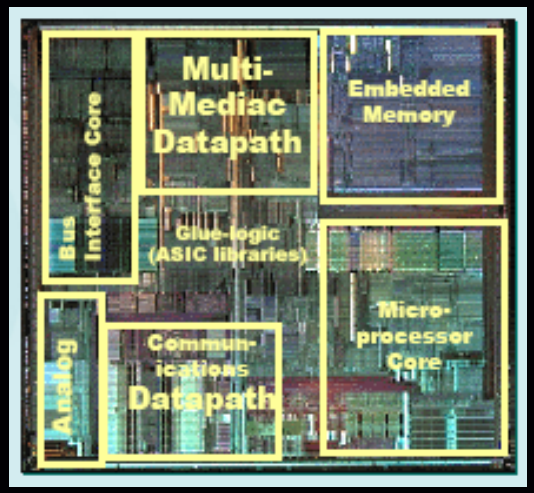
Boolean Satisfiability (SAT) Algorithms

Chung-Yang (Ric) Huang
Dept. EE/GIEE, NTU
FLOLAC, 2009

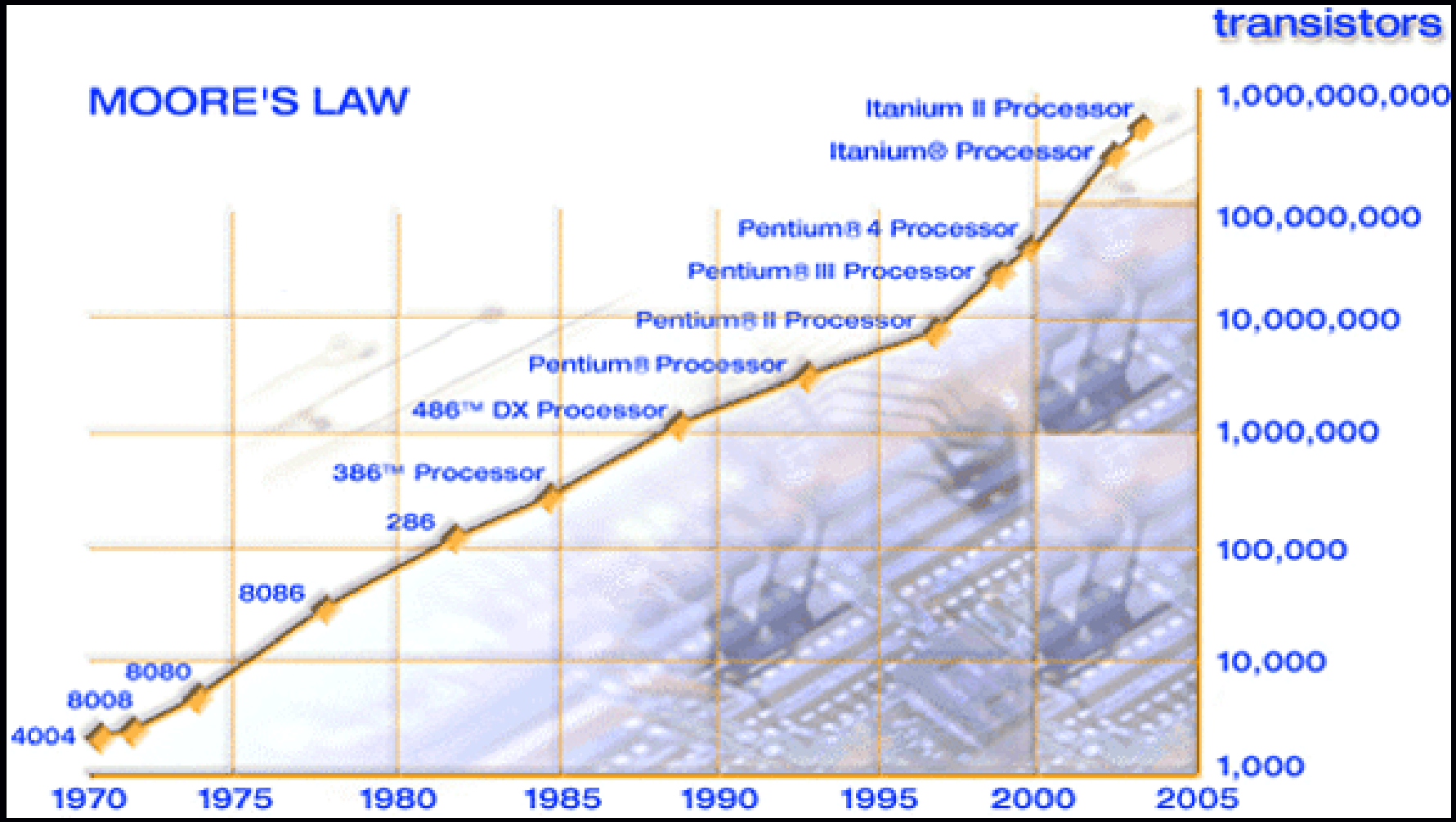
Outline

What is Verification?

- What's the problem?
  - You may have learned that modern IC designs are ---
    - Extremely complex
    - With very high time-to-market pressure



Famous Moore's Law



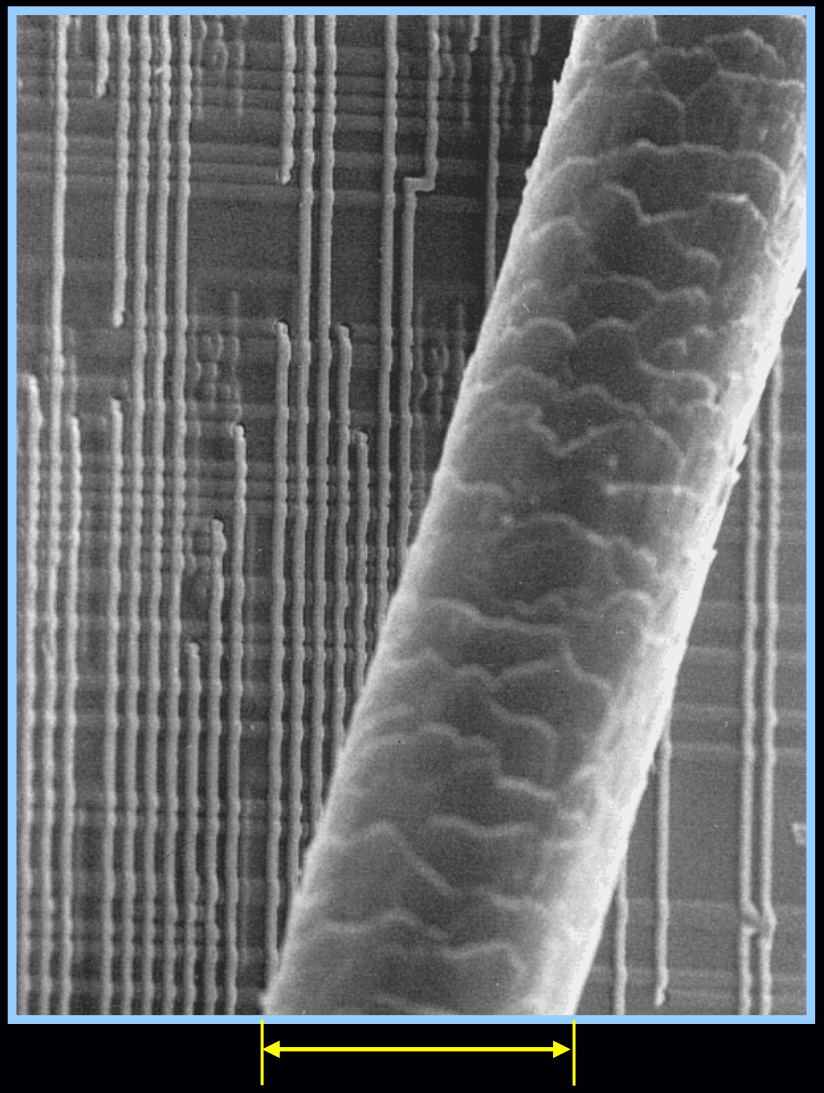
- #Transistors: double/2 years → double/1.5 years

Source: Intel Corp.

#Transistors in Intel's CPUs

	Year of Introduction	Transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium 4 processor	2000	42,000,000
Itanium® processor	2002	220,000,000
Itanium II processor	2003	410,000,000

A Closer Look at Modern IC

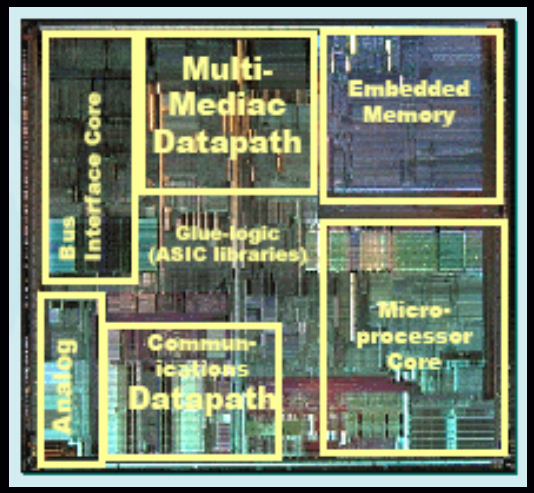


Typical hair diameter: 20 ~ 180 um

Multi-layer metal connection

What is Verification?

- What's the problem?
  - You may have learned that modern IC designs are ---
    - Extremely complex
    - With very high time-to-market pressure



☞ Are you sure your design is correct under all scenarios? (Have you fixed all the bugs?)

Verification: A high stake game

US ‘02 wireless telecom revenue: $76B (CNet)

- 7 M gates
- 500K lines of code

- DoCoMo: 23 M consumers with Java based phones.
- Mandatory recall cost: $4.2B (2001.06)

Kashai/Verisity, HLDVT 2003

And of course,
don’t forget the infamous
Pentium division bug…

Pentium FDIV Bug

- Pentium CPU
  - Shipped on 03/22/93
  - 5V → 3.3V
  - The name “Pentium” was mainly for better trademark protection
  - Superscalar, 64-bit datapath
  - Better floating point calculation
    - 3 X scalar; 5 X vector
- A bug was found in floating-point division
  - By Prof. Thomas Nicely, Lynchburg College, Virginia, USA, 10/30/94

Improvement on Floating-Point Execution

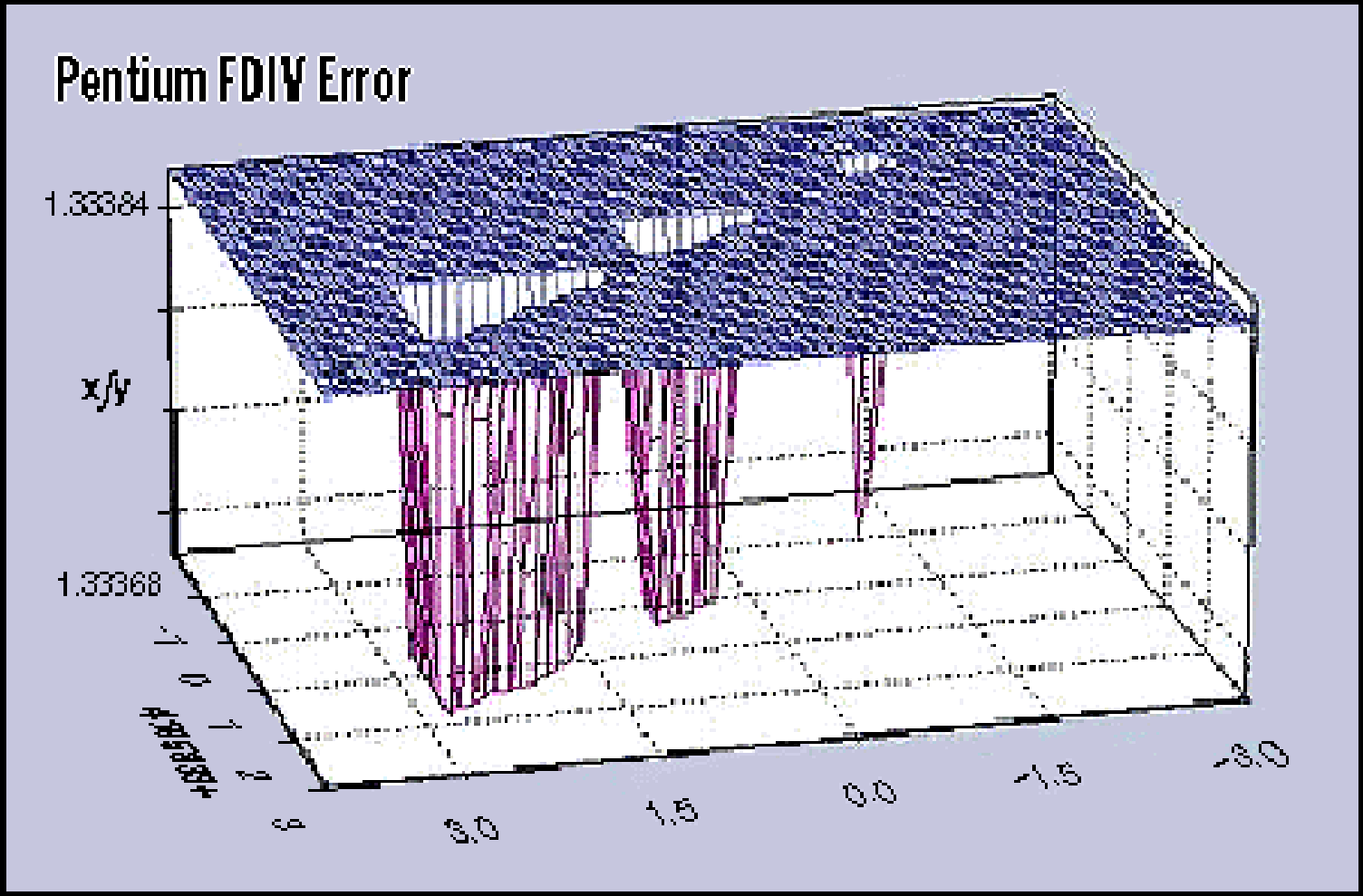
- Using SRT instead of “shift-and-subtract” algorithm for division
  - Generate two quotient bits per clock cycle
  - A lookup table to calculate the intermediate quotients necessary for floating-point division
    - 1066 table entries
- The FDIV bug
  - 5 were not downloaded into the programmable logic array (PLA lookup table)
  - When FPU accesses any of these 5 cells, it fetches zero instead of +2

  ➔ Less precise result!!

The Probability of the FDIV Bug....

- At worst, the 4^{th} significant digit of a decimal number
  - Probability: 1/ 360 Billions
- Most common: 9^{th} and 10^{th} digit
  - Probability: 1 / 9 Billions
- The occurrence of the problem is highly dependent on the input data

➔This result was quickly verified by other people around the Internet, and became known as the **Pentium FDIV bug**



A 3-D plot of the ratio 4195835/3145727 calculated on a Pentium with FDIV bug. The depressed triangular areas indicate where incorrect values have been computed. The correct values all would round to 1.3338, but the returned values are 1.3337, an error in the fifth significant digit. Byte Magazine, March 1995.

Intel's Reaction

- This report stirred up a huge controversy. Intel at first denied that the problem existed.
- Later, Intel claimed that it was not serious and would not affect most users.
  - However, people who could prove that they were affected would get their processor replaced by Intel
- However, although most independent estimates found the bug to be of little importance and have negligible effect on most users, it has caused a great public outcry.
  - Companies like IBM (whose "586" microprocessor competed at that time with the Intel Pentium line) joined the condemnation.
- Finally, Intel was forced to offer to replace all flawed Pentium processors, at huge potential cost to the company
  - However, it turned out that only a small fraction of Pentium owners actually bothered to get their chips replaced
  - Intel's stock price actually rose the day they finally acknowledged

http://en.wikipedia.org/wiki/Pentium_FDIV_bug

Aftermath...

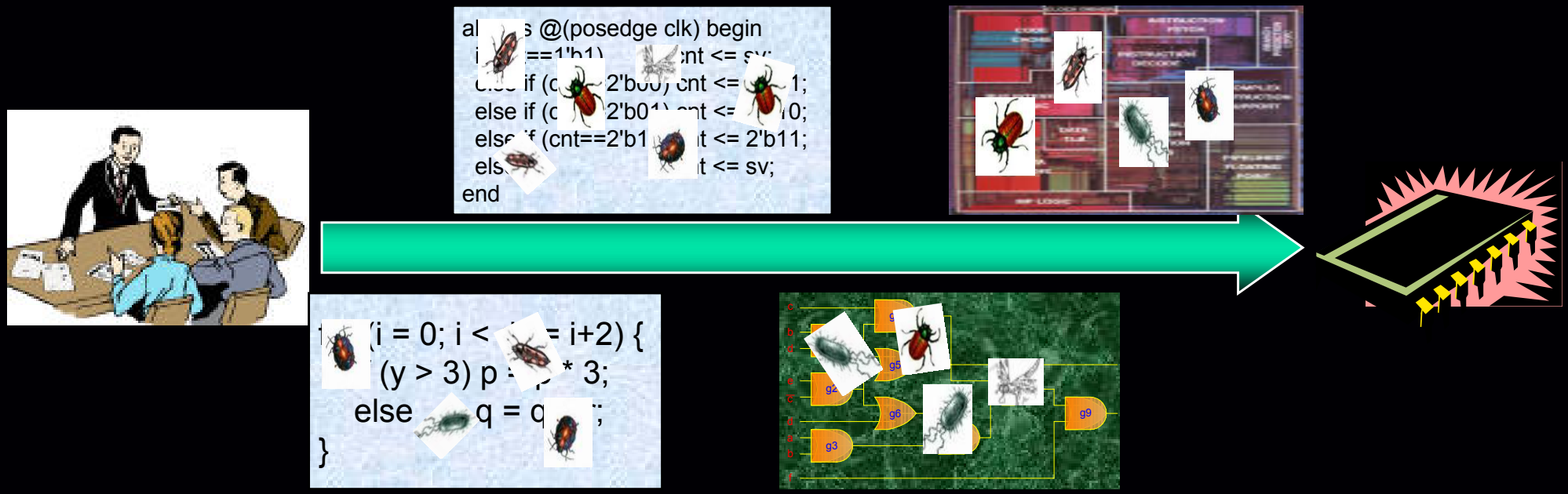
- Intel hired 1000+ PhDs for strategic CAD research, especially in the design verification area
- However, there were still several bugs found in later Pentium chips
  - e.g. CMPXCHG8B instruction, Dan-0411, Pentium FO bugs, etc

Where's the problem?
Why is it so difficult?

☞ Let's first review a little bit about the typical design process…

What is Design Verification?

To verify the correctness of your design



Bugs may exist anywhere in the design...

➔ Find as many design bugs as possible !!

How do you

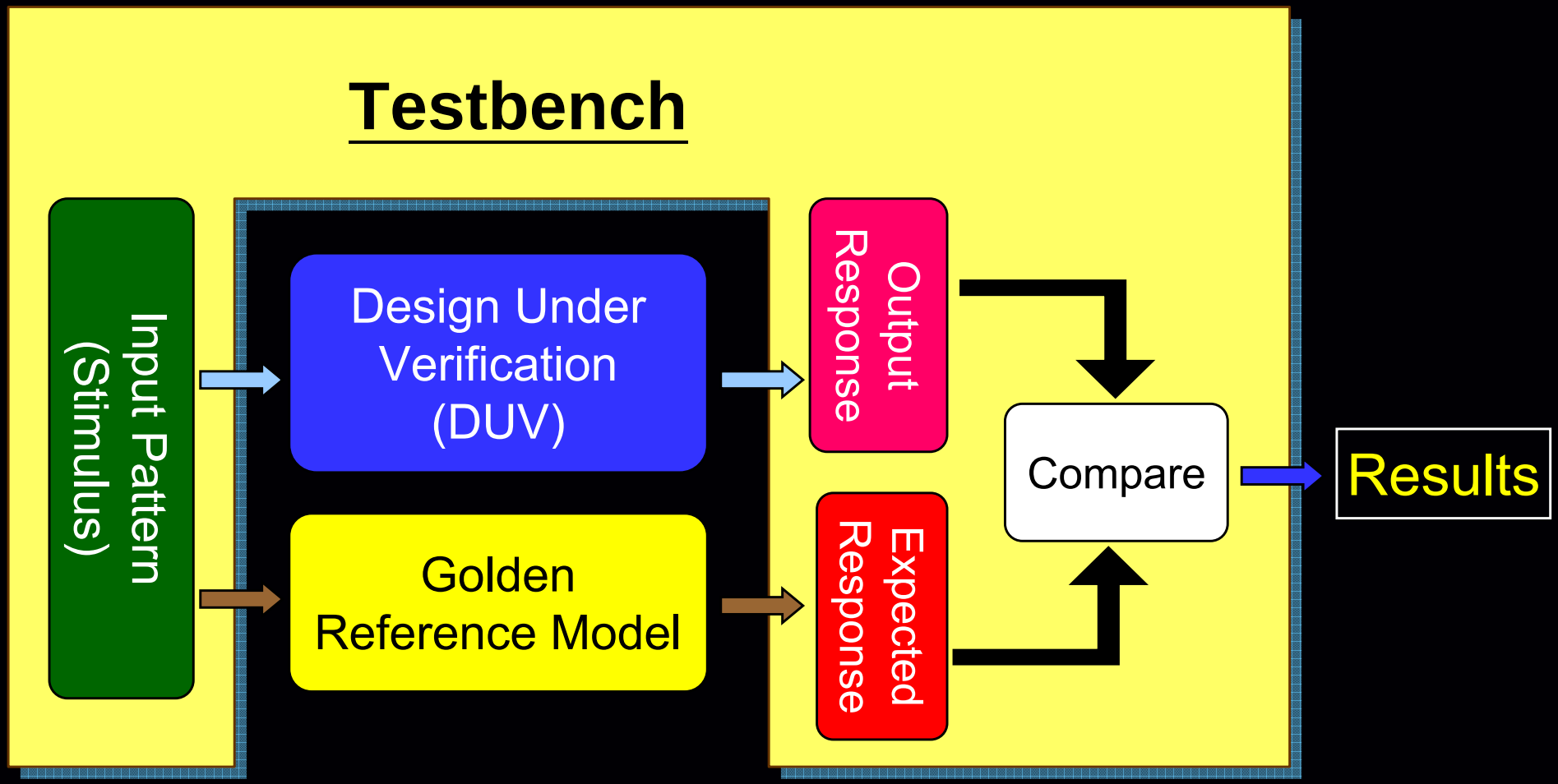
verify

your design

?

Yes, most people verify their designs by simulation and debug by examining the output responses

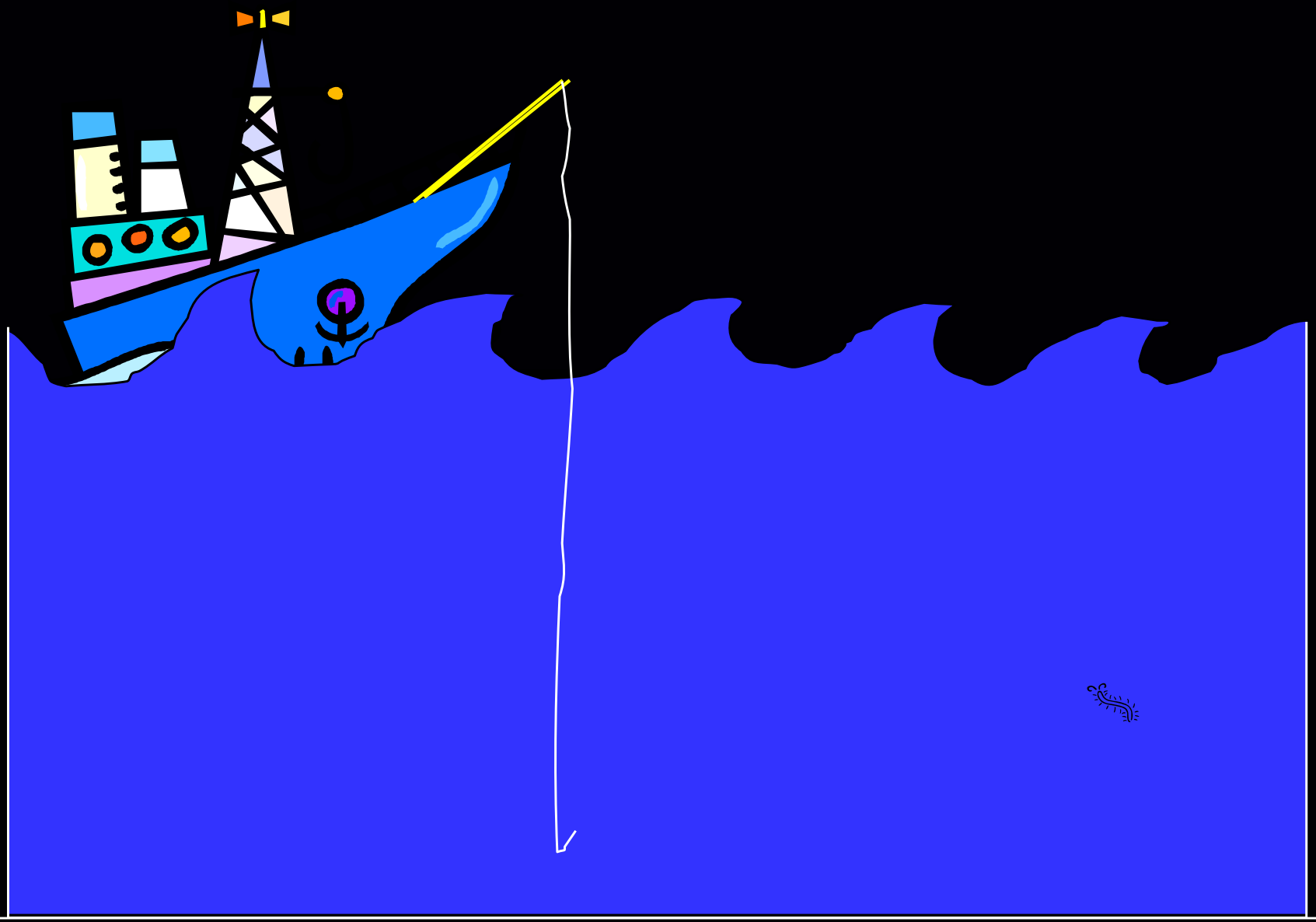
Simulation-Based Verification



Let's do some math

- Suppose a circuit has 100 inputs (this is considered tiny in modern design)
  - Total number of input combinations
    = 2^{100} = 10^{30} = 10^{24}M = 1.6 mole Mega
  - Requires (in the worse case) 10^{24}M test patterns to exhaust all the input scenarios
  - Let alone the sequential combinations
  - For an 1 MIPs simulator
    ➔ runtime = 10^{24} seconds = 3 * 10^{16} years

Like Finding a Bug in an Ocean…



What is worse, when design gets bigger…

➔Simulation runs much slower (exponential complexity)…

- e.g. Time to boot VxWorks
  - 1 million instructions, assume 2 million cycles
  - Today's verification choices:
    - 50M cps: 40 msec Actual system HW
    - 5M cps: 400 msec Logic emulator [1] (QT Mercury)
    - 500K cps: 4 sec Cycle-based gate accelerator [1] (QT CoBALT)
    - 50K cps: 40 sec Hybrid emulator/simulator [2] (Axis)
    - 5K cps: 7 min Event-driven gate accelerator [2] (Ikos NSIM)
    - **50 cps: 11 hr CPU and logic in HDL simulator [3] (VCS)**

1: assumes CPU chip 2: assumes RTL CPU 3: assumes HDL CPU

About VxWorks (http://www.faqs.org/faqs/vxworks-faq/part1/)

source: Kurt Keutzer, UCB

Then why is simulation still the mainstream approach in verification?

How can it be useful?

Simulation is useful because…

1. In early design cycle, most bugs are easy to find
   - Like something contaminates the ocean…
2. Use “design intent” to guide the testbench
   - Guided test pattern generation
   - Real-life stimulus
3. Make the DUV smaller
   - Lower level of hierarchy
   - Higher-level of abstraction
   - Design constraint
4. And ??

And of course,
simulation approach is
easier to learn, simpler to use,

so it is the most popular.

But corner-case bugs are still very tough…

Can we do better?

(Speed up the simulation?)

Using hardware to speed up simulation

1. Emulation
   - Map the design into FPGA
   - 100x - 10000x Speedup; but very expensive
2. Hardware Accelators
   - Compared to emulation

   ➔ lower at cost, but sacrificed in speed
3. Rapid Prototyping
   - Prototyping modules
   - FPGA for IP cores
   - Software
   - Debugging interfaces

Other than using hardware to speed up the simulation…

Let's look at the difference between design verification and manufacturing testing...

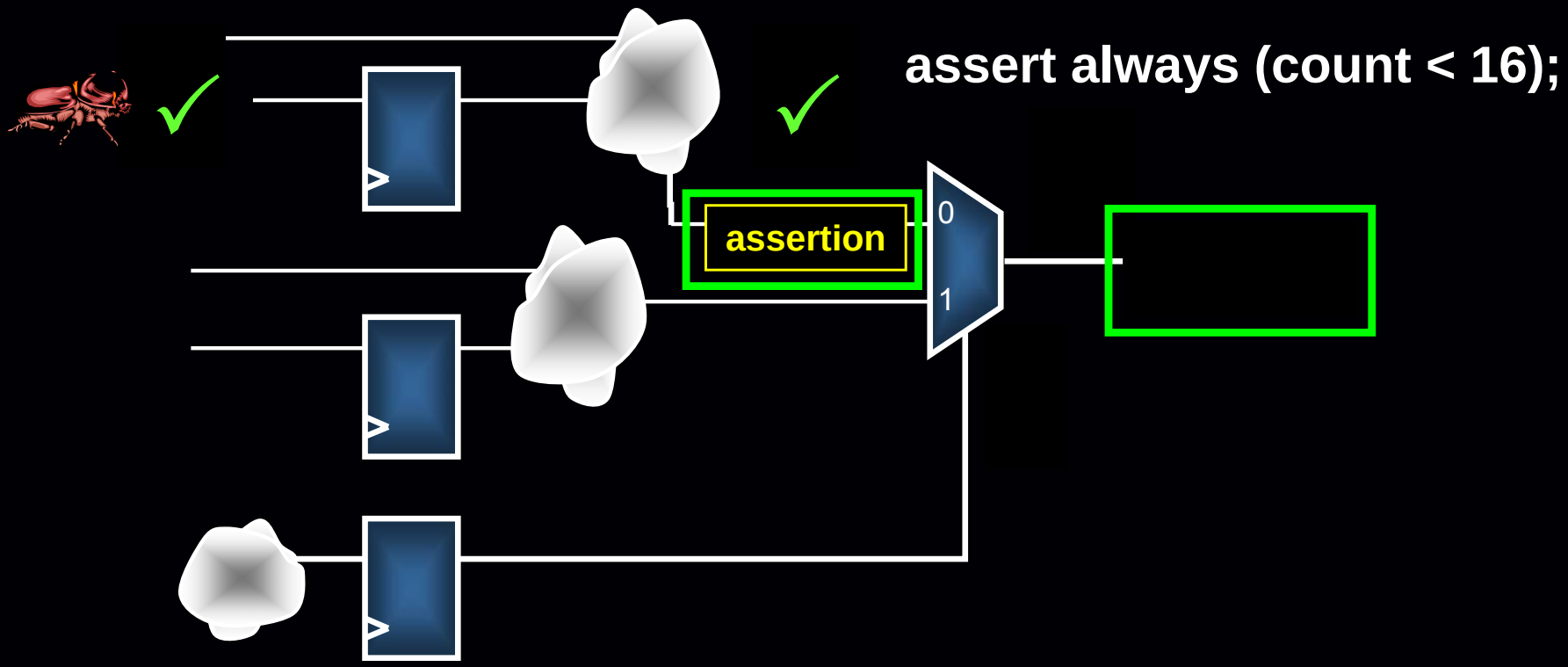
Verification vs. Testing

	Verification	Testing
Objective	Design (SW)	Chip (HW)
Environment	Simulator, debugger (tools)	Test equipments (HW)
Observation points	Any signal in the design	Chip outputs

- Unlike testing, verification does not have the observability problem

Observability Problem

- Bugs are detected at the observation points like POs and registers



source: Harry Foster

Observability Problem --- Solved

- Insert "*assertions*" in the middle of the circuit, and flag the simulator whenever the violation occurs
  - ➔ Increase the observability of the bugs

The difference between
hardware and software simulations

But, how to describe assertions?
Assertion specification languages?

Complex or Simple Assertions?

- With the very high expressiveness of some assertion languges, we can describe very complex design properties by using assertion specification languages
  - However, this may scare off many users...
  - Another bigger problem: how do you know you write the correct property?
    - What if the property you wrote is wrong?

- Assertions should be simple!!
  - The main purpose should be finding bugs
  - Most assertions needed in the design are very simple assertions

The Fact

- More than 90% of properties written for hardware verification are simply "*safety (invariance)* " properties
  - e.g. assert_never(readEn && writeEn);
  - e.g. assert_next(req, ack);

  ➔ Easier to write

  ➔ Higher proof completion percentage

  ➔ Enough to detect bugs

  How to quickly prove all of them is the key issue

ABV Example --- OVL

```
assert_never  underflow  ( clk, reset_n,
     (q_valid==1'b1) && (q_underflow==1'b1));
```


```
module assert_never (clk, reset_
 input clk, reset_n, test_expr;
 parameter severity_level = 0;
 parameter msg="ASSERT NEVER VIOLATION";
// ASSERT: PRAGMA HERE
 //synopsys translate_off
 `ifdef ASSERT_ON
  integer error_count;
  initial error_count = 0;
  always @(posedge clk) begin
   `ifdef ASSERT_GLOBAL_RESET
    if (`ASSERT_GLOBAL_RESET != 1'b0) begin
   `else
    if (reset_n != 0) begin // active low reset_n
   `endif
     if (test_expr == 1'b1) begin
      error_count = error_count + 1;
      `ifdef ASSERT_MAX_REPORT_ERROR
       if (error_count<=`ASSERT_MAX_REPORT_ERROR)
      `endif
         $display("%s : severity %0d : time %0t : %m", msg, severity_level, $time);
      if (severity_level == 0) $finish;
     end
    end
  end
 `endif
//synopsys translate_on
endmodule
```

source: Harry Foster

ABV Example --- OVL

```
module assert_never (clk, reset_n,
    test_expr);
  input clk, reset_n, test_expr;
  parameter severity_level = 0;
  parameter msg="ASSERT NEVER
    VIOLATION";
// ASSERT: PRAGMA HERE
  //synopsys translate_off
  `ifdef ASSERT_ON
    integer error_count;
    initial error_count = 0;
    always @(posedge clk) begin
      `ifdef ASSERT_GLOBAL_RESET
        if (`ASSERT_GLOBAL_RESET !=
      1'b0) begin
      `else
        if (reset_n != 0) begin
          // active low reset_n
      `endif
        if (test_expr == 1'b1) begin
          error_count = error_count + 1;
          `ifdef
        ASSERT_MAX_REPORT_ERROR
          if (error_count <=
        `ASSERT_MAX_REPORT_ERROR)
          `endif
            $display("%s : severity %0d :
                time %0t : %m", msg,
                severity_level, $time);
          if (severity_level == 0) $finish;
      end
    end
  end
  `endif
 //synopsys translate_on
endmodule
```

OVL based SVA library

assert_always
assert_always_on_edge
assert_change
assert_cycle_sequence
assert_decrement
assert_delta
assert_even_parity
assert_fifo_index
assert_frame
assert_handshake
assert_implication
assert_increment
assert_never
assert_next
assert_no_overflow
assert_no_transition
assert_no_underflow
assert_odd_parity
assert_one_cold
assert_one_hot
assert_proposition
assert_quiescent_state
assert_range
assert_time
assert_transition
assert_unchange
assert_width
assert_win_change
assert_win_unchange
assert_window
assert_zero_one_hot

Any Problem? Whose Problem?

"My biggest problem about design verification is that *the time is never enough*. I can only do my best to run more simulation, but I don't know whether it is sufficient."

--- Broadcom designer (similar comments from many others)

"It is very hard to write the testbenches and assertions for the design, since *I am not a designer*. Ask the designer to do it more? No way!!"

--- Sun Microsystems verification engineer (similar comments from many others)

“Where am I going to find time to write assertions? I don’t even have time to write comments!”

--- Conexant design engineer

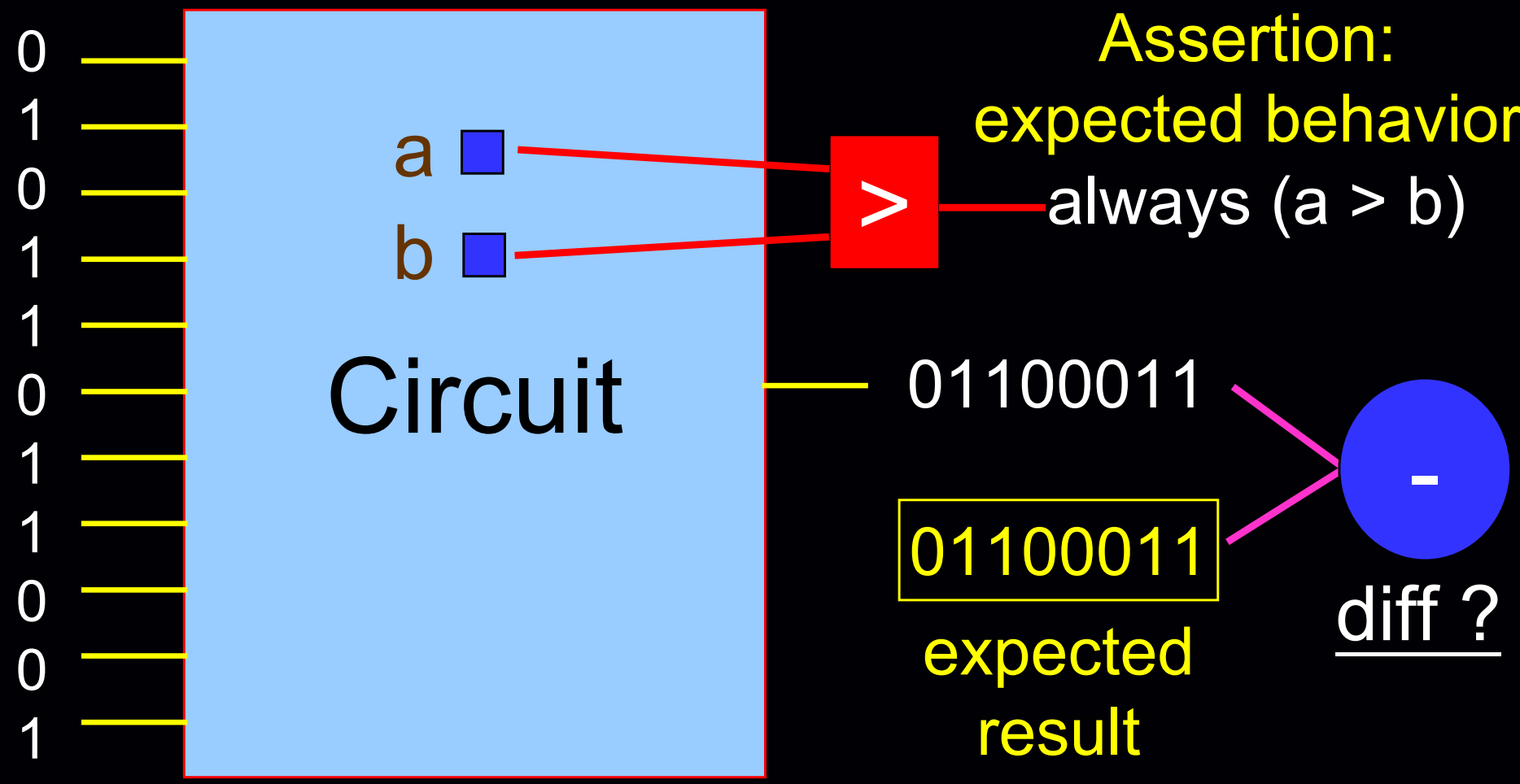
Think ---

- If you simulate your design for 3 days and 3 nights, and you don't find any new bug. And the "coverage metric" says that the coverage is 100%. What does that mean?
  - Is your design bug-free?
  - Or the quality of your testbench is NOT good?
- If a miss of a bug in the HW chip may cost you millions of dollars…
  - Can you sleep nice without further seeking any "proof" for the correctness of your design?

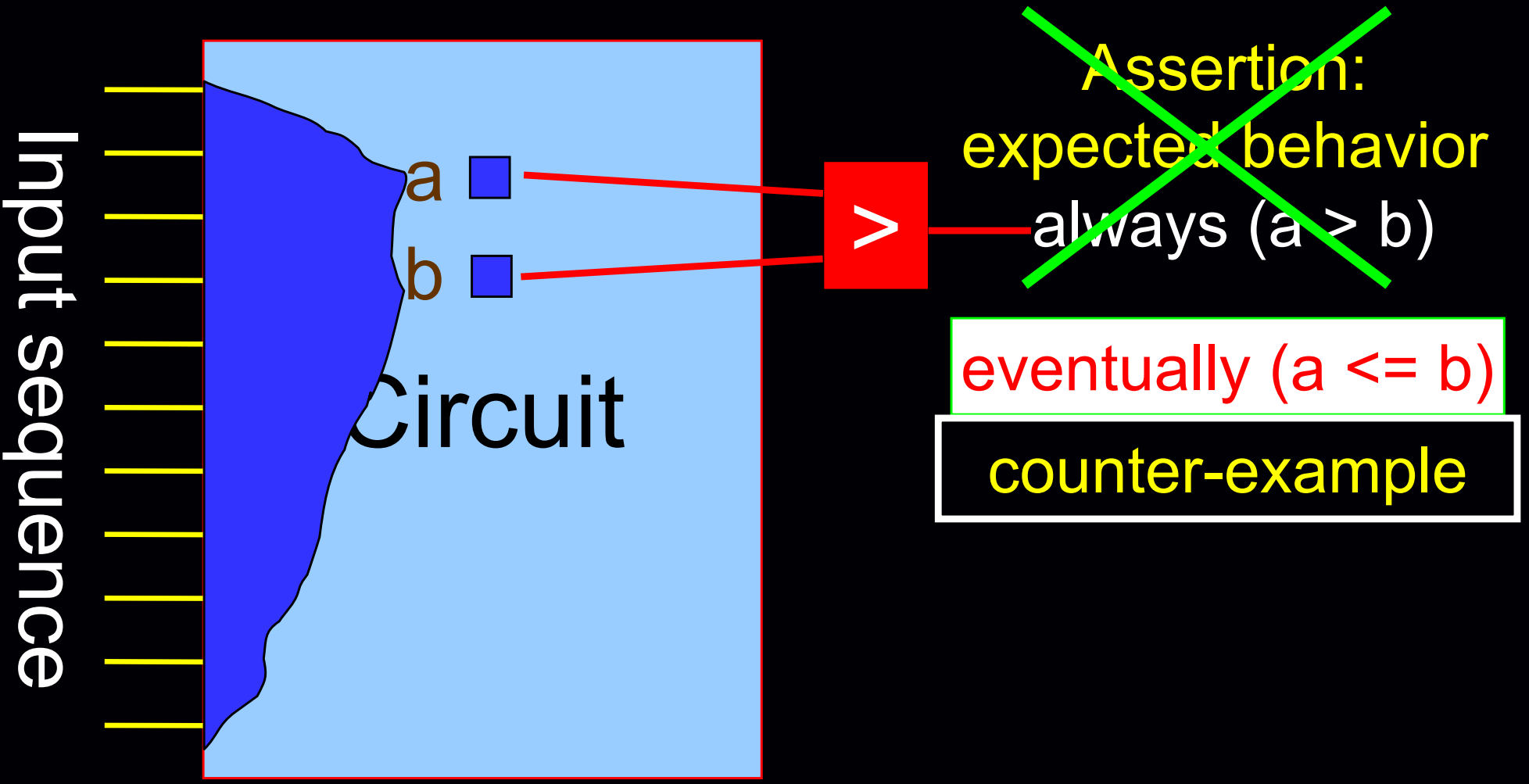
Can simulation answer these questions?

Any alternative to simulation/emulation?

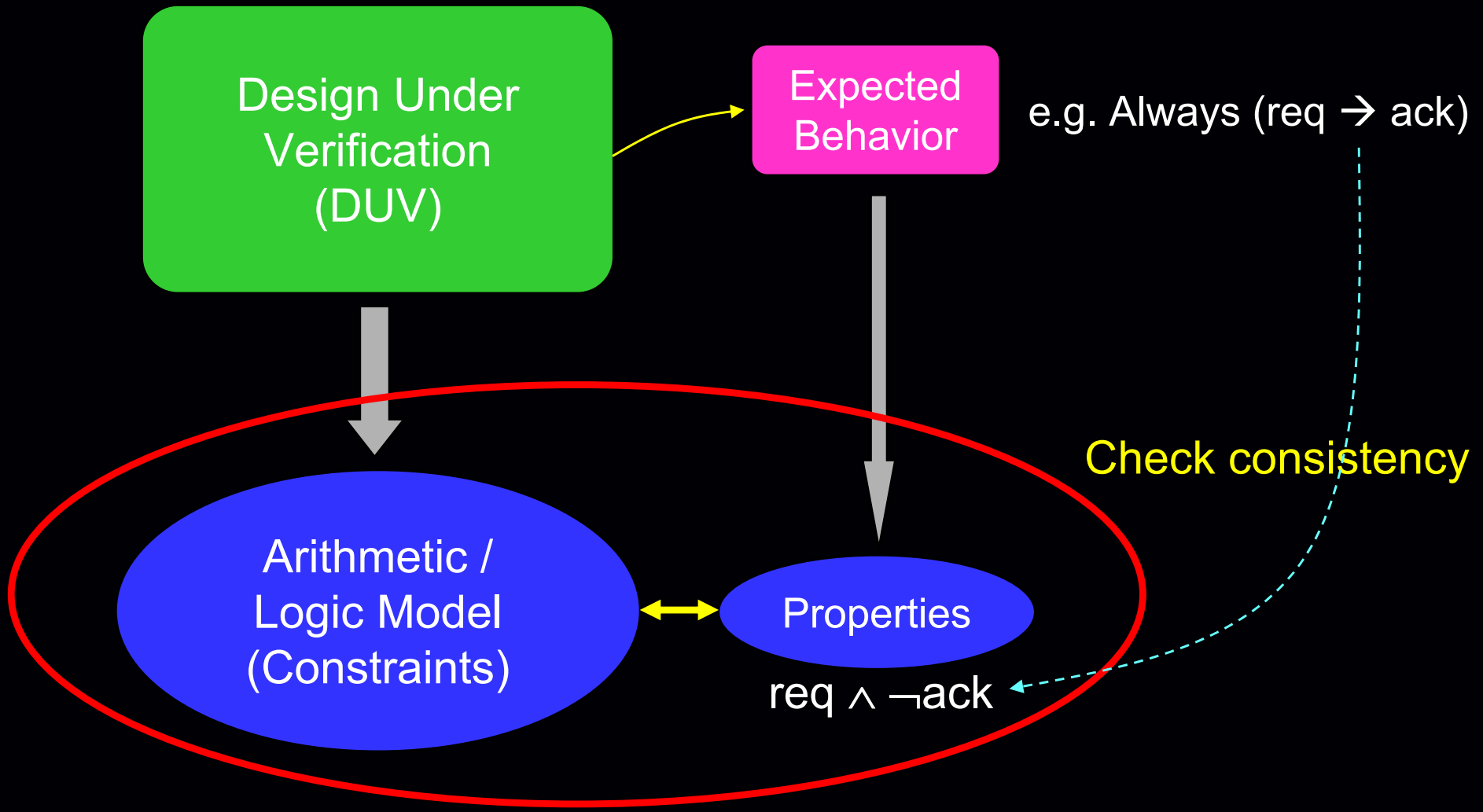
Simulation and Assertion-Based Verification



Finding Counter-Example

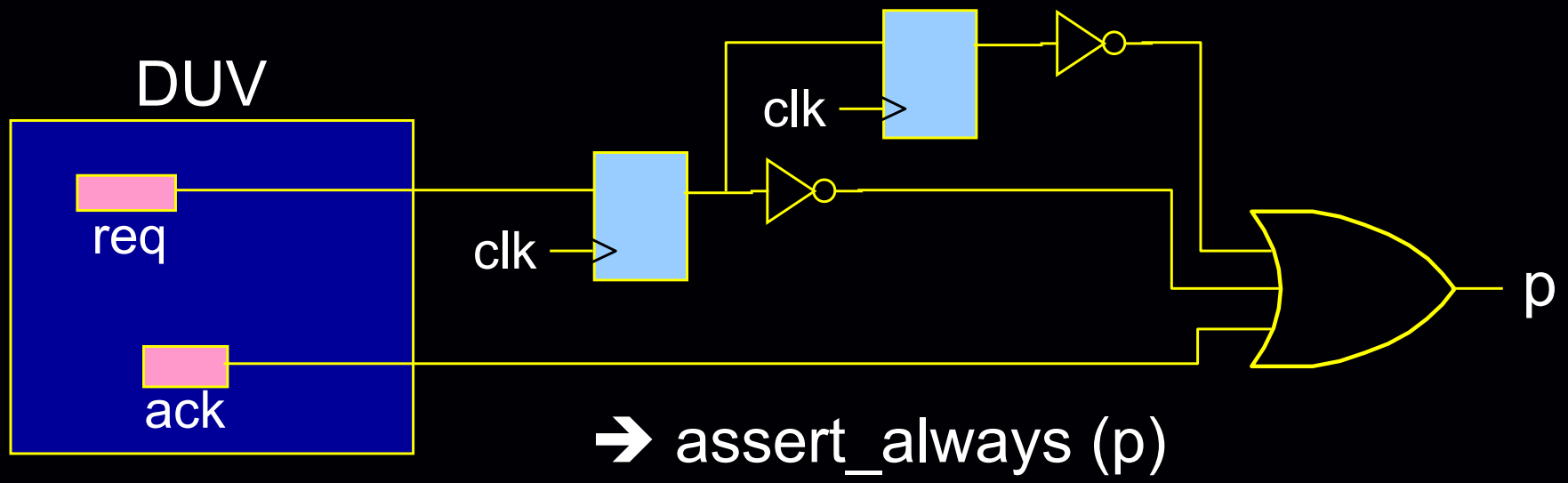


Formal Verification Technologies



Safety Property (Invariance)

- Something GOOD should always hold;
  Something BAD should never happen
- Without lost of generality, an assertion property on a circuit can be transformed into an “assert_always (atomic_proposition)” property with some extra gates
  - e.g. assert_never(p) ≡ assert_always(¬p);
  - e.g. assert property
    (@(posedge clk) req |-> ##[1:2] ack)



Proving “assert_always(p)”

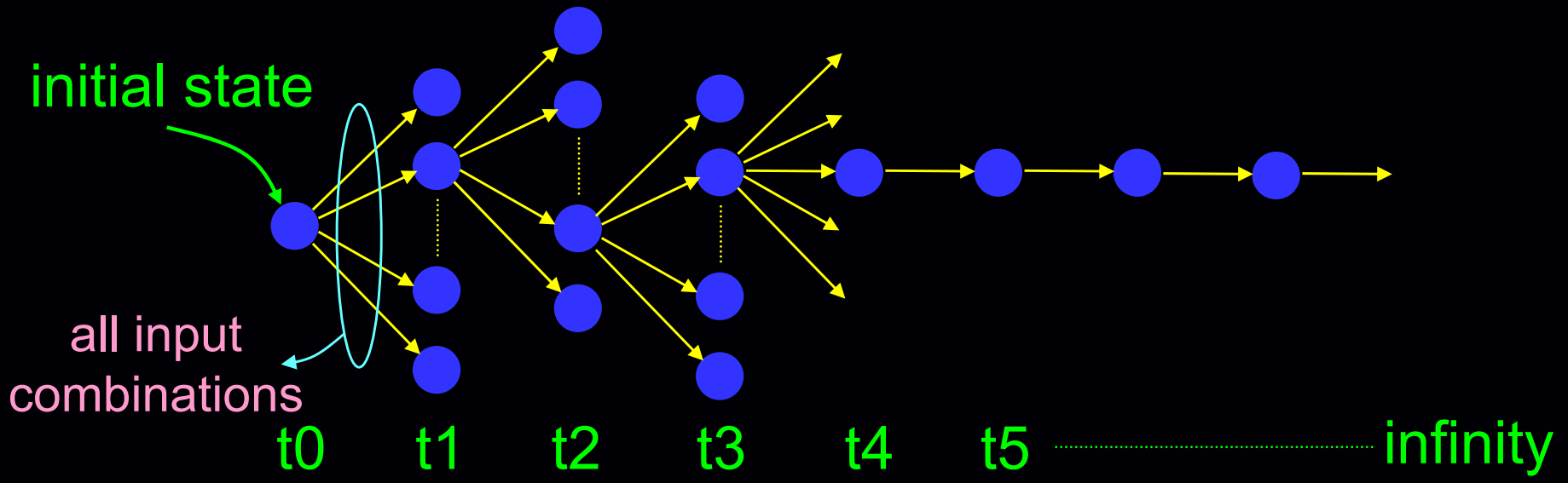
- In later slides, we will mostly focus on how to prove the property “assert_always(p)”, instead of proving a complex temporal logic formula

- assert_always(p) ≡ AG (p) ≡ ¬ EF (¬p)

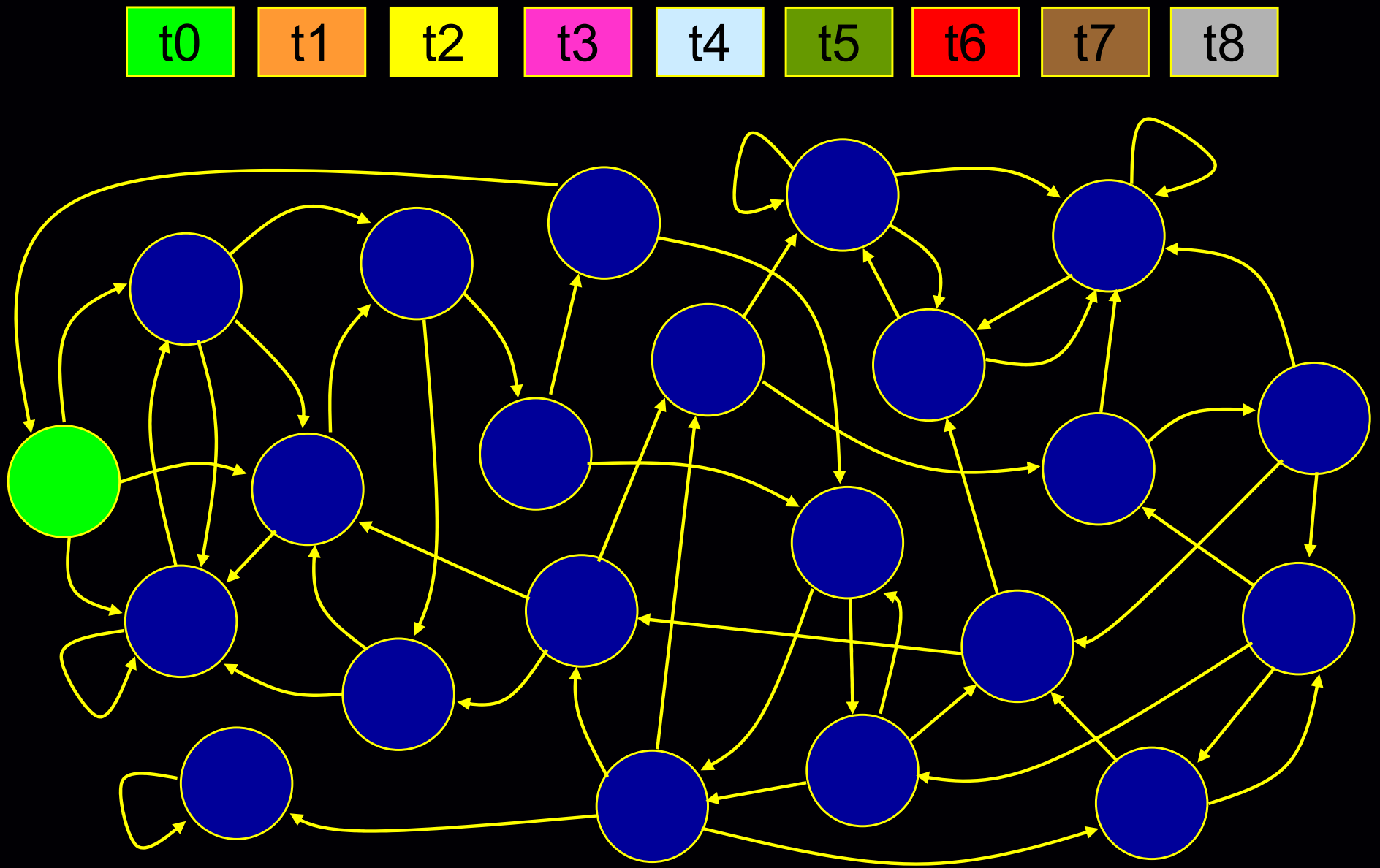
- Either
  “proving p is true for all states on the state transition graph” or
  “finding a trace that can disprove p”

"Mathematical Certainty" in Formal Verification

- Space exhaustiveness
  - Verify all input combinations of the system
- Time exhaustiveness
  - Verify system behavior from initial state to time infinity

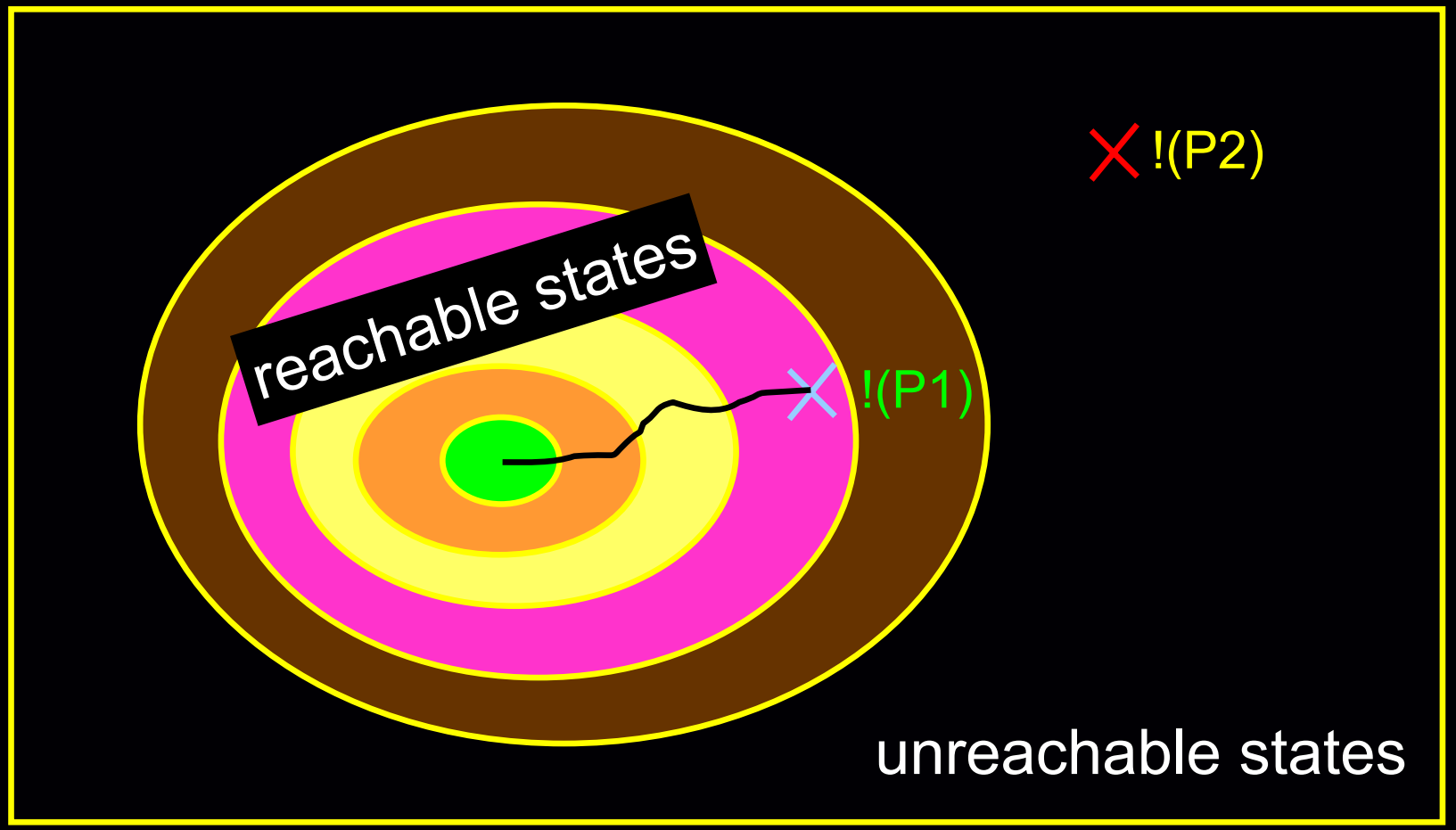


Set of Reachable States



Boolean State Space

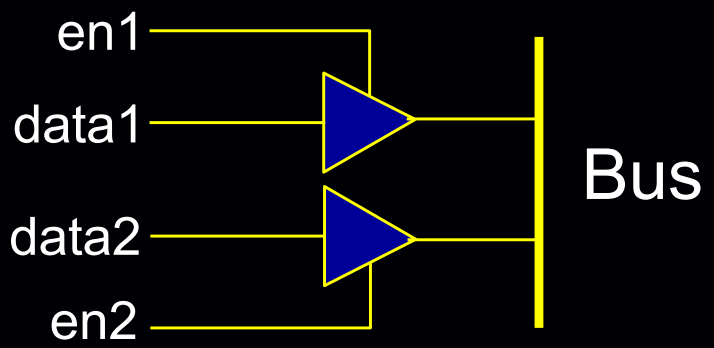
Boolean space of n state variables (2^n)



➔ AG (p1) ≡ false; AG (p2) ≡ true

Combinational Invariance

- It is worthwhile to know that some invariant assertions are actually combinational properties
  - ➔ Properties should hold "no matter what the state is" (reachable or unreachable)
  - e.g. Bus contention



- Need to assure bus won't have conflict signals even at random power-on or scan-in states

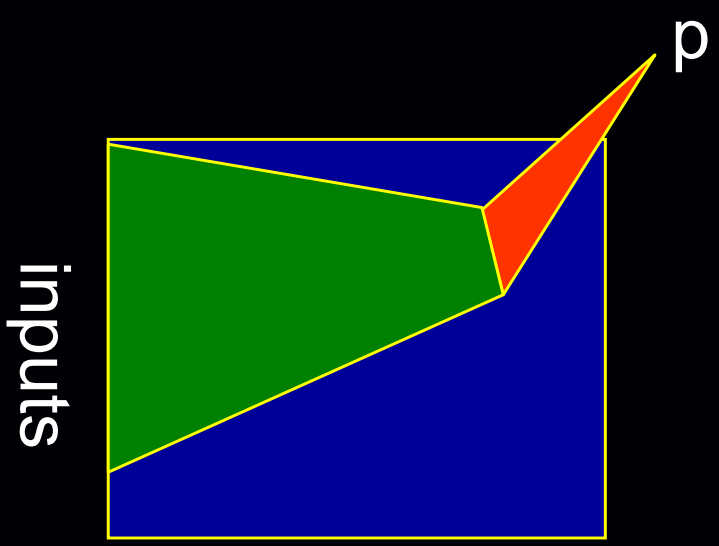
  - e.g. Logic equivalence checking

    → Most synthesis tools perform combinational optimization only

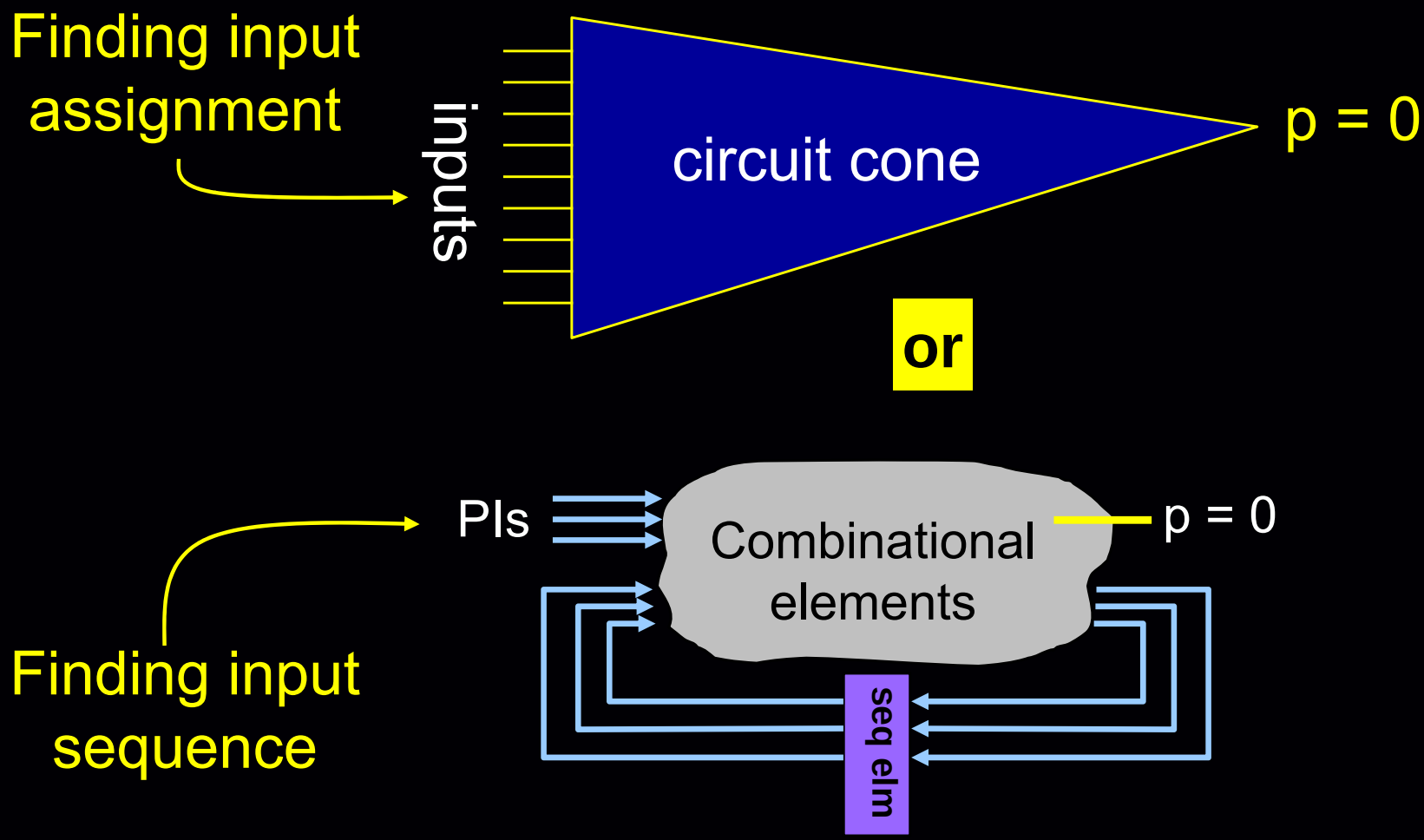
    → Just need to make sure combinationally equivalence

Proving Combinational Invariance

- No need to perform state reachability analysis
  - ➔ To prove a proposition p, just construct the fanin cone (functions) of p and prove that no input assignment can produce “p = 0”



In this topic, we will use SAT engine to prove the assertion property, or say, to prove that the counter-example does not exist!



Outline

Boolean Satisfiability (SAT)

Fundamental problem in computer science

- Given a Boolean network F: $B^n \rightarrow B$,
  where B = { 0, 1 }, and
  n is the number of inputs I = { x_1, x_2,... , x_n }

- Boolean Satisfiability
  - ➔ Finding an input assignment

    A: { $x_1 = a_1$, $x_2 = a_2$,... , $x_n = a_n \mid a_i \in B$ }

    such that F = 1.

- Exponential complexity...?

Boolean Satisfiability Solvers

- Boolean SAT solvers have been very successful recent years in the verification area
  - More popular than other techniques (e.g. BDDs)
  - Applications
    - Equivalence checking, property checking, etc
  - Applicable even for million-gate designs
  - For both combinational and sequential problems

- Most popular ones
  - miniSat, zChaff, BerkMin, Csat, Grasp, SATO,... etc.
  - http://www.satcompetition.org/

Complexity of SAT solver

- We have learned an efficient logic implication algorithm with watch scheme
- However, the complexity for the Satisfiability problem has been proven to be NP-complete (S. Cook, 1971)
  - Given n variables, the number of decisions can be as many as 2^n...

➔ How can SAT be useable for million-gate designs?

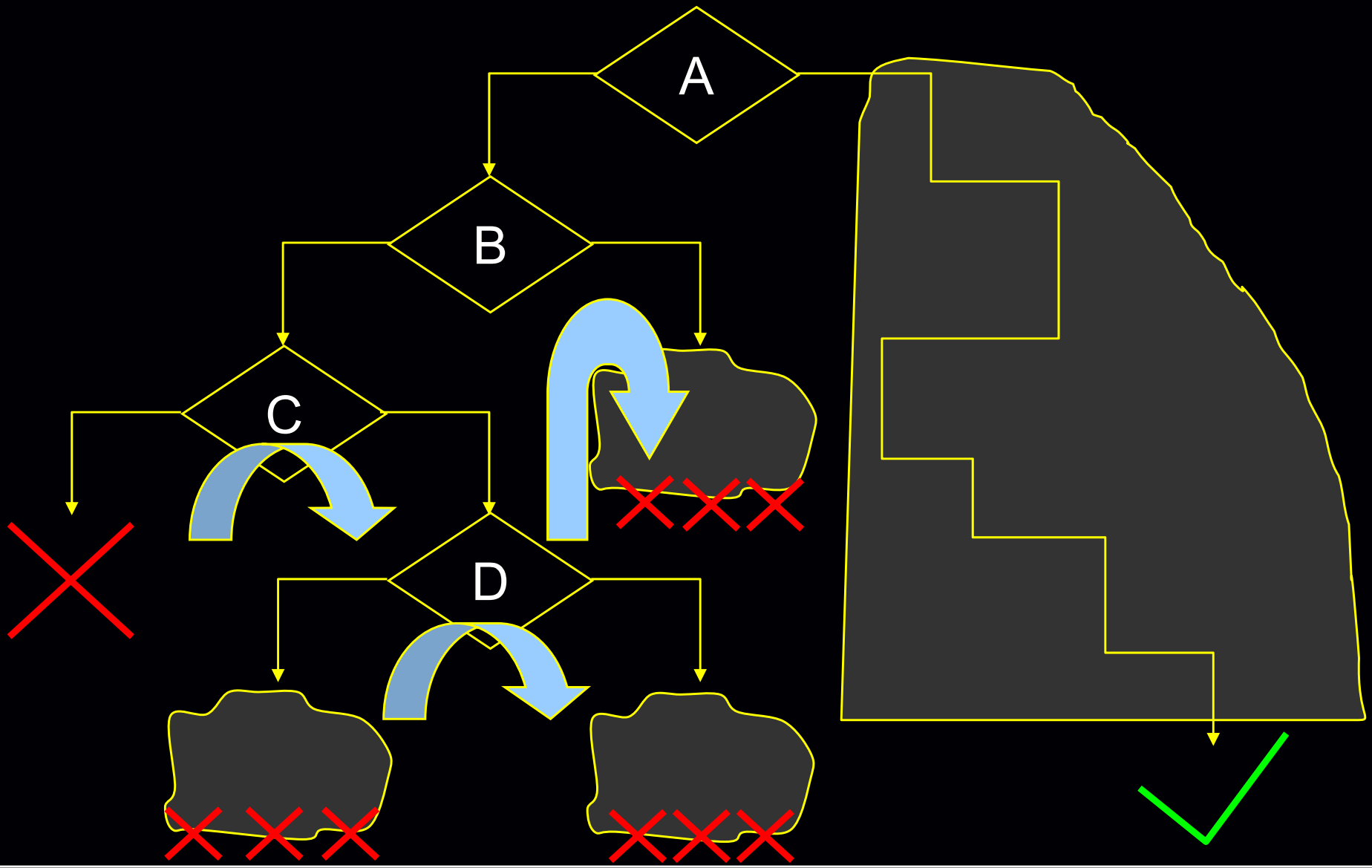
Think about the Human Search Process

(aka: search for $\neg$p)

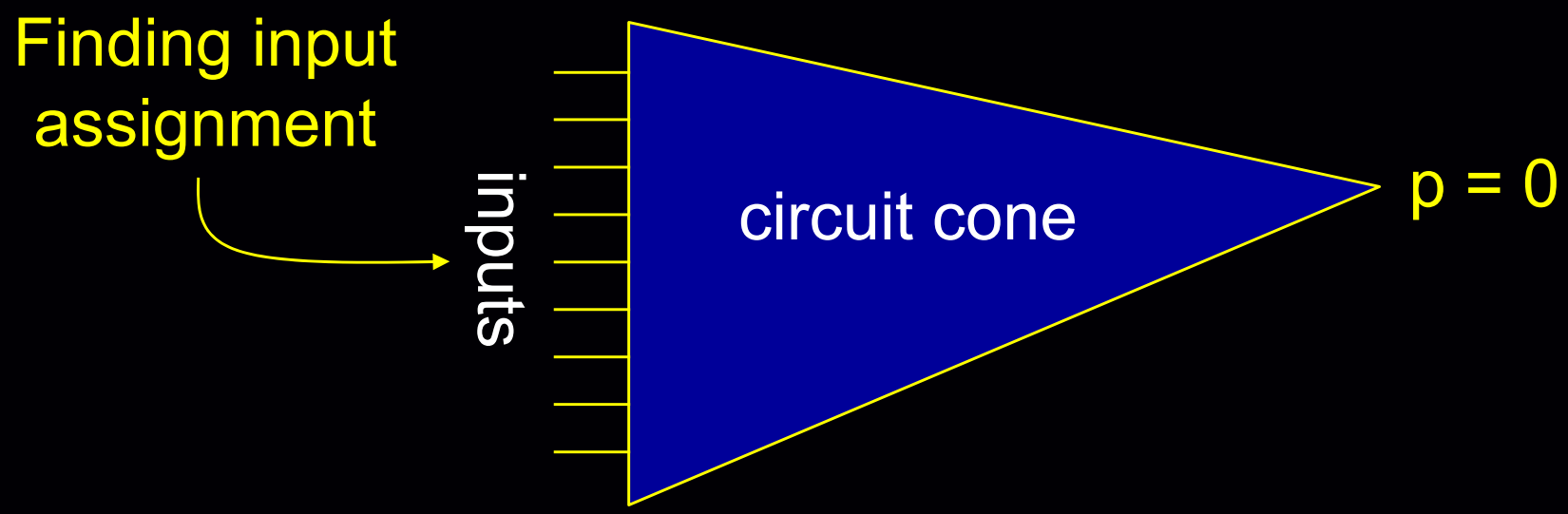
- When you search for something, you usually seek for one clue after another ---
  - Containing a keyword
  - Asking a question
  - By a direction, in a room, etc

- After some steps, if it is surely not there, you will reverse or revise some of the previous guesses/decisions and continue...

Something like --- A Decision Tree

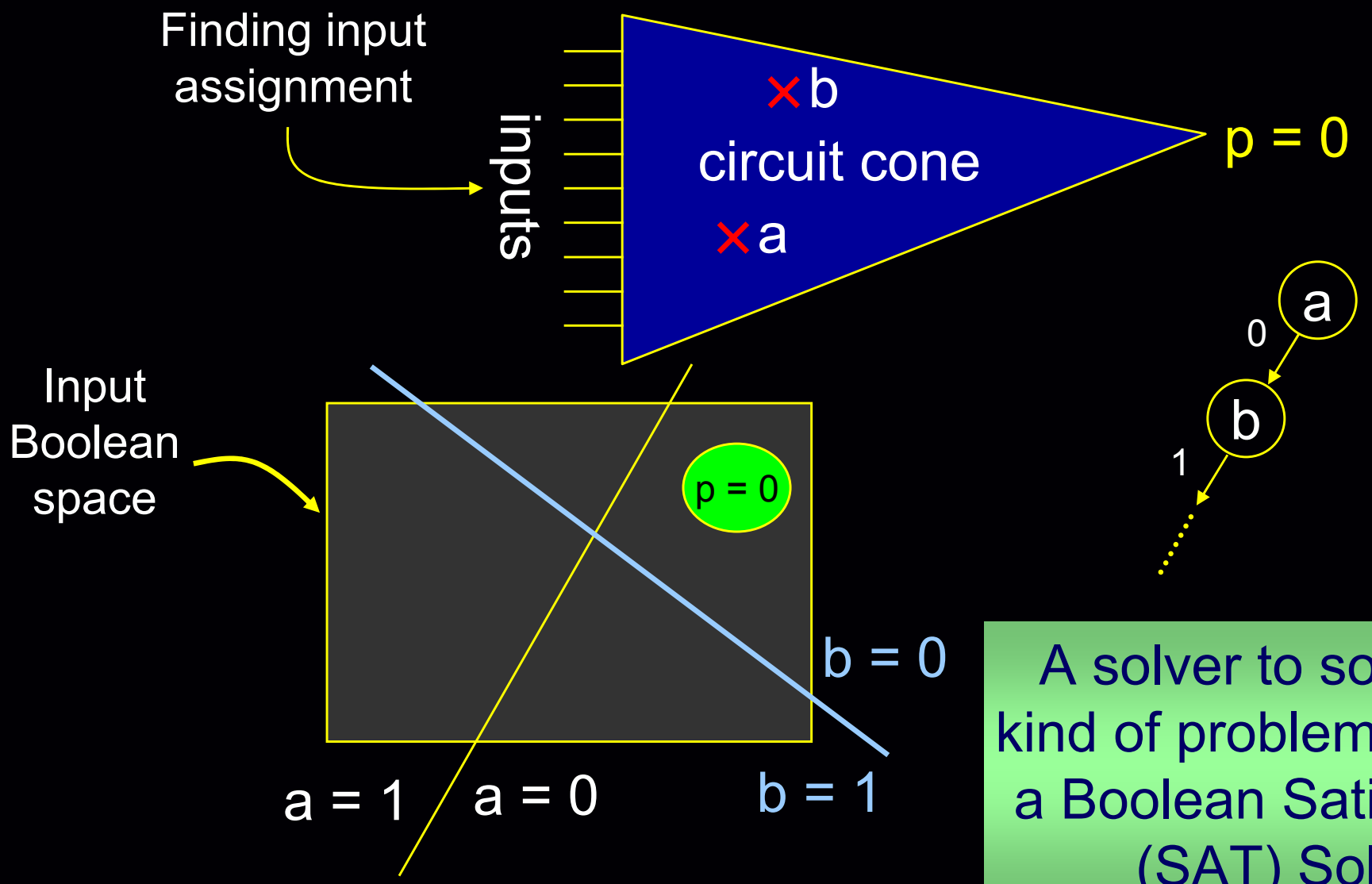


What about circuit properties?



- What are the “decisions”?
  - “Assigning input value one at a time” ?
    - ➔ Enumeration method: exponential complexity...
  - “Assigning values to internal signals” ?
    - ➔ Still exponential complexity...

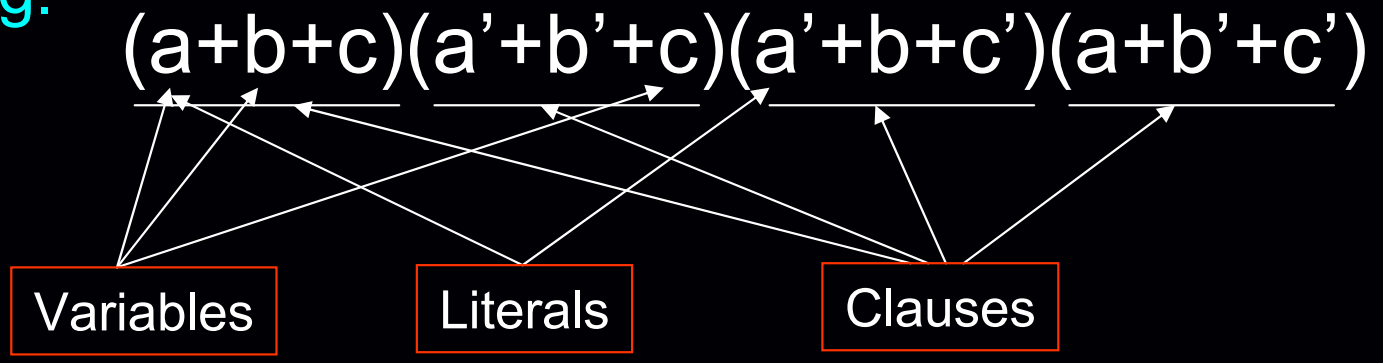
Boolean Satisfiability Checking for Properties on Circuit



A solver to solve this kind of problem is called a Boolean Satisfiability (SAT) Solver

Types of Boolean Satisfiability Solvers

1. Conjunctive Normal Form (CNF) Based
   - Boolean function is represented as a CNF (i.e. Product of Sum, POS format)
   - e.g.



   - To be satisfied, all the clauses should be '1'

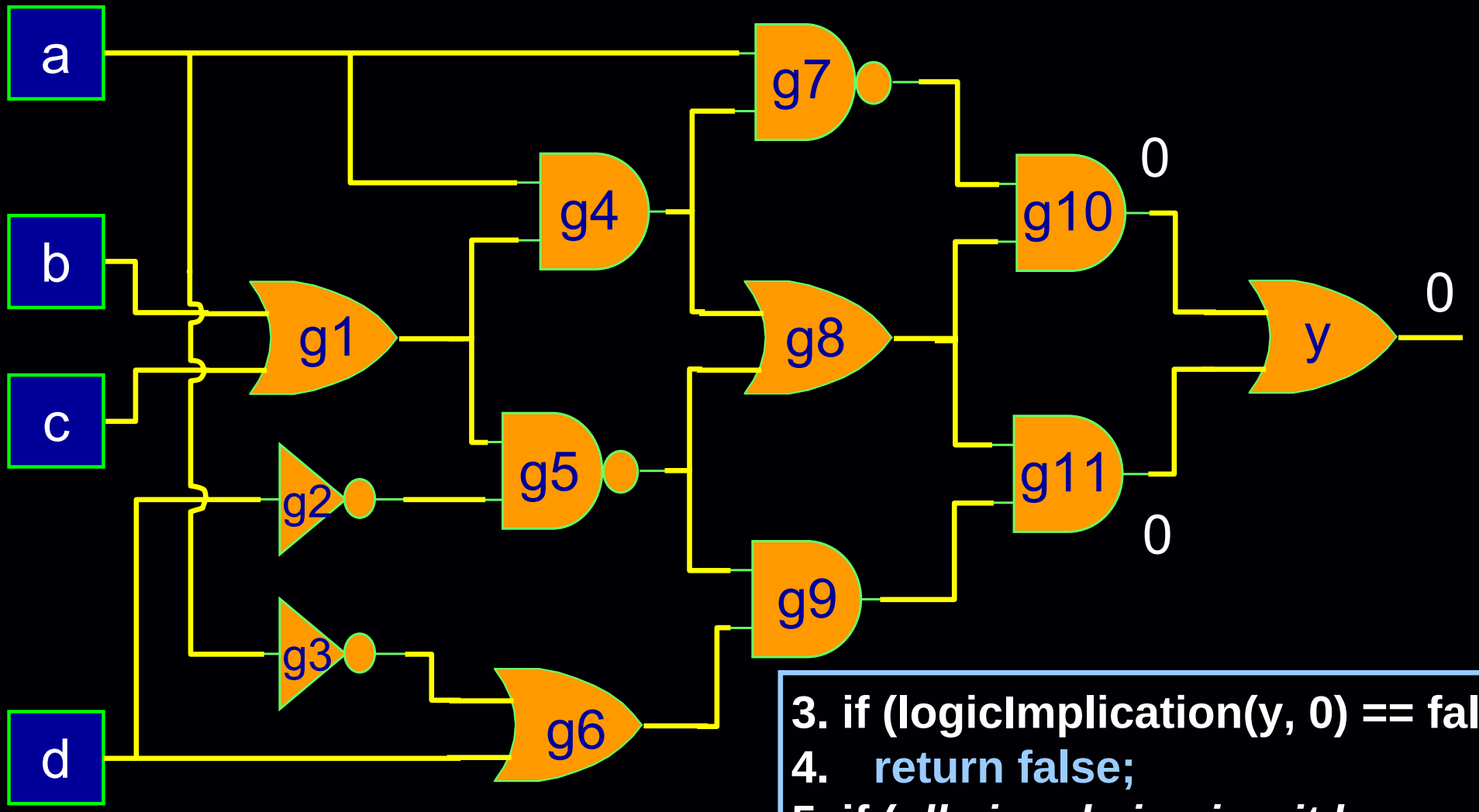
2. Circuit-Based
   - Boolean function is represented as a circuit netlist
   - SAT algorithm is directly operated on the netlist

A Typical Combinational SAT Algorithm

```
1. bool combSat(Gate g, value v)
2. {
3.    if (logicImplication(g, v) == false)
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (combSat(s, V0) == true)
9.       return true;
10.   backtrack(s);
11.   if (combSat(s, ~V0) == true)
12.      return true;
13.   backtrack(s);
14.   return false;
15.}
```

Proving always(y == 1)

combSat(y, 0)



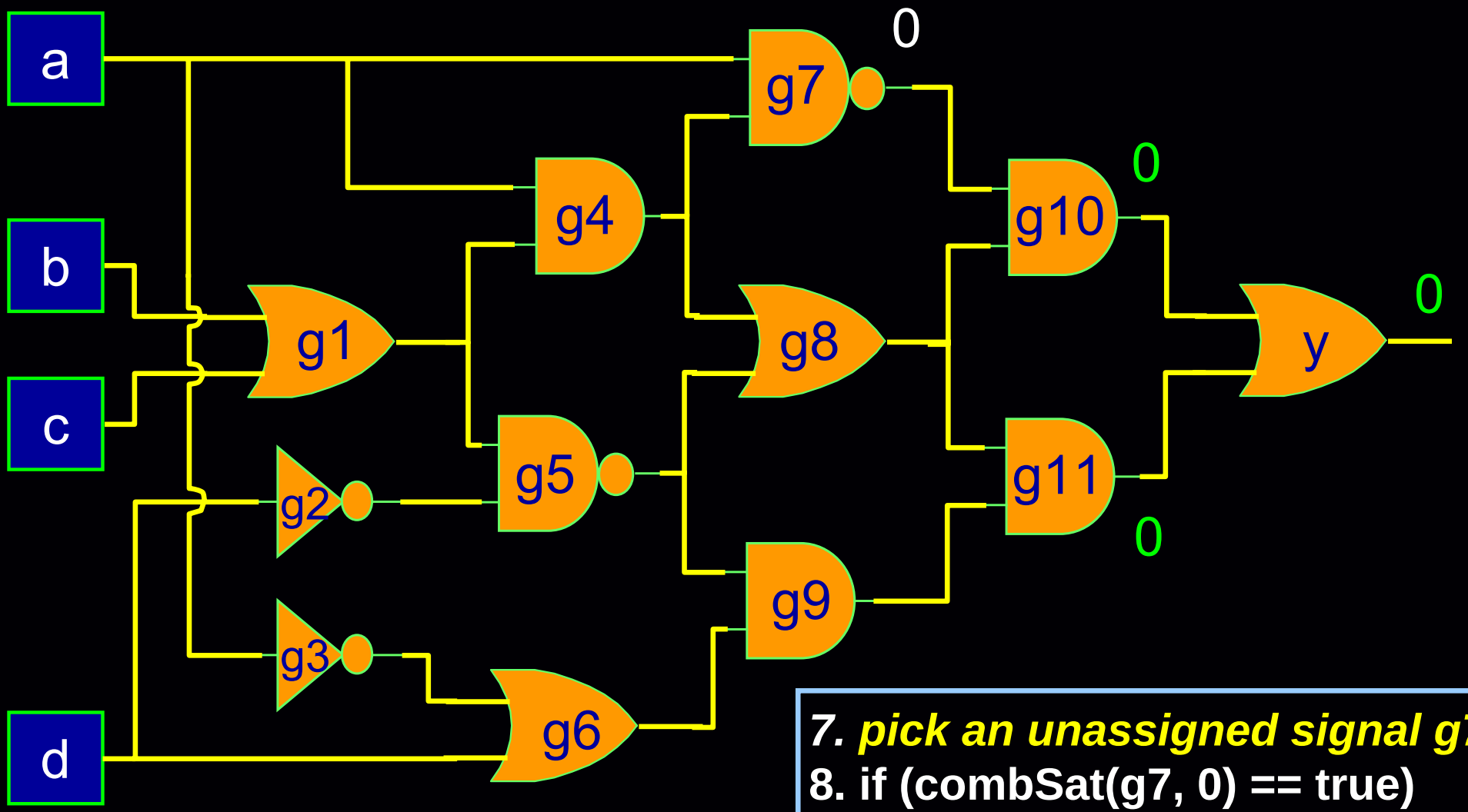

```
3. if (logicImplication(y, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

A Typical Combinational SAT Algorithm

```
1. bool combSat(Gate g, value v)
2. {
3.    if (logicImplication(g, v) == false)
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (combSat(s, V0) == true)
9.       return true;
10.   backtrack(s);
11.   if (combSat(s, ~V0) == true)
12.      return true;
13.   backtrack(s);
14.   return false;
15.}
```

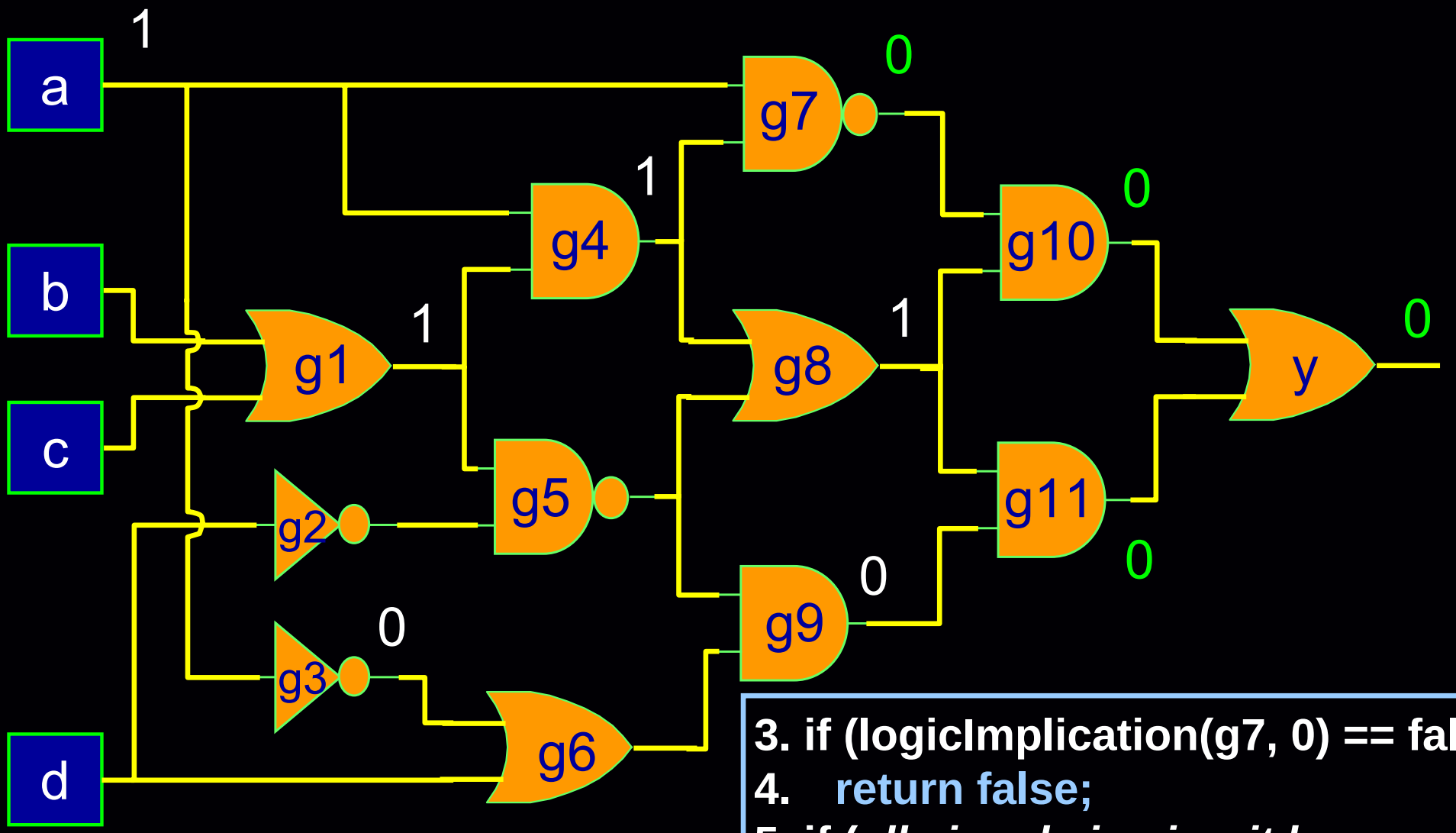
combSat (y, 0)

combSat (g7, 0)



7. *pick an unassigned signal g7*
8. if (combSat(g7, 0) == true)
9. return true;

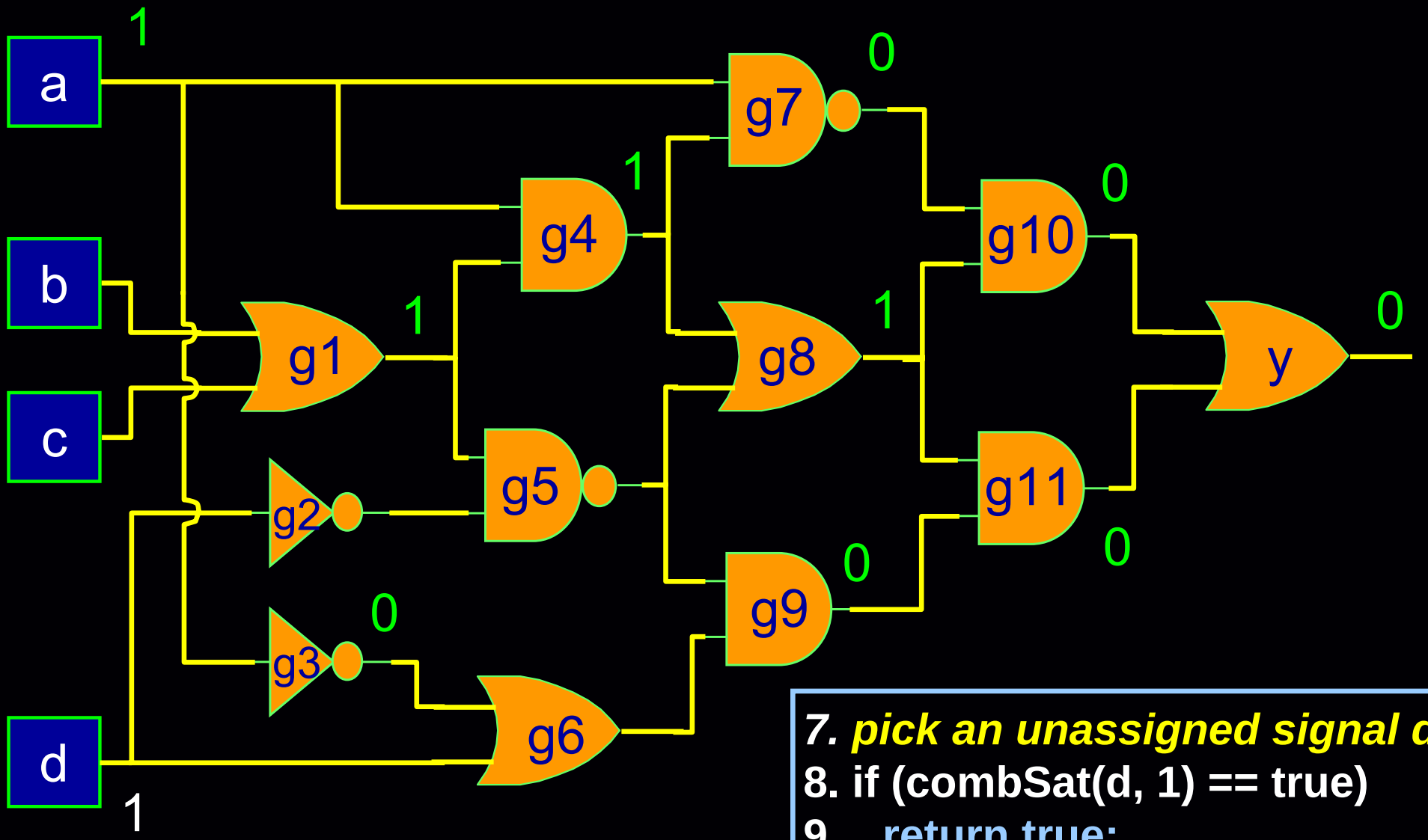
combSat (g7, 0)




```
3. if (logicImplication(g7, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

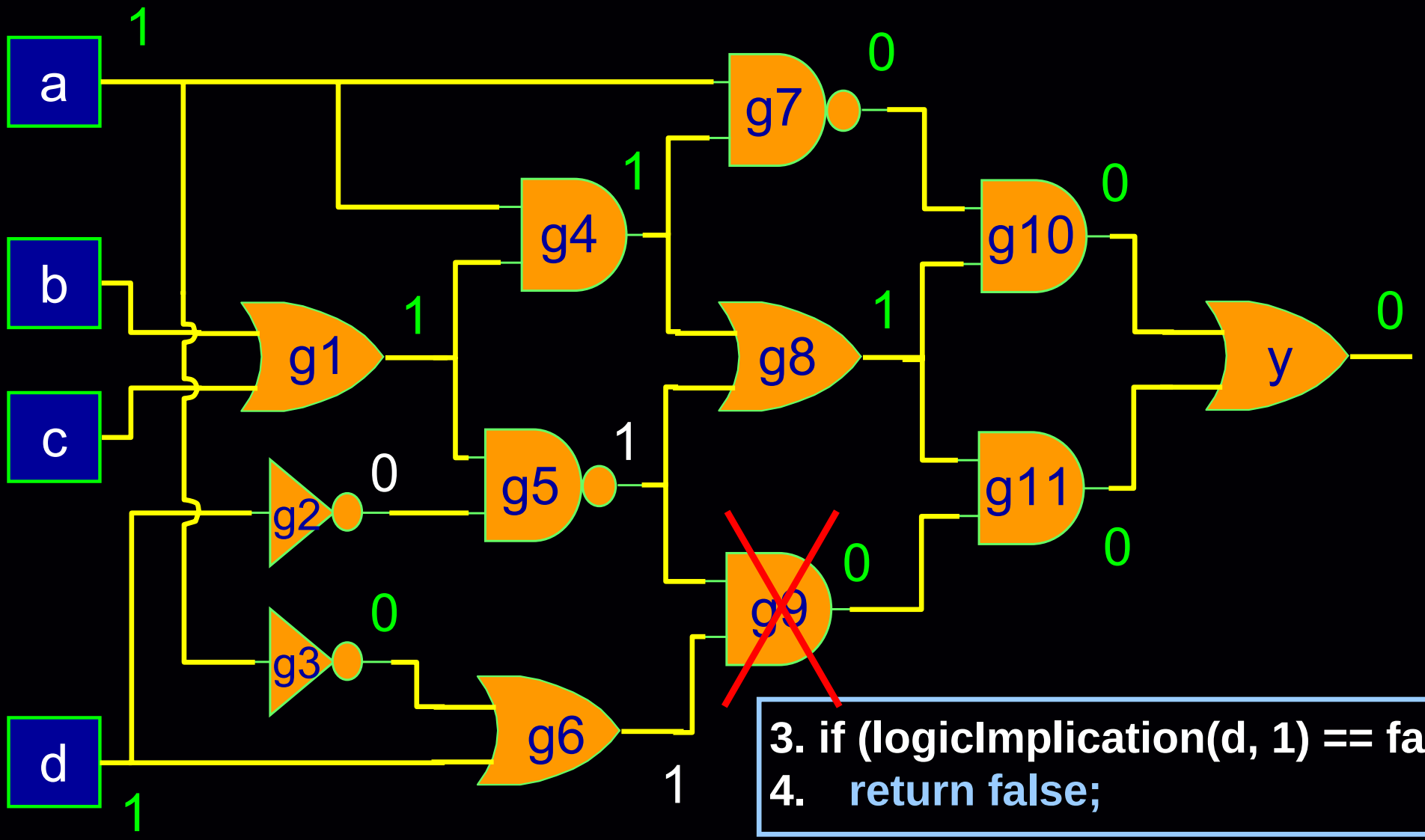
combSat (g7, 0)

combSat (d, 1)



7. *pick an unassigned signal d*
8. if (combSat(d, 1) == true)
9. return true;

combSat (d, 1)

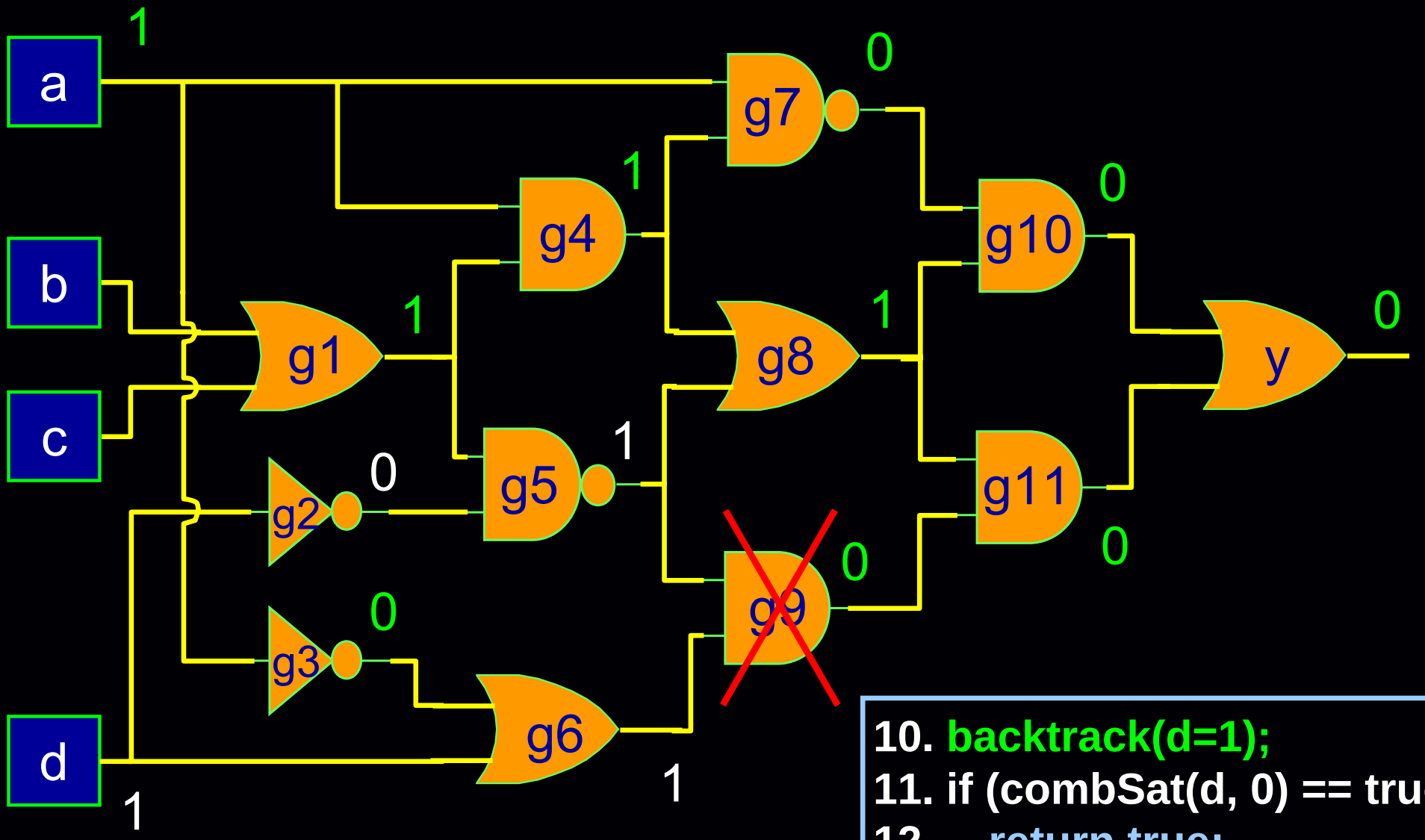



```
3. if (logicImplication(d, 1) == false)
4.    return false;
```

A Typical Combinational SAT Algorithm

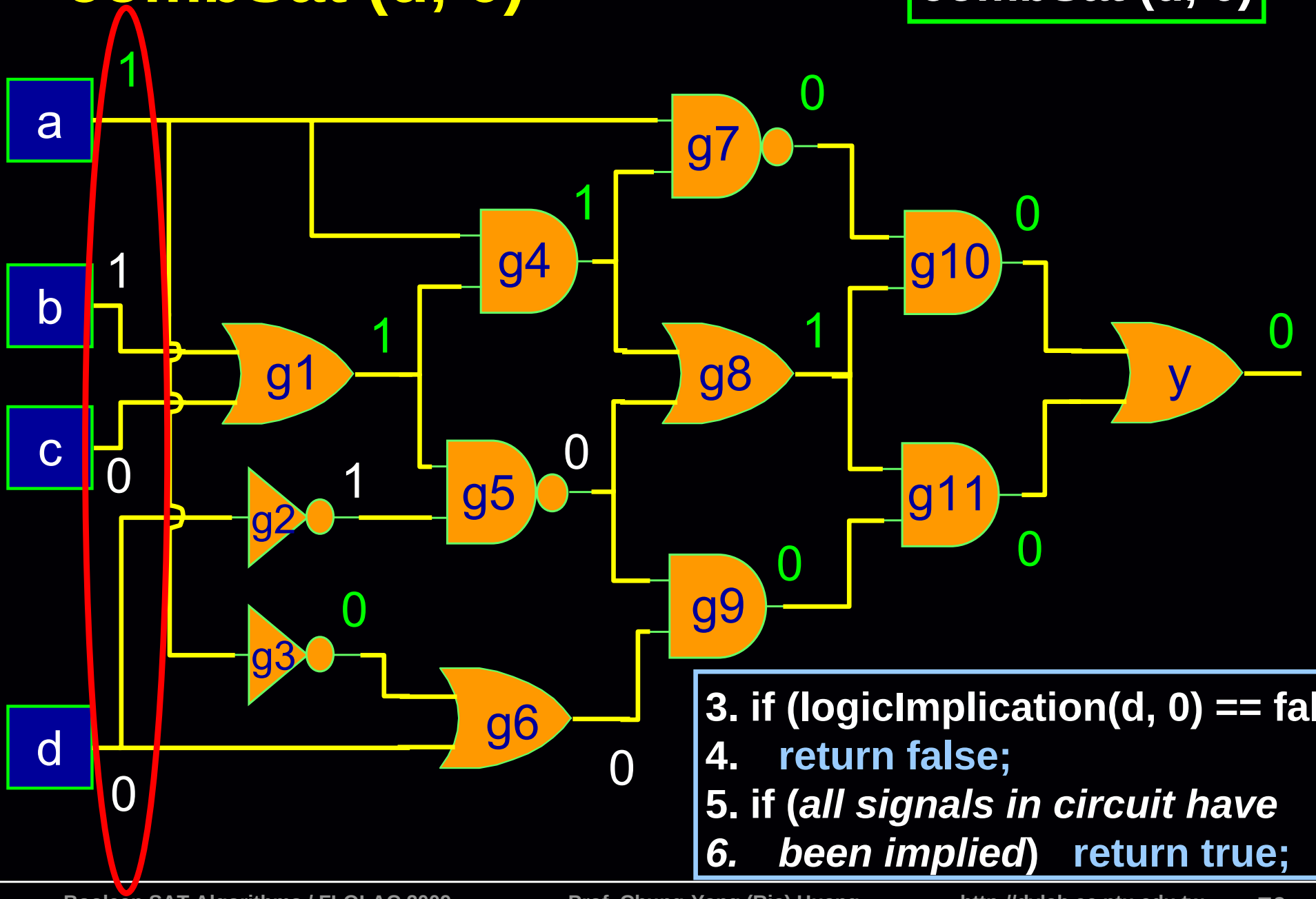
```
1. bool combSat(Gate g, value v)
2. {
3.    if (logicImplication(g, v) == false)
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (combSat(s, V0) == true)
9.       return true;
10.   backtrack(s);
11.   if (combSat(s, ~V0) == true)
12.      return true;
13.   backtrack(s);
14.   return false;
15.}
```

combSat (d, 1)




```
10. backtrack(d=1);
11. if (combSat(d, 0) == true)
12.     return true;
```

combSat (d, 0)



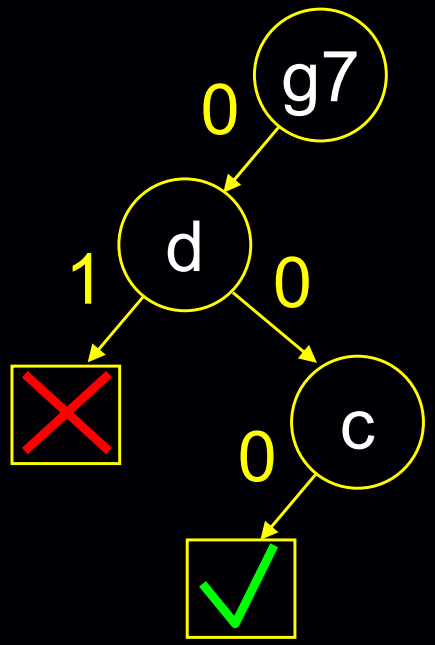
combSat (d, 0)

```
3. if (logicImplication(d, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

How many decisions did we make?

There are 12 gates and 4 inputs (2^4 decisions?)

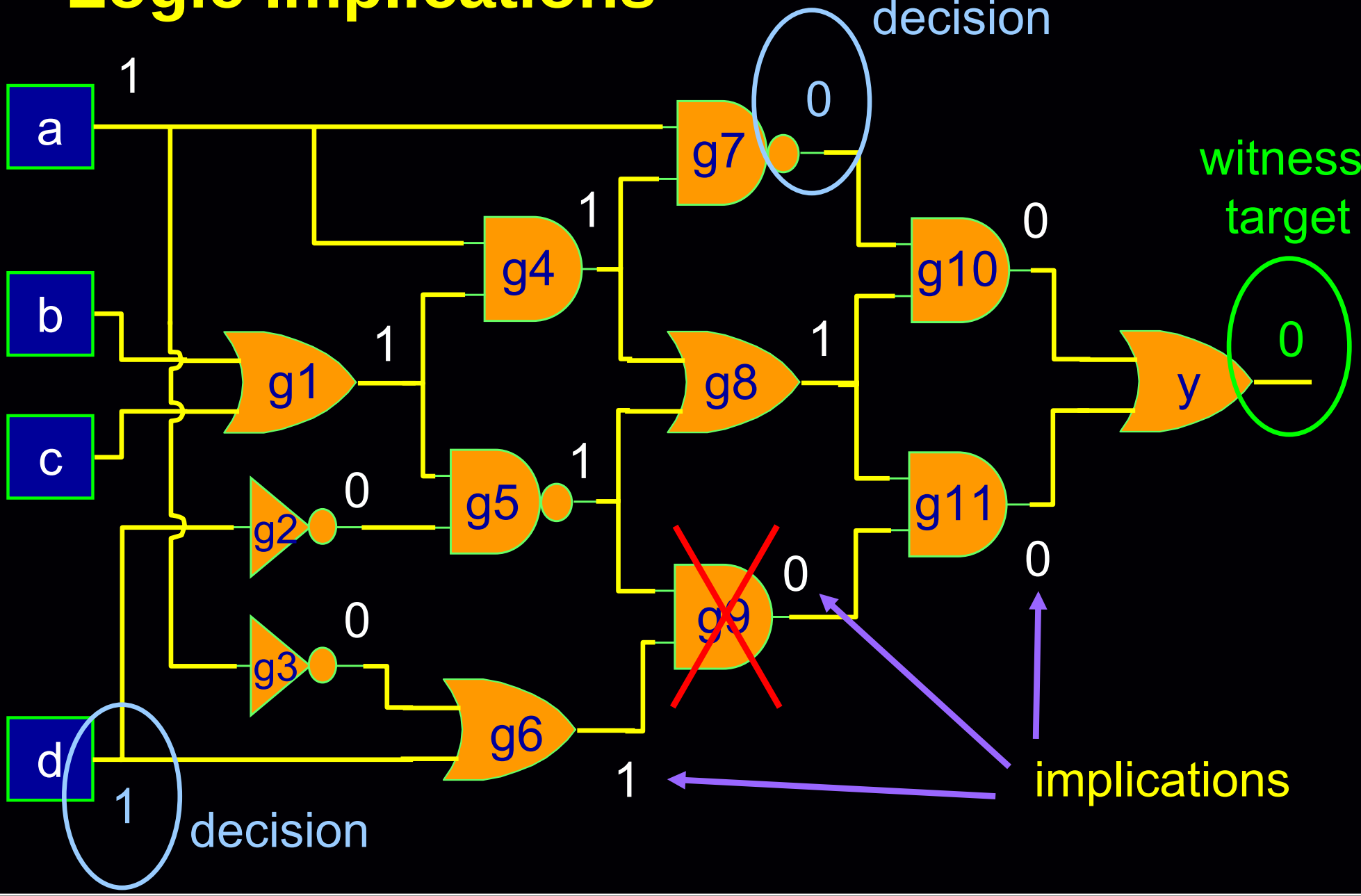
- The decision tree



← What make it so simple?

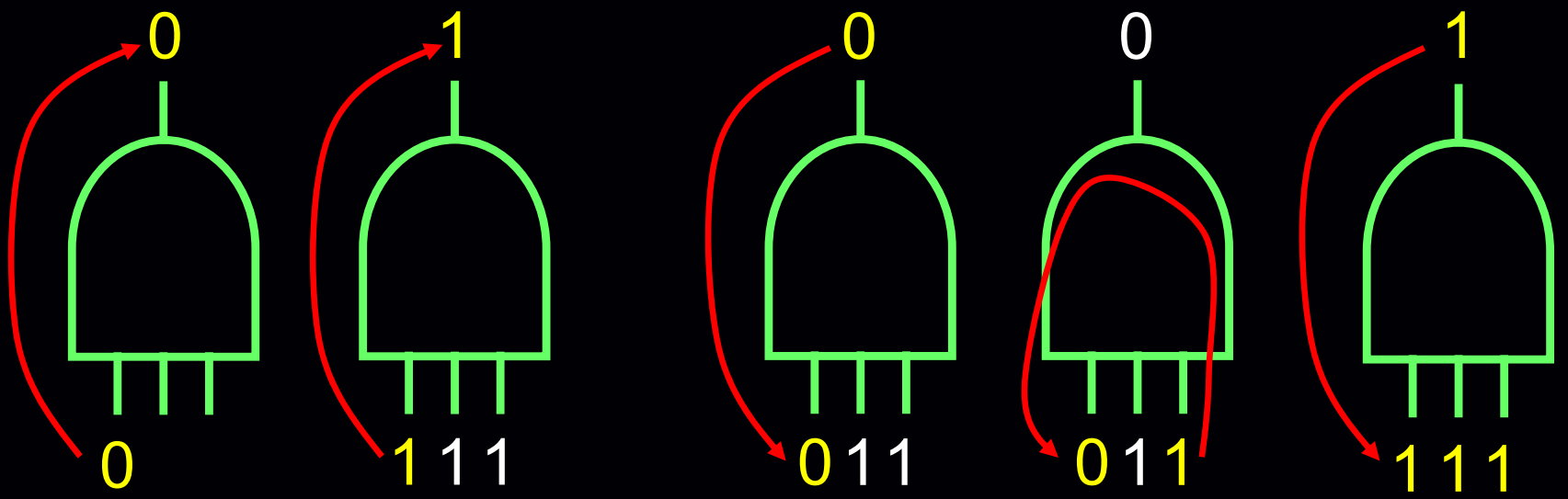
"Logic implications"

Logic Implications



Logic Implications

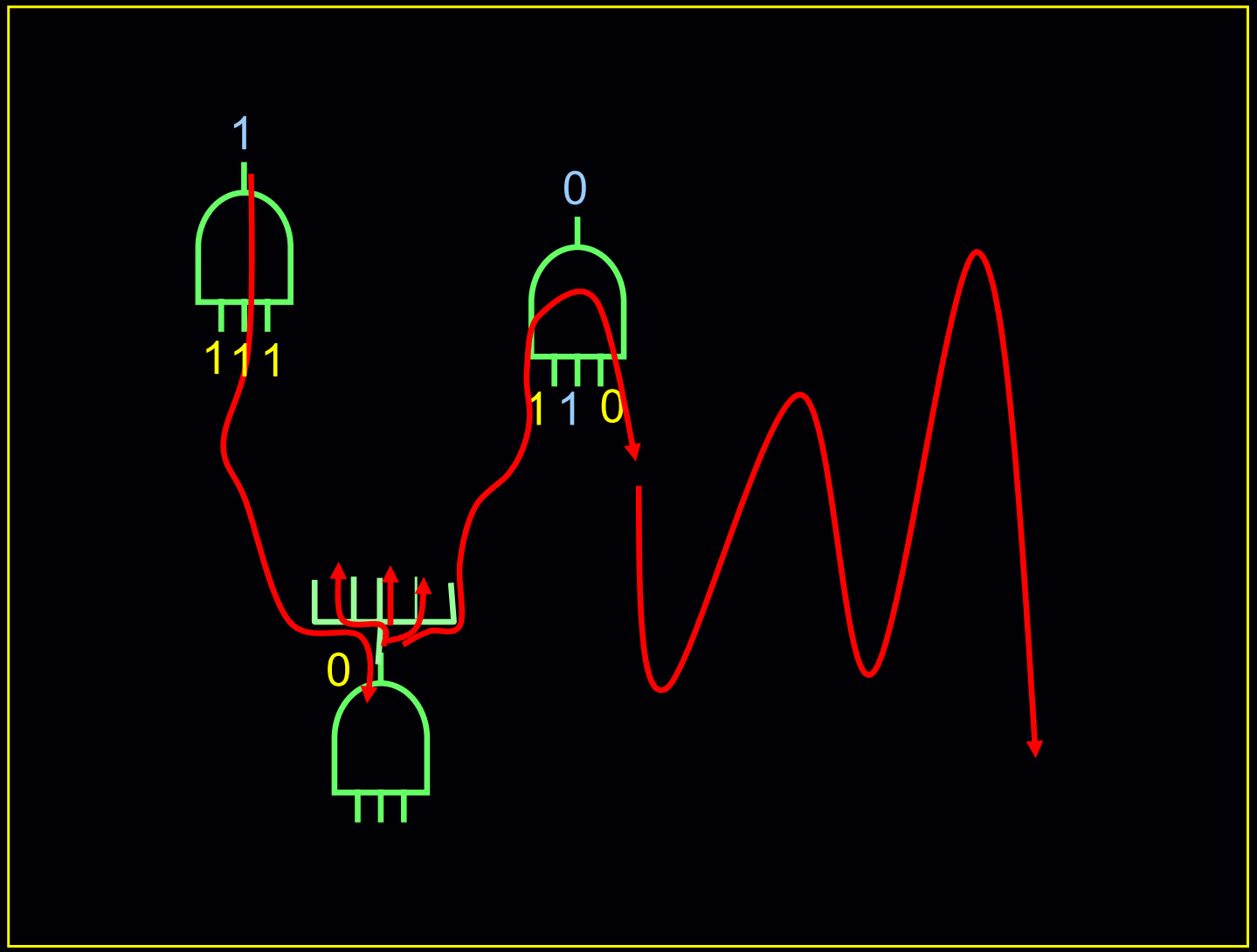
- Also called “Boolean Constraint Propagation” (BCP)
- Imply values to other gates in both forward and backward directions



Forward implications

Backward implications

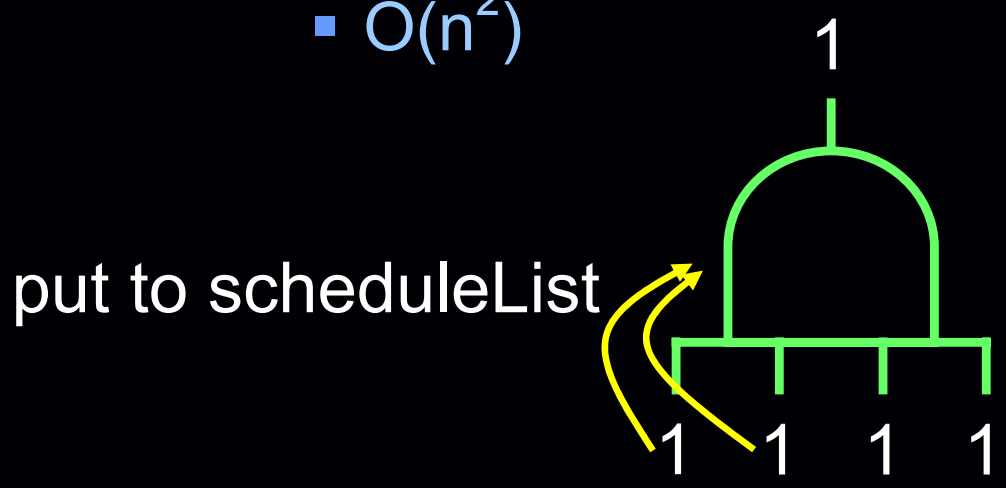
“Omni-directional” in a netlist



How to schedule and evaluate these implications??

Fruitless Implication Checking

- Checking if a gate can be implied, but usually no implication
- For example, the all-1's forward implication of an AND gate
  - Only the last '1' can trigger the forward implication
    - The first (n-1) checks are useless
  - Worse case: for n-input AND gate
    - Need to check (1 + 2 + …+ n) times of fanins
    - $O(n^2)$



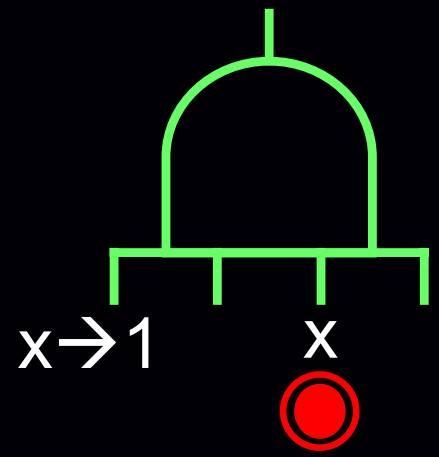
Trying to avoid fruitless implication checking

- Use a counter to record how many ‘x’ are in the fanins of a gate (e.g. AND gate)
  - Decrement by 1 when fanin is implied
  - Inccrement by 1 when fanin value is backtracked
  - When no ‘x’ fanin (x count = 0) ➔ forward implication

- Although this can avoid fruitless implication checking, but the overhead in maintaining the counts could be an overkill…
  - ‘x’ count could be: 5 → 4 → 3 → 4 → 5 → 4 → 5...

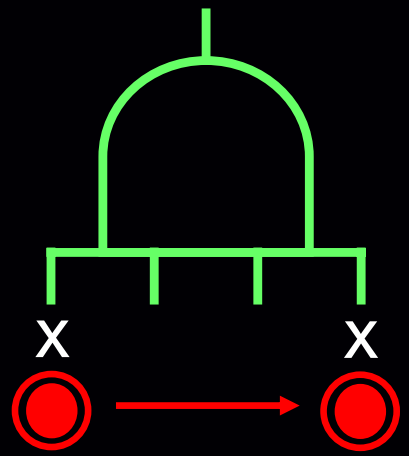
Watched-fanin concept

- In the n-input AND gate case, keep a pointer to one of its fanin that has value ‘x’ (watched fanin)
  - If other fanin gets implication ‘1’ ➔ no operation
  - If this watched fanin gets implication ‘1’, try to find another ‘x’ fanin to be new watched fanin
    - If found, update the pointer
    - If not found ➔ imply ‘1’ on the gate output



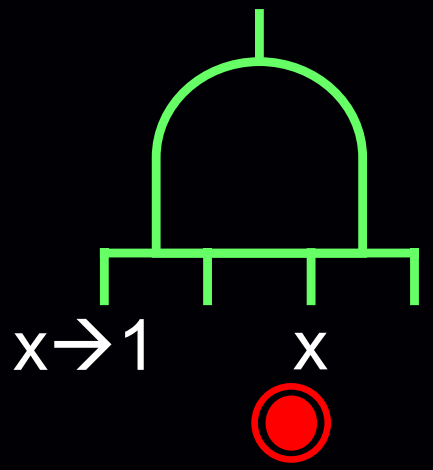
What's the improvement?

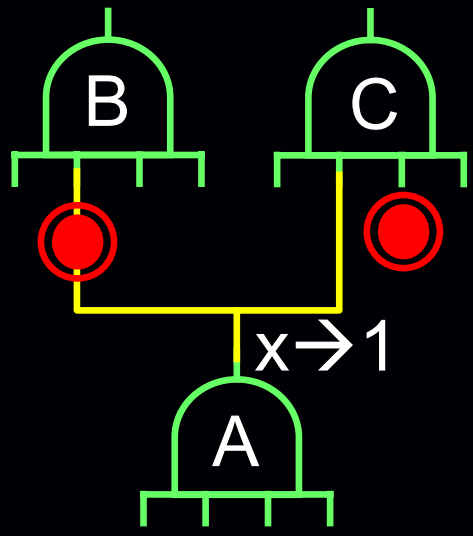
- Worse case $O(n^2)$?? No
- Suppose watched fanin points to the 1st fanin in the beginning
  - We always follow the same direction to find the next 'x' fanin
  - Complexity : O(n)



Watched fanin vs. Watching list

- Watched fanin
  - When one fanin gets an implication, check if this fanin is “watched fanin”
  - ➔ Still need to check for every fanin implication
- Watching list of watched fanin
  - The watched fanin (A) keeps the list of gates it is watching (i.e. { B })
    - When a watched fanin gets an implication
    - ➔ Update the watched fanins of the gates in the watching list (B); remove this watching list
    - ➔ Create the watching lists for the new watched fanins
  - Don’t need to evaluate a gate (e.g. C) if it is not in the watching list of any implied fanin (e.g. A → C)





Sounds good for all-1's forward implication of an AND gate, but what about 0 implication, backward implication, and other types of gates?

Different watched schemes?

(Could be very complex…)

That's why simpler data structure like CNF-based SAT engine can be more efficient sometimes

Logic Implication for CNF-Based SAT

- Refresh:
  - Boolean function is represented as a CNF (i.e. Product of Sum, POS format)
    - Can arbitrary Boolean function be converted into CNF?
  - e.g. (a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')
  - To be satisfied, all the clauses should be '1'

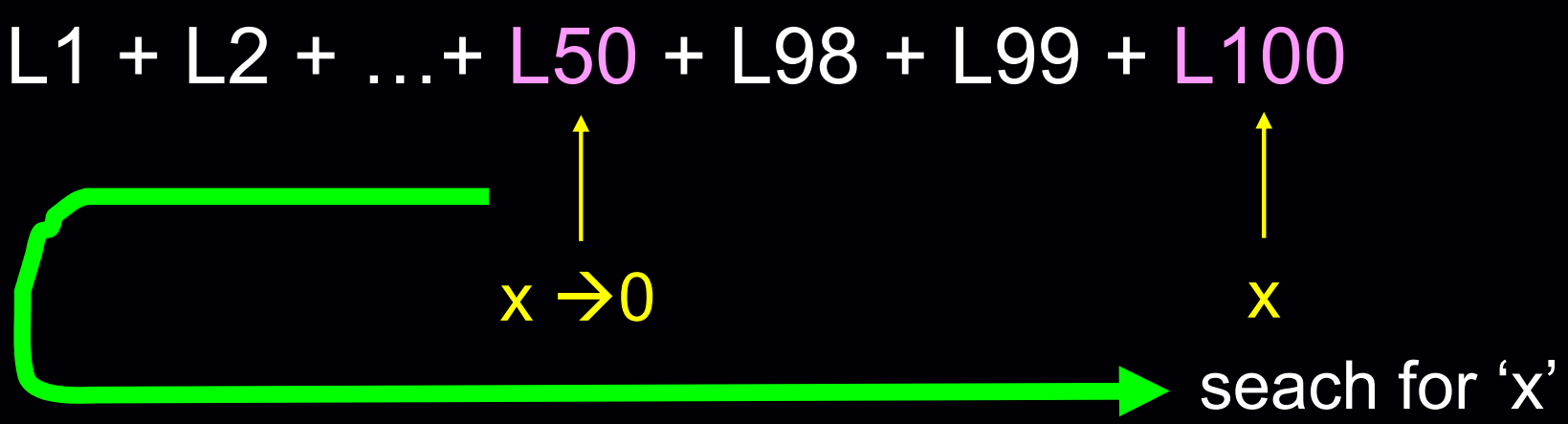
Logic implication for CNF-based SAT is simple ---

- If a literal in a clause gets an implication '1'
  - ➔ The clause is satisfied
- If a literal in a clause gets an implication '0'
  - ➔ Check: how many literals in the clause have unknown value?
  - >= 2 : no operation
  - = 1 : the remaining literal will be implied '1'
  - = 0 : the clause is evaluated to '0' ➔ a conflict !!

2-Watched-Literal Algorithm

H. Zhang, SATO, CADE 97; M. Moskewicz *et al*, Chaff, DAC 2001

- For each clause, keep 2 pointers on 2 literals that have “non-0” values
  - If any watched literal gets implication ‘0’
    - Scan in the clause for another literal with “non-0” value
    - If found, update the watched literal pointer
      Else, imply the other watched literal with value ‘1’



2-Watched-Literal Algorithm Example

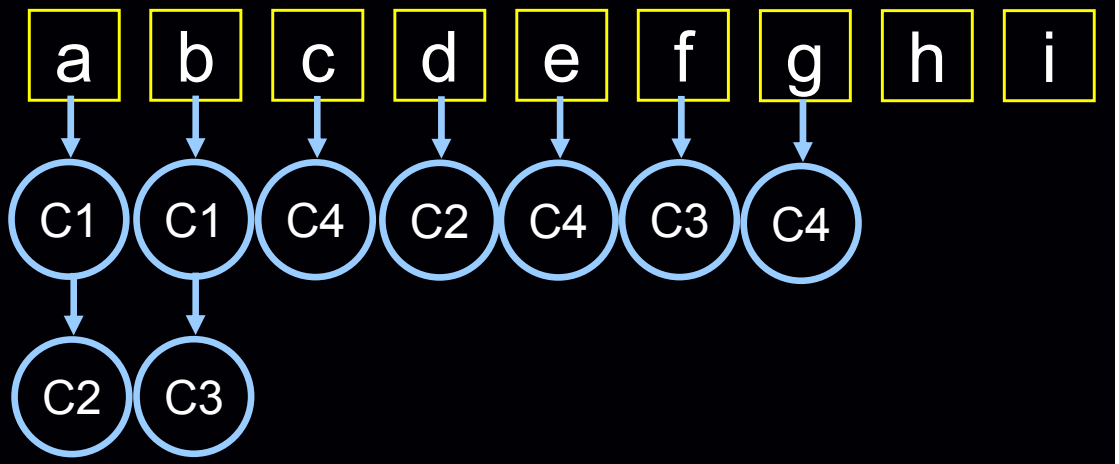
Each clause stores:

2 watched literal pointers

Each literal stores:

A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f)
C4: (c + e + g + h + i)



c ← 0

- Update watched literal pointer for C4 (for example, to ‘g’)
- Erase c’s watching-clause list
- Add ‘C4’ to g’s watching-clause list

[Note] Don’t need to check ‘C1’

2-Watched-Literal Algorithm Example

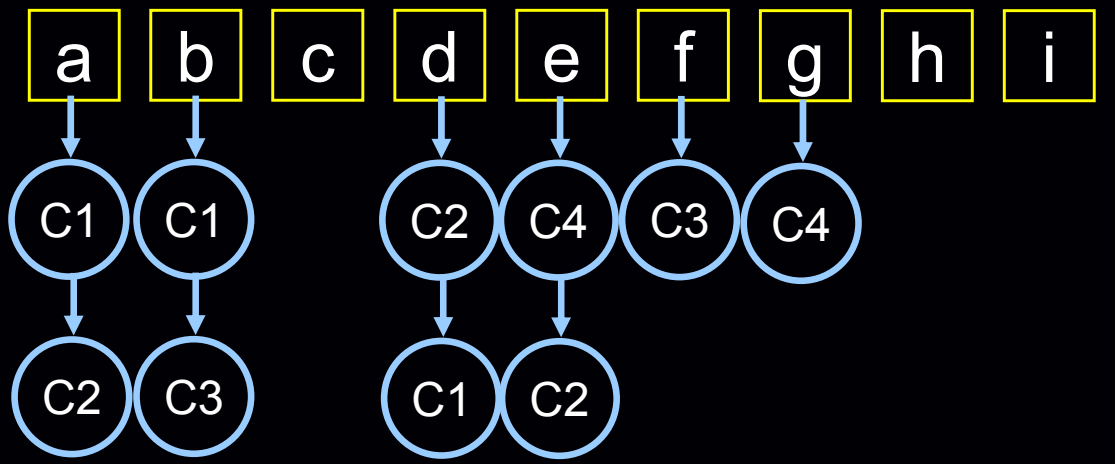
Each clause:

2 watched literal pointers

Each literal:

A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f)
C4: (c + e + g + h + i)



a ← 0

- Update watched literal pointer for C1 (only choice, to ‘d’)
- Update watched literal pointer for C2 (for example, to ‘e’)
- Erase a’s watching-clause list
- Add ‘C1’ to d’s and ‘C2’ to e’s watching-clause lists

2-Watched-Literal Algorithm Example

Each clause:

2 watched literal pointers

Each literal:

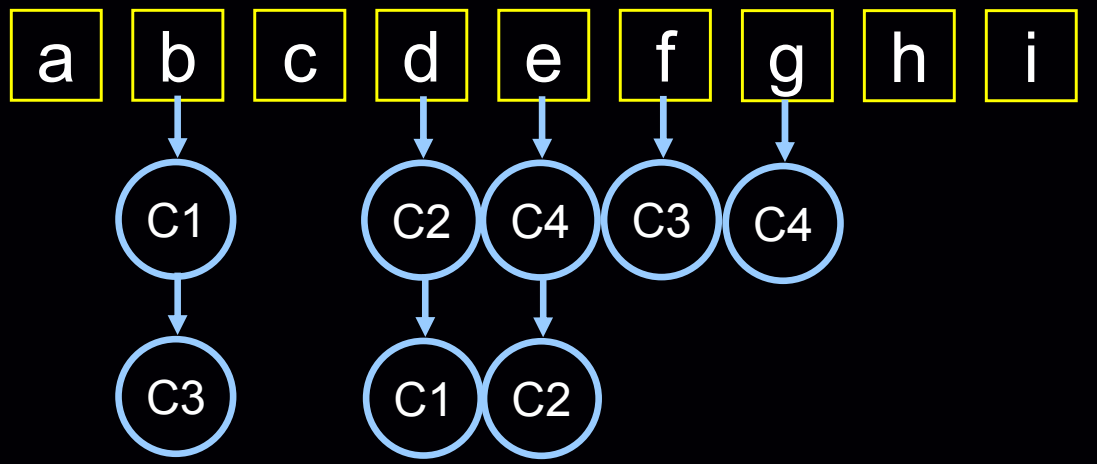
A list of watching clauses

C1: (a + b + c + d)

C2: (a + d + e + f + g)

C3: (b + f)

C4: (c + e + g + h + i)



b ← 0

- No more unknown literal for C1 : d = 1
- No more unknown literal for C3 : f = 1

[Note] No change on watched literals

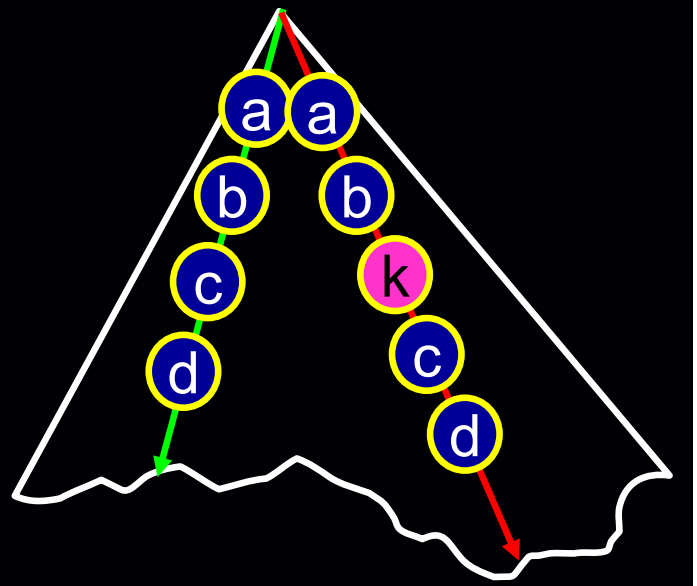
Caching Effect: Reducing from O(n) to almost O(C)

- The fact
  - Most of the time, the decision orderings at different parts of the decision tree are quite similar during a proof (or even from proof to proof)

  ➔ Literals in a clause get the implications almost by the same order every time
- Watched literal

  → point to the last implied literal

  ➔ Don't update watched literals after backtrack. After backtracks, no evaluations from the other unwatched literals.



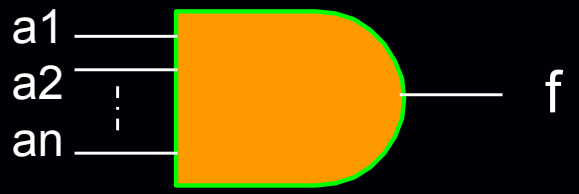
(L1 + L2 + L3 + L4 + L5 + L6)

Logic implication can be very efficient for CNF-based SAT by using “watch” scheme.

Can this idea be applied to circuit-based SAT?

Difference between circuit and CNF SAT

- Circuit-based SAT: gates



- CNF-based SAT: clauses, variables, literals
  - n+1 clauses
    - (a1 + f ')(a2 + f ')...(an + f ')
    - (a1' + a2' + ... + an' + f)
  - 1 variable (f)
  - 2 literals (f, f ')

Converting a Circuit to a CNF

Circuit_to_CNF(Circuit c)

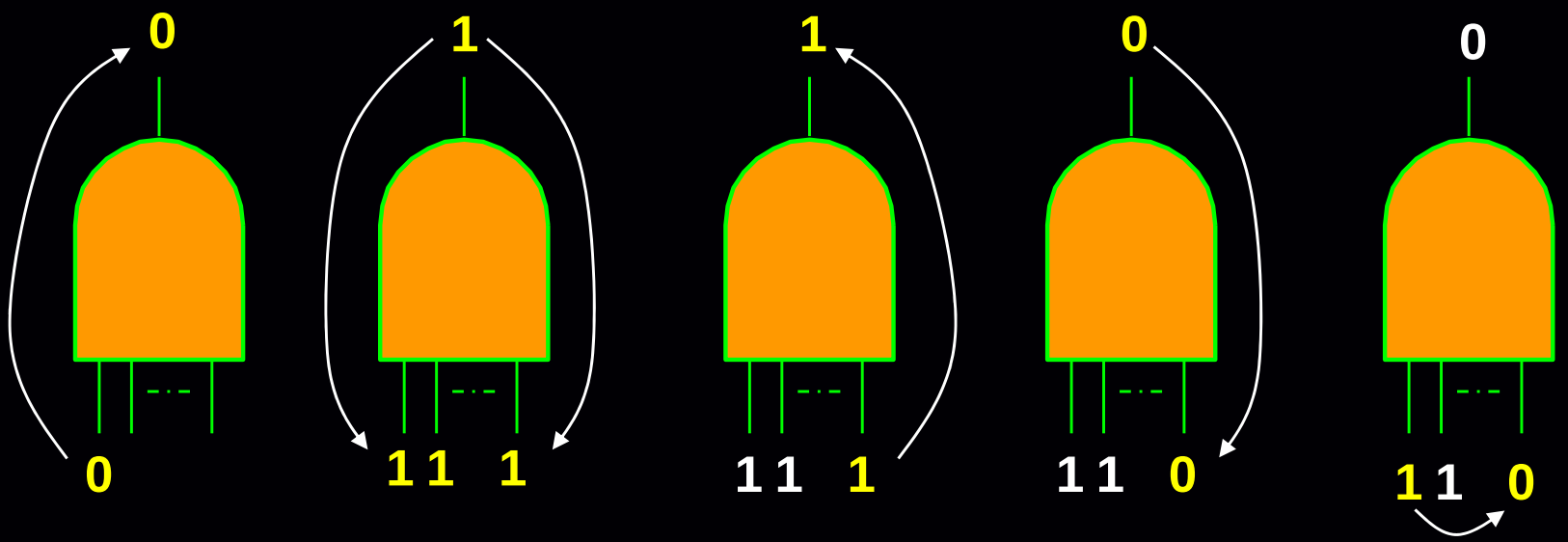
- Assign one variable to each gate in the circuit, including PIs
- For each gate in the circuit, generate the clauses for the characteristic function of this gate
- Conjunct all the clauses in step 2, return the CNF

Write_CNF(gate g)

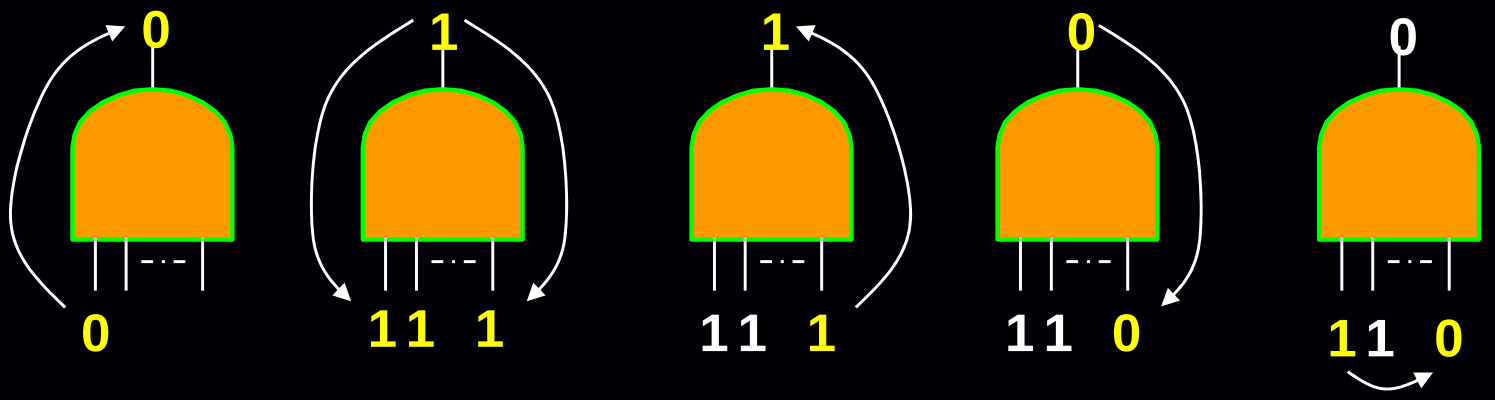
- Enumerate all the forward implications of this gate
- For each implication, generate a clause by the rule: $p \rightarrow q \equiv (p' + q)$

Any problem?

- Remember there is only one type of implication in CNF SAT
  - Logic implication can be very efficient
- But implications on an AND gate

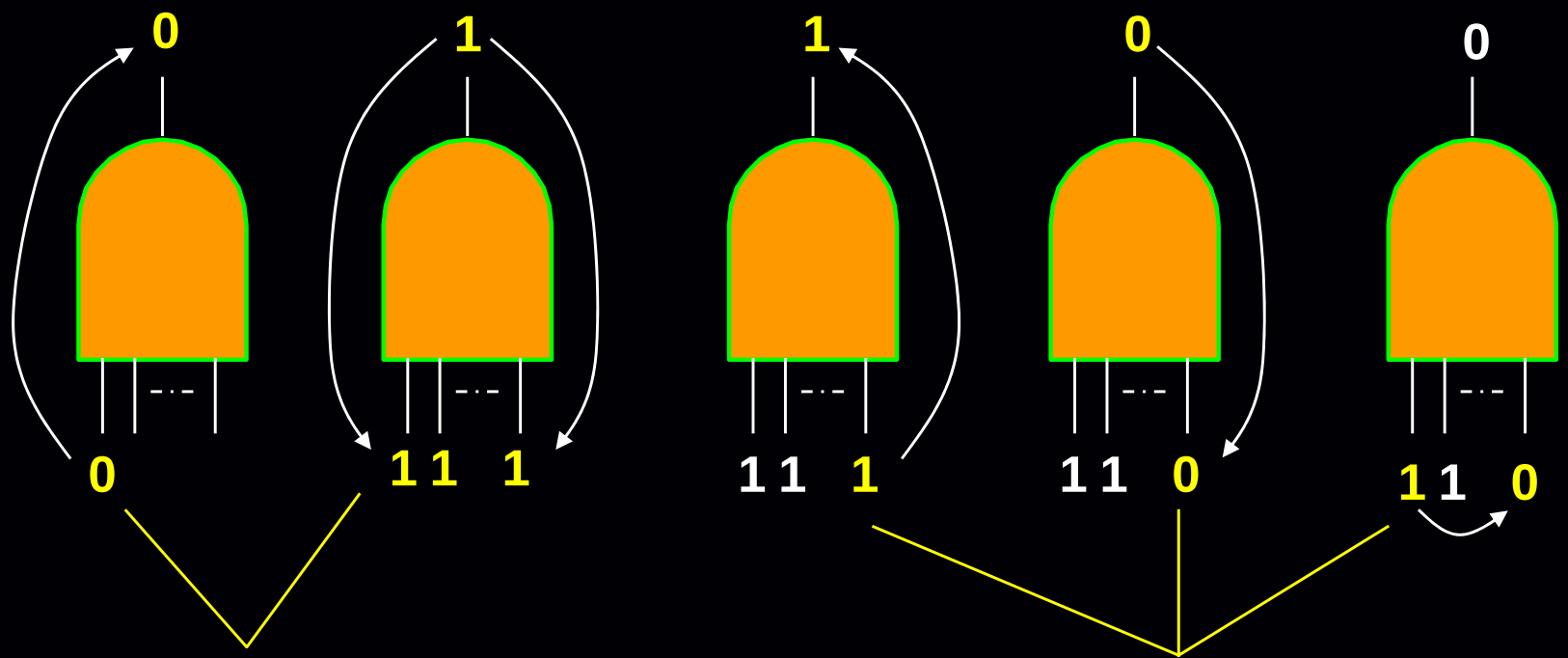


Can circuit SAT do 2-watch?



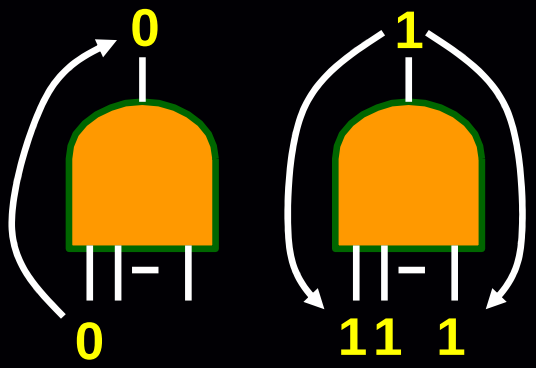
- Watch 2 fanins?
  - What's the watch value?
  - How about gate output?
  - How about OR, NOR, NAND,.. gates?
  - How about XOR, MUX, ... complex gates?

A Closer Look



Different implications on circuit-based SAT actually map to the same implication on CNF SAT

Direct vs. Indirect Implications



1. Direct implication
   - Corresponding n 2-literal clauses in CNF SAT
   - Single implication source

   ➔ No need to watch

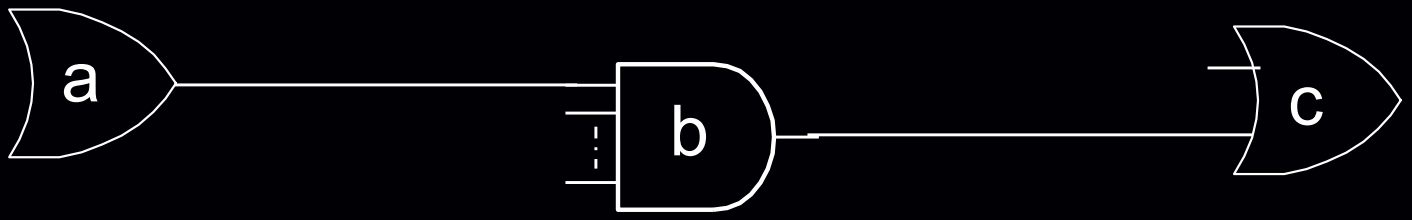
Direct Implication

1. Single source for each implication
2. Only depends on netlist structure; has nothing to do with the proving process (e.g. decisions, etc)
3. Should never encounter “CONFLICT” during the proof process (assume circuit is a DAG)

Direct Implication

➔ Construct a "*direct implication graph*" in the preprocessing step

➔ Apply direct implications whenever a gate is implied to a value

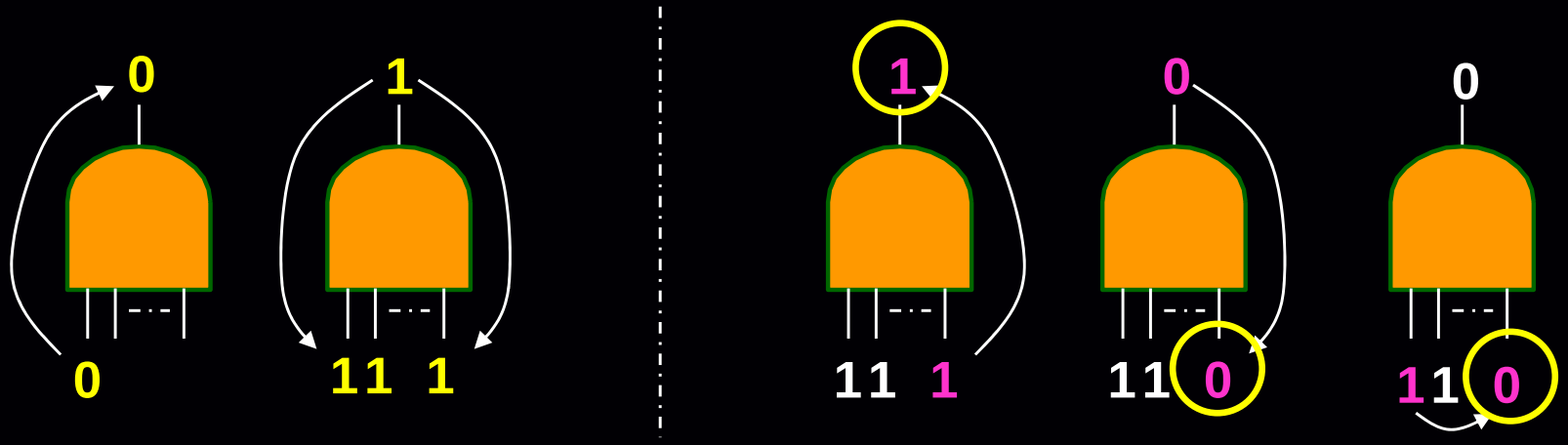


_0DirImps: { $\overline{b}$, ... }
_1DirImps: { ... }

_0DirImps: { ... }
_1DirImps: { a, c, ... }

_0DirImps: { $\overline{b}$, ... }
_1DirImps: { ... }

Direct vs. Indirect Implications



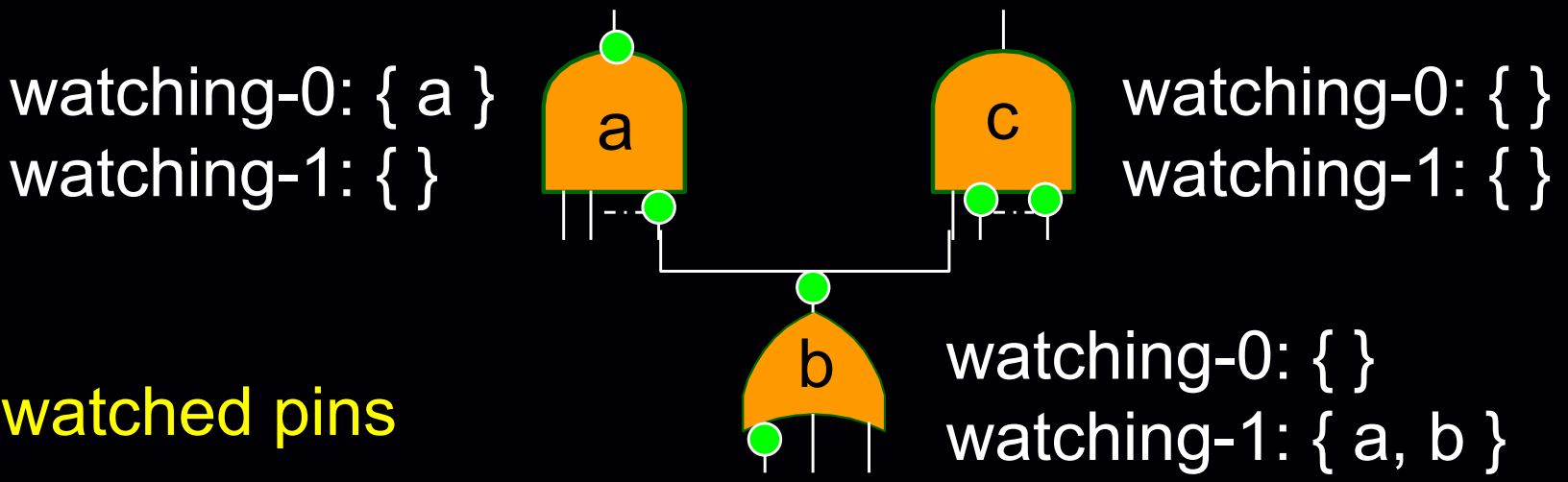
1. Direct implication
   - Corresponding n 2-literal clauses in CNF SAT
   - Single implication source

   ➔ No need to watch
2. Indirect implication
   - Corresponding to the same (n+1)-literal clause
   - Only the last implied pin has different value

   ➔ 2 watches: among all fanins and the gate itself

Indirect Implication (AND gate)

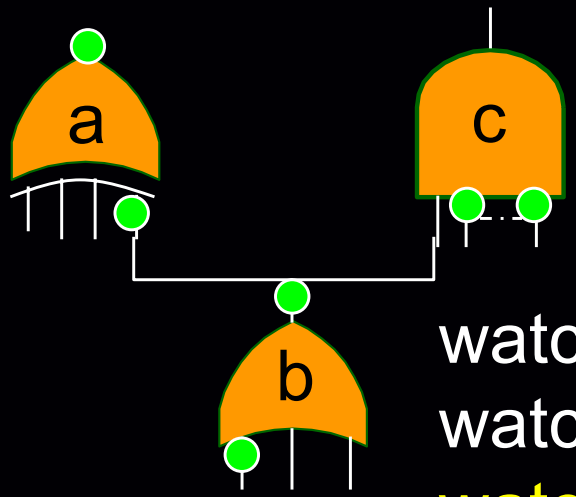
- Select 2 pins (fanins or the gate itself) in a gate to watch
  - Almost the same as CNF SAT, except that the "watched value" depends on the gate type and I/O
  - For each gate, two lists of watching gates
    - When a gate gets a value, perform direct implication and/or update watch for the gates on the watching list



Watch Scheme for XOR Gate

- n-input XOR gate
  - 2^n (n+1)-literal clauses
  - e.g. $(a + b + \overline{f})\ (\overline{a} + b + f)\ (a + \overline{b} + f)\ (\overline{a} + \overline{b} + \overline{f})$
- Implication occurs only when n variables become "known"
  - ➔ 2-watch; watch-known

watching-0: { }
watching-1: { }
watching-known: { a }



watching-0: { }
watching-1: { }
watching-known: { }

watching-0: { }
watching-1: { b }
watching-known: { a }

Generic Watch Scheme

- It can be shown that the watch scheme can be extended to complex gates such as MUXes, Pseudo Boolean gates, etc.
- For more details, please refer to:
  - "QuteSAT: A Robust Circuit-based SAT Solver for Complex Circuit Structure", DATE 2007.

Applications of Logic Implication

- We have learned that logic implication can be very efficient for both CNF and circuit-based SAT solvers
- Logic implication is actually also a powerful approach in exploring signal correlations in the circuit
- Any application?

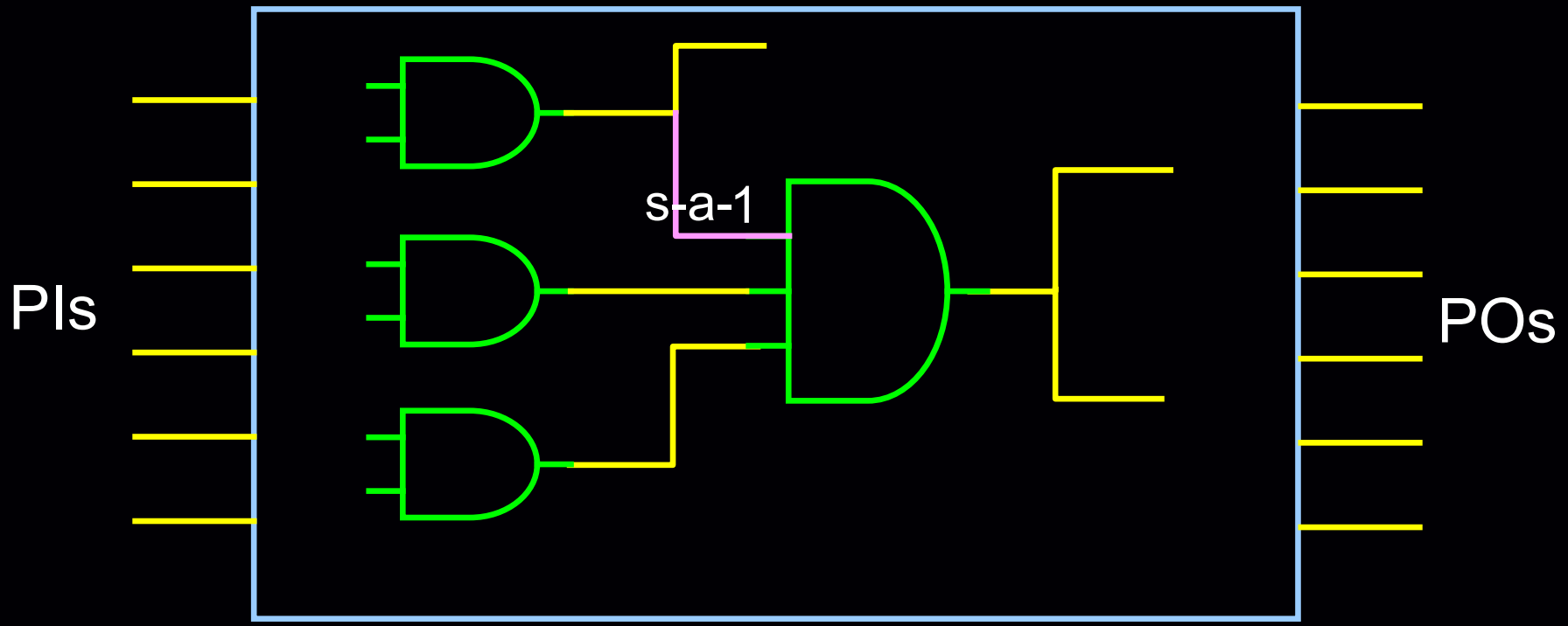
  Redundancy addition and removal

Redundancy Addition and Removal (RAR)

- Redundancy to a circuit
  - When removing or adding some signal/gate to a circuit, the circuit functionality remains unchanged
- Motivations
  - Removing redundancy in a circuit can gradually lead to small area, timing, power, etc
  - When (*deliberately*) adding some redundancy to a circuit, we may cause ***other*** part of the circuit become redundant
    - Incremental circuit restructuring (rewiring)
    - Can be used for incremental optimization (e.g. timing, area, etc)

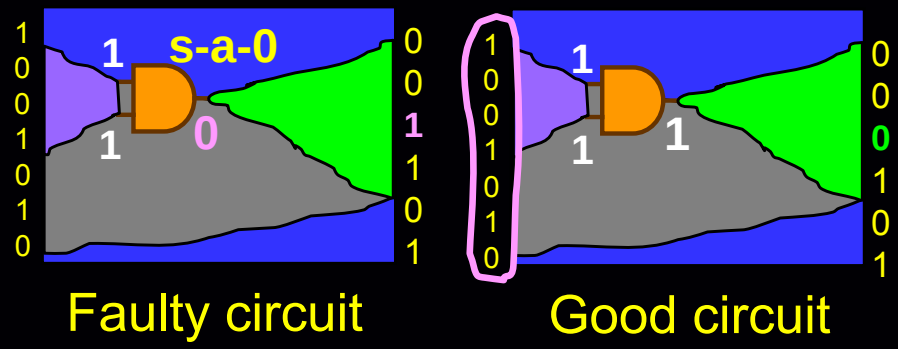
Redundancy in a Combinational Circuit

- Redundancy in a combinational circuit
  = Single stuck-at fault untestable



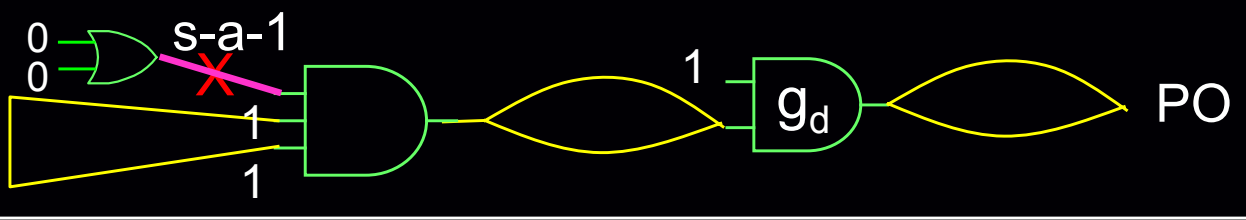
Background: Single Stuck-at Fault Untestable

- Sufficient untestable condition
  - ➔ The mandatory assignment (MA) of the stuck-at fault has conflict



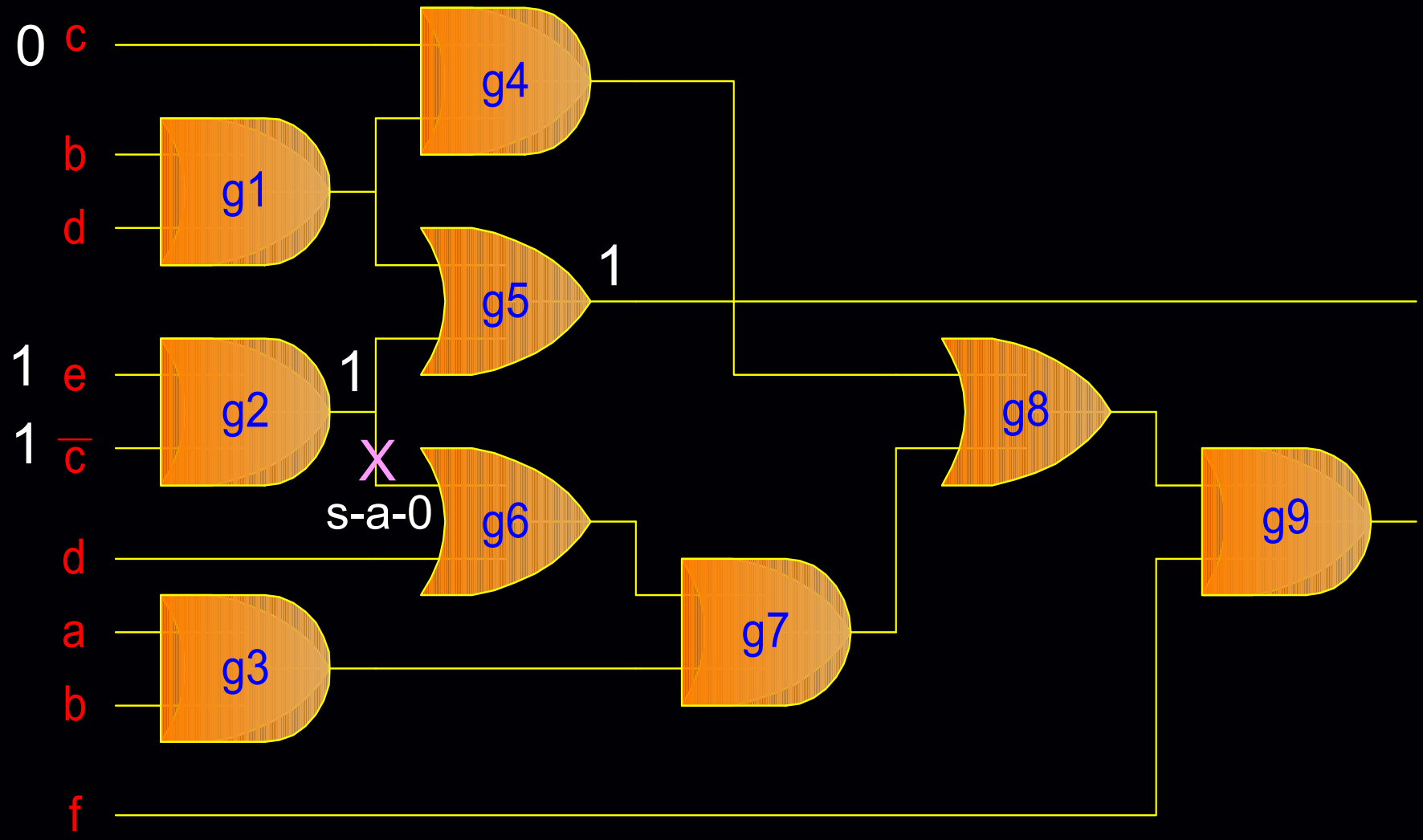
- Mandatory assignment of a fault
  - Denoted as MA(w) or MA(g), where ‘w’ or ‘g’ is the fault location (wire or gate)
  - Implications of
    1. Fault sensitization @ fault site
    2. Fault propagation @ the side inputs of the dominators

- Dominators of a fault
  - The gates where all the paths from the fault site to the POs must intersect



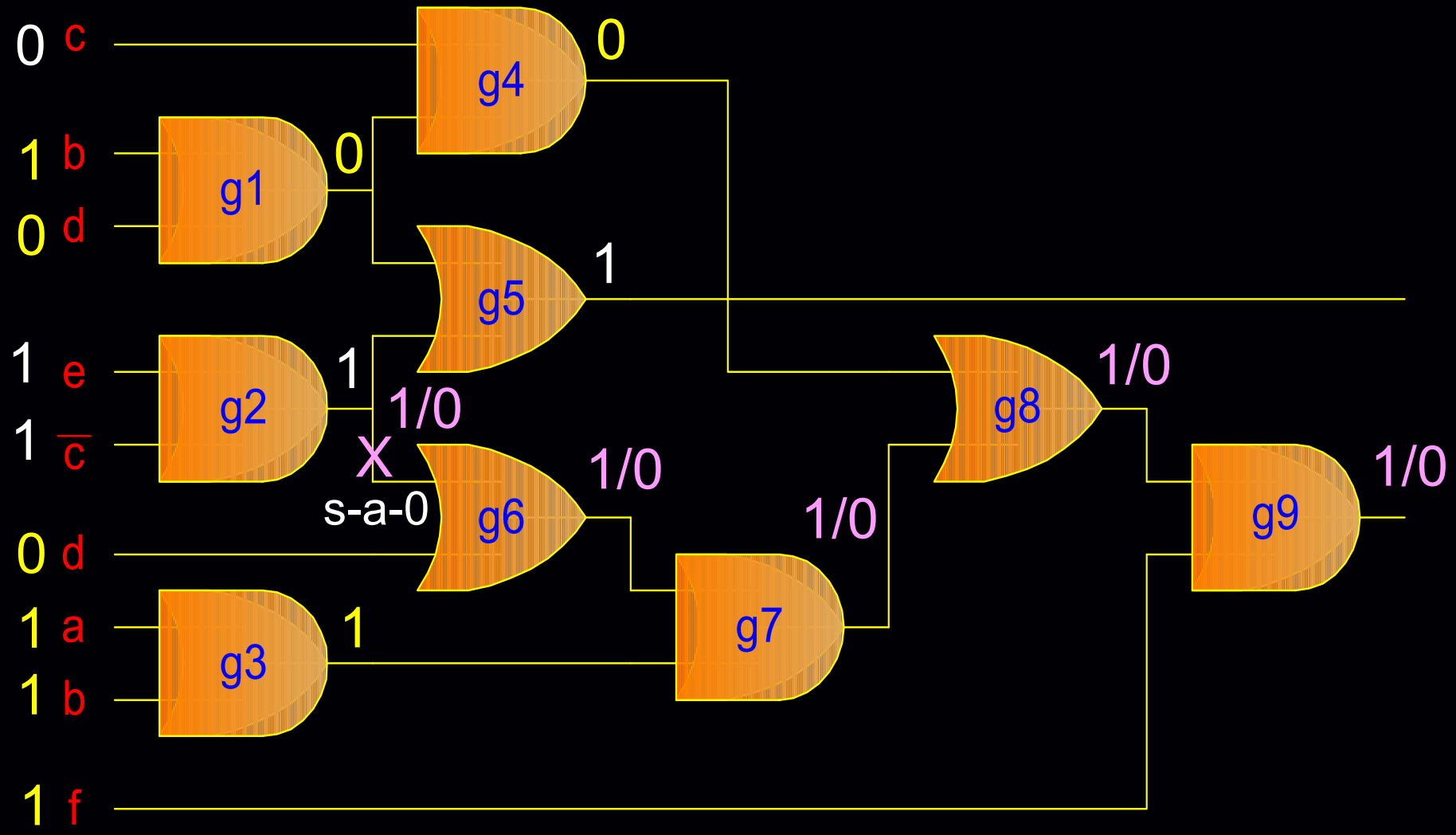
Mandatory Assignment Example

(1) Fault sensitization: g2 = 1



Mandatory Assignment Example

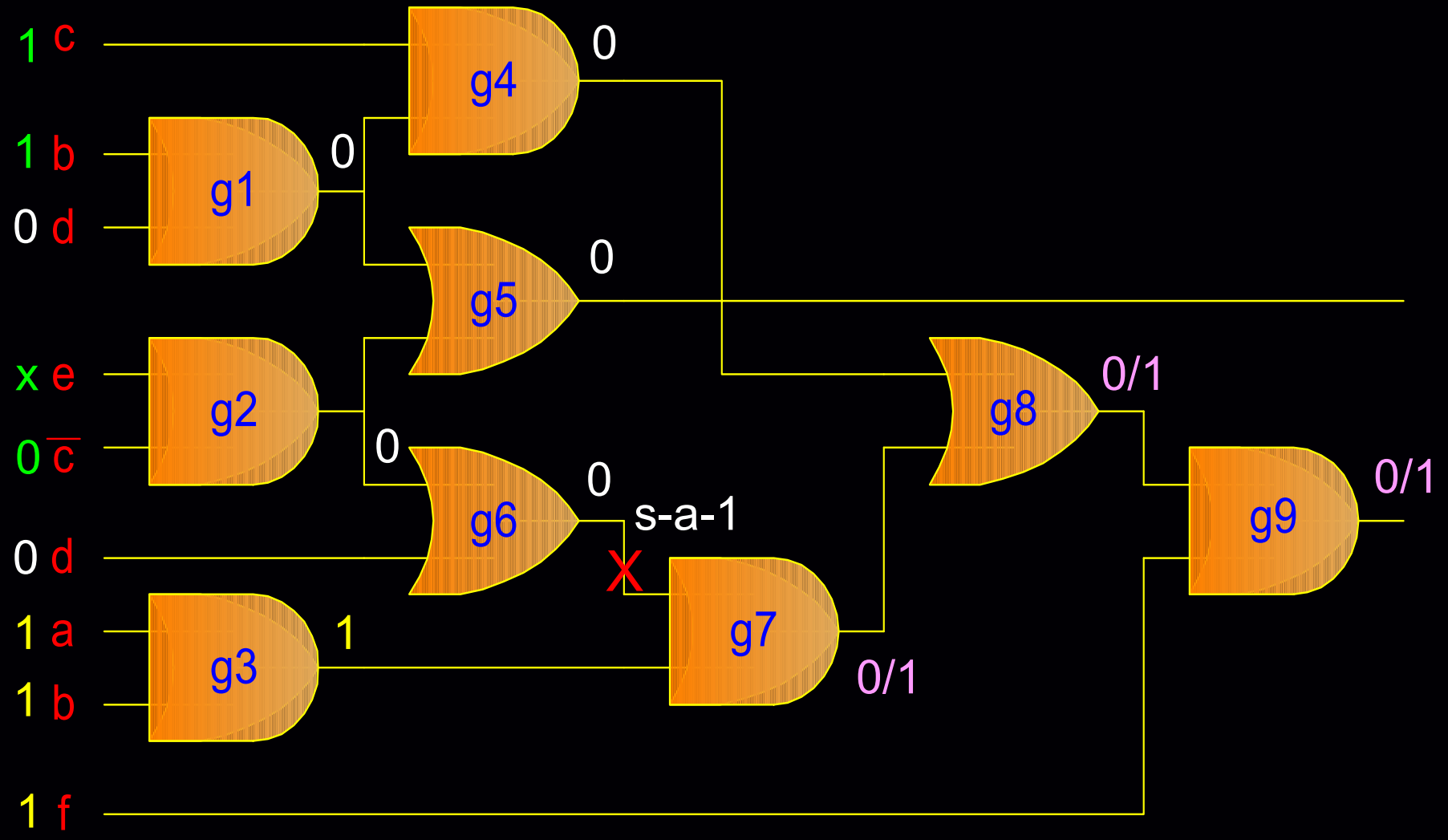
(2) Fault propagation: d = 0, g3 = 1, g4 = 0, f = 1



1. How do we know a wire in a combinational circuit is redundant?
   - ➔ Its corresponding stuck-at fault is untestable (s-a-1 for AND inputs; s-a-0 for OR inputs), or
   - ➔ MA of the fault has conflict

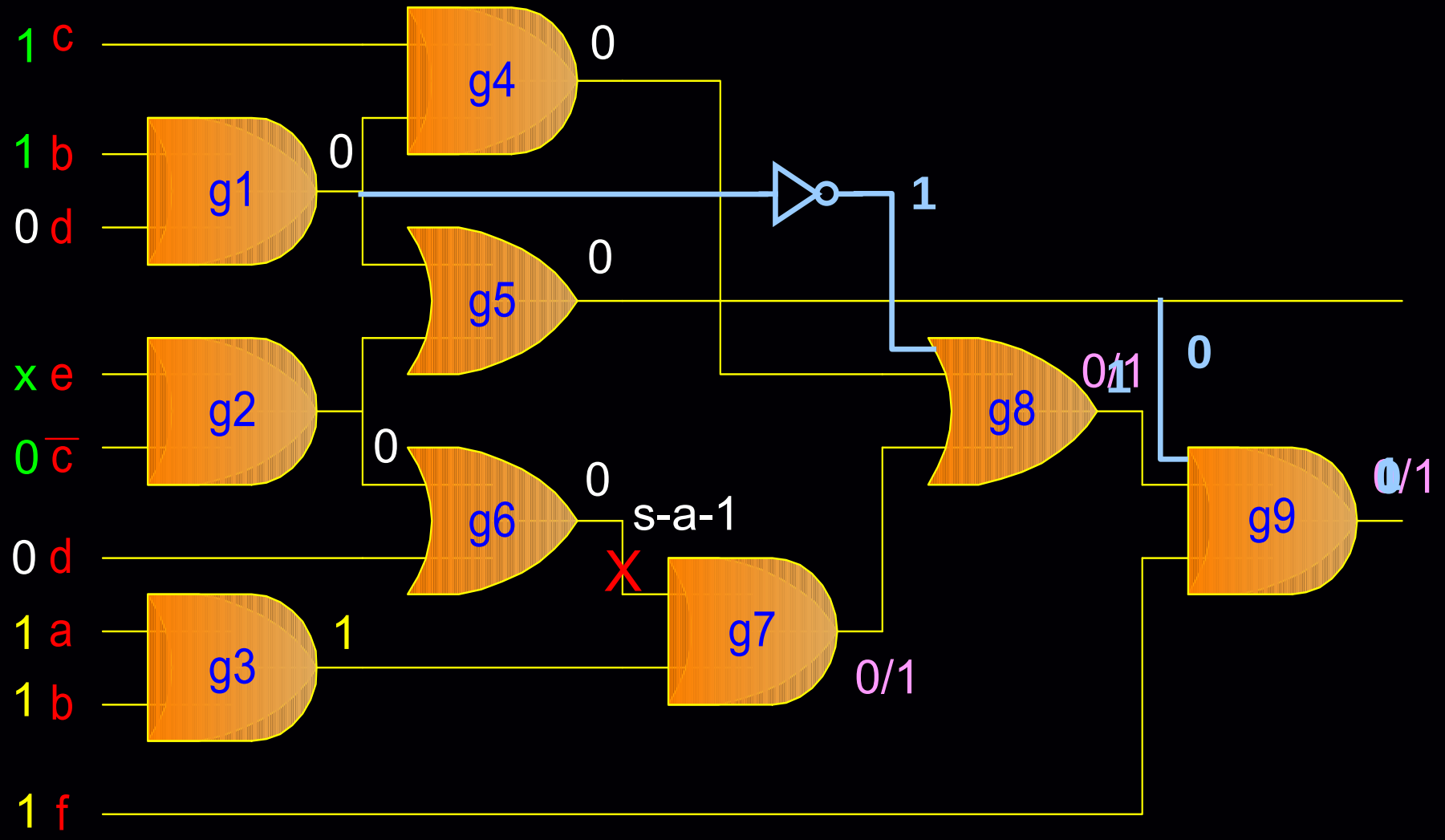
2. If a wire is NOT redundant, can we add an extra wire to make this wire redundant?
   - ➔ Yes, but the extra wire itself must be redundant
   - ➔ Add a redundant wire to make the originally irredundant wire become redundant

Target: remove g6



g6 is testable and thus NOT redundant

How to add an extra wire to make the s-a-1 fault @ g6 untestable?



Adding a wire (or with inverter) from any implied gate to a dominator

But remember,

the added wire must be redundant!!

RAMBO: Redundancy Addition and removal for Multi-level Boolean Optimization

[Cheng et.al. TCAD 1995]

1. Given a target wire, perform its mandatory assignments (MA) for its corresponding s-a fault

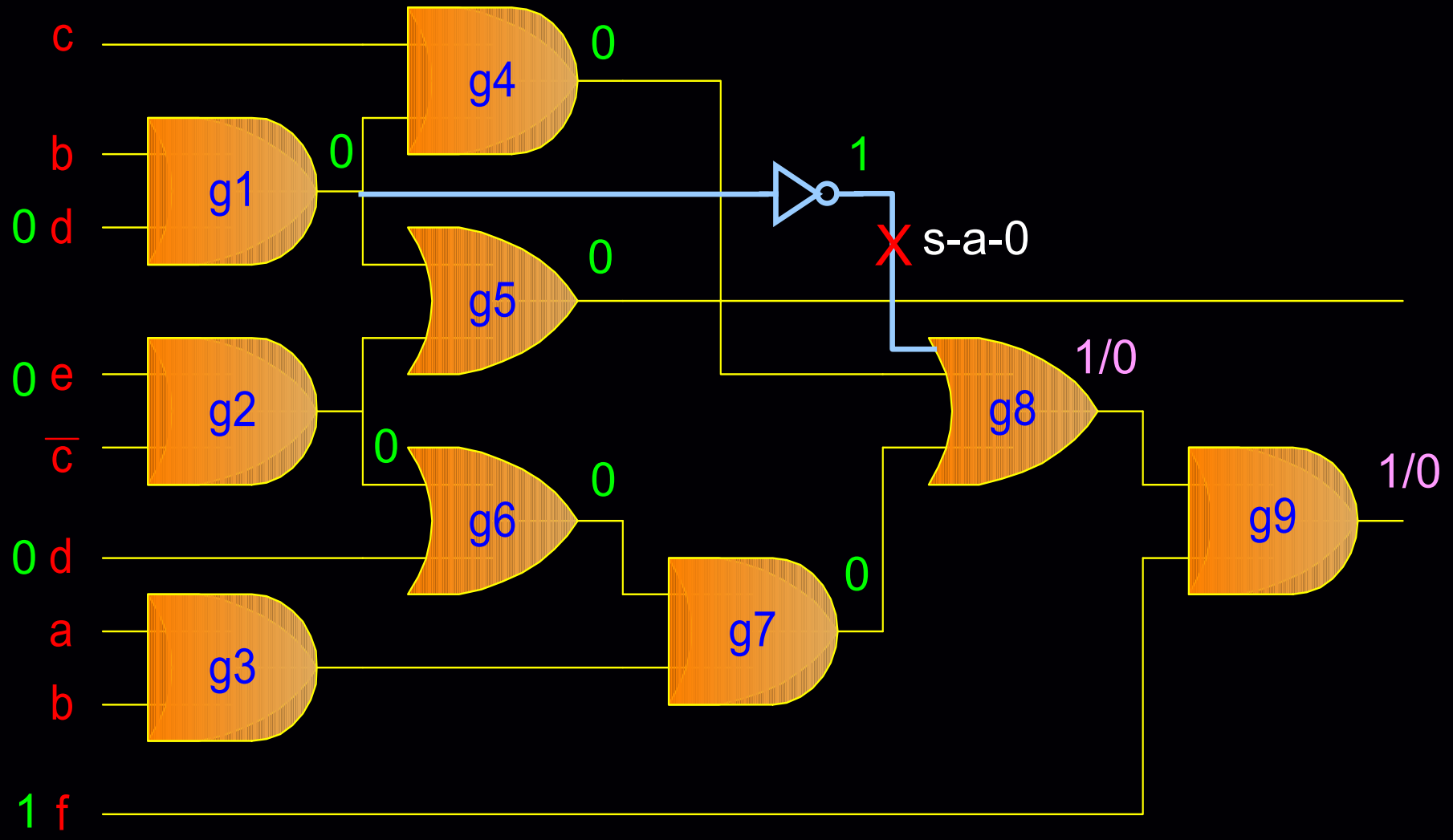
2. For each gate g_m in the set of MA,

   For each dominator g_d, test the fault on the added wire (g_m → g_d)

   a. If value(g_m) = 0 and g_d is an AND ➔ direct connection
   b. If value(g_m) = 1 and g_d is an AND ➔ add an inverter
   c. If value(g_m) = 0 and g_d is an OR ➔ add an inverter
   d. If value(g_m) = 1 and g_d is an OR ➔ direct connection

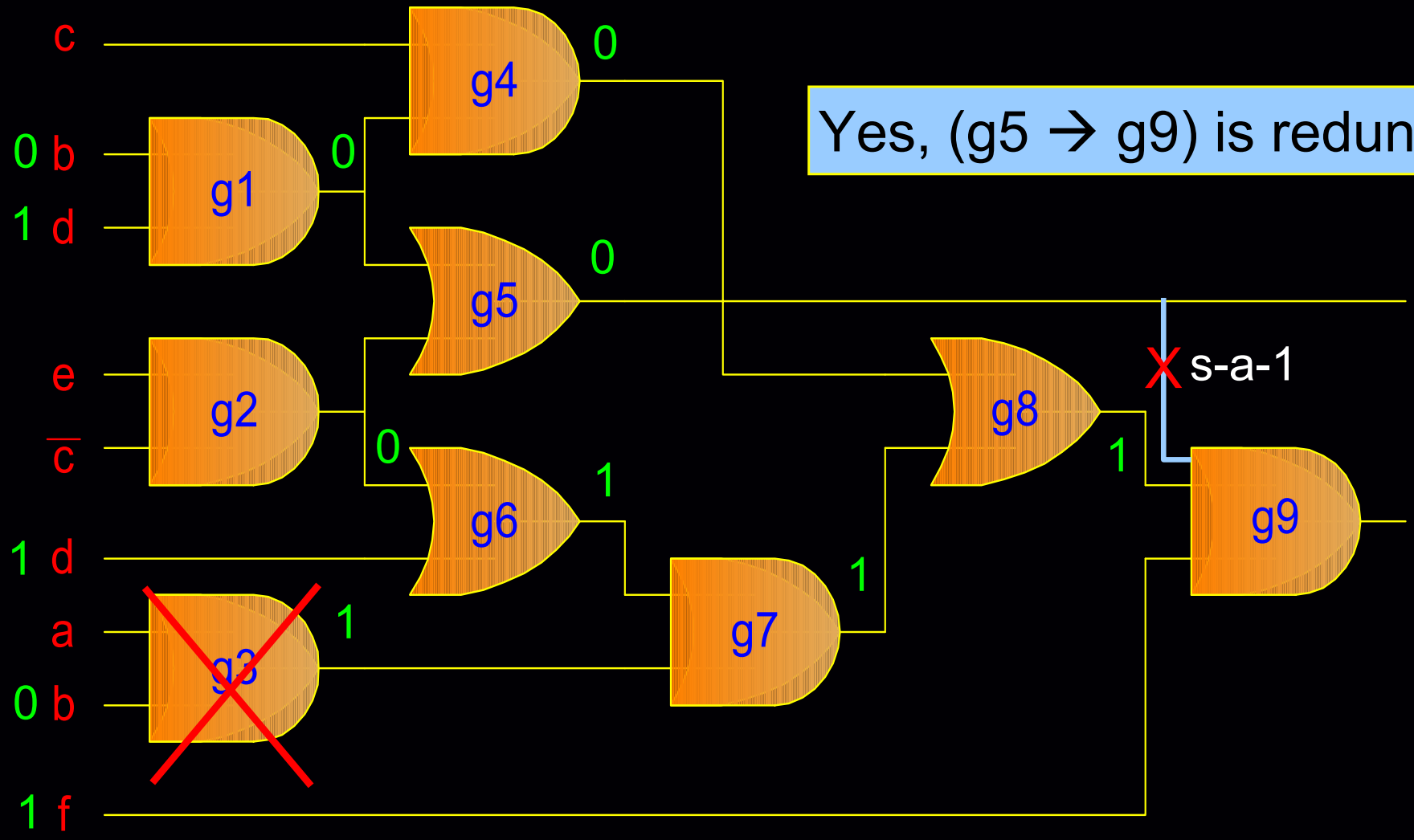
3. If the fault on the added wire in 2.a ~ 2.d is untestable,
   ➔ the added wire is redundant and can be an alternative wire to remove the target wire

Is (!g1 → g8) redundant?



No, (!g1 → g8) is NOT redundant

Is (g5 → g9) redundant?



Yes, (g5 → g9) is redundant

We can remove g6 and then g7

RAMBO Algorithm Complexity

- Need to perform (M * D) redundancy tests
  - M: number of gates in MA
  - D: number of dominators
  - ➔ Could be a BIG number

- "Perturb and Simplify" (Chang, et. al. TCAD 1996)
  - Propose several rules to filter out impossible candidates

Filtering Out Impossible Candidate Wires
(Chang, et. al. TCAD 1996)

- "Forced MAs" are the MAs ---
  1. To activate the fault site
  2. Obtained by setting the side inputs of dominators to non-controlling values
  3. Obtained by backward implications of 1 & 2
- If w_a (n_s → n_d) is an alternative wire of wire w_t, and n_d is an AND(OR) gate, then for w_t stuck-at-fault test ---
  1. n_s must have a MA 0(1)
  2. n_d must have a forced MA 1 or $\overline{D}$ (0 or D)

  → Still many candidates

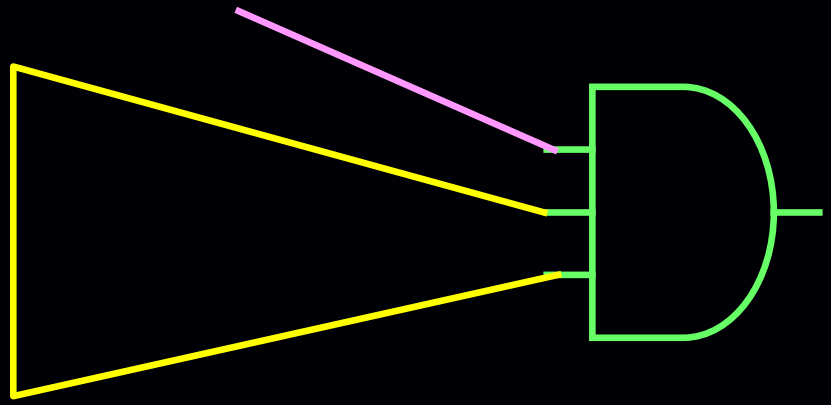
In short, given a target wire, it is easy to add a wire to its dominator to make this wire redundant.

The problem is, need to make sure the added wire is redundant. This may require a large number of fault tests.

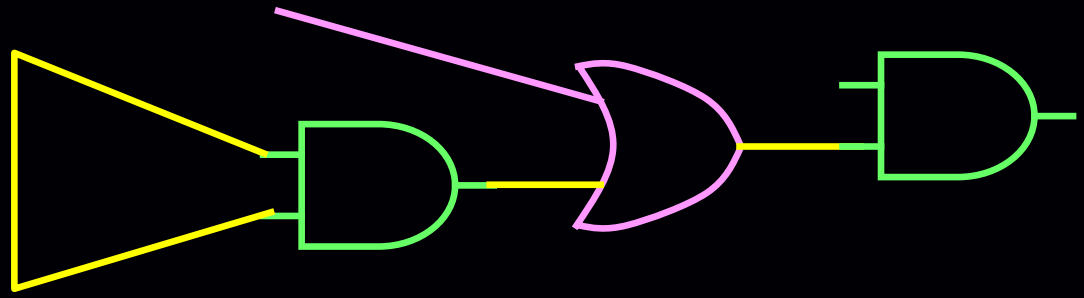
So, can we deliberately add something to a circuit, and guarantee that it is redundant?

How do we add “something” to a circuit and guarantee it is redundant?

- Add a wire

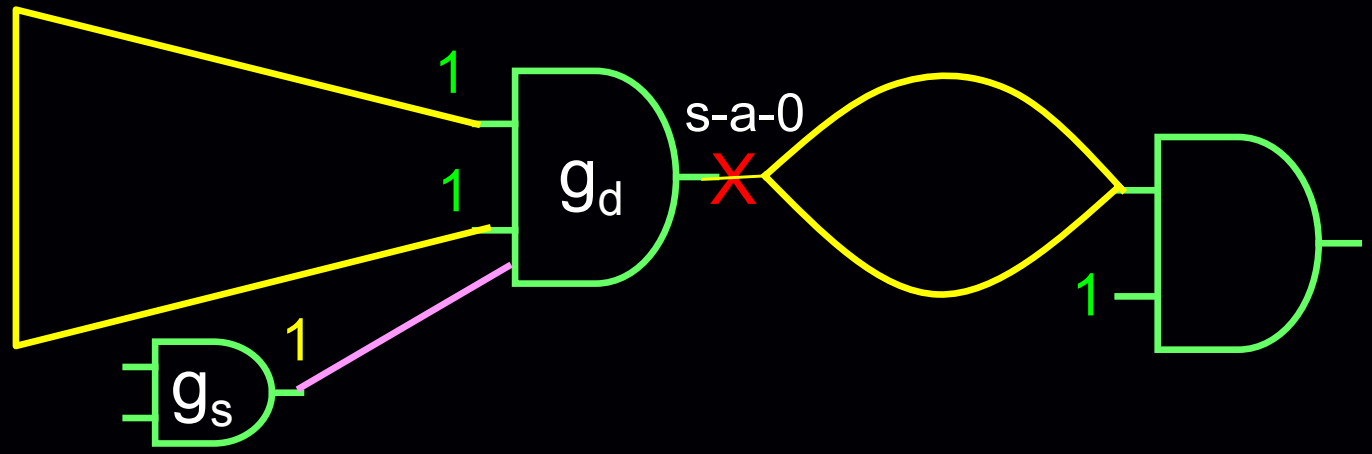


- Add a gate



Creating a Redundant Wire

- e.g. Add to the input of an AND gate g_d
  1. Test the output s-a-0 fault of this AND gate
  2. Perform MA of this fault

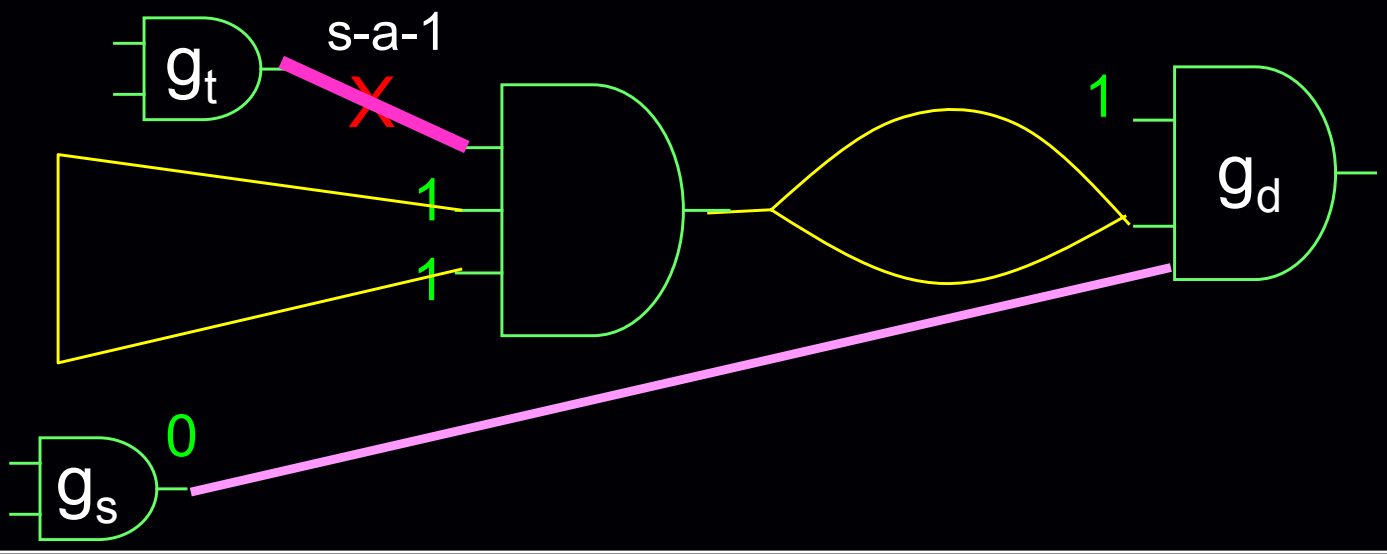


  3. For each gate g_s in the MA, there is a corresponding redundant wire (or with inverter) to g_d

Why??

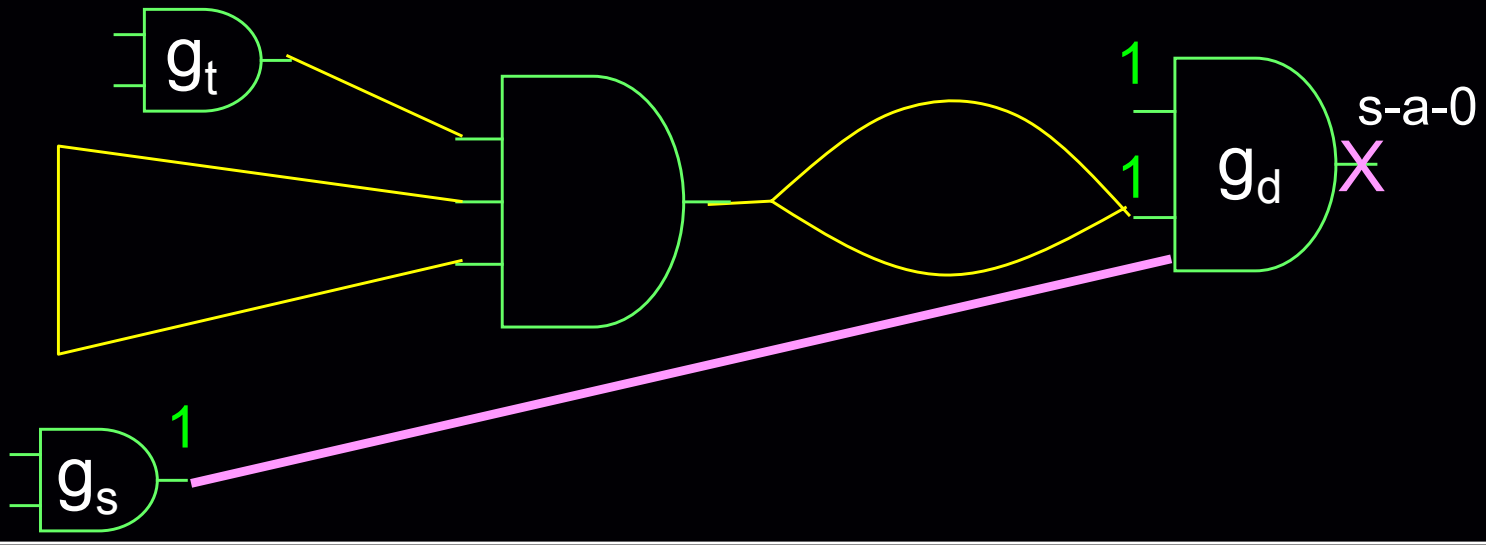
A Two-Way Redundancy Addition and Removal (2-Way RAR) Algorithm

1. Given a target wire on g_t, perform MA(g_t)
   - Adding a wire from a gate g_s in MA(g_t) to any of its dominator g_d can make this target wire redundant
   - e.g. value(g_s) = 0 → AND gate g_d

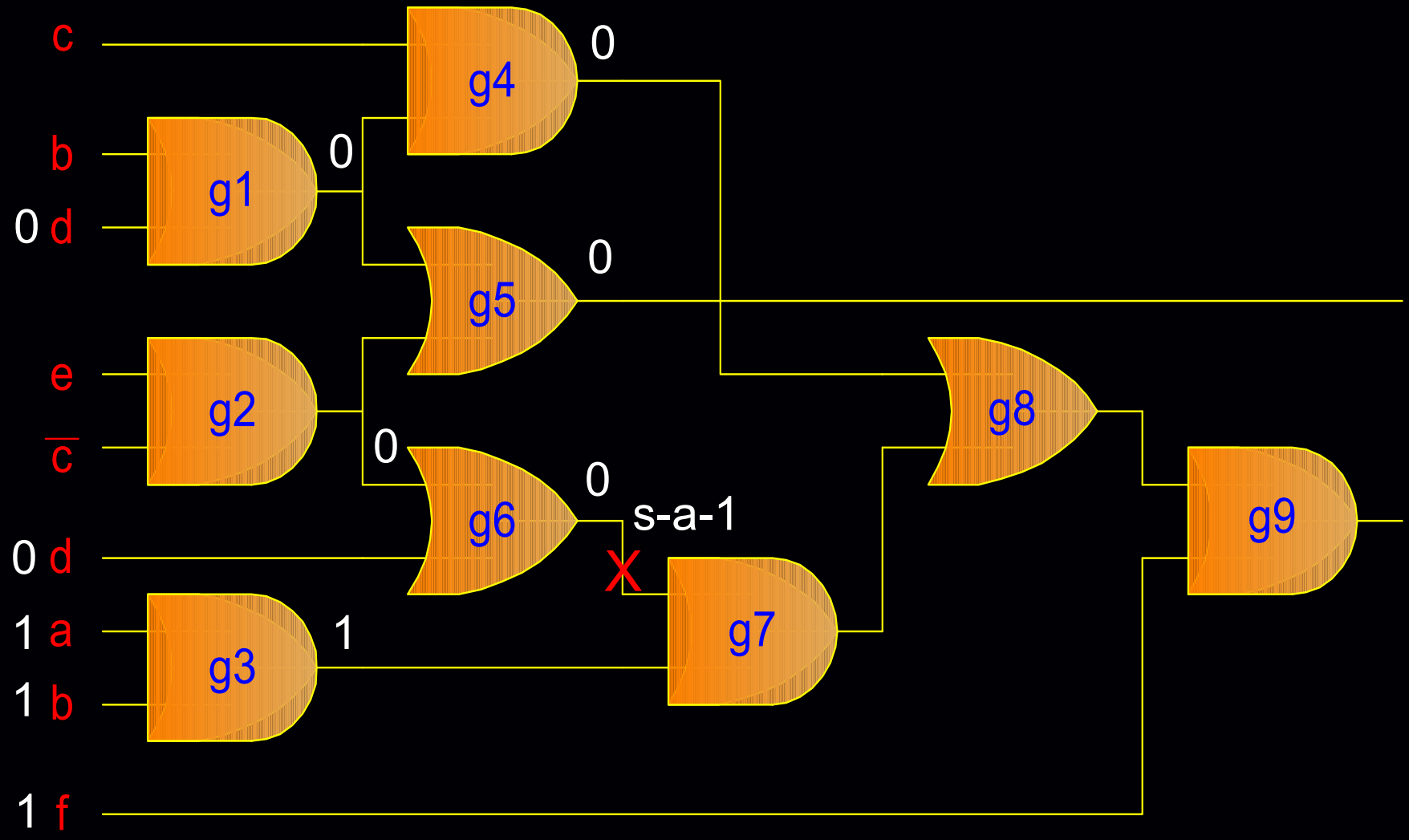


2-Way RAR Algorithm

1. Given a target wire on g_t, perform MA(g_t)
   - Adding a wire from a gate g_s in MA(g_t) to any of its dominator g_d can make this target wire redundant
   - e.g. value(g_s) = 0 → AND gate g_d
2. Given a destination gate g_d(dominator of the target wire g_t), perform MA(g_d)
   - Any wire from a gate g_s in MA(g_d) to this gate g_d can be redundant
   - e.g. value(g_s) = 1 → AND gate g_d

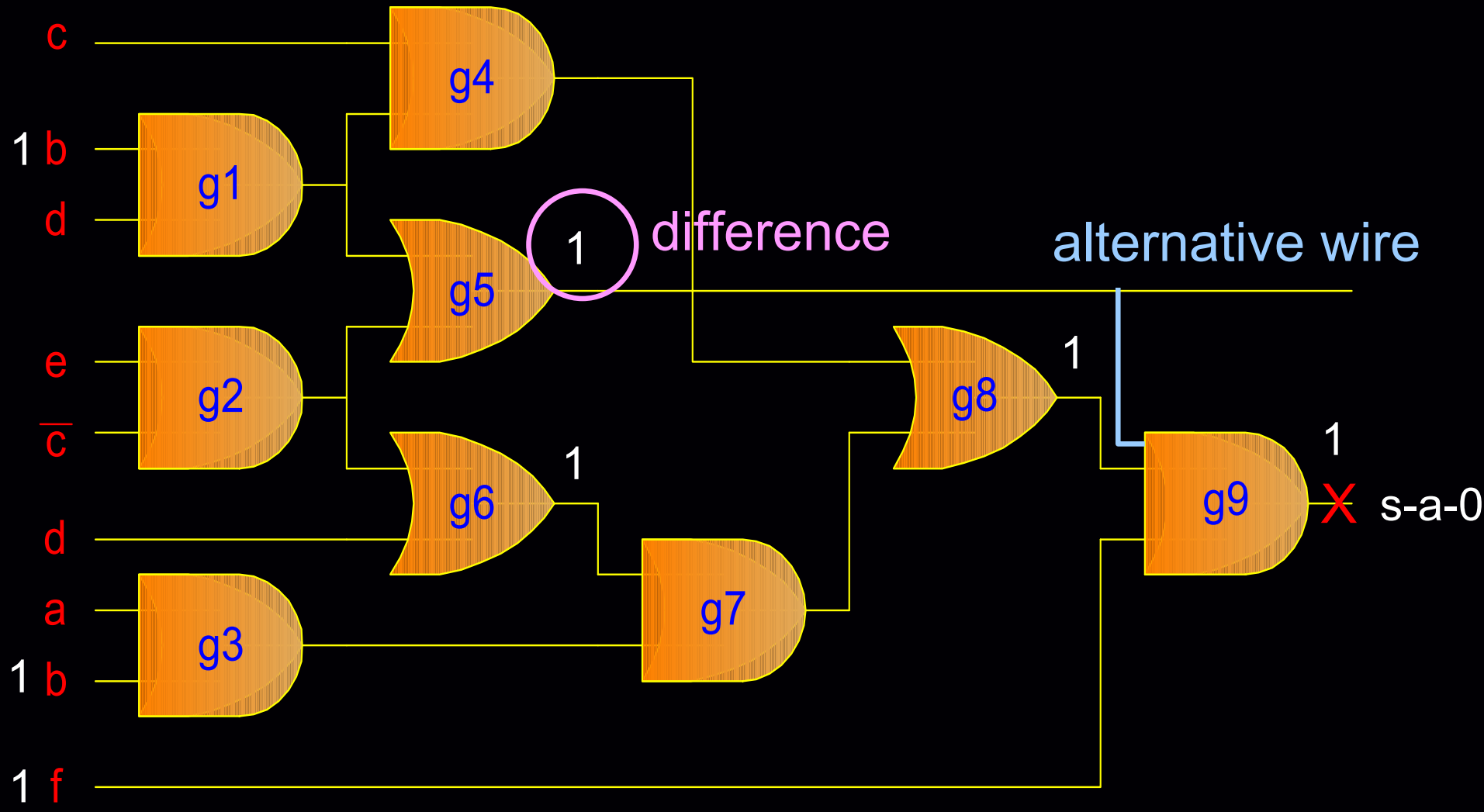


RAR Example (w_t: g6 → g7)



1. MA of w_t : g6 → g7 s-a-1

RAR Example (w_t: g6 → g7)



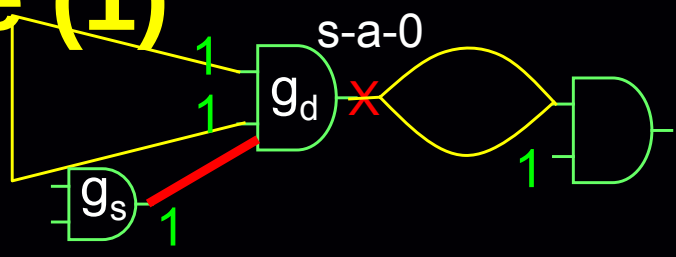
2. Try MA of g_d : g9 s-a-0

2-Way RAR Algorithm

1. Given a target wire on g_t, perform MA(g_t)
2. Given a destination gate g_d(dominator of the target wire g_t), perform MA(g_d)
3. Perform an intersection of (1) & (2)
4. Any contradiction on a gate g_s, implies an alternative wire ($g_s \rightarrow g_d$) for the target wire on g_t
   - Can be generalized for adding a gate or adding a sub-circuit

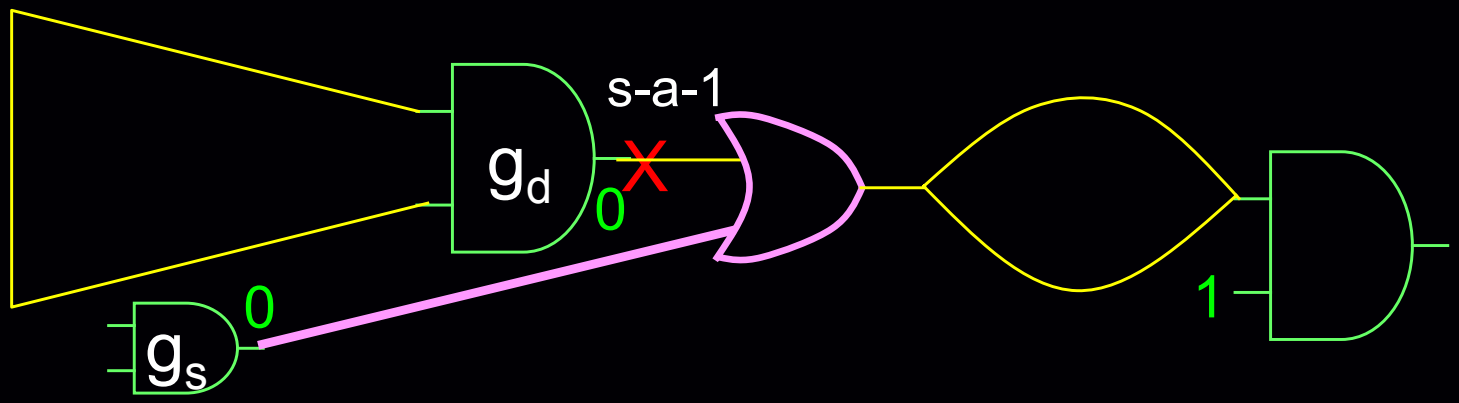
[ref: Huang ISPD 1998]

Creating a Redundant Gate (1)

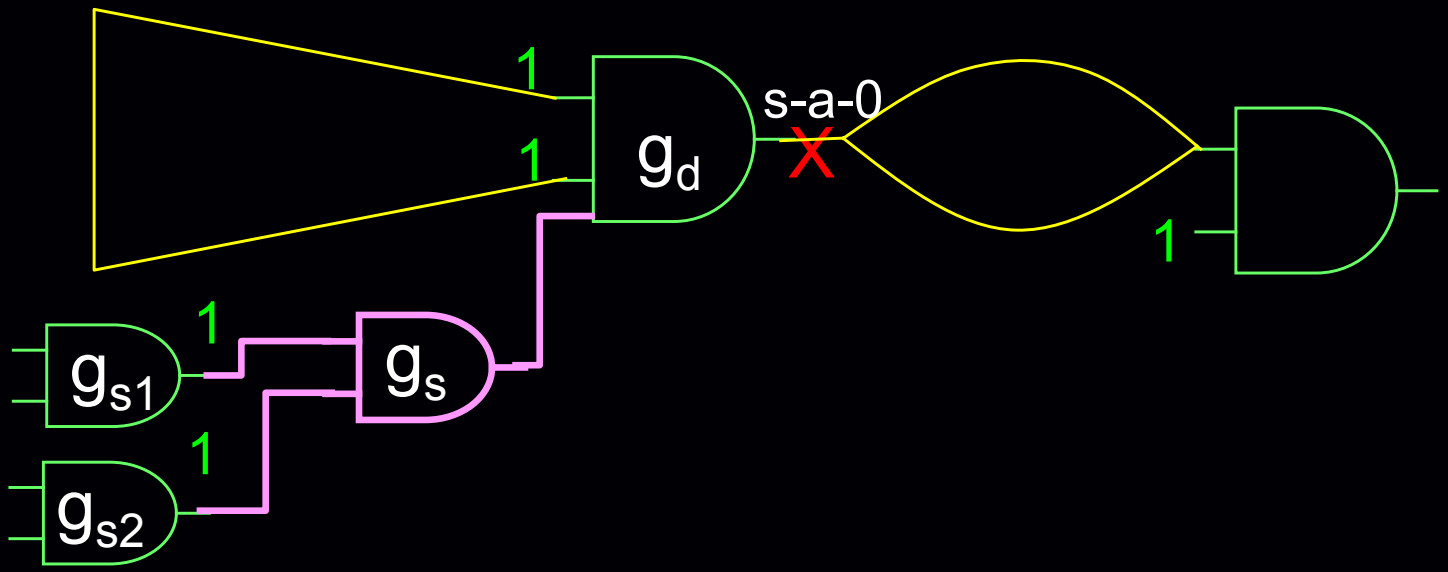


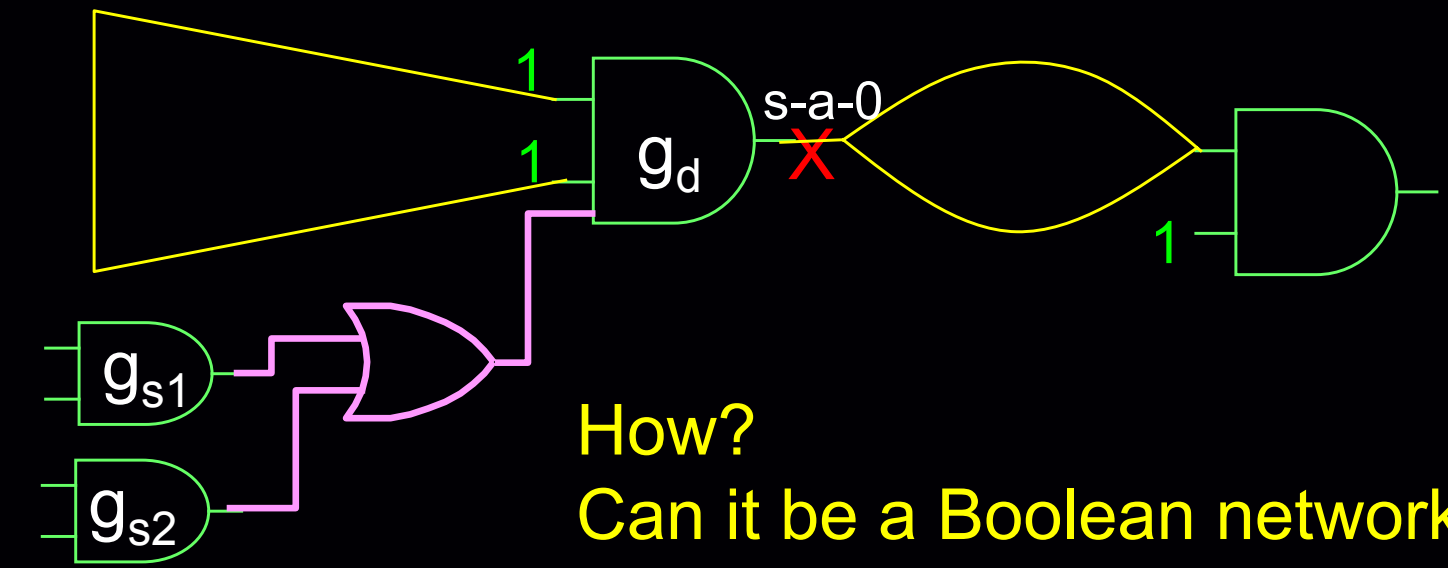
- Refresh "add a redundant wire":
  e.g. Add to the input of an AND gate g_d
  1. Test the output s-a-0 fault of this AND gate
  2. Perform MA of this fault
  3. For each gate g_s in the MA, there is a corresponding redundant wire (or with inverter) to g_d

- How about adding a redundant gate?
  ➔ Test the output s-a-1 fault of an AND gate?



Creating a Redundant Gate (2)





How?
Can it be a Boolean network?

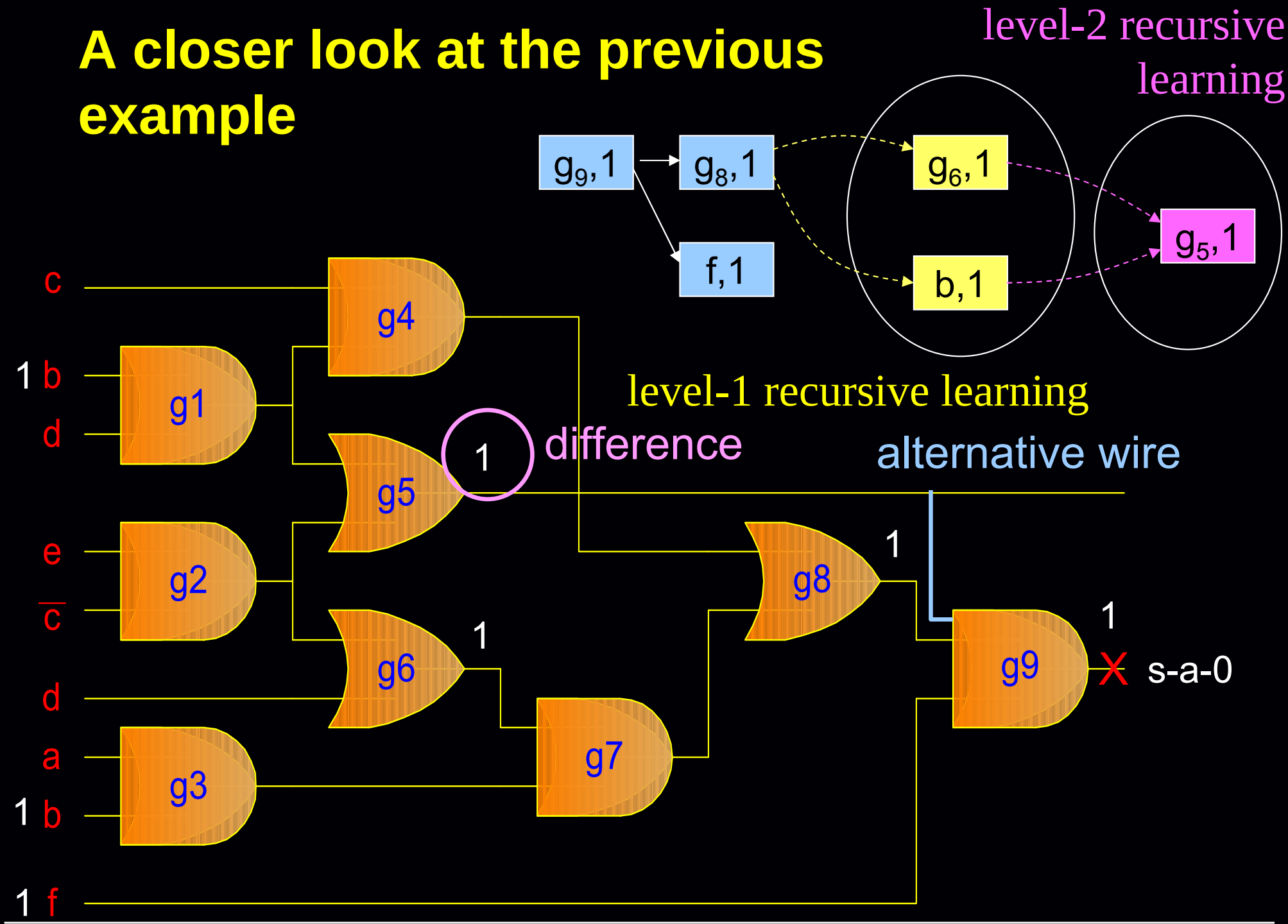
2-Way RAR Algorithm

- Pros
  - No need to perform (M*D) redundancy test as in RAMBO
  - Potential orders of speed-up
- Cons
  - Only connect to dominators?
    (Can we connect to fanins of dominators?)
  - Still need to try for each dominator
  - MA on target wire may NOT intersect with MA on dominators
    ➔ Or just find some trivial alternative wires (e.g. DeMorgan Law)

➔ Methods to deriving more MAs (e.g. Recursive learning) are often used (but could be expensive)

➔ How can we increase the number of MAs?

A closer look at the previous example



SAT-Controlled RAR (SatRAR) [Huang, ASPDAC 2009]

- Problems with previous RAR techniques
  - RAMBO: too many redundancy tests
  - 2-Way RAR: expensive implication technique needed
- SAT-controlled RAR
  - NOT just take the advantage of the advancements from the modern SAT solvers (covered later)
    - Efficient BCP, conflict-driven learning, etc
  - A seamless integration of SAT and RAR algorithms
  - Extensions for general RAR
    - Alternative wire, gate, sub-circuit identification
  - Options to “control” the RAR optimization quality

Single Wire Replacement Theorem in SatRAR

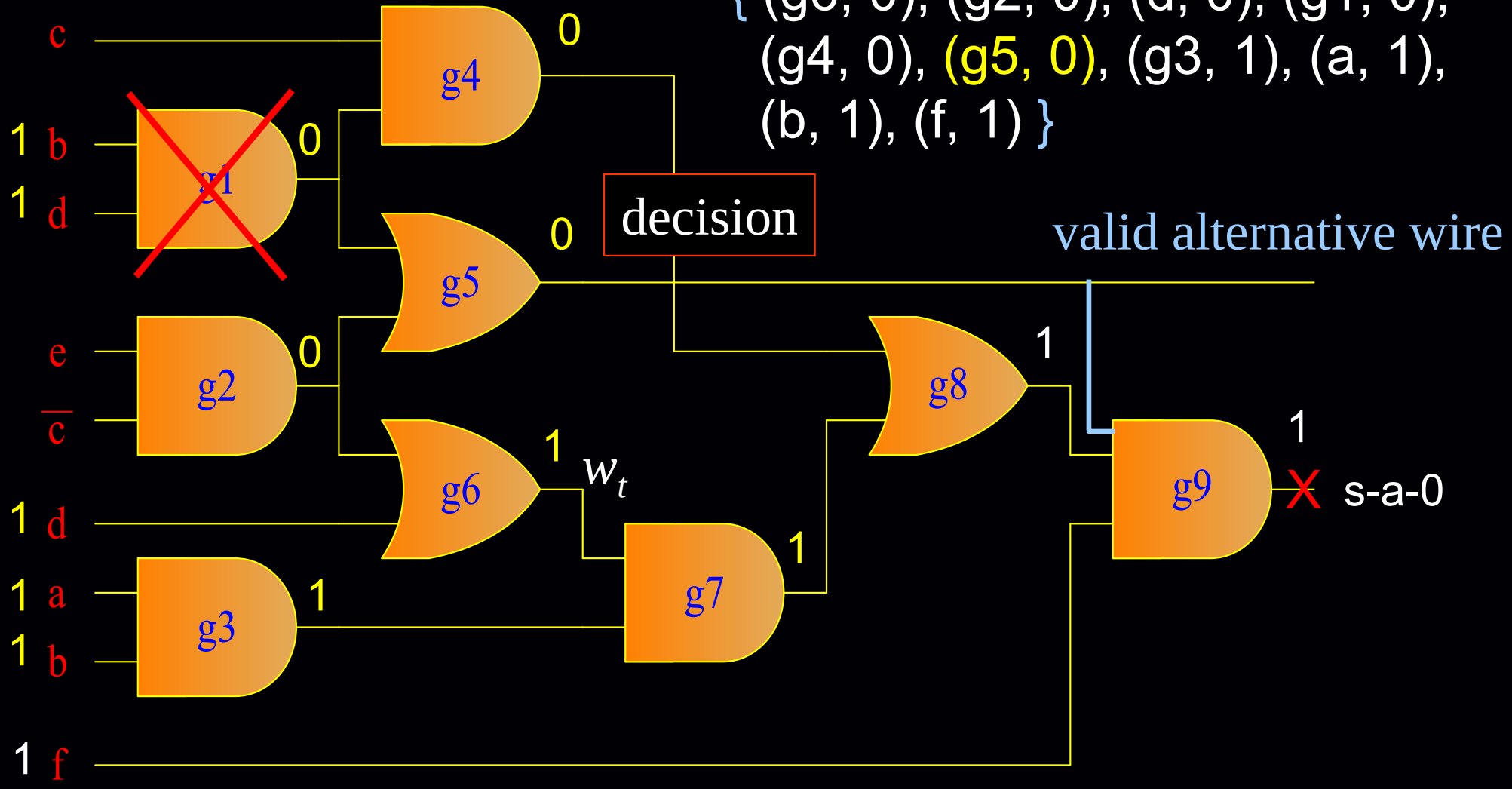
- *Let **MA(w_t)** and **MA(g_d)** be the mandatory assignments for the fault tests of the target wire w_t and its dominator g_d, respectively.*
- *Let **<g_s, v>** belong to MA(w_t) but not MA(g_d), and g_s be not in the fanout cone of g_d.*
- *If we make a decision <g_s, v> on top of MA(g_d) and encounter a conflict, then*

  (i) MA(g_d) $\Rightarrow$ <g_s, $\neg$v>

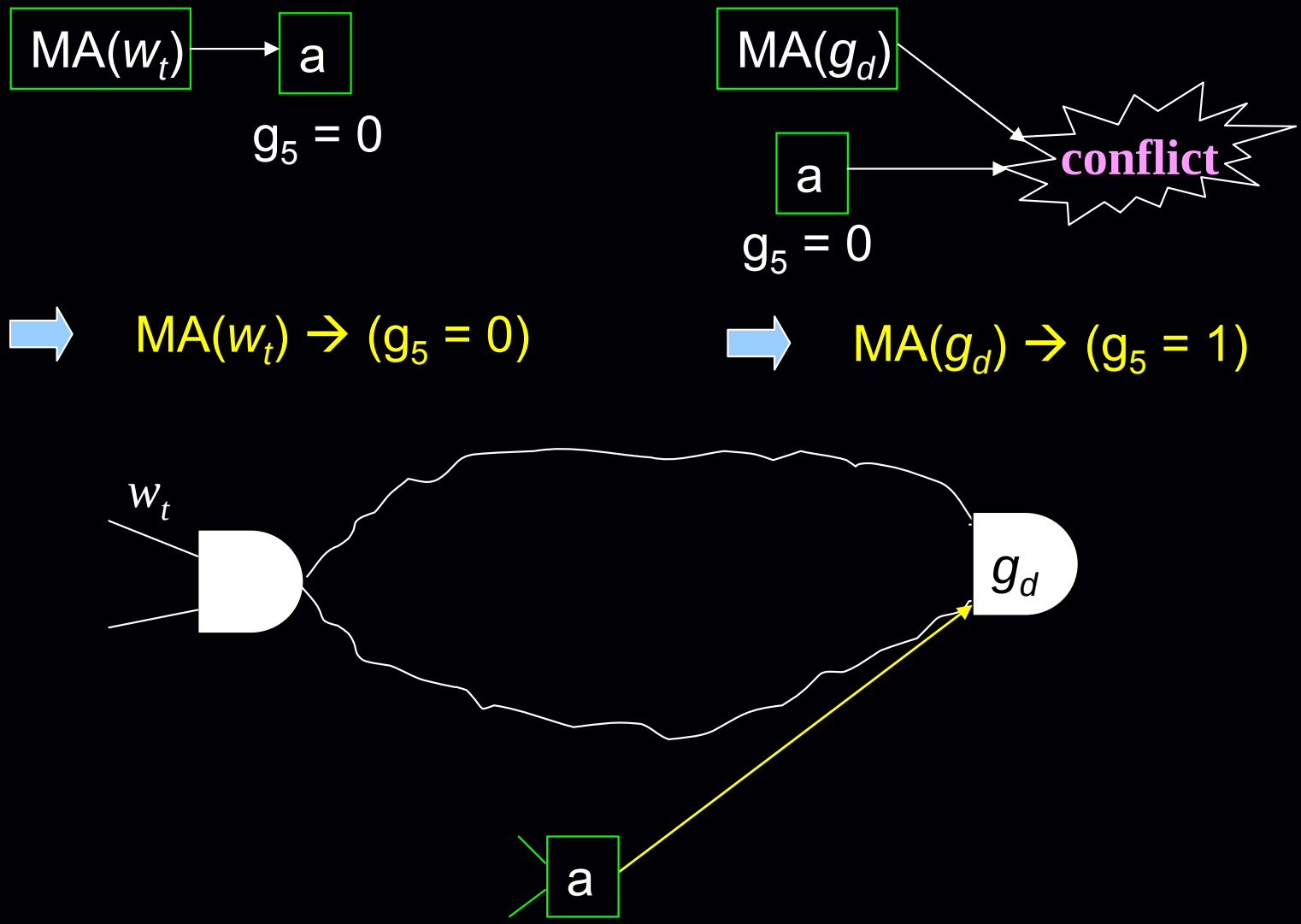
  (ii) ($g_s \rightarrow g_d$) or ($g_s \rightarrow^{\circ} g_d$) must be a valid alternative wire for w_t

Single Wire Replacement in SatRAR

$MA(w_t = g_6)$ =
{ (g6, 0), (g2, 0), (d, 0), (g1, 0),
(g4, 0), (g5, 0), (g3, 1), (a, 1),
(b, 1), (f, 1) }

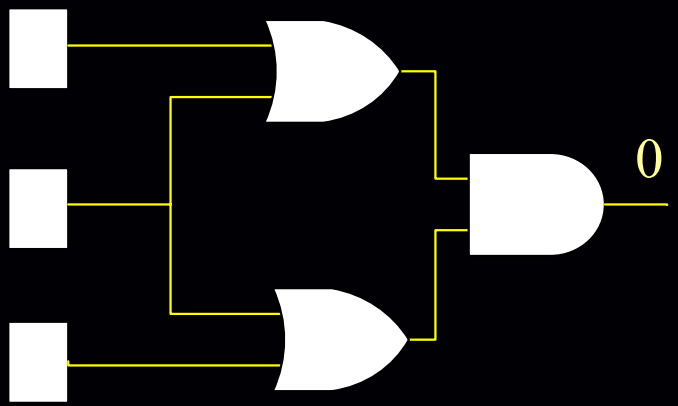


A closer look...



SRAR-Wire vs. 2-Way RAR

- Similarity
  - Based on the conflicting implications between MA(w_t) and MA(g_d)
- Difference
  - SAT decision (conflict-driven leanring) vs. Recursive learning



- Recursive learning:

  f = 0 $\Rightarrow$ d = 0 or e = 0

  $\Rightarrow$ { a=0, b=0 } or { b=0, c=0 }

  $\Rightarrow$ b = 0 (Cannot be recorded)

- Conflict-driven learning:

  f = 0; decision b = 1 results in conflict

  $\rightarrow$ f = 0 $\Rightarrow$ b = 0 (Recorded!!)

SRAR-Wire vs. Original RAMBO (FYI)

- Similarity (looks like...)
  - For each assignment g_s in { MA(w_t) - MA(g_d) }... vs.
    For each assignment g_s in MA(w_t)...
- Difference
  - Incremental SAT vs. Independent redundancy tests
- Incremental SAT in SatRAR
  - MA(g_s) is performed on top of MA(g_d)
    - Sharing of different MA(g_{di})
  - Conflict-driven learning
    - Learning & RAR at the same time
  - Implication filter
    - Reduce #decisions
  - More importantly, can be extended for alternative gate/sub-circuit replacements

Single Gate Replacement Theorem in SatRAR

- *Let **MA(w_t)** and **MA(g_d)** be the mandatory assignments for the fault tests of the target wire wt and its dominator gd, respectively.*
- *Let both **<g_s, u>** and **<g_t, v>** belong to MA(w_t), and be not in the fanout cone of gd.*
- *Suppose we make the decision <g_s, u> after MA(g_d) and result in an implication <g_t, $\neg$v>.*
- *Let a gate g_n = AND(<g_s, u>, <g_t, v>). Then*

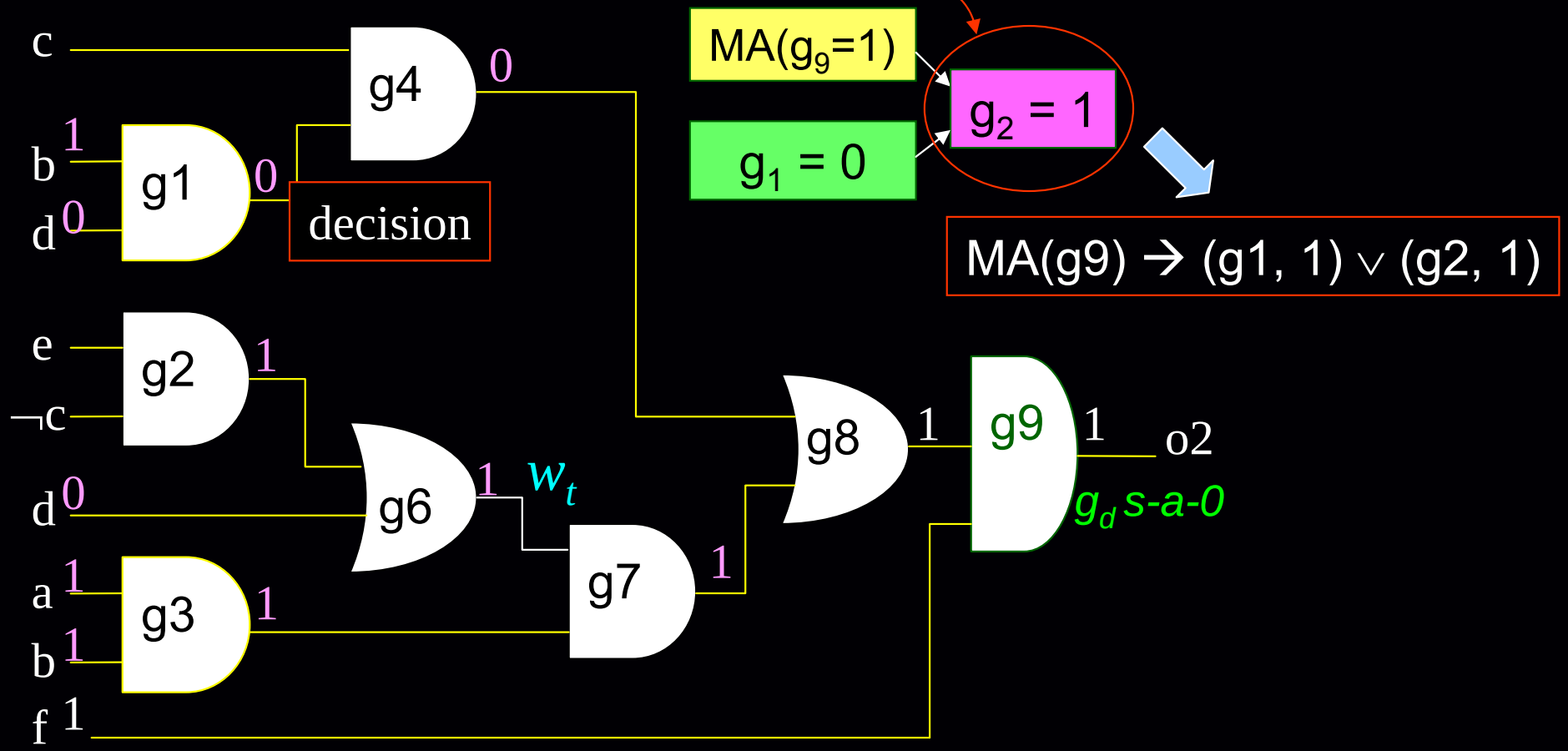
  (i) MA(g_d) $\Rightarrow \neg g_n$,

  (ii) g_n or $\neg g_n$, when connected to g_d, must be a valid alternative gate for w_t

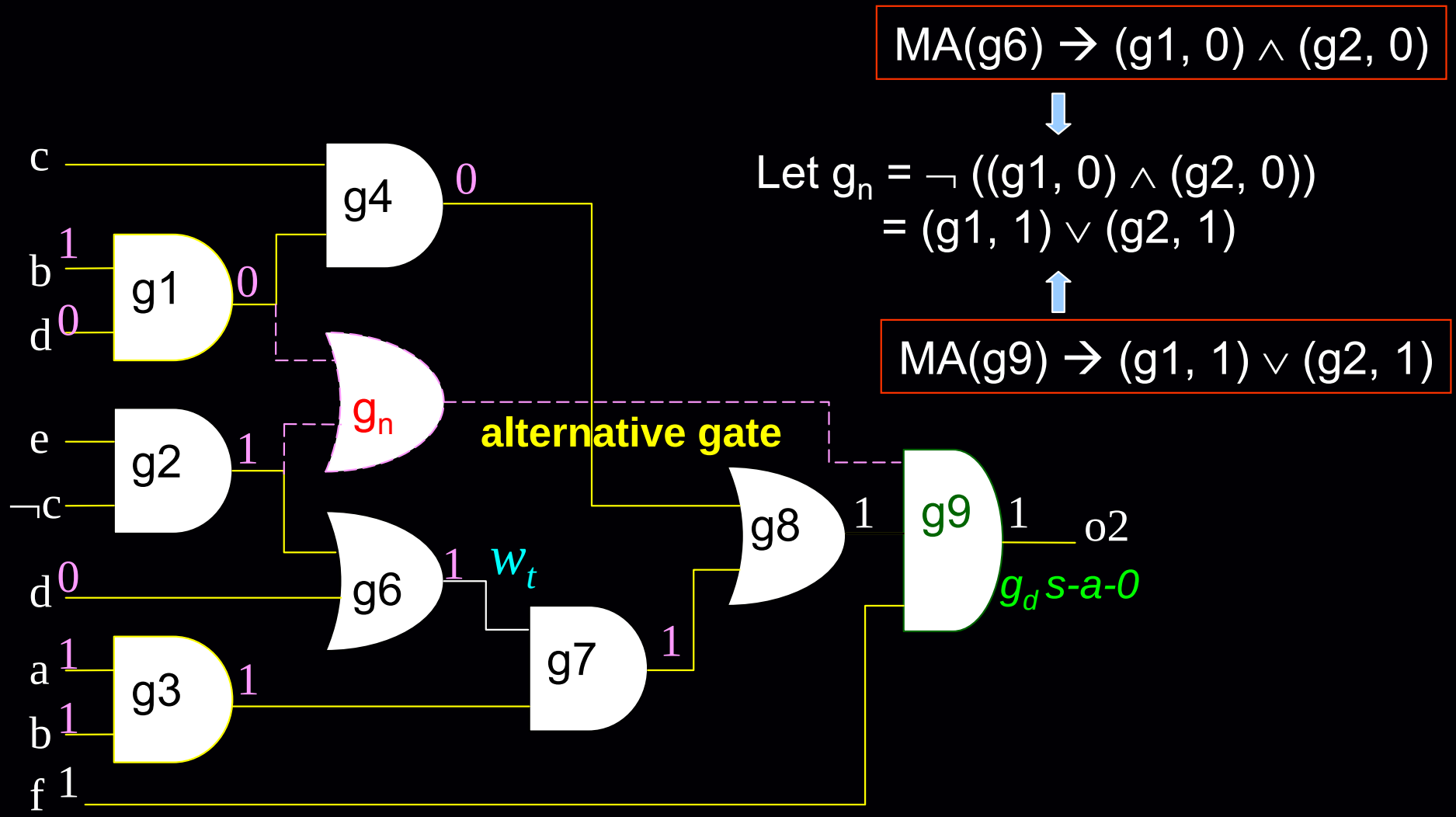
Single Gate Replacement in SatRAR

* MA(g6) = { (g6, 0), (g2, 0), (d, 0), (g1, 0), (g4, 0), (g3, 1), (a, 1), (b, 1), (f, 1) }

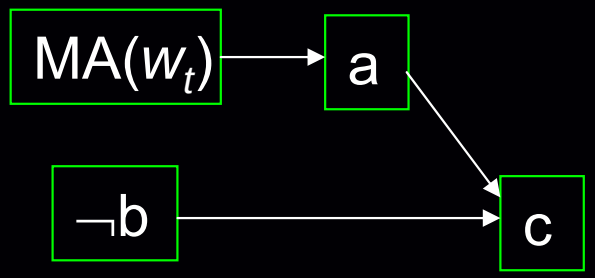
MA(g6) → (g1, 0) ∧ (g2, 0)

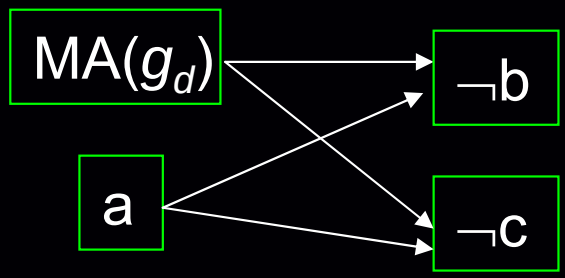


Single Gate Replacement in SatRAR



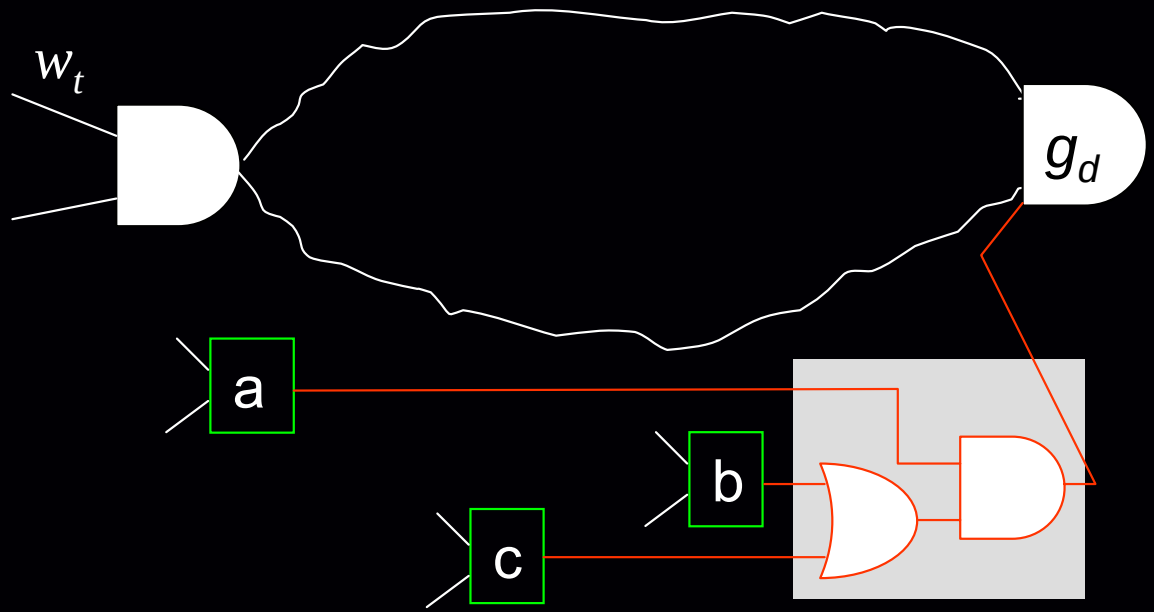
Alternative Sub-circuit by SatRAR





⇨ MA(w_t) → a ∧ (b ∨ c)

⇨ MA(g_d) → ¬ (a ∧ (b ∨ c))



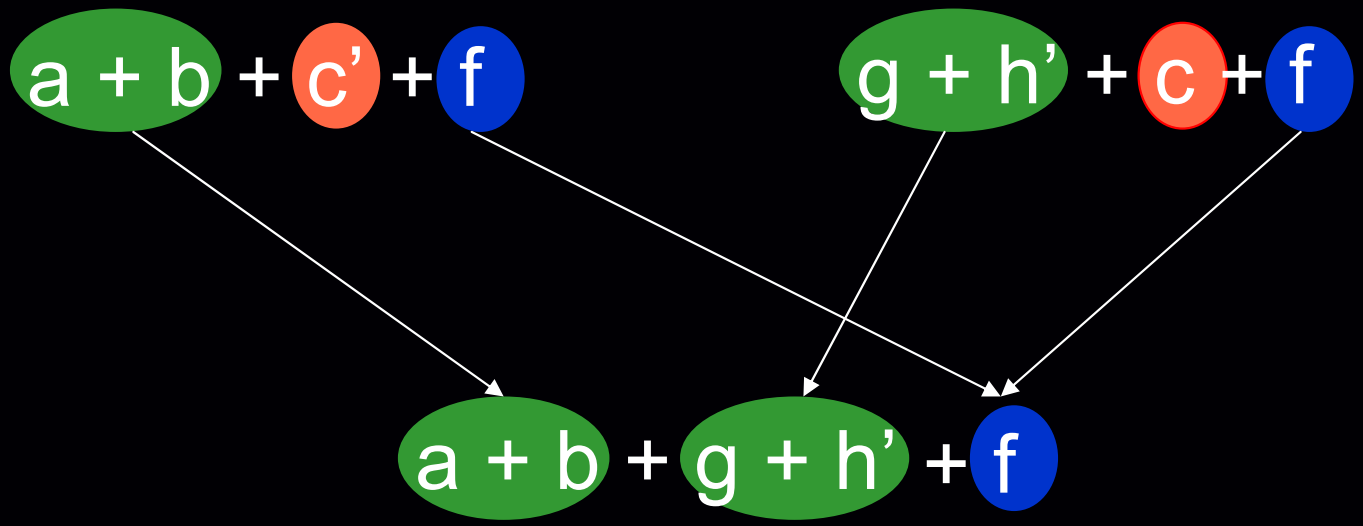
Outline

CNF-Based SAT Algorithm

1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, Logemann, Loveland, (DLL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Resolution

- Resolution of a pair of clauses with exactly ONE incompatible variable
  - Two clauses are said to have distance 1

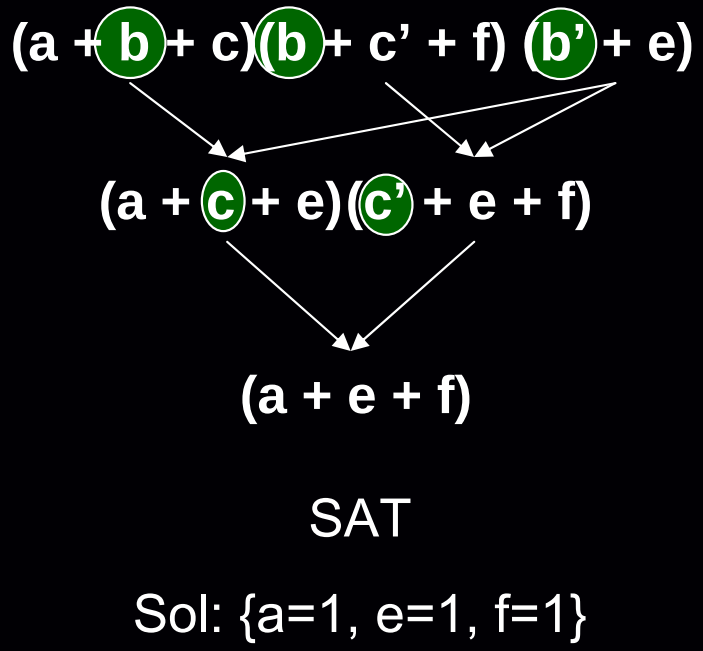


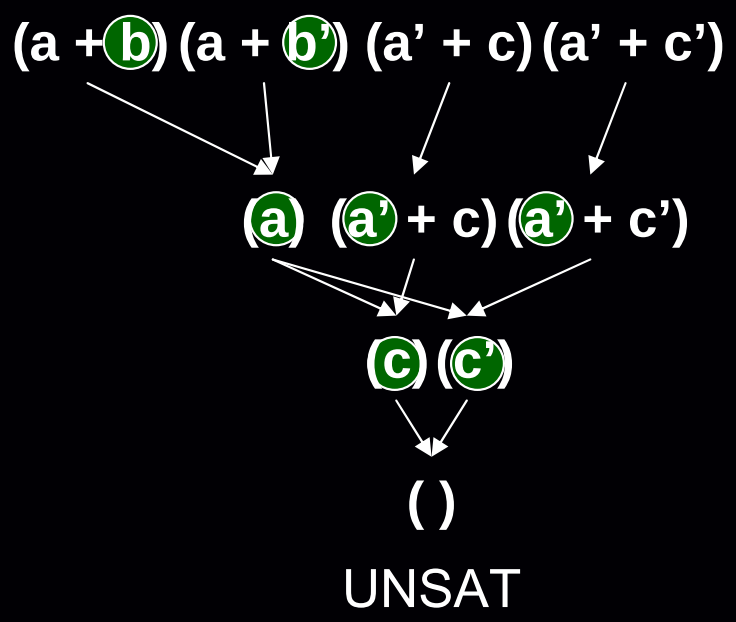
Souce: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Davis Putnam Algorithm

M .Davis, H. Putnam, “A computing procedure for quantification theory", *J. of ACM*, Vol. 7, pp. 201-214, 1960 (360 citations in citeseer)

- Existential abstraction using resolution
- Iteratively select a variable for resolution till no more variables are left.





Potential memory explosion problem!

Souce: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Boolean Satisfiability (SAT) Algorithm

1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, (Putnam), Logemann, Loveland, (D(P)LL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

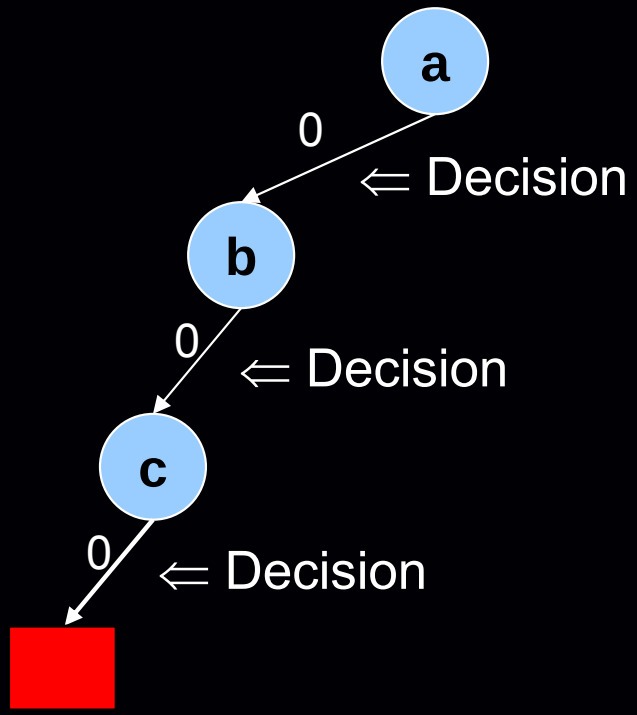
(a + c’ + d)

(a + c’ + d’)

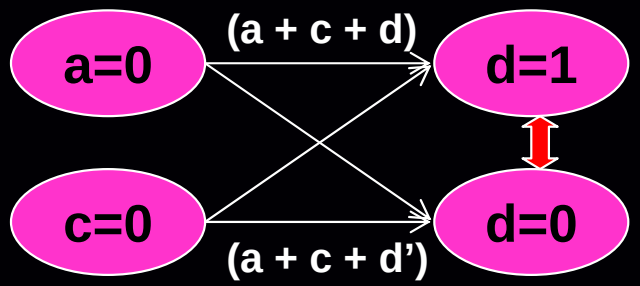
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

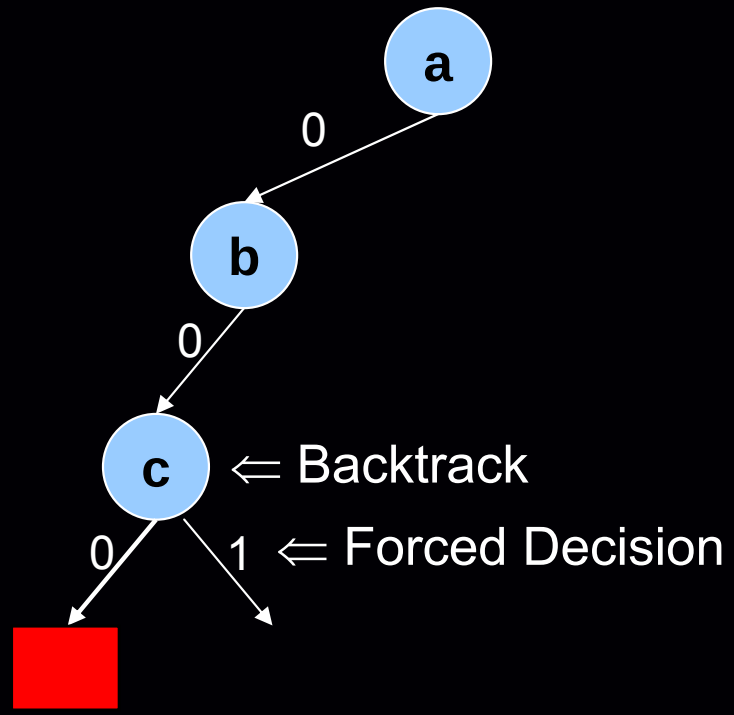
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

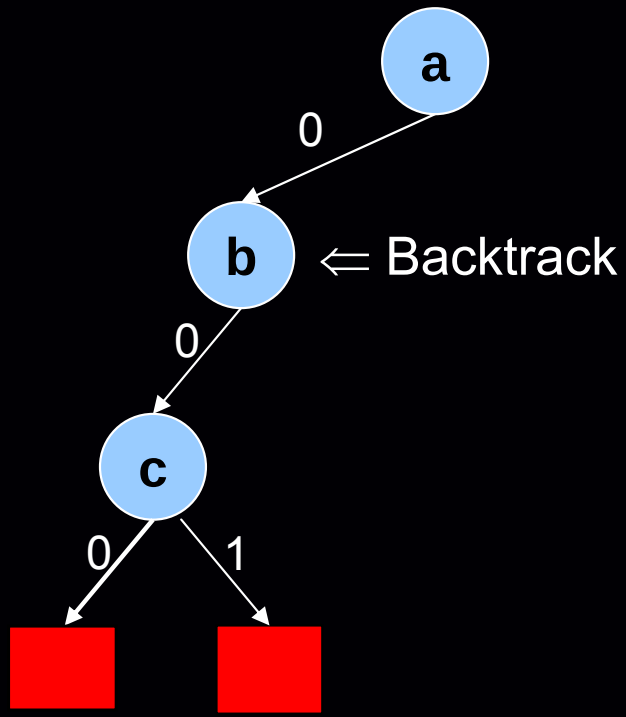
(a + c’ + d)

(a + c’ + d’)

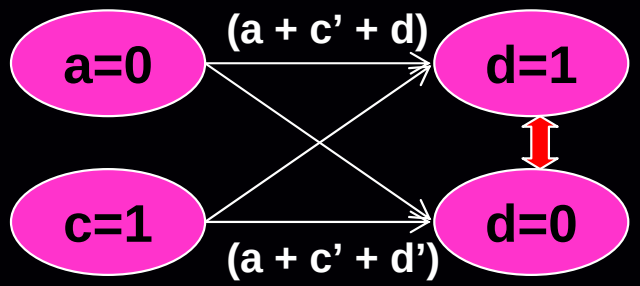
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

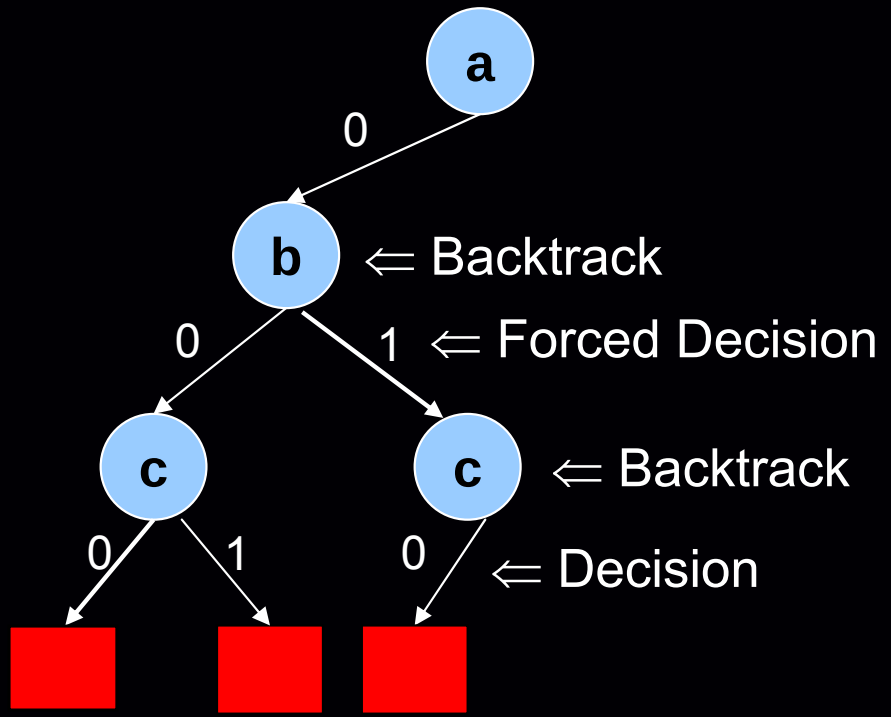
(a + c’ + d)

(a + c’ + d’)

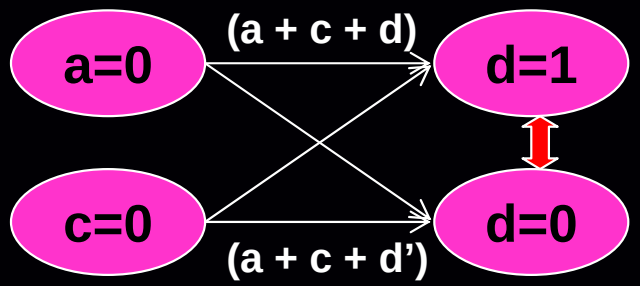
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

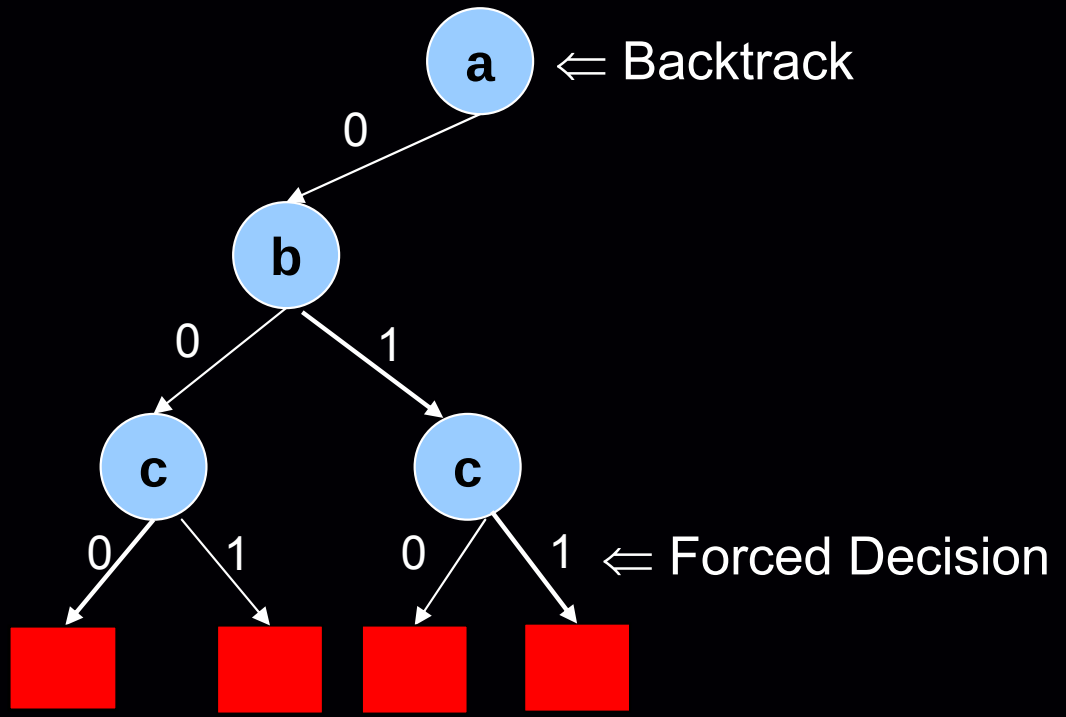
(a + c’ + d)

(a + c’ + d’)

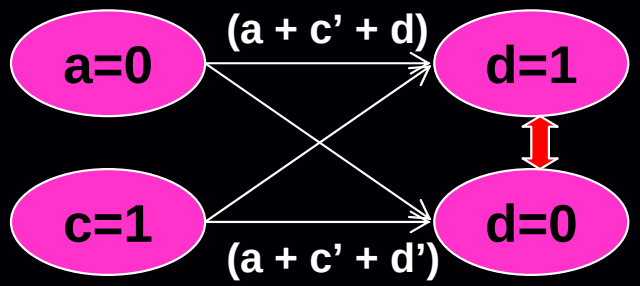
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a' + b + c)

(a + c + d)

(a + c + d')

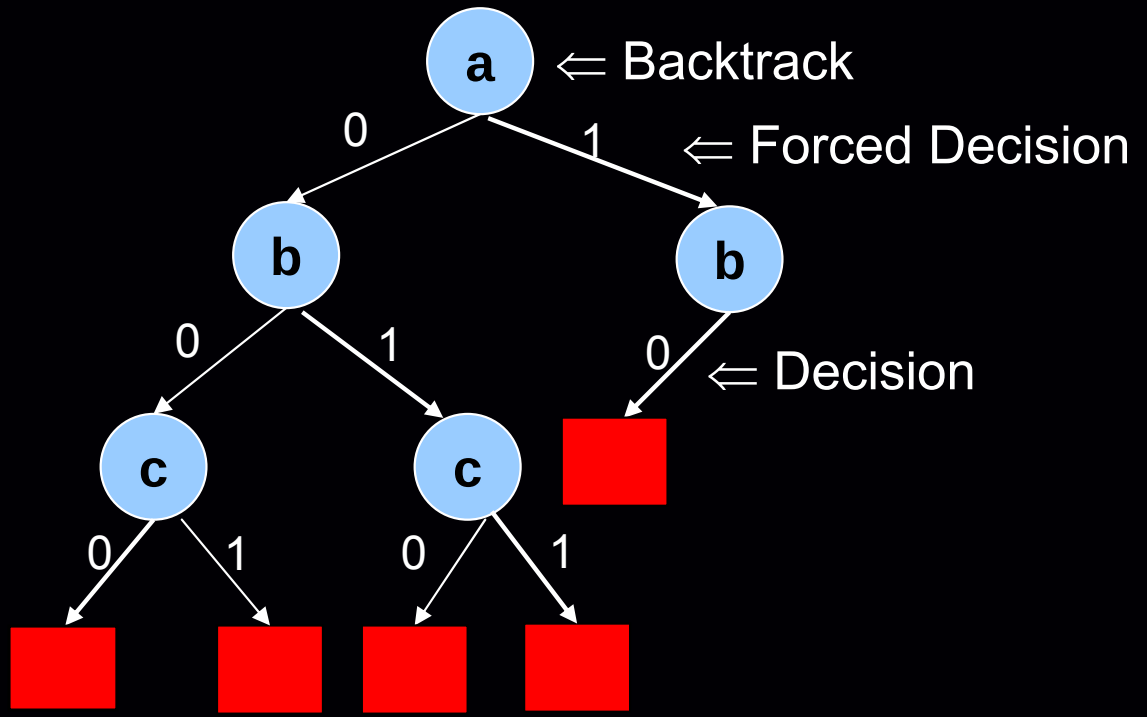
(a + c' + d)

(a + c' + d')

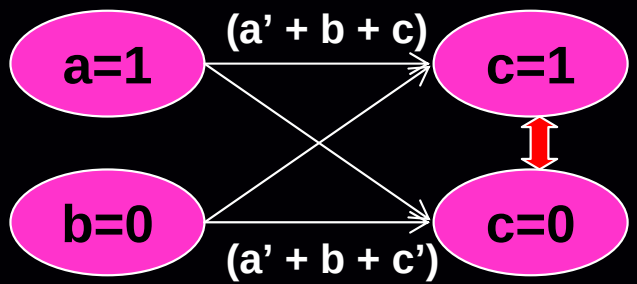
(b' + c' + d)

(a' + b + c')

(a' + b' + c)



Implication Graph



Conflict!

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

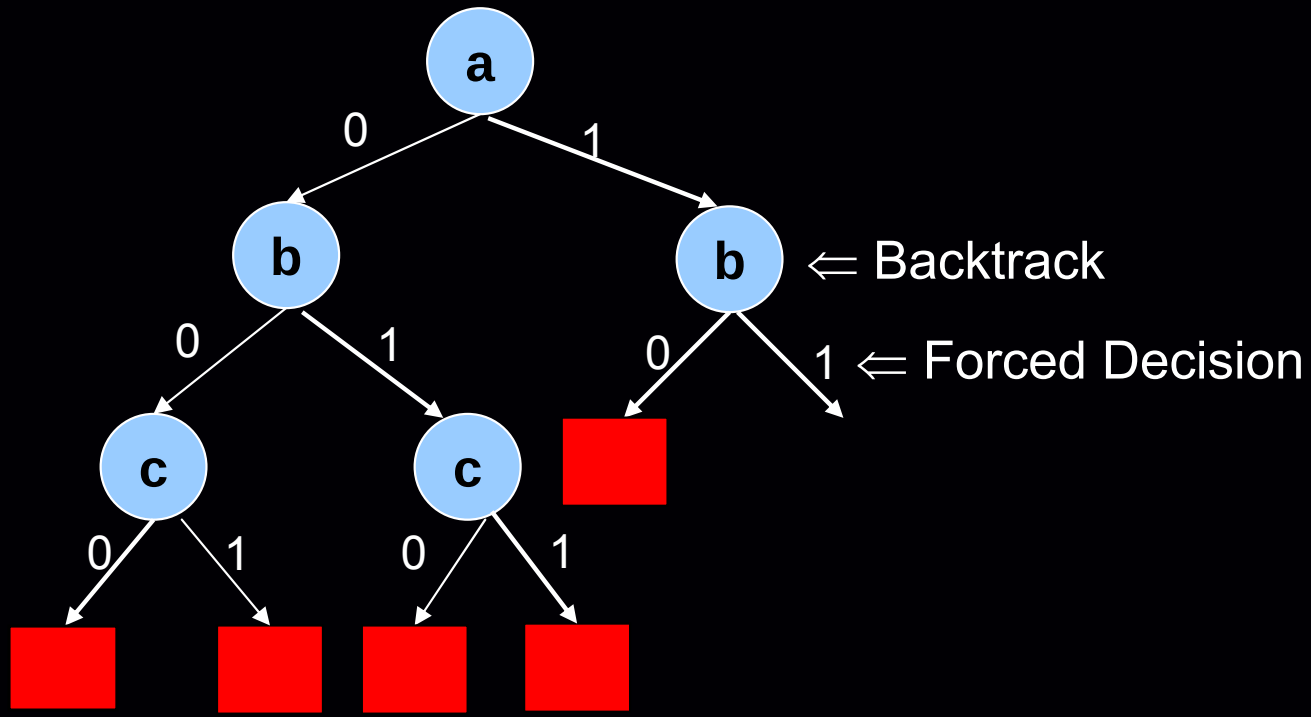
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

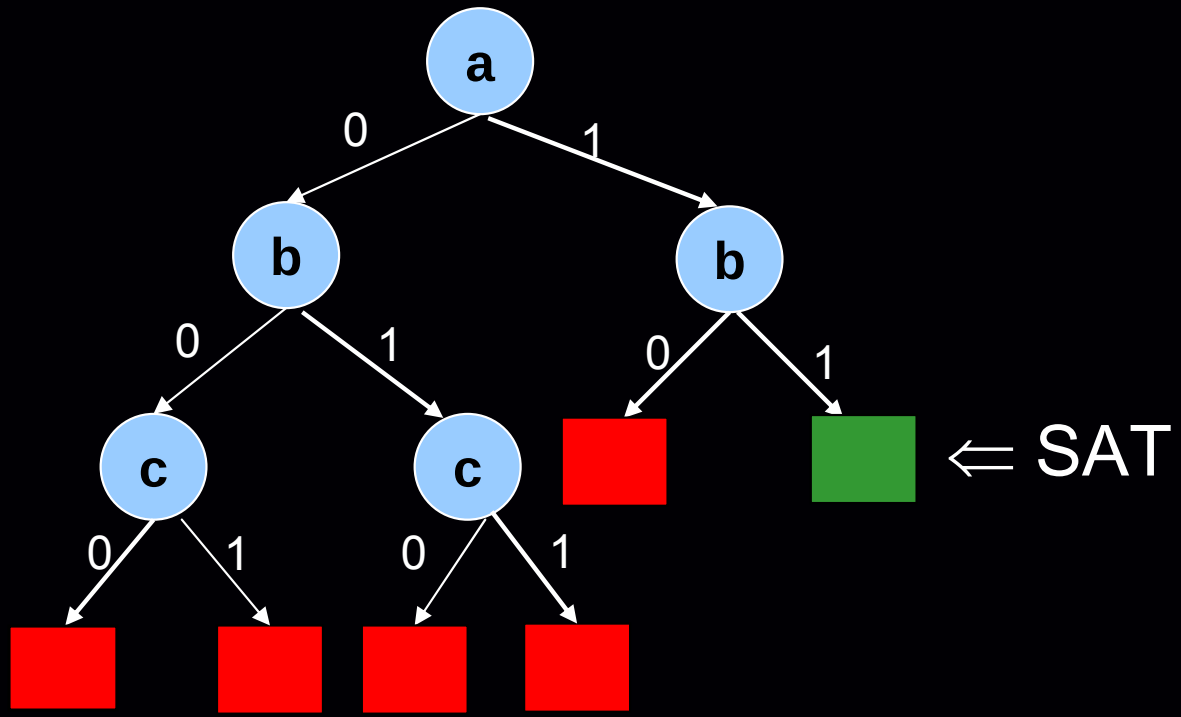
(a + c’ + d)

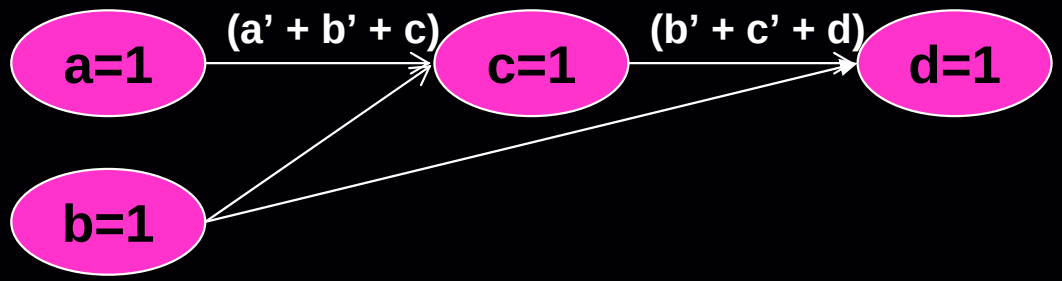
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)





Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Potentially exponential complexity!!

Did you see any unnecessary work?

SAT Improvements

1. Conflict-driven learning
   - Once we encounter a conflict
     ➔ Figure out the cause(s) of this conflict and prevent to see this conflict again!!

Conflict-Driven Learning

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

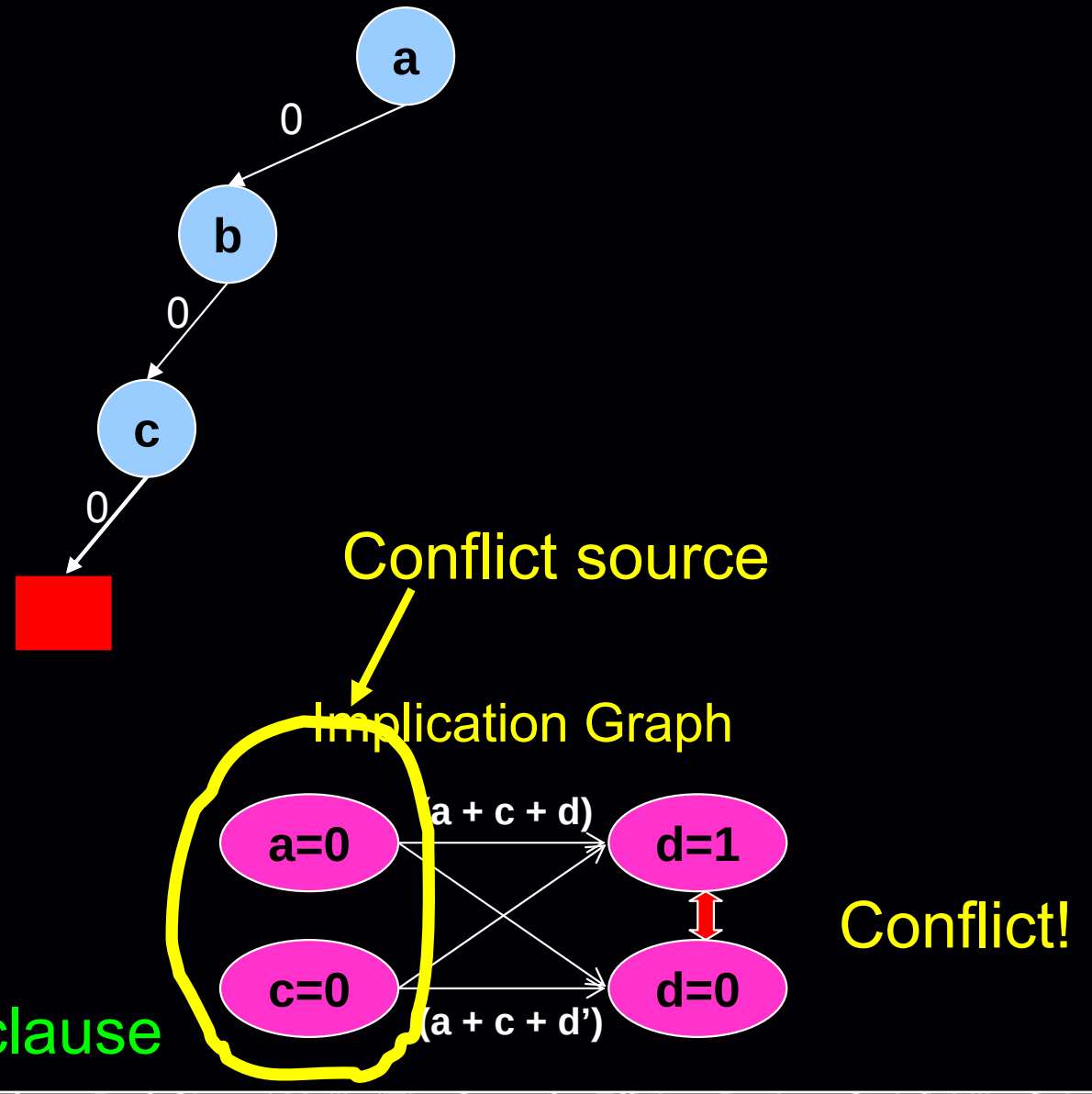
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c)

Learned clause



SAT Improvements

2. Non-chronological backtracking
   - Since we get a learned clause from the conflict analysis…

     ➔ Instead of backtracking 1 decision at a time, backtrack to the "next-to-the-last" variable in the learned clause

Non-Chronological Backtracking

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

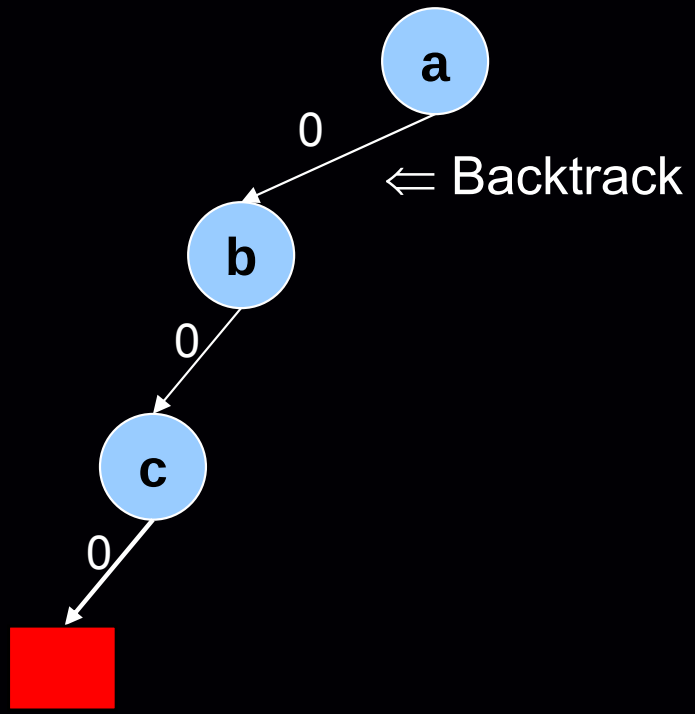
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c) Learned clause



- ‘a’ is the next-to-the-last variable in the learned clause
- Backtrack c = 0 && b = 0

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

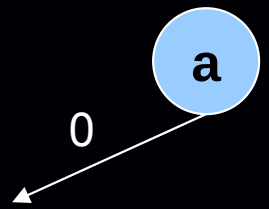
(a + c’ + d’)

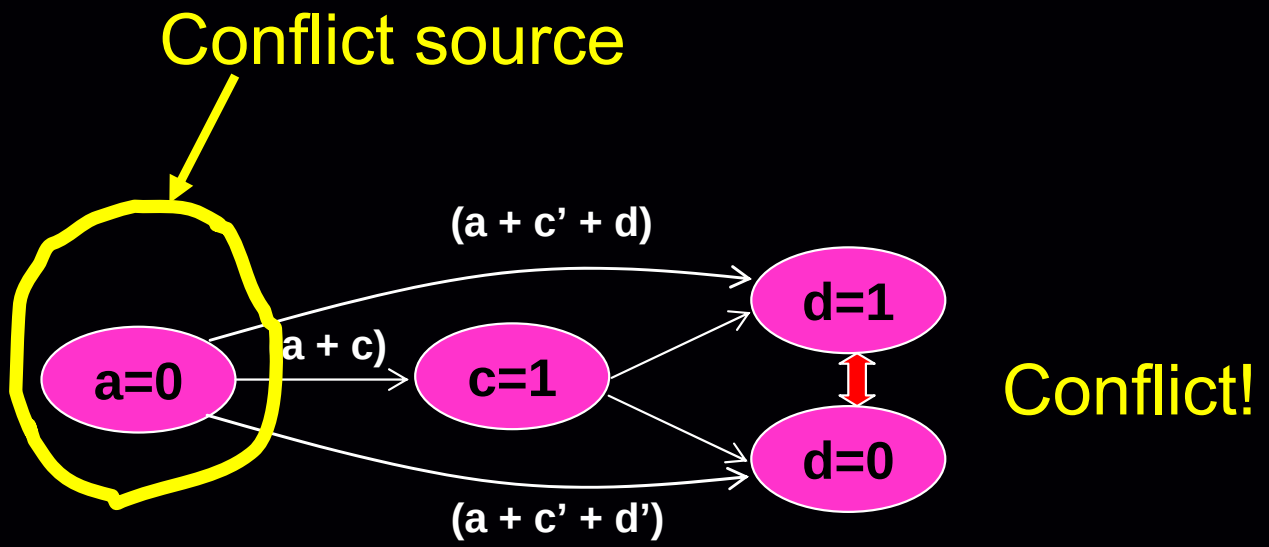
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c)





Deduced Implication from Learned Clause

(a' + b + c)

(a + c + d)

(a + c + d')

(a + c' + d)

(a + c' + d')

(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c)

(a) Learned clause

- Since there is only one variable in the learned clause

  → No one is the next-to-the-last variable

- Backtrack all decisions

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

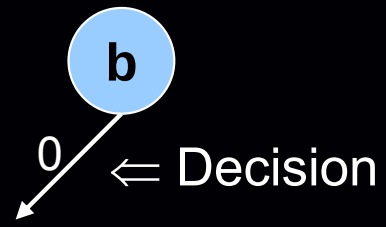
(b’ + c’ + d)

(a’ + b + c’)

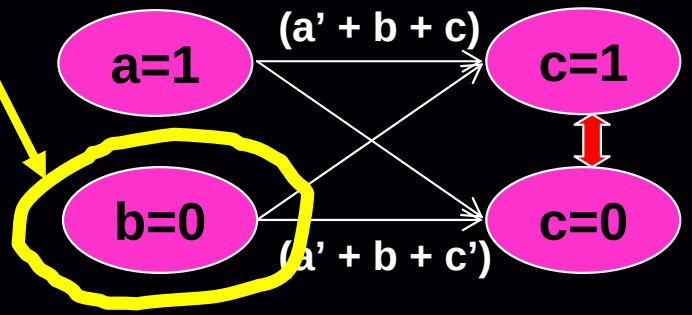
(a’ + b’ + c)

(a + c)

(a)



Conflict source



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

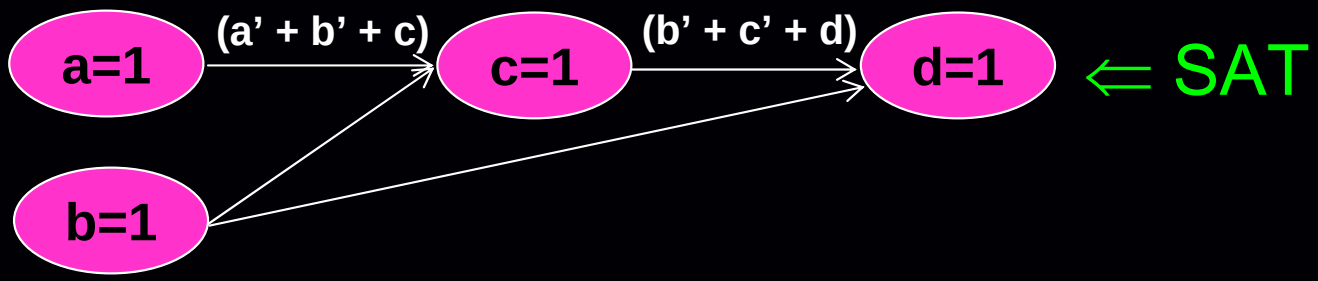
(b’ + c’ + d)

(a’ + b + c’)

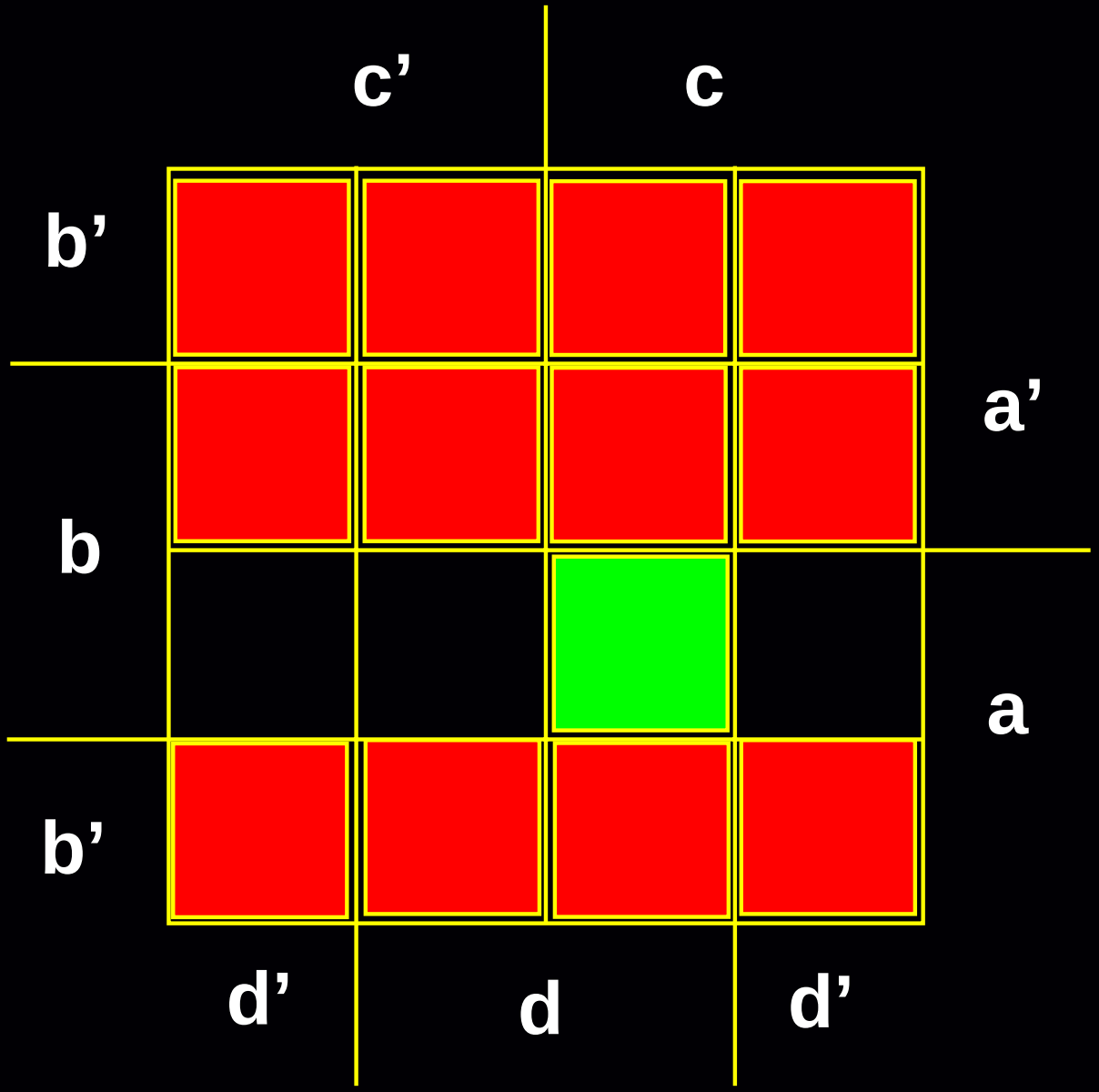
(a’ + b’ + c)

(a + c)

(a) (b) Learned clause



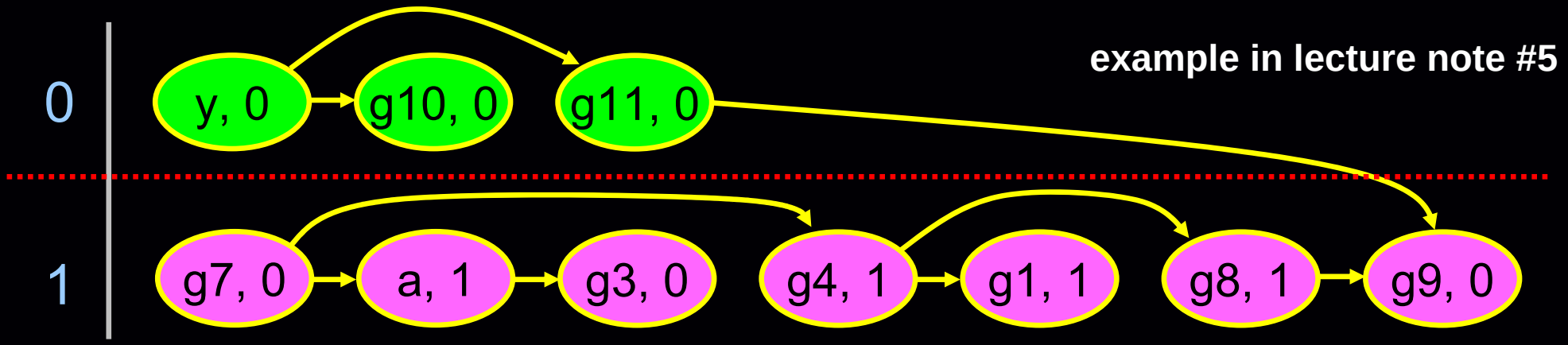
What does conflict learning tell us?



Decision: a = 0
Decision: b = 0
Decision: c = 0
conflict!!
Learned: (a + c)
Backtrack: c = 0, b = 0
Implied: c = 1
Decision: b = 0
conflict!!
Learned: (a)
Implied: a = 1
Decision: b = 0
conflict!!
Learned: (b)
Implied: b = 1
Implied: c = 1, d = 1
SAT!!

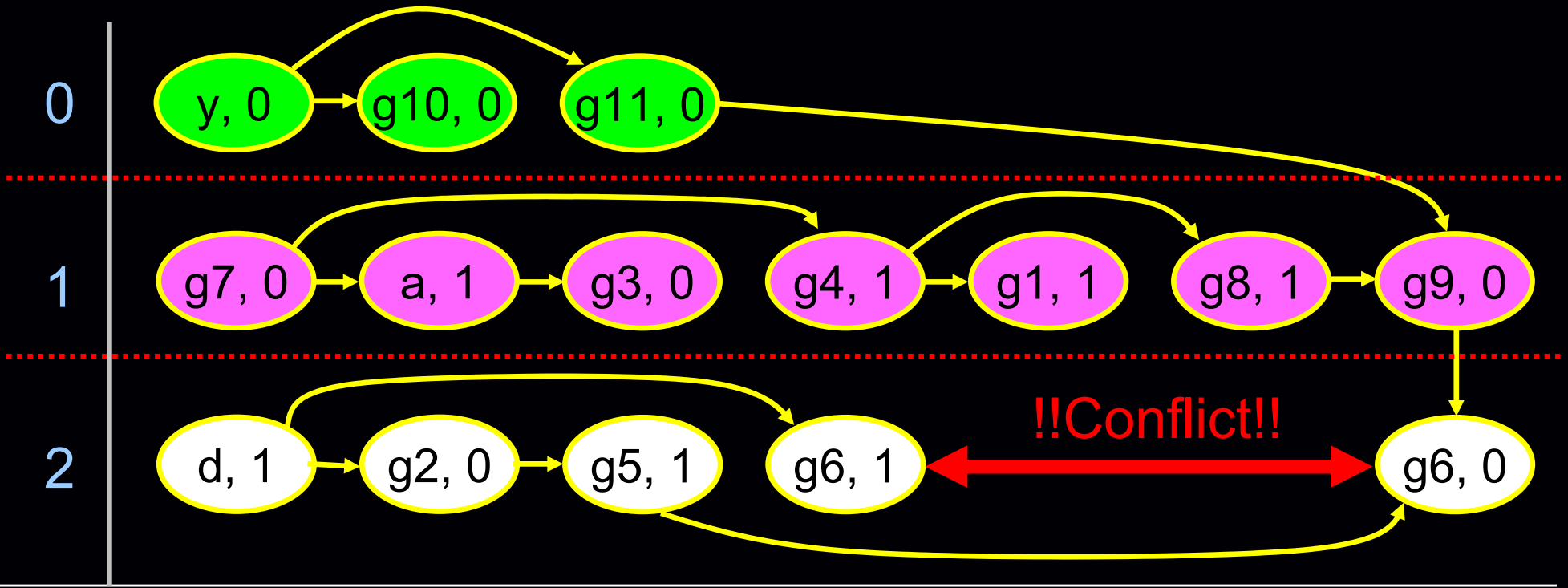
A Closer Look at the Implication Graph (a conceptual implementation)

- Implications are grouped into different decision levels
  - Level 0: target imp; constants
  - Level 1+: decisions
- Node (gate, value): implications
- Incoming edge(s) of a node: implication sources (reasons)
  - The nodes with no incoming edges are called “root implication nodes”
  - There should only be ONE root implication node for each decieion level >= 1 (which is the decision in that level)

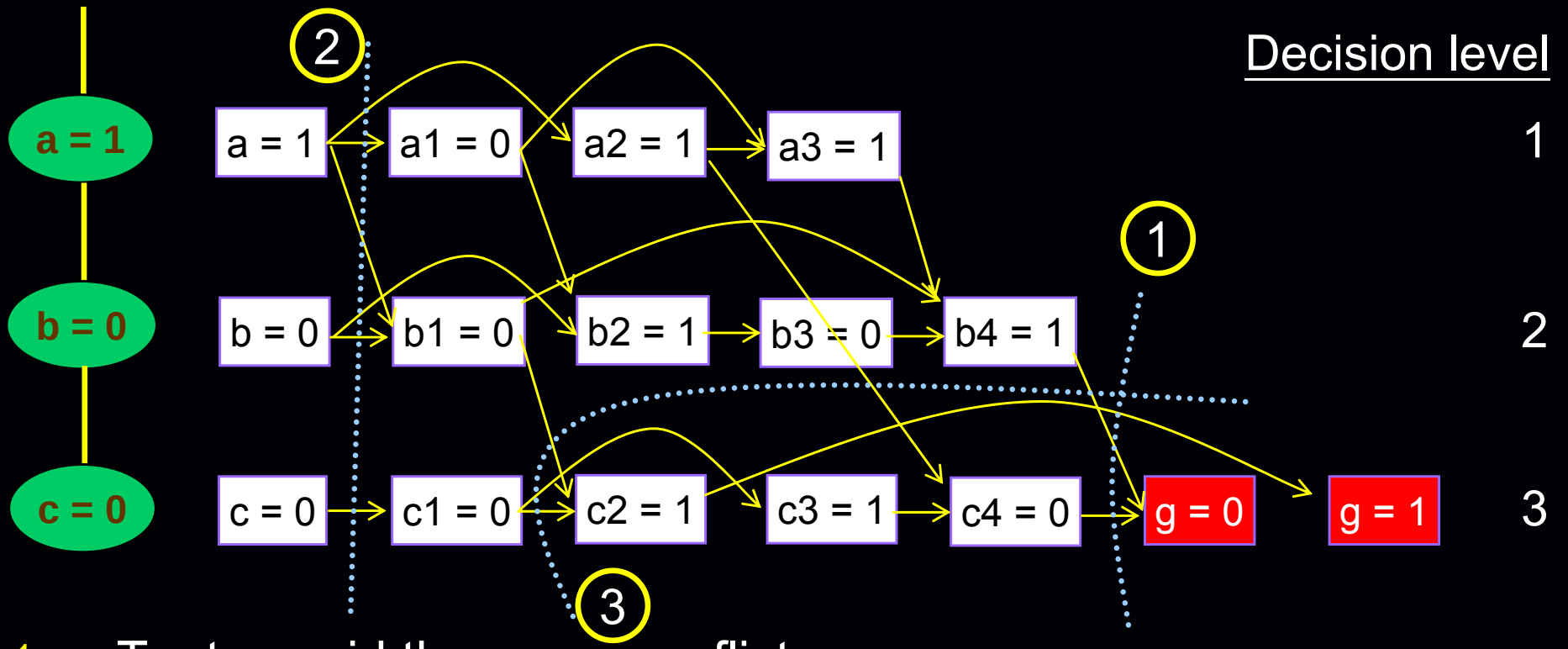


Conflict Analysis

- When we encounter a decision conflict, we want to figure out the causes so that ---
  1. Try to avoid the same conflict
  2. Backtrack as many decisions as possible

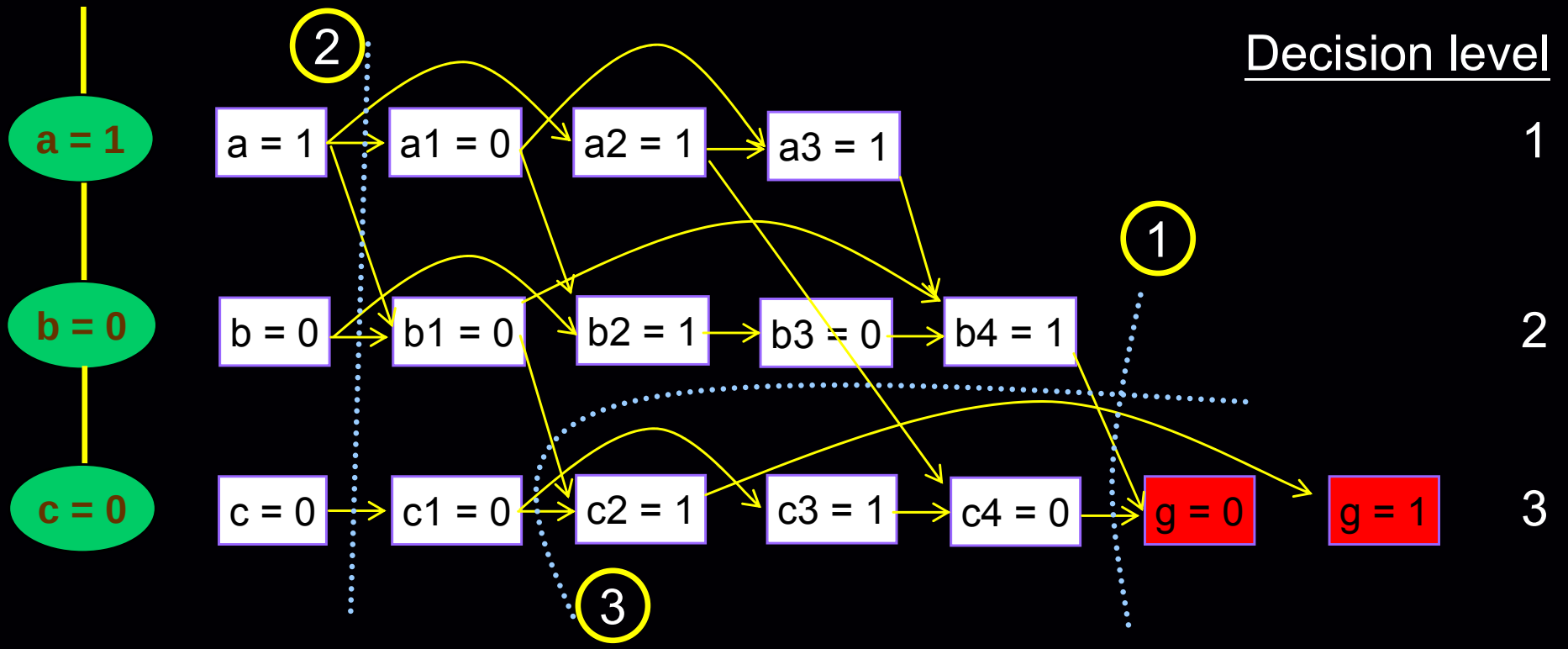


Conflict Analysis



1. Try to avoid the same conflict
   - Starting from the conflict implications (g = 0) & (g = 1), backward trace their implication sources
   - (An informal explanation) Any cut in the implication graph defines a set of conflict causes
   - Add a constraint for the conflict causes to prevent the conflict from happening again

Conflict-Driven Learning



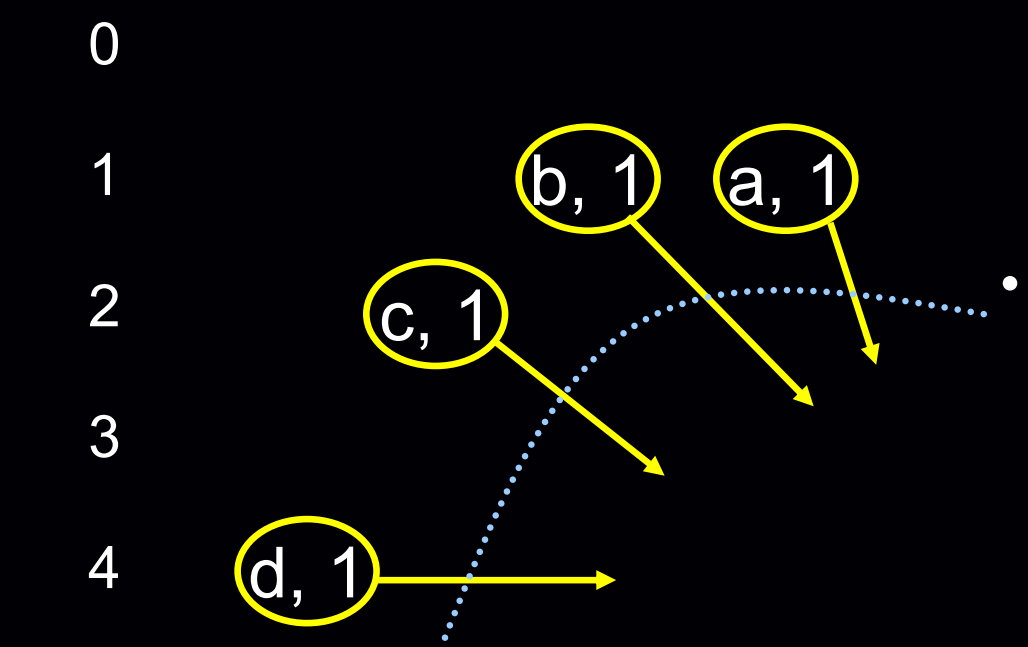
- Add a constraint to prevent the same conflict
  1. b4 && c2 && c4' = 0; ➔ (b4' + c2' + c4)
  2. a && b' && c' = 0; ➔ (a' + b + c)
  3. b4 && a2 && b1' && c1' = 0; ➔ (b4' + a2' + b1 + c1)

Which constraint is the best to add?

- [Zhang, *et al*, ICCAD 2001] Experiment shows that “first-UIP” (1st-UIP) is the best
  - UIP: Unique Implication Point
    - In a cut that there is only one node in the last (where conflict happens) decision level (why UIP cut?)
    - Starting from the conflict gate, the first encountered UIP is namely first UIP
    - The cut with only decision nodes is called the last-UIP
      - In the previous example, (2) is the last UIP, and (3) is the first UIP

UIP for Non-chronological Backtracking

- Since in UIP cut there is only one node with the last decision level…
- And we add a constraint for the UIP cut



Constraint

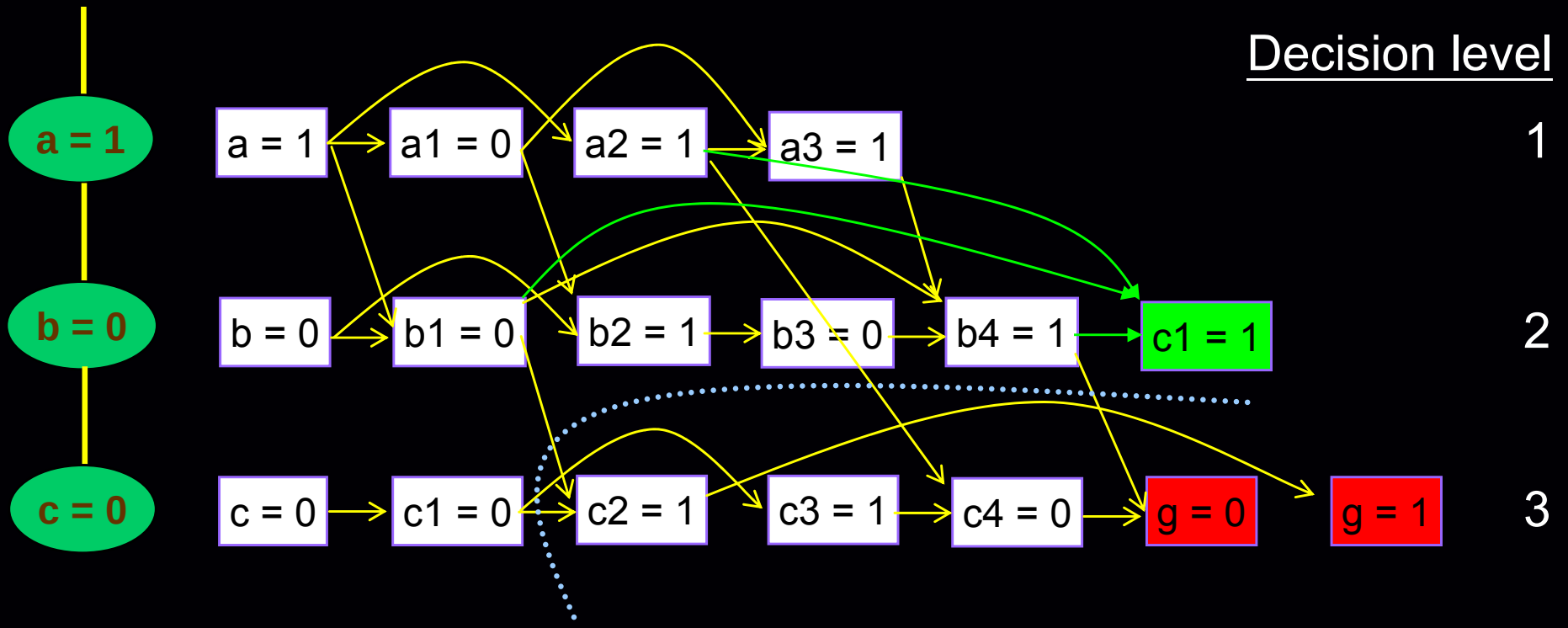
(a && b && c && d) = 0

(a && b && c) ➔ d'

If we backtrack to the max decision level of { a, b, c }

1. { a, b, c } still have the original implications
2. d can be implied with the opposite value at the max level above

Conflict-Driven Learning



Learned implication

b1' && a2 && b4 ➔ c1

Conflict-Driven Non-Chronological Backtracking --- Algorithm

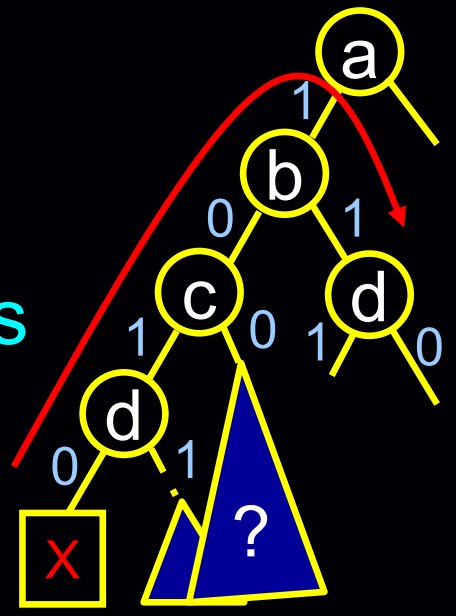
1. When conflict occurs, check if the conflict level == 0 (implication level for the SAT target)
   a) If yes, return *unsatisfiability* (Why?)
   b) Else, continue to 2
2. Find the 1st-UIP cut as the conflict causes
3. Backtrack to the max decision level of the nodes other than UIP in the cut
4. The UIP gate will be implied with the opposite value
5. Perform the new implication
6. If conflict, go to 1, else continue for the next decision

A closer look at binary decision tree

In general, is non-chronological backtracking safe?

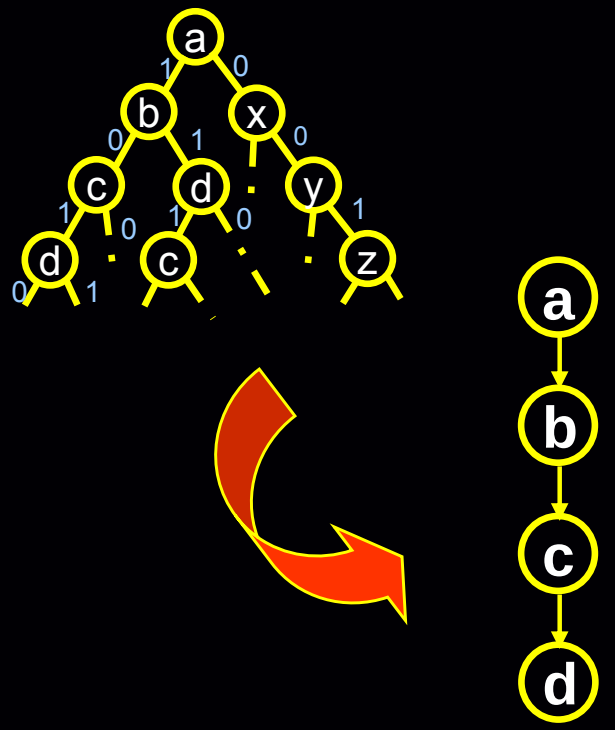
- May lead to SAT solution ealier
- But some portion of the decision tree may not be covered
  - Not a complete search anymore
  - May also miss some bugs

➔ Difficult to record which branches haven't been searched



Conflict-Driven Non-Chronological Backtracking --- Completeness

- But with conflict-driven learning, SAT search is still guaranteed to be complete
- SAT search is not a binary decision tree anymore…
  - Becomes a decision stack
  - Conflict
    - → Learned clause (gate)
    - → Indicate where to backtrack
    - → Learned implication



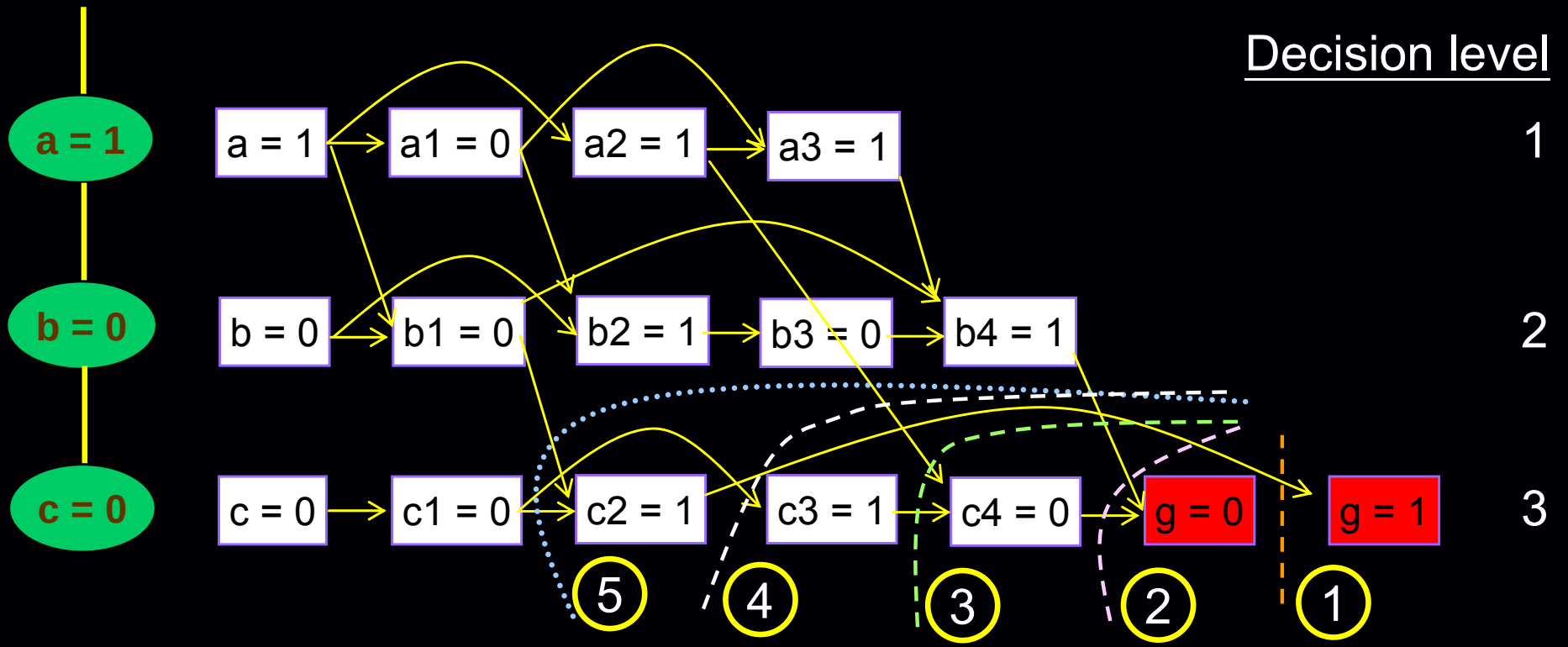
Conflict-Driven Non-Chronological Backtracking --- Completeness

- Branch-and-bound algorithm for Constraint Satisfaction Problem (CSP) becomes a "constraint refinement process"

➔ Search region is gradually narrowed down

➔ At the end, either becomes empty, or finds the solution !!

Implication graph, resolution, and learning



(1): (c2' + g)
(2): (b4' + c4 + g') → (b4' + c2' + c4)
(3): (a2' + c3' + c4') → (a2' + b4' + c2' + c3')
(4): (c1 + c3) → (a2' + b4' + c1 + c2')
(5): (b1 + c1 + c2) → (a2' + b1 + b4' + c1)

What we have learned on SAT...

1. Efficient logic implication using watches
2. Conflict-driven learning for non-chronological backtracking

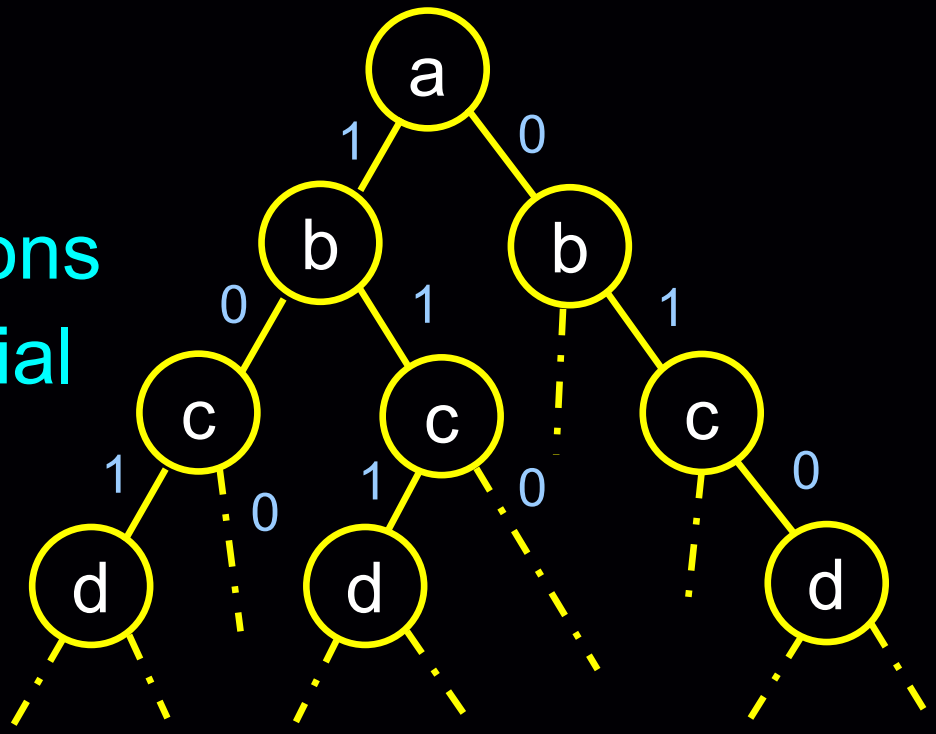
- More heuristic...

3. Decision ordering / Restart
4. Various learning techniques

Impact of Decision Ordering

- Decision ordering: the order of gates that the corresponding decisions are made
  1. Order of gates
  2. Decision values

  ➔ Good and bad decisions can lead to exponential difference (e.g. 2^{10} vs. 2^{50})



- (Think) Does the decision value matter? (i.e. should we decide on '1' or '0' first?)

Static Decision Ordering

- Decision order and values are pre-computed in the beginning and remain unchanged

1. Topological
   - Depth-first
   - Breadth-first
   - Guided by gate types
2. Probability-based
   - Controllability / Observability
   - Signal probability
   - (Weighted) Random
3. Influence-based
   - Literal count
   - #fanins / #fanouts
   - Influence of implications

Dynamic Decision Ordering

- Decision order and values are dynamically determined based on current implication values, justification frontier, etc.
  - Use similar criteria as static method
  - But can mix different rules dynamically
- Pros
  - May lead to better decisions
  - Avoid useless decisions
- Cons
  - Overhead in computing dynamic ordering may be high
  - Effectiveness sometimes is hard to predict

➔ However, experiences show that the best is:

1. Has a good initial decision ordering
2. Adaptively adjust the decision order after a certain amount of backtracks

zChaff's Variable State Independent Decaying Sum (VSIDS) Decision Heuristic

(1) Each variable in each polarity has a counter, initialized to 0.

(2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.

(3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.

(4) Ties are broken randomly by default, although this is configurable

(5) Periodically, all the counters are divided by a constant.

Zhang, et al, DAC 2001

Berkmin – Decision Making Heuristics

E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver", *Proc. DATE* 2002, pp. 142-149.

- Identify the most recently learned clause which is unsatisfied
- Pick most active variable in this clause to branch on
- Variable activities
  - updated during conflict analysis
  - decay periodically
- If all learnt conflict clauses are satisfied, choose variable using a global heuristic
- Increased emphasis on "locality" of decisions

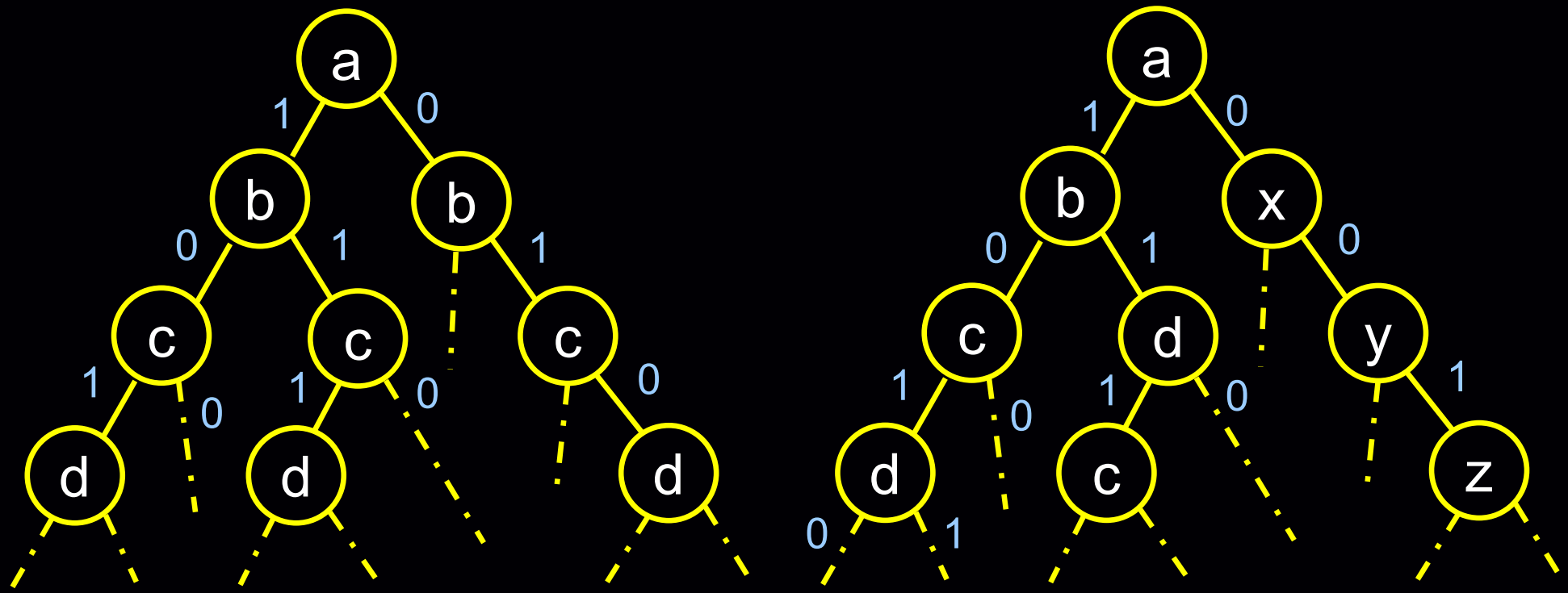
More decision heuristics...

- Variable Move-To-Front (VMTF)
- Clause Based Heuristic (CBH)
- Resolution Based Scoring (RBS)
- ...

- In general, there is no single decision heuristic that works for every case.
  - ➔ How to adaptively move to a good decision heuristic may be the winner...

A closer look at binary decision tree

Should the decision orderings on all branches be the same?



Remember when we talked about conflict-driven learning,

we mentioned that

by adding a learned clause

we can do non-chronological backtracking,

while still achieve complete proof

How??

The Constraint Refinement Process

- Search region is gradually narrowed down by the learned constraints
- Learned information is universally true
  - Independent of the target implication, only related to the circuit function
  - The proof efforts between different properties can be shared

    ➔ Incremental SAT
- Decision process can “restart” any time any where!!
  - Can use different decision ordering to explore different area in the decision tree
    - Previous efforts will not be wasted

Various Learning Techniques

- Other than conflict-driven learning, there are many other learning techniques that can help
  - Derive more implications
    → may help find the conflict earlier
  - Provide information for decision ordering

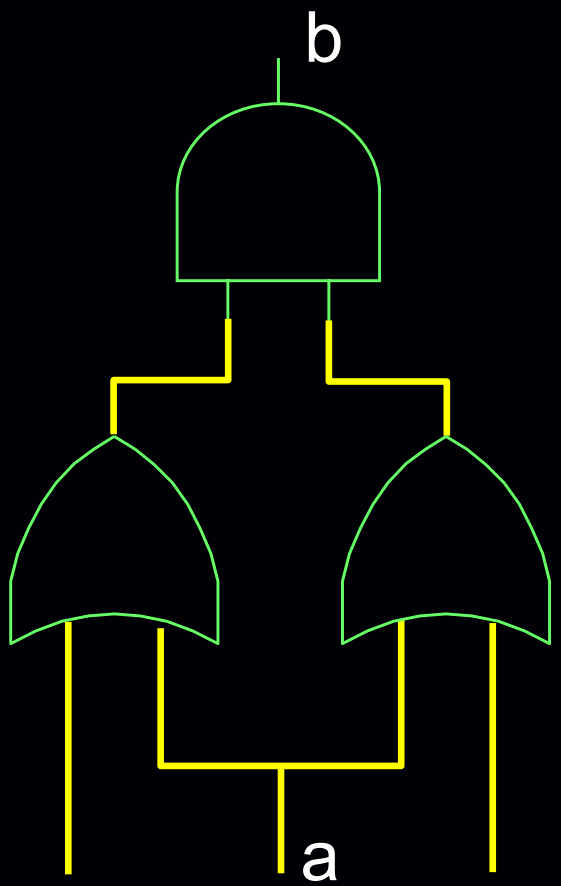
1. Static learning
2. By signal correlations
3. Recursive learning
4. Success-driven learning

Static Learning

- Learn by contrapositive

  (a → b ≡ !b → !a)

- e.g.



a = 1 → b = 1

Learned b = 0 → a = 0

The question is:
which gate to learn??

Ref: "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", Schulz *et.al*, TCAD 1988

Learned by Signal Correlations

- A proof-based approach
  - Since learned information is universally true, we can create some internal interesting properties, and use these properties to derive some interesting learning (by conflict analysis)

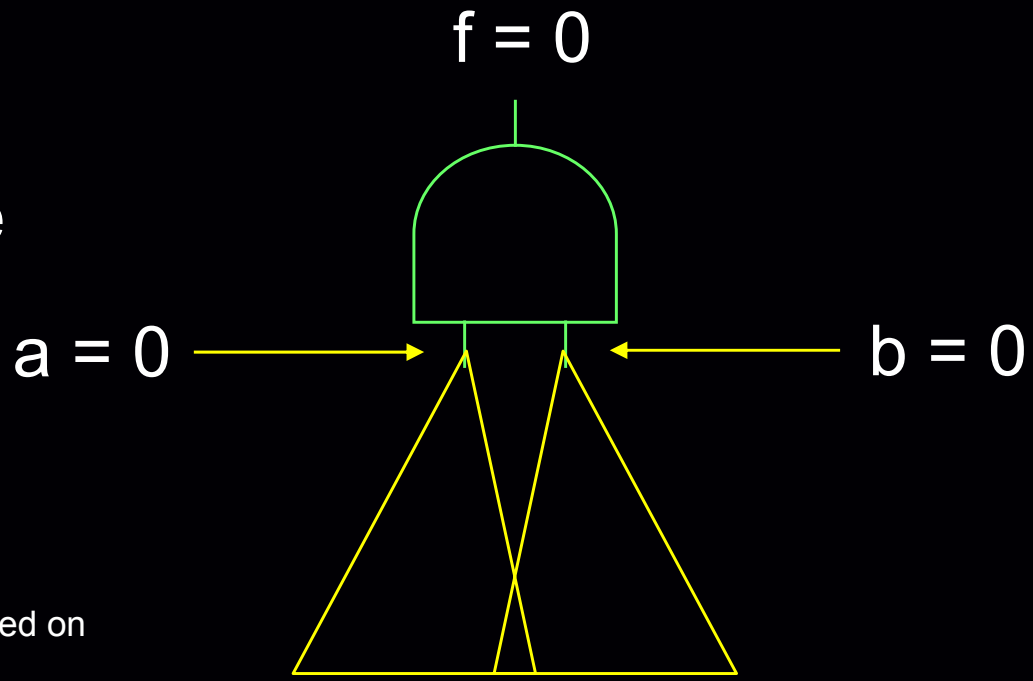
- e.g. By simulation, if we find a gate ‘g’ is very likely to stuck at some value ‘v’
  - ➔ Witness “g = ¬v” (should produce many conflicts)
- e.g. By simulation, if two signals respond almost the same
  - ➔ Witness “p != q”

- No matter the proof is finished or not
  - We can always learn something

Ref: Feng Lu, *et. al*, “A Circuit SAT Solver with Signal Correlation Guided Learning”, DATE 2003

Recursive Learning

- To justify f = 0
  - (a = 0) or (b = 0)
  - Let S_a and S_b be the set of implications from (a = 0) and (b = 0), respectively
  - Let $S = S_a \cap S_b$
    ➔ (f = 0) implies S
- A recursive process
- Deep recursion could be very expensive



Ref: “HANNIBAL: an efficient tool for logic verification based on recursive learning”, Wolfgang Kunz, ICCAD 1993

Conflict vs. Success-Driven Learning

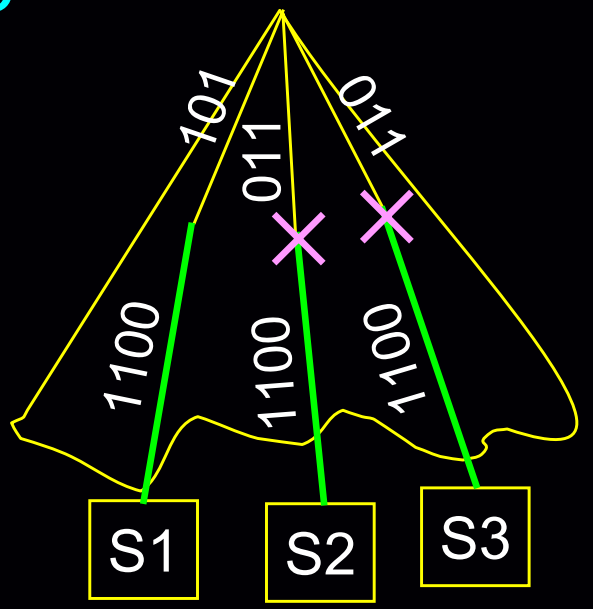
Motivation: Traditional SAT approach finds only 1 solution, can we find more (or all) the solutions?

- How to record the solutions?
  - Hash table? (too expensive)
- Success-driven learning
  - Similar to conflict learning
  - When we find one solution, say (v1, v2, …, vn), add a blocking gate "v1 && v2 && … vn = 0" so that
    - This solution won't be repeated
    - May lead to new implication
    - Can continue the justification process for the next solution
  - At the end, all the solutions are recorded as set of blocking gates (or clauses)

Conflict vs. Success-Driven Learning

- However, the number of solutions in a SAT problem can be very huge!!
  - ➔ Some solutions may look alike ---
    e.g. 1010011, 1100011, 0110011...
    s1 s2 s3

Can we predict that the sub-solutions under the sub-search tree are already covered?

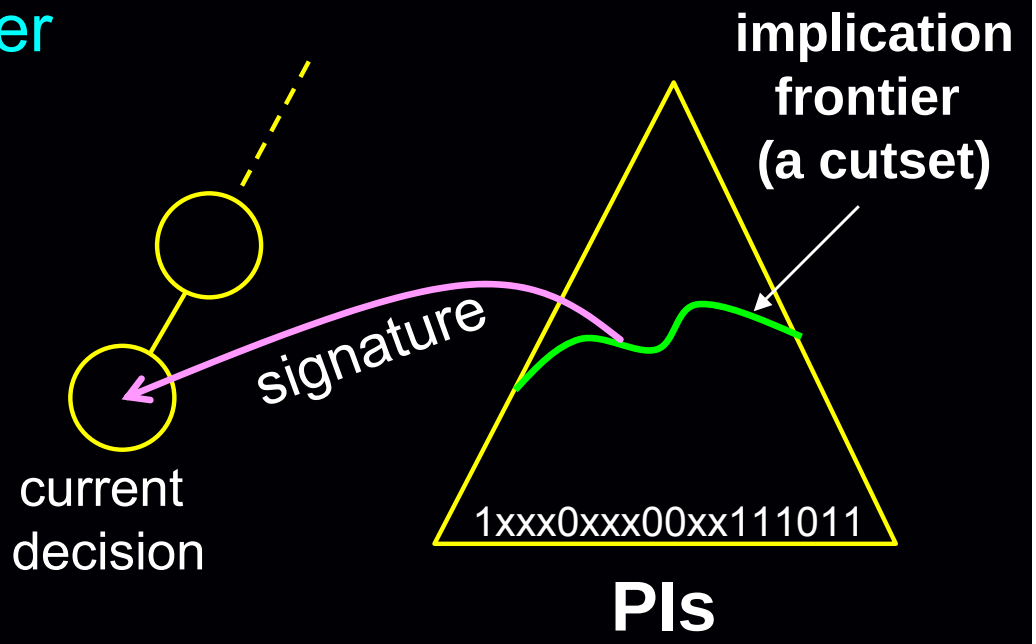


Success-Driven Learning

- Ref: Shuo, et al. DATE 2003
- Assume
  - ATPG-based technique (work on circuit)
  - Decisions on PIs only → forward implications
- Search State Equivalence
  - If two decisions have the same signature

  → The "sub-solutions" under the sub-search space are the same!!

  → No need to search
- Note: they also store the solutions in a "free BDD"



Although “learning” in general can lead to more implications and possibly lead to conflicts earlier (i.e. bound earlier) ---

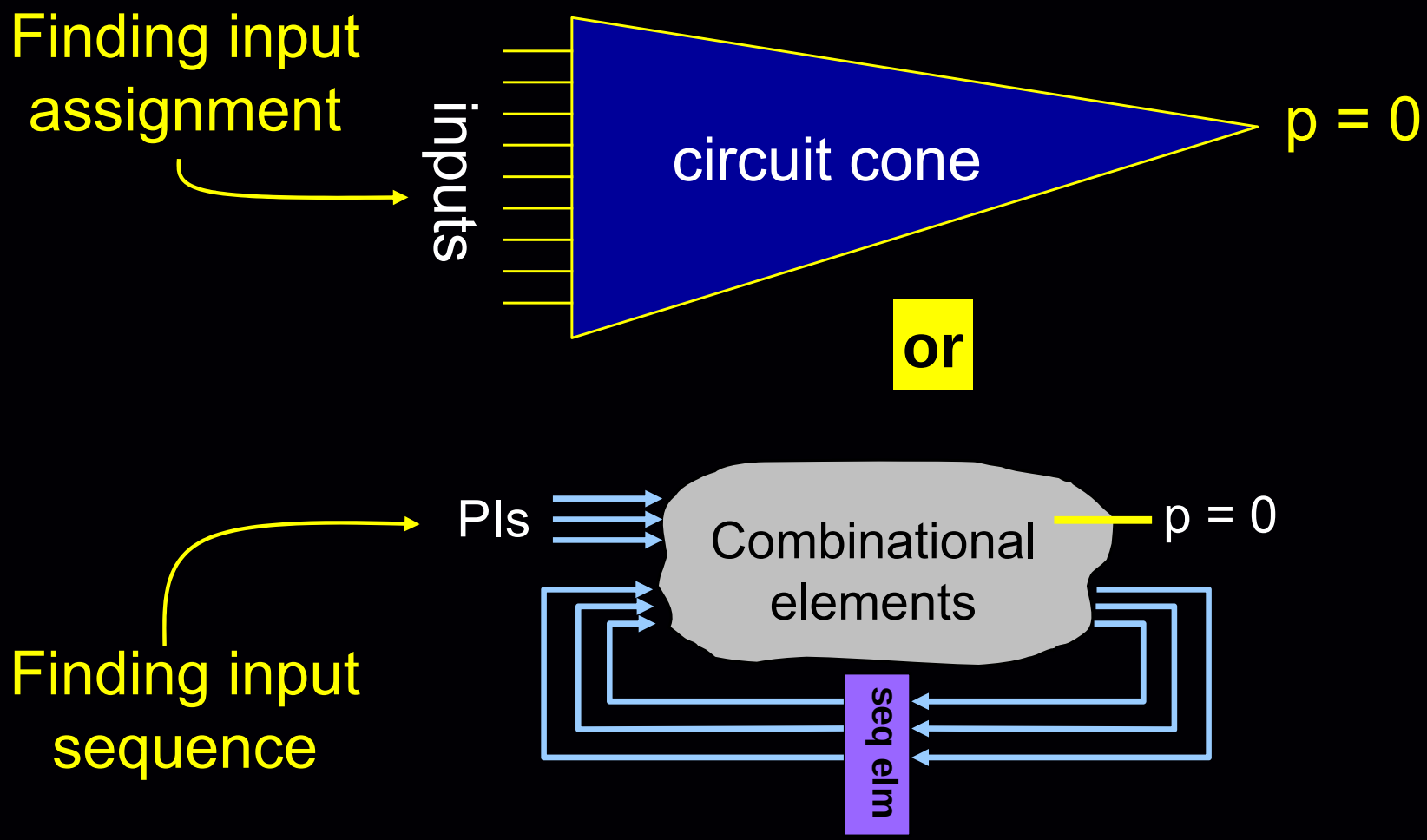
1. It may slow down the implication process
2. It may affect the decision ordering, which may not necessarily reduce the #decisions

What can we do to make the learning useful?

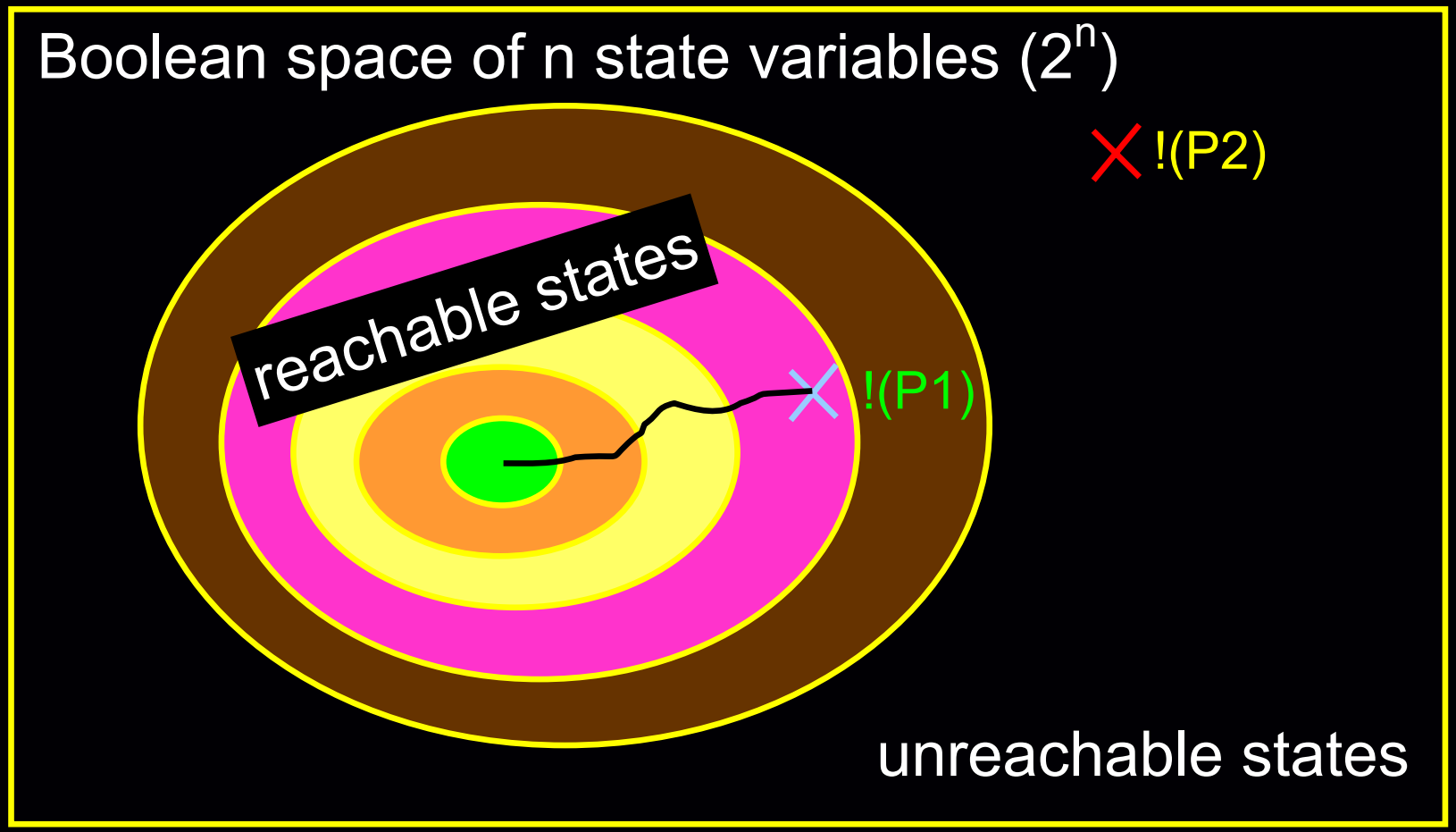
1. Use learning to find better decision ordering
   - zChaff uses learned information to refine the decision ordering
   - BerkMin uses learned information to increase emphasis on “locality” of decisions
2. With conflict analysis, decision can restart any time
   - Change to different decision ordering heuristic to explore different areas in the input space
3. Modify the learned information
   - Remove least-used learned information
   - Simplify or synthesize the learned information
   - Any other idea?

Outline

Recalled, the SAT algorithms we have covered so far are good for "combinational invariance" checking. In reality, most assertions are "sequential"



Recalled: Reachability Analysis vs. Assertion Checking

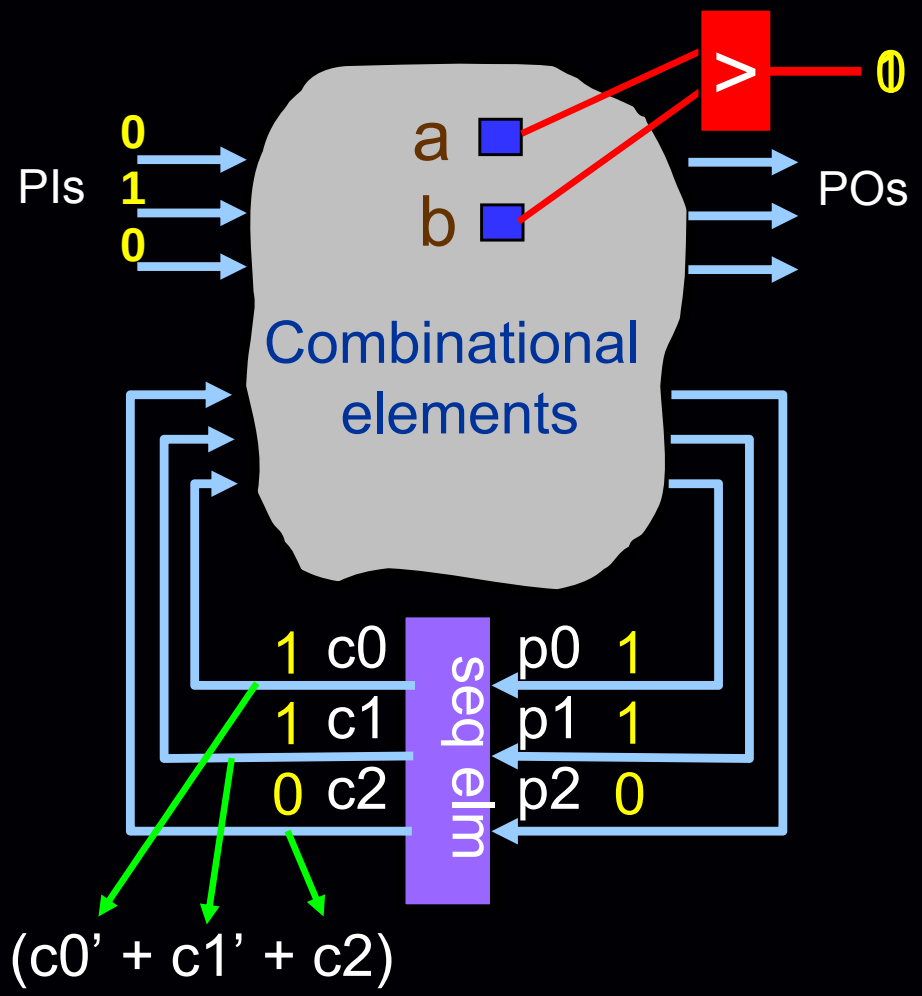


➔ AG (p1) ≡ false; AG (p2) ≡ true

SAT-Based Verification

- We know that SAT engine can work on the "assignment problem"
  - Find an assignment for a set of constraints
  - To prove that the counter-example of an combinational invariance does not exit
- How can we apply SAT on sequential assertion checking?
  - We prove that counter-example does not exist in one timeframe, not in two timeframes, three, four, ... to infinity?
  - How can we assure that ALL the reachable states have been reached?

Using Blocking Clauses for Sequential SAT



Suppose we are solving the property "a > b"

1. Use SAT to get a solution on the registers
   e.g. (c0, c1, c2) = (1,1, 0)
2. Add a "blocking clause" (c0' + c1' + c2) to the original CNF
   ➔ Won't get the same state again!!
3. Repeat 2 for another solution..., or
4. Apply the solution "(c0, c1, c2) = (1, 1, 0)" to the previous state as "(p2, p1, p0) = (1, 1, 0)" and continue to the search in the previous timeframe

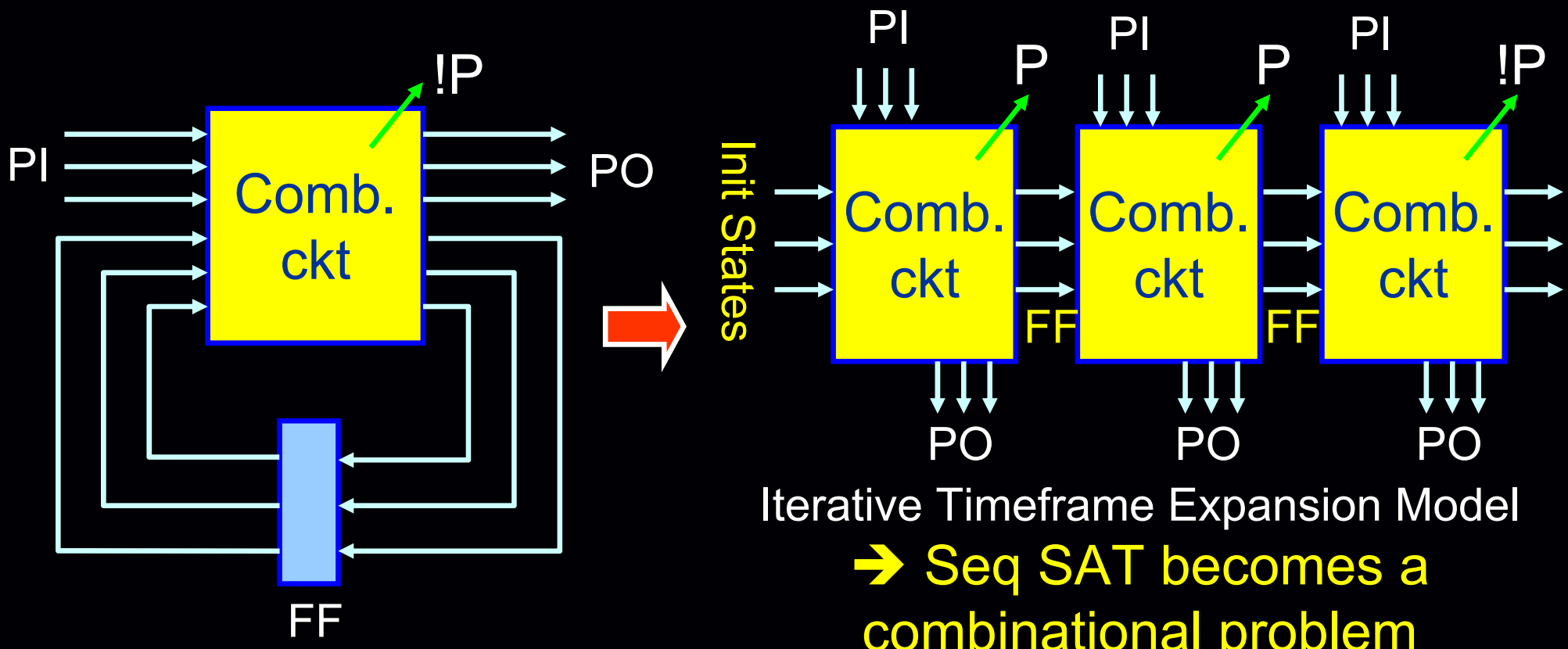
- However, in the above approach, we are solving one state (cube) at a time.

  The number of states is exponential to the number of registers...

  SAT seems inefficient...

Bounded Model Checking

- There's another way of using SAT for sequential property checking



Iterative Timeframe Expansion Model

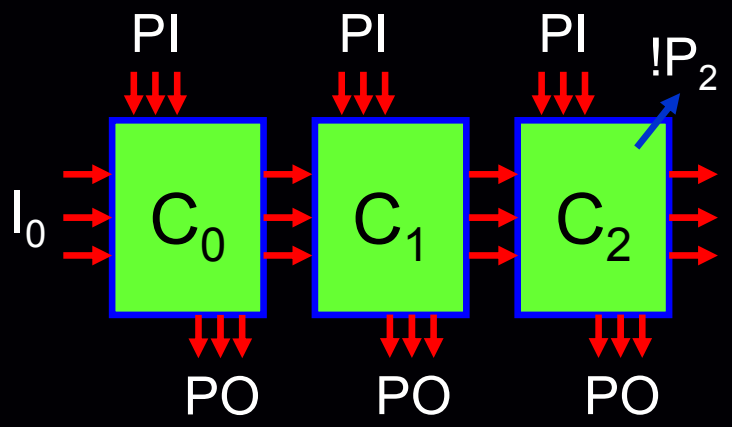
➔ Seq SAT becomes a combinational problem

Bounded Model Checking (BMC) Algorithm

- Let 'C' be the set of constraints on the combinational circuit

  ➔ For an iterative model that unfolds the circuit for n times, let 'C_i' correspond to the i-th iteration of the circuit constraint (0 <= i <= n - 1)

- Let 'I_0' be the initial state value
- Let 'P' be the property to prove



- **BMC(P) {**

```
    let k = 1;
  loop:
    if (SAT(I0∧C0∧...∧Ck-1∧!Pk-1))
      return "Find a counter-example @ (K-1)";
    k = k + 1;
    goto loop;
}
```

How far should we go?

- What's the limit of K?

  (How many iterations do we need before concluding the property is always true?)

  ➔ Impossible to know in the above BMC algorithm

  ➔ A loose upper bound is 2^N (N is the number of registers)

Application of BMC

- If the property is false, BMC can find a counter-example with the shortest length
- However, if the property is true, BMC cannot produce any conclusive result...

(BMC is best used in “bug-finding”)

Extension of BMC for Unbounded Proof

- BMC, combined with various techniques, can be extended to unbounded model checking

1. K-step Induction
2. Simple-path constraint

(Covered in next topic)

3. Counter-example-based abstraction
4. Proof-based abstraction
5. Image computation by SAT
6. Over-approximated image computation using interpolation

etc...

K-induction

SSS2000

- Induction:

$$\frac{P(s_0) \quad \forall i: P(s_i) \Rightarrow P(s_{i+1})}{\forall i: P(s_i)}$$

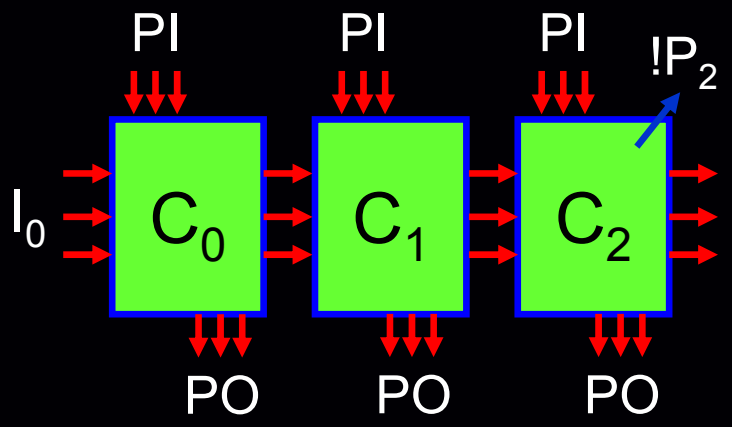
- k-step induction:

$$\frac{P(s_{0..k-1}) \quad \forall i: P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

K-induction with a SAT solver

- Let:

$$U_k = C_0 \wedge C_1 \wedge \ldots \wedge C_k$$



- Two formulas to check:
  - Base case:

$$I_0 \wedge U_{k-1} \Rightarrow P_0 \ldots P_{k-1}$$

  - Induction step:

$$U_k \wedge P_0 \ldots P_{k-1} \Rightarrow P_k$$

- If both are valid, then P always holds.
- If not, increase k and try again.

Induction SAT

```
for (k = 0 to infinity)
    S = U_k ∧ F_k  // F_k = P_0 ∧ ... ∧ P_{k-1} ∧ !P_k
    T = I_0  ∧ S
    // induciton step
    if (SAT(S) == false)
        return NO_SOLUTION;  // i.e. P is true
    // normal proof: base case for next k
    if (SAT(T) == true)
        return HAS_SOLUTION; // i.e. CEX is found
    if (effort exceeds limit)
        return ABORT;
endfor
```

Induction SAT

- In other words, let

  $S(k) = U_k \wedge F_k$ // induction step

  $T(k) = I_0 \wedge S$ // BMC step

- Induction SAT...

```
if (S(0) == UNSAT)   return UNSAT;
if (T(0) == SAT)     return SAT;
if (S(1) == UNSAT)   return UNSAT;
if (T(1) == SAT)     return SAT;
...
```

Does “Induction SAT” guarantee convergence?

i.e. Will we either

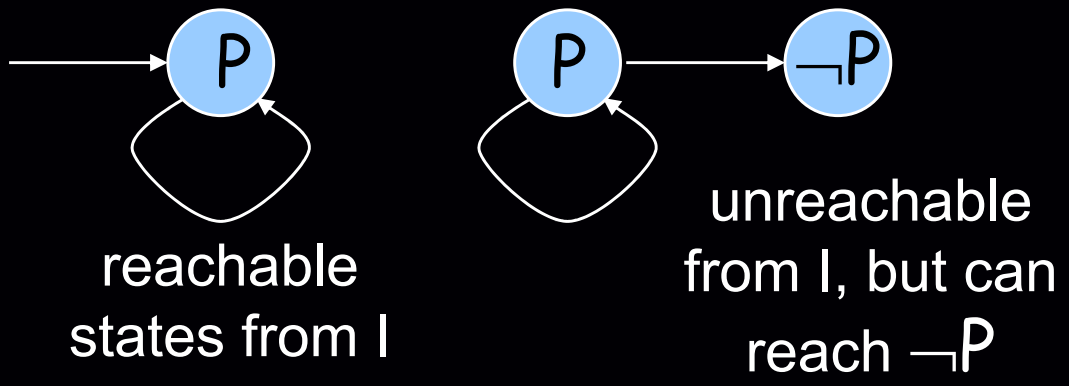
1. conclude no solution in induction step

or 2. find a counter-example in normal proof

with a finite number k ???

Simple path assumption

- Unfortunately, k-induction is not complete.
  - Some properties are not k-inductive for any k.



- Simple path restriction:
  - There is a path to ¬P iff there is a *simple* path to ¬P (path with no repeated states).

Induction over simple paths

- Let simple($s_{0..k}$) be defined as:
  - $\forall i,j \text{ in } 0..k : (i \neq j) \Rightarrow s_i \neq s_j$
- k-induction over simple paths:

$$\frac{P(s_{0..k-1}) \qquad \forall i: \text{simple}(s_{0..k}) \wedge P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

Must hold for k large enough, since a simple path cannot be unboundedly long. Length of longest simple path is called *recurrence diameter*.

...with a SAT solver

- For simple path restriction, let:

  $S_k = \forall t=0..k, t'=t+1..k: \neg (\forall v \text{ in } V : v_t = v_{t'})$

  (where V is the set of state variables).
- Two formulas to check:
  - Base case:

    $$I_0 \wedge U_{k-1} \Rightarrow P_0 ... P_{k-1}$$

  - Induction step:

    $$S_k \wedge U_k \wedge P_0 ... P_{k-1} \Rightarrow P_k$$
- If both are valid, then P always holds.
- If not, increase k and try again.

Is the recurrence diameter the same as the diameter (the distance from initial state to any state, i.e. depth of fixed point)??

Termination

- Termination condition:

  k is the length of the longest simple path of the form

  $$P^* \neg P$$

- This can be exponentially longer than the diameter.
  - example:
    - loadable mod 2^N counter where P is (count $\neq 2^N-1$)
    - diameter = 1
    - longest simple path = 2^N
- Nice special cases:
  - P is a tautology (k=0)
  - P is inductive invariant (k=1)

Outline

SAT-Based Verification

- BMC
  - Can find the shortest counter-example if it exists
  - But cannot prove invariance
- BMC + induction
  - Can prove invariance... only for cases that do not have a sequential feedback loop
- BMC + induction + simple-path constraints
  - Can guarantee convergence... but the overhead induced by the constraints can be very high
- What we are missing for SAT-based sequential proof...
  - An efficient method to “record” reachable states
  - ➔ Not a natural SAT application?

Symbolic model checking without BDD's

- There are some algorithms to compute the set of reachable states by SAT
- Fixed point characterization

  [ref: Abdulla, Bjesse and Een 2000
  Williams, Biere, Clarke and Gupta 2000]

  - Syntactic quantifier elimination
  - Blocking clauses
  - CTL Model Checking with SAT

  ➔ Need some background that is beyond the scope of this class...

SAT-based image

- May provide a good alternative when other technique (e.g. BDD) fails.
- Does not take advantage of SAT solver's ability to filter out irrelevant facts, since exact image is computed.

- Think:
  - Our goal of proof is: (1) Either find "a" counter-example, or (2) Prove that no counter-example can be found in reachable states

  ➔ Do we really need to compute the "exact" image of the reachable states?

Limitation of Formal Engine

- Still bounded by exponential complexity
  - When design gets big, or property becomes complex, it is very often that the formal engine cannot conclude the proof result
  - ➔ We have “coverage metric” for simulation, what about formal method?
- However, property checking by formal engine is somewhat analogous to reversely reasoning on designer’s intent
  - Designer’s intent should not be too complicated for one local module
  - It’s the interaction on the modules that make the problem difficult

Abstraction and Refinement

If we can confine our search/reasoning on some boundary (e.g. local module, FF boundary), we can simplify our proof

➔ Abstraction

But simplifying something means something is ignored.... the result may not be accurate

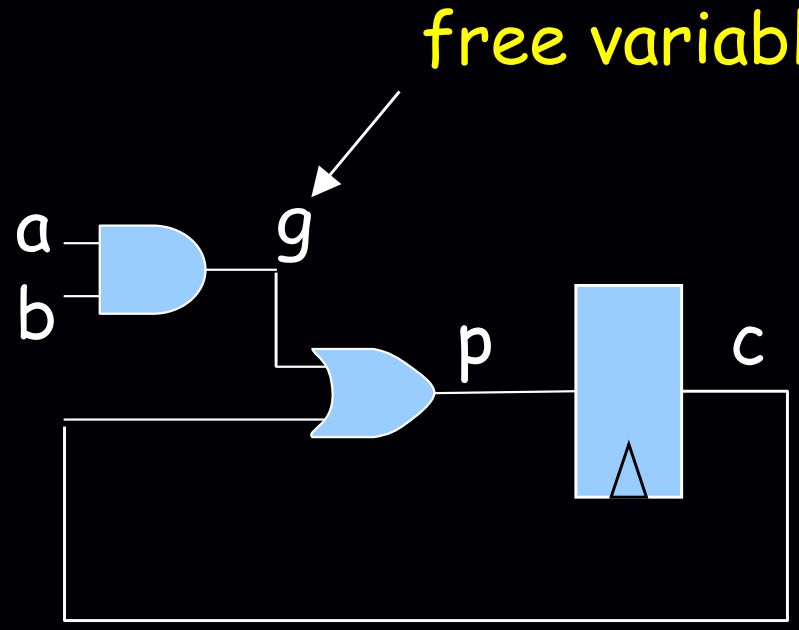
➔ Refinement

...refined to a bigger search region

Localization abstraction

Kurshan

- Property: G (c $\Rightarrow$ X c)



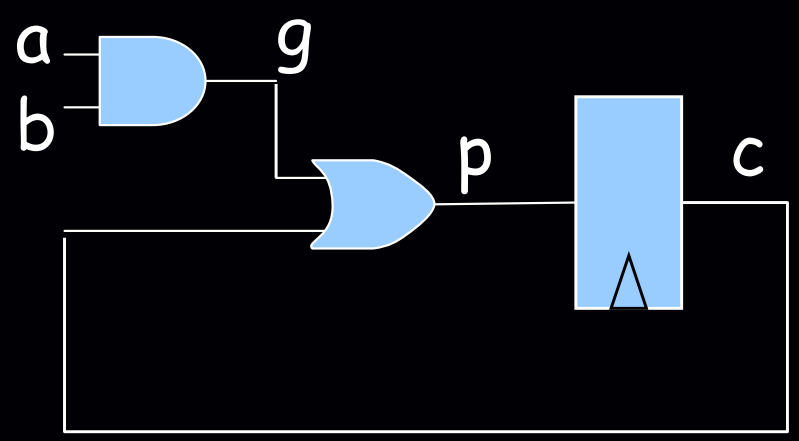
Model:

$C' = \{$
$g = a \wedge b,$
$p = g \vee c,$
$c' = p$
$\}$

$$\frac{C' \Rightarrow \text{property},\ C \Rightarrow C'}{C \Rightarrow \text{property}}$$

Constraint granularity

Most authors use constraints at "latch" granularity...



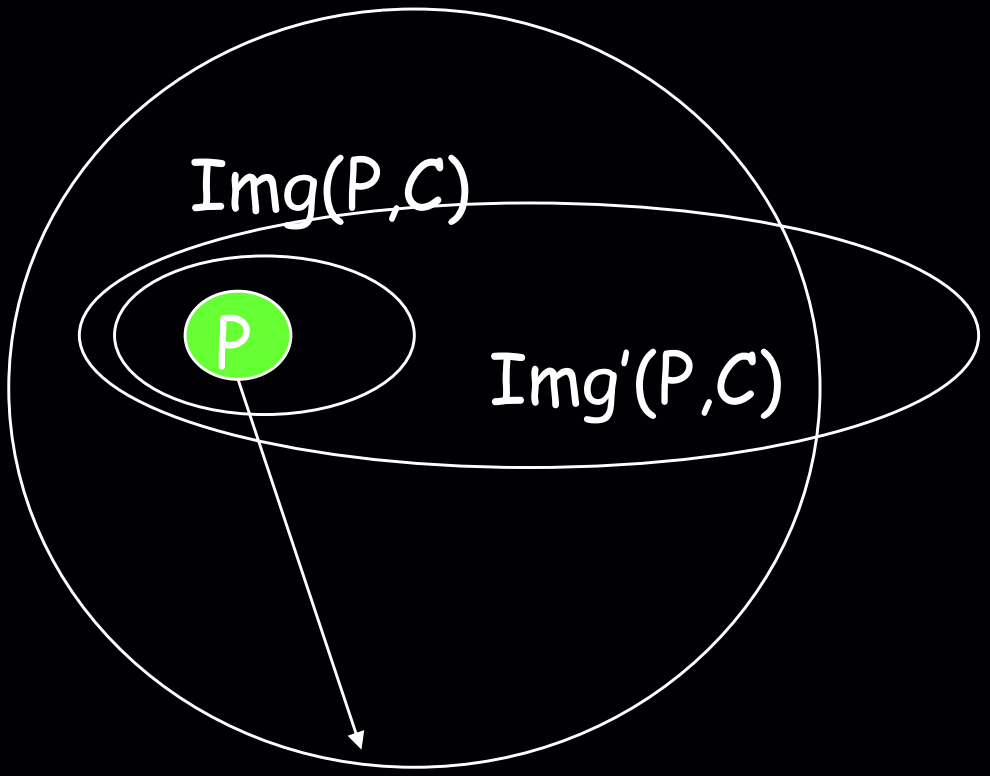
Model:

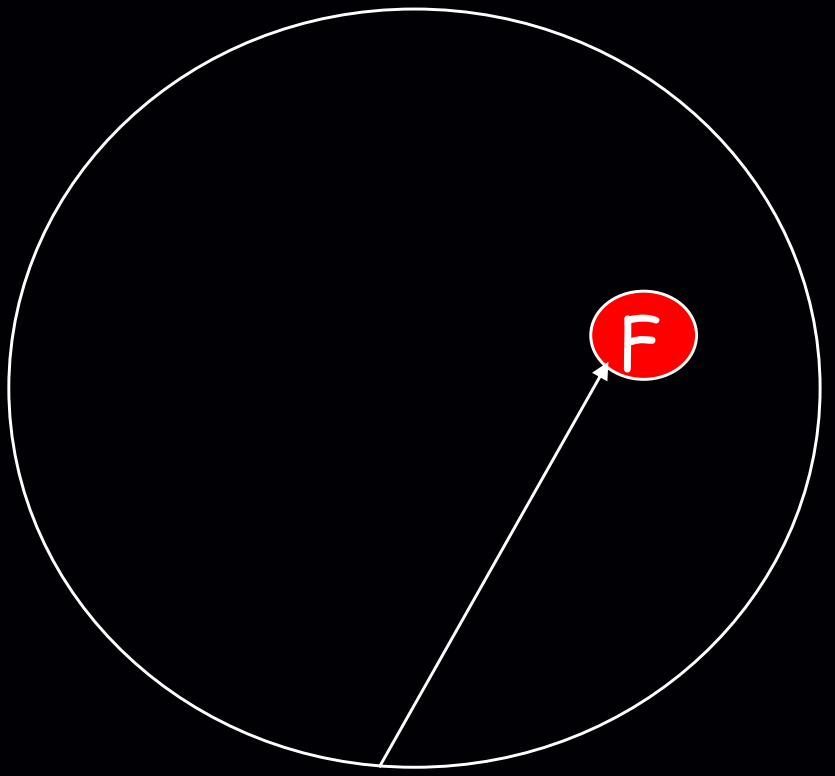
$C = \{$
$c' = (a \wedge b) \vee c$
$\}$

...however, techniques we will consider can be applied at both "gate" and "latch" granularity.

Image and over-approximated image

- Img → Img'
- R → R'
- If (R' ∧ F == ∅) ➔ (R ∧ F == ∅)
- If (R' ∧ F != ∅) ➔ Need to refine R' (Img')





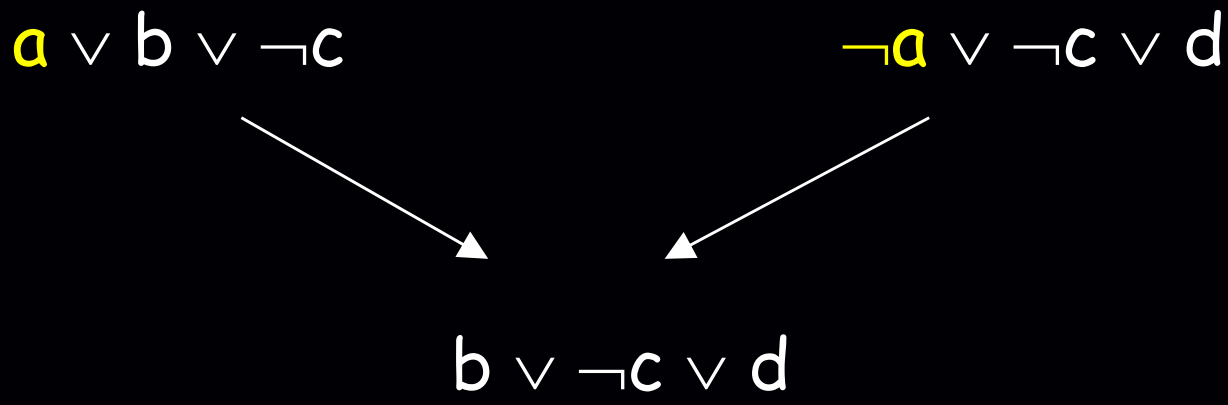
But, how to use SAT to compute the over-approximated image?

You need to understand two key techniques ---

Unsatisfiability Core

Interpolation

Remember: Resolution



When a conflict occurs, the implication graph is used to guide the resolution of clauses, so that the same conflict will not occur again.

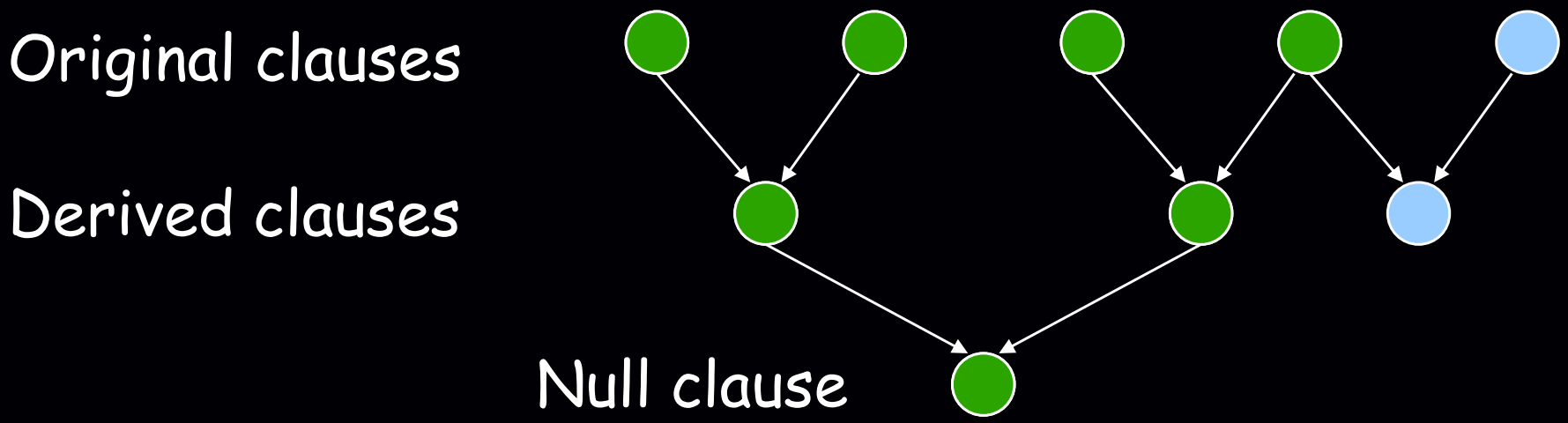
Conflict Clauses (cont.)

- Conflict clauses:
  - Are generated by resolution
  - Are implied by existing clauses
  - Are in conflict in the current assignment
  - Are safely added to the clause set

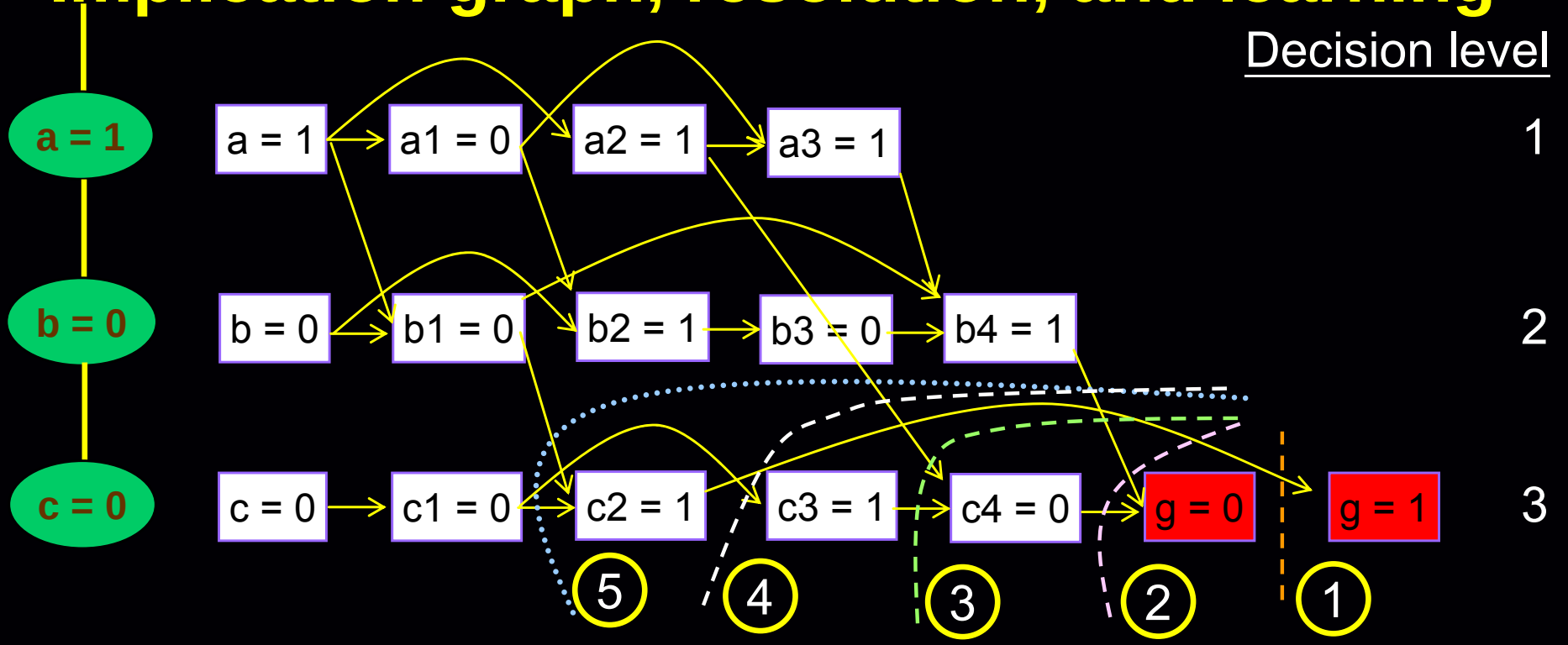
Many heuristics are available for determining when to terminate the resolution process. (e.g. first UIP)

Generating refutations

- Refutation = a proof of the null clause
  - Also called "proof core" or "UNSAT core"
  - Record a DAG containing all resolution steps performed during conflict clause generation.
  - When null clause is generated, we can extract a proof of the null clause as a resolution DAG.



Refresh:
Implication graph, resolution, and learning



(1): (c2' + g)
(2): (b4' + c4 + g') → (b4' + c2' + c4)
(3): (a2' + c3' + c4') → (a2' + b4' + c2' + c3')
(4): (c1 + c3) → (a2' + b4' + c1 + c2')
(5): (b1 + c1 + c2) → (a2' + b1 + b4' + c1)

Extraction of Unsatisfiability Core

1. For each conflict, record the resolution graph for the learned clause
2. A learned clause may depend on other learned clause, so does its resolution graph may build upon other's resolution graph
3. The root clauses for the last conflict (i.e. conflict at decision level 0) will be the unsatisfiability core

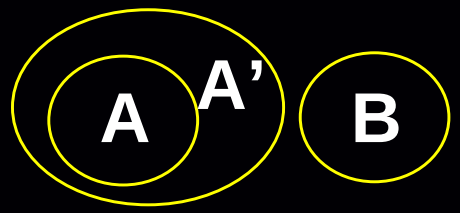
Interpolation

(Craig,57)

- If A $\wedge$ B = false, there exists an *interpolant* A' for (A,B) such that:

$$A \Rightarrow A'$$

$$A' \wedge B = \text{false}$$



A' refers only to common variables of A,B

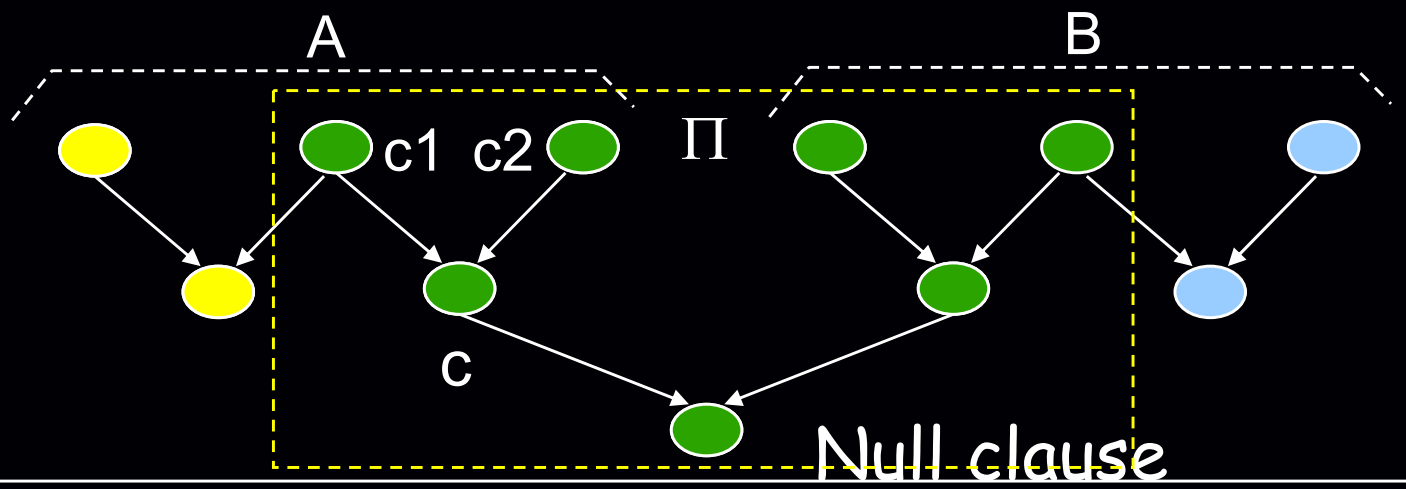
- Example:
  - A = p $\wedge$ q, B = $\neg$q $\wedge$ r, A' = q
- New result
  - given a resolution refutation of A $\wedge$B,

    A' can be derived in linear time.

(Pudlak,Krajicek,97)

Some Definitions for Unsatisfiability Proof

- Let (A,B) be a pair of clause sets and let Π be a proof of unsatisfiability of $A \cup B$
  - Π is a DAG (V_Π, E_Π)
  - Each vertex $c \in \Pi$ in the graph corresponds to a clause and has exactly 2 predecessors, say c1, c2
    - c is called the "resolvent" of c1 and c2
    - The resolved variable v is called the "pivot" variable
  - Π has exactly 1 leaf vertex which is a False (null clause)
  - The roots are original clauses in $A \cup B$



Some Definitions for Unsatisfiability Proof

- Let (A,B) be a pair of clause sets and let Π be a proof of unsatisfiability of $A \cup B$
  - Π is a DAG (V_Π, E_Π)
  - Each vertex $c \in \Pi$ in the graph corresponds to a clause and has exactly 2 predecessors, say c1, c2
    - c is called the "resolvent" of c1 and c2
    - The resolved variable v is called the "pivot" variable
  - Π has exactly 1 leaf vertex which is a False (null clause)
  - The roots are original clauses in $A \cup B$
- Global/Local variable/literal
  - With respect to (A,B), a variable/literal is global if it appears in both A and B
  - It is called local to A if it appears only in A

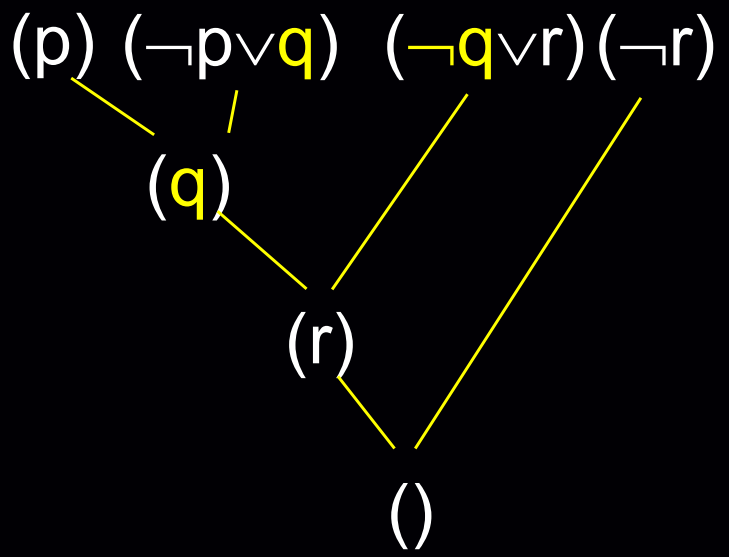
Interpolants from Proofs

- Deriving interpolant from Π
  - ➔ Calling itp(leaf vertex)
- **itp(c) { // c $\in$ V_Π let p(c) be a**

```
    if c is a root, then
      if c ∈ A then
        itp(c) = the disjunction of the
                 global literals in c
      else itp(c) = constant True
    else, let c1, c2 be the predecessors of c
          and let v be their pivot variable
      if v is local to A
        then itp(c) = itp(c1) ∨ itp(c2)
      else itp(c) = itp(c1) ∧ itp(c2)
  }
```

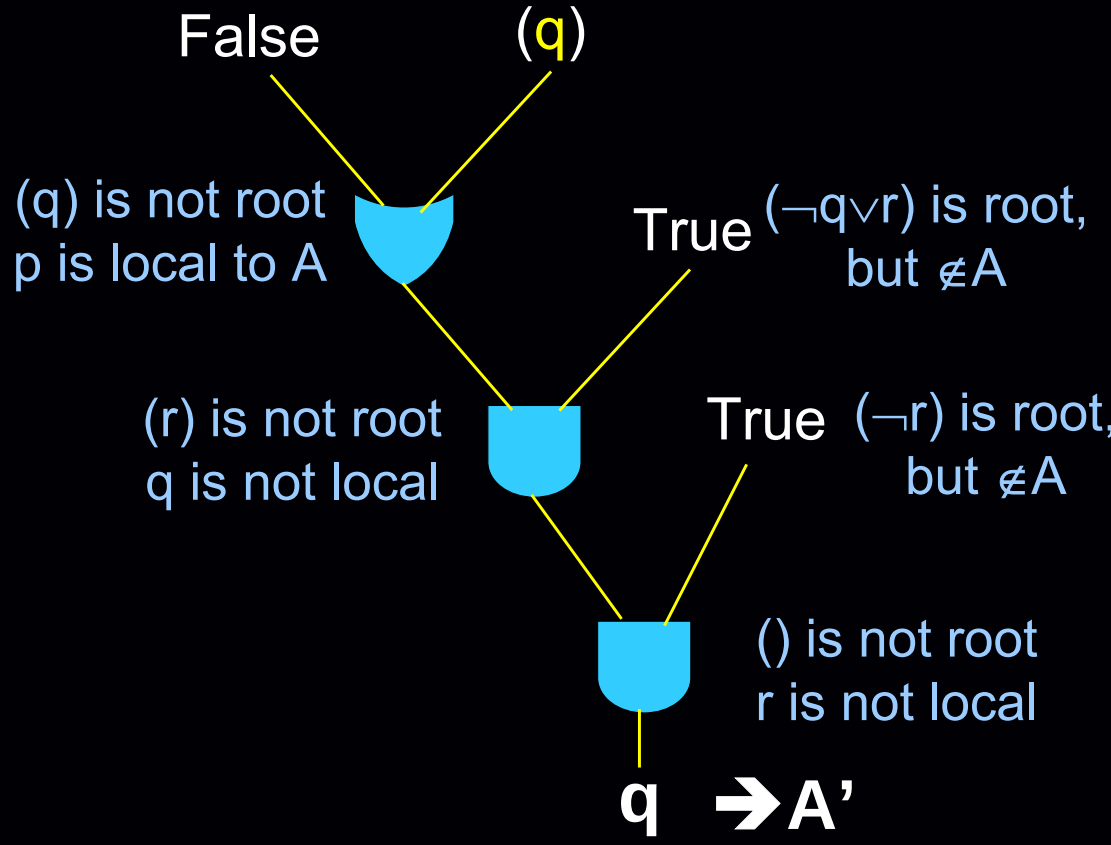
Interpolants from Proofs

A = (p)(¬p∨q) B = (¬q∨r) (¬ r)



(p) is root and ∈ A
but global literals = ∅

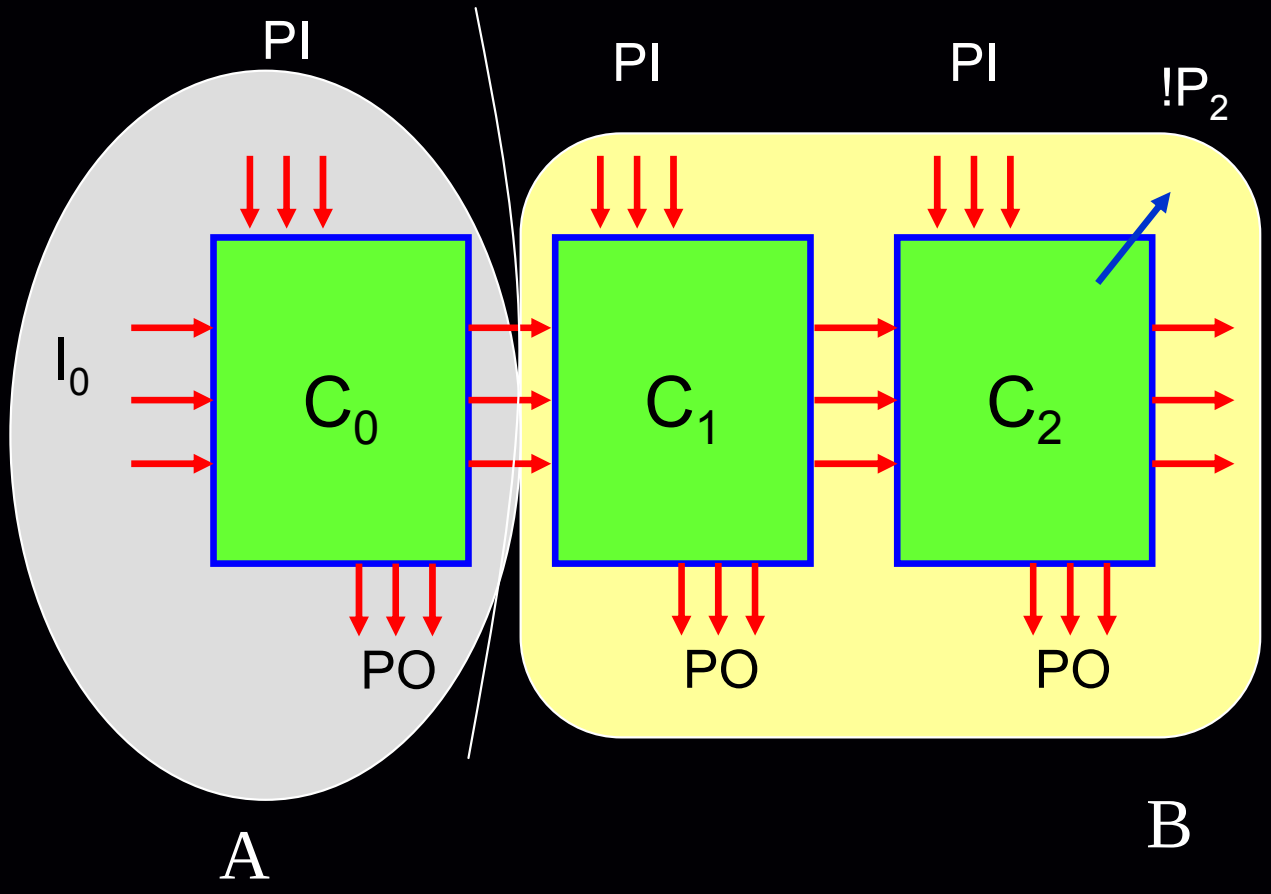
(¬p∨q) is root and ∈ A
global literals = {q}



itp(c) { // c ∈ VΠ let p(c) be a
** if c is a root, then**
**   if c ∈ A then**
**   itp(c) = the disjunction of the global literals in c**
**   else itp(c) = constant True**
** else, let c1, c2 be the predecessors of c**
**   and let v be their pivot variable**
**   if v is local to A**
**   then itp(c) = itp(c1) ∨ itp(c2)**
**   else p(c) = p(c1) ∧ p(c2)**
}

SAT to compute set of reachable states?

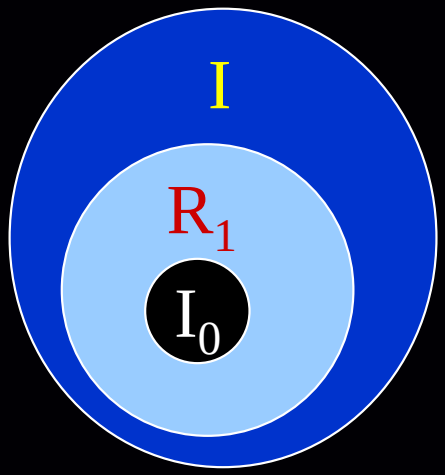
What is the set of reachable states here?



What are the common variables? What is the interpolant?

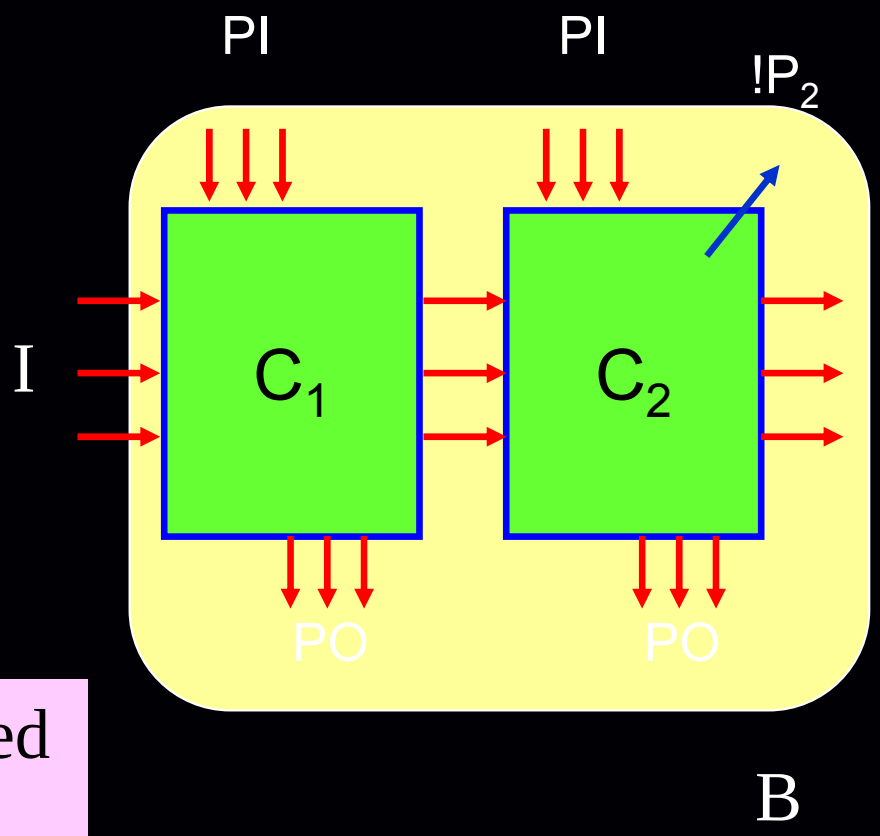
SAT to compute set of reachable states?

What is the set of reachable states here?



$R_1 = A(I_0)$

I is an over-approximated image of I_0



What are the common variables? What is the interpolant?

Overapproximation

- An overapproximate image op. is Img' s.t.

  for all P, Img(P,C) implies Img'(P,C)

- Overapprimate reachability:

$$R'_0 = I$$

$$R'_{i+1} = R'_i \vee Img'(R'_i, C)$$

$$R' = \cup R'_i$$

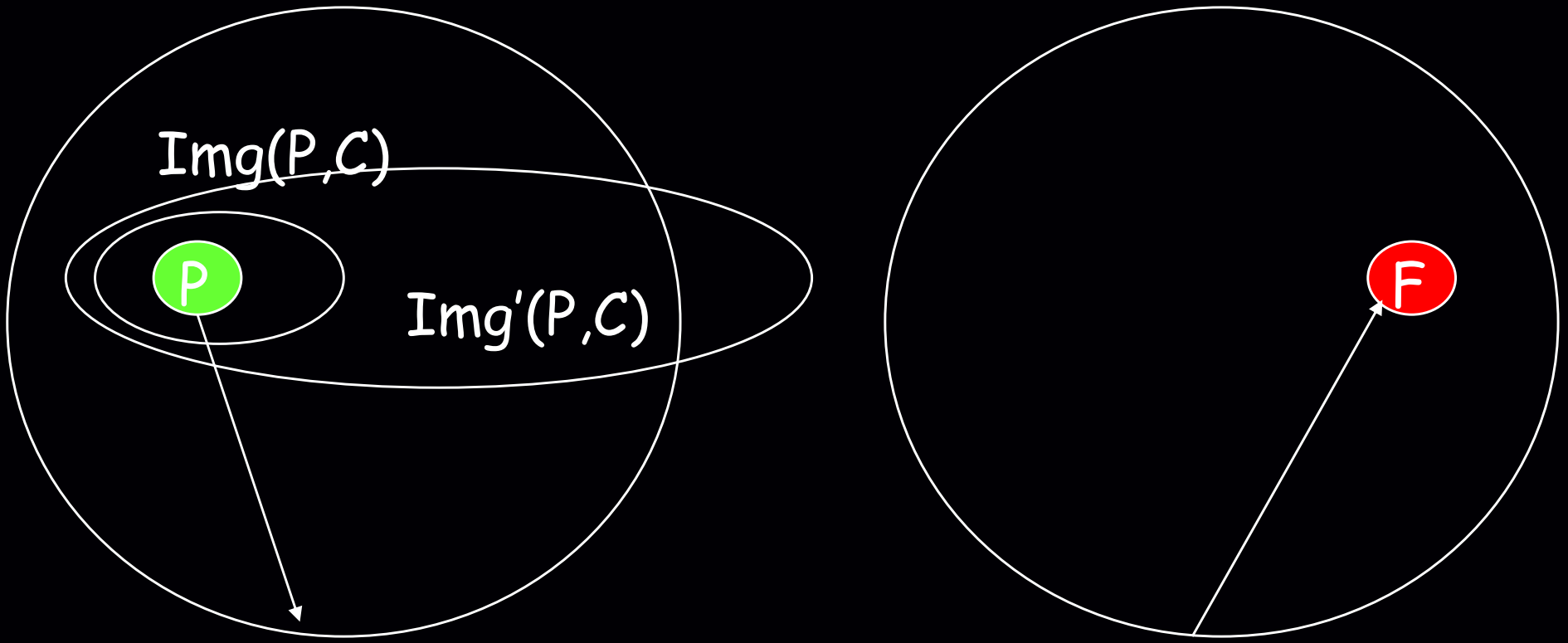
- Img' is adequate (w.r.t.) F, when
  - if P cannot reach F, Img'(P,C) cannot reach F
- If Img' is adequate, then
  - F is reachable iff $R' \wedge F \neq$ false

k-adequate image operator

- Img' is k-adequate (w.r.t.) F, when
  - if P cannot reach F,

    Img′(P,C) cannot reach F within k steps
- Note, if k > diameter, then k-adequate is equivalent to adequate.

Adequate image



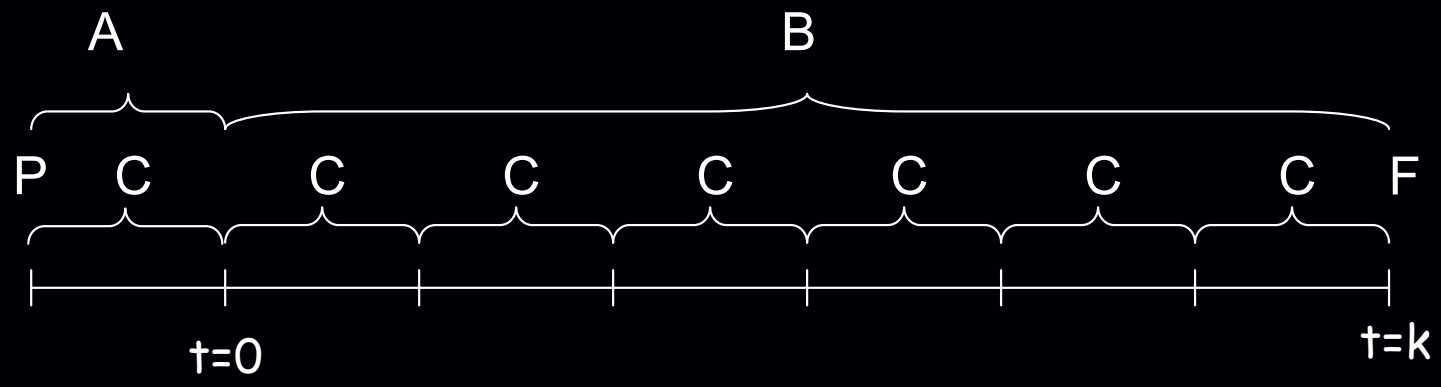
But how do you get an adequate Img'?

Interpolation-based image

- Idea -- use unfolding to enforce k-adequacy

$A = P_{-1} \wedge C_{-1}$

$B = C_0 \wedge C_1 \wedge \ldots \wedge C_{k-1} \wedge F_k$

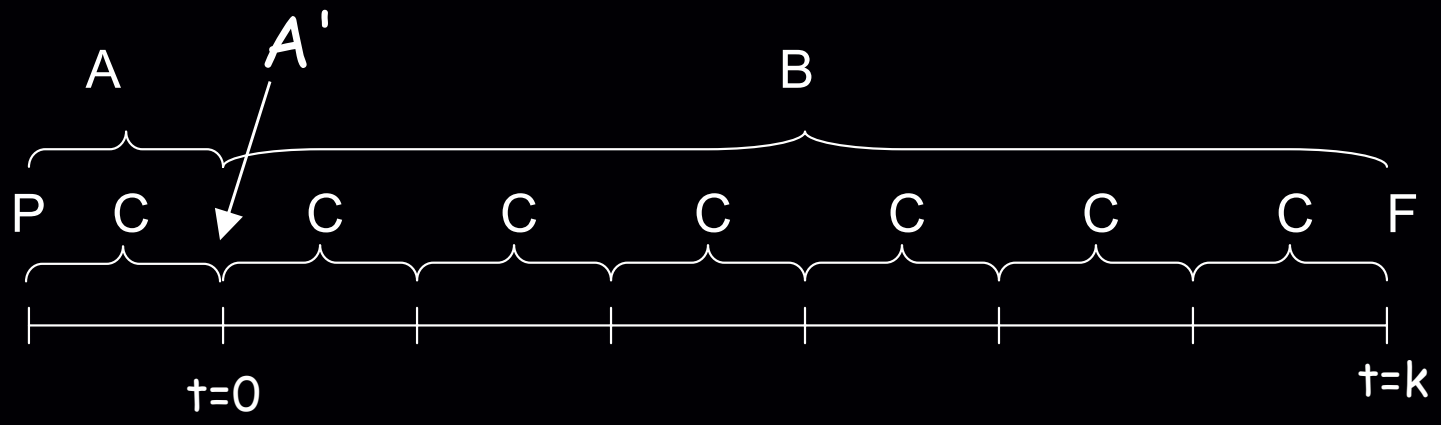


Let $Img'(P)_0 = A'$,
where A' is an interpolant for (A,B)...

- remember: A' contains the common variables of (A, B)

Img' is k-adequate!

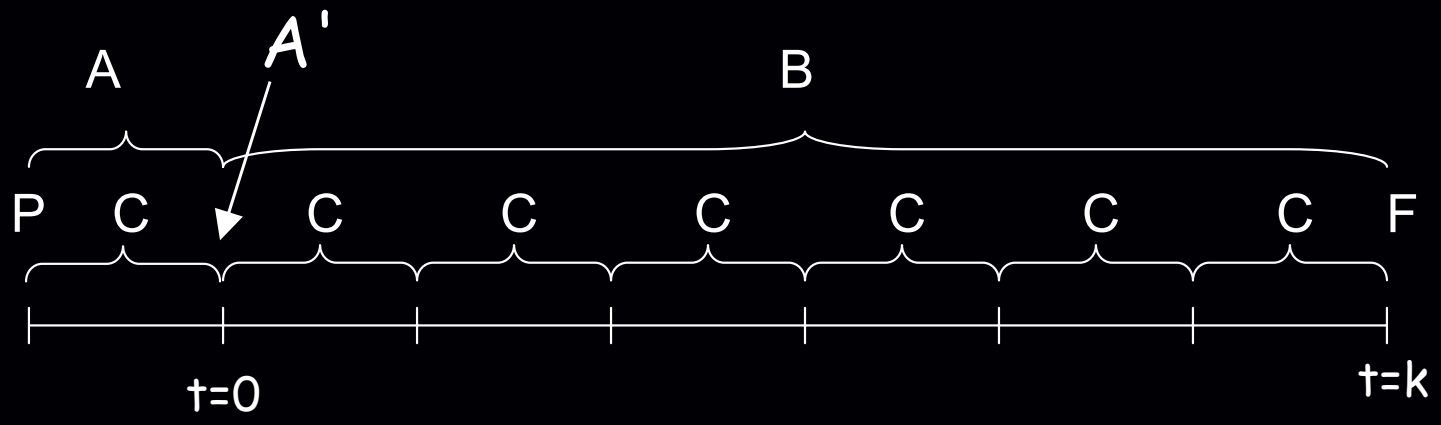
Huh?



- $A \Rightarrow A'$
  - $Img(P,C) \Rightarrow Img'(P,C)$
- $A' \wedge B$ = false
  - $Img'(P,C)$ cannot reach F in k steps
- Hence Img' is k-adequate overapprox.

But note, Img' is partial -- not defined if $A \wedge B$ is sat.

Intuition



- A' tells everything the SAT solver deduced about the image of P in proving it can't reach F in k steps.
- Hence, A' is in some sense an abstraction of the image relative to the property.

Reachability algorithm

```
let k = 0
repeat
   if I can reach F within k steps, answer reachable
   R = I
   while Img'(R,C) ∧ F  = false
        R' = Img'(R,C) ∨ R
        if R' = R answer unreachable
        R = R'
   end while
   increase k
end repeat
```

Termination

- Since k increases at every iteration, eventually k > d, the diameter, in which case Img' is adequate, and hence we terminate.

Notes:

- don't need to know when k > d in order to terminate
- often termination occurs with k << d
- depth bound for earlier method (Sheeran et al '00) is "longest simple path", which can be exponentially longer than diameter

Interpolation-based MC

- Fully SAT-based.
- Avoid computing exact image.
- Inherits SAT solvers ability to concentrate on facts relevant to a property.
  - Maintain SAT solver's advantage of filtering out irrelevant facts.
- For true properties, appears to converge for smaller k values.

Outline

Conclusion on Boolean SAT Algorithms

- Traditional NP-complete problem
- With several advanced techniques, modern SAT solvers can be very efficient
  - Able to work on million-gate designs
- Applications
  - SAT-based verification
  - Redundancy addition and removal
  - SAT-assisted logic optimization (e.g. using interpolation technique)
  - In other fields (e.g. OR, AI,...)

Future Research Directions

- Word-level SAT engine
  - Able to handle constraints beyond Boolean
    - Bit-vector, integer, uninterpreted function, etc
  - For the future system-level designs
- SAT-assisted logic optimization
  - Interpolation techniques
  - Pseudo Boolean constraint solver

Some popular SAT engines

- SATO
  - http://www.cs.uiowa.edu/~hzhang/sato.html
- GRASP
  - http://www.eecs.umich.edu/~faloul/Tools/satire/Grasp2.exe
- zChaff
  - http://www.princeton.edu/~chaff/
- BerkMin
  - http://eigold.tripod.com/BerkMin.html
- CSAT
  - http://cadlab.ece.ucsb.edu/downloads/CSAT.htm
- miniSAT
  - http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/
- PBS
  - http://www.eecs.umich.edu/~faloul/Tools/pbs4/
- Pueblo
  - http://www.eecs.umich.edu/~hsheini/pueblo/

That’s all.

Any questions?