From And-Inverter Graphs to Majority-Inverter Graphs

Felipe L. Machado, Vinicius N. Possani[1],
Augusto S. Neutzling[1]
[1]PPGC, Institute of Informatics
UFRGS, Porto Alegre, Brazil
{flmachado, vnpossani, ansilva}@inf.ufrgs.br

Renato P. Ribas[1, 2], André I. Reis[1, 2]
[1]PPGC / [2]PGMicro, Institute of Informatics
UFRGS, Porto Alegre, Brazil
{rpribas, andreis}@inf.ufrgs.br

***Abstract*— In VLSI design, the adopted data structure to perform logic synthesis has a direct impact in the software scalability and also in the quality of the final chip, i.e., area, power and delay. In the last decades different data structures were explored during the logic synthesis phase, the most commons are Directed-Acyclic Graphs (DAG), Binary-Decision Diagram (BDD) and And-Inverter Graph (AIG). The most recent approach presented in the literature was the Majority-Inverter Graph (MIG), which is a potential data structure to the next generation of logic synthesis CAD tools. In this sense, this work presents an automated method to convert a circuit from an AIG format to a MIG format. The trivial conversion performs a one-to-one conversion, i.e., each AIG node is directly translated do a MIG node. However, the proposed approach aims to find out majority functions in an AIG in order to represent them into a single majority node. Such a logic coupling, leads to a more compact MIG representation. Our experimental results demonstrate that the proposed method was able to achieve up to 53.89% of node count reduction compared to the trivial conversion, with an average node reduction of 25.77%.**

Keywords—VLSI degisn; logic synthesis; data structures; graph; and inverter graph; majority inverter graph.

I. Introduction

The microelectronics evolution is supported by the transistor scaling and enhanced CAD tools acting into the Electronic Design Automation (EDA) field. The Very Large Scale Integration (VLSI) design flow comprises many different phases, starting from a Register Transfer Level (RTL) description of the target design until achieve the final circuit layout. In this context, the microelectronics CAD tools must implement appropriated data structures to represent the circuit design at each phase of the project. The adopted data structure has a direct impact in the software scalability and also in the quality of the final chip, i.e., area, power and delay. In this work we are focusing in some data structures applied during the logic synthesis phase of the VLSI flow.

Some well-known data structures adopted by academy researchers and by the microelectronics industry to represent logic networks are: Sum of Products (SOP) [4], Directed-Acyclic graphs (DAG) [5], Binary-Decision Diagram (BDD) [6] and And-Inverter Graph (AIG) [7-8]. In the last decades, logic synthesis CAD tools migrated from Boolean networks described as DAGs, where nodes represent logic gates of any arbitrary logic function and edges represent interconnection, to the AIG structure. The AIG is a homogeneous Directed Acyclic Graph (DAG), where each vertex represents a 2-input *AND* function and inversions are represented with complemented attributes on the edges.

The most recent logic synthesis data structure presented in the literature is called Majority-Inverter Graph (MIG) [1-3]. Similarly to the AIG, the MIG structure is a DAG where each vertex represents a 3-input *MAJ* (majority function) and inversions are represented in complemented edges. In the paper [1], Amarú *et al.* present the formal definition of the MIG structure together with a majority-based Boolean algebra to perform logic optimization on top the MIG. The author present a set of experiments demonstrating advances in different criteria such as area, power and delay, for both the Application-Specific Integrated Circuit (ASIC) and the Field-Programmable Gate Array (FPGA) design flows.

Since new approaches, e.g., new data structures [1-3], are potential candidates to be adopted by the industry and academy, it is needed a gradual migration and adaptation of the CAD tools, as well as, of the well-known benchmark sets. In this context, the conversion from an AIG structure to a MIG representation is one of the first steps of the migration process. There is a trivial way to perform such conversion, where each AIG node (2-input *AND* node) is directly converted to a MIG node (3-input *MAJ* node) by set a constant value in one of the three inputs of the *MAJ* node, i.e., $MAJ(x, y, z) = AND(x, y)$ when $z = 0$ and $MAJ(x, y, z) = OR(x, y)$ when $z = 1$. By doing that, we have a one-to-one conversion, where the obtained MIG has the same number of nodes of the AIG representation. However, it possible to explore the logic representativity of the 3-input *MAJ* nodes from MIG by finding out majority functions in the AIG and merge each one in a single *MAJ* node of the MIG structure.

In this sense, this work presents a method to translate a circuit from an AIG structure to a MIG structure. The main idea behind the method is to traverse the AIG in order to find patterns that match with two different majority function representations. The proposed approach delivers a more compact MIG than the solution obtained by only applying the one-to-one conversion. Such compactness can provide a good start point for logic optimizations on top of the MIG structure by applying the *MIGhty* synthesis tool presented in [1]. Our experiments demonstrated up to 53.89% of reduction in the node count when converting circuits from AIGs to MIGs, with an average reduction of 25.77%.

The remainder of this paper is organized as follows. Section II present some preliminary concepts and definitions. Section III presents the proposed method. Our experimental results are presented and discussed in the Section IV.

II. Background

A. *AND Inverter Graph (AIG)*

And-Inverter-Graph (AIG), is a specific type of Directed Acyclic Graph (DAG), where each node has either 0 incoming edges – *primary inputs* (PI) – or 2 incoming edges – *AND* nodes, as show in Fig. 1(a). Each edge can be complemented or not. Some nodes are marked as *primary outputs* (PO) [7-8].

B. *Three Input Majority Function (MAJ3)*

The majority function is a three input function which expression is:

$$MAJ(x, y, z) = x \,.\, y + x \,.\, z + y \,.\, z \tag{1}$$

The majority function is symmetrical (i.e. any input permutation gives the same function):

$$\begin{aligned} MAJ(x, y, z) &= MAJ(x, z, y) = MAJ(y, x, z) \\ &= MAJ(y, z, x) = MAJ(z, x, y) = MAJ(z, y, x) \end{aligned} \tag{2}$$

Typically, Boolean functions are represented as expressions using *AND*, *OR* and *INV* as operations. Any expression of such form is trivially transformed into an equivalent majority expression (i.e. using the majority function as basic element) using the following relations:

$$AND(x, y) = MAJ(x, y, 0) \tag{3}$$

$$OR(x, y) = MAJ(x, y, 1) \tag{4}$$

C. *Majority Inverter Graph (MIG)*

MIG is also a specific type of Directed Acyclic Graph (DAG), representing a homogeneous logic network with an indegree equal to 3 and each node representing the majority function, as shown in Fig. 1(b). In a MIG, edges are marked by a regular or complemented attribute [1-3].

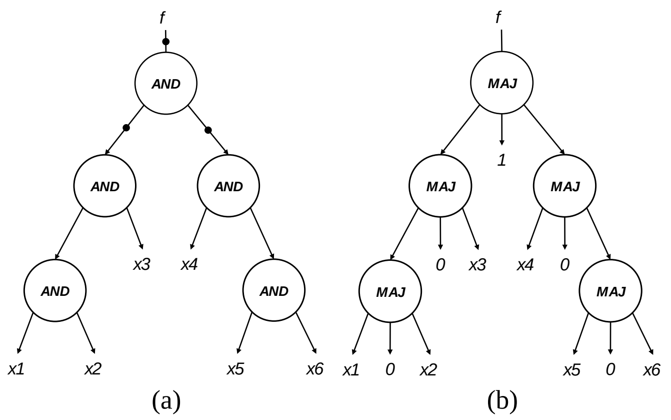


Fig. 1. In (a) AIG and in (b) MIG representation of the function *f = (x1 . x2 . x3) + (x4 . x5 . x6).*

D. *Classes of Boolean Functions (NPN)*

By considering a set of all functions with up to *n* variables, these functions can be grouped in *classes*, as illustrated in Fig. 2 [11]. Boolean functions can be grouped taking into account the negation (*x*), and/or the permutation of variables (*y*), and/or the negation of function value (*z*). For instance, NPN-class corresponds to the set of distinct Boolean functions obtained by permuting and complementing the input variables and complementing the output [9].

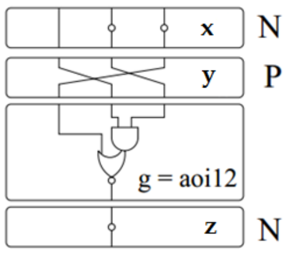


Fig. 2. Types of Boolean equivalence used to group functions in classes, image adapted form [11].

III. Proposed Method

The proposed method starts from an AIG structure and aims to provide as output a MIG representation of the circuit. As mentioned before, the proposed approach aims to avoid the trivial one-to-one conversion from the entire AIG nodes to MIG nodes. In this sense, in a first step the proposed method traverse the AIG finding for majority function patterns and convert each matched portion of the AIG to a single *MAJ* node. In a second step, the method coverts all the remainder *AND* nodes to *MAJ* nodes by considering the property presented in (3). In the following subsection, we define the two representative majority function patterns adopted in our approach and the how the algorithm searches for such patterns on top of the AIG.

A. *3-input MAJ patterns on AIG structures*

The 3-input majority function can be implemented according to the ISOP form described in equation (1). Thus, in order to represent equation (1) in an AIG structure we firstly convert the *OR* (+) operators into *AND* (.) operators by applying the De Morgan's Laws, resulting in the following equation:

$$MAJ(x ,y, z) = !((!(x \,.\, y) \,.\, !(x \,.\, z)) \,.\, !(y \,.\, z)) \tag{6}$$

Another alternative way to represent a 3-input majority function is through the factored form of (1):

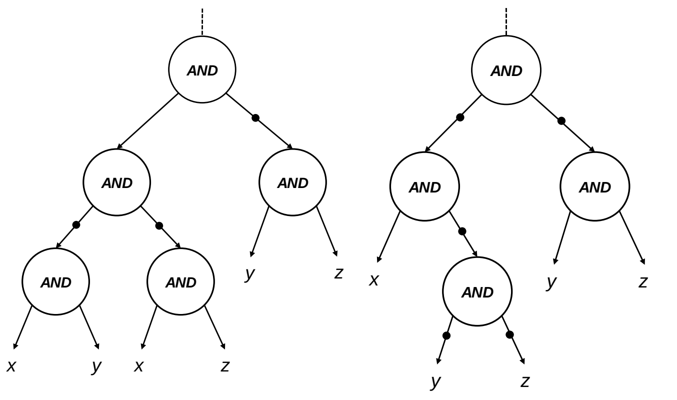
$$MAJ(x, y, z) = x \,.\, (y + z) + y \,.\, z \tag{7}$$

Analogously, by applying the De Morgan's Laws over (7) it is possible to achieve the following representation expressed only with 2-input *AND* and inversions:

$$MAJ(x, y, z) = !(!(x \,.\, !(!y \,.\, !z)) \,.\, !(y \,.\, z)) \tag{8}$$

Figure 3 presents the AIG representations for the majority function described in equations (6) and (8). In this work, we called these two generic patterns present in Fig. 3(a) and

Fig. 3(b) as *ISOP pattern* (*MAJ_{ISOP}*) and *factored form patter* (*MAJ_{fac}*), respectively. It is important to mention that in both patterns the inputs signal *x*, *y* and *z* can appear in any possible permutation and/or in positive or complementary polarity. Besides, the output of the *MAJ* can also appears in direct or complementary polarity. In other words, since one of the structural patterns presented in Fig. 3 is found on the AIG, the algorithm check the NPN-equivalence of the inputs and output to ensure an accurate matching.



(a) (b)

Fig. 3. Two representative patterns for the 3-input *MAJ*: in (a) and AIG obtained form (6) (*MAJ_{ISOP}*) and in (b), AIG obtained from (8) (*MAJ_{fac}*).

B. Finding for 3-input MAJ patterns on AIGs

The Algorithm 1 presents the proposed approach to find out the majority function patterns, presented in Fig. 3, on top the AIG. The algorithm starts from the primary outputs PO of the AIG and for each PO the algorithm traverse the graph recursively until reach the PIs. As the recursion returns the algorithm tries to match portion of the graph with the *MAJ_{ISOP}* and *MAJ_{fac}* patterns. When there is a successful matching, the *makeMAJ* procedure replaces the set of nodes *S* that represent the majority function from the AIG by a single representative 3-input *MAJ* node.

Algorithm 1 Pseudocode of the pattern matcher routine.

```
1:  patternMatcher( currentNode )
2:    if currentNode == PI then  // PI is the primary input set
3:      return
4:    endif
5:    for each node n in currentNode.inNodes do
6:      patternMatcher( n )
7:    endfor
8:    set < node > S ← ∅
9:    matching ← checkISOPPattern( currentNode, S )  // $MAJ_{ISOP}$
10:   if matching == false then
11:     S ← ∅
12:     matching ← checkFactoredPattern( currentNode, S )  // $MAJ_{fac}$
13:   endif
14:   if matching == true then
15:       makeMAJ( S )
16:   endif
17: end
```

Basically, the procedures *checkISOPPattern* and *checkFactoredPattern* consider the *currentNode* as the root node of the majority function, i.e., the root is the *AND* node at the top of the patterns shown in Fig. 3. Thus, the algorithm looks two levels ahead in order to check if the incoming nodes respect the desired patterns. If there is a successful matching the involved nodes are registered into the set *S* in order to perform the AIG rewriting. Otherwise, no changes are done on the graph and the algorithm returns one level back into the recursion. The algorithm tries the matching again, until return to the POs. The last step of the algorithm is convert the remainder *AND*(*x*, *y*) nodes into *MAJ*(*x*, *y*, 0) nodes by set one input of the majority node to zero. This last step is trivial so it is not explicitly described in pseudocode.

IV. Experimental Results

In order to evaluate the designed method, we perform a set of experiments over the IWLS'05 Open Cores benchmark circuits derived from the experiments performed in [1], which are available to download in [10]. The motivation to use this benchmark in our experiments is that such circuits were derived from the MAJ-based synthesis tool *MIGhty* [1]. Thus, it is expected that a large number of MAJ nodes can found in the AIG description of such circuits. As the designed method starts from and AIG structure, the first step was translate the circuits available in [10] from Verilog to the AIG format. In the sequence, the proposed approach was applied on top the AIG in order to find out the sets of nodes that can be coupled into a single MAJ node.

Our experimental results are summarized in Table I. From the left to the right, the columns of the table present the following information: benchmark circuit name; the number nodes in the input AIG |AIG|; the number of majority functions found in the factored form pattern shown in Fig. 3(b) $|MAJ_{fac}|$; the number of MAJ node found in the ISOP form pattern shown in Fig. 3(a) $|MAJ_{ISOP}|$; the total number of MAJ nodes |MAJ|, i.e., the sum of the two previous columns; the number of AIG nodes that are covered by the found MAJ nodes $|MAJ_{cov}|$; the resultant node count into the final MIG |MIG| and finally, the perceptual reduction in the number of nodes.

As can be seen in the Table I, the proposed method was able to find out a significant number of MAJ nodes in the input AIGs. In the most part of the cases, the MAJ nodes were found in the ISOP form. This result is strongly related (dependent) to the structure of the input AIG. The $|MAJ_{cov}|$ is defined as follows:

$$|MAJ cov| = |MAJ_{fac}| \,.\, 4 + |MAJ_{ISOP}| \,.\, 5,$$

where 4 and 5 are the number of AIG nodes into the factored and ISOP patterns. This way, the node count into the final MIG is defined as follows:

$$|MIG| = |AIG| - |MAJ_{cov}| + |MAJ|,$$

where the set of AIG nodes covered by a given majority node are replaced by a representative MAJ node.

For the evaluated circuits, the proposed approach was able to find up to 53.89% of reduction in the node count when converting circuits from AIGs to MIGs. On average, we achieve 25.77% of node reduction. In is important to mention that the proposed approach can be used to convert any circuit described in an AIG format to a MIG description. Thus, the obtained reductions can provide a good start point to apply a

MIG-based synthesis with the optimizations described in the *MIGhty* [1] and other future synthesis method based on majority functions.

V. Conclusions and Future Works

This paper presented and automated method to find out majority function patterns on top of AIG circuit representation, allowing a compact conversion from AIG to the MIG representation. A set of experiments was carried out the representative benchmark of circuits. The proposed approach was able to achieve significative reduction rate in the node count when compared to a trivial one-to-one conversion of nodes from the AIG to the MIG. As future works we intend to investigate alternative optimizations on MIGs, based on the MIG Boolean algebra defined in [1].

Acknowledgment

This work was supported in part by Brazilian founding agencies CNPq, CAPES and FAPERGS.

References

[1] L. Amarú, P. E. Gaillardon and G. De Micheli. "Majority-Inverter Graph: A New Paradigm for Logic Optimization," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806-819, May 2016.

[2] L. Amarú, P. E. Gaillardon and G. De Micheli. “Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization,” Proc. DAC’14.

[3] L. Amarú, P. E. Gaillardon and G. De Micheli. “Boolean Logic Optimizationin Majority-Inverter Graph,” Proc. DAC’15.

[4] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill, New York, 1994.

[5] R. K. Brayton, et al. “MIS: A Multiple-Level Logic Optimization System,” in *IEEE Trans. CAD*, 6(6): 1062-1081, 1987.

[6] R. E. Bryant. “Graph-based algorithms for Boolean function manipulation,” *IEEE TCOMP*, C-35(8): 677-691, 1986.

[7] ABC synthesis tool - available online at http://www.eecs.berkeley.edu/~alanmi/abc/.

[8] R. K. Brayton, A. Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool,” Proc. CAV, 2010.

[9] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," Proc. ICFPT'13.

[10] http://lsi.epfl.ch/MI

[11] U. Hinsberger and R. Kolla. “Boolean matching for large libraries,” in *Proc. Of Design Automation Conference (DAC)*, pp. 206-211, 1998.

TABLE I. SUMMARY OF RESULTS OBTAINED BY THE PROPOSED APPROACH WHEN PROCESSING THE BENCHMARK CIRCUITS FROM [10].

Benchmark	**|AIG|**	**|MAJ|**	**|MAJ$_{fac}$|**	**|MAJ$_{ISOP}$|**	**|MAJ$_{cov}$|**	**|MIG|**	**% Node Reduction**
vMIG.ac97_ctrl	13625	720	0	720	3600	10745	21.14
vMIG.aes_core	28991	2011	0	2011	10055	20947	27.75
vMIG.comp	26081	2096	199	1897	10281	17697	32.15
vMIG.des_area	5822	409	0	409	2045	4186	28.10
vMIG.des_perf	77454	2565	0	2565	12825	67194	13.25
vMIG.diffeq1	22857	1300	17	1283	6483	17657	22.75
vMIG.div16	9486	1278	0	1278	6390	4374	53.89
vMIG.DSP	50036	2462	6	2456	12304	40188	19.68
vMIG.ethernet	64251	1574	1	1573	7869	57955	9.80
vMIG.hamming	2871	200	0	200	1000	2071	27.86
vMIG.i2c	1163	48	0	48	240	971	16.51
vMIG.log2	38942	1916	12	1904	9568	31278	19.68
vMIG.MAC32	11654	582	0	582	2910	9326	19.98
vMIG.max	7790	895	0	895	4475	4210	45.96
vMIG.mem_ctrl	8407	316	0	316	1580	7143	15.04
vMIG.MUL32	12764	917	0	917	4585	9096	28.74
vMIG.mult64	30336	1141	0	1141	5705	25772	15.04
vMIG.pci_bridge32	23187	1146	0	1146	5730	18603	19.77
vMIG.pci_spoci_ctrl	1660	182	0	182	910	932	43.86
vMIG.revx	11040	884	3	881	4417	7504	32.03
vMIG.sasc	777	39	0	39	195	621	20.08
vMIG.simple_spi	1165	82	0	82	410	837	28.15
vMIG.spi	4197	215	0	215	1075	3337	20.49
vMIG.sqrt32	4616	615	0	615	3075	2156	53.29
vMIG.square	22007	1030	0	1030	5150	17887	18.72
vMIG.ss_pcm	597	50	0	50	250	397	33.50
vMIG.systemcaes	11215	417	0	417	2085	9547	14.87
vMIG.systemcdes	3513	265	0	265	1325	2453	30.17
vMIG.tv80	9597	550	0	550	2750	7397	22.92
vMIG.usb_funct	15827	708	0	708	3540	12995	17.89
vMIG.usb_phy	496	32	1	31	159	368	25.81
Average	**16852.39**	**859.52**	**7.71**	**851.81**	**4289.87**	**13414.32**	**25.77**
Stand. Dev.	**18907.41**	**740.07**	**35.70**	**728.42**	**3688.10**	**16405.05**	**11.06**