The Synthesis of Cyclic Combinational Circuits*

Marc D. Riedel
riedel@paradise.caltech.edu

Jehoshua Bruck
bruck@paradise.caltech.edu

California Institute of Technology
Mail Code 136–93, Pasadena, CA 91125

ABSTRACT

Digital circuits are called combinational if they are memoryless: they have outputs that depend only on the current values of the inputs. Combinational circuits are generally thought of as acyclic (i.e., feed-forward) structures. And yet, cyclic circuits can be combinational. Cycles sometimes occur in designs synthesized from high-level descriptions. Feedback in such cases is carefully contrived, typically occurring when functional units are connected in a cyclic topology. Although the premise of cycles in combinational circuits has been accepted, and analysis techniques have been proposed, no one has attempted the synthesis of circuits with feedback at the logic level.

We propose a general methodology for the synthesis of multilevel combinational circuits with cyclic topologies. Our approach is to introduce feedback in the substitution / minimization phase, optimizing a multilevel network description for area. In trials with benchmark circuits, many were optimized significantly, with improvements of up to 30% in the area.

We argue the case for radically rethinking the concept of "combinational" in circuit design: we should no longer think of combinational logic as acyclic in theory or in practice, since nearly all combinational circuits are best designed with cycles.

Categories and Subject Descriptors

B.6.1 [**Hardware**]: LOGIC DESIGN—*combinational logic*

General Terms

Algorithms, Design

Keywords

Logic Synthesis, Combinational Logic, Cycles, Feedback

*Supported in part by the "Alpha Project" at the Center for Genomic Experimentation and Computation, a National Institutes of Health Center of Excellence in Genomic Sciences. The Alpha Project is supported by a grant from the National Human Genome Research Institute (grant no. P50 HG02370).

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

1. INTRODUCTION

The term *combinational* means that a circuit has outputs that depend only on the current values of the inputs (i.e., it is memoryless); the term *sequential* means that a circuit has outputs that may depend upon past as well as current input values (i.e., it has memory).

x_3	x_2	x_1	x_0	Digit
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

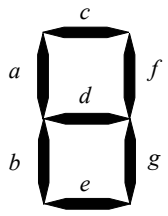

Figure 1: 7-Segment display decoder.

$$
\begin{aligned}
a &= \bar{x}_0\bar{x}_3\bar{c} + \bar{x}_1 c \\
b &= \bar{x}_0(\bar{x}_1\bar{x}_2 + x_1\bar{x}_3) \\
c &= x_0x_2\bar{x}_3 + \bar{x}_2(\bar{x}_1x_3 + e) \\
d &= x_1\bar{x}_2\bar{x}_3 + a(x_2 + x_3) \\
e &= x_0\bar{x}_3 d + b \\
f &= \bar{x}_2 c + \bar{x}_3\bar{e} \\
g &= x_0\bar{x}_3 + a
\end{aligned}
$$

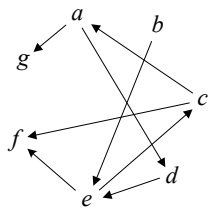

Figure 2: Cyclic network.

Combinational circuits are generally thought of as acyclic structures, and sequential circuits as cyclic structures. In fact, *combinational* and *sequential* are often defined this way. A collection of logic gates connected in an acyclic (i.e., loop-free) topology is clearly combinational. Regardless of the initial values on the wires, once the values of the inputs are fixed, the signals propagate to the outputs. There is a clear correspondence between the electrical behavior of the circuit and the abstract notion of the boolean functions that it implements. The behavior of a circuit with feedback is generally more complicated. Such a circuit may exhibit timing-dependent behavior (as in the case of an `R-S` Latch), and it may be unstable (as in the case of an oscillator).

And yet, cyclic circuits can be combinational. Consider the example shown in Figure 1, ubiquitous in introductory logic design courses: a 7-segment display decoder. The inputs are four bits, x_0, x_1, x_2, x_3, specifying a number from

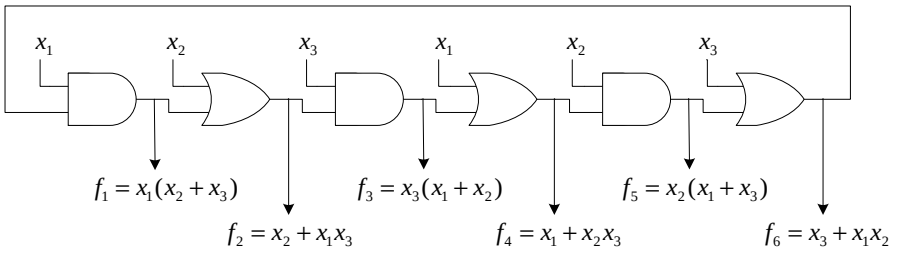

Figure 3: A cyclic combinational circuit due to Rivest [11].

0 to 9. The outputs are 7 bits, $a, b, c, d, e, f,\ g$, specifying which segments to light up in an LED display. A network for the 7-segment display decoder is shown in Figure 2. Note that there is a cycle through nodes a, d, e, and c. Nevertheless, the circuit is combinational. There is feedback in a *topological* sense, but not in an *electrical* sense.

To see this, consider specific input values. For instance, with $x_3 = 0, x_2 = 0, x_1 = 0, x_0 = 1$, the network simplifies to that shown in Figure 4, yielding the correct solution: the digit 1.

$$\begin{array}{lll} a & = c & = 0 \\ b & = 0 & \\ c & = e & = 0 \\ d & = 0 & \\ e & = b + d & = 0 \\ f & = \bar{e} + c & = 1 \\ g & = 1 & \end{array}$$

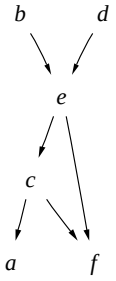

Figure 4: Network of Figure 2 with inputs $x_3 = 0, x_2 = 0, x_1 = 0, x_0 = 1$ (the digit 1).

With $x_3 = 0, x_2 = 1, x_1 = 0, x_0 = 1$, the network simplifies to that shown in Figure 5, yielding the correct solution: the digit 5.

$$\begin{array}{lll} a & = c & = 1 \\ b & = 0 & \\ c & = 1 & \\ d & = a & = 1 \\ e & = b + d & = 1 \\ f & = \bar{e} & = 0 \\ g & = 1 & \end{array}$$

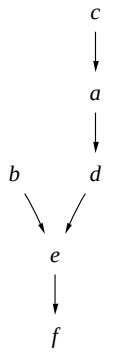

Figure 5: Network of Figure 2 with inputs $x_3 = 0, x_2 = 1, x_1 = 0, x_0 = 1$ (the digit 5).

The reader may verify that the network implements all the digits correctly. The cost of the network, as measured by the literal count, is 34. For comparison, the command `full_simplify` in the Berkeley SIS package [12] yields an acyclic network with a cost of 37.

We argue that introducing loops in the design of combinational circuits is advantageous. The intuition behind this is that with feedback, all nodes can potentially benefit from work done elsewhere; without feedback, nodes at the top of the hierarchy must be constructed from scratch.

1.1 Prior Work

In 1992, Stok observed that cycles sometimes occur in combinational circuits synthesized from high-level designs [18]. In such examples, feedback is carefully contrived, occurring when functional units are connected in a cyclic topology. Recently, Edwards pointed out that cycles arise in circuits synthesized from synchronous languages such as Esterel [5]. Most logic synthesis and verification tools balk when given cyclic designs for combinational logic. Stok's solution to this dilemma is to disallow the creation of cycles in the resource-sharing phase of high-level synthesis; Edwards' approach is to transform cyclic designs into equivalent acyclic ones.

In 1994, Malik addressed the issue of analyzing cyclic combinational circuits [8]. He proved that deciding whether a cyclic circuit is combinational or not is co-NP-complete, and he formulated an analysis algorithm for this task based on ternary-valued simulation. He also addressed the issue of timing analysis and testing for faults [9]. In 1996, Shiple extended Malik's work and set it on a firm theoretical footing [13]. He showed that the class of circuits that Malik's procedure decides to be combinational are precisely those that are well-behaved electrically, according to the up-bounded inertial delay model [4]. He proposed refinements to Malik's algorithm [14] and extended the concept to combinational logic embedded in sequential circuits [15].

Although the premise of cycles in combinational logic has been established, combinational circuits are not designed with feedback in practice. The examples that can be found all have simple and regular feedback structures stemming from high-level constructs [5], [17]. No one has attempted the synthesis of circuits with feedback at the logic level.

As early as 1960, Short argued that permitting cyclic topologies could reduce the size of relay networks [16]. Around 1970, Huffman and Kautz argued that cyclic combinational circuits could have fewer logic gates than acyclic forms [6], [7]. Unfortunately, the examples that they gave, although plausible, are not combinational in a strict sense.

In 1977, Rivest presented a convincing example of a family of cyclic combinational circuits [11]. For any odd integer n greater than 1, the circuit consists of n `AND` gates alternating with n `OR` gates in a single cycle, with n inputs repeated. The circuit for $n = 3$ is shown in Figure 3. Rivest showed that the circuit is combinational and that each gate computes a distinct output function depending on all n variables. Sig-

nificantly, he also proved that this circuit is optimal in terms of the number of fan-in two gates used, and he proved that the smallest acyclic circuit implementing the same $2n$ output functions requires at least $3n-2$ fan-in two gates. Thus, asymptotically, this cyclic circuit is two-thirds the size of any equivalent acyclic form.

1.2 Contributions

Inspired by the work of Rivest, we have generated a variety of cyclic examples with the same property as his circuit: they have provably fewer gates than equivalent acyclic circuits. Most notably, we have found a family of circuits that are asymptotically one-half the size. This work will be presented elsewhere.

In this paper, we explore the topic of cyclic combinational circuit *synthesis* and demonstrate that these are not isolated examples: cyclic topologies are superior to acyclic topologies for a broad range of networks, from randomly generated designs to small- and large-scale networks encountered in practice.

We propose a general methodology for the synthesis of multilevel combinational circuits with cyclic topologies. Our approach is to introduce feedback in the substitution / minimization phase. We have incorporated the technique in the Berkeley SIS package [12].

Our focus in the present work is on optimizing area (as measured by the literal count). In trials, benchmark circuits were optimized significantly with feedback, with improvements of up to 30%. In trials with randomly generated examples, very nearly *all* had cyclic solutions superior to acyclic forms, with improvements averaging up to 15%. In [10], we discuss analysis aspects in the context of synthesis, including timing analysis of our cyclic designs.

1.3 Notation and Definitions

We use the standard notation: addition (+) denotes disjunction (OR), multiplication ($\cdot$) denotes conjunction (AND), and an overline ($\bar{x}$) denotes negation (NOT).

Our model is at the level of abstraction applicable in the technology-independent phase of logic synthesis. Our goal is to construct a network that computes boolean **target functions** $g_i(x_1, \ldots, x_m)$, $1 \leq i \leq n$, of boolean input variables $x_1, \ldots, x_m$. Internally, the network is specified as a collection of nodes. Associated with each node is a **node function** f_i and an **internal variable** y_i, $1 \leq i \leq n$. The node functions depend on internal variables and input variables. A directed edge is drawn from node i to node j iff the node function f_j associated with node j depends on the internal variable y_i associated with node i. A subset of the nodes are designated as **output nodes**. For these, the target functions are the requisite output functions[1].

A network is **combinational** iff it computes unique boolean output values for each boolean input assignment. If there are "don't care" conditions on the inputs, then it is sufficient if the network computes unique boolean values for input assignments in the "care" set.

[1]In our examples, for the sake of readability, we use the same symbol for the node function, the target function, and the associated internal variable: on the left-hand side, a symbol such as f_i refers to a function; on the right-hand side, it refers to the corresponding internal variable.

This computation must hold:

- regardless of the initial state
- and independently of all timing assumptions.

Our **cost measure** for area is the sum of the literals in the factored form of the node expressions (see [3]).

2. SYNTHESIS

The goal in multilevel logic synthesis (also sometimes called random logic synthesis) is to obtain the best multilevel, structured representation of a network. The process typically consists of an iterative application of minimization, decomposition, and restructuring operations [3]. An important operation is **substitution** (also sometimes called "resubstitution"), in which node functions are expressed, or re-expressed, in terms of other node functions as well as of their original inputs. Consider the target functions in Figure 6.

$$\begin{aligned} f_1 &= \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_2(x_1 + x_3) \\ f_2 &= \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1(x_2 + x_3) \\ f_3 &= \bar{x}_3(\bar{x}_1 + \bar{x}_2) + \bar{x}_1 \bar{x}_2 \end{aligned}$$

Figure 6: Target functions for synthesis.

In our implementation, we use the `simplify` command of the Berkeley SIS package to perform substitution/minimization [12]. For instance, substituting f_3 into f_1, we get

$$f_1 = f_3(x_1 + x_2) + \bar{x}_2 x_3.$$

Substituting f_3 into f_2, we get

$$f_2 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{f}_3.$$

Substituting f_2 and f_3 into f_1, we get

$$f_1 = \bar{x}_2 x_3 + \bar{f}_2 f_3.$$

For each target function, we can try substituting different sets of functions. Call such a set a **substitutional set**. Different substitutional sets yield alternative functions of varying cost. In general, augmenting the set of functions available for substitution leaves the cost of the resulting expression unchanged or lowers it. (Strictly speaking, this may not always be the case since the algorithms used are heuristic.)

2.1 Substitutional Orderings

In existing methodologies, a total ordering is enforced among the functions in the substitution phase to ensure that no cycles occur. This choice can influence the cost of the solution. With the ordering shown on the right in Figure 7, substitution yields the network shown on the left with a cost of 14.

Enforcing an ordering is limiting since functions near the top cannot be expressed in terms of very many others (the one at the very top cannot be expressed in terms of *any* others). Dropping this restriction can lower the cost. For instance, if we allow every function to be substituted into every other, we obtain the network shown on the left in Figure 8, with a cost of 12. This network is cyclic, with the dependency shown on the right. It is *not* combinational.

$$
\begin{aligned}
f_1 &= \bar{x}_2 x_3 + \bar{f}_2 f_3 \\
f_2 &= \bar{x}_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{f}_3 \\
f_3 &= \bar{x}_3(\bar{x}_1 + \bar{x}_2) + \bar{x}_1 \bar{x}_2.
\end{aligned}
$$

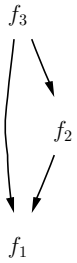

Figure 7: Acyclic substitution order.

$$
\begin{aligned}
f_1 &= \bar{x}_2 x_3 + \bar{f}_2 f_3 \\
f_2 &= x_1 \bar{f}_3 + \bar{x}_3 \bar{f}_1 \\
f_3 &= f_1 \bar{f}_2 + \bar{x}_2 \bar{x}_3
\end{aligned}
$$

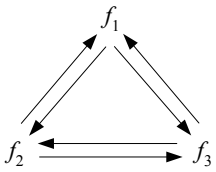

Figure 8: Unordered substitution (*not* combinational).

An essential step in the synthesis process is the analysis for combinationality, a topic previously addressed by Malik [8]. We have formulated a new algorithm for combinationality and timing analysis that can be applied efficiently in the context of synthesis. The details of our approach are presented in [10].

2.2 Branch-and-Bound Algorithms

The goal of the synthesis process is to select a choice of node functions that minimizes the cost while satisfying the condition for combinationality. For each node, we expect the lowest cost expression to be obtained with the full substitutional set (i.e., all other node functions) and the highest cost expression to be obtained with the empty set. For a network with a non-trivial number of nodes, a brute-force exhaustive search is evidently intractable. With n nodes, there are 2^{n-1} substitutional sets for each node, for a total of $n \cdot 2^{n-1}$ possibilities. We describe a branch-and-bound approach, as well as various heuristics.

2.2.1 The "Break-Down" Approach

With this approach, the search is performed *outside* the space of combinational solutions. A branch terminates when it hits a combinational solution. The search begins with a densely connected network, such as that in Figure 8. This initial branch provides a lower bound on the cost. As edges are excluded in the branch-and-bound process, the cost of the network remains unchanged or increases. (Again, since the substitution step is heuristic, this may not be strictly true.) The algorithm:

1. Analyze the current branch for combinationality. If it is combinational, add it to the solution list. If it is not, select a set of edges to exclude based on the analysis.
2. For each edge in the set, create a new branch. Create a node expression, excluding the incident node from the substitutional set. If the cost of the new branch equals or exceeds that of a solution already found, kill the branch.
3. Mark the current branch as "explored"
4. Set the current branch to be the lowest cost unexplored branch.
5. Repeat steps 1 – 4 until the cost goal is met.

A sketch of the algorithm is shown in Figure 9. (This is *not* a complete trace of the search; only the trajectory to the solution is shown.) For the target functions in Figure 6, the algorithm yields a cyclic combinational solution with a cost of 13, shown in Figure 11.

Many ideas immediately suggest themselves for expediting the search heuristically. We can prioritize progress slightly, at the expense of quality (i.e., choose branches that are "closer" to being combinational, according to the details provided by the analysis algorithm). Also, we can limit the density of edges a priori or prune the set of edges before creating new branches.

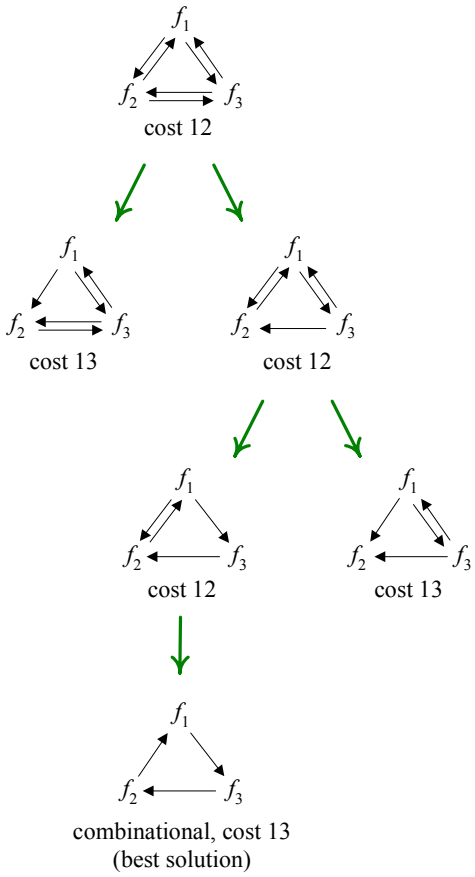

Figure 9: "Break-Down" Search Strategy.

2.2.2 The "Build-Up" Approach

With this approach, the search is performed *inside* the space of combinational solutions. A branch terminates when it hits a non-combinational solution. The search begins with an empty edge set (i.e., the target functions). Edges are added as the substitutional sets of nodes are augmented. As edges are included, the cost of the network remains the same or decreases. The algorithm:

1. Analyze the current branch for combinationality. If it is not combinational discard it. If it is combinational, select a set of edges to include based on the analysis.
2. For each edge in the set, create a new branch. Create a new node expression, including the incident node from the substitution set.

3. Mark the current branch as "explored."
4. Set the current branch to be the lowest cost unexplored branch.
5. Repeat steps 1 – 4 until the cost goal is met.

A sketch of the algorithm is shown in Figure 10. (Again, *not* a complete trace). The algorithm yields the same cyclic combinational solution, shown in Figure 11.

With this method, we cannot prune branches through a lower-bound analysis. However, exploring within the space of combinational solutions ensures that incrementally better solutions are found as the computation proceeds. In fact, as an alternative starting point, we can use an existing acyclic solution. Adding edges reduces the cost, while potentially introducing cycles.

The break-down approach seems to perform best on dense examples, whereas the build-up approach performs better on sparse examples.

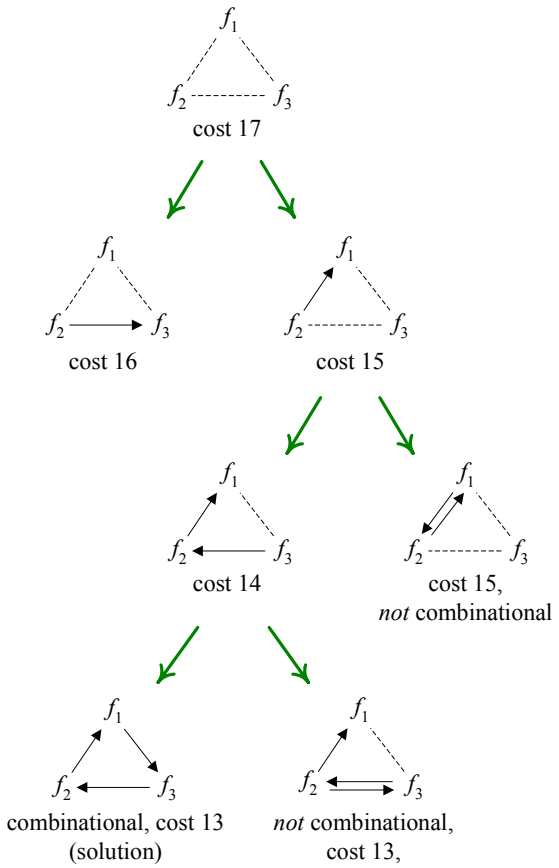

Figure 10: "Build-Up" Search Strategy.

3. RESULTS

The most salient result to report, and the main message of this paper, is that cyclic solutions are not a rarity; they can readily be found for most networks that are not trivially simple or sparse. We have run trials with our program, called CYCLIFY, on a range of randomly generated examples and benchmark circuits. We note that solutions for most of the examples have deeply nested loops, with dozens or even hundreds of cycles.

$$f_1 = \bar{x}_3\bar{f}_2 + \bar{x}_2x_3$$
$$f_2 = \bar{x}_1\bar{x}_2\bar{x}_3 + x_1\bar{f}_3$$
$$f_3 = \bar{x}_1f_1 + \bar{x}_2\bar{x}_3$$

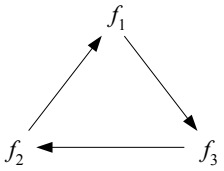

Figure 11: Cyclic solution for target functions in Figure 6.

3.1 Methodology

We present a simple comparison between the cost of cyclic versus acyclic substitutions. The input consists of a collapsed network. The substitution and minimization operation is performed with the `simplify` command in the Berkeley SIS package [12], with parameters: method = `snocomp`, dctype = `all`, filter = `exact`, accept = `fct_lits`. The cost given is that of the resulting network, as measured by the literal count of the nodes expressed in factored form. This is compared to the cost of the network obtained by executing `simplify` directly with the same parameters. The trials were run on an Intel Pentium IV, 1.8 GHz workstation. For the larger circuits, the amount of improvement drops off due to time limits imposed on the search.

3.2 Benchmarks

For benchmark circuits, we used the usual suspects, namely the Espresso [2] and LGSynth93 [1] benchmarks. Examples were selected based on size and suitability (generally, circuits with fewer than 30 inputs and fewer than 30 outputs). For circuits with latches, we extracted the combinational part. In Figure 12, we present those circuits for which cyclic solutions were found. Column 4 gives the improvement, and Column 5 the computation time.

3.3 Randomly Generated Functions

Since randomly generated functions are very dense, they are not generally representative of functions encountered in practice. Nevertheless, it is interesting to examine the performance of the CYCLIFY program on these. We present results from random trials in Figure 13. Each row lists the results of 25 trials. Cyclic solutions were found in nearly all cases (3rd column). The average improvement is given in the 4th column, and the range of improvement in the 5th column.

4. CONCLUSIONS/FURTHER DIRECTIONS

We feel that we have made the case for a paradigm shift in combinational circuit design: we should no longer think of combinational logic as acyclic in theory or in practice, since nearly all combinational circuits are best designed with cycles.

Our focus in the present work is on optimizing area. Our search algorithms, while heuristic in nature, can effectively tackle circuits of sizes that are of practical importance. We note that the implementation of more sophisticated search algorithms is an obvious evolution of the project.

In future work, we will extend the method to the technology mapping phase of logic synthesis, and we will address the topic of optimizing cyclic designs for power and delay.

Espresso Benchmarks				
	`Simplify`	`Cyclify`	Improvement	Time (H:M:S)
p82	104	90	13.5%	00:02:03
t4	109	89	18.3%	00:00:02
dc2	130	123	5.4%	00:01:34
apla	185	131	29.2%	00:00:31
tms	185	158	14.6%	00:01:17
m2	231	207	10.4%	00:06:02
t1	273	206	24.5%	00:21:40
b4	292	281	3.8%	00:09:50
exp	320	260	18.8%	00:33:26
in3	361	333	7.8%	00:22:06
in2	397	291	26.7%	00:00:45
b10	398	359	9.8%	00:08:29
gary	421	404	4.0%	00:18:15
m4	439	412	6.2%	00:07:22
in0	451	434	3.8%	00:05:53
max1024	793	774	2.4%	00:00:29
LGSynth93 Benchmarks				
ex6	85	76	10.6%	00:00:06
inc	116	112	3.4%	00:00:04
bbsse	118	106	10.2%	00:00:08
sse	118	106	10.2%	00:00:10
5xp1	123	109	11.4%	00:00:01
s386	131	113	13.7%	00:00:08
bw	171	163	4.7%	00:15:41
s400	179	165	7.8%	00:02:12
s382	180	165	8.3%	00:02:30
s526n	194	189	2.6%	00:00:29
s526	196	188	4.1%	00:00:25
cse	212	177	16.5%	00:00:05
clip	213	193	9.4%	00:00:01
pma	226	211	6.6%	00:04:30
dk16	248	233	6.0%	00:00:53
s510	260	227	12.7%	00:00:05
ex1	309	276	10.7%	00:09:11
s1	332	322	3.0%	00:03:34
duke2	415	397	4.3%	00:02:58
styr	474	443	6.5%	00:03:24
planet1	550	517	6.0%	05:09:19
planet	555	504	9.2%	02:57:47
s1488	622	589	5.3%	00:47:04
s1494	659	634	3.8%	05:19:41
table3	1287	1175	8.7%	12:39:20
table5	1059	1007	4.9%	14:10:10
s298	2598	2445	5.9%	10:15:03
ex1010	3703	3593	3.0%	10:57:58

Figure 12: Cost (literals in factored form) of Berkeley SIS `Simplify` vs. `Cyclify` for benchmarks.

Randomly Generated Networks				
# In.	# Out.	Cyclic Solns. Found	Avg. Improvement	Range
5	5	100%	8.5%	3% – 17%
5	7	96%	9.1%	0% – 18%
5	10	100%	12.0%	2% – 20%
5	15	100%	13.4%	7% – 23%
5	20	100%	14.2%	8% – 18%
7	10	96%	5.6%	0% – 11%
7	15	88%	3.6%	0% – 10%

Figure 13: Cost improvement (literals in factored form) of `Cyclify` over Berkeley SIS `Simplify` for randomly generated networks (25 trials per row).

5. REFERENCES

[1] Benchmarks from the 1993 Int'l Workshop on Logic Synthesis, available at `http://www.cbl.ncsu.edu/`.

[2] Benchmarks from "Logic Minimization Algorithms for VLSI Synthesis," by R. K. Brayton et al., available at `ftp://ic.eecs.berkeley.edu/`.

[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, "Multilevel Logic Synthesis," Proceedings of the IEEE, Vol. 78, No. 2, pp. 264 – 300, 1990.

[4] J. A. Brzozowski and C.-J. H. Seger, "Asynchronous Circuits," Springer-Verlag, 1995.

[5] S. A. Edwards, "Making Cyclic Circuits Acyclic," Design Automation Conf., 2003.

[6] D. A. Huffman, "Combinational Circuits with Feedback," Recent Developments in Switching Theory, A. Mukhopadhyay, ed., pp. 27 – 55, 1971.

[7] W. H. Kautz, "The Necessity of Closed Circuit Loops in Minimal Combinational Circuits," IEEE Trans. Comp., Vol. C-19, pp. 162 – 166, 1970.

[8] S. Malik, "Analysis of Cyclic Combinational Circuits," IEEE Trans. Computer-Aided Design, Vol. 13, No. 7, pp. 950 – 956, 1994.

[9] A. Raghunathan, P. Ashar, and S. Malik, "Test Generation for Cyclic Combinational Circuits," IEEE Trans. Computer-Aided Design, Vol. 14, No. 11, pp. 1408 – 1414, 1995.

[10] M. Riedel and J. Bruck, "Cyclic Combinational Circuits: Analysis for Synthesis," Int'l Workshop Logic and Synthesis, 2003.

[11] R. L. Rivest, "The Necessity of Feedback in Minimal Monotone Combinational Circuits," IEEE Trans. Comp., Vol. C-26, No. 6, pp. 606 – 607, 1977.

[12] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton and A. Sangiovanni-Vincentelli, "SIS: A System For Sequential Circuit Synthesis," Tech. Rep., UCB/ERL M92/41, Electronics Research Lab, University of California, Berkeley, 1992.

[13] T. R. Shiple, "Formal Analysis of Synchronous Circuits," Ph.D. Thesis, University of California, Berkeley, 1996.

[14] T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Analysis of Combinational Cycles in Sequential Circuits," IEEE Int'l Symp. Circuits and Systems, Vol. 4, pp. 592 – 595, 1996.

[15] T. R. Shiple, G. Berry, and H. Touati, "Constructive Analysis of Cyclic Circuits," European Design and Test Conf., pp. 328 – 333, 1996.

[16] R. A. Short, "A Theory of Relations Between Sequential and Combinational Realizations of Switching Functions," Stanford Electronics Laboratories, Tech. Rep. 098-1, pp. 33 – 34, 102 – 114, 1960.

[17] A. Srinivasan and S. Malik, "Practical Analysis of Cyclic Combinational Circuits," IEEE Custom Integrated Circuits Conf., pp. 381 – 384, 1996.

[18] L. Stok, "False Loops Through Resource Sharing," Int'l Conf. Computer-Aided Design, pp. 345 – 348, 1992.