XbarGen: A memristor based boolean logic synthesis tool

Marcello Traiola, Mario Barbareschi, Antonino Mazzeo, Alberto Bosio

XbarGen: a Memristor Based Boolean Logic Synthesis tool

Marcello Traiola[1], Mario Barbareschi[1], Antonino Mazzeo[1], Alberto Bosio[2]

[1]DIETI, University of Naples Federico II
Naples, Italy
m.traiola@studenti.unina.it
{mario.barbareschi, mazzeo}@unina.it

[2]LIRMM, Université Montpellier
Montpellier, France
bosio@lirmm.fr

***Abstract*—The shrinking process of CMOS technology is reaching its physical limits, thus impacting on several aspects, such as *performances*, *power consumption* and many others. Alternative solutions are under investigation in order to overcome CMOS limitations. Among them, the memristor is one of the promising technologies. Several works have been proposed so far, describing how to implement boolean logic functions employing memristors in a crossbar architecture. In this paper, we propose a tool able to automatically map any boolean function to a memristor based crossbar implementation. The proposed tool helps to perform a design space exploration to identify the best implementation w.r.t. performances and area overhead.**

Index terms—Memristor crossbar, Design Space Exploration, Boolean Functions. Circuit Synthesis

I. Introduction

Today's computing devices are based on the CMOS technology, that is the subject of the famous Moore's Law [1] predicting that the number of transistors in an integrated circuit will be doubled every two years. Despite the advantages of the technology shrinking, we are facing the physical limits of CMOS. Among the multiple challenges arising from technology nodes lower than 20 nm, we can highlight the high leakage current (i.e., high static power consumption), reduced performance gain, reduced reliability, complex manufacturing process leading to low yield and complex testing process, and extremely costly masks [2], [3], [4], [5].

Additionally, the expected never-ending increasing of performances is indeed no longer true. Looking in more detail, the classical computer architectures, either *von Neumann* or *Harvard,* divide the computational element (i.e., CPU) from the storage element (i.e., memory). Therefore, data have to be transferred inside the computational element in order to be processed and then transferred back to be stored. The main problem of this paradigm is the bottleneck due to the data transfer time limited by the bandwidth. For example, transferring one TeraByte at the rate of 1Gbit/second requires more than two hours.

Many new technologies are under investigation, among them the *memristor* is a promising one [9]. The memristor is a non-volatile device able to act as both storage and information processing unit that presents many advantages: CMOS process compatibility, lower cost, zero standby power, nanosecond switching speed, great scalability, high density and non-volatile capability [10], [11]. Thanks to its nature (i.e., computational as well as storage element), the memristor is exploited in different kind of applications, such as neuromorphic systems [12], non-volatile memories [13], computing architecture for data-intensive applications [6].

A fundamental component of any kind of computing architecture is the implementation of boolean logic functions. In [14], the authors proposed a methodology for the synthesis of boolean logic function on a memristor crossbar. Their work showed that is possible to implement any kind of boolean function on a memristor crossbar. However, the experimental results have been carried out on a couple of small circuits only due to the lack of an automated synthesis tool. In this paper, we aim to extend the work of [14] by presenting a tool able to automatically map any boolean function to a memristor based crossbar implementation. Moreover, we investigate the impact of different synthesis optimization parameters on the memristor crossbar to evaluate area and performances.

The remainder of the paper is structured as following. Section II presents the state of the art and provides the required background on the memristor based computation. Section III details the proposed memristor and crossbar model as well as the synthesis methodology, while the Section IV gives the experimental results. Finally, the Section V draws the conclusions.

II. Background and State of the Art

In this section we provide the basics about the memristor modeling, as well as the way how the memristor can be exploited to implement a given boolean function.

A. Memristor model

A memristor is a non-linear electrical component whose electrical resistance is not constant but depends on the history of the charge flowed through the device itself. Since we intend to implement a digital circuit, we refer to the memristor Voltage-Current relation depicted in Figure 1, detailed in [8],

as the best solution for modeling the memristor's behavior (i.e., thanks to the ideal response to a pulse-wave). Thus, as the Figure 1 shows, the voltage applied to the memriristor's terminals does not change its resistance until it crosses a threshold. In the adopted ideal model, the upper and the lower thresholds have the same absolute value.

We resort to the Snider Boolean Logic (SBL) [8] convention whereby a lower resistance (steeper curve denoted as R_{ON}) represents a logic '0' while an higher resistance (lower slope curve denoted as R_{OFF}) represents a logic '1'.

Two basic operations can be performed, defined as SET and RESET. The first one allows to program the memristor to R_{ON} and thus at logic '0', while the second one programs the memristor to R_{OFF} that corresponds to logic '1'. The Figure 1 depicts SET and RESET operations as described by Xie et al. in [14]

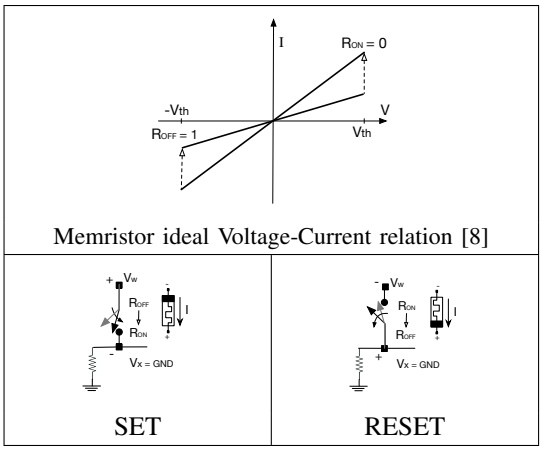


Figure 1. Set and Reset operations [14]

B. Fast Boolean Logic Circuits

Snider proposed in [8] a design methodology for memristor crossbar that aimed to implement boolean functions. This design was then improved by Xie et al. in [14]. Let us briefly recall their proposition referring to it as Fast Boolean Logic Circuit (***FBLC***). First, the logic circuit requires that the Boolean function is expressed in the SOP format:

$$f = M_1 + M_2 + ... + M_n = \overbrace{\overline{M_1} \cdot \underbrace{\overline{M_2} \cdot \cdots \cdot \overline{M_n}}_{\text{NAND}}}^{\text{AND}}\ \text{NOT}$$

Then, as Figure 2-a shows, FBLC is divided in blocks, useful to accomplish FSM's steps (fig 2-b) which are:

INA:	**in**itialize **a**ll the memristors to R_{OFF};
RI:	the input block **r**eceives the **i**nputs;
CFM	**con**figure all **m**interms simultaneously, in parallel;
EVM:	**ev**aluate all **m**interms simultaneously (NAND);
EVR:	**ev**aluate **r**esults: $\overline{f}$ is calculated (AND);
INR:	**in**vert **r**esults: $\overline{f}$ need to be inverted to achieve f
SO:	**s**end **o**utputs: the result captured in OL is sent out.

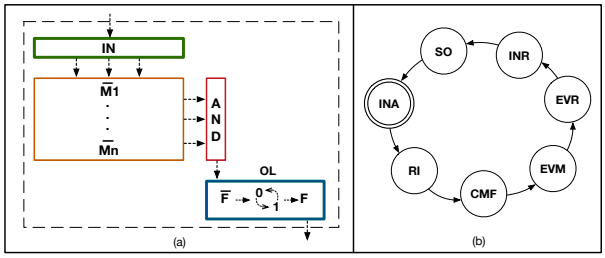


Figure 2. Fast Boolean Logic Circuit

Below, the description of the blocks:

- input block, where inputs are stored during the RI step;
- minterms block, where minterms are configured during CFM and evaluated during EVM;
- AND block, where results of EVM are stored and AND operation is performed during EVR;
- output block, where results of EVR are stored and inversion operation is performed during INR;

For the purpose of realizing each step of the FSM, the authors proposed some *primitive operations* that we summarize in Figure 3.

Each of these operations can be performed using as many input and output memristors as desired.

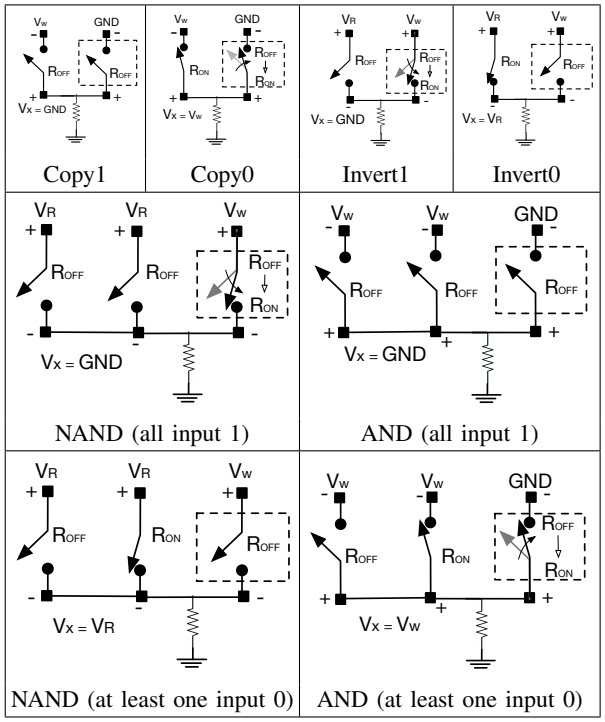
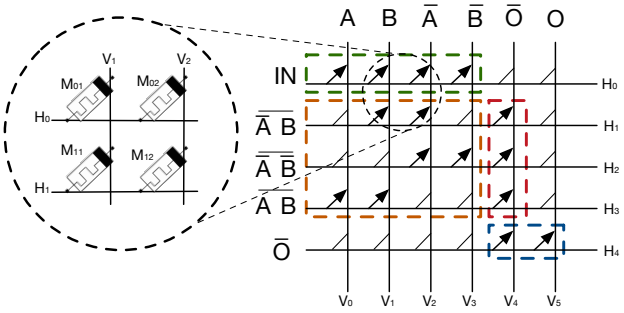


Figure 3. FBLC primitive operations

By driving the crossbar's nano-wires with the right voltages during each step, it is possible to calculate a boolean function in a constant number of steps. We report below in Figure 4 an example of crossbar for a simple 2 inputs / 1 output function, built following the FBLC approach.



$$O = AB + \overline{A}B + \overline{AB} = \overline{\overline{AB} \cdot \overline{\overline{A}B} \cdot \overline{A\overline{B}}}$$

Figure 4. FBLC example

III. Towards Automatic Crossbar Synthesis

Due to the lack of an automated process of translation from a given boolean function to a memristor based crossbar architecture, deeply investigating about memristor based crossbar circuits turns out to be really hard to accomplish. Therefore, in order to overcome these problems, we developed XbarGen.

XbarGen is a command line tool written in C++ which, starting from a boolean function described in the Synopsys equation format - EQN - (fig 5), executes the mapping to a memristor based crossbar architecture and produces a schematic view of the crossbar(s).

```
INORDER = a b c;
OUTORDER = o1;
n5 = !b * c;
n6 = a * !n5;
n7 = b * !c;
n8 = !n5 * !n7;
n9 = !a * !n8;
o1 = n6 + n9;
```

```
INORDER = a b c;
OUTORDER = o1;
o1 = !a*b*!c +
!a*!b*c + a*!c + a*b;
```

Figure 5. Synopsys equation format (EQN)

First, the tool operates an analysis of the given function. As the figure 5 shows, it is possible either that a function's output depends directly on its inputs either that it depends on "*intermediate values*", which indeed depends on inputs.

Thus, we will call level a set of inputs and outputs such that:

- inputs are independent from one another
- each output depends only on inputs

In this way we are able to describe a boolean function as a set of levels each one dependent from one another (figure 6)

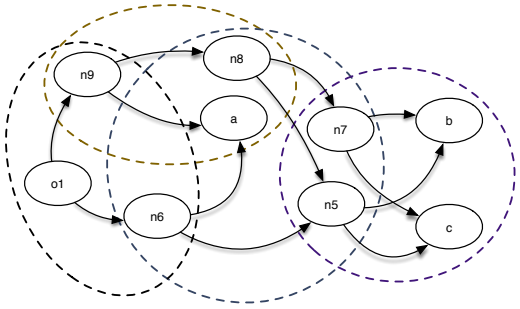


Figure 6. Function's levels

Second, the tool performs a mapping to one or more crossbars depending on how many levels it finds. Mapping to a crossbar means that inputs, outputs and related minterms of each level are translated into a FBLC, as explained in [14] and briefly reminded above. It is worth noting that, when more crossbars are produced, they must be connected together in series, as proposed by Snider in [8] and depicted in figure 7. Consequently, the latency of the circuit grows proportionally to the number of serially connected crossbars.

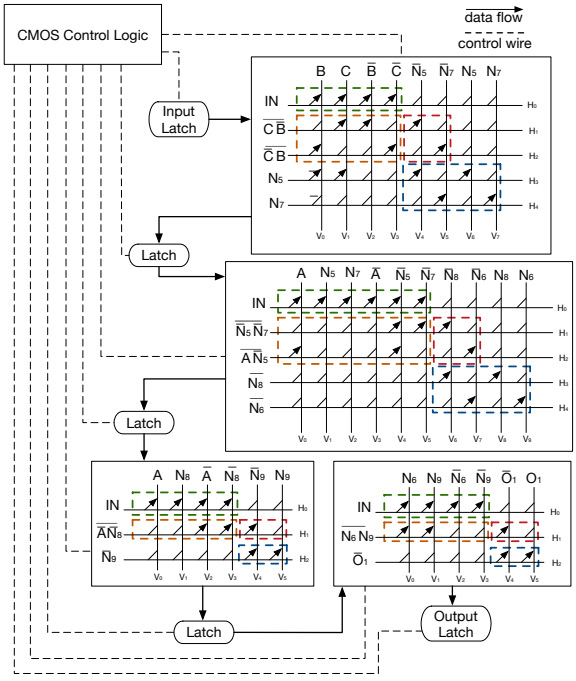


Figure 7. Snider Boolean Logic Circuit

IV. Experiments and Results

This section provides the experimental results achivied by the proposed synthesis tool and the adopted crossbar model. We also compare the proposed approach with the state of the art.

bmark	Inputs	Chakraborti et al. [7]				Single Crossbar						Multiple Crossbars						Multiple Crossbars (optimized)					
		M_s	OP_s	M_p	OP_p	N_{min}	N_{mem}	A	C	N_C	T_{XbG}	N_{min}	N_{mem}	A	C	N_C	T_{XbG}	N_{min}	N_{mem}	A	C	N_C	T_{XbG}
rd53f1	5	7	58	15	34	6	48	96	7	1	4,40771	16	120	492	56	8	13,1694	11	87	378	35	5	9,26288
rd53f2	5	8	59	17	35	20	132	264	7	1	5,71666	27	203	1318	49	7	17,8544	25	191	1108	49	7	17,5566
rd53f3	5	7	41	11	35	16	108	216	7	1	7,27523	22	156	894	42	6	15,5731	13	93	326	42	6	10,0649
xor5_d	5	7	41	11	35	16	108	216	7	1	6,88398	22	156	894	42	6	14,4109	13	93	326	42	6	10,9131
con1f1	7	8	59	17	47	5	37	98	7	1	4,11653	13	105	338	49	7	17,004	9	69	242	28	4	9,66641
con2f2	7	9	47	18	47	6	38	96	7	1	4,71909	16	122	490	49	7	18,7644	10	74	284	28	4	9,93731
rd73f1	7	8	84	18	48	42	330	704	7	1	10,9064	113	851	15232	98	14	92,7335	80	606	9354	70	10	69,9187
rd73f2	7	7	57	13	49	64	528	1056	7	1	14,0526	34	244	1842	56	8	25,8204	19	139	706	42	6	12,7191
rd73f3	7	7	106	25	46	35	275	592	7	1	9,78765	59	459	4186	91	13	39,1519	40	312	2212	63	9	27,0125
newill_d	8	11	104	33	50	22	177	432	7	1	6,02758	51	401	3040	91	13	36,2447	23	177	884	70	10	16,5744
newtagd	8	7	63	14	51	14	107	288	7	1	5,35582	24	192	900	77	11	18,9949	10	78	260	35	5	8,82142
rd84f1	8	8	99	21	57	120	1098	2196	7	1	32,4858	188	1418	37746	119	17	163,551	144	1082	26978	84	12	126,552
rd84f2	8	7	63	15	57	128	1170	2340	7	1	28,5734	40	288	2892	56	8	30,3401	22	162	1028	42	6	14,4034
rd84f3	8	7	62	14	56	1	27	54	7	1	3,52854	7	63	126	49	7	12,4683	7	63	294	21	3	7,68629
rd84f4	8	10	135	33	57	162	1476	2952	7	1	42,7301	176	1336	34468	119	17	158,735	142	1082	26504	84	12	130,351
max46_d	9	28	456	150	72	47	475	980	7	1	10,6472	185	1379	37264	112	16	210,26	156	1188	30628	98	14	141,284
sao2f1	10	9	187	46	73	10	124	264	7	1	5,7189	41	323	2320	91	13	27,696	39	309	2496	70	10	26,6478
sao2f2	10	9	179	46	83	20	242	484	7	1	6,67963	60	474	5226	98	14	49,9699	45	345	3974	63	9	36,1185
sao2f3	10	9	149	34	71	92	914	2068	7	1	23,0123	130	1002	18000	126	18	105,365	80	628	9160	84	12	65,3744
sao2f4	10	9	159	32	69	85	898	1914	7	1	24,8451	137	1051	19684	133	19	111,094	83	651	9810	84	12	66,9386
sym10_d	10	11	196	40	70	837	9229	18458	7	1	592,392	639	4721	372748	154	22	887,967	466	3494	240148	119	17	625,83
t481_d	16	5	137	26	107	1547	21699	52666	7	1	2476,55	657	5033	292290	203	29	878,071	427	3253	177764	161	23	508,84
5xp1	7	14	283	84	73	70	385	2754	7	1	19,3511	140	985	23218	70	10	171,619	101	740	13030	63	9	86,1893
alu2	10	12	1030	284	148	257	2282	8448	7	1	106,399	404	3100	74522	280	40	4250,49	363	2767	71596	217	31	3192,4
alu4	14	8	3634	642	334	1791	19947	79200	7	1	4330,92	740	5706	199562	294	42	54801,5	657	5061	191338	238	34	39688,8
apex1	45	12	7975	1626	705	206	3018	44000	7	1	1471,98	2677	19430	5110198	189	27	11086,1	2020	14817	3827274	119	17	7804,18
apex2	39	10	1701	122	237	1035	15610	85198	7	1	1811,35	445	3627	134188	203	29	614,329	268	2134	71438	140	20	315,608
apex4	9	11	5727	2073	447	438	5489	24678	7	1	2875,89	3466	24824	9858816	147	21	18019,5	2744	19958	7830882	119	17	15452,3
apex5	117	16	6630	806	888	1160	8004	495908	7	1	1401,94	1293	9630	1239620	147	21	3189,77	832	6213	629036	84	12	1863,43
apex6	135	22	3761	770	1169	656	4926	353808	7	1	669,392	740	5138	569256	105	15	1405,3	674	4678	505306	98	14	1227,78
apex7	49	28	1937	290	437	507	4126	93740	7	1	315,733	252	1769	53172	98	14	279,106	197	1404	36774	91	13	232,827
b9	41	19	634	125	298	107	642	15996	7	1	73,3105	113	861	20768	70	10	117,437	92	690	15366	56	8	95,8065
clip	9	21	485	120	89	166	1078	4816	7	1	41,0498	180	1354	33454	77	11	175,761	111	829	14098	63	9	107,43
cm150a	21	21	199	56	127	17	142	836	7	1	10,778	62	468	4502	91	13	50,1216	47	363	4102	63	9	48,2901
cm162a	14	14	198	46	102	43	230	1862	7	1	32,4292	38	298	2216	63	9	31,2726	33	257	1626	56	8	34,8671
cm163a	16	8	176	42	116	42	229	2016	7	1	23,054	36	288	2148	56	8	30,6175	32	254	1810	49	7	34,41
dalu	75	13	4856	627	470	2224	26700	407862	7	1	5701	1387	10169	879430	245	35	20674,4	1122	8270	700910	217	31	15307,3
e64	65	24	840	94	456	65	2470	34060	7	1	125,161	1437	12134	416608	448	64	2436,47	520	3885	348982	70	10	1088,73
ex1010	10	11	6606	1984	396	1024	18950	41400	7	1	108896	3350	23952	8992430	168	24	17142,5	2616	19132	7027470	119	17	15397,7
ex4	128	7	3076	256	928	620	5220	124460	7	1	359,811	478	3672	299178	112	16	735,931	408	3124	248724	98	14	618,099
frg2	143	22	8433	803	1005	4159	35729	2424636	7	1	13511,8	1208	8712	1265816	91	13	3972,73	728	5142	518280	77	11	1768,91
misex1	8	18	231	83	69	18	132	780	7	1	7,30883	72	504	7314	49	7	59,2965	61	423	5740	42	6	46,7335
misex3	14	14	2969	444	185	1426	15559	80696	7	1	2430,17	1582	11604	1803240	161	23	4496,13	1167	8691	1280548	140	20	3152,18
misex3c	14	10	2453	429	239	297	2710	18096	7	1	701,002	734	5380	483346	161	23	1188,36	590	4340	405166	126	18	1003,96
parity	16	7	119	23	113	32768	557090	1114180	7	1	1,95466e+06	46	346	3956	56	8	51,0531	46	346	3956	56	8	50,9398
pdc	16	17	3658	507	142	2192	61318	250096	7	1	1,65169e+06	1624	11994	1398822	182	26	4124,41	840	6290	627542	147	21	1741,17
seq	41	11	10808	1566	692	1066	14502	167504	7	1	2323,22	2413	17877	3580692	203	29	9025,33	1775	13437	2541004	154	22	6382,24
squar5	5	18	224	93	56	30	261	1014	7	1	15,6276	68	484	6936	49	7	55,5393	48	326	3928	42	6	34,2392
t481	16	11	137	26	107	481	5267	16422	7	1	232,543	1875	13495	3463670	147	21	7784,26	803	5875	857962	91	13	2170,26
table5	17	14	4068	580	168	158	2566	11136	7	1	416,991	2001	14648	2370246	182	26	5730,48	1481	11082	1841246	154	22	4415,77
tcon	17	7	166	57	118	32	154	3234	7	1	18,7235	48	298	6034	21	3	35,5259	40	226	5218	14	2	29,4298
term1	34	10	896	105	260	257	2298	23584	7	1	88,2616	313	2420	107300	112	16	393,415	150	1165	29526	84	12	162,668
too_large	38	14	2866	282	232	1035	15610	85198	7	1	1389,22	827	6375	411848	210	30	1394,65	463	3621	194054	168	24	699,077
vda	17	11	3039	805	273	93	1544	14896	7	1	376,691	955	6855	640324	112	16	2693,42	650	4638	444516	70	10	1737,85
x1	51	11	2810	230	398	309	2624	59340	7	1	156,096	400	3007	179366	91	13	544,872	286	2091	123350	70	10	371,451
x2	10	11	206	60	80	17	118	850	7	1	7,6228	60	428	5680	49	7	52,4899	46	314	3636	42	6	42,7509
x3	135	22	3761	770	1169	738	5535	392184	7	1	648,998	841	6241	893392	105	15	1961,42	611	4527	550626	77	11	1221,77
x4	94	39	3042	401	642	530	3431	198660	7	1	329,832	470	3378	207320	70	10	660,801	341	2403	112264	63	9	406,664

Table I
COMPARISON OF BENCHMARKS

A. Experiments

In order to investigate about the impact of different synthesis optimization parameters on the memristor crossbar in terms of area and performances, we carried out several experiments.

Given a function expressed in BLIF format, we first used synthesis tools (i.e. ABC and SIS by Berkeley) in order to produce EQN format. Specifically we used:

- SIS to generate equations in which outputs depend directly on inputs, so they can be translated to a single crossbar
- ABC to generate equations in which outputs depend on *intermediate values,* so they can be translated to more crossbars. Also an *"optimized"* version has been produced using the *resyn2* script [15]

An overview about achieved results is presented and discussed hereafter. In table II a legend of the used labels is reported.

B. Discussion

Table I shows the synthesis results on a number of combinational benchmarks. We noticed that:

1) As the levels' number of a function grows - thus the number of crossbars and, hence, the latency - the number of memristor grows instead of decreasing.
2) Given the number of crossbars, the response time of the circuit remains constant for any function.
   a) Therefore, in terms of both area and performances, is better to have only a single crossbar in order to obtain a tiny and constant time and use fewer memristors.
3) The number of minterms has an high impact on the memristors' total number.
   a) At the current state of art, there is not any tool that makes possible to optimize a boolean function in terms of number of minterms. Maybe the next direction could be such.

Symbol	Description	Expressed as
p_{ij}	Whether the *i-th* minterm is in the *j-th* level (boolean)	Given
$N_{in}(l_j)$	Num. of input for the *j-th* level	Given
$N_{out}(l_j)$	Num. of output for the *j-th* level	Given
$N_{occ}(m_i, l_j)$	Num. of occurrences of the *i-th* minterm in the *j-th* level	Given
$N_{lit}(m_i)$	Num. of literals of the *i-th* minterm	Given
N_{mem}	No. of memristors in the circuit	$\sum_j \left[2 * N_{in}(l_j) + \sum_i (N_{occ}(m_i, l_j)) + \sum_i (N_{lit}(m_i) * p_{ij}) + 2 * N_{out}(l_j) \right]$
N_{min}	No. of minterms in the circuit	Given
A	Total area	$\sum_j \left\{ [2 * N_{in}(l_j) + 2 * N_{out}(l_j)] * \left[1 + \sum_i (p_{ij}) + N_{out}(l_j) \right] \right\}$
t_M	Memristor switching time	Given
N_{Steps}	Num. of steps of a Crossbar computation	Given
T_C	"Latency" of a Crossbar	$t_M * N_{Steps}$
N_C	Num. of Crossbar in the circuit	Given
C	Needed Cycles to complete computation	$T_C * N_C$
T_{XbG}	Time (in milliseconds) needed by XbarGen to execute	Experimental
T_{sim}	Time (in milliseconds) for simulate VHDL design (50ns long)	Experimental
Chakraborti et al. [7]		
M_s	No. of memristors (serial)	
OP_s	No. of memristor micro-operations (serial)	
M_p	No. of memristors (parallel)	
OP_p	No. of memristor micro-operations (parallel)	

Table II
LEGEND

In addition, with regard to feasability of translation and simulation of memristor crossbar behavioral circuits, we collected the time that XbarGen needs to translate a boolean function and produce the crossbar schematic view (table I). XbarGen experiments were run on a dual-core i7 MacBook Pro with 2,8GHz clock, 4 GB RAM and running OSX v10.11.4.

C. Comparison

Chakraborti et al. in [7] proposed an architecture based on material implication operation implemented using memristors. In brief, they came up with a realization of 2-to-1 multiplexer using memristors, and a synthesis methodology that represents a given Boolean function as a Reduced Ordered Binary Decision Diagram (ROBDD) and maps it to memristor implementation. They carried out some benchmarks too (table I), reporting interesting results: comparing experiments that exploit parallelism and those that do not, the inverse ratio time-area is respected; from our side, instead, it is not true. On the other hand, execution time of the architecture in [7] depends on the function under consideration; execution time of Snider architecture could be independent from the function, if it is translated to a single crossbar. This could mean that one would prefer an architecture such as proposed in [7] in case of area constraints and an architecture such as Snider proposed in [8] in case of time constraints.

V. Conclusion

The memristor is one of the most promising technologies which is able to deal with the CMOS limitations. In particular, since the memristor is inherently able to behave also as a non-volatile device, it allows to overcome the bottleneck of the data transfer to the computational unit and back to storage elements. The research community is facing with the synthesis of boolean functions by exploiting memristor-based crossbars and, in this paper, we advanced the state-of-the-art with a comparison with the state-of-the-art of performances w.r.t. several benchmarks. Indeed, we developed XbarGen, a tool which is able to process boolean equations for the mapping over memristor crossbars.

References

[1] ITRS 2013 report. [Online]. Available: http://www.itrs.net/

[2] B. Hoefflinger, "Chips 2020: A Guide to the Future of Nanoelectronics", The Frontiers Collection, Springer Berlin Heidelberg, 2012, pp. 421–427

[3] J. McPherson, "Reliability trends with advanced CMOS scaling and the implications for design", in IEEE Custom Integrated Circuits Conference, 2007, pp. 405–412

[4] S. Borkar, "Design perspectives on 22Nm CMOS and beyond", in Proceedings of the 46th Annual Design Automation Conference, 2009, pp. 93-94

[5] G. Gielen, et al., "Emerging yield and reliability challenges in nanometer CMOS technologies", in Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1322-1327, 2008

[6] S. Hamdioui, et al., 'Memristor based computation-in-memory architecture for data-intensive applications', in Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1718-1725, 2015

[7] S. Chakraborti, P. V. Chowdhary, K. Datta and I. Sengupta, "BDD based synthesis of Boolean functions using memristors," 2014 9th International Design and Test Symposium (IDT), Algiers, 2014, pp. 136-141.

[8] G. Snider, "Computing with hysteretic resistor crossbars," Applied Physics A, vol. 80, pp. 1165–1172, 2005.

[9] L. Chua, 'Memristor-the missing circuit element', IEEE Transactions on Circuit Theory, vol. 18, no. 5, pp. 507–519, 1971

[10] R. Waser et al., "Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges," Advanced Materials, vol. 21, pp. 2632–2663, 2009

[11] J. J. Yang et al., "Memristive devices for computing," Nature nanotechnology, vol. 8, pp. 13–24, 2013

[12] J. R. Burger et al., "Variation-tolerant computing with memristive reservoirs," IEEE/ACM International Symposium in Nanoscale Architectures (NANOARCH), 2013, pp. 1–6.

[13] K.-H. Kim et al., "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," Nano letters, vol. 12, pp. 389–395, 2011.

[14] Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Said Hamdioui, Koen Bertels, “Fast Boolean Logic Mapped on Memristor Crossbar”, IEEE International Conference on Computer Design (ICCD), pp. 335-342, 2015.
[15] ABC User Guide. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/