

Introduction to ABC: An Open-Source Synthesis and Verification CAD Tool for FPGAs and ASICs

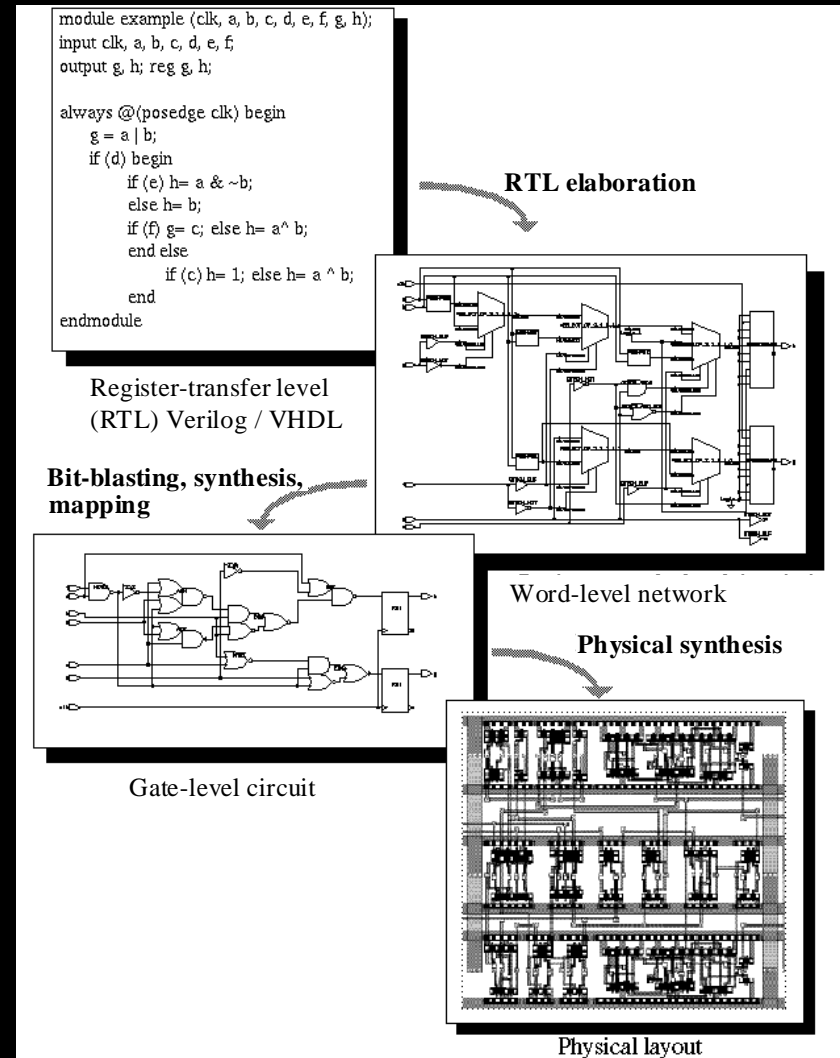
Alan Mishchenko

University of California, Berkeley

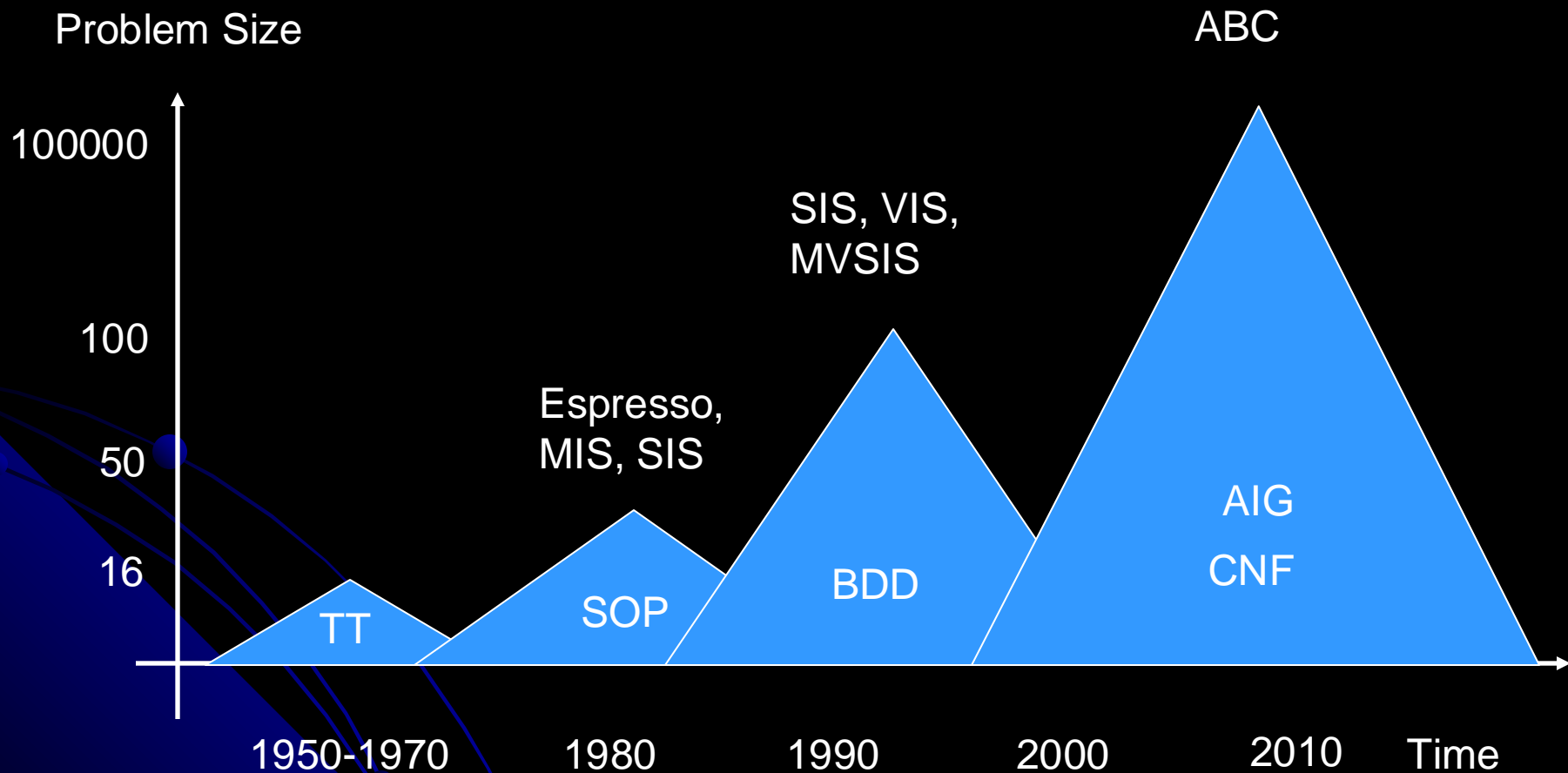


Synthesis/Verification in a Design Flow

- Stages of the design flow
 - System specification
 - Design architecture exploration
 - High-level synthesis
 - RTL elaboration
 - Word-level transformations
 - Bit-level logic synthesis/mapping
 - Physical synthesis
 - Mask generation
 - Fabrication
 - Packaging and testing
- Verification is typically needed
 - Between high-level description and a gate-level circuit
 - Between different stages of synthesis and mapping



Academic Synthesis/Verification Tools: Historical Perspective



Outline

- Basic level
 - Boolean calculator and visualizer
 - Standard commands and scripts
- Advanced level
 - Key packages and data-structures
 - Ways to improve runtime and memory usage
- Hits and misses
- Future research directions

Outline

- Basic level
 - Boolean calculator and visualizer
 - Standard commands and scripts
- Advanced level
 - Key packages and data-structures
 - Ways to improve runtime and memory usage
- Hits and misses
- Future research directions

Boolean Calculator

- Read/generate small functions/networks
 - Automatically extracted or manually specified
- Compute basic functional properties
 - Symmetries, decomposability, unateness, etc
- Transform them in various ways
 - Minimize SOP, reorder BDD, extract kernels, etc
- Visualization
 - Use text output, dot / GSView, etc

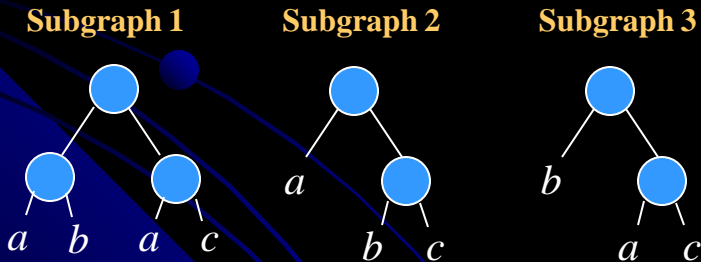
Standard Commands/Scripts

- Technology independent synthesis
- Logic synthesis for PLAs
- Technology mapping for standard cells
- Technology mapping for LUTs
- Sequential synthesis
- Verification

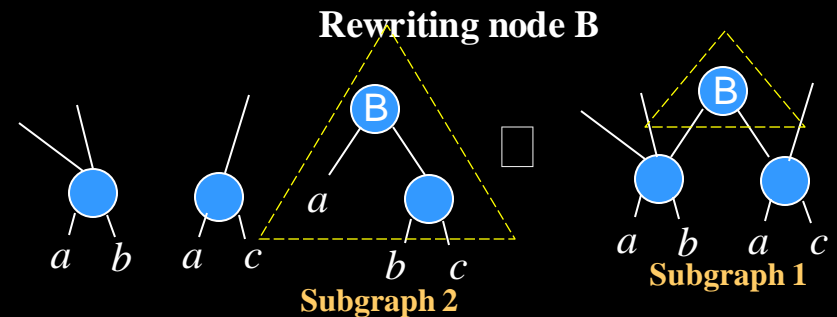
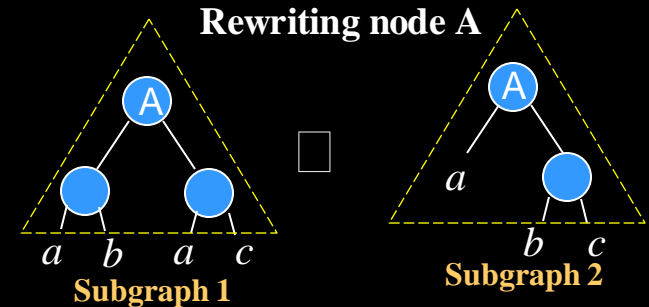
AIG Rewriting

- **AIG rewriting** has the goal of minimizing the number of AIG nodes

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



Rewriting AIG subgraphs



In both cases 1 node is saved

Technology Independent Synthesis

- AIG rewriting for area
 - Scripts *drwsat*, *compress2rs*
- AIG rewriting for delay
 - Scripts *dc2*, *resyn2*
 - Scripts *&syn2*, *&synch2*
- High-effort delay optimization
 - Perform SOP balancing (*st; if -g -K 6*)
 - Follow up with area-recovery (*resyn2*) and technology mapping (*map*, *amap*, *if*)
 - Iterate, if needed

Logic Synthesis for PLAs

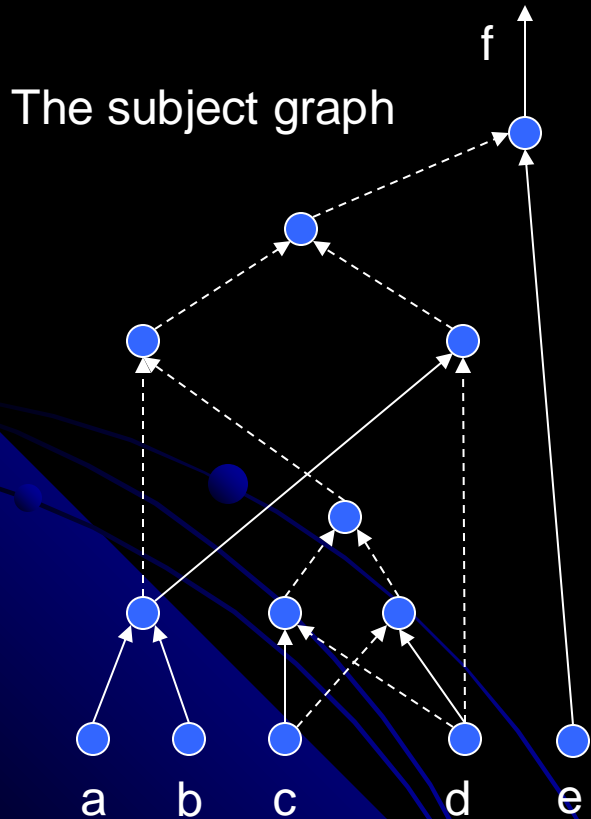
- Enter PLA (.type fd) into ABC using *read*
- Perform logic sharing extraction using *fx*
 - If *fx* is complaining that individual covers are not prime and irredundant, try *bdd; sop; fx*
- After *fx*, convert shared logic into AIG and continue AIG-based synthesis and mapping if needed
- Consider using high-effort synthesis with don't-cares
 - First map into 6-LUTs (*if -K 6; ps*), optimize (*mfs2*), synthesize with choices (*st; dch*) and map into 6-LUTs (*if -K 6; ps*)
 - Iterate until no improvement, then remap into target technology
- To find description of PLA format, google for “Espresso PLA format”, for example:
 - <http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/espresso.5.html>

```
.i 2
.o 2
00 00
01 01
10 01
11 10
.e
```

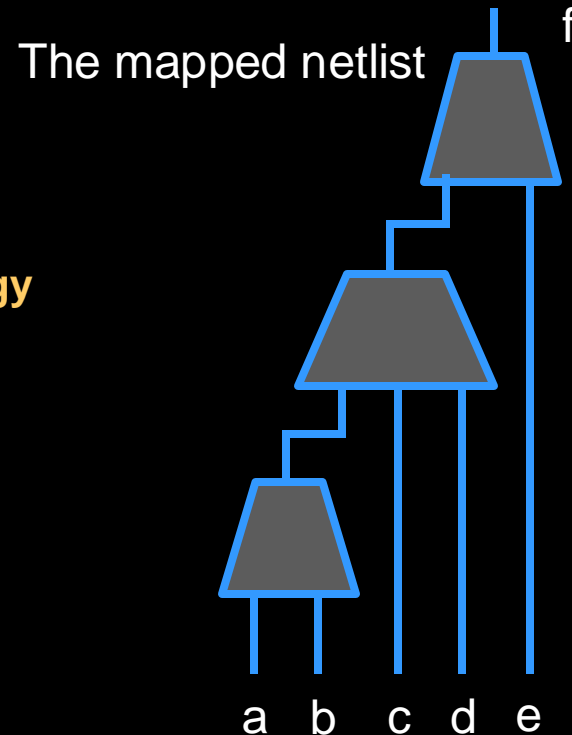
LUT Mapping

Input: A Boolean network
(And-Inverter Graph)

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



**Technology
Mapping**



Technology Mapping for SCs

- Read library using
 - *read_genlib* (for libraries in GENLIB format)
 - *read_liberty* (for libraries in Liberty format)
- For standard-cells
 - *map*: Boolean matching, delay-oriented, cells up to 5 inputs
 - *amap*: structural mapping, area-oriented, cells up to 15 inputs
- If Liberty library is used, run *topo* followed by
 - *stime* (accurate timing analysis)
 - *buffer* (buffering)
 - *upsize; dnsize* (gate sizing)
- Structural choices are an important way of improving mapping (both area and delay)
 - Run *st; dch* before calling *map* or *amap*

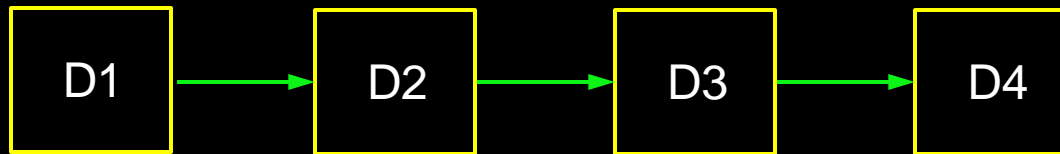
Technology Mapping for LUTs

- It is suggested to use mapper *if -K <num>*
 - For area-oriented mapping, try *if -a*
 - For delay-oriented mapping, try delay-oriented AIG-based synthesis with structural choices
- Structural choices are an important way of improving mapping (both area and delay)
 - Run *st; dch* before calling *if*

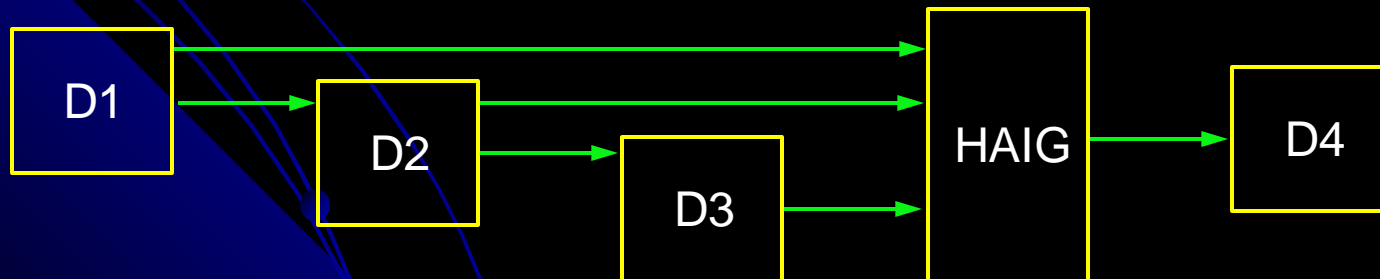
Synthesis With Structural Choices

- Traditional synthesis produces one “optimized” network
- Synthesis with choices produces several networks
 - These can be different snapshot of the same synthesis flow
 - These can be results of synthesizing the design with different options
 - For example, area-oriented and delay-oriented scripts

Synthesis



Synthesis with structural choices



Sequential Synthesis

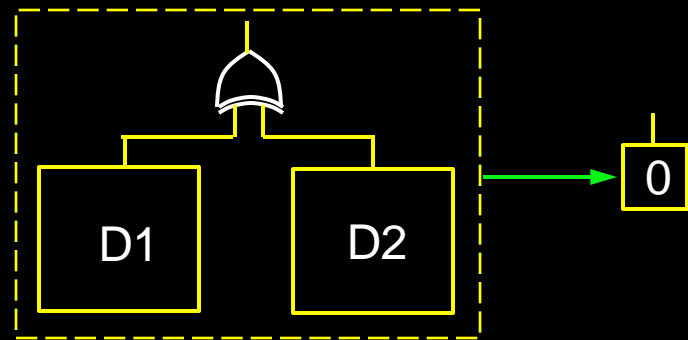
- Uses reachable state information to further improve the quality of results
 - Reachable states are often approximated
- Types of AIG-based sequential synthesis
 - Retiming (*retime*, *dretime*, etc)
 - Detecting and merging sequential equivalences (*lcorr*, *scorr*, *&scorr*, etc)
- Negative experiences
 - Sequential redundancy removal is often hard
 - Using sequential don't-cares in combinational synthesis typically gives a very small improvement

Formal Verification

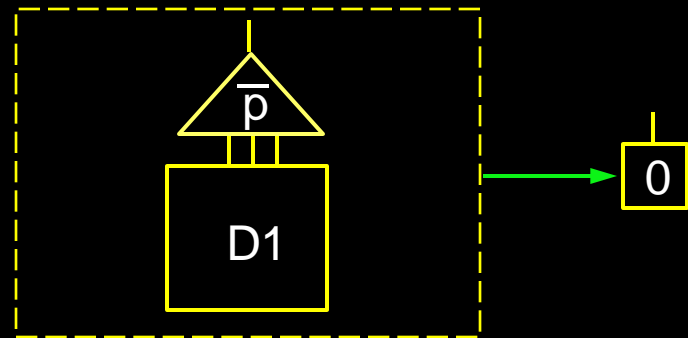
- **Equivalence checking**
 - Takes two designs and makes a miter (AIG)
- **Model checking**
 - Takes design and property and makes a miter (AIG)

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

Equivalence checking



Property checking



Verification Commands

- Combinational verification
 - *r <file1>; cec <file2>* (small/medium circuits)
 - *&r <file1.aig>; &cec <file2.aig>* (large circuits)
- Sequential verification
 - *r <file1>; dsec <file2>*
- Running *cec* or *dsec* any time in a synthesis flow compares the current network against its spec
 - The spec is the circuit obtained from the original file
- Verification and synthesis are closely related and should be co-developed

Outline

- Basic level
 - Boolean calculator and visualizer
 - Standard commands and scripts
- **Advanced level**
 - **Key packages and data-structures**
 - **Ways to improve runtime and memory usage**
- Future research directions

Key Packages

- AIG package
- Technology-independent synthesis
- Technology mappers
- SAT solver
- Combinational equivalence checking
- Sequential synthesis
- Sequential verification engine IC3/PDR

Key Packages

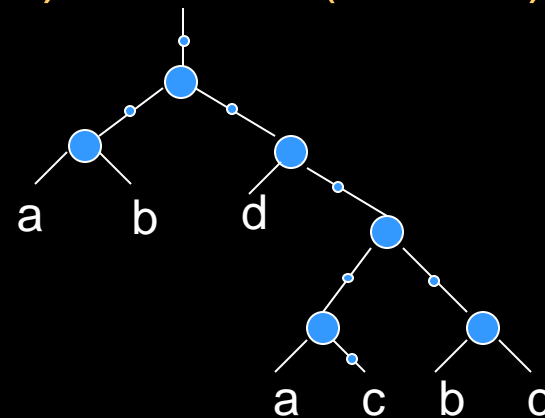
- AIG package
- Technology-independent synthesis
- Technology mappers
- SAT solver
- Combinational equivalence checking
- Sequential synthesis
- Sequential verification engine IC3/PDR

And-Inverter Graph (AIG)

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

cd^{ab}	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

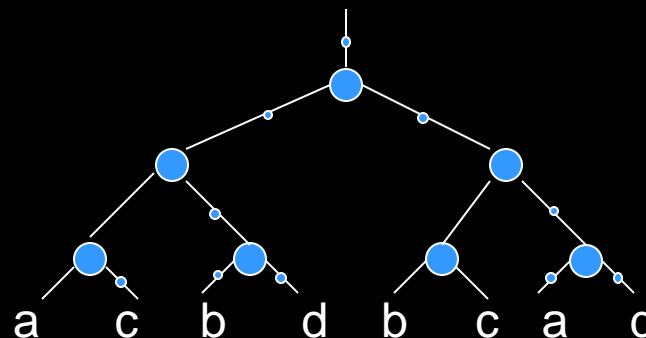
$$F(a,b,c,d) = ab + d(a!c+bc)$$



6 nodes
4 levels

cd^{ab}	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

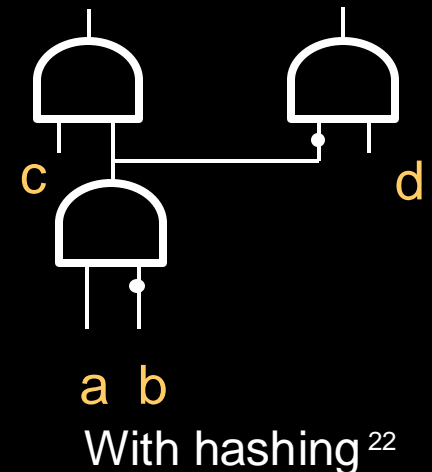
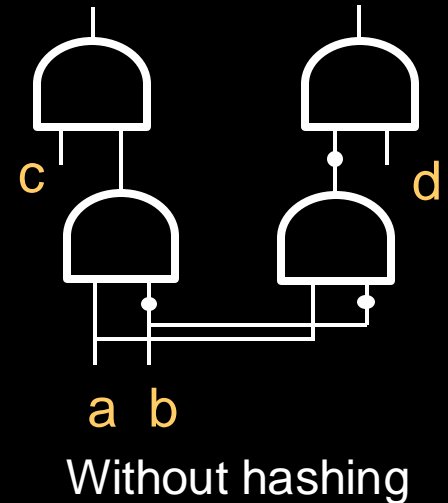
$$F(a,b,c,d) = a!c(b+d) + bc(a+d)$$



7 nodes
3 levels

Components of Efficient AIG Package

- **Structural hashing**
 - Leads to a compact representation
 - 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 custom memory manager
 - Even dynamic fanout can be implemented this way
 - Allocates memory for nodes in a topological order
 - Optimized for traversal using this topological order
 - Small static memory footprint for many applications
 - Computes fanout information on demand



“Minimalistic” AIG Package

- Designed to minimize memory requirements
 - Baseline: 8 bytes/node for AIGs (works up to 2 billion nodes)
 - Structural hashing: +8 bytes/node
 - Logic level information: +4 bytes/node
 - Simulation information: +8 bytes/node for 64 patterns
- Each node attribute is stored in a separate array
 - No “Aig_Node” struct
 - Attributes are allocated and deallocated on demand
 - Helps improve locality of computation
 - Very useful to large AIG (100M nodes and more)
- Maintains minimum memory footprint for basic tasks, while allowing the AIG package to have several optional built-in features
 - Structural hashing
 - Bit-parallel simulation
 - Circuit-based SAT solving

Outline

- Basic level
 - Boolean calculator and visualizer
 - Standard commands and scripts
- Advanced level
 - Key packages and data-structures
 - Ways to improve runtime and memory usage
- Hits and misses
- Future research directions

ABC Hits

- It is based on cutting-edge research ideas
- It offers a low-cost and often competitive implementation of fundamental algorithms
 - AIG rewriting, tech-mapping, SAT sweeping, retiming, equivalence checking, etc
- It tends to be fast and low-memory
- It is reliable (if we use it in a known way)
- It is actively developed and supported

ABC Misses

- Inadequate Verilog parser
- Does not natively support much of the “industrial stuff” (complex flops, multiple clocks, memories, design constraints, etc)
 - requires elaborate workarounds to be useful
- Poor documentation
- A lot of redundant source code

Lessons: Front-End and Back-End

- Having a variety of formats is useful, but...
- Reading and writing Verilog is a must!
 - Yosys is used as an external front-end
- Need well-documented APIs for integrating with external tools
 - This has been addressed to some extent

Lessons: Optimization Flow

- AIG is a good unifying data-structure
 - Do not hesitate to base computations on AIGs
- Need parametrizable optimizers
 - Rather than having optimizations geared to a specific representation (AIG/MIG/XMIG/etc)
- Need one generic cut-based tech-mapper for all technologies (gates, LUTs, etc)
- **Need to support the “industrial stuff”!**

Lessons: Data Structures

- Develop a clean minimalistic data-structure for each package (conversions between data-structures are easy and fast)
- Reduce memory used for large data-structures and runtime will be reduced
 - true about AIG, logic network, hierarchical netlist
- Whenever possible, use 32-bit integers
 - a MiniSAT-like SAT solver is a good example

Lessons: Programming

- Strive for maintainability
 - Minimize dependency between packages
- Strive for reproducibility
 - Implement your own floating point number
- Strive for thread-safety
 - Have no global and static variables
- Spend time to build a set of handy tools

Outline

- Basic level
 - Boolean calculator and visualizer
 - Standard commands and scripts
- Advanced level
 - Key packages and data-structures
 - Ways to improve runtime and memory usage
- Hits and misses
- Future research directions

Research Directions

- Ongoing
 - Deep integration of simulation and SAT
 - Word-level optimizations (e.g. memory abstraction)
 - Logic synthesis for machine learning
 - As opposed to machine learning for logic synthesis!

Industrial Supporters (since 2005)

- CAD tool companies
 - Synopsys, Mentor (Siemens), Cadence, Verific, Magma (Synopsys), Atrenta (Synopsys), Jasper (Cadence), Oasys (Mentor (Siemens))
- FPGA companies
 - Xilinx (AMD), Altera (Intel), Synplicity (Synopsys), Actel (Microsemi (Microchip)), Abound Logic (Lattice)
- Design companies
 - IBM, Intel
- Grants from federal and industrial funding agencies
 - NFS, NSA, SRC

Contributors to ABC

- Fabio Somenzi (U Colorado, Boulder) - **BDD package CUDD**
- Niklas Sorensson, Niklas Een (Chalmers U, Sweden) - **MiniSAT v. 1.4 (2005)**
- Gilles Audemard, Laurent Simon (U Artois, U Paris Sud, France) - **Glucose 3.0**

- Hadi Katebi, Igor Markov (U Michigan) - **Boolean matching for CEC**
- Jake Nasikovsky - **Fast truth table manipulation**
- Wenlong Yang (Fudan U, China) - **Lazy man's synthesis**
- Zyad Hassan (U Colorado, Boulder) - **Improved generalization in IC3/PDR**
- Augusto Neutzling, Jody Matos, Andre Reis (UFRGS, Brazil) - **Technology mapping into threshold functions**
- Mayler Martins, Vinicius Callegaro, Andre Reis (UFRGS, Brazil) – **Boolean decomposition using read-polarity-once (RPO) function**
- Mathias Soeken, EPFL - **Exact logic synthesis**
- Ana Petkovska, EPFL – **Hierarchical NPN matching**
- Bruno Schmitt (UFRGS / EPFL) - **Fast-extract with cube hashing**
- Xuegong Zhou, Lingli Wang (Fudan U, China) - **NPN classification**
- Yukio Miyasaka, Masahiro Fujita (U Tokyo, Japan) - **Custom BDD package for multiplier verification**
- Siang-Yun Lee, Roland Jiang (NTU, Taiwan) - **Dumping libraries of minimum circuits for functions up to five input variables**
- He-Teng Zhang (NTU, Taiwan) – **Circuit-based SAT solver, enhanced SAT sweeper**

ABC Resources

- Source code
 - <https://github.com/berkeley-abc/abc>
- “Getting started with ABC”, a tutorial by Ana Petkovska
 - https://www.dropbox.com/s/qrl9svlf0ylxy8p/ABC_GettingStarted.pdf
- An overview paper: R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", Proc. CAV'10.
 - http://www.eecs.berkeley.edu/~alanmi/publications/2010/cav10_abc.pdf
- Windows binary
 - <http://www.eecs.berkeley.edu/~alanmi/abc/abc.exe>
 - <http://www.eecs.berkeley.edu/~alanmi/abc/abc.rc>

Conclusions

- If you have patience and time to figure it out, ABC can be useful
- Do not hesitate to contact me if you have questions or ideas
- Consider contributing something that could be helpful for others, for example
 - the code used in your paper
 - your course project
 - etc