The Synthesis of Cyclic Combinational Circuits

Marc D. Riedel
California Institute of Technology
Mail Code 136-93
Pasadena, CA 91125
riedel@paradise.caltech.edu

Jeoshua Bruck
California Institute of Technology
Mail Code 136-93
Pasadena, CA 91125
bruck@paradise.caltech.edu

ABSTRACT

Combinational circuits are generally thought of as acyclic structures. It is known that cyclic structures can be combinational, and techniques have been proposed to analyze cyclic circuits to determine whether this is the case [7]. Cycles sometimes occur in designs synthesized from high-level descriptions, as well as in bus-based designs [15]. However, feedback in such cases is carefully contrived, typically occurring when functional units are connected in a cyclic topology. 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. The technique, applicable in the substitution phase of logic synthesis, optimizes a multilevel description, introducing feedback and potentially reducing the size of the resulting network. We have incorporated the technique in a general logic synthesis environment and performed trials on benchmark circuits and randomly generated examples. Many of the benchmark circuits were optimized significantly, with improvements of up to 30%. In trials with thousands of randomly generated examples, very nearly *all* had cyclic solutions superior to acyclic forms, with average improvements in the range of 5 to 15%.

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.

Keywords

Feedback, Logic Synthesis, Combinational Circuits

1. INTRODUCTION

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 better definition is that combinational circuits have outputs that depend only on the current values of the inputs; sequential circuits have outputs that may depend upon past as well as current input values.

A combinational circuit computes boolean-valued functions $f_i(x_1, \ldots, x_m)$, $1 \leq i \leq n$ of boolean inputs $x_1, \ldots, x_m$. A collection of logic gates connected in an acyclic (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 sequential behavior (as in the case of an `R-S` Latch), or it may be unstable (as in the case of an oscillator).

And yet, cyclic circuits can be combinational. Consider the network shown in Figure 1 consisting of three nodes, g_1, g_2 and g_3 producing output functions f_1, f_2 and f_3, respectively. Note that there is a cycle: g_1 receives f_3 as an input, g_2 receives f_1 as an input, and g_3 receives f_2 as an input. Nevertheless the circuit is combinational. To see why, note that if $a = 0$, then g_1 does not depend on f_3; if $b = 0$, then g_2 does not depend on f_1; if $a = 1$ and $b = 1$, then g_3 does not

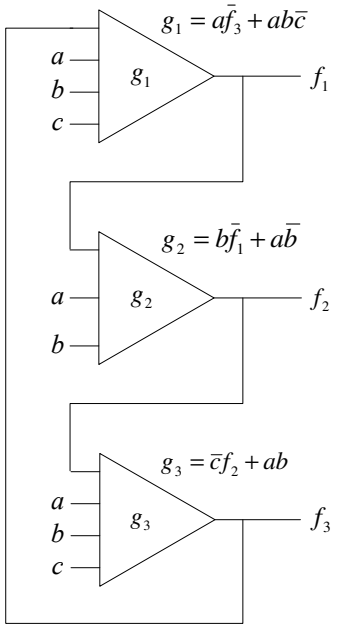


Figure 1: A cyclic combinational circuit.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Design Automation Conference '03 Anaheim, California USA
Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

depend on f_2. The circuit implements the functions

$$\begin{aligned} f_1(a,b,c) &= a\bar{b}c + ab\bar{c}, \\ f_2(a,b,c) &= \bar{a}b + a\bar{b} + bc, \\ f_3(a,b,c) &= ab + a\bar{c} + b\bar{c}. \end{aligned}$$

This network can be mapped to a circuit consisting of 9 fan-in two gates. In contrast, the best acyclic design that we could find requires 10 fan-in two gates. This demonstrates that cyclic circuits can be smaller than equivalent acyclic forms. 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. Figure 2 illustrates this. In the cyclic network, g_1 depends on g_3, g_2 depends on g_1 and g_3 depends on g_2. In the acyclic network, g_1 does not depend upon the other nodes. As a result additional gates are required to implement it.

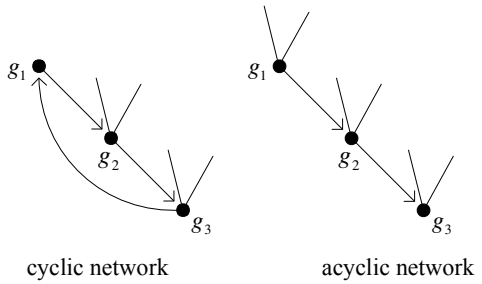


Figure 2: Cyclic vs. acyclic structures.

1.1 Prior Work

In 1992, Stok pointed out that cycles sometimes occur in circuits synthesized from high-level designs as well as in circuits with bus structures [15]. Cycles were observed in designs that were optimized to reuse functional units. For instance, given functional units $f(x)$ and $g(x)$ (these could be operations like "add" and "shift" on a datapath x) and a controlling variable y, one might implement

$$z(x) = \texttt{if } y \texttt{ then } f(g(x)) \texttt{ else } g(f(x)).$$

Feedback in such designs is carefully contrived, typically occurring when functional units are connected in a cyclic topology. Stok noted that while high-level synthesis tools and/or human designers sometimes create such cyclic designs, logic synthesis and verification tools used at later stages in the design process cannot handle cycles. His solution was to disallow the creation of cycles in the resource-sharing phase of high-level synthesis. In 1994 Malik proposed a technique for analyzing cyclic combinational circuits [7]. He formulated an efficient algorithm based on ternary-valued simulation to decide whether a circuit is combinational, and he proved that the problem is NP-complete. He also addressed the issue of timing analysis as well as fault testing [8]. In 1996, Shiple extended Malik's work and set it on firm theoretical footing [10]. 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 [11] and extended the concept to combinational logic embedded in sequential circuits [12].

To summarize, combinational circuits have been observed, and their behavior formalized, well understood and accepted. However, except for relatively simple cases of feedback at the level of functional units, combinational circuits are not designed with feedback in practice, and 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 [13]. Around 1970, Huffman and Kautz argued that cyclic combinational circuits could have fewer logic gates than acyclic forms [5], [6]. Unfortunately, the examples that they gave, although plausible, do not meet the rigorous condition for stability and uniqueness. Shiple demonstrated that the example given by Kautz fails the test and may not work correctly [10]; the same can be said for the example given by Huffman.

In 1977 Rivest presented a convincing example of a family of cyclic combinational circuits [9]. 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 twice. 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. Significantly, 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. Rivest states that in general

> "it remains an open problem to determine the extent to which feedback can yield economical realizations."

Twenty-five years later, the topic of incorporating feedback in the design of combinational circuits is still an open one, both in theory and in practice.

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 any 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 *design* 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. The technique, applicable in the substitution phase of logic synthesis, optimizes a multilevel description, introducing cycles and potentially reducing the size of the resulting network. We have incorporated the technique in a general logic synthesis environment, namely the Berkeley SIS package, and performed trials on benchmark circuits and randomly generated examples. Many of the benchmark circuits were optimized significantly with feedback, with improvements of up to 30%. In trials with thousands of randomly generated examples, very nearly *all* had cyclic solutions superior to acyclic forms, with average improvements in the range of 5 to 15%.

1.3 Notation and Definitions

$\bar{x}$ indicates the negation of a literal x, the symbol '+' denotes disjunction ("OR"), and multiplication denotes conjunction ("AND").

A boolean **network** consists of **input variables** $x_1, \ldots, x_m$, **node functions** $g_1, \ldots, g_n$, and **output functions** $f_1, \ldots, f_p$.

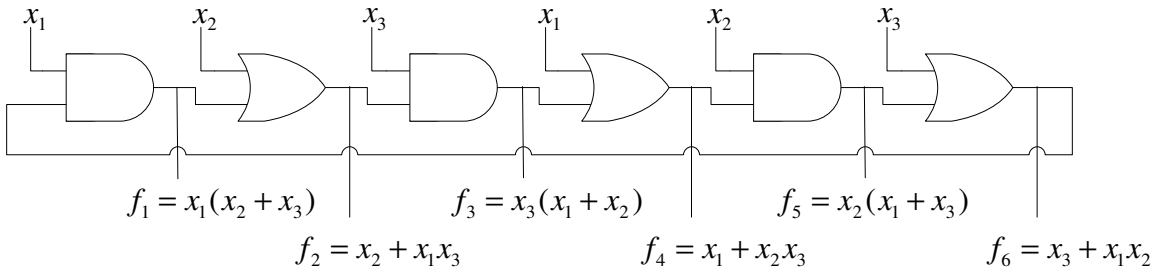


Figure 3: A cyclic combinational circuit due to Rivest [9].

An **internal variable** y_i is associated with each node, $1 \leq i \leq m$. Each node function is a logical expression consisting of input variables and internal variables. Each output function is a logical expression consisting of input variables only. Our **cost measure** is the sum of the literals in the factored form of node expressions (see [3]).

We use the term **combinational** with respect to an input assignment to mean that the network computes a unique value for each output. (Shiple uses the term "combinationally output-stable" [10].) This computation must hold:

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

A more precise circuit model and definition is given in [10]. The output functions f_i, $1 \leq i \leq p$ are only defined for combinational input assignments.

We use the term combinational with respect to a network to mean that the network is combinational for all input assignments in its "care" set. $C(g_1, \ldots, g_n)$ is a logical expression consisting of input variables that denotes the necessary and sufficient condition for stability and uniqueness. It evaluates to 1 for input assignments for which the network is combinational. The network is combinational iff the "care" set is contained in C.

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 their original inputs as well as other node functions. For instance, given functions

$$\begin{aligned} f_1(a,b,c) &= a\bar{b}c + ab\bar{c}, \\ f_2(a,b,c) &= \bar{a}b + a\bar{b} + bc, \\ f_3(a,b,c) &= ab + a\bar{c} + b\bar{c}. \end{aligned}$$

we can substitute f_1 into f_2 to obtain

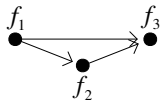
$$f_2 = g_2(a,b,f_1) = \bar{f}_1 b + a\bar{b},$$

or we could substitute f_2 into f_1 to obtain

$$f_1 = g_1(b,c,f_2) = f_2\bar{b}c + \bar{f}_2 b.$$

If we try to substitute f_3 into f_1, we find that it isn't helpful. For a given function f_i, dependent only on the primary inputs, substitution of a set of functions F_1 yields an expression dependent on a set of functions $F_2 \subseteq F_1$. In general, the resulting expression is not unique. Substitution may yield several *alternative* functions of varying cost. Also, 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 in logic synthesis are heuristical, but exceptions are rare.)

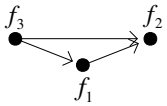
In existing methodologies, a total ordering is enforced among the functions in the substitution phase to ensure that no cycles occur. This choice can influence the cost of the solution. For instance, with the ordering



substitution yields

$$\begin{aligned} f_1 &= g_1(a,b,c) = a(b\bar{c} + \bar{b}c) \\ f_2 &= g_2(a,b,f_1) = b\bar{f}_1 + a\bar{b} \\ f_3 &= g_3(a,b,c,f_1) = a\bar{f}_1 + b\bar{c} \end{aligned}$$

with a cost of 13; whereas the ordering



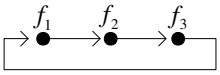
yields

$$\begin{aligned} f_1 &= \bar{b}cf_2 + b\bar{f}_2 \\ f_2 &= b(c + \bar{a}) + a\bar{b} \\ f_3 &= a\bar{f}_1 + b\bar{c} \end{aligned}$$

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

$$\begin{aligned} f_1 &= \bar{f}_2 b + \bar{f}_3 a \\ f_2 &= \bar{f}_1 b + a\bar{b} \\ f_3 &= \bar{f}_1 a + b\bar{c} \end{aligned}$$

with a cost of only 12. This network is cyclic and *not* combinational. This may be verified according to Malik's procedure. (Informally, note what happens when $a = 1$ and $b = 1$: we have $f_1 = \bar{f}_2 + \bar{f}_3$, $f_2 = \bar{f}_1$, and $f_3 = \bar{f}_1 + \bar{c}$.) However, suppose that we restrict the order of substitution to



We obtain,

$$\begin{aligned} f_1 &= a(\bar{f}_3 + b\bar{c}) \\ f_2 &= b\bar{f}_1 + a\bar{b} \\ f_3 &= \bar{c}f_2 + ab \end{aligned}$$

also with a cost of 12. As discussed in the introduction, this network is combinational.

2.1 Problem Statement

We first state the problem in its most general form, and then we discuss various heuristics that have been explored to make the search computationally tractable. We assume that the network is given to us in collapsed form, that is to say, we are given a set of output functions $f_1, \ldots, f_p$ to implement. We have the substitution operation at our disposal. For each output function f_i, we try substituting different sets of functions. Call such a set a **substitutional set**. For each substitutional set F_j we generate a node function $g_i^{(j)}$ (or several functions $g_i^{(j,1)}, g_i^{(j,2)}, \ldots$). For instance, for the output function

$$f_1 = a(b\bar{c} + \bar{b}c)$$

the substitutional set $\{f_2, f_3\}$ generates a node function

$$g_1^{(1)} = \bar{f}_2 b + \bar{f}_3 a.$$

We list the node functions obtained in this way in increasing order of cost. For a three-node network, the list might be

f_1		f_2		f_3	
$\{f_2, f_3\}:$	$g_1^{(1)}$	$\{f_1, f_3\}:$	$g_2^{(1)}$	$\{f_1, f_2\}:$	$g_3^{(1)}$
$\{f_2\}:$	$g_1^{(2)}$	$\{f_1\}:$	$g_2^{(2)}$	$\{f_1\}:$	$g_3^{(2)}$
$\{f_3\}:$	$g_1^{(3)}$	$\{f_3\}:$	$g_2^{(3)}$	$\{f_2\}:$	$g_3^{(3)}$
$\emptyset:$	f_1	$\emptyset:$	f_2	$\emptyset:$	f_3

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.

The goal of the synthesis process is to select a choice of node functions that minimizes the cost while satisfying the condition for stability and uniqueness. For a three-node network, choose the i-th expression from the first column, the j-th expression from the second column and the k-th expression from third column such that we get

$$\min_{i,j,k} : \text{cost}(g_1^{(i)}) + \text{cost}(g_2^{(j)}) + \text{cost}(g_3^{(k)})$$

while satisfying the condition for stability and uniqueness

$$C(g_1^{(i)}, g_2^{(j)}, g_3^{(k)}).$$

For a network with a non-trivial number of nodes, an exhaustive search is evidently intractable. With n nodes, there are 2^{n-1} substitutional sets for each node, for a total of 2^n possibilities. Fortunately, the structure of the search space immediately suggests the application of heuristics. We have considered two such heuristics: branch-and-bound, and dynamic programming. Due to space restrictions, we only describe the branch-and-bound approach here.

2.2 Branch-and-Bound Algorithms

An important step in the synthesis process is the analysis for combinationality. We have devised an algorithm that is recursive, and thus it provides information about the stability and uniqueness of components of the network. Also it lends itself well to the caching of analysis results for common subnetworks through iterations of the search. We do not describe the details here due to space restrictions. We discuss two branching search algorithms.

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 **maximally connected** network. For each node f_i, we generate a node expression $g_i^{(1)}$ based on the complete substitutional set (i.e., all other nodes in the network). Call the dependency of a node function on another node an "edge". This initial branch has the densest set of edges, and its cost provides a lower bound on the cost of the solution. As edges are excluded in the branch-and-bound process, the cost of the network increases. (Again, since the substitution step is based on heuristics, 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.

The process is illustrated in Figure 4. 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.

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 flat node functions). Edges are added as the substitutional sets of nodes are augmented. As edges are included, the cost of the network decreases. The process is shown in Figure 4. 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.

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. Although we haven't explored this possibility, a hybrid "build-up"/"break-down" approach might be feasible.

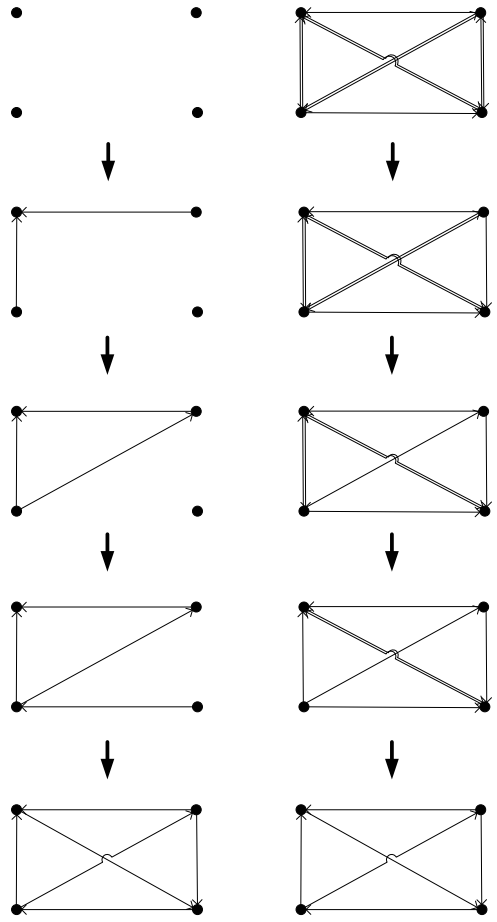


Figure 4: Search Strategies: "Build-Up" on the left-hand side and "Break-Down" on the right-hand side.

2.3 Example: Binary to 7-Segment Decoder

We illustrate the synthesis method on a common circuit, ubiquitous in introductory logic design courses: the "binary to 7-segment display" decoder. The inputs are four bits, x_0, x_1, x_2 and x_3, decoding a number from 0 to 9. The outputs are 7 bits, a, b, c, d, e, f and g, specifying which segments in an LED display light up, as shown in Figure 5. The node functions for this circuit are

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

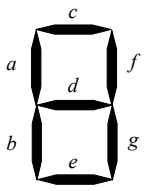
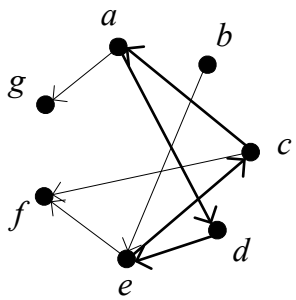


Figure 5: Seven segment display.

with a cost of 55. With the "Build-Up" algorithm, 8 edges are added:



Note that there is a cycle, shown in bold. This yields a cyclic combinational network,

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

with a cost of 34. With the "Break-Down" algorithm, the initial network has 17 edges. Nine edges are deleted, yielding the same cyclic combinational network as above. For comparison, executing the command `full_simplify` in the Berkeley SIS package yields an acyclic network with a cost of 37.

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 as well as on some of the usual suspects, namely the Espresso [2] and LGSynth93 [1] benchmarks. For benchmarks ciruits with latches, we extracted the combinational part. We note that solutions for many of the examples contain dozens or even hundreds of cycles. We note that solutions for many of the examples do not contain just one or just a few cycles; they contain dozens or even hundreds of cycles.

3.1 Methodology

We present a simple comparison between the cost of cyclic versus acyclic substitutions. We have also investigated the role of feedback in other phases of logic synthesis, namely decomposition and technology mapping. However, we do not discuss these aspects here due to space restrictions.

The input consists of a collapsed network. The substitution and minimization operation is performed with the `simplify`

command in the Berkeley SIS package, 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. Only a very modest amount of computation was applied for the results presented here (on the order of minutes on a Pentium IV, 1.8 GHz workstation).

3.2 Benchmarks

Examples were selected based on size and suitability. We mostly considered circuits with fewer than 30 inputs and fewer than 30 outputs. In Figure 6, we present those for which cyclic solutions were found.

LGSynth93 & Espresso Benchmarks					
	# In.	# Out.	`Simplify`	`Cyclify`	Diff.
dc1	4	7	39	34	12.8 %
ex6	8	11	85	76	10.6 %
p82	5	14	104	90	13.5 %
t4	12	8	109	89	18.3 %
inc	7	9	116	107	7.8 %
bbsse	11	11	118	106	10.2 %
sse	11	11	118	106	10.2 %
5xp1	7	10	123	109	11.4 %
dc2	8	7	130	123	5.4 %
s386	11	11	131	113	13.7 %
dk17	10	11	160	136	15.0 %
bw	5	28	171	163	4.7 %
s400	24	27	179	165	7.8 %
s382	24	27	180	165	8.3 %
apla	10	12	185	131	29.2 %
tms	8	16	185	158	14.6 %
s526n	24	27	194	189	2.6 %
s526	24	27	196	188	4.1 %
cse	11	11	212	177	16.5 %
clip	9	5	213	189	11.3 %
pma	11	13	226	211	6.6 %
m2	8	16	231	207	10.4 %
dk16	7	9	248	233	6.0 %
s510	25	13	260	227	12.7 %
t1	21	23	273	206	24.5 %
b4	33	23	292	281	3.8 %
ex1	13	24	309	276	10.7 %
exp	8	18	320	262	18.1 %
s1	13	11	332	322	3.0 %
in3	35	29	361	333	7.8 %
in2	19	10	397	291	26.7 %
b10	15	11	398	359	9.8 %
duke2	22	29	415	394	5.1 %
gary	15	11	421	404	4.0 %
m4	8	16	439	411	6.4 %
in0	15	11	451	434	3.8 %
styr	14	15	474	443	6.5 %
planet1	13	25	550	517	6.0 %
planet	13	25	555	504	9.2 %
s1488	14	24	622	589	5.3 %
s1494	14	25	659	634	3.8 %
max1024	10	6	793	774	2.4 %
table3	14	14	1287	1175	8.7 %
table5	17	15	1059	1007	4.9 %
s298	11	14	2598	2445	5.9 %
ex1010	10	10	3703	3593	3.0 %

Figure 6: Cost (literals in factored form) of Berkeley SIS `Simplify` vs. `Cyclify` for benchmarks.

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 7. Each row lists the results of 25 trials. Strikingly, CYCLIFY yielded cyclic solutions in nearly all cases (see 3rd column). The cost reduction is significant and very consistent. The average improvement is given in the 4th column, and the range of improvement in the 5th column.

Randomly Generated Networks				
# In.	#. Out.	Cyclic Solns.	Avg. Improvement	Range
5	5	96%	7.5%	0% – 16%
5	7	96%	9.0%	0% – 18%
5	10	100%	11.6%	2% – 20%
5	15	100%	13.7%	4% – 22%
5	20	100%	14.2%	8% – 18%
7	10	92%	4.7%	0% – 10%

Figure 7: Analysis of randomly generated examples.

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. We have formulated a general methodology for the synthesis of cyclic combinational circuits and incorporated it into a logic synthesis environment. Our search algorithms, while heuristical, can effectively tackle circuits of sizes that are of practical importance. We note that the implementation of more sophisticated search algorithms, such as stochastic search, is an obvious evolution of the project. Also, we are exploring parallelization. Further directions: on the theoretical side, we would like to approach the problem from a circuit complexity perspective. On the practical side, we will incorporate the techniques into different synthesis environments.

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

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

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

[8] 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.

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

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

[11] 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.

[12] T. R. Shiple, G. Berry, and H. Touati, "Constructive Analysis of Cyclic Circuits," European Design and Test Conf., 1996.

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

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

[15] L. Stok, "False Loops Through Resource Sharing," Int'l Conf. Computer-Aided Design, Santa Clara, 1992.