Hardware Verification

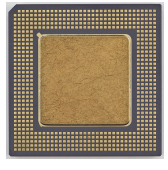
SE206 – Modélisation, génération de code et vérification

Ulrich Kühne

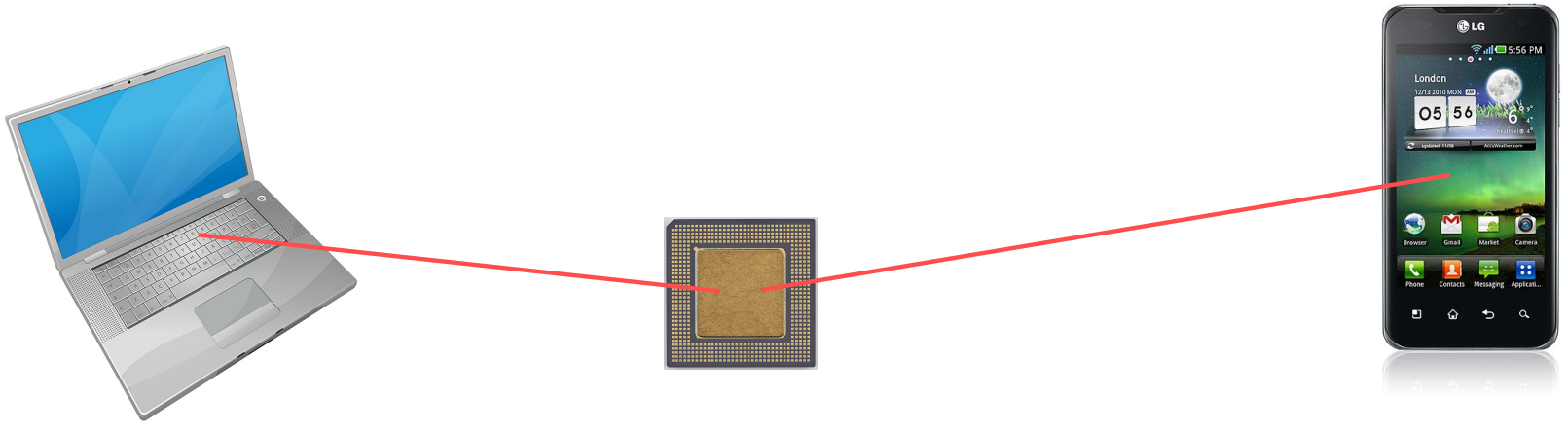
2020/2021

Outline

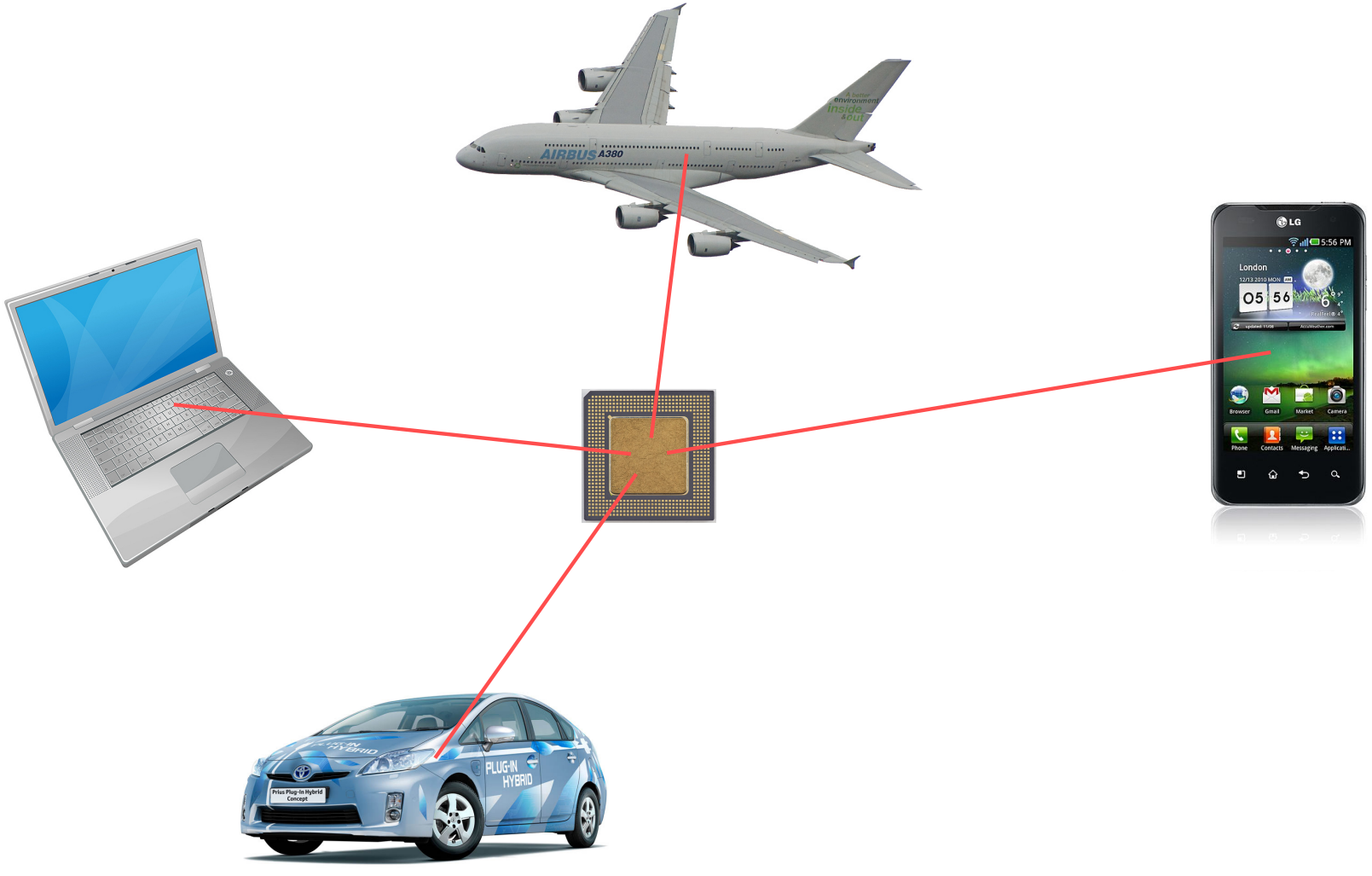
Motivation



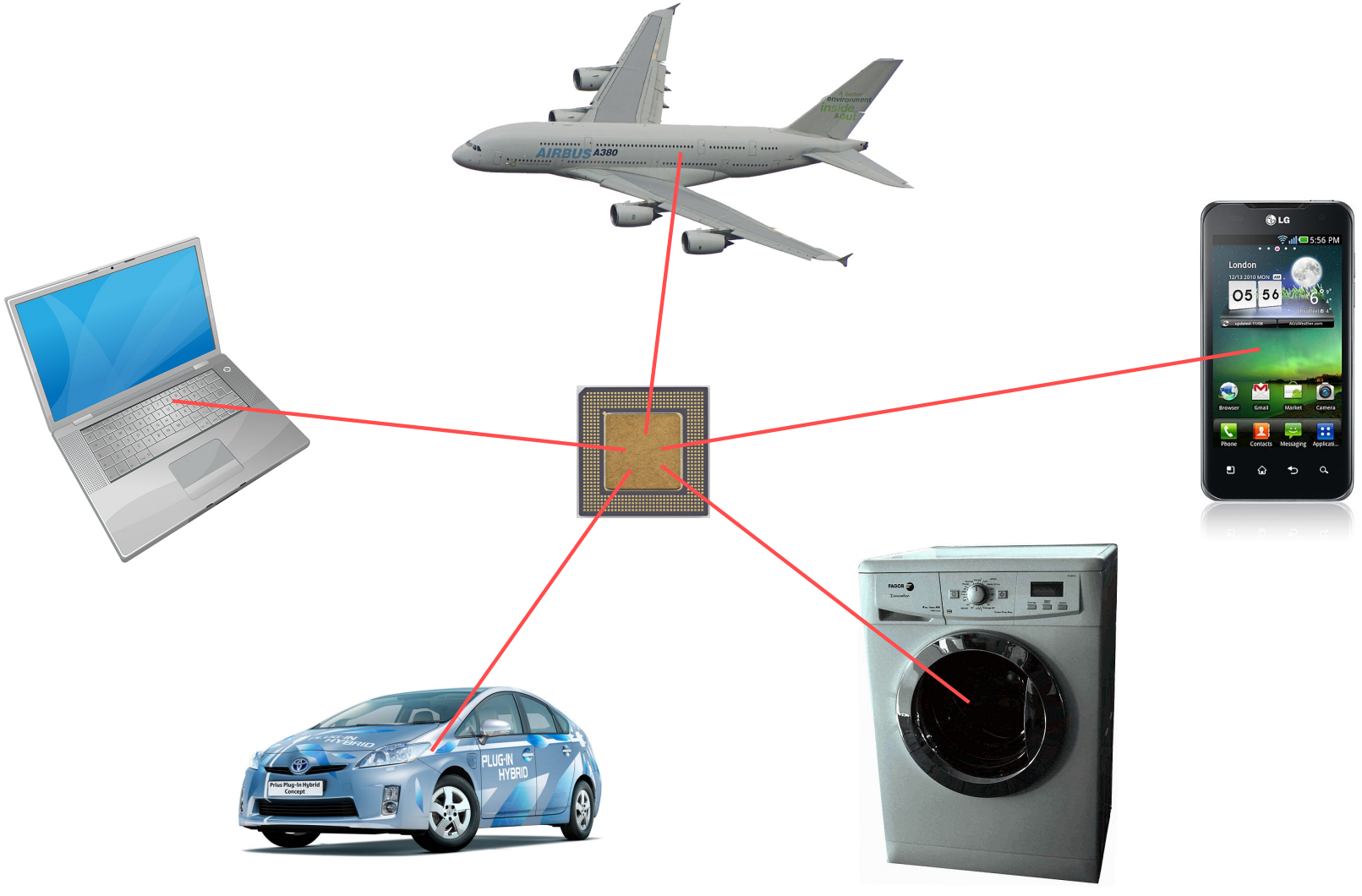
Motivation


LG
5:56 PM
London
05 56
Browser
Gmail
Market
Camera
Phone
Contacts
Messaging

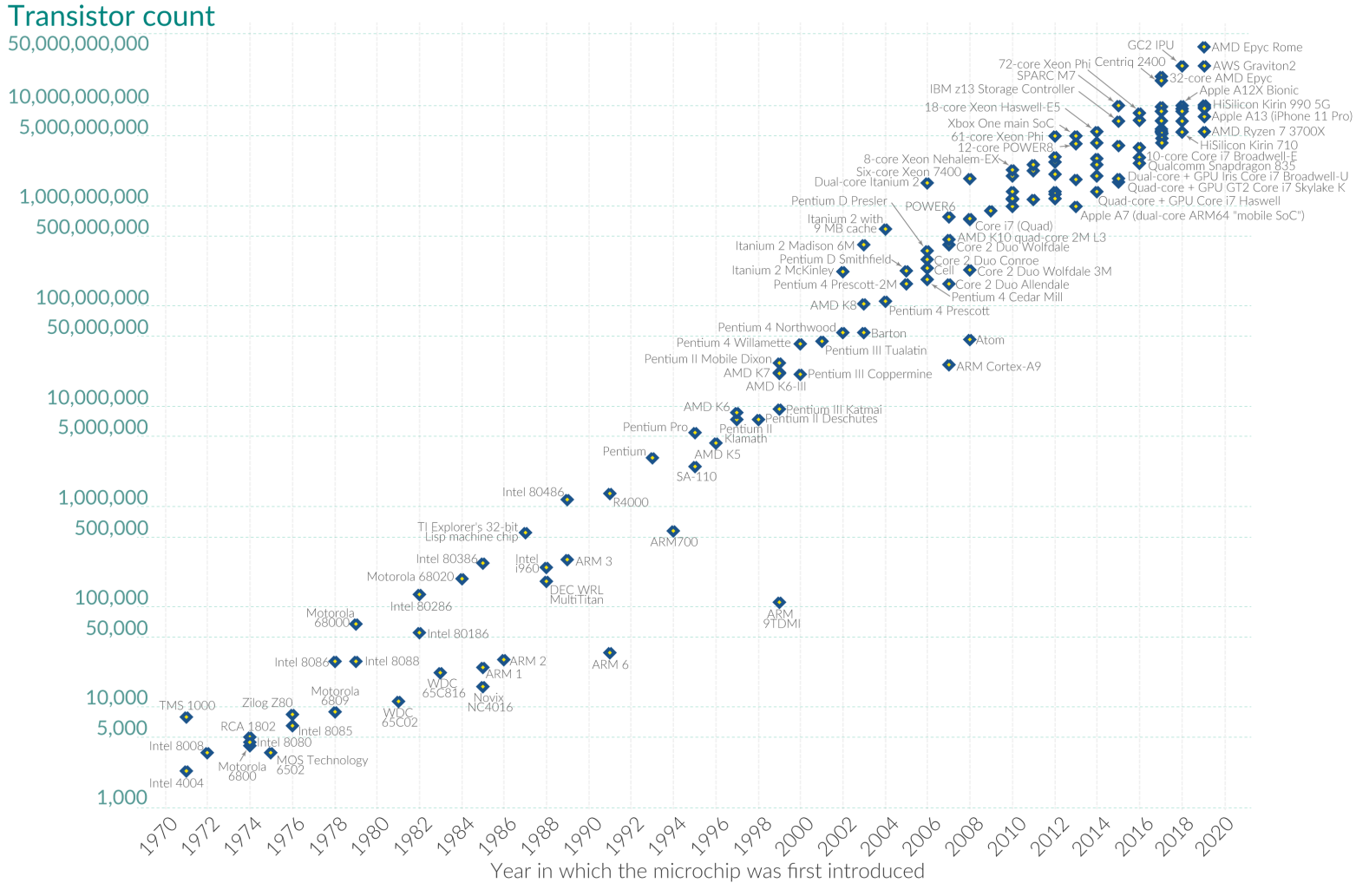
Motivation


AIRBUS A380
PLUG-IN HYBRID
Prius Plug-In Hybrid Concept
LG
5:56 PM
London

Motivation

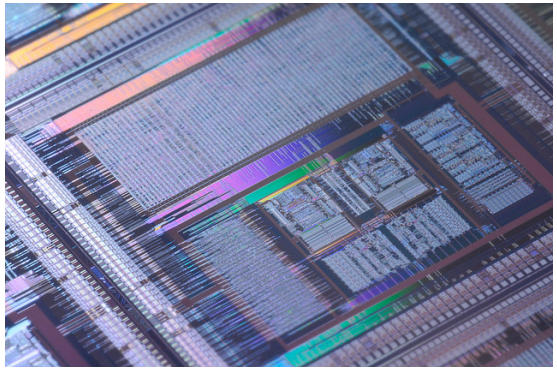

AIRBUS A380
PLUG-IN HYBRID
Prius Plug-In Hybrid Concept
LG
5:56 PM
London
05 56
Browser
Gmail
Market
Camera
Phone
Contacts
Messaging

Motivation



[Source: Wikipedia, CC-BY Hannah Ritchie and Max Moser]

Motivation

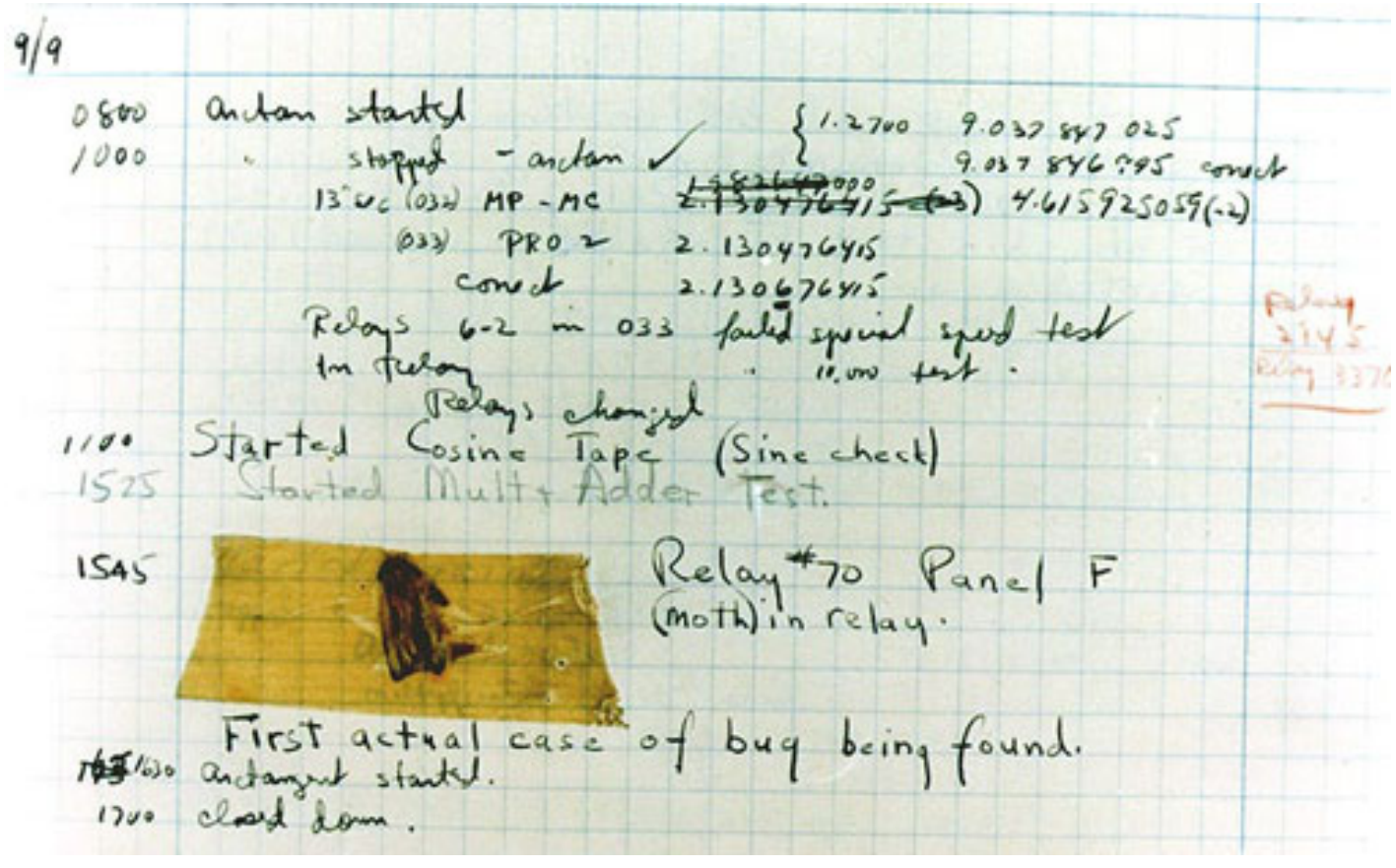


[© photo by mark.sze]

- Transistor count of > 10 billion
- Gate level models are huge
- Big designer teams (several hundreds)
- Big correctness issues
- Late bugs are extremely expensive

Motivation

The First Bug (1947)



[Photo: U.S. Naval Historical Center]

Motivation

The Pentium FDIV Bug (1994)

let $x = 4195835,\ y = 3145727 \quad \Rightarrow \quad x - \frac{x}{y} \cdot y =$

Motivation

The Pentium FDIV Bug (1994)

let $x = 4195835,\ y = 3145727 \quad \Rightarrow \quad x - \frac{x}{y} \cdot y = 256$

- Bug in floating point unit $\Rightarrow$ $ 450 Mio. loss for Intel

Motivation

The Pentium FDIV Bug (1994)

let $x = 4195835,\ y = 3145727 \quad \Rightarrow \quad x - \frac{x}{y} \cdot y = 256$

- Bug in floating point unit $\Rightarrow$ $ 450 Mio. loss for Intel

820 Chipset MTH Bug (2000)

- Error in memory translator hub
- Recall of around 1 Mio. motherboards
- $ 253 Mio. financial loss

Motivation

The Pentium FDIV Bug (1994)

let $x = 4195835,\ y = 3145727 \quad \Rightarrow \quad x - \frac{x}{y} \cdot y = 256$

- Bug in floating point unit $\Rightarrow$ $ 450 Mio. loss for Intel

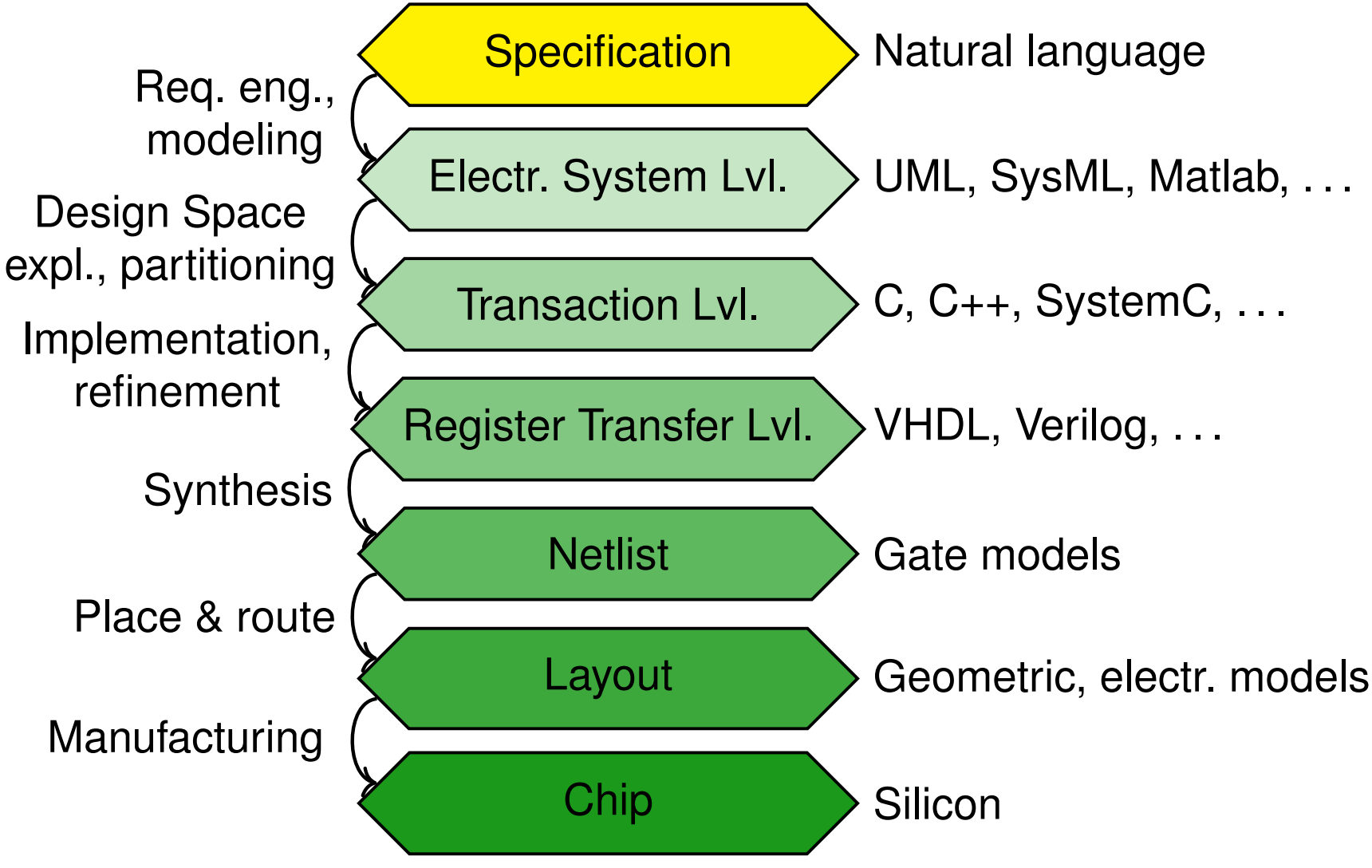
820 Chipset MTH Bug (2000)

- Error in memory translator hub
- Recall of around 1 Mio. motherboards
- $ 253 Mio. financial loss

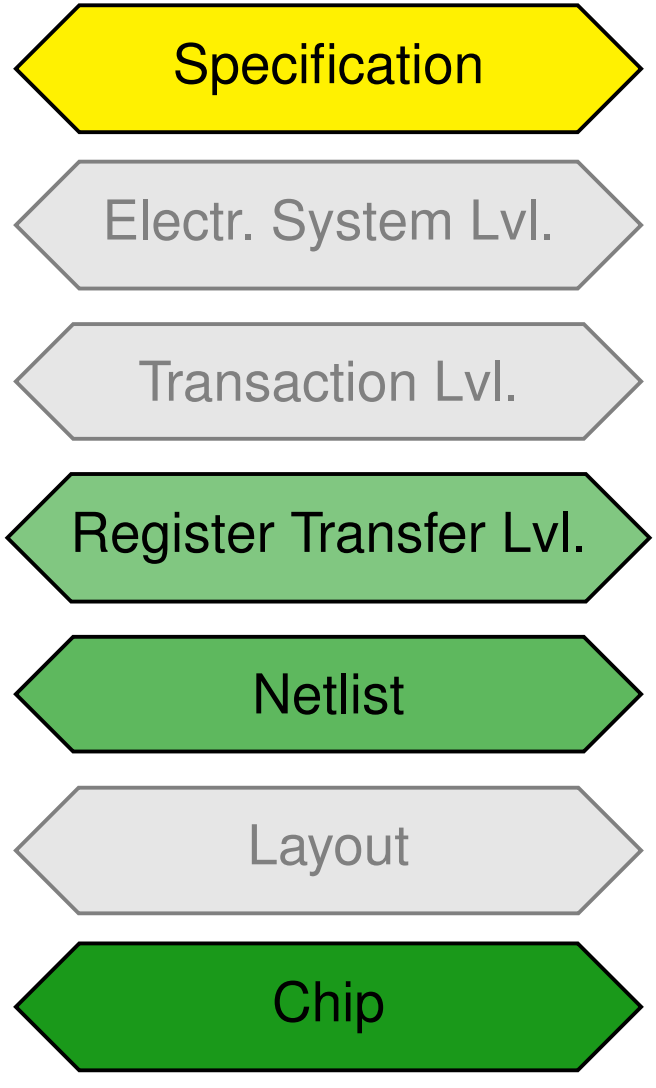
Intel i6/i7 (Skylake) Hyperthreading Bug (2017)

- Specific operating conditions cause unpredictable behavior
- Found by Linux/OCaml developers

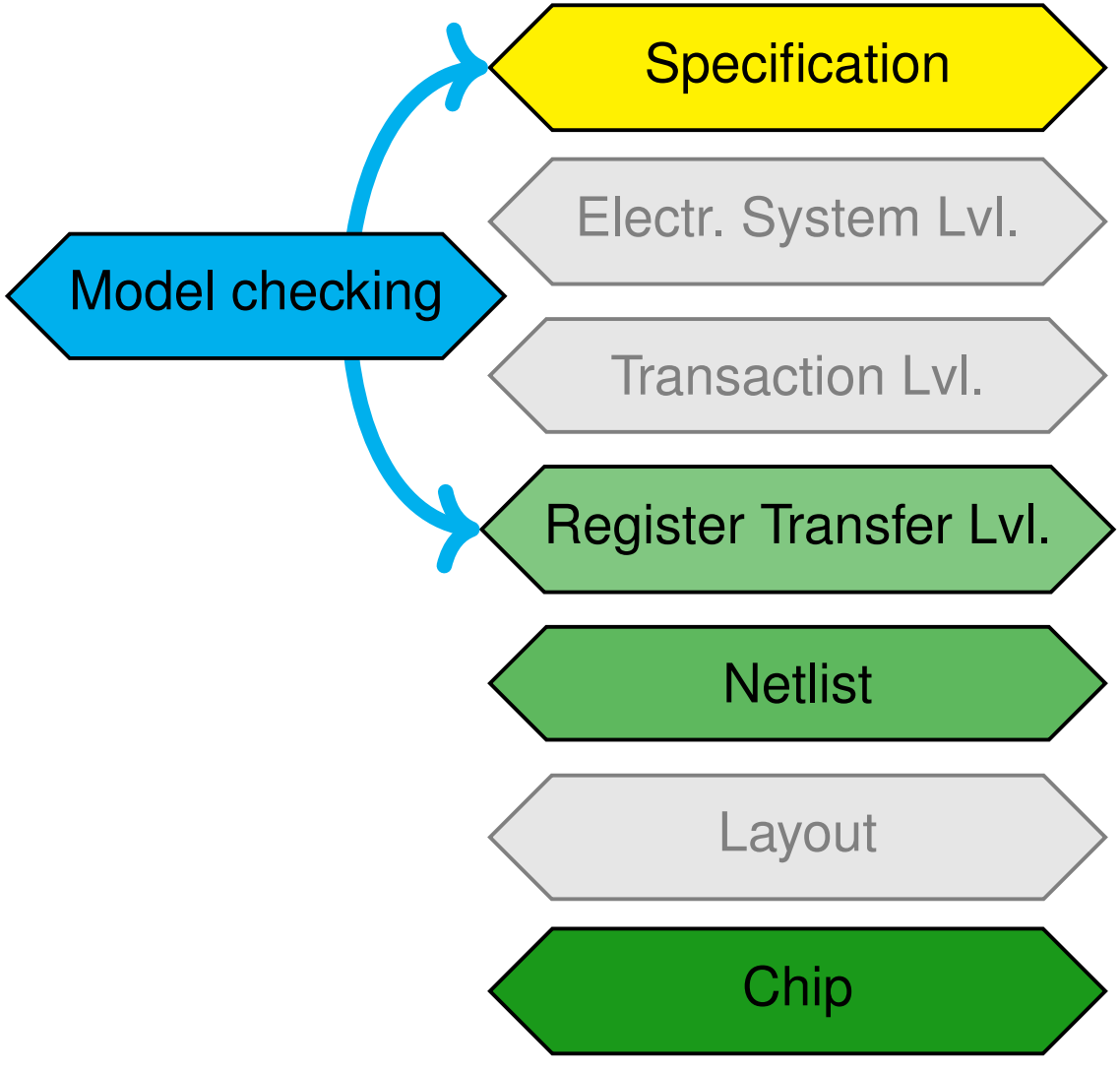
Hardware Design Flow



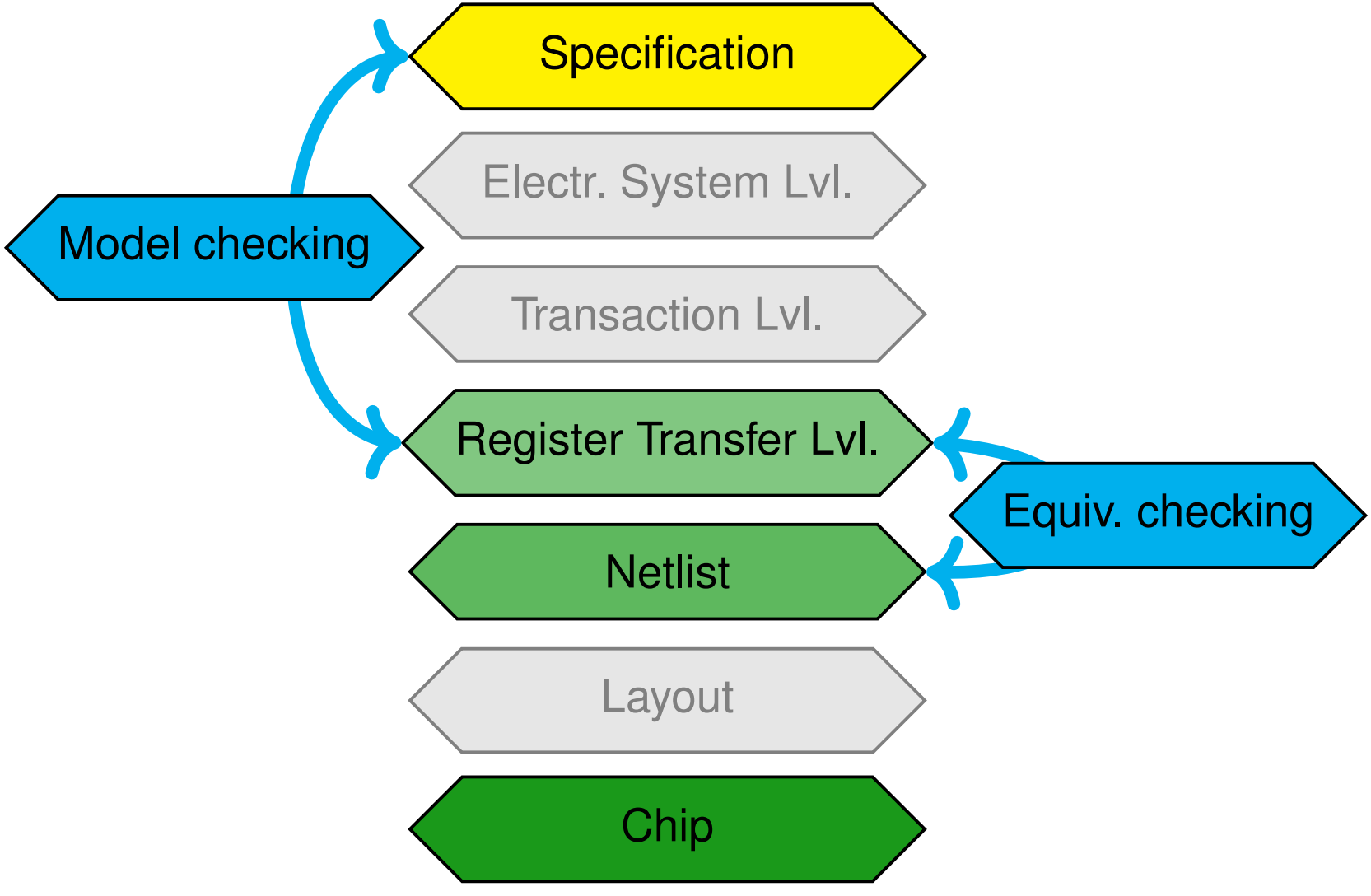
Hardware Verification Flow



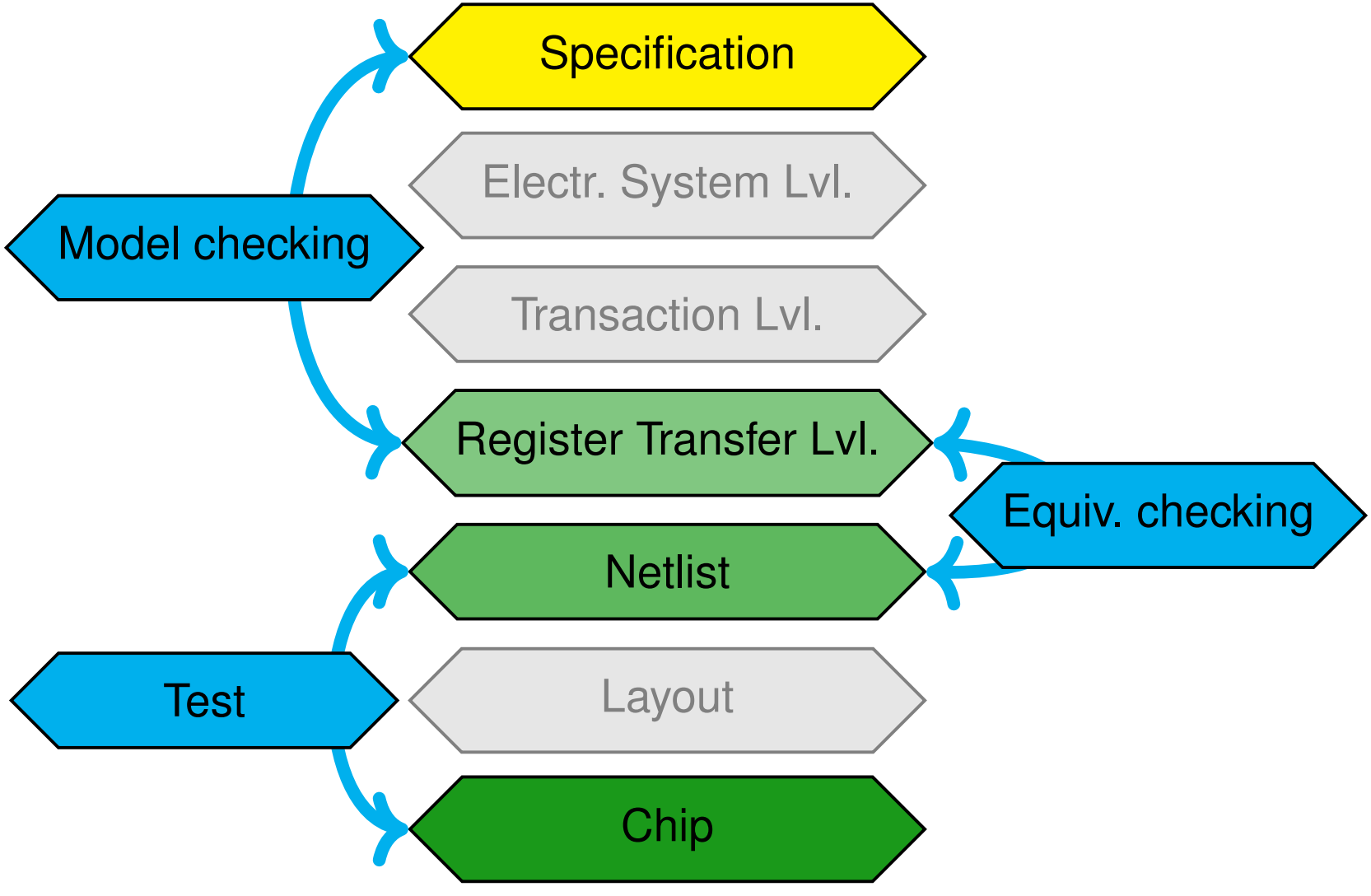
Hardware Verification Flow



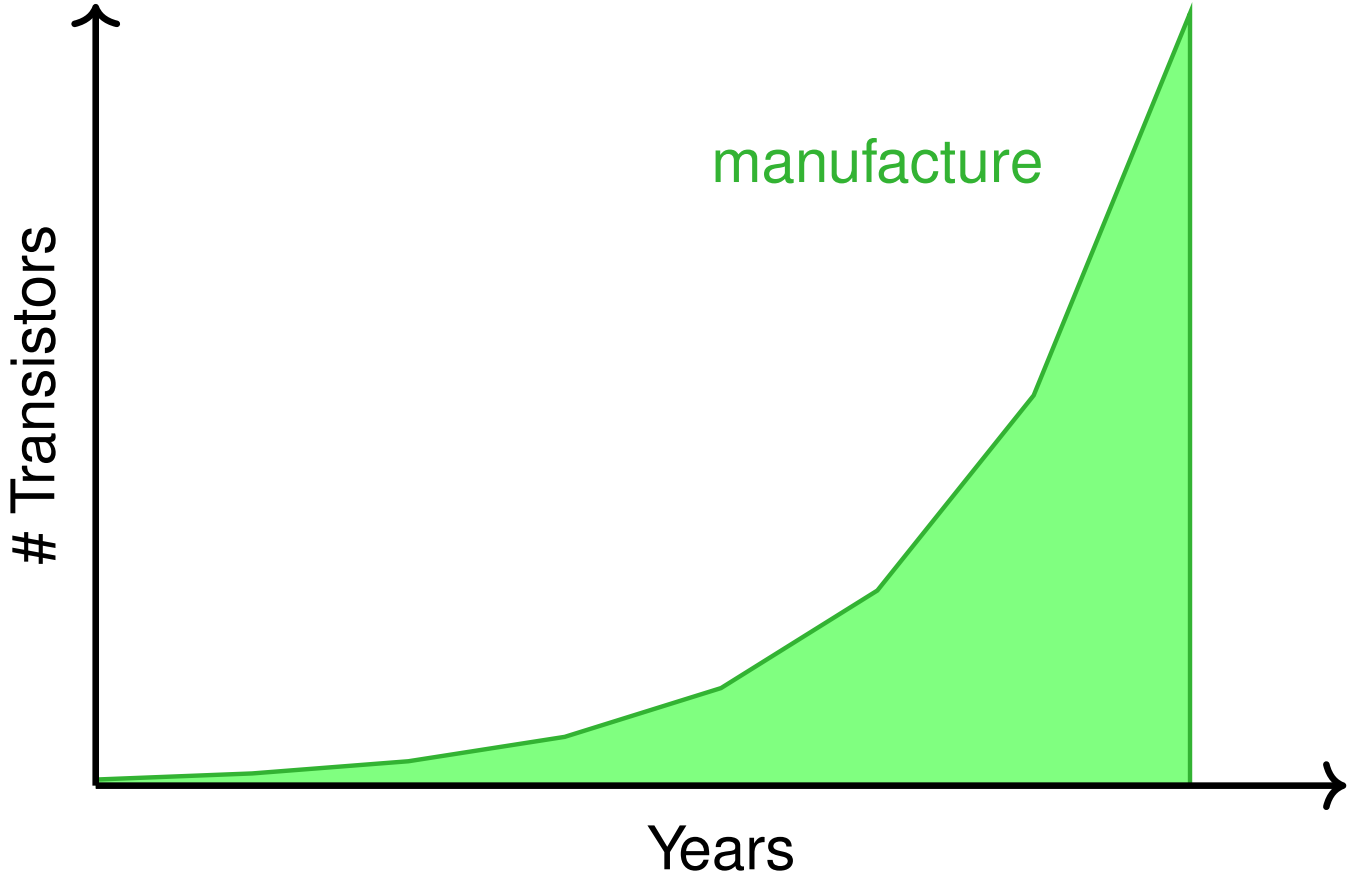
Hardware Verification Flow



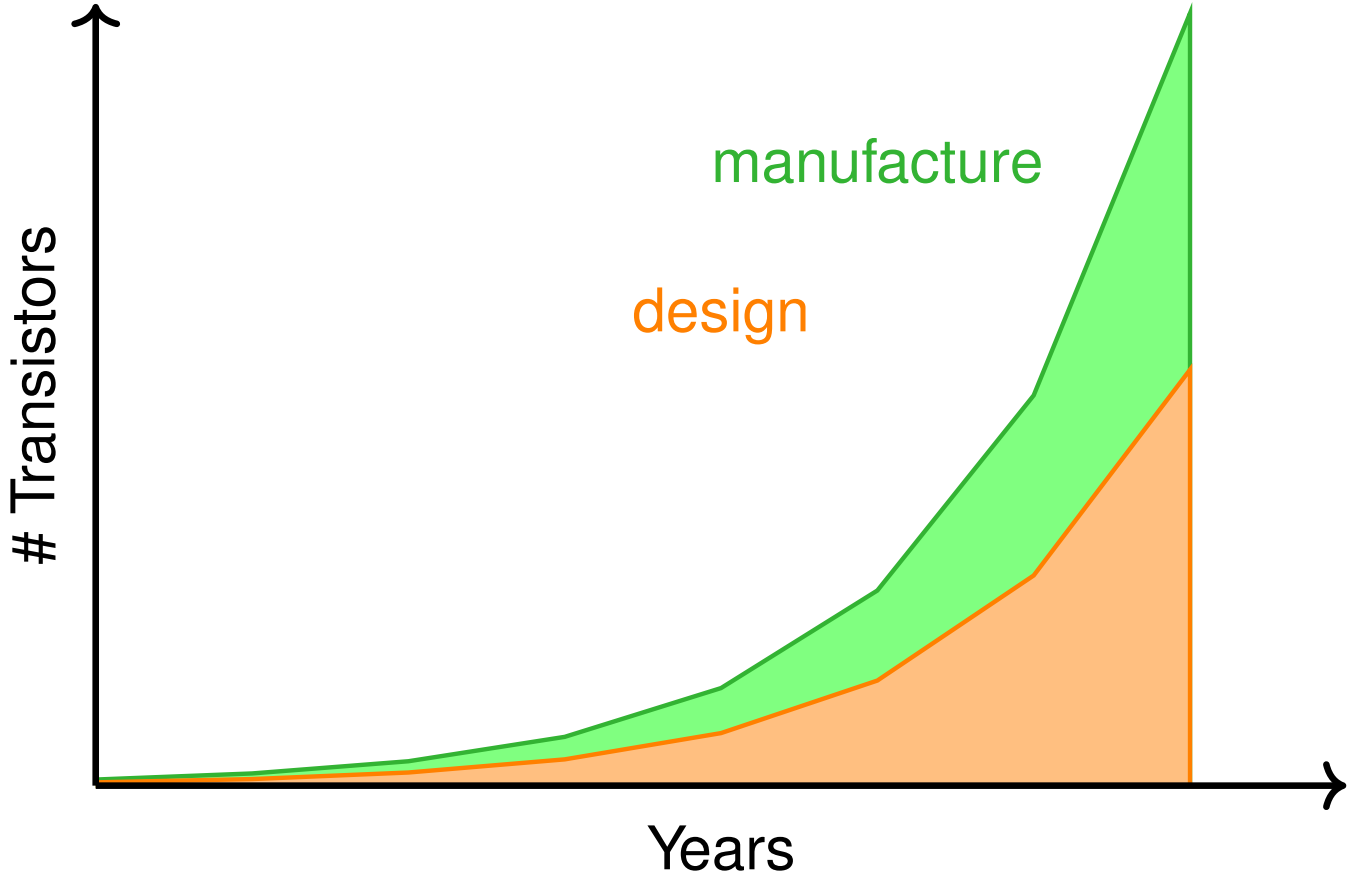
Hardware Verification Flow



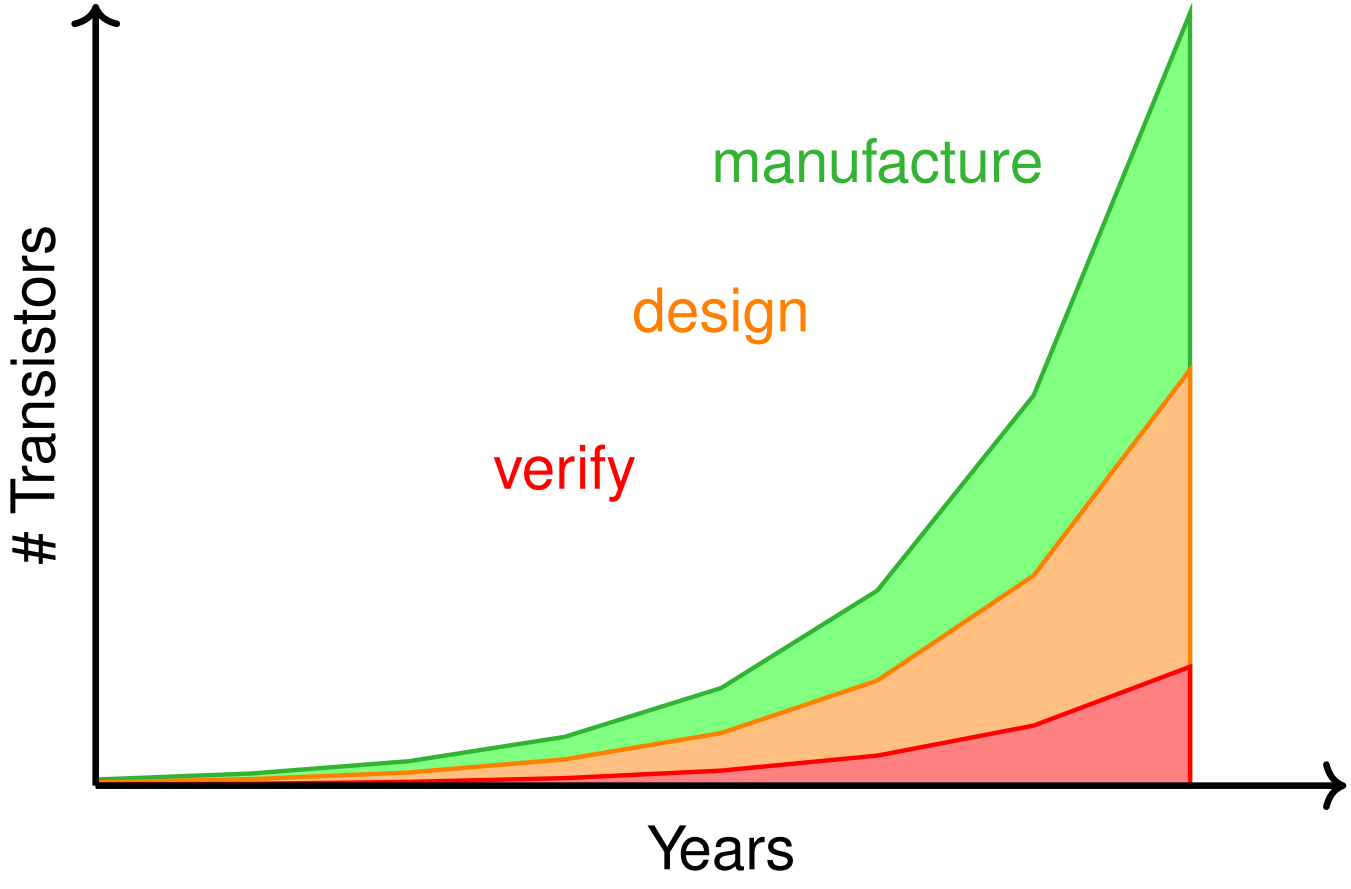
Design Gap – Verification Gap



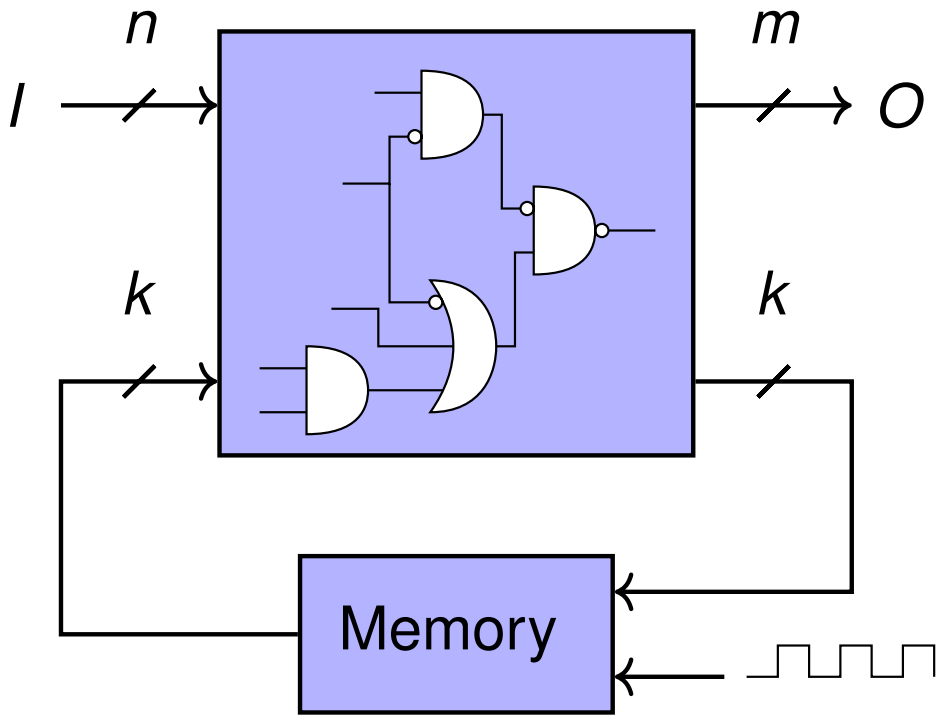
Design Gap – Verification Gap



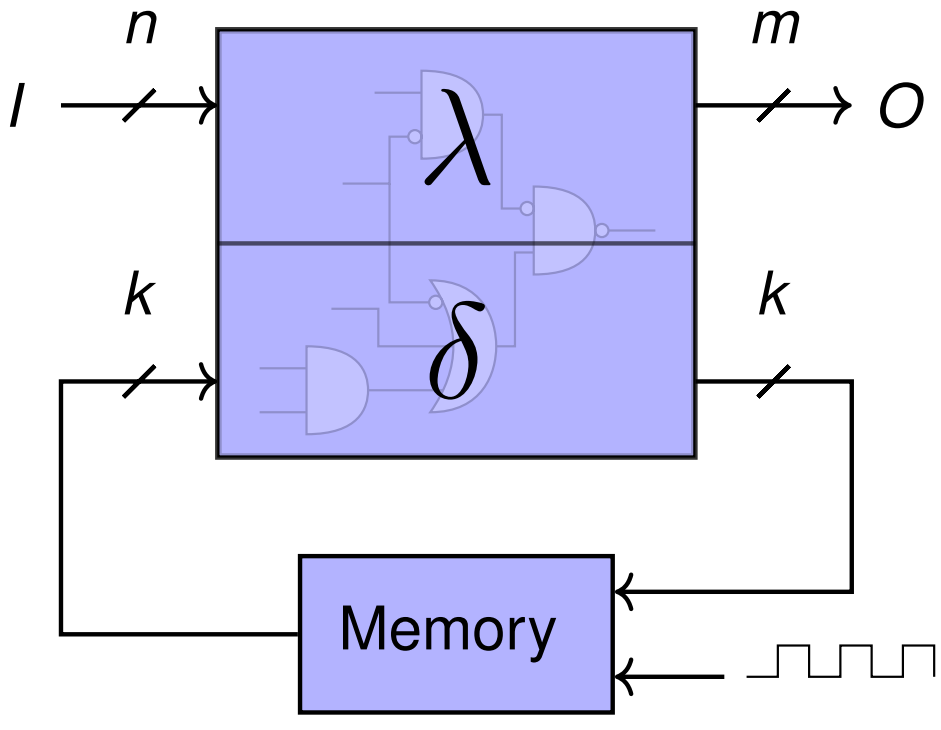
Design Gap – Verification Gap



Sequential Circuit Model



Sequential Circuit Model



Mealy Machine:

$$\mathcal{M} = (I, O, S, S_0, \delta, \lambda)$$

$$\delta : S \times I \rightarrow S$$
$$\lambda : S \times I \rightarrow O$$
$$S_0 \subseteq S$$

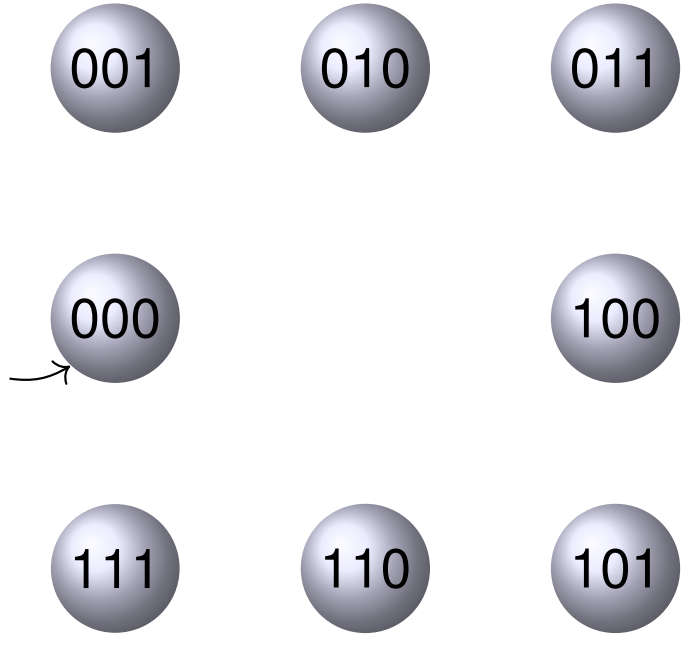
$$I = \{0,1\}^n$$
$$O = \{0,1\}^m$$
$$S = \{0,1\}^k$$

From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input       CLK, EN, CLR;
  output reg  S0, S1, S2;
  output      V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```

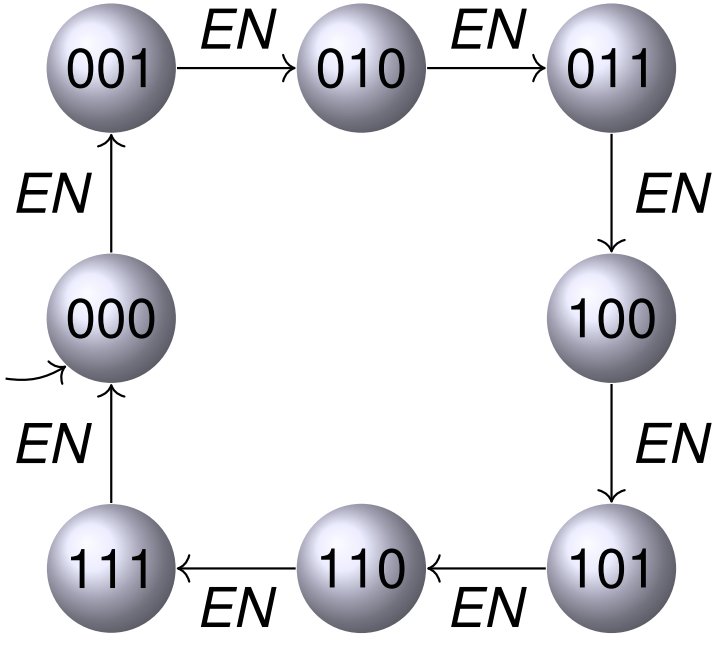
From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input      CLK, EN, CLR;
  output reg S0, S1, S2;
  output     V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```



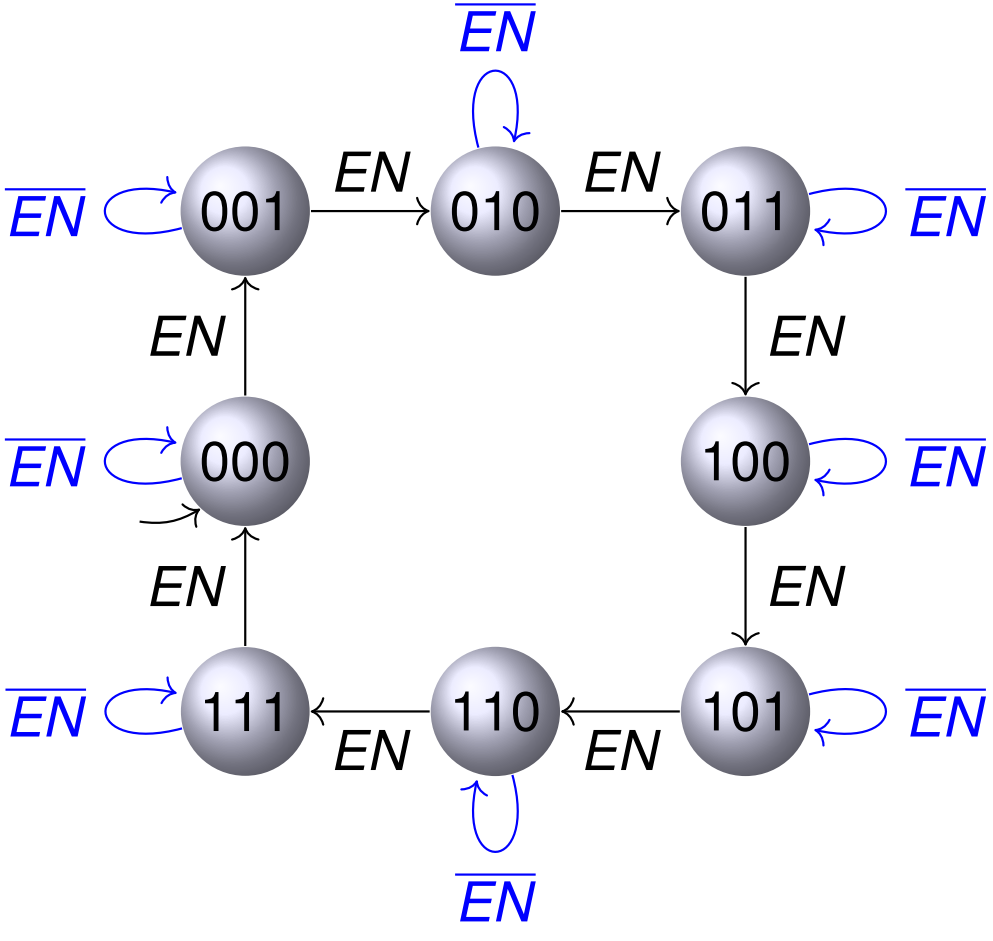
From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input       CLK, EN, CLR;
  output reg  S0, S1, S2;
  output      V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```



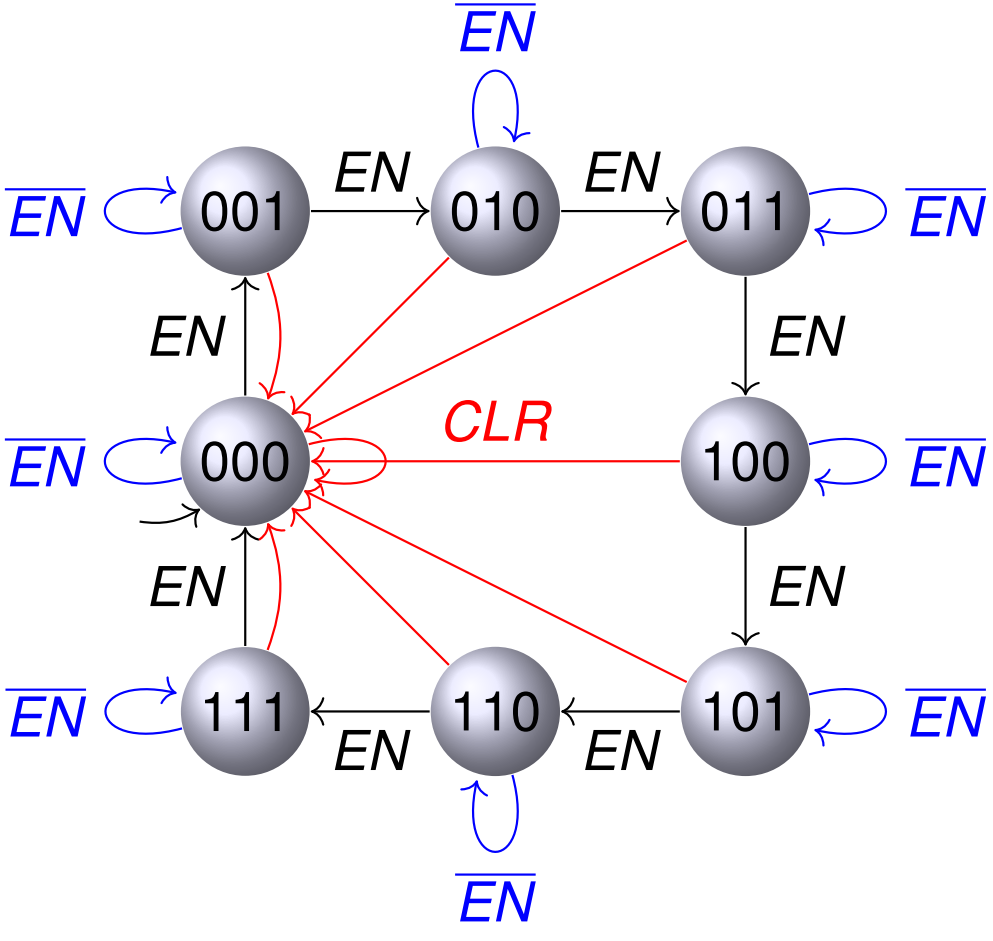
From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input       CLK, EN, CLR;
  output reg  S0, S1, S2;
  output      V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```



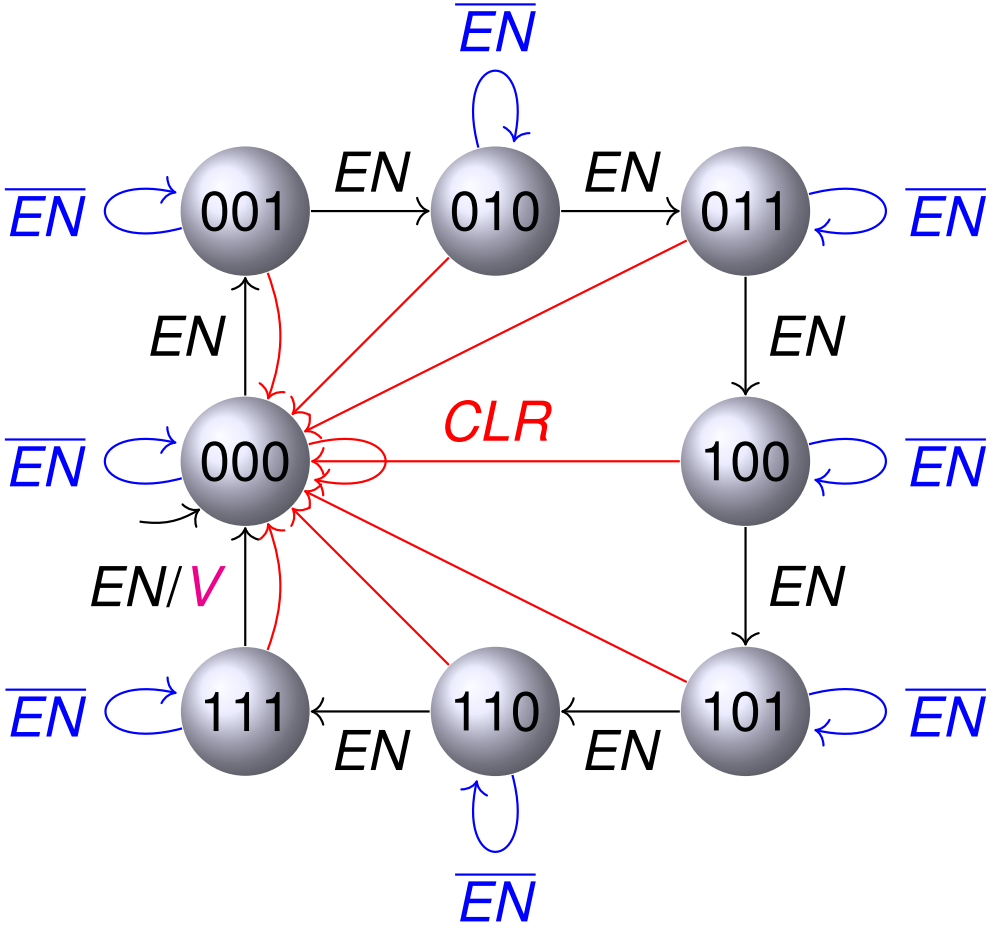
From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input       CLK, EN, CLR;
  output reg  S0, S1, S2;
  output      V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```



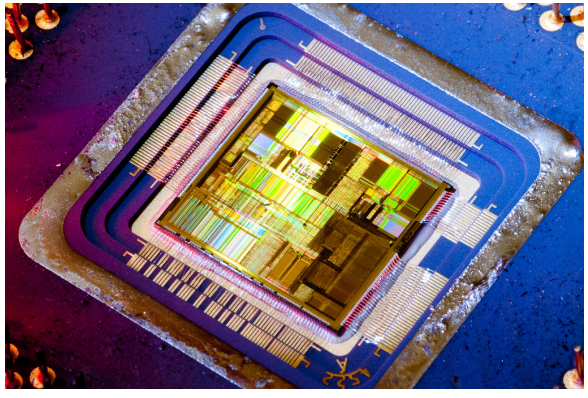
From Verilog to Mealy Machine

```
module count(CLK, EN, CLR,
             S0, S1, S2, V);

  input       CLK, EN, CLR;
  output reg  S0, S1, S2;
  output      V;

  assign V = S0 & S1 & S2 &
             !CLR & EN;

  always @(posedge CLK) begin
    if (CLR)
      {S2, S1, S0} <= 0;
    else if (EN)
      {S2, S1, S0}
        <= {S2, S1, S0} + 1;
  end
endmodule // count
```

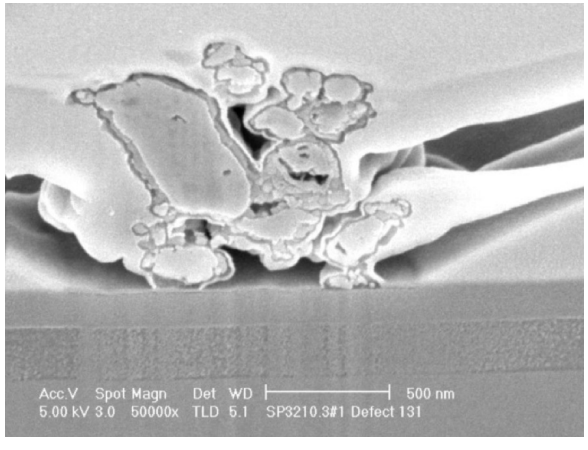


Model or Implementation?

- Fully automated synthesis from HDL
- Abstracts electronic/physical properties
  - Technology (e.g. CMOS)
  - Timing of combinatorial logic
  - Manufacturing defects
  - Cross-talk
- Abstracts non-functional properties
  - Energy consumption
  - Area on chip
  - Reliability



[cc photo by mark.sze]



[Ng Boon Haur & Oh Siew Khim / spectrum.ieee.org]

Outline

What do we want to verify?

Safety

Something bad will never happen, e.g.
“The stack pointer will never overflow”
“The traffic lights will never be green at the same time”

What do we want to verify?

Safety

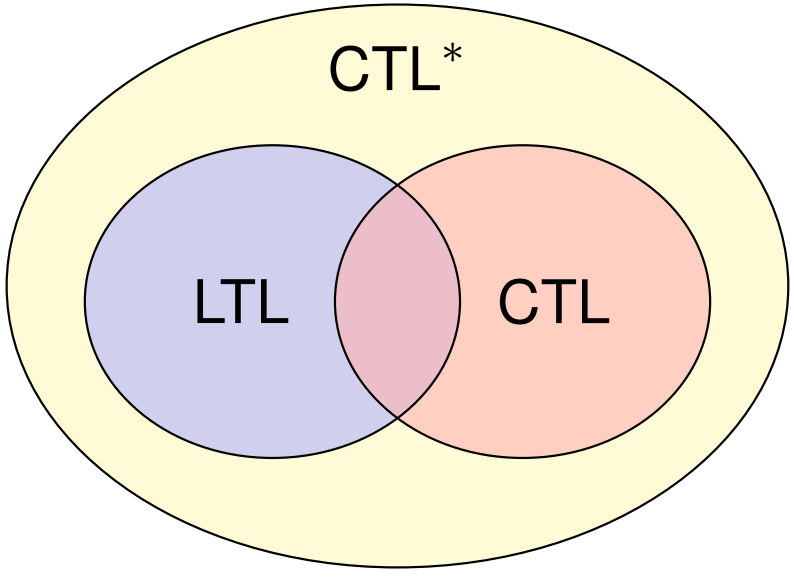
Something bad will never happen, e.g.
“The stack pointer will never overflow”
“The traffic lights will never be green at the same time”

Liveness

Something good will eventually happen, e.g.
“Every request will be granted”
“The cache and the main memory will eventually be consistent”

How to specify such properties?

- Temporal logic = propositional logic + time
- Discrete vs. continuous time
- Linear time view
- Branching time view



Model Checking

Model Checking

Given a Mealy Machine $\mathcal{M}$ and a temporal logic formula φ, check if $\mathcal{M} \models \varphi$.

Model Checking

Model Checking

Given a Mealy Machine $\mathcal{M}$ and a temporal logic formula φ, check if $\mathcal{M} \models \varphi$.

How do we do this?

Model Checking

Model Checking

Given a Mealy Machine $\mathcal{M}$ and a temporal logic formula φ, check if $\mathcal{M} \models \varphi$.

How do we do this?

It's complicated... (more in SE303)

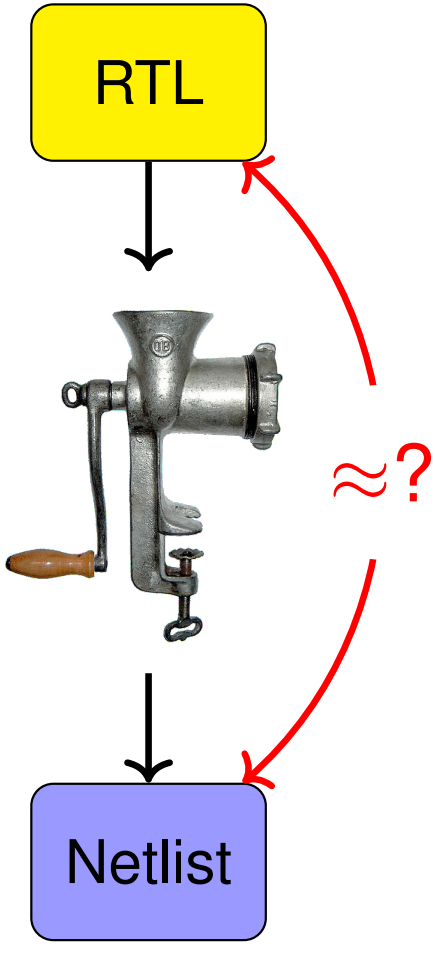
Outline

Equivalence Checking

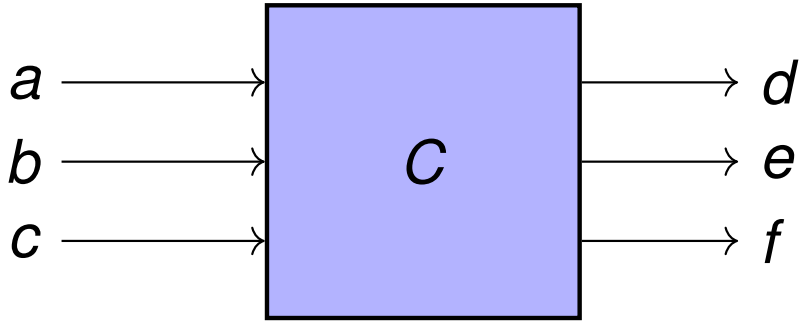
- Hardware *synthesis* is complex
- Aggressive *optimization*
- Automatic pipelining, retiming, . . .
- Technology mapping
- Tools are closed source

Equivalence Checking

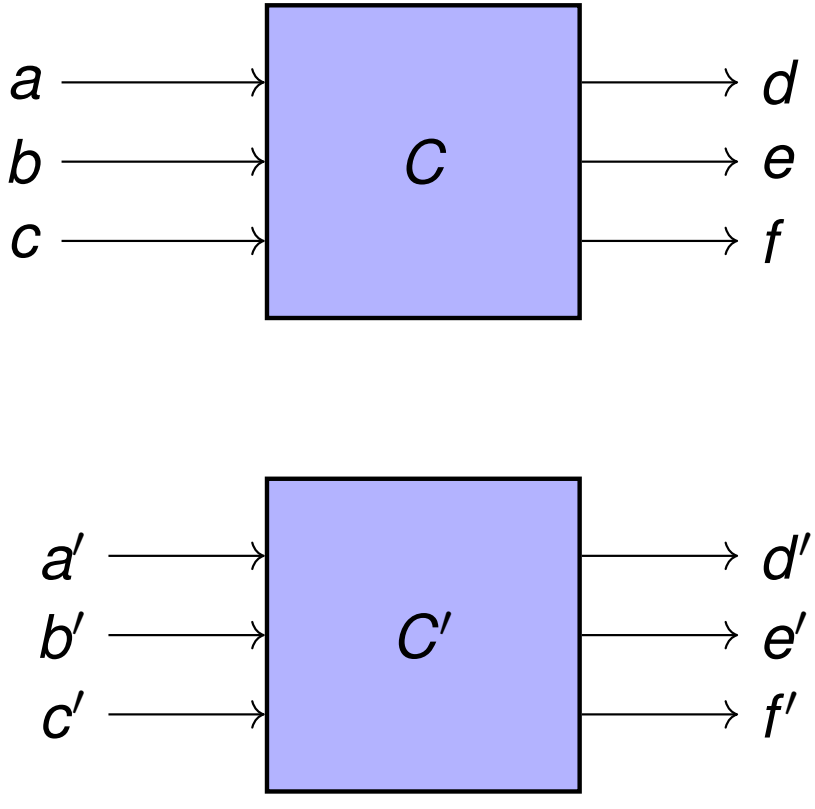
- Hardware *synthesis* is complex
- Aggressive *optimization*
- Automatic pipelining, retiming, . . .
- Technology mapping
- Tools are closed source
- Does the netlist do what we think it does?



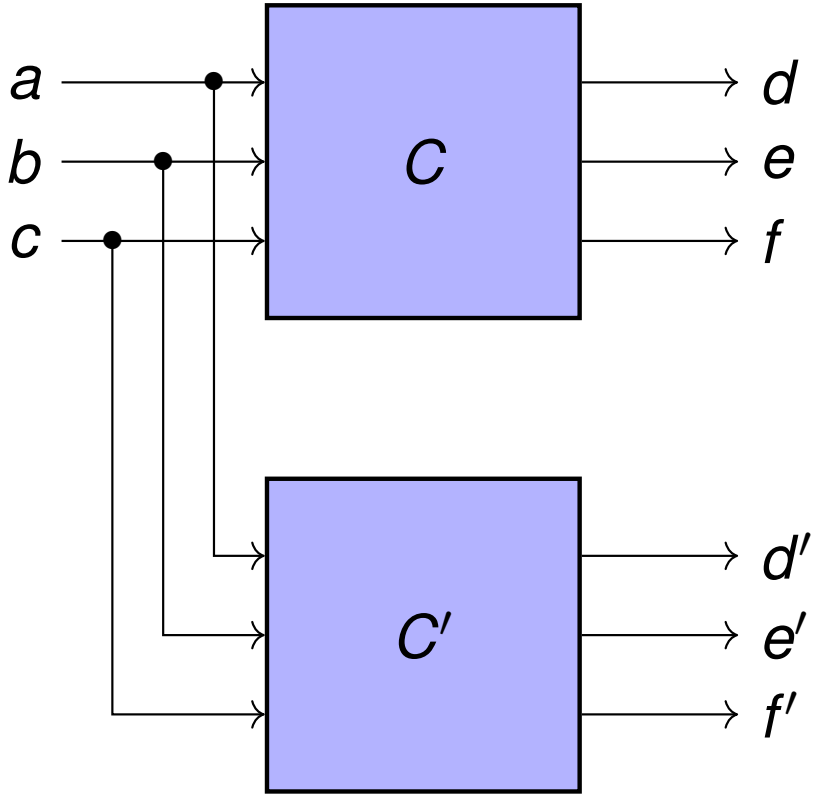
Miter Structure



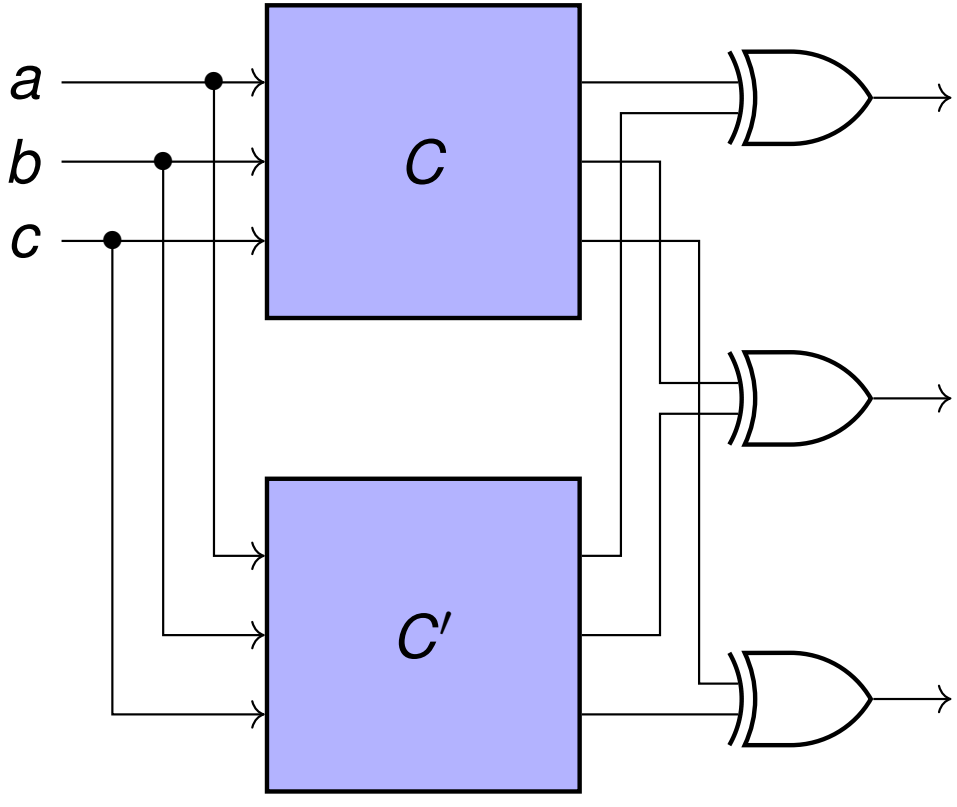
Miter Structure



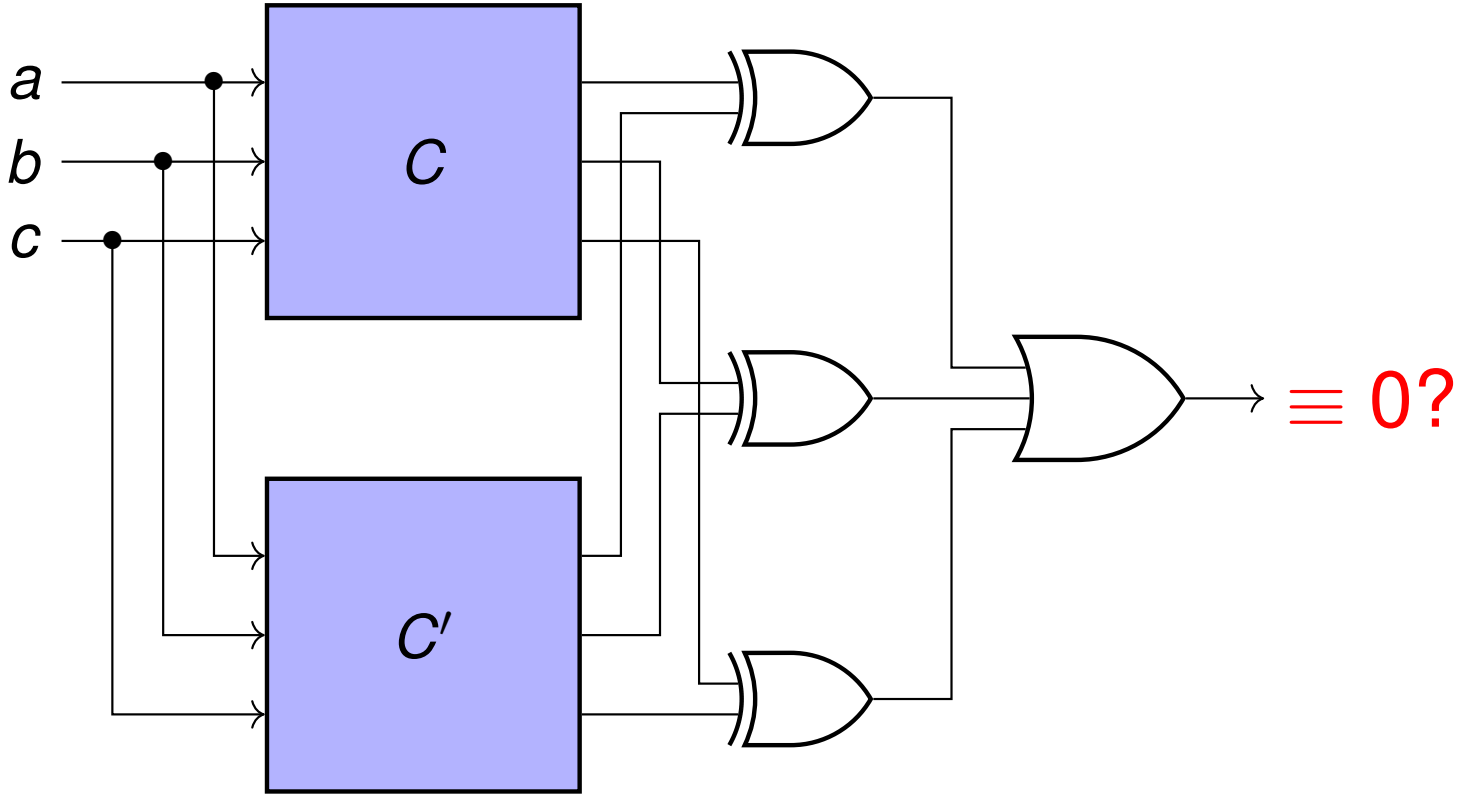
Miter Structure



Miter Structure



Miter Structure



Boolean Satisfiability (SAT)

SAT Problem

Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$ (in conjunctive normal form), is there an assignment $X \in \{0,1\}^n$, such that $f(X) = 1$?

Conjunctive Normal Form

A Boolean formula over variables $X = \{x_0 \dots x_n\}$ is in conjunctive normal form if it is a conjunction of clauses $(l_{1,1} \vee l_{1,2} \vee \cdots \vee l_{1,m_0}) \wedge \cdots \wedge (l_{k,1} \vee \cdots \vee l_{k,m_k})$. A clause is a disjunction of literals $l = x_i$ or $l = \neg x_i$ for some $x_i \in X$.

Boolean Satisfiability (SAT)

SAT Problem

Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$ (in conjunctive normal form), is there an assignment $X \in \{0,1\}^n$, such that $f(X) = 1$?

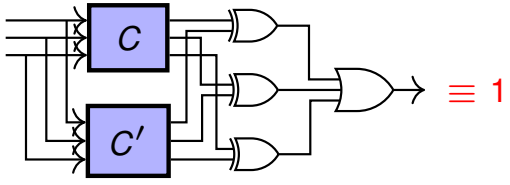
Conjunctive Normal Form

A Boolean formula over variables $X = \{x_0 \dots x_n\}$ is in conjunctive normal form if it is a conjunction of clauses $(l_{1,1} \vee l_{1,2} \vee \cdots \vee l_{1,m_0}) \wedge \cdots \wedge (l_{k,1} \vee \cdots \vee l_{k,m_k})$. A clause is a disjunction of literals $l = x_i$ or $l = \neg x_i$ for some $x_i \in X$.

- NP-complete problem [Cook, 1971]

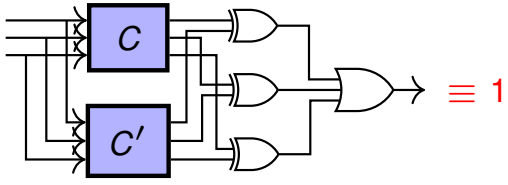
SAT-Based Equivalence Checking

Problem



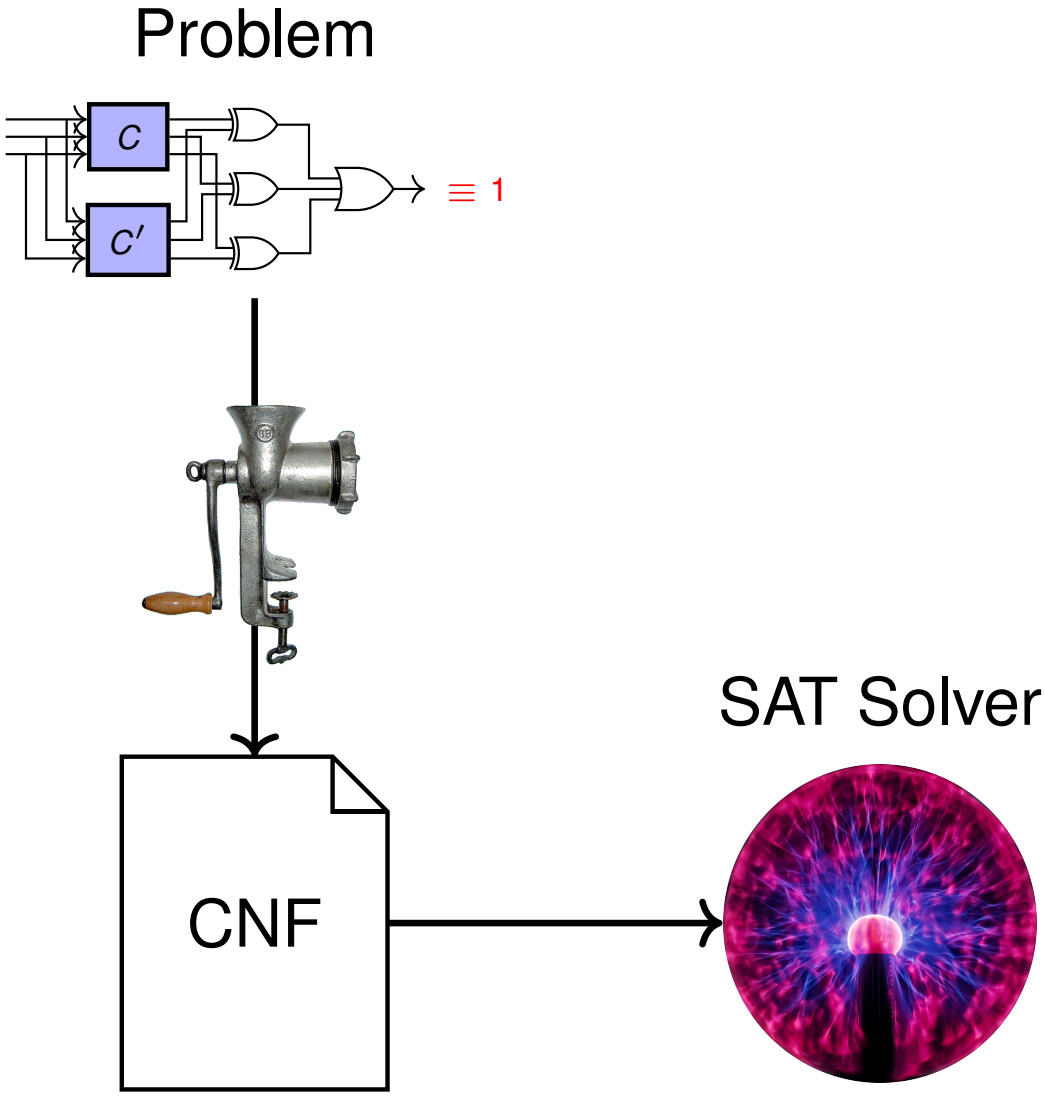
SAT-Based Equivalence Checking

Problem

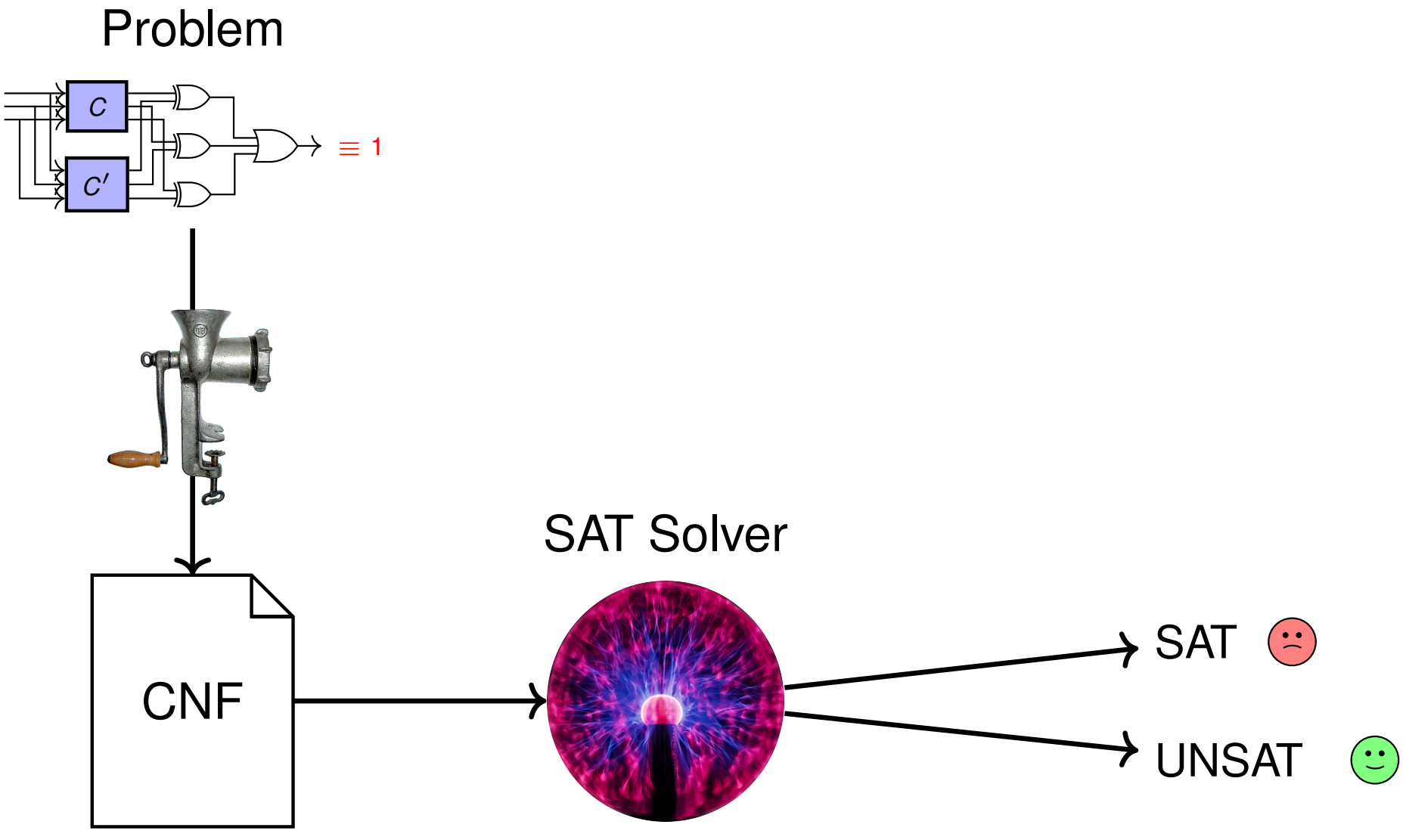


CNF

SAT-Based Equivalence Checking

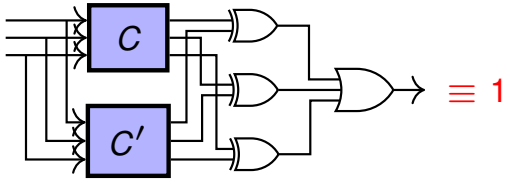


SAT-Based Equivalence Checking



SAT-Based Equivalence Checking

Problem



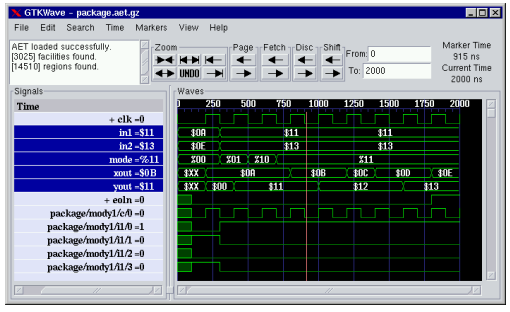
CNF

SAT Solver

SAT

UNSAT

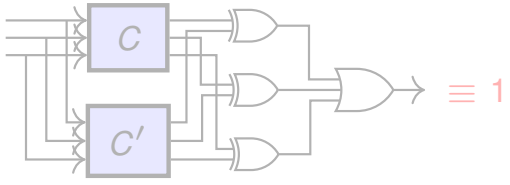
Counterexample



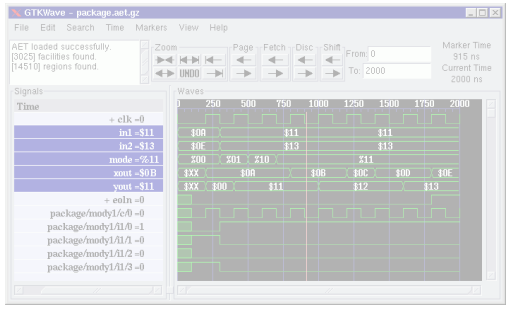
TELECOM ParisTech

SAT-Based Equivalence Checking

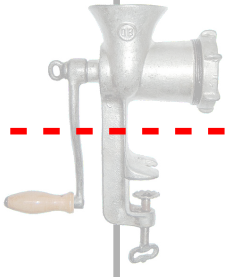
Problem



Counterexample

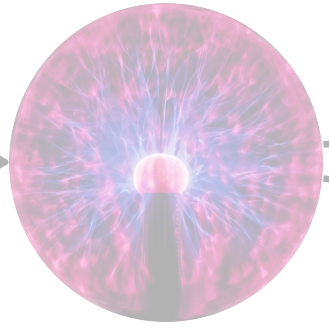


Problem



Encoding

CNF

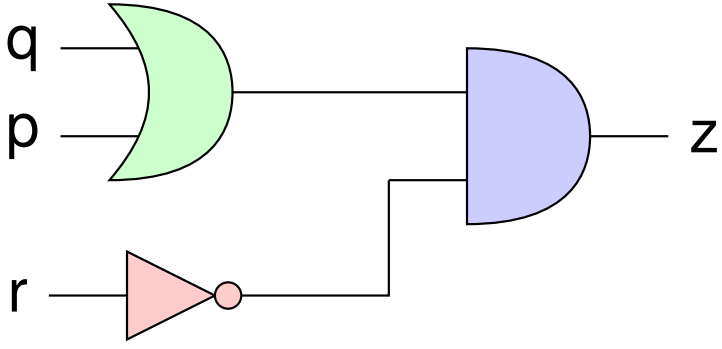


SAT

UNSAT

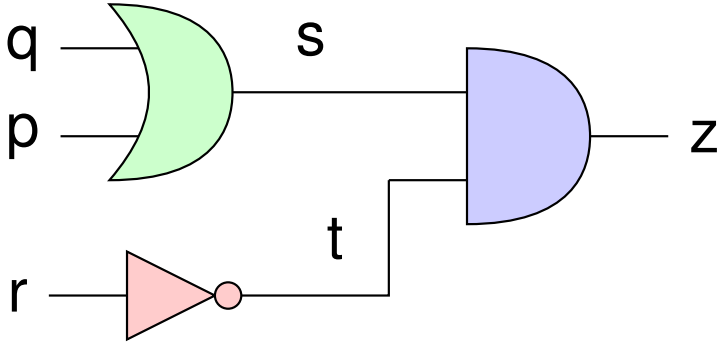
TELECOM ParisTech

Tseytin Transformation



$$z \leftrightarrow (p \vee q) \wedge \neg r$$

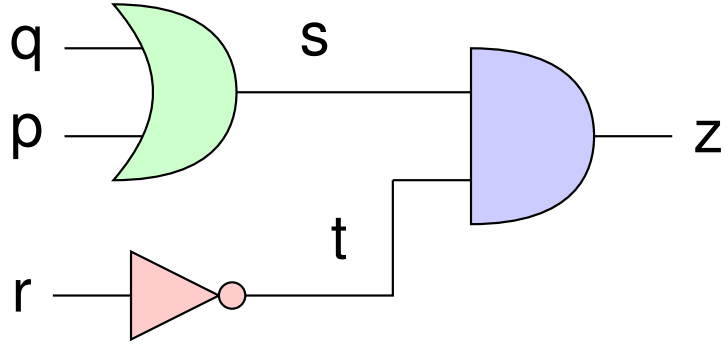
Tseytin Transformation



$$z \leftrightarrow (p \vee q) \wedge \neg r$$

$$s \leftrightarrow p \vee q$$
$$t \leftrightarrow \neg r$$
$$z \leftrightarrow s \wedge t$$

Tseytin Transformation

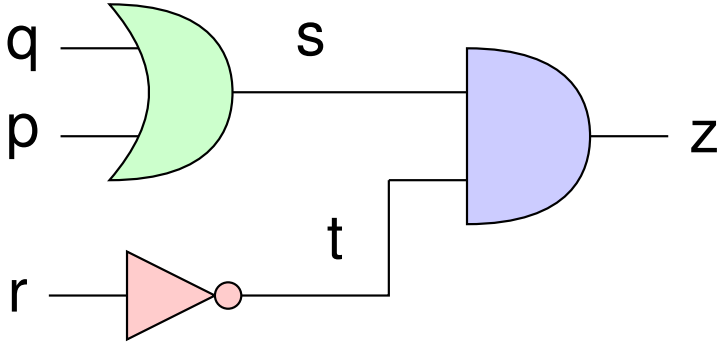


$$z \leftrightarrow (p \vee q) \wedge \neg r$$

$$s \leftrightarrow p \vee q$$
$$t \leftrightarrow \neg r$$
$$z \leftrightarrow s \wedge t$$

$$\begin{aligned} s \leftrightarrow p \vee q &\equiv (s \rightarrow p \vee q) \wedge (p \vee q \rightarrow s) \\ &\equiv (\neg s \vee p \vee q) \wedge (\neg(p \vee q) \vee s) \\ &\equiv (\neg s \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee s) \\ &\equiv (\neg s \vee p \vee q) \wedge (\neg p \vee s) \wedge (\neg q \vee s) \end{aligned}$$

Tseytin Transformation



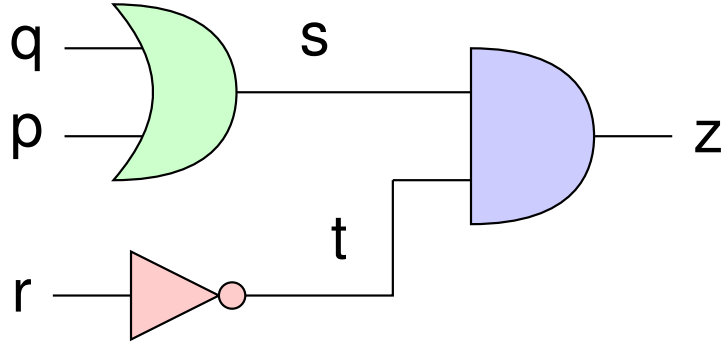
$$z \leftrightarrow (p \vee q) \wedge \neg r$$

$$s \leftrightarrow p \vee q$$
$$t \leftrightarrow \neg r$$
$$z \leftrightarrow s \wedge t$$

$$\begin{aligned}
s \leftrightarrow p \vee q &\equiv (s \rightarrow p \vee q) \wedge (p \vee q \rightarrow s) \\
&\equiv (\neg s \vee p \vee q) \wedge (\neg(p \vee q) \vee s) \\
&\equiv (\neg s \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee s) \\
&\equiv (\neg s \vee p \vee q) \wedge (\neg p \vee s) \wedge (\neg q \vee s)
\end{aligned}$$

$$t \leftrightarrow \neg r \equiv (\neg t \vee \neg r) \wedge (t \vee r)$$

Tseytin Transformation



$$z \leftrightarrow (p \vee q) \wedge \neg r$$

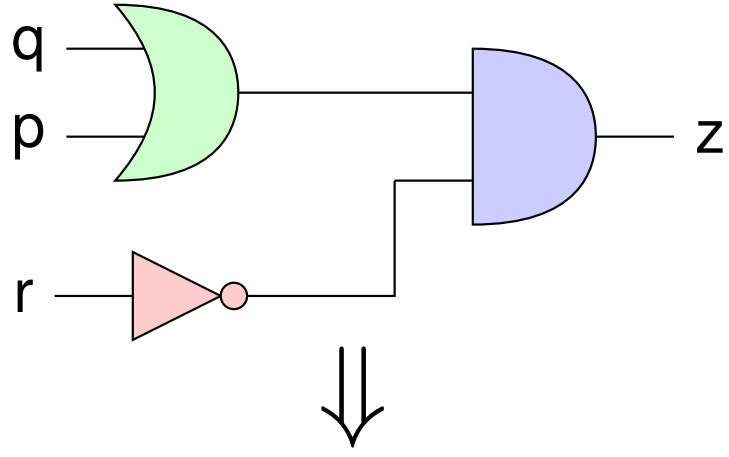
$$s \leftrightarrow p \vee q$$
$$t \leftrightarrow \neg r$$
$$z \leftrightarrow s \wedge t$$

$$\begin{aligned}
s \leftrightarrow p \vee q &\equiv (s \rightarrow p \vee q) \wedge (p \vee q \rightarrow s) \\
&\equiv (\neg s \vee p \vee q) \wedge (\neg(p \vee q) \vee s) \\
&\equiv (\neg s \vee p \vee q) \wedge ((\neg p \wedge \neg q) \vee s) \\
&\equiv (\neg s \vee p \vee q) \wedge (\neg p \vee s) \wedge (\neg q \vee s)
\end{aligned}$$

$$t \leftrightarrow \neg r \equiv (\neg t \vee \neg r) \wedge (t \vee r)$$

$$z \leftrightarrow s \wedge t \equiv (\neg s \vee \neg t \vee z) \wedge (s \vee \neg z) \wedge (t \vee \neg z)$$

Tseytin Transformation



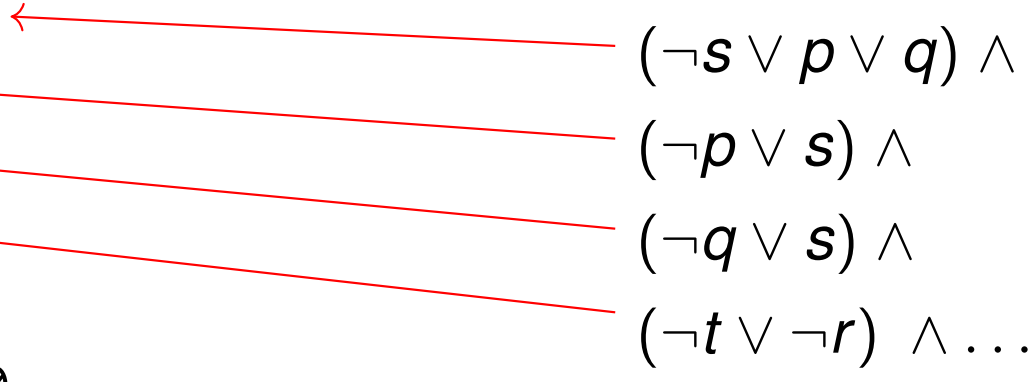
$$(\neg s \vee p \vee q) \wedge (\neg p \vee s) \wedge (\neg q \vee s) \wedge$$
$$(\neg t \vee \neg r) \wedge (t \vee r) \wedge$$
$$(\neg s \vee \neg t \vee z) \wedge (s \vee \neg z) \wedge (t \vee \neg z)$$

How it Looks Like in Practice...

```
c  example circuit
c
p cnf 6 8
-4 1 2 0
-1 4 0
-2 4 0
-5 -3 0
5 3 0
-4 -3 6 0
4 -6 0
5 -6 0
```

How it Looks Like in Practice...

```
c  example circuit
c
p cnf 6 8
-4 1 2 0
-1 4 0
-2 4 0
-5 -3 0
5 3 0
-4 -3 6 0
4 -6 0
5 -6 0
```



$$(\neg s \vee p \vee q) \wedge$$
$$(\neg p \vee s) \wedge$$
$$(\neg q \vee s) \wedge$$
$$(\neg t \vee \neg r) \wedge \ldots$$

DPLL Algorithm

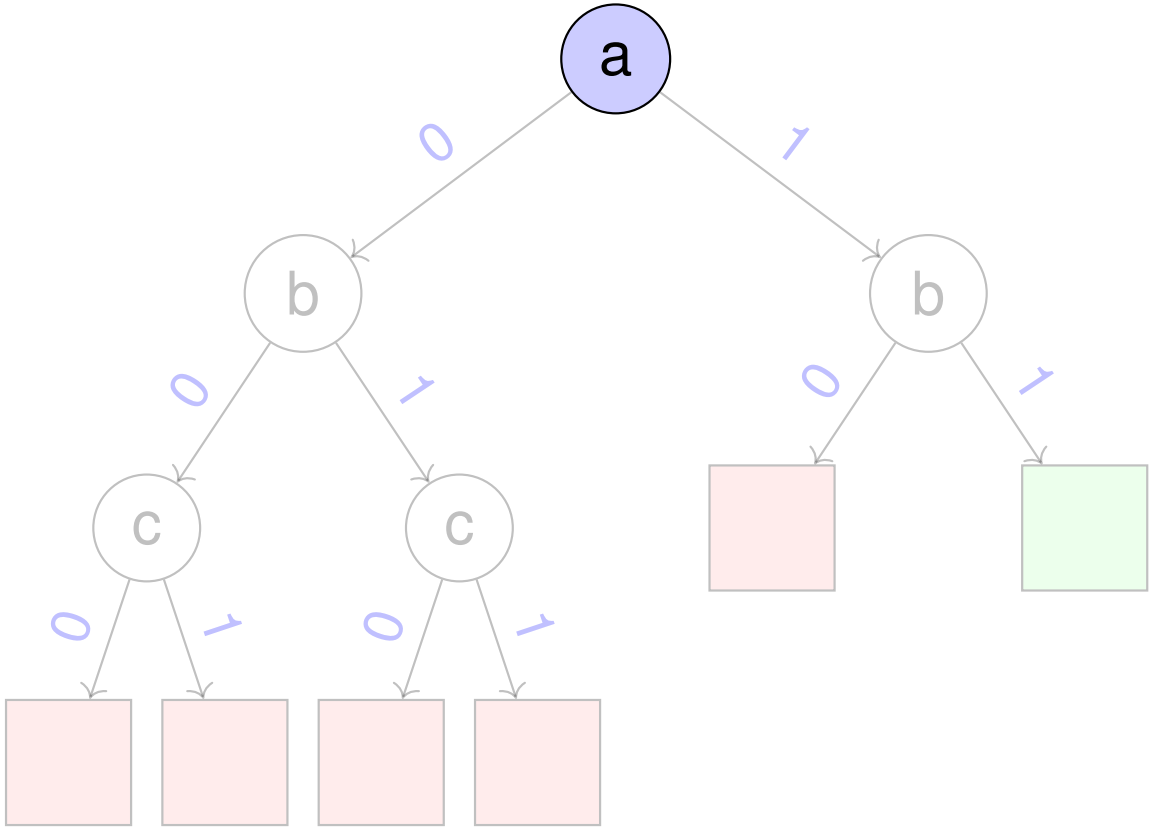
- Davis-Putnam-Logemann-Loveland [Davis et al., 1962]

DPLL

Data: Set of clauses $\mathcal{C}$
DPLL($\mathcal{C}$) **begin**
  if *all clauses satisfied* **then**
    return *SAT*
  if $\mathcal{C}$ *contains empty clause* **then**
    return *UNSAT*
  Propagate();
  $\ell \leftarrow$ *ChooseLiteral*();
  return *DPLL*($\mathcal{C} \wedge \ell$) *or DPLL*($\mathcal{C} \wedge \neg\ell$)

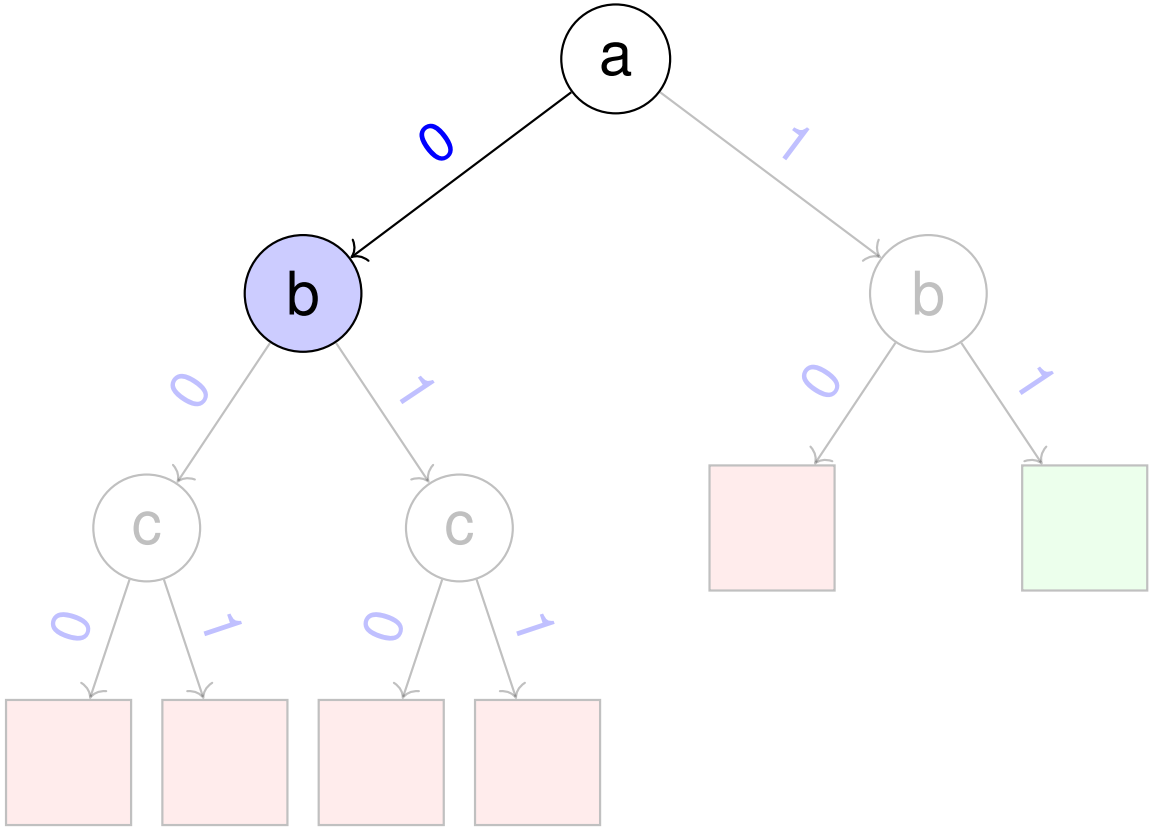
DPPL SAT Solving in a Nutshell

$$(\neg a \vee b \vee c)$$
$$(a \vee c \vee d)$$
$$(a \vee c \vee \neg d)$$
$$(a \vee \neg c \vee d)$$
$$(a \vee \neg c \vee \neg d)$$
$$(\neg b \vee \neg c \vee d)$$
$$(\neg a \vee b \vee \neg c)$$
$$(\neg a \vee \neg b \vee c)$$



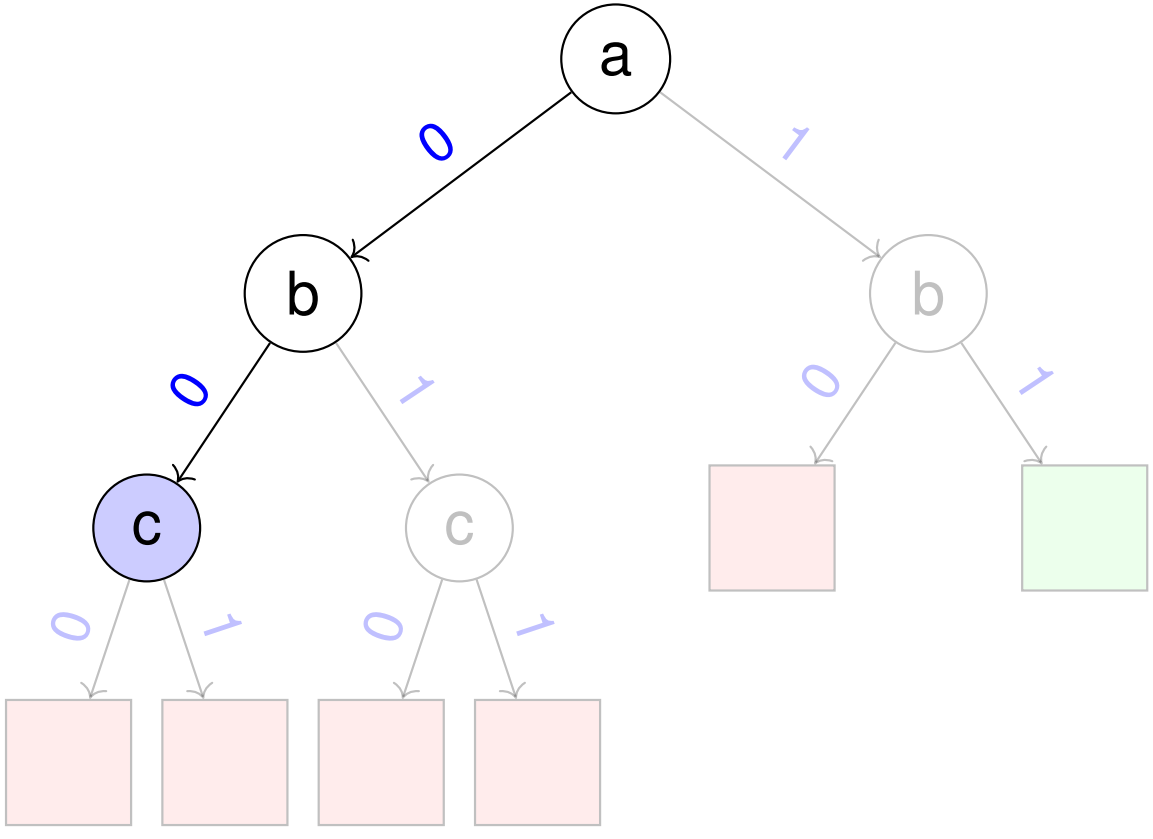
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



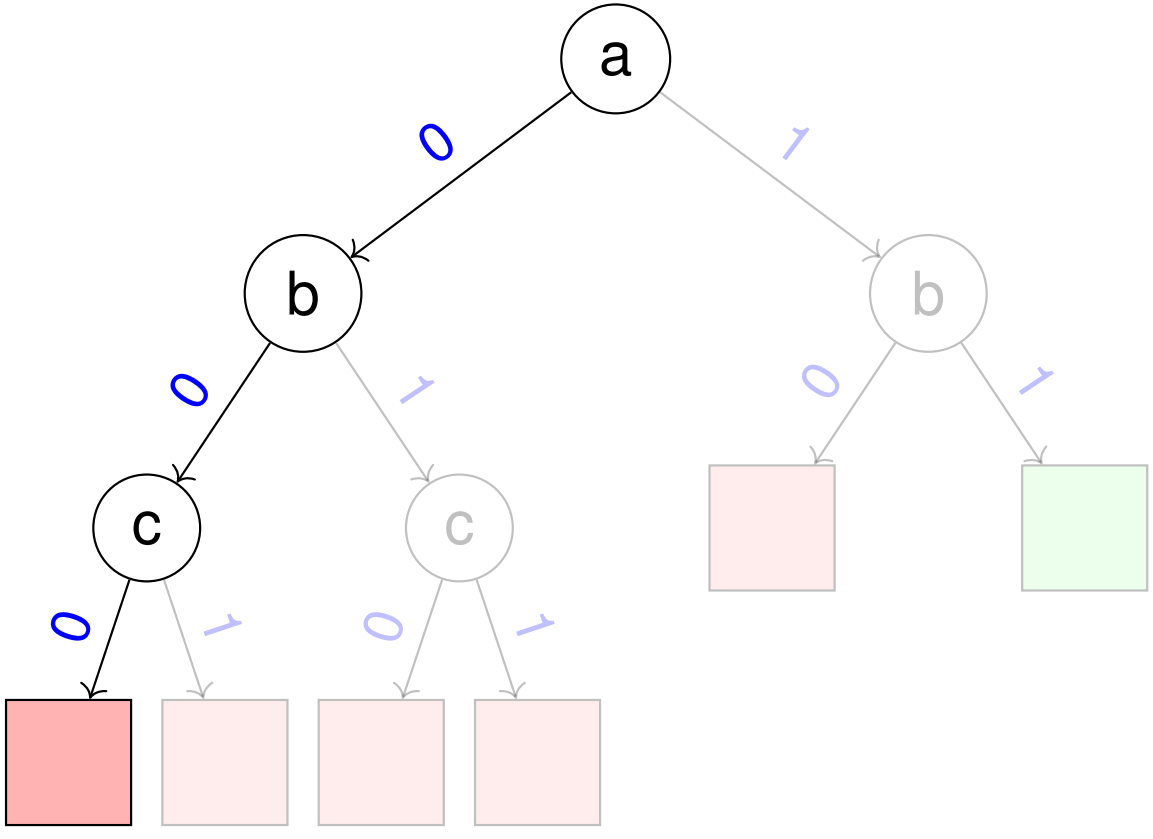
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



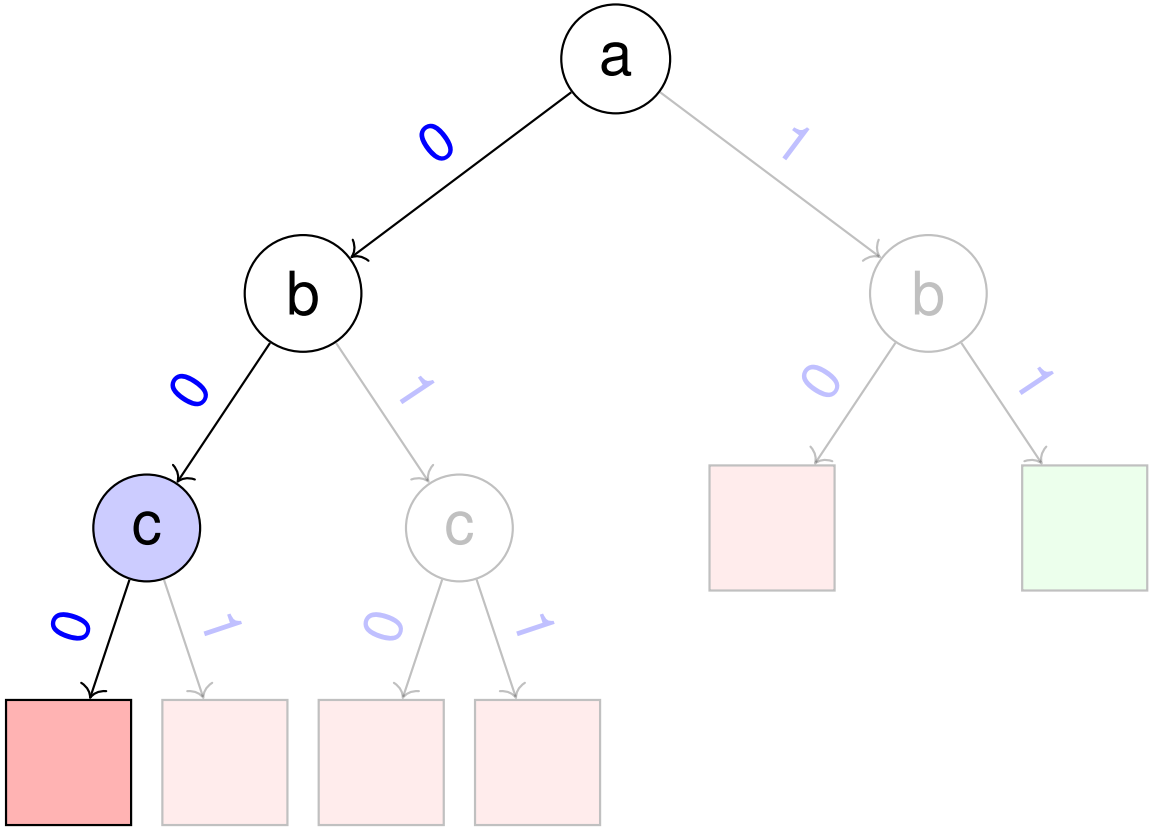
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



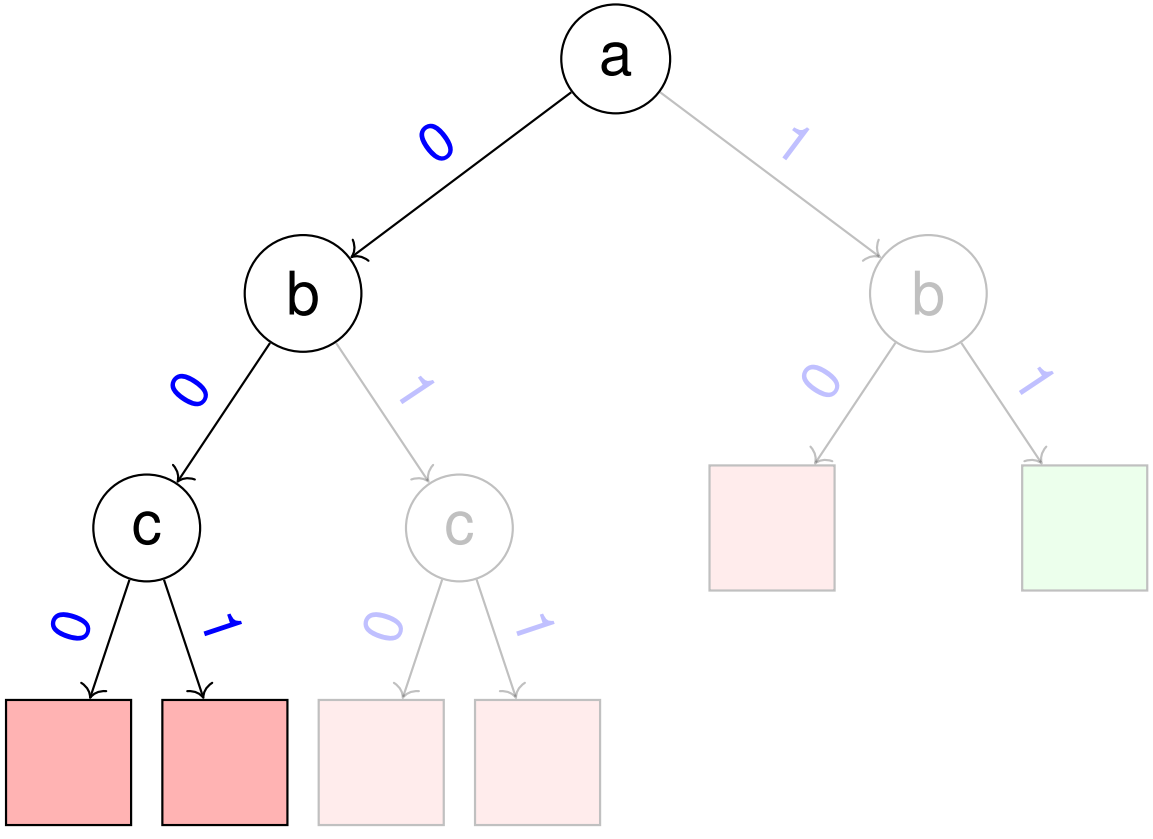
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



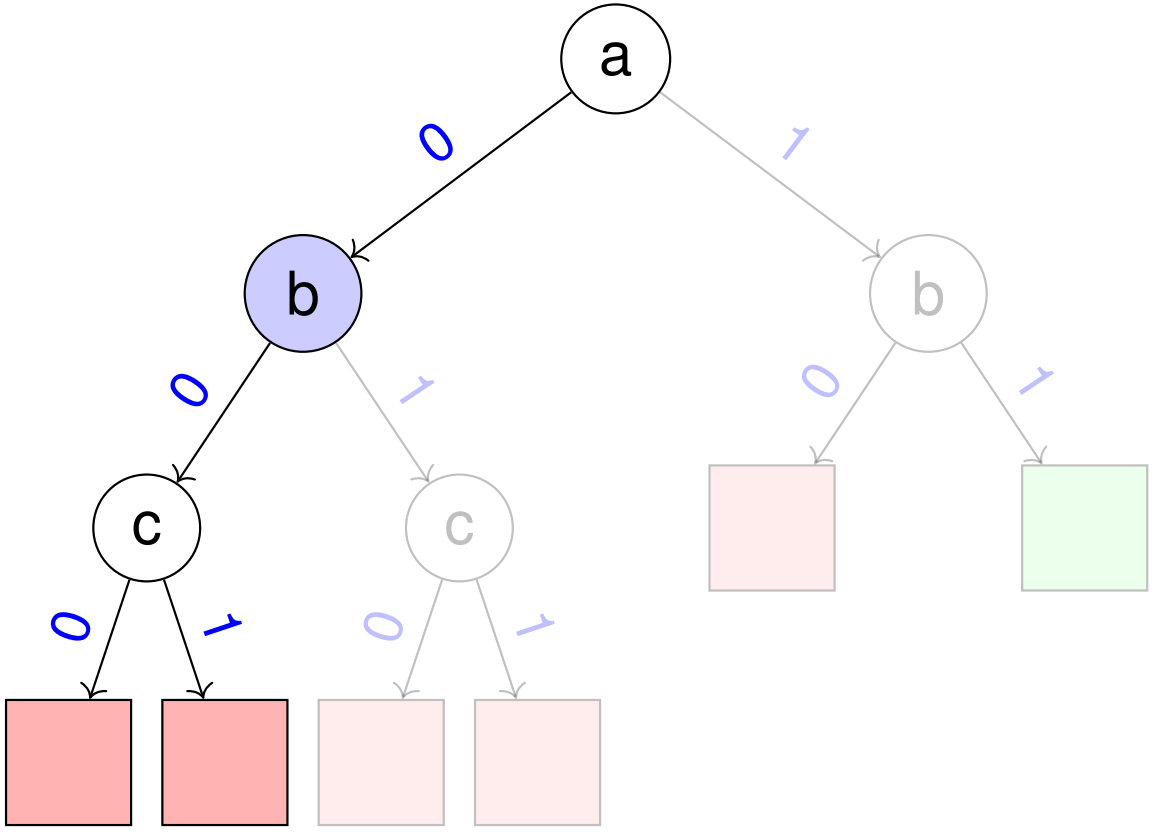
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



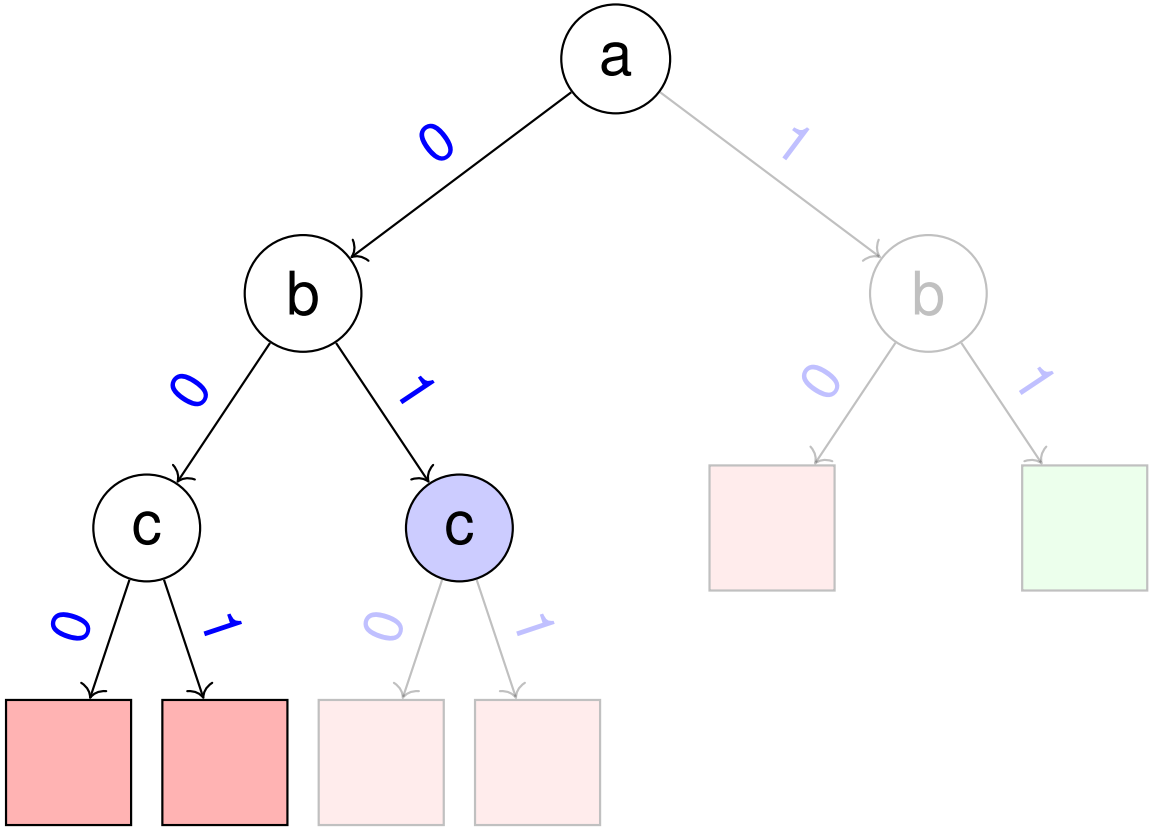
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



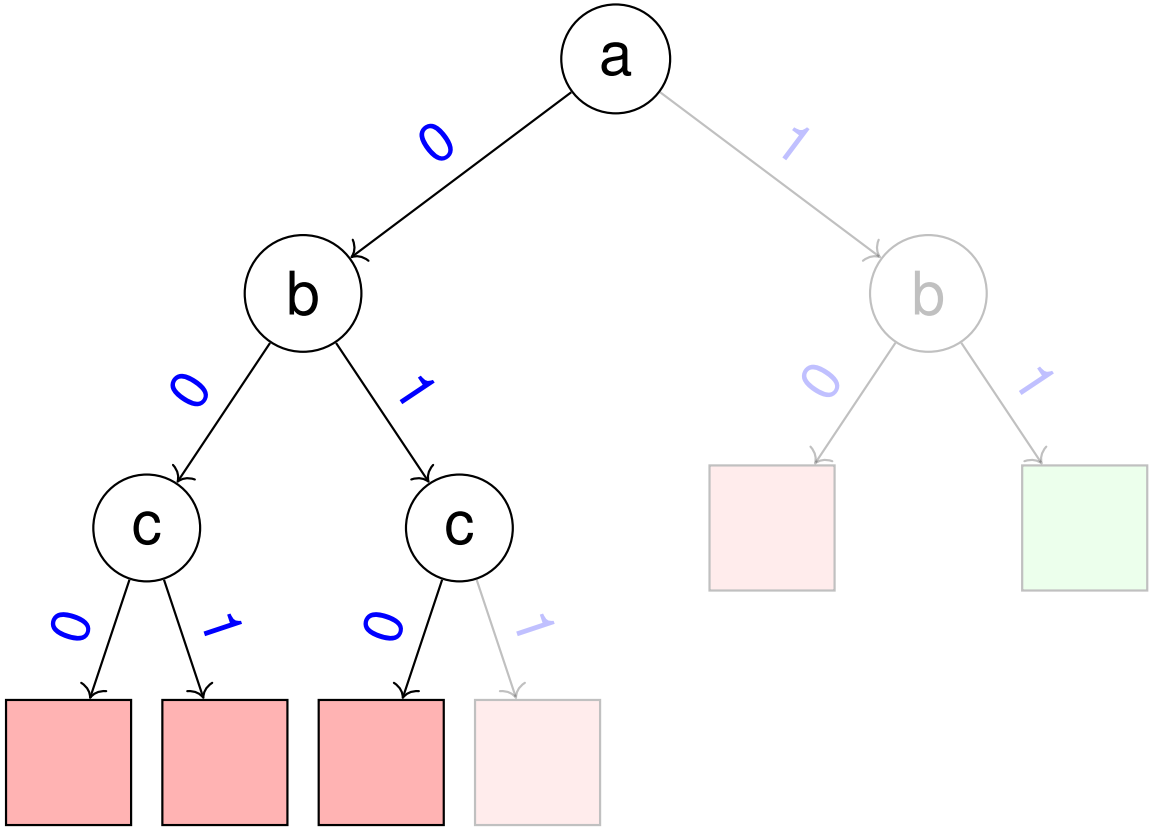
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



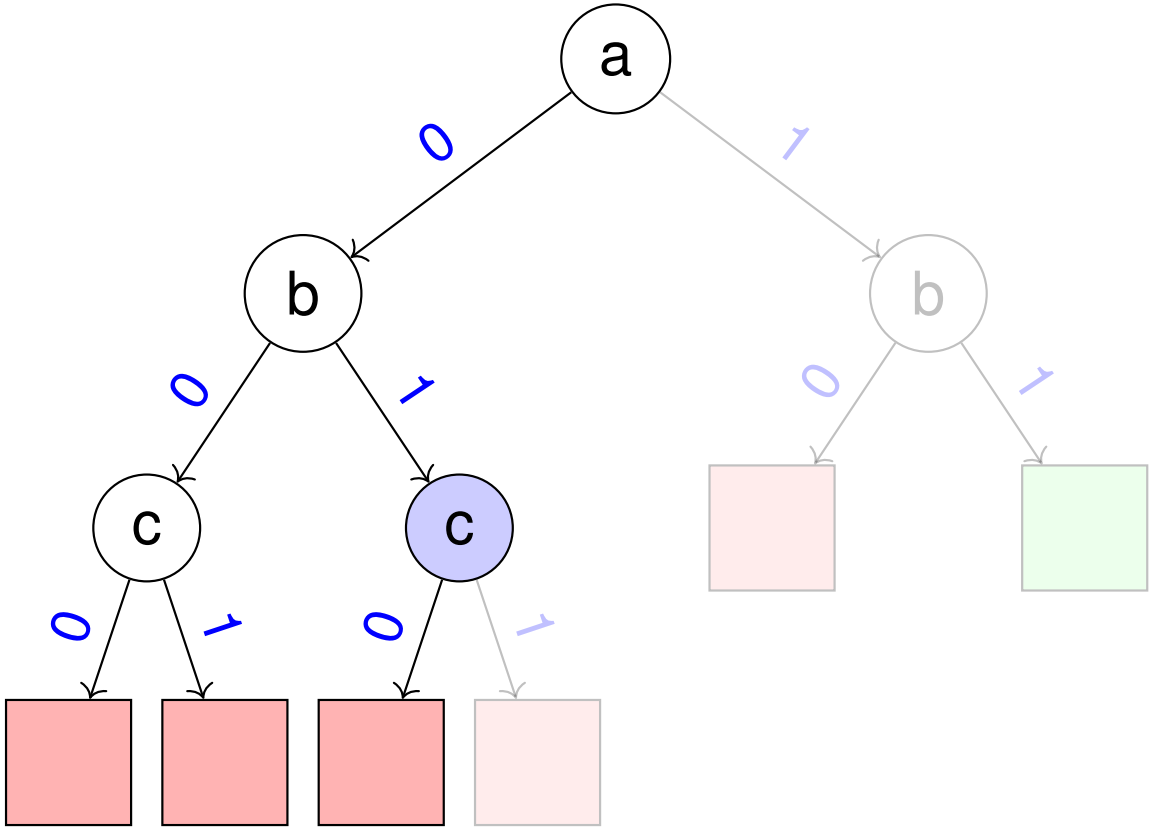
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



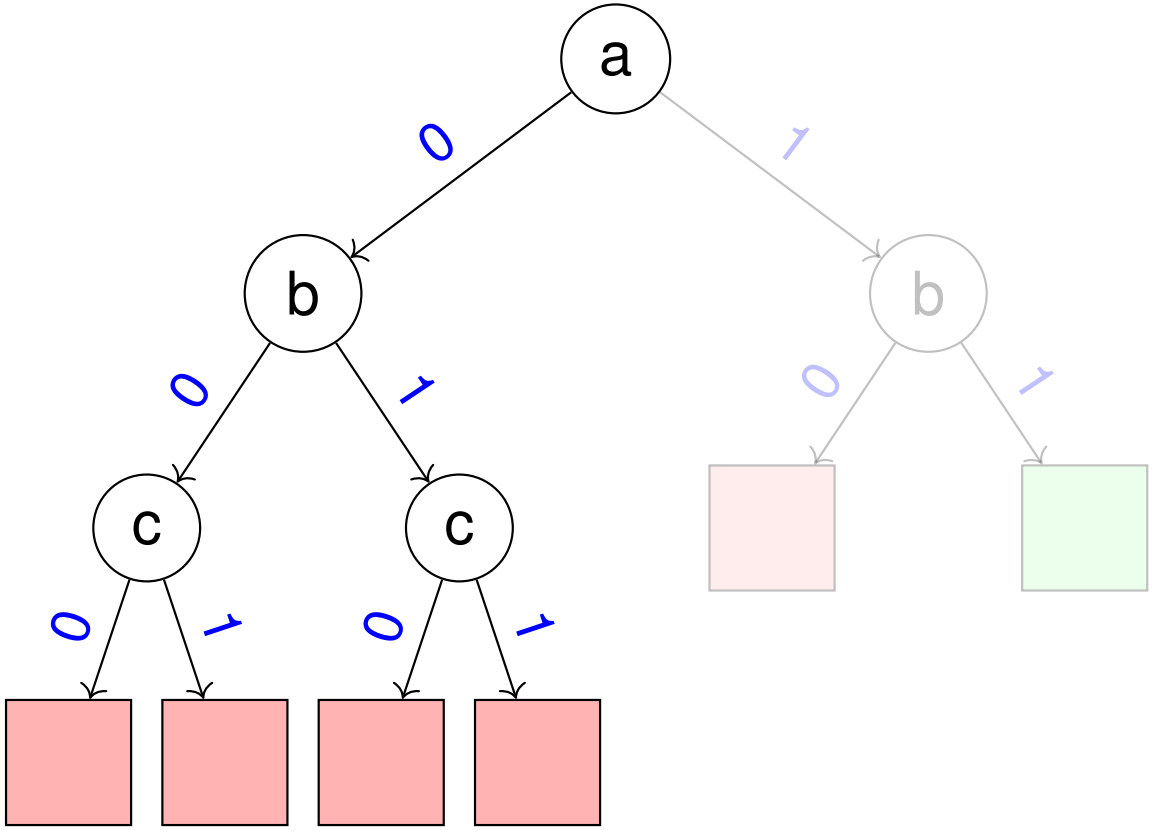
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



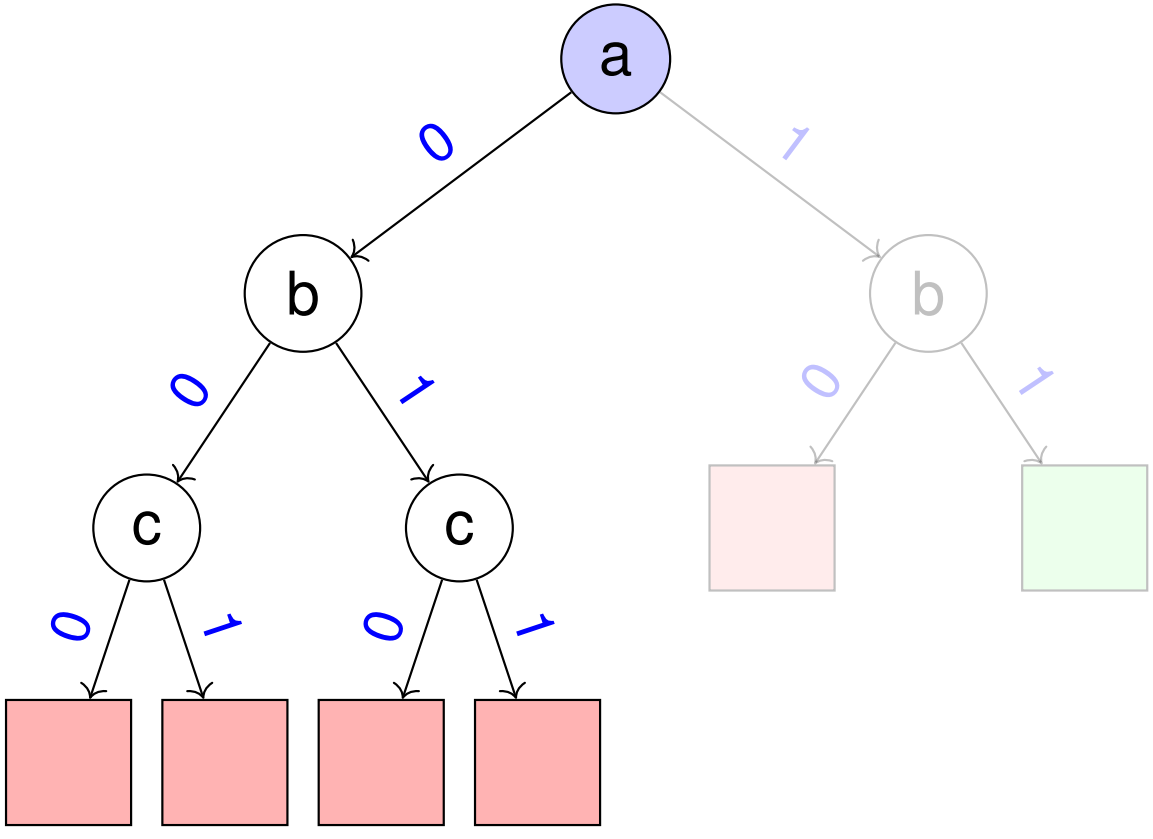
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



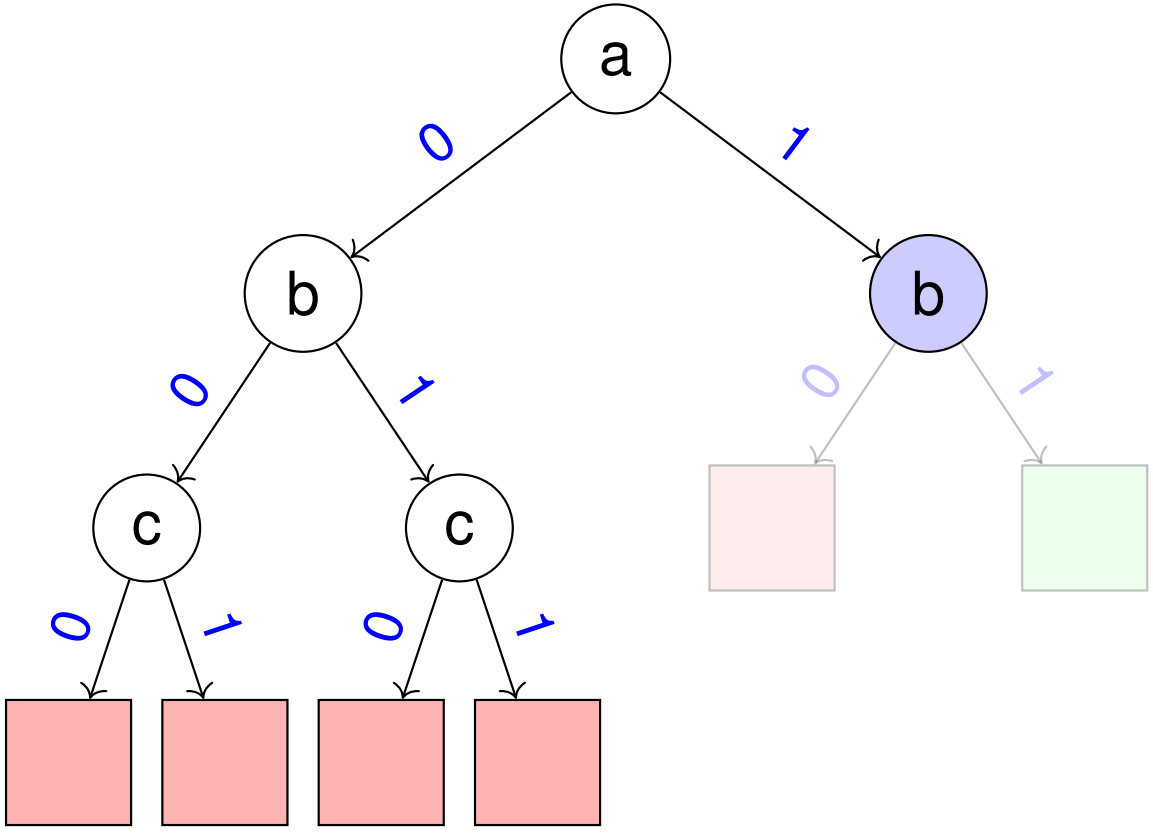
DPPL SAT Solving in a Nutshell

$$(\neg a \vee b \vee c)$$
$$(a \vee c \vee d)$$
$$(a \vee c \vee \neg d)$$
$$(a \vee \neg c \vee d)$$
$$(a \vee \neg c \vee \neg d)$$
$$(\neg b \vee \neg c \vee d)$$
$$(\neg a \vee b \vee \neg c)$$
$$(\neg a \vee \neg b \vee c)$$



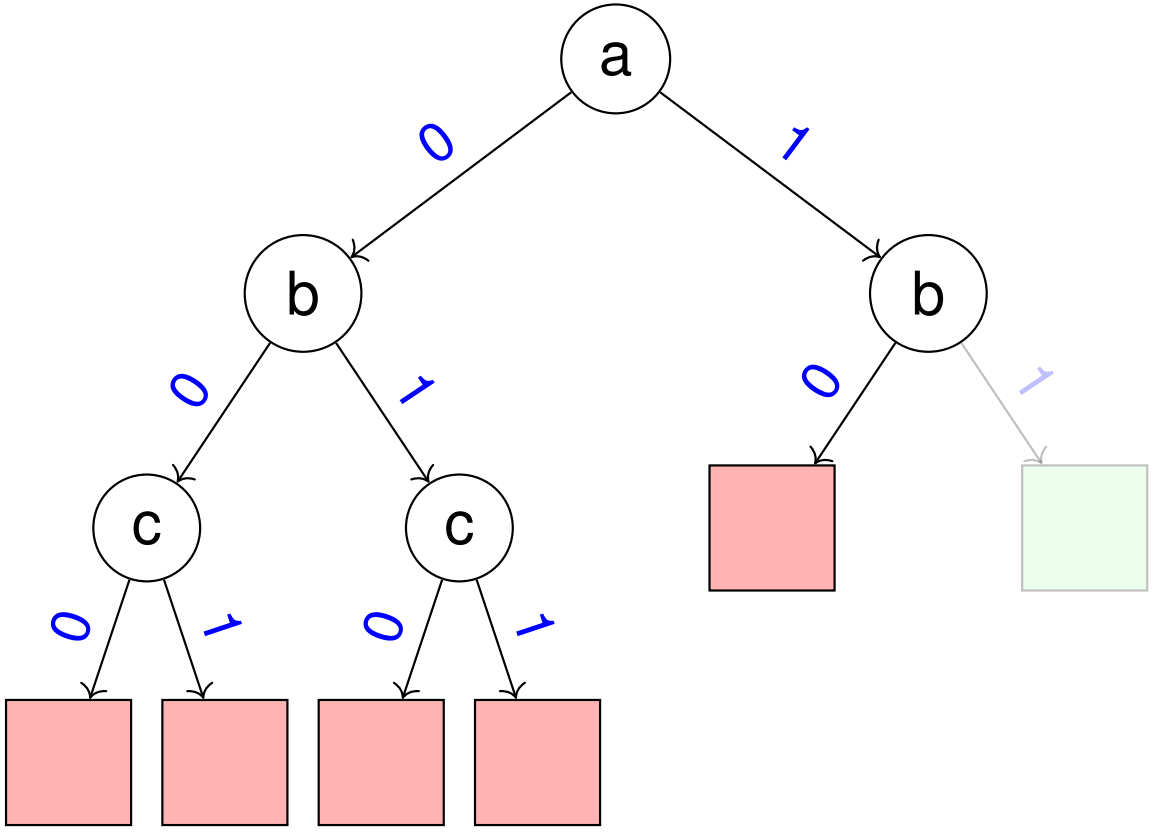
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



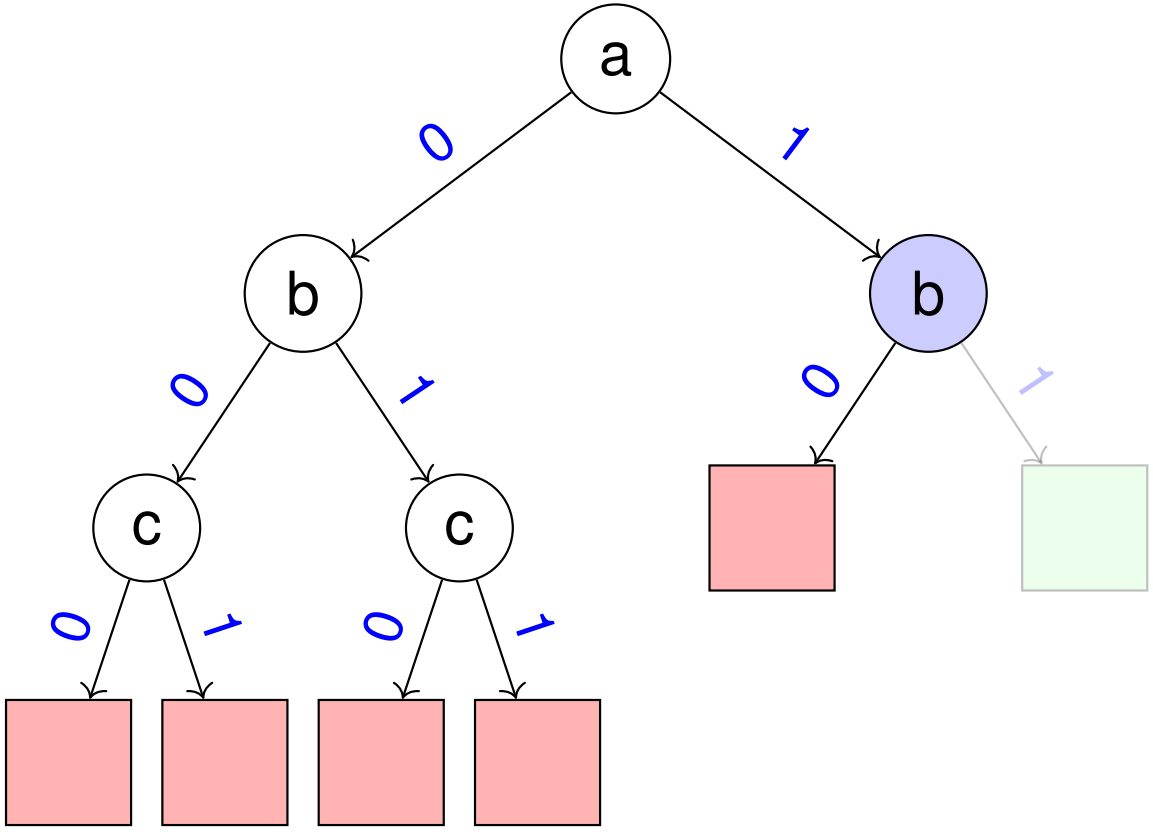
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



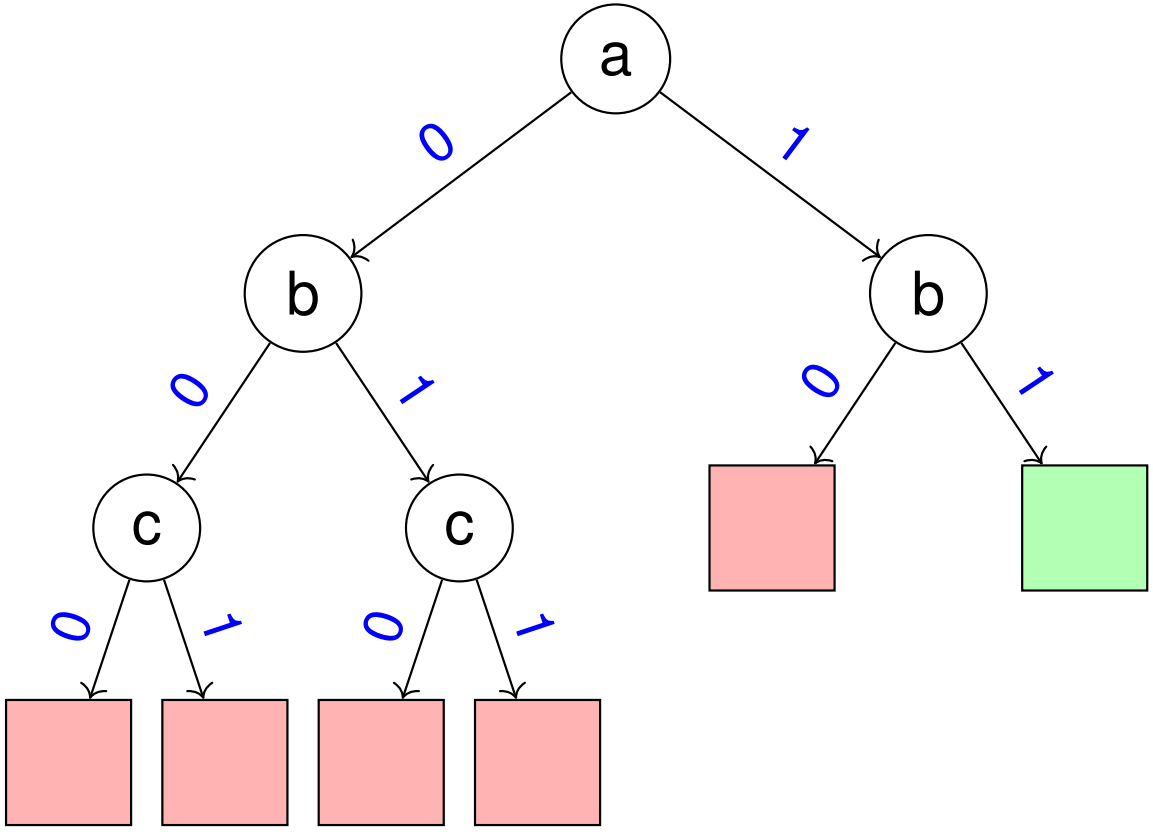
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



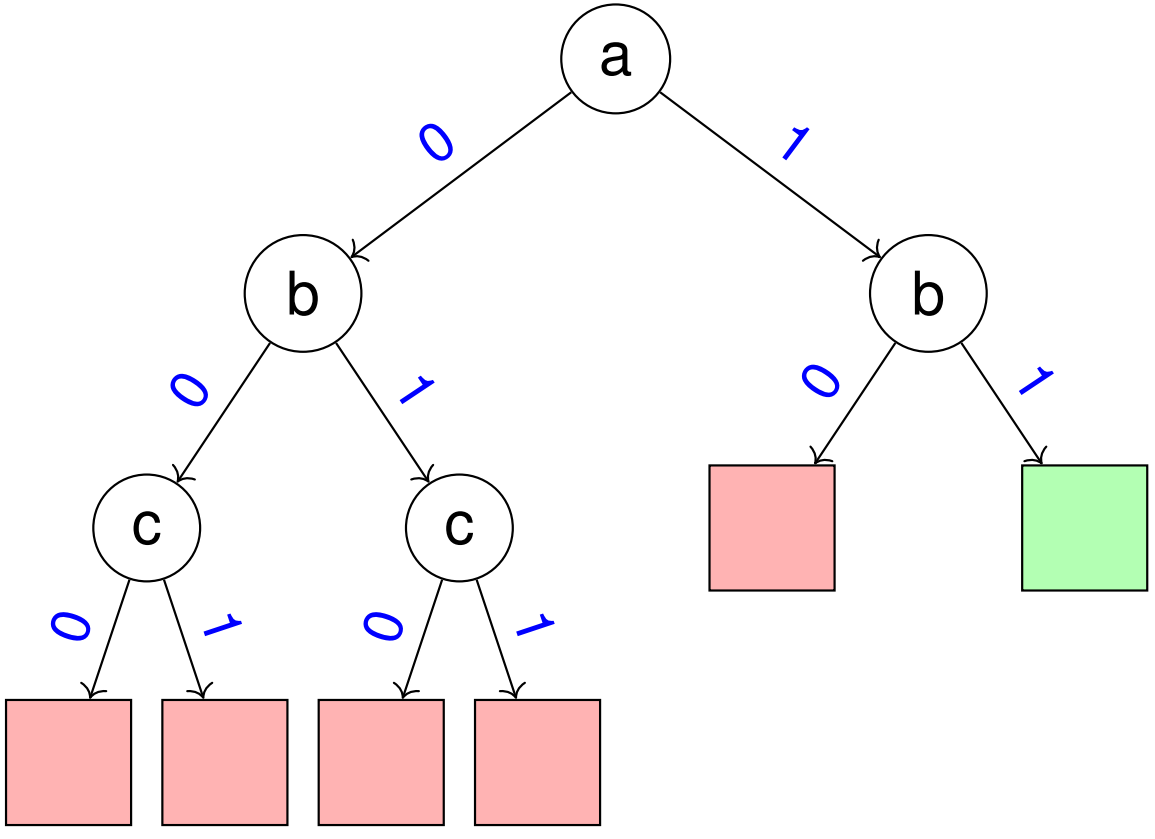
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



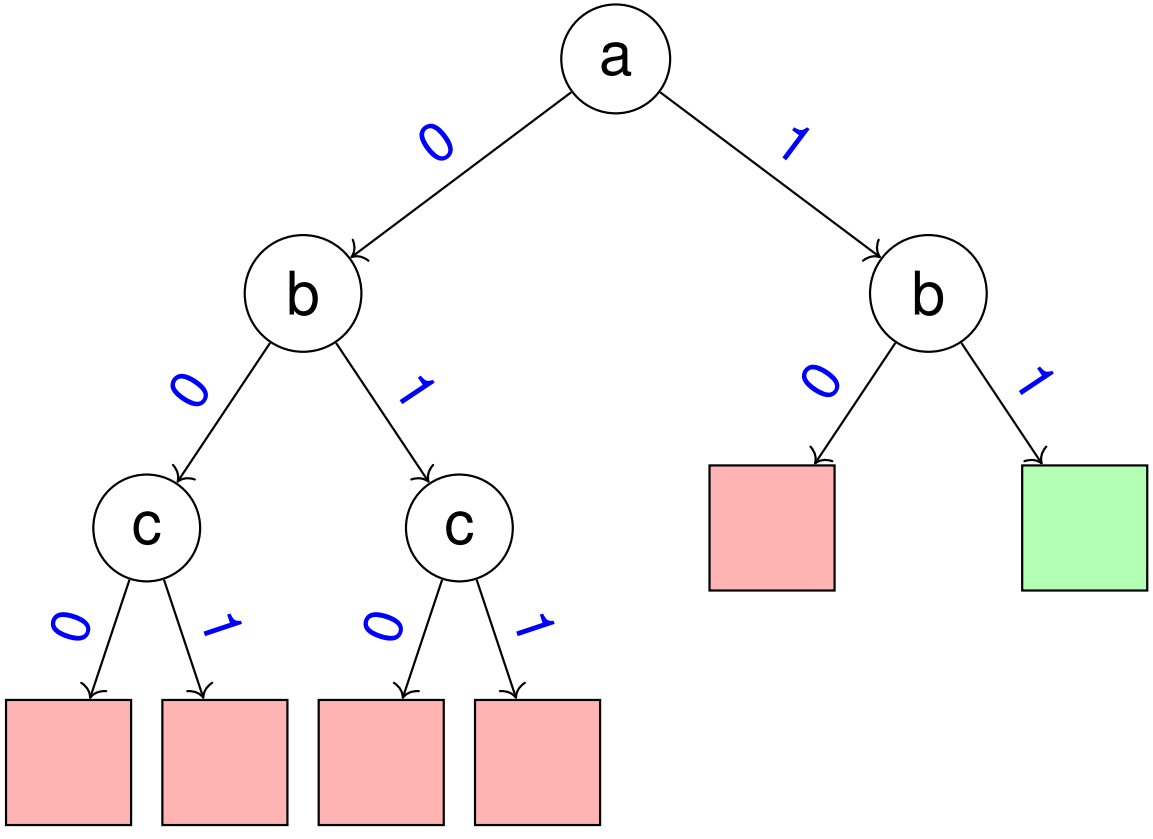
DPPL SAT Solving in a Nutshell

$(\neg a \vee b \vee c)$
$(a \vee c \vee d)$
$(a \vee c \vee \neg d)$
$(a \vee \neg c \vee d)$
$(a \vee \neg c \vee \neg d)$
$(\neg b \vee \neg c \vee d)$
$(\neg a \vee b \vee \neg c)$
$(\neg a \vee \neg b \vee c)$



DPPL SAT Solving in a Nutshell

$$(\neg a \vee b \vee c)$$
$$(a \vee c \vee d)$$
$$(a \vee c \vee \neg d)$$
$$(a \vee \neg c \vee d)$$
$$(a \vee \neg c \vee \neg d)$$
$$(\neg b \vee \neg c \vee d)$$
$$(\neg a \vee b \vee \neg c)$$
$$(\neg a \vee \neg b \vee c)$$



Advancements in SAT Solving

- **1962:** DPLL backtracking algorithm [Davis et al., 1962]
- **1996:** Conflict learning [Silva and Sakallah, 1996]
- **2001:** Local search, decision heuristics, engineering ...

Advancements in SAT Solving

- **1962:** DPLL backtracking algorithm [Davis et al., 1962]
- **1996:** Conflict learning [Silva and Sakallah, 1996]
- **2001:** Local search, decision heuristics, engineering ...

- Modern solvers handle millions of variables and clauses
- Popular solvers:
  - MiniSAT
  - Glucose
  - Lingeling
- Extensions of SAT:
  - Satisfiability Modulo Theories (SMT)
  - Quantified Boolean Formulae (QBF)
  - Max-SAT

Outline

Hardware Verification vs Test

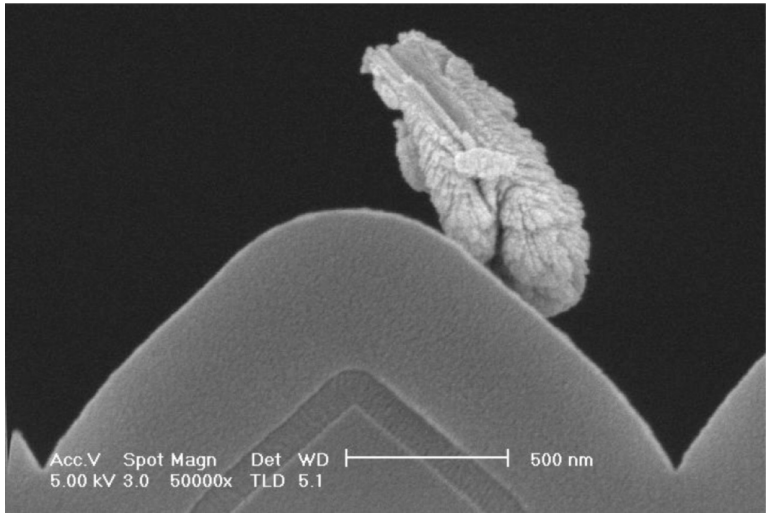
Verification

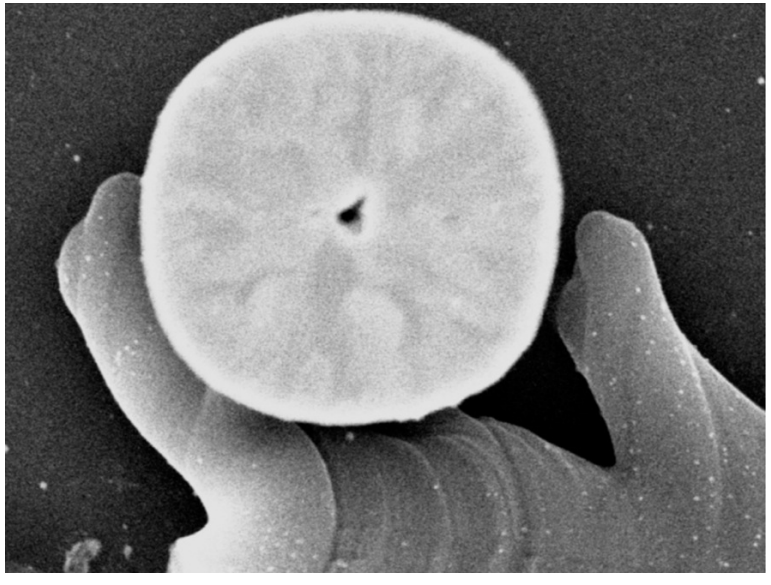
- Detect design bugs
- Extract properties from requirements
- Applied on RTL code
- High manual effort

Test

- Detect physical defects
- Test generation from netlist according to fault model
- Applied on fabricated chips
- High automation

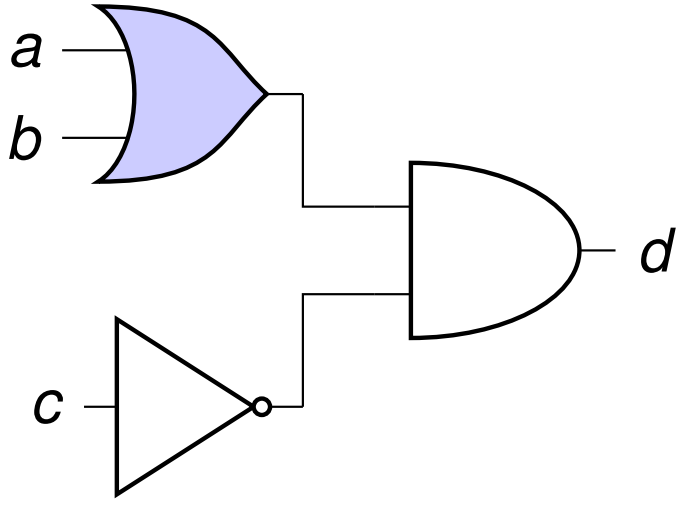
Physical Defects





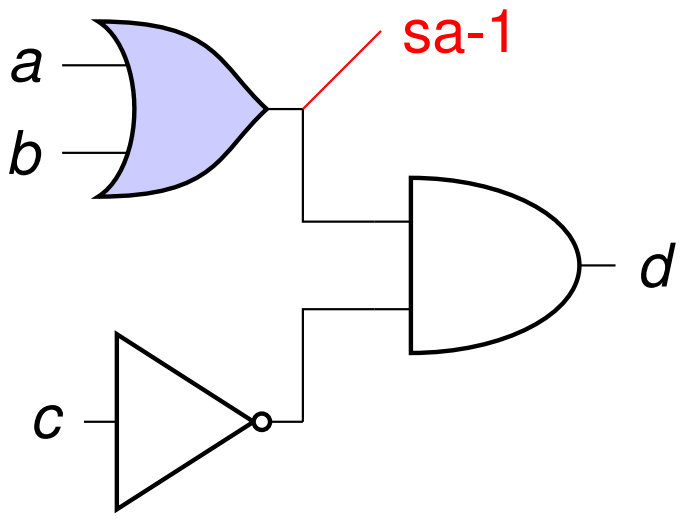
[Source: IEEE Spectrum “The Art of Failure”]

Stuck-at Fault Model



a	b	c	d
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

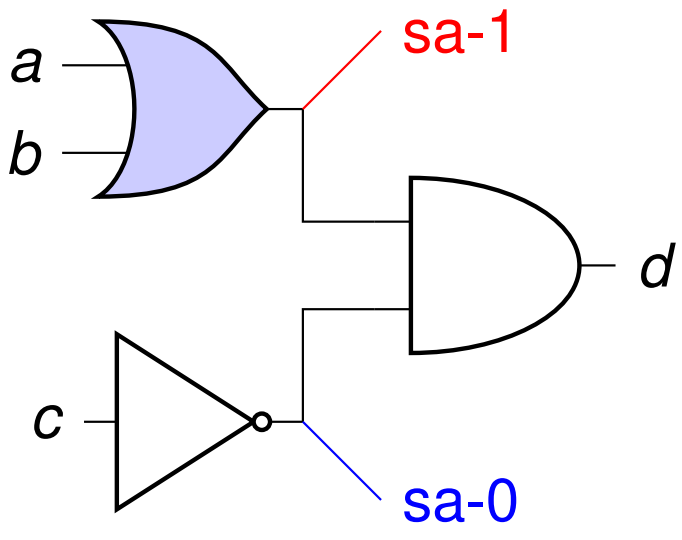
Stuck-at Fault Model



a	b	c	d
0	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	1
1	1	1	0

- $\langle 000 \rangle$ is a test vector for the shown stuck-at-1 fault

Stuck-at Fault Model



a	b	c	d
0	0	0	**0/1**
0	0	1	0
0	1	0	**1/0**
0	1	1	0
1	0	0	**1/0**
1	0	1	0
1	1	0	**1/0**
1	1	1	0

- $\langle 000 \rangle$ is a test vector for the shown stuck-at-1 fault
- $\{\langle 010 \rangle, \langle 100 \rangle, \langle 110 \rangle\}$ are test vectors for the stuck-at-0 fault

Automatic Test Pattern Generation

ATPG

- Create a list of all possible (stuck-at) faults
- For each fault:
  - Find a test pattern
  - Drop all other faults detected by this pattern

- Untestable faults?
- Hard to test faults?
- Sequential tests?
- Test compression?

Thank you for your attention

References I


Cook, S. (1971).
The complexity of theorem proving procedures.
In *3. ACM Symposium on Theory of Computing*, pages 151–158.


Davis, M., Logemann, G., and Loveland, D. (1962).
A machine program for theorem-proving.
Commun. ACM, 5(7):394–397.


Silva, J. a. P. M. and Sakallah, K. A. (1996).
Grasp – a new search algorithm for satisfiability.
In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA. IEEE Computer Society.