Incremental Synthesis

Daniel Brand, IBM Research Division, Yorktown Heights, NY
Anthony Drumm, IBM AS/400 Division, Rochester, MN
Sandip Kundu, IBM Research Division, Yorktown Heights, NY
Prakash Narain, IBM Microelectronics, Endicott, NY

Abstract

A small change in the input to logic synthesis may cause a large change in the output implementation. This is undesirable if a designer has some investment in the old implementation and does not want it perturbed more than necessary. We describe a method that solves this problem by reusing gates from the old implementation, and restricting synthesis to the modified portions only.

1. Introduction

It is common for a designer to have an investment in the implementation of his design. Examples of such investment are effort to synthesize, expense of physical design, mask generation, any manual changes, or simply designer's time spent understanding the implementation. This investment may be jeopardized if the designer has to modify the specification. Such modifications, commonly called "engineering changes" or ECs, are necessitated by changes in requirements, errors, or efforts to speed up the logic. If the designer synthesizes his new specification then he may get a completely different implementation, because synthesis tries to find a minimal representation of the new function, and a small change in the specification of a function may cause a large change in its optimal implementation. Moreover, synthesis systems are heuristics, making some "arbitrary" choices, which further contribute to large changes in the implementation.

For this reason it is a common practice to "freeze" a design, meaning that after a design is sufficiently stable, synthesis is no longer used, and any modifications are done manually in both the specification and the implementation. This is undesirable for three reasons. First, it is very time-consuming for a designer to understand how a synthesized implementation relates to his original specification. Secondly, it is very easy for the designers to make a mistake in modifying the implementation. Thirdly, the modified implementation may be wrong even if no mistake is ever made. The reason is that some optimization performed on the old version might be invalid for the new function.

Therefore, several approaches exist for automating this process. These approaches differ by their objectives. Some methods restrict themselves to a specific set of ECs [7,8]. Others have the goal of preserving the final layout, and allowing modifications only around the perimeter of the implementation [6,10]. There are several methods with the same goal as ours, namely, reusing as many gates from the old implementation as possible. An example is [9], whose method is applicable if synthesis does not change logic structure. The method closest to ours is that of [5], which can handle arbitrary structural differences. The main distinguishing feature of our method is a preprocessing step calculating correspondence between the new specification and the old implementation, which then gives us not only efficiency, but allows greater reuse of the old implementation.

2. Methodology

Regular (i.e., non-incremental) synthesis has the following steps, which are identical for all versions of a design:

READ(Specification)
REGULAR SYNTHESIS
WRITE(Implementation)

In case of incremental synthesis we have an old and a new version of a design. The old version is given by Specification0 (e.g., -- Figure id 'oldsrc' unknown --) and Implementation0 (e.g., -- Figure id 'oldimpl' unknown --). The new version is given by Specification1 (e.g., -- Figure id 'newsrc' unknown --). We will assume all three to be given as gate networks, which is the normal representation for logic synthesis. Correspondence between primary IOs and other nets is indicated by the common letter forming the name. (This commonality of names is used for exposition only; our programs do not rely on names.)

For version1 we wish an Implementation1 with the functionality of Specification1, that reuses as much of Implementation0 as possible. We use the following steps:

FUNCTIONAL_CORR(Specification0, Implementation0)
READ(Specification1)
STRUCTURAL_CORR(Specification0, Specification1)
INCREMENTAL SYNTHESIS
REGULAR SYNTHESIS
POST_PROCESSING
WRITE(Implementation1)

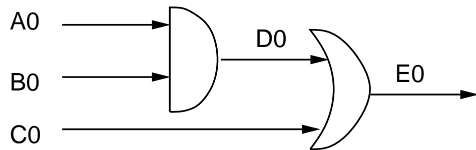

Figure 1. Specification0

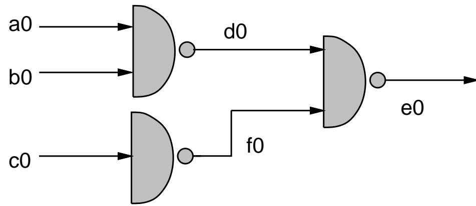

Figure 2. Implementation0

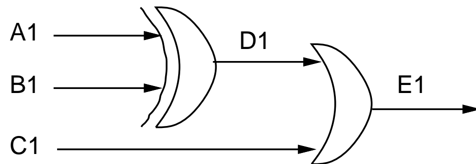

Figure 3. Specification1

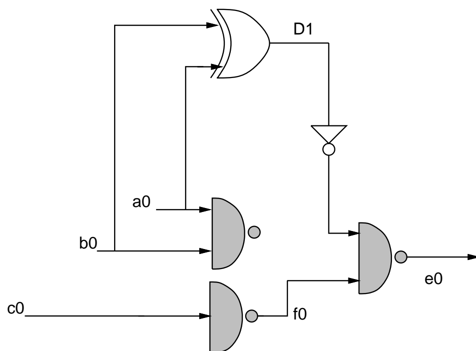

Figure 4. Result of incremental synthesis

The main subject of this paper is the step INCREMENTAL SYNTHESIS (described later), which forms a preliminary implementation of Specification1 (see Figure 4). This preliminary implementation has three kinds of gates -- some gates from Implementation0 (the shaded NAND gates), some gates from Specification1 (the XOR gate) and some inverters needed to connect the first two kinds of gates. We will refer to the gates from Implementation0 (shaded in our figures) as "old gates" and all the others as "new gates". Our objective is to minimize the number of new gates. In particular, primary outputs that retain their specification should also retain their implementation.

Once this preliminary implementation is formed it is run through regular synthesis (including optimization, technology mapping, etc.) in order to process the new gates. During this regular synthesis all the old gates are marked as protected from any synthesis transformations, and remain unchanged.

The result of regular synthesis must be run through a post-processing step, where the protection is removed from the old gates for several reasons. A primary output, which retains its old implementation, may nevertheless have different delay because of different loading of some nets. Also some of its faults might become untestable. The post-processing step may be needed to adjust power levels and improve testability.

3. Correspondence calculation

Our general approach to incremental synthesis tries to identify pieces of logic in Specification1 (e.g., the OR gate) that can be replaced by pieces of logic from Implementation0 (the two NAND gates) without altering the function of Specification1. This is, in general, a difficult problem, because given a piece of logic in Specification1, it is not clear what would be good candidate logic in Implementation0 for the replacement. It is not clear because there is in general no relationship between Specification1 and Implementation0. First of all, they are functionally different; they may even have different primary IOs. Secondly, they are structurally different because synthesis tends to make drastic logic restructuring. In order to derive a correspondence between Specification1 and Implementation0 (see -- Figure id 'diagram' unknown --), we use Specification0, which is functionally related to Implementation0 (both should implement the same function), and it is also structurally related to Specification1, (in case of only an incremental change to the specification, the two specifications "look" similar).

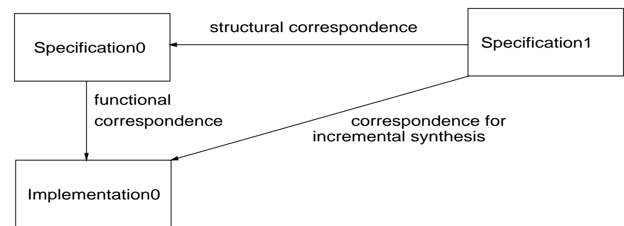

Figure 5. Correspondences

To derive *structural correspondence* between Specification0 and Specification1 we use the algorithm of [1]. For each net X1 in Specification1 that algorithm computes a net X0 in Specification0, which structurally corresponds to X1 (if such an X0 exists). (In Figure 1 and Figure 3 corresponding nets are indicated by the same letter.) Structural correspondence is a mapping from nets and gates of Specification1 to nets and gates of

Specification0, where corresponding primary IO must be mapped to each other. The algorithm [1] tries to maximize the number of gates that are in "agreement". A gate G1 is in complete agreement with a corresponding gate G0 if they are of the same type, and are connected to corresponding nets. If two cones are isomorphic then all their gates can be put in complete agreement, which is important to detect, so that incremental synthesis can preserve the implementation of unmodified functions. In addition, we try to calculate correspondence that will also put other gates in complete or partial agreement. Two gates are in only a partial agreement, if they differ in function or some of their nets do not correspond to each other. (Even partial agreement is useful in minimizing the impact of an EC.) In our example of Figure 1 and Figure 3 the two OR gates are in complete agreement, while the XOR gate is in partial agreement with the AND gate.

To derive the *functional correspondence* between Specification0 and Implementation0 we use the algorithm of [2]. That algorithm computes for each net X0 in Specification0 a net x0 in Implementation0, which functionally corresponds to X0 (if such an x0 exists). That does not mean that X0 and x0 have the same function, merely that the function of x0 can replace the function of X0 without altering the functionality of Specification0. (In Figure 1 and Figure 2 corresponding nets are indicated by the same letter, but we use small case for names in the implementation.) Since implementations tend to use negative logic, it is common that X0 cannot be replaced by x0, but it may be replaceable by $\overline{x0}$; therefore our algorithm, in addition to computing x0 also computes the right polarity. (In our example, the correspondence <D0, d0> is negative.)

4. Incremental synthesis algorithm

Having calculated the structural and functional correspondences we compose them by transitivity to obtain a correspondence between the nets of Specification1 and Implementation0. (The correspondence is calculated before incremental synthesis starts and does not change during the incremental synthesis algorithm.) When a net X1 in Specification1 corresponds to a net x0 in Implementation0 then it is likely that x0 (or $\overline{x0}$) can functionally replace X1. We could be assured of the legality of such a replacement if Specification0 and Specification1 were identical and if the replacements were carried out in the order calculated by functional correspondence. However, since the two specifications are not identical we must check it explicitly and make the replacement only if it does not change the function of Specification1.

Incremental synthesis is done in three phases. Phase I deals with primary inputs and latches, Phase III deals with primary outputs and latches, while Phase II handles the combinational logic in between. For lack of space we will explain the algorithm only in the simplest case, where there are no latches and the new and old versions have the same primary IOs.

Phase I places Specification1 and Implementation0 together, sharing primary inputs only (see -- Figure id 'after1' unknown --). The resulting network still has two sets of primary outputs.

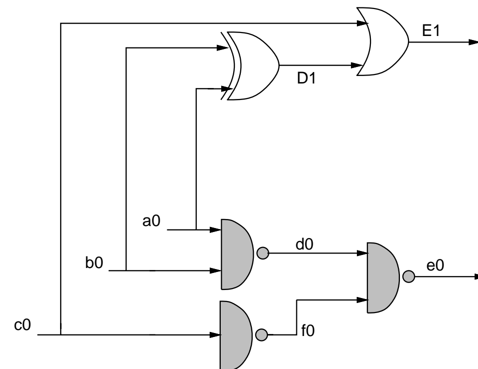

Figure 6. After Phase I

Phase II takes the two pieces of logic sharing primary inputs only, and introduces more sharing between them. The result of Phase II is a network, where Specification1 and Implementation0 are interconnected, and some pieces of logic are left unused. Phase II may change the functionality of the primary outputs for Implementation0 (e0), but it must preserve the functionality of the primary outputs of Specification1 (E1).

Phase II proceeds through the nets of Specification1 in topological order from inputs to outputs. Given a net X1 in Specification1, the composition of structural and functional correspondence gives us a corresponding net x0 in Implementation0. We check whether x0 may replace X1 without changing the functionality of Specification1 (using the algorithm of [2]).

Case A: If the answer is "yes", then the net x0 (or $\overline{x0}$) gets connected wherever X1 used to be connected.

Case B: If the answer is "no", then the net X1 (or $\overline{X1}$) gets connected wherever x0 used to be connected.

Case A (the desirable case) represents the situation of logic X1 that was not modified and thus can use the gates of x0. The cone of X1 becomes unnecessary. Case B represents logic X1 that was modified and thus cannot use the gates of x0. In Case B, replacing x0 with X1 makes it likely that as the algorithm proceeds it will encounter Case A further on.

In our example, we first apply Case B to <D1, d0> with negative polarity (see -- Figure id 'afterB' unknown --). This step effectively replaces a NAND gate from Implementation0 by the XOR gate from Specification1; the XOR gate is connected through an inverter because the correspondence between D1 and d0 is negative. Then we apply Case A to <E1, e0> (see -- Figure id 'afterA' unknown --). -- Figure id 'afterA' unknown -- shows the network after Phase II is finished.

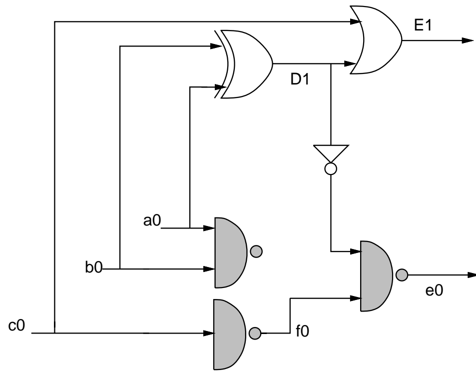

Figure 7. After Case B applied to <D1, d0>

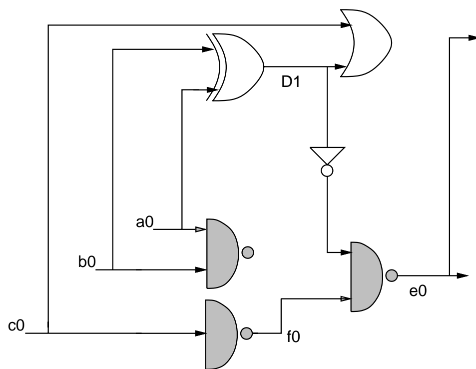

Figure 8. After Case A applied to <E1, e0>

It should be clear that after Case A, the functionality of Specification1 is preserved. After Case B, it is certainly possible for the functionality of Implementation0 to change, which is all right. However, the way we described it above, it is also possible for the functionality of Specification1 to change, which is not all right. The reason is that several nets in Specification1 may correspond to one net in Implementation0 and more generally, proceeding in topological order in Specification1 does not imply topological order in Implementation0. Case B replacing some x0 by X1 may be preceded by Case A replacing some Y1 with x0. More generally, it may be preceded by Case A replacing some Y1 with y0, where y0 is in the transitive fanout of x0.

To guarantee that Case B will never change the functionality of Specification1, the net X1 may not replace connections of x0 to new gates. More generally, after performing Case A on some Y1 and y0 we "commit" the whole transitive fanin of y0. That means, all the input pins in the transitive fanin of y0 are marked as committed and may not be changed. Then in Case B net X1 replaces the connections of x0 only at those pins that are not committed. The net x0 remains connected to the committed pins.

This mechanism of committing a cone of logic is also used to ensure that the new implementation of unmodified primary outputs is identical to the old one. In Phase I we commit the whole transitive fanin of any primary output whose cone is isomorphic in the two specifications (identified by structural correspondence).

At the end of Phase II we are still left with two sets of primary outputs. Each pair of corresponding primary outputs ends up connected to the same net (e0 in Figure 8). To guarantee this property, the correspondence calculation makes sure that primary IOs correspond to each other and no other nets. Phase III simply deletes all the primary outputs of Specification1. The result is a network with the functionality of Specification1 and with as many gates of Implementation0 as introduced by Case A. As explained in Section 2 the network of Figure 8 has to be run through regular synthesis, where all the shaded gates are protected from any transformations. Figure 4 shows the result of simplifying away one unused gate; the shaded unused gate will be also deleted, but is left in Figure 4 for purposes of exposition only.

5. Experimental results

In TABLE 1 we show results for several experiments. All of the circuits were taken from production parts, with the exception of C6288, which was taken from [3] and random change was introduced. (We included C6288 so that the reader can relate the other pieces of logic to something that has been discussed in the literature.)

The column SOURCE LINES shows the difference between new and old version before synthesis: number of lines in old source \ number of lines reused in new source \ number of new or different lines. These numbers are as seen by the designer, rather than synthesis. For example, the first two circuits have the same source data, but the first one had different preprocessing applied to the old and new versions, which presents different looking pieces of logic to incremental synthesis.
The column IMPLEMENTATION GATES indicates the amount of reuse in the implementation: number of gates in Implementation0 \ number of gates reused in Implementation1 \ number of new gates in Implementation1. The last number refers to new gates introduced by incremental synthesis and then processed by regular synthesis and the post-processing step.

Under the column WITH \ WITHOUT INCREMENTAL SYNTHESIS we are comparing two implementations of Specification1 -- incremental synthesis followed by regular synthesis for new gates (including post-processing), versus regular synthesis for all the gates of Specification1. The columns AREA and SLACK show how much area and delay we are sacrificing by wanting the new Implementation1 to be as similar to the old as possible. Under CPU TIMES we are reporting the time to calculate functional correspondence, and separately the time to do the incremental synthesis together with structural correspondence. We are singling out the

computation of functional correspondence because it can be done off line before the new version exists, and thus need not impact the design cycle. All CPU times are scaled to a 100 MHz machine cycle.

From the experimental results we see that the amount of gate reuse is related to the amount change in the specification, which is designers' expectation. Except in two circuits, we get an area penalty, which is caused by preventing any simplification between old and new gates. Sometimes we also get a delay penalty; this is again caused by no simplification between old and new gates.

Contrary to designers' expectations no significant saving in CPU time occurs; in fact by using incremental synthesis CPU time may increase. There are several reasons for this. First of all, substantial amount of CPU time is spent in calculating the functional correspondence, as well as the incremental synthesis step itself. Secondly, regular synthesis spends a lot of CPU time even if it needs to synthesize only very few new gates. (It spends this time on calculating information (e.g. timing) about all the old gates.) In order to reduce the turn-around time for incremental synthesis it is important to calculate the functional correspondence immediately after synthesizing the old version, without waiting for the new version.

6. Conclusions

We have presented an automated way of performing incremental synthesis. Its main benefit lies in reducing the design cycle, which happens in several ways. First, it automates the implementation of specification changes late in the design cycle. Secondly, it allows a designer to make modifications (e.g. speed up) in one area without changing the implementation of areas he is already happy with. And thirdly, it preserves the information from the old version (e.g. net names), which helps in analyzing the new implementation.

Acknowledgements

L. Stok, L. Trevillyan, R. Damiano, M. Berkelaar, R. Bergamashi, I. Spillinger, W. Kunz read the manuscript and made many helpful suggestions. We are especially grateful to Andreas Kuehlmann for valuable discussions. Lakshmi Reddy made an important contribution to the test generator, on which this approach is based.

References

[1] D. Brand, "The Taming of Synthesis", *International Workshop on Logic Synthesis*, RTP, May 1991.

[2] D. Brand, "Verification of Large Synthesized Designs", *Proc. of ICCAD*, November 1993, pp. 534-537.

[3] F. Brglez, P.Pownall, R. Humm, "Accelerated ATPG and Fault Grading via Testability Analysis", *IEEE International Symposium on Systems and Circuits*, June 1985, pp. 695-698.

[4] P.Y. Chung, I.N. Hajj, "ACCORD Automatic Catching and CORrection of Logic Design Errors in Combinational Circuits", *International Test Conference*, September 1992.

[5] M. Fujita, T. Kakuda, Y. Matsunaga, "Redesign and Automatic Error Correction of Combinational Circuits", *Logic and Architecture Synthesis*, ed. G. Saucier, North-Holland: Elsvier Science Publishers B.V., pp. 253-262.

[6] M. Fujita, Y. Matsunaga, K.C. Chen, "On Application of Boolean Unification to Combinational Logic Synthesis", *Proc. of ICCAD*, November 1991, pp. 510-513.

[7] J.C. Madre, O. Coudert, J.P. Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM", *Proc. of ICCAD*, November 1989, pp. 30-33.

[8] I. Pomeranz, S.M. Reddy, "On Diagnosis and Correction of Design Errors", *Proc. of ICCAD*, November 1993, pp. 500-507.

[9] T.Shinsha, T. Kubo, Y. Sakataya, J. Koshishita, K. Ishihara, "Incremental Logic Synthesis Through Gate Logic Structure Identification", *Proc. of DAC*, June 1986, pp. 391-397.

[10] Y. Watanabe, R.K. Brayton, "Incremental Synthesis for Engineering Changes", *Proc. of ICCAD*, November 1991, pp. 40-43.

NAME	AMOUNT OF CHANGE OLD \ REUSED \ NEW		WITH \ WITHOUT INCREMENTAL SYNTHESIS					
			QUALITY		CPU TIMES (SEC)			
	SOURCE LINES	IMPLEMENTATION GATES	AREA (CELLS)	SLACK (NS)	FUNCT. CORRESP.	INCREM. SYNTHESIS	REGULAR SYNTHESIS	TOTAL
A1	7315 \ 7314 \ 1	973 \ 800 \ 251	11498 \ 10245	-0.8 \ -0.8	232 \ 0	124 \ 0	698 \ 1110	1054 \ 1110
A2	7315 \ 7314 \ 1	973 \ 952 \ 3	9806 \ 9949	-0.8 \ -0.9	232 \ 0	403 \ 0	234 \ 855	869 \ 855
B	1440 \ 1439 \ 1	2676 \ 2675 \ 2	6622 \ 6621	-1.7 \ -1.7	128 \ 0	244 \ 0	315 \ 806	687 \ 806
C	2514 \ 2492 \51	940 \ 895 \ 134	3238 \ 3092	-0.9 \ -0.8	26 \ 0	40 \ 0	207 \ 561	273 \ 561
D1	12792 \12785 \19	4199 \ 4137 \ 107	40116 \ 39491	-1.0 \ -1.0	3718 \ 0	1168 \ 0	1275 \ 4485	6161 \ 4485
D2	12804 \12795 \23	4124 \ 4077 \ 93	39938 \ 38937	-1.2 \ -0.9	3588 \ 0	1449 \ 0	1244 \ 4381	6281 \ 4381
D3	12792 \12779 \41	4199 \ 4152 \ 64	39715 \ 38937	-1.0 \ -0.9	3718 \ 0	1283 \ 0	1192 \ 4381	6193 \ 4381
C6822	6735 \ 6734 \ 1	2210 \ 2210 \ 0	7018 \ 7018	-5.9 \ -5.9	51 \ 0	108 \ 0	209 \ 1458	368 \ 1458

TABLE 1: Results of incremental synthesis