Topic IX

Formal Hardware Verification (II)
ATPG/SAT-Based Engine Basics

系統晶片驗證

SoC Verification

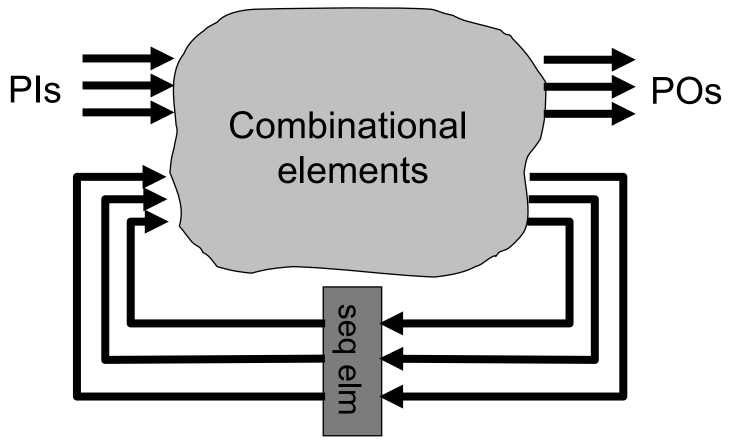
Sep, 2004

What we will cover in this topic ---

1. Circuit modeling summary
   - Assertion property modeling
2. Combinational ATPG/SAT algorithms
   - Logic implications
   - Branch and bound process
   - Learning
   - Dynamic decision ordering
3. Word-level ATPG
4. RAMBO

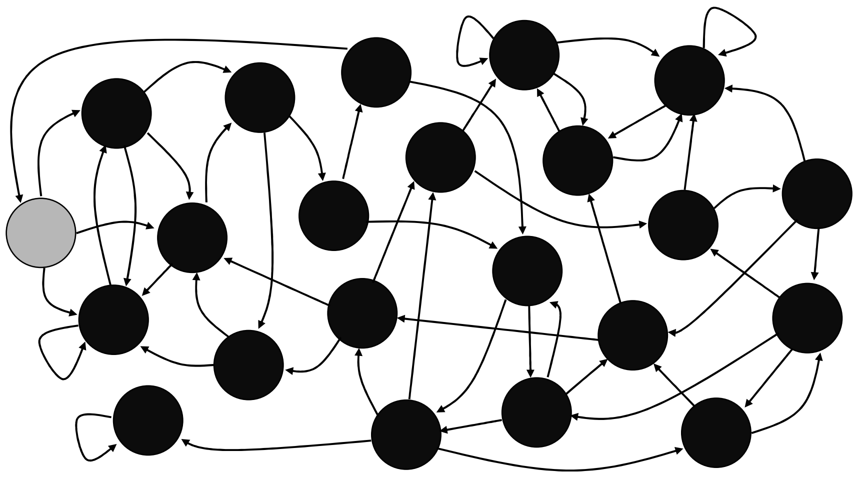
Different ways to view the circuit modeling

1. Hoffman Model



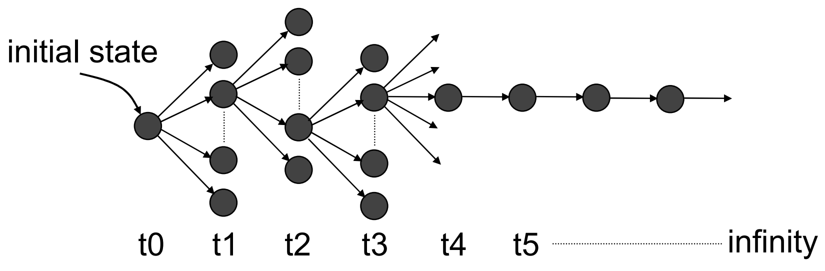
Different ways to view the circuit modeling

2. Finite State Machine (State Transition Graph)



Different ways to view the circuit modeling

3. Computational Tree Logic (CTL)



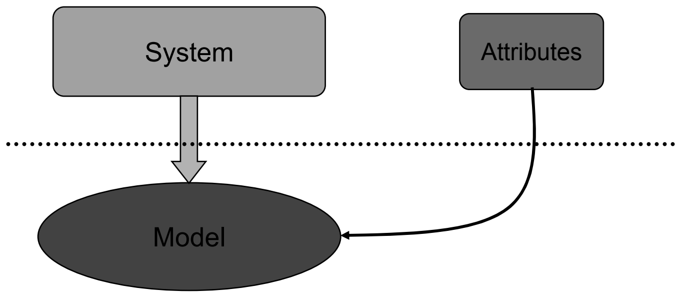
Different ways to view the circuit modeling

4. Source codes

```
class Circuit
{
public:
   Circuit(const string& inFile);
   ~Circuit();

   bool simulate(const string&
    vectorFile);
   friend ostream& operator <<
    (ostream&, const Circuit&);

private:
   vector<Gate *>      _piList;
   vector<Gate *>      _poList;
   vector<Gate *>      _seqElmList;
};
```

```
class Gate
{
public:
   Gate(unsigned numFanins = 0);
   virtual ~Gate();

   const string& getName() const;
   virtual bool simulate() = 0;
   friend ostream& operator <<
    (ostream&, const Gate*);

protected:
   string              _name;
   unsigned            _id;
   GateValue           _value;
   GateFlag            _flag;
   vector<Gate *>      _faninList;
   vector<Gate *>      _fanoutList;
};
```

Remember, formal verification is ---



- ◆ Variety of verification methods (often quite different)
- ◆ What distinguishes formal?
  - ● Conveys a promise of mathematical certainty
    ➔ Guarantees that no behavior or execution of the model can ever be found to contradict the attributes

Attributes (or called "properties") to a circuit in general are *temporal logic*

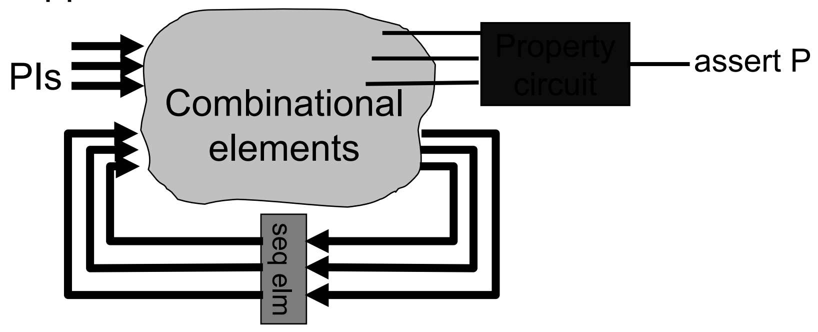
We will cover the details of temporal logic later

To simplify the explanation of the formal verification engines,

Let's first focus on the simplest one, assertion property

Assertion Property Modeling

◆Assure that something bad should never happen



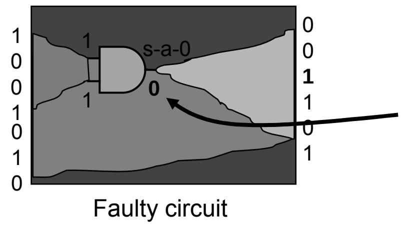
(Note: we don't care about POs)

In other words,
proving assert P is true
= proving there is no counter-example for ~P

➔ Use a "search engine"

Automatic Test Pattern Generation (ATPG)

- Deterministic algorithms of generating test patterns for manufacturing faults
- Developed as early as in 1960's
- In essence, a branch-and-bound algorithm
- 2 major steps:
  fault sensitization + fault propagation



What verification needs is just the "fault sensitization" part

The traditional sequential ATPG is very difficult (not applicable to mid-size design).

However, for verification without considering the fault propagation, the sequential ATPG can sometimes be used for large circuit.

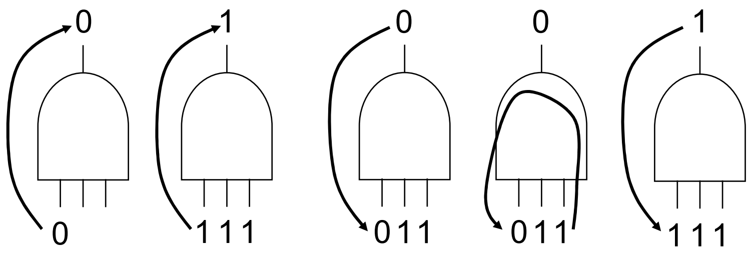
To simplify the explanation of ATPG algorithm, we will start from combinational ATPG first.

Combinational ATPG Algorithm for Verification

```
bool witnessValue(Gate g, value v)
{
   if (logicImplication(g, v) == false)
      return false;
   if (all signals in circuit have been implied)
      return true;
   pick an unassigned signal s
   if (witnessValue(s, V0) == true)
      return true;
   backtrack(s);
   if (witnessValue(s, ~V0) == true)
      return true;
   backtrack(s);
   return false;
}
```

Logic Implication

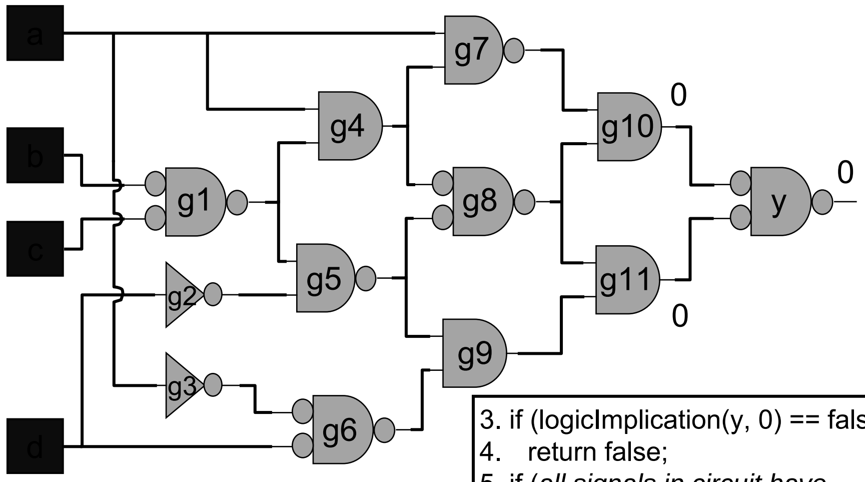
- ◆Also called “Boolean Constraint Propagation” (BCP)
- ◆Imply values to other gates in both forward and backward directions



Forward implications Backward implications

Proving always(y == 1)

witnessValue(y, 0)



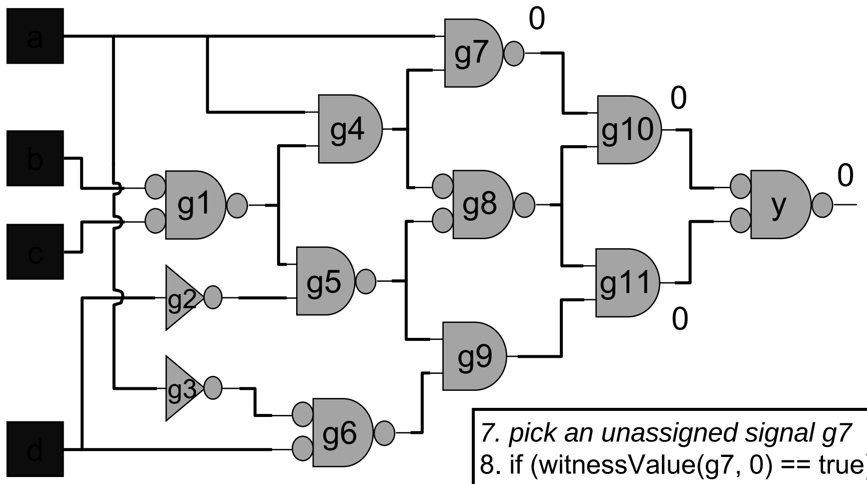
```
3. if (logicImplication(y, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

Combinational ATPG Algorithm for Verification

```
1.  bool witnessValue(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 (witnessValue(s, V0) == true)
9.        return true;
10.    backtrack(s);
11.    if (witnessValue(s, ~V0) == true)
12.       return true;
13.    backtrack(s);
14.    return false;
15.}
```

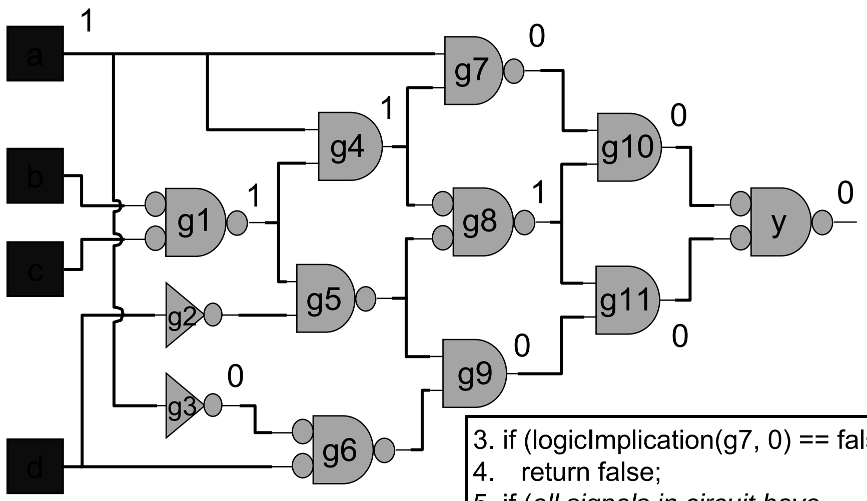
Witness (y, 0)

Decision (g7, 0)



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

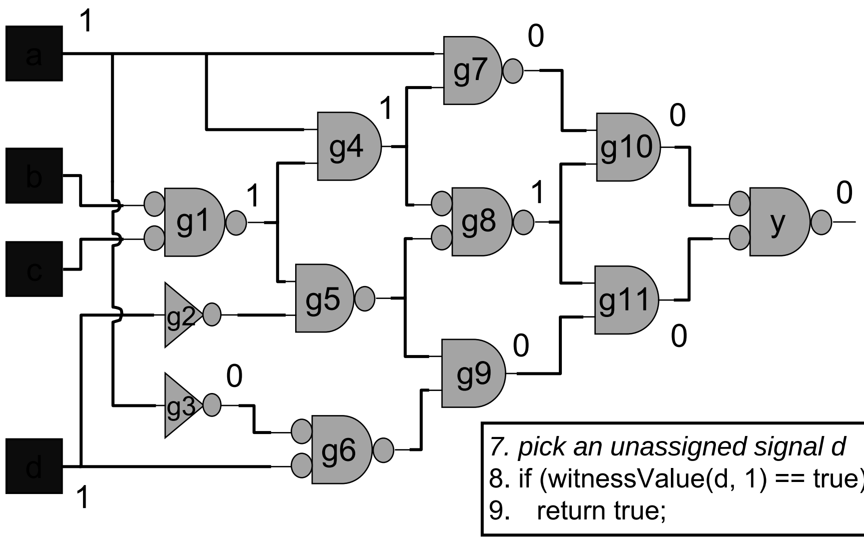
Witness (g7, 0)



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

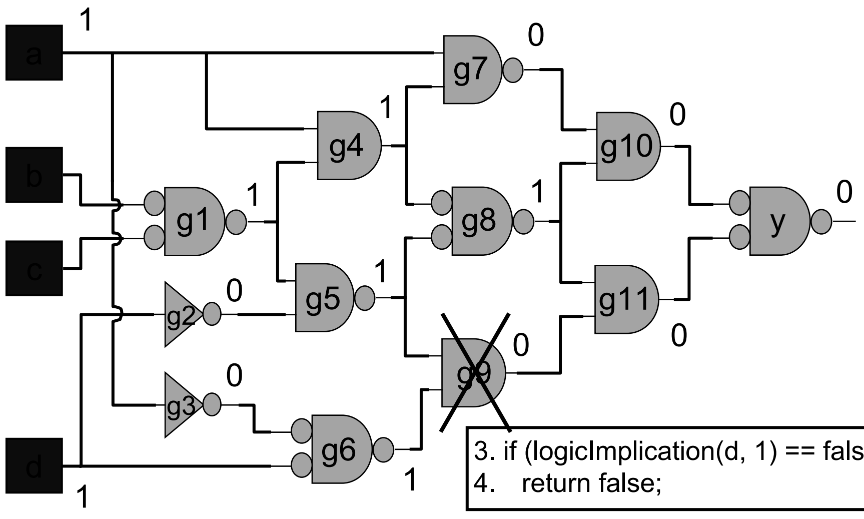
Witness (g7, 0)

Decision (d, 1)



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

Witness (d, 1)

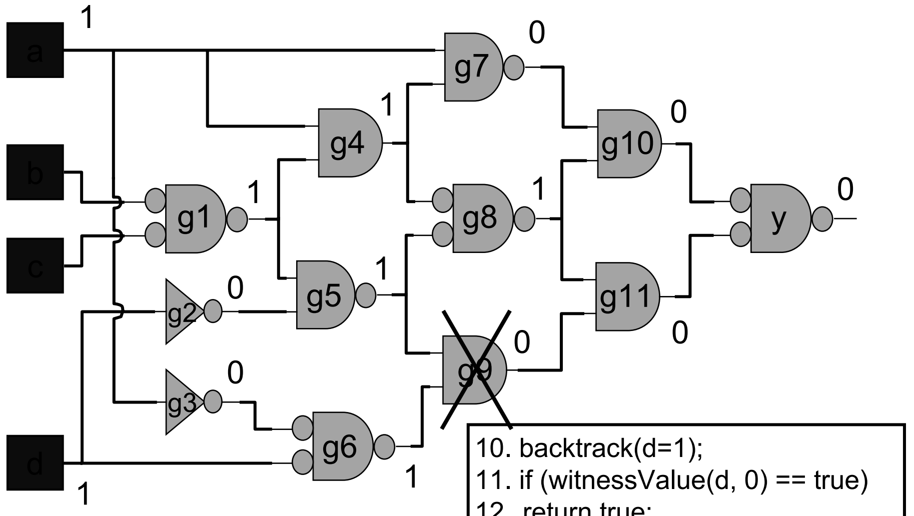


```
3. if (logicImplication(d, 1) == false)
4.    return false;
```

Combinational ATPG Algorithm for Verification

```
1. bool witnessValue(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 (witnessValue(s, V0) == true)
9.       return true;
10.   backtrack(s);
11.   if (witnessValue(s, ~V0) == true)
12.      return true;
13.   backtrack(s);
14.   return false;
15.}
```

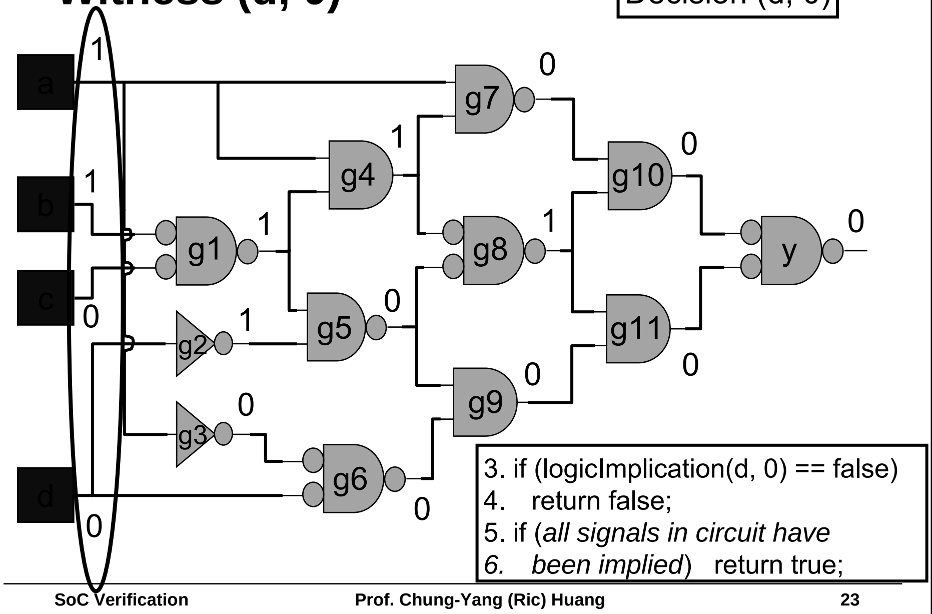
Witness (g7, 0)



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

Witness (d, 0)

Decision (d, 0)



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

Do you see that ATPG algorithm is very similar to SAT?

Actually, they are in principle the same, but just different in data structure

SAT is sometimes called "Satisfiability-based ATPG"

ATPG is sometimes called "Circuit-based SAT"

The Best

Combined advantages from both sides

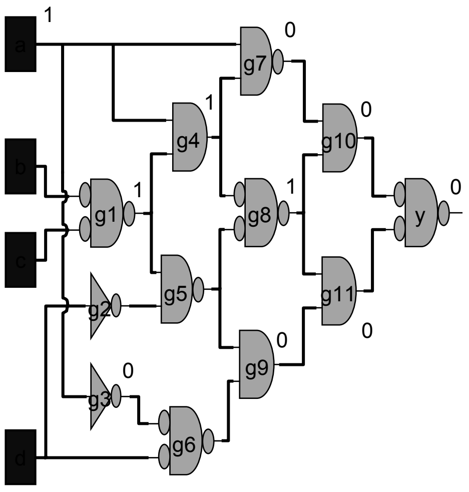
Logic Implication Routines in ATPG/SAT

1. Logic Implication by Recursion
2. Logic Implication in Topological Order
3. 2 Watch Literals algorithm in zChaff

Logic Implication by Recursion

```
bool logicImplication(Gate g, value v)
{
   if (g->hasConflictValue(v)) return false;
   if (g->getValue() != ‘x’) return true;
   // g has value ‘x’ ➔ can be implied
   g->setValue(v);
   // fanins and fanouts that may be
   //    implied by g
   List impliedList = checkImplication(g);
   for_each_gate_value(impliedList, gg, vv)
      if (logicImplication(gg, vv) == false)
         return false;
   return true;
}
```

Logic Implication by Recursion Example



$$\begin{tikzcd}
g7 \arrow[r] \arrow[ddr] \arrow[dddddr] & a \arrow[r] \arrow[dr] & g3 \arrow[r] & g6 & \\
 & & g4 & & \\
 & g4 \arrow[r] \arrow[dr] \arrow[ddr] & a & & \\
 & & g1 \arrow[r] & g5 & \\
 & & g8 \arrow[r] \arrow[dr] & g10 & \\
 & g10 & & g11 \arrow[r] & g9
\end{tikzcd}$$

Recursion tree

Logic Implication by Recursion

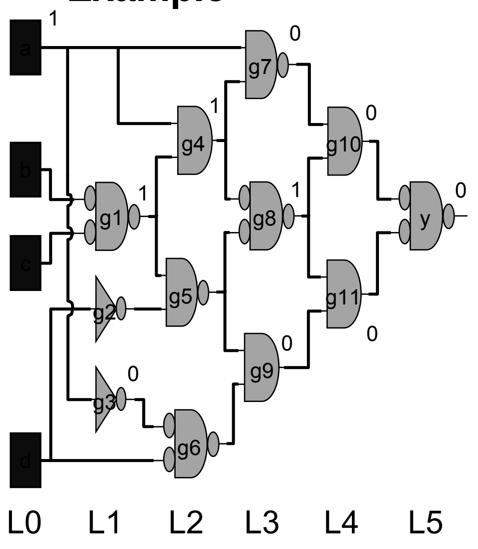
- Pros
  - Easy to implement
  - May lead to conflict earlier
- Cons
  - A gate may be put into `impliedList` many times by different neighboring gates
    1. Value has already been implied, or
    2. Too early to evaluate

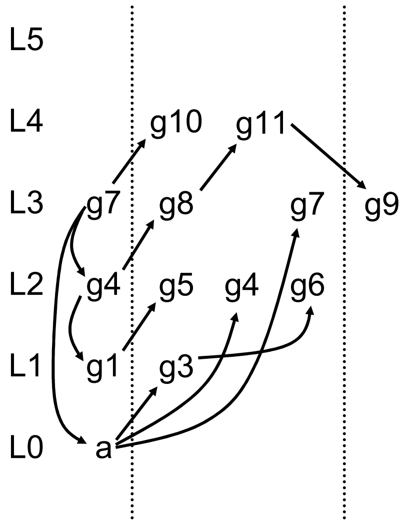
Logic Implication in Topological Order

- Preprocess
  - All gates are sorted topologically
  - Each gate has a field "_level" = max(fanin->_level) + 1

```
bool logicImplication()
{
   while (more scheduled gates)
      for scheduled gates (level = maxLevel to 0)
         apply backward implications and
            schedule forward implications
         return false if any conflict;
      for scheduled gates (level = 0 to maxLevel)
         apply forward implications and
            schedule backward implications;
         return false if any conflict;
   return true;
}
```

Logic Implication in Topological Order Example





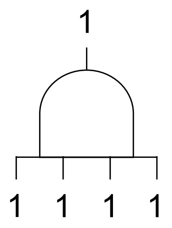
Logic Implication in Topological Order

◆Pros
- Minimize the number of scheduling
- Each gate is checked only once in each forward or backward implication

◆Cons
- A gate may be checked more times than necessary

Fruitless Implication Checking

- ◆ Checking if a gate can be implied, but it cannot
- ◆ 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)



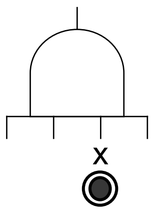
Trying to avoid fruitless implication checking

- ◆ 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 ➔ forward implication

- ◆ Although this can avoid fruitless implication checking, but the overhead in maintaining the counts could be an overkill…

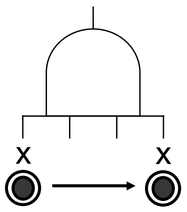
Watched-fanin concept

- ◆ 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’ ➔ don’t care
  - ● 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’ at the gate



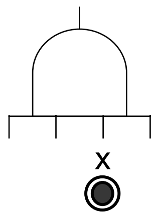
What’s the improvement?

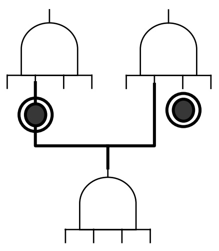
- ◆ Worse case $O(n^2)$?? No
- ◆ Suppose watched fanin points to the 1st fanin in the beginning
  - ● We always follow the same direction to find the next ‘x’ fanin
  - ● Complexity O(2n) → O(n)



Reducing from O(n) to almost O(C)

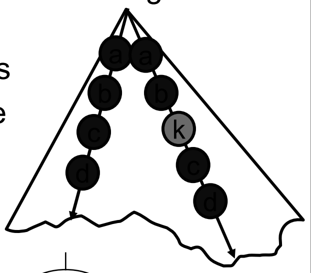
- ◆ Watched fanin
  - When one fanin gets an implication, check if this fanin is “watched fanin”
  - ➔ Still need to check for each fanin implication
- ◆ Watching list from watched fanin
  - The watched fanin keeps the list of gates it is watching
    - When a watched fanin gets an implication
    - ➔ Update the watched fanins of the gates in the watching list; remove this watching list
    - ➔ Create the watching lists for the new watched fanins
  - Don’t need to evaluate a gate if it is not in the watching list of any fanin

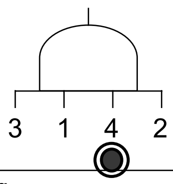




Caching Effect

- ◆ 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)
  - ➔ Fanins of a gate get the implications almost at the same order every time
- ◆ Watched fanin
  - → point to the last implied fanin
  - ➔ After the first backtrack, no evaluations for the previous fanins





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

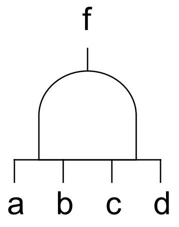
Different watched schemes?

(Could be very complex…)

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

Review on SAT Clause Construction

- For an n-input AND gate ➔ (n + 1) clauses
  - 1 clause for all-1's implication
    - Non-controlling fanin value
    - (a & b & c & d) → f
    - (!a + !b + !c + !d + f)
    - (n + 1)-literal clause
  - n clauses for 0 implications
    - Controlling fanin value
    - !a → !f
    - (a + !f)
    - 2-literal clauses



Logic Implication for SAT

- Also called “Boolean Constraint Propagation (BCP)” (why?)
- If a literal in a clause gets an implication ‘1’
  - ➔ The clause is satisfied
- If a literal in a clause gets an implication ‘0’
  - ➔ If all but one literals have values ‘0’
    - The remaining literal will get implication ‘1’
  - ➔ If all literals have values ‘0’
    - The clause is evaluated to ‘0’ ➔ a conflict !!

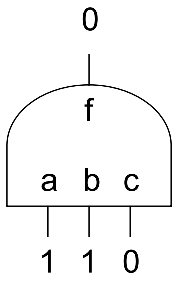
Efficient BCP Routine in zChaff

- For 2-literal clauses
  - If any literal gets an implication
    - We can conclude the implication of this clause immediately
- For n-literal clauses (n > 2)

  e.g. f = a & b & c
  - 4-literal clause (!a + !b + !c + f)
  - (!a = 0) & (!b = 0) & (!c = 0) → (f = 1)
    - a & b & c → f
  - (!a = 0) & (!b = 0) & (f = 0) → (!c = 1)
    - a & b & !f → !c

  ➔ Watch f together with a, b, and c

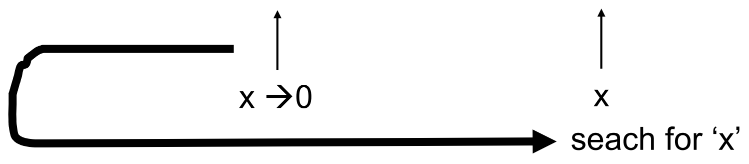
  ➔ 2 watch literals (why?)



2-Watched-Literals algorithm in zChaff

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

L1 + L2 + …+ L50 + L98 + L99 + L100



Comments on ATPG and SAT Implication

- zChaff proposed a very simple yet efficient BCP
  - All types of implications can be implemented in the same 2-watched-literal algorithm
- Can ATPG mimic the similar “watched xxx” concept?
  - The previous watched fanin does
    - But only for non-controlling fanin’s forward implication
  - (How?) Can it be efficient?

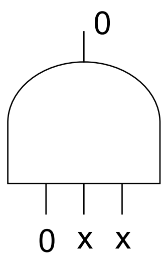
Combinational ATPG Algorithm for Verification

```
bool witnessValue(Gate g, value v)
{
   if (logicImplication(g, v) == false)
      return false;
   if (all signals in circuit have been implied)
      return true;
   pick an unassigned signal s
   if (witnessValue(s, V0) == true)
      return true;
   backtrack(s);
   if (witnessValue(s, ~V0) == true)
      return true;
   backtrack(s);
   return false;
}
```

How do we know that the target (~P) has been justified?

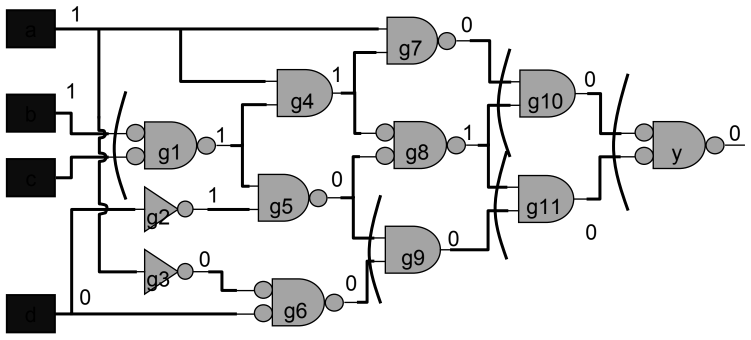
◆ Bottom line
- The implied values on PIs ➔ Test pattern
- If we simulate the test pattern, we should be able to witness the target value

◆ A conservative approach
- All the signals in the circuit have been implied
- Some are redundant assignments
  - e.g. AND gate with value '0'



Justification Frontier

◆ Keep a set of gates that are not yet justified
- Update the frontier after each decision implications
- Restore the frontier on backtracks
- When frontier becomes empty, the target is satisfied



Justification Frontier (Cont'd)

◆ Pros
- Only justify a minimal set of gates
- Minimize the assignments on test vectors
- May avoid useless decisions

◆ Cons
- Overhead in maintaining the justification frontier (in both implementation and runtime)

With or without justification frontier??

Combinational ATPG Algorithm for Verification

```
bool witnessValue(Gate g, value v)
{
   if (logicImplication(g, v) == false)
      return false;
   if (all signals in circuit have been implied)
      return true;
   pick an unassigned signal s
   if (witnessValue(s, V0) == true)
      return true;
   backtrack(s);
   if (witnessValue(s, ~V0) == true)
      return true;
   backtrack(s);
   return false;
}
```

Branch-and-bound Algorithm

- Binary decision tree
  - Branched by picking a gate for decision
  - Bounded by logic implication conflict
  - Exponential complexity
- When bounded…
  - Backtrack the last implications
  - Make decision on the opposite value, or ---
  - If both values have been tested, return false to parent decision (and it will be bounded, too)

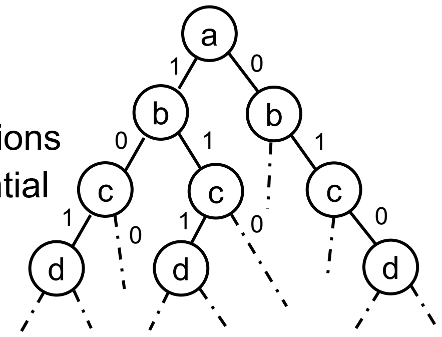
Speed-ups on branch-and-bound process

1. Better decision ordering
2. Learning
3. Bound earlier

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

Dynamic Decision Ordering

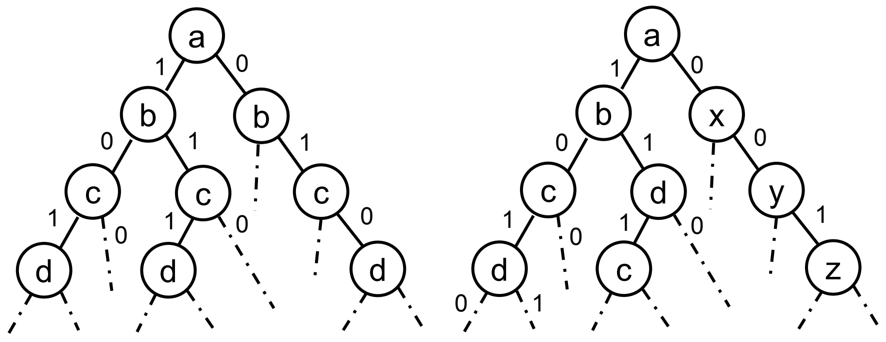
◆ Decision order and values are dynamically determined based on current implication values, justification frontier, etc.
- Use similar criteria as static method
- But can mix different rules dynamically

◆ Pros
- May lead to better decisions
- Avoid useless decisions

◆ Cons
- Overhead in computing dynamic ordering may be high
- Effectiveness sometimes is hard to predict

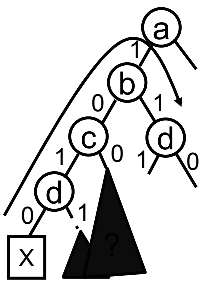
A closer look at binary decision tree

1. Should the decision orderings on all branches be the same?



A closer look at binary decision tree

2. When conflicts, do we always need to backtrack one decision at a time? Or can we backtrack multiple decisions at once?
   - Called non-chronological backtracking
   - Some portion of the decision tree may not be covered
     - Not a complete search anymore
     - May also miss some bugs
     - But may also find bugs earlier



Remember when we talked about

SAT engine,

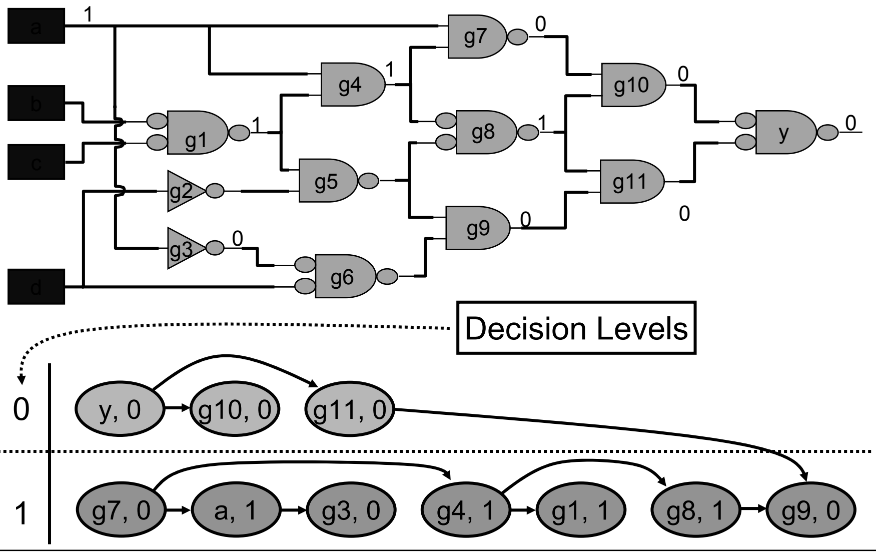
we mentioned that

using “conflict analysis”

we can do non-chronological backtracking, while still achieve complete proof

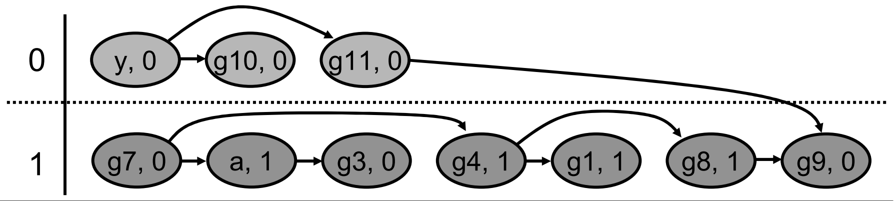
How??

Implication Graph



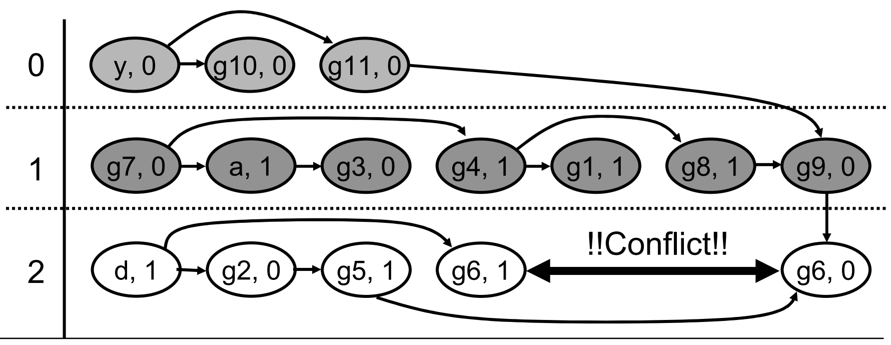
Implication Graph (an exemplar implementation)

- 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)

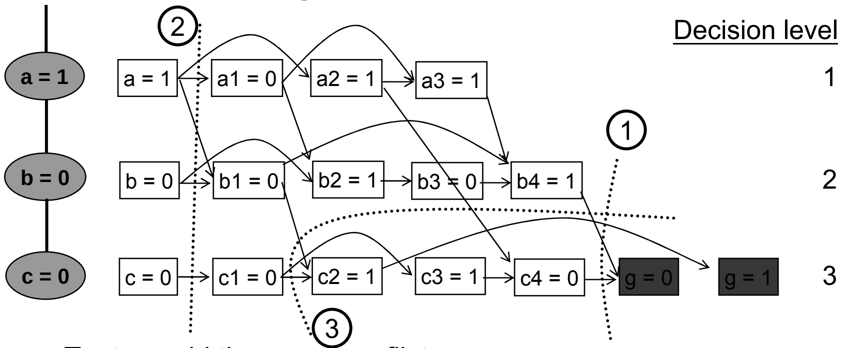


Conflict Analysis

- 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

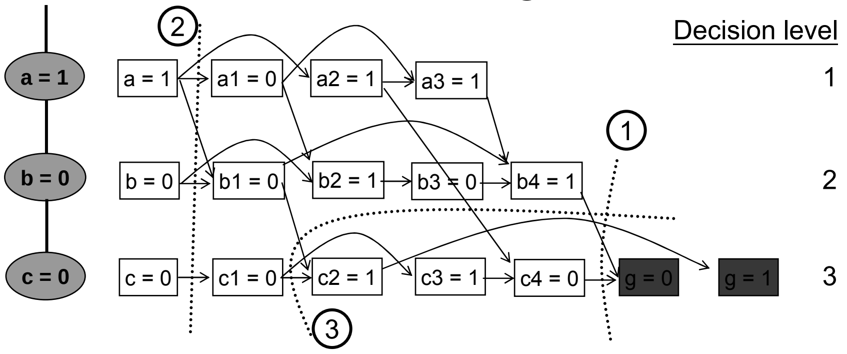


Conflict Analysis



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 is a set of conflict causes
   - Add a constraint for the conflict causes to prevent the conflict from happening again

Conflict-Driven Learning



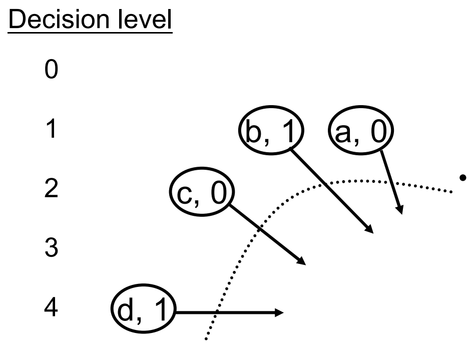
- Add a constraint to prevent the same conflict
  1. b4 && c2 && !c4 = 0;
  2. a && !b && !c = 0;
  3. b4 && a2 && !b1 && !c1 = 0;
- For SAT, add a learned clause (!l1 + !l2 + … + !ln)

Conflict-Driven Learning

- ◆ But which constraint to add?
- ◆ [Zhang, *et al*, ICCAD 2001] Experiment shows that "first-UIP" (1-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
    - ▪ 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

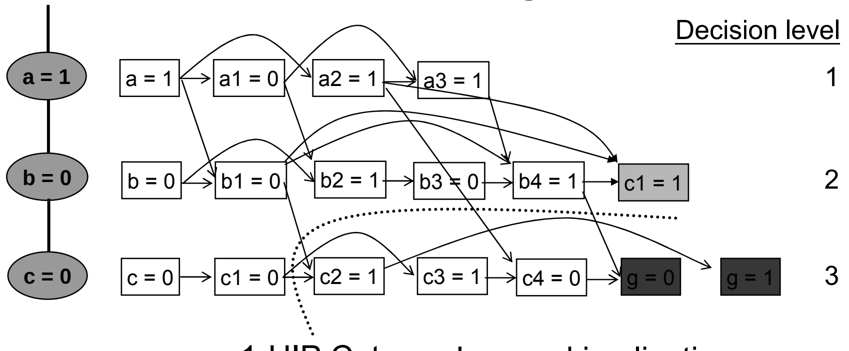
⇩

(!a && b && !c) ➔ !d

⇩

- • If we backtrack to the max decision level of { a, b, c }
  1. { a, b, c } still have the orig implications
  2. d can be implied with the opposite value at the max level above

Conflict-Driven Learning



1-UIP Cut

Learned implication

!b1 && a2 && b4 ➔ c1

Conflict-Driven Non-Chronological Backtracking --- Algorithm

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

Conflict-Driven Non-Chronological Backtracking --- Properties

◆ A complete (exhaustive) proof

◆ Not a binary decision tree anymore…

- Becomes a decision queue
- Conflict → Learned gate
  → Learned implication
- Keeps adding learned gates
  - Implication may be slowed down due to increasing #gates
  - Re-synthesize the learned gates???

Conflict-Driven Non-Chronological Backtracking --- Properties (cont'd)

◆ 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 TPG/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

Amazing Beauty

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

Speed-ups on branch-and-bound process

1. Better decision ordering
2. Learning
3. Bound earlier

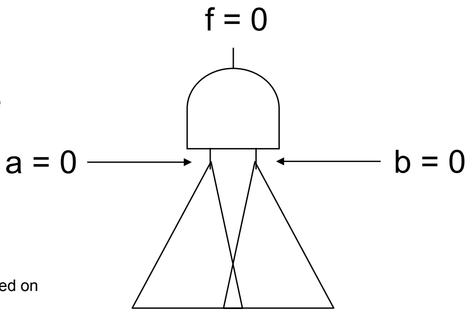
The constraints learned by conflict analysis are sometimes called

"dynamic learning"

because they are learned during the proof process

Another Kind of Dynamic Learning: 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 $S = S_a \cap S_b$
    $\Rightarrow$ (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

Complexity Analysis

1. Conflict analysis
   - O(N), N is the number of nodes in the last decision level of the implication graph
2. Recursive learning
   - $O(M^L)$, M is the complexity of BCP, and L is the level of recursion

◆ [Think] Recursive learning seems very expensive, how can it be useful??

Other than
"dynamic learning",

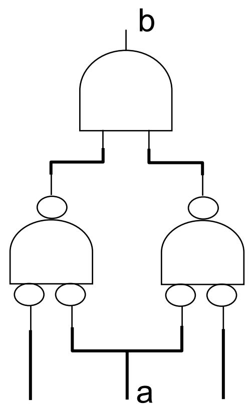
we can also preprocess the circuit to derive some
"static learning"
information

Static Learning (1)

◆ 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

Static Learning (2)

- ◆ Proof-based
  - 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

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

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

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 increased emphasis on "locality" of decisions
2. With conflict analysis, decision can restart any time
   - Change to different decision ordering heuristic to explore different area in the input space
3. Re-synthesis the learned information
   - Two-level to multi-level circuit
   - Any other idea?

Conflict vs. Success-Driven Learning

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

- How to record the solutions?
  - Hash table?
- 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

What we will cover in this topic ---

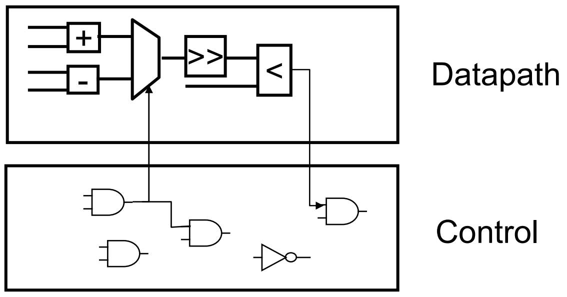
1. Circuit modeling summary
   - Assertion property modeling
2. Combinational ATPG/SAT algorithms
   - Logic implications
   - Branch and bound process
   - Learning
   - Dynamic decision ordering
3. Word-level ATPG
4. RAMBO

Word-Level ATPG

- ◆ Motivations
  - High-level (RTL and up) design usually contains many word-level constructs like “adder”, “data bus”, etc
  - Arithmetic constraints are best solved by arithmetic methods (e.g. Linear algebra)
- ◆ Observations
  - Control signals are usually good candidates for earlier decisions
  - Signals on a data bus usually share same operators
    - Better treated as a word rather than several bits

Word-Level ATPG

1. Divide circuit into “control” and “datapath”



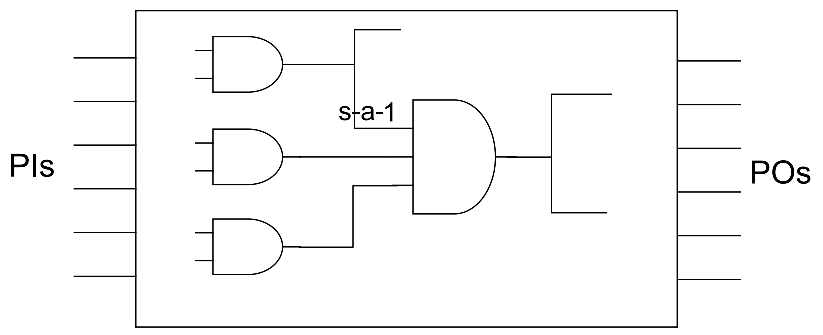
2. Apply branch-and-bound algorithm on control circuit first
3. Solve the remaining datapath equations by arithmetic techniques
   - If solved, done
   - Else find another solution from the control circuit and repeat 3

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

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

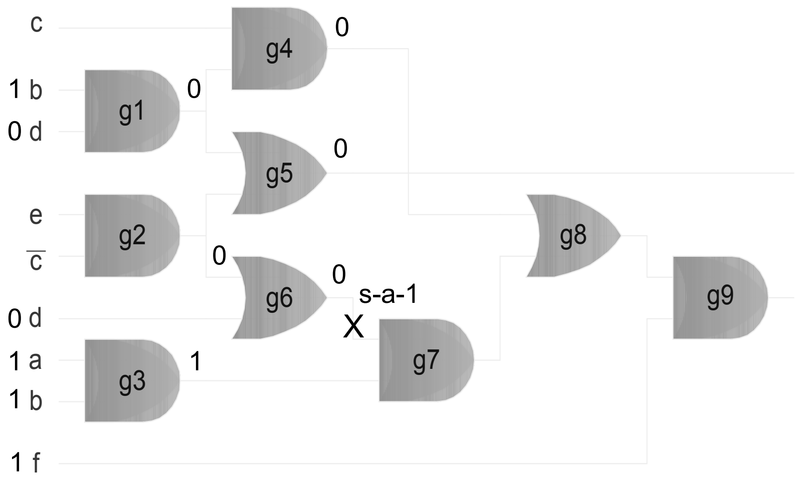
Redundancy in a Combinational Circuit

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



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
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

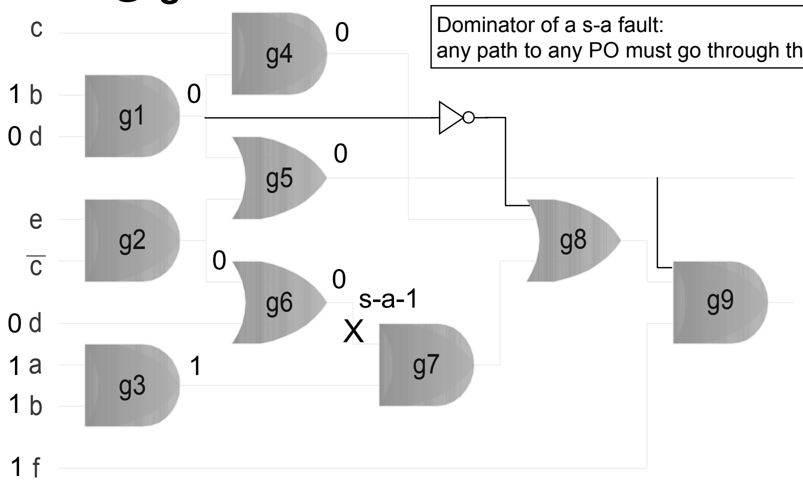
Target: remove g6



g6 is NOT redundant

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

Dominator of a s-a fault:
any path to any PO must go through this gate



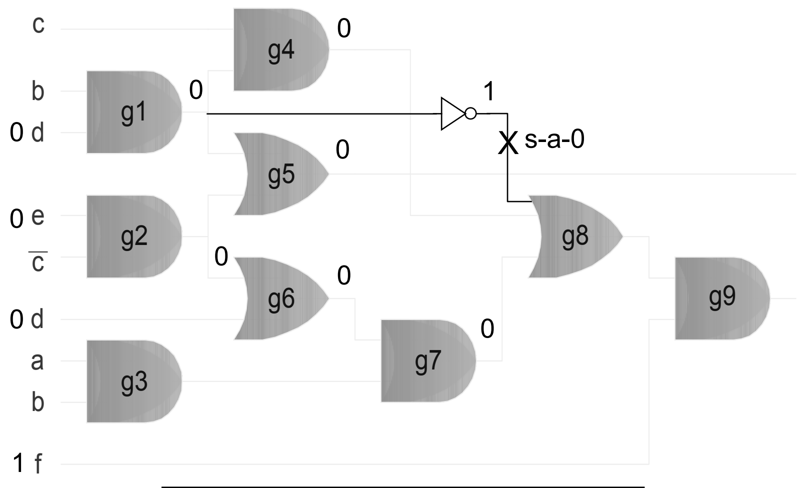
Adding a wire (with inverter) from any implied gate to a dominator

RAMBO Algorithm

[Cheng et.al. TCAD 1995]

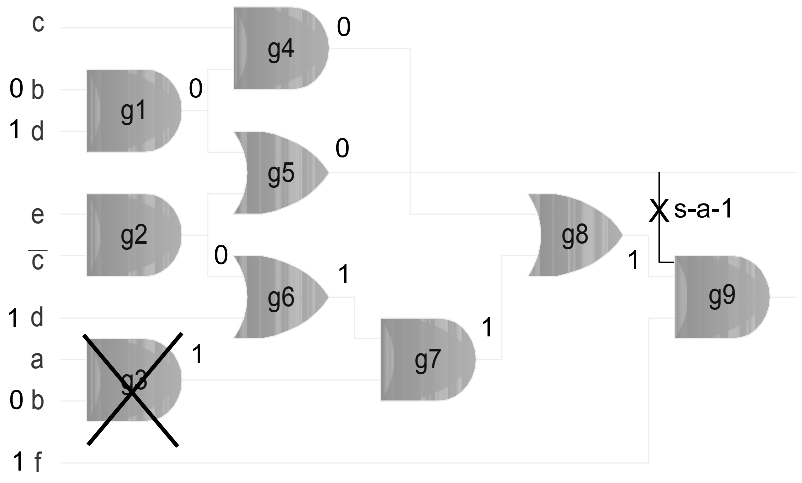
1. Given a target wire, perform its mandatory assignments (MA) for its corresponding s-a fault
   - MA = Implications of
     1. Fault sensitization @ fault site
     2. Fault propagation @ the side inputs of the dominators
2. For each gate g_m in the set of MA,
   For each dominator g_d, test the fault on the added wire ($g_m \rightarrow g_d$)
   a. If value(g_m) = 0 and g_d is an AND ➔ direct connection, s-a-1 fault
   b. If value(g_m) = 1 and g_d is an AND ➔ add an inverter, s-a-1 fault
   c. If value(g_m) = 0 and g_d is an OR ➔ add an inverter, s-a-0 fault
   d. If value(g_m) = 1 and g_d is an OR ➔ direct connection, s-a-0 fault
3. Any untestable fault in 2.a ~ 2.d corresponds to a redundant wire to remove the target wire

Is (!g1 → g8) redundant?



No, (!g1 → g8) is NOT redundant

Is (g5 → g9) redundant?



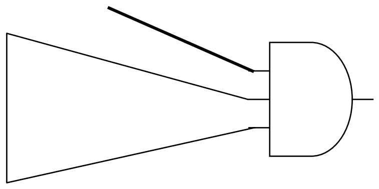
Yes, (g5 → g9) is redundant

RAMBO Algorithm Complexity

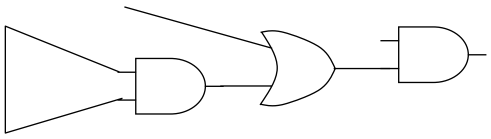
- Need to perform (M * D) redundancy tests
  - M: number of gates in MA
  - D: number of dominators
  - ➔ Could be a BIG number
- "Perturb and Simplify" (Chang, et. al. TCAD 1996)
  - Propose several rules to filter out impossible candidates
  - ➔ Possibly some big number

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

- Add a wire

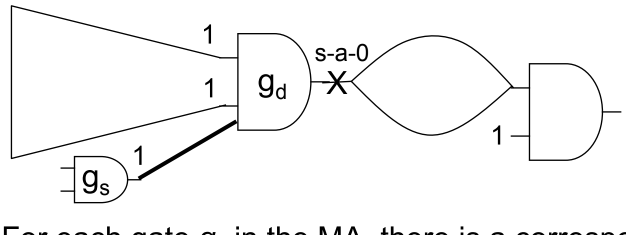


- Add a gate



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

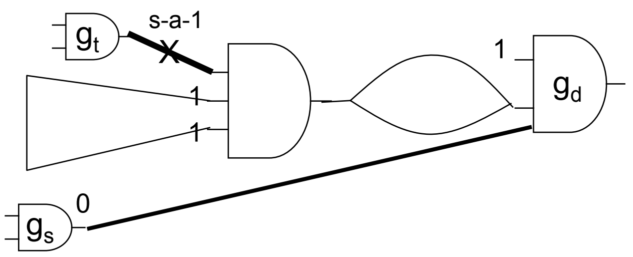


  3. For each gate g_s in the MA, there is a corresponding redundant wire (or with inverter) to g_d

Why??

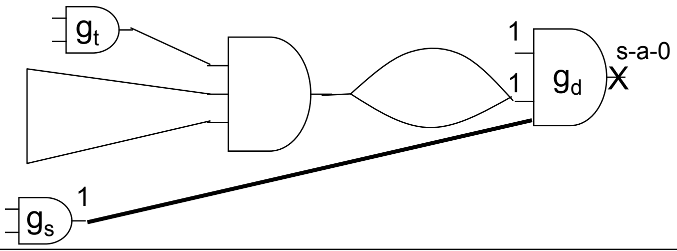
An Efficient Redundancy Addition and Removal Algorithm

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



An Efficient Redundancy Addition and Removal 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



An Efficient Redundancy Addition and Removal Algorithm

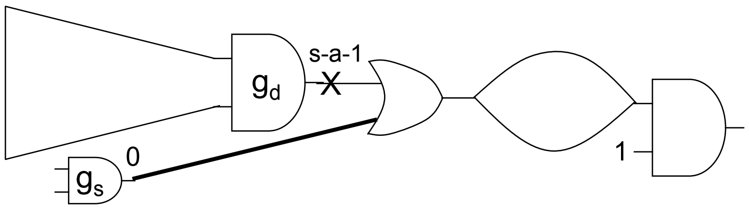
1. Given a target wire on g_t, perform MA(g_t)
   - Adding a wire from a gate g_s in MA(g_t) to any of its dominator g_d can make this target wire redundant
   - e.g. value(g_s) = 0 → AND gate g_d
2. Given a destination gate g_d(dominator of the target wire g_t), perform MA(g_d)
   - Any wire from a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

➔ Perform an intersection on MA(g_t) and MA(g_d),
any contradiction on g_s, implies an alternative wire (g_s → g_d) for the target wire on g_t

[ref: "A New Reasoning Scheme for Efficient Redundancy Addition and Removal", Chang, et. al. TCAD 2003]

Add a Redundant Gate

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

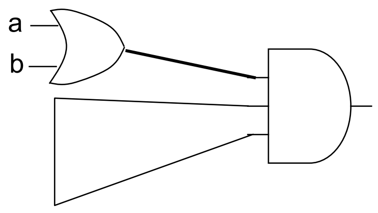


  3. For each gate g_s in the MA, there is a corresponding redundant gate (or with inverter) on g_d

Why??

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

- ◆ Add a wire
- ◆ Add a gate @ destination
- ◆ Add a gate @ source



- ◆Add a circuit (How??)