Also, LCF often reaches optimal or near-optimal solutions in a much shorter time, e.g., in cases 5 and 17. Therefore, LCF is suitable as the first attempt to simplify the circuit, making the whole process more scalable for large circuits.

TABLE III
RESULTS OF APPLYING EACH TECHNIQUE INDIVIDUALLY ON THE CAD CONTEST BENCHMARKS.

Case	Original Cost	Achieved Cost			Runtime (second)		
		PR	SR	LCF	PR	SR	LCF
1	35	**2**	-	**2**	0.01	219.02	0.12
2	58	**2**	-	**2**	0.14	218.81	0.13
3	313	**3**	**3**	**3**	0.02	96.00	0.13
4	7416	**6**	**6**	**6**	20.90	296.45	6.24
5	1290	**6**	-	**6**	236.86	230.90	0.23
6	4959	**3**	**3**	**3**	111.34	151.90	1.63
7	2845	10	**7**	8	81.53	184.45	2.75
8	2696	**4**	-	-	114.56	175.97	4.84
9	7250	-	**3**	-	63.44	246.35	10.11
10	2658	**4**	**4**	**4**	62.93	292.11	1.89
11	1795	**2**	-	**2**	58.99	358.34	0.28
12	11127	**23**	-	**23**	30.31	217.93	41.13
13	931	-	**17**	-	0.00	1095.00	120.00
14	66	66	**2**	3	0.00	23.72	0.29
15	3468	3458	**9**	42	357.49	168.63	8.57
16	721	-	**5**	6	0.00	132.63	6.26
17	938	935	**7**	9	43.40	49.98	0.79
18	94	81	**7**	8	0.01	42.72	1.07
19	1981	**50**	100	91	0.44	1170.44	3.98
20	4836	**18**	-	24	1.03	330.43	52.54
21	1620	53	-	**49**	0.16	419.37	40.12
22	9184	**25**	-	63	2224.58	234.21	73.44
23	299	**3**	-	102	0.01	162.23	0.43
25	215	-	**20**	111	0.00	32.75	13.20
29	688	-	**69**	-	144.30	4592.40	41.68
30	14371	**18**	-	25	96.60	396.54	77.99

VII. CONCLUSIONS

We proposed a word-level function extraction flow combining techniques of polynomial rewriting, symbolic regression, and linear coefficient fitting. It assumes no intermediate word structure and can extract complex functions without matching to some known modules. The extended flow can also handle designs without given input- and output-pin information. Experimental results on the CAD Contest benchmarks demonstrate the superiority of our approach compared to the winning teams in both efficiency and solution quality.

ACKNOWLEDGMENTS

We thank Alan Mishchenko for helpful discussions and comments.

References

[1] C. Saint, *IC Mask Design: Essential Layout Techniques*. McGraw-Hill Education, 2002.

[2] S. E. Quadir *et al.*, "A survey on chip to system reverse engineering," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 1, Apr. 2016.

[3] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[4] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *Proc. DAC*, 2019, pp. 1–6.

[5] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA-2.0: SCA-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal," *IEEE TCAD*, vol. 41, no. 5, pp. 1573–1586, 2022.

[6] C.-H. Chou, C.-J. Hsu, C.-A. Wu, and K.-H. Tu, *Problem A: Learning arithmetic operations from gate-level circuit*, Feb. 2022. [Online]. Available: http://iccad-contest.org/2022/Problems.html.

[7] M. Fyrbiak *et al.*, "HAL—The missing piece of the puzzle for hardware reverse engineering, Trojan detection and insertion," *IEEE TDSC*, vol. 16, no. 3, pp. 498–510, 2019.

[8] L. Azriel, J. Speith, N. Albartus, R. Ginosar, A. Mendelson, and C. Paar, "A survey of algorithmic methods in IC reverse engineering," *J. Cryptogr. Eng.*, vol. 11, no. 3, pp. 299–315, Jul. 2021.

[9] W. Li *et al.*, "WordRev: Finding word-level structures in a sea of bit-level gates," in *Proc. HOST*, 2013, pp. 67–74.

[10] Z. He, Z. Wang, C. Bail, H. Yang, and B. Yu, "Graph learning-based arithmetic block identification," in *Proc. ICCAD*, 2021, pp. 1–8.

[11] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, "Simulation graphs for reverse engineering," in *Proc. FMCAD*, 2015, pp. 152–159.

[12] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *Proc. HOST*, ser. HOST '12, 2012, pp. 83–88.

[13] C. Yu and M. Ciesielski, "Automatic word-level abstraction of datapath," in *Proc. ISCAS*, 2016, pp. 1718–1721.

[14] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, "Formal verification of arithmetic circuits by function extraction," *IEEE TCAD*, vol. 35, no. 12, pp. 2131–2142, 2016.

[15] Y. Wang, N. Wagner, and J. M. Rondinelli, "Symbolic regression in materials science," *MRS Commun.*, vol. 9, no. 3, pp. 793–805, 2019.

[16] I. Icke and J. C. Bongard, "Improving genetic programming based symbolic regression using deterministic machine learning," in *Proc. CEC*, 2013, pp. 1763–1770.

[17] O. Giustolisi and D. A. Savic, "Advances in data-driven analyses and modelling using EPR-MOGA," *J. Hydroinformatics*, vol. 11, no. 3-4, pp. 225–236, Jul. 2009.

[18] M. Cranmer, *PySR: Fast & parallelized symbolic regression in Python/Julia*, Sep. 2020.

[19] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021.

[20] S. Coward, G. A. Constantinides, and T. Drane, "Automatic datapath optimization using e-graphs," Apr. 2022. arXiv: 2204.11478.

[21] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, "Formal verification of modular multipliers using symbolic computer algebra and Boolean satisfiability," in *Proc. DAC*, 2022, pp. 1183–1188.

[22] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proc. ICCAD*, 2005, pp. 519–526.