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
1Group of Computer Architecture
University of Bremen, Germany
2Cyber Physcial Systems,
DFKI GmbH, Bremen, Germany
grosse@cs.uni-bremen.de

Rolf Drechsler
1Group of Computer Architecture
University of Bremen, Germany
2Cyber Physcial 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:

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 Majority-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 Yise1. 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

1Yise package is publicly available at: https://gitlab.com/arunc/yise
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 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
\[
(a, b, c) := ab + bc + ca,
\]
a Y-function from the 6 input variables a, b, c, d, e, f is given by the formulation
\[
y(a, b, c, d, e, f) = \langle (a, b, c), (b, d, e), (c, e, f) \rangle \tag{1}
\]
i.e., Y-function is formed using the majority of three variables. 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 (a, b, c), (b, d, e) and (c, e, f) 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.

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,
\[
y(a, b, c, d, e, f) = \langle (a, b, c), (b, d, e), (c, e, f) \rangle
\]

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

□

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
\[
f(x_1, x_2, x_3, ..., x_n) = f(\overline{x_1}, \overline{x_2}, \overline{x_3}, ..., \overline{x_n}).
\]
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 (un complemented) [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 \( \text{Y Function Table yt} \), 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.
We have used a wide range of standard benchmark circuits to evaluate ABC [11], which heavily uses AIGs. These results are summarized in 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, 4th 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 representation 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

Table 1: Comparison of YIG vs AIG

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PI</th>
<th>PO</th>
<th>nodes</th>
<th>levels</th>
<th>Δ%</th>
<th>Benchmark</th>
<th>PI</th>
<th>PO</th>
<th>nodes</th>
<th>levels</th>
<th>Δ%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round robin arbiter</td>
<td>256</td>
<td>129</td>
<td>11839</td>
<td>87</td>
<td>11647</td>
<td>86</td>
<td>1.62</td>
<td>Adder</td>
<td>256</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>Alu control unit</td>
<td>7</td>
<td>26</td>
<td>174</td>
<td>10</td>
<td>138</td>
<td>9</td>
<td>20.68</td>
<td>Barrel shifter</td>
<td>135</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Coding-cavlc</td>
<td>10</td>
<td>11</td>
<td>693</td>
<td>16</td>
<td>644</td>
<td>16</td>
<td>7.07</td>
<td>Divisor</td>
<td>128</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Decoder</td>
<td>8</td>
<td>256</td>
<td>304</td>
<td>3</td>
<td>304</td>
<td>3</td>
<td>0.00</td>
<td>Hypotenuse</td>
<td>256</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>f2 controller</td>
<td>147</td>
<td>142</td>
<td>1342</td>
<td>20</td>
<td>1223</td>
<td>20</td>
<td>8.86</td>
<td>Log2</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Int to float converter</td>
<td>11</td>
<td>7</td>
<td>260</td>
<td>16</td>
<td>235</td>
<td>15</td>
<td>9.61</td>
<td>Max</td>
<td>512</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>Memory controller</td>
<td>1204</td>
<td>1231</td>
<td>46836</td>
<td>114</td>
<td>42046</td>
<td>110</td>
<td>10.22</td>
<td>Multiplier</td>
<td>128</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>Priority encoder</td>
<td>128</td>
<td>8</td>
<td>978</td>
<td>250</td>
<td>914</td>
<td>247</td>
<td>6.54</td>
<td>Sine</td>
<td>24</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>LookAhead XY router</td>
<td>60</td>
<td>30</td>
<td>257</td>
<td>54</td>
<td>245</td>
<td>51</td>
<td>4.66</td>
<td>Square-root</td>
<td>128</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Voter</td>
<td>1001</td>
<td></td>
<td>13758</td>
<td>70</td>
<td>11608</td>
<td>68</td>
<td>15.62</td>
<td>Square</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Circuit</th>
<th>PI</th>
<th>PO</th>
<th>#BDD</th>
<th>YIG</th>
<th>Circuit</th>
<th>PI</th>
<th>PO</th>
<th>#BDD</th>
<th>YIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1908</td>
<td>33</td>
<td>25</td>
<td>c2670</td>
<td>157</td>
<td>64</td>
<td>7469</td>
<td>580</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e1355</td>
<td>41</td>
<td>32</td>
<td>c3540</td>
<td>50</td>
<td>22</td>
<td>27666</td>
<td>940</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e499</td>
<td>41</td>
<td>32</td>
<td>c7552</td>
<td>207</td>
<td>108</td>
<td>9808</td>
<td>1879</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Circuit</th>
<th>PI</th>
<th>PO</th>
<th>#BDD</th>
<th>YIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1908</td>
<td>33</td>
<td>25</td>
<td>c2670</td>
<td>157</td>
</tr>
<tr>
<td>e1355</td>
<td>41</td>
<td>32</td>
<td>c3540</td>
<td>50</td>
</tr>
<tr>
<td>e499</td>
<td>41</td>
<td>32</td>
<td>c7552</td>
<td>207</td>
</tr>
</tbody>
</table>

[Citations]


