Algorithmic Aspects of Cyclic Combinational Circuit Synthesis

Marc D. Riedel and Jehoshua Bruck
California Institute of Technology
Mail Code 136-93, Pasadena, CA 91125
E-mail: {riedel, bruck}@paradise.caltech.edu

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, as well as in bus-based designs [16]. 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 [7], no one has attempted the synthesis of circuits with feedback at the logic level.**

We have argued the case for a paradigm shift in combinational circuit design [10]. We should no longer think of combinational logic as acyclic in theory or in practice, since most combinational circuits are best designed with cycles. We have proposed a general methodology for the synthesis of multilevel networks with cyclic topologies and incorporated it in a general logic synthesis environment. In trials, benchmark circuits were optimized significantly, with improvements of up to 30% in the area.

In this paper, we discuss algorithmic aspects of cyclic circuit design. We formulate a symbolic framework for analysis based on a divide-and-conquer strategy. Unlike previous approaches, our method does not require ternary-valued simulation. Our analysis for combinationality is tightly coupled with the synthesis phase, in which we assemble a combinational network from smaller combinational components. We discuss the underpinnings of the heuristic search methods and present examples as well as synthesis results for benchmark circuits.

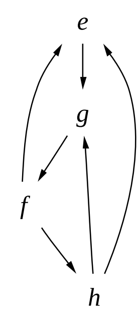
Keywords— **Feedback, Logic Synthesis, Combinational Circuits**

I. 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 $g_i(x_1, \ldots, x_m)$, $1 \le i \le n$ 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 timing-dependent behavior (as in the case of an R-S Latch), and it may be unstable (as in the case of an oscillator).

	d, c, b, a	π	h, g, f, e
0	0 0 0 0	3	0 0 1 1
1	0 0 0 1	1	0 0 0 1
2	0 0 1 0	4	0 1 0 0
3	0 0 1 1	1	0 0 0 1
4	0 1 0 0	5	0 1 0 1
5	0 1 0 1	9	1 0 0 1
6	0 1 1 0	2	0 0 1 0
7	0 1 1 1	6	0 1 1 0
8	1 0 0 0	5	0 1 0 1
9	1 0 0 1	3	0 0 1 1
10	1 0 1 0	5	0 1 0 1
11	1 0 1 1	8	1 0 0 0
12	1 1 0 0	9	1 0 0 1
13	1 1 0 1	7	0 1 1 1
14	1 1 1 0	9	1 0 0 1
15	1 1 1 1	3	0 0 1 1



$$
\begin{aligned}
e &= \bar{f}(a\bar{h} + c) + d\bar{h} + \bar{b} \\
f &= \bar{a}\bar{d}\bar{g} + a(\bar{b}d + bc) \\
g &= \bar{a}b\bar{c} + \bar{h}(a\bar{e} + \bar{a}d + \bar{b}c) \\
h &= \bar{f}(a(c + d) + cd)
\end{aligned}
$$

Fig. 1. Example: Lookup table for the digits of π.

And yet, cyclic circuits can be combinational. Consider the example shown in Figure 1, a lookup table for the first 16 digits of π. Given inputs a, b, c, d specifying a number i between 0 and 15 (in binary), the network yields outputs, e, f, g, h, specifying the i-th digit of π (in binary). Each output is specified as a function of the input variables and the other output functions. As shown, the network contains cycles ((e, g, f), (e, g, f, h) and (f, h, g)). In spite of this, the network is combinational. For each combination of input values, the network produces the correct outputs, regardless of the initial state and independently of all timing assumptions. To see this, consider specific input values. For instance, with $a = 0, b = 0, c = 0, d = 0$, the network simplifies to that shown in Figure 2, yielding the correct value of $e = 1, f = 1, g = 0, h = 0$ (the first digit of π, namely 3). With $a = 1, b = 1, c = 1, d = 1$, the network simplifies to that shown in Figure 3, yielding the correct value of $e = 1, f = 1, g = 0, h = 0$ (the 16th digit of π, namely 3). The reader may verify that the network implements all the values in between 0000 and 1111 correctly. Although it is straightforward to verify that a cyclic network is combinational, it is not obvious how to go about designing such networks, nor is it clear why one would want to go to the

$$\begin{aligned} e &= & 1 \\ f &= \bar{g} &= 1 \\ g &= & 0 \\ h &= & 0 \end{aligned}$$

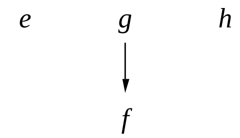


Fig. 2. Network in Figure 1 with $a = 0, b = 0, c = 0, d = 0$.

$$\begin{aligned} e &= \bar{f} + \bar{h} &= 1 \\ f &= &= 1 \\ g &= \bar{e}\bar{h} &= 0 \\ h &= \bar{f} &= 0 \end{aligned}$$

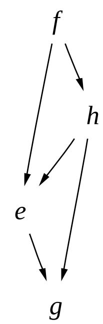


Fig. 3. Network in Figure 1 with $a = 1, b = 1, c = 1, d = 1$.

trouble. In fact, feedback is highly advantageous. In [10] we demonstrated that cyclic networks are generally 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 4 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.

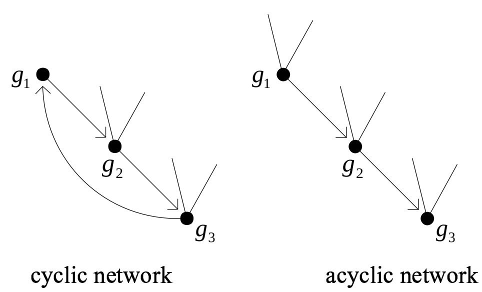


Fig. 4. Cyclic vs. acyclic structures.

A. 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 [16]. 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 based on ternary-valued simulation [7]. He also addressed the issue of timing analysis as well as fault testing [9], [15]. In 1996, Shiple discussed the theoretical underpinnings of this work, extending the concept to combinational logic embedded in sequential circuits, and he proposed refinements to Malik's algorithm [12], [13], [14].

Although the premise of cycles in combinational circuits has been established, combinational circuits are not designed with feedback in practice. Except for relatively simple cases of feedback at the level of functional units, no one has attempted the synthesis of circuits with feedback at the logic level.

B. Contributions

We have proposed a general methodology for the synthesis of multilevel networks with cyclic topologies and incorporated it in a general logic synthesis environment, namely the Berkeley SIS package. Our approach is to optimize a multilevel description in the substitution phase, introducing feedback and potentially reducing the area. In trials with benchmark circuits, many were optimized significantly, with improvements of up to 30% in the cost (as measured by the literal count of the nodes expressed in factored form). In trials with randomly generated examples, very nearly *all* had cyclic solutions superior to acyclic forms.

In this paper, we discuss algorithmic aspects of cyclic circuit design. We formulate a symbolic framework for analysis that obviates the need for ternary-valued simulation. Our algorithm for deciding combinationality, based on a divide-and-conquer strategy, analyzes components of the network. It is tightly coupled with the synthesis phase, in which we assemble a combinational network from smaller combinational components. We discuss the underpinnings of the heuristic search methods and present examples as well as synthesis results for benchmark circuits.

C. Definitions and Notation

The exposition in this paper is based upon symbolic operations. By this we mean algebraic operations[1] on a symbolic representation of boolean functions. The representation that we used in our implementation is based on Binary Decision Diagrams (BDDs) [4].

We use the standard notation: addition ($+$, $\sum$) denotes disjunction (OR), multiplication ($\cdot$, $\prod$) denotes conjunction (AND), and an $\bar{x}$ denotes negation (NOT). The **restriction** operation (also known as the cofactor) of a function f with respect to a variable x,

$$f|_{x=v}.$$

refers to the assignment of the constant value $v \in \{0, 1\}$ to x. The **composition** operation of a function f with respect

[1]Here we mean "algebraic" in the mathematical sense; it is not a reference the term "algebraic" (as opposed to "boolean") for methods in logic decomposition/restructuring.

to a variable x and a function g,

$$f|_{x=g},$$

refers to the substitution of g for x in f. A function f **depends** upon a variable x iff $f|_{x=0}$ is not identically equal to $f|_{x=1}$. Call the variables that a function depends upon its **support set**.

For the following definitions, we divide the variables into two subsets (the x_i's and the y_j's, corresponding to the inputs and the internal variables, respectively, defined in Section I-D). An operation with respect to a subset (the y_j's) yields an expression in terms of the remaining variables (the x_i's).

The **universal quantification** operation (also known as consensus) yields a function

$$\forall\,(y_1,\ldots,y_n)f$$

that is true iff the given function f is true for all 2^n assignments of boolean values to the variables $y_1,\ldots,y_n$. The **existential quantification** operation (also known as smoothing) yields a function

$$\exists\,(y_1,\ldots,y_n)f$$

that is true iff the given function f is true for *some* assignment of boolean values to the variables $y_1,\ldots,y_n$. The **marginalize** operation yields a function

$$f\downarrow(y_1,\ldots,y_n)$$

that is true iff the given function f is invariant for all 2^n assignments of boolean values to $y_1,\ldots,y_n$. For a single variable y, it is true iff $f|_{y=0}$ agrees with $f|_{y=1}$,

$$f\downarrow y = f|_{y=0}\cdot f|_{y=1} + \overline{f|_{y=0}}\cdot\overline{f|_{y=1}}.$$

(This is the complement of what is known as the boolean difference). For several variables $y_1,\ldots,y_n$, it is computed as the universal quantification of the product of the marginals:

$$f\downarrow(y_1,\ldots,y_n) = \forall\,y_1,\ldots y_n\,\left[(f\downarrow y_1)\cdot(f\downarrow y_n)\right].$$

For example, with

$$f = x_1 + x_2y_1 + x_3y_2 + x_4y_1y_2,$$

we have,

$$\begin{aligned}
f\downarrow y_1 &= x_1 + x_3y_2 + \bar{x}_2(\bar{x}_4+\bar{y}_2),\\
f\downarrow y_2 &= x_1 + x_2y_1 + \bar{x}_1(\bar{x}_4+\bar{y}_1),\\
f\downarrow(y_1,y_2) &= x_1 + \bar{x}_2\bar{x}_3\bar{x}_4.
\end{aligned}$$

Note that the marginalize operator requires a linear number of symbolic operations.

D. Network Model

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 functions of boolean input variables $x_1,\ldots,x_m$. Internally, the network is specified as a collection of nodes $\mathcal{N}$. Associated with each node $1\le i\le n$ is a **node function** f_i and an **internal variable** y_i. The node functions depend on input variables as well as on internal variables. In the **dependency** graph, 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.

Also associated with each node is a **target function** g_i (in the case of acyclic networks this would be the "collapsed" function). The target functions depend on the input variables only. A subset of the nodes are designated as **output nodes**. For these, the target functions are the requisite output functions. If we substitute the target function g_j for each corresponding internal variable y_j in a node function f_i, we get the corresponding target function g_i,

$$f_i|_{y_1=g_1,\ldots,y_n=g_n} = g_i.$$

We use the notation

$$\mathcal{N}|_{y_i}$$

to mean that the target function g_i is substituted for the corresponding internal variable y_i in *every* node function of the network. Consider a network with node functions[2]

$$\begin{aligned}
f_1 &= \bar{x}_1y_2 + \bar{x}_2\bar{x}_3\\
f_2 &= \bar{x}_2(x_3+x_1) + \bar{x}_3\bar{y}_3\\
f_3 &= \bar{x}_2y_1 + x_1x_2
\end{aligned}$$

and target functions

$$\begin{aligned}
g_1 &= \bar{x}_3(\bar{x}_1+\bar{x}_2) + \bar{x}_1\bar{x}_2\\
g_2 &= \bar{x}_1x_2\bar{x}_3 + \bar{x}_2(x_1+x_3)\\
g_3 &= \bar{x}_2(\bar{x}_1+\bar{x}_3) + x_1x_2.
\end{aligned}$$

In this example, if we substitute g_2 for y_2 in f_1, we get g_1:

$$f_1|_{y_2=g_2} = \bar{x}_1g_2 + \bar{x}_2\bar{x}_3 = g_1.$$

For a fixed assignment of boolean input values, a node function may no longer depend on some of the internal variables in its support set. In the example above, f_2 depends on y_3 in general. Indeed, for an input vector $x_1=0, x_2=0, x_3=0$,

$$f_2(0,0,0,y_3) = \bar{y}_3.$$

However, for $x_1=1, x_2=0, x_3=0$, f_2 does not depend on y_3,

$$f_2(1,0,0,y_3) = 1.$$

For a fixed assignment of inputs, call the network the **induced** network, and call the associated dependency graph the **induced** dependency graph. In the induced network, if a node function f_i doesn't depend upon any internal variable (i.e., it evaluates to 0 or 1), then we may substitute this value for the corresponding internal variable y_i in other expressions. In this way, we can continue to simplify the network, until no further simplifications are possible. Call the result the **simplified induced** network.

[2]We use x_i, y_i, f_i, g_i when we refer to networks in the abstract. However, for the sake of readability, in our examples we use $a, b, c, \ldots$ for the input variables. We use $e, f, g, \ldots$ for the node functions, the internal variables, and the target functions: on the left-hand side of an equation the symbol refers to either a node function or a target function depending on the context; on the right-hand side, it refers to the associated internal variable.

E. Definition of Combinational

A network is **combinational** iff it computes unique boolean output values for each boolean input vector.[3] We sometimes abuse this terminology and say that a network is combinational for a *specific* input vector, meaning that it computes unique boolean output values for that input vector. If there are "don't care" conditions on the inputs, then it is sufficient if the network computes unique boolean values for input vectors in the "care" set. This computation must hold:

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

Proposition 1 *A network is combinational iff, for each assignment of boolean values to the inputs, all output nodes in the simplified induced network evaluate to definite boolean values.*

This definition of combinational is functionally equivalent to that proposed in earlier work. Malik [7] suggested the ternary model for the analysis of cyclic combinational circuits. Following Bryant [5], his approach for deciding combinationality is based on ternary-valued simulation. He uses a "dual-rail" encoding (10 for one, 01 for zero, and 11 for "unknown") to reduce the problem to boolean simulation. Shiple [12] and Mendler [8] elaborated on Malik's approach, putting the work on a firm theoretical footing by showing that the definition of combinational corresponds to that of circuits that are well-behaved electrically, according to the up-bounded inertial delay model [6].

II. Analysis

We formulate a symbolic framework for analysis that obviates the need for ternary-valued simulation. We tackle the problem with a divide-and-conquer approach: progressively smaller components of the network are analyzed for combinationality. We note that if a network's dependency graph can be divided into several distinct strongly-connected components, then the analysis may be performed separately on each component. For simplicity, we assume that each node in the network is an output node.

A. Symbolic Framework

The marginalize operator, defined in Section I-C, specifies when a node function is independent of the internal variables in its support set. For a node function f_i, dependent on a set of internal variables Y_i, if $(f_i \downarrow Y_i)$ holds, then f_i has a definite boolean value equal to the corresponding target function g_i.

For a network $\mathcal{N}$, the restriction $\mathcal{N}|_{y_i}$ means that we cut node i from the network, replacing it with the corresponding target function g_i (expressed entirely in terms of the input variables). This is accomplished by substituting g_i for the corresponding internal variable y_i in *every* node function of the network.

The following theorem states a necessary and sufficient condition for combinationality.

[3] Shiple uses the term "combinationally output-stable" [12]

Theorem 1

$$C(\mathcal{N}) = (f_1 \downarrow Y_1) \cdot C(\mathcal{N}|_{y_1}) + \cdots + (f_n \downarrow Y_n) \cdot C(\mathcal{N}|_{y_n}).$$

Proof Sketch: We argue that for each input vector at least one node function must evaluate to a definite boolean value independently of all the others. Indeed, if none of the functions evaluate to a definite boolean value, then no simplifications are possible and the network is not combinational. A function evaluates to a definite boolean value independently of the others iff the marginal holds

$$(f_i \downarrow Y_i).$$

Now, if a node function f_i evaluates to a definite boolean value, this value is given by the corresponding target function g_i. If we cut this node from the network, then the rest of the network must be combinational, that is

$$C(\mathcal{N}|_{y_i})$$

must hold. Indeed, if a component of the network viewed in isolation is not combinational, then the entire network is not combinational. □

We illustrate the analysis with two examples. Consider the target functions,

$$\begin{aligned} d &= \bar{c}(\bar{b} + \bar{a}) + \bar{a}\bar{b} \\ e &= \bar{a}b\bar{c} + \bar{b}(c + a) \\ f &= \bar{b}(\bar{c} + \bar{a}) + ab. \end{aligned}$$

Example 1

Consider the network $\mathcal{N}_1$, shown in Figure 5. Note that the

$$\begin{aligned} d &= \bar{a}e + \bar{b}\bar{c} \\ e &= \bar{b}(c + a) + \bar{c}\bar{f} \\ f &= \bar{b}d + ab \end{aligned}$$

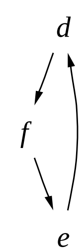


Fig. 5. Example: Network $\mathcal{N}_1$.

dependency graph is a single cycle. The necessary and sufficient condition is

$$\begin{aligned} C(\mathcal{N}_1) &= [d \downarrow e] \cdot C(\mathcal{N}_1|_d) + \\ &\quad [e \downarrow f] \cdot C(\mathcal{N}_1|_e) + \\ &\quad [f \downarrow d] \cdot C(\mathcal{N}_1|_f). \end{aligned}$$

The marginals are

$$\begin{aligned} d \downarrow e &= a + \bar{b}\bar{c} \\ e \downarrow f &= c + a\bar{b} \\ f \downarrow d &= b. \end{aligned}$$

Since we have a single cycle,

$$C(\mathcal{N}_1|_d) = C(\mathcal{N}_1|_e) = C(\mathcal{N}_1|_f) = 1.$$

Thus,

$$C(\mathcal{N}_1) = a + \bar{b}\bar{c} + c + a\bar{b} + b = 1.$$

We conclude that the network is combinational for all input vectors.

Example 2

Now consider the network $\mathcal{N}_2$ shown in Figure 6.

$$\begin{aligned} d &= \bar{b}f + \bar{c}e \\ e &= d(\bar{f} + a) + \bar{b}c \\ f &= a\bar{e} + \bar{b}d \end{aligned}$$

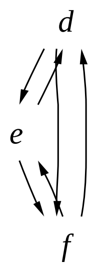


Fig. 6. Example: Network $\mathcal{N}_2$.

Note that the dependency graph is the complete graph on three nodes. The necessary and sufficient condition is

$$\begin{aligned} C(\mathcal{N}_2) &= [d \downarrow (e, f)] \cdot C(\mathcal{N}_2|_d) + \\ &\quad [e \downarrow (d, f)] \cdot C(\mathcal{N}_2|_e) + \\ &\quad [f \downarrow (d, e)] \cdot C(\mathcal{N}_2|_f). \end{aligned}$$

The marginals are

$$\begin{aligned} d \downarrow (e, f) &= bc \\ e \downarrow (d, f) &= \bar{b}c \\ f \downarrow (d, e) &= \bar{a}b. \end{aligned}$$

For the restriction $\mathcal{N}_2|_d$, we compute

$$\begin{aligned} e|_d &= \bar{b}(a + c) + \bar{a}\bar{c}\bar{f} \\ f|_d &= \bar{b}(\bar{a} + \bar{c}) + a\bar{e}. \end{aligned}$$

For this restriction, the marginals are

$$\begin{aligned} (e|_d) \downarrow f &= a + c \\ (f|_d) \downarrow e &= \bar{a} + \bar{b}\bar{c} \end{aligned}$$

Now, recursively,

$$\begin{aligned} C(\mathcal{N}_2|_d) &= [(e|_d) \downarrow f] \cdot (1) + [(f|_d) \downarrow e] \cdot (1) \\ &= a + c + \bar{a} + \bar{b}\bar{c} \\ &= 1. \end{aligned}$$

Similarly, we compute

$$\begin{aligned} C(\mathcal{N}_2|_e) &= a\bar{c} + b \\ C(\mathcal{N}_2|_f) &= \bar{b} + c. \end{aligned}$$

Thus,

$$\begin{aligned} C(\mathcal{N}_2) &= (bc) \cdot (1) + (\bar{b}c)(a\bar{c} + b) + \bar{a}b(\bar{b} + c) \\ &= bc. \end{aligned}$$

We conclude that the network is combinational iff bc holds.

B. Complexity

In the recursive decomposition of the necessary and sufficient condition for combinationality in a network, one may encounter the same sub-network several times. Restriction is invariant to order so that for any i, j,

$$(\mathcal{N}|_{y_i})|_{y_j} = (\mathcal{N}|_{y_j})|_{y_i}.$$

Thus we need not recompute the condition for the same component encountered twice. For instance, in a network with nodes, $f_1, f_2, \ldots$, we compute

$$C(\mathcal{N}) = (f_1 \downarrow y_1) \cdot C(\mathcal{N}|_{y_1}) + (f_2 \downarrow y_2) \cdot C(\mathcal{N}|_{y_2}) + \cdots .$$

Recursively, we compute

$$C(\mathcal{N}|_{y_1}) = ((f_2|_{y_1})|_{y_2}) \cdot C((\mathcal{N}|_{y_1})|_{y_2}) + \cdots ,$$

and

$$C(\mathcal{N}|_{y_2}) = ((f_1|_{y_2})|_{y_1}) \cdot C((\mathcal{N}|_{y_2})|_{y_1}) + \cdots .$$

We needn't recompute $(\mathcal{N}|_{y_2})|_{y_1}$, as it is equal to $(\mathcal{N}|_{y_1})|_{y_2}$.

For a network corresponding to a complete graph on n nodes, the analysis requires on the order of $n \cdot 2^n$ steps (there are 2^n subsets of n nodes, each of which has n terms to evaluate). For less densely connected networks, the analysis is of course less complex. Malik has shown that the problem of analyzing a network to determine if it is combinational is co-NP-complete [7]. His approach for analysis, based on ternary simulation, seems on the surface to be completely different from ours. However, it may be shown that the complexity of both approaches is the same. If we were to translate Malik's approach into our framework, we would perform the exact same sequence of restrictions and marginals, albeit in a different order.

III. Synthesis Algorithms

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 "re-substitution"), in which node functions are expressed, or re-expressed, in terms other node functions as well as of their original inputs.[4]

A. Substitution

Consider the example in Figure 1 from the Introduction. The target functions are

$$\begin{aligned} e &= \bar{b} + a\bar{c}\bar{d} + d(\bar{a} + c) \\ f &= a(bc + \bar{b}d) + \bar{d}(\bar{a}\bar{b}\bar{c} + bc) \\ g &= ac(\bar{b}d + b\bar{d}) + \bar{a}(\bar{b}c\bar{d} + \bar{c}(b + d)) \\ h &= \bar{a}cd + a(\bar{b}c\bar{d} + b\bar{c}d). \end{aligned}$$

[4] In our implementation, we do not use the Berkeley SIS "resub" command; rather we use the full power of the "simplify" command.

Substituting e into h, we get

$$h = c(a\bar{d}e + \bar{a}d) + d\bar{e}.$$

Substituting f into h, we get

$$h = \bar{f}(a(c + d) + dc).$$

Substituting g into h, we get

$$h = \bar{g}(d(b\bar{c} + \bar{a}) + \bar{b}c).$$

Substituting e, f, g into h we get

$$h = c\bar{f}\bar{g} + d\bar{e}.$$

For each target function, we can try substituting different sets of functions. Call such a set a **substitutional set**. For each substitutional set we generate a node function (or several functions). 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 shown on the right in Figure 7, substitution yields the network shown on the left with a cost of 33. With the ordering shown on the right in Figure 8, substitution yields the network shown on the left with a cost of 32. 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 9, with cost 26. This network is cyclic, with the dependency shown on the right. It is *not* combinational.

$$\begin{aligned} e &= a\bar{g}\bar{h} + \bar{a}d + \bar{b} \\ f &= \bar{a}\bar{g}\bar{h} + a(de + g) \\ g &= \bar{a}b\bar{c} + \bar{h}(c(a\bar{d} + \bar{b}) + \bar{a}d) \\ h &= \bar{a}cd + a(\bar{b}c\bar{d} + b\bar{c}d) \end{aligned}$$

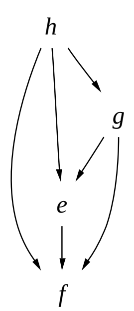


Fig. 7. Acyclic substitution order.

$$\begin{aligned} e &= a\bar{c}\bar{d} + d(\bar{a} + c) + \bar{b} \\ f &= \bar{a}\bar{g}\bar{h} + a(de + g) \\ g &= \bar{a}b\bar{c} + \bar{h}(a\bar{e} + \bar{a}d + \bar{b}c) \\ h &= c(a\bar{d}e + \bar{a}d) + d\bar{e} \end{aligned}$$

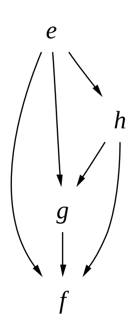


Fig. 8. Another acyclic substitution order.

$$\begin{aligned} e &= a\bar{g}\bar{h} + \bar{a}d + \bar{b} \\ f &= \bar{a}\bar{g}\bar{h} + a(de + g) \\ g &= \bar{a}\bar{f}\bar{h} + f(a\bar{e} + \bar{b}c) \\ h &= c\bar{f}\bar{g} + d\bar{e} \end{aligned}$$

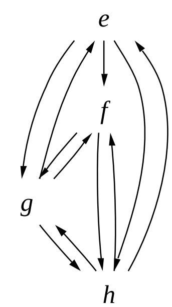


Fig. 9. Unordered substitution.

A cyclic soultion with cost 31 is shown in Figure 1. Its dependency graph is not as dense as that in Figure 9, and accordingly it is more costly. However, it may be verified according to the procedure in Section II that it is combinational.

The goal of the synthesis process is to select a choice of node functions that minimizes the cost while satisfying the condition for combinationality. We have explored several approaches, including dynamic programming and branch-and-bound algorithms (see [10]). Here we discuss the interplay of analysis and synthesis in the design process.

The analysis method described in Section II is formulated recursively. Accordingly, it lends itself well to the caching of analysis results for common sub-networks through iterations of the search. Suppose that in the course of our search for a low-cost combinational solution we consider a network $\mathcal{N}_1$ with node functions

$$f_1, \ldots, f_n.$$

Analysis for combinationality entails evaluating the expression $C(\mathcal{N}_1)$, given in Theorem 1. Next, suppose that we consider a network $\mathcal{N}_2$ with node functions

$$f_1', \ldots, f_n'.$$

Analysis entails evaluating $C(\mathcal{N}_2)$. Now suppose that some of the node functions in $\mathcal{N}_1$ are identical to those in $\mathcal{N}_2$. Let $\mathcal{S}$ be the subset of nodes that are equal:

$$\forall\, i \in \mathcal{S}, f_i \equiv f_i'.$$

The evaluation of $C(\mathcal{S})$ figures in both $C(\mathcal{N}_1)$ and $C(\mathcal{N}_2)$, and so it need not be repeated. If, in the process of evaluating $C(\mathcal{N}_1)$, we find that $C(\mathcal{S}) = 0$, then we rule out $\mathcal{N}_1$ as well as $\mathcal{N}_2$ (and all other networks that contain $\mathcal{S}$). Otherwise, we find that $C(\mathcal{S}) = 1$, and we need not re-evaluate it when evaluating $\mathcal{N}_2$ (or any other network that contains $\mathcal{S}$). We illustrate with examples.

Example 1

Consider again the example in Figure 1. Suppose that we have constructed the network for nodes f and g (assuming that nodes e and h are given) shown in Figure 10. Analysis

$$\begin{aligned} f &= \bar{a}\bar{g}\bar{h} + a(de + g) \\ g &= \bar{a}\bar{f}\bar{h} + f(a\bar{e} + \bar{b}c) \end{aligned}$$

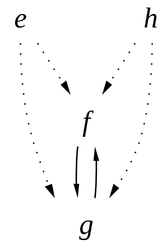


Fig. 10. A non-combinational component.

according to Theorem 1 tells us that this component is *not* combinational. Thus, we exclude this pair of node functions as candidates for f and g.

Example 2

Now suppose that we have constructed the candidates for nodes e, f, and g shown in Figure 11.

$$\begin{aligned} e &= \bar{f}(a\bar{h} + c) + d\bar{h} + \bar{b} \\ f &= \bar{a}\bar{d}\bar{g} + a(\bar{b}d + bc) \\ g &= \bar{a}b\bar{c} + \bar{h}(a\bar{e} + \bar{a}d + \bar{b}c) \end{aligned}$$

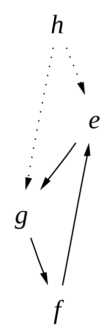


Fig. 11. A combinational component.

Analysis tells us that this component *is* combinational. We can proceed to select a candidate for h. The possibilities are

$$\begin{aligned} h_1 &= c(a\bar{d}e + \bar{a}d) + d\bar{e} \\ h_2 &= \bar{f}(a(c + d) + cd) \\ h_3 &= \bar{g}(d(b\bar{c} + \bar{a}) + \bar{b}c) \\ h_4 &= c\bar{f}(a + d) + d\bar{e} \\ h_5 &= \bar{f}\bar{g}(c + d) \\ h_6 &= c\bar{f}\bar{g} + d\bar{e}. \end{aligned}$$

When analyzing networks constructed with these candidates for h, we need not re-evaluate the component e, f, g from Figure 11. We find that h_2 combined with this component yields a combinational circuit (that shown in Figure 1).

IV. Results

From our experiments, we conclude 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. The dependency graph of the cyclic solution for one of the Espresso benchmark circuits, "exp", is shown in Figure 12.

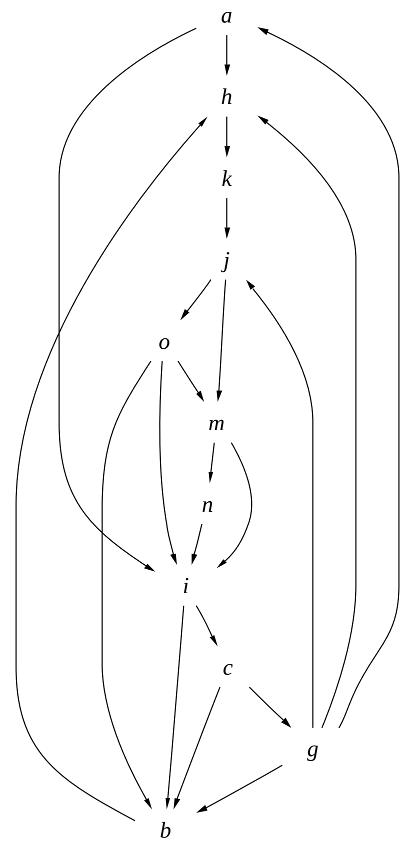


Fig. 12. Topology of network for Espresso benchmark circuit "exp" with 8 inputs, 18 outputs, and cost 262.

A. 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.

B. 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 13, we present those for which cyclic solutions were found. Since our goal is a proof-of-concept, only a very modest amount of computation was applied for the results presented here. For the larger circuits, the amount of improvement drops off due to time limits imposed on the search.

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 %

Fig. 13. Cost (literals in factored form) of Berkeley SIS `Simplify` vs. `Cyclify` for benchmarks.

V. Discussion

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 twice. 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 said, "it remains unknown to what extent feedback can yield economical realizations in general."

Twenty-five years later, the topic of incorporating feedback in the design of combinational circuits remained an open one, both in theory and in practice. Inspired by the work of Rivest, we 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.

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. With the symbolic framework presented here, the behavior of cyclic combinational circuits can be described in terms of successively smaller components. Circuits can be synthesized incrementally by adding combinational sub-components. Also, given an acyclic design we can re-synthesize the circuit by introducing feedback.

Our focus in the present work is on optimizating area. In future work, we will discuss issues related to timing and testing in the context of synthesis. On the practical side, we will incorporate the techniques into different synthesis environments, and report results for the decomposition and technology mapping phases.

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] R. E. Bryant, "Boolean Analysis of MOS Circuits," IEEE Trans. Computer-Aided Design, pp. 634–649, 1987.

[5] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," IEEE Trans. Computers, Vol. C-35, pp. 677–691, 1986.

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

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

[8] M. Mendler and M. Fairtlough, "Ternary Simulation: A Refinement of Binary Functions or an Abstraction of Real-Time Behavior, " M. Sheeran and S. Singh, ed., 3rd Workshop on Designing Correct Circuits, Springer Electronic Workshops in Computing, 1996.

[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, "The Synthesis of Cyclic Combinational Circuits," to appear in Design Automation Conf., 2003, available as Tech. Rep. ETR052, `http://www.paradise.caltech.edu/ETR.html`.

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

[13] 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.

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

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

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