Re-Encoding Sequential Circuits to Reduce Power Dissipation*

Gary D. Hachtel Mariano Hermida † Abelardo Pardo Massimo Poncino ‡ Fabio Somenzi

University of Colorado
Dept. of Electrical and Computer Engineering
Boulder, CO 80309

Abstract

We present a fully implicit encoding algorithm for minimization of average power dissipation in sequential circuits, based on the reduction of the average number of bit changes per state transition.
We have studied two novel schemes for this purpose, one based on recursive weighted non-bipartite matching, and one on recursive mincut bi-partitioning. We employ ADDs (Algebraic Decision Diagrams) to computate the transition probabilities, to measure potential area saving, and in the encoding algorithms themselves.
Our experiments show the effectiveness of our method in reducing power dissipation for large sequential designs.

1 Introduction

The importance of low-power, high-throughput microelectronic systems is rapidly increasing. High power dissipation limits the developments of portable applications that demand intensive high-speed computation. Further, excessive power dissipation is a limiting factor in integrating more transistors on a single chip.
Power directed synthesis techniques can significantly reduce power dissipation [10, 6]. They can be viewed as straightforward modifications of conventional logic synthesis approaches, and are applicable to a very broad class of digital designs. All of these works exploit the estimated transition frequency (switching activity) in directing the synthesis [9], because in CMOS circuits there is little standby power consumption.
Regarding *sequential* synthesis for low power, previous work on encoding [10] follows the paradigm of conventional FSM encoding algorithms, such as [2, 4], where the states are referred to *explicitly*, and therefore cannot be applied to large practical state machines.
We employ the recently developed ADD (Algebraic Decision Diagram, [1]) technology to overcome these limitations. ADDs are a form of multi-terminal BDDs that support algebraic and arithmetic operations on their terminal nodes, which can hold objects drawn from an arbitrary set, *e.g.*, real numbers. ADDs are the key to the Markov analysis which gives the edge weights which drive the re-encoding algorithm. ADDs have allowed us to solve the Chapman-Kolmogorov equations [10, 6] for realistic machines (million of states) that are not manageable by conventional sparse matrix techniques.
In this paper, we focus on power optimization of sequential logic by *re-encoding* an existing circuit. We try to find an encoding of the states such that the average number of bit changes per state transition is minimized. By doing this, besides minimizing the toggling of the latches, we possibly reduce the switching activity in the combinational logic that implements the next-state and the output functions.

*This work is supported in part by NSF/DARPA grant MIP-9115432 and SRC contract 92-DJ-206.

†Mariano Hermida is with the Universidad Politecnica De Madrid, Facultad Informatica, Madrid, Spain

‡Massimo Poncino is also with the Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, ITALY 10129.

Our method initially computes the steady-state probabilities, and the state *transition* probabilities, as discussed in [7]. We then re-encode the circuit so as to minimize latch activity. We have studied two novel encoding strategies, one based on recursive weighted non-bipartite matching, and one on recursive mincut bi-partitioning, that combine the well-known Dolotta-McCluskey method with the multi-level area optimization target approach of the MUSTANG algorithm [2] by building a weight matrix which is a convex combination of transition probabilities and area optimization potential. The next-state and output functions are then re-encoded by solving a boolean equation.
While our experimental results are preliminary, they demonstrate that our ADD technology is practically significant.

2 Matching Based Re-Encoding

Our first method is based on recursive maximum weighted matching, which we solve with an ADD extension of a BDD-based mincut/maxflow algorithm [8]; the levels of recursion correspond to encoding bits as in [3]. At each step of the recursion the states with the greatest transition probability are matched, and the half with the greatest match weight are grouped in the same partition, and receive the same code bit.
Figure 1 shows the re-encoding algorithm. Variable sets x and y encode rows and columns, respectively.

```
Encode (G(x, y), N) {
    i = 0;  G^i(x, y) = G(x, y);
    for (i = 0; i < N; i++) {
        M^i(x, y) = Max_Weight_Matching (G^i(x, y));
        Code[i](x) = ∃_y^+ (M(x, y) · (x > y));
        Pairs(x, y) =  Pairs_Of_Matched_Nodes(M^i(x, y));
        G^{i+1}(x, y) =  Total_Weight(G^i(x, y), Pairs(x, y));
    }
}
```

Figure 1: High-Level Encoding Procedure.

The procedure **Encode** gets the matrix G and the number of latches of the circuit N as inputs.
Transition probabilities in G are non-zero only for states in the terminal SCCs of the transition graph. The reduction in the switching activity is obtained from the analysis of this (sometimes very small) subset of the actually reachable states. This is correct from the standpoint of *average* power dissipation, since transient states play a negligible role in the *average* power dissipation. Transient states are indeed relevant for area minimization, and they are taken into account for potential area savings.
At each iteration, we first find a maximum matching M^i in the matrix G^i (`Max_Weight_Matching`). Then, a "1" is arbitrarily assigned to the i-th bit of the new encoding of one member of each pair of matched nodes, and a "0" to the other. The node with higher index ($x > y$) is chosen as representative of each matched pair. The re-encoding is given as a *trans-coding* function, which expresses the bit of the new encoding as a function of the bits of the old one. After assigning to every state one of the bits of the encoding, we shrink the graph. That is, we find a new

weighted graph G^{i+1}, in which the new nodes are the pairs of matched states. After the k-th matching stage of this recursion, a supernode corresponds to 2^k original states. Since every new node (state) gets either a "1" or a "0" in the next least significant bit of the new encoding, all the original nodes which formed the current (super)node get the same value for the k-th bit. Therefore, they differ in the bit corresponding to the iteration in which they were matched, and possibly in bits of preceding steps.
At each stage we must match all the states; hence the matching contains some degree of arbitrariness, since the matching of unreachable states is irrelevant. If no area saving weight is included, the same holds for states which are outside the terminal SCC of the transition graph. This degree of arbitrariness in the re-encoding can be used later in the resynthesis phase, for example, to simplify the logical expression or to try to minimize delay.

2.1 Weighted Non-Bipartite Matching

We now focus on finding a maximum weighted matching on a graph with an implicit algorithm. Our purpose is to solve this problem for very large graphs, whose size is beyond the possibility of traditional explicit algorithms.
The theoretical best solution is to use some implicit variant of Edmonds's matching algorithm [5]. For general non-bipartite graphs, blossoms (odd cycles) significantly increase the complexity of the matching algorithm: It is difficult to deal with blossoms symbolically. On the other hand, we can trade off accuracy for time and memory by targeting an approximate solution to the problem.
In the following we deviate from Edmonds's algorithm in two ways. First, we get rid of odd cycles as soon as they appear (suppressing the edges which cause them). Second, after reducing the graph in this manner, we find an approximate maximum weighted matching.
We have developed a heuristics for quasi-maximum weighted matching that is amenable to symbolic implementation. It is based on (1) sorting the edges according to their weight, and (2) selecting a subset of the edges in decreasing order of weight. These edges are then used to find a first maximum *unweighted* matching. If this matching is not complete, we then add more edges to the selected set of edges and try to match nodes which are not yet match. The process is continued until a complete matching is obtained.
The core of the matching is based on growing *alternating trees* from every unmatched node, in order to find an alternating path to another unmatched node. We build alternating paths rooted at exposed nodes till they meet another exposed node. However, to make our approach symbolic, we grow trees from *every* exposed node, and terminate the growth when they meet another tree. In this sense, our symbolization of Edmonds's algorithm is similar to the symbolization of Dinits's algorithm in [8].
Figure 2 shows the augmenting paths growing algorithm. The parameter E is the edge relation of the graph, and M is an initial matching; i represent the iteration index.

3 Mincut Based Re-Encoding

The alternative to maximum weighted matching is recursive mincut bi-partiti-oning. We extend the original work by Lin and Kernighan to the symbolic BDD/ADD based setting. Although our symbolic version gives up the linear complexity update of the Kernighan-Lin scheme, symbolic processing vastly extends capacity in cases amenable to BDD representations.
The idea is illustrated in Figure 3. Given an initial partition of the states into L and R, migration groups MR (Move Right) and ML (Move Left) are computed as follows. MR is the set of states s for which the sum of the weights of edges going from $s \in L$ to R, minus the sum of the weights of edges going from $s \in L$ to L is less that ϵ (a user-defined threshold). ML is similarly defined. This

```
Tree_Matching (E, M, i) {
    i = 0; Matching^i = M;
    while (Matching^i = Matching^{i+1}){
        Roots = complement(Matching^i);
        if (odd_cycle = 0) {
          Remove_Odd_Cycles (Roots);
          continue;
        }
        if (augmenting_path = 0) {
          Matching^i = Modify_Matching(Roots);
          continue;
        }
        New_Interior_Points^i = Interior_Points(Roots);
        New_Exterior_Points^i = Exterior_Points(Roots);
        if (New_Interior_Points^i = 0) return (Matching^i);
        if (odd_cycle = 0) {
          Remove_Odd_Cycles (New_Exterior_Points);
          continue;
        }
        if (augmenting_path = 0) {
          Matching^i = Modify_Matching(New_Exterior_Points);
          continue;
        }
        return (Matching^i);
    }
}
```

Figure 2: High-Level Tree Matching Procedure.

process is iterated until no such states exist. The process may then be repeated with a smaller ϵ, if appropriate.

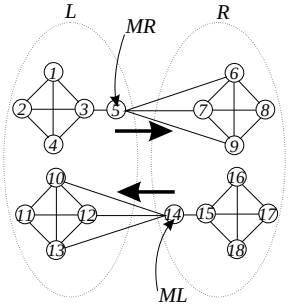

Figure 3: Basic Step in the Group Migrate Procedure.

Once the partition on the global set of states is obtained, we assign one binary value to each block of the partition, corresponding to the most significant bit of the encoding, and we recur on the two halves. At the end, every node will have a distinct code. States having high mutual transition probabilities will get low-distance encodings. The algorithm is shown in the following figures. The high level procedure is shown in Figure 4. In the following, r and c variables encode *row* and *column* indices.
The procedure `Recursive_Mincut` gets the current weight (sub)matrix $A(r, c)$, the partition (L, R), the coding functions $Codes$, and the current index of the state bit n as inputs. Initially, A is the transition probability matrix, L and R are the tautology, $Codes$ is the zero function, and n the number of state bits. $Codes$ is a *trans-coding* function, which will give each bit of the new encoding as a function of the old coding bits.
Line 1 shows the termination condition. When the current subspace consists of two states only, we do not attempt futher moves, and immediately update the LSB of the new encoding. In Line 2, an initial partition $(\widetilde{L}, \widetilde{R})$ is computed by `split_set`. We cannot simply split on the top variable since $\widetilde{L}$ and $\widetilde{R}$ must have the same size. This is because we will eventually assign a 1 in the i-th code bit to all states of one of the two partitions, and there must be the same number of states with a given code bit set to 1 and to 0.
The effectiveness of a partitioning algorithm depends on

```
Mincut (A(r, c), L(r), R(r), Codes(r), n) {
    if (|A| = 2) {
1       Codes[0](r) = Codes[0](r) + L(r);
        return;
    }
2   L̃(r) = Split_Set(L(r)); R̃(r) = L(r) · ¬L̃(r);
3   Cut = ∃_xy(L̃(r) · A(r, c) · R̃(c));
4   MR(r) = Group_Migrate_Right (L̃(r), A(r, c), p(r), 0);
5   ML(r) = Group_Migrate_Left (R̃(r), A(r, c), p(r), 0);
6   CS(r) =Correction_Set(MR(r), ML(r), |MR(r)| − |ML(r)|);
    (L(r), R(r)) = Update_Sets (L̃(r), R̃(r), MR(r), ML(r), CS(r));
    NewCut = ∃_xy(L(r) · A(r, c) · R(c));
7   if (NewCut> Cut) {
8       L(r) = L̃(r); R(r) = R̃(r);
    }
9   Codes[n](r) = Codes[n](r) + L(r);
10  Mincut (L(r) · A(r, c) · L(c), L(r), L(r), Codes(r), n − 1);
11  Mincut (R(r) · A(r, c) · R(c), R(r), R(r), Codes(r), n − 1);
}
```

Figure 4: High-Level Procedure.

the initial partition. An alternative for the initial guess is the partition produced by a single run of the symbolic matching algorithm described in Section 2.

In Line 3 we compute the cut-size of the current partition. In Lines 4 and 5, the migration groups MR and ML are computed, as detailed in Figure 5. The blocks of the initial partition are then updated.

In general, the migration procedures compute sets of different size. Since the partition must be balanced, the procedure `correction_set` (Line 6), computes a set of states $CS(r)$ whose size equals the difference $||ML| - |MR||$. Then, we update the partition according to the migration groups and the correction set to get the new L and R. At this point the new cut-size is computed and compared to the previous one. If the new one is larger, we restore the initial partition. This case may arise since we force to move equally sized sets of states.

In Line 9, we update the i-th code bit function with the L function, as obtained by the migration procedures. Notice that the coding functions are built by accumulating partial results at the same level of recursion. We then recur on L (Line 10). The set $L(r) \cdot A(r,c) \cdot L(c)$ represents the subgraph consisting of nodes and edges in L. In Lines 11 we repeat the same operation for states in R.

In the group migration algorithm, *mindepth* is a parameter which prevents the move of too large sets of states. In the early stages of the recursion, it is unlikely that moves of large sets of nodes are profitable. Since we want to reduce the "expensive" operations, we allow them only at depth greater than *mindepth*. Clearly, $mindepth \leq n$.

The procedures receives the sets L and R, the weight matrix A, a function p which represents the cofactor cube, and the maximum depth n of the recursion.

```
Group_Migrate_Right (L(r), A(r, c), R(c), p(r), n) {
1   if(L(r) · A(r, c) · R(c) = 0) return(0)
2   if(|p| ≥ mindepth) {
3       w_R(r) = ∃⁺_rc(L(r) · A(r, c) · R(c));
4       w_L(r) = ∃⁺_rc(L(r) · A(r, c) · ¬R(c));
5       if ((w_R − w_L) ≤ α · w_L) return group(r);
    }
    r_n = Top_Variable(L(r), A(r, c));
6   E = Group_Migrate_Right(L_¬r_n, A_¬r_n, R(c), (p(r) · ¬r_n), n + 1);
7   T = Group_Migrate_Right(L_r_n, A_r_n, R(c), (p(r) · r_n), n + 1);
8   Result = ite(r_n, T, E);
    return (Result);
}
```

Figure 5: MR Computation.

In Line 1, we check whether the cutsize between L and R is 0. Clearly, we do not move anything and return 0. If we have reached the required depth (*mindepth*) in the recursion (Line 2), we check if the current set is a candidate for being moved. The function $L(r) \cdot A(r,c) \cdot R(c)$ in Line 3 represents the subgraph consisting of the edges originating from nodes in L, and ending on nodes of R. By existentially abstracting the variables r and c from this function, we get the *sum* of all the edge weights (w_R) over all the nodes in L for all the end nodes in R. Since we are dealing with ADDs, abstraction implies *algebraic* sum.

Similarly, $L(r) \cdot A(r,c) \cdot \overline{R(c)}$ in Line 4 is the subgraph consisting of the edges from nodes in L, and into nodes *outside* R; w_L is the sum of the edge weights over the all the nodes in L for all the end nodes *not* in R. In Line 5, we return from the recursion if L satisfies *on average* the threshold requirement. With this strategy, we need a *relative* threshold rather than a static one. In Line 5, *alpha* is a number between 0 and 1. If the requirement is satisfied, we return the current L.

In Lines 6 and 7 we recur by splitting on the variable r_n corresponding to the current recursion index. We do not recur on R, which therefore remains unchanged. Finally, we combine the results of the recursive calls with `ITE`.

The algorithm for computing ML is similar to the previouse procedure: The sets L and R are interchanged, and we now recur on R, leaving L unchanged. However, we now take into account the already computed MR set: In computing w_L, we do not consider edges going from nodes of R to nodes in MR; similarly, in computing w_L, we consider the set MR as belonging to the current R set. Furthermore, to guarantee a move which is balanced with respect to the MR migration group, we check during the recursion if the current set is of the same size as MR. If so, we return without further recursion.

4 Area Minimization

If we use only the transition probabilities matrix as the underlying graph of the algorithm, we may get an encoding that, though optimal for minimizing the transition on the state lines, may cause an increase in the circuit area. This increase in area, besides being undesirable by itself, could eventually mask the saving in power, since the extra gates will dissipate some power. Therefore, we need a mechanism which allows to control the area build up. Since area is not the main concern of our algorithm, we can accept a relatively simple measure, which is easily amenable to a symbolic formulation, like the one used in MUSTANG [2]. In MUSTANG, the attractive force between states is related to the ability of extracting common cubes from either the next state or output functions, according to the two variants (*fanout-oriented* and *fanin-oriented*) of the algorithm. In the fanout-oriented algorithm, states that have a common fanout state are attracted; in the fanin-oriented, states that have a common fanin state are attracted. Both algorithms build a weight matrix, which expresses the attraction between pairs of states. We compute the weight matrices symbolically, by representing them as ADDs. In the following description, $T(s,x,t)$ represents the transition relation of the STG.

Fanout-oriented algorithm. First we build a matrix $S(s,t)$; entry s_{ij} gives the number of input patterns that label the transition between s_i and state s_j. ($10 - -1$ yields a value of 4, because it represents four patterns.) S is obtained as:

$$S(s,t) = \exists_x^+(T(s,x,t)).$$

Then we build a second matrix $Z(s,o)$, having one row for each state, and one column for each output. Entry z_{ij} gives the number of input patterns that, on the arc from state s_i, assert output z_j. To compute Z, we need the output functions $\lambda(x,s)$. The variables denoted with o encode primary outputs by randomly selecting a binary code for each output variable. The i-th column of Z is given by

$$Z_i(s) = (\exists_x^+(\lambda_i(s,x))).$$

Then, the weight matrix is obtained as:

$$W(s,t) = \frac{N_b}{2} \cdot (S(s,u) \times S^T(u,t)) + (Z(s,o) \times Z^T(o,t)).$$

N_b is the number of encoding bits. The factor $\frac{N_b}{2}$ says that the number of occurrences of the common cube depends on the number of 1's in the next state code. In the end, W is a square matrix with as many rows and columns as the number of states.

In a similar fashion we compute the weight matrix $\widetilde{W}(t,s)$ for the fanin-oriented algorithm. The two variants may be used either separately or together. In the second case, we get a weight matrix whose elements are the average of the weight values of W and $\widetilde{W}$. The overall weight matrix is an input to the encoding algorithm, together with the transition probability matrix.

5 Experimental Results

We have applied our encoding algorithm to examples taken from the ISCAS'89 and MCNC benchmarks. Table 1 shows the results obtained on a DECstation 5000/200 with 80 MB of memory. The columns labeled *Bit Changes* give the average number of bit changes in the encoding, for both the original circuit (column *Orig*) and the re-encoded one (column *Re-enc*). This quantity corresponds to the *cost* of the encoding $C = \sum_{i,j} w_{ij} \cdot d(e_i, e_j)$, where w_{ij} is the value of the weight matrix, and $d(e_i, e_j)$ is the Hamming distance between the codewords e_i and e_j. The values in Table 1 are obtained with the matching algorithm, and show that this cost function is significantly reduced in most examples.

Circuit	*Latches*	*States*	*Bit Changes*		*Time*
			Orig	*Re-enc*	
s27	3	8	0.90	0.89	0.2
keyb	5	32	0.83	0.65	1.8
tbk	5	32	1.53	1.21	2.0
styr	5	30	1.26	0.61	3.9
s820	5	32	0.78	0.74	1.6
s1488	6	64	0.82	0.35	9.0
s386	6	64	1.03	1.00	3.3
scf	7	128	2.32	1.31	18.6
s208	8	256	0.50	0.50	1.3
tlc	10	1024	1.98	1.72	4.6
s298	14	16384	1.96	1.80	134.6
s420	16	65536	0.50	0.50	165.6
s400	21	2e6	1.99	1.37	1204.3

Table 1: Encoding Results.

To get an estimate of the actual dissipated power, we have used the method by Ghosh *et al.* [6]. In the following experiments, the circuits are optimized using the `script.rugged` *sis* standard script when the size of the circuit makes it possible, and with `script.algebraic` otherwise. The circuit are mapped with the `map -AFG -n 1` command using the `lib2` and `lib2_latch` standard libraries. Since the delay of the circuit plays a role in determining the dissipated power, we mapped to reduce the circuit delay, rather than area.

To ensure a high similarity in the structure of the two circuits, each re-encoded circuit is translated to a multi-level representation obtained from their BDDs. We keep a similar structure in the compared circuits in order to isolate the effect of re-encoding.

We ran experiments comparing the re-encoded circuit with the original circuit. We should emphasize that our method targets multi-level circuits, and that we are interested in the problem of re-encoding an already existing circuit. Therefore, we do not compare with traditional encoding methods that start from a symbolic specification of a STG (e.g., a description in KISS format). In this sense, the comparison with multi-level sequential designs is more challenging, since they are synthesized under some encoding which is likely to be close to the optimum, at least for area.

Table 2 shows the comparison of the original and the re-encoded circuits. The values in Column "*Logic*" are the numbers of literals in factored form after optimization (Column *OPT*), and the area of the mapped circuit (Column *Area*). Power dissipation is computed assuming a unit-delay model, to allow estimation for larger circuits. Power roughly scales with area, but for cnt8, s1488, and scf, despite an increase in area, the resulting power is lower.

Circuit	*Original Circuit*			*Re-Encoded Circuit*		
	Logic		*Power*	*Logic*		*Power*
	OPT	*Area*	(μW)	*OPT*	*Area*	(μW)
s820	348	508080	860	296	448688	649
s386	101	194416	280	98	181888	273
s1488	833	1127056	1840	995	1316832	1263
scf	1533	1698240	4034	1722	1931168	2921
s208	45	85376	102	43	82592	78
cnt8	70	134560	412	108	208336	210

Table 2: Estimated Power Results.

The largest circuits of Table 1 are missing from Table 2 because the experimental procedure has two major bottlenecks. First, computing the next-state and output functions of the re-encoded circuit is rather expensive, in terms of memory. Moreover, the *blif* files generated from the BDDs are, in some cases, too large to be processed by `sis`. Second, the power estimator of [6] is very memory intensive. Therefore, although we have been able to generate encodings for some large circuits, we could not evaluate the dissipated power.

6 Conclusions and Future Work

We have described two symbolic methods for re-encoding a circuit to reduce the dissipated power; they have proven effective for designs that are too large for traditional encoding techniques. Although the experimental results are promising, the synthesis procedure still needs further refinements in both the circuit transformation and the power estimation phases.

The matching algorithm gives more accurate results, while the mincut approach takes less memory, and therefore can be applied to larger circuits.

We are currently investigating the possibility to account for unreachable states already in the encoding processes. This implies executing the algorithms on a subgraph that contains only the reachable states. The coding functions obtained in this case would be incompletely specified, and the problem of assigning a distinct code word to each state can be addressed in the transformation phase.

This technique may especially benefit the mincut approach, where we address nodes. In the matching approach, where we match *edges*, the impact may be more limited.

References

[1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In Proceedings of the International Conference on Computer-Aided Design, pages 188–191, Santa Clara, CA, November 1993.

[2] S. Devadas, H.-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines for optimal multi-level logic implementations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, CAD-7:1290–1300, December 1988.

[3] T. A. Dolotta and E. J. McCluskey. The coding of internal states of sequential machines. IEEE Transactions on Electronic Computers, EC-13:549–562, October 1964.

[4] X. Du, G. D. Hachtel, and P. H. Moceyunas. MUSE: A MUltilevel Symbolic Encoding algorithm for state assignment. In Proceedings of the Hawaii International Conference on Systems Science, pages 367–376, January 1990.

[5] J. Edmonds. Paths, trees and flowers. Canadian Journal of Mathematics, 17:449–467, 1965.

[6] A. Ghosh, S. Devadas, K. Keutzer, and J. White. Estimation of average switching activity in combinational and sequential circuits. In Proceedings of the Design Automation Conference, pages 253–259, Anaheim, CA, June 1992.

[7] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Probabilistic analysis of large finite state machines. In Proceedings of the Design Automation Conference, San Diego, CA, June 1994.

[8] G. D. Hachtel and F. Somenzi. A symbolic algorithm for maximum flow in 0-1 networks. In Proceedings of the International Conference on Computer-Aided Design, pages 403–406, Santa Clara, CA, November 1993.

[9] F. N. Najm. Transition density, a stochastic measure of activity in digital circuits. In Proceedings of the Design Automation Conference, pages 644–649, San Francisco, CA, June 1991.

[10] K. Roy and S. Prasad. SYCLOP: Synthesis of CMOS logic for low power applications. In Proceedings of the International Conference on Computer Design, pages 464–467, Cambridge, MA, October 1992.