

Introduction to Logic Synthesis with ABC

Alan Mishchenko

UC Berkeley

Overview

- (1) Problems in logic synthesis
 - Representations and computations
- (2) And-Inverter Graphs (AIGs)
 - The foundation of innovative synthesis
- (3) AIG-based solutions
 - Synthesis, mapping, verification
- (4) Introduction to ABC
 - Differences, fundamentals, programming
- (5) Programming assignment

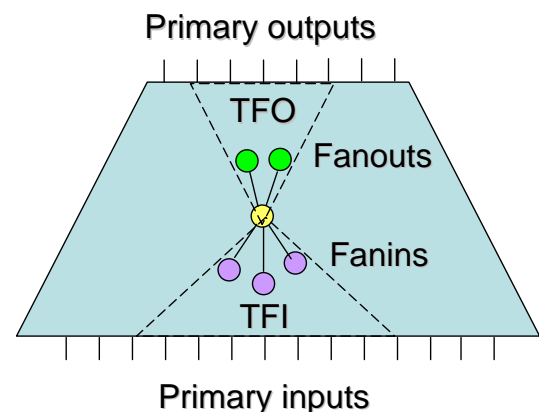
(1) Problems In Synthesis

- What are the objects to be “synthesized”?
 - Logic structures
 - Boolean functions (with or without don't-cares)
 - State machines, relations, sets, etc
- How to represent them efficiently?
 - Depends on the task to be solved
 - Depends on the size of an object
- How to create, transform, minimize the representations?
 - Multi-level logic synthesis
 - Technology mapping
- How to verify the correctness of the design?
 - Gate-level equivalence checking
 - Property checking
 - etc

3

Terminology

- Logic function (e.g. $F = ab+cd$)
 - Variables (e.g. b)
 - Minterms (e.g. $ab\bar{c}d$)
 - Cube (e.g. ab)
- Logic network
 - Primary inputs/outputs
 - Logic nodes
 - Fanins/fanouts
 - Transitive fanin/fanout cone
 - Cut and window (defined later)



4

Logic (Boolean) Function

- Completely specified logic function

		ab			
		00	01	11	10
cd	00	0	0	1	0
	01	0	0	1	0
	11	1	1	1	1
	10	0	0	1	0

- Incompletely specified logic function

		ab			
		00	01	11	10
cd	00	0	0	1	0
	01	0	-	-	-
	11	1	1	1	-
	10	0	0	1	0

		ab			
		00	01	11	10
cd	00	0	0	1	0
	01	0	0	0	0
	11	1	1	1	0
	10	0	0	1	0

On-set

		ab			
		00	01	11	10
cd	00	1	1	0	1
	01	1	0	0	0
	11	0	0	0	0
	10	1	1	0	1

Off-set

		ab			
		00	01	11	10
cd	00	0	0	0	0
	01	0	1	1	1
	11	0	0	0	1
	10	0	0	0	0

DC-set

Relations

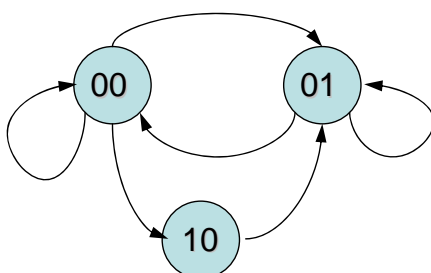
- Relation $(a_1, a_2) \rightarrow (b_1, b_2)$

- $(0,0) \rightarrow (0,0)$
- $(0,1) \rightarrow (1,0)(0,1)$
- $(1,0) \rightarrow (1,1)$
- $(1,1) \rightarrow (1,0)$

Characteristic function

		a1 a2			
		00	01	11	10
b1 b2	00	1	0	0	0
	01	0	1	0	0
	11	0	0	0	1
	10	0	1	1	0

- FSM



Current state

		Next state			
		00	01	11	10
Next state	00	1	1	-	0
	01	1	1	-	1
	11	-	-	-	-
	10	1	0	-	0

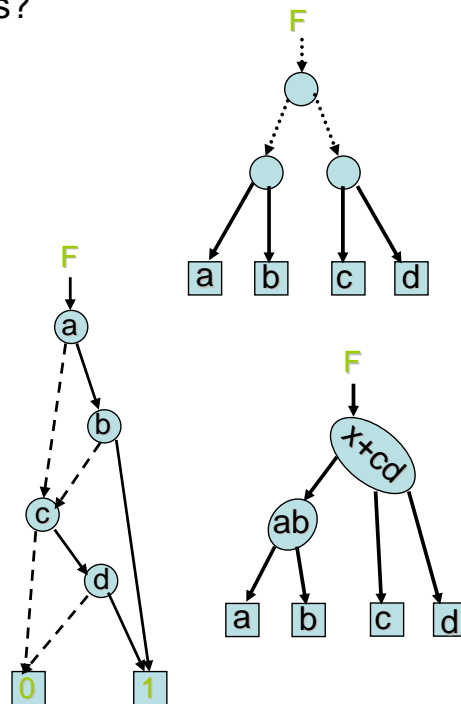
Representation Zoo

Find each of these representations?

- Truth table (TT)
- Sum-of-products (SOP)
- Product-of-sums (POS)
- Binary decision diagram (BDD)
- And-inverter graph (AIG)
- Logic network (LN)

$$F = ab+cd$$

$$F = (a+c)(a+d)(b+c)(b+d)$$



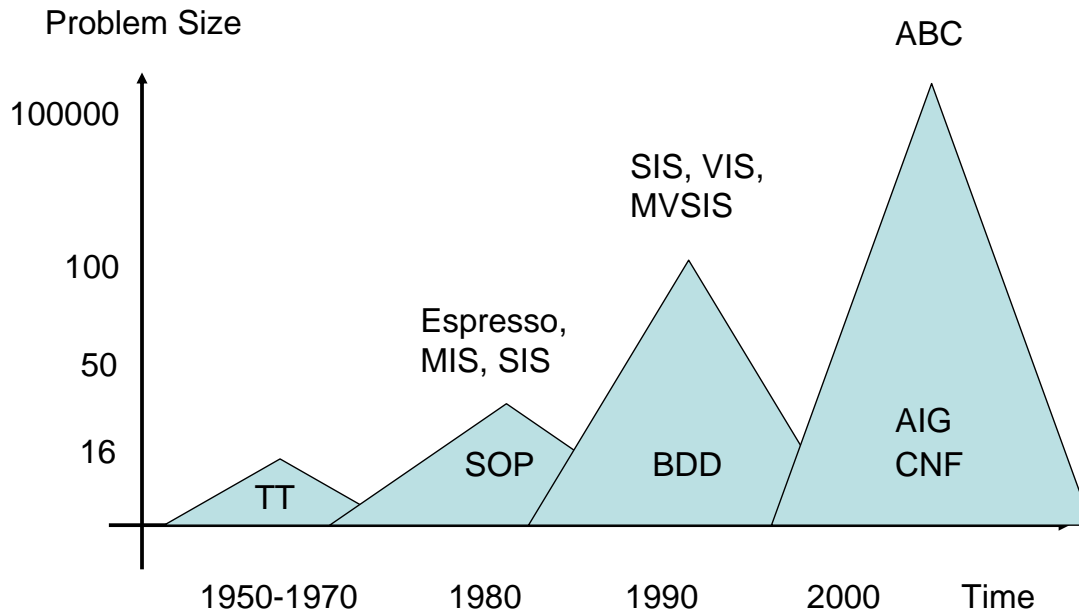
abcd	F
0000	0
0001	0
0010	0
0011	1
0100	0
0101	0
0110	0
0111	1
1000	0
1001	0
1010	0
1011	1
1100	1
1101	1
1110	1
1111	1

7

Representation Overview

- **TT are the natural representation of logic functions**
 - Not practical for large functions
 - Still good for functions up to 16 variables
- **SOP is widely used in synthesis tools since 1980's**
 - More compact than TT, but not canonical
 - Can be efficiently minimized (SOP minimization by Espresso, ISOP computation) and translated into multi-level forms (algebraic factoring)
- **BDD is a useful representation discovered around 1986**
 - Canonical (for a given function, there is only one BDD)
 - Very good, but only if (a) it can be constructed, (b) it is not too large
 - Unreliable (non-robust) for many industrial circuits
- **AIG is an up-and-coming representation!**
 - Compact, easy to construct, can be made “canonical” using a SAT solver
 - Unifies the synthesis/mapping/verification flow
 - The main reason to give this talk 😊

Historical Perspective



9

What Representation to Use?

- For small functions (up to 16 inputs)
 - TT works the best (local transforms, decomposition, factoring, etc)
- For medium-sized functions (16-100 inputs)
 - In some cases, BDDs are still used (reachability analysis)
 - Typically, it is better to represent as AIGs
 - Translate AIG into CNF and use SAT solver for logic manipulation
 - Interpolate or enumerate SAT assignments
- For large industrial circuits (>100 inputs, >10,000 gates)
 - Traditional LN representation is not efficient
 - AIGs work remarkably well
 - Lead to efficient synthesis
 - Are a natural representation for technology mapping
 - Easy to translate into CNF for SAT solving
 - etc

10

What are Typical Transformations?

- Typical transformations of representations
 - For SOP, minimize cubes/literals
 - For BDD, minimize nodes/width
 - For AIG, restructure, minimize nodes/levels
 - For LN, restructure, minimize area/delay

11

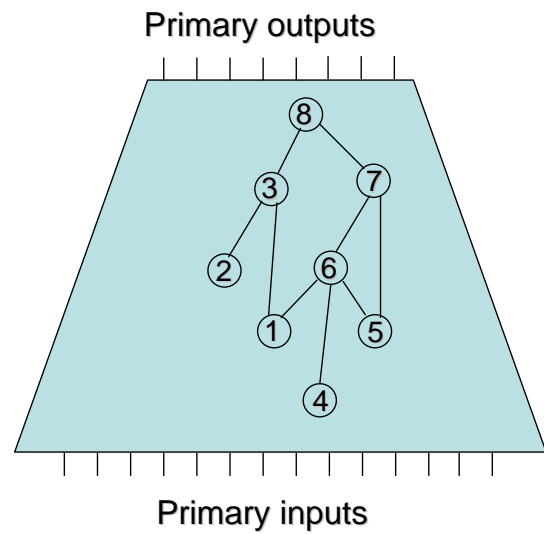
Algorithmic Paradigms

- Divide-and-conquer
 - Traversal, windowing, cut computation
- Guess-and-check
 - Bit-wise simulation
- Reason-and-prove
 - Boolean satisfiability

12

Traversal

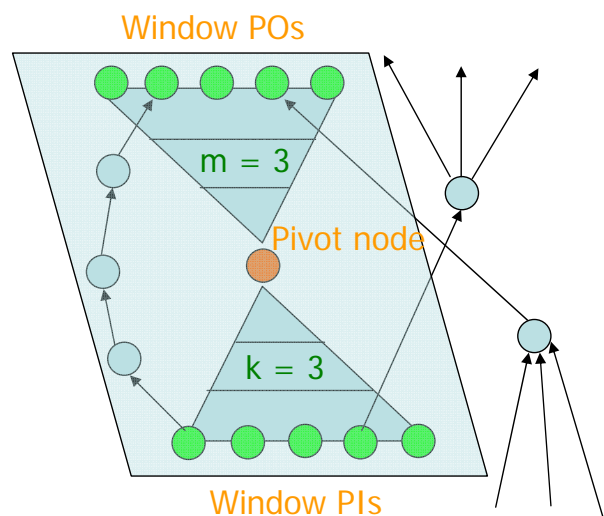
- **Traversal** is visiting nodes in the network in some order
- **Topological order** visits nodes from PIs to POs
 - Each node is visited after its fanins are visited
- **Reverse topological order** visits nodes from POs to PIs
 - Each node is visited after its fanouts are visited



Traversal in a topological order

Windowing

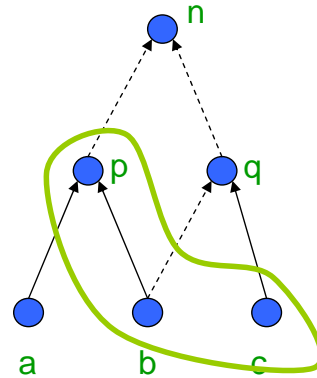
- **Definition**
 - A **window** for a node is the node's context, in which an operation is performed
- A window includes
 - k levels of the TFI
 - m levels of the TFO
 - all re-convergent paths between window PIs and window POs



Structural Cuts in AIG

A **cut** of a node n is a set of nodes in transitive fan-in such that every path from the node to PIs is blocked by nodes in the cut.

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



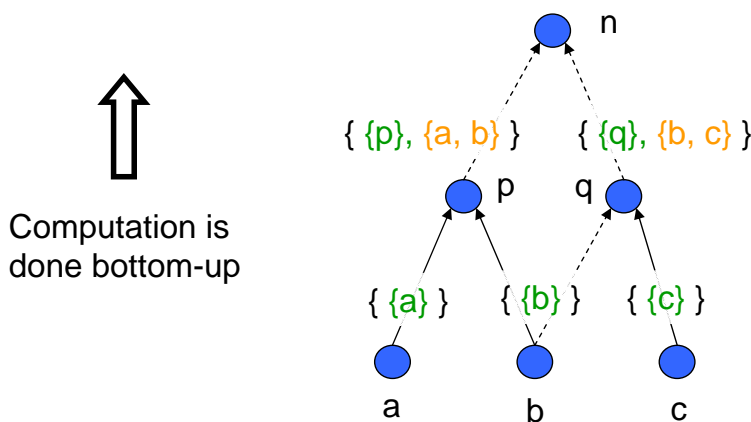
The set $\{p, b, c\}$ is a 3-feasible cut of node n . (It is also a 4-feasible cut.)

k -feasible cuts are important in FPGA mapping because the logic between root n and the cut nodes $\{p, b, c\}$ can be replaced by a k -LUT

15

Cut Computation

$\{\{n\}, \{p, q\}, \{p, b, c\}, \{a, b, q\}, \{a, b, c\}\}$



Computation is done bottom-up

k	Cuts per node
4	6
5	20
6	80
7	150

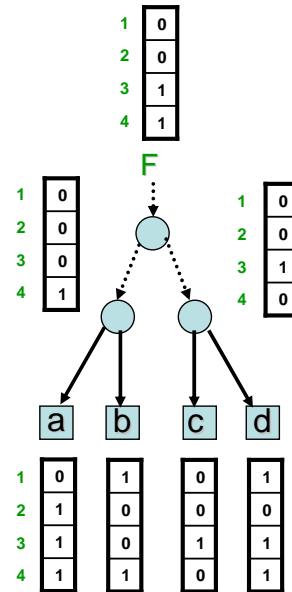
The set of cuts of a node is a 'cross product' of the sets of cuts of its children.
Any cut that is of size greater than k is discarded.

(P. Pan et al, FPGA '98; J. Cong et al, FPGA '99)

16

Bitwise Simulation

- Assign particular (or random) values at the primary inputs
 - Multiple simulation patterns are packed into 32- or 64-bit strings
- Perform bitwise simulation at each node
 - Nodes are ordered in a topological order
- Works well for AIG due to
 - The uniformity of AND-nodes
 - Speed of bitwise simulation
 - Topological ordering of memory used for simulation information



17

Boolean Satisfiability

- Given a CNF formula $\varphi(x)$, **satisfiability problem** is to prove that $\varphi(x) \equiv 0$, or to find a counter-example x' such that $\varphi(x') \equiv 1$
- Why this problem arises?
 - If CNF were a canonical representation (like BDD), it would be trivial to answer this question.
 - But CNF is not canonical. Moreover, CNF can be very redundant, so that a large formula is, in fact, equivalent to 0.
 - Looking for a satisfying assignment can be similar to searching for a needle in the hay-stack.
 - The problem may be even harder, if there is no needle there!

18

Example (Deriving CNF)

CNF

$$(a + b + c)$$

$$(a + b + c')$$

$$(a' + b + c')$$

$$(a + c + d)$$

$$(a' + c + d)$$

$$(a' + c + d')$$

$$(b' + c' + d')$$

$$(b' + c' + d)$$

	ab			
cd	00	01	11	10
00	0	0	0	0
01	0	1	0	0
11	0	0	0	0
10	0	0	0	0

Cube: bcd'

Clause: $b' + c' + d$

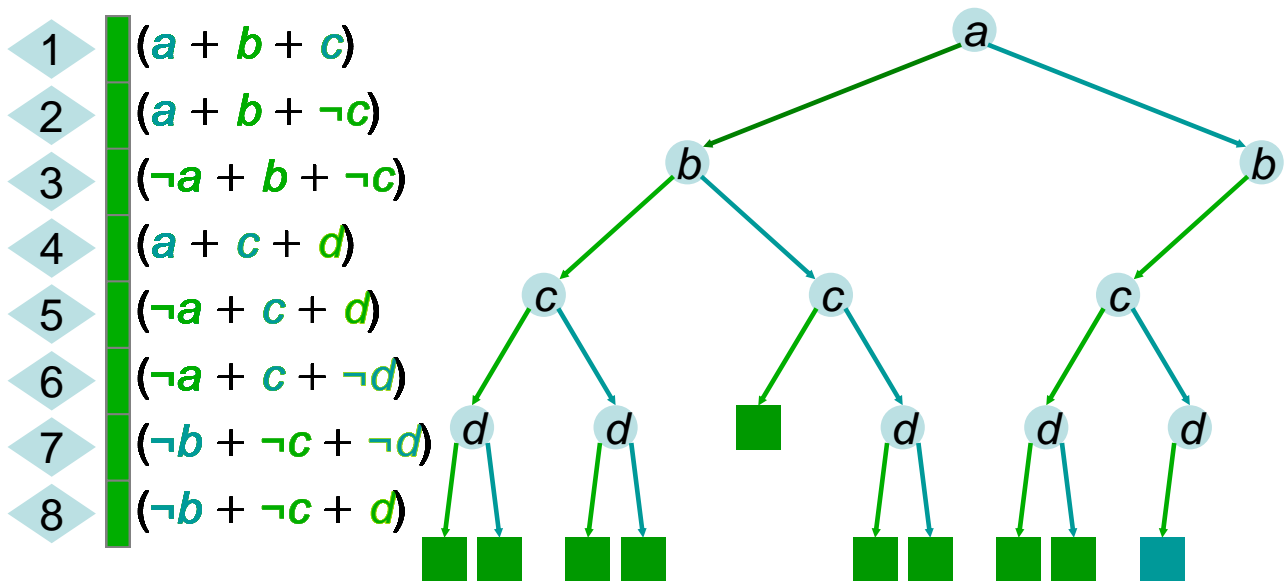
19

SAT Solver

- SAT solver types
 - CNF-based, circuit-based
 - Complete, incomplete
 - DPLL, saturation, etc.
- Applications in EDA
 - Verification
 - Equivalence checking
 - Model checking
 - Synthesis
 - Circuit restructuring
 - Decomposition
 - False path analysis
 - Routing
- A lot of magic is used to build an efficient SAT solver
 - Two literal clause watching
 - Conflict analysis with clause recording
 - Non-chronological backtracking
 - Variable ordering heuristics
 - Random restarts, etc
- The best SAT solver is MiniSAT (<http://minisat.se/>)
 - Efficient (won many competitions)
 - Simple (600 lines of code)
 - Easy to modify and extend
 - Integrated into ABC

20

Example (SAT Solving)



Courtesy Karem Sakallah, University of Michigan

21

(2) And-Inverter Graphs (AIG)

- Definition and examples
- Several simple tricks that make AIGs work
- Sequential AIGs
- Unifying representation
- A typical synthesis application: AIG rewriting

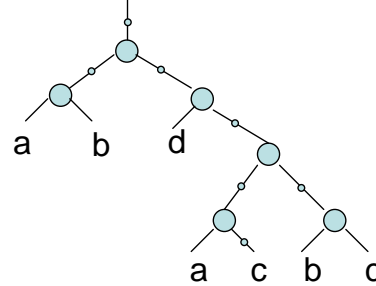
22

AIG Definition and Examples

AIG is a Boolean network composed of two-input ANDs and inverters.

ab \ cd	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

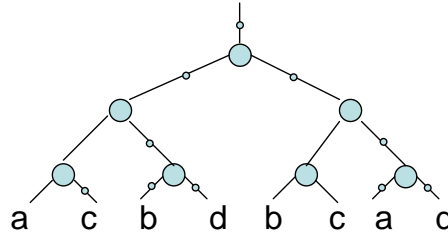
$$F(a,b,c,d) = ab + d(ac' + bc)$$



6 nodes
4 levels

ab \ cd	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

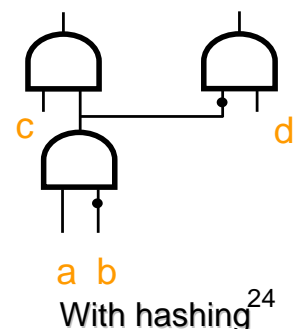
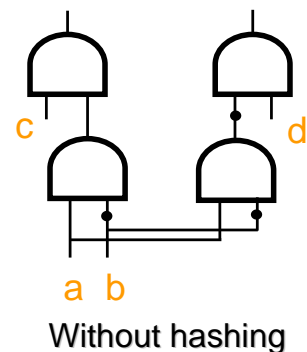
$$F(a,b,c,d) = ac'(b'd) + c(a'd) = ac'(b+d) + bc(a+d)$$



7 nodes
3 levels
23

Three Simple Tricks

- **Structural hashing**
 - Makes sure AIG is stored in a compact form
 - Is applied during AIG construction
 - Propagates constants
 - Makes each node structurally unique
- **Complemented edges**
 - Represents inverters as attributes on the edges
 - Leads to fast, uniform manipulation
 - Does not use memory for inverters
 - Increases logic sharing using DeMorgan's rule
- **Memory allocation**
 - Uses fixed amount of memory for each node
 - Can be done by a simple custom memory manager
 - Even dynamic fanout manipulation is supported!
 - Allocates memory for nodes in a topological order
 - Optimized for traversal in the same topological order
 - Small static memory footprint for many applications
 - Computes fanout information on demand



Sequential AIGs

- Sequential networks have memory elements in addition to logic nodes
 - Memory elements are modeled as D-flip-flops
 - Initial state $\{0,1,x\}$ is assumed to be given
- Several ways of representing sequential AIGs
 - Additional PIs and POs in the combinational AIG
 - Additional register nodes with sequential structural hashing
- Sequential synthesis (in particular, retiming) annotates registers with additional information
 - Takes into account register type and its clock domain

25

AIG: A Unifying Representation

- An underlying data structure for various computations
 - Rewriting, resubstitution, simulation, SAT sweeping, induction, etc are based on the same AIG manager
- A unifying representation for the whole flow
 - Synthesis, mapping, verification use the same data-structure
 - Allows multiple structures to be stored and used for mapping
- The main functional representation in ABC
 - A foundation of new logic synthesis

26

(3) AIG-Based Solutions

- Synthesis
- Mapping
- Verification

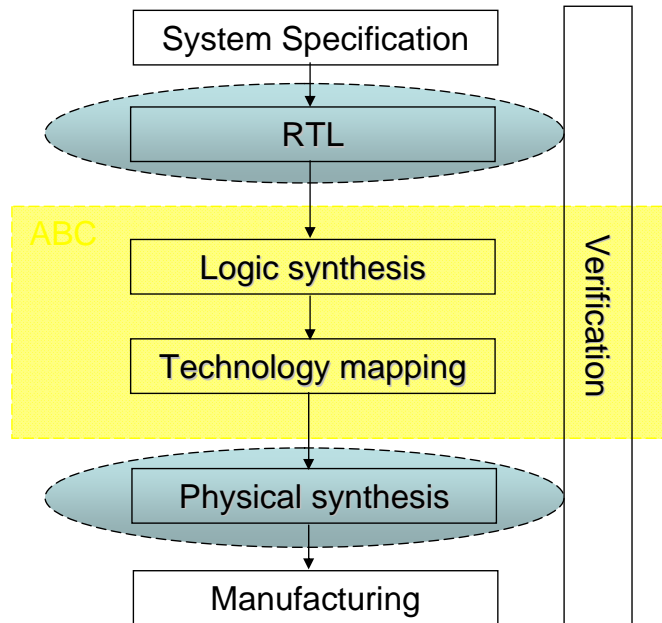
27

What Is Berkeley ABC?

- A system for logic **synthesis** and **verification**
 - Fast
 - Scalable
 - High quality results (industrial strength)
 - Exploits **synergy** between synthesis and verification
- A programming environment
 - Open-source
 - Evolving and improving over time

28

Design Flow



29

Screenshot

```
ex "C:\_projects\abc\_TEST\abc.exe"
UC Berkeley, ABC 1.01 (compiled Aug  3 2008 09:41:23)
abc 01> read wb2\blif\cloud.blif
Warning: The network contains hierarchy.
Hierarchy reader flattened 48190 instances of logic boxes and left 9839 black boxes.
Hierarchy reader converted 9839 instances of blackboxes.
abc 02>
abc 02> ps; st; ps; [redacted] ps; time
cloud      : i/o = 27526/13552  lat = 36862  nd = 92798  edge = 267760  cube = 164666  lev = 23
cloud      : i/o = 27526/13552  lat = 36862  and = 227678 (exor = 9964) (mux = 34186) (pure and =
95228)  lev = 42
cloud      : i/o = 27526/13552  lat = [redacted] nd = [redacted] edge = 236594  aig = 273622  lev = 8
elapsed [redacted] seconds, total: 16.57 seconds
abc 04>
abc 04> read wb2\blif\cloud.blif
Warning: The network contains hierarchy.
Hierarchy reader flattened 48190 instances of logic boxes and left 9839 black boxes.
Hierarchy reader converted 9839 instances of blackboxes.
abc 05>
abc 05> ps; st; zero; [redacted] ps; if -K 6; ps; time
cloud      : i/o = 27526/13552  lat = 36862  nd = 92798  edge = 267760  cube = 164666  lev = 23
cloud      : i/o = 27526/13552  lat = 23944  and = 163840 (exor = 8825) (mux = 25584) (pure and =
60613)  lev = 42
cloud      : i/o = 27526/13552  lat = [redacted] nd = [redacted] edge = 174861  aig = 199632  lev = 8
elapsed [redacted] seconds, total: 33.27 seconds
abc 09>
```

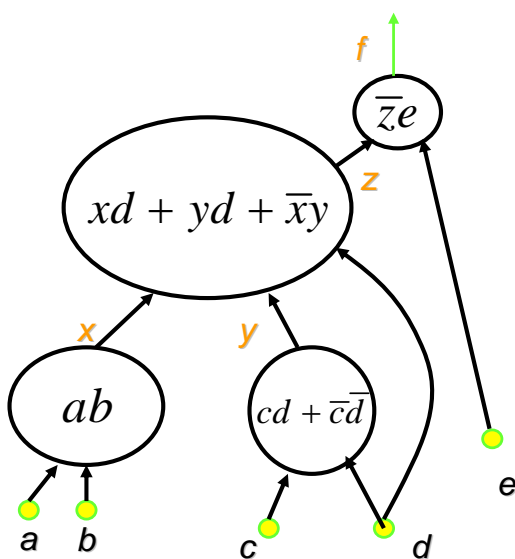
ABC vs. Other Tools

- Industrial
 - + well documented, fewer bugs
 - black-box, push-button, no source code, often expensive
- SIS
 - + traditionally very popular
 - data structures / algorithms outdated, weak sequential synthesis
- VIS
 - + very good implementation of BDD-based verification algorithms
 - not meant for logic synthesis, does not feature the latest SAT-based implementations
- MVSIS
 - + allows for multi-valued and finite-automata manipulation
 - not meant for binary synthesis, lacking recent implementations

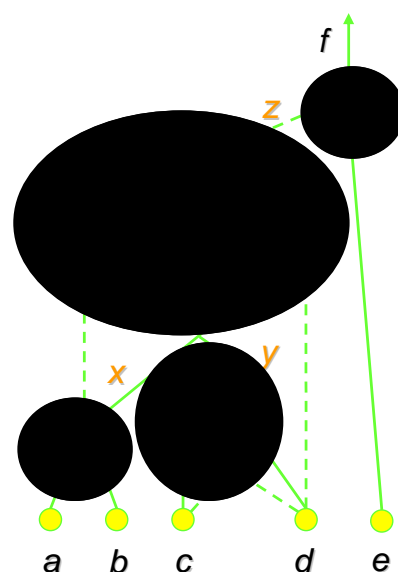
31

How Is ABC Different From SIS?

Boolean network in SIS



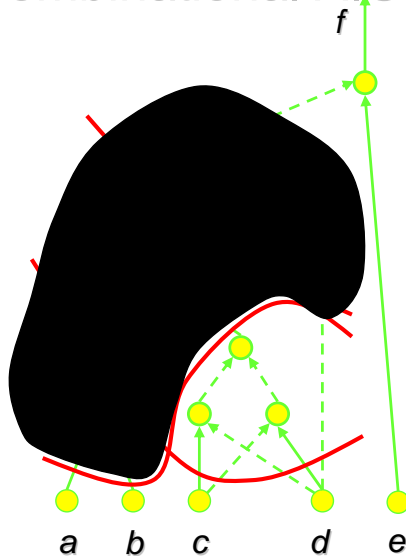
Equivalent AIG in ABC



AIG is a Boolean network of 2-input AND nodes and invertors (dotted lines)³²

One AIG Node – Many Cuts

Combinational AIG



Different cuts for the same node

- Manipulating AIGs in ABC
 - Each node in an AIG has many cuts
 - Each cut is a **different** SIS node
 - No a priori fixed boundaries
- Implies that AIG manipulation with cuts is equivalent to working on **many** Boolean networks at the same time

33

Comparison of Two Syntheses

“Classical” synthesis

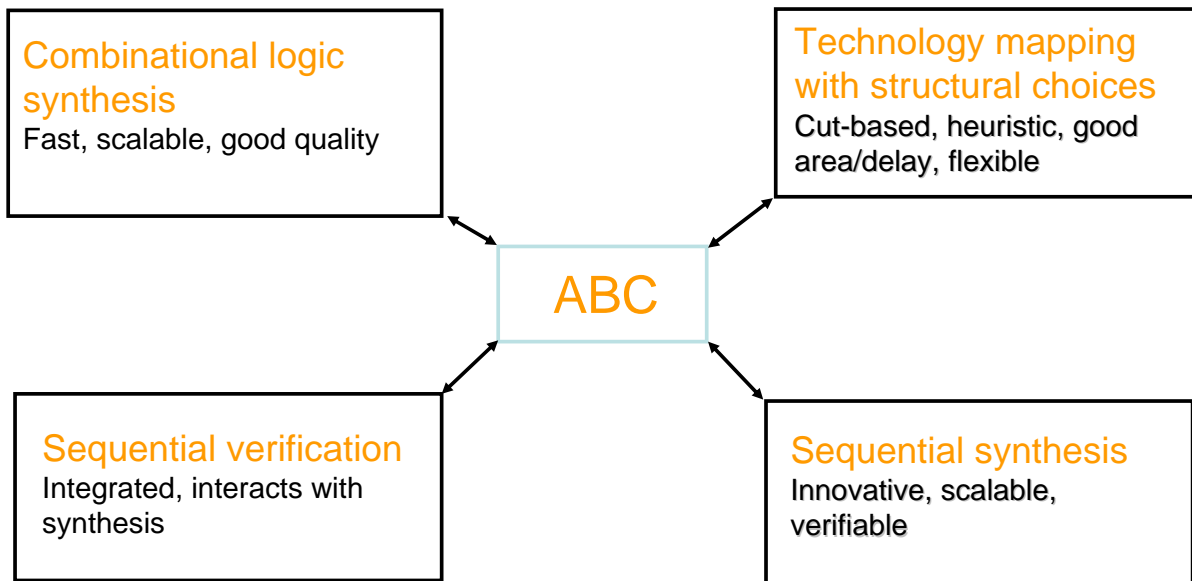
- Boolean network
- Network manipulation (algebraic)
 - Elimination
 - Factoring/Decomposition
 - Speedup
- Node minimization
 - Espresso
 - Don't cares computed using BDDs
 - Resubstitution
- Technology mapping
 - Tree based

ABC “contemporary” synthesis

- AIG network
- DAG-aware AIG rewriting (Boolean)
 - Several related algorithms
 - Rewriting
 - Refactoring
 - Balancing
 - Speedup
- Node minimization
 - Boolean decomposition
 - Don't cares computed using simulation and SAT
 - Resubstitution with don't cares
- Technology mapping
 - Cut based with choice nodes

34

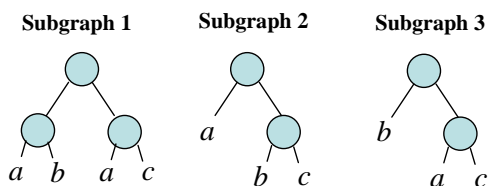
Existing Capabilities (2005-2008)



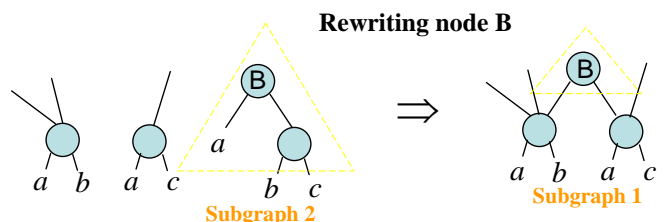
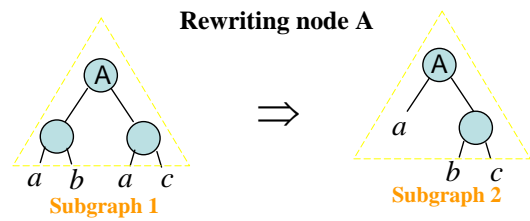
Combinational Synthesis

- **AIG rewriting** minimizes the number of AIG nodes without increasing the number of AIG levels

- Pre-computing AIG subgraphs
 - Consider function $f = abc$



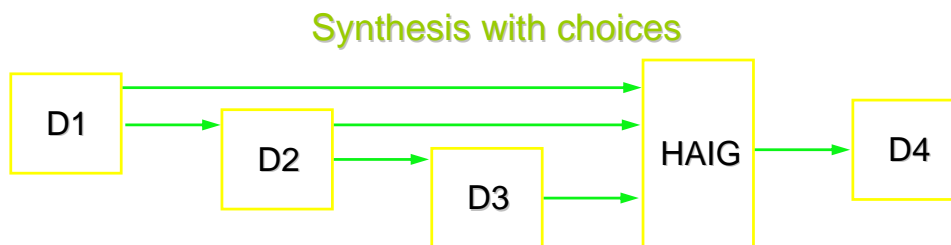
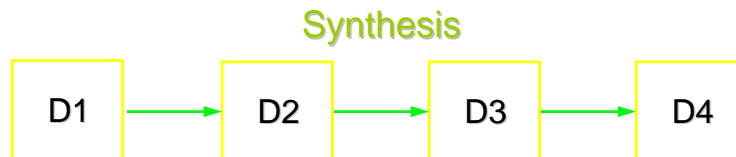
Rewriting AIG subgraphs



In both cases 1 node is saved

AIG-Based Solutions (Synthesis)

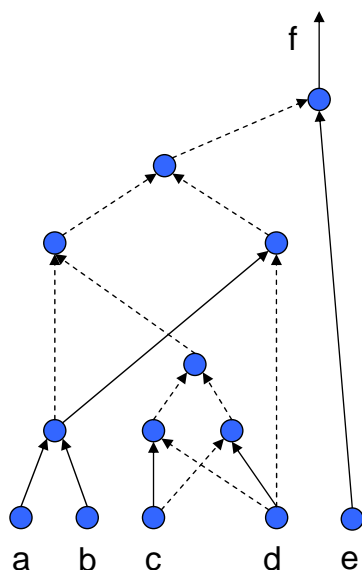
- Restructures AIG or logic network by the following transforms
 - Algebraic balancing
 - Rewriting/refactoring/redecomposition
 - Resubstitution
 - Minimization with don't-cares, etc



37

AIG-Based Solutions (Mapping)

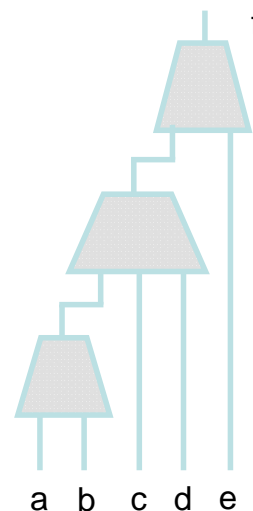
Input: A Boolean network
(And-Inverter Graph)



The subject graph

Output: A netlist of K -LUTs implementing
AIG and optimizing some cost function

Technology
Mapping



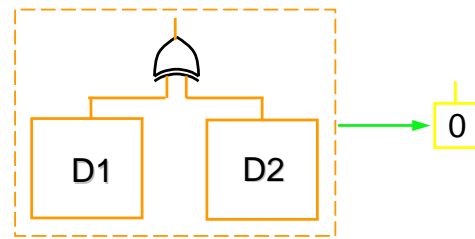
The mapped netlist

38

Formal Verification

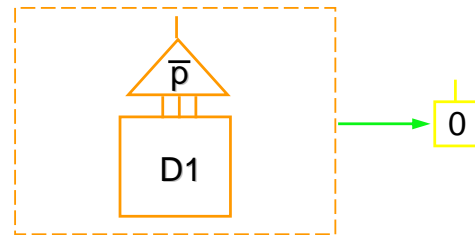
- **Equivalence checking**
 - Takes two designs and makes a miter (AIG)
- **Model checking *safety* properties**
 - Takes design and property and makes a miter (AIG)

Equivalence checking



The goals are the same: to transform AIG until the output is proved constant 0

Property checking



Breaking News: ABC won a model checking competition at CAV in August 2010

39

Model Checking Competition

A screenshot of a Mozilla Firefox browser window displaying the results of the Hardware Model Checking Competition 2010. The browser's address bar shows the URL <http://fmv.jku.at/hwmc10/results.html>. The page content includes the HWMCC'10 logo, a navigation menu with links for Benchmarks, Organizers, Results, and Rules, and a main section titled "Results".

Results

The results have been presented at [HVVW'10](#) with the following [slides](#).

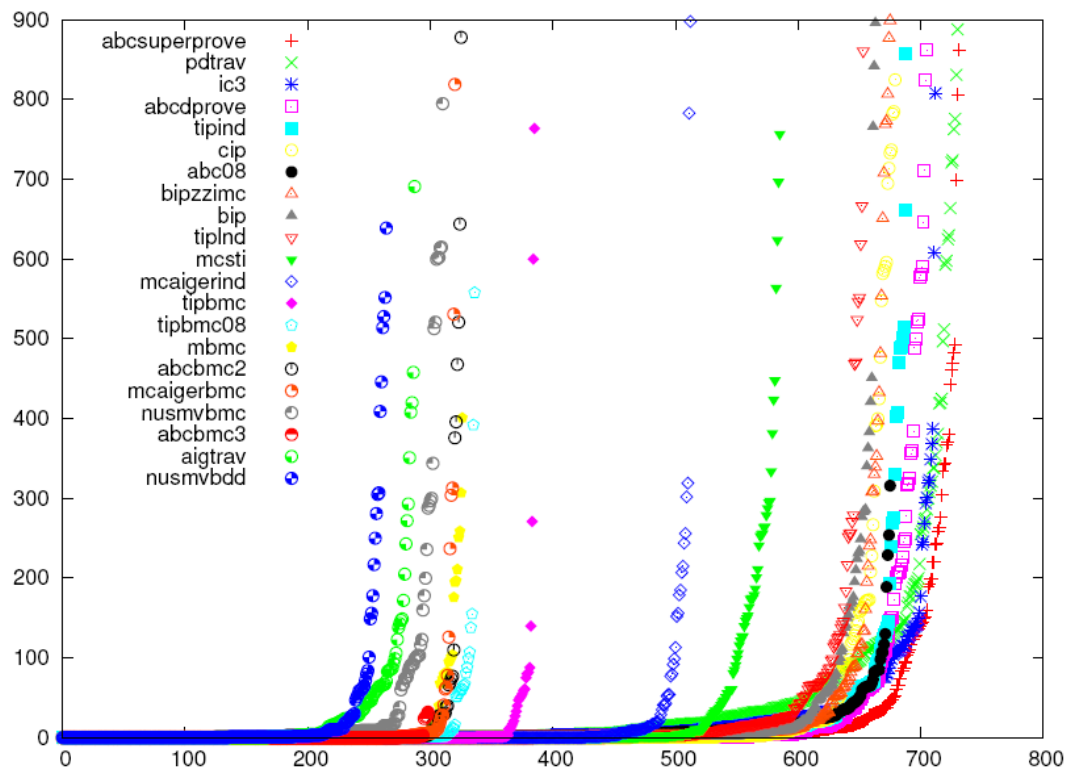
The winners are:

ALL	abcsuperprove	University of California, Berkeley
SAT	abcbmc2	University of California, Berkeley
UNSAT	pdtrav	Politecnico di Torino

For more information on the set-up please consult the [slides](#) of the [HVVW'10](#).

More details can be found in the following files: [table.xls](#), [table.csv](#), [details.bt](#), and [checked.txt](#).

40



Armin Biere – FMV – JKU Linz

Further Reading: ABC Tutorial

- For more information, please refer to
- R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", Proc. CAV'10, Springer, LNCS 6174, pp. 24-40.
- http://www.eecs.berkeley.edu/~alanmi/publications/2010/cav10_abc.pdf

Summary

- Introduced problems in logic synthesis
 - Representations and computations
- Described And-Inverter Graphs (AIGs)
 - The foundation of innovative synthesis
- Overviewed AIG-based solutions
 - Synthesis, mapping, verification
- Introduced ABC
 - Differences, fundamentals, programming

43

Assignment: Using ABC

- Using BLIF manual
<http://www.eecs.berkeley.edu/~alanmi/publications/other/blif.pdf>
create a BLIF file representing a 2-bit full-adder
- Perform the following sequence:
 - read the file into ABC (command "read")
 - check statistics (command "print_stats")
 - visualize the network structure (command "show")
 - convert to AIG (command "strash")
 - visualize the AIG (command "show")
 - convert to BDD (command "collapse")
 - visualize the BDD (command "show_bdd")

44

Assignment: Programming ABC

- Write a procedure in ABC environment to iterate over the objects of the network and list each object ID number and type on a separate line. Integrate this procedure into ABC, so that running command "test" would invoke your code, and print the result. Compare the print-out of the new command "test" with the result of command "show" for the full-adder example above
- Comment 1: For commands "show" and "show_bdd" to work, please download the binary of software "dot" from GraphVis webpage and put it in the same directory as the ABC binary or anywhere else in the path: <http://www.graphviz.org>
- Comment 2: Make sure GSview and Ghostscript are installed on your computer. <http://pages.cs.wisc.edu/~ghost/gsview/>

45

Programming Help

- Example of code to iterate over the objects

```
void Abc_NtkCleanCopy( Abc_Ntk_t * pNtk )
{
    Abc_Obj_t * pObj;
    int i;
    Abc_NtkForEachObj( pNtk, pObj, i )
        pObj->pCopy = NULL;
}
```
- Example of code to create new command "test"
Call the new procedure (say, Abc_NtkPrintObjs) from Abc_CommandTest() in file "abc\src\base\abc\abc.c"

```
Abc_NtkPrintObjs( pNtk );
```

46