A General Approach for Regularity Extraction in Datapath Circuits

Amit Chowdhary Sudhakar Kale Phani Saripella Naresh Sehgal

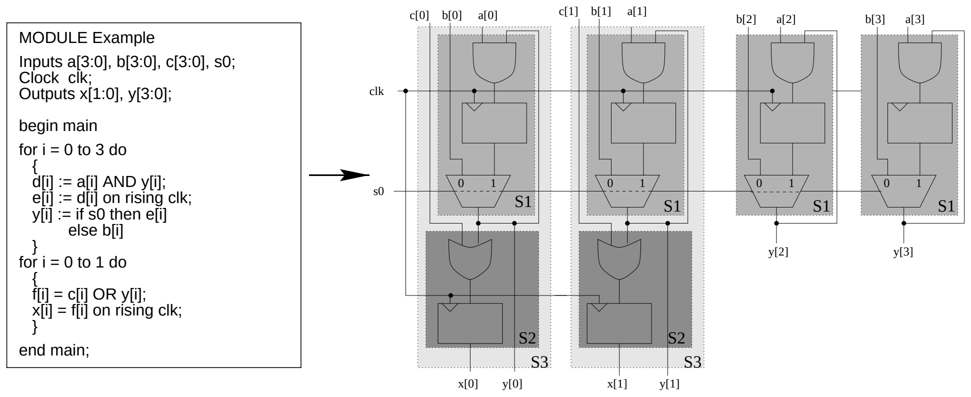
Intel Corporation
Santa Clara, CA 95052

Rajesh Gupta

University of California
Irvine, CA 92697

ABSTRACT

In majority of high-performance custom IC designs, designers take advantage of the high degree of regularity present in circuits to generate efficient layouts in terms area and performance as well as to reduce the design effort. In this paper, we explain how regularity manifests itself at functional, structural and topological levels. Using these notions, we present a general and comprehensive approach to extract functional regularity for datapath circuits from their high-level or gate-level descriptions. The fundamental step is the generation of a large set of templates, where a template is a subcircuit with multiple instances in the circuit. Two novel template generation algorithms are presented — one for templates with a tree structure, and the other for a special class of multi-output templates, called *single-principal-output* (*single-PO*) templates, where all outputs of a template are in the transitive fanin of a particular output. The set of templates generated is shown to be complete under a few simplifying, yet practical, assumptions, which is key in obtaining a desirable cover of the circuit using templates. We show that the generation of such a large set of templates results in excellent covers for various circuits, including several ISCAS benchmark circuits. We also demonstrate that the regularity extracted from these circuits can be used to easily understand their structure. We have successfully used our approach to identify bit slices of very large datapath circuits from general-purpose microprocessors, which would lead to efficient layouts with a significant reduction in the overall design effort.



```
MODULE Example
Inputs a[3:0], b[3:0], c[3:0], s0;
Clock  clk;
Outputs x[1:0], y[3:0];

begin main
for i = 0 to 3 do
   {
   d[i] := a[i] AND y[i];
   e[i] := d[i] on rising clk;
   y[i] := if s0 then e[i]
           else b[i]
   }
for i = 0 to 1 do
   {
   f[i] = c[i] OR y[i];
   x[i] = f[i] on rising clk;
   }
end main;
```

Figure 1: High-level description of a circuit, where the regularity is evident from the corresponding gate-level description. Three templates $S1$, $S2$ and $S3$ are identified.

1 Introduction

Datapath circuits in general-purpose microprocessors as well as application-specific integrated circuits (ASICs) perform a variety of boolean and arithmetic operations on busses with a width of up to 64 bits. Such circuits have a very high degree of regularity. Designers often exploit this regularity in circuits to achieve regular layouts with a small area and a high performance. The datapath regularity defines a natural hierarchy of the circuit, which simplifies the overall design process. As a result, the total design effort is reduced by identifying regularity in circuits, thus improving the productivity of designers. Therefore, a very important task in datapath design is to extract the regularity inherent in the circuits. Existing CAD tools can not extract and utilize regularity to the extent necessary for competitive designs. Therefore, datapath circuits in general-purpose microprocessors are currently designed almost entirely by hand [6]. There is an urgent need for a regularity extraction approach that would speed up the design of efficient regular layouts of datapath circuits.

We assume that the input circuit is described by a hardware description language (HDL), such as Verilog or VHDL. The operators in the HDL descriptions can be either logic gates, such as AND, OR and multiplexors, or arithmetic operators, such as adders, subtracters and shifters. For example, Fig. $1a$ illustrates the high-level description of a small circuit, along with the corresponding gate-level description. Regularity in a circuit implies that there exists subcircuits, called *templates*, which have multiple instances in the circuit. The circuit of Fig. $1a$ has four instances of template $S1$, and two of templates $S2$ and $S3$ each. The task of regularity extraction is to identify a set of templates, and cover the given circuit by a subset of these templates, where the objective is to use large templates which have a large number of instances. The regularity extraction involves a tradeoff, since a large template usually has a few instances, while a small template has a high number of

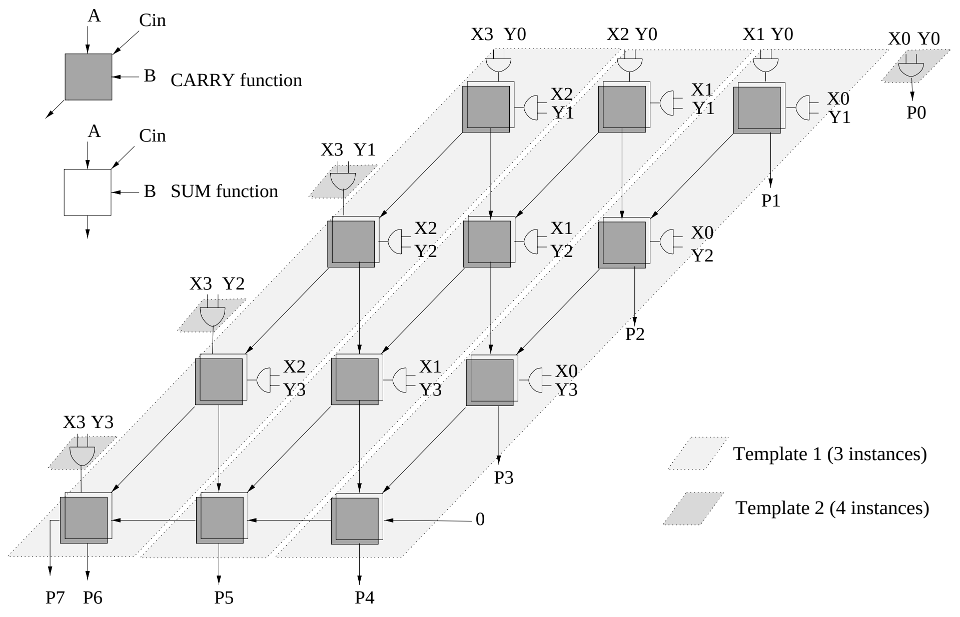


Figure 2: A 4×4 multiplier with inputs $X3, \ldots, X0$, and $Y3, \ldots, Y0$, which is covered by two templates.

instances. Usually, a large template implies a better optimization of area and performance, while a template with more instances requires less design effort, since a template is designed only once for all its instances. For example, the circuit in Fig. $1a$ can be covered by two instances of $S2$ and four of $S1$, or by two instances of $S3$ and two of $S1$. The first cover results in a lower design effort, while the second cover might lead to a more efficient layout, since it can optimize across the boundary between $S1$ and $S2$. Another example is the 4×4 multiplier, whose gate-level representation is shown in Fig. 2. It is composed of SUM, CARRY and AND functions. It can be covered by three instances of template $S1$ which is a diagonal array, and four instances of template 2 which is just an AND gate. If the structure of the multiplier is unknown, then the extraction technique should generate a cover of these two templates. In general, an extraction approach should be able to generate a range of covers to assist the designer in selecting the most desirable cover.

Regularity in a given circuit can be classified as either functional, structural or topological. Given a high-level (behavioral or structural) description, a functionally-regular circuit uses a set of functionally-equivalent operations or subcircuits (templates). Functional regularity is an essential first step towards the generation of a compact and regular layout of the circuit. It can also be used to restructure the HDL code, for instance to improve the quality of high-level synthesis results by identifying opportunities for resource sharing [13]. Structure in an HDL description typically refers to declaratively specified blocks [8] consisting of a netlist where the nets or signals can be classified as control or data. A structurally regular description can be described schematically by assigning a horizontal or vertical direction to the nets. Finally, a topologically regular design consists of an ordered set of blocks which gives a good initial placement for the circuit. Our synthesis approach for datapath circuits identifies functional and structural regularity in HDL descriptions and uses it to build topologically regular circuits. In this paper, we are concerned with identification of functional regularity in high-level descriptions.

Several techniques for extraction of functional regularity have been proposed in the literature [17, 2, 16, 15, 11, 14, 1]. Most of these techniques focus on mapping the circuit by templates, assuming that a template library is provided by the user. Very few techniques address the problem of generating a good set of templates. Given a library of templates, Corazao *et al.* [2, 16] address the problem of mapping a circuit described at a behavioral level using templates from the target library. Their approach addresses several key subproblems, such as finding complete as well as partial matches of a template and selecting a good set of templates to optimize the clock period. Rao and Kurdahi [17] represent the input circuit as well as templates from the given library by strings, and use a string matching algorithm to find all instances of the template in the circuit. A subset of the template set is then heuristically chosen to cover the input circuit. Rao and Kurdahi [17] present a simple heuristic to generate a set of templates. The final cover is very sensitive to these templates. Odawara *et al.* [15] presented a methodology to identify structural regularity in highly-regular datapaths. Their method chooses latches driven by the same control signals as initial templates, and uses them to grow larger templates. Odawara's approach identifies one-dimensional regularity in terms of bit-slices of the datapath. Other approaches by Nijssen *et al.* [14] and Arikati *et al.* [1] extend Odawara's methodology to identify bit slices as well as stages of datapath circuits. These structural methods perform well for highly-regular circuits, but might not work for circuits with a mix of datapath and control logic. A problem similar to regularity extraction is technology mapping, where the input circuit is covered by cells (templates) from a given library. Keutzer [12] proposed an approach of partitioning the circuit into rooted trees, followed by mapping the trees using library cells by dynamic programming. The above techniques address the problem of covering a circuit by templates, where the templates are either provided by the user or generated in an ad-hoc manner. None of these techniques deal with the systematic generation of a set of templates for a given circuit. From our experiments, formulation of a good set of templates is crucial for two reasons: (a) it allows tradeoffs among multiple criteria, such as area, timing or power, and (b) it provides controllability of the datapath synthesis process by the designer to support multi-technology designs, such as those using a combination of static and dynamic logic.

We propose a novel approach to extraction of functional regularity, where the set of all possible templates is generated automatically for the input circuit under a set of simplifying but practical assumptions discussed in Section 3. These assumptions significantly reduce the number of templates addressed to $O(V^2)$, where V is the number of components of the input circuit. Our approach then covers the circuit by selecting a subset of this set of templates based on various criteria which correlate to area, performance and design effort. We demonstrate that a wide range of efficient covers are obtained from this set of $O(V^2)$ templates. The major contributions of this paper are two algorithms that generate a sufficiently large set of templates for a given circuit, one to generate templates with a tree structure, and the other to generate a special class of multi-output templates, where every output of the template lies in the transitive fanin of a particular output. We will present the effectiveness of our approach by identifying regularity in several circuits, including several ISCAS benchmarks, from their high-level descriptions. The high-level structure of these ISCAS circuits have been identified earlier by reverse engineering [9, 19] to help in various CAD applications, such as high-level test generation and hierarchical timing analysis.

Our extraction approach has numerous interesting and useful extensions. We can identify multi-output bit slices of more general structure. We can represent the templates hierarchically, which enhances users' understanding of the circuit and provides user the flexibility to work at the desired level of hierarchy. In the event that a template is specified, our approach can be used to generate its all possible instances, either complete or partial, in the input circuit.

The rest of the paper is organized as follows. Section 2 formulates the regularity extraction problem in terms of two subproblems — template generation and then circuit covering by these templates. The complexity of generating the complete set of templates of a circuit is discussed in Section 3. Section 4 discusses the algorithm for generation of templates with a tree structure. Section 5 extends the algorithm to a special class of multi-output templates. The heuristic technique of covering the circuit by templates is discussed in Section 6. Several interesting extensions of our template generation algorithm are described in Section 7. We extract regularity of various benchmark circuits and present the results in Section 8. Section 9 concludes with some future extensions.

2 Problem Formulation

We first present a graph-theoretical formulation of the problem of regularity extraction. The input to regularity extraction is a circuit C composed of logic components that can be either small logic blocks, such as AND gates, OR gates, multiplexors and latches, or arithmetic blocks, such as adders and shifters. The input circuit C is usually described using an HDL. We represent C by a directed graph $G(V, E)$, where the nodes in V correspond to the logic components or the primary inputs of C, and the edges in E correspond to the interconnection among the components and primary

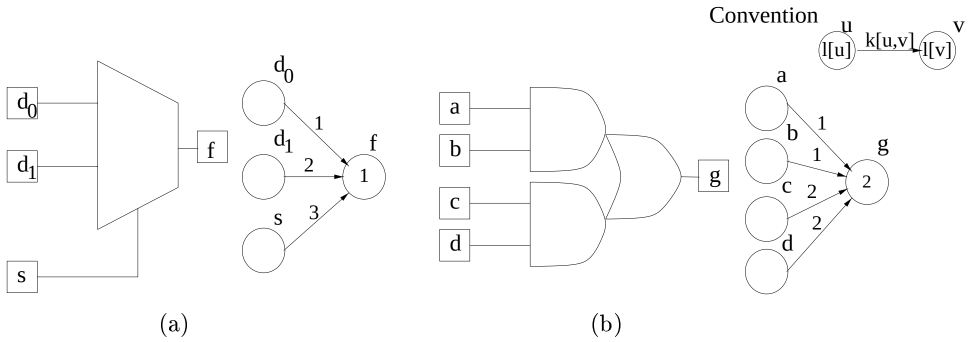


Figure 3: Representing the logic functions of circuit components in G: (a) 2-to-1 mux; (b) AND-OR gate.

inputs of C. The set V can be partitioned into two subsets I and L, which correspond to the sets of primary inputs and logic components, respectively. The set O of primary outputs is a subset of L. We represent the logic functions of components of C in G by a pair of functions. We first define a logic function $l : L \rightarrow \{1, .., l_0\}$, where l_0 is the total number of distinct types of logic functions. If $l[u] = l[v]$, then u and v correspond to the same logic function, *e.g.* a 2-to-1 multiplexor. Similarly, we associate an index $k : E \rightarrow \{1, .., k_0\}$ with every edge in E, where $k(u_1, v) = k(u_2, v)$ implies that the two incoming edges of v are equivalent. Figure 3a shows a multiplexor whose input edges have all distinct indices, while the AND-OR gate of Fig. 3b has four edges assigned to only two indices. The graph G of the multiplier of Fig. 2 is shown in Fig. 4.

A subgraph of G is a graph $G_i(V_i, E_i)$ such that $V_i \subseteq V$ and $E_i \subseteq E$. V_i is partitioned into I_i and L_i. The set O_i of primary outputs is again a subset of L_i. A subgraph of G corresponds to a subcircuit of C. We consider only those subgraphs which satisfy the condition that if $v \in L_i$, then $u \in I_i \cup L_i$ for every node u connected to v by an edge (u, v) in G. We call the subgraphs which satisfy this condition *feasible subgraphs* of G, since they correspond to meaningful subcircuits of C. From here on, a subgraph will imply a feasible subgraph.

We consider two subgraphs G_i and G_j functionally equivalent, if and only if (a) they are *isomorphic*, i.e. there exists a one-to-one mapping ϕ between V_i and V_j [10], (b) the logic functions of corresponding nodes are same, i.e. $l[v] = l[\phi[v]]$, and (c) the indices of corresponding edges are also the same, i.e. $k[u, v] = k[\phi[u], \phi[v]]$. We call the equivalence class of this relation a *template*. Given any set S of subgraphs of G, the subgraphs can be partitioned into m templates, $S_1, \ldots, S_m$, where a template S_i contains $|S_i|$ subgraphs. We estimate the area of a subcircuit that corresponds to the template S_i by $area[S_i] = \sum_{v \in L_i} a[l[v]]$, where $a[j]$ is the area estimate of a node of logic function j.

A *cover* of G is a set $C(G) = \{G_1, \ldots, G_n\}$ of feasible subgraphs of G that satisfies the following

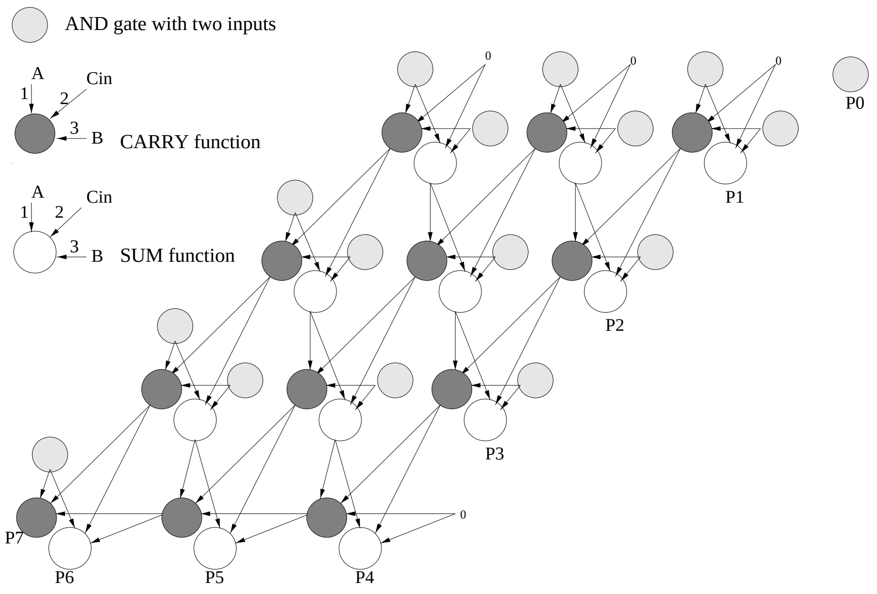


Figure 4: The graph of the 4 × 4 multiplier of Fig 2.

conditions:

1. Every node of G belongs to at least one subgraph in $C(G)$, i.e. $V \subseteq V_1 \cup \ldots \cup V_n$.

2. If a node v is a primary input of a subgraph, then it is either a primary input of G or an output of another subgraph, i.e. for all $v \in I_i$, $v \in I \cup O_1 \cup \ldots \cup O_n$.

The problem of regularity extraction can now be stated as follows.

Regularity Extraction Problem: Given a circuit represented by a graph $G(V, E)$, find a cover $C(G) = \{G_1, \ldots, G_n\}$, which is partitioned into m templates $S_1, \ldots, S_m$, such that the number n of subgraphs and the overall area of the templates, given by $A[C] = \sum_{i=1}^{m} area[S_i]$, are maximized. $\square$

If we assume that every template is synthesized and laid out only once for all its subgraphs, then maximizing the number of subgraphs will reduce the overall design effort. Maximizing $A[C]$ will improve the overall area and delay, since larger templates would lead to better optimization during subsequent technology mapping and physical design stages. The above two objectives are, however, conflicting, since a large template usually has only a few subgraphs, while a smaller template has a large number of subgraphs. For example, the cover of multiplier shown in Fig. 2 has two templates with a total of 7 subgraphs and an area $A[C]$ of 13 units, assuming every component has a unit area. On the other hand, a trivial cover of three templates — an AND gate, SUM and CARRY functions, has a total of 40 subgraphs and an area $A[C]$ of three units.

The problem of finding an optimal cover is NP-complete, even when the subgraphs are selected from a given set. Here, the problem is even more complex, since there is no such set of subgraphs for selecting the cover. We reduce the complexity of regularity extraction by decomposing it into two steps, where a set of templates is first generated, followed by selecting a subset of the template set to cover G. We state these two sub-problems below.

Template Generation Problem: Given a circuit represented by a graph $G(V, E)$, generate the complete set of templates where each template has at least two subgraphs. $\square$

The problem of generating all templates of G is similar to enumerating the equivalence classes of subgraphs of G under isomorphism, which is inherently difficult. As mentioned earlier, prior techniques do not address this problem due to its high complexity. However, we present a few simplifying assumptions in the next section, which will reduce the number of templates addressed to within V^2. (These assumptions will be justified in the context of regularity extraction.) We will later demonstrate that this set of at most V^2 templates will lead to efficient covers for various datapath circuits.

Graph Covering Problem: Given a circuit represented by a graph $G(V, E)$ and its set $S_T(G)$ of templates, find a cover $C(G, S_T) = \{G_1, \ldots, G_n\}$ of G, which is partitioned into $m(\leq p)$ templates $S_1, \ldots, S_m$, such that the number n of subgraphs and the overall area of the templates, given by $\sum_{i=1}^{m} area[S_i]$, are maximized. $\square$

The graph covering problem is the similar to the binate-covering problem [5, 4], which has been well studied in various CAD areas, including regularity extraction [17].

3 Complexity of template generation

We now discuss the complexity of the template generation problem, and present several assumptions to make it tractable. We first show by an example that the number of templates of $G(V, E)$ can be $O(2^V)$. Consider a graph $G_1(V_1, E_1)$ which is a binary tree with n nodes such that every internal (non-leaf) node in L_1 has the same logic function of $l[v] = 1$, and every node v at the leaf level has a distinct function $l[v] = 2, \ldots, \frac{n+1}{2}$; see Fig. 5. We can easily identify $2^{\frac{n+1}{2}}$ subgraphs of G_1, where each subgraph contains all internal nodes of L_1, but only a subset of $\frac{n+1}{2}$ leaf nodes. Thus, the number of subgraphs of G_1 is $O(2^V)$. Now consider a graph G_2 which consists of two unconnected copies of G_1. The number of templates of G_2 is then $O(2^V)$, where each template has at least two subgraphs. We now make the following assumption to reduce the number of templates addressed by the template generation problem. We represent the set of templates of G addressed here by S_T, and the corresponding set of subgraphs by S.

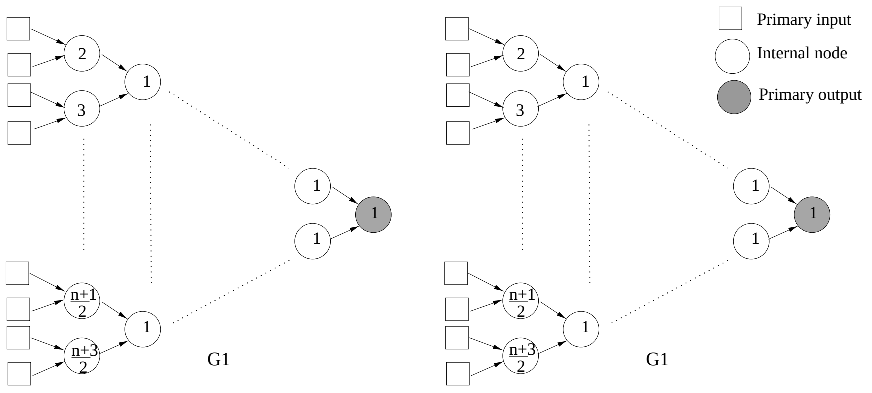


Figure 5: A graph G_2 with two unconnected equivalent trees G_1 has $O(2^V)$ templates, where V is the number of nodes.

Assumption 1. Restrict the set S of subgraphs of G to include only those subgraphs of G which are not a subgraph of any other subgraph in S and which have at least one distinct equivalent subgraph in S. □

The basis for the above assumption is that we would like to extract the maximum degree of regularity. Using Assumption 1, the set S of G_2 contains only two subgraphs, both of which are equivalent to G_1. Thus, G_2 has only one template. We can then generate the templates for G_1, thus resulting in a hierarchy of templates.

The number of templates can still be $O(2^V)$ even after considering Assumption 1. Consider the graph G' of Fig. 6a composed of two unconnected trees, where the incoming edges of every node have the same index k. It has two templates shown in Fig. 6b. Now, consider the graph G of Fig. 6c which is composed of two unconnected binary trees such that all the internal nodes have the same function $l[v] = 1$, while the leaf level is composed of one of the two subgraphs, G_1 or G_2. The number of templates of G is $O(2^V)$, since every pair of subgraphs G_1 and G_2 can be matched using either of the templates of Fig. 6b.

We make the following assumption that does not allow permuting the incoming edges of a node even though the two edges (u_1, v) and (u_2, v) have the same index $k[u_1, v] = k[u_2, v]$. For example, the two input edges of a node corresponding to an OR gate would be assigned different indices, even though they are equivalent.

Assumption 2. For every node v of G with incoming edges from nodes $u_1, \ldots, u_f$, every edge is assigned a unique index of $k[u_i, v] = i$, for all $1 \leq i \leq f$. □

The above assumption will rule out S_2 (Fig. 6b) as a template for the graph of Fig. 6a. As a result, the graph G' of Fig. 6c also has a single template. The justification for the above assumption is that G is constructed from an HDL description of the input circuit C, which ensures that nodes

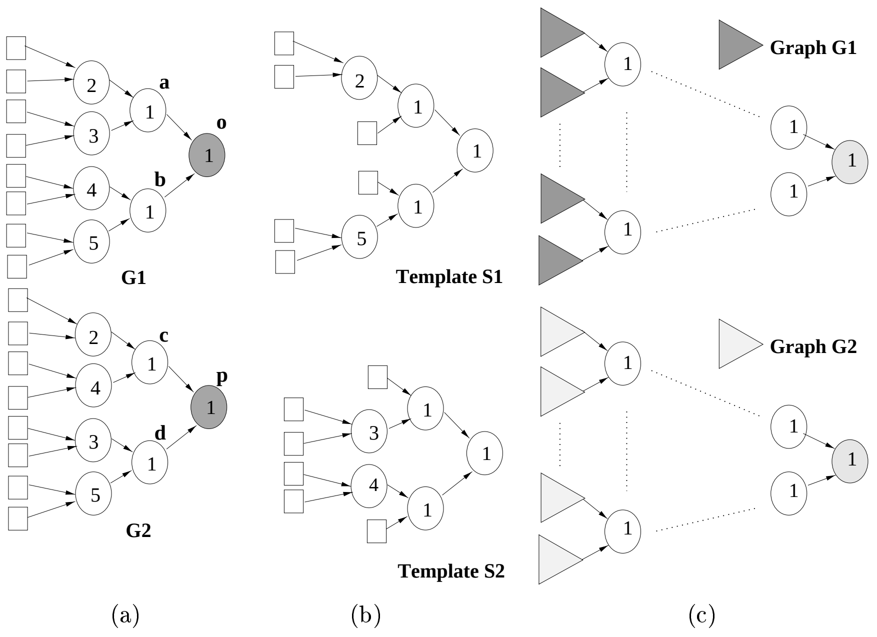


Figure 6: (a) A graph G' composed of G_1 and G_2; (b) the two templates of G' obtained by permuting incident edges of the nodes; (c) the graph G has $O(2^V)$ templates.

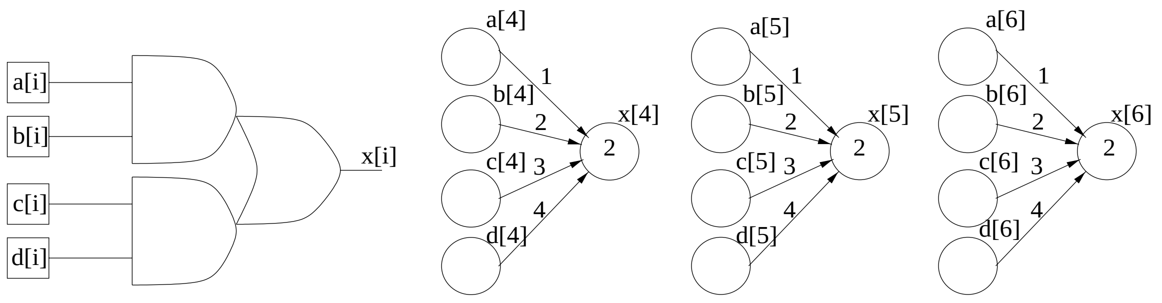


Figure 7: Representing the HDL assignment "`for i = 4 to 6` { `x[i] = a[i]· b[i] + c[i]· d[i]`}" in G. Note that the indices of edges are different from those in Fig. $3b$ as a result of Assumption 2.

with the same function are defined identically. For example, the HDL assignment statement " `for i = 4 to 6` { `x[i] = a[i]· b[i] + c[i]· d[i]` }" will correspond to a set of three nodes with the same logic function which are transformed identically in building G; see Fig. 7. Therefore, the above assumption does not rule out the regularity inherent in the HDL description.

4 Generation of tree templates

A *tree template*, as the name implies, is a template, which has a single output and no internal reconvergence. We present an algorithm for generating all tree templates of a given graph G under Assumptions 1 and 2. The number of tree templates is reduced to within V^2 under these two assumptions, which will directly follow from the description of the algorithm. The upper bound of V^2 makes the enumeration of such templates feasible.

We first explain a scheme to compactly store the set of tree templates, assuming that the fanin of nodes of G are bounded. The templates are stored in a set $S_T = \{S_1, \ldots, S_m\}$, where every template S_i is a class of functionally-equivalent subgraphs. Instead of storing each template completely, we store a template as a set of hierarchically organized templates. A template S_i can be completely defined by the logic function of its root node, denoted by $root\_fn[i]$, and the list of templates $children\_templates[i] = \{T_1, \ldots, T_f\}$ to which the subgraphs rooted at the f fanin nodes of the root node belong to. For example, Fig. 8 illustrates the templates of the graph G' shown earlier in Fig. 6a. The template S_8 can be precisely defined by $root\_fn[8] = 1$ and $children\_templates[8] = \{T_6, T_7\}$. We also reduce the space required for storing the subgraphs of each template by simply storing the root node of the subgraphs in the list $root\_nodes[i]$. In case of the template S_8 in Fig. 8b, $root\_nodes[8] = \{o, p\}$. It can be shown easily that the subgraphs of a template S_i can be precisely reconstructed using $root\_fn[i]$, and the lists $children\_templates[i]$ and $root\_nodes[i]$. Thus, the storage requirements of a template is $O(f + V)$, or $O(V)$ for bounded-fanin graphs.

The algorithm for generation of tree templates is presented in Fig. 9. For efficiency reasons, we sort the template list S_T by a composite key of size $f+1$, defined as $key = \{root\_fn, children\_templates\}$. We explain the algorithm using the example of Fig. 8. First the nodes of G are topologically sorted. Then, for every pair of nodes, the function *Largest_Template* generates a template with two subgraphs, once rooted at each node. *Largest_Template* compares the logic function of the two nodes, and then constructs the list of children templates. The template S_m, thus generated, is compared with previously-generated templates by a binary search on the list S_T using key. If S_m is equivalent to an existing template S_k, then its subgraphs are added to S_k; otherwise S_m is stored in S_T as a new template. For the graph of Fig. 8a, first the trivial templates $S_1, \ldots, S_4$ are generated. Then, from the remaining nodes $\{a, b, c, d, o, p\}$, template S_5 is generated by comparing a and b, and S_6 is generated by comparing a and c. The template obtained by comparing a and d is found to be equivalent to S_5, so no new template is created, but d is stored in the $root\_nodes$ of S_5. The remaining two templates, S_7 and S_8, are generated by comparing the node pairs, (b, d) and (o, p), respectively. *Largest_Template* returns a NULL template, in the case of remaining node pairs. Note that every template has only two subgraphs, except S_5 with six subgraphs given by $root\_nodes = \{a, b, c, d, o, p\}$.

The execution time of levelizing a graph is $O(V + E)$ [3], or $O(V)$ for bounded-fanin graphs. The function *Largest_Template* takes a constant time, since the fanin of nodes is assumed to be

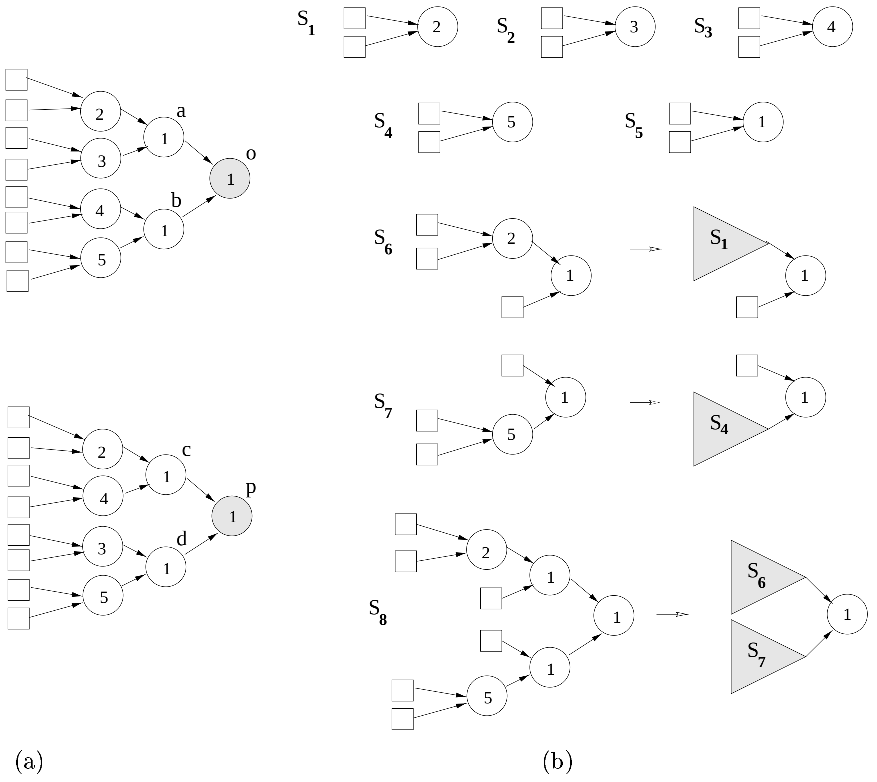


Figure 8: (a) The graph G' of Fig. 6a; (b) tree templates generated by the algorithm of Fig. 9. Note that S_8 can compactly and precisely represented by $root\_fn[8] = 1$, $children\_templates = \{T_6, T_7\}$ and $root\_nodes = \{o, p\}$.

```
**Generate_Templates**($G(V, E)$)
/* A tree template $S_i$ is completely defined by *(i) root_fn*$[i]$ — logic function of the root node;
*(ii) children_templates*$[i]$ — list of children templates that form $S_i$; *(iii) root_nodes*$[n]$ — a list of the
root nodes of the subgraphs of $S_i$ (all the subgraphs of $S_i$ can be constructed by these three fields) */
begin
topologically sort the nodes of $G$ as $\{v_1, \ldots, v_N\}$;
$S_T := \emptyset$;    /* $S_T$ stores the list of templates */
$m := 0$;    /* $m$ is the number of templates generated so far */
$template[v_1 \ldots v_N, v_1 \ldots v_N] := 0$;    /* $template[v_i, v_j]$, if non-zero, gives the index of the template
          to which the functionally-equivalent subgraphs rooted at nodes $v_i$ and $v_j$ belong to */
for $i = 1$ to $N$
    for $j = i + 1$ to $N$
        $m := m + 1$;    /* the new template will be stored in $S_m$ */
        $S_m :=$ **Largest_Template**($v_i$, $v_j$);    /* generates a template with two largest functionally-
              equivalent subgraphs $G_i$ and $G_j$ rooted at $v_i$ and $v_j$, respectively */
        if $S_m \neq \emptyset$
            $k :=$ **Find_Equivalent_Template**($S_m$, $S_T$);    /* find $S_k$ in $S_T$ equivalent to $S_m$ */
            $template[v_i, v_j] := k$;
            if $k = m$    /* $S_m$ is a new template */
                $S_T := S_T \cup \{S_m\}$;    /* add $S_m$ to $S_T$, such that it remains sorted */
            else
                $root\_nodes[k] := root\_nodes[k] \cup \{v_i, v_j\}$;    /* add, only if not present already */
                $m := m - 1$;
return $S_T$;
end

**Largest_Template**($u$, $v$)
/* generates the largest trees rooted at nodes $u$ and $v$ that are functionally equivalent */
if $l[u] \neq l[v]$    /* if $u$ and $v$ have different logic functions, then there is no template */
    return $\emptyset$;
else
    $root\_fn[m] := l[u]$;    /* setting the fields of template $S_m$ */
    for $i = 1$ to $f$ do    /* both $u$ and $v$ have $f$ fanin nodes, $\{u_1, \ldots, u_f\}$ and $\{v_1, \ldots, v_f\}$, each */
        if $u_i$ and $v_i$ have a single fanout each
            add $template[u_i, v_i]$ to $children\_templates[m]$;
    $root\_nodes[m] := \{u, v\}$;    /* $S_m$ has only two subgraphs, $G_u$ and $G_v$ */
return $S_m$;

**Find_Equivalent_Template**($S_m$, $S_T$)
/* $S_T$ is a list $\{S_i, \ldots, S_j\}$ of templates sorted by $key = (root\_fn, children\_templates)$.
This function finds the template in $S_T$, equivalent to $S_m$, by performing a binary search */
if $S_T = \emptyset$
    return $m$
if $key[m] < key[\frac{i+j}{2}]$    /* check in the first half of $S_T$ */
    return **Find_Equivalent_Template**($S_m$, $\{S_i, \ldots, S_{\frac{i+j}{2}-1}\}$);
else if $key[m] > key[\frac{i+j}{2}]$    /* check in the second half of $S_T$ */
    return **Find_Equivalent_Template**($S_m$, $\{S_{\frac{i+j}{2}+1}, \ldots, S_j\}$);
return $\frac{i+j}{2}$;    /* $S_{\frac{i+j}{2}}$ and $S_m$ are equivalent */
```

Figure 9: Algorithm for generating the complete set of tree templates of G under Assumptions 1 and 2.

bounded. The function *Find_Equivalent_Template* takes $O(logV)$ for the binary search on the list S_T of at most V^2 templates [3]. Insertion of a template S_m in S_T (line 15) also takes in $O(logV)$ time, since S_T is already sorted. The above functions are performed for every node-pair, resulting in the overall time complexity of *Generate_Templates* of $O(V^2.logV)$. We store the *root_fn* and the list *children_templates* for every template, which results in a memory requirement of $O(V^2)$. The subgraphs of the templates are stored using a list of their root nodes in *root_nodes*. Since the maximum number of subgraphs generated is $O(V^2)$, the storage required for the subgraphs is also $O(V^2)$. Thus, the overall storage complexity is $O(V^2)$.

5 Generation of multi-output templates

The template generation algorithm of Fig. 9 gives excellent covers for sparse graphs, but it might not perform well for graphs with a high number of nodes with multiple fanout. If we apply this algorithm to the 4×4 multiplier shown as a graph in Fig. 4, then three trivial tree templates — AND gate, CARRY and SUM functions, are obtained. We now extend the algorithm for tree templates to multi-output templates. We restrict ourselves to only those multi-output subgraphs, whose every output lies in the transitive fanin of a particular output. We refer to this particular output as the *principal output* of the subgraph, and such a subgraph (template) as a *single principal-output* subgraph (template) or a *single-PO* subgraph (template). The remaining outputs are called *secondary outputs*. For example, the two subgraphs shown in Fig. 10*a* of the graph of Fig. 4 are single-PO graphs with $P5$ and $P4$ as the respective principal outputs.

We briefly analyze some interesting properties of the class of single-PO graphs (SPOGs). A SPOG is a directed graph which has at least one path from every output to the principal output. Thus, SPOGs can have internal reconvergence as well as cycles, and can have any number of outputs. Figure 11 shows the relation of SPOGs to widely-studied classes of circuit graphs — directed-acyclic graphs (DAGs), fanout-free cones (FFCs), leaf-DAGs and trees[1]. SPOGs can be used to represent sequential circuits as well, since they allow cycles, unlike DAGs. Besides regularity extraction, there can be other useful applications of SPOGs; circuit decomposition in technology mapping is one such example. A circuit graph is usually decomposed into trees prior to technology mapping. However, for a circuit graph G with V nodes and O primary outputs, the number of trees can be as high as V, which might result in a poor mapping solution. An alternate decomposition is in to O SPOGs, where O is usually much smaller than V. For example, the multiplier of Fig. 4 is decomposed into as many as 40 trees, but can be decomposed into only 8 SPOGs.

[1]A DAG is a directed graph with no cycles, a fanout-free cone is a DAG with a single output, a leaf-DAG is a fanout-free cone where only leaf nodes can have multiple fanout.

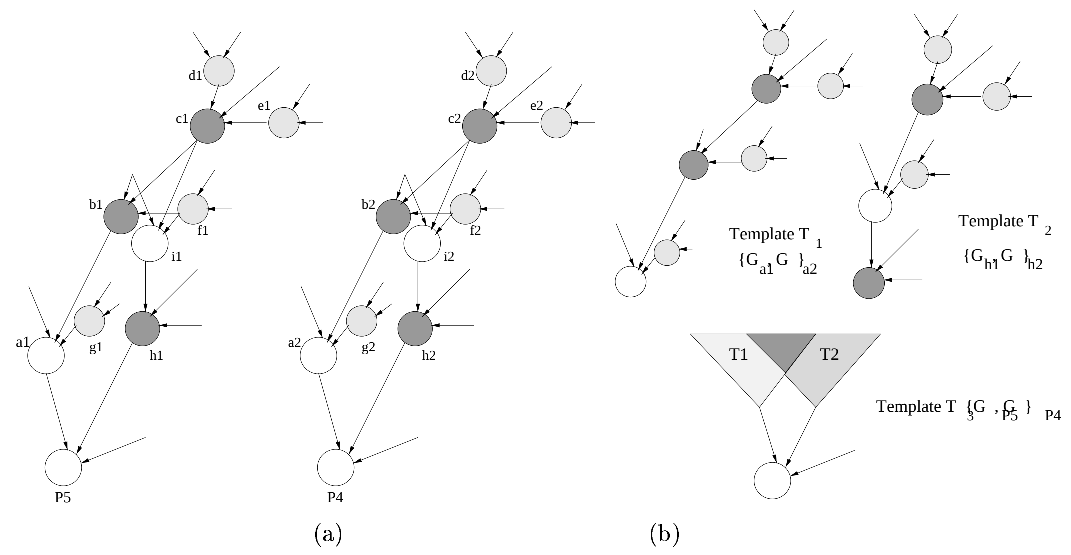


Figure 10: (a) Two functionally-equivalent single-PO subgraphs G_{P5} and G_{P4} of the graph of Fig. 4; (b) the two templates with overlapping nodes which are merged to form the template S_3.

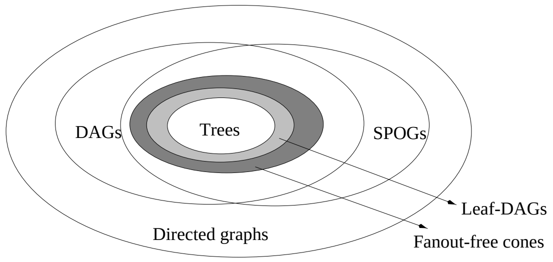


Figure 11: The relation among the various classes of circuit graphs.

The main advantage of using single-PO subgraphs in regularity extraction is that despite their complex structure, the number of such subgraphs of G is restricted to V^2 under the Assumptions 1 and 2, similar to the tree case. However, the generation of single-PO subgraphs (and templates) requires more time and memory, which is analyzed next. The storage scheme used earlier for tree templates cannot be extended to single-PO templates. A tree template was earlier represented by a list of children templates, since these children templates are non-overlapping. However, in case of single-PO templates, the children templates can overlap with each other. Figure 10*c* shows the template S_3 which contains the two subgraphs of Fig. 10*a-b*. S_3 has two children templates, S_1 and S_2, which have overlapping nodes, such as $c1$, $f1$ for subgraph G_{P5} and $c2$, $f2$ for subgraph G_{P4}. Therefore, S_3 cannot be completely specified just by the list of its children templates. Instead, we have to specify every template individually.

We represent a single-PO subgraph G_u by a list of its nodes, *nodelist*, which stores the nodes in a depth-first search order. The motivation for using a depth-first order is that it is unique for all isomorphic subgraphs. The subgraph of template S_1 in G_{P_5} has $nodelist = \{a1, b1, c1, d1, e1, f1, g1\}$. With every node in *nodelist*, we also store its fanin and fanout nodes as well. Thus, memory required to store a subgraph is $O(V)$, since every node requires a fixed storage for bounded-fanin graphs.

We replace the two functions in the template generation algorithm of Fig. 9 by the corresponding functions in Fig. 12 in order to generate the complete set of single-PO templates. We explain these functions with the aid of the example of Fig. 10. Prior to the call $Largest\_Template(P5, P4)$, the template S_1 is already generated with two subgraphs, G_{a_1} and G_{a_2}. Similarly, S_2 is also generated with two subgraphs, G_{h_1} and G_{h_2}. First, $nodelist[G_{P5}]$ and $nodelist[G_{P4}]$ are set as $\{P5\}$ and $\{P4\}$, respectively (lines 05-06). The nodelists of G_{a_1} and G_{h_1} (G_{a_2} and G_{h_2}) are then added to the nodelist of G_{P5} (G_{P4}). The nodelists after lines 07-09 are given below.

$$nodelist[G_{P5}] = \{P5, a1, b1, c1, d1, e1, f1, g1, h1, i1, c1, d1, e1, f1\}$$
$$nodelist[G_{P4}] = \{P4, a2, b2, c2, d2, e2, f2, g2, h2, i2, c2, d2, e2, f2\}$$

There can be multiple paths from a node w to the root node v through different incoming edges of v. As a result, w occurs multiple times in $nodelist[G_v]$. For example, $c1$ is connected to $P5$ through the edges $(a1, P5)$ and $(h1, P5)$ in Fig. 10*a*, and hence, it occurs twice in the $nodelist[G_{P5}]$. We define a list $path[w, v]$ which contains the indices of the incoming edges of v through which w is connected to v, e.g. $path[b1, P5] = \{1\}$, while $path[c1, P5] = \{1, 2\}$. The *path* lists for all nodes of the subgraph G_u and G_v, rooted at nodes u and v, are evaluated in lines 10-12.

```
    /* $S_T$ stores the set of all templates $\{S_1, \ldots, S_m\}$. Every subgraph $G_u$ of a template $S_i$
    is represented by a node-list $nodelist[G_u]$ stored in the depth-first order */
01  **Largest_Template**($u$, $v$)
    /* generates the largest single-PO subgraphs rooted at nodes $u$ and $v$ that are equivalent */
02  if $l[u] \neq l[v]$
03      return $\emptyset$;
04  else
05      $nodelist[G_u] := \{u\}$;     /* the root node is always the first node in $nodelist$ */
06      $nodelist[G_v] := \{v\}$;
07      for $i = 1$ to $f$ do     /* both $u$ and $v$ have $f$ fanin nodes, $\{u_1, \ldots, u_f\}$ and $\{v_1, \ldots, v_f\}$, each */
08          add $nodelist[G_{u_i}]$ at the end of $nodelist[G_u]$;
09          add $nodelist[G_{v_i}]$ at the end of $nodelist[G_v]$;
10          for $w_1 \in nodelist[G_{u_i}]$ and $w_2 \in nodelist[G_{v_i}]$
11              add $i$ to $path[w_1, u]$;     /* there is a path from $w_1$ ($w_2$) to $u$ ($v$) through the incoming */
12              add $i$ to $path[w_2, v]$;     /* edge of $u$ ($v$) with index $i$ */
13      for $w_1 \in nodelist[G_u]$ and $w_2 \in nodelist[G_v]$
14          if $path[w_1, u] \neq path[w_2, v]$
15              delete all copies of $w_1$ ($w_2$) from $nodelist[G_u]$ ($nodelist[G_v]$)
16          else if $path[w_1, u]$ has more than one elements     /* here, $path[w_1, u] = path[w_2, v]$ */
17              delete all remaining copies of $w_1$ ($w_2$) from $nodelist[G_u]$ ($nodelist[G_v]$)
18  $S_m := \{G_u, G_v\}$;
19  return $S_m$;

20  **Find_Equivalent_Template**($S_m$, $S_T$)
    /* $S_T$ is a list of $k$ templates, $S_1, \ldots, S_k$.
    This function finds the template in $S_T$, equivalent to $S_m$, if any; otherwise, returns $m$ */
21  for $i = 1$ to $k$
22      if $nodelist[S_i] = nodelist[S_m]$
23          return $i$;
24  return $m$;
```

Figure 12: Algorithm to generate the complete set of single-PO templates of G under Assumptions 1 and 2.

We then pairwise compare the nodes in *nodelist* of G_u and G_v (line 13). If the *path* lists of the corresponding nodes are different, then these nodes have to be removed from the respective subgraphs (lines 14-16). Otherwise, if the *path* lists of the corresponding nodes are the same and have multiple indices, then the remaining copies of these nodes are removed from their respective *nodelist*'s. For example, the second occurrence of the node $c1$ ($c2$) in the graph G_{P5} (G_{P4}) is deleted. The final nodelists of G_{P5} and G_{P4} (line 21) are given below.

$$nodelist[G_{P5}] = \{P5, a1, b1, c1, d1, e1, g1, h1, i1\}$$
$$nodelist[G_{P4}] = \{P4, a2, b2, c2, d2, e2, g2, h2, i2\}$$

The function $Find\_Equivalent\_Template$ compares a template with every other template in the set S_T by matching corresponding nodes in the two *nodelist*'s. Since the depth-first order of the nodes of a graph is unique, two graphs are isomorphic if and only if the corresponding nodes in their *nodelist*'s are the same in terms of their logic functions as well as their lists of fanouts.

We now analyze the complexity of the algorithm of Fig. 12. The *nodelist* created after line 12 in $Largest\_Template$ is $O(f \cdot V)$, or $O(V)$ for bounded fanin graphs. The unmatched or duplicate nodes are deleted from the *nodelist* by a single traversal (lines 13-17), which also takes $O(V)$ time. Thus, the time complexity of $Largest\_Template$ is $O(V)$. The comparison of two *nodelist*'s (line 24) in the function $Find\_Equivalent\_Template$ takes $O(V)$ time, and is called for every template in S_T resulting in a time complexity of $O(V^3)$. Finally, since these two functions are called for every node-pair (line 07-08, Fig. 9), the overall time complexity is $O(V^5)$. As discussed earlier, every subgraph requires a storage of $O(V)$. Since the maximum number of subgraphs generated is V^2, the storage complexity is $O(V^3)$. If the number of single-PO templates of G are bounded by S, then the overall time and space complexity are given by $O(S^2 \cdot V)$ and $O(S \cdot V)$, respectively.

6 Covering of graph by templates

So far, we have presented algorithms to generate a set S_T of templates of G. S_T can be either a set of all tree templates or a set of all single-PO templates of G under the Assumptions 1 and 2. Let S denote the set of all subgraphs in the templates stored in S_T. Now, we present a solution to the graph covering problem, where given G and S_T, the objective is to find a subset $C(G, S_T)$ of the set S of all subgraphs subgraphs that forms a cover of G.

Since a large set S of subgraphs are generated to choose the cover and the binate covering problem is inherently difficult, we focus on efficient heuristics to solve the covering problem. Our

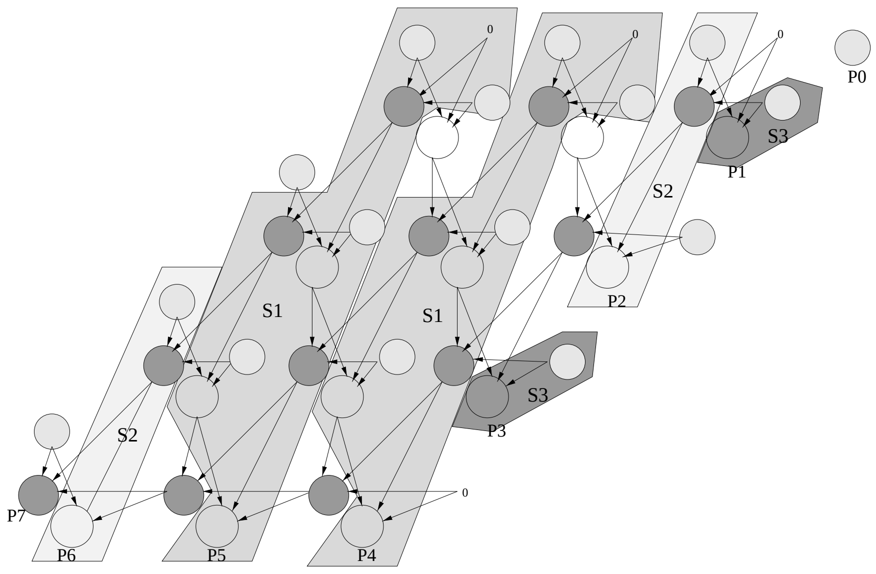


Figure 13: The cover of the 4×4 multiplier of Fig 2 obtained using LFF heuristic on the set of single-PO templates. The nodes shown uncovered are covered by single-node (trivial) templates.

approach, at every step, selects a template S_i with the maximum objective function out of all templates in S_T, deletes all nodes of G that belong to the non-overlapping subgraphs of S_i, and then generates the set S_T of templates for the remaining graph. This step is repeated until either all nodes of G are covered, or if S_T is found to be NULL. If some nodes are left uncovered and S_T becomes NULL, then we store the remaining nodes in a template with a single subgraph. (In case of datapath circuits, this template will correlate to the control logic.)

We use the following two heuristics for graph covering based on the objective function used for selecting templates from the set S_T.

1. *Largest-Fit-First (LFF)* heuristic: Select the template S_i with the the maximum area $area[S_i]$ (defined in Section 2).
2. *Most-frequent-Fit-First (MFF)* heuristic: Select the template S_i with the maximum number $|S_i|$ of subgraphs.

Usually, these two heuristics give different covers, since a template with a large area has few subgraphs, and vice-versa. The cover of the 4×4 multiplier of Fig. 4 obtained using the LFF heuristic is shown in Fig. 13. (The cover of two templates shown in Fig. 2 cannot be obtained, since our algorithm is restricted to single-PO templates.) If the MFF heuristic is used, then the cover of three small templates — AND gate, CARRY and SUM functions, is obtained.

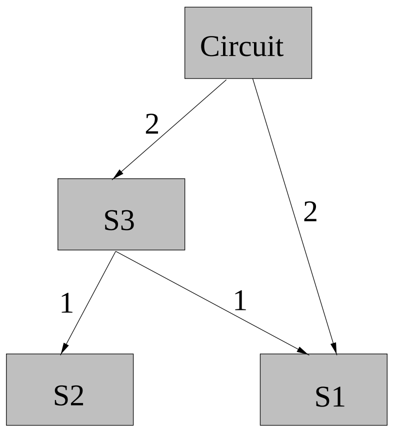


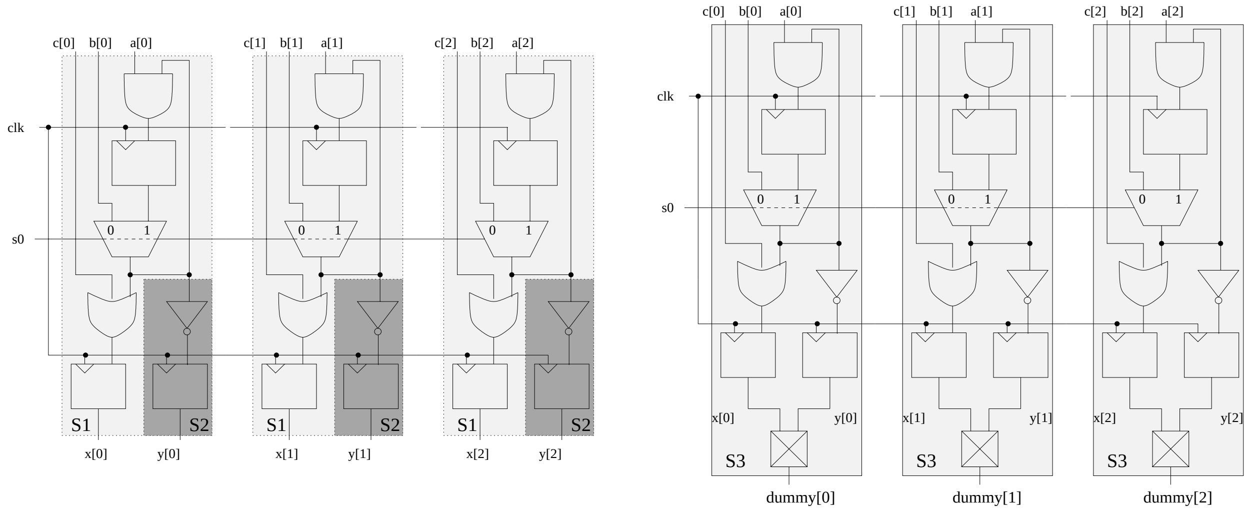
Figure 14: The template hierarchy of the circuit of Fig. 1 obtained by our extraction approach.

7 Applications of template generation

We now discuss several interesting applications of the template generation algorithm, which further generalizes our regularity extraction methodology.

Hierarchical representation of regularity: Consider the two covers for the circuit graph of Fig. 1 generated by our extraction approach — one with two subgraphs of $S3$ and $S1$ each, and another one with two subgraphs of $S2$ and four of $S1$. The fact that $S3$ is composed of $S1$ and $S2$ is not captured by these two covers. We can compactly represent these two covers by identifying the hierarchy of templates. Earlier in the case of tree templates, we hierarchically stored a template as a set of children templates. We now generalize this notion of *template hierarchy*. For a given G, every template is either hierarchically composed of other templates or is a leaf template. Let $S_1, \ldots, S_m$ be the templates in a cover generated by the regularity extraction approach. We can generate the complete template hierarchy by recursively extracting the regularity from the graph composed of $S_1, \ldots, S_m$, until we are left with leaf templates only. The templates in the two covers of Fig. 1 discussed above can be compactly represented by the hierarchy shown in Fig. 14. In general, any set of covers of G can be represented by a template hierarchy, which allows the user to select the most desirable cover by descending the hierarchy.

Generating subgraphs for a given template: Given a template S, we can modify the template generation algorithm to identify all subgraphs of S as well as its children templates. For example, if the user provides the template $S3$ for the circuit in Fig. 1, then all subgraphs of $S3$ as well as its children templates $S1$ and $S2$ can be generated. The only modification to the template generation algorithm is that function *Largest_Template* (line 10, Fig. 9) should be called for every node-pair (v_i, v_j), where v_i and v_j belong to G and S, respectively. The covering step can be generalized,



(a) (b)

Figure 15: Larger templates can be identified by creating a dummy output bus for a datapath circuit – (a) a cover of two single-PO templates; (b) a cover of a single template using the heuristic of a single dummy output bus.

such that G is covered by a mix of automatically-generated and user-supplied templates.

General multi-output templates: Usually the primary outputs of datapath circuits are specified by busses. We can group primary outputs with the same bus index using a dummy node, thus creating a dummy bus of the same width. For example, the two output busses $x[2..0]$ and $y[2..0]$ of the circuit of Fig. 15 can be grouped into a single dummy bus. Using the heuristic of adding a dummy output bus, our template generation algorithm can result in a general multi-output template, as shown in Fig. 15b.

8 Experimental results

The only input to our regularity extraction technique is the graph G of a circuit C. The input circuit can be described in any format, such as an HDL or the gate-level *blif* format [7], from which G can be constructed in a straightforward manner. We extract the regularity for a variety of circuits, including adders, $74X$ series circuits [18] and ISCAS benchmark circuits. The ISCAS benchmarks are already described in the blif format. We have written gate-level descriptions for adders and $74X$ circuits from their functional descriptions. We obtained a set of four covers for each circuit, depending on whether tree templates or single-PO templates are generated, or whether the LFF or MFF covering heuristic is used. In fact, other covers can be obtained by using alternate covering heuristics.

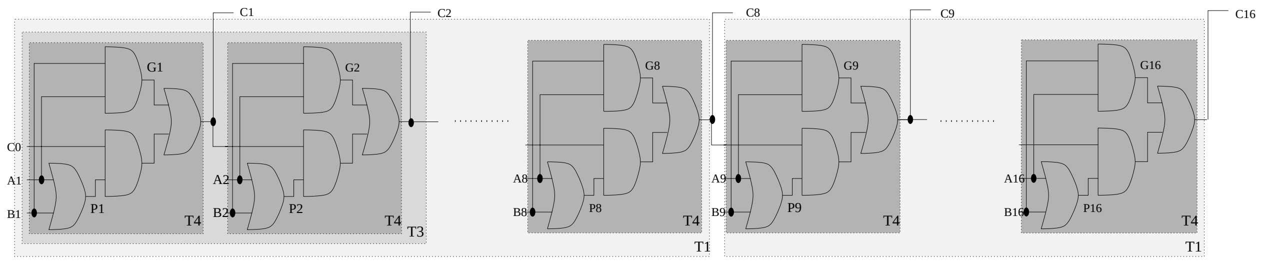


Figure 16: A 16-bit ripple carry function illustrating the template hierarchy.

We first analyze some interesting covers for several circuits, and then summarize the results for the complete set of circuits.

Ripple-carry function: Figure 16 shows a 16-bit ripple-carry function. A cover of a single template $S1$ with two instances is obtained by using the LFF heuristic on the set of single-PO templates. If only the tree templates are considered, then the cover of a smaller template $S4$ with 16 instances is obtained. In fact, we recursively extract the regularity from template $S1$ to get a set of covers given by $\{T1(2)\}$, $\{T2(4)\}$, $\{T3(8)\}$, and $\{T4(16)\}$, where $\{Ti(n_i)\}$ implies a cover of n_i instances of template Si. Thus, various covers can be compactly described by a template hierarchy.

Carry-lookahead function: Figure 17 shows a 16-bit carry-lookahead logic block which is realized using sub-blocks of four bits each. The largest single-PO template is $S1$ with two instances, one rooted at $C8$ and the other at $C16$. Selecting $S1$ results in the cover $\{T1(2), T2(4), T3(4), T4(4)\}$ shown in Fig. 17. Further, $S1$ can be shown to be composed of just one template with two instances. Thus, the algorithm is able to detect multi-bit regularity efficiently.

74181 **4-bit ALU**: The 74181 4-bit ALU of Fig. 18a [9] is found to have a single-PO template $S1$ with four instances which cover most of the circuit. The carry-lookahead logic is, however, not covered by any template.

7485 **magnitude comparator**: The gate-level realization of 74L85 magnitude comparator [18], shown in Fig. 18b, is composed of two carry-lookahead modules. The cover has two templates — a template composed of an AND gate and a carry-lookahead module, and a template with an XOR gate. The uncovered logic of the AND gate is represented by a template with a single instance.

c499 (c1355): This ISCAS-85 benchmark circuit of Fig. 19, described in [9], is a single-error-correcting circuit which reads in a 32-bit bus, generates a set of eight syndrome lines, and then corrects the appropriate bit in the output bus. The largest single-PO template is shown to cover all of the syndrome generation logic and a part of the remaining error correction logic. The remaining circuit is covered by five templates. The c1355 benchmark implements the same logic as c499, except that each XOR gate is represented by a set of four NAND gates. As expected, we get the

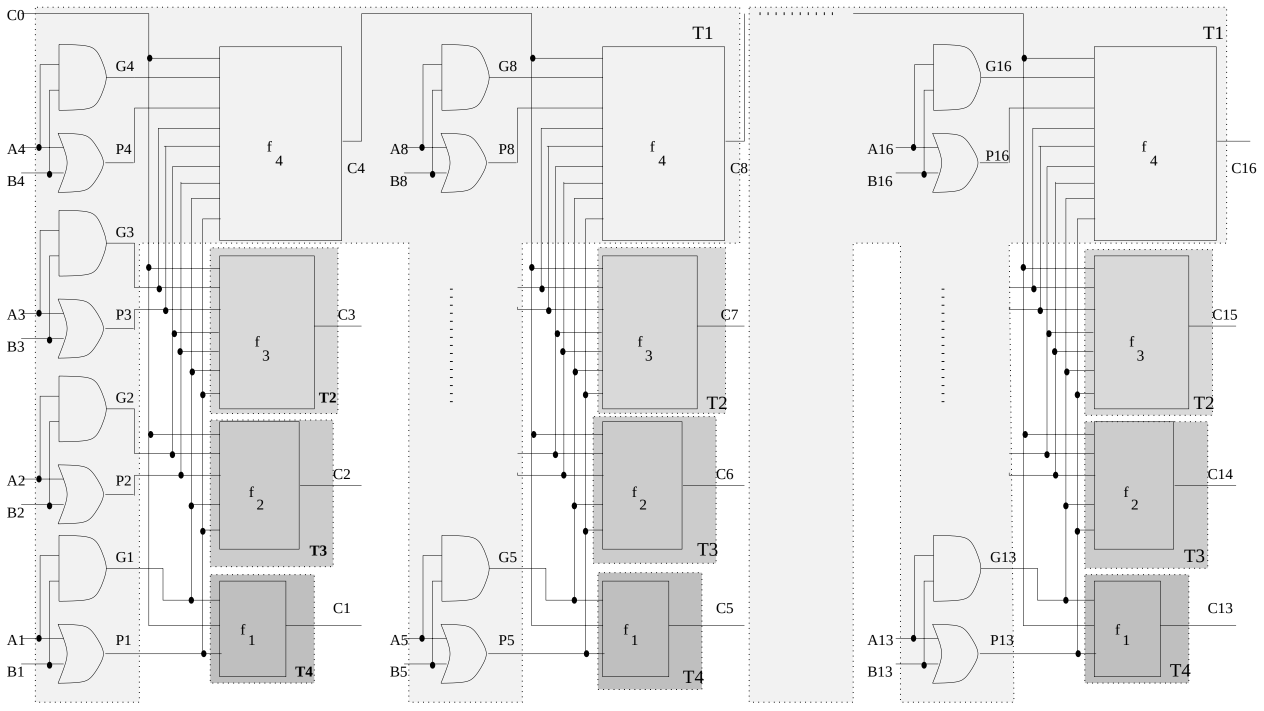


Figure 17: A 16-bit carry-lookahead circuit; most of its nodes are covered by the largest template $S1$.

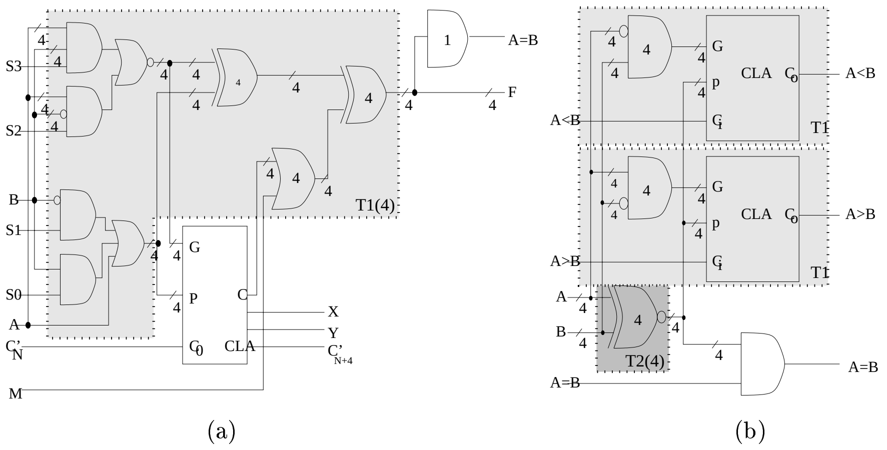


Figure 18: (a) The 74181 4-bit ALU has a single template with four instances; (b) the 74L85 magnitude comparator with its optimal cover.

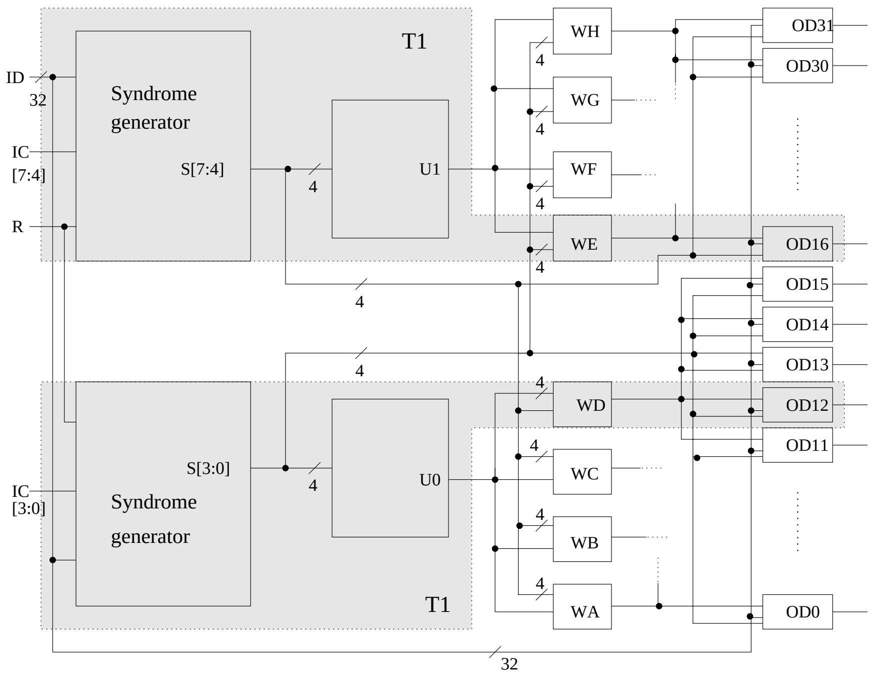


Figure 19: The c499 ISCAS circuit [9], where the largest template covers the entire syndrome generator logic and a part of the remaining error correction logic.

same largest template as for c499.

c2670: This ISCAS benchmark is an ALU with two identical comparator subcircuits [19] apparently used for fault-tolerant reasons. As expected, we are able to identify two instances of a template which corresponds to the comparator of 12-bit inputs.

Finally, we summarize the results of regularity extraction on the above set of circuits. Tables I and II give the covers obtained by applying the covering heuristics on the sets of tree and single-PO templates, respectively. We define *regularity index* of a cover $C(G)$ as the area of all templates in the cover, given by $\sum_{i=1}^{m} area[S_i]$, as a percentage of the total area of G, given by $\sum_{i=1}^{m} |S_i| \cdot area[S_i]$, The regularity index correlates with the reduction in the design effort, assuming that a template is synthesized only once for all its subgraphs. We can compare the quality of covers using the regularity index and the area of the largest template. A small regularity index implies that a low effort is needed for synthesis and layout of the circuit, while a large template implies that a better optimization can be achieved during subsequent synthesis and layout stages. The results indicate that the LFF heuristic generates covers with large templates, e.g., the two instances of the largest single-PO template of c1355 together account for two-thirds of the overall area. Such covers have a high regularity index which can be reduced by hierarchically extracting regularity. On the other hand, covers obtained using the MFF heuristic have a small regularity index as well as small templates. Figure 20 shows the variation in the regularity indices for the covers of some benchmark circuits. In fact, alternate covers can be obtained by using a combination of LFF and MFF heuristics, or other covering heuristics.

		LFF heuristic			MFF heuristic		
Ckt.	No. of gates	# templates (# subgraphs)	Largest template	Regularity index	# templates (# subgraphs)	Largest template	Regularity index
ripple-carry	64	1(16)	6.3	6.3	2(64)	1.3	3.1
16-bit CLA	48	6(40)	6	17	6(48)	2	13
74181	41	4(17)	7.3	33	5(21)	7.3	32
7485	15	3(7)	33	26.7	4(15)	6.7	26.7
4-bit mult.	40	4(40)	2.5	10	4(40)	2.5	10
c432	160	9(89)	3.1	11.9	7(159)	0.6	5
c499	202	7(66)	8.5	17	6(202)	0.5	3
c880	383	18(178)	3.6	15.1	9(383)	0.3	2.3
c1355	546	8(298)	3.1	5.1	7(546)	0.2	1.3
c1908	880	18(425)	0.8	5.2	12(879)	0.1	1.5
c2670	1193	23(604)	2.7	11.6	12(1193)	0.1	1
c3540	1669	44(652)	3.9	21.2	15(1669)	0.1	0.9
c5315	2307	37(845)	0.6	7.8	15(2307)	0.1	0.6

Table I: Covers composed of tree templates obtained using largest-fit-first and most-frequent-fit-first heuristics.

	LFF heuristic			MFF heuristic		
Ckt.	# templates (# subgraphs)	Largest template	Regularity index	# templates (# subgraphs)	Largest template	Regularity index
ripple-carry	1(2)	50	50	2(64)	1.6	3.1
16-bit CLA	4(14)	38	44	6(48)	2	13
74181	2(5)	22	32	5(21)	7.3	32
7485	3(7)	33	26.7	4(15)	6.7	26.7
4-bit mult.	6(16)	25	45	4(40)	2.5	10
c432	9(58)	7.5	24.4	7(159)	0.6	5
c499	6(42)	29.7	36.1	6(202)	0.5	3
c880	19(127)	12.3	35.2	9(383)	0.3	2.3
c1355	7(74)	31.3	35.5	7(546)	0.2	1.3
c1908	27(171)	5	44	12(879)	0.1	1.5
c2670	26(262)	15.5	43.7	12(1193)	0.1	1
c3540	38(224)	28.8	43.4	15(1669)	0.1	0.9
c5315	30(264)	17.3	40.4	15(2307)	0.1	0.6

Table II: Covers composed of single-PO templates obtained using the two covering heuristics.

9 Conclusions

We have presented a comprehensive approach to extract regularity inherent in datapath circuits. Our approach reduces the problem complexity by first generating a set of templates and then selecting its subset to cover the input circuit. The major contributions of this paper are the novel

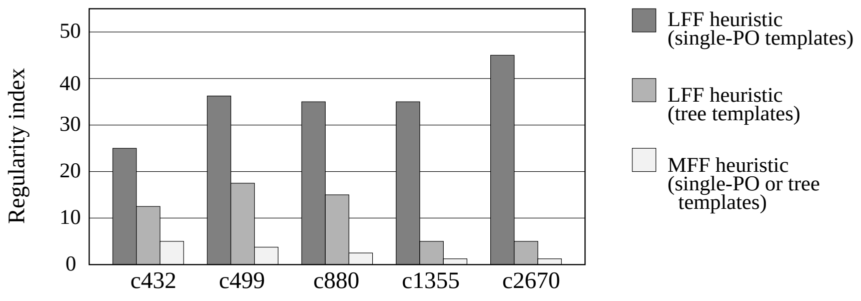


Figure 20: The regularity indices of the covers obtained for some ISCAS benchmarks.

algorithms developed to generate two special classes of templates — the class of tree templates and the class of single-PO templates. Under a few practical assumptions, our algorithm generates the complete set of these classes of templates, which is key to achieving efficient covers. These assumptions have been justified in the case of circuits described using behavioral or structural level HDL. We have obtained a variety of covers for several benchmark circuits by using two different heuristics for selecting templates to cover the graph. We have also demonstrated that the covers generated by our approach help in understanding the underlying structure of the circuits. The identification of regularity would result in a significant reduction of effort in the subsequent synthesis and layout design stages. A hierarchical representation of the regularity in a circuit can also be obtained by recursive application of our approach. Given a user-defined template, all its subgraphs can be easily generated, thus providing the feature of covering the circuit by a mix of user-defined and automatically-generated templates.

We presented a heuristic of grouping output busses into a single bus, that allows us to identify general multi-output templates for some circuits. A useful extension to our extraction approach is to generate general multi-output templates for any given circuit, but the number of such templates is not restricted by V^2 under the two assumptions presented in this paper. Such an algorithm would lead to more efficient covers, such as the cover of just two templates for the 4×4 multiplier (Fig. 2). Furthermore, our approach explicitly enumerates the templates of a circuit, which raises the following question: is it possible to consider all the templates in the covering step without explicitly enumerating the set of templates generated by our algorithm?

References

[1] S. R. Arikati and R. Varadarajan. A signature based approach to regularity extraction. In *Proc. Int'l Conf. on CAD*, pages 542–545, Nov. 1997.

[2] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey. Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Trans. on CAD*, 15(8):877–887, Aug. 1996.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 1990.

[4] O. Coudert and S. Liao. xxxxx. In *Proc. 34th Design Automation Conf.*, pages xx–xx, June 1997.

[5] G. de Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, New York, 1994.

[6] D. W. Dobberpuhl. Circuits and technology for Digital's StrongARM and ALPHA microprocessors. In *Proc. Conf. on Advanced Research in VLSI*, pages 2–11, Sept. 1997.

[7] E. *et al.* Detjens. Technology mapping in MIS. In *Proc. Int'l Conf. on CAD*, pages 116–119, 1987.

[8] R. Gupta and S. Liao. Using a programming language for digital hardware design. *IEEE Design and Test of Computers*, pages 72–80, April 1991.

[9] M. C. Hansen and J. P. Hayes. High-level test generation using physically-induced faults. In *Proc. VLSI Test Symp.*, pages 20–28, May 1995.

[10] F. Harary. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969.

[11] M. Hirsch and D. Siewiorek. Automatically extracting structures from a logical design. In *Proc. Int'l Conf. on CAD*, pages 456–459, Nov. 1988.

[12] K. Keutzer. Dagon: Technology binding and local optimization by DAG matching. In *Proc. 24th Design Automation Conf.*, June. 1987.

[13] J. Li and R. Gupta. HDL code restructuring using TDTs. In *Proc. Int'l Workshop on Codesign*, March 1998.

[14] R. X. T. Nijssen and C. A. J. van Eijk. Regular layout generation of logically optimized datapaths. In *Proc. Int'l Symp. on Physical Design*, pages 42–47, 1997.

[15] G. Odawara, T. Hiraide, and O. Nishina. Partitioning and placement technique for CMOS gate arrays. *IEEE Trans. on CAD*, 6(3):355–363, May 1987.

[16] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architecures. *IEEE Design and Test of Computers*, pages 40–51, June 1991.

[17] D. S. Rao and F. J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Trans. on CAD*, 12(8):1198–1208, Aug. 1993.

[18] Texas Instruments Inc. *The TTL Data Book*, Dallas, Texas, 1988.

[19] H. Yalcin, J. P. Hayes, and K. A. Sakallah. An approximate timing analysis method for datapath circuits. In *Proc. Int'l Conf. on CAD*, pages 114–118, Nov. 1996.