Yise - A novel Framework for Boolean Networks using Y-Inverter Graphs

Arun Chandrasekharan
Group of Computer Architecture
University of Bremen, Germany
arun@cs.uni-bremen.de

Daniel Große
[1]Group of Computer Architecture
University of Bremen, Germany
[2]Cyber Phyiscal Systems,
DFKI GmbH, Bremen, Germany
grosse@cs.uni-bremen.de

Rolf Drechsler
[1]Group of Computer Architecture
University of Bremen, Germany
[2]Cyber Phyiscal Systems,
DFKI GmbH, Bremen, Germany
drechsle@cs.uni-bremen.de

ABSTRACT

In this paper we introduce the novel framework *Yise* for representing logic. Unlike the conventional approaches, *Yise* uses a Y-Inverter Graph (YIG) to represent the Boolean network at hand. Such a YIG represents Y-functions, which are single output, six input Boolean functions composed of three majority functions connected in a triangular (Y) fashion. We show that YIGs are a super set of the well-known and very successful logic representation data-structures AND/OR/Majority/Inverter Graphs which include AIGs and MIGs. Our results on a wide range of benchmarks show very compact representations of the logic without compromising system requirements. Up to 33% reduction in the node count can be achieved compared to AIGs without increasing the number of logic levels.

ACM Reference format:
Arun Chandrasekharan, Daniel Große, and Rolf Drechsler. 2017. Yise - A novel Framework for Boolean Networks using Y-Inverter Graphs. In *Proceedings of MEMOCODE '17, Vienna, Austria, September 29-October 2, 2017,* 4 pages.
https://doi.org/10.1145/3127041.3127065

1 INTRODUCTION

Logic synthesis is a crucial step for the design of today's and future VLSI systems. Hence, the performance of the implemented design in terms of area, power and timing, highly depends on the quality of the logic synthesis step. In order to achieve this, the synthesis tool requires a *compact representation* of the *Boolean network*. Throughout the history of *Electronic Design Automation* (EDA), several forms of Boolean networks have been used such as *Sum-of-Products* (SOP), *Binary Decision Diagrams* (BDD) [5], *And Inverter Graphs* (AIGs) [11], and recently *Majorty-Inverter Graphs* (MIGs) [1]. However, the circuit sizes and the complexity of the designs continue to increase in accordance with the Moore's law, and this puts a high demand on the EDA tools to improve.

Another challenge for EDA is to cope up with the emerging technologies of the future. These are the novel devices and fabrication technologies that are expected to be mainstream once the *beyond CMOS* era starts. Examples for such technologies are quantum dot based logic [16], spintronics logic devices [17], DNA based logic [9], resistive RAM devices [12] etc. Several of these technologies depend on median algebra since the fundamental device is best described as a *majority voter*, rather than a digital switch. Even though there is a considerable amount of research in this field, EDA tools developed for these technologies are still in infancy.

In this paper we propose to use *Y-Inverter Graphs* (YIGs) for representing logic. A YIG is a *Directed Acyclic Graph* (DAG). A node in the YIG, called Y-gate, represents a single output, six input Boolean Y-function with optional inversions at the inputs and the output. A Y-function consists of three majority functions connected in a triangular (Y) fashion. As a proof-of-concept we have implemented the YIG framework *Yise*[1]. We show that YIGs compactly represent Boolean networks by comparing them to the state-of-the-art data structures.

The remainder of this paper is structured as follows. In Section 2, the necessary background for this work is discussed. The details about YIGs and the developed framework *Yise* are provided in Section 3. In Section 4 the experimental results are given. The concluding remarks are provided in the final section.

2 BACKGROUND

A *Boolean network* is a DAG where nodes (vertices) represent logic gates or Primary Inputs/Primary Outputs (PIs/POs), and edges represent wires that form the interconnection among the gates. Note that in a general Boolean network representation edges and nodes can have polarity showing inversion. An *And-Inverter Graph* (AIG) is special Boolean network where each node is an AND gate with two inputs and one output [11]. For AIGs the number of nodes correspond to the area of the technology independent synthesized circuit and the maximum level of the AIG correspond to the delay of the same. Note that the maximum level is alternately called the *longest path* or *height* of the AIG. This number represents the maximum distance in terms of number of nodes from any primary input to any primary output. AIG is the state-of-the-art logic representation choice for several synthesis tools [6, 11].

Another special type of DAG which is popular in EDA tools is *Binary Decision Diagram* (BDD) [5]. BDDs are formed by applying Shannon decomposition for the considered Boolean function. *Reduced Ordered Binary Decision Diagram* (ROBDD) are the *canonical* version of BDDs where no sub-BDD is represented more than

[1] *Yise* package is publicly available at: https://gitlab.com/arunc/yise

MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria

ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127065

once. For the remainder of this paper, "BDD" stands for ROBDD. BDDs are unique to a given input variable order. Hence, the logical equivalence of two designs can be easily determined by comparing the BDDs of both functions for a fixed variable order. Note that in contrast an AIG is fundamentally non-canonical.

3 Y-INVERTER GRAPH AND YISE

As mentioned before we introduce a new class of Boolean network called *Y-inverter graph* (YIG) based on majority logic. We study and compare the results obtained with YIG with other forms such as AIG and BDD. The Y-inverter graph and its properties are explained in the this section, followed by the details on our framework *Yise*. The experimental results and the comparison with other DAGs are provided separately in Section 4 after this.

3.1 Y-Inverter Graph

A Y-inverter graph is a homogeneous Boolean network with 6-inputs and 1-output where each node represent a Y-function. If we represent the majority function of variables a, b and c using the notation

$$\langle a, b, c\rangle := ab + bc + ca, \quad (1)$$

a *Y-function* from the 6 input variables *a, b, c, d, e, f* is given by the formulation

$$y(a, b, c, d, e, f) = \langle\langle a, b, c\rangle, \langle b, d, e\rangle, \langle c, e, f\rangle\rangle \quad (2)$$

i.e., Y-function is formed using the majority of three majority functions. We follow the notation used in [8] for majority and Y-functions. The inter-connections of the edges a, b, c, d, e and f can be easily visualized with the help of Fig 1. The three majority gates consisting of $\langle a, b, c\rangle$, $\langle b, d, e\rangle$ and $\langle c, e, f\rangle$ form the three triangles shown in shaded color. Each of these smaller triangles form a 3-input majority function with inputs at the vertex of the triangle. The output of these three majority gates are further given as input to a next level of majority gate to form the final output. A Y-gate is formed with Y-function as the node and optional inversions in the inputs and outputs. The Y-inverter graph is composed of only Y-gates. We now consider some properties of YIGs.

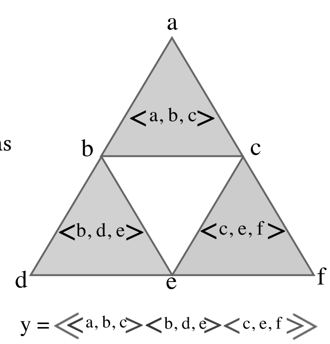

Figure 1: Y-gate visual representation

A YIG can represent several other Boolean networks such as *Majority-Inverter Graph* (MIG), *And/Or-Inverter Graph* (AOIG) and *And-Inverter Graph* (AIG). This is formally stated as follows:

THEOREM 1. *YIG ⊃ MIG ⊃ AOIG ⊃ AIG*

We provide the proof for the first part of Theorem 1 here, i.e., *YIG ⊃ MIG*. We refer to [8] for the remaining part of the theorem.

PROOF. From the definition of YIG in Equation 2,
$y(a, b, c, d, e, f) = \langle\langle a, b, c\rangle, \langle b, d, e\rangle, \langle c, e, f\rangle\rangle$

Now, assign $d = c$, $e = a$ and $f = b$:
$y(a, b, c, c, a, b) = \langle\langle a, b, c\rangle, \langle b, c, a\rangle, \langle c, a, b\rangle\rangle = \langle a, b, c\rangle$

□

In other words we have shown that a YIG node can contain any 3-input majority gate (MIG). An MIG node contains an AOIG node and is an universal representation [8]. Hence, together with Theorem 1 we get the following

COROLLARY 1. *YIG is a universal representation form.*

Therefore, a YIG is sufficient to implement any Boolean function. Besides, a single Y-gate (node) can compactly represent several popular Boolean functions such as any 3-input majority gate or any 3 (or 2) input and-or-inverter gates[2]. Note that a basic primitive in MIG, AOIG or AIG cannot contain a YIG primitive. Furthermore, the three input form of the AND and OR logic gate cannot be contained in a single MIG node, but only in a single YIG node.

There are several other interesting properties of Y-functions: A Boolean function f is *self-dual* when it satisfies the property $\overline{f(x_1, x_2, x_3, ..., x_n)} = f(\overline{x_1}, \overline{x_2}, \overline{x_3}, ..., \overline{x_n})$. Y-functions are self-dual [8]. This property is important in technologies such as quantum-dot cellular automata where the cost of the inverter is significant [10]. It allows to reduce the inverter count significantly by transferring the inversions from the input side to the output side or vice-versa. A Boolean function f is positive (negative) *unate* in x if and only if there exists a normal form of the function in which x does not appear complemented (uncomplemented) [15]. Y-functions are unate in all of its arguments. Furthermore, Y-functions have strong error correction properties. As an example, from the Equation 2, when $a = d$ and $c = e$, any error in the input f does not affect the output of the Y-gate. However, in comparison with a BDD, a representation based on YIG is not canonical.

The number of nodes in the YIG corresponds to the area of the technology independent synthesized circuit and the maximum level of the YIG corresponds to the delay of the circuit[3].

3.2 Yise

Yise is a YIG framework built entirely using Y-functions. *Yise* has been developed in C++. Currently *Yise* can read in a circuit, convert it into an equivalent YIG representation and write out the synthesized results in Verilog format. We also provide a textual representation of the YIG. The textual format follows the recommendations in [14].

Yise uses a Boolean matching algorithm to convert the input design specification to a YIG. This approach is illustrated in Algorithm 1. The algorithm uses a pre-computed look-up table[4] comprising of the functions (in terms of truth-table) for all the input variable combinations of the Y-function. Each of these functions is mapped to an optimal representation of Y-gates in the look-up table. Note that this mapping is computed offline and stored in *Yise* as a hash table. This Y-function map is shown in Algorithm 1 as *Y_FunctionTable yt*, and is an input to the algorithm. The design is parsed and the corresponding DAG is formed (Line 3 in

[2] i.e., both 2-input and 3-input AND, NAND, OR, NOR gates.
[3] In this work, we restrict ourselves to CMOS technologies that follow Boolean logic. Other post CMOS devices are left for future work.
[4] We use NPN-classification [4] to generate the look-up table.

Algorithm 1). Further 6-input sub-graphs (*cut-set* of the DAG) are enumerated in the design in the reverse topological order starting from the primary outputs and ending at the primary inputs. The local function of each of these sub-graphs is computed and matched with the stored Y-function hash table (Lines 4, 5 and 6). The best Y-gate representation is selected from this hash table corresponding to a given local function and the YIG is constructed from these Y-gates (Lines 7 and 8). Finally the algorithm returns the complete YIG representation of the input design.

Algorithm 1 *Yise* YIG construction

```
1:  function CREATE_YIG ( Design f, Y_FunctionTable yt )
2:      set yig ← initialize ()
3:      set dag ← parse_design ( f )
4:      set cuts ← enumerate_six_input_cuts ( dag )
5:      for each C ∈ cuts do
6:          set l ← local_function ( C )
7:          set y ← select_best_graph ( l, yt )
8:          set yig ← add_to_yig ( y )
9:      end for
10:     return yig
11: end function
```

We explain next the experimental results obtained using *Yise*.

4 EXPERIMENTAL RESULTS

We have used a wide range of standard benchmark circuits to evaluate *Yise*. The evaluation is carried out in the number of nodes, the maximum level of the YIG graph and the time taken to synthesize a given circuit. The experiments have been carried out on a laptop computer running Ubuntu 16.04 edition Linux with Intel Core-i5 CPU. We have taken five different standard benchmarks from the EPFL circuits [2], ISCAS-85 [7], LGSynth-91 [18], arithmetic circuits from [3] and the circuits distributed as part of [14]. All the results obtained using *Yise* are formally verified to be equivalent with the initial circuit specification.

4.1 YIG Comparison with AIG

In this section we compare the results obtained using *Yise* with ABC [11], which heavily uses AIGs. These results are summarized in Table 1. The general structure of Table 1 is as follows. The first three columns give the circuit details such as the name of the circuit and the number of primary inputs/outputs. The next two columns provide the number of nodes and the maximum level of the AIG graph. After this the corresponding numbers for the YIG obtained using *Yise* is provided, followed by the relative reduction in the number of nodes. Recall that the number of nodes represent the area of the technology independent circuit and the maximum level corresponds to the delay.

The number of nodes and the maximum level for YIG is same or better than AIG in all the circuits reported. For several arithmetic circuits such as adders and multipliers the reduction is more pronounced (see for e.g., results from set:4 in the Table 1). The multipliers (Array, Wallace and Dadda) in set:4 have a reduction in the node count of more than 30%. This has to be expected since an important section of the logic in these circuits consists of majority function which are represented very compactly using YIG.

4.2 YIG Comparison with BDD

A comparison of YIG generated using *Yise* package and BDD benchmarks taken from [13] is given in Table 2. The first three columns of the Table 2 are circuit name, number of primary inputs, outputs. The next column is the BDD node count followed by the YIG node count. It can be easily seen that YIG outperforms BDD. There is a significant difference in the number of nodes between YIG and BDD. However, note that as mentioned before BDDs have a very important property of canonicity, which YIG lacks. The Table 2 is sufficient to show the general trend. Hence, further evaluation is omitted due to lack of space.

4.3 Scalability and Run Time

Yise takes about 1 sec CPU time to read in and write out the results for most of the circuits in Table 1. Note that the EPFL benchmarks (set:1) are among the biggest combinational benchmarks publicly available. Furthermore, the arithmetic circuits given in set:4 includes large multipliers such as 128-bit array multiplier. The biggest circuit evaluated with *Yise* is the *hypotenuse* from EPFL (set:1, 4^{th} entry on the right side of Table 1). This circuit with a node count over 200K Y-gates is synthesized under 2 sec.

5 CONCLUDING REMARKS

The results confirm the potential of the introduced *Yise* package. *Yise* can generate a compact representations of logic using YIGs in very short run times.

There are several directions of future work. One main aspect is to extend the *Yise* framework with a technology mapper and optimization techniques. Using *Yise* for novel technologies such as quantum dot is also another important direction.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and in the project MANIAC (DR 287/29-1), the German Federal Ministry of Education and Research (BMBF) within the project 01IW16001 (SELFIE), the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative and by the German Academic Exchange Service (DAAD).

REFERENCES

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2014. Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization. In *DAC*. 194:1–194:6.

[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL Combinational Benchmark Suite. In *IWLS*. http://infoscience.epfl.ch/record/207551

[3] Aoki 2016. *Aoki Laboratory - Graduate School of Information Sciences. Tohoku University*. http://www.aoki.ecei.tohoku.ac.jp/arith.

[4] Luca Benini and Giovanni De Micheli. 1997. A Survey of Boolean Matching Techniques for Library Binding. *TODAES* 2, 3 (July 1997), 193–226.

[5] R. E. Bryant. 1995. Binary Decision Diagrams and Beyond: Enabling Techniques for Formal Verification. In *ICCAD*. 236–243.

[6] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. 2016. Approximation-aware rewriting of AIGs for error tolerant applications. In *ICCAD*. 83:1–83:8.

[7] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. 1999. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *D&T* 16, 3 (1999), 72–80.

[8] Donald E. Knuth. 2011. *The Art of Computer Programming, Section 7.1.1. Boolean Basics*. Vol. 4A. Addison-Wesley, Upper Saddle River, New Jersey.

Table 1: Comparison of YIG vs AIG

set:1 EPFL circuits [2]			AIG		YIG			set:1 EPFL circuits [2]			AIG		YIG		
Benchmark	PI	PO	nodes	levels	nodes	levels	Δ%	Benchmark	PI	PO	nodes	levels	nodes	levels	Δ%
Round robin arbiter	256	129	11839	87	11647	86	1.62	Adder	256	129	1020	255	893	129	12.45
Alu control unit	7	26	174	10	138	9	20.68	Barrel shifter	135	128	3336	12	3080	11	7.67
Coding-cavlc	10	11	693	16	644	16	7.07	Divisor	128	128	57247	4372	56718	4343	0.92
Decoder	8	256	304	3	304	3	0.00	Hypotenuse	256	128	214335	24801	211715	24801	1.22
i2c controller	147	142	1342	20	1223	20	8.86	Log2	32	32	32060	444	31572	442	1.52
Int to float converter	11	7	260	16	235	15	9.61	Max	512	130	2865	287	2864	286	0.03
Memory controller	1204	1231	46836	114	42046	110	10.22	Multiplier	128	128	27062	274	26937	271	0.46
Priority encoder	128	8	978	250	914	247	6.54	Sine	24	25	5416	225	5226	219	3.50
Lookahead XY router	60	30	257	54	245	51	4.66	Square-root	128	64	24618	5058	20700	4127	15.91
Voter	1001	1	13758	70	11608	68	15.62	Square	64	128	18484	250	18173	250	1.68
set:2 Benchmarks from [14]			AIG		YIG			set:3 LGSynth-91 [18]			AIG		YIG		
circuit	pi	po	nodes	levels	nodes	levels	Δ%	circuit	pi	po	nodes	levels	nodes	levels	Δ%
circuit0	128	160	8136	23	7457	22	8.34	alu2	10	6	325	46	267	44	17.84
circuit1	128	94	5326	25	4965	25	6.77	alu4	14	8	622	55	529	54	14.95
circuit2	207	108	2893	61	2212	49	23.53	cm163a	16	5	37	13	31	10	16.21
circuit3	512	130	2832	184	2828	184	0.14	count	35	16	128	18	95	18	25.78
circuit4	20	1	1991	60	1439	39	27.72	dalu	75	16	1306	26	1110	26	15.01
circuit5	32	32	7002	703	6329	516	9.61	frg1	28	3	186	39	151	38	18.81
circuit6	65	16	744	58	744	58	0.00	term1	34	10	180	16	148	16	17.77
circuit7	78	120	1235	84	1187	79	3.88	unreg	36	16	83	5	82	4	1.20
circuit8	420	1	1186	94	1181	92	0.42	x2	10	7	42	8	33	7	21.42
circuit9	12	1	772	109	713	109	7.64	z4ml	7	4	36	17	30	12	16.66
set:4 Arithmetic circuits [3]			AIG		YIG			set:5 ISCAS-85 [7]			AIG		YIG		
circuit	pi	po	nodes	levels	nodes	levels	Δ%	circuit	PI	PO	nodes	levels	nodes	levels	Δ%
HanCarlson_add_32	64	33	430	15	426	15	0.93	c7552	207	108	2074	29	1879	27	9.40
BrentKung_add_32	64	33	361	20	360	20	0.27	c1908	33	25	341	27	318	25	6.74
KoggeStone_add_32	64	33	577	13	569	13	1.38	c17	5	2	6	3	5	2	16.66
Adder4	64	18	537	20	415	19	22.71	c1355	41	32	502	25	396	21	21.11
MAC_32	48	33	857	65	605	40	29.40	c5315	178	123	1776	37	1471	36	17.17
Array_Mult_32	64	64	11712	200	7872	110	32.78	c499	41	32	398	19	364	18	8.54
Array_Mult_64	128	128	48000	408	32128	222	33.06	c432	36	7	208	26	196	23	5.76
Wallace_Mult_32	64	64	12184	187	8356	79	31.41	c3540	50	22	1024	41	940	38	8.20
Wallace_Mult_64	128	128	49312	378	33456	144	32.15	c2670	157	64	716	20	580	18	18.99
Dadda_Mult_32	64	64	11712	186	7872	70	32.78	c880	60	26	325	25	303	24	6.76
Dadda_Mult_64	128	128	48000	378	32128	134	33.06	c6288	32	32	2337	12	2336	120	0.04
Array_Mult_128	256	256	150407	753	117022	344	22.19								

PI, PO: Number of primary inputs, primary outputs AIG: And-Inv Graph results from ABC YIG: Y-Inv Graph from *Yise*
nodes, levels: Number of nodes and maximum level in AIG and YIG (corresponds to area and delay of the DAG graph)
Δ: Percentage change in the number of nodes in YIG relative to the number of nodes in AIG. $\Delta = \frac{\text{nodes}_{\text{AIG}} - \text{nodes}_{\text{YIG}}}{\text{nodes}_{\text{AIG}}}$
EPFL benchmarks from [2]. Benchmarks are from [14]. Arithmetic circuits from [3], ISCAS-85 from [7] and LGSynth-91 from [18]

Table 2: YIG vs BDD node count for ISCAS-85 circuits [7]

Circuit	PI	PO	#BDD	#YIG	Circuit	PI	PO	#BDD	#YIG
c1908	33	25	7764	318	c2670	157	64	7469	580
c1355	41	32	29609	396	c3540	50	22	27666	940
c499	41	32	34113	364	c7552	207	108	9808	1879

PI, PO: Primary inputs, outputs. #BDD, #YIG: BDD, YIG node counts
#BDD is taken from [13], #YIG from *Yise*

[9] Wei Li, Yang Yang, Hao Yan, and Yan Liu. 2013. Three-Input Majority Logic Gate and Multiple Input Logic Circuit Based on DNA Strand Displacement. *Nano Letters* 13, 6 (2013), 2980–2988.

[10] W. Liu, L. Lu, M. O'Neill, and E. E. Swartzlander. 2014. A First Step Toward Cost Functions for Quantum-Dot Cellular Automata Designs. *IEEE Transactions on Nanotechnology* 13, 3 (2014), 476–487.

[11] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. 2006. DAG-aware AIG Rewriting a Fresh Look at Combinational Logic Synthesis. In *DAC*. 532–535.

[12] S. M. Patil and S. R. S. Prabhaharan. 2015. Memory array with complementary resistive switch with memristive characteristics. In *Intl. Conf on Advances in Computing, Communications and Informatics*. 508–513.

[13] C. Scholl, D. Moeller, P. Molitor, and R. Drechsler. 1999. BDD Minimization Using Symmetries. *TCAD* 18, 2 (1999), 81–100.

[14] M. Soeken. 2017. IWLS 2017 Programming Contest: "Y Logic Synthesis". In *IWLS*.

[15] Y. Tohma. 1964. Decompositions of Logical Functions Using Majority Decision Elements. *IEEE Transactions on Electronic Computers* (1964), 698–705.

[16] P. D. Tougaw and C. S. Lent. 1994. Logical devices implemented using quantum cellular automata. In *Joural of Applied Physics*. 1811–1817.

[17] J. Wang, H. Meng, and J.P. Wang. 2005. Programmable spintronics logic device based on a magnetic tunnel junction element. In *Joural of Applied Physics*. 1811–1817.

[18] Saeyang Yang. 1991. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0. (1991).