A New Technology Mapping Algorithm for the Design and Evaluation of Fuse/Antifuse-based Field-Programmable Gate Arrays

Alessandro Bedarida, Silvia Ercolani and Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

We describe in this paper a new approach for technology mapping of fuse/antifuse-based field-programmable gate arrays (FPGAs). These are arrays of uncommitted modules, where the personalization and wiring is achieved by fuse/antifuse technology and can be modeled by stuck-at and/or bridging inputs. The method extends a previous approach presented in [9].

The contribution of this paper is twofold. First we show a new efficient way of determining a matching that requires input bridging. Second, we compare the ease of mapping of standard logic benchmark circuits into different FPGAs and the relative importance of the personalization by means of stuck-at and bridging. Experimental results are reported and critically compared.

1 Introduction

There has been an increasing interest in digital-system prototyping using *Field-Programmable Gate Arrays* (FPGAs) due to their fast turn-around time and low non-recurrent engineering costs. One class of FPGAs uses anti-fuse technology, where logic gates and their interconnections are programmed by shorting wire segments in prescribed locations. These FPGAs consists of an array of identical multiple-input/single-output logic modules. A module can be configured to implement a logic function by forcing any input to logic 0 or logic 1 or by bridging inputs. An example of circuits in this class are those manufactured by Actel Inc. [1].

System design with FPGAs requires specific logic design tools. In particular, *technology mapping* is crucial for achieving an efficient implementation. Technology mapping is the process of transforming a set of logic equations into an interconnection of parts that are instances of the elements in a given library. In the case of FPGAs, the "library" consists of the set of combinational logic gates that can be derived from the uncommitted module.

FPGAs were conceived before efficient logic design tools were available for this technology. As a result, FPGA modules were devised on the basis of their electrical and geometrical properties (*i.e.* area, timing, wirability). A thorough analysis of the ease of mapping logic circuits into FPGAs was not possible at first due to the lack of specialized tools.

Previous existing approaches to technology mapping include algorithms and tools that support an explicit arbitrary library definition, such as *MisII* [4], *Ceres* [5] and commercial products. These tools need a library of cells explicitly derived from the uncommitted module. Since the enumeration of the library cells may be long, a subset of the library may be used to increase the mapping speed at the expense of the quality of the mapping.

Mis-pga [2] and *Amap* [3] are specialized technology mappers for the FPGAs fabricated by Actel Inc. These programs exploit the particular structure of the uncommitted module, based on multiplexers, in the mapping process. Conversely, the approach presented here is general in nature and it applies to any kind of FPGA based on fuse or antifuse technology.

In general, existing logic design tools for FPGAs rely on a library description and assume a given programmable module. Therefore it is hard to use them to explore the impact of choosing different programmable modules, because the full enumerated library has to be derived for each module.

The authors presented in [9] a new approach to technology mapping for fuse/antifuse-based FPGAs (called Electrical PGAs) that:

- Does not require an explicit library enumeration, nor a manual process to tailor the algorithms on the particular module used in the FPGA.
- Supports generic FPGAs which can be personalized according to a stuck-at and bridging model. The uncommitted module is assumed to be an arbitrary single-output combinational function.
- Can be used as an exploratory tool to benchmark the mapping performance of different uncommitted modules.

In this approach the full library is replaced by the description of the uncommitted module only. The technology mapping algorithms check whether a given logic function can be implemented by programming the uncommitted module.

Indeed, the process of personalizing an uncommitted module can be modeled by tying some input to logic 0 (*input stuck-at-0*) or logic 1 (*input stuck-at-1*) or by connecting two inputs together (*input bridging*) on the uncommitted module. Therefore, the FPGA library can be represented by the set of equivalent gates that the personalization induces on the uncommitted module. As a result, any FPGA can be used as a target architecture by just describing the logic function of the uncommitted module, *i.e.* by a single Boolean expression.

The contribution of this paper is twofold. First we show an enchancement of the matching technique presented in [9] that achieves comparable results in a much shorter computing time. Second, we compare the ease of mapping of standard logic benchmark circuits and we consider the effects of using *input stuck-at 0/1* and input *bridging* separately, to contrast the relative merit of the two techniques. Indeed, the major contribution of this research has been the creation of an *exploratory tool* to evaluate fuse/antifuse programmable modules for FPGAs, to deepen the understanding between personalization features and quality of the final logic implementation.

2 Technology Mapping for FPGAs

Algorithms for technology mapping were pioneered by Keutzer [7], Rudell [8] and Detjens [10]. Similarly to their approach, we use a heuristic method based on three different tasks. *Partitioning* a network into a collection of multiple-input/single-output combinational sub-networks. *Decomposition* of each sub-network into two-input functions, to increase the network granularity. *Covering* each decomposed network by committed modules so that either area or delay is optimized.

Standard techniques are used for the first two tasks [7, 8, 10]. The covering algorithm is borrowed from Mailhot [5]. It uses the notion of *cluster* and *module* functions. The former represents a portion of the network to be covered, the latter the FPGA primitive. Details of the covering algorithms are presented in [9].

The major contribution of this paper is related to solving the *matching* problem for FPGAs. The algorithms used here, as well as those presented in [9], perform a run-time customization of the FPGA module: by comparing the cluster and the module functions, it determines which module inputs should be set to 0/1, which should be bridged together, and which input ordering, if any, makes the module implement the cluster function. If no match is found, the algorithm returns that no matching exists and another cluster function is tried.

We denote the module and cluster functions by $G(x_1, x_2, \ldots, x_n)$ and $F(y_1, y_2, \ldots, y_m)$ respectively. We call *stuck-at set* of the module the set of variables that are set to 0/1 and we denote it by: S. We call *bridge set* B_j $(j = 1, 2, \ldots, l)$ each set of variables that are bridged together and we denote by B the *class of the bridge sets* (or, briefly, the *bridge class*), *i.e.* $B = \{B_j\}$. We define G_S the function obtained from G by setting each variable $x_i \in S$ to 0/1. Similarly, we define G_{SB} to be the function obtained from G_S by bridging the inputs corresponding to variables $x_i \in B_j$, $\forall\ B_j$. We represent by R the set of independent input variables, i.e. those that are not stuck-at and one for each bridge set. We define the matching problem as follows:

Given a cluster function $F(y_1, \ldots, y_m)$, and the module function $G(x_1, \ldots, x_n)$ $m \leq n$ find a stuck-at set S, a bridge class B and an ordering $\Omega(R)$ such that F and G_{SB} are functionally equivalent.

In our previous approach, we considered first a simplified matching problem:

Given a cluster function $F(y_1, \ldots, y_m)$, $m \leq n$, and the module function $G(x_1, \ldots, x_n)$ find a stuck-at set S, and an ordering $\Omega(R)$ such that F and G_S are functionally equivalent.

We showed how this problem can be solved efficiently by means of *Global Binary Decision Diagrams* (GBDDs). Then we showed how to compute a solution to the full matching problem by solving repeatedly the simplified matching problem.

We present here an extended set of results of applying the algorithms presented in [9] to three different FPGA modules: $act0$, $act1$ and $act2$. Their logic circuit diagrams are shown in Fig. 1. While only $act1$ and $act2$ are commercially available, we consider also $act0$ since this cell is the fundamental block of which both $act1$ and $act2$ can be considered extensions.

Circuit	*act0*		*act1*		*act2*	
duke2	212	211	178	175	164	162
f51m	85	83	65	59	52	50
bw	89	87	67	63	64	61
clip	86	86	74	68	62	60
vg2	57	57	46	45	41	41
rd84	92	87	75	65	63	61
5xp1	66	65	54	50	48	46
C499	279	175	274	170	170	170
C1908	307	244	280	206	209	204
C5315	1048	932	912	796	729	723
misex1	30	30	25	24	23	23
misex2	52	52	46	42	40	39
apex6	426	417	411	383	295	288
apex7	137	135	122	114	108	104
des	1995	1945	1783	1673	1404	1384

Table 1: Number of cells needed to implement a set of benchmarks using FPGA $act0$, $act1$ and $act2$. For any of the modules, the first column reports the cost only for the stuck-at personalization and the second one the cost for the complete personalization.

Table 1 shows the results on several ISCAS and MCNC benchmarks. For each of the uncommitted modules, we report the number of cells needed to implement the circuit when only stuck-at inputs are used in the personalization

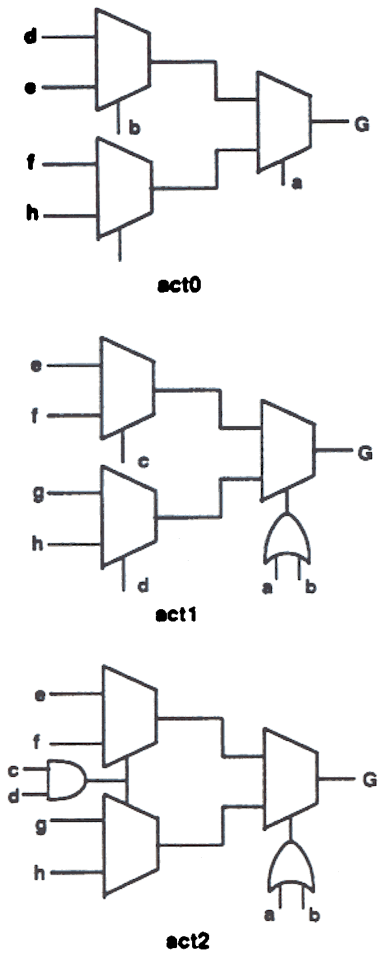

Figure 1: FPGA uncommitted modules

(simplified matching problem), as well as when stuck-at and bridged inputs are used (full matching problem). Run times are in the order of few seconds for the simple matching problem and several minutes for the full one.

Comparing the results was very interesting since we noticed that, in most cases, the full personalization of the cells by input bridging decreases the mapping cost by only a few cells. This observation led us to think that considering all the possible bridging between inputs is formally correct, but of little practical importance for the cells considered. It is also the case in practice [15] that bridging is rarely used, usually between only two inputs. Moreover, the average execution time of the algorithm for the full matching problem is much higher than the one for the simplified problem.

For the reasons above, we introduced a simplified bridging procedure that bridges only any two inputs of the module cell. This procedure turned out to be very effective, confirming our hypothesis about the relevance of input bridging in the personalization process.

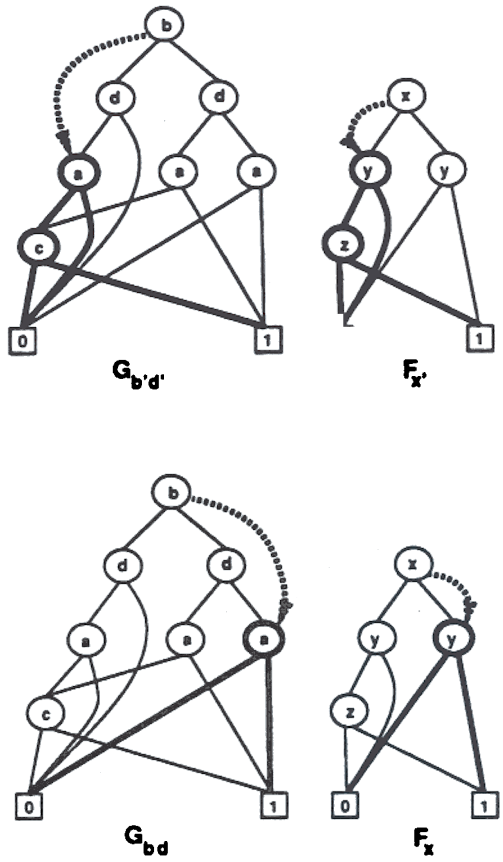

Figure 2: When bridging is allowed only for two input variables, the matching problem is equivalent to checking for a double sub-graph isomorphism.

3 A New Simplified Bridging Algorithm: Onebridge

The basic idea of the algorithm is to express a bridging between two input variables as a pair of double stuck-at's. Suppose that the variables to be bridged are b and d. The bridge is represented by the equation $b = d$. This can also be expressed as $b'd' + bd$, or $((b = 0)$ AND $(d = 0))$ OR $((b = 1)$ AND $(d = 1))$.

Let us now refer to Fig. 2 and consider the module function $G = ab + a'd'c$ whose BDD is drawn for the input ordering (b, d, a, c) and the cluster function $F = xy + x'y'z$ whose BDD is drawn for (x, y, z).

A solution to the matching problem with one bridge between b and d for the given input ordering exists if and only if the equations $G_{b'd'} = F_{x'}$ and $G_{bd} = F_x$ are simultaneously verified. Therefore, this problem of one simple bridge (i.e. one bridge between two input variables of the module function) is reduced to verifying that the BDD of $G_{b'd'}$ is isomorphic to the BDD of $F_{x'}$ and that the BDD of G_{bd} is isomorphic to the BDD of F_x.

Since in this algorithm we associate the bridged variables of G with the first variable of F, we have to perform a cyclic permutation of the variables of F, so that each of them, in

turn, is in first position. The algorithm can be described as follows:

```
Matching_3( G, F )
{
    /* No bridging at first attempt */
    if (Matching_1( G, F))
        return(TRUE);
    m = |sup(F)|;
    for (each cyclic permutation of F inputs) {
        node_f = Root(BDD_F);
        for (each of the subgraphGBDD_G of height m + 1 ) {
            node_g = Root(subgraphGBDD_G);
            if (OneBridgeBdd(node_g, node_f)) {
                Ω = DetermineOmega( );
                S = DetermineStuckatSet( );
                return(TRUE);
            }
        }
    }
    return(FALSE);
}

OneBridgeBdd(node_g, node_f)
{
    /* Initialize pointers to cluster subBDDs */
    low_f = node_f->low;
    high_f = node_f->high;
    /* Initialize pointers to module subBDDs */
    low_g = node_g->low;
    if (levelof(low_g->low) == levelof(low_g) - 1)
        low_g = low_g->low;
    high_g = node_g->high;
    if (levelof(high_g->high) == levelof(high_g) - 1)
        high_g = high_g->high;
    return (DoubleIsomorphism(low_g, low_f, high_g, high_f))
}
```

The functions *Matching_1*, *DetermineOmega* and *DetermineStuckatSet* are described in [9]. The former solves the simplified matching problem. The other two determine the input permutation and the detailed *stuck-at* set.

In function *DoubleIsomorphism* shown below, the two pairs of BDDs *(node_x1, node_y1)* and *(node_x2, node_y2)* must be isomorphic under the same mapping of levels and nodes. The function *isomorph_body* is similar to *Isomorph* as defined in [9], the only difference being that the initialization is performed outside the function.

```
DoubleIsomorphism(node_x1, node_y1, node_x2, node_y2)
{
    Level_Table = InitializeLevelTable( );
    Node_Table = InitializeNodeTable( );
    return(isomorph_body(node_x1, node_y1) &&
        isomorph_body(node_x2, node_y2));
}
```

This algorithm has advantages over both the simplified matching and the full matching algorithms described in [9]. It delivers results very close (and in most cases identical) to those obtained with the latter, but requires a computational effort comparable to the former. The implementation cost for our set of benchmarks is shown in Table 2. It should be remarked that the reason why the algorithm is much faster than the full matching one is twofold: 1) fewer candidate solutions are considered and 2) the particularly simple case of one bridge of two variables allows a very efficient implementation.

Circuit	*act0*	*act1*	*act2*
duke2	211	175	164
f51m	83	59	52
bw	87	63	64
clip	86	68	62
vg2	57	45	41
rd84	87	65	63
5xp1	65	50	48
C499	175	170	170
C1908	244	207	206
C5315	932	796	724
misex1	30	24	23
misex2	52	42	40
apex6	417	383	292
apex7	135	114	107
des	1945	1673	1404

Table 2: Implementation cost for FPGA *act0*, *act1* and *act2* when *onebridge* is applied.

This algorithm can be extended to the more general case of one bridge of many variables. Indeed, we tried this approach and we obtained the same results of the full matching algorithm. Even though the execution time became larger than for the *onebridge* algorithm, it was still much smaller than the time required by the *fullbridge*.

4 Program Implementation

The algorithms presented here have been incorporated in *Ceres* [5] to form an option called *Proserpine*. The program reads the logic description of the module and creates the global BDD data structure. The partitioning, decomposition and covering tasks are those of *Ceres*, while the matching algorithm is based on the BDD sub-isomorphism described in this paper. Three options can be chosen: *zerobridge* corresponds to allowing only input stuck-at's as a means of module personalization, *onebridge* allows bridging between any two inputs and *fullbridge* solves the full matching problem. *Proserpine* has been implemented in C and has been tested on the MCNC and ISCAS benchmarks. The results are summarized in Tables 1 and 2. Run times are in the order of a few seconds (on a DECstation 5000) for the *zerobridge* and *onebridge* options. They are significantly higher in the case of *fullbridge*.

So far, we considered FPGAs where each module is a multiple-input/single-output combinational logic gate [1]. Extensions to FPGAs with a multiple-output module and latching capability, as very recently proposed [16], are possible. They are straightforward when no functional sharing is done inside each module.

5 Evaluation of FPGA Module Functions

In this section, we want to show that *Proserpine* can be a useful tool in researching module primitives that support efficient implementations. Uncommitted modules have been chosen by looking at the circuit configurations and performance. No analysis to study and evaluate the ease of mapping logic circuits onto different FPGA architectures during their design phase has been done. By means of *Proserpine*, some very important issues in FPGAs design can be addressed.

Comparing the mapping behavior of *act*1 and *act*2 was quite interesting. Even though these two cells have similar area/delay characteristics, their performances turned out to be quite different. The module *act*2 outperforms *act*1 in all the cases that we tried. In Table 3, we compare the geometric mean of the results of Table 1 and Table 2 related to *act*1 and *act*2.

	zerobridge	*onebridge*	*fullbridge*
*act*1	133.6	118.9	118.8
*act*2	110.5	110.2	108.0

Table 3: Geometric mean of the implementation costs of *act*1 and *act*2 when the three different options of *Proserpine* are applied.

An interesting result was the different mapping behavior of these cells when using *zerobridge* and *fullbridge*. The two cells have a different functionality with respect to the stuck-at personalization. In the case of *act*2, the geometric means that we obtained for the two options have almost identical values (110.5-108.0). For the other cell, instead, there is a fairly large gap (133.6-118.8). Therefore *act*2 requires fewer bridges than *act*1.

Limiting the number of bridges in an FPGA is important for other remarks. There are two technical problems related to bridging. A first issue is the added cost required by customizing a cell by input bridging. In the architecture we considered, antifuse elements are located in such a way that setting any of the inputs to ground/power requires simply blowing an antifuse. A bridge between two inputs can be achieved in two ways. If an antifuse connects the inputs we want to bridge, the cost is equivalent to that of a stuck-at. However, some input pairs are not connected by antifuse. In this case, the bridge requires an added wiring cost.

Also, in the presence of bridges, load problems might occur. Suppose a net feeds several cells that need multiple bridges. In this case the total load on the fanout stem is affected by the input bridging within the cells with possible consequences on the driving capability of the net. Therefore, an ideal cell for FPGAs is one requiring the smallest number of bridges.

It was rather interesting to notice that little variations to a FPGA cell can affect its performance to mapping. Here we applied *Proserpine* to two new uncommitted module cells, whose logic diagram is shown in Fig. 3. These are simple variations of *act*1 and *act*2, swapping OR and AND gates. The results are given in Table 4.

By comparing the results in Tables 1, 2 and 4 we remark that *test*1 performs much worse than *act*1, while *test*2 is comparable to *act*2. This is an example of how *Proserpine* can be used by a designer of FPGAs to explore the impact of different module cells.

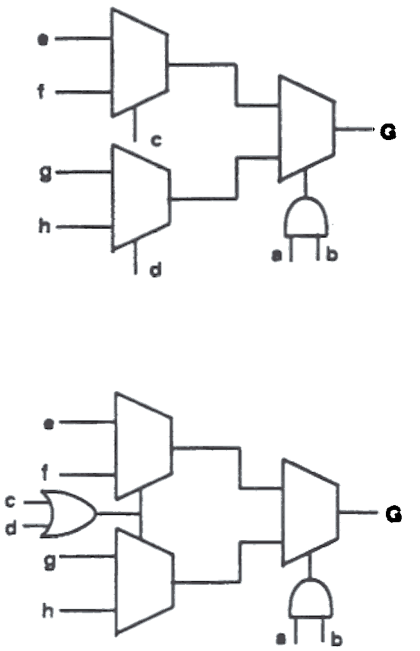

Figure 3: New FPGA uncommitted modules

Circuit	*test1*		*test2*	
duke2	185	178	164	164
f51m	73	62	52	52
bw	83	79	64	64
clip	76	72	62	62
vg2	50	46	41	41
rd84	87	74	63	63
5xp1	61	56	48	48
C499	279	173	170	170
C1908	296	220	209	206
C5315	931	802	729	724
misex1	28	26	23	23
misex2	48	45	40	40
apex6	329	305	295	292
apex7	123	114	108	107
des	1568	1455	1404	1404

Table 4: Number of cells needed to implement the benchmarks using FPGA *test*1, *test*2, when *zerobridge* and *onebridge* are applied.

6 Conclusions

We have extended a set of techniques for technology mapping of field-programmable gate arrays [9], that personalize an uncommitted module to perform a desired logic function, when possible, by determining the set of input variables that need to be stuck at 0/1 or bridged together. This matching algorithm allows a mapping program to capture the entire family of functions that can be implemented by a module by describing only one logic function, thus avoiding the enumeration of the entire library. In addition, the algorithm is not specific to a type of module, but can be applied to any module that can be represented by a single-output combinational logic function.

We have introduced here a novel algorithm, called *one-bridge*, that is a heuristics to solve a simplified version of the full matching problem. The algorithm has fast execution times and it performs nearly as well as a previous one that solves exactly the full matching problem, but that is much slower.

The matching algorithm has been implemented as part of a technology mapping program called *Proserpine*. We have tested the program on a set of benchmarks and have concluded that it compares favorably to other approaches. An outcome of this research has been the development of a tool to evaluate the effectiveness of different modules, for use in future FPGAs.

7 Acknowledgments

The authors would like to thank Frederic Mailhot for several stimulating discussions. This research was sponsored by an AEI/IEEE scholarship and by DEC and AT&T, jointly with NSF, under a PYI award. We acknowledge also support from ARPA, under contract No. J-FBI-89-101.

References

[1] A. El Gamal, J. Greene, J. Reynery, E. Rogoyski, Kaled A. El Ayat, and Amr Mohsen, "Architecture for Electrically Configurable Gate Arrays", *IEEE Journal of Solid-State Circuits*, April 1989, pp. 394-398

[2] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays", *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990, pp. 620-625

[3] K. Karplus, "Amap: a Technology Mapper for Selector-based Field-Programmable Gate Arrays", *28th ACM/IEEE Design Automation Conference*, San Francisco, June 1991, pp. 244-247

[4] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD*, November 1987, pp. 1062-1081

[5] F. Mailhot and G.De Micheli, " Technology Mapping with Boolean Matching ", *European Design Automation Conference*, Glasgow, Scotland, March 1990, pp. 212-216

[6] J. Rose, R.Francis, D.Lewis and P.Chow, "Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency", *IEEE Journal of Solid State Circuits*, Vol. 25, No.5, Oct. 1990, pp. 1217-1225

[7] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching", in *24th ACM/IEEE Design Automation Conference*, 1987, pp. 341-347

[8] R. Rudell, *Logic Synthesis for VLSI Design*, PhD thesis, U. C. Berkeley, April 1989, and Memorandum UCB/ERL M89/49

[9] S. Ercolani and G. De Micheli, " Technology Mapping for Electrically Programmable Gate Arrays", *Design Automation Conference*, San Francisco, June 1991, pp. 234-239.

[10] E. Detjens, G. Gannot, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. "Technology mapping in MIS", *International Conference on Computer-Aided Design*, November 1987, pp. 116-119

[11] R.J. Francis, J. Rose, K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays", *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990, pp. 413-419

[12] S.B. Akers, "Binary Decision Diagrams", *IEEE Transactions on Computer*, June 1978, pp. 509-516

[13] R. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computer*, August 1986, pp. 677-691

[14] K.Brace, R. Rudell and R.Bryant, "Efficient Implementation of a BDD package", *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990, pp. 40-45

[15] A. El Gamal, *private communication*, April 1991

[16] S. Baker, " Quicklogic unveils FPGA", *EE Times*, issue 639, April 29, 1991.