International Conference on VLSI Design (VLSI Design'01)
Bangalore, India, January 2001

Performance Driven Optimization for MUX based FPGAs

Wolfgang Günther
Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg, Germany
guenther@informatik.uni-freiburg.de

Rolf Drechsler
Siemens AG
Corporate Technology
81730 Munich, Germany
Rolf.Drechsler@mchp.siemens.de

Abstract

Logic synthesis and technology mapping for Multiplexor (MUX) based Field Programmable Gate Arrays (FPGAs) have widely been considered. In this paper, an algorithm is proposed that applies techniques from logic synthesis during mapping to minimize the delay. By this, the target technology is considered during the minimization process. Binary Decision Diagrams (BDDs) are used as an underlying data structure for netlists, combining both structural and functional properties. To evaluate a netlist, a fast technology mapper is used. Since most of the changes to a netlist are local, also re-mapping can be done locally, allowing a fast but reliable evaluation after each modification. We give experimental results that show large improvements for most of the instances we considered.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have been widely used in many implementations. Since time-to-market aspects become more and more important, they are commonly applied for rapid prototyping due to the short turnaround time.

FPGAs are mainly based on a basic cell type that consists e.g. of a *Look-Up Table* (LUT) or a *Multiplexor* (MUX) structure. In the meantime, not only the size of FPGA implementations has grown a lot (nowadays FPGAs can provide more than one million gates), but also their speed has increased tremendously, and therefore it is possible to use FPGAs even in time-critical applications.

Logic synthesis for FPGAs is usually divided into a technology independent logic optimization phase followed by technology mapping. For area minimization, many successful approaches to combine these phases of the design process targeting *Pass Transistor Logic* (PTL) have been proposed in the past. In [4] a decomposed BDD representation and variable reordering is used. In [5], a BDD based synthesis flow for PTL is presented. Basic operations to cells are applied. Also local don't cares are used to minimize the BDDs of cells. Another approach is to perform logic optimization after technology mapping, i.e. to carry out logic restructuring operations on the mapped netlist. This was mainly proposed to minimize the delay [20, 7, 12]. For an overview on FPGA design, see e.g. [2, 18].

In this paper we present an approach to combine logic synthesis and technology mapping. We use FPGAs based on Actel-1 cells [1] as a target technology. The algorithm starts with a pre-optimized netlist, obtained by a standard tool, like SIS [23]. Then different optimization operators are carried out, using technology mapping and delay computation to evaluate the intermediate steps. By this, the actual influence of the modification to the area and delay of the resulting FPGA implementation is found out.

It has been observed in [8, 14] that the combination of these two steps tends to be time consuming. However, since most modifications only affect small parts of the netlist, mapping and the delay computation have to be re-run only on the modified regions to update the delay value, allowing a fast evaluation. An efficient representation of the subcircuits, i.e. *Binary Decision Diagrams* (BDDs) [3, 9], is used to represent the function of cells which allows to merge cells very quickly. This technique combines both functional and structural properties of the netlist. To minimize the size and the delay of the BDDs, we apply problem specific variable ordering techniques.

The algorithm has been incorporated in the program ACTion [13]. We give comparisons to SIS [23] mapping for delay and to a recently published logic restructuring approach [12] which is used as a post-processing to results which were obtained by a commercial tool, namely the Actel Designer[1]. It is shown that a smaller delay is obtained in most cases compared to all these approaches and a significant reduction can be observed on average.

[1] http://www.actel.com/products/systems/design_tools.html

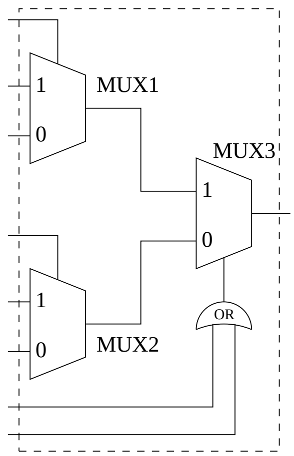
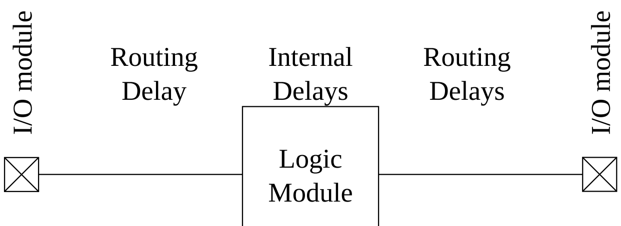


Figure 1. Architecture of Actel-1



fanout	delay
1	1.4 ns
2	2.2 ns
3	3.3 ns
4	4.8 ns

delay
4.5 ns

fanout	delay
1	1.4 ns
2	2.2 ns
3	3.3 ns
4	4.8 ns

Figure 2. Delay Model of Actel-1

2 Problem Formulation

During technology mapping a given netlist is transformed to a functional equivalent representation with respect to a target technology. One of the main problems results from the fact that this target technology is usually not considered during logic synthesis.

The approach in this paper focuses on one specific cell type, i.e. the Actel-1 cell. The basic cell is given in Figure 1. Several technology mappers have been proposed in the past for this architecture (see e.g. [16, 17]). Note that our approach can easily be extended to other cell types. However, a MUX based structure is of advantage because of the similarity to BDDs.

For the rest of the paper we only focus on delay minimization, i.e. we try to find a representation that minimizes the delay of the resulting implementation. We use the original Actel delay model [1] (see Figure 2), i.e. the propagation delay is modeled using the fanout driven by a module.

```
map(MUX netlist)
    set_of_gates_to_realize := set of output gates;
    while (set_of_gates_to_realize is not empty)
        g := element of set_of_gates_to_realize
                with minimal fanout;
        do
            for (all fanin gates h, sorted by the
                        number of resulting inputs)
                merge g and h, if the resulting cell
                is realizable with one Actel-1 cell;
        while (improvement);
        insert Actel-1 cell to mapped netlist;
        insert fanin gates to set_of_gates_to_realize;
```

Figure 3. Sketch of mapping algorithm

3 Combining Logic Synthesis and Technology Mapping

First, the underlying data structure is outlined. Then the mapping and delay computation algorithms are explained. The preprocessing phase is outlined in Section 3.4. In Section 3.5, the overall algorithm is described. Finally, some details of the algorithm are given.

3.1 Representation of the Netlist

A data structure based on BDDs is used to represent a netlist having cells of arbitrary functionality: the function of each cell is described by a BDD. By this it is possible to decompose the function into blocks which can be represented by Actel-1 cells. Also merging of cells is easy, since only the BDD for the combined cell has to be constructed. In summary, this representation combines both structural and functional properties of the netlists.

3.2 Actel-1 Mapper

To map a netlist to Actel-1 cells, the netlist is first mapped to MUX cells: each node of the BDDs representing the blocks' functions are replaced by a MUX gate. Note that this mapping is done on a copy of the netlist, since the netlist should not be changed by the mapping algorithm.

A relatively simple greedy strategy is used to map MUX netlists to Actel-1 cells. Starting from the primary outputs of the circuit, a gate is chosen and assigned to a new Actel-1 cell. Then the fanin gates are added to the cell one after another until no further gate fits into the cell. Now all gates in the fanin which could not be added to the cell become "output" cells since they have to be represented by separate cells. A sketch of the mapping algorithm is given in Figure 3.

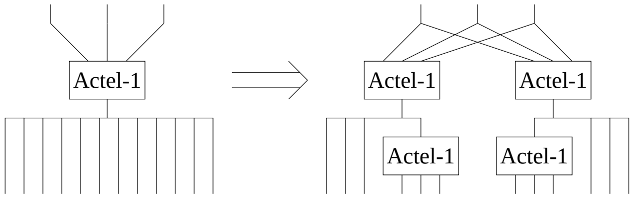


Figure 4. Fanout tree to limit fanout

To decide whether a gate can be added to an Actel-1 cell, in a preprocessing step the set of functions which can be realized with one Actel-1 cell is computed. A signature-based approach is used to speed up Boolean matching (for more details, see [13]).

3.3 Delay Computation

Applying the Actel-1 mapper just described results in a mapped netlist. Since we restrict to delay minimization and the delay of logic cells becomes very large if the fanout is high (see Figure 2), we use a fanout restriction of 4 in the following. To achieve this limitation, we use a combination of building fanout trees and gate duplication [6]. More precisely, the fanout restriction to 4 is computed using the following algorithm:

```
for (each Actel-1 cell g of the circuit,
        traversed in topological order)
  if (fanout(g) > 4)
    duplicate g until the fanout of all inputs is ≤ 4;
    compute balanced fanout tree for each duplicate;
```

In Figure 4 an example of splitting a large fanout is given. No further timing optimization has been implemented to keep the runtimes moderate.

3.4 Preprocessing

Library-invariant optimization tools are used to find a good starting point. We used three different starting points, which are mainly based on SIS script *delay* and two variants of a global BDD construction (if possible within given memory limits, see [13]). The best starting point (in terms of delay of the mapped circuit) is chosen.

3.5 Overall Algorithm

Starting from a netlist obtained in the preprocessing step, local synthesis operations are applied. These modifications may result in cells consisting of BDDs with many nodes. We used the following operations:

1. Change polarity of one cell [19].
2. Merge one cell with all its fanin cells. In a first step, each cell is merged with its fanin cells one after another. To avoid local minima, in a second step also larger regions of fanin cells are merged, unless the BDDs for the cells become too large.
3. Merge one cell with all its fanout cells.
4. Merge two cells with similar inputs. As the resulting cell has two outputs, the shared BDD representation for the cell is mapped to MUX cells. By this, many small cells are created, which enables further merging steps.

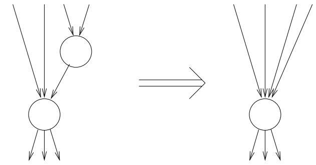


Figure 5. Merging two cells

After each modification the resulting number of Actel-1 cells and the delay is computed using the mapper and delay model of Section 3.2 and Section 3.3, respectively. Since by most modifications only small parts of the netlist are affected, the evaluation has to be updated only locally. If the overall delay caused by the transformation is worse, the last modification is undone and the previous netlist is restored. If the resulting delay is equal, then the area value is used to decide whether the modification was an improvement. The optimization steps are iterated until no further improvement can be obtained (or a user defined time limit is reached).

Then, the netlist is mapped to a multiplexor netlist, leading to tiny cells consisting of only one MUX gate. This is also done in each mapping step, but now the small cells are part of the new netlist, whereas the netlist remains unchanged in each mapping step.

If the size of the intermediate BDDs grows too much, the merging process can be stopped, as it is very unlikely (yet desirable) that the delay can be reduced by further merging steps.

The local optimization cycle is carried out three times. All in all, the flow of the algorithm is very simple and the good quality (see below) mainly results from the technology dependent optimizations during *local optimization* and the good BDD minimization techniques.

3.6 Merging Cells

Merging two cells is the main operation of local optimization. Basically, this is done by building the BDD for the merged cell, i.e. a variable is substituted by another BDD (see Figure 5).

As the size of the resulting BDD directly influences the mapping result, a minimal representation is very desirable. Therefore, we apply different methods for variable reordering. If the number of variables is small

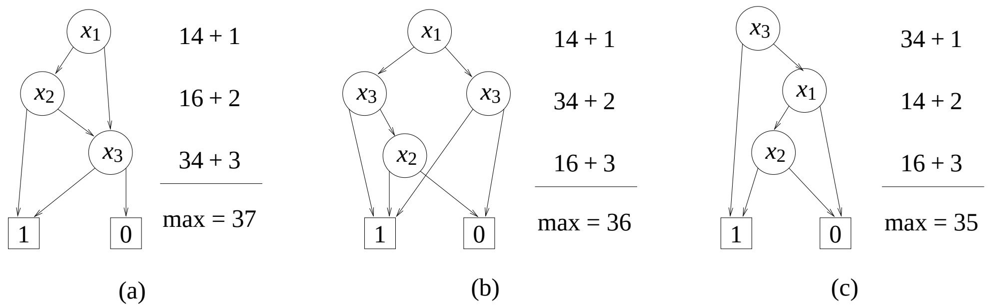


Figure 6. Delay sifting of variable x_3

enough (less than 10), an exact algorithm [10] is used first, and second delay sifting [22] is called, which will be briefly described below. If the number of variables is too large for the exact algorithm to finish quickly, only delay sifting is used. By this procedure, in most cases a good variable ordering is found. To further minimize the BDD sizes, we make use of local don't cares [15], since smaller BDD sizes lead to smaller fanout values, which may reduce the total delay.

3.6.1 Delay Sifting

One drawback of BDD based synthesis in general is that BDDs have linear depth, resulting in large delays of the circuit. To overcome this limitation, it has been shown in [22] that for sifting, different optimization criteria than the node count can be used. In [22], the delay of the inputs were used to minimize the delay of a BDD, using PTL.

The main idea of delay sifting is that the different BDD variables correspond to outputs of cells which may have a different delay. This delay information is used to minimize the delay of the BDD output: the delay is estimated by

$$delay(\text{BDD}) = \max_{x_i} (\text{delay}(x_i) + \text{level}(x_i))$$

(see Figure 6). To estimate the delay of the input variables, the unit delay model is applied to the (unmapped) netlist, assuming that a BDD consisting of n variables has delay n.

Ordinary sifting [21] tries to optimize the position of each variable, assuming that the relative order of the other variables remains the same. The optimal position is found by moving the considered variable to all positions of the variable ordering, using the total node count as cost function. For delay sifting, the cost function is extended to

$$\alpha \cdot size + \beta \cdot delay.$$

In our implementation we used $\alpha = 1$ and $\beta = 2$, i.e. giving more weight to delay than to area. However, it turned out in the experiments that small changes of these parameters do not have much influence on the final results, although the weight for area should not be chosen too small, since this leads to large BDD sizes which result in large fanout values, increasing the overall delay.

4 Experimental Results

In this section we describe our experimental results. All experiments are carried out on a *SUN Ultra 4* workstation. The CUDD package [24] is used as underlying BDD package. A memory limit of 256 MB and a runtime limit of 10,000 CPU seconds is set in all experiments. The high runtime limit is used because the quality of the results was our main interest. Our technique has been implemented in the program ACTion [13] and in the following it is denoted as ACTion:–D.

In a first series of experiments, we compared ACTion:–D to the results of the ite_map algorithm[2] which is implemented in SIS [23] (see Table 1). We first run *script.delay* for both approaches. In column SIS, area (in terms of Actel-1 cells) and delay of ite_map is given. The delay values of both approaches were computed using the algorithm from Section 3.3. For ACTion:–D, also the CPU time is given. The best values are given in bold. A large reduction can be seen for many instances, sometimes even more than 50% (see e.g. *alu2*, *x2*). Moreover, also area is reduced in many cases (see e.g. *e64* for a reduction of almost 70%). On average, delay is 16.7% smaller and at the same time area is reduced by 15.3%.

In [12], a post-processing step was presented which is based on redundancy addition and removal [11]. In a second series of experiments, we compared ACTion:–D to this approach (see Table 2). In columns *in* and *out*, the number of primary inputs and outputs of each circuit are given. The results of [12] starting from SIS *script.delay* using *act_map* before postprocessing are given in column *SIS*, whereas in column *Designer* the Actel Designer, a commercial tool, was used instead of *act_map*. The numbers are taken from [12]. In the last

[2] using `ite_map -n 2 -r`, minimizing for delay (which requires to modify a hard-encoded parameter inside of SIS).

Table 1. Comparison to *ite_map*

circuit	SIS		ACTion:–D		
	area	delay	area	delay	time
5xp1	87	66.6	**60**	**40.4**	93.6
9sym	194	88.3	**17**	**41.2**	93.5
alu2	295	152.6	**170**	**63.0**	7703.7
alu4	612	182.0	**329**	**77.5**	10000.0
apex2	327	113.5	**292**	**84.0**	10000.0
apex3	1576	113.7	**876**	**96.8**	10000.0
apex6	571	95.8	**543**	**70.3**	10000.0
apex7	156	67.4	**140**	**50.4**	7439.7
b9	**72**	**44.4**	87	47.1	1385.4
bw	160	67.5	**120**	**53.8**	250.1
c1355	264	**107.1**	**259**	123.0	10000.0
c17	4	15.9	**3**	**14.8**	2.2
c1908	439	146.8	**379**	**144.8**	10000.0
c2670	489	169.2	**437**	**128.3**	10000.0
c3540	970	221.0	**852**	**201.5**	10000.0
c432	133	162.9	**127**	**138.8**	10000.0
c5315	**1767**	183.0	1900	**176.4**	10000.0
c6288	2128	510.8	**1936**	**478.8**	10000.0
c7552	1881	**159.5**	**1846**	214.9	10000.0
c8	76	47.0	**55**	**31.8**	51.6
c880	318	140.9	**276**	**111.5**	10000.0
cc	**34**	37.7	43	**25.9**	48.1
clip	131	94.2	**48**	**49.7**	123.4
cm150a	26	52.9	**15**	**31.8**	135.0
cm151a	**12**	31.7	13	**25.9**	10.6
cm162a	**20**	42.9	30	**31.8**	22.4
cm163a	24	35.1	**17**	**23.3**	9.6
cm85a	**16**	**38.5**	21	38.9	84.3
comp	80	78.0	**59**	**42.9**	10000.0
cordic	**31**	46.9	37	**43.7**	86.2

circuit	SIS		ACTion:–D		
	area	delay	area	delay	time
cu	34	37.7	**26**	**25.2**	45.7
dalu	**899**	143.2	930	**124.3**	10000.0
des	**3013**	**113.7**	3121	150.1	10000.0
duke2	466	82.4	**366**	**68.2**	10000.0
e64	828	**62.2**	**251**	76.6	10000.0
ex4	**355**	75.0	374	**67.4**	10000.0
example2	258	**72.4**	**231**	81.7	10000.0
f51m	87	57.3	**45**	**38.5**	63.5
frg1	82	78.3	**58**	**62.9**	860.3
frg2	705	**83.3**	**580**	102.9	10000.0
i2	119	66.3	**78**	**48.6**	4115.6
i4	126	66.3	**103**	**44.3**	1178.9
i5	223	70.6	**92**	**53.4**	6231.6
k2	1177	126.4	**1036**	**119.8**	10000.0
majority	5	21.0	**3**	**17.4**	2.2
misex2	78	47.8	**60**	**41.2**	131.0
my_adder	161	125.7	**147**	**113.8**	10000.0
pair	1183	**91.9**	**1153**	103.6	10000.0
rd84	95	75.9	**46**	**44.5**	494.8
rot	496	**92.3**	**455**	92.6	10000.0
sao2	**92**	71.4	**92**	**46.3**	1313.0
spla	711	95.9	**572**	**92.4**	10000.0
t481	768	132.3	**11**	**32.5**	49.4
term1	140	71.4	**78**	**46.3**	8235.5
unreg	82	44.5	**46**	**26.7**	42.4
vg2	110	**56.2**	**69**	60.8	739.3
x2	28	57.3	**24**	**25.9**	13.6
x3	**438**	**61.5**	541	69.8	10000.0
x4	**332**	78.3	450	**77.6**	10000.0
z4ml	40	47.8	**9**	**24.1**	8.8
sum	26024	5620.1	**22034**	**4682.4**	

column, the results of ACTion:–D are given. It can be seen that due to the combination of logic synthesis and technology mapping, most of the resulting delays are smaller than using SIS or the Actel Designer. Compared to Actel Designer, reductions of more than 40% are possible (see e.g. *cm151a*). On average a reduction of more than 10% can be observed.

5 Conclusions

In this paper we presented an approach to combine logic synthesis and technology mapping for performance-driven optimization of MUX based FPGAs. BDDs were used to represent cells of a netlist. A fast technology mapper that can be restricted to parts of the netlist was used to evaluate each modification. The user can specify parameters, like a runtime limit or the maximal size of intermediate BDDs, in order to trade off quality of the results versus runtime. A comparison on a large set of benchmarks to the best previously published results and a commercial tool are given, clearly demonstrating the efficiency of the approach.

References

[1] Actel. ACT™1 series FPGAs. Also available at http://www.actel.com/docs/databook97/section01/97s01d07.pdf, 1997.

[2] S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publisher, 1992.

[3] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[4] P. Buch, A. Narayan, A.R. Newton, and A.L. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *Int'l Conf. on CAD*, pages 663–670, 1997.

Table 2. Comparison to [12]

circuit	in	out	SIS		Designer		ACTion:–D		
			area	delay	area	delay	area	delay	CPU time
alu4	14	8	394	172.4	415	96.4	**329**	**77.5**	10000.0
apex6	135	99	423	97.2	**406**	**59.0**	543	70.3	10000.0
c17	5	2	7	15.1	**3**	**14.8**	**3**	**14.8**	2.2
c8	28	18	69	47.2	56	41.7	**55**	**31.8**	51.6
cc	21	20	40	43.7	**27**	26.6	43	**25.9**	48.1
cm150a	21	1	16	39.7	**11**	**27.9**	15	31.8	135.0
cm151a	12	2	**9**	35.1	13	43.6	13	**25.9**	10.6
cm162a	14	5	24	39.6	**19**	49.9	30	**31.8**	22.4
cm85a	11	3	22	**35.4**	**15**	36.7	21	38.9	84.3
cordic	23	2	**30**	39.5	36	**35.5**	37	43.7	86.2
cu	14	11	29	42.5	**22**	27.8	26	**25.2**	45.7
f51m	8	8	36	76.1	**34**	52.0	45	**38.5**	63.5
frg2	143	139	**454**	116.5	499	**88.8**	580	102.9	10000.0
i5	133	66	181	69.8	**67**	70.8	92	**53.4**	6231.6
majority	5	1	7	26.9	**3**	**15.9**	**3**	17.4	2.2
rot	135	107	458	124.8	**325**	114.3	455	**92.6**	10000.0
term1	34	10	99	70.9	97	61.5	**78**	**46.3**	8235.5
x2	10	7	—	—	**24**	32.5	**24**	**25.9**	13.6
x3	135	99	438	100.2	**421**	**62.3**	541	69.8	10000.0
z4ml	7	4	21	42.9	10	37.4	**9**	**24.1**	8.8
sum			2757	1235.5	**2503**	995.4	2942	**888.5**	

[5] R. Chaudhry, T.-H. Liu, A. Aziz, and J.L. Burns. Area-oriented synthesis for pass-transistor logic. In *Int'l Conf. on Comp. Design*, pages 160–167, 1998.

[6] C.-H. Chen and C.-Y. Tsui. Timing optimization of logic network using gate duplication. In *ASP Design Automation Conf.*, pages 233–234, 1999.

[7] K.C. Chen and S. Muroga. Timing optimization for multi-level combinational networks. In *Design Automation Conf.*, pages 339–344, 1990.

[8] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., 1994.

[9] R. Drechsler and B. Becker. *Binary Decision Diagrams – Theory and Implementation.* Kluwer Academic Publishers, 1998.

[10] R. Drechsler, N. Drechsler, and W. Günther. Fast exact minimization of BDDs. *IEEE Trans. on CAD*, 19(3):384–389, 2000.

[11] L.A. Entrena and K.-T. Cheng. Combinational and sequential logic optimization by redundancy addition and removal. *IEEE Trans. on CAD*, 14(7):909–916, 1995.

[12] J. Espejo, L. Entrena, E. San Millan, and E. Olias. Logic restructuring for MUX-based FPGAs. In *EUROMICRO*, pages 161–168, 1999.

[13] W. Günther and R. Drechsler. ACTion: Combining technology mapping and logic synthesis for MUX based FPGAs. In *EUROMICRO*, pages 130–137, 2000.

[14] G. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publisher, 1996.

[15] Y. Hong, P.A. Beerel, J.R. Burch, and K.L. McMillan. Safe BDD minimization using don't cares. In *Design Automation Conf.*, pages 208–213, 1997.

[16] K. Karplus. Amap: a technology mapper for selector-based field-programmable gate arrays. In *Design Automation Conf.*, pages 244–247, 1991.

[17] R. Murgai, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. An improved synthesis algorithm for multiplexor-based PGA's. In *Design Automation Conf.*, pages 380–386, 1992.

[18] R. Murgai, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. *Logic Synthesis for Field-Programmable Gate Arrays.* Kluwer Academic Publisher, 1995.

[19] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Design Automation Conf.*, pages 620–625, 1990.

[20] B. Rohfleisch, B. Wurth, and K. Antreich. Logic clause analysis for delay optimization. In *Design Automation Conf.*, pages 668–672, 1995.

[21] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

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

[23] E. Sentovich, *et al.* SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.

[24] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0.* University of Colorado at Boulder, 1998.