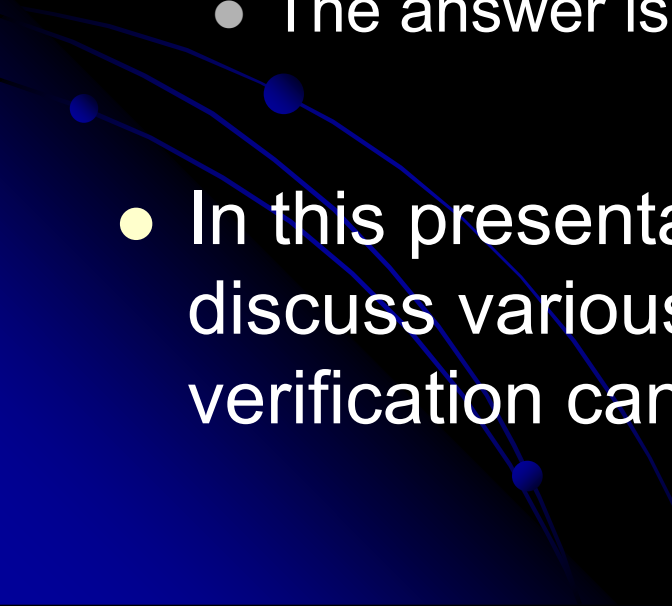


Towards Next Generation Logic Synthesis and Verification



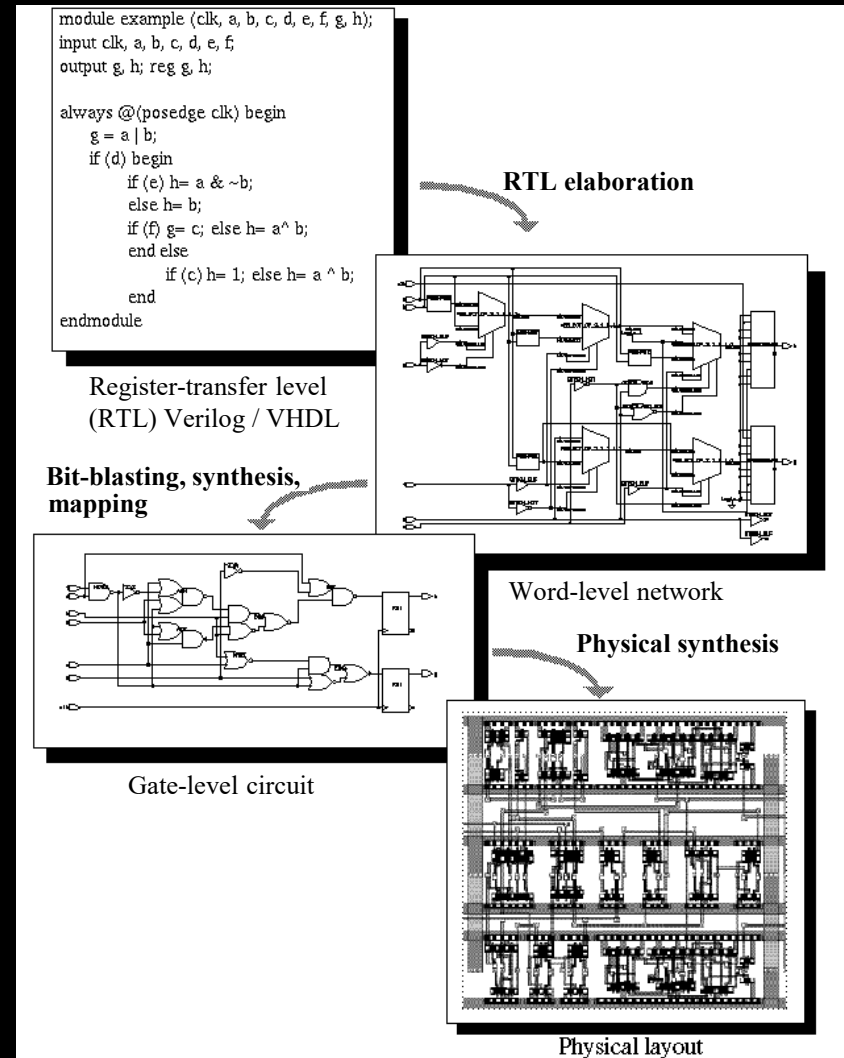
Alan Mishchenko
UC Berkeley

Overview

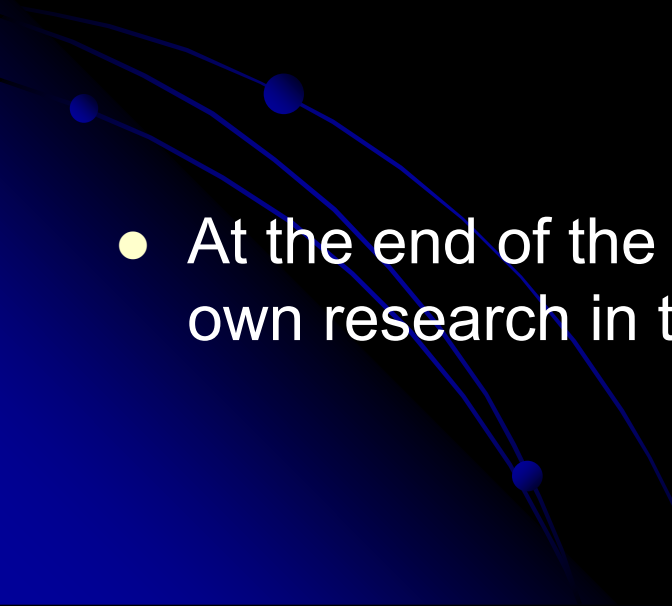
- Why talking about the next-gen developments?
 - Did we accumulate a critical amount of innovation, which is currently not used in the tools?
 - The answer is a resounding “yes”!
 - In this presentation, we review the design flow and discuss various ways logic synthesis and formal verification can be moved forward
- 

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

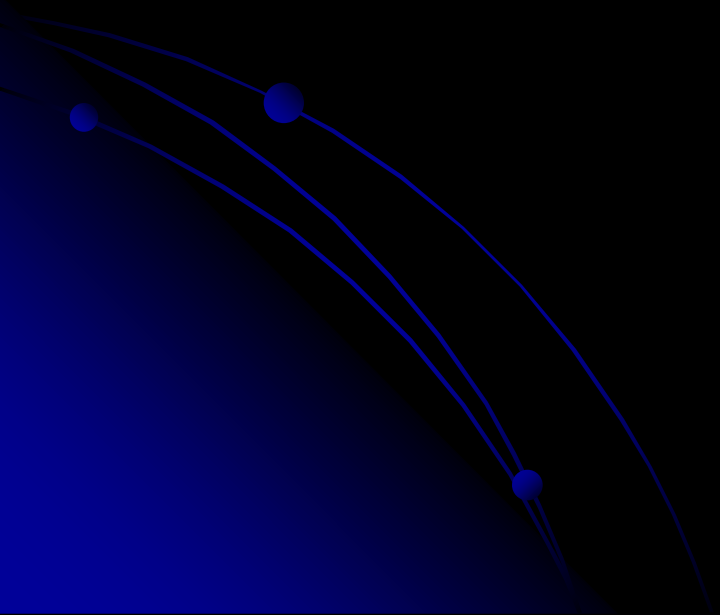


Topics Revisited

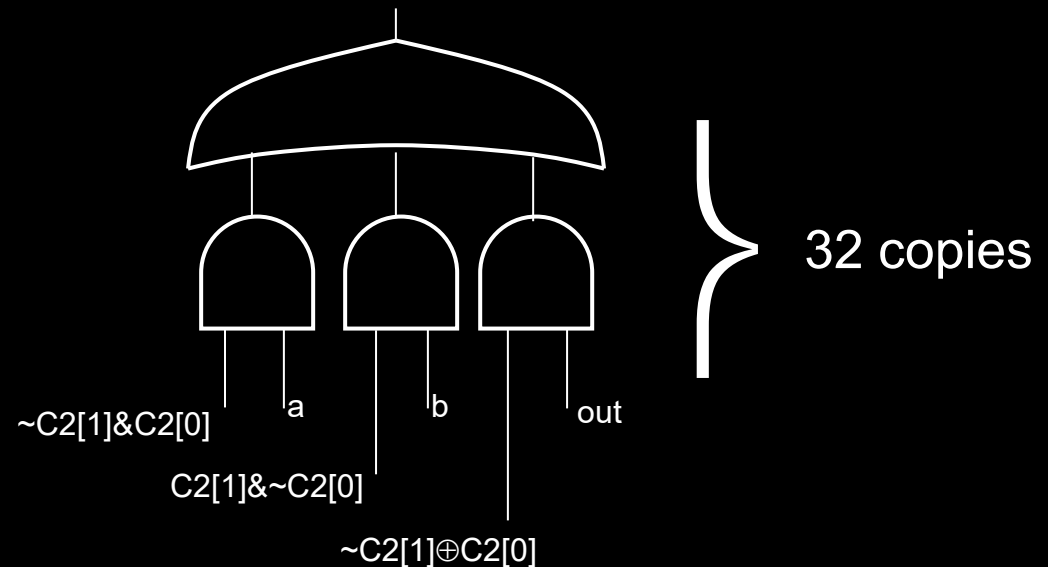
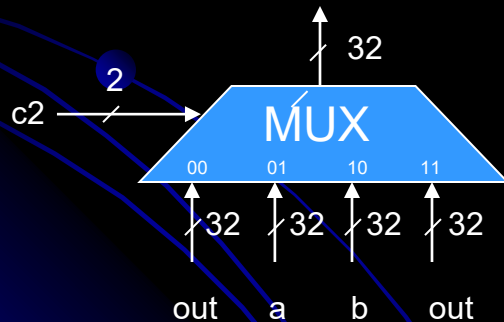
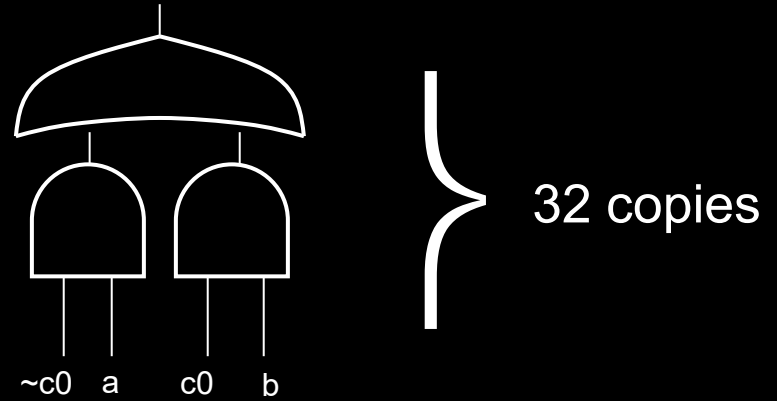
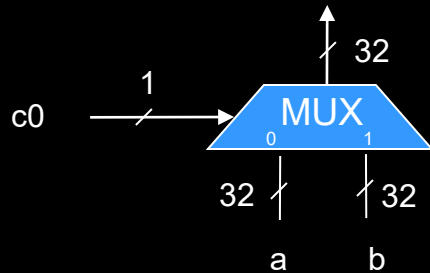
- Word-level to bit-level conversion
 - Hierarchical bit-level circuit representation
 - Computation engines (simulator, SAT solver, etc)
 - Technology-independent logic synthesis
 - Mapping/retiming used as a single transform
 - Verification and ECO
-
- 
- At the end of the talk, we also discuss how to begin your own research in this area, and what projects to take

Topics Revisited

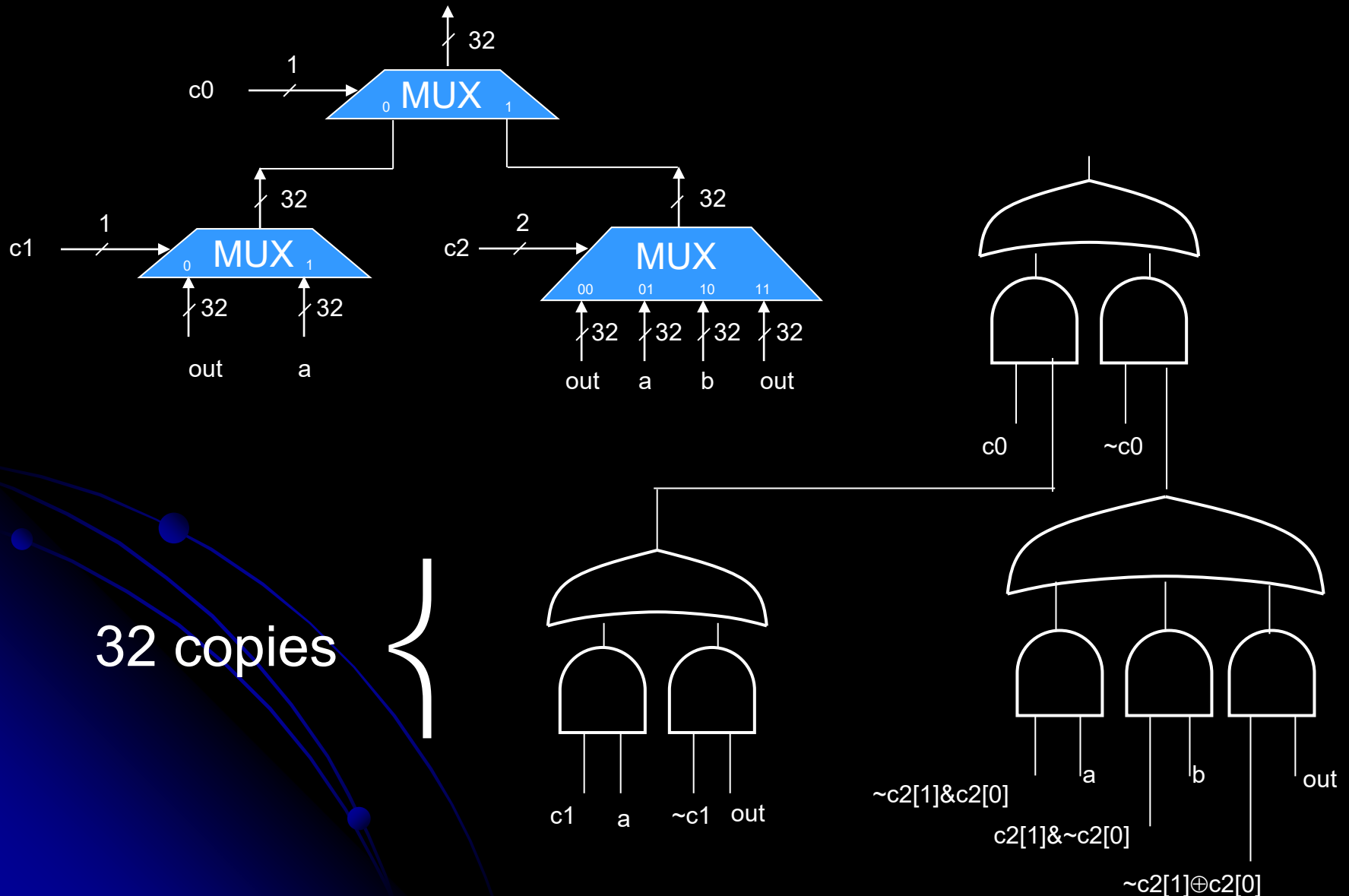
- Word-level to bit-level conversion (bit-blasting)
- Hierarchical bit-level circuit representation
- Computation engines (simulator, SAT solver, etc)
- Technology-independent logic synthesis
- Mapping/retiming used as a single transform
- Verification and ECO



Example of Bit-Blasting (Operator)



Example of Bit-Blasting (Circuit)



Word- to Bit-Level Conversion

- Current

- Available in Yosys, Verific, etc
- Works fine (may be slow and suboptimal sometimes)
- Currently done for each node without its context

- Future

- Can be made faster and better quality
- Can be context-dependent for each node
 - “Delay-aware bit-blasting”
 - “Congestion-aware bit-blasting”
- Can include targeted word-level transforms
 - “Control logic extraction”
- Can be coupled with a next-gen and-inverter-graph (AIG) package to extract box information

Example of Verilog Elaboration

```
module example ( clk, c0, c1, c2, a, b, out );
```

```
input clk;
```

```
input c0, c1;
```

```
input [1:0] c2;
```

```
input [31:0] a, b;
```

```
output [31:0] out;
```

```
reg [31:0] out;
```

```
always @ (posedge clk)
```

```
if (c0 == 1'b0)
```

```
begin
```

```
if (c1 == 1'b1)
```

```
out <= a;
```

```
end
```

```
else // c0 == 1'b1
```

```
begin
```

```
case (c2)
```

```
2'b00: ;
```

```
2'b01: out <= a;
```

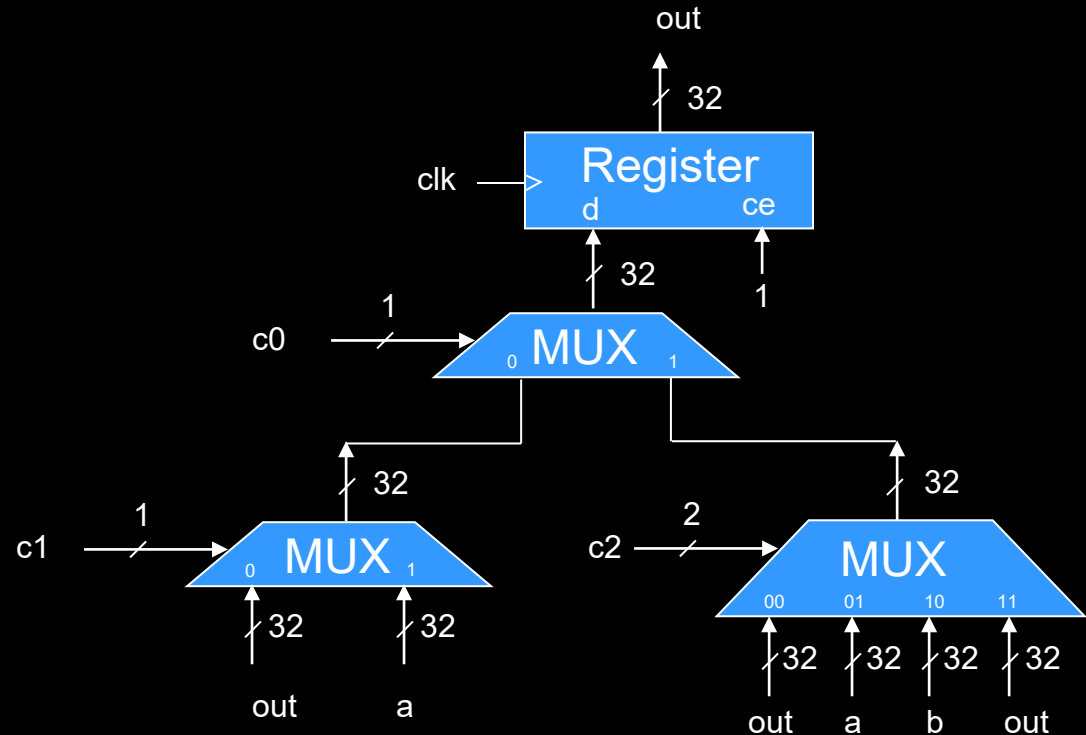
```
2'b10: out <= b;
```

```
2'b11: ;
```

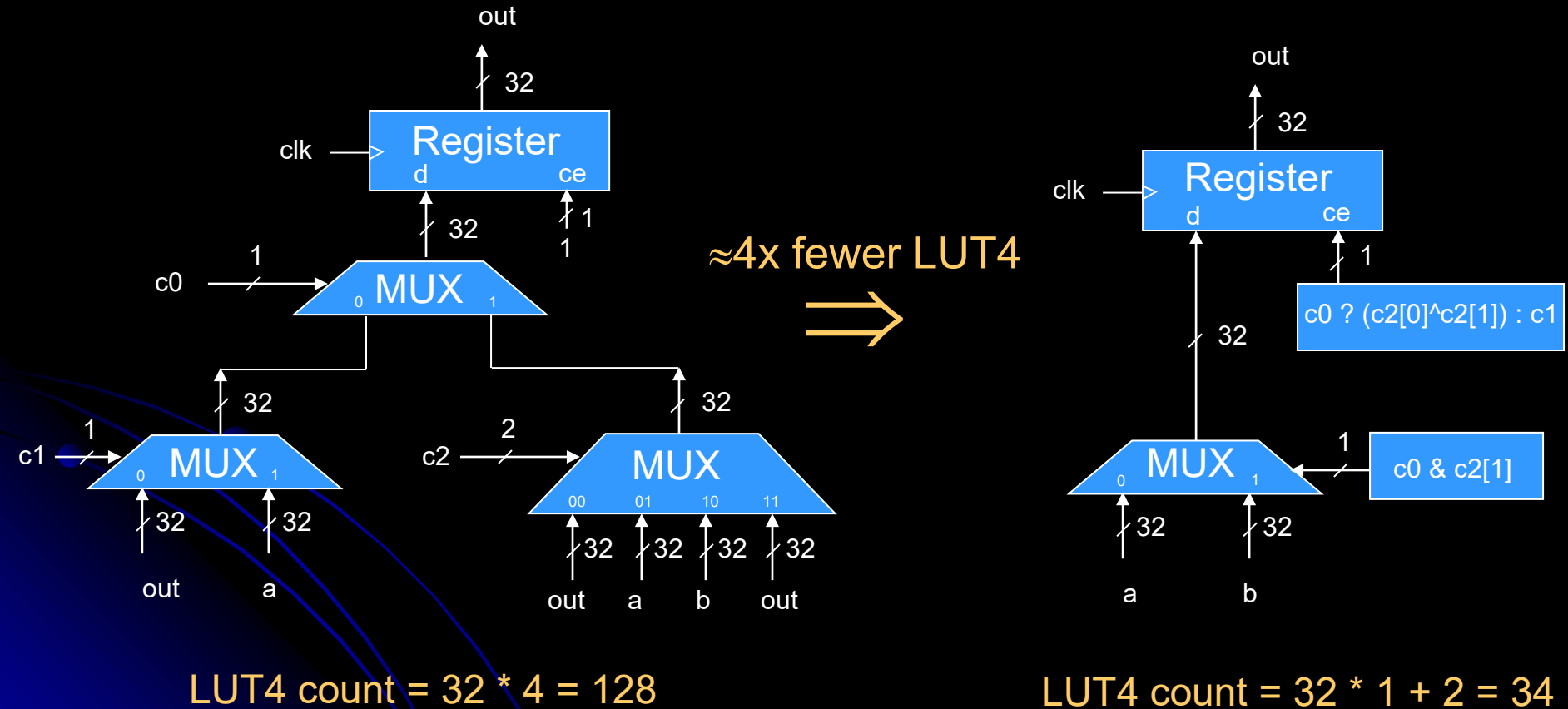
```
endcase
```

```
end
```

```
endmodule
```



Word-Level Transformation Before Bit-Blasting



Experimental Setup

- We considered 10 out of 30 designs from the OpenCores repository, for which applying MUX restructuring was most helpful. For the remaining 20 designs, there was no improvement or a minor improvement (about 1% in area).
- The optimization was applied to the whole design after flattening the user hierarchy. The results were verified by extracting a comb AIG before and after the transform and comparing these AIGs using command &cec in ABC.
- Conclusions
 - For the selected designs, unoptimized AIG is reduced by 40%.
 - After AIG rewriting, the average reduction in area is about 11.5%.
 - After AIG rewriting, the average reduction in AIG level is about 4%.

Experimental Results

Design name	Design statistics				Baseline			Transformed		
	PI	PO	Reg	FF	Base	BaseOpt	BaseLOpt	Trans	TransOpt	TransLOpt
double_fpu	136	70	132	5222	73425	62006	114	72808	60309	114
mem_ctrl	115	152	59	1138	21064	8798	38	11117	7607	38
nova	85	89	565	6429	559282	219827	118	148228	121603	108
picorv32	102	307	23	701	11599	6819	71	7022	6149	72
reedsolomn	11	10	245	3059	72915	25776	32	31408	22582	31
sudoku	732	731	171	3160	127355	63122	90	97836	59030	64
uart16550	51	37	32	352	4776	2560	18	2942	2374	18
usb	128	121	113	1762	16551	13240	41	14793	12472	41
vga_led	89	109	56	830	11682	5321	32	5513	5055	33
wb_dma	217	215	41	796	73642	41074	20	57802	42401	20
Geomean					1.000	1.000	1.000	0.596	0.885	0.959

PI (PO) is the number of bit-level primary inputs (outputs).

Reg is the number of word-level registers in the design.

FF is the number of bit-level flops. (Some of them are part of word-level registers.)

Baseline is the result without MUX restructuring. **Transformed** is the result with MUX restructuring.

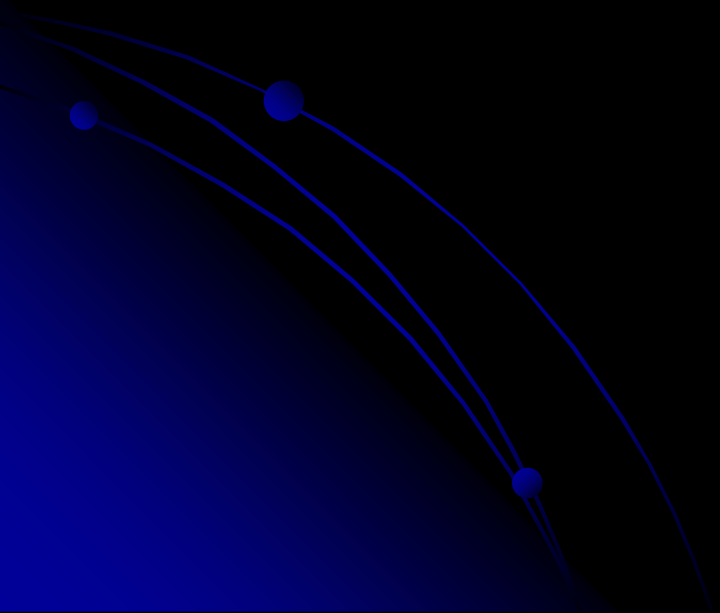
Base (Trans) are AIG sizes after structural hashing.

BaseOpt (TransOpt) is the AIG sizes after applying &dc2 in ABC.

BaseLOpt (TransLOpt) is the AIG level count after applying &dc2 in ABC.

Topics Revisited

- Word-level to bit-level conversion
- Hierarchical bit-level circuit representation
- Computation engines (simulator, SAT solver, etc)
- Technology-independent logic synthesis
- Mapping/retiming used as a single transform
- Verification and ECO

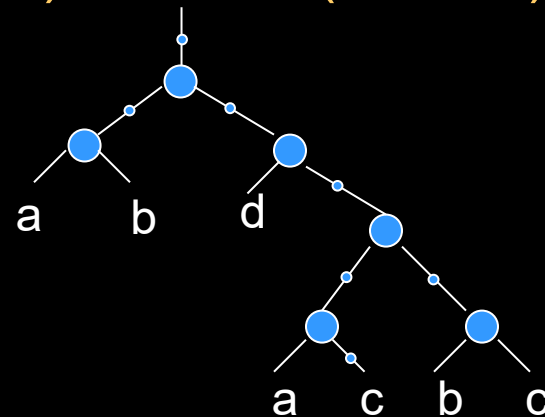


And-Inverter Graph (AIG)

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

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

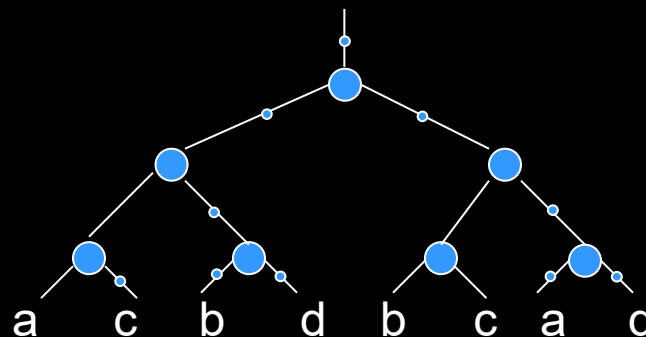


6 nodes

4 levels

a \ bcd	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	0
10	0	0	1	0

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

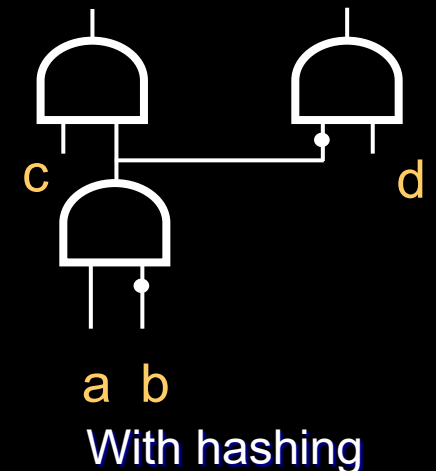
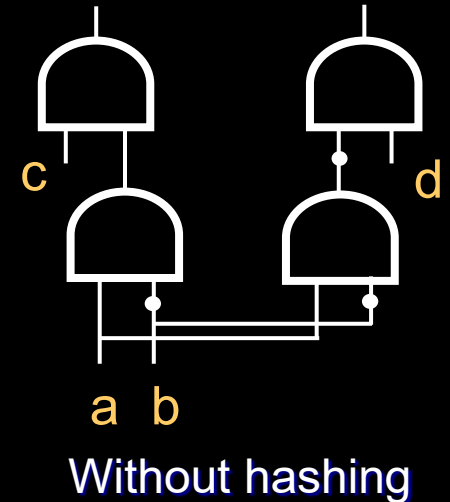


7 nodes

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



AIG Package Improvements

- Current

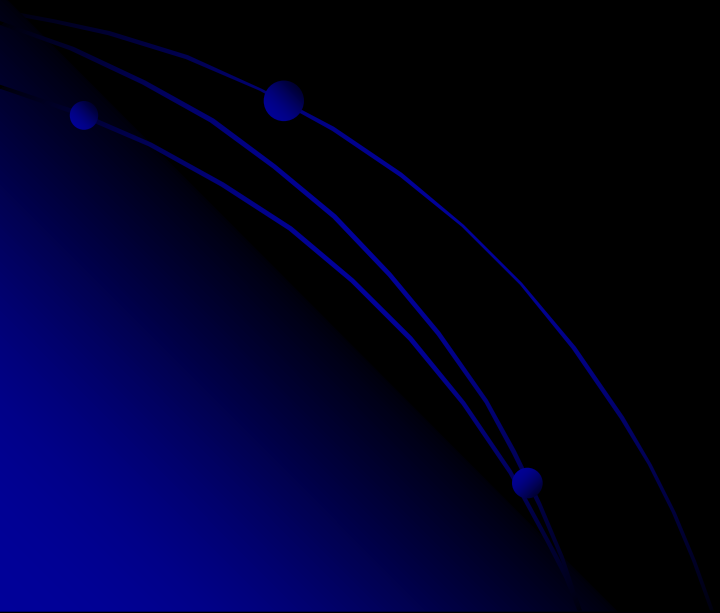
- Readily available in AIGER, Yosys, ABC, etc
- Well-motivated and well-understood
- Uniform representation for the flow
 - But used for a flat logic representation

- Future

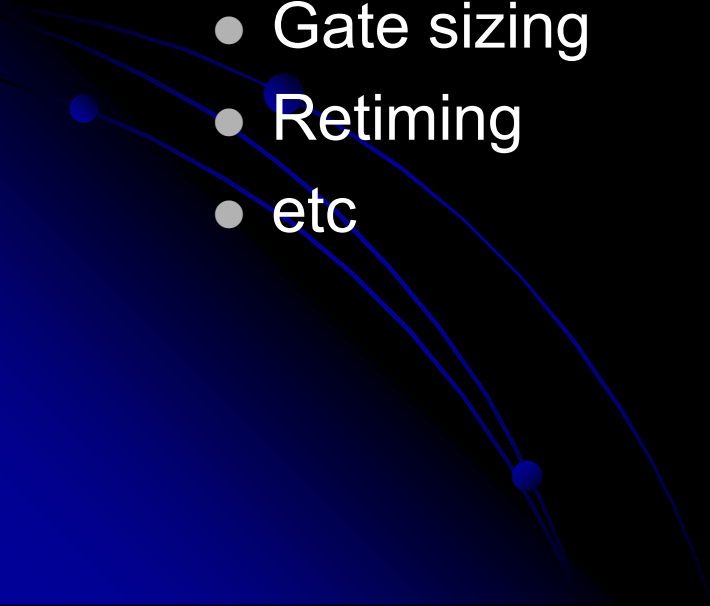
- Better integration with boxes (beyond “ABC9”)
- Better APIs for SAT solvers and simulators
- Can be extended to represent design hierarchy
- Can be extended to use XOR, MUX, MAJ, etc

Topics Revisited

- Word-level to bit-level conversion
- Hierarchical bit-level circuit representation
- Computation engines (simulator, SAT solver, etc)
- Technology-independent logic synthesis
- Mapping/retiming used as a single transform
- Verification and ECO

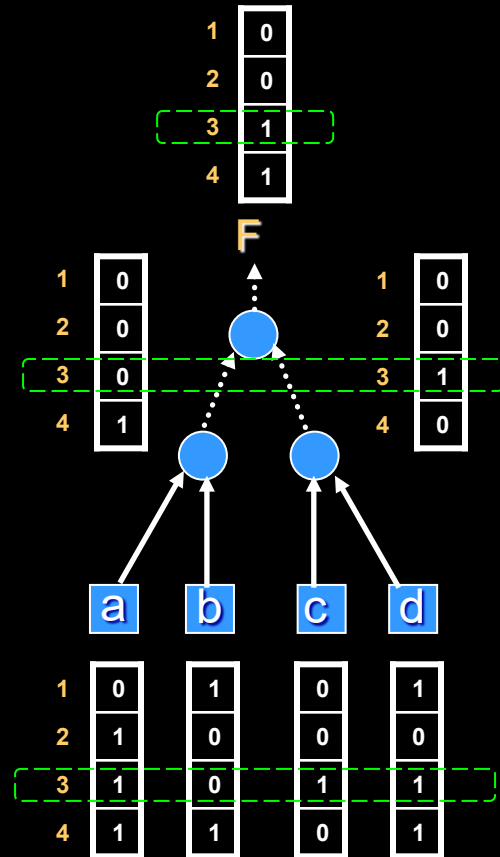


Engines Used in the Flow

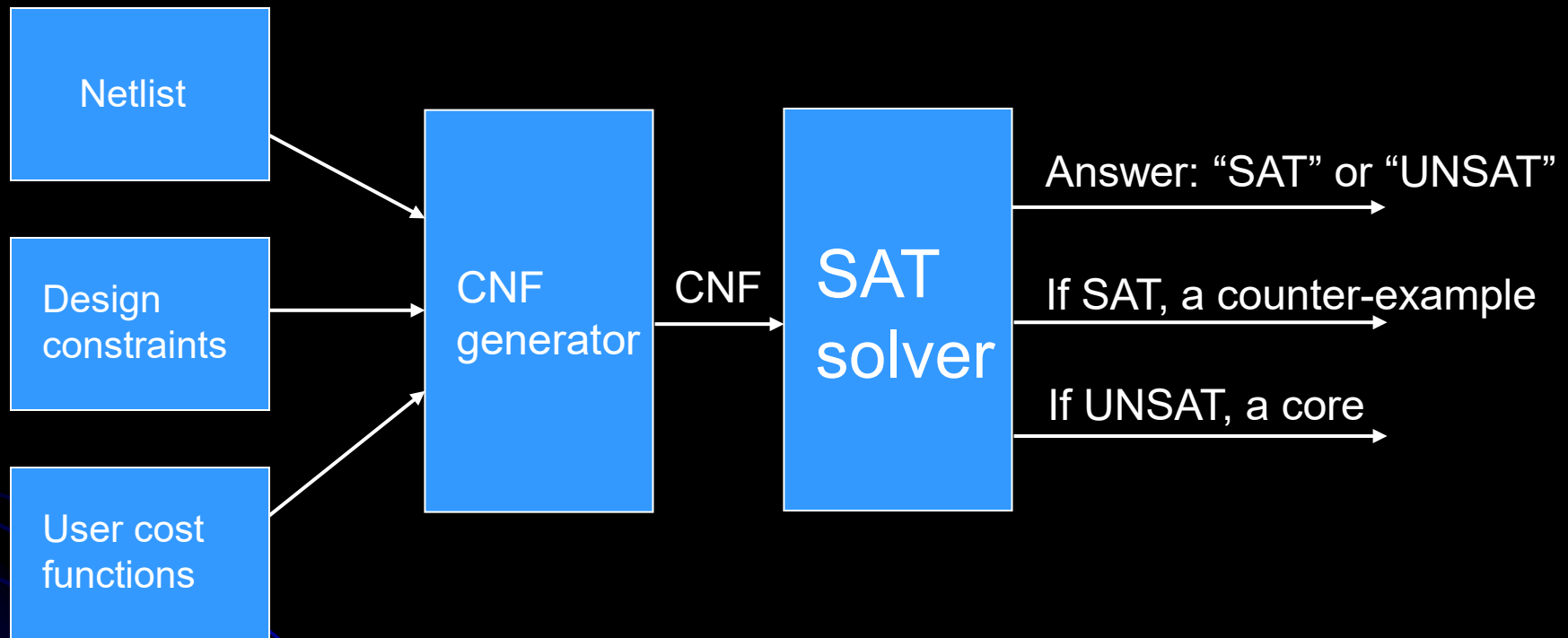
- Optimization engines
 - Delay optimization
 - Logic sharing extraction
 - Bit-level circuit rewriting
 - Technology mapping
 - Gate sizing
 - Retiming
 - etc
 - Computation engines
 - AIG package
 - Bit-level simulator
 - SAT solver
 - SAT sweeper
 - NPN matcher
 - SOP minimizer
 - etc
- 

Bitwise Simulation (SIM)

- 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



Satisfiability Solving (SAT)



Both counter-examples and cores are useful in SAT-based applications.

In practice, cores are often represented as subsets of assumptions that make the problem UNSAT.

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, scheduling, etc
- A lot of complex ideas 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
- MiniSAT is a popular SAT solver (<http://minisat.se/>)
 - Efficient (won many competitions)
 - Simple (600 lines of code)
 - Easy to modify and extend
 - Integrated into ABC

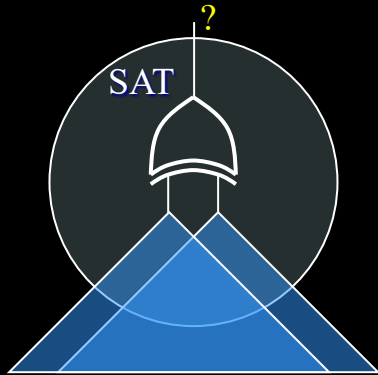
SIM and SAT (Present and Future)

- Bit-parallel SIM (scan forward)
- Reverse SIM (search backward)
- SAT solving (search backward and forward)
- In the future
 - Boundary between SIM and SAT will continue to blur
 - SIM and SAT will be better integrated
 - New engines will be developed with the role of both
 - Both SIM and SAT will become “features” (APIs) of the next-gen AIG package

SAT Solving

- SAT solving is a fundamental computation
 - Currently, it is used everywhere in logic synthesis, replacing BDDs as a Boolean computation engine
 - Circuit-based solvers have advantages over CNF-based solvers
- In the future
 - The boundary between circuit- and CNF-based solvers will continue to blur
 - Synthesis will continue to be based on fast light-weight solvers
 - Verification will periodically need strong heavy-weight solvers

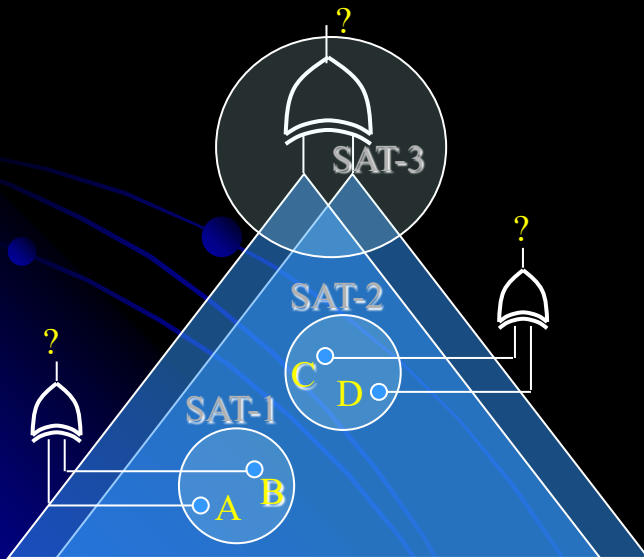
SAT Sweeping (Efficient CEC)



Applying SAT to the output

- **Naïve CEC approach – SAT solving**
 - Build output miter and call SAT
 - works well for many easy problems

- **Better CEC approach – SAT sweeping**
 - based on incremental SAT solving
 - Detects possibly equivalent nodes using **simulation**
 - Candidate constant nodes
 - Candidate equivalent nodes
 - Runs **SAT** on the intermediate miters in a topological order
 - Refines the candidates using counterexamples



Proving internal equivalences
in a topological order

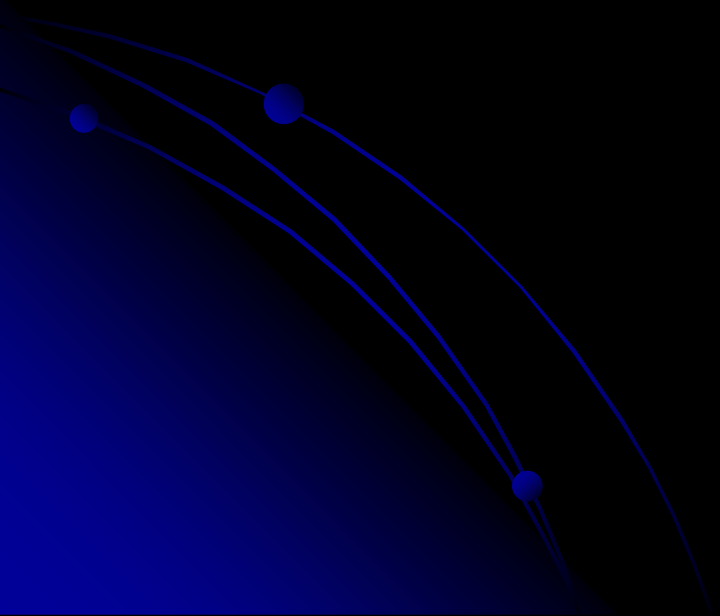
CEC = Combinational Equivalent Checking

Improvements to SAT Sweeping

- Runtime and scalability (10x+)
- Applicability to sequential blocks, arithmetic blocks, etc
- Native integration with the next-gen AIG package
- How the flow can benefit from improved engines?
 - Faster runtime (3x)
 - Larger logic blocks (100K gates → 10M gates)
 - QoR hard to estimate (possibly 5% in area and 10% in delay)
 - Automation (less parameter tweaking)
 - Easier development of future applications in the flow

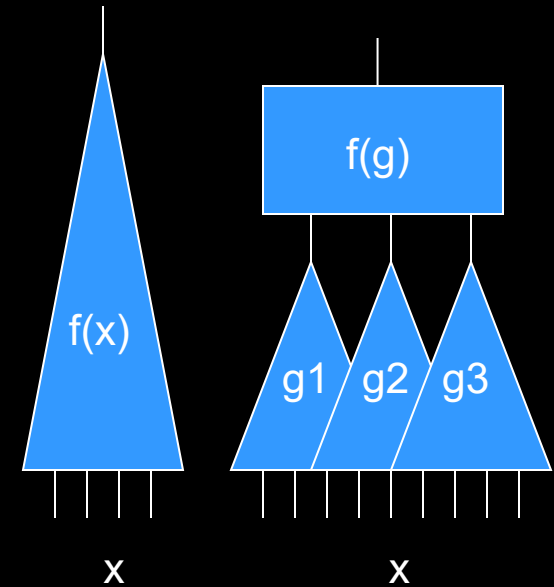
Topics Revisited

- Word-level to bit-level conversion
- Hierarchical bit-level circuit representation
- Computation engines (simulator, SAT solver, etc)
- Technology-independent logic synthesis (resubstitution)
- Mapping/retiming used as a single transform
- Verification and ECO

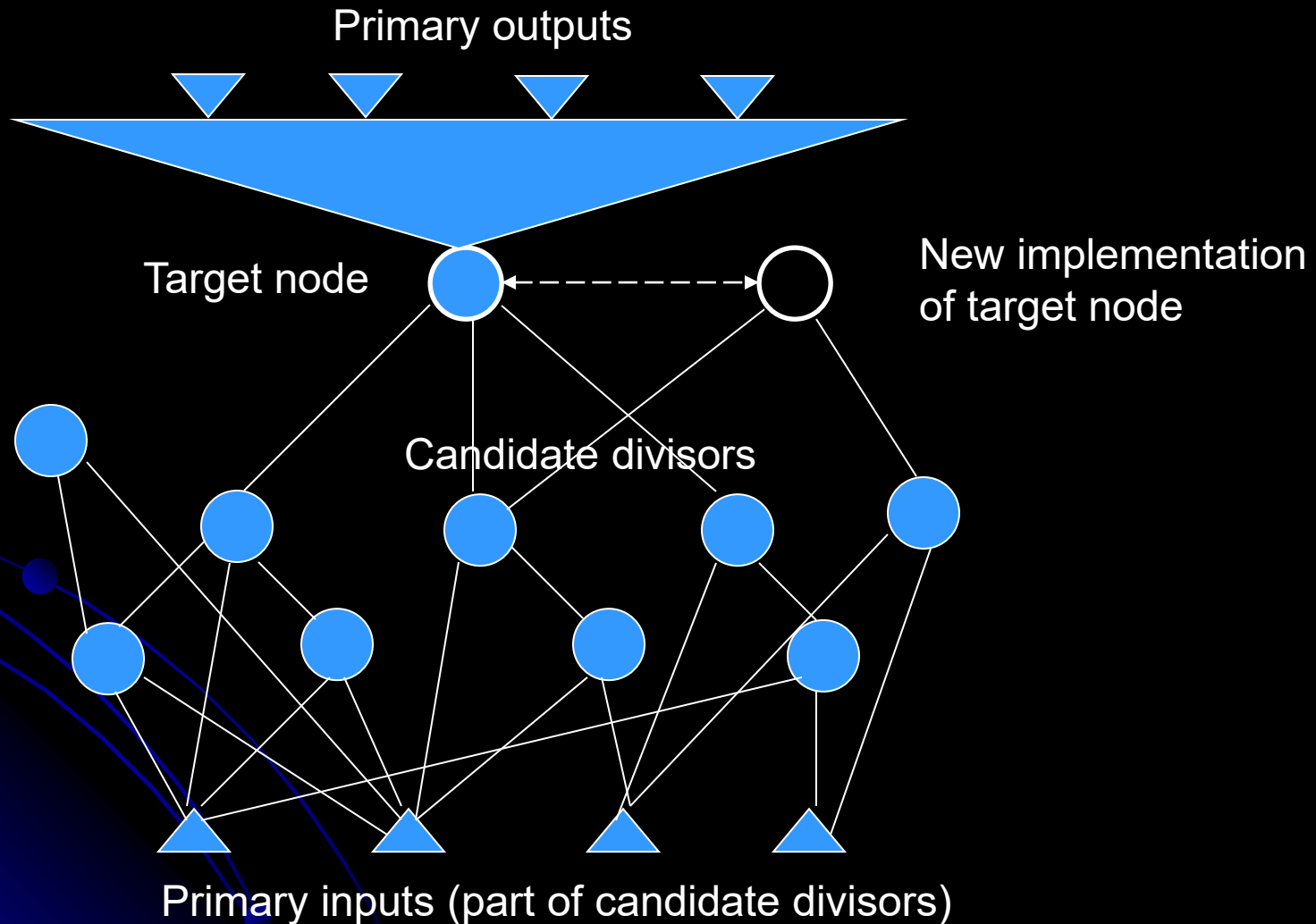


Resubstitution

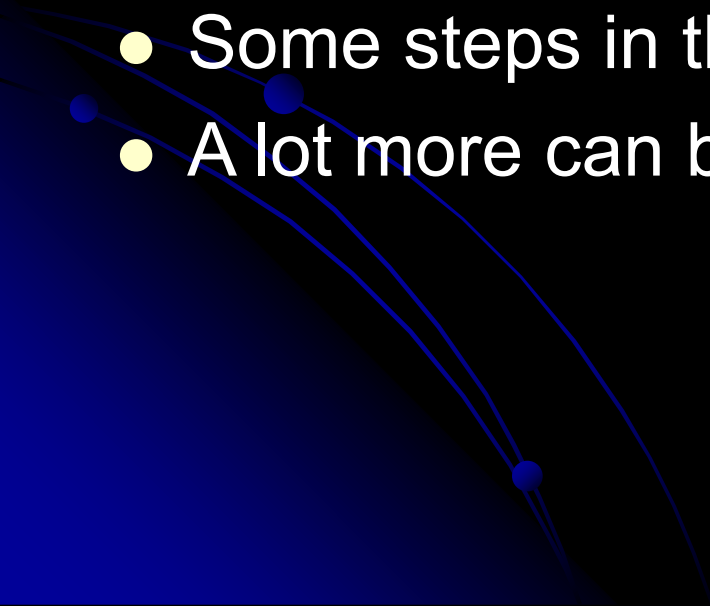
- **Resubstitution** means expressing one function in terms of others
 - Given $f(x)$ and $\{g_i(x)\}$, is it possible to express f in terms of a subset of functions g_i ?
 - If so, what is function $f(g)$?



Resubstitution



Resubstitution

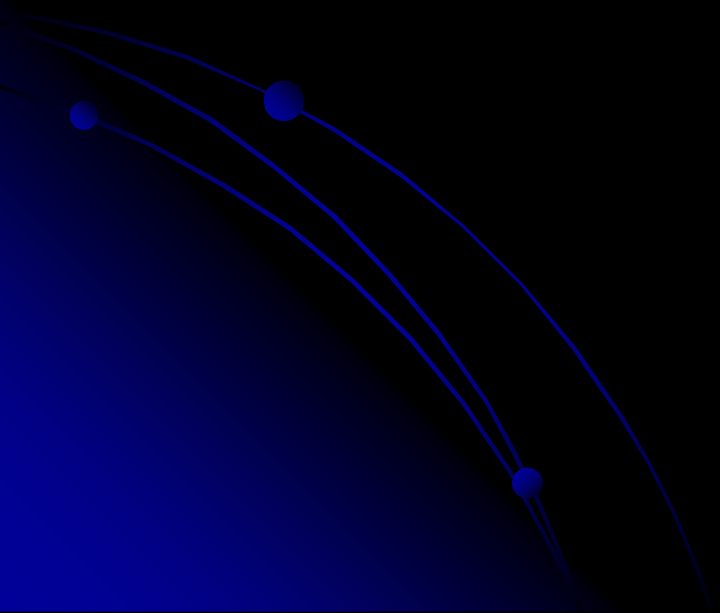
- It can be argued that most of the logic synthesis transforms are special cases of resubstitution
 - Developing a robust (simulation-based) resubstitution engine can solve many of the disparate synthesis tasks
 - Some steps in this direction have been made
 - A lot more can be done in the new flow
- 

Applications in Logic Synthesis

- Optimization of a large network
 - Similar to AIG rewriting
 - Iterate through each node and try to resub it
 - Small runtime budget for each node
- Solving a localized Boolean problem
 - Useful in placement-aware resynthesis, etc
 - Consider one or several nodes and resub them
 - Larger runtime budget for each node is possible

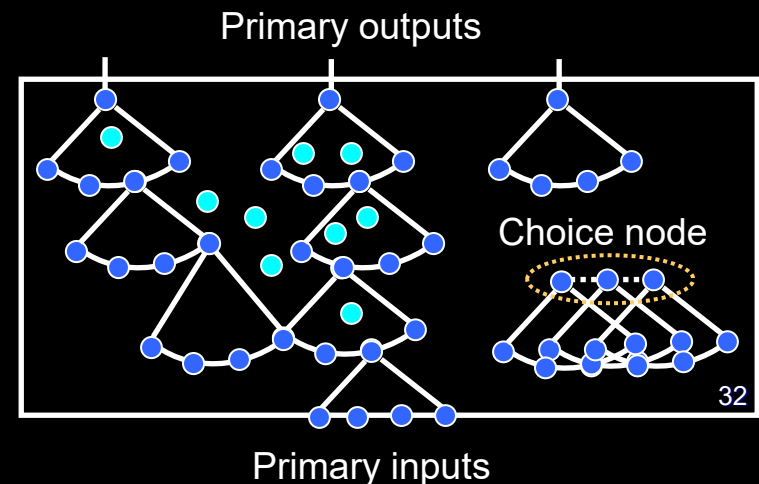
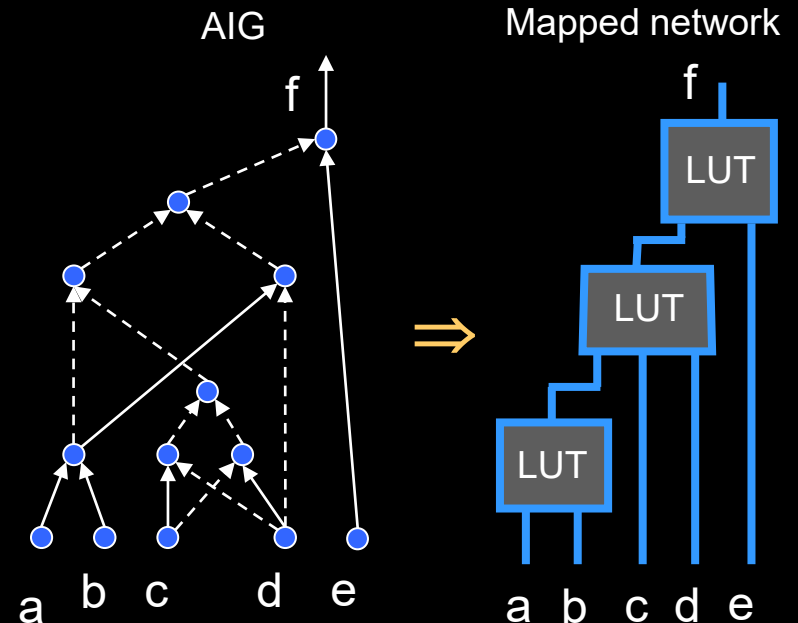
Topics Revisited

- Word-level to bit-level conversion
- Hierarchical bit-level circuit representation
- Computation engines (simulator, SAT solver, etc)
- Technology-independent logic synthesis
- Mapping/retiming used as a single transform
- Verification and ECO



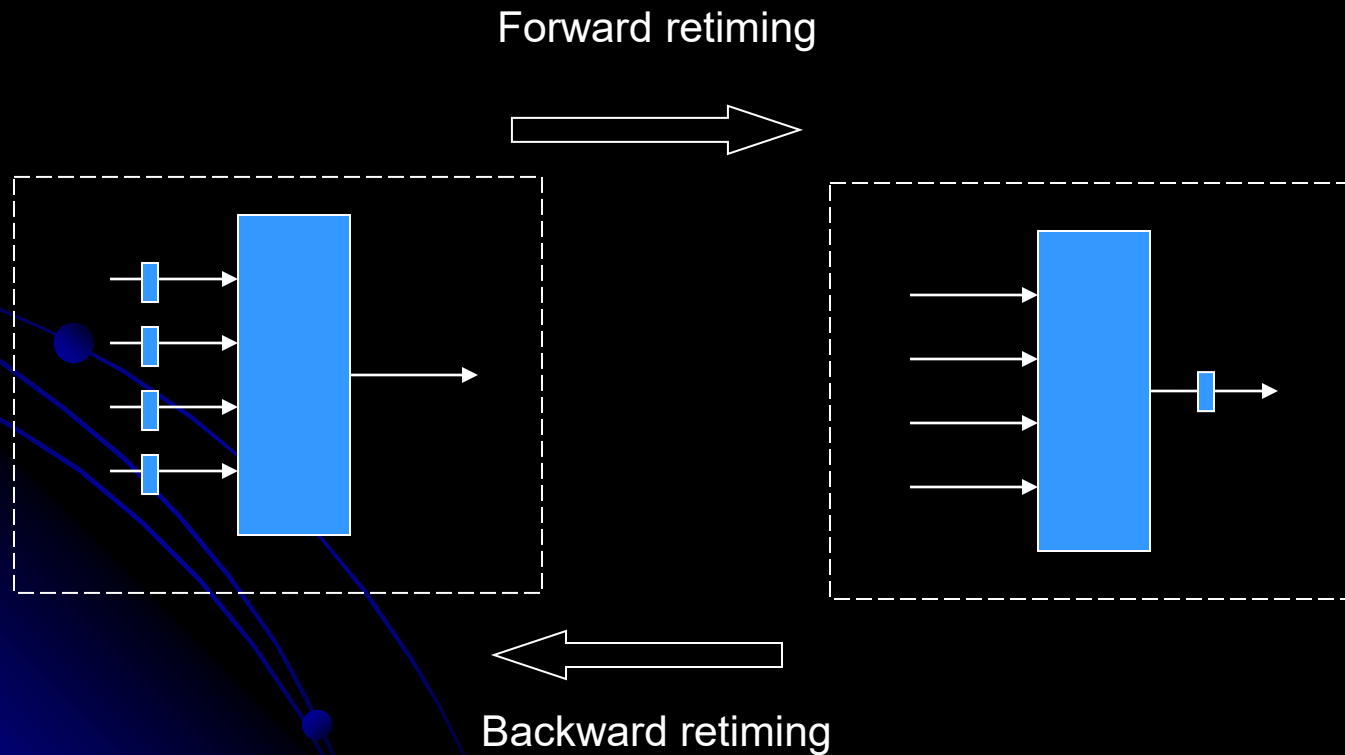
Mapping in a Nutshell

- **AIGs represent logic functions**
 - A good subject graph for mapping
- **Technology mapping expresses logic functions to be implemented**
 - Uses a description of a technology
- **Technology**
 - Primitives with delay, area, etc
- **Structural mapping**
 - Computes a cover of AIG using primitives of the technology
- **Cut-based structural mapping**
 - Computes cuts for each AIG node
 - Associates each cut with a primitive
 - Selects a cover with a minimum cost
- **Structural bias**
 - Good mapping cannot be found because of the poor AIG structure
- **Overcoming structural bias**
 - Need to map over a number of AIG structures (leads to choice nodes)



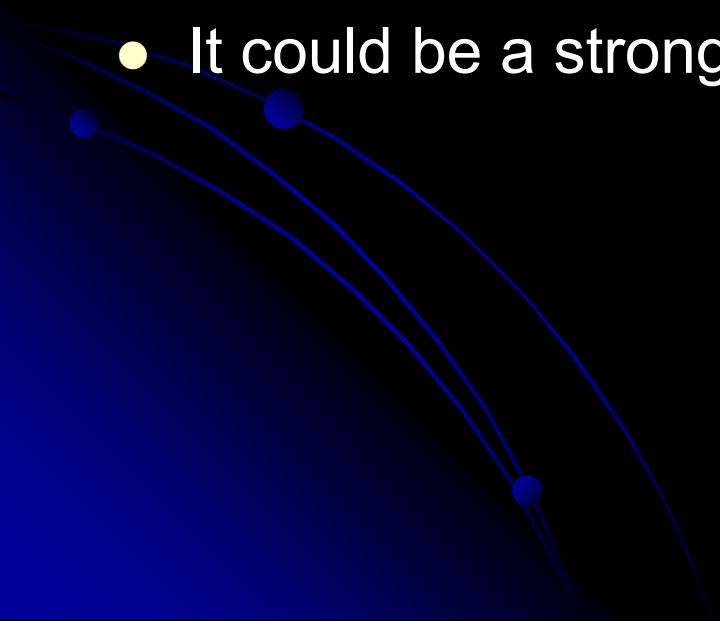
Retiming

- Retiming moves registers across logic nodes to reduce delay or area



Mapping + Retiming

- A methodology for performing mapping and retiming together was proposed 25 years ago by Peichen Pan
 - As opposed to mapping followed by retiming, or vice versa
 - The benefit is up to 25% delay reduction vs a typical 5%
 - Almost no area penalty!
- The potential of this has never been fully explored
- It could be a strong differentiator for the next-gen flow



Experimental Results

No retiming

	FPGA	SC
1. mapping (M)	1.0	1.0
2. integrated choices and mapping (MC)	.96	.95

Retiming, no choices

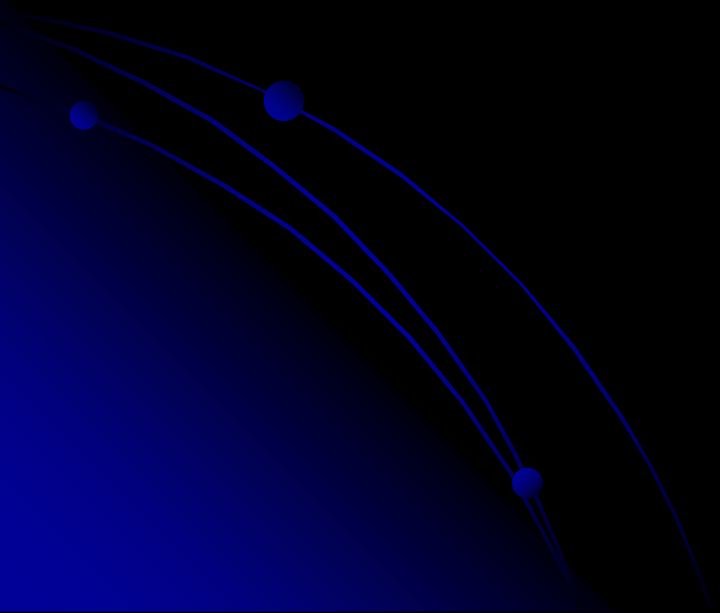
3. mapping followed by retiming (M+R)	.97	.96
4. integrated mapping and retiming (MR)	.82	.84

Retiming and choices

5. mapping with choices followed by retiming (MC+R)	.95	.91
6. integrated choices, mapping, and retiming (MCR)	.74	.76

Topics Revisited

- Word-level to bit-level conversion
- Hierarchical bit-level circuit representation
- Computation engines (simulator, SAT solver, etc)
- Technology-independent logic synthesis
- Mapping/retiming used as a single transform
- Verification and ECO

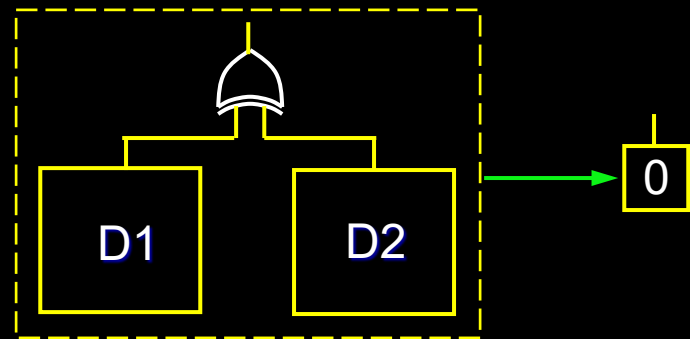


Formal Verification

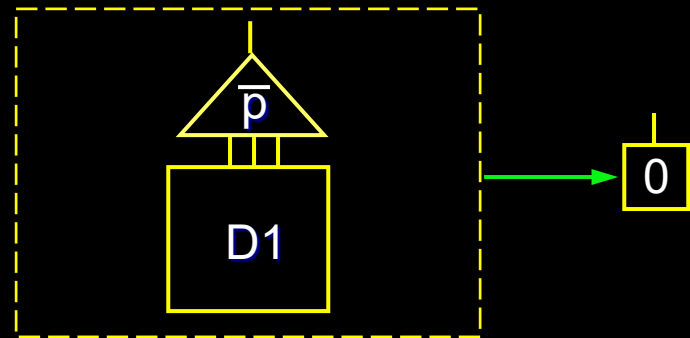
- **Equivalence checking**
 - Takes two designs and makes a miter (AIG)
- **Model checking**
 - Takes design and property and makes a miter (AIG)

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

Equivalence checking



Property checking



Better Verification

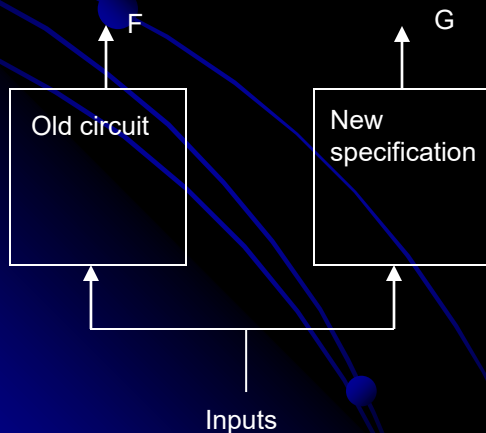
- Verification is not just CEC (combinational equivalence checking)
- There is a lot more to it
 - Sequential designs
 - Designs with memories
 - Verifying arithmetic blocks
- Robust integration of these features is currently missing and can be developed in the new flow

Better ECO

- Problem statement:
 - Synthesize a minimal patch to update an already implemented design (old circuit) after specification has changed
- Solution can be based on the next-gen flow

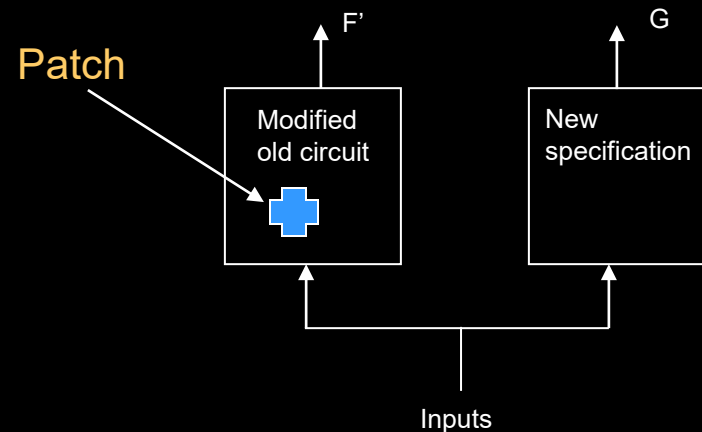
Before ECO

$$F \neq G$$



After ECO

$$F' = G$$



How to Begin Your Own Research

- Take a logic synthesis course at your university or find course materials available online at other universities
- Study public-domain systems, for example:
 - MiniSAT <http://minisat.se/>
 - AIGER <http://fmv.jku.at/aiger/>
 - SIS <https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/sis/index.htm>
 - ABC <https://github.com/berkeley-abc/abc>
 - Yosys <https://github.com/YosysHQ/yosys>
 - Replace <https://github.com/The-OpenROAD-Project/RePIAce>
- Learn how to develop CAD tools by implementing simple basic computations
 - Simple SAT solver
 - Simple SOP minimization program
 - Simple BDD package
 - Decomposition of functions using truth tables

(Any of these can be a beginning of a big project, leading to a Ph.D. or a startup)
- Participating in the next IWLS context
 - Learn about the 2022 contest <https://github.com/alanminko/iwls2022-ls-contest>
 - Follow announcements (in about 6 months) and participate in the 2023 contest
- Read a paper and implementing it
- Send me an email telling about your interests
 - Let us together decide what project you can work on

Example of a Good Project

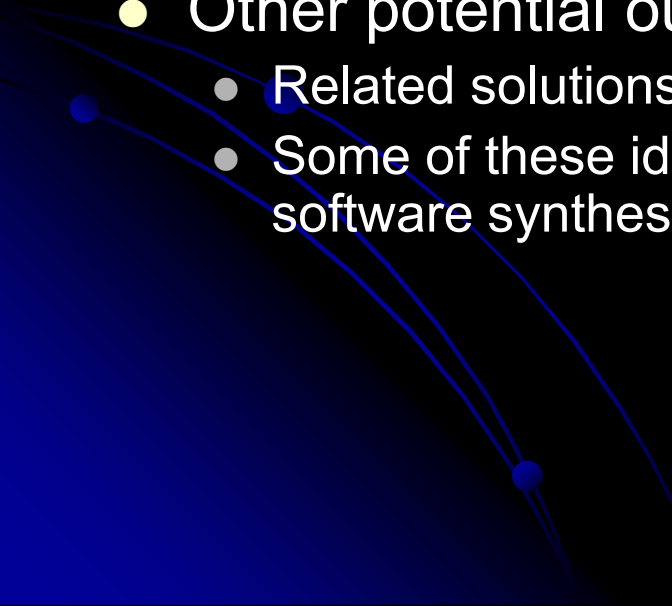
- Well-tuned version based on MiniSAT

```
abc 01> &r corrsrm06.aig; &sat -v -C 100
CO =      98192  AND =    544369  Conf =    100  MinVar =   2000  MinCalls =   200
Unsat calls 32294 ( 32.89 %)  Ave conf =     4.6  Time =     2.12 sec ( 15.35 %)
Sat   calls 65540 ( 66.75 %)  Ave conf =     0.6  Time =     9.38 sec ( 67.82 %)
Undef calls   358 (  0.36 %)  Ave conf =   101.6  Time =     0.98 sec (  7.08 %)
Total time =    13.83 sec
```

- Version based on circuit-based solver

```
abc 01> &r corrsrm06.aig; &sat -vc -C 100
CO =      98192  AND =    544369  Conf =    100  JustMax =   100
Unsat calls 31952 ( 32.54 %)  Ave conf =     3.3  Time =     0.12 sec ( 14.51 %)
Sat   calls 65501 ( 66.71 %)  Ave conf =     0.3  Time =     0.42 sec ( 52.77 %)
Undef calls   739 (  0.75 %)  Ave conf =   102.3  Time =     0.20 sec ( 24.48 %)
Total time =     0.80 sec
```

Conclusion

- A new synthesis / verification flow is well-motivated
 - With key improvements in several related areas
 - Our primary target area is FPGA synthesis
 - Secondary target is ASIC synthesis
 - The flow can be based on ABC
 - With Yosys integrated as an RTL elaborator
 - Other potential outcomes
 - Related solutions for verification and ECO
 - Some of these ideas could also be used as building blocks of a software synthesis/verification/debugging flow
- 

Abstract

- Given the progress achieved over the last fifty years in logic synthesis and verification, it is tempting to believe that most of the research discoveries have already been made, and the role of future researchers and engineers is just to maintain the CAD tools and occasionally make small changes, such as adding concurrency or employing machine learning to generate better scripts. Nothing could be farther from the truth.
 - In this talk, we explore several orthogonal innovations in the fundamental research used to build synthesis tools targeting FPGAs and ASICs. These innovations include using novel data structures, leveraging synergistic optimization engines, and simultaneously exploring previously-unrelated search spaces. Most of these improvements are work-in-progress with early results demonstrating better quality and faster runtimes.
- 