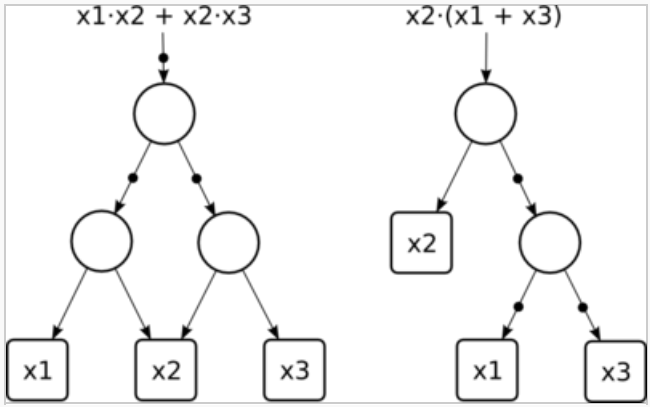
And-inverter graph

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(July 2014)*

An **and-inverter graph (AIG)** is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network. An AIG consists of two-input nodes representing logical conjunction, terminal nodes labeled with variable names, and edges optionally containing markers indicating logical negation. This representation of a logic function is rarely structurally efficient for large circuits, but is an efficient representation for manipulation of boolean functions. Typically, the abstract graph is represented as a data structure in software.



Two structurally different AIGs for the function f(x1, x2, x3) = x2 * (x1 + x3)

Conversion from the network of logic gates to AIGs is fast and scalable. It only requires that every gate be expressed in terms of AND gates and inverters. This conversion does not lead to unpredictable increase in memory use and runtime. This makes the AIG an efficient representation in comparison with either the binary decision diagram (BDD) or the "sum-of-product" (ΣoΠ) form,[*citation needed*] that is, the canonical form in Boolean algebra known as the disjunctive normal form (DNF). The BDD and DNF may also be viewed as circuits, but they involve formal constraints that deprive them of scalability. For example, ΣoΠs are circuits with at most two levels while BDDs are canonical, that is, they require that input variables be evaluated in the same order on all paths.

Circuits composed of simple gates, including AIGs, are an "ancient" research topic. The interest in AIGs started in the late 1950s[1] and continued in the 1970s when various local transformations have been developed. These transformations were implemented in several logic synthesis and verification systems, such as Darringer et al.[2] and Smith et al.,[3] which reduce circuits to improve area and delay during synthesis, or to speed up formal equivalence checking. Several important techniques were discovered early at IBM, such as combining and reusing multi-input logic expressions and subexpressions, now known as structural hashing.

Recently there has been a renewed interest in AIGs as a functional representation for a variety of tasks in synthesis and verification. That is because representations popular in the 1990s (such as BDDs) have reached their limits of scalability in many of their applications.[*citation needed*] Another important development was the recent emergence of much more efficient boolean satisfiability (SAT) solvers. When coupled with *AIGs* as the circuit representation, they lead to remarkable speedups in solving a wide variety of boolean problems.[*citation needed*]

AIGs found successful use in diverse EDA applications. A well-tuned combination of *AIGs* and boolean satisfiability made an impact on formal verification, including both model checking and equivalence checking.[4] Another recent work shows that efficient circuit compression techniques can be developed using AIGs.[5] There is a growing understanding that logic and physical synthesis problems can be solved using AIGs simulation and boolean satisfiability compute functional properties (such as symmetries[6]) and node flexibilities (such as don't-cares, resubstitutions, and SPFDs[7]). This work shows that AIGs are a promising *unifying* representation, which can bridge logic synthesis, technology mapping, physical synthesis, and formal verification. This is, to a large extent, due to the simple and uniform structure of AIGs, which allow rewriting, simulation, mapping, placement, and verification to share the same data structure.

In addition to combinational logic, AIGs have also been applied to sequential logic and sequential transformations. Specifically, the method of structural hashing was extended to work for AIGs with memory

elements (such as D-type flip-flops with an initial state, which, in general, can be unknown) resulting in a data structure that is specifically tailored for applications related to retiming.[8]

Ongoing research includes implementing a modern logic synthesis system completely based on AIGs. The prototype called ABC features an AIG package, several AIG-based synthesis and equivalence-checking techniques, as well as an experimental implementation of sequential synthesis. One such technique combines technology mapping and retiming in a single optimization step. These optimizations can be implemented using networks composed of arbitrary gates, but the use of AIGs makes them more scalable and easier to implement.

Implementations [edit]

- Logic Synthesis and Verification System ABC
- A set of utilities for AIGs AIGER
- OpenAccess Gear

References [edit]

1. ^ L. Hellerman (June 1963). "A catalog of three-variable Or-Inverter and And-Inverter logical circuits". *IEEE Trans. Electron. Comput*. EC-12 (3): 198–223. doi:10.1109/PGEC.1963.263531.
2. ^ A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan (1981). "Logic synthesis through local transformations". *IBM J. of Research and Development* **25** (4): 272–280. doi:10.1147/rd.254.0272.
3. ^ G. L. Smith, R. J. Bahnsen, H. Halliwell (1982). "Boolean comparison of hardware and flowcharts". *IBM J. of Research and Development* **26** (1): 106–116. doi:10.1147/rd.261.0106.
4. ^ A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai (2002). "Robust boolean reasoning for equivalence checking and functional property verification". *IEEE Trans. CAD* **21** (12): 1377–1394.
5. ^ P. Bjesse and A. Boralv. "DAG-aware circuit compression for formal verification". *Proc. ICCAD '04*. pp. 42–49.
6. ^ K.-H. Chang, I. L. Markov, V. Bertacco. "Post-placement rewiring and rebuffering by exhaustive search for functional symmetries". *Proc. ICCAD '05`pages=56–63*.
7. ^ A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske (May 2006). "Using simulation and satisfiability to compute flexibilities in Boolean networks". *IEEE Trans. CAD* **25** (5): 743–755.
8. ^ J. Baumgartner and A. Kuehlmann. "Min-area retiming on flexible circuit structures". *Proc. ICCAD'01*. pp. 176–182.

See also [edit]

- Binary decision diagram
- Logical conjunction

This article is adapted from a column in the ACM SIGDA e-newsletter by Alan Mishchenko
Original text is available here.

Categories: Graph data structures | Diagrams | Electrical circuits | Electronic design automation | Formal methods