Optimizing Designs Containing Black Boxes*†

Tai-Hung Liu Khurram Sajid Adnan Aziz
Electrical and Computer Engineering
The University of Texas
Austin TX

Vigyan Singhal
Cadence Berkeley Labs
Cadence
Berkeley CA

Abstract

We define a notion of equivalence for designs containing black boxes i.e., components whose functionality is not known; these arise naturally in the course of hierarchical design. Using this notion, we describe a sound and complete methodology for optimizing such designs.

1 Introduction

Modern gate-level hardware designs often contain "black boxes" (components whose functionality is not known). These black boxes can arise in many ways:

1. In hierarchical synthesis, components are recursively optimized from the "bottom-up" and then treated as fixed blocks.
2. The choice for the implementation of certain components may not have been made yet; this happens when different parts are designed separately.
3. A conscious decision may have been made not to synthesize certain components; these could be predefined blocks which have been already carefully designed and hand-optimized.

In the past, the approach taken for synthesis as well as verification of designs with black boxes has been to make the inputs to the black boxes primary outputs, and output of the black boxes primary inputs.

We will show that, for logic optimization, this approach is pessimistic in theory and in practice; the flexibility afforded by observability and controllability don't cares in the portion of the design to be synthesized is not fully used. Additionally, the fact that certain components may be instantiations of the same "variety" of black box, and consequently when presented with the same input are constrained to produce the same output is not used in this approach.

*Research supported in part by a DAC Graduate Fellowship.

To the best of our knowledge, there has been no past work on synthesizing designs with black boxes which has done more than treat the black box inputs and outputs as design outputs and inputs. In the verification community, Jones, Dill, and Burch have addressed the problem of verifying designs with "uninterpreted functions" (UIFs). These UIFs arise in the context of verifying complex operations in microprocessors, and provide a useful mechanism for abstraction [8]. They can be viewed in some sense as being black boxes. However, they are applied to complex datatypes (such as integers), and the decision procedure for verification in the presence of UIFs is based on a rewrite system [13], which is completely distinct from our approach, which is based on gate level designs and BDD based optimization.

The remainder of this paper is structured as follows: we begin in Section 2 by giving syntax and semantics to designs using finite state machines and netlists; this is extended to designs containing black boxes. In Section 3, we formulate the appropriate notion of equivalence for such designs. The inadequacy of existing approaches to synthesizing designs with black boxes is described in Section 4; we then formulate a sound and complete synthesis procedure using the concept of an "observability network", and present experimental results. We conclude with a summary of our contributions and suggestions for future work in Section 5.

2 Models for Hardware

There are two main formalisms for expressing designs, namely finite state machines and netlists. A detailed discussion on FSMs can be found in Hopcroft [6]. Netlists are "structural" and are more relevant to the present discussion.

2.1 Netlists

A netlist is a representation of a design at the *structural level*. We define two types of netlists, namely, *simple* and *complex*.

A *simple netlist* is a directed graph, where the nodes correspond to *primitive circuit elements* which could be gates, latches or primary inputs, and the edges correspond to wires connecting these elements. For simplicity, we will assume that the netlist is *Boolean* i.e., all variables take values in $\{0, 1\}$. A simple netlist will be referred to

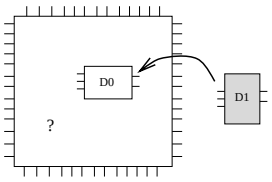

Figure 1: Plug-in plug-out replaceability.

as a *combinational* netlist if no latches occur in it; otherwise, it will be referred to as a *sequential* netlist.

A *complex netlist* (or simply, a *netlist*) is a similar to a simple netlist, with the addition of black boxes to the set of primitive circuit elements. Black boxes of the same variety are required to have the same number of inputs. A more detailed description of the syntax and semantics of netlists can be found in [7].

By definition, we require that there be no *combinational cycles*, where a combinational cycle is a cycle of gates. Issues related to combinational cycles have been dealt with elsewhere [3, 9].

3 Design Equivalence

If design D_1 is equivalent to design D_0 then the composition of D_1 with any environment should be equivalent to the composition of D_0 with the same environment; this is similar to the argument of plug-in plug-out replaceability given in [14] – see Figure 1.

Here we will discuss only about combinational designs. We can reason about sequential designs in a similar manner: details can be found in [7]. Since we require that there be no combinational loop in a design (a reasonable requirement), we also require that both composition and instantiation not create any loops as well.

First we consider the problem of equivalence for simple combinational netlists. Formally, let D_1 and D_2 be two simple combinational designs. Let the primary inputs of D_1 be $x_1, \ldots, x_n$ and the primary outputs be $y_1, \ldots, y_m$ similarly, let the primary inputs and outputs of D_2 be $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$.

Definition 1 *Two simple combinational designs D_1 and D_2 are equivalent if $f_{D_1} = f_{D_2}$, where f_{D_1} and f_{D_2} are the logic functions implemented by D_1 and D_2.*

We can now define equivalence for complex combinational designs:

Definition 2 *Two complex combinational designs D_1 and D_2 are equivalent if for any combinational instantiation μ of the black boxes appearing in D_1 and D_2, we have $f_{D_1[\mu]} = f_{D_2[\mu]}$.*

An example of complex combinational designs which are combinationally equivalent is given in Figure 2.

4 Synthesis

We now describe algorithms for optimizing designs with black boxes.

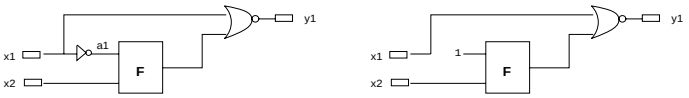

Figure 2: Combinationally equivalent complex combinational netlists: for both designs, when x_1 is 1, y_1 is zero and when x_2 is 0, $y_1 = F(1, x_2)$.

4.1 Traditional Approaches

Traditionally, designs with black boxes have been synthesized by making the inputs to the black boxes primary outputs, and output of the black boxes primary inputs, and synthesizing the simple logic [5, 1].

However, this approach is pessimistic for reasons given in Section 1. This is illustrated in Figure 2, in which gate a_1 can be safely replaced by the constant one, even though it is an input to the black box.

4.2 Sound and Complete Synthesis

Our approach to synthesizing combinational netlists will be to first identify all the flexibility available for synthesizing the simple portion of the netlists. This is then used to minimize the simple logic using existing logic optimization techniques. In particular, the notion of "don't cares" sets, i.e., inputs for which a gate can output any value carries over from logic synthesis on simple combinational networks [10]; the same is true of "compatible sets of permissible functions" [4, 11] is useful. The latter correspond to subsets of the complete set of don't cares for individual gates with the property that gates can be independently simplified with respect to these subsets, without requiring that don't cares to be recomputed.

Definition 3 *Let D be a complex combinational design. For a gate G in D on n inputs with function f_G, the input set $S \subset 2^n$ is a* don't care *set, if the gate function f_G can be replaced by any function f_G^* so that f_G^* takes the same values as f_G on any input $c \in \{0,1\}^n - S$ (on inputs from S, f_G^* is allowed to take arbitrary values), while preserving the equivalence of the resulting complex combinational design and D.*

By treating the inputs and outputs of the black boxes as primary inputs and primary outputs, we can use conventional methods to compute don't care sets for the gates of the design. However, as illustrated in Figure 2, this approach is suboptimal. In order to compute the full set of don't cares at a gate we use the concept of a *consistency network*.

Let D be a complex combinational network. We construct the consistency network as follows:

Step 1: Form a new netlist $D^{\texttt{PM}}$ from D: duplicate D to obtain $D_{\texttt{dup}}$ and combine D and $D_{\texttt{dup}}$ by merging corresponding primary inputs, and creating a single primary output which is 1 precisely when there is a

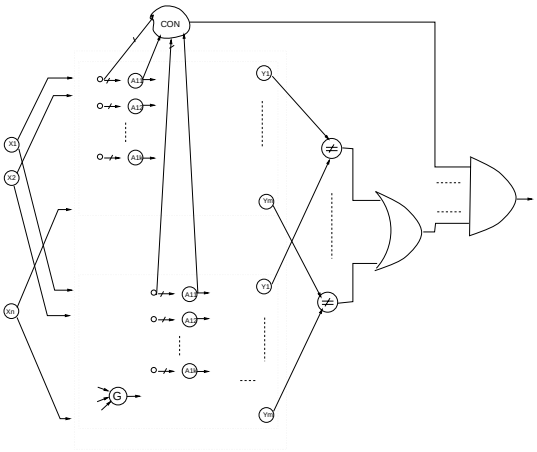

Figure 3: Redundancy removal for complex combinational netlists. The netlist depicted is $D^{\texttt{CHECKER}}$. The nodes marked A_{1j} are primary inputs derived from black boxes from single variety A; consistency logic has been shown for a single pair. (Consistency logic exists for pairs within a netlist too.)

primary output of D which is not equal to the corresponding primary output of $D_{\texttt{dup}}$.

Step 2: Replace all black boxes in $D^{\texttt{PM}}$ by primary inputs to form the simple combinational netlist $D^{\texttt{SIMP}}$.

Step 3: For each pair of primary inputs in $D^{\texttt{SIMP}}$ which correspond to black boxes of the same variety in $D^{\texttt{PM}}$, add to $D^{\texttt{SIMP}}$ a "consistency" gate, which outputs 1 exactly when the pair of primary inputs have the same value or the inputs to the corresponding black boxes from $D^{\texttt{PM}}$ take distinct values. Call this netlist $D^{\texttt{CONSISTENT}}$.

Step 4: Form the gate $G_{\texttt{CONSISTENT}}$ by taking the conjunction of all consistency logic nodes and the output of $D^{\texttt{CONSISTENT}}$. Add this gate to $D^{\texttt{CONSISTENT}}$; designate $G_{\texttt{CONSISTENT}}$ to be the only primary output. Call the resulting netlist $D^{\texttt{CHECKER}}$; we will refer to $D^{\texttt{CHECKER}}$ as the *consistency network*.

The result of this construction is illustrated in Figure 3; note that the resulting netlist is a simple combinational netlist. It should be clear from the construction that the output of $D^{\texttt{CHECKER}}$ is 0 for any input.

4.2.1 Logic Optimization

We claim that the consistency network embodies all the flexibility available for synthesis. In order to illustrate this claim, we consider a simple yet surprisingly powerful global optimization technique known as *redundancy removal* [12]. This consists of identifying gates which can be replaced by a constant valued gate, while ensuring that the resultant design is equivalent to the original design. These constants are subsequently used to simplify the logic.

The concept of redundancy removal can be extended to complex combinational netlists:

Definition 4 *A gate is stuck-at-1(0) redundant in a combinational complex netlist when it can be replaced by a gate taking the constant value* $1(0)$ *and the resulting netlist is equivalent to the original netlist, where the notion of equivalence is that for complex combinational netlists, as given in Definition 2.*

The following theorem (the proof of which is given in [7]) demonstrates that we can perform redundancy removal on the simple logic associated with D by performing redundancy removal on the nodes corresponding to D in the consistency network.

Theorem 4.1 *Let* α *be any gate in* D*. Then* α *is s-a-1(0) redundant if and only if the node corresponding to* α *in* $D^{\texttt{CHECKER}}$ *is s-a-1(0) redundant.*

More generally, one can compute don't cares which can be used to simplify the individual gates in D. It is tedious but straightforward to prove an analog of Theorem 4.1 to the effect that the don't cares computed for the gate corresponding to α in $D^{\texttt{CHECKER}}$ are exactly the don't cares for α in D. This gives us a mechanism for computing the don't cares for D.

After simplifying gate G in D, the don't cares for other gates may have changed. Thus it is necessary to recompute the don't cares for the remaining gates, which is potentially expensive computationally. Almost the same degree of optimization can be achieved using the concept of *compatible don't cares* [4, 11]. These don't cares can be used independently to optimize the gates; they too can be directly computed from the network $D^{\texttt{CHECKER}}$.

4.3 Experiments

In this section we report experimental results on the synthesis of complex combinational netlists; these experiments were performed in the SIS environment [12].

Specifically, we took several ISCAS benchmark circuits, and "cut out" a region of the logic internal to the design; this was treated as a black box. We then formed the consistency network, computed the compatible observability don't cares for the gates as described in Section 4.2.1, and simplified the functions for the gates using these don't cares [2]. All this was relatively simple to achieve, since we were able to use the existing code for the `full_simplify` command in SIS which computes compatible observability don't cares; we simply restricted the nodes chosen to be simplified in the consistency network to be from D and not from $D_{\texttt{dup}}$ or the consistency logic.

In Table 1 we provide a comparison of our procedure with the conventional approach of making the black box

Benchmark	Initial size	*Conventional Optimization*			*Complete Optimization*		
	(literals)	Reduction	Time (sec)	BDD Size	Reduction	Time (sec)	BDD Size
pm1	144	62	0	12	84	1	12
b9	338	124	1	52	138	4	63
i4	496	156	1	44438	318	6	44438
s208	206	34	1	35	37	2	1901
9symml	720	446	4	58	512	10	58
cordic	266	86	1	139	95	3	175
apex6	1869	988	7	454	1054	47	256
comp	404	238	0	21128	291	3	166125

Table 1: Conventional vs. complete logic optimization.

inputs and outputs primary outputs and primary inputs. For each design, we report the initial size of the circuit, the literal savings after optimization, the time taken for optimization, and the size of the largest BDD built in the course of computing the don't cares [11].

Not surprisingly, the experiments show that there is more reduction to be achieved by using the new procedure; in many cases, this difference is substantial, e.g., `i4`.

However, the true significance of these results is not the amount of reduction offered; these examples do not reflect the kind of hierarchical designs our procedure is suited to. The main point to note is that the penalty in running time and memory usage is not significant. Thus we feel our procedure should take the same order of magnitude of running time as existing synthesis routines on more realistic designs, while providing substantial improvements to the quality of the resulting design.

5 Conclusion

To summarize, we have met our goal of establishing a sound and complete methodology for optimizing designs containing black boxes. We formalized the notion of a design containing black boxes, developed criterion for equivalence, and characterized all the flexibility available for synthesizing such designs; we pointed out the limitations in existing approaches. Preliminary experiments performed indicate that the additional flexibility can be useful for optimization, and the increased time taken for synthesis is acceptable. We have been offered a large hierarchical industrial design which contains many uninstantiated components; we are working on applying our procedure to it.

In the long term future, we would like to relate our results, which stem from work on logic synthesis, to high-level synthesis. We feel existing high-level synthesis procedures lack a well defined characterization of the set of permissible implementations; we believe we can contribute to an enhanced understanding of these issues.

References

[1] D. P. Appenzeller and A. Kuehlmann. Formal Verification of a PowerPC Microprocessor. In *Proc. Intl. Conf. on Computer Design*, pages 79–84, Austin, TX, October 1995.

[2] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[3] J.R. Burch, D. Dill, E. Wolf, and G. DeMicheli. Modeling Hierarchical Combinational Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[4] M. Damiani and G. De Micheli. Don't care Set Specifications in Combinational and Synchronous Logic Circuits. Technical Report CSL-TR-92-531, Stanford University, Computer Systems Laboratory, Stanford, CA 94305-4055, July 1992.

[5] L. Stok et al. BooleDozer: Logic Synthesis for ASICs. *IBM J. Res. and Devep.*, pages 407–430, July 1996.

[6] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[7] Tai-Hung Liu. `church.ece.utexas.edu/~tai/dac-bb-97.ps`.

[8] David E. Long. *Model Checking, Abstraction and Compositional Verification* . PhD thesis, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[9] S. Malik. Analysis of Cyclic Combinational Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(7):950–956, July 1994.

[10] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[11] Hamid Savoj. *Don't Cares in Multi-Level Network Optimization*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[12] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proc. Intl. Conf. on Computer Design*, pages 328–333, October 1992.

[13] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, 1979.

[14] Vigyan Singhal. *Design Replacements for Sequential Circuits*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, 1996.