Rewriting And-inverter graphs (AIG)

December 5, 2016

A project report submitted in partial fulfillment of the requirements of the course ECEN 5139 titled Computer Aided Verification.

Lakhan Shiva Kamireddy

Contents

1 INTRODUCTION

The choice of representation for circuits and boolean formulae in a formal verification scenario is important. This is mainly attributed to the goal to keep memory consumption low by making use of compact representations. Also, many formal verification algorithms are sensitive to the redundancies in the design that is processed. To address these concerns we try to use various compression techniques. And-inverter graph rewriting is focused in this direction. And-inverter graph (AIG) is a directed, acyclic graph. It is one way to represent the logic functionality of a circuit or network. AIG consists of 2-input nodes representing logical conjunction, terminal nodes labeled with variable names, and edges optionally containing markers indicating logical negation. It is an efficient representation for manipulation of Boolean functions. AIGs can be small compared to Conjunctive Normal Form(CNF) representation. Some circuits that are linear in AIGs are exponential in CNF. We studied another form of representing logic functionalities of a network in class, viz. Binary Decision Diagrams (BDD). Conversion from a network of logic gates to AIGs is fast and scalable. It only requires every gate be expressed in terms of AND gates and inverters. Below is an illustration of how a circuit can be converted to an And-inverter graph.

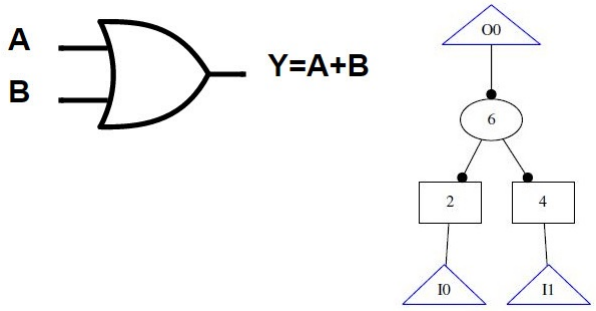


Fig. 1. Constructing And-inverter graphs

AIGs don't have a canonical representation like BDDs. This is the reason that makes AIG an efficient representation in comparison with BDDs with loss of canonicity. Also, techniques like rewriting can be applied to AIGs unlike BDDs for the same reason that they are not canonical.

AIG rewriting is an optimization technique that is alternated to reduce area by sharing common logic without increasing delay. It is an innovative technique for combinational logic synthesis. My work is centered around implementing the DAG-Aware Rewriting algorithm proposed by Berkeley Logic Synthesis and Verification Group in [6]. It has been further showed in [6] that this methodology scales to very large designs and is several orders of magnitude faster than logic synthesis tools like SIS and MVSIS without compromising the quality of the network after mapping.

2 AIG REWRITING

Rewriting is a fast greedy algorithm for minimizing the AIG size by iteratively selecting AIG subgraphs rooted at node and replacing them with smaller pre-computed subgraphs.

During the construction of the And-inverter graph, each vertex is entered into a hash table using the vertices of the two input operands and their polarities as key. Identical vertex keys are a sufficient condition for structural equivalence. The hash table can thus be used during the graph construction to map isomorphic parts of the two circuits onto the same subgraph. For example, when the functions ab and $\bar{a}\bar{b}$ are present in a circuit, they are identified as structurally equivalent because of modulo inversion and mapped to the same vertex in the graph model. I have worked with AIGs making extensive use of the AIGER library. The input and output And-inverter graphs were taken in AIGER format as described in [1] and [4]. I have run the AIG rewriting implementation against some of the benchmarks written for the new AIGER format and presented the results.

3 BOOLEAN NETWORKS

A Logic Network is a directed acyclic graph.

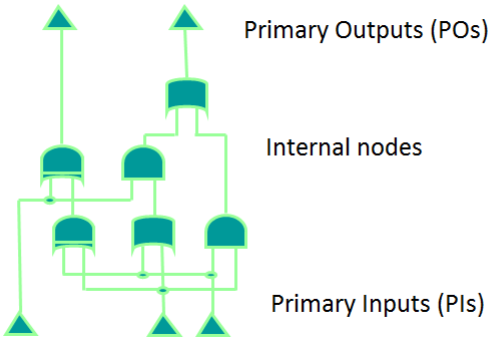


Fig. 2. Logic Network

Each node has only one output. A node can have any number of inputs (fanins) and can be input to any number of nodes (fanouts)

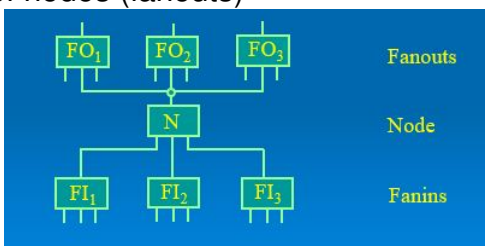


Fig. 3. Node, Fanin and Fanout

Transitive fanin or transitive fanout means, there is either a direct or indirect connection between the gates.

Cut:

A cut of a node n is a set of nodes(leaves) in transitive fan-in such that every path from the node to PIs is blocked by nodes (at least one leaf) in the cut.

A k-feasible cut means the size of the cut must be k or less.

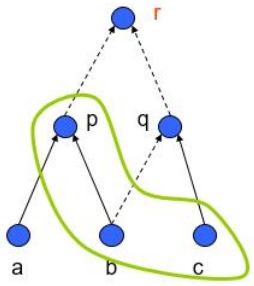


Fig. 4. The set of leaves $\{p, b, c\}$ is a 3-feasible cut of the node r. It is also a 4-feasible cut.

As a side note to this information k-feasible cuts are important in technology mapping (especially FPGA mapping) since the logic between a node and the nodes in its cut can be replaced by a k-LUT(look up table).

k-feasible cut computation:

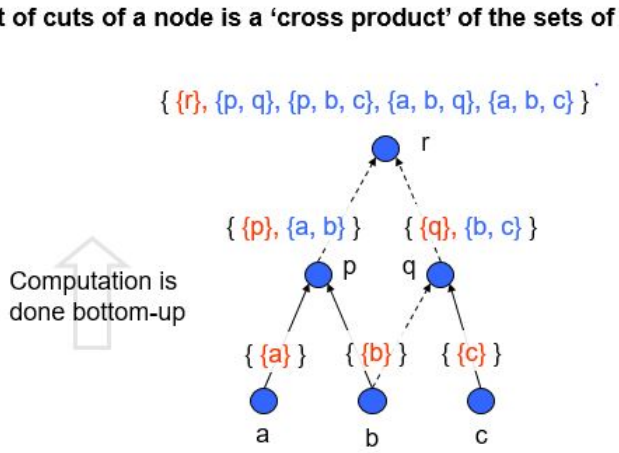


Fig. 5. k-feasible cut computation

Note that any cut that is of size greater than k is discarded. The cut function is the function of node n in terms of the cut leaves.

The fact that many different 2-input functions may have same gate level implementation gives rise to the idea of equivalence. P equivalence between two functions is obtained when it is possible to achieve identical values for both truth table outputs by permuting the function inputs. Functions that are P equivalent can be grouped into P classes. The most important property of P equivalent functions is that they can always be implemented with the same cell from a library (circuit). If we look further in this direction, it is possible to see that even P classes may have similar implementations. These functions may be grouped into NPN equivalence class.

NPN equivalence:

Two Boolean functions, F and G belong to same NPN-class (are NPN-equivalent) if F can be derived from G by negating (N) and permuting (P) inputs and negating (N) the output.

For example, $F = ab + c$ and $G = ac + b$ are NPN-equivalent because swapping b and c make them identical. Functions $F = ab + c$ and $G = ab$ are not NPN-equivalent because no amount of permuting and complementing variables can make a 3-variable

function equivalent to a 2-variable function.

If there are n variables, there are 2^n combinations, let us say there are 2 variables, we have 4 combinations. Each of these combinations can be true or false, i.e. for n variables there are 2^{2^n} boolean functions possible. We note that here in spite of existence of 16 different two-input functions, there are only 4 different 2-input NPN classes. There are two NPN classes composed of 2 functions, one NPN class composed of 4 functions, and one NPN class composed of 8 functions. NPN equivalent functions can be implemented with the same circuit plus some inverters (that can be used to negate the inputs and outputs, if necessary). This way it is possible to use a smaller library composed of one representative gate for each NPN class plus one inverter cell.

Implementation of n-input functions

$f_0 = 0$	
$f_1 = \overline{A}.\overline{B}$	
$f_2 = \overline{A}.B$	
$f_3 = \overline{A}$	
$f_4 = A.\overline{B}$	
$f_5 = \overline{B}$	
$f_6 = A \oplus B$	
$f_7 = \overline{A.B}$	
$f_8 = A.B$	
$f_9 = \overline{A \oplus B}$	
$f_{10} = B$	
$f_{11} = \overline{A.\overline{B}}$	
$f_{12} = A$	
$f_{13} = \overline{\overline{A}.B}$	
$f_{14} = \overline{\overline{A}.\overline{B}}$	
$f_{15} = 1$	

Fig. 6. Combinations of all 2 variable functions. There are 2^{2^2} total combinations.

f_0	
f_1	
f_2 f_4	
f_3 f_5	
f_6	
f_7	
f_8	
f_9	
f_{10} f_{12}	
f_{11} f_{13}	
f_{14}	
f_{15}	

Fig. 7. Identifying all P classes. Only permutations on input variables allowed (to be

equivalent).

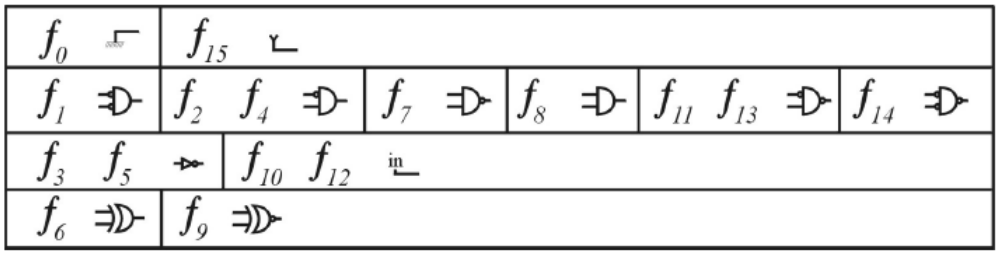


Fig. 8. Identifying all NPN classes. Permutations and negations on input variables allowed and only negation on output allowed (to be equivalent).

Data structure used for AIG representation:

A simple AIG data structure allows quick and cheap structural hashing among AIG nodes. Two AIG nodes with the same inputs under the same complementation conditions are merged (something like the reduction rule in ROBDD). Unlike ROBDD, however AIG representation is not canonical even when structural hashing is applied.

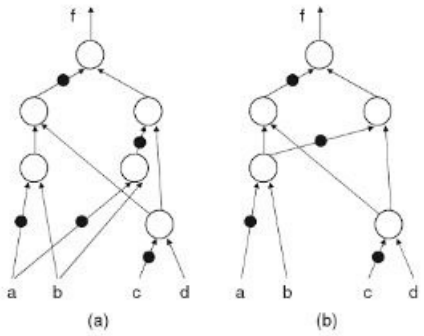


Fig. 9. Enumerating Structural hashing. AIGs for function $a\bar{c}d+\bar{b}\bar{c}d$ (a) without structural hashing (b) with structural hashing

Structural Hashing: Structural hashing applied during AIG construction propagates constants and ensures that each node is structurally unique. Complemented edges: Accordingly AIGs are stored in a compact form. AIGs represent inverters as attributes on edges and therefore don't require extra memory. Regularity: As a result of regularity, memory management of an AIG package can be done by a simple customized memory manager which uses fixed amount of memory for each node (thanks to the fixed number of inputs to each node). By allocating memory for nodes in a topological order, we can optimize AIG traversal which is repeatedly performed in many logic synthesis algorithms, in the same order. AIGs can also be used in verification applications, such as equivalence checking and even model checking. For instance, checking if two given AIGs under comparison are functionally equivalent can be reduced to a SAT checking by adding an XNOR (XOR) gate which can be expressed in terms of AND2 and INV gates with its two

inputs fed in by the outputs of the two AIGs. The two AIGs are equivalent if and only if the output of the XNOR (or XOR) gate is unsatisfiable. When it comes to synthesis, AIGs are used in multilevel logic minimization and technology mapping. It is used as a unifying data structure for both logic synthesis and verification. A new binary format called AIGER was recently proposed to enable compact representation of the AIGs in files and memory. With memory requirements of about three bytes for AIG node, AIGER has become a standard representation for circuit based problems in SAT competitions and Hardware model checking competitions organized annually as a part of International conference on theory and applications of satisfiability theory and International conference on CAV.

Alan Mishchenko, et al have proposed new technology independent combinational logic synthesis flow using fast local transformations of And-Inverter-Graphs. The flow improves on the traditional logic synthesis by addressing the above difficulties, advantages are as follows:

1. While still being heuristic and suboptimal, the new algorithm does not require as much hand-tuning and trial and error.

2. Improvements in the complexity of the logic are measured by AIG nodes and levels, is in better correspondence with both standard-cell and FPGA mappers, which use AIGs or similar data structures as subject graphs.

3. It is much simpler. A robust implementation reported in Alan's paper took a few person-weeks to implement.

4. It is orders of magnitude faster than the traditional flow, even when compared with its most rugged and robust versions, while the quality is comparable or better when measured by the delay and area of the network after technology mapping.

AIG rewriting is local; however, rewriting is very fast and can be applied to the network many times. For example, performing ten rewriting passes over a typical network is still at least an order of magnitude faster than running the resource-aware implementation of the traditional flow in MVSIS. By applying rewriting many times, the scope of changes is no longer local. The result is that the cumulative effect of several rewriting passes is often superior to traditional synthesis in terms of quality.

4 ALGORITHM

It is a Fast greedy algorithm.

The idea is to iteratively select AIG subgraphs rooted at a node and replacing them with smaller pre-computed subgraphs, while preserving the functionality of the root node. The following steps are followed.

-> For the purpose of 4-input AIG rewriting, all 4-feasible cuts of the nodes are enumerated using the procedure described above.

-> For each cut, the boolean function is computed and its NPN-class is determined by hash-table lookup.

-> Fast manipulation of 4-variable functions is achieved by representing them using truth tables stored as 16-bit bit-strings.

As per [3], although there are 222 NPN equivalence classes of 4-variable functions, out of those, only about 100 appear more than once as functions of 4-feasible cuts in the

numerous benchmarks, and only about 40 of these have been found experimentally to lead to improvements in rewriting.

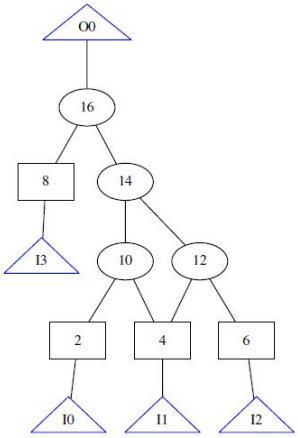
Pseudocode:

```
Rewriting(network AIG, hash table PrecomputedStructures, bool UseZeroCost)
{
    for each node N in the AIG in the topological order{
        for each 4-input cut C of node N computed using cut enumeration{
            F=Boolean function of N in terms of the leaves C
            PossibleStructures=HashTableLookup(PrecomputedStructures, F);
            //find the best logic structure for rewriting
            BestS = NULL; BestGain=-1;
            for each structure S in PossibleStructures{
                NodesSaved=DereferenceNode(AIG, N);
                NodesAdded=ReferenceNode(AIG, S);
                Gain = NodesSaved - NodesAdded;
                Dereference(AIG,S); Reference(AIG,N);
                if (Gain > 0 || (Gain = 0 && UseZeroCost))
                    if (BestS = NULL || BestGain < Gain)
                        BestS = S;BestGain = Gain;
            }
            if (BestS == NULL) continue;
            //use the best logic structure to update the netlist
            NodesSaved = DereferenceNode(AIG,N);
            NodesAdded = ReferenceNode(AIG,S);
            assert (BestGain = NodesSaved - NodesAdded);
        }
    }
}
```

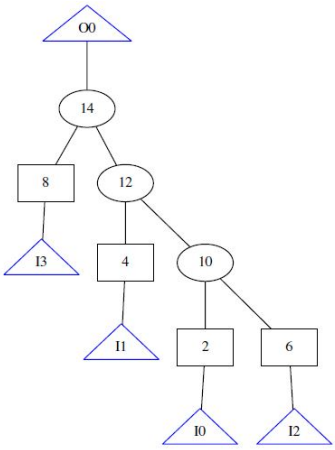
5 RESULTS

The results were as follows:

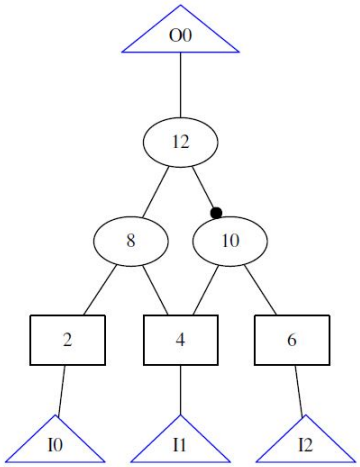
Input AIG:



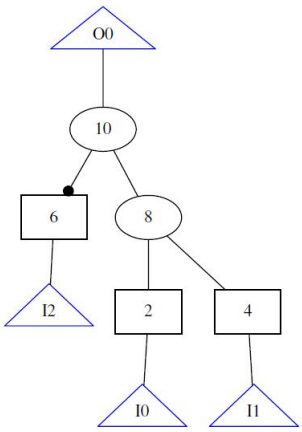
Output AIG:



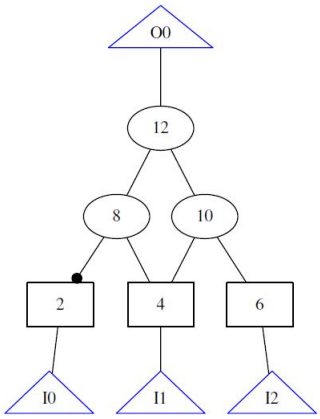
Input AIG:



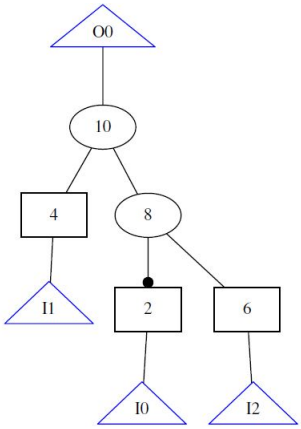
Output AIG:



Input AIG:



Output AIG:



Improvement using the current AIG rewriting implementation:

S.no	No. of and gates before rewriting	After rewriting
1.	4	3
2.	3	2
3.	3	2

A quick test on AIGER benchmarks has not showed much improvement in rewriting the BEEM benchmark AIGER files. This may be due to the fact that only 40 NPN classes have found experimentally to lead to improvements in rewriting. The structural rewriting that i have implemented does not exhaustively cover all the NPN equivalence classes, rather covers some structures. The absence of rewriting for most of these structures in the implementation should be the reason for its poor performance on benchmarks. However, in future a hash table look up implementation technique (pre-computed structures stored at one place) attempting rewriting for most of these structures can be made to achieve better results.

References

[1] Armin Biere. Aiger (aiger is a format, library and set of utilities for and-inverter graphs (aigs)), 2006.

[2] P. Bjesse and A. Boralv. Dag-aware circuit compression for formal verification. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 42–49, Nov 2004.

[3] Vinícius P Correia and André I Reis. Classifying n-input boolean functions. In *VII Workshop Iberchip*, page 58, 2001.

[4] Johannes Keppler University Formal Verification Group. Benchmarks. Technical report, http://fmv.jku.at/aiger/, 2004.

[5] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 263–268, New York, NY, USA, 1997. ACM.

[6] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535. ACM, 2006.