FROSTY: A Fast Hierarchy Extractor for Industrial CMOS Circuits*

Lei Yang and C.-J. Richard Shi
Department of Electrical Engineering, University of Washington
Seattle, WA 98195
{yanglei, cjshi}@ee.washington.edu

Abstract: This paper presents FROSTY, a computer program for automatically extracting the hierarchy of a large-scale digital CMOS circuit from its transistor-level netlist description and a library of subcircuits. To handle the complexity of industrial circuits, FROSTY combines traditional structural recognition and pattern matching methods into a two-step extraction process. First, gate structures based on channel-connected-components are recognized from a circuit netlist and library subcircuits. Then annotated graphs representing the connectivity and properties of gate structures are constructed. Comparing to transistor-level netlists, these graphs are much smaller in size, more distinguishable in structure, and are thus more suitable for labeling based pattern matching. An efficient pattern matching algorithm is applied to extract the circuit hierarchy from these condensed circuit graphs. FROSTY has been demonstrated to be orders of magnitude faster than the best known extraction program SubGemini, capable of extracting the entire hierarchy of industrial designs with several hundred thousand transistors in a few minutes on a Sun workstation. Further FROSTY is scale with the size of a circuit.

1. INTRODUCTION

With the rapid development of IC industry, continuously increasing CMOS circuit complexity poses a great challenge to CAD tools, and makes hierarchical expression of circuits very important. There are several levels of abstractions to represent circuits. *Transistor level* describes circuits through a number of transistors and their interconnections. *Gate level* represents logic gates as building blocks to describe circuits. In digital CMOS designs, there is another higher level of circuit, which includes functional blocks consisting of a number of gates, for example: latch, flip-flop, adder, etc. This block level provides a behavioral description of digital integrated circuits.

Automatic recognition of a high level structure from the transistor level netlist of a circuit design is important for many tasks in VLSI design. The early automatic extractors have been developed mostly for functional verification of a circuit layout with respect to its netlist [1][2]. Later, researchers have also shown how to extract higher level structures to speed up the simulation [3]. If the circuit is described at the transistor level, the simulation time is long compared to a behavior block level simulation. This is extremely useful for post-layout simulation before the tapeout. Hierarchy extraction has also been used in formal verification, as well as circuit diagnosis and test generation [4].

Existing extraction algorithms appeared in literature can be classified to two categories: structural recognition and pattern matching. Structural recognition uses rule-based techniques to identify logic gates from sets of channel-connected MOS transistors [5][6]. This category of algorithms is fast but it can only recognize structures with generic rules, for example, static CMOS gates with complementary structures between p-part and n-part. It cannot handle well irregular-structured blocks, for example, DFF, latches, or high-level blocks with structures that are hard to pre-defined as rules.

Pattern matching based extraction algorithms map a flat circuit to a graph, in which transistors are nodes and interconnection wires are edges. Then a subgraph-isomorphism technique is applied to find a one-to-one correspondence between nodes and edges within the two graphs [7][8][9]. However, finding subcircuits in a transistor level ***object*** circuit is a NP-complete problem and is much slower compared to structural recognition. The complexity of pattern matching is determined by two factors [10]. The first factor is how to construct a discriminative graph labeling algorithm. If the model graph vertices carry unique labels that correspond to the labels of the vertex images in an object graph, then subcircuit recognition is a relatively easy task (*more distinguishable in structure*). Unfortunately, the graphs representing directly the transistor-level netlist are hard to be distinguishable, since both the connectivity and the types of transistors a node connected to can be in-distinguishable for most circuit nodes in digital CMOS circuits. As a result, the construction of a discriminative labeling algorithm is a difficult task. The second factor is how to efficiently find subcircuits in the object circuit. The labeling procedure and the recognition strategy are related and both affect the performance of the subcircuit extraction program.

Some efforts have been dedicated to develop good pattern matching algorithms. SubGemini [9] is one of them. It labels part of the nodes with the node's information as well as its neighbors' information and then performs breadth-first-search in the object graph. SubGemini has been demonstrated to be faster than the previous pattern matching algorithms.

In this paper, we propose to combine structural recognition and pattern matching into a two-step extraction process. In the first step, a structural recognition algorithm is applied to transistor level circuits to extract gate level structures. The second step entails generating a directed graph based on the gate level strcutures. Every node in this graph corresponds to one gate, every edge represents one interconnection wire, and the edge direction stands for the signal flow in a circuit. Then the pattern matching process can be applied to recognize the user-defined behavior blocks.

The proposed two-step process has been implemented into a computer program called FROSTY. It is very fast, due to the following reasons. First, compared with the transistor level pattern matching algorithms, the gate level pattern matching algorithm

* This research was supported by DARPA NeoCAD Program under Grant No. N66001-01-8920 and NSF CAREER Award under Grant No. 9985507.

can significantly reduce the size of the graph because every graph node is a gate instead of a transistor. Secondly, pattern matching of directed graphs (gate level) is faster than undirected graphs (transistor level). Finally, every node in the graph can be labeled according to its gate property, including gate type, the gate logic function, fanout number of gate, number of inputs, etc; this can guarantee most of the nodes in the graph have discriminative labels.

Given a transistor level circuit and a used-defined library file, FROSTY recognizes all CMOS gates and user-defined blocks in the library file and outputs a block level netlist. The design of FROSTY is driven by the observation that for industrial CMOS designs, every design company introduces a considerable amount of its own custom behavioral blocks. If those blocks can be extracted from the transistor level circuit, it will be very helpful for design verification, test generation, and fast simulation. In this paper, we use a set of post layout designs from Boeing's digital CMOS ASIC divisions.

The remainder of this paper is organized as follows. An overview of FROSTY is given in Section 2. Section 3 presents the detailed FROSTY algorithm and its time complexity analysis. Section 4 describes performance results on benchmarks from the Boeing Company. Concluding remarks are made in Section 5.

2. PROGRAM OVERVIEW

FROSTY reads in a transistor level digital CMOS netlist (*object circuit*), and a library file in the SPICE format, as shown in Figure 1. The library file contains user specified subcircuit blocks that are to be recognized from the object circuit. After extraction, FROSTY outputs the object circuit description in terms of standard CMOS logic gates and user-defined blocks in the library using VHDL or Verilog formats. FROSTY also produces a header file that contains the functional definitions of all used standard CMOS gates. Together with VHDL or Verilog model descriptions of the library blocks, the extracted block level netlist and the header file can be used for the post-layout simulation of a transistor-level netlist at higher levels.

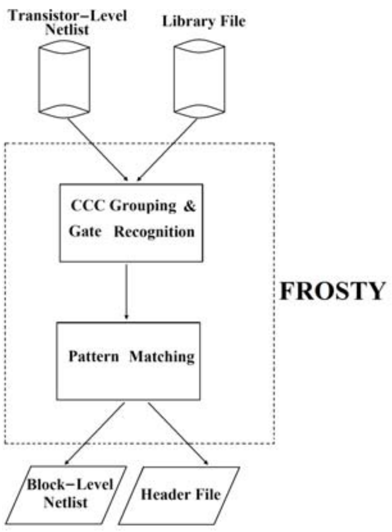


Figure 1. FROSTY flow and architecture.

3. THE TWO-STEP FROSTY ALGORITHM

FROSTY consists of two major steps. The first step is from the transistor level netlist to the gate level, and the second step is from the gate level to the user-defined behavior block level. They are described in 3.1 and 3.2, respectively.

3.1 GATE RECOGNITION

3.1.1 CCC GROUPING

After a netlist is read, the structural recognition algorithm is used for CMOS gate recognition. First, the circuit is divided into Channel-Connected-Components (CCC), which are clusters of transistors connected at the sources and drains. The recognition process starts from every VDD or GND connected transistor, and ends at the connection node between the p-tree and n-tree. All the channel-connected transistors on this path will be grouped together as a p-tree or n-tree. A shared connection node between a p-tree and an n-tree are considered as one CCC. Figure 2 shows the grouping process.

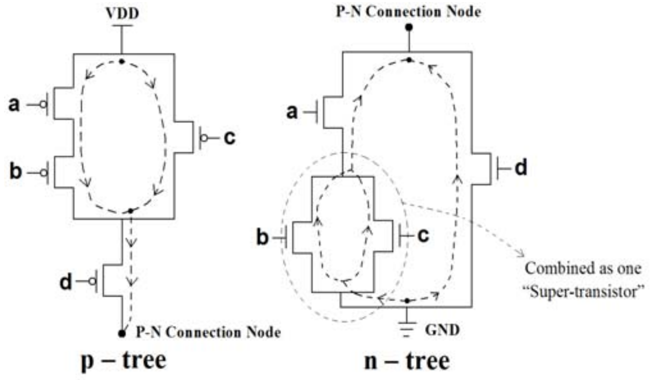


Figure 2. Group the channel-connected blocks.

3.1.2 LOGIC FUNCTION RECOGNITION

For each p-tree and n-tree in the circuit, FROSTY performs the following steps to recognize its logic function. First, FROSTY performs a parallel search inside the p-tree or n-tree. After finding all the transistors that are in parallel, FROSTY replaces them with a "super-transistor", as shown in Figure 2. Then FROSTY performs a serial search, finding all transistors connected in series and replaces them with a "super-transistor". The parallel and series search continues until only one "super-transistor" is left in the tree, at which time the logic function of the tree can be recognized. For example, in Figure 2 the logic function (seen at the P-N connection Node) of this p-tree and n-tree are:

$$f_{p-tree} = (\overline{a} \cdot \overline{b} + \overline{c}) \cdot \overline{d} \qquad f_{n-tree} = a \cdot (b + c) + d$$

Every CCC contains one p-tree and one n-tree. According to the logic function relationships of the p-tree and the n-tree, the CCC can be divided into two types of gates:

- ***Standard Gate***

If a CCC's p-tree and n-tree logic functions have a complementary relationship, the CCC is a standard CMOS gate.

- ***Pseudo Gate***

A CCC whose p-tree and n-tree logic functions are not complementary is called a pseudo gate. Figure 3 is an example of a pseudo gate, one tri-state inverter in a latch. Because the logic functions of the tri-state inverter's p-tree and n-tree do not form a complementary relationship, it is considered to be a pseudo gate.

In static digital CMOS circuits, pseudo gates always exist as part of behavior models (flip-flop, latch…). When all the behavior models have been extracted, there should be no pseudo gates left in the circuit.

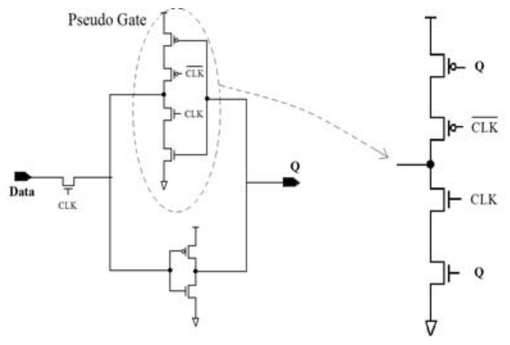


Figure 3. Pseudo CMOS Gate.

Even after structural grouping and logical recognition of the gates, there may be some transistors that cannot be grouped into any CCC. Examples are transistors from pass transistor logic, as shown in Figure 4, which are also recognized by FROSTY.

- ***Pass Transistor Logic***

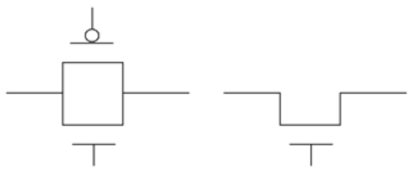


Figure 4. Pass Transistor Logic.

Currently, FROSTY can recognize static digital circuits. After gate recognition is finished, the circuit can be classified into three categories: gates, pseudo gates and pass transistor logic. For dynamic logic circuits, more categories need to be created.

3.2 USER-DEFINED-BLOCK RECOGNITION

In Step 1, the gate level structures are generated for the object circuit and all the blocks in the library. In Step 2, those structures are converted to directed graphs, then a pattern matching algorithm is applied to recognize all the behavior blocks from the object circuit.

3.2.1 DIRECTED GRAPH GENERATION

After Step 1, the circuit has been transformed into a gate-interconnected structure. With each such gate represented by a node, the circuit is then characterized as a graph with both directed edges and un-directed edges.

- ***Directed edge and undirected edge***

A directed edge represents a wire from the output of one gate to the input of another gate, which describes the signal flow in the circuit. For pass transistor logics, it is hard to detect the signal flow. So the wires connected to pass transistor logics can be considered as undirected edges.

- ***Node property***

The characteristics of the gates, such as gate type, logic function of the gate, number of inputs of the gate, gate fanout number, fanout gates properties, etc. are expressed as node properties in the graph. For example, the node that represents the tri-state inverter in Figure 3 has the node property in Table 1.

Table 1. Node property of the pseudo gate in Figure 3.

Gate type		Pseudo gate
Logical function	p-tree	$f = \overline{a} \cdot \overline{b}$
	n-tree	$f = a \cdot b$
Number of inputs		2
Fanout number		2, transmission gate + inverter

Here, let us use the D-flip-flop shown in Figure 5 and transform it from a circuit to a graph. Using the partition and gate recognition algorithm in Step 1, the circuit can be divided into 10 gates. Notice that gate 3 is a pseudo gate made up of two tri-state inverters controlled by the clock signal. Because the two tri-state inverters have the same p-n connection node (the two gates outputs are connected together), the program considers them as one CCC. Gate 10 is a transmission gate, so the edges connected to gate 10 (gate 5 - gate 10, gate 7 - gate 10, gate 8 - gate 10) are undirected edges. Other gates are standard gates.

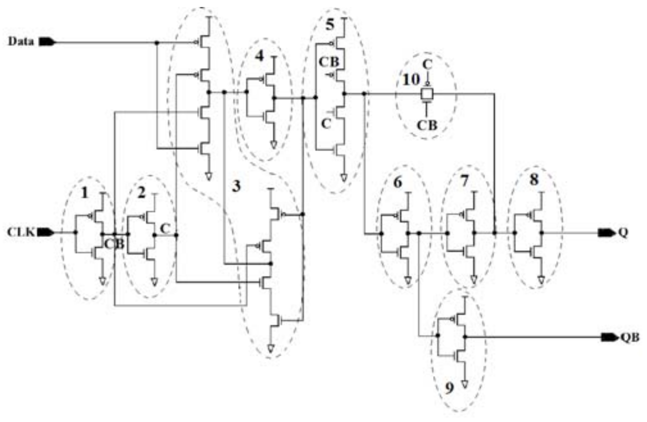
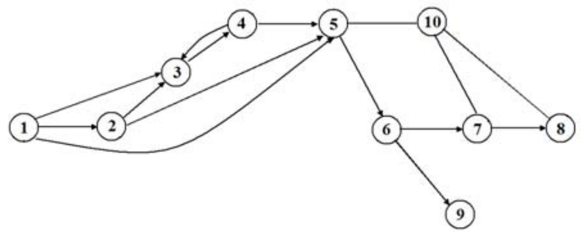


Figure 5. D flip-flop (DFF) circuit after gate recognition.

According to the connection relationships among the gates, a directed graph for this D-flip-flop can be constructed as shown in Figure 6.



$$\begin{tikzcd}
 & & & 4 \arrow[ld, bend left] \arrow[rr] & & 5 \arrow[r, dash] \arrow[rd] & 10 \arrow[rd] \arrow[rrd, dash] & & \\
 & & 3 \arrow[ru] & & & & 6 \arrow[r] \arrow[rd] & 7 \arrow[r] & 8 \\
1 \arrow[rru] \arrow[r] \arrow[rrrrru, bend right] & 2 \arrow[ru] \arrow[rrrru] & & & & & & 9 &
\end{tikzcd}$$

Figure 6. Graph representation of the DFF.

3.2.2 PATTERN MATCHING

After the equivalent graphs are constructed for the object circuit and the library subcircuits, a pattern matching algorithm is employed to locate all of the defined subcircuits in the object circuit.

The basic pattern matching process is illustrated with the following example. Consider the DFF in Figure 5 as a subcircuit block defined in the library file. The object circuit, shown in Figure 7, contains the DFF. The corresponding graph of this circuit is shown in Figure 8. The final pattern matching result is shown in Figure 9.

In order to find the block DFF in the object circuit, we should apply pattern matching to the subgraph (shown in Figure 6) in object graph (shown in Figure 8). This means that for every node in the subgraph, we should find its corresponding node in the object graph. In FROSTY, two node-stacks are set up to hold all the matched nodes, shown in Table 2.

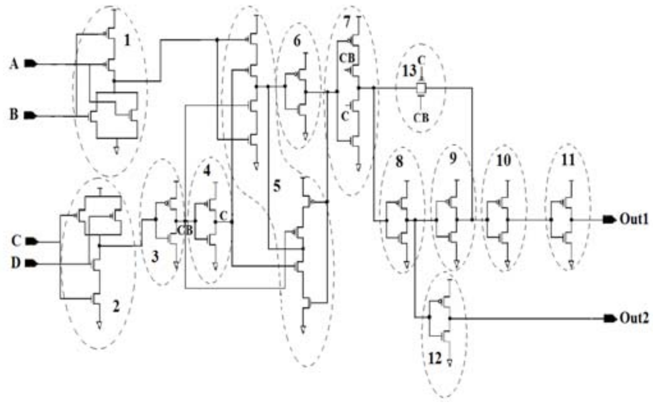


Figure 7. The object circuit after the gate recognition.

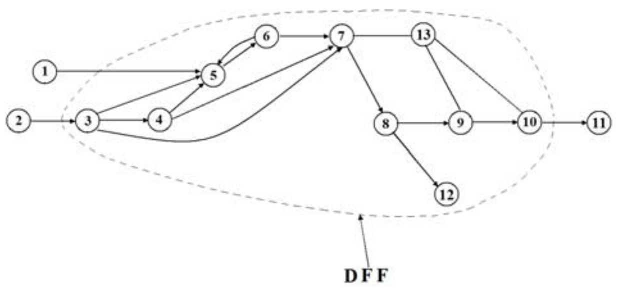


Figure 8. Graph representation of the object circuit.

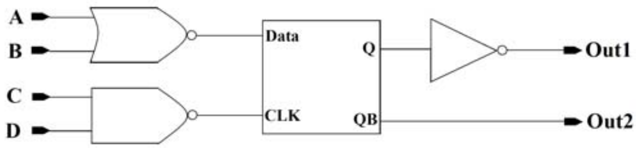


Figure 9. The extracted block-level structure.

The pattern matching process employed in FROSTY is called gradual matching [7]. It begins with a starting node in a block graph, any object graph node with the same "Node Property" as the starting node is identified as a possible location of the subcircuit. Then FROSTY verifies whether there is an actual subcircuit at each possible location.

The first step of the gradual matching process is to locate the starting node in a block graph. From this starting node, all other nodes can be reached through directed or undirected edges. This node is also called "*source node*". In order to locate this source node in the block graph, we pick up a random node first, and then backtrack to its parent nodes. This backtracking is done recursively until a node that has no parent nodes is reached. This node is a "source node". In the DFF block graph in Figure 6, the source node is node 1. However, sometimes we cannot find the source node because the graph may be a ring, as shown in Figure 10. In this case, we can pick any node to be the source node.

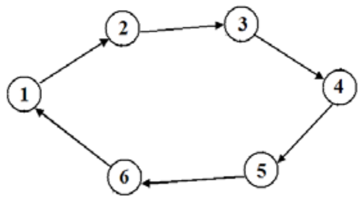


Figure 10. Ring structure of a graph.

After the source node in the block graph is found, all the nodes in the object graph will be searched to locate nodes with similar "Node Properties" as the source node. Any one of these nodes is a possible location of the subcircuit. For every such node (for example, node 3 in Figure 8), the source node and this possible node will be pushed into the block graph node-stack and object graph node-stack, respectively, to begin the gradual matching process; this is Loop 1 in Table 2.

Table 2. Pattern matching process for the example.

Matching process	Block Graph Node-Stack	Object Graph Node-Stack
Loop 1 (source node)	1	3
Loop 2	2	4
	3	5
	5	7
Loop 3	4	6
	6	8
	10	13
Loop 4	9	12
	7	9
	8	10

Then in Loop 2, the matching process starts from this pair of matched nodes in the stacks ($\text{node1}_{\text{block graph}}$-$\text{node3}_{\text{object graph}}$). In the block graph, node1 connects with nodes 2, 3, and 5, while in object graph node3 connects with nodes 4, 5, and 7. After comparing the "Node Properties" of the two series of nodes, we find the following node pairs, $\text{node2}_{\text{block graph}}$-$\text{node4}_{\text{object graph}}$, $\text{node3}_{\text{block graph}}$-$\text{node5}_{\text{object graph}}$, $\text{node5}_{\text{block graph}}$-$\text{node7}_{\text{object graph}}$, have the same "Node Property", respectively. Those pairs are identified to be matched node pairs and pushed into the stacks.

In Loop 3, the matching process starts from all of the newly matched node pairs in the previous loop. For example, from matched node pair $\text{node3}_{\text{block graph}}$-$\text{node5}_{\text{object graph}}$ in the stacks, we can match $\text{node4}_{\text{block graph}}$-$\text{node6}_{\text{object graph}}$; from matched node pair $\text{node5}_{\text{block graph}}$-$\text{node7}_{\text{object graph}}$, we can match $\text{node6}_{\text{block graph}}$-$\text{node8}_{\text{object graph}}$, $\text{node10}_{\text{block graph}}$-$\text{node13}_{\text{object graph}}$; from node pair $\text{node2}_{\text{block graph}}$-$\text{node4}_{\text{object graph}}$, we can match $\text{node5}_{\text{block graph}}$-$\text{node7}_{\text{object graph}}$. All these newly found node pairs are also pushed into the stacks.

The process in Loop 3 is iteratively performed until every block graph node matches its corresponding node in object graph. The whole process is shown in Table 2. If any conflict occurs during the gradual matching process, the process fails, and the node-stacks are emptied for next matching process.

In some cases, there may be more than one "source node" in the graph, as shown in Figure 11. In this example, either node 1 or node 2 can be a "source node". For this case, the program picks the starting node randomly. Suppose that node 1 is chosen here, the searching process will be 1-3-5-6-7-8. Since node 2 and 4 cannot be searched, backtracking will be applied. After checking the stack, unmatched node 4 is connected with matched node 6. So backtracking from node 6-4-2 is performed until all the nodes are matched.

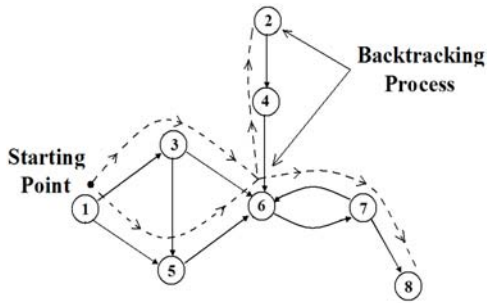


Figure 11. An illustration of the backtracking process.

3.3 Overall Algorithm and Complexity

The entire algorithm of FROSTY is shown in Table 3. In FROSTY, hash tables are used wherever possible due to its linear search time. The complexity of the program is $O(k*n + g1*g2)$, here $k*n$ represents the complexity of Step 1, n is the number of transistors in the circuit, k is an integer number, one can see that the CPU time for Step 1 is linear to n; $g1*g2$ represents the complexity of Step 2, where $g1$ and $g2$ are the number of gates in library file and number of gates in the object circuit after Step 1.

Table 3. Algorithm FROSTY.

```
PREPROCESS FOR LIBRARY:
LOOP: for i = Block_1 : Block_n (in library file) {
        Divide the Block_i into channel-connected-components (CCC)
        Recognize pass transistor logic
        Recognize the logic function of every CCC in the Block_i
      }
INITIALIZATION:
Construct hash tables to store transistors and nodes of the circuit
Divide the circuit into channel-connected-components(CCC)
Recognize transmission gates in the left transistors
Recognize the logic function of every CCC
LOOP: for i = Block_1 : Block_k (in library file)
{
OuterLoop:
  Find "source node" in the Block_i and push it into node-stack
  InnerLoop:
  for j = Node_1 : Node_n (in object graph) {
     if (Node Property (Node_i) = Node Property (Source Node)) {
        Push Source Node and Node_i into the node-stack.
        do{ Searching from matched nodes in node-stack to find new matched nodes, and push them into node-stack. }
        while {conflict happens or all nodes in Block_i has been matched}
     }
  }
}
Output the blocks and gates to a Verilog or VHDL block-level netlist
```

4. EXPERIMENTAL RESULTS

FROSTY was written in C++ and executed on SUN Fire V480 server with 900MHz UltraSparc-III processors and 16GB RAM. Results from applying FROSTY to several industrial circuits from Boeing are presented in this section.

Table 4 shows the statistics of a set of test circuits and the FROSTY CPU time for recognizing all the gates and blocks. Test circuits PSM, PSM-7, PSM-17, PSM-43 are a set of digital CMOS designs in Boeing's "Power Supply Monitor ASIC" on F22 airplanes. Test circuits CEGRP, CEGRP-3, CEGRP-5, CEGRP-7, DFGRP, DFGRP-2, DFGRP-4, DFGRP-6 are a set of digital designs in Being's "Pressure Belt Chip". This chip is used in Boeing's airplanes to determine the structural load on aircraft wings by measuring the pressure distributed on the top and bottom of the wing. These test circuits contain a lot of Boeing defined behavior blocks, such as DFFs, latches, MUXs, adders, etc. Table 5 shows in detail how many blocks are contained in the circuits. For example, in PSM, there are 3 different structures of DFFs and the total number of DFFs is 122. Using the library file provided by Boeing, FROSTY extracts all of the blocks in the library file and outputs a behavioral Verilog/VHDL netlist containing the recognized blocks and the standard CMOS gates that do not belong to any block.

Table 5. Blocks types and numbers in test circuits.

		PSM	CEGRP	DFGRP
DFF	Types	3	10	5
	Number	122	1304	1436
Latch	Types	1	-	-
	Number	15	-	-
Adder	Types	-	3	5
	Number	-	118	646
MUX	Types	1	3	4
	Number	27	1049	508
XNOR	Types	-	2	2
	Number	-	290	74

In Table 4, we compare our results with SubGemini [9]. For each test circuit in Table 4, we try to use SubGemini to extract all of the behavior blocks in Boeing's library. However, SubGemini fails to recognize some blocks. The recognized block numbers and CPU time of SubGemini are listed in columns 8 and 9 in Table 4. The CPU time comparison between FROSTY and SubGemini is shown in Figure 12.

Table 4 also shows the speed of FROSTY. For test circuit CEGRP-7 (729652 transistors), recognizing 74998 gates and 19327 behavior blocks only takes 305.32 seconds. FROSTY is faster than SubGemini for larger circuits and libraries because it performs pattern matching at the gate level. For example, it is 20 times faster than SubGemini for the CEGRP-7 circuit.

The CPU time of FROSTY depends on two factors: 1) the size of a circuit and 2) the number of behavior blocks in the library file. To observe the relationship between circuit sizes and CPU times, we use a set of PSM circuits and perform extraction with the same library file. In Figure 13 the relationship between CPU time and circuit sizes is displayed. We can see that FROSTY CPU time is linear to the size of a circuit with the same library file.

Table 4. Results of FROSTY and SubGemini.

Circuits	#Transistors	# CMOS gates	# Behavior blocks	FROSTY CPU Time (s)			SubGemini Results	
				Setup and Step-1	Step 2	Total	# Extracted Behavior blocks	CPU Time (s)
PSM	4520	1516	164	1.17	0.58	1.75	156	1.4
PSM - 7	31640	10612	1148	7.83	3.97	11.8	1091	13.6
PSM - 17	76840	25772	2788	18.98	9.88	28.86	2651	48.4
PSM - 43	194360	65188	7052	49.9	25.2	75.1	6714	277.1
CEGRP	104236	10714	2761	26.45	15.03	41.48	1419	94.1
CEGRP - 3	312708	32142	8283	81.57	46.83	128.4	4568	1113.5
CEGRP - 5	521180	53570	13805	135.0	78.31	213.35	7093	3178.1
CEGRP - 7	729652	74998	19327	191.63	113.67	305.32	9926	5856.2
DFGRP	119257	10048	2664	30.1	45.8	75.9	2001	108.0
DFGRP - 2	238514	20096	5328	60.1	91.5	151.6	3074	1113.9
DFGRP - 4	477028	40192	10656	121.2	169.96	219.2	5652	2456.9
DFGRP - 6	715542	60288	15984	182.45	233.7	416.2	8802	4317.2

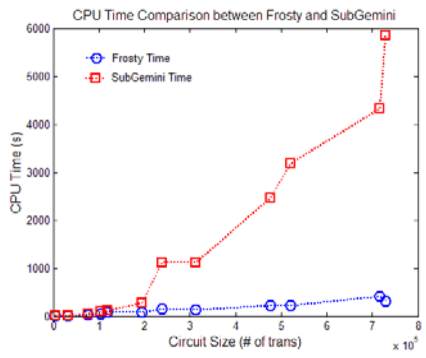


Figure 12. FROSTY and SubGemini CPU time comparison.

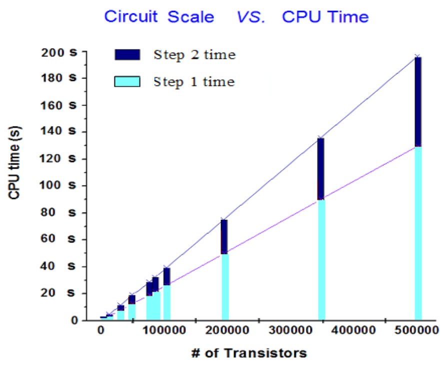


Figure 13. CPU time vs. circuit scale.

5. CONCLUSIONS

This paper presented FROSTY, a computer program for the automatic extraction of circuit hierarchy targeted for the post-layout simulation and verification of library-based large-scale CMOS circuit design. By condensing both the object circuit and library circuits into graphs of blocks and then applying the pattern matching algorithm at the gate level, FROSTY has demonstrated that it can extract an industrial design with seven hundred thousand transistors in less than five minutes on a modern Sun workstation. By representing the extracted hierarchy using high-level descriptions such as VHDL and Verilog, the output netlist can be simulated by any high-level behavioral simulator.

6. REFERENCES

[1] T. Watanabe, M. Endo, and N. Miyahara, "A new automatic logic interconnection verification system for VLSI design", *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. CAD-2, no. 2, pp. 70-76, 1982.

[2] M. S. Abadir and J. Ferguson, "An improved layout verification algorithm (LAVA)", *Proc. European Design Automation Conference*, pp. 391-395, 1990.

[3] T. J. Thatcher and R. A. Saleh, "Automatic partitioning and dynamic mixed-mode simulation", *Proc. IEEE Custom Integrated Circuits Conference*, pp. 12.7.1-12.7.4, 1992.

[4] S. Kundu, "GateMaker: A transistor to gate level model extractor for simulation, automatic test pattern generation and verification", *Proc. of International Test Conference*, pp. 372-381, 1998.

[5] M. Boehner, "LOGEX – an automatic logic extractor from transistor to gate level for CMOS technology", *Proc. IEEE/ACM Design Automation Conference*, pp. 517-522, 1988.

[6] A. Lester, P. Bazargan-Sabet and A. Greiner, "YAGLE, a second generation functional abstractor for CMOS VLSI circuits", *Proc. of the Tenth International Conference on Microelectronics,* pp. 265-268, 1998.

[7] F. Luellau, T. Hoepken and E. Barke, "A technology independent block extraction algorithm", *Proc. IEEE/ACM Design Automation Conference*, pp. 610-615, 1984.

[8] G. Pelz and U. Roettcher, "Pattern matching and refinement hybrid approach to circuit comparison", *IEEE Transactions on Computer-Aided Design*, pp. 264-275, vol. 13, no. 2, Feb. 1994.

[9] M. Ohlrich, C. Ebeling and E. Ginting, "SubGemini: Identifying subcircuits using a fast subgraph isomorphism algorithm", *Proc. IEEE/ACM Design Automation Conference*, pp. 31-37, 1993.

[10] N. Rubanov, "SubIslands: The probabilistic match assignment algorithm for subcircuit recognition", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, pp. 26-38, Jan. 2003.