Eliminating False Loops Caused by Sharing in Control Path *

Alan Su† Ta-Yung Liu† Yu-Chin Hsu† Mike Tien-Chien Lee‡

†Department of Computer Science
University of California
Riverside, CA 92521

‡Fujitsu Laboratories of America
3350 Scott Blvd., Bldg. #34
Santa Clara, CA 95054

Abstract

In high level synthesis, resource sharing may result in a circuit containing false loops that pose great difficulty in timing validation during design sign-off phase. It is hence desirable to avoid generating any false loops in a synthesized circuit. Previous work [1, 2] considered mainly data path sharing for false loop elimination. However, for a complete circuit with both data path and control path, false loops can be created due to control logic sharing, even though the loops caused by data path sharing have all been removed. In this paper, we present a novel approach to detect and eliminate the false loops caused by control logic sharing. An effective filter is devised to reduce the computation complexity of false loop detection, which is based on checking the level numbers that are propagated from data path operators to inputs/outputs of the control path. Only the input/output pairs of the control path identified by the filter are further investigated by traversing into the data path for false loop detection. A removal algorithm is then applied to eliminate the detected false loops, followed by logic minimization to further optimize the circuit. Experimental results show that for nine example circuits we tested, the final designs after false loop removal and logic minimization give only slightly larger area than the original ones that contain false loops.

1. Introduction

A false path is a combinational path which will never be activated during circuit execution. A false loop is a special case of false path where the starting and ending points of the false path are identical. A circuit containing false loops is not timing analyzable because most timing analysis tools cannot handle false loops to evaluate the circuit's clock period. Therefore, a designer has to manually identify all the false loops and mask them in order to complete timing validation. For a circuit produced by automatic synthesis, this task can become very difficult because the designer has no clue as where the false loops are. So, it is desirable for a synthesizer to generate circuits which are false-loop-free.

In [1, 2], methods were proposed to generate a false-loop-free data path by considering only data path sharing during high level synthesis. Stok [1] developed a *chain annotated compatibility graph* to constrain resource sharing for false loop elimination, at the cost of extra functional units. Huang *et al.* [2] addressed this problem using the concept of *delayed binding*. Delay binding technique eliminates false loops in scheduling phase and guarantees the data path generated satisfies the resource constraint. If scheduling an operation into the current control step produces a false loop, the operation will be "delayed" until next control step. This approach may introduce extra control steps in order to find a false-loop-free data path.

The two algorithms above tackle the false loop problem only with resource sharing in the data path. However, for a complete circuit with both data path and control path, false loops can still be created due to control logic sharing, even though the loops caused by data path sharing have all been removed. This is because, for a false-loop-free data path circuit generated by [1] or [2], control logic synthesizer will perform logic sharing using such information as don't cares, which introduces the possibility of creating false loops. This don't care information allows an output signal of the controller to be realized by reusing, or sharing, some logic of other independent control path functions for minimization purpose. For example, for an input signal x which is don't care to an output signal s which depends only on y, control logic synthesis can implement s as a function of y as well as x if the function of s is still correct and the resulting logic is minimized. In this case, if there is already a combinational path from the control output s through the data path to the control input x, a loop will be created after control logic synthesis. However, because x is don't care to s, this loop will never be functionally activated. Since such false loops are

* This work has been supported in part by the UC MICRO, Fujitsu Labs. of America, Quickturn co., and National Semiconductor Inc.

caused by sharing of random logic in the control path, they are more difficult to detect and remove than those in the data path.

This kind of control sharing false loop can be generated by most high-level synthesis tools. Because data path (computation) and control path (random logic) have different design characteristics, usually high-level synthesis tools seperate the design into data and control paths to apply different synthesis algorithms for optimization. When data path and control unit are optimized seperately, there is no way to tell if there are false loops accross data path and control unit.

This paper proposes a novel approach to detect and eliminate such false loops caused by sharing in the control path. We start with a false-loop-free register-transfer level (RTL) design obtained by the algorithm in [1] or [2]. Therefore, any possible false loop created subsequently in the circuit can only be due to control logic sharing. There can be two possible strategies to solve this problem. The first approach is to devise logic synthesis so as to prevent the false loop due to logic sharing from happening. The second strategy is to detect all such false loops and remove them efficiently. The first approach requires logic synthesis to check false loops through the data path at every optimization step, which requires very expensive global computation. In this paper the second approach, false loop detection and elimination, is adopted. Since there can be a lot of such false loops and detecting each of them by traversing the circuit can be also computationally expensive, a filter is devised to effectively reduce the computation complexity of detection. The filter is based on checking the topological level numbers that are assigned to each data path operators and propagated to each input and output of the control path. Only the input/output pairs of the control path identified by the filter are further investigated by traversing the circuit for false loop detection. A removal algorithm is then applied to eliminate the detected false loops, followed by logic minimization to further optimize the circuit. Experimental results show that for the nine example circuits we tested, the final designs after false loop removal and logic minimization give only slightly larger area than the original ones that contain false loops.

This paper is organized as follows. Section 2 classifies two types of false loops and introduces the false loop problem caused by control logic sharing. Section 3 proposes our approach for solving the false loop problem caused by control logic sharing. Section 4 provides the experimental results using the proposed approach on nine example circuits. Section 5 gives the conclusion.

2 Classification of False Loops

There are two types of false loops – those caused by resource sharing in data path and those caused by resource sharing in control path. We will briefly overview the false loop problem caused by data path sharing in Section 2.1, and focus on the problem caused by control path sharing in Section 2.2 and the remainder of the paper.

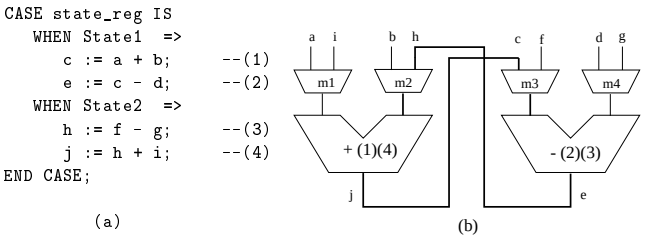

Figure 1. False loop caused by data path sharing and contained within the data path only: (a) circuit description; (b) its synthesized data path.

2.1 False Loops Caused by Sharing in Data Path

Resource sharing by data path operations may create false loops which span either only within the data path or to the control unit [1]. Figure 1 shows an example where the false loop is contained within the data path. The behavioral description in Figure 1(a) is synthesized into the data path in Figure 1(b) under the resource constraint of one adder and one subtracter. However, due to resource sharing by the data path operations during synthesis, a false loop, as indicated in thick line in Figure 1(b), is created in the data path. On the other hand, sharing in data path can also create false loops that span across both data path and control path. This is because the output of some data path operator, such as a comparator, feeds back to the control path which also controls the execution of the same operator. Effective algorithms have been proposed to eliminate such false loops caused by data path sharing during either allocation [1] or scheduling [2].

2.2 False Loops Caused by Sharing in Control Path

Although the data path synthesis algorithms proposed in [1, 2] can derive a false-loop-free data path, subsequent control path synthesis must guarantee, when connecting the synthesized controller to the data path, the entire circuit be false-loop-free as well. Otherwise, false loops can still be introduced by logic sharing in the control path itself. Furthermore, such false loops caused by sharing of random logic are even more difficult to detect and remove than that in the data path. It is therefore important for a control path synthesis tool to ensure no false loops be created.

Creation of false loops by sharing in control path is because of don't care information used by control path synthe-

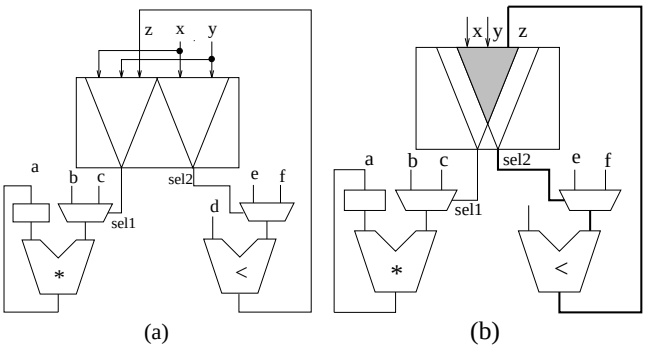

Figure 2. (a) Example of false-loop-free RTL design; (b) false loop introduced by control logic sharing.

sis. In a control path, if there exists a portion of logic that can be shared by other control functions, a single copy of the shared logic can be used to implement the control functions. Such control path sharing occurs during control logic synthesis, which is illustrated by the example in Figure 2. In Figure 2(a), sel1 and sel2 are two independent control functions before logic synthesis, as is indicated by the two non-overlapping logic cones, and there are no false loops in the circuit. Suppose control input z is don't care to sel2. Then after logic synthesis, sel1 and sel2 may share some portion of logic, as shown by that the shaded area in Figure 2(b). A loop is therefore created, from the output of the adder through the control unit and the multiplexer to the same adder, as shown in thick lines. Since z is don't care to sel2, the path from z to sel2 will never be functionally sensitized, or activated, which makes the loop a false one.

A concrete example of control path is used below to demonstrate how false loops can be created by control path sharing. Suppose the the control path has three control output functions:

$$\begin{aligned} \mathsf{OUT1} &= u(z+w) \\ \mathsf{OUT2} &= v(\bar{z}+w) \\ \mathsf{OUT3} &= uvw \end{aligned}$$

and OUT3 has a combinational path to z in the data path. Although OUT3 is defined by the three inputs u, v, and w, control logic synthesis can implement its function by production of OUT1 and OUT2 because $u(z+w)\cdot v(\bar{z}+w)$ equals to uvw. Such logic sharing makes z also an input of OUT3 even though z is don't care to the computation of OUT3. A path from z to OUT3 is hence created but will never be sensitized. So, a false loop containing this path is introduced by control logic sharing.

3 The Proposed Approach

The goal of our proposed approach is to detect and eliminate false loops caused by control logic sharing for a given false-loop-free RTL design. This approach isolates the problem from logic synthesis thus can easily adopted by any logic synthesis tools. The other approach, by enhancing logic synthesis to prevent logic sharing false loops, is computationally expensive. For every logic sharing operation, logic synthesis needs to scan through the CDFG to determine false loops. In our approach, we first perform topological ordering on the elements of function units, multiplexers, and registers in the given data path. Based on the topological order, we assign to each data path element a level number which is used next as a filter for screening out the cases where loops are guaranteed not to exist. Only the loops that pass the filter are further investigated by traversing the circuit for false loop detection. When a false loop is detected, it is removed immediately by a false loop removal algorithm. This detection/removal process iterates until no more false loops are found. Details of our approach are described in Sections 3.1 to 3.3.

Several notations are defined here and will be used in the remainder of the paper. *Data path element* is used to denote an object in the RTL data path such as a functional unit, a multiplexer, or a register. For an output signal s of the control path, its function is represented as $f_s(x_1, \cdots, x_{n_s})$ where x_i, $1 \le i \le n_s$, is an input signal of s. The set of x_i, $1 \le i \le n_s$, is called the *support* of s, and a tuple (x_i,s) is used to denote the control input/output pair of x_i and s.

3.1 Topological Ordering of Data Path Elements

For a given false-loop-free RTL design, we can derive an *architectural graph* to model its data path architecture and control unit to facilitate our false loop identification algorithm. A node in the architectural graph represents either a data path element or the control unit (CU). The state flip-flops in the CU are not represented in the graph for our loop detection purpose. There are two types of arcs in the graph: solid arcs and dashed arcs. A solid arc from nodes a to b corresponds to a physical connection from a's output to b's input in the RTL design. To simplify the graph, no solid arc is used to represent connections from the CU node to a register node and vice versa, because obviously a combinational loop cannot have a register on it. A dashed arc from nodes a to b represents a control dependency in the RTL data path where a's output controls the execution of b. Notice that such a control dependency is implied only by data path operators and does not correspond exactly to a physical connection in the circuit. Figure 3 shows an example of architectural graph for a false-loop-free RTL design. We can see that the two dashed arcs are used to denote the control dependen-

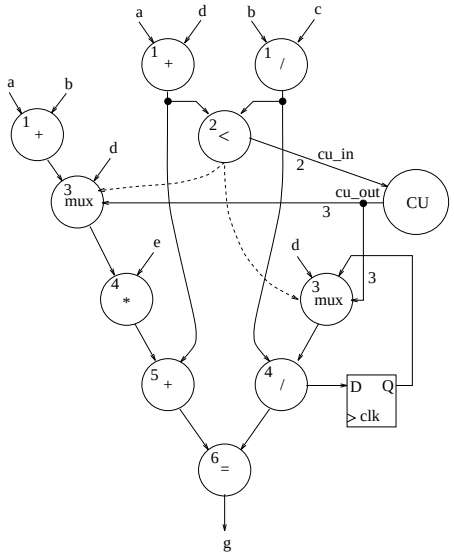

Figure 3. An example of architectural graph.

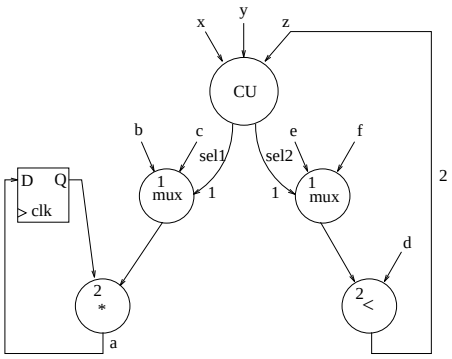

Figure 4. Architectural graph of Figure 2(a) and (b)

cies from the comparator $<$ to two multiplexers respectively. There are no physical connections in the RTL design corresponding exactly to the dashed arcs.

Topological ordering from inputs to outputs is first performed on the nodes corresponding only to the data path elements. Since registers break combinational paths, the nodes of registers are not considered for ordering, and their outputs are treated as primary inputs. A level number can be assigned accordingly based on the topological order. That is, excluding the register nodes as well as the CU node, the level number of each node equals to 1 plus the largest level number among immediate parent nodes. Figure 3 shows the level numbers assigned to the nodes according to the topological order.

The level numbers are then propagated to the input and output arcs of the CU node for false loop detection purpose. For an input arc from node a to the CU node, a's level number is assigned to the arc. As to the output arcs, since a control output signal can have multiple fanout nodes and each fanout connection has a corresponding output arc, we assign to all these arcs the same level number which is the smallest one of the fanout nodes. In Figure 3, the arc for input cu_in of the CU node is assigned to the same level number of $<$. For the output cu_out which has two fanout arcs, the smallest level number of the two fanout nodes, which are the same in this case, is assigned to the arcs.

3.2 False Loop Identification

A false loop can be identified by traversing the architectural graph using a depth-first search or breadth-first search algorithm. However, since there can be numerous false loops, direct application of such an algorithm to identify each of them can be computational expensive. To reduce complexity, a filter is devised which is based on checking the level numbers assigned to the input/output arcs of the CU nodes in the architectural graph.

For any control input/output pair (x_i, s) in a given RTL design with x_i in the support of s, if (x_i, s) obeys correct topological order, then the circuit is guaranteed to be false-loop-free. That is, if the level number of s (output of CU) is larger than the level of x_i (input of CU) for any (x_i, s), the circuit must be false-loop-free. This is because, for the data path elements O_s and O_{x_i} that determine the level numbers of s and x_i respectively, O_s must also have a larger level number than O_{x_i}, which implies there cannot exist a path in the data path from O_s to O_{x_i} according to the topological ordering policy. Therefore, no loop can be formed. The circuit in Figure 3 gives one such example where the control input/output pair $(\mathsf{cu\_in}, \mathsf{cu\_out})$ has correct topological order. Therefore, if the topological order of (x_i, s) is obeyed, there is no need to traverse the architectural graph from s for loop detection.

However, if the topological order is not obeyed, which means the level number of s is equal to or smaller than the level number of x_i, then whether or not there exists a false loop passing through s and x_i needs to be verified by traversing the architectural graph from s. This is because although O_s has its level number equal to or smaller than O_{x_i}, O_s does not necessarily have a path to O_{x_i}. This can be illustrated by the architecture graph depicted in Figure 4. The control input/output pair $(z, \mathsf{sel1})$ violates topological order but contains no false loop after verification by traversal. We can see that there is no path from the multiplexer controlled by sel1 to the comparator $<$ generating z. However, for the control input/output pair $(z, \mathsf{sel2})$ that violates topological order, the false loop will be detected by traversal. Therefore, traversal is required to verify if false loops actually ex-

```
for each output signal s = f_s(x_1, ..., x_{n_s}) of the control path {
    X = ∅;
    for each x_i, 1 ≤ i ≤ n_x {
        if ((x_i, s) violate topological order) {
            perform depth-first traversal from s in architecture graph;
            record in X all x_k's that are on detected false loops;
            if (X ≠ ∅) {
                perform false loop removal with X for s; /* Section 3.3 */
                break;
            }
        }
    }
}
```

Figure 5. False loop identification algorithm.

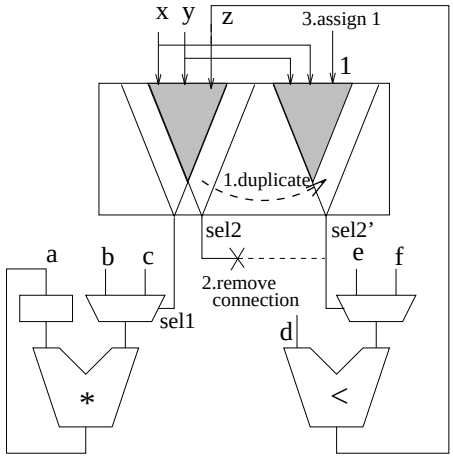

Figure 6. Illustration of false loop removal for the RTL design in Figure 2(b).

ist when topological order is violated.

Based on the discussion above, a false loop identification algorithm is described in Figure 5. In the algorithm, checking topological order is used as a filter to avoid traversing for each (x_i, s) if the order is correct. In case the order is incorrect, a depth-first traversal of the architecture graph starting from s is performed to identify all the false loops passing through s. All the control input/output pairs that are identified on these false loops are recorded in a variable X, which will be used in the false loop removal step to be discussed in Section 3.3.

3.3 False Loop Removal

Given the control output signal s and the variable X which stores all x_k's by the false loop identification algorithm discussed in Section 3.2, the false loop removal algorithm can be described by the following four steps:

The algorithm can be illustrated by applying to

step 1: Duplicate in s' the logic of $s = f_s(x_1, \cdots, x_{n_s})$ and its input/output connections.

Since the false loop problem was caused by control logic sharing, this step undoes the sharing by duplicating s.

step 2: Disconnect the output of s.

This is because s' will be responsible for the functionality of s. In this step, all the false loops passing s are broken.

step 3: Assign 0 or 1 to each control input x_k of s' if x_k is in X.

Since each x_k is considered don't care to s', the boolean difference of $s' = f_s(x_1, \cdots, x_{n_s})$ with respect to x_k should be 0. That is,

$$\frac{df_s}{dx_k} = f_s(x_k = 0) \oplus f_s(x_k = 1) = 0,$$

which means $f_s(x_k = 0) = f_s(x_k = 1)$. So, by Shannon expansion of f_s, we have

$$\begin{aligned} f_s &= x_k \cdot f_s(x_k = 1) + \overline{x_k} \cdot f_s(x_k = 0) \\ &= x_k \cdot f_s(x_k = 1) + \overline{x_k} \cdot f_s(x_k = 1) \\ &= f_s(x_k = 0) \\ &= f_s(x_k = 1). \end{aligned}$$

So we can assign either 0 or 1 to these don't care inputs without changing the actual functionality of f_s.

step 4: Perform control logic synthesis again to s and s' separately to remove redundancy without creating false loops again.

In this step, the information of the don't care inputs of s' assigned with 0 or 1 and the disconnected output of s is used by logic optimization to remove the redundant logic introduced by duplicating s in step 1. Notice that the logic synthesis is re-applied separately on both original and duplicate logic. This is to prevent the regeneration of false loops just removed.

the RTL design in Figure 2(b) to remove the false loop passing z and sel2. The removal steps are shown in Figure 6. The logic cone of sel2 and its input/output connections are first duplicated into sel2$'$. Then, the output of sel2 is disconnected, and 1 is assigned to z of sel2$'$. Logic optimization can then be performed to remove redundancy logic.

4 Experimental Result

We implemented the proposed approach in our high-level synthesis compiler MEBS [3], and run it on a Sun Sparc 20 work station. Three examples which contain false loops caused by control logic sharing were used for experiment. The first example, *Cone*, is an example circuit used to test

Table 1. Experimental results of real cases

Examples (Lines of RTL code)	Area of circuits which contain false loops			Area of circuits after eliminating false loops			Overhead
	Flip-Flop	Logic	Total	Flip-Flip	Logic	Total	
Cone(106)	43	61	**104**	43	62	**105**	0.96%
BJ(1354)	946	1362	**2308**	946	1363	**2309**	0.04%
Toner(1804)	2476	1950	**4426**	2476	1950	**4426**	0.00%

Table 2. Experimental results of cases with false loop inserted

Examples (Lines of RTL code)	Area of circuits which contain false loops			Area of circuits after eliminating false loops			Overhead
	Flip-Flop	Logic	Total	Flip-Flip	Logic	Total	
Answer(633)	616	948	**1564**	616	949	**1565**	0.06%
Ceps1(397)	3129	1400	**4529**	3129	1401	**4530**	0.02%
Vending(341)	277	224	**501**	277	225	**502**	0.19%
VCR(1235)	7229	2019	**9248**	7229	2020	**9249**	0.01%
Candy(656)	206	255	**526**	206	256	**527**	0.19%
Computer(438)	758	826	**1584**	758	827	**1585**	0.06%

the existence of false loops. The second example, *BJ*, is a Black Jack game machine. The third example, *Toner*, is a fuzzy logic controller for the copier toner. We compared the areas, in terms of basic cells for a specific technology library, before and after applying the false loop removal algorithm. The results are shown in Table 1, which reports for each example the areas of flip-flops, combinational logic of both data path and control path, and the entire circuit. The size of each example, in number of lines in RTL VHDL code, is listed inside the parenthesis after each example name. Our proposed algorithm detected 1 false loop in *Cone*, 2 in *BJ*, and 3 in *Toner*. The results show that only slight area overhead is imposed. In the case of *Toner*, we even see that the areas before and after applying the false loop removal algorithm are the same. This suggests that the overhead introduced by duplicating logic for false loop removal can be limited if logic minimization is applied subsequently.

We wanted to further observe the average overhead introduced by our algorithm. We selected six examples with nested control paths and manually inserted conditions to produce false loops during logic synthesis. Among these six examples *Answer* is an answer machine, *Ceps1* is a special RAM, *Vending* is a vending machine, *VCR* is a VCR controller, *Candy* is a candy machine, and *Computer* is a simple 4-bit computer. The experimental results shown in Table 2 still give low overhead. The average overhead of all examples in Tables 1 and 2 is 0.17%.

We also analyzed these examples and found only the portion in the control path where the fanout violates the topological order is identified by the false loop detection algorithm. Therefore, only a small number of gates in the control logic are duplicated. Furthermore, in Step 2 of the false loop removal algorithm, the fanout of the original logic being duplicated is disconnected, so the original logic may be eliminated by reapplying the logic synthesis. Therefore, the limited overhead caused by logic duplication can be expected. In Figure 7 for example, since one logic A fanin has larger level number than the fanout, only logic A is duplicated and its fanout is disconnected.

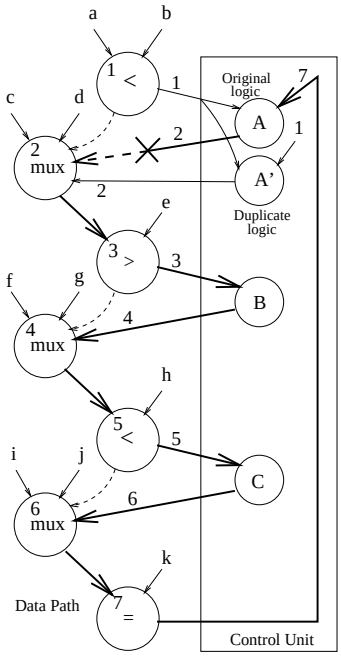

Figure 7. An example to show small overhead due to logic duplication.

5 Conclusion

In this paper, we analyzed the false loops caused by control logic sharing due to the usage of don't cares. We proposed an effective false loop identification/removal algorithm to eliminate such false loops. To reduce the computation complexity of false loop detection, a filter is devised to efficiently screen out the cases where false loops are guaranteed not to exist. Although the proposed algorithm duplicates logic for false loop removal, the experimental results show the average area overhead is 0.17%.

References

[1] L. Stok. False loops through resource sharing. In *Proceedings of International Conference of Computer-Aided Design*, pages 345–348, Nov. 1992.

[2] S. Huang, T. Liu, Y. Hsu, and Y. Oyang. Synthesis of false loop free circuits. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 55–60, 1995.

[3] Y. C. Hsu, T. Y. Liu, F. S. Tsai, S. Z. Lin, and C. Yu. Digital design from concept to prototype in hours. In *Asian-Pacific Conference on Circuits and Systems*, Dec. 1994.