Identifying Transparent Logic in Gate-Level Circuits

Yu-Yun Dai[1], Robert K. Brayton[2]
[1,2]Department of EECS, University of California, Berkeley, U.S.A.
{[1]yunmeow, [2]brayton} @berkeley.edu

ABSTRACT

Many reasons exist for high-level information to be unavailable for a design. Identifying high-level constructs, e.g. control paths and words, from gate-level circuits, can assist verification, equivalence checking, reverse engineering, etc.. Word-level identification can be done by structural methods, but we focus on functional approaches because they only depend on dependencies between signals of a circuit. We introduce *transparent logic*, based on *functional isomorphism*, and provide algorithms to recognize control signals, data paths, internal words, and boundaries between different types of logic. Experiments show that the proposed algorithms can re-assemble words effectively from unstructured or synthesized circuits.

1. INTRODUCTION

In hardware, control logic regulates the data flow and dictates circuit functionalities. Logic can be classified as that where data is simply moved from one part of a circuit to another part without modifying it. Such logic is referred to as *transparent*. Another category transforms data by some word-level operator, e.g. a bit-vector operator defined in Verilog. A third category, control, determines which data is moved and when, or which operation is applied and when. Efficient recognition of such logic can benefit circuit verification, e.g. [4] as a guide to abstraction. To identify word-level operators in a gate-level circuit, it is crucial to find words and locate the boundaries (inputs/outputs) of arithmetic operators [10].

The basic example of transparent logic is a multiplexer (MUX) structure, which selects from several data signals and forwards it unaltered towards the outputs. Identification of MUXes can be performed over gate-level circuits very quickly using structural matching, but can be unreliable, especially if synthesis has been applied.

In this paper, we focus on functional approaches which do not depend on the actual gate-level structure of the circuit. These can augment structure methods and provide a much more reliable technique as we show in the experiments.

In general, functional methods, which rely only on functional dependencies, have been used to augment structural approaches. Examples are,

- Li and Subramanyan et. al. [6, 11] identified internal words based on *bitslice aggregation* (functional approach) and *shapehashing* (structural approach.) The candidate words found were used as boundaries of operators for further recognizing.

- Li et. al. [6, 7] identified functional operators in gate-level circuits, based on an existing library of blocks. Word-level information at the primary inputs was assumed available, but in many applications this information is not known.

- Sterin et. al. [10] extracted word-level operators functionally, given a library of operators and a slice of logic containing inputs and outputs of such operators. Word-level information was not required nor was the possible location and ordering of the inputs and outputs of an operator.

We present methods to identify *functional transparent logic*. This is inherited from *functional isomorphism*. Using this, we propose an algorithm to identify words, word-level operator boundaries, and control logic in gate-level circuits and apply this to a variety of test cases. Once operator boundaries are located (roughly), techniques like those in Sterin et. al. [10] can be used to identify the more precise location of the operators as well as their functionalities.

The paper is organized as follows. Section 2 introduces *functional* isomorphism. In Section 3, we describe the definition and propagation of *transparent* logic. Proposed algorithms for identifying transparent logic are given in Section 4. Experimental results are shown in Section 5, while Section 6 concludes this paper.

2. OVERVIEW

Roughly, a transparent path in a circuit has width n and a set of controls $\{s^i\}$, which when evaluated appropriately at a minterm $s^i = m_{s^i}$, moves a data-word (width n) from the beginning of the path to the end. Such paths can fork and join in the circuit, and can begin and end at a set of inputs, outputs or internal signals. A path is maximal if there is no transparent path that can extend it. The terminals of maximal transparent paths are of interest because they likely delineate the input or output of an operator, e.g. an arithmetic function.

A sink terminal can have many source terminals. Each signal in a sink terminal is a Boolean function of a) data signals $D_k = \{d_k^j\}$ in the source terminals and b) the set of associated controls $\{s^i\}$ of transparent segments of any path from source to sink. Such a set of functions at a sink forms an npn equivalence class (or equivalently an npn isomorphism class). Each sink bit function typically (with some exceptions) looks like $f_k = \sum_{j \in D_k} (\prod_{i \in path} m_{s^i}) d_k^j$. The isomorphism between the inputs of any two signals f_p and f_q (where $p, q \in [1, n]$) in the sink terminal is $d_p^j \leftrightarrow d_q^j$ i.e. different bit positions in the same data word are isomorphically mapped to each other, while control signals s^i are isomorphically mapped into themselves. Each coefficient $(\prod_{i \in path} m_{s^i})$ is the predicate of the control signals under which a path from source d_k^j to the sink f_k becomes transparent. The predicates are disjoint. It is possible that some bits of a terminal have been inverted, hence npn equivalence is considered in the subsequent discussions.

Thus a transparent path's outputs is a subset of an npn

isomorphism class. The isomorphism helps distinguish between different data-words by factoring out common predicates ($\prod_{i \in path} m_{s^i}$) in the representative function of the equivalence class.

2.1 Npn Isomorphism

Two graphs, $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are *isomorphic*, if there exists a bijective mapping, $\mathbf{M}_{12}$: $V_1 \rightarrow V_2$, such that any two vertices u and v are adjacent in G_1, if and only if $\mathbf{M}_{12}(u)$ and $\mathbf{M}_{12}(v)$ are adjacent in G_2 [2]. Two circuits, C_1 and C_2, are isomorphic to each other if their logic gates and connections form two isomorphic graphs, while any gate g of C_1 and the mapped gate $\mathbf{M}_{12}(g)$ in C_2 are the same type. The relation between C_1 and C_2 is called *structural isomorphism*, which has been applied to reverse engineering [5].

In contrast, *functional isomorphism* is a relation between two signals in a circuit. A signal f in a circuit, supported by a set of other signals, S_f, is a Boolean function of these inputs: $f : \mathbf{B}^{|S_f|} \rightarrow \mathbf{B}$, for $\mathbf{B} = \{0, 1\}$.

In the following sections, for a Boolean variable x_i with its polarity p_i, $(x_i)^{p_i}$ represents the function: $p_i = 0 \rightarrow (x_i)^{p_i} \equiv x_i$ and $p_i = 1 \rightarrow (x_i)^{p_i} \equiv inv(x_i)$.

Definition 1: A pair of Boolean functions $f(x_1, \ldots x_n)$ and $g(y_1, \ldots, y_n)$ are *npn isomorphic*[1], if there exists a permutation π of size n and polarities p_{out} and $\{p_1, \ldots, p_n\} \in \mathbf{B}^n$ such that

$$f(x_1, \ldots, x_n) = g^{p_{out}}(x_{\pi(1)}^{p_1}, \ldots, x_{\pi(n)}^{p_n}) \quad (1)$$

i.e., g can be made equivalent to f by selectively negating inputs, permuting inputs, and negating the output. The implied isomorphic mapping between the supports of g and f is $\{y_i, x_{\pi(i)}^{p_i}\}$ and p_i is said to be the relative polarity between inputs y_i and $x_{\pi(i)}$.

A set of signals in a circuit, in which every pair is functionally npn isomorphic is called an *npn isomorphism class*.

2.2 Composition of Npn Isomorphism

Although improved methods for computing npn equivalence can be found in Soekin et. al. [9], this still can be time-consuming. This effort can be reduced immensely by proving npn isomorphism on smaller logic blocks and composing proved classes to obtain larger ones. Larger classes help extend paths of transparency (discussed in Section 3) in a circuit and to more reliably find transparency boundaries, and hence the input/output boundaries of word-level operators.

The following discussion details when compositions lead to larger npn isomorphisms.

Definition 2: (*polar consistency*) Let $(f(s), g(t))$ be a pair of npn isomorphic functions with sets of supports $s = \{s_i\}$ and $t = \{t_j\}$, respectively. Suppose each pair of mapped input supports, $s_i \leftrightarrow t_j$, are npn isomorphic functions, i.e. $s_i(x)$ is npn isomorphic to $t_j(y)$. Let p_{out}^{ij} be the relative output polarity between $s_i(x)$ and $t_j(y)$, and p_{ij} be the relative input polarity between inputs s_i and t_j in the npn isomorphism between $f(s)$ and $g(t)$. The compositions $f(s(x))$ and $g(t(y))$ are *polar consistent*, if $p_{out}^{i\pi(i)} = p_{i\pi(i)}$, where π is the permutation in the isomorphism mapping of $(f(s), g(t))$.

Theorem 1: The compositions of $(f(s(x)), g(t(y)))$ are polar consistent if and only if $f(s(x))$ and $g(t(y))$ are npn isomorphic.

[1]or Negation-Permutation-Negation (NPN) equivalent

3. TRANSPARENT LOGIC

As stated identifying maximal *transparent logic*, can be used to identify input/output boundaries of arithmetic operators.

3.1 Transparent Words

Intuitively, a *transparent word* is a set of signals, $\{w^k\}$, with supports, $\{S^k\}$, where under some evaluation of $\cap^k S^k$ (*common control*), $\{w^k\}$ is equivalent to a subset (*data-word*) of $\cup^k S^k$. In other words, the control evaluation makes the word transparent from some input data-word.

Example: Outputs of a set of 2-to-1 multiplexers (MUX) controlled by the same selector signal s,

$$C[m-1:0] = s?A[m-1:0] : B[m-1:0], \quad (2)$$

comprises a transparent word C, where $\forall j \in [0, m-1]$, $(C[j] = sA[j] + s'B[j])$. For this case, word C is transparent from word A or word B, depending on the value assigned to s.

Definition 3: Functions $W = \{w^k | k \in [1, m]\}$ of an npn isomorphism class comprise an m-bit *transparent word*, if:

1. Each function $w^k : \mathbf{B}^{S^k} \rightarrow \mathbf{B}$, has support $S^k = (Control, Data^k)$, (i.e. *Control* is the set of common signals), and each bit of *Control* is isomorphically mapped into itself.

2. The following formula is *True*, (where m_c is a minterm of *Control*, $\equiv$ denotes functional equivalence, and $w_{m_c}^k$ denotes the co-factor of function $w^k(Control, Data^k)$ with respect to m_c).

$$(\exists_{m_c} \forall_k \exists_{d_i^k \in Data^k} \exists_{p_i^k} (w_{m_c}^k(Data^k) \equiv (d_i^k)^{p_i^k})). \quad (3)$$

3. For any $(w^x, w^y) \in W$, the associated isomorphic support mapping $\mathbf{M}_{xy}$, satisfies $\mathbf{M}_{xy}(Data^x) = Data^y$.

Thus a transparent word W is conditionally (by m_c) equivalent to an input data word $[(d_i^1)^{p_i^1}, \ldots, (d_i^m)^{p_i^m}]$. Based on the above definition, the vector of conditionally equivalent data support bits that have a common condition m_c, $D_i = \{d_i^1, \ldots, d_i^m\}$ is called an *input word*.

Given a transparent word, $W = \{w_k\}$, with the corresponding support partitions $\{(Control, Data^k)\}$, the entire support set of W can be partitioned into **Control** and $\mathbf{Data}^W = \bigcup^i D_i$. The definition of transparent words can be restated as follows:

Definition 4: A transparent word W is a set of npn isomorphism functions supported by control **Control** and data $\mathbf{Data}^W = \bigcup^i D_i$, such that the following formula is *True*:

$$\forall_{D_i \in \mathbf{Data}^W} \exists_{m_c} \exists_{P_i} (W_{m_c}(\mathbf{Data}^W) \equiv (D_i)^{P_i}), \quad (4)$$

where P_i set of polarity bits for D_i.

Although, for an input word D_i, there could be multiple minterms of *Control* satisfying Formula (4), the assignments of $m_c \in Control$ for different D_is are disjoint.

Example: Consider Equation(1): for each $C[j]$, the support set $\{s, A[j], B[j]\}$ can be partitioned into $Data^j = \{A[j], B[j]\}$ and $Control = \{s\}$, such that $(s = 1) \Rightarrow (C[j] = A[j])$ and $(s = 0) \Rightarrow (C[j] = B[j])$. Hence a

common (control) assignment applied to all bits of the transparent word, makes them transparent from the corresponding supports simultaneously. The supports of C can be partitioned into $\mathbf{Data}^C \equiv \{A[m-1:0], B[m-1,0]\}$, and $\mathbf{Control} \equiv \{s\}$.

Since negations of some bits of transparent words might occur during synthesis, it seems reasonable to consider the logic still as "transparent". Note that in the example: $C[j] = sA[j] + s'B[j]$ the negation of bit $C[j]$ can be done by negating the data inputs, $A[j]$ and $B[j]$:

$$\begin{aligned} inv(C[j]) &= inv(sA[j] + s'B[j]) \\ &= s\ inv(A[j]) + s' inv(B[j]). \end{aligned} \tag{5}$$

but C (with some phase changes) can still be considered transparent from A and B because the assignments to the control bits are unchanged.

In Section 3.2 and 3.3, as we compose transparent sections to form a larger transparent path, we will need to resolve cases where only some bits of a transparent word are negated. However, for composing transparencies to find larger ones, it is required that the polarities of the inputs and outputs are consistent. This can be done by negating some of the inputs of the path (using npn isomorphism) to get a compatible polarity at the output that feeds into another transparent word.

Theorem 2: Given a transparent word W, the negation of any output bit w^k can be done by negating the corresponding input data support bits, without changing any control assignment.

The upshot is that when finding another transparent section of logic and composing it to extend a transparent path, this can *always* be done simply by negating the inputs to get compatible polarities at the point of composition.

3.2 Composition of Transparency

Similar to the composition of npn isomorphism, larger transparent functions are frequently created by composing smaller transparent blocks.

Example: In Figure 1, word C is transparent from A and B under the control of s_1, while a second transparent block consists of word E, transparent from C and D under the control of s_2. Thus $(s_1 = 1, s_2 = 1) \rightarrow E \equiv A$, while $(s_1 = 0, s_2 = 1) \rightarrow E \equiv B$ i.e. transparency of E from A and B is obtained by composing of smaller transparent blocks. If some bits of C are negated before feeding into the MUXes controlled by s_2, the composition can be done by pushing the negation to the corresponding bits of A and B to maintain the polar consistency.

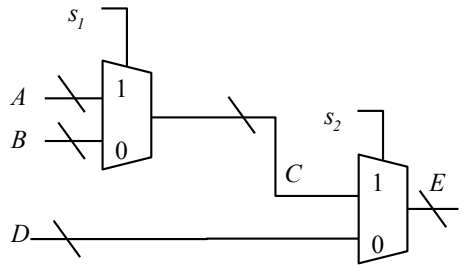

Figure 1: A transparent word can be implemented by composing smaller transparent words.

Definition 5: Let $\mathbf{W} = \{W^k(X) | k = [1, n]\}$ be a set of n m-bit transparent words, and let $Y = \{y^j | j = [1, m]\}$ be another transparent word with support $Data^Y = \mathbf{W} \bigcup V$ and $Control^Y$. Suppose each input word of Y is exactly one transparent word in $\mathbf{W}$ or one word in V. The set of compositions,

$$Z = \{z^j\} = \{y^j(\mathbf{W}(X), V, Control^Y)\} \tag{6}$$

form a *compound word*, and are denoted as $Z = Y \circ \mathbf{W}$.

Theorem 3: Assume Y is a transparent word and $\mathbf{W}$ is a set of transparent words. Let $\{\alpha_i^k\}$ be the set of minterms of $Control^k$, which enable W^k to be transparent from an input word $x_i^k \in Data^k$, and $\{\beta^k\}$ be the set of minterms of $Control^Y$ for $(Y \equiv W^k)$. Using the notation:

$$Control^Z = Control^Y \bigcup [\cup^k Control^k],$$
$$Data^Z = V \bigcup [\cup^k Data^k],$$

a compound word, $Z \equiv Y \circ \mathbf{W}$ is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha_i^k}\} \cap \{\hat{\beta^k}\} \neq \emptyset) \tag{7}$$

is *True*, where $\{\hat{\alpha_i^k}\}$ and $\{\hat{\beta^k}\}$ are $\{\alpha_i^k\}$ and $\{\beta^k\}$ extended to cubes of the larger space of $Control^Z$, respectively.

Proof

1. Based on Theorems 1 and 2, Z can be an npn isomorphism class by flipping the polarities of W^kwhenever its output polarity is not consistent with the input polarities of y^k.

2. Because Y is a transparent word, for each input word in V, there must exist an assignment of $Control^Y$ to enable the transparency from V.

3. Conditions satisfying Formula 7 imply that for each input word x_i^k of W^k, there exists an assignment of $Control^Z$ such that a) W^k is transparent from x_i^k, b) Y is transparent from W^k, and c) Y is transparent from x_i^k. Therefore, $Z \equiv Y \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

3.3 Propagation of Transparency

Example: Figure 2 illustrates how a longer transparency can be composed from non transparent sections of logic. C is transparent from A when $s_1 = 1$, and D is transparent from B when $s_2 = 1$, but the logic block from C and D to E is not transparent (there is no common control support for each bit of E). However, E is transparent from A when $(s_1 = 1, s_2 = 0)$, while $(s_1 = 0, s_2 = 1)$ makes E transparent from B.

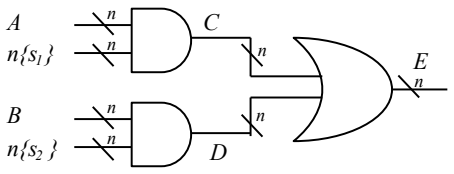

Figure 2: A longer transparent word may be composed of smaller transparent words and an npn isomorphism class.

When a transparent function block is composed of non-transparent sections, it is called *propagation of transparency.*

The conditions when this can happen are stated in the following.

Definition 6: (*prodeeding word*) Let $\mathbf{W}$ be a set of n m-bit transparent words, and let $Y(\mathbf{W}) = \{y^j(\mathbf{W})\}$ be an npn isomorphism class. Suppose each y^j is supported by exactly one bit of each W^k, and the isomorphically mapped supports of y^j are always from the same word of $\mathbf{W}$. The compositions, $Z = \{z^j\} = \{y^j(\mathbf{W}(X))\}$ are said to form a *proceeding word*.

Theorem 4: Assume Y and $\mathbf{W}$ are as in Definition 6 and the supports of W^k are $supp(W^k) = (Control^k, Data^k)$. Let $\{\alpha_i^k\}$ be the set of minterms of $Control^k$ which cause $(W^k \equiv x_i^k)$, and $\{\beta^k\}$ be minterms of $\cup^k Control^k$ which cause $(Y \equiv W^k)$. Using $Control^Z = \cup^k Control^k$, and $Data^Z = \cup^k Data^k$, a proceeding word, $Z \equiv Y \circ \mathbf{W}$, is a transparent word controlled by $Control^Z$ if

$$\forall_k \forall_i (\{\hat{\alpha}_i^k\} \cap \{\beta^k\} \neq \emptyset), \tag{8}$$

where $\{\hat{\alpha}_i^k\}$ refers to $\{\alpha_i^k\}$ extended to cubes of $Control^Z$.

Proof

1. Similar to the proof of Theorem 3, Z can be an npn isomorphism class by flipping the polarities of W^k if needed.
2. For each input word x_i^k in $Data^Z$, Formula 8 implies that there exists an assignment of $Control^Z$, such that $W^k \equiv x_i^k$, $Y \equiv W^k$, and thus, $Y \equiv x_i^k$, implying $Z \equiv Y \circ \mathbf{W}$ is a transparent word with $(Control^Z, Data^Z)$ as control and data supports.

Example: In Figure 2, $\mathbf{W} = (C, D)$ and $\{\hat{\alpha}_1^k\} = s_1 s_2 + s_1 \overline{s}_2$ makes C transparent from A, while $\{\hat{\alpha}_2^k\} = s_1 s_2 + \overline{s}_1 s_2$ makes D transparent from B. $\{\beta^k\} = s_1 \overline{s}_2$ $(\overline{s}_1 s_2)$ causes $E \equiv C$ $(E \equiv D)$. Note that $\{\hat{\alpha}_1^k\} \cap \{\beta^k\} = s_1 \overline{s}_2 \neq \emptyset$ and $\{\hat{\alpha}_2^k\} \cap \{\beta^k\} = \overline{s}_1 s_2 \neq \emptyset$. Thus the conditions for propagation of transparency are met, and therefore E is transparent from A and B.

4. TRANSPARENCY IDENTIFICATION

The functional approach proposed for transparency identification relies only on dependencies among signals. It can be used to complement a structural approach, leading to a method that is more efficient with more reliable results.

In general, we want to identify transparent logic anywhere it occurs in the circuit - from inputs to internal words (forward), from internal words to outputs (backward), and between internal words. Two problems are formulated and solved in this paper: forward and backward transparency.

Forward Transparency: Given a boundary of input supports, e.g. primary inputs, find (1) input words, (2) transparent words in the fanout cones, (3) the corresponding partition of supports for each proved word and (4) partial assignments of **Control** for enabling the transparencies.

Backward Transparency: Given an output boundary of transparent words (candidates), e.g. primary outputs, find (1) transparent words on the boundary, (2) the boundary of supports in the fanin cones, (3) the corresponding partition of the supports and (4) partial assignments of **Control** for enabling the transparencies.

In both cases, we want to find the longest path for each transparency We provide details and discuss the common sub-problems for both forward and backward transparency algorithms and the challenges of this approach.

4.1 Proving Transparency of Sub-circuits

Both forward and backward transparency problems have common sub-problems: given a) the boundaries of supports b) targets (word candidates), and c) a set of proved words, find (1) transparent words on the target boundary, (2) support partitions and (3) partial assignments of **Control** for input words on the support boundary. Figure 3 outlines the steps for solving this sub-problem.

```
Algorithm: Find Transparent Words
Input:
  Circuit //one combinational gate-level circuit.
  Boundary = (Supports, Targets)
//two sets of signals in the circuit.
  ProvedWords // a set of proved words
Output:
  NewWords
  //a set of newly proved input and output words

01. NewWords = ∅
02. (NpnIsoClasses, SuppMaps) = NpnIsoClasses(Circuit,
...     Boundary)
03. For each c in NpnIsoClasses
04.   Words = splitClass( c, SuppMaps, ProvedWords)
05.   For each w in Words
06.     (Control, Data) = classifySupports(w, SuppMaps,
...       ProvedWords )
07.     If getAssignments(w, Control, Data, ProvedWords)
08.       addWords(NewWords, w, Data)
09. Return NewWords
```

Figure 3: Algorithm for proving transparency of a sub-circuit.

Given a sub-circuit specified by **Circuit** and **Boundary**, the function *NpnIsoClasses(...)* at Line 2 returns the npn isomorphism classes among signals in **Targets** with their isomorphic support mappings. Each npn isomorphism class can contain more than one word, so *splitClass(...)* works on those isomorphic signals, analyzes their supports and splits those signals into different candidate words. It decomposes each npn isomorphism class into sub-classes, such that all signals are driven by the same controls and each contain a bit from each of a common group of words.

Then *classifySupports(...)* at line 6 partitions supports into *Data* and *Control*, and groups the bits of each identified input word together (details are given in Section 4.2).

For each candidate word, the function *getAssignments(...)* at Line 7 formulates QBF problems as Equation (5) and apply a QBF solver to those problems to find assignments for control supports. Once the QBF solver proves the candidate word is indeed a transparent word, this word is kept in **NewWords** along with the corresponding assignments and input words.

For *compound words*, there exists a complete transparent block inside the sub-circuit, and all essential supports can be found on the boundary. Hence the QBF problems can be formulated only with the signals on the support boundary. In contrast, for *proceeding words*, the circuit defined by **Boundary** might only contain npn isomorphism classes, but no whole transparent words. Hence the QBF problems should consider the supports of input words on the boundary.

To illustrate how this is done for a *proceeding word*, consider Figure 2. The non-isomorphism class (the logic block

from C and D to E) inside the sub-circuit is not a transparent block. For this case, the function *getAssignments(...)* uses the control supports of the proved words (s_1 and s_2 for C and D, respectively) to find the feasible assignments for the transparent condition.

4.2 Support Classification

Classifying supports into control and data types is critical for proving transparent words. According to Definition 2, all bits of a transparent word must each have support bits from the same set of input words, while mapped control supports must be identical, i.e. the same set of control bits appear in every support of the transparent word bits and they are mapped isomorphically to themselves.

Finding an ideal mapping cannot fully rely on the mappings found for npn functional isomorphism, because a legal mapping for two signals in the same npn isomorphism class could be illegal for transparent logic (see Figure 4).

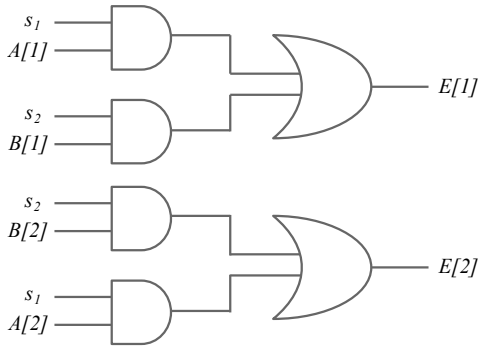

Figure 4: An example with inappropriate support mapping between signals in the same isormorphism class.

The circuit in Figure 4 is extracted from Figure 2, but with some input permutations. Given the support boundary, (A[1], A[2], B[1], B[2], s_1, s_2) and the targets, (E[1], E[2]), E[1] and E[2] are classified into the same npn isomorphism class. Due to symmetry of some Boolean functions, the support mapping is not unique.

For example, the support mapping, $(s_1, A[1], s_2, B[1])_{E[1]} \rightarrow (s_2, B[2], s_1, A[2])_{E[2]}$ satisfies the definition of isomorphism, i.e. when the same values are applied to the mapped inputs (i.e. s_1 and s_2), E[1] and E[2] should evaluate to the same value. However, this requirement blocks out any legal assignments of (s_1, s_2) for E being transparent from A and B; it conflicts with the control condition for transparency: $(s_1 = 1, s_2 = 0) \Rightarrow (E \equiv A)$ and $(s_1 = 0, s_2 = 1) \Rightarrow (E \equiv B)$.

For this case, the mapping of control supports can be revised easily, because the definition of transparent words requires that each control support is isomorphically mapped to itself. Moreover, if A and B have been proved already as words, the mapping issue can be resolved by forcing bits of the same word to be mapped to the same word.

Unfortunately, if input words have not been proved yet, it is necessary to enumerate all legal isomorphism mappings in order to determine correct mappings. This issue can be moderated by decomposing an input circuit into several sub-circuits properly. First, input words can be proved before being considered as supports of others. Also, as the sizes of the input circuits decrease, the number of feasible support mappings for npn isomorphism is reduced.

In the experiments for the results shown in Section 5, support classification only finds isomorphism mappings and uses the above heuristics, but does not enumerate all feasible mappings. Thus some transparent words could have been missed in the experiments.

4.3 Forward Transparency Algorithm

With the assistance of the algorithm outlined in Figure 3 , the algorithm for forward transparency is shown in Figure 5.

Algorithm: **Forward Transparency**
Input:
  Circuit
Output:
  ProvedWords

01. **ProvedWords**= $\emptyset$
02. **InternalControls** = $findHighFanoutSignals$(**Circuit**)
03. $decideForwardOrder$(**InternalControls**)
04. **For** each *control* in **InternalControls**
05. **Fanouts** = $collectImmediateFanouts$(*control*)
06. **Supports** = $collectSupports$(**Fanouts**, **ProvedWords**)
07. **Targets** = $collectFanouts$(*control*, **Supports**)
08. **NewWords** = $findTransparentWords$(**Circuit**,
... **Supports**, **Targets**, **ProvedWords**)
09. **ProvedWords** $\bigcup$ = **NewWords**
10. **ProceedingWords** = $propagateWords$(**ProvedWords**)
11. **ProvedWords** $\bigcup$ = **ProceedingWords**
12. **Return ProvedWords**

Figure 5: Algorithm for identifying transparent words in a gate-level circuit - forward transparency case

The function *findHighFanoutSignals(...)* in Line 2 uses the fact that all bits of a transparent word should be controlled by the same condition, so it collects all signals with more than 3 immediate fanouts as candidate controls. Also, *decideForwardOrder(...)*, sorts control signals in ascending topological order according to its greatest topologically immediate fanout.

Lines 5 to 9, are repeated for all control signals. In Line 5, the function *collectImmediateFanouts(...)* collects all immediate fanouts of the current *control*, while *collectSupports(...)* decides the boundary of supports for identifying npn isomorphism classes. In practice, this function backtracks from all signals in **Fanouts** and collects the nearest bits in **ProvedWords** or nearest primary inputs. The support boundary is a subset of the union of proved words and primary inputs. Then the function *collectFanouts(...)* collects all signals only driven by signals in **Supports** and saves them as the target boundary.

Based on the support and target boundaries, the function *findTransparentWords(...)* applies the algorithm of Figure 3 to find transparent words in the specified subcircuits. Those new words are added to **ProvedWords** in Line 9 and utilized in later procedures.

Finally, *propagateWords(...)* addresses the possible existence of proceeding words (Section 3.3) to enlarge transparent logic blocks. This searches for npn isomorphism classes in the fanout cones of the deepest transparent words and reuses the proved assignments based on Theorem 4. Those new words will be added to **ProvedWords**. This procedure is repeated until no new words are found.

Example: Consider Figure 1 as input **Circuit**. In the beginning, s_1 and s_2 are recognized as high-fanout signals.

The greatest immediate fanout of s_2 is greater than that of s_1, hence signals in **InternalControl** are ordered as $s_1 \rightarrow s_2$. Based on controls in **InternalControl**, all bits of C are considered as isomorphism targets first, and C is proved as a transparent word from A and B, under the control of s_1. Here A, B, and C are added to **ProvedWords**. Then all immediate fanouts of s_2, bits of E, are identified as an isomorphism class, with the support boundary (s_2, C, D). The function *getAssignments(...)* finds assignments for (s_1, s_2), such that E is conditionally equivalent to C and D, and hence to A and B. The proved words, A, B, C, D, E, are returned as **ProvedWords**.

4.4 Backward Transparency Algorithm

The main differences between the forward and backward algorithms are (1) how to define support boundaries, (2) the challenge of classifying supports for sub-circuits and 3) all proved bits in the forward algorithm are transparent from certain primary inputs, but in the backward algorithm, only parts of the proved bits are transparent to the primary outputs. The proposed algorithm for backward transparency is shown in Figure 6.

```
Algorithm: Backward Transparency
Input:
  Circuit
Output:
  ProvedWords, TransBits

01. ProvedWords= ∅
02. InternalControls = findHighFanoutSignals(Circuit)
03. decideBackwardOrder(InternalControls)
04. For each control in InternalControls
05.   Fanouts = collectImmediateFanouts(control)
06.   Supports = collectFanins(Fanouts)
07.   Targets = collectFanouts(Supports)
08.   NewWords = findTransparentWords(Circuit,
...       Supports, Targets, ProvedWords)
09.   ProvedWords ⋃= NewWords
10. ProceedingWords = propagateWords(ProvedWords)
11. ProvedWords ⋃= ProceedingWords
12. TransBits = finalizeBackward(ProvedWords)
13. Return (ProvedWords, TransBits)
```

Figure 6: Algorithm for identifying transparent words in a gate-level circuit.

As in the forward algorithm, all high-fanout signals are collected and sorted by their greatest immediate fanouts, but sorted in descending topological order.

The immediate fanouts of each control signal are saved temporarily as **Fanouts**, while their immediate fanins form the support boundary. This is different than in *collectSupports(...)* in Figure 5, which traverses circuits until reaching primary inputs or proved words. At Line 7, all signals driven only by signals in **Support** are regarded as the target boundary. When *collectFanouts(...)* traverses forward from signals in **Support**, it might reach some signals which have been proved as input words for other transparent words. Those signals are included in **Targets**, while the traversal stops moving forward from them.

Lines 4 to 9 prove a set of disconnected transparent words, because they are proved in reverse topological order. Since there are no proved words in the support boundaries, when *findTransparentWords(...)* works on each sub-circuit, the functions *splitClass(...)* and *classifySupports(...)* in Figure 3 only can use data dependencies and control signals inside the sub-circuit. In this case, the issues mentioned in Section 4.2 would arise.

Like the forward algorithm, *propagateWords(...)* enlarges the proved transparent blocks. Finally, the function *finalizeBackward(...)* at Line 12 composes the whole set of proved words into larger transparent logic blocks and saves those bits which are transparent to primary outputs as **TransBits**.

Example: Consider Figure 1 using the backward algorithm. Signals in **InternalControls** are ordered as $s_2 \rightarrow s_1$. Under control s_2, E is proved first as a transparent word from C and D. Then C is proved to be conditionally equivalent to A and B under control s_1. The function *finalizeBackward(...)* verifies the composition of assignments for (s_1, s_2), and hence the two proved words are concatenated into a larger transparent function block.

Example: When the backward algorithm is applied to the circuit in Figure 2, it can find $(s_1 = 1) \rightarrow (C \equiv A)$ and $(s_2 = 1) \rightarrow (D \equiv B)$ first. Then *propagateWords(...)* will do the same thing as in the forward algorithm. In the end, the function *finalizeBackward(...)* finds all proved words can be transparent to primary outputs.

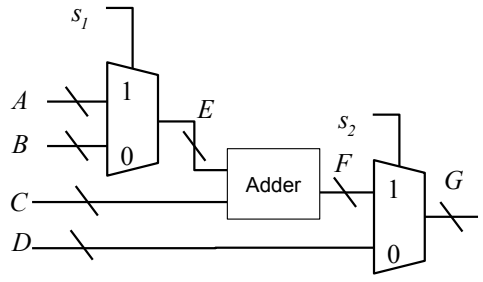

Figure 7: An example with disjointed transparent logic blocks.

Before *finalizeBackward(...)*, the proved words can be disjoint, and are not guaranteed to be reachable from the primary outputs. In the circuit in Figure 7, the backward algorithm finds two disconnected transparent words: $G = s_2?F : D$ and $E = s_1?A : B$, which are separated by an adder. *propagateWords(...)* will confirm that there is no way to connect the two words with transparent paths. Hence *finalizeBackward(...)* only returns one transparent word $G = s_2?F : D$, which is reachable from primary outputs.

5. EXPERIMENTAL RESULTS

The proposed algorithms were implemented in ABC [3]. All experiments were performed on a 16-core 2.60GHz Intel(R) Xeon(R) CPU with no time limit. All cases were processed as AIGs and analyzed for forward and backward transparency. Sequential circuits were converted into combinational designs by replacing flip-flops inputs and outputs with primary outputs and inputs respectively.

As a reference for the functional approach, we implemented a pure structural approach: 1) structural matching is used to locate all 2-to-1 MUXes in the AIGs, 2) signals with the same control are grouped into one word, and these connected words are collected into larger transparent blocks, and 3) words which are reachable from primary inputs (outputs) for forward (backward) transparency are returned.

We wanted to compare the efficiency and effectiveness of the structural algorithms versus our functional algorithms applied to highly-transparent cases. To select these, the functional forward algorithm was applied to all 230 cases of the single-output track in the Hardware Model Checking Competition 2014 [1]. For each case, some POs were proved conditionally equivalent to certain primary inputs. We computed the proportion of those POs to all POs and ran experiments on the top 10 cases with the highest percentages of transparent POs. Among the 230 cases, there are 20 cases with more than 50% transparent POs, while another 38 cases have more than 25% transparent POs. Table 1 shows the statistics of the selected cases after they were converted to combinational circuits. The last column of Table 1 indicates the percentages of transparent POs over all POs. The *6sxxx* cases are industrial problems from IBM and the *beem* examples come from different applications areas like protocols, planning, scheduling, communication, or puzzles.

Table 1: Statistics of the selected benchmarks from HWMCC'14 [1].

Case Name	PI #	PO #	AND #	Trans. PO %
6s195.aig	1344	1258	8046	87.1
beemfrogs1b1.aig	323	159	8493	86.0
6s171.aig	1357	1263	8074	84.6
beemloyd3b1.aig	237	118	3970	82.1
6s282b01.aig	1977	1934	10264	81.2
6s384rb024.aig	22367	14953	47933	79.0
6s206rb103.aig	37847	28644	103375	71.4
6s302rb09.aig	36962	27777	100571	70.3
6s348b53.aig	15797	15561	89567	70.1
beemldelec4b1.aig	2559	1215	34252	67.5

5.1 Experiments for Structural Approach

Table 2 shows the results for the structural approach coded for the experiments. Column 2 indicates the total number of signals, i.e. AIG nodes or primary inputs, that were classified as belonging to words. Column 3 lists the total number of structural MUXes recognized. Columns 4-7 (labeled *Forward Transparency*) give the statistics of the transparencies found using the forward algorithm. Column 4 shows the total number of transparent words reachable from primary inputs (including input words); Column 5 lists the number of AIGs plus inputs covered by all the transparent logic blocks found; Column 6 gives the (minimum, maximum) widths (the number of MUXes grouped together as a word) of found words, and Column 7 shows the (minimum, maximum) depths of transparent words on boundaries. The depth of each word is the total number of AIG nodes between itself and the primary inputs, where one MUX is counted as depth 2.

Columns 8-11 (labeled *Backward Transparency*) show similar statistics for the backward case: Column 8 lists the total number of transparent words reachable from the primary outputs. Depth here is the number of AIGs between internal transparent words on boundaries and the primary outputs. The last column shows the combined run-time for the forward and backward structural approaches. Here we only identify 2-to-1 MUXes and MUXes with negation on outputs or inputs. We omit counting words with less than 4 bits

Discussion of Structural Results

Table 2 shows that most benchmarks contain wide transparent words. Some contain very deep transparent paths as well as some with only 1 or 2 levels of MUXes. The runtimes show that this approach is very efficient as expected.

Although these cases have high percentages of transparent POs, for some cases the structural approach cannot find any transparent words. Many MUXes are recognized but there are several reasons why the structural approach misses many transparent words:

1. structural matching only considers standard 2-to-1 multiplexers, while there are other types of transparent functions.

2. Many of the identified MUXes are controlled by different selection signals, and thus lead to words of less than 4 bits, which are excluded in the analysis.

3. Transparent words are required to be reachable from primary inputs through fully transparent paths. If a transparent word originates from the output word of an arithmetic operator (e.g. word G and F in Figure 7), it would not be reported, yet many MUXes would be involved in such a transparency.

Although quite fast, this approach itself is not enough for finding many of the whole transparent blocks that exist in these benchmarks as shown in the following section which shown the total words found by the functional approaches.

5.2 Experiments for Functional Approach

In the experiments for the functional approach, the function *NpnIsoClasses(...)* is created as an ABC command, *&iso* [8]. Table 3 shows the experimental results for both the forward and backward cases. The columns are similar Table 2.

Comparing Functional and Structural

We observe for the forward case:

1. The functional approach finds many more and wider transparent words in all cases. For example, in the last case, beemldelec4b1.aig, the functional approach finds many transparent words, while the structural approach finds none. One reason is the functional approach addresses all isomorphism classes and tries to prove transparency for them, while the structural method only considers 2-to-1 MUX cases.

2. The functional approach finds much more logic involved in transparent paths than the structural approach, on average about 2x more signals.

3. The functional approach finds deeper words than the structural method. As mentioned, the current structural approach cannot find any depth-1 transparent logic (a MUX has depth 2).

4. The runtime of the functional approach increases with circuit size, while the structural approach is much faster. The functional approach requires many circuit traversals and manipulations. In contrast, the structural approach only goes through an entire circuit once to collect MUXes, and then works on groups of MUXes as candidate words.

For backward transparency, we observe the following:

1. For the three cases where the structural approach cannot find any words, the functional approach finds several depth-1 transparent words.

Table 2: Experimental results of the structural approach on ten selected cases from HWMCC'14 [1]. N/A here means no transparent word found.

Case Name	Total Sig. #	MUX #	Forward Transparency				Backward Transparency				Runtime(s)
			Words#	Sig.#	Widths	Depths	Words #	Sig. #	Widths	Depths	
6s195.aig	9390	2357	18	4552	8, 512	2, 12	287	5005	4, 72	2, 6	0.056
beemfrogs1b1.aig	8816	2016	33	520	8, 8	32, 32	0	0	N/A	N/A	0.056
6s171.aig	9431	2362	11	4413	16, 512	2, 12	289	5034	4, 73	2, 6	0.058
beemloyd3b1.aig	4207	985	23	352	8, 8	22, 22	0	0	N/A	N/A	0.049
6s282b01.aig	12241	2472	65	1803	6, 66	2, 6	31	3643	6, 1031	2, 4	0.056
6s384rb024.aig	70300	14492	889	21278	4, 64	2, 4	992	45174	4, 4250	2, 8	0.122
6s206rb103.aig	141222	28684	2083	56295	4, 193	2, 4	2456	96678	4, 1398	2, 8	0.238
6s302rb09.aig	137533	27818	2054	55274	4, 191	2, 4	2421	94243	4, 1061	2, 8	0.235
6s348b53.aig	105364	28775	484	28850	4, 262	2, 12	304	58875	4, 734	2, 16	0.458
beemldelec4b1.aig	36811	8458	0	0	N/A	N/A	0	0	N/A	N/A	0.591

Table 3: Experimental results of the functional approach on ten selected cases from HWMCC'14 [1]

Case Name	Forward Transparency					Backward Transparency					
	Words#	Sig.#	Widths	Depths	Runtime(s)	Words#	Trans. W #	Trans. S #	Widths	Depths	Runtime(s)
6s195.aig	1003	7921	4, 512	3, 14	3.106	352	242	5025	4, 72	2, 6	4.130
beemfrogs1b1.aig	622	4153	6, 8	9, 40	3.753	464	19	151	7, 8	1, 1	2.689
6s171.aig	490	8036	4, 512	3, 14	3.471	407	190	4641	4, 67	1, 7	3.100
beemloyd3b1.aig	89	712	8, 8	2, 26	1.815	231	12	96	8, 8	1, 1	1.390
6s282b01.aig	268	6996	4, 966	2, 10	2.623	205	20	2744	4, 999	3, 4	2.989
6s384rb024.aig	2587	48311	4, 2308	3, 7	66.533	2295	1336	40036	4, 3338	1, 9	99.395
6s206rb103.aig	6552	105493	4, 1042	3, 12	297.935	6076	3472	79660	4, 305	1, 8	471.489
6s302rb09.aig	6583	103437	4, 872	3, 12	295.97	6071	3493	78696	4, 296	1, 8	451.814
6s348b53.aig	1879	58874	4, 367	3, 18	72.454	2644	715	57521	4, 598	1, 11	238.205
beemldelec4b1.aig	574	2688	4, 8	2, 3	27.368	2183	43	223	4, 21	1, 1	17.976

2. Although the functional backward algorithm spends a lot of time proving internal transparent words, many are unreachable from the primary outputs.

3. Unlike the forward case, the functional backward approach seems to miss some words. The reason is that the present implementation of *NpnIsoClasses(...)* (*&iso* in ABC) only reports isomorphism classes in which the polarities of one word must be all the same. The backward case is more likely to have mixed polarity npn isomorphism classes, because some bits of input words for transparent logic blocks might have been synthesized with their supports, which are parts of operators. Therefore, some bits are excluded from our current implementation and this can have a more significant effect on the backward case.

Comparing the forward and backward functional approaches, we observe:

1. Due to the difficulties of support classification (mentioned in Section 4.2), the backward algorithm misses some transparent paths proved by the forward algorithm. We need to understand this better and improve the backard strategies.

2. For large cases, the backward algorithm takes much more time than the forward one because:

   - For some backward cases, there is no proved word on the support boundary that would be useful for classifying supports. Then it takes more time to prove transparent words. In contrast, the forward algorithm collects and proves candidate words in a topological order and the proved words can be used to prove candidates in their fanout cones.
   - The forward algorithm only checks each signal once, but the backward one needs to revisit some target signals to consider different support boundaries. The reason is, if the support boundary defined earlier cannot be used to prove transparent words (due to the support classification issue), those signals will be re-visited when a different support boundary is proposed.

Table 3 shows that the proposed algorithm finds a significant number of transparent words in the selected cases, even though the isomorphism algorithm used in these experiments is an early version largely limited by the structure of the circuit [8]. This may cause larger words to be broken down into smaller sub-words. A better isomorphism method, using npn isomorphism, has been developed but not yet integrated into the present implementation [9]. Also, run-time performance should improve as the implementation matures, e.g. run-time might be improved by skipping some intermediate levels of words, but then intermediate words might be missed. Also structural and functional methods can be inter-mixed. For real applications, the particular final usage of the found words will dictate a suitable balance between performance and the number of proved words.

6. CONCLUSIONS

This paper presented algorithms for finding transparent logic, which can be used to highlight word-level information in gate-level circuits. A functional approach was proposed to identify transparent logic in combinational circuits. Experimental results demonstrated that the proposed algorithms can be very effective in extracting words as well as some control logic.

Future work will include:

- new npn isomorphism functional methods will be integrated into the current approach,
- experiments will be conducted on examples from unrolled sequential circuits to try deeper circuits,
- a composite method combining both structural and functional approaches will be developed to achieve efficiency and effectiveness at the same time,

- an algorithm will be developed for finding internal transparency blocks, not just forward or backward transparencies, and appropriate examples will be created to measure its effectiveness,
- the method will be used to find all word -> operator and operator -> word boundaries in a gate-level design and integrated with reverse engineering functional approaches that can identify the operators.

The final goal is a fully functional approach to the reverse engineering of gate-level designs.

7. ACKNOWLEDGEMENTS

This work is supported in part by SRC contract 2265.001 and by the NSA via the TRUST grant. We also thank industrial sponsors of BVSRC, Altera, Atrenta, Cadence, Calypto, IBM, Intel, Mentor Graphics, Microsemi, Synopsys, and Verific for their continued support.

8. REFERENCES

[1] *Hardware Model Checking Competition 2014*. http://fmv.jku.at/hwmcc14cav/.

[2] S. Awodey. *Category theory*. Clarendon Press Oxford University Press, Oxford Oxford New York, 2006.

[3] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

[4] Y.-Y. Dai, K.-Y. Khoo, and R. Brayton. Sequential equivalence checking of clock-gated circuits. In *Design Automation Conference*. ACM, 2015.

[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*, (3):72–80, 1999.

[6] W. Li. Formal methods for reverse engineering gate-level netlists. Master's thesis, 2013.

[7] W. Li, G. Adria, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pages 67–74. IEEE, 2013.

[8] A. Mishchenko, N. Een, R. Brayton, M. Case, P. Chauhan, and N. Sharma. A semi-canonical form for sequential aigs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 797–802. EDA Consortium, 2013.

[9] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. K. Brayton, and G. D. Micheli. Heuristic npn classification for large functions using aigs and lexsat.

[10] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton. Reverse engineering with simulation graphs. In *Formal Methods in Computer-Aided Design*, 2015.

[11] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Y. Tan, A. Tiwari, N. Shankar, S. Seshia, and S. Malik. Reverse engineering digital circuits using structural and functional analyses. *Emerging Topics in Computing, IEEE Transactions on*, 2(1):63–80, 2014.