

DAG-Aware AIG Rewriting

A Fresh Look at Combinational Logic Synthesis

Alan Mishchenko Satrajit Chatterjee Robert Brayton

Department of EECS
University of California, Berkeley
Berkeley, CA 94720

{alanmi, satrajit, brayton}@eecs.berkeley.edu

ABSTRACT

This paper presents a technique for preprocessing combinational logic before technology mapping. The technique is based on the representation of combinational logic using And-Inverter Graphs (AIGs), the networks of two-input ANDs and inverters. The optimization works by alternating DAG-aware AIG rewriting, which reduces area by sharing common logic without increasing delay, and algebraic AIG balancing, which minimizes delay without increasing area. The new technology-independent flow is implemented in a public-domain tool ABC. Experiments on large industrial benchmarks show that the proposed methodology scales to very large designs and is several orders of magnitude faster than SIS and MVSIS while offering comparable or better quality when measured by the quality of the network after mapping.

1 INTRODUCTION

Optimization of multi-level logic networks using logic synthesis [6][7] plays an important role in automated design flow. Logic synthesis is often applied to the network derived by compiling HDLs, such as VHDL or Verilog, before performing technology mapping [14] for standard cells or programmable devices. Other uses of logic synthesis include hardware emulation and design complexity estimation. Traditional combinational logic synthesis, exemplified by SIS [26] and MVSIS [23], applies a sequence of optimization steps, having the goal of removing redundant nodes (*sweep*), finding better logic boundaries (*eliminate, resubstitute*), discovering shared logic (*fast_extract*), and simplifying the node representations (*simplify, full_simplify*).

Traditional synthesis has several drawbacks:

- It often relies on trial-and-error and hand-tuning of the optimization scripts.
- Improvements are measured using the reduction in the number of literals in the factored forms of the node SOPs, while DAG-mappers [9][16] often use cost functions not correlated with the literal counts. It is known that the quality of FPGA mapping, measured by the total number of LUTs and the number of LUT levels on the critical path, hardly correlates with the number of literals in the factored forms.
- It is complicated and hard to implement. An implementation of a robust technology-independent synthesis flow in SIS and MVSIS takes several person-months or even –years, in addition to in-depth knowledge about logic synthesis.
- Even in its robust implementations, with resource limits controlling runtime and memory, traditional synthesis is often slow because it involves time-consuming steps, such as computation of internal don't-cares [25][19].

We propose a new technology-independent logic synthesis flow, based on a sequence of fast local transformations performed

on an AIG representation of the combinational logic. The new flow provides improvements of traditional logic synthesis by addressing the above difficulties. Advantages are summarized as follows:

- While still being heuristic and suboptimal, the new algorithm does not require hand-tuning and trial-and-error.
- Improvements in the complexity of the logic are measured by AIG nodes and levels, in better correspondence with both standard-cell and FPGA mappers [20][21], which use AIGs or similar data structures as subject graphs.
- It is much simpler. A robust implementation reported in this paper took a few person-weeks to conceive and implement.
- It is orders of magnitude faster than the traditional flow, even when compared with its most rugged and robust versions, while the quality is comparable or better when measured by the delay and area of the network after technology mapping.

AIG rewriting is local; however, rewriting is very fast and can be applied to the network many times. For example, performing 10 rewriting passes over a typical network is still at least an order of magnitude faster than running the resource-aware implementation of the traditional flow in MVSIS. By applying rewriting many times, the scope of changes is no longer local. The result is that the cumulative effect of several rewriting passes is often superior to traditional synthesis in terms of quality.

The rest of the paper is organized as follows. Section 2 describes the background. Section 3 discusses the related work. Section 4 discusses the proposed algorithm. Section 5 presents experimental results. Section 6 concludes the paper and outlines future work.

2 BACKGROUND

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires. The terms network, Boolean network, design, and circuit are used interchangeably.

A node has zero or more *fanins*, i.e. nodes that are driving this node, and zero or more *fanouts*, i.e. nodes driven by this node. The *primary inputs* (PIs) of the network are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, flip-flop outputs/inputs are treated as additional PIs/POs.

A *cut* C of node n is a set of nodes of the network, called *leaves*, such that each path from PIs to n passes through at least one leaf. A *trivial cut* of the node is the node itself. A cut is *K-feasible* if the number of nodes in it does not exceed K .

A transitive *fanin (fanout) cone* of node n is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted

towards the path lengths but the PIs are not. The network *depth* is the largest level of an internal node in the network.

The area and delay of an FPGA mapping is measured by the number of LUTs and LUT levels in the resulting network. The delay of a standard cell mapping is computed using pin-to-pin delays of gates assigned to implement a cut.

Two Boolean functions, F and G , belong to the same NPN-class (are *NPN-equivalent*) if F can be derived from G by possibly negating (N) and permuting (P) inputs and negating (N) the output.

Example. Functions $F = ab + c$ and $G = ac + b$ are NPN-equivalent because swapping b and c make them identical. Functions $F = ab + c$ and $G = ab$ are not NPN-equivalent because no amount of permuting and complementing variables can make a 3-variable function equivalent to a 2-variable function.

3 PREVIOUS WORK

The approach presented in this paper was inspired by the recent work on DAG-aware rewriting as a way to compress circuits in formal verification [4]. The paper also contains an excellent overview of recent work on circuit rewriting and functional representations suited for the purpose.

Rewriting in [4] works in two phases. In the first phase, which happens only once when the rewriting package starts, all two-level AIG subgraphs are pre-computed and stored in a hash table by their functionality. The table contains all non-redundant AIG implementations of logic functions with four variables or less.

The second phase involves traversing the AIG of the circuit in a topological order. At each node, its two-level AIG subgraph is found and its Boolean function is computed. This function is used to access the hash table to find equivalent subgraphs. Each of the subgraphs is tried as an implementation of the given node while taking into account logic sharing between the new subgraph nodes and the nodes existing in the circuits. The subgraph leading to the largest improvement in the number of nodes is used to replace the current one in the circuit. The rewriting is *DAG-aware* because it makes use of the logic sharing.

If none of the new subgraphs leads to improvement but there is a new subgraph that keeps the number of nodes constant, it is used for rewriting. Accepting *zero-cost replacements* is a useful heuristic proposed in [4] for changing the AIG structure, which may lead to improvements in subsequent rewriting.

Example. Figure 1 shows three AIG subgraphs for the function $F = abc$. Although the AIG subgraphs differ in the number of nodes, all of them are pre-computed and stored.

Figure 2 shows two instances of rewriting. The upper part of the figure shows the situation when Subgraph 1 is detected in the circuit and replaced by Subgraph 2. The lower part shows two nodes $\text{AND}(a, b)$ and $\text{AND}(a, c)$ already existing in the network. In this case, Subgraph 2 can be replaced by Subgraph 1. In both cases, the network has one node less.

It is instructive to compare this type of rewriting with early logic synthesis systems: SOCRATES [13] and the IBM system [12]. In those systems, rewriting rules were used to replace certain combinations of library gates with other combinations if doing so would improve area or delay. In SOCRATES, an expert system was used to manage the application of these rules – which ones to fire and when. The expert system also allowed uphill moves (moves which increased delay or area) in order to escape local optima. SOCRATES had 163 rules – based on intuition of human designers and observing the results from many examples.

Besides having more rules, and using an automated analysis to derive those rules, our rewriting differs from in several ways. First, we decide the applicability of a set of rules by functional

matching, instead of structural matching. This recognizes that the actual combination of gates is not as important as the function they compute. Second, the DAG-aware rewriting means that a smaller set of gates can be potentially replaced by a larger set of gates, provided most of those gates are already present in the circuit (as in the second example in Figure 2). This reduces the need for a global common sub-expression extraction technique, such as *fast_extract* [24] in SIS [26]. Both SOCRATES and the IBM system also featured dedicated global common subexpression extraction algorithms in addition to rule-based rewriting. Finally, these systems worked with a netlist of library gate structures while we work with the simpler AIG structure, which leads to very fast computation.

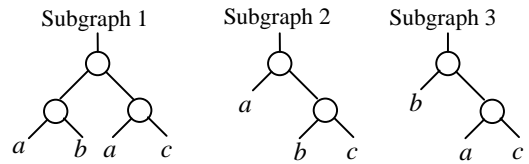


Figure 1. Examples of different AND-Inverter structures for the same function.

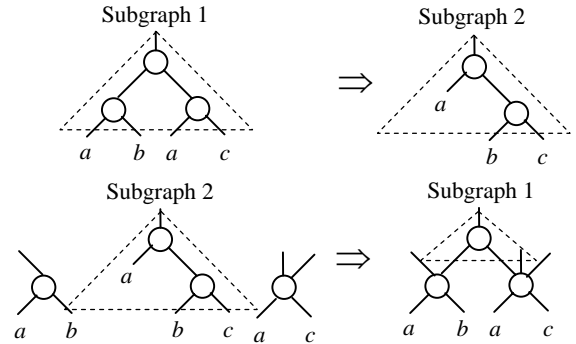


Figure 2. A simple example of AIG rewriting.

4 AIG REWRITING

The AIG rewriting as proposed in this paper has the following improvements, compared to [4]:

- Use of 4-input cuts instead of two-level subgraphs.
- Restricted rewriting to preserve the number of logic levels.
- Variations of AIG rewriting that (a) look at larger subgraphs and (b) attempt to reduce the delay.
- Experimental tune-up for logic synthesis applications.

In the following subsections, we discuss these improvements.

4.1 Using 4-input cuts

The cut computation [10][21] starts at the PIs and proceeds in the topological order to the POs. For a PI, the set of cuts contains only the trivial cut. For an internal node n with two fanins, a and b , the cuts $\Phi(n)$ are computed by *merging* the cuts of a and b :

$$\Phi(n) = \{\{n\}\} \cup \{u \cup v \mid u \in \Phi(a), v \in \Phi(b), |u \cup v| \leq 4\}.$$

Informally, merging two sets of cuts adds the trivial cut of the node to the set of pair-wise unions of cuts belonging to the fanins, while keeping only 4-feasible cuts.

For the purposes of AIG rewriting, we find all 4-feasible cuts of all nodes. For each cut, the Boolean function is computed and its NPN-equivalence class is determined by a table look-up. Manipulation of the 4-variable functions is fast because they are stored using 16-bit bitstrings representing truth tables. Altogether there are 222 equivalence classes of 4-variable functions [22].

Only about 100 of them appear as functions of 4-feasible cuts in any of the available benchmarks. Only about 40 of these have been found experimentally to lead to improvements in rewriting.

We pre-computed all non-redundant AIG implementations of the representative functions of the 40 equivalence classes. All the AIG subgraphs are stored in a shared DAG with approximately 2,000 nodes. The DAG is compiled into our program as an integer array. This noticeably reduced the setup time of the rewriting package because the AIGs do not have to be computed from scratch each time the package starts.

As in previous work, for each cut, all pre-computed subgraphs are considered; however, our subgraphs can span more than two-levels. The subgraphs are derived from the subgraphs of the representative of the cut function’s NPN-equivalence class. The new subgraph that leads to the largest improvement at a node is chosen. If there is no improvement but zero-cost replacements are enabled, a new subgraph that does not increase the number of nodes is used.

Logic sharing between the new subgraphs and nodes already in the network can be checked efficiently using an AIG with reference counters implemented similar to those in a BDD package [5]. In this case, the old subgraph is dereferenced and the number of nodes, whose reference counts became 0, is returned. This is the number of nodes saved by not having the old subgraph in the network. Then, a new subgraph is added to the network while counting the number of new nodes and the nodes whose reference counter changes from 0 to a positive value. This is the increase in the number of nodes due to having the new subgraph in the network. The difference between the former and the latter numbers is the decrease or gain in the number of nodes if the replacement is done.

Using 4-input cuts, instead of the two-level subgraphs extends the scope of rewriting; for each cut, there are on average 3.2 four-feasible cuts, instead of just one two-level subgraph. Additionally, for each cut, we try on average 5 different subgraphs, which often extend beyond the two levels.

4.2 Delay-aware AIG rewriting

Previous AIG rewriting was not concerned with a possible increase in the number of AIG levels because the only goal in formal verification is circuit compression. One of the extensions needed for logic synthesis is to make AIG rewriting delay-aware, without which rewriting tends to increase the delay after mapping, even if area is reduced.

To ensure that the number of logic level does not increase after AIG rewriting, an incremental computation of the slacks at each internal AIG node is supported. Slack at a node is defined as the largest increase of its level that does not lead to an increase in level of any POs. Using the slack, we compute the level of the root node after trying each new subgraph. A subgraph replacement is not accepted if the resulting increase in the number of AIG levels exceeds the slack. After a replacement, slack is incrementally updated.

4.3 Variations of AIG rewriting

The baseline AIG rewriting described in the previous section may occasionally reduce the number of levels, but typically keeps it constant because it is geared for area minimization. Therefore, we developed several new variations of AIG rewriting for processing subgraphs with more than four inputs (Section 4.3.1), and for reducing the number of AIG levels (Section 4.3.2).

4.3.1 AIG refactoring

AIG refactoring is a type of rewriting that works for larger cuts. For each AIG node, in a topological order, one K -feasible cut is computed with $10 \leq K \leq 20$. This cut is selected heuristically such that the number of cut leaves is minimized while the number of reconvergent paths covered by the cut is maximized. The Boolean function of the cut is computed using BDDs and converted into an SOP by the Minato-Morreale algorithm [18][11]. The SOP is factored [6], resulting in an AIG subgraph that is processed using the baseline AIG rewriting. This results in a replacement subgraph for the cut, which is used if the number of nodes and levels does not grow.

When interleaved with the baseline AIG rewriting, refactoring tends to produce deeper permutations of the logic structures improving the effectiveness of rewriting. The runtime of AIG refactoring is comparable to that of AIG rewriting.

4.3.2 AIG balancing

AIG balancing is performed in one linear-time sweep over the network in a topological order. At each node, the associative transform $a(bc) = (ab)c = (ac)b$ is applied to maximally reduce the number of AIG levels. As a result, often the total number of AIG levels is reduced. The experiments have shown that interleaving AIG balancing, which tries to recover delay without increasing area, with AIG rewriting or refactoring, which tries to recover area without increasing delay, is a very good heuristic approach. Balancing takes negligible runtime.

4.4 An example of rewriting

Figure 3 shows a realistic example of AIG rewriting for the small ISCAS benchmark *s27.blif*. Rewriting was applied to the 4-node 3-level subgraph on the left, resulting in a 3-node 3-level subgraph on the right. The POs are on top; the PIs are at the bottom. Triangles stand for the PIs/POs while the rectangles denote latch outputs/inputs. The bubbles are two-input ANDs; the dotted edges have complemented attributes.

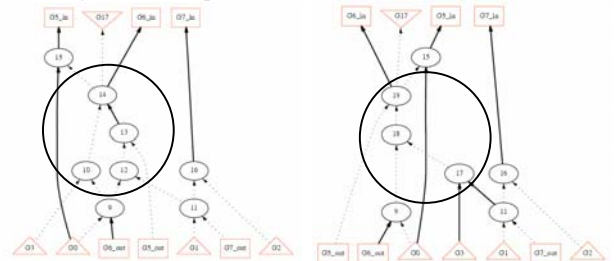


Figure 3. An example of AIG rewriting.

5 EXPERIMENTAL RESULTS

AIG rewriting is implemented in the sequential logic synthesis and verification system, ABC [3], as commands *rewrite*, *refactor*, and *balance*. A rewriting script, *resyn2*, was defined as an alias in the resource file *abc.rc* [3]. This script performs 10 passes over the network as follows: *b*; *rw*; *rf*; *b*; *rw*; *rwz*; *b*; *rfz*; *rwz*; *b*. In the abbreviated notation, *b* (*balance*) stands for AIG balancing, *rw/rf* (*rewrite/refactor*) stands for AIG rewriting/refactoring, and *rwz/rfz* is the same, but with zero-cost replacements allowed.

The *resyn2* script optimizes area under delay constraints. It starts by using balancing to reduce delay upfront as much as possible. Next, rewriting/refactoring and balancing are interleaved. During this, rewriting/refactoring tries to reduce area while not increasing delay. Balancing tries to reduce delay while not increasing area. Zero-cost replacements are enabled later in the script to facilitate creating new rewriting opportunities. This

process in *resyn2* is stopped after three iterations. Generally, this approach seems to work well for a variety of benchmarks.

One of the difficulties in comparing the quality of AIG rewriting with the traditional logic synthesis is the use of different cost functions. Traditionally, improvements are measured by counting the sum total of literals in the factored forms while AIG rewriting looks at the total number of AIG nodes and the maximum number of AIG levels. A direct translation between these two cost function leads to distortions.

Therefore, in Tables 2 and 3, we compare the impact of AIG rewriting to that of logic synthesis in SIS and MVSIS, *after technology mapping*. We used the technology mappers in ABC, for standard cells [8] and FPGAs [21]. The standard cell library *mcnc.genlib* from the SIS distribution was used in the experiments. A load-independent timing model was assumed. Our experiments with a load-independent combinational mapper in an industrial setting confirm that gate sizing and buffering can be done in later stages of the flow in conjunction with placement.

Experiments were performed on several sets of public-domain benchmarks, including industrial circuits from the IWLS 2005 benchmark set [15]. Section 5.1 analyzes the performance of the rewriting script. Section 5.2 compares AIG rewriting with logic synthesis scripts in SIS and MVSIS. Section 5.3 gives detailed statistics for IWLS benchmarks, showing the impact of AIG rewriting on tech-mapping for standard cells and FPGAs.

In all cases, the netlists produced by SIS, MVSIS and ABC were structurally hashed and balanced for minimum delay (Section 4.3.2) in ABC before mapping. The resulting netlists were verified using a SAT-based equivalence checker [20].

5.1 Performance and runtime analysis

The results of the case study are listed in Table 2. The first column lists the benchmarks. The next five columns show the number of primary inputs (PI), primary outputs (PO), latches (Latch), AIG nodes (AND2), and logic levels of two-input AND gates (Lev). The number of gates and logic levels is given for an AIG after structural hashing and algebraic balancing.

The next eight columns show the AIG rewriting statistics after two successive applications of *rwz* to the original benchmarks. The columns show the number of 4-input cuts computed for all internal nodes (“Cuts”), the number of subgraphs tried during rewriting (“Subgrs”), the number of times a rewriting was accepted (“Upds”), and the improvement in the number of AIG nodes after each pass of rewriting.

The data show that the second pass of rewriting leads to smaller but still non-negligible gains in the number of AIG nodes (18%, compared to the first pass). This confirms that the zero-cost replacements restructure logic and open new rewriting possibilities. Without zero-cost replacements, the second pass improves by only 11% (data is not shown in the table). With replacements the first pass reduces the number of nodes by 14%, while without zero-cost replacements, by 12%.

The last three columns of the table show the runtime of logic synthesis in MVSIS (script *mvsis.rugged*), ABC (*resyn2*), and, as a sanity check, the runtime of technology mapping for standard cells in ABC (command *map -s*). The runtimes are reported on a 1.6GHz laptop.

In summary, AIG rewriting as implemented in ABC (*resyn2*) performs 10 passes over the network, gradually improving the area and delay of the AIG. It is also clear that it is much faster than the resource-aware traditional logic synthesis script in MVSIS.

5.2 Comparison with SIS and MVSIS using MCNC benchmarks

In Table 1, we compare the average ratios of improvements achieved by technology mapping for standard cells and FPGAs after running several optimization scripts. The complete set of MCNC benchmarks [27] is used in this experiment. The results of mapping unoptimized circuits are used as the base for comparison (Line 1 of Table 1). The optimization in SIS (*script.rugged*) did not complete on several benchmarks, which were therefore excluded from the comparisons.

The last column shows the average ratios of runtime using AIG rewriting (*resyn2*) as the base. On these relatively small benchmarks, MVSIS is 7 times slower while SIS is slower by several orders of magnitude, depending on the script used. In terms of quality, rewriting tends to produce better area and worse delay than the combination of *script.rugged* followed by *speed up* in SIS. It is likely that a more powerful rewriting that uses larger cuts will outperform SIS in delay while taking only a small fraction of the SIS runtime.

Table 1. Summary of comparison on MCNC benchmarks.

Logic synthesis flow used for optimization	Stand. cells		FPGAs		Runtime
	Area	Delay	Area	Delay	
No optimization	1.00	1.00	1.00	1.00	0.00
ABC (AIG rewriting)	0.87	0.96	0.93	0.98	1.00
MVSIS (<i>mvsis.rugged</i>)	0.91	1.10	0.93	1.03	7.12
SIS (<i>script.delay</i>)	0.94	0.99	0.98	0.97	~100.00
SIS (<i>script.rugged+speed up</i>)	0.94	0.90	0.98	0.94	~1000.00

5.3 Comparison with MVSIS using IWLS 2005 benchmarks

This section compares AIG rewriting in ABC with logic synthesis in MVSIS on the relatively large benchmarks in IWLS 2005. A similar comparison proved impossible for ABC vs. SIS because several key commands in SIS timed out on the largest benchmarks in the set.

The following notation is used in Table 3. Columns “Original”, “MVSIS”, and “ABC” show the results of mapping of the original circuit, the circuit optimized by *mvsis.rugged* in MVSIS, and the same circuit optimized by *resyn2* in ABC, respectively. Two sets of mapping results are reported, one for LUT-based FPGAs and another for standard cells using *mcnc.genlib*.

In summary, the ratios of improvements demonstrate that AIG rewriting on average performs better than traditional synthesis. In particular, the results of technology mapping for FPGAs confirm that literal-based optimization in MVSIS does not reduce area and delay while AIG rewriting reduces both.

It should be pointed out that the original IWLS 2005 benchmarks were optimized somewhat prior to distribution. The delay improvements after AIG rewriting would be more substantial if this were not the case, and also if the AIG balancing was not applied to the original benchmarks before mapping.

6 CONCLUSIONS AND FUTURE WORK

This paper presents AIG rewriting, an innovative technique for combinational logic synthesis. The technique was inspired by research in the field of formal verification where a similar algorithm was used for fast compression of redundant logic circuits before formal verification was attempted. Our experiments show that AIG rewriting often leads to improvements in quality comparable or better than those afforded by the logic synthesis scripts in MVSIS and SIS while being one or two orders of magnitude faster as well as applying to larger examples. The new technique has a potential for replacing the traditional logic

synthesis in the CAD tools. The extreme speed and good quality of the proposed algorithm might make the new synthesis flow useful in a variety of applications from hardware emulation to early estimation of the design complexity.

Future work will include extending the baseline AIG rewriting to use even larger cut sizes. The challenge is search a much larger space of possible replacements while keeping runtime low, which will allow multiple optimization passes.

7 REFERENCES

- [1] P. A. Abdullah, P. Bjesse, and N. Eén, "Symbolic reachability analysis based on SAT-solvers", *Proc. IACAS '00*.
- [2] H. Andersen and H. Huulgaard, "Boolean expression diagrams", *Proc. Symp. Logic in Comp. Science, 1997*.
- [3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. December 2005 Release. <http://www-cad.eecs.berkeley.edu/~alanmi/abc>
- [4] P. Bjesse and A. Boraly, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.
- [5] K. S. Brace, R. L. Rudell, R. E. Bryant, "Efficient implementation of a BDD package", *Proc. DAC '90*, pp. 40-45.
- [6] R. Brayton and C. McMullen, "The decomposition and factorization of Boolean expressions," *Proc. ISCAS '82*, pp. 29-54.
- [7] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, "Multilevel logic synthesis", *Proc. IEEE*, Vol. 78, Feb.1990.
- [8] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05*, pp. 519-526.
- [9] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs", *IEEE Trans. CAD*, vol. 13(1), January 1994, pp. 1-12.
- [10] J. Cong, C. Wu and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," *Proc. FPGA '99*, pp. 29-35.
- [11] O. Coudert, J. C. Madre, H. Fraise, H. Touati, "Implicit prime cover computation: An overview", *Proc. SASIMI '93*.
- [12] A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan, "Logic Synthesis Through Local Transformations," IBM J. Of Research and Development, Vol. 2(4), 1981, pp 272-281.
- [13] D. Gregory, K. Bartlett, A. de Geus, G. D. Hachtel, "SOCRATES: a system for automatically synthesizing and optimizing combinational logic," *Proc. DAC '86*, pp 79-85.
- [14] S. Hassoun and T. Sasao, eds., *Logic synthesis and verification*, Kluwer 2002, Chapter 5, "Technology mapping", pp. 115-140.
- [15] IWLS 2005 Benchmarks. <http://iwls.org/iwls2005/benchmarks.html>
- [16] E. Y. Kukimoto, R. Brayton, P. Sawkar, "Delay-optimal technology mapping by DAG covering", *Proc. DAC '98*, pp. 348-351.
- [17] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. CAD*, Vol. 16(8), 1997, pp. 813-833.
- [18] S. Minato, "Fast generation of irredundant sum-of-products forms from binary decision diagrams", *Proc. SASIMI '92*.
- [19] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization", *Proc. DATE '05*, pp. 418-423.
- [20] A. Mishchenko, S.Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification", *ERL Technical Report*, EECS Dept., UC Berkeley, March 2005.
- [21] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs", *Proc. FPGA '06*.
- [22] S. Muroga, *Logic design and switching theory*, John Wiley & Sons, Inc., New York, NY, 1979
- [23] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System*. UC Berkeley. <http://www-cad.eecs.berkeley.edu/mvsis/>
- [24] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of Boolean expressions", *IEEE Trans. CAD*, vol. 2(6), 1992, pp. 778-793.
- [25] H. Savoj. *Don't cares in multi-level network optimization*. Ph.D. Dissertation, UC Berkeley, May 1992.
- [26] E. Sentovich et al. SIS: A system for sequential circuit synthesis. *Technical Report*, UCB/ERI, M92/41, ERL, Dept. of EECS, UC Berkeley, 1992.
- [27] S. Yang. *Logic synthesis and optimization benchmarks*. Version 3.0. Tech. Report. Microelectronics Center of North Carolina, 1991.

Table 2. IWLS benchmark statistics, rewriting performance, and runtime comparison.

IWLS benchmarks	Network statistics					First iteration (<i>rwz</i>)				Second iteration (<i>rwz</i>)				Runtime, s		
	PI	PO	Latch	AND2	Lev	Cuts	Subgrs	Upds	Gain	Cuts	Subgrs	Upds	Gain	MVSIS	ABC	Map
ac97_ctrl	84	48	2199	14261	11	30583	114770	4105	3242	21081	102688	2108	135	20.46	1.13	0.91
aes_core	259	129	530	21125	21	64314	350417	10849	697	60386	331836	10205	141	175.80	5.54	1.57
des_area	240	64	128	4781	27	28344	121199	1513	330	27322	115917	1426	24	13.29	1.64	0.74
des_perf	234	64	8808	76716	17	394629	1701867	37530	4935	381979	1687585	30212	969	1010.70	33.23	8.87
ethernet	98	115	2235	19654	27	55413	326972	7401	4619	38365	238450	4080	381	50.39	3.81	0.99
i2c	19	14	128	1151	12	2678	16422	415	108	2342	15304	346	21	2.58	0.89	0.26
Mem_ctrl	115	152	1083	15191	28	45670	297941	8257	5416	28759	188528	3762	686	17.20	1.95	0.72
pci_bridge32	162	207	3359	22742	22	67838	331636	7865	3624	53148	279038	5191	155	51.57	3.14	1.46
pci_spoci_ctr	25	13	60	1369	16	3888	23098	578	291	3213	19365	407	101	4.46	1.28	0.21
Sasc	16	12	117	772	8	1413	5002	189	136	1018	4208	158	10	1.22	0.48	0.22
simple_spi	16	12	132	1031	10	2150	10159	328	170	1629	8301	253	21	2.07	0.57	0.25
Spi	47	45	229	3768	31	13511	60660	1267	305	12215	60661	1254	109	11.48	0.92	0.45
ss_pcm	19	9	87	405	7	636	2235	91	10	595	2662	76	2	0.65	0.40	0.14
systemcaes	260	129	670	12279	44	48620	164882	4539	1186	39962	145844	3993	273	28.17	1.87	0.87
systemcdes	132	65	190	2933	23	15246	64309	1045	288	13795	58191	967	52	23.71	1.00	0.38
tv80	14	32	359	9503	42	36843	184125	3494	1340	30106	162391	2894	227	50.07	1.75	0.72
usb_funct	128	121	1746	15670	23	40237	195654	4679	1383	35017	171805	3730	325	66.59	2.29	0.96
usb_phy	15	18	98	456	9	799	4320	145	60	652	3631	115	17	3.59	0.79	0.74
vga_lcd	89	109	17079	126687	19	463129	2887484	51088	34208	292358	2077801	32477	145	1998.27	32.55	8.51
wb_conmax	1130	1416	770	47535	18	159658	978491	15460	1891	149295	993207	16907	862	2323.01	12.38	3.31
wb_dma	217	215	563	4044	22	10722	60208	1052	407	8725	49281	858	101	12.67	1.09	0.30
Ratio						1.00	1.00	1.00	1.00	0.82	0.88	0.79	0.18	24.53	1.00	0.40

Table 3. Effect of AIG rewriting on technology mapping for LUT-based FPGAs ($k = 5$) and standard cells (*mcnc.genlib*).

IWLS benchmarks	Results of mapping into LUTs ($k = 5$)						Results of mapping into mcnc.genlib					
	Original		MVSIS		ABC		Original		MVSIS		ABC	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay	Area	Delay
ac97_ctrl	3391	4	3864	5	3532	3	25961	9.20	23494	13.80	19491	8.30
aes_core	6772	7	7214	8	7180	6	39635	17.70	38855	20.30	38555	17.30
des_area	1581	9	1682	10	1657	8	9562	22.90	8999	25.30	8589	22.00
des_perf	19177	5	23406	5	19163	5	162228	14.10	155708	17.50	145133	14.80
ethernet	4665	9	5170	9	4297	8	33949	22.40	29180	24.40	23142	21.30
i2c	370	4	335	4	320	4	2113	10.10	1678	11.80	1782	10.00
mem_ctrl	4854	9	4551	10	3191	9	25521	23.30	23537	26.50	15865	21.10
pci_bridge32	6150	8	5888	9	5908	7	40322	18.60	35254	20.60	34860	17.70
pci_spoci_ctrl	382	6	336	6	306	5	2426	13.40	1653	19.30	1646	12.10
sasc	207	2	165	3	166	3	1303	7.00	1143	10.70	1178	7.40
simple_spi	245	4	247	5	247	3	1706	9.40	1535	10.20	1446	9.10
spi	1112	8	1180	10	990	8	6977	24.70	6553	25.50	6272	21.80
ss_pcm	120	2	121	3	136	2	805	6.80	788	8.40	841	6.60
systemcaes	2547	9	2770	13	2329	10	21715	28.70	16483	34.60	16533	28.10
systemcdes	783	7	851	8	809	7	6327	20.10	5935	23.00	5451	19.90
tv80	2651	14	2594	14	2467	14	17396	33.70	13939	37.40	13568	33.90
usb_funct	4530	7	4475	8	4030	7	27617	17.80	24386	28.30	23637	19.70
usb_phy	167	3	163	4	162	3	900	8.30	791	10.90	767	7.10
vga_lcd	28458	8	28866	8	29562	7	240071	15.70	169276	16.50	201141	15.50
wb_conmax	16073	7	17165	8	13370	7	82353	15.90	87082	17.60	66124	15.90
wb_dma	1316	8	1283	9	1247	8	7805	18.90	6913	21.40	6675	18.60
Ratio	1.00	1.00	1.01	1.17	0.94	0.97	1.00	1.00	0.88	1.22	0.83	0.97