WordRev: Finding Word-Level Structures in a Sea of Bit-Level Gates

Wenchao Li*, Adria Gascon§, Pramod Subramanyan‡, Wei Yang Tan*,
Ashish Tiwari†, Sharad Malik‡, Natarajan Shankar†, Sanjit A. Seshia*
*University of California, Berkeley {wenchaol, tanweiyang, sseshia}@eecs.berkeley.edu
†SRI International {ashish.tiwari, natarajan.shankar}@sri.com
‡Princeton University {psubrama, sharad}@princeton.edu
§Universitat Politècnica de Catalunya {agascon}@lsi.upc.edu

Abstract—**Systems are increasingly being constructed from off-the-shelf components acquired through a globally distributed and untrusted supply chain. Often only post-synthesis gate-level netlists or actual silicons are available for security inspection. This makes reasoning about hardware trojans particularly challenging given the enormous scale of the problem. Currently, there is no mature methodology that can provide visibility into a bit-level design in terms of high-level components to allow more comprehensive analysis. In this paper, we present a systemic way of automatically deriving word-level structures from the gate-level netlist of a digital circuit. Our framework also provides the possibility for a user to specify sequences of word-level operations and it can extract the collection of gates corresponding to those operations. We demonstrate the effectiveness of our approach on a system-on-a-chip (SoC) design consisting of approximately 400,000 IBM 12SOI cells and several open-source designs.**

I. Introduction

Systems are increasingly being constructed from off-the-shelf components acquired through a globally distributed supply chain. There is a rising concern over the trustworthiness of these components, especially when used in mission-critical applications, as a disruptive threat known as Hardware Trojan begins to surface. Hardware Trojan is a malicious modification of the circuity of an integrated circuit (IC) that aims to compromise the integrity, security and reliability of the IC. A hardware trojan can provide a foothold for software based attacks, where the attacks are orchestrated by colluding sofware [6]. It can also act as a portal for leaking sensitive information [9], or simply subvert the operation of the system under special conditions (e.g. special instruction sequences that trigger the trojan) [7].

The context we address is that of an attacker having maliciously altered the design before or during fabrication. With only a few lines of modification in Hardware Description Languages (HDLs), one can introduce malicious behavior that can undermine the correct operation of the entire system. Such design-time injections are also especially difficult to detect due to possible obfuscation [13] and small physical footprints [15]. Since they may be activated under very specific conditions, they are unlikely to be triggered and detected in simulation or functional tests. Even if suspicion is raised during the operation or inspection of a system, currently there is no way to zoom into a particular portion (say the ALU unit) of a system by simply looking at the overall gate-level netlist. Most high-level structures such as word declaration, modularization, function separation are lost once the design has undergone logic synthesis, thus making it extremely difficult to perform targeted function search in the flattened netlist. In this paper, we present a systematic way of automatically deriving word-level structures from the gate-level netlist of a digital circuit. A *word* is simply a bounded array of bits. A word-level structure then is an operation defined on words. For example, the snippet of Verilog code below describes an 8-bit addition operation.

```
wire [7:0] a, b, c;
assign c = a + b;
```

Our framework also allows the user to specify sequences of word-level operations and it can extract the collection of gates (as well as the appropriate side conditions, for reasons that will become obvious in Section VI) corresponding to those operations.

Algorithmic reverse engineering of a unstructured netlist is particularly challenging due to optimizations performed by RTL synthesis tools. First, hierarchy and module information is lost when the netlist is flattened. Second, logic synthesis techniques such as multi-level logic minimization, technology mapping and retiming further destroy high-level structures in the netlist, and can result in overlapping functional blocks and gate sharing. Both of these make it difficult to apply a direct divide-and-conquer approach.

Starting with a flattened netlist, our approach uncovers the flow of information at the word-level via a combination of three steps. First, it tries to figure out how wires in the netlist may constitute *words*. Having identified some candidate words, it will then try to propagate them across the netlist to infer as many other words as possible. Lastly, it looks for word-level computation that may take place between these inferred words.

Reverse engineering this flow of information brings significant advantages. Instead of looking at individual gates and wires, users can now navigate the netlist at a higher level, by following the flow of *words*. The word-level computation structure extraction process also enables localizing the set of

This work was done when the second author was visiting SRI International.

978-1-4799-0601-7/13/$31.00 ©2013 IEEE

gates for a target function, thus allowing more comprehensive analysis to be performed on a much smaller netlist. Finally, it provides a gateway for automatic inference techniques to be performed directly on a graph formed by the flow of words.

To summarize, we make the following contributions:

- We present a systemtic framework that can extract word-level structures from the bit-level netlist of a design. The extraction process uses a portfolio of algorithms that identifies words, propagates words and reasons about operations between words.
- We study the problem of checking if a logical block in a netlist implements a specific function and formulate it as solving a Quantified Boolean Formula (QBF). This formulation addresses the challenge of having extensive gate sharing in an optimized flattened netlist.
- We demonstrate the effectiveness of our reverse engineering framework on a complete SoC design with approximately 400,000 IBM 12SOI cells and several open-source designs.

The rest of the paper is organized as follows. We begin with a brief survey on related work. Section III presents our solution overview. We describe our word identification algorithm in detail in Section IV, followed by word propagation in Section V, and then by word operation reasoning in Section VI. Section VII reports experimental results. Lastly, we conclude in Section VIII with a discussion on future work.

II. Related Work

In this section, we review related work that aims to derive high-level functions of a design from its gate-level netlist.

Hansen et al. [5] present a study of reverse engineering the well-known ISCAS-85 combinational circuits. They present several strategies, mostly manual, to reverse engineer circuit functionality from a gate-level schematic. Some of these include looking for common library components, repeated structures, computing truth tables of small blocks, and identifying bus structures and control signals. Our work takes this direction further, by providing automatic ways to lift the netlist to the word level and identify logical block that corresponds to word operations. We also demonstrate the effectiveness of our approach on netlists that are several orders of magnitude larger than the ones they considered.

Subramanyan et al. [14] and Li et al. [8] are two recent contributions following a similar direction. In [14], the authors propose a variety of techniques to identify high-level components such as register files, counters, adders and subtracters. Our work complements well with their solution. In fact, words generated by their bitslice aggregation algorithm are used as candidate words in our word propagation technique to infer more words. Our framework also provides additional features, such as the capability of navigating the netlist at the word level and that of handling gate sharing in module identification.

In [8], the authors formalize the problem of reverse engineering the high-level description of a netlist and present a method for matching an unknown sub-circuit against a library of *abstract components*. Our technique is different because we analyze an unstructured netlist as opposed to sub-circuit matching. In addition, their technique assumes the availability of tools to produce candidate sub-circuit. Hence, its usefulness is somewhat contingent upon having good quality (e.g. well functionally separated) sub-circuits. The framework proposed in this paper takes a different angle. We first lift the netlist to a high-level by extracting its word-level information flow, and then reason about operation between words by also taking into account the effect of gate sharing. This helps to functionally isolate logic blocks and complements the work in [8] by generating candidate blocks.

Torrance and James [16] describe the practice of reverse engineering semiconductor-based products. Their approach includes product tear-downs (stripping packaging and disassembling the unit), “system-level analysis” (identifying components on a board and performing functional analysis through probing), process analysis, and circuit extraction (deriving a schematic from a stripped IC). Our work is complementary to this effort. Once a gate-level schematic is derived, our techniques can be applied to to find word-level structures.

Finally, our technique is complementary to other recent work on malicious trojan circuit detection (e.g., [6], [13]). We do not seek to find trojans directly, instead focusing on providing visibility in terms of high-level structures to an unstructured netlist. In the case of encountering suspicious behavior of the circuit, our tool can be used to narrow down to a small set of gates that may be responsible for this behavior, e.g. finding the adder if an “ADD” instruction fails unexpectedly. Additionally, our technique enables scalable analysis to be performed to a high-level abstraction of the netlist.

III. Solution Overview

We approach the problem in three stages. The first stage identifies *candidate words*. We employ two techniques for solving this problem, one based on bitslice aggregation [14] and the other based on a notion called *shapehashing*. The first technique uses functional matching while the second one uses structural information to group “equivalent” wires into words. We discuss these techniques in detail in Section IV. Starting with the candidate words and other known words (such as ones at the primary input and output), the second stage infers more words by iteratively propagating them across gates in the netlist. The final stage performs computation structure extraction for word-level operations, such as addition and rotation. Figure 1 illustrates the overall tool flow.

The word-level program is essentially a combinational circuit with word-level signals and operators. Currently, we support (conditional) assignment, indexing into a sub-word, concatenation, addition, subtraction, Boolean operation, rotation and shifting. For example, the Verilog program below describes a sequence of operation that first conjuncts two 8-bit words and then rotates the lower 4 bits by 2 bits to the left to form a

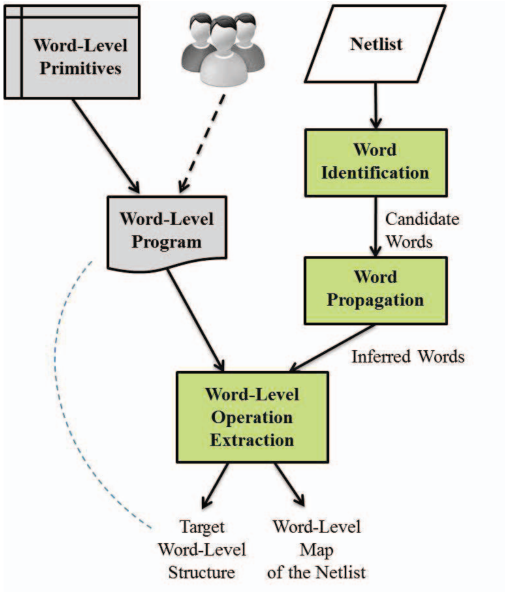


Fig. 1: Overview of the Word-Level Structure Extraction Flow.

new 4-bit word.

```
wire [7:0] a, b;
assign c = a & b;
assign d = {c[1:0],c[3:2]};
```

IV. Word Identification

The goal of word identification is to find wires that can be grouped together into meaningful *words*. A word is a bounded array of bits. We denote w_i as the i^{th} bit of word w, and denote $w_{[i,j]}$ as the subword of w from the i^{th} bit to the j^{th} bit inclusively for $j > i$. In this section, we describe both the structural and functional matching techniques we use to find candidate words given the netlist.

1) Shape Hashing: The idea of *shape hashing* is to assign each wire in the netlist a *shape*, and then create a hash function for all the shapes such that we can easily identify equivalent wires if they have the same shape. The *shape* of a wire w is defined as the directed graph formed by the set of gates *backward reachable*[1] from w. A *k-bounded shape* is simply a shape where all the gates in the graph are backward reachable from the root w within k steps, and at least one of them is reachable at exactly k steps. When the directed graph is cyclic, we unroll the loops by duplicating gates along the loops until the gates are not backward reachable from w in k steps. In our experiments, we used all values of $k \in \{2, 3, 4\}$[2].

To compute a hash key from each shape efficiently, we perform a depth-first-search traversal of the DAG (i.e. shape) backward starting from w to produce a serialization of the DAG using gate and wire types. Multiple children gates in the traversal are tie-broken lexicographically. However, we do not check for graph isomorphism, especially one that is induced by having gates with commutative ports, for efficiency reasons.

We further refine the equivalence classes by taking into account the relative location of the wires. We define the distance between two wires as the smallest number of gates between them in the netlist. With this distance measure, we form equivalence subclasses for wires that have the same shape. In each subclass, the wires are located to one another by at most d distance, where d is the cardinality of the original equivalent class. The collection of wires in each subclass then forms a candidate word. Wires of the same word are ordered arbitrarily.

2) Bitslice Aggregation [14]: In addition to finding structurally similar wires, we also employ the functional matching approach described in [14] for identifying functionally equivalent wires. The function of a wire w in the netlist can be characterized by a *feasible cut* of w. This is defined as a set of wires in the transitive fan-in cone of w such that an arbitrary assignment of truth values to each wire in the set completely determines the value of w [3]. A cut is said to be *k-feasible* if it has no more than k inputs. As in [14], we enumerate the set of 6-*feasible cuts* for each node. Each cut then forms a *bitslice* rooted at w, which is a Booolean function with a single output and no more than 6 inputs.[3] Once all the bitslices are identified, they can be grouped into equivalence classes using permutation-independent Boolean matching. For example, a bitslice matching the function $y = ab + c$ and a bitslice matching the function $y = bc + a$ are grouped into the same class.

Now that we have found all the duplicated bitslices across the netlist that compute a particular function, we can look for *aggregates* of them that are connected in specific patterns. Following the approach described in [14], we group all matching bitslices that (1) have a common select signal; (2) the output of one bitslice feeds to the input of another (e.g. carry chain in a ripple carry adder). Since aggregated bitslices are essentially circuits that operate on sets of nodes simultaneously. We group the inputs or outputs of aggregated bitslices together to form candidate words. Note that in the case of a carry chain, the words are ordered in the carry direction.

V. Word Propagation

An important piece of the overall word structure extraction process is an algorithm for propagating words. Intuitively, we want to see if arbitrary values of a word can propagate across a set of gates to reach a new set of wires. To do this efficiently, we use symbolic evaluation [2], which allows the evaluation of a set of values simultaneously in a single run.

[1] In general, one can use both forward and backward reachable gates to assess structural similarity. In our case, the set of backward reachable gates also describes a Boolean function with a single output at w.

[2] As k increases further, even wires originally declared as parts of the same word become structurally dissimilar in an optimized netlist.

[3] The number 6 is chosen for efficiency reasons, as the number of cuts for $k > 6$ is significantly higher. Also, most common bitslices have less than six inputs, e.g., a full adder bitslice has 3 inputs [14].

Similar to Roth's D-calculus [12], we redefine the functions of the logic gates in the netlist to operate over the expanded domain $\{0, 1, D, \overline{D}, X\}$, where D represents a symbolic value in $\{0, 1\}$, $\overline{D}$ is the negation of D, and X represents an unknown/undetermined value. Figure 2 shows two examples of symbolic evaluation for basic Boolean gates.

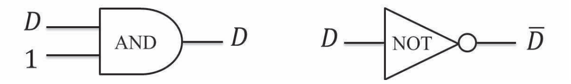
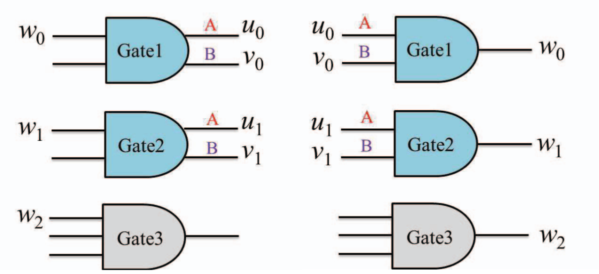


Fig. 2: D-calculus Examples for Boolean "AND" and "NOT"

Our solution for word propagation uses a "guess-and-check" approach. Starting from some known or candidate word, we first try to find a set of wires that are located one level of gates away from this word. For simplicity of discussion, we search *forward* for such a set of wires, using a procedure called **FindForwardWordPairs**. This set of wires constitutes a target word, i.e. the word to propagate to. Next, we construct a netlist slice for symbolic evaluation using **SetupSymbolicEvaluation**. Finally, we check if symbolic values of the source word can indeed be propagated to the target word, by using the procedure **TryPropagate**.

FindForwardWordPairs, i.e., the "guessing" stage of the algorithm, consists of finding promising target words. Figure 3 shows two structural heuristics we use to find target words forward and backward. The idea is to group gates according to their function types and group wires according to the ports they connect to. For instance, consider the source word w as shown in Figure 3a, which is 3-bit wide, and assume that "Gate1" and "Gate2" have the same function type, we guess two target words u and v, each of 2-bit wide. We then check if the subword $w_{[0,1]}$ can propagate to u and v respectively.



(a) Forward Search (b) Backward Search

Fig. 3: Structural Heuristics for Finding Target Words from Source Word w. Gates of different function types are colored differently.

SetupSymbolicEvaluation is the first step of the "checking" part of our algorithm. In general, if we just evaluate the gates between the source word and the target word, by assigning X to the other inputs of these gates, we would not be able to propagate to the target word. The key insight here, which leads to effective word propagation, as we will show in Section VII, is that if the target word is indeed a word, then often times there exist a few *nearby* (in the fan-in cone of these gates) wires which behave as *control signals*. This means that for some specific concrete assignments to these control signals, the target word will take the values of the source word (or their negation). This pattern is quite common in digital circuits, such as in the case of a multiplexer, or a conditional assignment in an "if-then-else" statement. In fact, even if the condition involvess many signals and Boolean operators, the synthesis tool will likely create a wire in the netlist that is the evaluation result of this condition. We elaborate on how we make use of this insight in Figure 4.

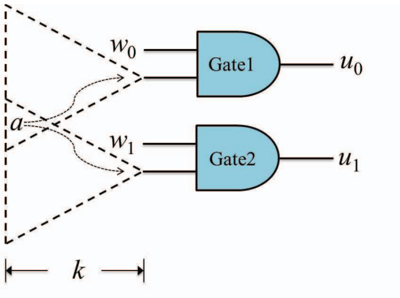


Fig. 4: To check if word $w = [w_0, w_1]$ propagates to word $u = [u_0, u_1]$, we find potential control signals such as a and include them as inputs in the netlist slice on which symbolic evaluation is performed.

We consider wires and gates in the fan-ins of "Gate1" and "Gate2", up to some small depth k. Any wire that lies in the intersection of these fan-ins is treated as *control*, e.g. wire a. We construct a netlist slice including the gates between the source word and the target word, as well as the gates in the aforementioned fan-ins. The fan-in gates are aggregated in a backward traversal manner up to depth k or when a sequential logic cell such as a flip-flop is reached. This ensures that the overall netlist slice forms a combinational circuit. This is the netlist that we will symbolically evaluate, and is set up in the following way.

- Source word: each bit is assigned the symbolic value D.
- Control wires: If the number of control wires is greater than 3, we enumerate all combinations of size 3 for the set of wires. In each combination, we further try all possible concrete assignments to the wires involved. The election of the constant value 3 is related to the insight mentioned above that the synthesis tool will likely add wires capturing the evaluation of complicated conditions.
- Other inputs: each wire is assigned the unknown value X.

TryPropagate is the second piece of the "checking" stage of the algorithm. For each set of concrete assignments to condition wires, the netlist slice is symbolically evaluated afresh. Because the netlist slice is a combinational circuit, symbolic evaluation can be done efficiently by evaluating each gate in the slice in a topological order. If every bit of the target word is evaluated to D or $\overline{D}$ for any concrete assignment to the control wires, then the target word is considered as a *true* word. In this case, the propagation process iterates and tries to use this new word to infer more words.

The overall algorithm is summarized in Algorithm 1. For simplicity, we show the only parts for forward propagation. The algorithm iterates over the set of source words W. At each iteration, a word w is removed from the source pool. **FindForwardWordPairs** then uses the structural heuristics described in Figure 3a to find promising pairs of words to test for propagation. For each pair of source word u and target word v, if no propagation has been attempted so far from u to v, the pair is added to $\mathcal{H}$ and we proceed to checking if u can propagate to v. This is achieved by **SetupSymbolicEval** which first sets up the netlist slice C' and control wires $\boldsymbol{cw}$ for symbolic evaluation as described in Figure 4, and **TryPropagate** which then evaluates C'. If propagation succeeds, this means we have verified u and v are words and we add them to the set of inferred words $\mathcal{W}'$. Since v may be further propagated forward, we add it to the set of source words $\mathcal{W}$. The algorithm terminates when we have tried all words that can be inferred from, i.e. $\mathcal{W}$ is depleted.

Algorithm 1 Symbolic Evaluation for Propagating Words Forward

```
Input: the set of candidate/source words $\mathcal{W}$, netlist $C$.
Output: the set of inferred words $\mathcal{W}'$.
Initialize: set $\mathcal{H}$ to $\emptyset$.
while $\mathcal{W} \neq \emptyset$ do
    $w$ = pop($\mathcal{W}$)
    $\mathcal{P}$ = FindForwardWordPairs($w$)
    for $(u, v) \in \mathcal{P}$ do
        if $(u, v) \notin \mathcal{H}$ then
            Add $(u, v)$ to $\mathcal{H}$.
            ($C'$,$\boldsymbol{cw}$) = SetupSymbolicEval($C$,$u$,$v$)
            if TryPropagate($C'$,$\boldsymbol{cw}$,$u$,$v$) succeeds then
                Add $u, v$ to $\mathcal{W}'$.
                if $v \notin \mathcal{W}$ then
                    Add $v$ to $\mathcal{W}$.
                end if
            end if
        end if
    end for
end while
```

To assist users with visualizing the data flow and to enable a higher level of inference, we also create a directed graph where each node in the graph represents an unique word and there is an edge from node u to node v if the word u can propagate to word v. We will explain this in detail with a case study in Section VII.

VI. Reasoning Word Operation

Now that we have all the words that can be inferred by propagation, we are interested in finding computation structures that operate on these words. The main idea is to cut out the portion of the netlist that lies *between words*, and then check if this structure implements a particular word operation. We currently support only operations that are combinational logic. However, these still include many that are commonly found in circuit designs, such as addition, subtraction, Boolean operation (e.g. NOT, AND, OR, XOR) and shifting/rotation. Note that the user can extend this set with other word operations by providing reference models for those operations.

To extract the netlist between words, we first arrange the words in topological order between sequential boundaries. For example, given three words w^a, w^b, w^c of the same width such that w^a and w^b are followed by w^c in the topological order, and we are interested in checking if $w^c = w^a + w^b$ modulo a carry-in, we can form a netlist slice by using the gates between w^a and w^c and those between w^b and w^c.

Due to optimizations performed by the synthesis tools, gates are often shared among different functions to minimize area or delay. For example, instead of having a separate set of gates implementing each opcode function, a single set of gates between the inputs and outputs of an ALU unit will suffice for all operations, each having overlapping logical block with another. In fact, gate-sharing can be a result of design choice and customization. The ripple-carry adder-subtractor design demonstrates this phenomenon where a single signal value can convert the adder to a subtractor.

The consequence of these is that the word operation may be embedded in the netlist slice which has side input signals. Depending on the assignment to these side inputs, the netlist may or may not behave according to a certain operation, e.g. addition or subtraction.

Our solution is to model the problem into a Quantified Boolean Formula (QBF) and make use of state-of-the-art QBF solvers to solve it. QBF is the canonical complete problem for PSPACE. It extends propositional formulas by including the universal quantifier $\forall$ and existential quantifier $\exists$. The particular instance of QBF we have formulated here involve a single alternation of $\forall$ and $\exists$, which is also known as 2QBF. Figure 5 illustrates the miter construction for creating the 2QBF formula.

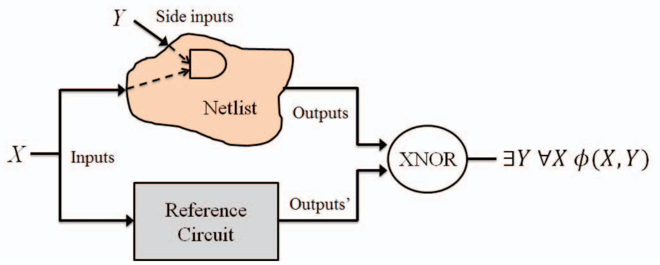


Fig. 5: Miter Construction and QBF Formulation

The reference circuit for the word operation we are interested in checking has inputs X. However, the extracted netlist slice also contains side inputs Y in addition to X. We can describe the miter circuit (as typically done for SAT-based combinational equivalence checking [4] in the verification literature) with a single Boolean formula ϕ such that ϕ is true if and only if the two circuits are equivalent. The intuition behind the existential quantification over Y is that if the netlist

TABLE I: Benchmark Netlist Information

Design	Gates	Nets	Latches	Description
router	896	984	182	CMP Router
open8	1807	1812	237	Open8 CPU
Cpu8080	2258	2368	243	8080 CPU
MIPS16	6986	11110	4380	MIPS-like core
oc8051	8093	10210	2748	8051 μcontroller
RISC FPU	14291	15740	3097	RISC FPU
BigSoC	375090	231736	34318	SoC

TABLE II: WordRev Statistics

Design	Input	Output	Cand.	Prop.	Runtime (min)
router	2	2	0	54	1.3
Open8	2	2	22	98	80.0
cpu8080	1	2	6	174	114.2
MIPS16	0	1	2	4	1.0
oc8051	6	7	12	113	329.9
RISC FPU	1	2	128	142	154.6
BigSoC	1	4	16	865	311.2

implements the function of the reference circuit, then there must exist a way to configure the side inputs for it to do so. For the previous example of checking if $w^c = w^a + w^b$, we now have inputs as w^a and w^b and outputs as w^c. The formula ϕ desribes the comparison of the netlist with the disjunction of two circuits, one implements the addition with carry-in equal to 1, and the other with carry-in equal to 0.[4]

An important assumption that we make here is that the ordering of the bits in the input and output words are known, with respect to those of the reference circuit. Li et al. [8] suggest a way to resolve correspondence between signals through behavioral mining from simulation traces. We plan to investigate this further in future work.

VII. Experiments

We have developed an inference tool using Python and C++ that implements the algorithms described in this paper. The tool takes as input a synthesized netlist, analyzes it and produces as output the set of word-level structures in the netlist, as well as the connectivity between them in the form of a directed graph. The tool uses DepQBF [10] as the backend solver for solving QBFs. Table I summarizes the netlist information of the designs that are used in this paper. The generic name BigSoC is used for confidentiality reasons. All the designs were synthesized with an IBM/ARM cell library for a 45nm SOI process using the Synopsys Design Compiler with its default optimization setting.

The CMP router is a simplified version of the chip-multiprocessor router proposed in [11]. BigSoC is a system-on-a-chip design which consists of 7 subsystems: a 32-bit ARM-compatible RSIC processor, a Singular Value Decomposition module, a SPI interface, a UART interface, an I^{2}C interface, a VGA controller and a memory controller. The subsystems are further interconnected through an AXI4S switch. The rest of the benchmarks are available on OpenCores [1].

A. WordRev

Table II summarizes the results of applying WordRev to the netlist designs. Columns 2 and 3 record the number of input and output words used respectively. Column 4 record the number of candidate words identified using the algorithms in Section IV. Column 5 is the number of words produced by word propagation. Column 6 is the runtime for word propagation in minutes. In all experiments, we limited the search to only words between 4 and 32 bits wide.

Input and output words were assumed to be known, since these are at the I/O of the design. Candidate words were selected from the word identification step described in Section IV. For BigSoC, we only selected 16 32-bit words identified by using the algorithm in Section IV-2. This is because a lot of words were identified as candidate words, which caused the tool to timeout as a result of trying to propagate each of them. Additionally, we consider words that can be propagated to be more valuable than just identified words, since propagation indicates that these arrays of bits are more likely to be operated together at the RTL level.

B. CMP Router

In this section, we use the CMP router to evaluate the effectiveness of WordRev in detail. Particularly, we focus on the following analysis.

- Usefulness of the flow of word-level information presented as a directed graph.
- Effect of the netlist being synthesized from different cell libraries.
- Effect of the netlist being synthesized from the same cell library but with different optimization settings.

For convenience, we called the router netlist used in the previous section the “TR Router”, the resulting netlist synthesized from a different cell library the “TS Router”, and the optimized netlist the “Optimized TR Router”.

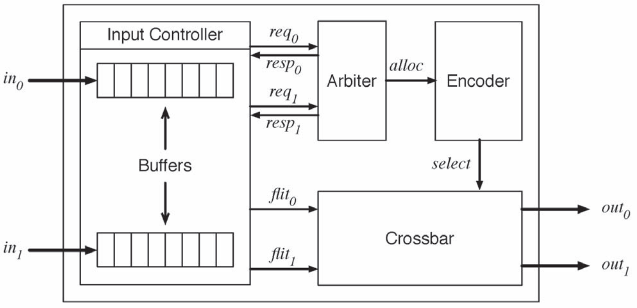


Fig. 6: CMP Router Comprising Four High-Level Modules

The overview of the CMP router design is illustrated in Figure 6. It is a composition of four high-level modules. The *input controller* comprises a set of FIFOs buffering incoming

[4] We consider two possible reference circuits that the netlist can be matched to because the carry-in bit is not a primary input to the miter circuit.

flits and interacting with the *arbiter*. A *flit* is a flow control unit. A data packet is composed of multiple flits. In this implementation, a flit has 12 bits, with the lower two fits as a header (used for channel reservation) and the remaining 10 bits as address (6 bits) and data (4 bits). Each *input controller* contains a circular FIFO buffer of 4 flits deep. When the arbiter grants access to a particular output port, it sends a signal to the input controller to release the flits from the buffers, and at the same time, it sends the allocation information to the *encoder* which in turn configures the *crossbar* to route the flits to the appropriate output port.

Word Graph: Figure 7 shows the word-level graph produced by our tool. We have highlighted regions of the graph that correspond to high-level modules of the router as desribed previously. The nodes in yellow are the known words that we start with at the primary input and output of the router. The number in each node denotes the numbering of the word (e.g. "w11") followed by the width of the word (e.g. 12 for "w11"). If a node is contained in another node, it indicates that the inner word is a subword of the outer word.

The top half of the word graph is in fact isomorphic to the bottom half, each corresponding to a port of the router. The input controller FIFO is the subgraph that has two special nodes, marked as "Writing into FIFO" and "Reading from FIFO". From the first node, which is the write pointer of the FIFO, it splits to four other words. Each of these words is then latched (as indicated by the red arrow). The latched output has three possible out-going paths: one going back to the itself (typical for state holding elements), one goes to the write pointer of the FIFO (multiplexed to determine whether it will get overwritten), and the last one goes to the read pointer of the FIFO (another multiplexing for determining which flit will be read). The crossbar on the right is easier to recognize – it consists of two 12-bit words that interweave to two other 12-bit words downstream. While the module identification described here was performed manually, comparing to Figure 6, the word graph in Figure 7 essentially reduces the router netlist with approximately 1100 cells to a single graph (which can fit on a page) that captures most aspects of the data-flow and architectural features of the router. This allows to a human analyst to look for patterns at a higher level of abstraction. Moreover, the word graph provides a structure for further automation, such as module identification, which we plan to explore in future work.

Different Cell Libraries: "TS Router" had about 4% less gates than "TR Router", but had the same number of latches.[5] Applying the same word propagation algorithm to "TS Router", 52 (instead of 54) words were found. Comparing the words inferred in the two netlists directly was difficult since the wires were named differently. However, upon close examination of the word graphs, we verified that the topology of the "TS Router" word graph is almost identifical to that of "TR Router". The only differences were at the read and write pointers of the FIFOs. In addition, the widths of the words were also the same, showing the splitting of I/O words into words of 10 bits wide. This shows that our heuristic for finding target words to propagate is robust to small change in the cell library used in logic synthesis.

Different Synthesis Parameters: "Optimized TR Router" contained about 25% less gates than "TR Router".[6] In addition, it used 176 instead of 182 latches. In this case, WordRev found 50 words, 4 less than the number of words found in "TR Router". We analyzed the word graph generated for "Optimized TR Router" and found interesting discrepancies. While the topology of the graph remained largely the same, which means the key features were again visible, the words being propagated were only 4 bits wide. In fact, they correspond to the 4 data bits in each flit. In "TR Router", both the 6-bit wide address field and the 4-bit wide data field were propagated together. This shows that while our algorithm can extract the word-level dataflow of a netlist, its performance can be influenced by the optimization setting during logic synthesis.

C. OC8051 Microcontroller

In this section, we focus on reporting results that demonstrate the effectiveness of using the QBF formulation to identify structures and conditions in reasoning various word operations.

TABLE III: QBF Statistics for Different Operations

Operation	Num of Vars	Num of Clauses	Runtime (s)
Addition	4525	11974	60
Subtraction	4438	11744	50
Rotate-right	4332	11416	30
Rotate-left	4332	11416	30
Not	4333	11415	64
And	4337	11428	34
Or	4337	11428	51

The OC8051 microcontroller is widely used in many embedded system products. It contains an 8-bit CPU optimized for control applications. Using word identification techniques described in Section IV and the structure extraction process described in Section VI, we extracted a netlist slice that was part of the ALU unit of the microcontroller. This netlist slice contained 435 12SOI gates. In addition to the two 8-bit input words, the netlist slice had 87 other side inputs. We formulated a QBF problem for each of the word-level operation (each word is 8-bit wide and each operation corresponds to a specific ALU opcode) shown in Table III, and verified that the *single* netlist slice could implement that operation by making appropriate assignments to the side inputs.

VIII. Conclusion

This paper describes a systematic way to reverse engineer word-level structures from an unstructured netlist. Our solution

[5] The "TS" and "TR" cell libraries differ only in the layout implementation, where one is speed optimized and the other is size optimized. Their logical functions are the same.

[6] "Optimized TR Router" was the result of using the additional Synopsys "compile_ultra" command, which does extra optimizations for high-performance (i.e., high clock frequency) designs.

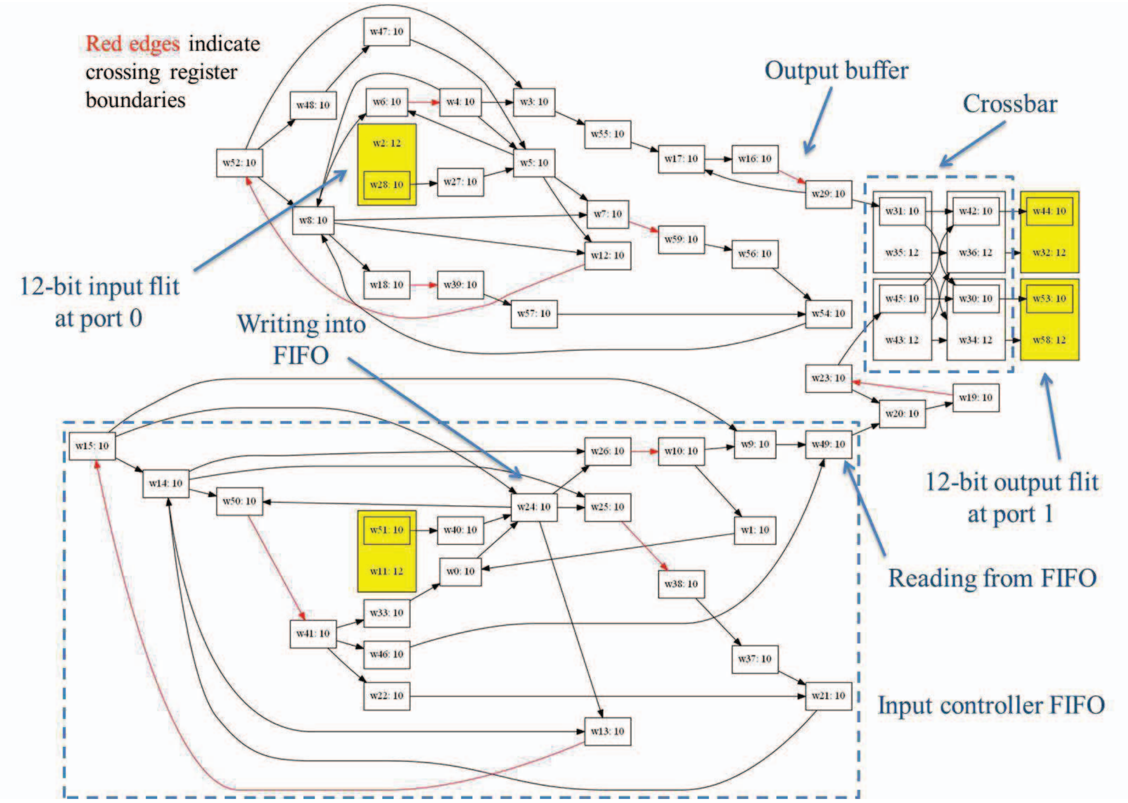


Fig. 7: Word Graph for the Same CMP Router Design

uses a portfolio of algorithms such as bitslice aggregation, shapehashing, symbolic evaluation and QBF solving for identifying and inferring these word structures. In addition, this work offers new possibilities in reverse engineering the high-level functions of a circuit. The word-level map of the circuit essentially lifts the netlist to a higher level of abstraction, where subgraph matching methods can be directly applied to find interesting circuit structures, such as FIFOs and crossbars. Hence, our approach allows the user to focus on the functionality of concrete parts of the circuit and make (possibly automated) inferences with the aim of identifying parts of the circuit that can be then extracted and independently formally verified. A natural extension of this work is to allow the user to define custom word operations that are implemented with sequential logic. We are currently investigating this extension.

References

[1] Opencores. http://opencores.org/.

[2] R. Bryant. Symbolic simulation-techniques and applications. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 517 –521, Jun 1990.

[3] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. Reducing structural bias in technology mapping. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 519 – 526, Nov 2005.

[4] E. Goldberg, M. Prasad, and R. Brayton. Using sat for combinational equivalence checking. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

[5] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[6] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 159 –172, May 2010.

[7] Y. Jin, N. Kupp, and Y. Makris. Experiences in hardware trojan design and implementation. In *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, pages 50 –57, july 2009.

[8] W. Li, Z. Wasson, and S. A. Seshia. Reverse engineering circuits using behavioral pattern mining. In *Proceedings of the IEEE Conference on Hardware-Oriented Security and Trust (HOST)*, Jun 2012.

[9] L. Lin, M. Kasper, T. Gneysu, C. Paar, and W. Burleson. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 382–395. Springer, 2009.

[10] F. Lonsing and A. Biere. Depqbf: A dependency-aware qbf solver. *JSAT*, 7(2-3):71–76, 2010.

[11] L.-S. Peh. *Flow control and micro-architectural mechanisms for extending the performance of interconnection networks*. PhD thesis, 2001.

[12] J. P. Roth. *Computer Logic, Testing and Verification*. W. H. Freeman & Co., New York, NY, USA, 1980.

[13] C. Sturton, M. Hicks, D. Wagner, and S. King. Defeating uci: Building stealthy and malicious hardware. In *Security and Privacy (SP), IEEE Symposium on*, pages 64 –77, May 2011.

[14] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik. Reverse engineering digital circuits using functional analysis. In *Design Automation and Test in Europe (DATE)*, Mar 2013.

[15] M. Tehranipour and F. Koushanfar. A survey of trojan taxonomy and detection. *IEEE Design & Test of Computers*, 27:10 – 25, 2010 2010.

[16] R. Torrance and D. James. The state-of-the-art in IC reverse engineering. In *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5747 of *Lecture Notes in Computer Science*, pages 363–381, 2009.