Function Decomposition and Synthesis Using Linear Sifting

Christoph Meinel
FB IV – Informatik
Univ. Trier
D – 54286 Trier

Fabio Somenzi
Electr. and Computer Engineering
Univ. of Colorado at Boulder
USA – Boulder, CO 80309-0425

Thorsten Theobald
Zentrum Mathematik
TU München
D – 80290 München

Abstract

In order to simplify a synthesis task for particularly hard functions it is sometimes inevitable to decompose the function in a preprocessing step. We propose a new algorithm for automatically decomposing a target function by extracting a linear filter within the synthesis process. The algorithm is an application of the Linear Sifting algorithm which has been proposed in [6]. Using this method we were able to synthesize functions with standard tools which fail otherwise.

1 Introduction

Decomposition techniques date back to the very early days of computer-aided circuit design [5]. By revealing the structural properties of a switching function it is possible to establish representations by means of several simpler functions. Utilizing these functional properties enables us to decompose large and complex circuits into a system of smaller subcircuits which may be readily available and economically maintained.

Recent developments concerning symbolic representations of switching functions by means of decision diagrams [2] have renewed the interest in decomposition techniques [3, 11]: By extracting functional properties and decomposing the switching functions the well-known power of symbolic manipulation techniques can be even further extended.

In particular, the recent work [3] dealt with the problem that most multi-level synthesis tools like SIS run into severe problems for complex functions. By decomposing the circuit into the cascade of two suitable subcircuits the synthesis task can be simplified. However, efficient algorithms to find such a decomposition are available only for very specific types of decomposition. An attractive candidate is to divide the function f into a linear block σ and a non-linear block f' (see Fig. 1, [15]) such that $f(x) = f'(\sigma(x))$. The main synthesis task now is to find a linear block σ that minimizes the complexity of f. In [3] this problem has been tackled by methods of spectral analysis. However, although there is a well-established theory on these techniques [4] the practical implementation of these ideas leads to very complex algorithms and the improvements are limited. Hence, an important question is not only to find better algorithms but also to find algorithms which can be integrated much more easily into existing design systems.

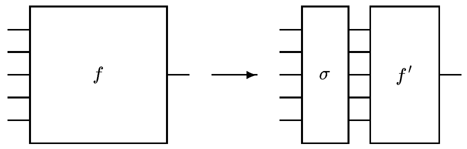


Figure 1: Linear decomposition.

*This work was supported in part by NSF grant MIP-94-22268, SRC contract 95-DJ-560 and DFG-grant Me 1077/12-1.

In this paper we propose to use the Linear Sifting algorithm [6] for the computation of the linear block σ. This algorithm is an extension of the well-known Sifting algorithm [12], which is currently the state-of-the-art method for finding good variable orderings of binary decision diagrams. In the Linear Sifting algorithm, not only swaps of two neighboring variables are applied as elementary operations, but also linear transformations among neighboring variables. This algorithm modifies a given function, but preserves the canonicity in representation and the efficiency of manipulation. We show in what respect Linear Sifting can be used to perform the above mentioned synthesis task, and we prove the validity of our approach by experimental results.

The paper is structured as follows: In Section 2 we recall definitions relevant to decision diagrams and linear transformations. In Section 3 we review the approach of [3] based on spectral analysis. In Section 4 we present our new approach based on Linear Sifting and contrast it to the existing approach. Experimental results are given in Section 5. Finally, the ideas and results are summarized.

2 Preliminaries

2.1 Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) [2] are rooted directed acyclic graphs representing switching functions. Each OBDD has two sink nodes which are labeled 1 and 0. Each internal (= non-sink) node is labeled by an input variable x_i and has two outgoing edges, labeled 1 and 0. A linear variable order π is placed on the input variables. The variable occurrences on each OBDD-path have to be consistent with this order. An OBDD computes a switching function $f : \mathbb{B}^n \to \mathbb{B}$, $\mathbb{B} = \{0, 1\}$, in a natural manner: each assignment to the input variables x_i defines a unique path through the graph from the root to a sink. The label of this sink defines the value of the function on that input.

The OBDD is called *reduced* if it does not contain any vertex v such that the 0-edge and the 1-edge of v lead to the same node, and it does not contain any distinct vertices v and v' such that the subgraphs rooted in v and v' are isomorphic. It is well-known that reduced OBDDs are a unique representation of switching functions $f : \mathbb{B}^n \to \mathbb{B}$ with respect to a given variable order [2]. The *size* of an OBDD is the number of its nodes. Several functions can be represented by a multi-rooted graph called *shared OBDD*. In the following, all functions are represented by a shared OBDD. Furthermore, typical OBDD-implementations use an additional edge attribute to represent a function and its complement by the same subgraph.

2.2 Linear Transformations

A Boolean function $\sigma : \mathbb{B}^n \to \mathbb{B}^n$ is called linear if there exists an $n \times n$-matrix m_σ with Boolean entries such that $\sigma(x) = m_\sigma \cdot x$ where the matrix product $m_\sigma \cdot x$ is evaluated over the Galois field $GF(2)$. Of course, σ and m_σ can be used interchangeably. Additionally we allow to complement some bits of $\sigma(x)$.

3 Synthesis by Spectral Analysis

In this section we revisit the algorithm of [3]. This description will form the basis for the development and judgement of our algorithm.

The problem: Given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$, find a linear transformation σ and a remaining function $f' : \mathbb{B}^n \to \mathbb{B}$ such that the decomposition $f(x) = f'(\sigma(x))$ simplifies the synthesis task and leads to smaller circuits.

The choice of suitable linear transformations is achieved by assigning each Boolean function a complexity measure that is heuristically related to the complexity of the final circuit implementation. The cost of the transformation σ is neglected in this complexity measure on the grounds that the transformation itself is typically much less complex than the remaining function after the transformation. (In our experience this is true in most cases.)

In order to estimate the complexity of the Boolean function f one can use the number of vector pairs (x_1, x_2) which have a Hamming distance $d_H(x_1, x_2)$ of 1 and which have identical function values, i.e.

$$C^n(f) = \#\{(x_1, x_2) : d_H(x_1, x_2) = 1 \text{ and } f(x_1) = f(x_2)\}.$$

This value lies in the range 0 to $n2^n$. As a heuristic criterion, large values of $C^n(f)$ indicate a simple AND/OR-implementation with the extreme case $C^n(1) = n2^n$. On the other hand, small values of $C^n(f)$ indicate an expensive AND/OR-implementation with the extreme case $C^n(x_1 \oplus x_2 \oplus \cdots \oplus x_n) = 0$. Due to the observation that many multi-level synthesis systems perform poorly without a good sum-of-product representation of the target function, the authors of [3] claim that the complexity measure C can also be used for multi-level synthesis.

It has been shown that the complexity measure $C(f)$ can be expressed by the Walsh spectrum of f [4]. The *Walsh spectrum* of f is the 2^n element vector

$$S^n(f) = (s_0, \ldots, s_{2^n-1}) = W^n Y^n(f),$$

where $Y^n(f)$ is the 2^n element truth table vector of f (with 0, 1 replaced by $+1$, -1) and W^n is the $2^n \times 2^n$ *Hadamard matrix* defined by

$$W^n = \begin{bmatrix} W^{n-1} & W^{n-1} \\ W^{n-1} & -W^{n-1} \end{bmatrix}, \qquad W^0 = 1.$$

If $||u||$ is defined as the number of 1's in the binary representation of u, then the complexity measure C can be written as a weighted sum of squares of the Walsh spectral coefficients

$$C^n(f) = n2^n - \frac{1}{2^n} \sum_{u=0}^{2^n-1} ||u||\ s_u^2 .$$

The main idea in [3] to find a suitable linear filter is to construct linear transformations that minimize the complexity C. In order to find suitable candidates for this minimization process the Walsh coefficient representation of the complexity measure is used. To avoid the exponential costs of computing the Walsh transformation symbolic BDD-techniques are used in connection with extensions of the Walsh transformation.

4 Synthesis by Linear Sifting

4.1 The Linear Sifting Algorithm

Originally starting from re-encodings of finite state machines and from the general optimization of BDD-representations two recent papers [7, 6] have investigated the effect of linear transformations on BDDs, concluding that they form a class very well suited to the implementation of efficient algorithms. In [6] an efficient algorithm is presented which combines the efficiency of Sifting and the power of linear transformations. We will briefly review this algorithm.

The well-known Sifting algorithm [12] tries to find a good variable ordering of an OBDD by successively investigating each variable: The current variable is moved through the whole ordering and finally put in the best position that has been found. The basic step during the movement of the variable is a pairwise exchange of variables. Suppose that variables x_i and x_j are to be swapped, and that x_i immediately precedes x_j before the swap. Then the effect of the exchange on each node labeled x_i can be easily seen by applying Boole's expansion theorem w.r.t. both x_i and x_j. Assuming f is the function of a node labeled x_i, we have:

$$f = x_i x_j f_{11} + x_i x_j' f_{10} + x_i' x_j f_{01} + x_i' x_j' f_{00}. \tag{1}$$

Formally exchanging x_i and x_j and rearranging terms yields the function $\phi(f)$:

$$\phi(f) = x_i x_j f_{11} + x_i x_j' f_{01} + x_i' x_j f_{10} + x_i' x_j' f_{00}. \tag{2}$$

In words, the effect of a swap is to interchange f_{10} and f_{01}. If we repeat the same process for the application of an elementary linear transformation $x_i \mapsto x_i \equiv x_j$, we obtain:

$$\phi(f) = x_i x_j f_{11} + x_i x_j' f_{00} + x_i' x_j f_{01} + x_i' x_j' f_{10}. \tag{3}$$

A comparison of Equations (2)-(3) shows that the only difference is that f_{00} is involved in the interchange with f_{10}, instead of f_{01}. The fundamental techniques applied to the efficient swapping of two variables can be used also for their linear combination. By combining swapping and linear transformations it is possible to reduce the size of the BDD in more cases than by swapping only. This observation forms the basis of the Linear Sifting algorithm, which proceeds as follows.

Each variable is considered in turn, and, as in Sifting, it is moved up and down in the order. Let x_i be the chosen variable, and let x_j be the variable immediately following it in the order. One basic step of Linear Sifting consists of the following three phases:

1. Variables x_i and x_j are swapped; let the size of the BDD after the swap be s_1.
2. The linear transformation $x_j \mapsto x_i \equiv x_j$ is applied; let the resulting size of the BDD be s_2.
3. If $s_1 \le s_2$ then the linear transformation is undone. This is obtained by simply applying the transformation again, since it is its own inverse.

The net effect of the three-phase procedure is that x_i is moved one position onward in the order, and possibly linearly combined with x_j.

The usual formulation of linear transformations is in terms of exclusive or, rather than its complement. We use the equivalence function ($\equiv$) because typically, BDDs use the complement arcs; hence, in order to preserve canonicity, we cannot swap f_{11} with one of $\{f_{10}, f_{01}, f_{00}\}$, all of which could be complemented. Since the 1-arcs cannot be complemented, we would have to change the complementation of arcs into the node labeled (initially) x_i. This would make the operation non-local, which is highly undesirable.

The Linear Sifting algorithm is very time-efficient, as it is shown in [6], and it sometimes leads to significant reductions in the sizes of the OBDDs compared to the "conventional" Sifting algorithm.

4.2 Applying Linear Sifting

Our heuristic for extracting a linear filter is to use the (shared) BDD sizes of the functions f', σ in the decomposition $f(x) = f'(\sigma(x))$ as a complexity measure:

$$C^n(f', \sigma) = \text{BDD-size}(f', \sigma).$$

Our algorithm for extracting a linear filter is to apply the Linear Sifting algorithm on the given target function as explained above. Hence, our complexity measure coincides with the optimization criterion of the Linear Sifting algorithm. Particularly, we can exploit the fact that our implementation represents the linear transformation σ within the shared BDD itself. The measure has been inspired by the observation that in some cases Linear Sifting yields significant reductions in the BDD size compared to the conventional Sifting algorithm. Although there is by far no strict correlation between BDD size and the difficulty in the mapping step, we expect functions with relatively small BDD size to be mapped easily. The advantage of our measure is that it is the most attractive one for the preprocessing step (minimizing the BDD size is equivalent to minimizing the memory consumption) and that it somehow seems to be particularly adequate to analyze Boolean functions whose implementation will be obtained through modern BDD-based logic synthesis tools.

Although we did not particularly aim at fully symbolic synthesis algorithms like the one by Minato [10] we also investigated these methods, see Section 5.3. These algorithms start by transforming the OBDD of a function into a ZDD (zero-suppressed BDD, [8]) of its prime-irredundant two-level form before proceeding.

5 Experimental Results

5.1 Main Setup

For the experimental evaluation we tried to use a very similar setup as the one in [3]. Our algorithm has been implemented using the CUDD BDD-package [14] and SIS 1.3 [13]. After building the BDDs in a suitable manner (see below) they are dumped to a BLIF file. In case of using Linear Sifting the transformation is also encoded into this file (in form of a subcircuit). Notice that, for sequential benchmarks, only the combinational portion of the logic has been considered, that is, state inputs and state outputs have been treated as primary inputs and primary outputs, respectively.

The functions are then synthesized under SIS 1.3 using `script.rugged`. Mapping was done using the library `lib2.genlib` with the command `map -m 0` which optimizes the circuit for area. The memory limit was 128 MB, the time limit 20000 s on a Sun Sparc 10.

Our results for a set of 62 benchmark circuits are shown in Table 2. The column `In` represents the number of inputs resp. outputs of the function. The column `Area` shows the area of the mapped circuit, where all numbers are divided by 464, the greatest common divisor of all gate sizes. The delay of the mapped circuit and the time that SIS needs for all the above mentioned tasks are listed in the next two columns. The following two columns show the size of the BDD after the preprocessing step (= the symbolic simulation using Sifting resp. Linear Sifting) and the running times of this preprocessing.

In our experiments we were especially interested in the question if it is possible by using an appropriate preprocessing to synthesize those functions which failed when using only conventional variable reordering. Hence, our experiments were conducted as follows: We used the publicly available very good variable orderings from the CUDD distribution [14] and used the mapping from the corresponding BDDs as our reference. These variable orderings were obtained by a large number of different variable reordering variants. Then we used three combinations of Sifting and Linear Sifting in order to see whether a circuit can be improved or a failed attempt can now be completed. The three combinations are:

1. Start from the well-known good ordering, then apply a final Linear Sifting.
2. Dynamically apply Linear Sifting.
3. Dynamically apply Linear Sifting until Convergence.

By these methods the number of fails could be reduced from 9 to 6. As 2 of these 6 functions completed when starting from a good order, we now only have 4 functions that did not complete for any variant. Concerning the size of the circuits for which both approaches succeed, the decomposition generally seems to produce some overhead. However, in some cases like s1423, C7552 or C2670 significant gains could be achieved by using Linear Sifting. As it is proven by the CPU times, the preprocessing step is very time-efficient.

As a further reference we want to mention the results of [3]. Unfortunately, although they report on a large number of intermediate results of their implementation, they give their final mapping results only for two circuits whose BDDs have more than 31 nodes and compare it to a mapping from a hardware description language, see Fig. 3.

		Starting from a HDL			Using transformation		
	In	Area	Delay	BDD	Area	Delay	BDD
sec	8	674	23.02	109	357	13.95	111
des	256	6213	126.36	7388	7824	117.53	4887

Figure 3: Reference results from [3].

5.2 Separate Mapping

In the above described experimental setup, we were mainly interested in the effect of Linear Sifting when acting as a preprocessing for introducing some structure into the function representation.

In particular for the cases where Linear Sifting leads to far superior results we were interested in the question in how far it is advisable to encode the transformation and the remaining function into one BLIF file or to do the mappings separately. Table 4 shows what happens to the circuits C499, C1355, C1908 when mapping the transformation and the remaining function separately. It reflects the general observation that separate mapping typically does not improve the mapping results in our environment. However, we did not use a specialized algorithm for mapping the transformation (like in [3]). This might improve our results.

5.3 Symbolic Synthesis

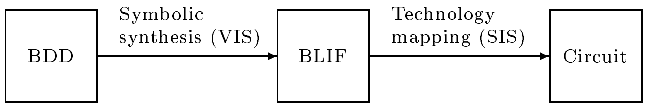
Minato [10] has proposed the use of symbolic BDD- and ZDD techniques for optimizing multi-level logic circuits. They are

		Starting from very good order					Best results - three Linear Sifting BDDs				
	In	Area	Delay	Time	BDD	PTime	Area	Delay	Time	BDD	PTime
9symml	9	116	11.8	8	25	0.7	115	9.9	5	25	0.1
alu4	14	1210	36.8	156	350	0.2	1251	35.6	251	350	0.4
t481	16	90	12.5	3	21	0.9	90	12.5	4	21	1.0
vda	17	1748	25.1	265	478	0.3	1800	29.7	269	478	0.5
cordic	23	137	14.4	6	42	0.1	145	18.0	5	31	0.2
s499	23	561	19.8	49	330	0.1	1031	29.5	105	336	0.6
s344	24	300	18.2	12	104	0.1	366	18.3	14	83	0.3
ttt2	24	341	19.1	14	107	0.1	349	21.6	12.5	107	0.2
s1196	32	1509	29.3	362	598	0.3	1462	33.5	842	599	0.8
C1908	33	TIME OUT			5526	1.7	5148	74.9	3714	1209	23.1
adder16	33	366	73.2	9	82	0.1	481	51.2	18	49	0.6
s635	34	569	40.4	24	128	0.2	569	40.4	25	128	0.6
C432	36	2308	43.5	1995	1064	0.3	2279	45.5	3168	1064	1.1
mm9b	38	4472	46.9	6800	1527	0.7	4394	48.6	12941	1502	1.8
toolarge	38	754	26.7	121	319	1.8	737	29.1	98	319	2.4
mm9a	39	3851	52.5	4259	1111	0.5	TIME OUT			980	2.0
C1355	41	TIME OUT			25866	5.1	3110	54.6	757	484	18.1
C499	41	TIME OUT			25866	4.1	3419	58.0	968	520	10.7
k2	45	3667	35.2	1209	1246	0.6	3681	53.9	1438	1236	4.8
s967	45	1088	22.3	163	366	0.2	1012	24.3	169	383	0.5
C3540	50	TIME OUT			23828	7.7	TIME OUT			23823	120.2
s1269	55	3872	42.3	16456	1695	0.4	3719	46.9	3573	1480	2.2
C880	60	13047	77.6	1761	4053	0.8	13150	78.7	1585	4052	5.2
comp32	64	568	80.1	42	97	0.1	527	76.2	38	97	0.7
adder32	65	734	146.2	30	162	0.2	1019	126.7	58	97	2.2
adsb32	65	1323	145.6	136	345	0.2	1323	110.6	82	191	1.5
s938	66	717	40.1	48	161	0.2	726	44.1	46	161	1.4
sbc	68	1869	21.7	222	917	0.5	1861	19.6	193	917	2.0
alu32	73	2032	95.4	307	478	0.4	2078	82.1	281	388	1.8
alu32r	73	2032	95.4	305	478	0.4	2100	95.1	279	388	3.0
dalu	75	1665	32.5	488	689	0.8	1634	42.1	382	594	12.1
s991	84	1329	79.0	82	328	0.3	1485	116.6	82	328	1.8
s1512	86	1451	23.6	104	566	0.4	1358	29.5	102	526	1.6
xi30	90	675	157.4	22	91	0.3	675	157.4	20	91	1.3
s1423	91	3025	39.7	898	1796	0.6	2533	35.8	577	1477	11.4
dpath32	93	1776	133.6	272	485	0.4	1511	100.9	254	396	3.2
mm30a	123	MEMORY OUT			11065	4.5	27038	226.1	8470	8979	47.8
i3	132	254	9.9	11	133	0.2	290	11.0	13	133	1.0
i8	133	2788	45.4	541	1276	1.3	2860	52.9	719	1276	4.1
apex6	135	1406	25.2	84	498	0.3	1432	34.3	158	498	1.6
s3271	142	2564	29.0	160	831	0.8	2666	29.4	145	671	2.9
frg2	143	1823	36.1	313	963	0.6	1837	35.1	321	921	3.5
s4863	153	MEMORY OUT			64088	26.3	MEMORY OUT			64079	419.0
C5315	178	6374	53.0	542	1719	1.3	5746	64.1	3448	1537	8.5
s5378	199	3727	27.6	468	1932	1.3	MEMORY OUT			1688	56.0
C7552	207	9189	143.3	638	2212	2.0	6182	113.5	315	1150	157.8
s3384	226	2605	83.3	141	693	1.2	2737	42.9	160	597	4.5
C2670	233	6475	71.1	433	1774	0.9	5003	51.8	308	1401	9.3
s9234.1	247	9007	45.1	733	3045	2.2	9045	48.1	688	2731	56.2
comp128	256	2296	318.9	3392	385	0.3	2103	311.2	2852	385	8.9
des	256	7148	120.6	1950	2945	1.7	8106	224.9	4012	2087	162.5
add128	257	2942	584.5	1220	642	0.9	4091	497.3	189	385	46.2
adsb128	257	5435	579.5	10750	1401	1.4	5550	428.8	366	767	24.5
i10	257	TIME OUT			20660	6.2	TIME OUT			19343	650.4
alu128	265	TIME OUT			1918	4.2	8550	323.4	623	1540	60.1
s6669	322	TIME OUT			18033	25.3	TIME OUT			17912	157.3
clma	415	1481	33.3	117	738	9.0	1501	29.7	128	718	13.0
clmb	415	1388	28.4	134	690	9.0	1244	23.7	103	618	24.2
dsip	452	5310	119.7	1192	3033	3.1	10595	155.8	3848	2703	31.1
bigkey	486	5529	126.5	1173	1595	1.4	7817	144.2	1480	1406	22.7
s15850.1	611	25154	111.8	4766	9712	15.8	31514	137.6	7727	10934	236.7
s13207.1	700	7487	55.0	1322	2863	12.8	8622	43.2	1187	2854	42.0

Figure 2: Experimental results.

		Mapping remaining function			Mapping transformation				
	In	Area	Delay	Time	Area	Delay	Time	BDD	PTime
C1355	41	1667	26.0	82	1817	68.6	75.9	484	18.1
C499	41	1663	25.2	93	2296	83.2	83.7	520	10.7
C1908	33	4400	42.6	1906	732	29.9	15.9	1209	32.1

Figure 4: Results from separate mapping.



		Mapping remaining function		
	In	Area	Delay	Time
C1355	41	1945	22.5	314
C499	41	1801	28.9	191
C1908	33	TIME OUT		

Figure 5: Symbolic synthesis: Idea and results.

mainly based on generating a ZDD-representation for a prime-irredundant two-level form of the given BDD and then using efficient divisor extraction algorithms on ZDDs for optimizing the circuit. As these techniques seem to be especially suited when starting a mapping procedure from a given BDD (instead e.g., of a multi-level description), we explored their use for our problem. In our experimental setup, we first used the symbolic algorithms for synthesizing the transformed function and then used SIS for doing the final mapping step, see Fig. 5. For the implementation of the symbolic synthesis algorithms we used the corresponding procedures of the newest VIS releases [1].

The results for some cases where Linear Sifting reduced the BDD size significantly are shown in Fig. 5. A comparison with Fig. 4 shows that the overhead from the two-phase mapping (VIS and SIS) seems to be too big in our context.

A very astonishing result which is also of independent interest in the connection between BDDs and ZDDs is obtained in the case of adders. When using Minato's algorithm to transform a BDD for an n-bit adder to the ZDD of its prime-irredundant two-level form, the ZDD is of linear size. When applying Linear Sifting on the BDD of the adder, the BDD size is reduced by 40 % of the original size. If, however, this linearly sifted BDD is transformed to the ZDD of its irredundant sum-of-product, the size of the resulting ZDD seems to grow exponentially. A thorough description of this effect can be found in the appendix.

6 Conclusion

We have presented and evaluated a new approach for synthesizing and mapping a Boolean function after performing a suitable decomposition. Our method is an alternative to the one proposed in [3]. The experimental results show that this approach may produce good results in cases where the function cannot be synthesized without a decomposition. Our approach is easy to implement in all environments which use dynamic reordering for BDDs, the resulting algorithm is simpler than the one in [3]. It also very well supports the concept of symbolic BDD-representations and may immediately profit from further improvements of the Linear Sifting algorithm.

In general, we think that the full power of decomposition techniques for mapping from a BDD has not been exploited so far. Additional investigations of the auxiliary routines (like Linear Sifting), their combination and the optimization criteria should help to close this gap. Specifically, the results of the appendix suggest that the minimization of the ZDD for the cover of the given function rather than the minimization of the BDD for the function itself may provide a more reliable cost function. (The number of nodes in that ZDD corresponds to the number of literals in a factored form for the function.)

Consequently, we have begun the implementation of Linear Sifting for ZDDs and we expect results soon.

REFERENCES

[1] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, et al. VIS: A system f. verification and synthesis. *CAV '96*, *LNCS* 1102, pp. 428–432. Springer, 1996.

[2] R. E. Bryant. Graph-based algorithms f. Boolean function manipulation. *IEEE Trans. Comp.*, C–35:677–691, 1986.

[3] J. P. Hansen, M. Sekine. Synthesis by spectral translation using BDDs. In *33th DAC*, pp. 248–253, 1996.

[4] S. L. Hurst, D. M. Miller, J. C. Muzio. *Spectral Techniques in Digital Logic.* Academic Press, 1985.

[5] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw Hill, 1978.

[6] Ch. Meinel, F. Somenzi, T. Theobald. Linear sifting of decision diagrams. In *34th DAC*, pp. 202–207, 1997.

[7] Ch. Meinel, T. Theobald. Local encoding transformations for optimizing OBDD-representations of FSMs. In *FMCAD '96*, *LNCS* 1166, pp. 404–418. Springer, 1996.

[8] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th DAC*, pp. 272–277, 1993.

[9] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD.* Kluwer Academic Publishers, 1996.

[10] S. Minato. Fast factorization method for implicit cube representation. *IEEE Trans. CAD*, 15:377–384, 1996.

[11] A. Narayan, J. Jain, M. Fujita, A. Sangiovanni-Vincentelli. Partioned OBDDs – a compact, canonical and efficiently manipulable representation for Boolean functions. In *ICCAD '96*, pp. 547–553, 1996.

[12] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93*, pp. 42–47, 1993.

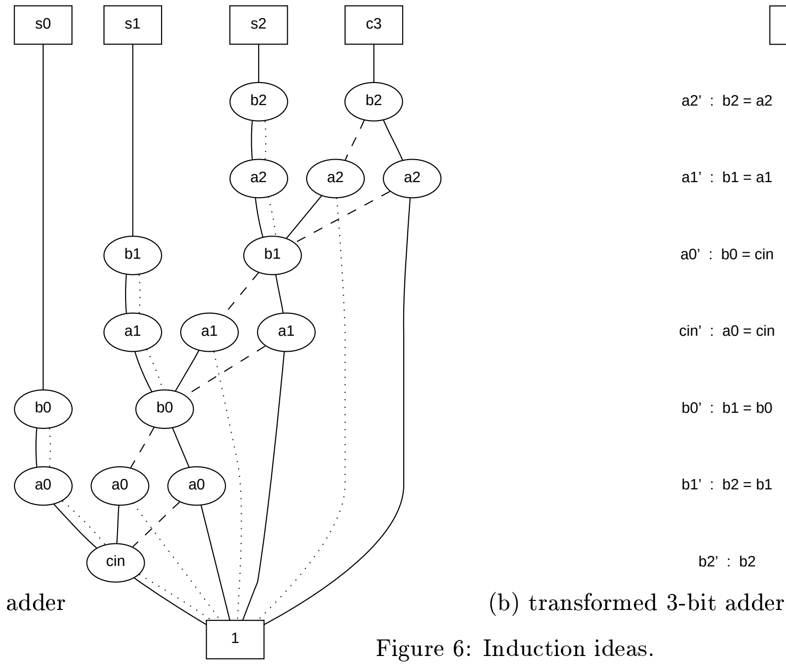
[13] E. M. Sentovich, K. J. Singh, L. Lavagno, et al. SIS: A system for sequential circuit synthesis. Tech. Report UCB/ERL M92/41, Univ. of California, Berkeley, 1992.

[14] F. Somenzi. *CUDD: Colorado University Decision Diagram Package.* ftp://vlsi.colorado.edu/pub/, 1996.

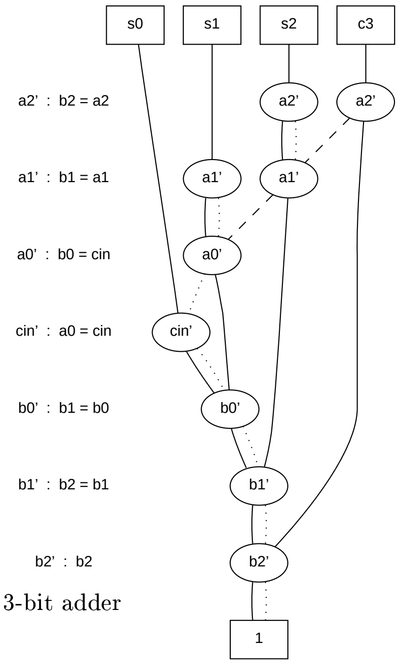
[15] D. Varma, E. A. Trachtenberg. Design automation tools for efficient implementation of logic functions by decomposition. *IEEE Trans. CAD*, 8:901–916, 1989.

Appendix

In this appendix we give some analysis of adder functions. The goal is not only to analyze the blowup for adders reported in Section 5.3 but also to provide a new reference result for unexpected yet important effects caused by combining seemingly



(a) 3-bit adder



(b) transformed 3-bit adder

Figure 6: Induction ideas.

unrelated algorithms on BDDs and ZDDs.

Let $f(a_{n-1}, b_{n-1}, \ldots, a_0, b_0, c_{in}) : \mathbb{B}_{2n+1} \rightarrow \mathbb{B}_{n+1}$ be the n-bit-adder function that takes as input two n-bit strings $a_{n-1} \ldots a_0$, $b_{n-1} \ldots b_0$ and an incoming carry c_{in}, and computes their binary sum $s_{n-1} \ldots s_0$ and the outgoing carry bit c_n. It is well-known that good BDD-orders for the n-bit adder require to keep a_i and b_i together in the ordering, $0 \leq i \leq n-1$. Moreover, an optimal OBDD for the adder function that may use complemented edges needs at least $5n + C$ nodes for a constant C. As it is shown in the following lemma the variable ordering $b_{n-1}, a_{n-1}, \ldots, b_0, a_0, c_{in}$ leads to this optimal bound of (asymptotically) $5n$.

Lemma 1 *For $n \geq 1$, the reduced OBDD (using complemented edges) for the n-bit adder w.r.t. the variable ordering $b_{n-1}, a_{n-1}, \ldots, b_0, a_0, c_{in}$ has exactly $5n + 2$ nodes.*

Proof The lemma can be proven by induction over the number of variables. The idea of the induction should become clear from Fig. 6 (a) which shows the OBDD for a 3-bit adder. In the diagram a 1-edge is indicated by a solid line, a 0-edge by a dashed line and a complemented 0-edge by a dotted line. □

Lemma 2 *For $n \geq 1$, there exists a linear transformation ρ_n such that the OBDD (using complemented edges) of the transformed n-bit adder has exactly $3n + 1$ nodes.*

The transformation in the previous lemma is exactly the one that is found in our implementation of Linear Sifting.

Proof The proof is also by induction. The idea of the induction should become clear from Fig. 6 (b) which can be extended to arbitrary n. On the left side of the figure the functional relation between the original and the transformed variables is given. □

The transformation from an OBDD to a ZDD for its prime-irredundant sum-of-product is achieved by using Morreale's algorithm which has been adapted to the decision diagram environment by Minato [9]. As this ISOP algorithm contains a double recursion, a detailed analysis of the resulting ZDD function quickly becomes quite tedious. However, the following can be proved formally:

Lemma 3 *For $n \geq 1$, the resulting ZDD (which depends on $2 \cdot (2n+1)$ variables) when applying Minato's algorithm on the adder of Lemma 1 has size $11n + 2$.*

Proof Here, the inductive proof idea is that the ZDD for the n-bit adder has the structure which is shown in Fig. 7 with some subgraphs rooted in the nodes A, B. Each variable of the original OBDD is now represented by two variables representing the positive and the negative literal. The shown ZDD structure can be exploited recursively to construct the ZDD for the $(n + 1)$-bit adder. □

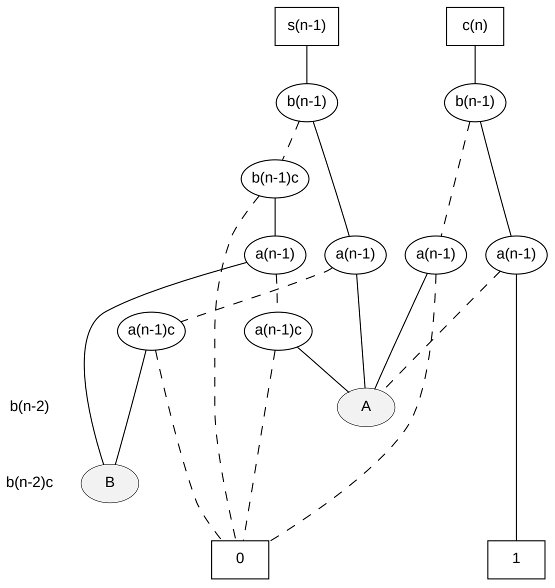


Figure 7: ZDD for the adder.

In contrast to the linear ZDD size resulting from the original adder, the ZDD size resulting from the *transformed* OBDD satisfies the recurrence equation $size(n) = 2 \cdot size(n-1) + 4n + 5$ for all $n \geq 3$ in our corresponding experiments (in particular $n \leq 16$) and hence seems to grow exponentially in n. ◇