# A General Approach for Regularity Extraction in Datapath Circuits

Amit Chowdhary[†], Sudhakar Kale[†], Phani Saripella[†], Naresh Sehgal[†], Rajesh Gupta[‡]
{amitc, skale, phani, nsehgal}@scdt.intel.com rgupta@ics.uci.edu

[†] Intel Corporation
Santa Clara, CA 95052

[‡] 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 present a general and comprehensive approach to extract functional regularity for datapath circuits from their behavioral or structural HDL 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 complete under a few simplifying, yet practical, assumptions. This is key to obtaining a desirable cover of the circuit using templates. We show that excellent covers are obtained for various circuits, including ISCAS benchmarks. We also demonstrate that the regularity extracted for these circuits can be used to understand their underlying structure. We have successfully used our approach to identify bit slices of very large datapath circuits from general-purpose microprocessors.*

## 1 Introduction

Datapath circuits perform various arithmetic and multiplexing operations on wide busses. Such circuits have a very high degree of regularity. Designers often exploit this regularity in circuits to achieve layouts with a small area and a high performance. Design effort can be reduced by identifying regularity in circuits, thus improving the productivity of designers. Therefore, an important task in circuit design is to extract the regularity inherent in the circuits. Currently, datapath circuits in general-purpose microprocessors are designed almost entirely by hand [5], since the existing CAD tools can not extract and utilize regularity to the extent necessary for competitive designs.

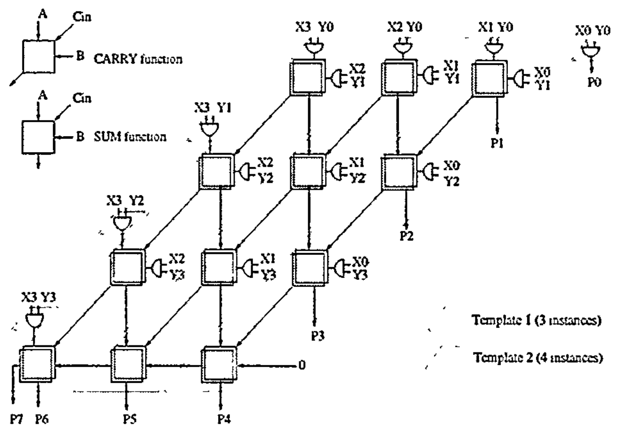

Figure 1: A $4 \times 4$ multiplier with inputs $X3, \ldots, X0$, and $Y3, \ldots, Y0$, which is covered by two templates.

We assume that circuits are 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 multiplexers, or arithmetic operators, such as adders and shifters. Regularity in a circuit implies that there exists subcircuits, called *templates*, which have multiple instances in the circuit. The task of regularity extraction is to identify a set of templates, and cover the circuit by a subset of these templates, where the objective is to use large templates with many instances. The regularity extraction involves a tradeoff, since a large template usually has a few instances, while a small template has many instances. For a $4 \times 4$ multiplier of Fig. 1, the template 1 composed of a diagonal array has only three instances, while a template composed of the SUM function has 12 instances. If the structure of the multiplier is unknown, then the extraction technique should generate a cover, such as the cover of only two templates (Fig. 1). In general, an extraction approach should be able to generate a range of covers given only its high-level description.

Regularity in a given circuit can be 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 can be used to restructure the HDL code, for instance to improve the quality of high-level synthesis results by identifying opportunities for resource sharing [9]. Structure in an HDL description typically refers to declaratively specified blocks [6] consisting of a netlist which 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. An ideal synthesis approach for datapath circuits should identify functional and structural regularity in HDL descriptions and use it to build topologically regular circuits. This paper concerns only with functional regularity.

Various techniques for extraction of functional regularity have been proposed in the literature [1, 3, 10, 11]. Rao and Kurdahi [11] use a string matching algorithm to find all instances of user-specified templates in the circuit, and then heuristically choose a subset of the set of templates to cover the circuit. The final cover is sensitive to the templates provided by the designer. Corazao *et al.* [3] also assume that a template library is provided, but they generate all complete as well as partial instances of a given template in the circuit. Other approaches by Nijssen *et al.* [10] and Arikati *et al.* [1] choose small logic components, such as latches, as templates, and then grow them to obtain bigger templates. These approaches are highly dependent on the initial choice of templates. The problem of finding all instances of a given template is similar to matching in technology mapping, where the input circuit is covered by cells (templates) from a given library [8]. We found that all prior techniques address the problem of covering a circuit by templates, where the templates are assumed to be either provided by the designer, e.g. as a library, or generated in an ad-hoc manner. None of these techniques deal with the systematic generation of a set of templates for a circuit. From our experiments, formulation of a good set of templates is crucial since: *(a)* it allows tradeoffs among multiple criteria, such as area, timing and power, and *(b)* it allows user to build multi-technology designs, such as 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. Our approach then chooses a subset of this set of templates to cover the circuit. 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. There has been no prior attempt at generating such a large set of templates, which is key to obtain a wide range of efficient covers. 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. We identify the structure of regular circuits of unknown description, since a template library need not be specified, unlike prior techniques. We will demonstrate the effectiveness of our approach by identifying regularity in several circuits, including some ISCAS benchmarks whose high-level structure have been identified earlier [7, 13] to help in various CAD applications, such as high-level test generation [7], hierarchical timing analysis [13] and FPGA technology mapping [2].

The rest of the paper is organized as follows. Section 2 formulates the regularity extraction problem in terms of template generation and then circuit covering by templates.

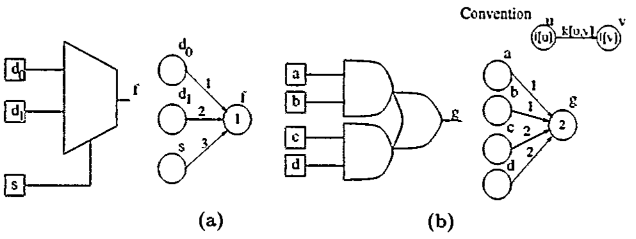

Figure 2: Representing the logic functions of circuit components in $G$: (a) 2-to-1 mux; (b) AND-OR gate.

The complexity of template generation is discussed in Section 3. Section 4 discusses the algorithm for generation of templates with a tree structure. Section 5 extends it to a special class of multi-output templates. The heuristic technique of circuit covering is discussed in Section 6. We present results on benchmark circuits in Section 7. Section 8 concludes with some future extensions.

## 2 Problem formulation

The input to regularity extraction is a circuit $C$ composed of logic components that can be either logic gates or arithmetic operators. $C$ 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 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 $l : L \rightarrow \{1, .., l_O\}$, where $l_O$ 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 multiplexer. Similarly, we associate an index $k : E \rightarrow \{1, .., k_O\}$ 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 2*a* shows a multiplexer whose input edges have all distinct indices, while the AND-OR gate of Fig. 2*b* has four edges assigned to only two indices.

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 such subgraphs *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 $\phi$ between $V_i$ and $V_j$, (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*. Any set $S$ of subgraphs of $G$ 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 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 is stated below.
**Regularity Extraction Problem:** Given a circuit represented by a graph $G$, 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 $\sum_{i=1}^{m} area[S_i]$ of the templates are maximized. □

Maximizing the number of subgraphs will reduce the effort needed to design the circuit, while maximizing the area of templates will reduce the overall area and delay by facilitating better optimization during technology mapping and layout. The above two objectives are conflicting, since a large template usually has only a few subgraphs.

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 problem complexity 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 subproblems below.

**Template Generation Problem:** Given a circuit represented by a graph $G$, generate the complete set of templates where each template has at least two subgraphs. □

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

As mentioned earlier, prior techniques do not address the template generation problem due to its high complexity. The graph covering problem is similar to the binate-covering problem [4], which has been well studied in various CAD areas, including regularity extraction [11].

## 3 Complexity of template generation

The problem of generating all templates of $G$ is similar to enumerating the equivalence classes of $G$ under isomorphism, which is inherently difficult. We now present a few practical assumptions, 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 templates will lead to efficient covers for various datapath circuits.

First, we propose the following assumption, since we would like to extract regularity to the maximum extent.

**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 $S$. □

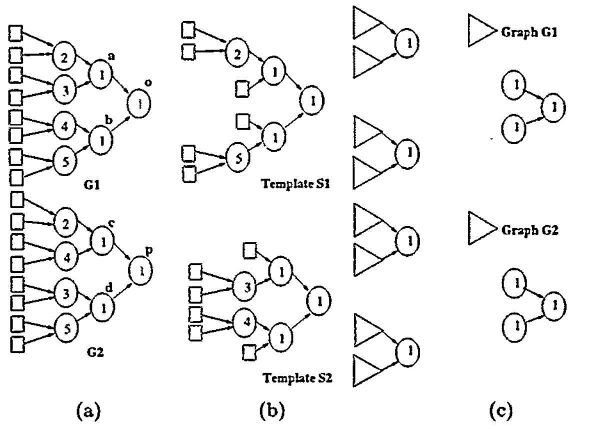

Figure 3: (a) The graph $G'$; (b) its two templates obtained by permuting the incoming edges of the nodes; (c) the graph $G$ with the number of templates given by $O(2^V)$.

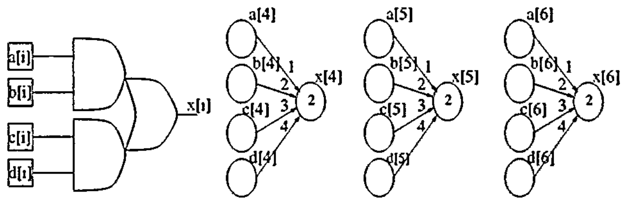

Figure 4: Representing the HDL assignment "`for i = 4 to 6 {x[i] = a[i]·b[i]+c[i]·d[i]}`" in $G$. Note that the edge indices are different from those in Fig. 2*b* as a result of Assumption 2.

The number of templates can be $O(2^V)$ even after considering Assumption 1. Consider the graph $G'$ of Fig. 3*a* composed of two unconnected trees, where the incoming edges of every node have the same index. It has two templates shown in Fig. 3*b*. Now, consider the graph $G$ of Fig. 3*c* 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. 3*b*.

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. 3*b*) as a template for the graph of Fig. 3*a*. As a result, the graph $G'$ of Fig. 3*c* also has a single template. The justification for the above assumption is that $G$ is constructed from an HDL description of $C$, which ensures that nodes 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 three nodes which are transformed identically in building $G$; see Fig. 4. Therefore, the above assumption does not rule out the regularity

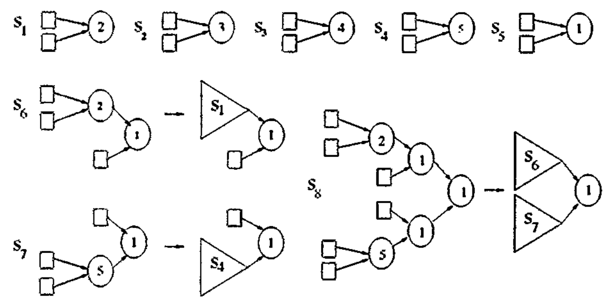

Figure 5: The tree templates for the graph of Fig. 3*a* generated by the algorithm of Fig. 6. $S_8$ is compactly represented by *root_fn*[8] = 1, *children_templates* = $\{S_6, S_7\}$ and *root_nodes* = $\{o, p\}$.

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. It can be shown that the number of tree templates is reduced to within $V^2$ under these two assumptions, which makes the enumeration of such templates practical. We will analyze the complexity for the case where the fanin of the nodes in $G$ is 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*[$i$] = $\{S_1, \ldots, S_f\}$ to which the subgraphs rooted at the $f$ fanin nodes of the root node belong to. For example, Fig. 5 illustrates the templates of the graph $G'$ shown earlier in Fig. 3*a*. The template $S_8$ can be precisely defined by *root_fn*[8] = 1 and *children_templates*[8] = $\{S_6, S_7\}$. 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. 5*b*, *root_nodes*[8] = $\{o, p\}$. It can be shown 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*[$i$].

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*}. The template generation algorithm is presented in Fig. 6. We explain the algorithm using the example of Fig. 5. 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 $S_T$ 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. 5*a*, first the trivial templates $S_1, \ldots, S_4$ are generated. Then, from the remaining nodes $\{a, b, c, d, o, p\}$ (Fig. 3*a*), $S_5$ is generated by comparing $a$ and $b$, and $S_6$ is generated by

```
/* A tree template $S_i$ is completely defined by
(i) root_fn[i]; (ii) children_templates[i] — list of
children templates; (iii) root_nodes[n] — list of
root nodes of subgraphs of $S_i$ */

Generate_Templates($G(V, E)$)
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 no. 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, is the index of template to
which equivalent subgraphs rooted at $v_i$ and $v_j$ belong */
for $i = 1$ to $N$
  for $j = i + 1$ to $N$
    $m := m + 1$;  /*new template to be stored in $S_m$*/
    $S_m$ := Largest_Template($v_i$, $v_j$);
    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 */
        $S_T := S_T \cup \{S_m\}$; /* $S_T$ remains sorted */
      else
        root_nodes[$k$] := root_nodes[$k$] $\cup \{v_i, v_j\}$;
        $m := m - 1$;
return $S_T$;
end


/* generates largest equivalent trees rooted at $u$ and $v$ */
Largest_Template($u$, $v$)
if $l[u] \neq l[v]$ /*$u$ and $v$ have different logic functions*/
  return $\emptyset$;
else
  root_fn[$m$] := $l[u]$; /* setting fields of template $S_m$*/
  for $i = 1$ to $f$ do
  /* $u(v)$ have $f$ fanin nodes $\{u_1, \ldots, u_f\}$ ($\{v_1, \ldots, v_f\}$) */
    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 two subgraphs */
  return $S_m$;


/* performs a binary search on $S_T = \{S_i, \ldots, S_j\}$ */
Find_Equivalent_Template($S_m$, $S_T$)
if $S_T = \emptyset$
  return $m$
if key[$m$] < key[$\frac{i+j}{2}$]   /* check 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 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 6: Algorithm for generating the complete set of tree templates of $G$ under Assumptions 1 and 2.

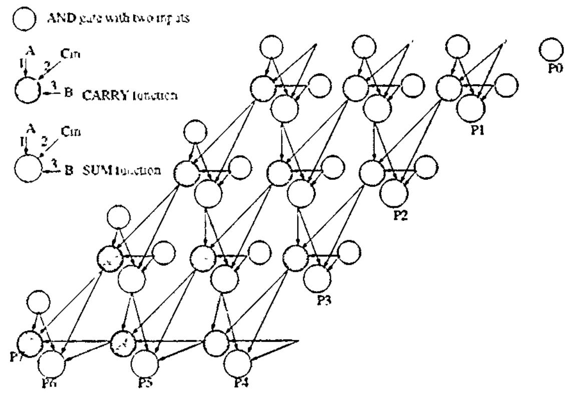
Figure 7: The graph of the 4 × 4 multiplier of Fig 1.

comparing $a$ and $c$. The template obtained by comparing $a$ and $d$ is found to be equivalent to $S_5$, so $d$ 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* $= \{a, b, c, d, o, p\}$.

*Largest_Template* takes a constant time for bounded-fanin graphs. Binary search on $S_T$ (lines 31-38) as well as insertion of $S_m$ in $S_T$ (line 15) take $O(logV)$ time, both of which are called for every node-pair. Thus, the overall time complexity is $O(V^2.logV)$. We store *root_fn* and *children_templates* for every template, which requires a memory of $O(V^2)$. The storage required for subgraphs is also $O(V^2)$, since a subgraph is stored just as its root node. Thus, the overall storage complexity is $O(V^2)$.

## 5 Multi-output templates

The template generation algorithm of Fig. 6 gives excellent covers for datapath circuits composed of sparsely interconnected subcircuits, but it might not perform well for circuits with a high number of multiple-fanout nodes. If we apply this algorithm to the multiplier of Fig. 1, also shown as a graph in Fig. 7, then three trivial tree templates — AND gate, CARRY and SUM functions, are obtained. We now extend the algorithm for tree templates to a special class of 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). For example, the two subgraphs shown in Fig. 8*a* of the graph of Fig. 7 are single-PO graphs with $P5$ and $P4$ as the respective principal outputs. Single-PO graphs have several interesting properties. They can have internal reconvergence as well as cycles, and can have any number of outputs, as opposed to trees. The main advantage of using single-PO subgraphs is that despite their complex structure, the number of such subgraphs of $G$ under the Assumptions 1 and 2 is also restricted to $V^2$, provided the subgraphs satisfy the convex property that if $u, v \in V(G_i)$, then every node $w$ on a path from $u$ to $v$ also belongs to $V(G_i)$.

A tree template was earlier represented by a list of children templates which are non-overlapping. However, the children templates can overlap in single-PO templates. Figure 8c shows the template $S_3$ with the two subgraphs of Fig. 8*a-b*. $S_3$ has two children templates, $S_1$ and $S_2$, which have overlapping nodes, such as c1 of subgraph $G_{P5}$ and c2 of $G_{P4}$. Therefore, $S_3$ cannot be completely specified just by the list of its children templates. Instead, every template has to be specified individually. We store the nodes of a subgraph $G_u$ by a list *nodelist* using the 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 $S_1$ rooted at node $a1$ has *nodelist* $= \{a1, b1, c1, d1, e1, f1, g1\}$. With every node in *nodelist*, we store its fanin and fanout links as well. Thus, memory required to store a subgraph is $O(V)$ for bounded-fanin graphs.

We replace the two functions in Fig. 6 by the corresponding functions in Fig. 9 in order to generate the complete set of single-PO templates. We explain these functions using the example of Fig. 8. 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 subgraphs, $G_{h_1}$ and $G_{h_2}$. The nodelists of $G_{a_1}$ and $G_{h_1}$ ($G_{a_2}$ and $G_{h_2}$) are combined to obtain the nodelist of $G_{P5}$ ($G_{P4}$). After lines 07-09, *nodelist*$[G_{P5}] = \{P5, a1, b1, c1, d1, e1, f1, g1, h1, i1, c1, d1, e1, f1\}$ and *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. 8*a*, and hence, it occurs twice in *nodelist*$[G_{P5}]$. We define a list *path*$[w, v]$ (lines 10-12) 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*$[c1, P5] = \{1, 2\}$. 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 14-15). Otherwise, if the two *path* lists are same, but have multiple indices, then the remaining copies of these nodes have to be removed. For example, the second occurrence of the node c1 (c2) in $G_{P5}$ ($G_{P4}$) is deleted. Finally, after line 21, *nodelist*$[G_{P5}] = \{P5, a1, b1, c1, d1, e1, g1, h1, i1\}$ and *nodelist*$[G_{P4}] = \{P4,$

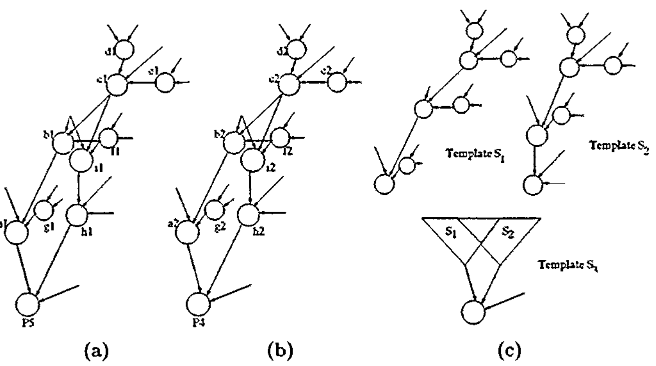
Figure 8: (a) Two functionally-equivalent subgraphs $G_{P5}$ and $G_{P4}$ of the graph of Fig. 7; (b) the two templates with overlapping nodes which are merged to form the template $S_3$.

```
/* generates the largest equivalent single-PO subgraphs
   rooted at $u$ and $v$ */
Largest_Template($u$, $v$)
if $l[u] \neq l[v]$
    return $\emptyset$;
else
    $nodelist[G_u] := \{u\}$; /*root node is the first node*/
    $nodelist[G_v] := \{v\}$;
    for $i = 1$ to $f$ do
  /* $u$ ($v$) have $f$ fanin nodes $\{u_1, \ldots, u_f\}$ ($\{v_1, \ldots, v_f\}$) */
        add $nodelist[G_{u_i}]$ at the end of $nodelist[G_u]$;
        add $nodelist[G_{v_i}]$ at the end of $nodelist[G_v]$;
        for $w_1 \in nodelist[G_{u_i}]$ and $w_2 \in nodelist[G_{v_i}]$
            add $i$ to $path[w_1, u]$;
            add $i$ to $path[w_2, v]$;
        /* there is a path from $w_1$ ($w_2$) to $u$ ($v$) through */
        /* the incoming edge of $u$ ($v$) with index $i$ */
    for $w_1 \in nodelist[G_u]$ and $w_2 \in nodelist[G_v]$
        if $path[w_1, u] \neq path[w_2, v]$
            delete all copies of $w_1 (w_2)$ from $nodelist[G_u(G_v)]$;
        else if $path[w_1, u]$ has more than one element
            delete remaining copies of $w_1 (w_2)$
             from $nodelist[G_u(G_v)]$;
    $S_m := \{G_u, G_v\}$;
    return $S_m$;

Find_Equivalent_Template($S_m$, $S_T$)
for $i = 1$ to $k$
    if $nodelist[S_i] = nodelist[S_m]$
        return $i$;
return $m$;
```

Figure 9: Algorithm to generate the complete set of single-PO templates of $G$ under Assumptions 1 and 2.

$a2, b2, c2, d2, e2, g2, h2, i2\}$. 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.

The procedure *Largest_Template* takes $O(V)$ time, since it constructs two *nodelist*'s and then traverses them twice. *Find_Equivalent_Template* takes $O(V^3)$ time, since it compares two *nodelist*'s at most $V^2$ times. These two functions are called for every node-pair (line 07-08, Fig. 6), resulting in the time complexity of $O(V^5)$. The *nodelist* of every subgraph requires a storage of $O(V)$, resulting in a storage complexity of $O(V^3)$. We found that the execution time for circuits with about 2,000 nodes was no more than a few minutes, since the total number of templates is very small compared to $V^2$. If the number of single-PO templates of $G$ is bounded by $S$, then the overall time and space complexity are given by $O(S^2 \cdot V)$ and $O(S \cdot V)$, respectively.

If designer provides a template $G_T$, we can generate all its complete as well as partial matches in the input graph $G$ by calling the function *Largest_Template* (line 10, Fig. 6) for every node-pair $(v_i, v_j)$, where $v_i$ and $v_j$ belong to $G$ and $G_T$, respectively. This feature allows the designer to control the extraction approach and improve the circuit cover as desired.

## 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 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 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 correlates to its control logic.)

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

1. *Largest-Fit-First (LFF)* heuristic: Select the template $S_i$ with the the maximum area $area[S_i]$.
2. *Most-frequent-Fit-First (MFF)* heuristic: Select $S_i$ 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 $4 \times 4$ multiplier of Fig. 7 obtained using the LFF heuristic contains six templates, where the largest template shown in Fig. 8 covers more than half of the circuit. (The cover of two templates shown in Fig. 1 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.

## 7 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 Berkeley logic interchange format (*blif*), from which $G$ can be constructed in a straightforward manner. We extract the regularity for a variety of circuits, including adders, $74X$ series circuits [12] and ISCAS benchmark circuits. The ISCAS benchmarks are already described in the blif format. We have written input 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.

We analyze interesting covers for several circuits, and then summarize the results for the complete set of circuits.
**Ripple-carry function:** Figure 10 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 $\{S1(2)\}$, $\{S2(4)\}$, $\{S3(8)\}$, and $\{S4(16)\}$, where $\{Si(n_i)\}$

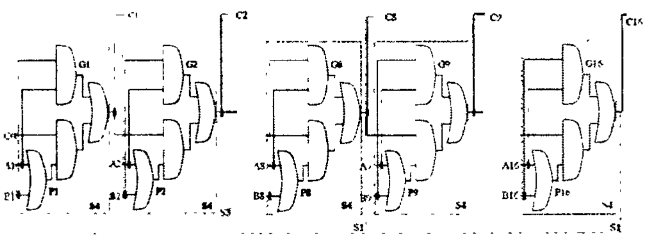
Figure 10: A 16-bit ripple carry function illustrating a hierarchy of templates.

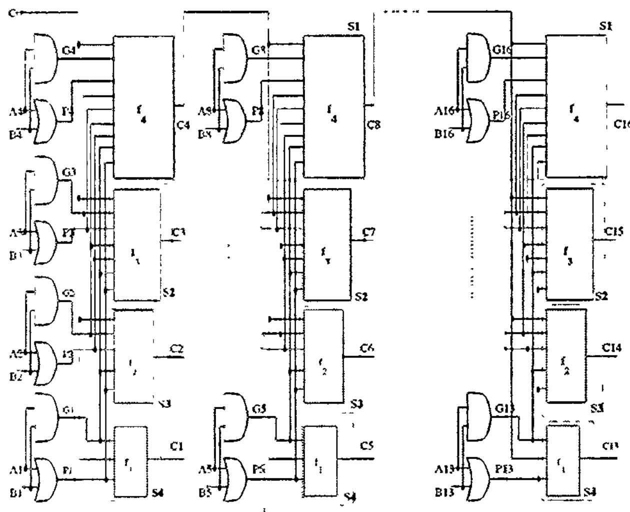
Figure 11: A 16-bit carry-lookahead circuit; most of its nodes are covered by the largest template $S1$.

implies a cover of $n_i$ instances of template $Si$. Thus, our technique can generate a hierarchy of template covers.

**Carry-lookahead function**: Figure 11 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 $\{S1(2), S2(4), S3(4), S4(4)\}$ shown in Fig. 11. Further, $S1$ can be shown to be composed of just one template with two instances.

74181 4-bit **ALU**: This ALU of Fig. 12 [7] is found to have a single-PO template $S1$ with four instances which cover most of the circuit, except the carry-lookahead logic.

7485 **magnitude comparator**: The gate-level realization of 74L85 magnitude comparator [12, 7] is composed of two

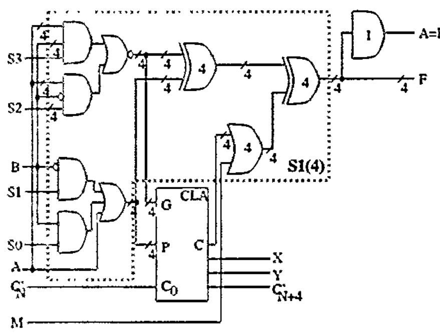

Figure 12: The 74181 ALU has a single template with four instances.

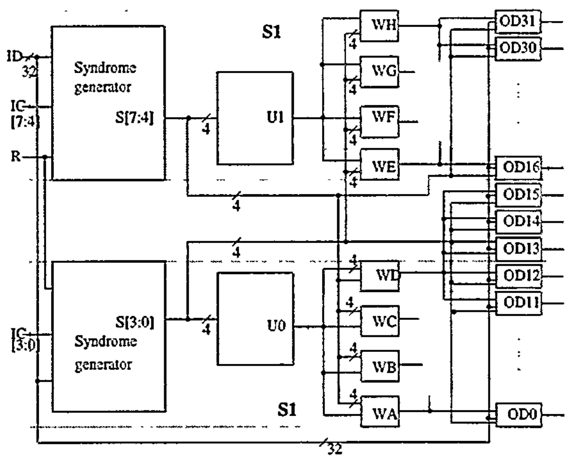

Figure 13: The largest template of the c499 ISCAS circuit covers the entire syndrome generator logic and a part of the remaining error correction logic.

carry-lookahead modules, which are identified as expected.

**c499 (c1355)**: This ISCAS-85 benchmark circuit of Fig. 13, described in [7], 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 some 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 four NAND gates. We get the same largest template as for c499, which proves that our algorithm effectively handles internal reconvergence in the circuit.

**c2670**: This ISCAS benchmark is an ALU with two identical comparator subcircuits [13] apparently used for fault-tolerant reasons. As expected, we are able to identify two instances of the comparator of 12-bit inputs.

Finally, we summarize the results obtained for above set of circuits. Table I gives the covers obtained by applying the covering heuristics on the sets of tree and single-PO templates. Every node is assigned a unit area. The quality of covers can be compared using the area of the largest template and a measure called *regularity index*, defined by 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 $\sum_{i=1}^{m} |S_i| \cdot area[S_i]$. Assuming that a template is synthesized only once for all its subgraphs, a small regularity index implies that a low effort is needed during synthesis and layout stages, while a large template implies that a better optimization can be achieved during synthesis and layout. 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 in the largest template. On the other hand, covers obtained using the MFF heuristic have a small regularity index as well as small templates. Figure 14 shows the wide variation in the regularity indices for the covers of some benchmark circuits. In fact, covers with intermediate values of regularity index can be obtained by using a combination of LFF and MFF heuristics, or other covering heuristics.

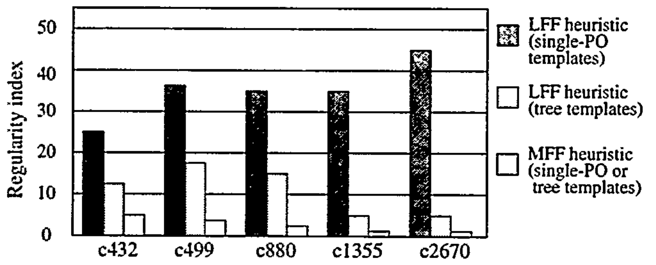

Figure 14: Regularity indices of the covers for some ISCAS benchmarks.

## 8 Conclusions

We have presented a comprehensive approach to extract regularity inherent in the behavioral or structural HDL descriptions of datapath circuits. Identifying regularity would significantly reduce the design effort in subsequent technology mapping and layout stages. Our approach reduces the problem complexity by first generating a set of templates and then selecting its subset to cover the circuit. The major contributions of this paper are the novel algorithms developed to generate two special classes of templates — tree templates and single-PO templates. Our algorithm generates the complete set of these two classes of templates under a few practical assumptions, which is key to achieving a range of efficient covers. We have obtained a variety of covers for several benchmark circuits by using two different covering heuristics. We have also demonstrated that these covers help in understanding the underlying structure of the circuits. A hierarchical representation of circuit regularity can also be obtained by recursive application of our approach.

A useful extension to our extraction approach is to generate templates with multiple outputs, which would lead to more efficient covers, such as the cover of just two templates for the 4 × 4 multiplier (Fig. 1). However, the number of such templates is not restricted by $V^2$ under the two assumptions presented in this paper. Our approach explicitly enumerates the templates, 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] Arikati, S. R., and Varadarajan, R. A signature based approach to regularity extraction. In *Proc. Int'l Conf. on CAD* (Nov. 1997), pp. 542–545.

[2] Chowdhary, A., and Hayes, J. P. Technology mapping for field-programmable gate arrays using integer programming. In *Proc. Int'l Conf. on CAD* (Nov. 1995), pp. 346–352.

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

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

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

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

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

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

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

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

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

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

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

| | No. | LFF heuristic (tree templates) | | | MFF heuristic (tree or single-PO templates) | | | LFF heuristic (single-PO templates) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ckt. | of gates | # templates (subgraphs) | Largest template | Reg. index | # templates (subgraphs) | Largest template | Reg. index | # templates (subgraphs) | Largest template | Reg. index |
| ripple-carry | 64 | 1(16) | 6.3 | 6.3 | 2(64) | 1.3 | 3.1 | 1(2) | 50 | 50 |
| 16-bit CLA | 48 | 6(40) | 6 | 17 | 6(48) | 2 | 13 | 4(14) | 38 | 44 |
| 74181 | 41 | 4(17) | 7.3 | 33 | 5(21) | 7.3 | 32 | 2(5) | 22 | 32 |
| 7485 | 15 | 3(7) | 33 | 26.7 | 4(15) | 6.7 | 26.7 | 3(7) | 33 | 26.7 |
| mult. | 40 | 4(40) | 2.5 | 10 | 4(40) | 2.5 | 10 | 6(16) | 25 | 45 |
| c432 | 160 | 9(89) | 3.1 | 11.9 | 7(159) | 0.6 | 5 | 9(58) | 7.5 | 24.4 |
| c499 | 202 | 7(66) | 8.5 | 17 | 6(202) | 0.5 | 3 | 6(42) | 29.7 | 36.1 |
| c880 | 383 | 18(178) | 3.6 | 15.1 | 9(383) | 0.3 | 2.3 | 19(127) | 12.3 | 35.2 |
| c1355 | 546 | 8(298) | 3.1 | 5.1 | 7(546) | 0.2 | 1.3 | 7(74) | 31.3 | 35.5 |
| c1908 | 880 | 18(425) | 0.8 | 5.2 | 12(879) | 0.1 | 1.5 | 27(171) | 5 | 44 |
| c2670 | 1193 | 23(604) | 2.7 | 11.6 | 12(1193) | 0.1 | 1 | 26(262) | 15.5 | 43.7 |
| c3540 | 1669 | 44(652) | 3.9 | 21.2 | 15(1669) | 0.1 | 0.9 | 38(224) | 28.8 | 43.4 |
| c5315 | 2307 | 37(845) | 0.6 | 7.8 | 15(2307) | 0.1 | 0.6 | 30(264) | 17.3 | 40.4 |

**Table I:** Covers obtained using largest-fit-first and most-frequent-fit-first heuristics. MFF heuristics on tree templates and single-PO templates result in identical covers. The largest template is specified in terms of its area as a percentage of overall circuit area.