Advanced Computer Systems (ACS'98)
pp. 517-524, Szczecin, Poland, November 1998

Exact Circuit Synthesis

Rolf Drechsler Wolfgang Günther

Institute of Computer Science, Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
{drechsle,guenther}@informatik.uni-freiburg.de

Abstract

In this paper we present an exact algorithm for finding a minimal circuit for a given Boolean function. The algorithm is based on graph enumeration and makes use of several pruning techniques to reduce the huge search space. It can handle not only single output but also multiple output functions. The exact netlists obtained are compared to circuits determined by SIS, a state-of-the-art synthesis tool. It turns out that the results can be improved up to 40% also on small examples.

Key words: logic synthesis, exact minimization, multilevel synthesis, pruning techniques

1 INTRODUCTION

Logic synthesis is one of the most important steps in the design process of ICs. Many heuristic synthesis approaches for circuit design have been proposed in the past for two-level and multi-level synthesis [1, 7, 8, 6].

While for two-level realizations powerful tools for exact minimization have been developed [2], exact multi-level optimization has only been considered in the 1960s [4, 3]. But these approaches were of limited practical interest, since they were only applicable to very small functions. However, meanwhile the computational power of modern workstations and the knowledge in efficient data structures has extremely increased. For this reason, exact circuit synthesis becomes an interesting topic again.

In this paper we present an extension of the approach described in [3]: A circuit is considered as a directed acyclic graph. Basically, all possible graphs are enumerated, but in our approach we make use of a clever pruning technique that considers several constraints, i.e. topology of the network, functional behavior, symmetry, and library-specific constraints. Based on these principles and an efficient backtracking method networks with up to 12 gates can be minimized exactly. Extension to optimization not only to area, but also delay are discussed. Furthermore, in contrast to [3] the algorithm can deal with arbitrary libraries (not only $NAND$ gates). Analogously to [5], we merge logic optimization and mapping. However, our approach guarantees optimality of the solution.

2 PRELIMINARIES

In general, a *Combinational Logic Circuit* (CLC) is defined over a fixed library and is modeled as a directed acyclic graph $C = (V, E)$ with some additional properties. Very often a standard library (STD) consisting of primary input (PI) and output ports (PO), the 2-input, 1-output AND and OR gates and the 1-input, 1-output inverter NOT is used. This library is denoted by $Lib_{STD} = \{PI, PO, AND, OR, NOT\}$. Alternatively, in the following $Lib_{NAND} := \{PI, PO, NAND\}$ is used, where $NAND$ denotes the 2-input, 1-output $NAND$ gate. Also other (more complex) libraries can be considered.

Definition 1 *An exact CLC over library* Lib *for a given Boolean function f is a netlist with the least number of gates from library* Lib *among all netlists representing function f.*

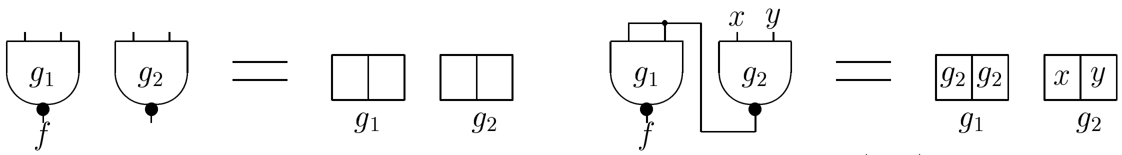

Figure 1: Initial situation and synthesized network for $f(x, y) = x \cdot y$.

Remark 1 *It is a straightforward extension of the approach in the following to introduce cost functions for each gate. By this, some gates that are "cheaper" to realize, e.g. for one specific target technology, can be preferred.*

3 EXACT ALGORITHM

In this section the exact algorithm is explained. First, we give some insight in the underlying data structure. The basic idea of graph enumeration by backtracking is introduced in Section 3.2. Then the pruning technique is described in Section 3.3

For simplicity of the presentation throughout this section library Lib_{NAND} is used, i.e. the exact algorithm determines the optimal *NAND* realization of a Boolean function. The straightforward extension to more complex libraries will be described in Section 4.2.

3.1 Data Structure

First, the underlying data structure is described: All gates in the netlist are stored in an array of size *num_gates*, wherein the first gates represent the output functions. A reference counter is used for both the variables and the gates.

Example 1 *To synthesize function $f(x, y) = x \cdot y$, there are at least two* NAND *gates necessary. The initial state where unassigned gate inputs are left empty and the obvious solution for f are given in Figure 1. For each gate only the connections of the two inputs have to be specified.*

Each gate is assigned an integer number g ($1 \leq g \leq num\_gates$). Output gates get the lowest integer numbers $1, \ldots, num\_outputs$. This simple data structure allows a fast processing and easy manipulation of the netlists. Furthermore, it is very memory efficient.

3.2 Graph Enumeration by Backtracking

Basically, in each step a realization of a given function with a fix number *num_gates* of gates is searched. Starting with 1, this number is incremented as long as no solution is found. The search for a realization with a given number *num_gates* of gates is done by iteratively generating all possible netlists with that many gates.

The generation of all possible netlists with a given number of gates starts with the generation of the gates. The first gates are the output gates and finally have to represent the given function. Initially, all gate inputs are unspecified. In successive steps, the inputs of these gates are assigned with either primary inputs or outputs of other gates. If during the assignments it turns out that it is not possible to complete the netlist so that it represents the correct function, a backtracking step has to be performed. Checking this criteria when backtracking is necessary is the core of the algorithm and will be explained in detail in Section 3.3.

Possible input values to gates are variables and other gates, both output gates and internal gates. When choosing gates, it has to be checked whether the resulting netlist is still acyclic. But these cycles are easy to identify: to decide whether gate h is a possible input to gate g, it only has to be searched for gate g in the set of successors of gate h.

We do not have to consider constant gate inputs for Lib_{NAND}:

input=0: The output of the *NAND* gate is constantly 1 and for this a simpler function exists.

input=1: Then the function realizes an inverter, but this is already generated if both inputs are connected to the same gate.

Similar simplifications hold for other gate types.

As *NAND* is a symmetric function, we can assume that the first input of a *NAND* gate is less or equal than the second one, whereas variables are ordered before gate numbers. A sketch of the basic algorithm is given in Figure 2.

Let us now consider in more detail how for a given netlist the subsequent netlist is determined. This is done in two parts:

```
compute_minimal_NAND_netlist(f)
    for increasing number num_gates of NAND gates
        initialize NAND gates representing the output functions;
        do {
            assign_input();
            if (backtracking_necessary())
                do_backtracking();
        } until netlist computes function f;
    return netlist;
```

Figure 2: Sketch of the algorithm

```
assign_input()
    find lowest unassigned gate input;
    if (it is the first input of a NAND gate)
            assign the variable x_1 to the gate input;
    else    assign the value of the assigned input to the gate input;
```

Figure 3: Sketch of assign_input()

- A first one that assigns gate inputs ("assign_input"): Naturally, assigning inputs is only done for netlists with still unassigned inputs and it has to respect the properties just described. A sketch is given in Figure 3.
- A second one that changes such assignments ("do_backtracking"): When called, in a first step the highest gate input which can be incremented is determined. If no such input exists, we have considered all netlists with the given number *num_gates* of gates. Otherwise, the input value is incremented (respecting again the properties above) and all higher gate input assignments are deleted. Figure 4 gives a sketch of that.

All in all, this algorithm computes the exact result, since all possible netlists are enumerated. But the search space is so large that a complete enumeration is often impossible. For this in the following techniques are described how intermediate netlists that can never compute the correct function can easily be identified.

We now focus on the pruning techniques checked in function *backtracking_necessary()* in the algorithm in Figure 2.

3.3 Pruning Techniques

In the trivial approach, backtracking would be called only if all gate inputs were assigned. But the structure of a partial netlist often allows to get some information, too. These properties can be divided into the following four classes:

1. Topological constraints
2. Symmetry constraints
3. Functional constraints
4. Library-specific constraints

Backtracking can be called each time one of these constraints is not fulfilled. As early calls of backtracking result in the pruning of large parts of the search space, this is very desirable. Note that by using all these constraints, the exactness of the result is not lost, i.e. correctness of the algorithm is preserved.

3.4 Topological constraints

We make use of three topological constraints:

1. The search space can be pruned if some gate occurs having exactly the same inputs as some other gate. Since fanout constraints are not considered, this redundancy can be removed.

```
do_backtracking()
    find last gate input i of gate g that can be incremented;
    increment input i of gate g;
    if (input i is not valid)
        do_backtracking();
```

Figure 4: Sketch of do_backtracking()

2. If the number of remaining gate inputs is less than the number of unused primary input variables, the search space can be pruned, assuming that the function depends essentially on all its input variables (which is checked as preprocessing). If there are r remaining two-input gates (gates with totally unassigned inputs), we know that at most $r+1$ input variables can be assigned to some remaining gate. Thus, backtracking can be performed if there are more than $r+1$ unassigned input variables.
3. Only inputs of referenced gates need to be assigned, i.e. we do not have to assign inputs to a gate g which is not yet connected to some other gate input.

3.5 Symmetry constraints

It has already been mentioned that we can take advantage of the symmetry of the *NAND* function. Additionally, symmetries of the input variables in the function that is to be synthesized are used. Symmetric variables can be exchanged without altering the function. Again in a preprocessing all symmetric variables are grouped. When assigning a variable out of a symmetry group for the first time, we can assume this variable to have the smallest index of the group. All remaining variables do not have to be considered at this position.

3.6 Functional constraints

Now we describe how the functional behavior of a partial netlist can be exploited. To motivate this approach, we first give an example.

Example 2 *Let $f = x + y$. For the first input of the first* NAND *gate g_1 possible values are variables x, y or an internal gate g_2. As x and y are symmetric variables, we only need to check x and g_2. Choosing x, the function cannot be synthesized, as for $x = 0$, the circuit implementation would result in* $\mathrm{NAND}(0, ?) = 1$*, yet the desired function is $f_{x=0} = y$. Therefore, x is not a possible input to gate g_1.*

Exploiting the functional behavior is the key to the effectiveness of our approach. Excluding assignments early allows us to omit many cases which otherwise had to be considered.

The functional behavior of partial netlists can be described using characteristic functions. For a given function $f : \mathbf{B}^n \rightarrow \mathbf{B}^m$, the characteristic function is defined as $\chi_f : \mathbf{B}^{n+m} \rightarrow \mathbf{B}$,

$$\chi_f(x_1, \ldots, x_n, y_1, \ldots, y_m) = 1 \;\equiv\; f(x_1, \ldots, x_n) = (y_1, \ldots, y_m).$$

For complete netlists, the characteristic function is determined by first taking the product of the characteristic functions of all gates and then existentially quantifying over all internal gate variables. For partial netlists, the product is only taken over assigned gates. For multi-output functions, output gates can occur as an input of some other gate. In this case, the desired output function of such output gates can be taken instead of the function represented by the partial netlist, as this gate finally has to represent this function. This results in a large gain for functions with many outputs.

Let χ_f denote the characteristic function of function f that is to be synthesized, and χ_{net} the characteristic function of the partial netlist. If for some input assignment $(x_1, \ldots, x_n)$ and some output assignment $(y_1, \ldots, y_m)$ it holds

$$\chi_f(x_1, .., x_n, y_1, .., y_m) > \chi_{net}(x_1, .., x_n, y_1, .., y_m),$$

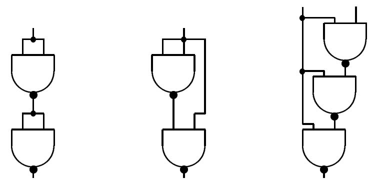

Figure 5: Omitable situations using NAND gates

the network will not realize function f for all possible extensions. Thus for all partial netlists, it must hold $\chi_f \leq \chi_{net}$. So we can prune the search space again if this condition does not hold.

Note that due to representing all characteristic functions as BDDs, the computation of $\chi_f \leq \chi_{net}$ can be done efficiently.

3.7 Library-specific constraints

When using only NAND gates, several combinations need not be considered. Some of them are given in Figure 5. They can be either realized more compact in any netlist realizing the function (since fanout constraints are not considered) or lead to a constant value.

4 EXTENSIONS

In the last section only the number of *NAND* gates has been considered as optimization goal. It is also possible to optimize for delay (see Section 4.1). Furthermore, extensions to other libraries than Lib_{NAND} are described in Section 4.2.

4.1 Optimization for Delay

When optimizing for area, a maximum gate limit has been increased until a feasible solution was found. When optimizing for delay, the gate number is (almost) unlimited; nevertheless a similar approach is possible. Instead of increasing the gate limit, a maximum delay limit has to be increased, until a feasible solution is found. This approach is straightforward and will not be described here in detail; we remark that similar pruning techniques can be used in this case. Also combinations of area/delay are easy to incorporate.

4.2 Arbitrary Gate Types

The approach can also be extended to arbitrary libraries. In this case, the functions *assign_input()* and *do_backtracking()* have to handle an additional parameter which can take the different gate types as values. This additional parameter is handled first, i.e. when assigning inputs, the gate type is determined first and then the real inputs are assigned (which need not be only two as in case of *NAND* gates).

Different pruning possibilities occur when using other gate types. For example, *AND* gates need not have two equal inputs, as in that case the gate could be left out. It may also be necessary to allow constant inputs, too, or arbitrary gates need not be symmetric, as in the case of *NAND* gates. This both holds is the case of libraries containing multiplexors.

5 EXPERIMENTAL RESULTS

The algorithm described above has been implemented in *C*. All experiments have been carried out on a *SUN Ultra 1* with 64 MBytes of main memory. All runtimes are given in CPU seconds. The exact algorithm is compared to SIS [7] using script *rugged* and mapped for area optimization.

5.1 Random Functions

We start by showing the effect of our pruning techniques. For this in a first series of experiments, the algorithm has been executed for all 256 functions with 3 variables using library Lib_{NAND}. The average improvement of the number of gates is 0.63. Some functions, denoted by "f3-xx", where a significant improvement could be obtained are given in Table 1. In column "i/o", the number of inputs and outputs is given, respectively. Column "sis" gives the number of *NAND* gates of the solution network obtained by SIS. The exact minimum of

circuit	i/o	sis	exact	no pruning	χ	pruning	gain
f3-24	3/1	10	7	6696.05 s	0.43 s	0.09 s	30 %
f3-200	3/1	5	3	0.06 s	0.02 s	0.01 s	40 %
f3-231	3/1	9	6	273.01 s	0.12 s	0.05 s	33 %
f4-0	4/1	13	9	—	215.09 s	28.54 s	31 %
f4-1	4/1	18	10	—	3216.14 s	296.17 s	44 %
f4-2	4/1	11	9	—	207.97 s	36.86 s	18 %
f4-3	4/1	14	10	—	2093.93 s	319.87 s	29 %
f4-4	4/1	13	8	—	69.48 s	11.75 s	38 %
f4-5	4/1	6	6	4207.54 s	0.14 s	0.09 s	0 %
f4-6	4/1	14	12	—	—	91059.31 s	14 %
f4-7	4/1	21	11	—	29975.08 s	2958.92 s	48 %
f4-8	4/1	15	11	—	97835.71 s	10609.16 s	27 %
f4-9	4/1	15	11	—	33406.46 s	2150.74 s	27 %

Table 1: Experimental results for random functions

circuit	i/o	sis	exact	time	gain
5xp1-hdl.out<1>	2/1	5	4	0.01 s	20 %
5xp1-hdl.out<2>	3/1	7	7	0.22 s	0 %
5xp1-hdl.out<3>	4/1	14	10	182.28 s	29 %
5xp1-hdl.out<9>	7/1	12	10	109.78 s	17 %
b1	3/4	12	10	105.02 s	17 %
c17	5/2	9	6	0.03 s	33 %
cm42a.e,f	4/2	14	9	19.74 s	36 %
cm42a.g,h	4/2	13	8	9.43 s	38 %
cm42a.i,j	4/2	13	8	9.47 s	38 %
cm42a.k,l	4/2	12	7	1.39 s	42 %
cm42a.m,n	4/2	13	8	9.33 s	38 %
decod.f,g	5/2	17	10	68.61 s	41 %
decod.h,i	5/2	18	11	1603.07 s	39 %
decod.j,k	5/2	18	11	1592.00 s	39 %
decod.l,m	5/2	19	12	27818.76 s	37 %
decod.n,o	5/2	18	11	1620.73 s	39 %
decod.p,q	5/2	19	12	26921.29 s	37 %
decod.r,s	5/2	19	12	26445.97 s	37 %
majority	5/1	14	9	37.43 s	36 %
x2.k,l	3/2	11	8	8.11 s	27 %
x2.m,o	4/2	11	10	119.66 s	9 %
z4ml.27	3/1	10	8	0.60 s	20 %
sum		298	201		33 %

Table 2: Experimental results for *NAND* networks

NAND gates determined by the algorithm described in this paper is given in column "exact". Column "no pruning" gives the runtime if no pruning technique is used at all, while in column "χ" runtimes are given for the case that the check for the characteristic function (see Section 3.6) is the only pruning technique. In column "pruning", all pruning techniques presented in Section 3.3 are included. We used an upper limit of 100000s for the CPU times.

The algorithm was also executed on 256 randomly chosen functions with 4 variables. Results for the first ten of these functions are given in lines "f4-xx" of the same table. For 4 out of 256 functions, the optimum results could not be computed within 3 CPU days. The average improvement of the number of gates for the remaining 252 circuits is 3.68. For the functions considered the runtime of the algorithm and the percentage of improvement on the circuit size are given in the last two columns of Table 1. For some examples almost 50% improvement can be obtained (see e.g. *f4-7*).

circuit	i/o	sis		exact		time	gain	
		gates	cost	gates	cost		gates	cost
5xp1-hdl.out<1>	2/1	5	13	4	11	0.03 s	20 %	15 %
5xp1-hdl.out<2>	3/1	7	18	6	16	0.92 s	14 %	11 %
5xp1-hdl.out<3>	4/1	14	38	9	25	4645.13 s	36 %	34 %
b1	3/4	11	29	9	24	203.75 s	18 %	17 %
c17	5/2	7	20	7	20	16.41 s	0 %	0 %
cm138a.l	6/1	8	21	6	17	1.83 s	25 %	19 %
cm138a.m	6/1	8	21	6	17	1.75 s	25 %	19 %
cm138a.n	6/1	7	19	6	17	1.62 s	14 %	11 %
cm42a.e,f	4/2	7	20	5	14	0.33 s	29 %	30 %
cm42a.g,h	4/2	8	22	6	16	2.82 s	25 %	27 %
cm42a.i,j	4/2	8	22	6	16	2.79 s	25 %	27 %
cm42a.k,l	4/2	9	24	6	16	2.53 s	33 %	33 %
cm42a.m,n	4/2	8	22	6	16	2.80 s	25 %	27 %
cm82a.ff	3/1	9	24	8	22	45.91 s	11 %	8 %
decod.f,g	5/2	9	26	6	17	4.60 s	33 %	35 %
decod.h,i	5/2	10	28	7	19	46.77 s	30 %	32 %
decod.j,k	5/2	10	28	7	19	46.65 s	30 %	32 %
decod.l,m	5/2	11	30	7	19	45.77 s	36 %	37 %
decod.n,o	5/2	10	28	7	19	47.66 s	30 %	32 %
decod.p,q	5/2	11	30	7	19	46.31 s	36 %	37 %
decod.r,s	5/2	11	30	7	19	46.59 s	36 %	37 %
decod.t,u	5/2	11	30	7	19	30.29 s	36 %	37 %
f51m.48	4/1	14	38	9	25	4395.36 s	36 %	34 %
f51m.49	3/1	7	18	6	16	0.87 s	14 %	11 %
f51m.50	2/1	5	13	4	11	0.03 s	20 %	15 %
majority	5/1	9	27	8	24	1476.35 s	11 %	11 %
z4ml.27	3/1	9	24	8	22	45.64 s	11 %	8 %
sum		243	663	180	495		26 %	25 %

Table 3: Experimental results for *AND/OR/NOT* networks

As can be seen the pruning techniques are very effective. Without pruning even very small functions with only three variables may have very long runtimes. Functions with more than four variables cannot be handled at all.

5.2 *NAND* Networks

In a second series of experiments, we compared the algorithm on several benchmark functions. Results are given in Table 2. For the benchmarks *cm42a* and *decod*, we first considered the outputs separately, but none could be improved in comparison to SIS. However, when combining two of the outputs, the results obtained by SIS could be improved significantly. E.g. for the benchmarks in Table 2 the average improvement of the gate numbers is 4.4 corresponding to more than 30%. Thus, it turns out that even for small functions with more than one output the results obtained by SIS can be improved.

5.3 *AND/OR/NOT* Networks

As described in Section 4.2, the algorithm can be extended to arbitrary gate types. As library we chose Lib_{STD}. *AND* and *OR* gates were assigned an area usage of 3, whereas *NOT* gates got a value of 2. Again, the algorithm has been executed on several benchmarks and the resulting area of the circuits has been compared to that obtained by SIS. The results are given in Table 3. In column "gates" the number of gates is given and column "cost" shows the corresponding cost with respect to the gate weights given above. Note that due to the larger library, more netlists have to be considered for the same maximum number of gates. However, the resulting netlists often need less gates and thus the algorithm is applicable to a similar set of benchmark functions.

circuit	i/ o	sis	post	gain
alu2	10/ 6	460	426	7 %
cc	21/20	56	56	0 %
cm151a	12/ 2	27	27	0 %
cm163a	16/ 5	60	47	22 %
cm42a	4/10	36	28	22 %
cm82a	5/ 3	26	18	31 %
cm85a	11/ 3	70	57	19 %
comp	32/ 3	159	135	15 %
cordic	23/ 2	102	84	18 %
cu	14/11	84	74	12 %
i3	132/ 6	318	254	20 %

circuit	i/ o	sis	post	gain
i5	133/66	198	198	0 %
parity	16/ 1	75	60	20 %
pcle	19/ 9	75	67	11 %
pm1	16/13	100	74	26 %
rd53-hdl	5/ 3	37	27	27 %
sao2-hdl	10/ 4	222	207	7 %
term1	34/10	214	203	5 %
ttt2	24/21	263	253	4 %
x1	51/35	410	386	6 %
x2	10/ 7	100	78	22 %
z4ml	7/ 4	49	37	24 %

Table 4: Heuristic results for NAND networks

For this library it turns out that the average improvement of the number of gates also becomes much better. For many benchmarks an improvement of more than 30% over SIS can be obtained.

6 CONCLUSIONS

We presented an exact algorithm for determining a minimal circuit for a given Boolean function. It is based on a complete enumeration of all possible graphs, but makes use of efficient pruning techniques. Experiments have shown that the exact method is applicable to networks with up to 12 gates. In many cases a significant improvement over SIS has been obtained.

It is possible to apply the exact approach to larger netlists optimized by SIS in the form of a post-processing. Small parts of a netlist are chosen and substituted by a functional equivalent netlist which is determined using the exact algorithm. Results for some larger circuits are given in Table 4 using library Lib_{NAND}. Column “sis” gives the number of NAND gates in the result of SIS. In column “post”, the results after the post-processing are given. These first experiments show that for most benchmarks further improvement (of up to 27%) can be observed.

REFERENCES

[1] R.K. Brayton, G.D. Hachtel, C. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[2] O. Coudert. Two-level logic minimization: an overview. *Integration the VLSI Jour.*, 17(2):97–140, 1994.

[3] E. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Trans. on Comp.*, 18:1098–1109, 1969.

[4] E.L. Lawler. An approach to multilevel boolean minimization. *Journal of the ACM*, 11:283–295, 1964.

[5] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. In *Int'l Conf. on CAD*, pages 264–271, 1995.

[6] C. Scholl and P. Molitor. Communication based FPGA synthesis for multi-output Boolean functions. In *ASP Design Automation Conf.*, pages 279–287, 1995.

[7] E. Sentovich, K. Singh, L. Lavagno, Ch. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.

[8] B. Wurth, K. Eckl, and K. Antreich. Functional multiple-output decomposition: Theory and implicit algorithm. In *Design Automation Conf.*, pages 54–59, 1995.