

Boolean Logic Synthesis

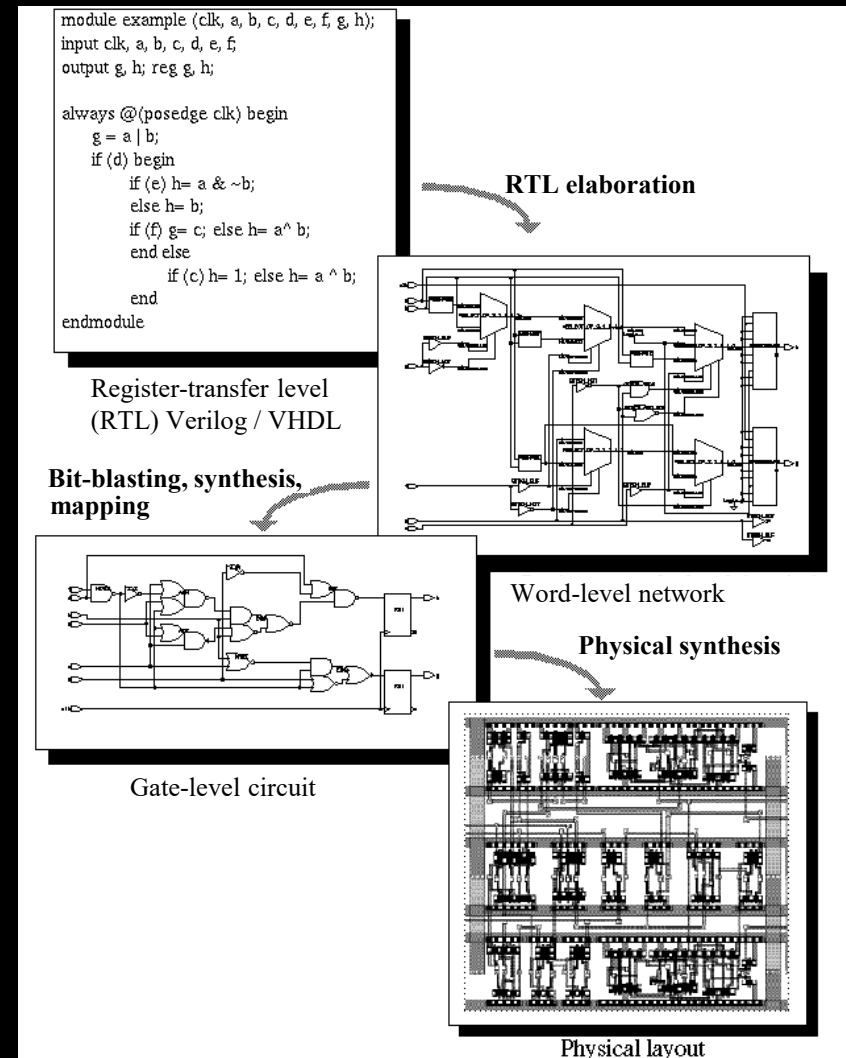
Alan Mishchenko

UC Berkeley



Logic Synthesis 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
- Engineering Change Orders (ECOs)
 - Fixing bugs and adding small features in the later stages of the design flow



Presentation Overview

- Logic synthesis
- Traditional logic synthesis
- Boolean logic synthesis
- Resubstitution formulation and classification
- Resubstitution methods
 - Small scale (usingunate divisors)
 - Large scale (using support minimization)
 - Multi-output exact (using SAT solving)
- Conclusions

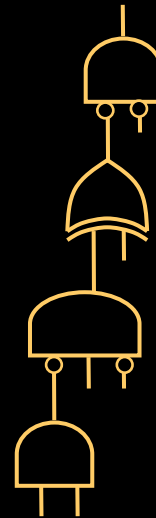
Logic Synthesis: Definition

- Given a function, derive (synthesize) a circuit
- Given a circuit, transform (optimize) it

Sum of products:

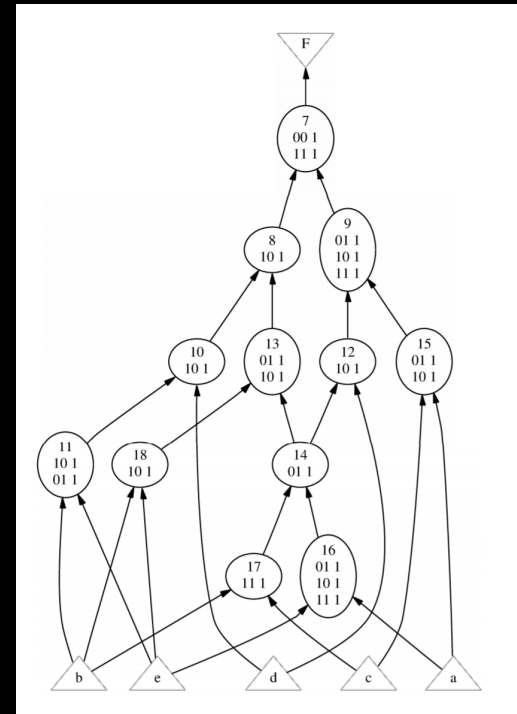
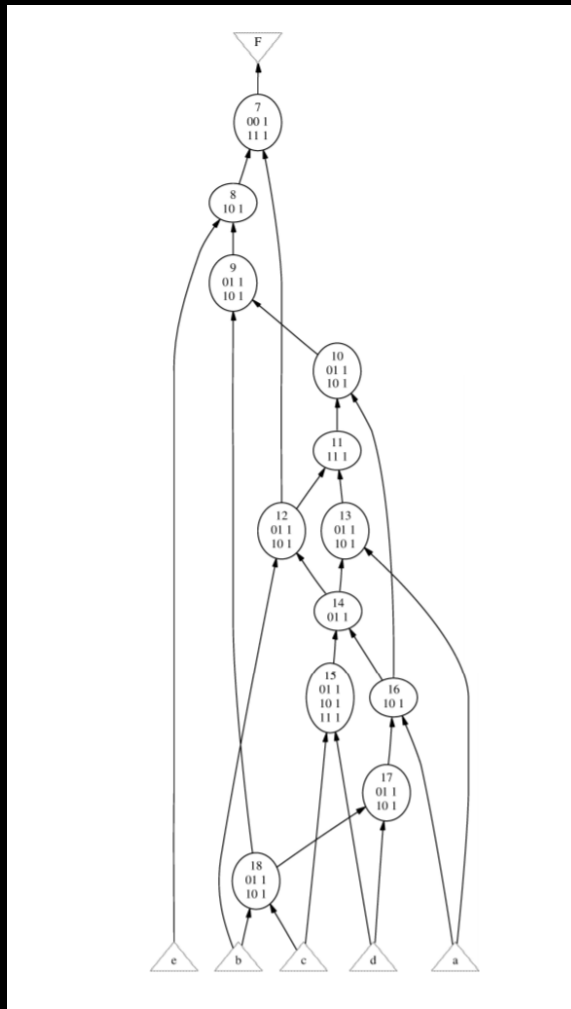
```
.i 6  
.o 1  
0-1--0 1  
-11--0 1  
100-00 1  
1000-0 1  
--1110 1  
.e
```

Circuit:

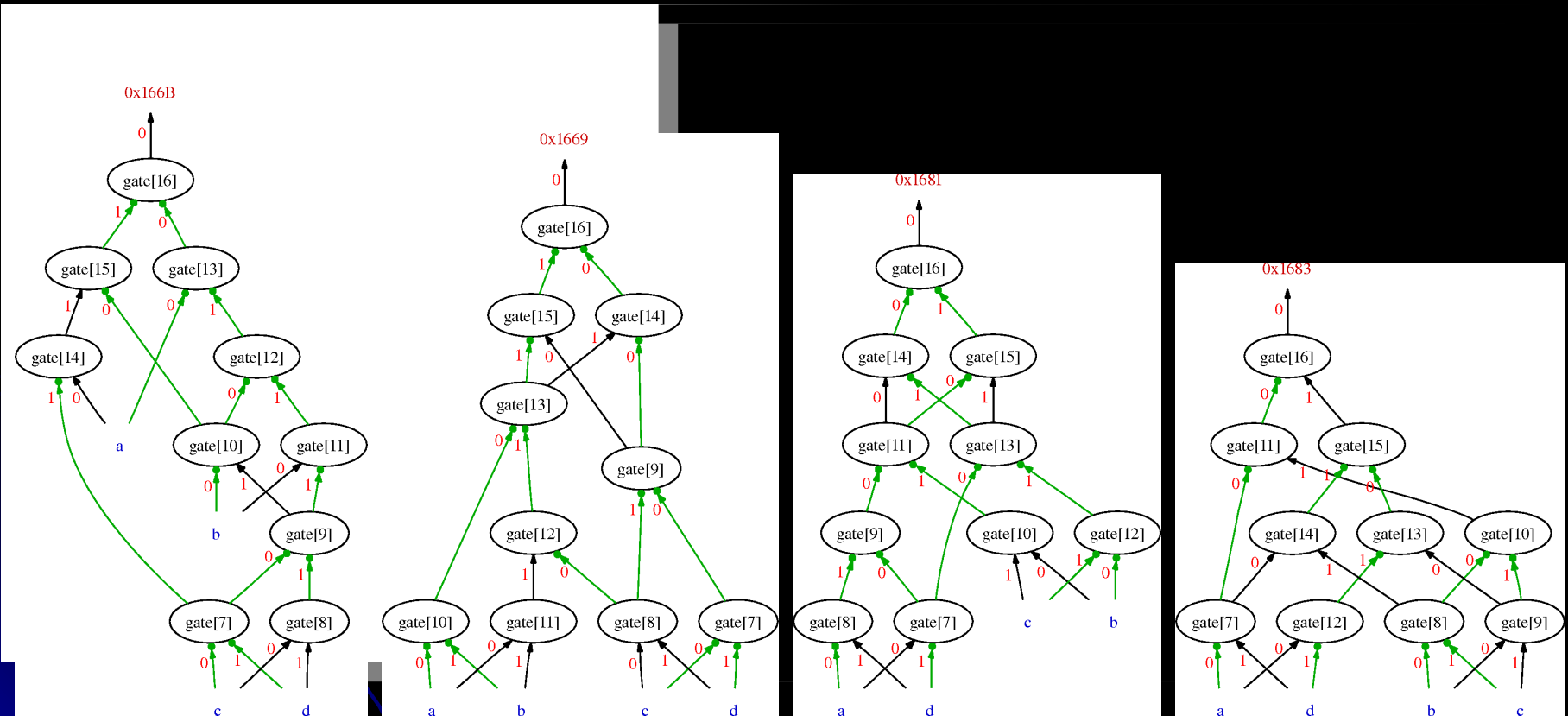


Two Different 12-Node Min-Circuits for The Most Complex 5-Function: 169ae443

Please note that XORs are used.



Four 4-Functions That Require 10 AND Nodes (XORs not used)



This slide is borrowed from presentation “Practical SAT” (2007) by Niklas Een

Boolean Synthesis Terminology

- **Boolean value**: 0 or 1
- **Boolean variable**: $x \in \{0,1\}$
 - A set of n Boolean variables, e.g. $n=4$, $\{a,b,c,d\}$
- **Minterm**: a complete assignment of a set of n Boolean variables, e.g. 0011 for vars $\{a,b,c,d\}$
 - The total number of minterms = 2^n
- **Boolean function**: a complete assignments of all 2^n minterms in terms of a set of variables
 - The total number of functions = 2^{2^n}
- **Truth table**: a simple representation of Boolean function, e.g. 1111 0001 0001 0001 = 0xF111
- **Function with don't-cares**: Output value $\in \{0,1, -\}$
- **Boolean relation**: Any subset of output minterms

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

Traditional Logic Synthesis

- SOP minimization
- Logic network representation
- Algebraic factoring
 - fast_extract algorithm
- Don't-care-based optimization

SOP Minimization

- SOP = Sum-of-Products
 - For example: $F = ab + cd + abc$
- Given an SOP, minimize the number of products
 - For example: $F = ab + cd + abc = ab + cd$
- SOP minimization is important because traditional logic synthesis uses SOPs for factoring and node minimization
- Tools:
 - MINI (IBM, 1974), by S. J. Hong, R. G. Cain, D. L. Ostapko
 - Espresso-MV (IBM / UC Berkeley, 1986) by R. Brayton, R. Rudell

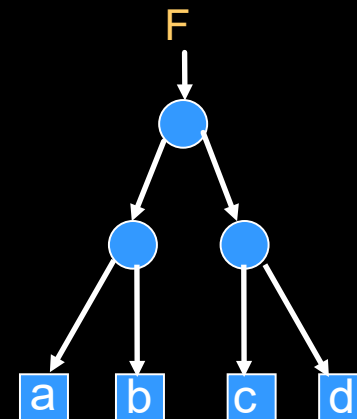
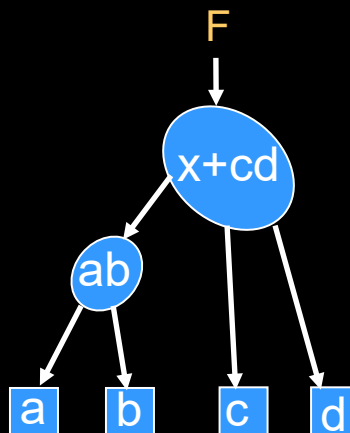
Logic Network Representation

- In traditional logic synthesis, netlists are represented as DAGs whose nodes can have arbitrary logic functions
 - Similar to standard-cell library, with any library gates
- This representation is general but not well-suited for implementing fast/scalable transformations
 - It has been shown that AIGs work better than logic networks

Logic network

And-Inverter Graph (AIG)

$$F = ab + cd$$



Algebraic Factoring

- SOP is a two-level (AND-OR) circuit
- In many applications, a multi-level circuit is needed
- Algebraic factoring converts SOP into a multi-level circuit
- Why it is called algebraic?
 - Because factoring of integers and polynomials is similar
 - $48 = 2 * 2 * 2 * 2 * 3$
 - $a^2 - b^2 = (a + b)*(a - b)$
- The result of factoring of an SOP is not unique
 - For example: $F = ab + ac + bc = a(b+c) + bc = b(a+c) + ac$

Algorithm *fast_extract*

- Algorithm *fast_extract* (J. Vasudevamurthy and J. Rajski, ICCAD'90) is used for algebraic factoring in many practical applications
- It considers all two-cube divisors and single-cube two-literal divisors and their complement
 - For example, the 6-input SOP shown on the right:
 - $D1 = !a + b$ is a two-cube divisor
 - $D2 = a * !b$ is a single-cube two-literal divisor
 - $D1 = !D2$
- It uses a priority queue to find what divisor to extract
- It updated the SOP and the priority queue after extracting each divisor
- The resulting factored form can be further optimized by other methods

.i 6

.o 1

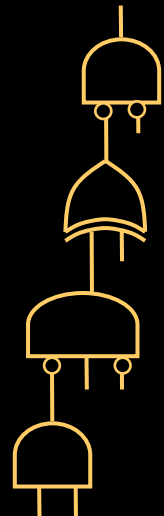
0-1--0 1

-11--0 1

100-00 1

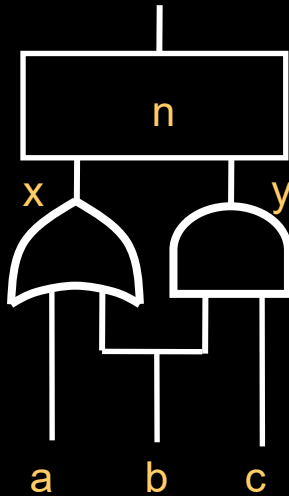
1000-0 1

--1110 1

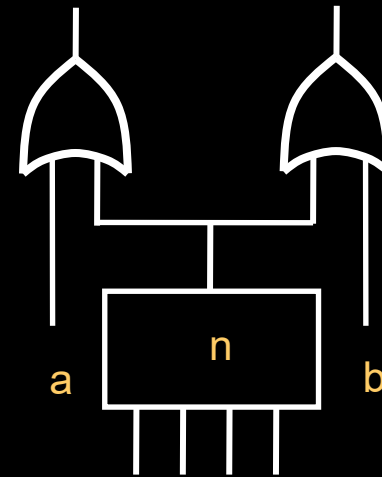


Don't-Cares

- **Don't-cares** are input combinations, for which node **n** in the network can produce any value



$(x = 0, y = 1)$ is a satisfiability
don't-care for node **n**



$(a = 1, b = 1)$ is an observability
don't-care for node **n**

Traditional Don't-Care-Based Optimization

- It is a complicated Boolean problem with many variations and improvements – hard to select one “classic algorithm”
 - Below we consider the work of H. Savoj and R. Brayton from early 1990's
- Compute compatible subsets of observability don't-cares (CODCs) for all nodes in one sweep
- For each node, project don't-cares into a local space, and minimize the node's representation
 - Don't-care are represented using BDDs, the node using an SOP
- The resulting network has smaller local functions
 - Quality of mapping in terms of area and delay are often improved
- Implementation in SIS (full_simplify) has not been widely used
 - Slow and unscalable in many cases

Example of Using Don't-Cares

Original function:

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

	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

Complete don't-cares:

	00	01	11	10
00	0	-	-	0
01	0	-	1	-
11	0	1	1	0
10	0	-	1	0

Optimized function:

$$F = b$$

	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

Presentation Overview

- Logic synthesis
- Traditional logic synthesis
- **Boolean logic synthesis**
- Resubstitution formulation and classification
- Resubstitution methods
 - Small scale (usingunate divisors)
 - Large scale (using support minimization)
 - Multi-output exact (using SAT solving)
- Conclusions

Boolean Logic Synthesis

- This synthesis uses the complete set of rules of Boolean algebra to optimize logic expressions

Identity: $A + 0 = A$ $A * 1 = A$

Complementarity: $A + A' = 1$ $A * A' = 0$

Idempotency: $A + A = A$ $A * A = A$

Involution: $(A')' = A$

Absorption: $A + (A * B) = A$ $A * (A + B) = A$

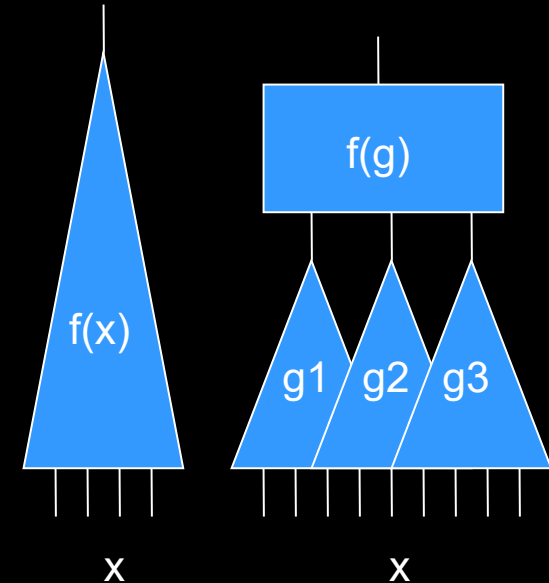
De Morgan's rule: $(A + B)' = A' * B'$ $(A * B)' = A' + B'$

Distributivity: $A + (B * C) = (A + B) * (A + C)$

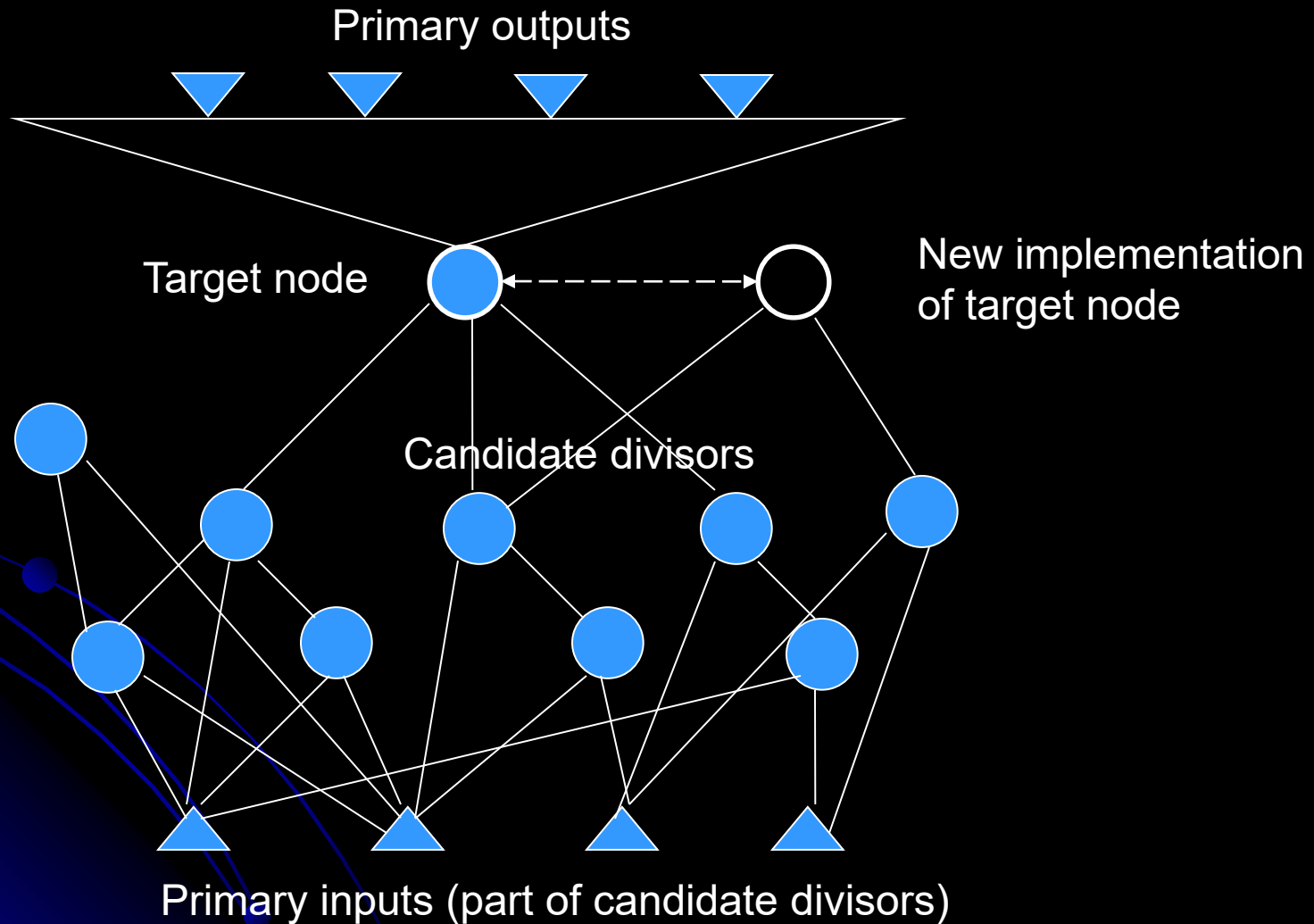
$$A * (B + C) = (A * B) + (A * C)$$

Resubstitution

- **Resubstitution** (aka **resub**) is arguably the simplest formulation of the main problem in logic synthesis
- It expresses one function in terms of others
 - Given function $f(x)$ and divisors $\{g_i(x)\}$, is it possible to represent f in terms of a subset of divisors g_i ?
 - If so, what is function $f(g)$?
- In general, resubstitution is Boolean (not algebraic)
- Variations:
 - Express function $f(g)$ using the fewest gates
 - Given many (~ 1000) divisors, find the smallest (~ 10) subset needed to express the function
- Solve the problem for
 - a completely-specified single-output function
 - a function with don't-cares
 - a multi-output function
 - a Boolean relation



Resubstitution



Classification of Resubstitution

- Functional representation
 - Truth tables
 - Decision diagrams
 - Simulation signatures verified by Boolean satisfiability
 - And-inverter graphs
- Flexibility exploited
 - Completely specified Boolean functions (CSFs)
 - Incompletely specified Boolean functions (ISFs, aka “functions with don’t-cares”)
 - Multi-output Boolean functions
 - Boolean relations
 - SPFDs

Classification (continued)

- Scope and approximate runtime budget
 - (1) Fast rewriting-style optimization
 - 10 divisors / transform, 100K transforms / second
 - (2) Mid-range resynthesis-style optimization
 - 1000 divisors, 100 transforms / second
 - (3) High-effort multi-output SAT-based optimization
 - 10 divisors, 1 transform / second
- Engines where resub is used in ABC
 - Technology-independent logic synthesis (rewriting, balancing, decomposition, resubstitution, etc)
 - Post-placement optimization (delay-opt, congestion-aware, etc)
 - Verification (miter optimization, creating equivalent nodes, etc)
 - Reverse engineering (synthesizing missing nodes to represent the boundary of a module)

Presentation Overview

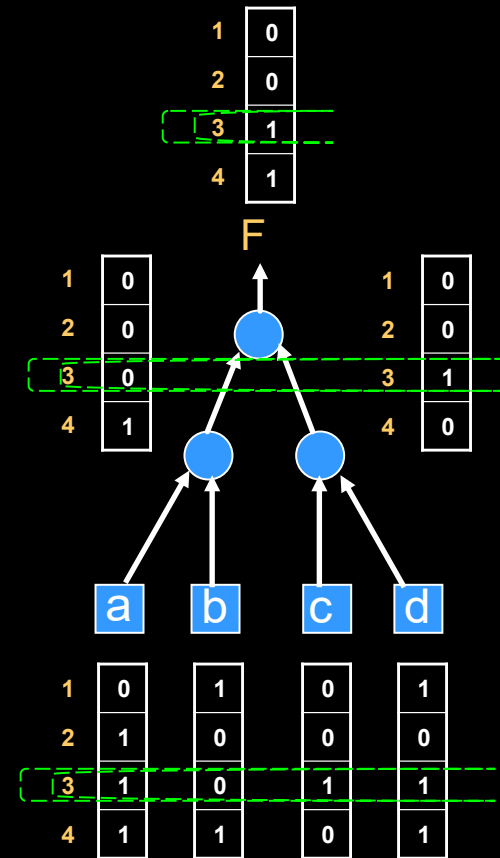
- Logic synthesis
- Traditional logic synthesis
- Boolean logic synthesis
- Resubstitution formulation and classification
- Resubstitution methods
 - ● **Small scale (usingunate divisors)**
 - Large scale (using support minimization)
 - Multi-output exact (using SAT solving)
- Conclusions

Functional Representation for Boolean Resubstitution

- Traditionally, SOPs, BDDs, or truth tables are used
- However, if the function is large (more than 10 inputs), it may be better to use simulation signatures to manipulate functions
 - This representation is assumed in this talk
- Using simulation signatures
 - Decide on a set of expressive simulation patterns
 - It can be the set of all minterms if the function is small
 - Simulate these patterns through the circuit
 - Use the values at each node as its simulation signature
 - It can be seen as an incomplete truth table of the node
 - If an error appears due to incompleteness, fix it later

Computing Simulation Signatures

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

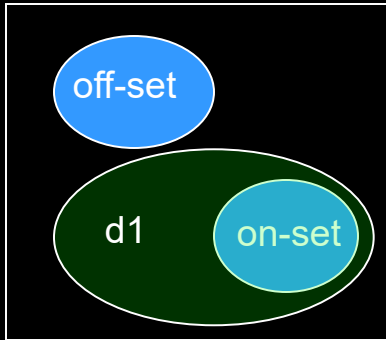


K-Resub Algorithm

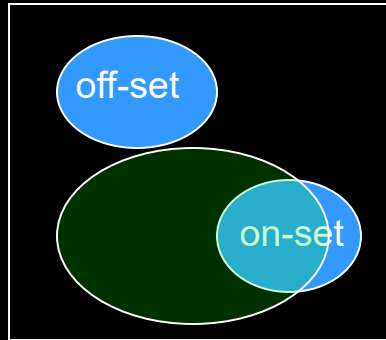
- This is the simplest form of resubstitution, based on smart enumeration of unate and binate divisors
- Pre-compute simulation info and iterate over the nodes
 - For each node, computes divisors
 - Tries to use unate divisors whenever possible
 - Checks correctness using SAT solver
 - Adds new simulation patterns if the SAT check fails
- Reference
 - S.-Y. Lee, H. Riener, A. Mishchenko, R. Brayton, and G. De Micheli, " A simulation-guided paradigm for logic synthesis and verification", IEEE Trans. CAD, Vol. 41(8), August 2022.
https://people.eecs.berkeley.edu/~alanmi/publications/2021/tcad21_sim.pdf₂₅

Unate Divisors

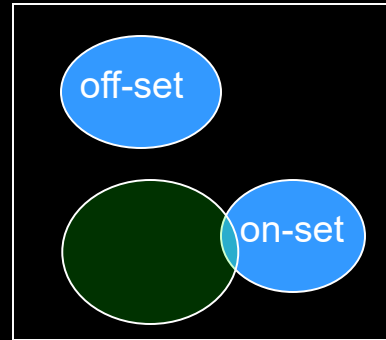
Ideal



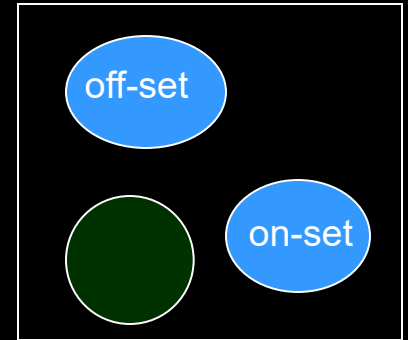
Good



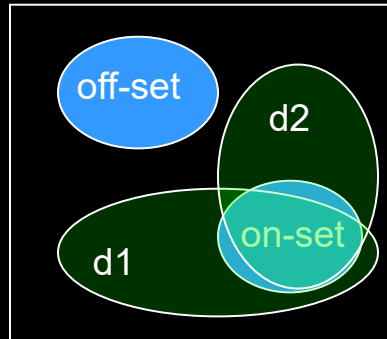
Bad



Irrelevant



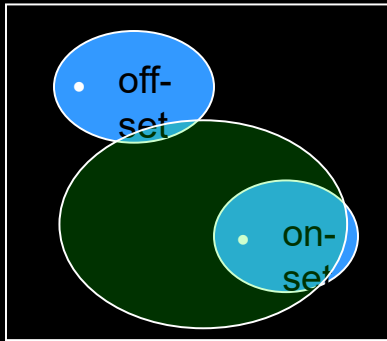
Two good divisors



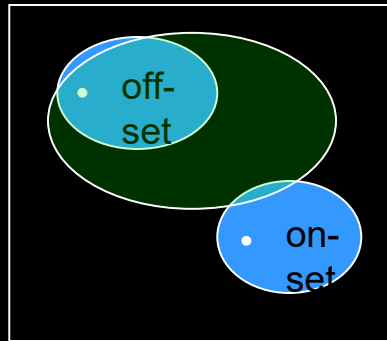
Unate divisor is a divisor that does not overlap with onset or with offset of F .
Divisor weight is the percentage of the on-set covered (the bigger, the better).
 If there is no ideal unate divisor, at least one gate is needed to realize the target.
 Two unate divisors are sufficient to express F , if they cover the on-set completely.

Binate Divisors

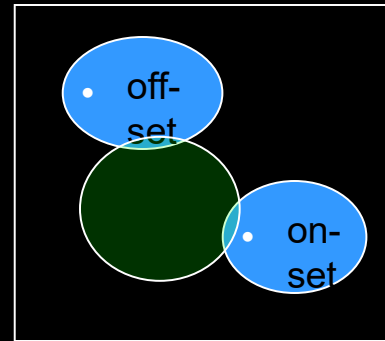
Good1



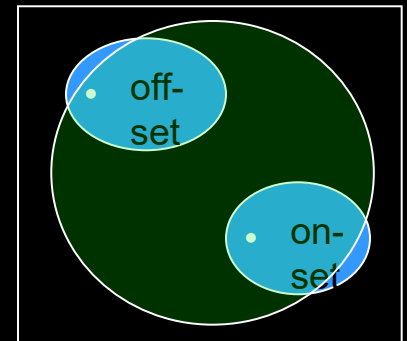
Good2



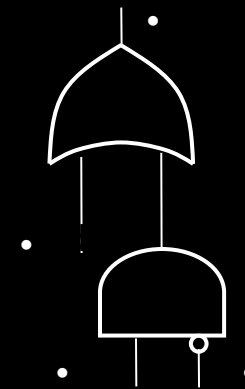
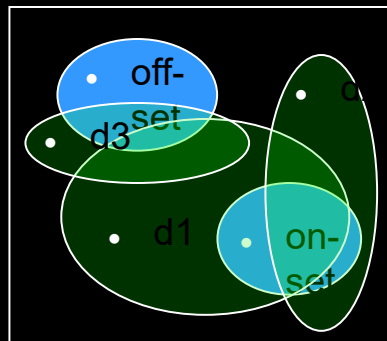
Bad1



Bad2



Result



Binate divisor is a divisor, which partially covers both onset and offset of F .

Divisor weight is the percentage of correctly covered on-set and off-set.

- At least two gates are needed to express F using a binate divisor, as shown.

Presentation Overview

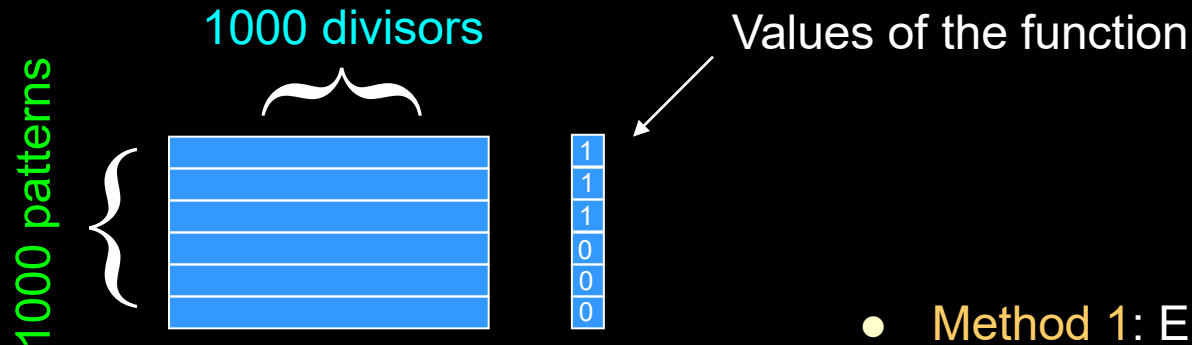
- Logic synthesis
- Traditional logic synthesis
- Boolean logic synthesis
- Resubstitution formulation and classification
- Resubstitution methods
 - Small scale (usingunate divisors)
 - Large scale (using support minimization)
 - Multi-output exact (using SAT solving)
- Conclusions

Resub with Support Minimization

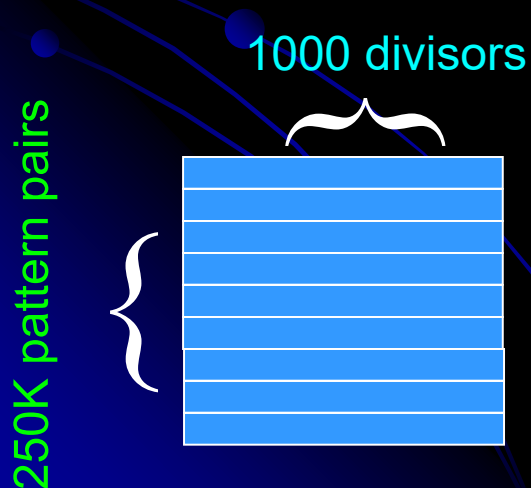
- **Input:** Function and **D** divisors (with or without DCs) while simulating **P** pattern
- **Outputs:** Circuit for the function with support of **S** divisors, composed of **N** gates
- Typical problem sizes
 - Resub in AIG rewriting (command “resub”): **D = 20, P = 64, S = 3, N = 2**
 - Window-based resub (command “mfs”): **D = 200, P = 1000, S = 6, N = 5**
 - High-effort resub-based applications: **D = 2000, P = 10000, S = 16, N = 10**
- **Support-size minimization methods:**
 - Method 1: Enumerating subsets
 - Method 2: Greedy divisor accumulation
 - Method 3: Covering table
 - Method 4: Heuristic support refinement
- **Circuit-size minimization methods:**
 - Minimize support first, then minimize the function (sub-optimal)
 - K-resub described above (up to 4 gates)
 - New method described below that was used in ICCAD CAD Competition
 - A. Q. Dao, N.-Z. Lee, L.-C. Chen, M. P.-H. Lin, J.-H. R. Jiang, A. Mishchenko, and R. Brayton, "Efficient computation of ECO patch functions", Proc. DAC'18.
 - https://people.eecs.berkeley.edu/~alanmi/publications/2018/dac18_eco.pdf

Support Computation Methods

Problem Formulation



Covering Table



- **Method 1:** Enumerating subsets
 - Not practical
- **Method 2:** Greedy divisor accumulation
 - Not accurate
- **Method 3:** Covering table
 - High memory usage, slow runtime
- **Method 4:** Heuristic support refinement (Stochastic iterative support refinement with dynamic approximate covering table)
 - A practical alternative

Method 4: Stochastic ($N=10$) Iterative ($M=50$) Support Computation Using Dynamic Covering Table

- Maintain globally best solution $S_{\text{global_best}}$
- Perform N random restarts
 - Reset dynamic covering Table
 - Empty Table
 - Add to Table a random subset of the complete covering table
 - Construct one feasible solution S using the original divisor information
 - Empty S
 - Iteratively add to S the divisor that covers most of the still uncovered rows of Table
 - Break ties randomly
 - $S_{\text{local_best}} = S$
 - Iterate M times
 - Minimize S
 - S_2 is obtained by removing one divisor from S
 - if (S_2 is feasible) $S = S_2$; else collect a random subset of failing pattern pairs
 - If ($S_{\text{local_best}} > S$) $S_{\text{local_best}} = S$
 - Add all collected failing pattern pairs to Table
 - Reconstruct S using the original divisor information
 - Empty S
 - Iteratively add to S the divisor that covers most of the still uncovered rows of Table
 - Break ties randomly
 - If ($S_{\text{global_best}} > S_{\text{local_best}}$) $S_{\text{global_best}} = S_{\text{local_best}}$
- Return $S_{\text{global_best}}$

Minimizing Resub Circuit Size Without Minimizing Support Size

- Perform stochastic iterative support minimization as before
- Add two changes
 - (1) At the end of each iteration, create new divisors
 - A new divisor is a 2-input gate on top of 2 available divisors
 - (2) When choosing a solution, minimize “support size + divisor cost”
 - Cost of an old divisor is 0
 - Cost of a new divisor is the number of gates needed to create it
- In the end, look for supports composed of a single divisor
 - This divisor is the root node of the circuit implementing the function

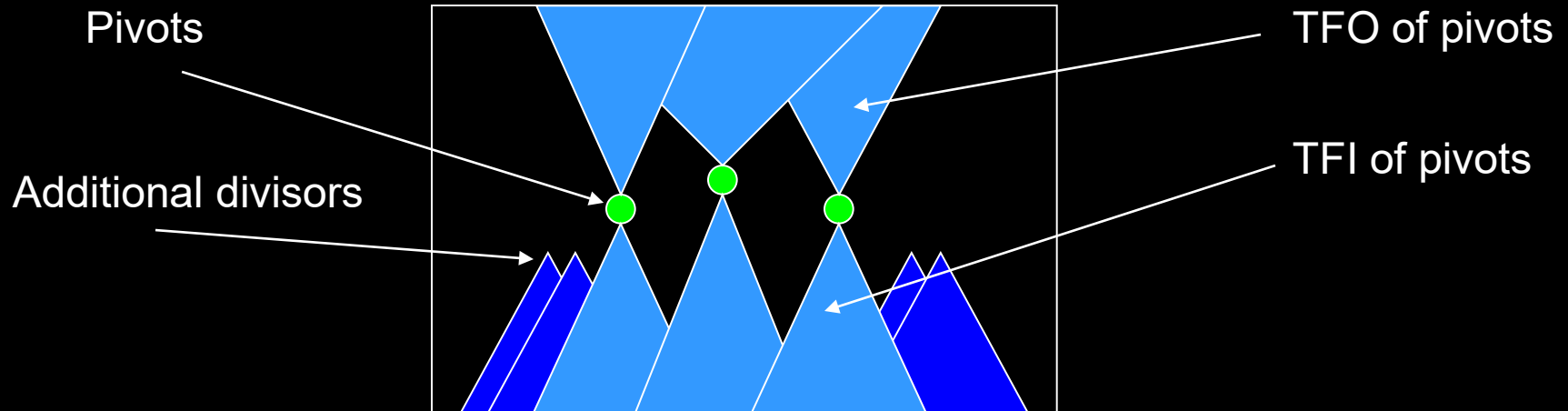
Expected Results

- Improving quality
 - Finding smaller resub functions
 - Possible because we do not focus on support minimization
- Improving runtime
 - Compact problem representation with don't-cares
 - Less runtime to access data in memory
 - Row-wise and column-wise bit-matrix
 - Always using word-level operators, instead of bit-sampling
 - Better computation flow
 - Appending columns on-the-fly to represent new divisors

Presentation Overview

- Logic synthesis
- Traditional logic synthesis
- Boolean logic synthesis
- Resubstitution formulation and classification
- Resubstitution methods
 - Small scale (usingunate divisors)
 - Large scale (using support minimization)
 - Multi-output exact (using SAT solving)
- Conclusions

Multi-Output Resub Problem



Assume small number of primary inputs (≤ 16)

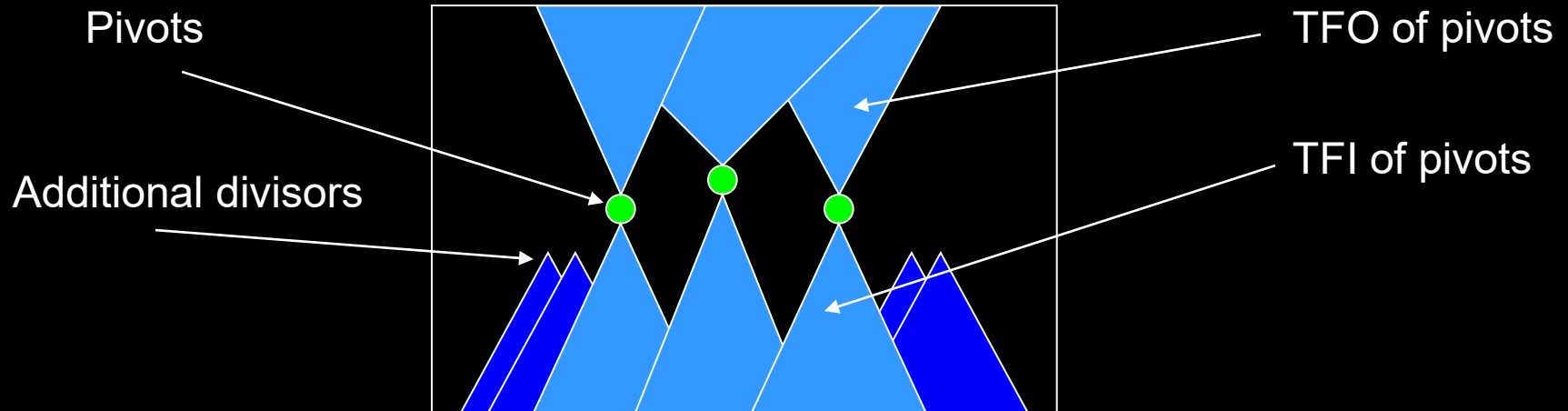
Select several (1 to 6) pivots

Collect TFI nodes and additional divisors (≤ 50)

Derive Boolean relation: primary inputs – pivots

Formulate a SAT problem

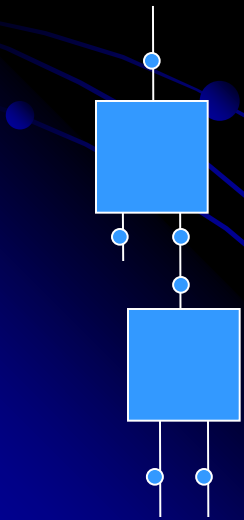
SAT-Based Formulation



The **SAT problem** is 'satisfiable' iff there exist N internal two-input nodes driving K pivots, with fanins selected out of D divisors, such that when this circuit is inserted, the network's output functions remain unchanged.

Encoding Circuit Structure

- Use $K \cdot (D+N)$ variables for each output (there are K outputs in total)
 - A variable is 1, iff the output is connected to the given node or divisor
- Use $N \cdot 2 \cdot (D+N)$ variables for each node (there are N nodes in total)
 - A variable is 1, iff the node has a given divisor as a fanin
- Only 1 out $D+N$ vars can be 1 for the solution to be valid
 - The 1-hot constraints can be added dynamically



For each node, we have 3 pins (encoded with different vars)

Suppose var v enables connection between pins $p1$ and $p2$

This is expressed as $v \rightarrow (p1 == p2)$, which yields two clauses

$$\sim v + \sim p1 + p2$$

$$\sim v + p1 + \sim p2$$

Encoding Node Functions

- Use 3 variables for each two-input node
 - They represent truth-table bits of the function (0, f0, f1, f2)
 - Further constrained to allow only AND or XOR with complemented inputs (rule out constants and buffers/inverters)

	0	1
0	0	f0
1	f1	f2

	0	1
0	0	0
1	0	1

AND (*)

	0	1
0	0	1
1	1	1

OR (+)

	0	1
0	0	1
1	1	0

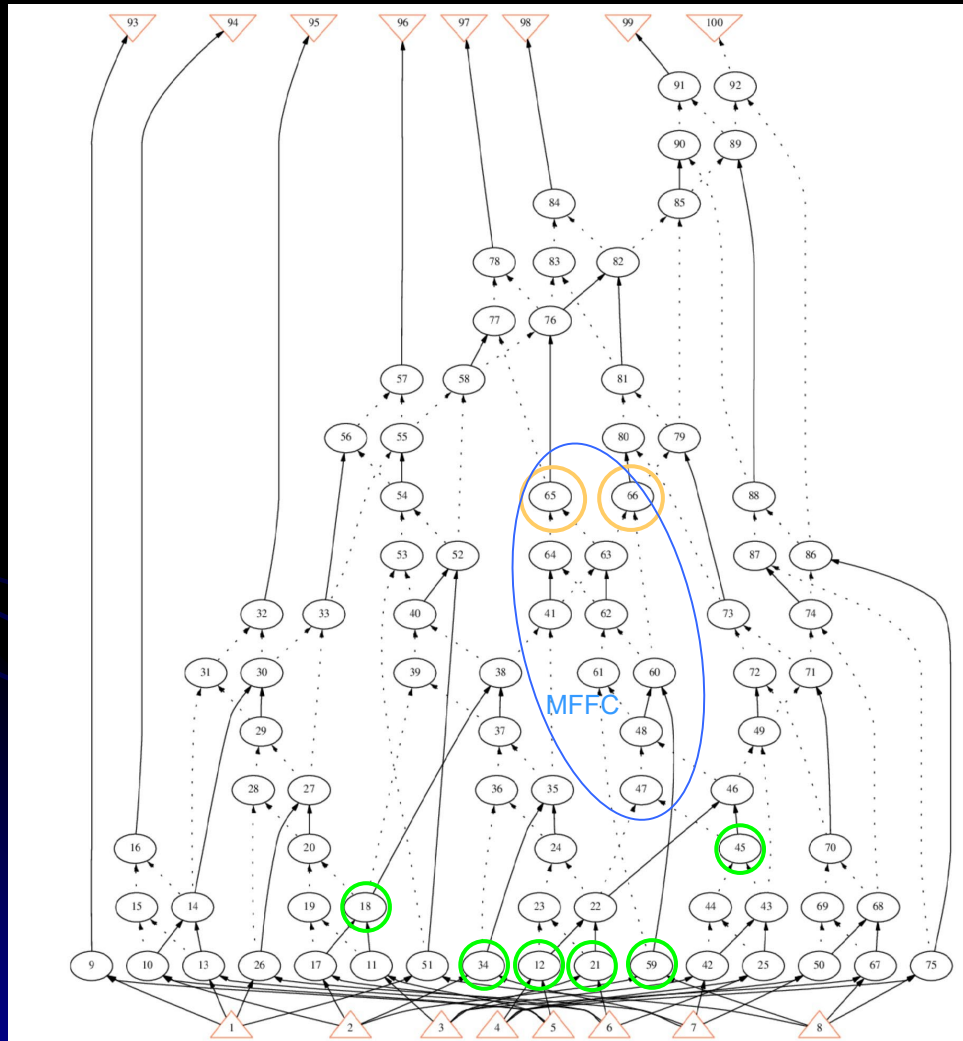
XOR (\oplus)

- For each one out of 3 minterms in the node's truth table, with inputs p0 and p1 and output p2, we have $(p0 \& p1) \rightarrow (p2 == f0)$, resulting in two clauses

$$\sim p0 + \sim p1 + \sim p2 + f0$$

$$\sim p0 + \sim p1 + p2 + \sim f0$$

Example

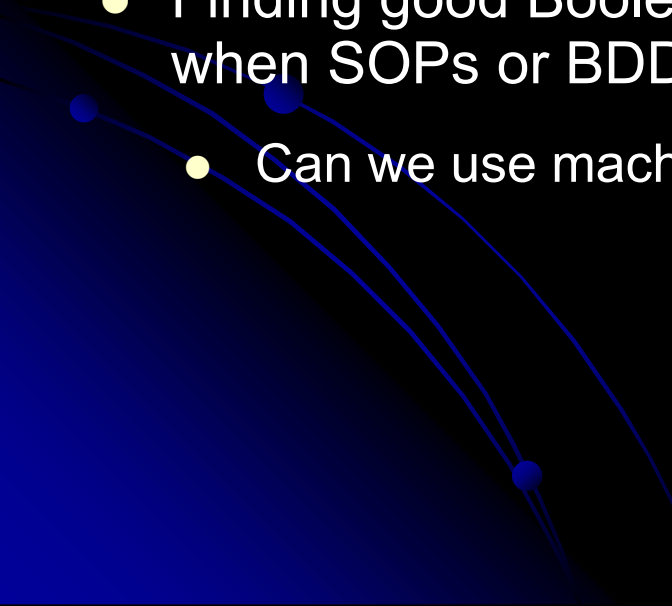


- The network has
 - 8 inputs
 - 8 outputs
- The window has
 - 6 inputs
 - 2 outputs
 - 10 nodes in MFFC
 - 15 divisors
 - 1 const node
 - 6 inputs
 - 8 nodes
- Boolean relation for this window can be computed by simulating 256 input patterns through the network

SAT-based Resub Summary

- For the first time, we have “simultaneous” multi-output resub (before we iterated over single-output resubs)
- SAT-based solution scales better when we have more divisors and fewer internal nodes
 - The limit is 12 new nodes for a 5-var function without side divisors
 - We can do 6 new nodes for a 5-var function with 20 divisors
- SAT-based solution may be the only way to minimize the number of AIG nodes beyond what is now possible
 - F. Reichl, F. Slivovsky and S. Szeider, “Circuit minimization with exact synthesis: From QBF back to SAT”, Proc. IWLS’23
- Another powerful resubstitution-based algorithm is called transduction
 - Y. Miyasaka, Transduction method for AIG minimization, Proc. IWLS’23.
 - <https://people.eecs.berkeley.edu/~alanmi/publications/iwls/iwls2023.pdf>

Open Research Problems

- Extremely fast logic synthesis (100M nodes in 1 min)
 - Currently, the runtime is 1M nodes in 1min
 - Deriving a small resulting circuit if it is known to exist, but present methods take 100x iterations of a synthesis script to derive it
 - These are examples when “traditional logic synthesis never ends” :)
 - Finding good Boolean divisors in multi-output logic synthesis when SOPs or BDDs cannot be constructed
 - Can we use machine learning for this?
- 

Conclusion

- Introduced logic synthesis
- Reviewed algebraic methods
- Defined Boolean resubstitution
- Considered three resubstitution methods
 - Based on unate divisors
 - Based on support minimization
 - Based on SAT solving
- Future work

Abstract

- The talk introduces logic synthesis and observes that the first logic synthesis methods used in the industrial EDA tools were algebraic methods, which are fast but tend to give suboptimal results because they rely on simplified rules to manipulate Boolean functions. The more general methods, known as Boolean logic synthesis, are harder to implement, lead to longer runtimes, but result in deeper minima and ability to solve more challenging optimization problems. The talk reviews several flavors of Boolean logic synthesis, in particular resubstitution, and shows how they can be used together in different parts of the design flow. One case-study includes the most general and the least scalable multi-output SAT-based exact synthesis. Another case study is the most scalable but somewhat less general formulation of resubstitution that minimizes support of the resulting function by a method that is similar to reinforcement learning. Other Boolean synthesis methods occupy the middle ground between these two. The implementations will be illustrated using the public domain tool ABC. Several open research problem will be introduced and motivated.