EE 382M-11
The University of Texas at Austin
Department of Electrical and Computer Engineering

Dr. Xiushan "Shaun" Feng
Formal Verification Group Leader,
Samsung Austin Research Center (SARC), Austin, TX
Samsung Advanced Computing Lab (ACL), San Jose, CA
Samsung G2 Design Center, Gyeonggi-do, Korea

Jan 28, 2021

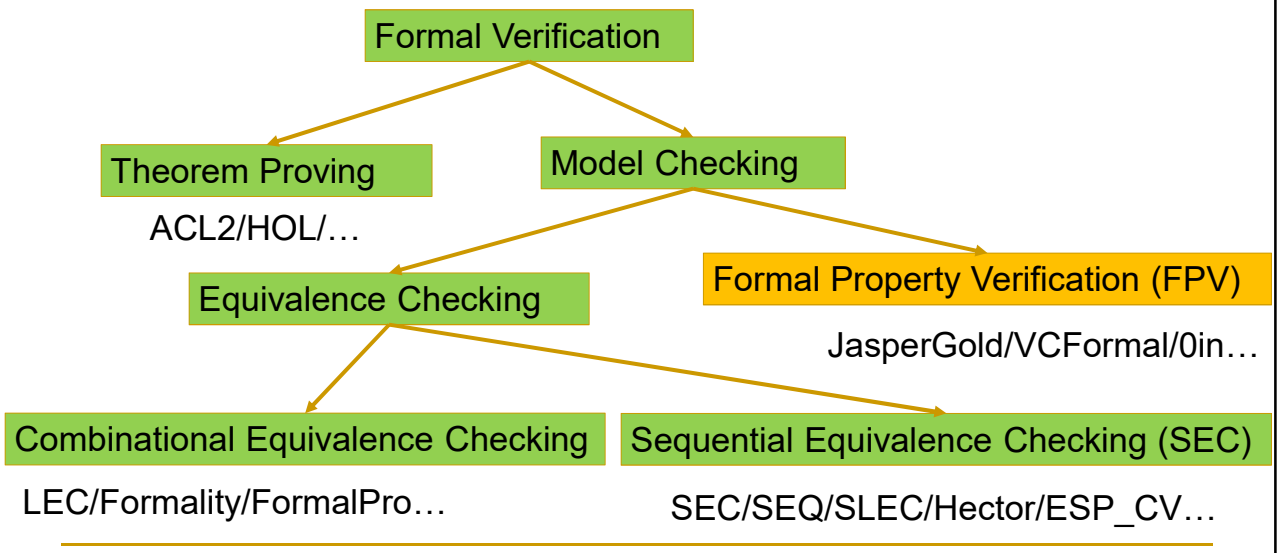
1

Introduction

- About myself
  - PhD degree, Computer Science Dept., UBC, Vancouver, Canada
  - Led formal verification at Samsung/Nvidia/Oracle/Freescale/AMD
  - Built formal verification teams at Samsung from scratch
  - About 20 years experience in formal verification
- Three lectures
  - Formal Equivalence Checking (Jan 28)
  - Symbolic Trajectory Evaluation, Term Rewriting (Feb 11)
  - Sequential Equivalence Checking (April 1)

2

What is Formal Verification?



3

Who Are Using Formal Verification?

- Software companies
  - Microsoft
    - SLAM: static driver verifier
    - VeriSol: Azure Blockchain smart contracts
    - SLAyer: memory safety of C programs
    - …
  - Google: DeepDive, static analyzer for vulnerabilities of android apps…
  - Facebook: INFER for mobile apps (Instagram/facebook/Msger)…
  - Amazon: formal reasoning on cloud security(AWS)…
- Hardware design companies
  - Intel, Apple, Nvidia, ARM, Samsung, ….: clock gating, floating points, formal properties, protocols,…
  - Google, Facebook, Amazon…: secret HW projects

4

4. Formal Equivalence Checking

Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin

Verification of Digital Systems
Spring 2020

January 30, 2020

Outline

- Review, representing logic
- Application of formal equivalence checking
  - Basics
  - Tool for equivalence checking
- Dealing with complexity in equivalence checking
- Things to watch out for in equivalence checking
- Functional partitioning

Acknowledgments: Jim Bitner (UT), Jawahar Jain (UT, Fujitsu Labs. and Samsung), Shahrzad Mirkhani (UT), Erik Seligman (Intel/Portland State University)

ECE Department, University of Texas at Austin Lecture 4. Formal Equivalence Checking Jacob Abraham, January 30, 2020 1 / 52

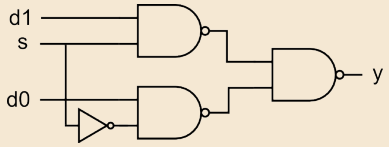
Equivalence Checking

- Validate that the implementation of a module is consistent with the specification
  - Can use simulation or formal techniques
  - Combinational or sequential modules

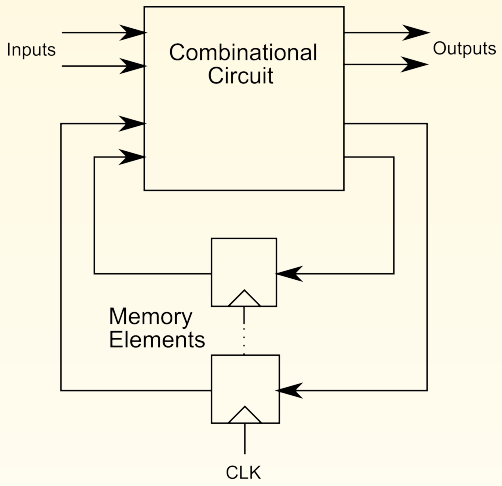
Example: Specification in RTL

```
module mux(input s, d0, d1,
           output y);
assign y = s ? d1 : d0;
endmodule
```

Example: Implementation at the gate level



Huffman Model of a Sequential Circuit



If the specification and implementation have identical memory elements, need to only check equivalence of the combinational portions

Formally Checking the Equivalence of Two Modules

- Canonical representations for combinational equivalence checking (why canonical?)
  - Tables
  - Equations
  - Graphs
- Representations for sequential equivalence checking
  - State and transitions
  - Tables, equations, graphs
- Issues
  - Complexity
  - Ease of automation

Some Boolean Connectives

$(P \rightarrow Q) := (\overline{P} + Q)$

$(P \oplus Q) := ((\overline{P} \cdot Q) + (P \cdot \overline{Q}))$

$(P \leftrightarrow Q) := ((P \cdot Q) + (\overline{P} \cdot \overline{Q}))$

$f(x_1, \ldots, x_i := 1, \ldots, x_n)$ is called the cofactor of f with respect to x_i, written $f|_{x_i}$

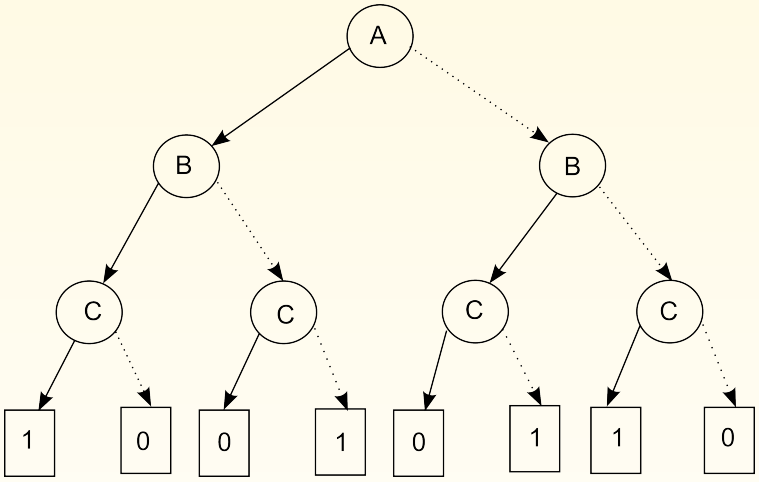
$f(x_1, \ldots, x_i := 0, \ldots, x_n)$ is called the cofactor of f with respect to $\overline{x_i}$, written $f|_{\overline{x_i}}$

Shannon Expansion: $f(x_1, \ldots, x_i, \ldots, x_n) = (x_i \cdot f|_{x_i} + \overline{x_i} \cdot f|_{\overline{x_i}})$

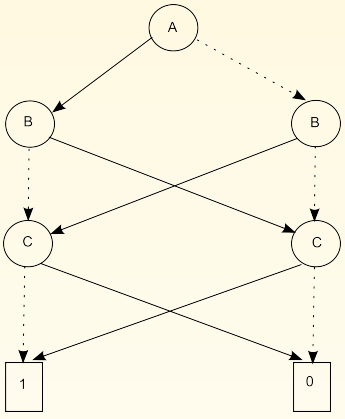
Example: $f = a\ b\ c + \overline{a}\ \overline{c}$ (Note: $\cdot$ implicit)
$f|_a = b\ c$, $f|_{\overline{a}} = \overline{c}$
$f = a(b\ c) + \overline{a}(\overline{c})$

Decision Tree for $A \oplus B \oplus C$

In order to prove that two Boolean functions are equivalent, they must be represented canonically (Why)?



Reduced, Ordered BDD (ROBDD)



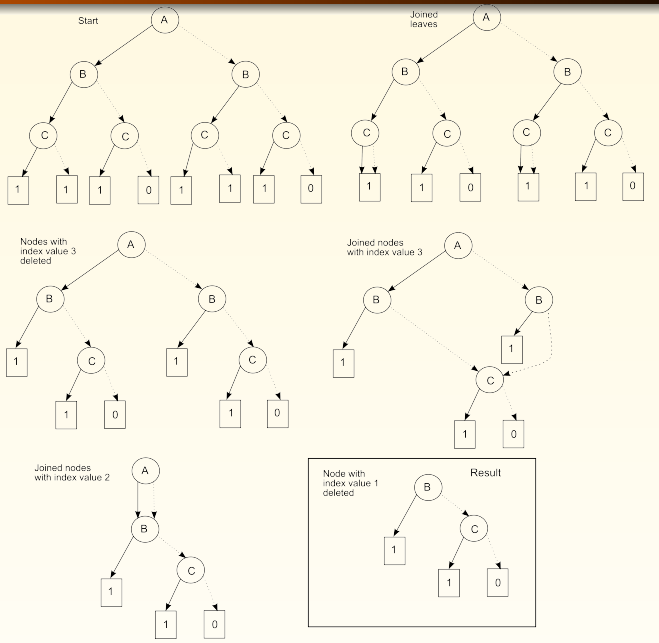
$F = A \oplus B \oplus C$

Reduced, Ordered BDDs (ROBDDs) are canonical

Can represent sets of states, state-transition relations, etc.

Structure and complexity of ROBDDs for *Symmetric Functions?*

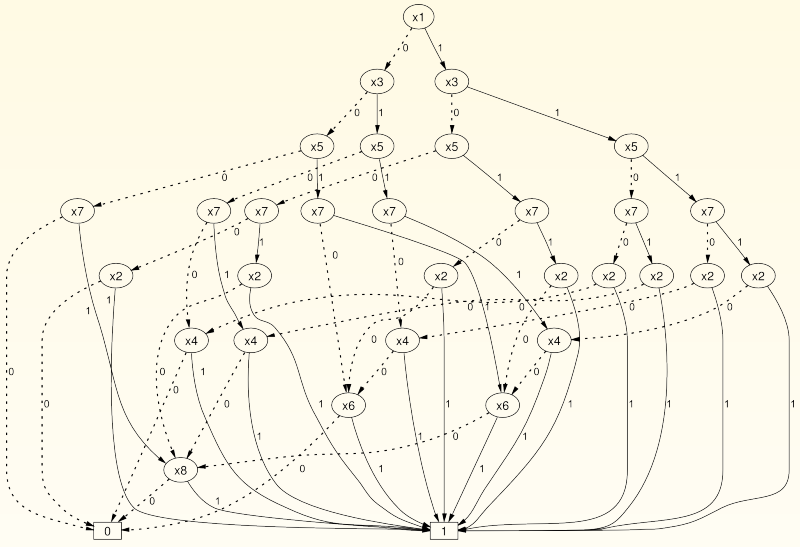
Example of ROBDD Reduction



Impact of BDD Variable Ordering

$$f(x_1, x_2, \ldots, x_8) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 + x_7 \cdot x_8$$

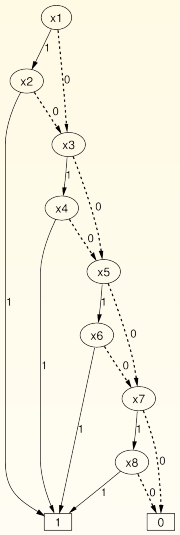
$$Ordering : x_1 < x_3 < x_5 < x_7 < x_2 < x_4 < x_6 < x_8$$



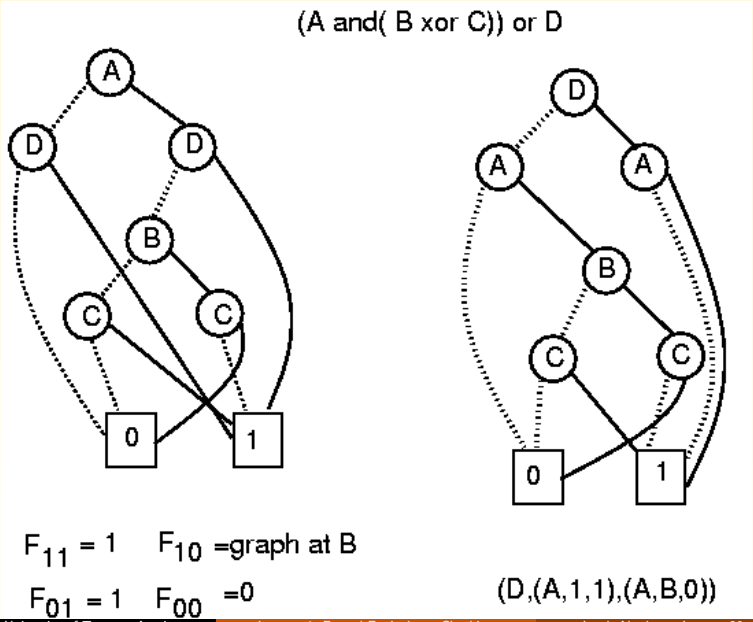
Impact of BDD Variable Ordering, Cont'd

$$f(x_1, x_2, \ldots, x_8) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 + x_7 \cdot x_8$$

$$Ordering : x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8$$



Variable Swapping – An example



Representing Functions and Relations

A relation can be used to represent a function, such as
$y \leftrightarrow f(x_1, \ldots, x_n)$
Example, consider the relational representation for an AND gate
$y \leftrightarrow x_1 \wedge x_2$

x_1	x_2	y	$y \leftrightarrow x_1 \wedge x_2$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

This representation for a function, in CNF, is used in SAT clauses

Satisfiability (SAT) Solvers

Can a Boolean Function be Satisfied?

- Cast an equivalence checking problem as a SAT problem
- Starts by converting Boolean formula into the Conjunctive Normal Form (CNF) – (product of sums)
  $(a + b + c)(a + \overline{e} + f)(\overline{c} + \overline{d} + g) \ldots$
- Goal is to find an assignment satisfying every term (if any clause is 0, there is no satisfying assignment)
- Commercial and Open SAT solvers available
- Most verification tools now use BDDs + SAT
- Some bring in ATPG ideas – called "structural SAT"

Truth Table to CNF

- Put negation of formula in DNF
  - For each "0" or "F" row in table, make a term equivalent to the corresponding assignment
- Negate the disjunction of the terms
  - By DeMorgan's Law, switch AND and OR, and complement literals

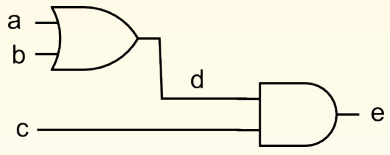
Example: Express $x \leftrightarrow y \quad (x \cdot y + \overline{x} \cdot \overline{y})$ in CNF

- Two terms for "0": x=1, y=0 and x=0, y=1
  $\Longrightarrow$ function is "0" when $x\overline{y} + \overline{x}y$
- CNF is: $(\overline{x} + y)(x + \overline{y})$

Circuit to CNF

$d \equiv (a + b)$

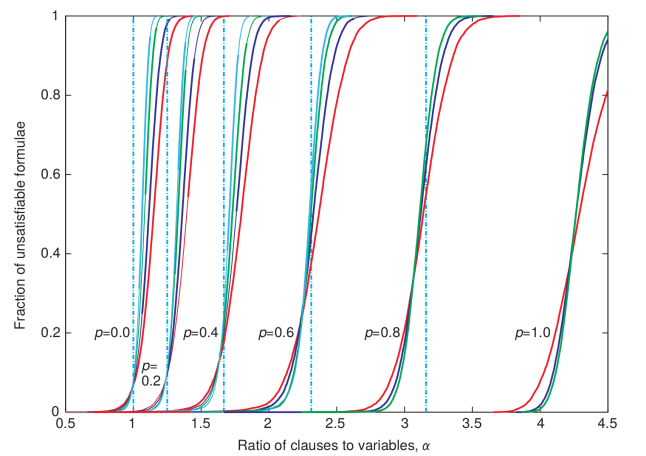
Clauses:
$(a + b + \overline{d})$
$(\overline{a} + d)$
$(\overline{b} + d)$



$e \equiv (c.d)$

Clauses:
$(\overline{c} + \overline{d} + e)$
$(d + \overline{e})$
$(c + \overline{e})$

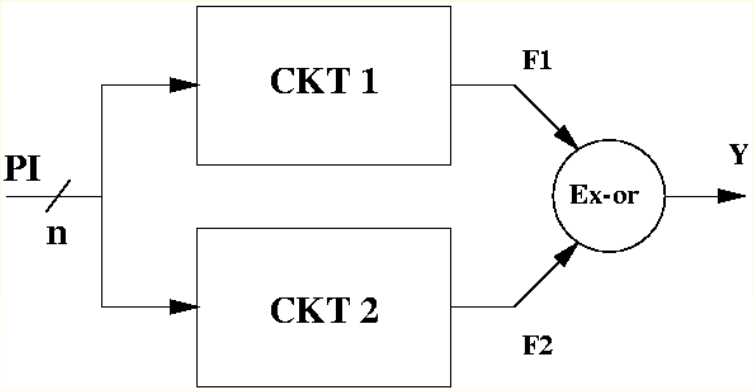
Complexity of SAT



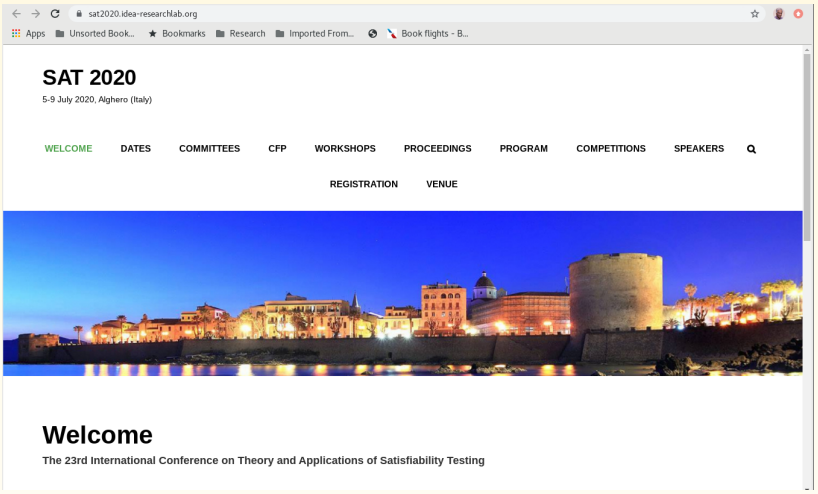
Monasson et al., "Determining computational complexity from characteristic 'phase transitions'," ture, vol. 400, 8 July, 1999

Equivalence Checking as a Satisfiability Problem

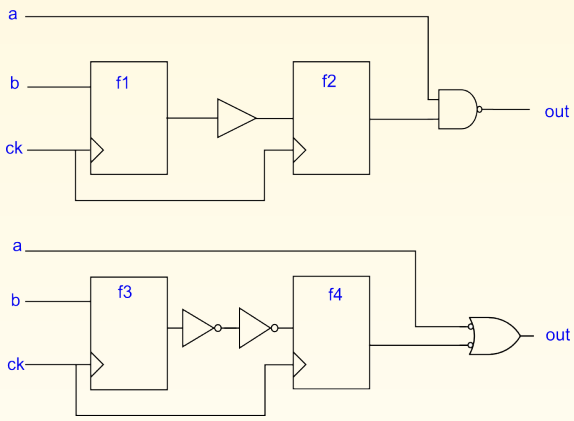
- Forcing a 1 at Y (for some input combination) will prove inequivalence of the two circuits
- Approach is not memory limited (like BDDs)



SAT Competitions

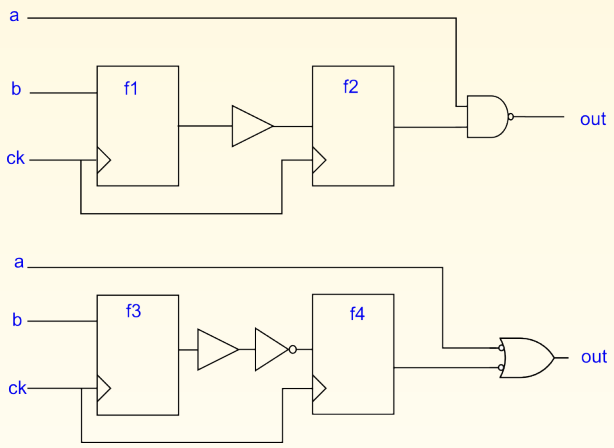


Are the Two Circuits Equivalent?



1. Map key points: inputs, outputs, f1 $\leftrightarrow$ f3, f2 $\leftrightarrow$ f4
2. Build equations: f1 $=$ b, f2 $=$ f1, out $= \overline{(a \cdot f2)}$
   f3 $=$ b, f4 $= \overline{\overline{(f3)}}$, out $= \overline{a} + \overline{f4}$
3. Compare equations: show logic of output signals is the same

Erroneous Design



In this case, the comparison of the equations will show that
$(f1 = b = f3)$ for the both circuits, and
$(f2 = f1 = f3)$ for the top, while $(f4 = \overline{f3})$ for the bottom circuit
which means, $f2 \neq f4 \implies$ ERROR

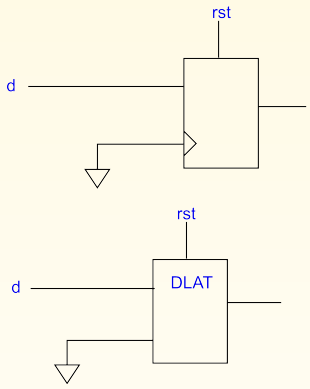
Debugging – Where to Look

- Fanin cones (*support set*)
  - Different fanin $\implies$ major issue
- Set of counterexample values
  - If only specific values cause an error, provides hint of root cause
- "Intelligent" hints from tools
  - Is an overall inversion suspected?
  - Identify similar areas of logic within cone?
  - Isolate error

Model Flattening

Minor exceptions to state matching – useful if flops/latches don't map

Example:



`set_parameters -flatten` *design*

Constraints in Equivalence Checking

Example of the need for constraints

RTL is often general

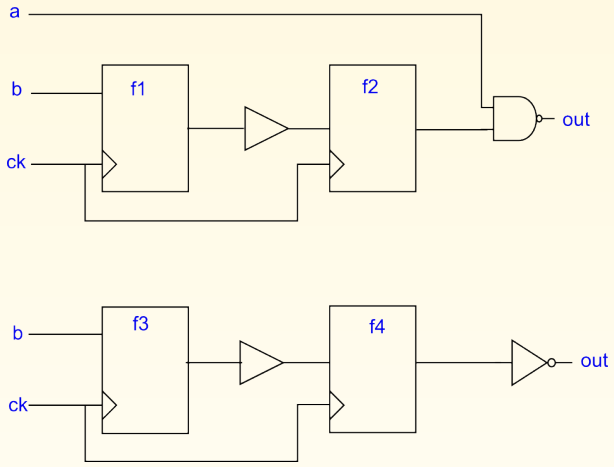
```
‘ifdef CHIP_VERSION_1
‘define A 0
‘else
‘define A 1
‘endif
```

When reusing part of the design:

```
assign A = 1b1;
...
if (!A) ...
```

Irrelevant RTL remains

Are the Two Circuits Equivalent?



No, but what if a is always 1?

Need to specify constraints to the equivalence checking tool

```
set_constant r:/WORK/a 1
```


Why Constraints Matter

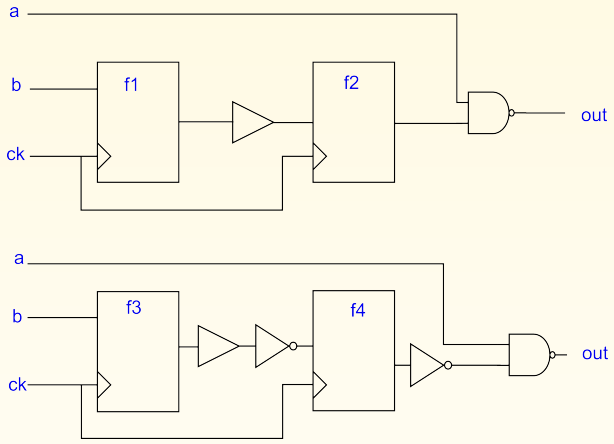
- Good synthesis tools take advantage of specified constraints
  - Assume constants to reduce size/scope
  - Don't synthesize masked-out RTL
  - Allow out-of-band constraint specifications in control files
- Equivalence checking tools must recognize constraints
- Otherwise, will get spurious mismatches
- No effort **if** constraints are visible at the equivalence checking level
  - But may be only in wrapper RTL
  - Or inside analog blackbox
  - Or could be due to software/outside specifications
- If not visible to tool, may need to specify it
  - Add constraints (on pin, between values, etc.)

Synthesized Netlists

- Synthesized netlists built from cell library
- Cells hide transistor-level logic
  - Delivered with behavioral descriptions
  - Library developers certify correctness
- Dealing with custom cells is difficult
- Checking full block at the transistor level is expensive
  - Formal verification of custom cells is a separate process
  - Tools have been developed for automatically extracting logic (and, in one case, RTL) descriptions from transistor-level designs
- Users may need to annotate transistor-level information when using an logic equivalence checking tool
  - Transistor signal flow directions
  - Nodes meant to hold state
  - Domino precharge nodes

State Negation

Are the two circuit below equivalent?

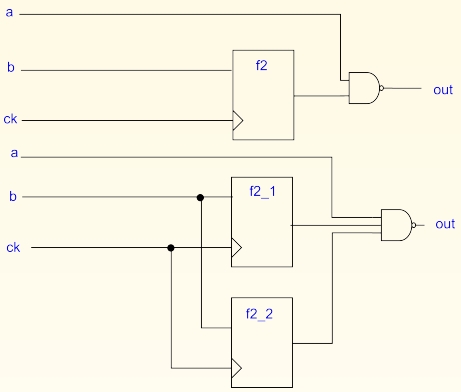


Yes: with State Negation (state f2 = ¬f4)

```
set_user_match type cell r:/WORK/f2 i:/WORK/f4 -inverted
```

State Replication

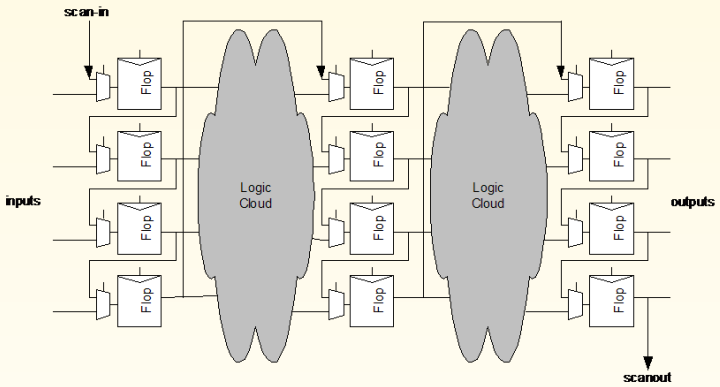
Are the two circuit below equivalent?




```
set_user_match -type cell r:WORK/f2 i:WORK/f2_1 i:WORK/f2_2
```


Scan Chains

Scan chains used for silicon bring-up and manufacturing test



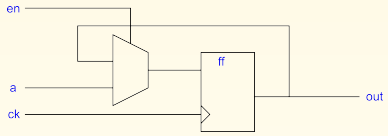
Use commands `guide_scan_input guide_scan_output`

Handling Scan Chains in Equivalence Checking

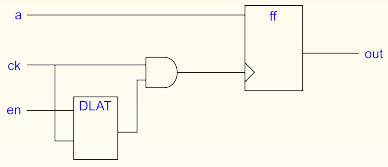
- Need to be careful – scan is in the netlists, not in RTL
  - Scan insertion during synthesis
- Identify scan enable conditions
  - May be simple scan enable pin
  - Or a combination of pins and states
- Use constraints to disable in netlist so that tool can deal with scan
- This could be a verification hole – need to be addressed
  - Gate-level simulation of netlist
  - Or custom scripts to walk chains

Clock Gating

- Another verification issue
- Stop clock to inactive flip-flops
  - No clock asserted $\Longrightarrow$ no switching power
- Automatically inserted in synthesis
  - Thus, in netlist, but not in RTL



Enabled flop



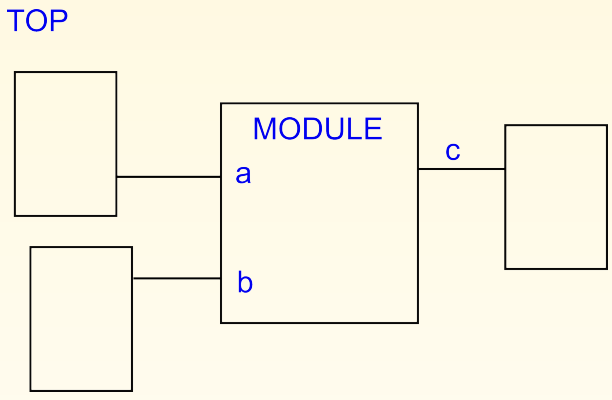
Enabled flop with clock gating

```
set verification_clock_gate_hold_mode low
```

Black Boxes

- Formal tool cannot deal with some blocks
- This logic is ignored by the tool
- Usually a Verilog module instance
- Examples
  - Analog circuits
  - "Hard IP" – externally supplied block (no alternative but to "trust" it)
  - Divide and conquer in large designs – different ownership of particular block
  - Large embedded memories (register files, caches, etc.) and multipliers – use specialized tools on them
- What is verified then?
  - Drivers of black-box input pins
  - Receivers of black-box input pins
  - Internals of black box are ignored – be careful!

Black Box Example

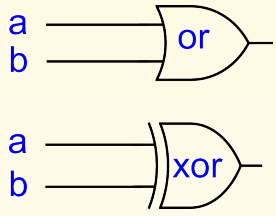


- TOP.MODULE is a single key point
  - But mapped only if a, b, and c have matches
  - Verify fails if logic driving a or b mismatches

```
set_black_box r:WORK/TOP/MODULE
set_black_box i:WORK/TOP/MODULE
```

Don't Care Spaces

Are these two equal?



Suppose source RTL is:

```
case ({a,b})
2b00:  out=0
    2b01:  out=1
    2b10:  out=1
endcase
```

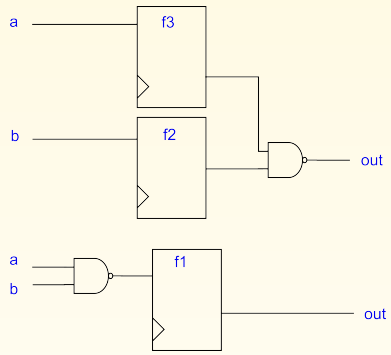
Unspecified case is a Dont-Care (DC)

Don't Care Cases

- Often result from underspecified RTL
- Synthesis has freedom to choose values
  - Can optimize for area, timing, etc.
- Formal tools can handle automatically
  - But, don't cares only come from *golden* model
  - Don't care in *netlist* model is an error
  - Asymmetry between *golden* and *netlist* models
- Sometimes, RTL can match two different netlists
- But, verifying the netlists against each other may produce an error

Pipeline Retiming

Are the two circuits equivalent?



In a sequential sense, the circuits are equivalent
Logic has been moved across a flop – pipeline retiming
`set_parameters -retimed` *design*

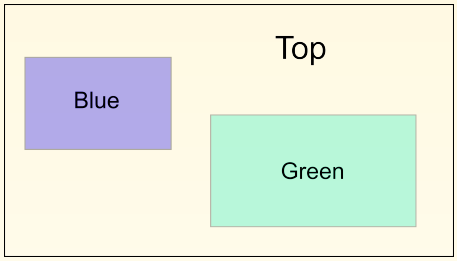
Pipeline Retiming and Equivalence Checking

- Retiming violates state matching
  - Should not expect generic equivalence tools to handle this
- Recent tools can handle some cases
- Tools are aware of synthesis techniques
  - Internally "push" logic along pipeline to match
- Many limitations, need to be careful
  - Retiming must be isolated to one module
  - Can cause runtime/memory complexity
  - Need sequential equivalence checking tool for more general cases

Dealing with Complexity

- Possible result on very complex module
  - Tool crash
  - Tool "aborts" – could not deal with that logic
- Check tool options to possibly resolve this problem
  - Increase "effort" of tool
  - Check for structures (for example, multiplier) and use different tool/techniques on them
  - Concentrate comparison on each standalone point
- Monitor process for memory blowup
- Run on larger server
- Attempt to parallelize comparisons
  - Using multiple machines on the network

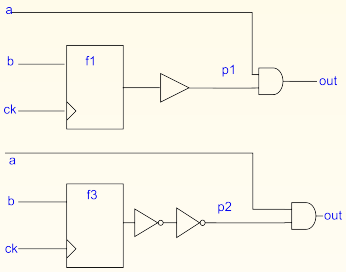
Hierarchical Equivalence Checking



- Verify entire design (Top) if possible
- Otherwise, perform equivalence checking three times
  - TOP.Blue
  - TOP.Green
  - TOP with Blue and Green black-boxed
- What about inputs to the black boxes?
  - Signals may be related (example, set of inputs "one-hot")
- Sometimes need to write wrappers for each black box

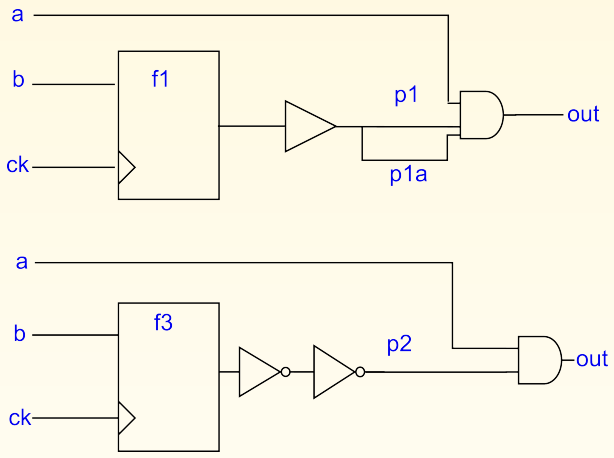
Cut Points

- Problem: verifying large combinational cones
- Solution: Divide logic at points other than states
  - Internal signals may correspond
  - In extreme cases, recode RTL to enable matching non-state points
- Cut point = non-state to treat as key point
  - Map and verify just like latches/flops
  - Reduces logic cones being analyzed



Add rule in tool to match p1 and p2

Possible Cut-Point Problem

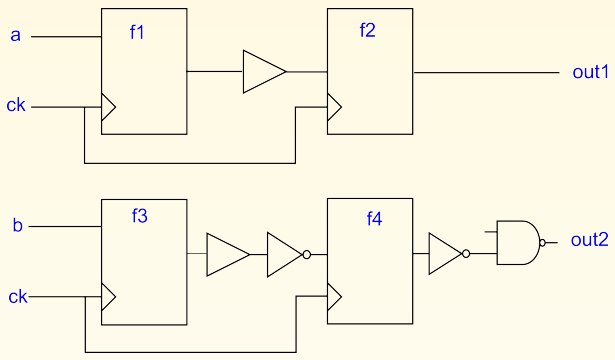


Is p1 still a useful cut point?
Constraint issues
`set_cut_point` *signal*

Case Splitting

- Formal techniques consider all cases together
  - Good tool engines may be smarter
- What if small inputs activate/deactivate lots of logic?
  - Example: mode bits
- Constrain appropriate pins to 1 or 0
  - Then compare twice
  - Or, in general, constrain n bits, then 2^n compares
- We will explore this in more detail in a formal context

Example of Case Splitting



Suppose compare of f2 and f4 aborts, and we want to case-split on f1/f3
First assign a constant 0 to flops
Then assign a constant 1
If both cases pass, circuits are equivalent

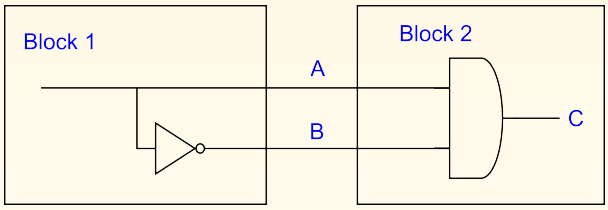
False Positives

A False Positive is a case where the equivalence checking tool incorrectly labels design as equivalent, even when they are not

- Very costly – equivalence checking typically gates next design phase
- Wrong checking answer at tapeout $\Longrightarrow$ silicon respin
- Not just a theoretical concept – has been seen in real designs
- This is not just from a bug in the tool – incorrect specification or use of the tool can lead to this problem
- See some examples in the following slides

Constraints in Equivalence Checking

- Constraints: Reduce set of possible values
  - Turn off scan
  - Known conditions on inputs
  - Eliminate unused state encodings



Constraint that A and B are inverse of each other needed to prove C==0 in Block 2

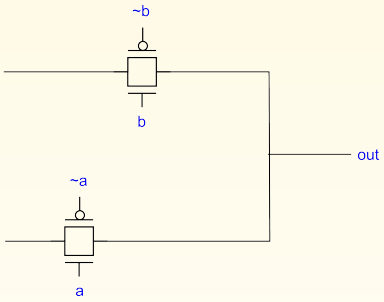
Bad constraints $\Longrightarrow$ False Positive

If constraint also included mapping a and b equal to 1 gives 1 on c (this implies that a and b have to be equal)

All inputs are illegal, so module passes check

Library Cells and Equivalence Checking

- Vendor-supplied libraries of cells used in design
- Logic representations trusted for checking
  - Library needs to be validated (example for assumptions on inputs)

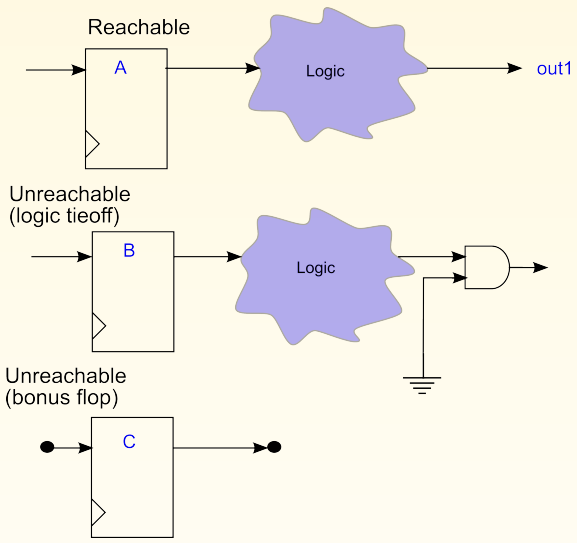


What about contention on the wires shorted together? Is the common point an AND or OR of the signals?
Cell is legal if we guarantee that a and b are complements

Unreachable Points in Equivalence Checking

- Unreachable: flop/latch that cannot affect output
  - Can be logically ignored
  - Must be ignored if not in both models
- Common causes
  - Synthesis optimizations
  - Tied off no-longer-relevant logic
- Must be careful about back end logic
  - Some flows (scan, bonus cells) connected later
  - So "unreachables" may be important!

Reachable and Unreachable Flops



Lost unreachables may result in false positives

Other Items to Watch

- Black boxing logic
  - Ensure that black-boxed logic and interfaces verified somewhere
- RTL language usage
  - Watch for ambiguous Verilog standards
  - Tools may disagree on interpretation
- Ideally, ensure independence between Synthesis and Checking
  - Different results from different vendor tools
  - Avoid Synthesis and Equivalence Checking from same vendor?
  - "Verification diversity" (may not be possible due to costs, etc.)
- Obscure tools behaviors
  - Multiple drivers on a net – report error, or treat as wired (AND or OR)?
  - Set to *wire* unless intention is for multi-drives
  - Input accidentally defined as output (really happened, and was caught during late inspection)
- Multiple checks – example, sanity check of simulation

Conformal LEC (Cadence) – Equivalence Checking Tool

Steps in running the tool

- Read the reference (golden) model
- Read the implementation model (to check)
- Define match points between reference and implementation models
- Verify the design
- Diagnose the error
- Debug the design

See the *Conformal LEC Tutorial* and References in the Lab 1 handout on Canvas

Partitioning Techniques

Orthogonal Partitions

- If F is complemented into regions π_1 and π_2 such that F_{π_1} and F_{π_2} are never *true* at the same time, then π_1 and π_2 form orthogonal partitions
- F can be regarded as a linear sum of functions F_{π_1} and F_{π_2}
- F_{π_1} and F_{π_2} can be evaluated and ordered independently
- Many function pairs, which otherwise would take exponential amount of computational resources for verification, can be efficiently (in polynomial time) verified through such partitions

Functional Partitioning

If F_{π_1} and F_{π_2} are never *true* at the same time, then π_1 and π_2 form orthogonal partitions

- F_{π_1} and F_{π_2} can be evaluated and ordered independently
- Many functions, which otherwise would take an exponential amount of resources for verification, can be verified efficiently (in polynomial time) using orthogonal partitions
- Example, the Fortune-Hopcroft-Schmidt (FHS) function

The FHS function requires an exponential number of ROBDD nodes to represent it. However, if it is partitioned into subfunctions based on the selection logic, each subfunction is easy to represent, and there are only O(n) of these orthogonal functions, allowing easy verification of the circuit