BDD Decomposition for the Synthesis of High Performance PTL Circuits

Rupesh S. Shelar, Sachin S. Sapatnekar
Department of Electrical and Computer Engineering,
University of Minnesota, Minneapolis, MN 55455.
Email: {rupesh, sachin}@ece.umn.edu

Abstract

In this paper, we address the problem of performance oriented synthesis of pass transistor logic (PTL) circuits using a binary decision diagram (BDD) decomposition technique. We transform the BDD decomposition problem into a recursive bipartitioning problem and solve the latter using a max-flow min-cut technique. We use area, delay cost of the PTL implementation of the logic function to guide the bipartitioning scheme. Using recursive bipartitioning and a one-hot multiplexer circuit, we show that our PTL implementation has $O(\log N)$ depth in terms of the maximum number of series transistors for any logic circuit, where N is the number of inputs. We present the results obtained by running our algorithm on a set of MCNC benchmarks and compare these results with those reported by other library-less synthesis techniques.

1 Introduction

Static CMOS has long been a favorite logic style of designers for area-efficient and high-speed implementations with good noise immunity properties. However, with the advent of the system-on-chip era and therefore, with the motivation of packing more logic functionality on the chip, other logic styles such as domino and pass transistor logic (PTL) are being explored as alternatives to static CMOS, mainly because of their lower transistor count and their potential for high-speed operation. The primary benefits of PTL include the potential for a lower transistor count, lower capacitance, smaller delays and reduced power consumption [1, 2]. Synthesis techniques for PTL circuits have been closely related to the binary decision diagram (BDD) representation of the logic functions implemented by the circuit, for several reasons. Firstly, it is well known that a simple mapping can be used to translate a BDD to a PTL circuit and this mapping also avoids sneak paths from power to ground. Secondly, minimization techniques for BDD's have been extensively studied, and numerous efficient algorithms are available. The approach of [3] proposes a greedy heuristic to decompose the BDD's into smaller BDD's whose sizes are kept under a specified threshold. Based on this greedy framework, strategies for heuristic optimization under various objective functions, such as area, delay and power, have been proposed. In [2], a pass transistor based cell library is proposed and the results of synthesis are favorably compared with static CMOS. For area-driven PTL synthesis, Chaudhary *et al.* propose an approach [4] similar to traditional multi-level logic optimization that comprises iterative application of node elimination and *don't care* minimization to reduce the size of BDD's and BDD representation is, then, mapped on to a PTL cell library. A similar philosophy has been used for performance driven synthesis, where sweep, elimination and reorder procedures have been proposed [5], taking into account the performance gain while using these procedures iteratively. Both [3] and [5] imply that multi-level BDD's are to be used, but the limitation of these approaches is that they are unable to predict the performance gain beforehand. Other relevant work on BDD optimization includes [6], in which transformations such as AND/OR decomposition based on 0/1 dominators, and XOR and functional MUX-based decompositions are proposed. Becker *et al.* reported the use of multiplexer circuits for area and delay optimizations of PTL circuits [7]. Unlike [3], they allowed varied threshold size of BDD and their cost function allows area and depth to be traded off.

In this paper, we present a novel approach to performing PTL synthesis through a decomposition of a large monolithic BDD representing a circuit. We employ a bipartitioning scheme that uses max-flow min-cut technique to halve the depth of a PTL implementation of a BDD with the least area overhead. We first illustrate how the BDD can be bipartitioned into smaller pieces and implemented as a PTL circuit using multiplexers. We apply bipartitioning recursively and

with the use of one-hot multiplexer circuit, we show that it results in the implementation with logarithmic depth in number of inputs. Another feature of our work is that unlike many previous techniques for PTL circuit synthesis, we predict the delays in the circuit beforehand through a theoretical analysis of the circuit delay that motivates our partitioning algorithm.

The organization of rest of the paper is as follows. In Section 2, we explain a general technique of BDD decomposition and its PTL implementation using one-hot multiplexers and regular multiplexers. In Section 3, we describe a one-hot multiplexer and its advantages over regular multiplexer. In Section 4, we transform the decomposition of BDD to bipartitioning problem and solve the latter using the max-flow min-cut technique. In Section 5, we present the experimental results. In Section 6, we conclude the paper.

2 PTL Implementation using Decomposed BDD's

2.1 The BDD-PTL Relationship

A BDD can be mapped to a PTL implementation. Each node of the BDD implements a Shannon expansion about the variable x associated with the node, and can be expressed as, F = $x \cdot F_x + x' \cdot F_{x'}$, where F_x and $F_{x'}$ are, respectively, the Shannon cofactors of the function F. This may be translated to a multiplexer that passes F_x when x is high, and $F_{x'}$ when x is low; the procedure can then be applied recursively to the functions F_x and $F_{x'}$. Therefore, for any logic function, the BDD representation can be used to directly arrive at its PTL implementation, as shown in Figure 1. For the purposes of this paper, all BDD's

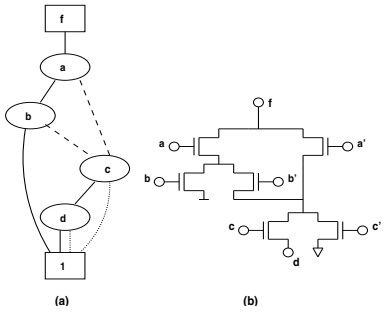

Figure 1: A translation from a BDD to a PTL circuit for the function F = AB + CD

are reduced ordered BDD's (ROBDD's), which implies that the order of variables on any path from an output node to a leaf node is identical.

Consider a PTL implementation of a function obtained by direct mapping of a BDD on N inputs. We estimate the delay along the critical path, $Delay_M$, as the delay along a path containing $N-1$ pass transistors in series. Let us assume that buffers are also added when k pass transistors are in series to regenerate the signal; we will assume $k = 3$ here, but a similar analysis can be carried out for any other value of k. To estimate $Delay_M$, we build an equivalent RC model for the pass transistor circuit. In Figure 2, we show the Elmore delay for one segment of a PTL circuit that consists of the maximum of 3 series transistors between buffers. Note that since each node is a multiplexer, only one of of the two transistors connected to any node will be on, implying that the equivalent RC network will be an RC line rather than a general RC tree. Assuming that the resistance of a pass transistor is R_p, and the resistance of the driver node (buffer) is R_d, we can calculate the Elmore delay of this structure as $R_d \cdot (3C_D + 3C_S + C_{Buff}) + R_p \cdot (6C_D + 3C_S + 3C_{Buff})$, where $C_{D(S)}$ is the drain (source) capacitance and C_{Buff} is the input capacitance of buffer. In case of a chain of $N-1$ pass transistors with one buffer placed at every third pass transistor, there are $\lfloor (N-1)/3 \rfloor$ three-transistor sections of the type shown in Figure 2. Therefore, the worst-case delay, corresponding to a path with $N-1$ transistors in series, is given by,

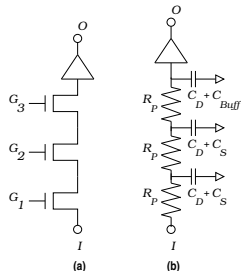

Figure 2: (a) Three pass transistors in series (b) The equivalent RC model

$$\begin{aligned} Delay_M &= \lfloor (N-1)/3 \rfloor (R_p(6C_D + 3C_S + 3C_{Buff}) \\ &\quad + R_d(3C_D + 3C_S + C_{Buff})) \\ &\approx D_{RC} \cdot (N-1) \end{aligned} \quad (1)$$

where, $D_{RC} = (R_p(2C_D + C_S + C_{Buff}) + R_D(C_D + C_S + C_{Buff}/3))$.

2.2 Decomposition of BDD

We see that mapping directly from a BDD to PTL results in delays that are linear in number of input variables, and decomposition can be used to reduce these delays. We outline a general BDD decomposition technique with the help of the following example. Consider a carry function for a 3-bit adder whose optimized BDD is shown in Figure 3(a). The BDD shown in Figure 3(a) is on 6 variables a0, b0, a1, b1, a2 and b2; while c3 is the output. We take a cut across the BDD as shown by the horizontal line in

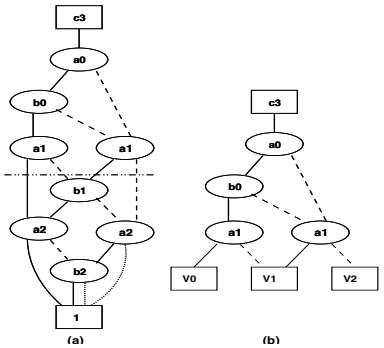

Figure 3: (a) Carry function for 3-bit adder, (b) Introducing dummy nodes V0, V1, V2

Figure 3(a). Because of this, dangling edges are created, for instance, edges from nodes labeled a1 to nodes labeled a2. We introduce dummy nodes V0, V1, V2 which replace nodes labeled a2 and b1, as shown in Figure 3(b). Dummy nodes V0, V1, V2 can be assigned unique codes using one-hot or minimum-bit encoding as shown in Table 1. After encoding, the next step in decomposition is to construct the BDD's corresponding to the *select* and *data* inputs of a multiplexer. Each such *select* input corresponds to a BDD representation that sets the leaf nodes according to the chosen encoding. As an example, the select bit O_0

One-hot Encoding		Minimum-bit Encoding	
terminal node	$O_0O_1O_2$	terminal node	S_0S_1
V0	100	V0	00
V1	010	V1	01
V2	001	V2	11

Table 1: One-hot and minimum-bit encoding for dummy terminal nodes

corresponds to the combination V0 = 1, V1 = V2 = 0. By substituting these values into the dummy terminals in Figure 3(b), we can obtain the BDD for the *select* input O_0 as shown in Figure 4(a). The BDD's for other *select* inputs such as O_1 and O_2 can be obtained similarly. Figure 4(b) shows the BDD's for *select* inputs S_0 and S_1, respectively. We observe that depth of the BDD's for the *select* inputs is the same for one-hot encoding as well as minimum-bit encoding. Note that in case of *select* functions obtained by one-hot encoding, for any assignment of a0, b0, a1, only one of the *select* functions is true and we can use a one-hot multiplexer circuit to implement c3. The implementation of c3 using a one-hot multiplexer and a regular multiplexer is shown in Figure 5. The *select* inputs are simply the PTL implementations of the BDD's shown in Figure 4. Clearly, the implementation using regular multiplexer has more delay than the implementation obtained using one-hot multiplexer. We also observe that in the decomposed implementation using the one-hot multiplexer, the depth of the circuit is halved as compared to the implementation obtained by a direct mapping of the BDD. In the following section, we will compare the one-hot multiplexer circuit with regular multiplexer circuit.

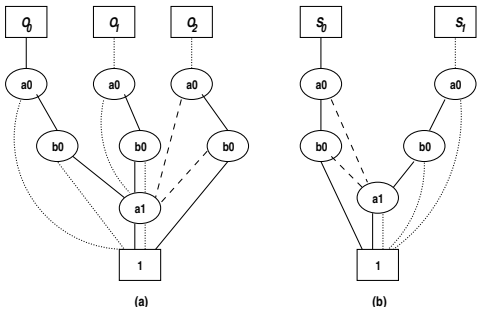

Figure 4: *Select* functions for (a) One-hot encoding, (b) Minimum-bit encoding

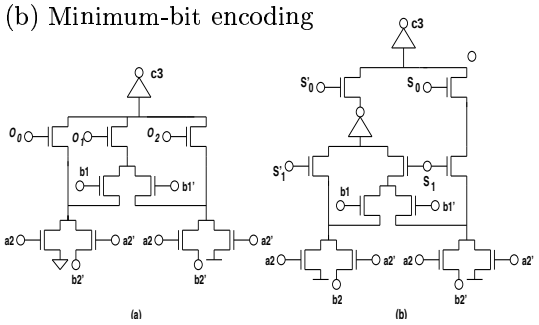

Figure 5: Implementation of c3 (a) Using one-hot multiplexer (b) Using regular multiplexer

3 One-hot Multiplexer and Regular Multiplexer

Figures 6(a),(b) shows transistor level implementations for 4:1 one-hot and regular multiplexers, respectively. In case of a one-hot multiplexer, four *select* inputs are required and only one of them can be high at a time and hence the name. In contrast, the regular multiplexer has two *select* inputs which are used to select among four *data* inputs. We observe that the depth of a one-hot multiplexer circuit, as measured by the maximum number of series transistors, is always one (constant), irrespective of number of *data* inputs. On the other hand, the depth of regular

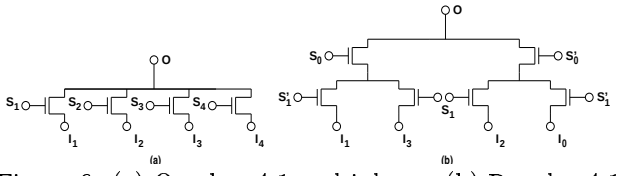

Figure 6: (a) One-hot 4:1 multiplexer, (b) Regular 4:1 multiplexer

multiplexer varies logarithmically with number of *data*

inputs. Apart from the performance advantage, the number of transistors required for implementation of one-hot multiplexer is linear in the number of *data* inputs while that required for implementation of regular multiplexer is super-linear in the number of *data* inputs. Thus, a one-hot multiplexer offers advantages in terms of area as well as performance over the regular multiplexer. However, the number of *select* inputs required for one-hot multiplexer is the same as the number of *data* inputs, and therefore, we need to generate a large number of *select* functions (which are one-hot) than for regular multiplexers. HSPICE simulations of transistor level implementations of one-hot and regular multiplexer also showed that one-hot multiplexer clearly outperforms regular multiplexer, as expected. However, we observed that delay of a one-hot multiplexer is not constant but increases with the number of *data* inputs. This is because, as the number of parallel transistors increase, the load driven by one-hot multiplexer increases, typically by the drain capacitance of a transistor for each additional data input.

4 Bipartitioning

In section 2, we presented techniques for decomposing a BDD and its PTL implementation using one-hot multiplexer and regular multiplexer. This decomposition can be thought of as a bipartitioning that halves the circuit depth and therefore, shortens the critical path and its delay. As seen in section 2, in case of mapping a monolithic BDD to PTL circuits, the delay is linear in N, the number of input variables. In the worst case, such a BDD has N nodes in series, translating to N series-connected PTL transistors (separated by buffers). If we take a single cut such that the critical path is halved, then using equation (1), we find that the delay using a one-hot multiplexer, which adds one extra series transistor, is given by,

$$Delay_{DC} \approx D_{RC} \cdot (N/2) \quad (2)$$

Therefore, we see that the delay is approximately halved. We can apply this bipartitioning procedure recursively, such that on each application of the procedure, the critical path is halved. The price being paid for delay reduction is in terms of area, since the number of transistors increases as we recursively bipartition the BDD. We can perform this bipartitioning in such a way that the area penalty is minimum. In the following subsection, we describe an algorithm that we have developed which uses the max-flow min-cut technique to find an optimum cut.

4.1 The Algorithm for Bipartitioning

Our aim is to find the optimum cut that halves the critical path, measured in terms of the number of BDD nodes along the path, and also has the least area penalty. We represent a BDD as a directed acyclic graph (DAG) where the nodes are identical to the nodes of the BDD, and the edges are identical to the BDD edges, and are assigned a direction corresponding to variable ordering, from a lower indexed variable to a higher indexed variable. To find a critical path, we assign the two distances to each node.

Distance from Top(D_{top}) : This is the maximum of the distances from all the primary output nodes to the given node.

Distance from Bottom(D_{bottom}) : This is the maximum of the distances of all the nodes which are reachable form a given node.

Note that the distances are measured in terms of number of nodes in the path and we have used the directed acyclic nature of the graph to define the distances. These distances can be found out using a PERT-like traversal on the digraph. Clearly, the critical path is a path on which the node with maximum distance from bottom lies. We define two types of nodes for delay balanced bipartitioning:

Essential Nodes : These are the nodes for which D_{bottom} equals half of the critical path length.

Candidate Nodes : These are the nodes for which D_{top} and D_{bottom} are both less than half of the critical path length.

The optimum cut will halve the critical path and also ensure that no other path in the decomposed implementation is longer than half the length of critical path. It is obvious that all essential nodes must be in the cut while we have a freedom to choose among the candidate nodes. We assign an area cost to each candidate node assuming PTL implementation obtained by direct mapping of BDD's and then use the max-flow min-cut technique [8] to find the optimum cut that halves the circuit depth at the minimum area cost.

Figure 7 shows the creation of the flow network

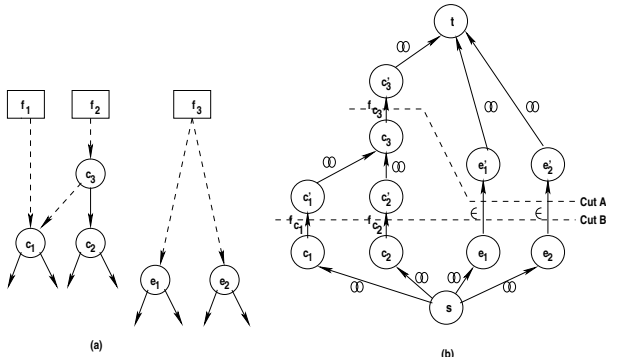

Figure 7: (a) Directed acyclic graph, (b) Corresponding flow network

from a digraph corresponding to the given BDD. Figure 7(a) shows the digraph corresponding to a BDD,

```
Input:  G(V,E) = Digraph corresponding to given BDD,
V = Nodes, E = Edges.
Output:  S_cut = Optimum cut-set.
Steps:
PERT_Traversal(G); /* Assign D_top, D_bottom ∀vεV */
L_crit = max{D_bottom ∀vεV};/* Length of critical path */
Nodes_Essential = {v:vεV and D_bottom = L_crit/2 };
Nodes_Candidate = {v:vεV and D_top, D_bottom < L_crit/2 };
AreaCostEstimate(Nodes_Candidate); /* Assume PTL
implementation by direct mapping of BDD's & estimate */
G_Flow = CreateFlowNetwork(Nodes_Essential,Nodes_Candidate,G);
Ford-Fulkerson(G_Flow, G, S_cut);   /* Find optimum cut */
```

Figure 8: Pseudocode for algorithm to find optimum cut

in which there are three nodes labeled f_1, f_2, and f_3 corresponding to the three primary outputs. There are three candidate nodes labeled c_1, c_2 and c_3 and two essential nodes labeled e_1 and e_2. Dashed edges in Figure 7(a) (for instance, an edge from f_1 to c_1) indicate that there are directed paths between the corresponding nodes. Figure 7(b) shows the corresponding flow network. There is one source node labeled s and one destination node labeled t. Each essential node in the digraph is split into two nodes, for instance, node e_1 in the digraph is represented by two nodes e_1 and e_1' with an edge from e_1 to e_1' of a small capacity ϵ. Similarly, candidate nodes in the digraph are represented by splitting them into two nodes, for instance, node c_1 in the digraph is represented by two nodes c_1 and c_1', respectively, with an edge of capacity f_{c1} from c_1 to c_1'. Note that since we want essential nodes to be included in the optimum cut, we assign a small capacity to the edge between the split essential nodes, and since we want to choose the candidate nodes with the least area penalty, we assign a capacity proportional to the area cost of candidate nodes to the edges between split candidate nodes. The remaining edges in the flow network are assigned a capacity of ∞, and therefore, will not appear in the cut. Thus, there are two possible cuts, Cut A and Cut B, corresponding to cut-sets $A_{cutset} = \{e_1, e_2, c_3\}$ and $B_{cutset} = \{e_1, e_2, c_1, c_2\}$ in the digraph corresponding to the given BDD. Application of the Ford-Fulkerson technique to find the minimum cut will result in one of these cuts, depending on values of f_{c_1}, f_{c_2} and f_{c_3}. The pseudocode for the overall algorithm is as shown in Figure 8. Once the cut is determined, the vertices in the cut are replaced by dummy terminal nodes, which can be assigned unique codes, and implemented using either a one-hot multiplexer or a regular multiplexer, respectively, as illustrated in subsection 2.2. The bipartitioning procedure can be applied recursively till no further depth reduction can be achieved and the resulting implementation has a depth which is logarithmic in terms of number of inputs, as stated by the following theorem.

Theorem 4.1 *The recursive application of an algorithm in Figure 8 to any BDD on N input variables with the use of one-hot multiplexers results in an implementation which has a depth of $O(\log N)$, in terms of number of series transistors.*

Proof 4.1 We observe that the BDD on N variables has at most N nodes in series and application of bipartitioning shown in Figure 8 results in halving the critical path length. It means that BDD's corresponding to *select* inputs to the multiplexer as well as *data* inputs has the path with at most $N/2$ nodes in series. Use of one-hot multiplexer adds depth of one transistor. Application of bipartitioning to *select* and *data* inputs will reduce their depth to $N/4$ with an addition of one transistor depth for one-hot multiplexer. Thus, bipartitioning can be recursively applied and every time it halves the depth. There can be $\log N$ such recursions and the resulting implementation has depth $O(\log N)$.

We point out that multiplexer based implementation for PTL circuits proposed by Becker *et al.* [7] obtains a logarithmic depth for *xor* functions. This is because in case of the *xor* function the cutset always contains two nodes (*xor* function on intermediate variables and its complement) and [7] used regular two-input multiplexers, which have a depth of one. On the other hand, the use of one-hot multiplexers and recursive bipartitioning results in a logarithmic depth implementation for any circuit, irrespective of the size of the cut-set.

5 Experimental Results

We have implemented the recursive bipartitioning algorithm and the decomposition procedure as a c++ program called PTLS (Pass Transistor Logic Synthesizer). PTLS uses the BDD package CUDD [9] for generating BDD's. We used NMOS transistors as pass transistors and the size of each transistor was $0.5\mu/0.25\mu$. We used inverters after every three transistors in series, and in case of implementations using PTLS, to drive the gates of one-hot multiplexer as well as to boost the output of the one-hot multiplexer. For the inverter, we chose $w_p/l_p = 1\mu/0.25\mu$ and $w_n/l_n = 0.5\mu/0.25\mu$. The delays were measured using static timing analysis of the resulting transistor netlist. In general, the delay in the transistor depends on the transition time of the input waveform, the load it drives and the transistor size. Since we keep the transistor sizes fixed ($0.5\mu/0.25\mu$) and input transition time same (1ns) throughout the experiment, we characterized pass transistors and inverters only for

variable loads and used the same model to evaluate delays using static timing analysis. We used PTLS to synthesize MCNC benchmark circuits and compared its results with other libraryless synthesis techniques such as TABA [10] and OTR [11]. Table 2 shows the number of transistors required for implementations using TABA, OTR and PTLS for several MCNC benchmarks; delay figures for TABA and OTR are unavailable. We observe that the implementations obtained by using PTLS has significantly less area than TABA and OTR in all cases except sao2. This is because BDD representations for all those examples are compact and more importantly, recursive bipartitioning using one-hot encoding results in simplified expressions. Table 3 shows a comparison of the number of transistors and the delay for the implementations obtained by mapping monolithic BDD's directly to PTL, with the number of transistors and delays in the implementations obtained using PTLS. We observe significant delay reductions in all cases with marginal increase in area.

Example	TABA [10]	OTR [11]	PTLS
	# of Trans.	# of Trans.	# of Trans.
5xp1	302	378	282
9sym	404	272	109
misex1	148	158	130
rd53	82	82	56
rd73	174	152	120
rd84	290	252	175
sao2	362	320	487
Total	1762	1614	1359

Table 2: Comparison with TABA, OTR

Example	Monolithic		PTLS	
	# of Trans.	Delay(ps)	# of Trans.	Delay(ps)
5xp1	284	1074.87	282	289.5
9sym	110	1723.89	109	638.66
c17	32	692.537	29	420.52
alu2	1002	2136.53	1188	568.90
cm138	70	758.909	98	590.26
cm163	126	845.847	159	312.86
cmb	158	1523.57	132	819.76
comp	1070	5295.79	1779	653.77
parity	112	3381.1	108	820.61
rd53	82	753.327	56	491.89
rd73	146	1364.68	120	590.26
rd84	202	1425.47	175	876.12
t481	160	2828.1	299	328.90
z4ml	130	840.131	94	593.53
sao2	326	1747.46	487	471.63
misex1	142	738.797	130	570.35
f51m	252	1391.0	573	439.624
my_adder	1204	5417.5	1313	1259.73

Table 3: Comparison of monolithic BDD implementation with PTLS

6 Conclusion

In this paper, we have presented a recursive bipartitioning approach to BDD decomposition. We observe that the use of recursive bipartitioning and one-hot multiplexers results in a logarithmic depth implementation of any circuit. A comparison of implementations obtained by our technique with those reported by other libraryless synthesis techniques show that our procedure clearly outperforms TABA [10] and OTR [11] in terms of area. The comparison with monolithic BDD implementation shows significant performance improvement with small area overhead.

References

[1] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigsahi, and A. Shimizu. A 3.8ns CMOS 16 x 16 multiplier using complementary pass transistor logic. *IEEE Journal of Solid State Circuits*, 25(2):388–395, 1990.

[2] K. Yano, Y. Sasaki, and K. Rikino. Top-down pass-transistor logic design. *IEEE Journal of Solid-State Circuits*, 31(6):792–803, 1996.

[3] P. Buch, A. Narayan, A. Richard Newton, and A. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *International Conference on Computer Aided Design*, pages 663–670, November 1997.

[4] R. Chaudhary, T. Liu, A. Aziz, and J. Burns. Area-oriented synthesis for pass-transistor logic. In *International Conference on Computer Design*, pages 160–167, October 1998.

[5] T. Liu, M. Ganai, A. Aziz, and J. Burns. Performance driven synthesis for pass-transistor logic. In *VLSI Design Conference*, pages 372–377, January 1999.

[6] C. Yang and M. Ciesielski. BDD decomposition for efficient logic synthesis. In *International Conference on Computer Design*, pages 626–631, October 1999.

[7] C. Scholl and B. Becker. On the generation of multplexer circuits for pass transistor logic. In *Design Automation and Test in Europe*, pages 372–378, March 2000.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* Prentice-Hall India, New Delhi, 1998.

[9] F. Somenzi. CUDD: CU Decision Diagram package, release 2.3.0. http://vlsi.colorado.edu/ fabio/CUDD/.

[10] A. Reis, R. Reis, D. Auvergne, and M. Robert. The library free technology mapping problem. In *IEEE/ACM International Workshop on Logic Synthesis*, May 1997.

[11] Y. Jiang, S. S. Sapatnekar, and C. Bamji. A fast global gate collapsing technique for high performance designs using static cmos and pass transistor logic. In *International Conference on Computer Design*, pages 276–281, October 1998.