



---

# Metodologie di progetto HW

## La verifica di circuiti digitali

Versione del 28/04/08

---



---

# Metodologie di progetto HW

## La verifica di circuiti digitali

Equivalence checking Basics

---



## ROBDD's (The Canonical Side)

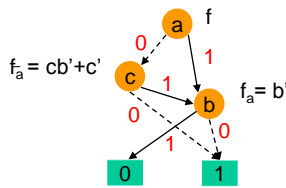
- Representation of a logic function as graph (DAG):
  - many logic functions can be represented compactly - usually better than SOP's
- Can be made **canonical** !!
- Many logic operations can be performed efficiently on BDD's:
  - usually linear in size of result - tautology and complement are constant time
- Size of BDD critically dependent on variable ordering

- 3 -



## Onset is Given by all Paths to "1"

$F = b' + a'c' = ab' + a'cb' + a'c'$  all paths to the 1 node



### Notes:

- By tracing paths to the 1 node, we get a **cover** of pair wise **disjoint** cubes.
- The power of the BDD representation is that it does **not** explicitly enumerate all paths; rather it represents paths by a graph whose size is measured by its nodes and not paths.
- A DAG can represent an **exponential number of paths** with a linear number of nodes.
- BDDs can be used to efficiently represent sets
  - interpret elements of the onset as elements of the set
  - $f$  is called the **characteristic function** of that set

- 4 -



## ROBDD's

- Directed acyclic graph (DAG)
- one root node, two terminals 0, 1
- each node, two children, and a variable
- Shannon co-factoring tree, except **reduced** and **ordered** (ROBDD)
 

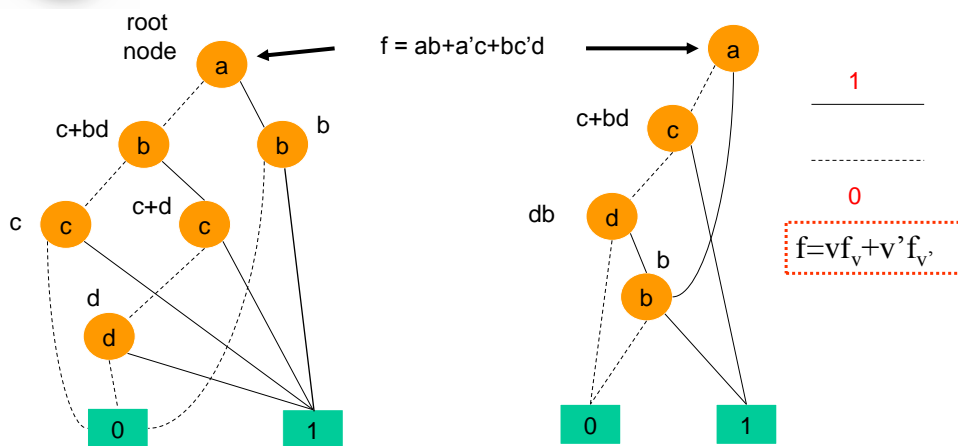
$f = vf_v + v'f_v'$

  - **Reduced:**
    - any node with two identical children is removed
    - two nodes with isomorphic BDD's are merged
  - **Ordered:**
    - Co-factoring variables (splitting variables) always follow the **same order along all paths**
$$x_{i_1} < x_{i_2} < x_{i_3} < \dots < x_{i_n}$$

- 5 -



## Example



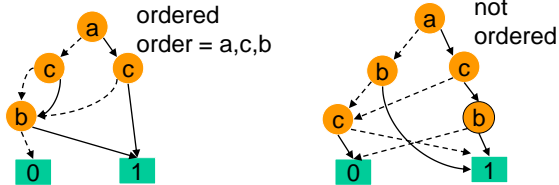
Two different orderings, same function.

- 6 -



## ROBDD

**Ordered BDD (OBDD)** Input variables are ordered - each path from root to sink visits nodes with labels (**variables**) in ascending order.



**Reduced Ordered BDD (ROBDD)** - reduction rules:

1. if the two children of a node are the **same**, the node is eliminated:  $f = vf + v'f$
2. if two nodes have **isomorphic** graphs, they are replaced by one of them

These two rules make it so that each node represents a distinct logic function.

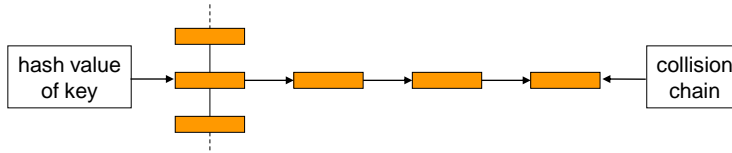
- 7 -



## Efficient Implementation of BDD's

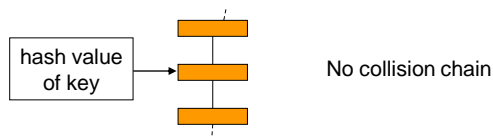
**Unique Table:**

- avoids duplication of existing nodes
  - Hash-Table:  $\text{hash-function}(\text{key}) = \text{value}$



**Computed Table:**

- avoids re-computation of existing results

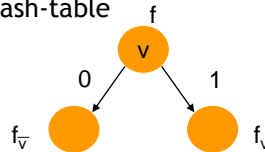


- 8 -



## Efficient Implementation of BDD's

- BDDs is a compressed Shannon co-factoring tree:
  - $f = v f_v + v' f_{v'}$
  - leaves are constants "0" and "1"
- Three components make ROBDDs canonical (Proof: Bryant 1986):
  - unique nodes for constant "0" and "1"
  - identical order of case splitting variables along each paths
  - hash table that ensures:
    - $(\text{node}(f_v) = \text{node}(g_v)) \wedge (\text{node}(f_{v'}) = \text{node}(g_{v'})) \Rightarrow \text{node}(f) = \text{node}(g)$
  - provides recursive argument that  $\text{node}(f)$  is unique when using the unique hash-table



- 9 -



## Recursive Formulation of ITE

$v$  = top-most variable among the three BDD's  $f, g, h$

$$\begin{aligned}
 \text{ite}(f, g, h) &= f g + \overline{f} h \\
 &= v(f g + \overline{f} h)_v + \overline{v}(f g + \overline{f} h)_{\overline{v}} \\
 &= v(f_v g_v + \overline{f}_v h_v) + \overline{v}(f_{\overline{v}} g_{\overline{v}} + \overline{f}_{\overline{v}} h_{\overline{v}}) \\
 &= \text{ite}(v, \text{ite}(f_v, g_v, h_v), \text{ite}(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}})) \\
 &= (v, \text{ite}(f_v, g_v, h_v), \text{ite}(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}}))
 \end{aligned}$$

- 10 -



## Recursive Formulation of ITE

```

Algorithm ITE(f, g, h)
  if(f == 1) return g
  if(f == 0) return h
  if(g == h) return g

  if((p = HASH_LOOKUP_COMPUTED_TABLE(f,g,h)) return p
  v = TOP_VARIABLE(f, g, h) // top variable from f,g,h
  fn = ITE(fv,gv,hv) // recursive calls
  gn = ITE(fv,gv,hv)'
  if(fn == gn) return gn // reduction
  if(!(p = HASH_LOOKUP_UNIQUE_TABLE(v,fn,gn)) {
    p = CREATE_NODE(v,fn,gn) // and insert into UNIQUE_TABLE
  }
  INSERT_COMPUTED_TABLE(p,HASH_KEY{f,g,h})
  return p
}

```

- 11 -



## ITE Operator $\text{ite}(f, g, h) = fg + \overline{f}h$

ITE operator can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of  $B^2$ :

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	AND(f, g)	fg	ite(f, g, 0)
0010	f > g	fg'	ite(f, g', 0)
0011	f	f	f
0100	f < g	f'g	ite(f, 0, g)
0101	g	g	g
0110	XOR(f, g)	f ⊕ g	ite(f, g', g)
0111	OR(f, g)	f + g	ite(f, 1, g)
1000	NOR(f, g)	(f + g)'	ite(f, 0, g')
1001	XNOR(f, g)	f ⊕ g	ite(f, g', g')
1010	NOT(g)	g'	ite(g, 0, 1)
1011	f ≥ g	f + g'	ite(f, 1, g')
1100	NOT(f)	f'	ite(f, 0, 1)
1101	f ≤ g	f' + g	ite(f, g, 1)
1110	NAND(f, g)	(fg)'	ite(f, g', 1)
1111	1	1	1

- 12 -



## Traversal Procedure: Boolean AND

```

bdd AND( bdd A, bdd B )
{
  (1) if ( A == 0 ) return 0;   if ( B == 0 ) return 0;
      if ( A == 1 ) return B;   if ( B == 1 ) return A;
      if ( A == B ) return A;   if ( A == B' ) return 0;
  (2) cache lookup
  (3) (A0,A1)=Cofactors(A,x); (B0,B1)=Cofactors(B,x);
  (4) R0 = AND( A0, B0 );      R1 = AND( A1, B1 );
  (5) R = ITE( x, R1, R0 );
  (6) cache insert
  (7) return R;
}

```

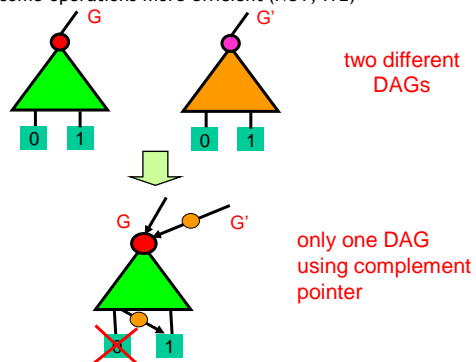
- 13 -



## Extension - Complement Edges

Combine inverted functions by using complemented edge

- similar to circuit case
- reduces memory requirements
- BUT MORE IMPORTANT:
  - makes some operations more efficient (NOT, ITE)

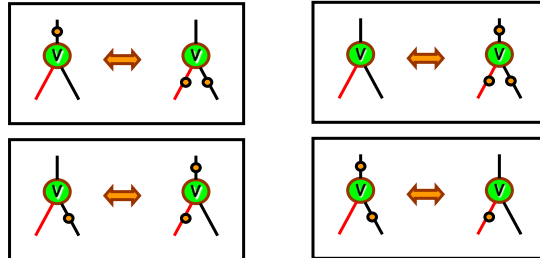


- 14 -



## Extension - Complement Edges

To maintain strong canonical form, need to resolve 4 equivalences:



**Solution:** Always choose one on **left**, i.e. the “then” leg must have **no** complement edge.

- 15 -



## Garbage Collection

- Very important to free and reuse memory of unused BDD nodes
  - explicitly free'd by an external `bdd_free` operation
  - BDD nodes that were temporary created during BDD operations
- Two mechanism to check whether a BDD is not referenced:
  - **Reference counter** at each node
    - increment whenever node gets one more reference (incl. External)
    - decrement when node gets de-references (`bdd_free` from external, de-reference from internal)
    - counter-overflow -> freeze node
  - **Mark and Sweep** algorithm
    - does not need counter
    - first pass, mark all BDDs that are referenced
    - second pass, free the BDDs that are not marked
    - need additional handle layer for external references

- 16 -



## Garbage Collection

---

- Timing is very crucial because garbage collection is expensive
    - immediately when node gets free'ed
      - bad because dead nodes get often reincarnated in next operation
    - regular garbage collections based on statistics collected during BDD operations
    - “death row” for nodes to keep them around for a bit longer
  
  - Computed table must be cleared since not used in reference mechanism
  
  - Improving memory locality and therefore cache behavior:
    - sort free'ed BDD nodes to
- 

- 17 -



## BDD Derivatives

---

- MDD: Multi-valued BDDs
    - natural extension, have more than two branches
    - can be implemented using a regular BDD package with binary encoding
      - advantage that binary BDD variables for one MV variable do not have to stay together -> potentially better ordering
  - ADDs: (Analog BDDs) MTBDDs
    - multi-terminal BDDs
    - decision tree is binary
    - multiple leafs, including real numbers, sets or arbitrary objects
    - efficient for matrix computations and other non-integer applications
  - FDDs: Free BDDs
    - variable ordering differs
    - not canonical anymore
  
  - and many more ....
- 

- 18 -

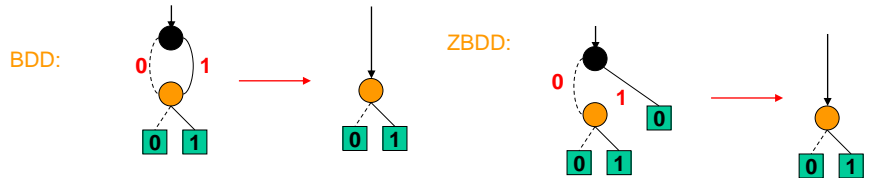


## Zero Suppressed BDD's - ZBDD's

ZBDD's were invented by **Minato** to efficiently represent **sparse** sets. They have turned out to be **useful** in implicit methods for representing primes (which usually are a sparse subset of all cubes).

Different reduction rules:

- **BDD**: eliminate all nodes where **then** edge and **else** edge point to the same node.
- **ZBDD**: eliminate all nodes where the **then** node points to 0. Connect incoming edges to **else** node.
- **For both**: share equivalent nodes.



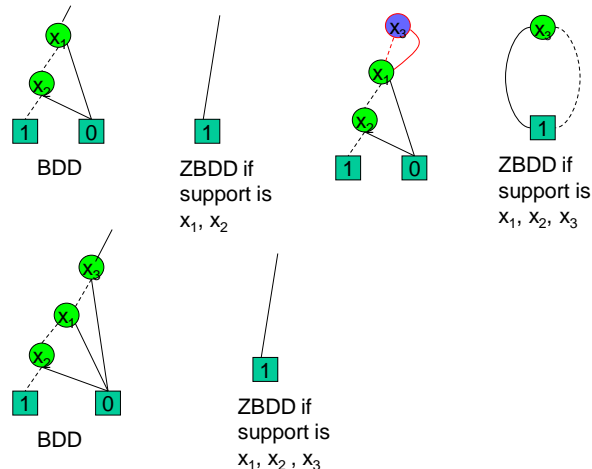
- 19 -



## Canonicity

**Theorem: (Minato)** ZBDD's are canonical given a variable ordering and the support set.

Example:

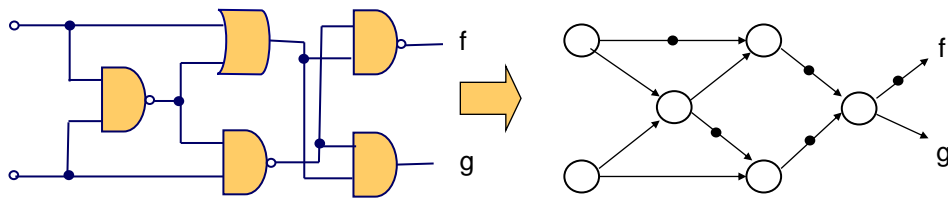


- 20 -



## AND-INVERTER Circuits

- Base data structure uses two-input AND function for vertices and INVERTER attributes at the edges (individual bit)
  - use De'Morgan's law to convert OR operation etc.
- Hash table to identify and reuse structurally isomorphic circuits



- 21 -



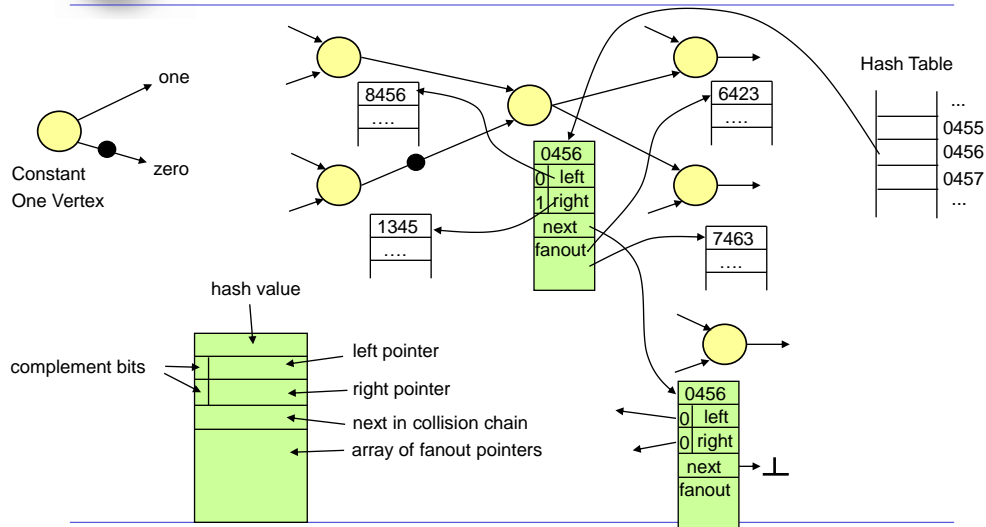
## Data Representation

- Vertex:
  - pointers (integer indices) to left and right child and fanout vertices
  - collision chain pointer
  - other data
- Edge:
  - pointer or index into array
  - one bit to represent inversion
- Global hash table holds each vertex to identify isomorphic structures
- Garbage collection to regularly free un-referenced vertices

- 22 -



## Data Representation



- 23 -



## Hash Table

```

Algorithm HASH_LOOKUP(Edge p1, Edge p2) {
    index = HASH_FUNCTION(p1,p2)
    p     = &hash_table[index]
    while(p != NULL) {
        if(p->left == p1 && p->right == p2) return p;
        p = p->next;
    }
    return NULL;
}

```

Tricks:

- keep collision chain sorted by the address (or index) of p
  - that reduces the search through the list by 1/2
- use memory locations (or array indices) in topological order of circuit
  - that results in better cache performance

- 24 -



## Basic Construction Operations

---

```
Algorithm AND(Edge p1,Edge p2){
  if(p1 == const1) return p2
  if(p2 == const1) return p1
  if(p1 == p2)      return p1
  if(p1 == ^p2)    return const0
  if(p1 == const0 || p2 == const0) return const0

  if(RANK(p1) > RANK(p2)) SWAP(p1,p2)

  if((p = HASH_LOOKUP(p1,p2)) return p
  return CREATE_AND_VERTEX(p1,p2)
}
```

---

- 25 -



## Basic Construction Operations

---

```
Algorithm NOT(Edge p) {
  return TOGGLE_COMPLEMENT_BIT(p)
}

Algorithm OR(Edge p1,Edge p2){
  return (NOT(AND(NOT(p1),NOT(p2))))
}
```

---

- 26 -



## SAT and Tautology

---

- **Tautology:**
  - Find an assignment to the inputs that evaluate a given vertex to “0”.
- **SAT:**
  - Find an assignment to the inputs that evaluate a given vertex to “1”.
  - Identical to Tautology on the inverted vertex
- **SAT on circuits is identical** to the justification part in ATPG
  - First half of ATPG: justify a particular circuit vertex to “1”
  - Second half of ATPG (propagate a potential change to an output) can be easily formulated as SAT (will be covered later)
- **Basic SAT algorithms:**
  - branch and bound algorithm as seen before
    - branching on the assignments of primary inputs only (Podem algorithm)
    - branching on the assignments of all vertices (more efficient)

---

- 27 -



## General Davis-Putnam Procedure

---

- search for **consistent** assignment to entire cone of requested vertex by systematically trying all combinations (**may be partial!!!**)
- keep a queue of vertices that remain to be justified
  - pick **decision vertex** from the queue and case split on possible assignments
  - for each case
    - propagate as many implications as possible
      - generate more vertices to be justified
      - if conflicting assignment encountered
        - » undo all implications and backtrack
    - recur to next vertex from queue

```
Algorithm SAT(Edge p) {  
    queue = INIT_QUEUE()  
    if (IMPLY(p) && QUEUE_EMPTY(queue)) return TRUE  
    return JUSTIFY(queue)  
}
```

---

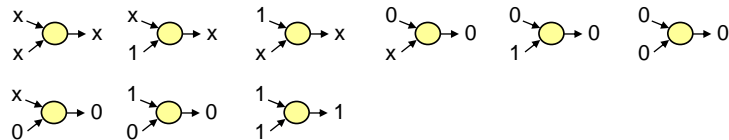
- 28 -



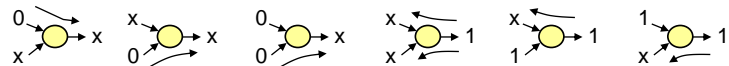
## Implication Procedure

- Fast implication procedure is key for efficient SAT solver!!!
  - don't move into circuit parts that are not sensitized to current SAT problem
  - detect conflicts as early as possible
- Table lookup implementation (27 cases):

- No implications:



- Implications:

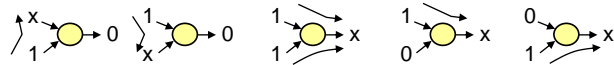


- 29 -

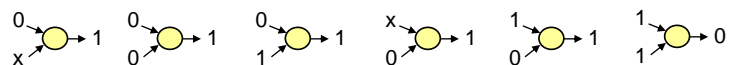


## Implication Procedure

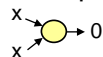
- Implications:



- Conflicts:



- Case Split:



- 30 -



## imply

```
Algorithm imply(Edge p) {
  assign(p,1)
  next_state = lookup(p)
  switch(next_state)
  {
    case CONFLICT: return FALSE;
    case CASE_SPLIT:
      add_vertex_to_queue(p,justification_queue);
      return TRUE;
    case NO_IMPLICATION: return TRUE;
    case PROP_BACK_LEFT_RIGHT:
      Edge lvalue = get_value(p->left);
      Edge rvalue = get_value(p->right);
      return imply(lvalue) && imply(rvalue);
    ...
  }
}
```

- 31 -



## General Davis-Putnam Procedure

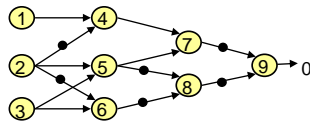
```
Algorithm JUSTIFY(queue) {
  if(QUEUE_EMPTY(queue)) return TRUE
  mark = ASSIGNMENT_MARK()
  v = QUEUE_NEXT(queue) // decision vertex
  if(IMPLY(NOT(v->left))) {
    if(JUSTIFY(queue)) return TRUE
  } // conflict
  UNDO_ASSIGNMENTS(mark)
  if(IMPLY(v->left)) {
    if(JUSTIFY(queue)) return TRUE
  } // conflict
  UNDO_ASSIGNMENTS(mark)
  return FALSE
}
```

- 32 -



## Example

SAT(NOT(9))??



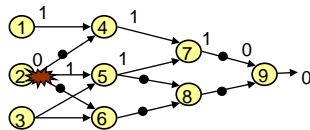
Queue

9

Assignments

9

First case for 9:



9

9  
7  
4  
5  
1  
2

**Conflict!!**

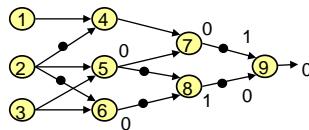
- undo all assignments
- backtrack

- 33 -



## Example

Second case for 9:



Note:

- vertex 7 is justified by 8->5->7

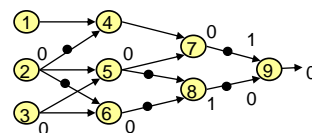
Queue

5  
6

Assignments

9  
7  
8  
5  
6

First case for 5:



9  
7  
8  
5  
6  
2  
3

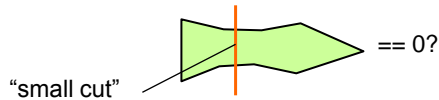
**Solution cube: 1 = x, 2 = 0, 3 = 0**

- 34 -



## Ordering of Case Splits

- various heuristics work differently well for particular problem classes
- often **depth-first heuristic** good because it generates conflicts quickly
- mixture of **depth-first and breadth-first** schedule
- other heuristics:
  - pick the vertex with the **largest fanout**
  - count the **polarities** of the fanout **separately** and pick the vertex with the highest count in either one
  - run a **full implication** phase on all outstanding case splits and count the number of implications one would get
    - some cases may already generate conflicts, the other case is immediately implied
- pick vertices that are involved in **small cut** of the circuit

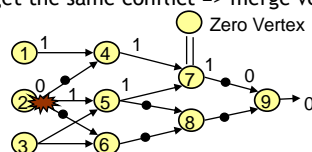


- 35 -



## Learning

- Learning is the process of adding “shortcuts” to the circuit structure that avoids case splits
  - static learning:
    - global implications are learned
  - dynamic learning:
    - learned implications only hold in current part of the search tree
- Learned implications are stores as additional network
- Back to example:
  - First case for vertex 9 lead to conflict
  - If we were to try the same assignment again (e.g. for the next SAT call), we would get the same conflict => merge vertex 7 with Zero-vertex



- if rehashing is invoked  
vertex 9 is simplified and  
and merged with vertex 8

- 36 -



## Static Learning

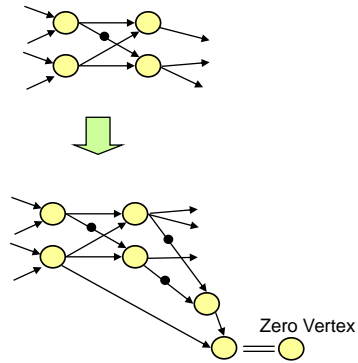
- Implications that can be learned structurally from the circuit
  - Example:  $((x \wedge y) \equiv 0) \wedge (x \wedge \bar{y}) \equiv 0 \Rightarrow (x \equiv 0)$

- Add learned structure as circuit

Use hash table to find structure in circuit:

```

Algorithm CREATE_AND(p1,p2) {
  . . . // create new vertex p
  if ((p' = HASH_LOOKUP(p1, NOT(p2))) ) {
    LEARN ((p=0) & (p' = 0) => (p1=0))
  }
  if ((p' = HASH_LOOKUP(NOT(p1), p2)) ) {
    LEARN ((p=0) & (p' = 0) => (p2=0))
  }
}
  
```

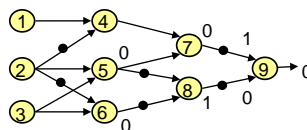


- 37 -



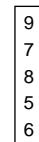
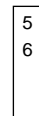
## Back to Example

Original second case for 9:

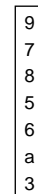
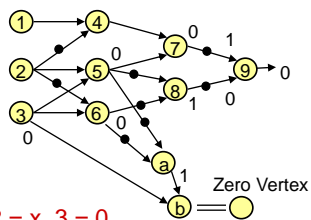


Queue

Assignments



Second case for 9 with static learning:



Solution cube:  $1 = x, 2 = x, 3 = 0$

- 38 -



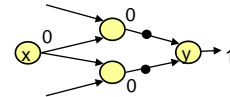
## Static Learning

- Socrates algorithm: based on contra-positive:  
 $(x \Rightarrow y) \Rightarrow (\bar{y} \Rightarrow \bar{x})$

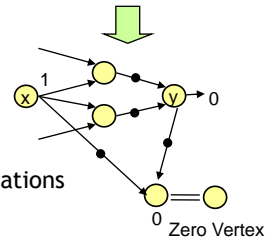
```

foreach vertex v {
  mark = ASSIGNMENT_MARK()
  IMPLY(v)
  LEARN_IMPLICATIONS(v)
  UNDO_ASSIGNMENTS(mark)
  IMPLY(NOT(v))
  LEARN_IMPLICATIONS(NOT(v))
  UNDO_ASSIGNMENTS(mark)
}

```



$$((x = 0) \Rightarrow (y = 1)) \Rightarrow ((y = 0) \Rightarrow (x = 1))$$



- Problem: learned implications are far too many
  - solution: restrict learning to non-trivial implications
  - mask redundant implications

- 39 -



## Metodologie di progetto HW La verifica di circuiti digitali

### BDD Sweeping



## Combining Structural Hashing and BDDs

---

- How can we build a hybrid SAT solver that takes advantage of multiple approaches?
  - Structural methods such as the AND/INVERTER graph are very efficient in making structurally easy decisions
    - However, they often cannot decide a problem
  - BDDs are the “power engine”
    - If we can build them the problem can be decided because of canonicity
    - However, often we cannot and run into a memory blow-up
  - Ultimate solution: Combine multiple approaches (e.g. structural hashing, BDDs, SAT) to cover a larger application range

---

- 51 -



## BDD Sweeping Algorithm

---

- Combines BDD and structural hashing
- Applied internal cutpoints for overlapped BDD propagation
  
- **Basic ingredient:**
  - AND/INVERTER graph
    - hashing of isomorphic structure
  - BDD propagation using a sorted heap (priority queue) controlled by the size of the BDDs
    - propagate smallest BDDs first, don't waste time building large BDD, might not be needed
  - Propagation of multiple BDD layers
    - simulated multiple, independent cut layers
    - avoids to put all bets into one cutset

---

- 52 -



## Heap Based BDD Propagation

BDD sweeping works on AND/INVERTER graph

- idea is to merge Miter from inputs to outputs

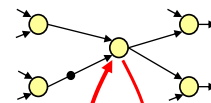
```

Algorithm SWEEP (Edge p) {
  if(p == const1) return SAT
  if(p == const0) return UNSAT

  forall input vertices i do {
    bdd_i = CREATE_BDD_VARIABLE()
    STORE_VERTEX_AT_BDD(bdd_i, i)
    PUT_ON_HEAP(heap, bdd_i)
  }
  return PROCESS_HEAP(heap, p)
}

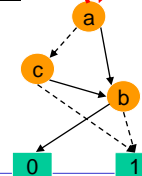
```

AND/INV Graph:



Cross-references

BDD:



- 53 -



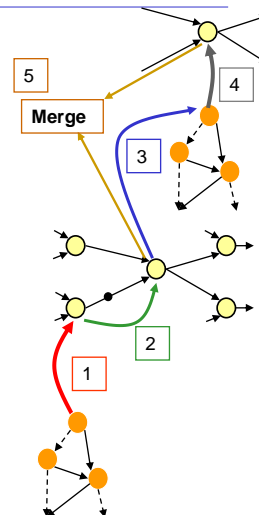
## Heap Based BDD Propagation

```

Algorithm PROCESS_HEAP (heap, Edge p) {
  while(heap != empty) {
    bdd = GET_SMALLEST_BDD(heap)
    v = GET_VERTEX_FROM_BDD(bdd) 1

    forall fanout vertices v_out of v do { 2
      bdd_left = GET_BDD_FROM_VERTEX(v_out->left)
      bdd_right = GET_BDD_FROM_VERTEX(v_out->right)
      bdd_res = BDD_AND(bdd_left, bdd_right)
      if(v_res = GET_VERTEX_FROM_BDD(bdd_res)
        MERGE_VERTICES_AND_HASH(v_res, v_out)
        if(p == const1) return SAT
        if(p == const0) return UNSAT
      }
      else {
        STORE_VERTEX_AT_BDD(bdd_res, v_out)
      }
      PUT_ON_HEAP(heap, bdd_res)
    }
  }
  return undecided;
}

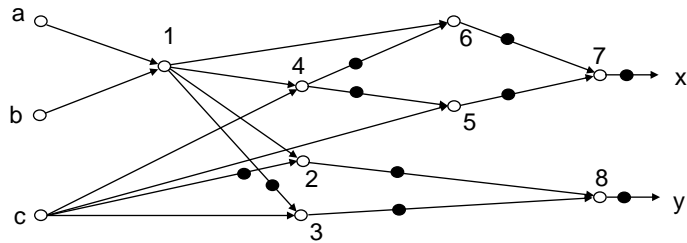
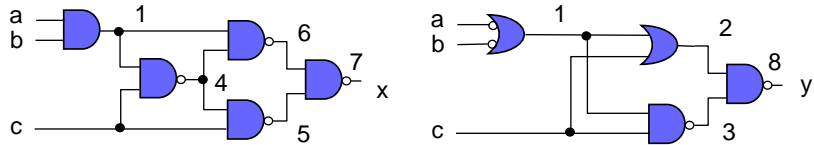
```



- 54 -



## Example

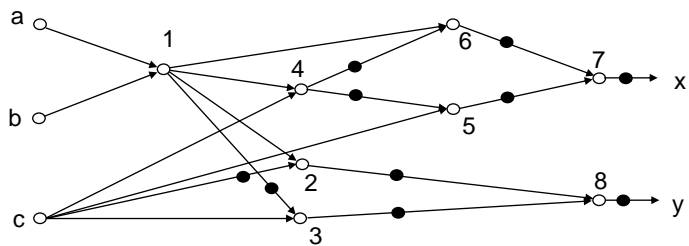


- 55 -



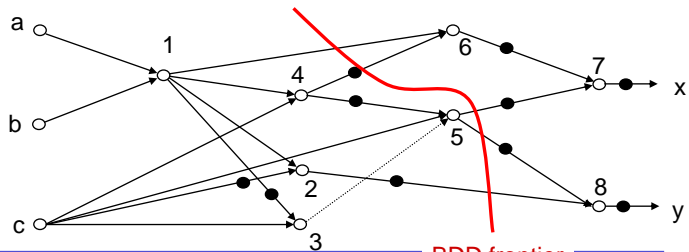
## Example

Original graph:



Graph after processing  
vertices 1,2,3,4,5

5 and 3 get merged



BDD frontier

- 56 -



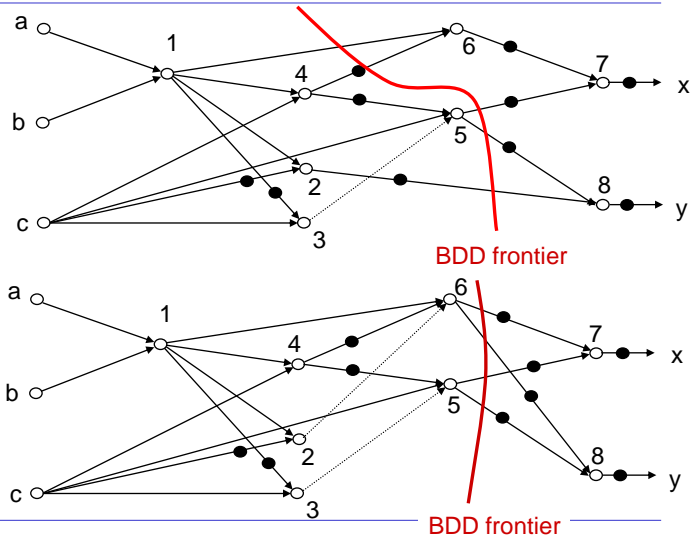
## Example

Graph after processing  
vertices 1,2,3,4,5

5 and 3 get merged

Graph after processing  
additional vertex 6

6 and 2 get merged

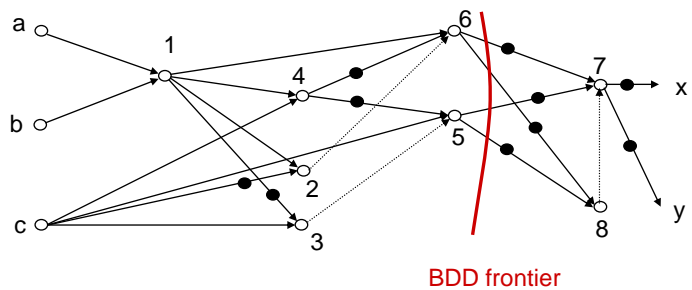


- 57 -



## Example

Hashing kicks in  
and 7 and 8 get merged



Output equivalence is proven without building BDDs for it  
• exponential savings

- 58 -



## BDD Sweeping Algorithm

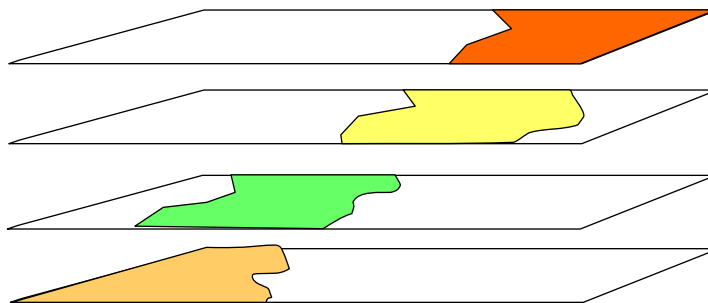
- BDD Sweeping works from the inputs toward the outputs until structural hashing kicks in
- What if logic too deep and no structural equivalence at outputs?
  - Propagate multiple layers of BDD by starting from cone intersection
  - Interleave with SAT solver running from the output
  - Combine with local rewriting to increase structural similarity
- What can we do about false negatives when BDD arrive at output from intermediate cutset
  - Apply BDD\_compose to resubstitute cutpoint variables using a heap based scheme

- 59 -



## Multiple Layer BDD Sweeping

- start new BDD layers at common cut-frontiers
- keep propagating them in parallel from the same heap
- multiple BDD per AND vertex



- 60 -



## BDD Sweeping and SAT

---

- Interleave the application of BDD sweeping and SAT solver
    - control the resources by BDD size limit for the sweeping
    - backtrack limit for the SAT solver
    - do not free BDDs once above size limit, just “hide” them
      - when size limit increases, BDDs “appear” again and sweeping continues
  
  - Effect:
    - Sweeping merges the Miter from the inputs toward the outputs
    - SAT attempts to solve it as early as possible
  
  - Further, interleave with random simulation to find mismatches
- 

- 61 -



## BDD Sweeping + SAT Solver

---

```
Algorithm SWEEP_SAT(Edge p) {
  if(p == const1) return SAT
  if(p == const0) return UNSAT
  forall input vertices i do {
    bdd_i = CREATE_BDD_VARIABLE()
    STORE_VERTEX_AT_BDD(bdd_i,i)
    PUT_ON_HEAP(heap,bdd_i)
  }
  for(size=size_lower_bound,size<size_upper_bound;size++) {
    if((res=PROCESS_HEAP(heap,size,p))!=undecided)
      return res
    if((res=SAT(back_track_limit))!=undecided)
      return res
  }
  return SAT(max_back_track_limit)
}
```

---

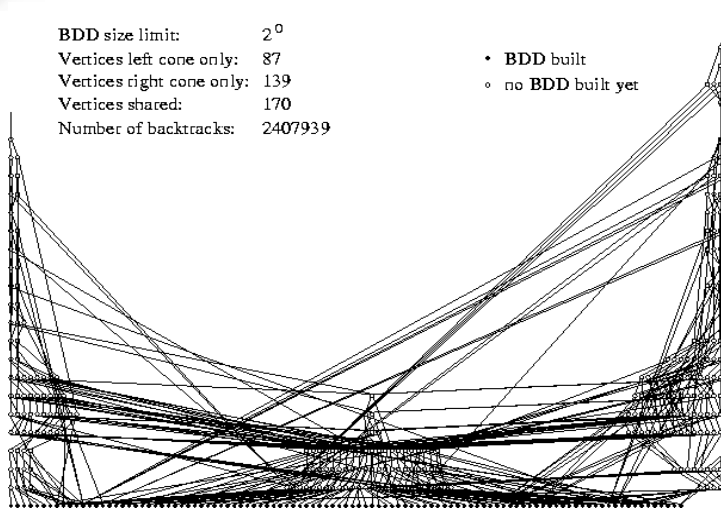
- 62 -



## Equivalence Checking Example No Sweeping

BDD size limit:  $2^{10}$   
Vertices left cone only: 87  
Vertices right cone only: 139  
Vertices shared: 170  
Number of backtracks: 2407939

- BDD built
- no BDD built yet

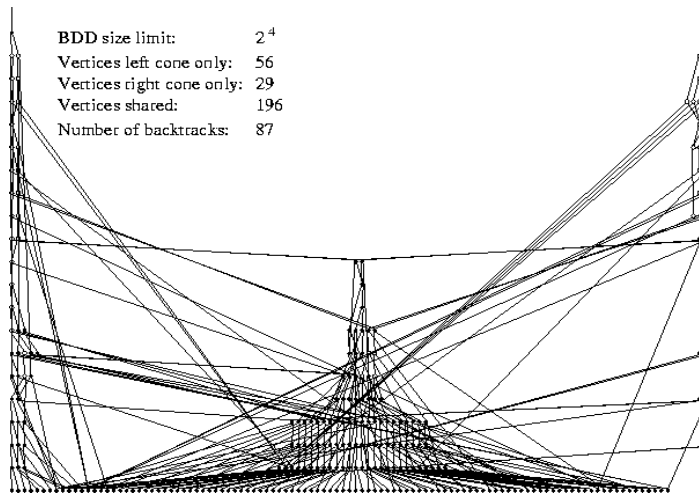


- 63 -



## Sweeping Half-way

BDD size limit:  $2^4$   
Vertices left cone only: 56  
Vertices right cone only: 29  
Vertices shared: 196  
Number of backtracks: 87



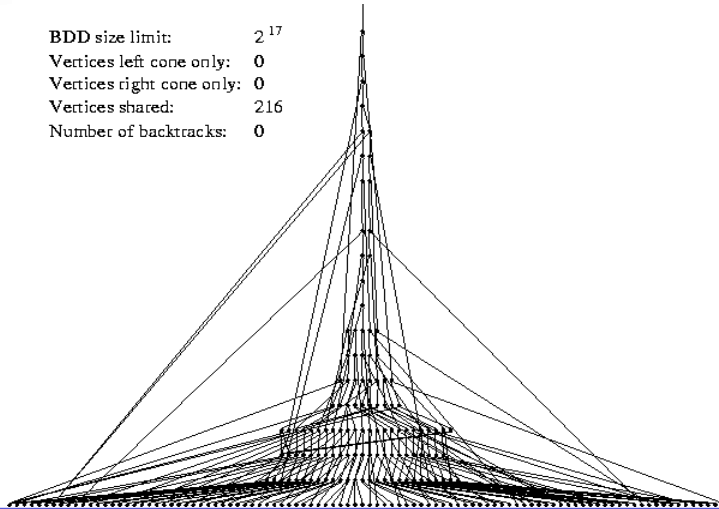
- 64 -



## Full Sweep

---

BDD size limit:  $2^{17}$   
Vertices left cone only: 0  
Vertices right cone only: 0  
Vertices shared: 216  
Number of backtracks: 0



- 65 -



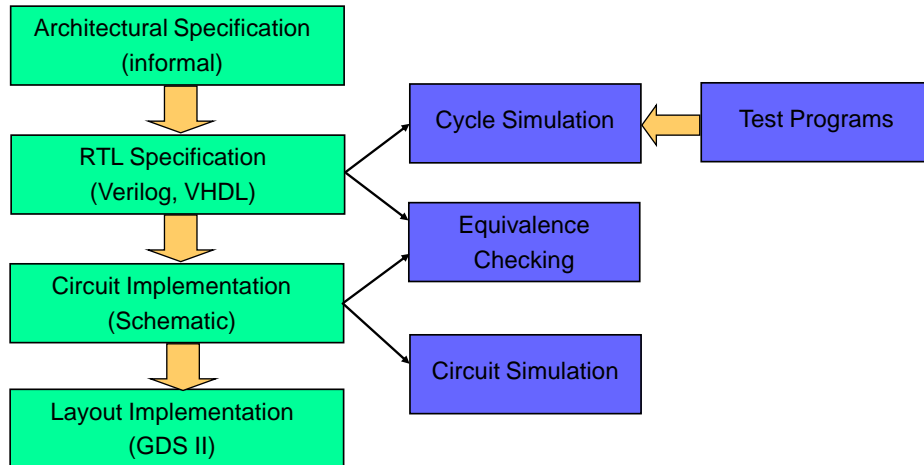
---

## Metodologie di progetto HW La verifica di circuiti digitali

Equivalence checking



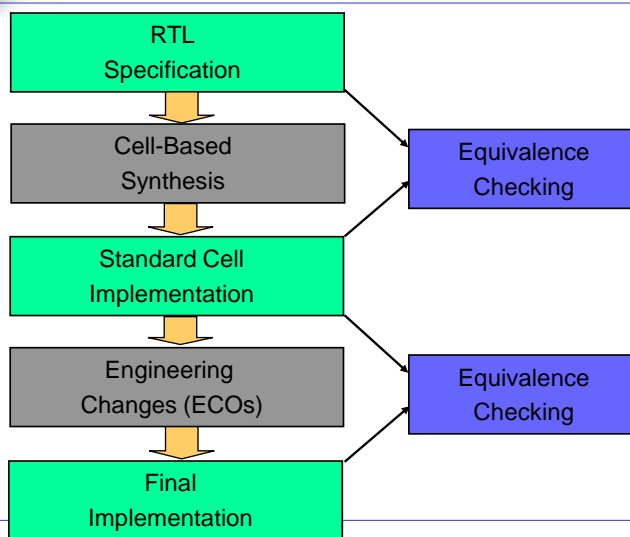
## Application of EC in $\mu$ P Designs



- 67 -



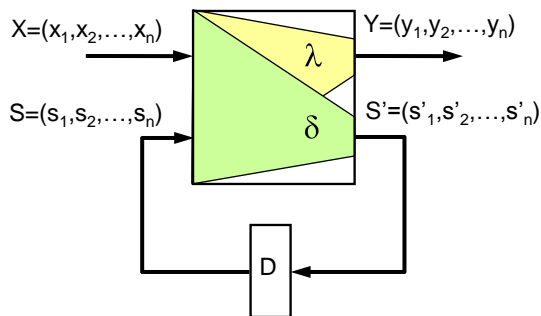
## Application of EC in ASIC Designs



- 68 -



## Basic Model Finite State Machines



$M(X, Y, S, S_0, \delta, \lambda)$ :

X: Inputs

Y: Outputs

S: Current State

$S_0$ : Initial State(s)

$\delta$ :  $X \times S \rightarrow S$  (next state function)

$\lambda$ :  $X \times S \rightarrow Y$  (output function)

Delay element:

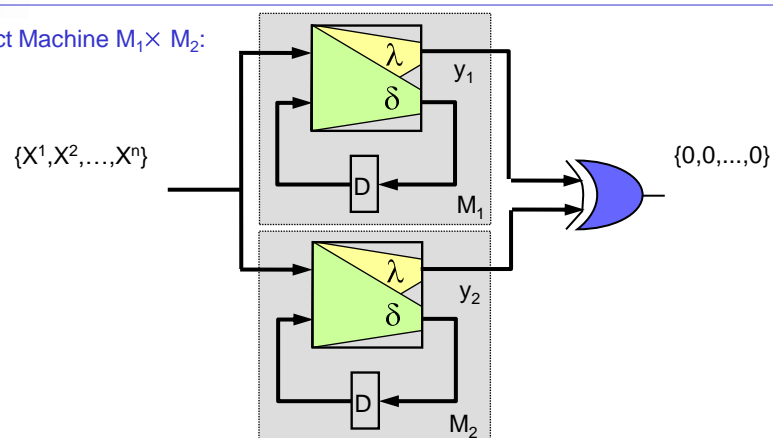
- Clocked: synchronous
  - single-phase clock, multiple-phase clocks
- Unclocked: asynchronous

- 69 -



## Finite State Machines Equivalence

Build Product Machine  $M_1 \times M_2$ :



Definition:

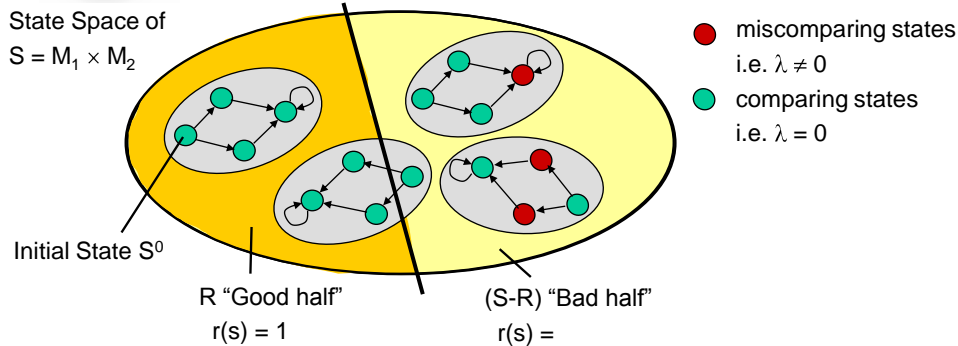
$M_1$  and  $M_2$  are functionally equivalent iff the product machine

$M_1 \times M_2$  produces a constant 0 for all valid input sequences  $\{X_1, \dots, X_n\}$ .

- 70 -



## General Approach to EC



### Inductive proof of equivalence:

Find subset  $R \subseteq S$  with characteristic function  $f: S \rightarrow \{0,1\}$  such that:

1.  $r(s^0) = 1$  (initial state is in good half)
2.  $(r(s) = 1) \Rightarrow r(\delta(x,s)) = 1$  (all states from good half lead go to states in good half)
3.  $(r(s) = 1) \Rightarrow \lambda(x,s) = 0$  (all states in good half are comparing states)



## How Do We Obtain R?

- **Reachability analysis:**
  - state traversal until no more states can be explored
    - forward
    - backward
    - explicit
    - symbolic
- **Relying on the design methodology to provide R:**
  - equivalent state encoding in both machines
  - synthesis tool provides hint for R from sequential optimization
    - manual register correspondence
    - automatic register correspondence
- **Combination of them**



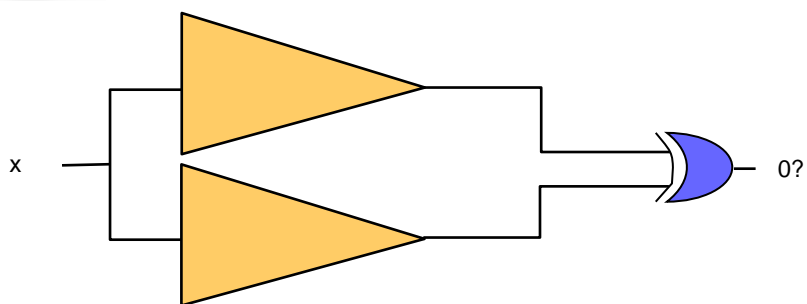
## Combinational EC

- Industrial EC checkers almost exclusively use a combinational EC paradigm
  - sequential EC is too complex, can only be applied to design with a few hundred state bits
  - combinational methods scale linearly with the design size for a given fixed size and “functional complexity” of the individual cones
- Still, pure BDDs are plain SAT solver cannot handle all cones
  - BDDs can be built for about 80% of the cones of high-speed designs
  - less for complex ASICs
  - plain SAT blows up on a “Miter” structure
- Contemporary method highly exploit structural similarity of designs to be compared

- 73 -



## Miter Structure for Combinational EC



Basic methods:

- random simulation, good for finding miscompares
- BDD based and modifications
- structural SAT based with modifications

- 74 -



## History of Equivalence Checking

---

- SAS (IBM 1978 - 1994):
    - standard equivalence checking tool running on mainframes
    - based on the DBA algorithm (“BDDs in time”)
    - verified manual cell-based designs against RTL spec
    - handling of entire processor designs
      - application of “proper cutpoints”
      - application of synthesis routines to make circuits structurally similar
      - special hacks for hard problems
  - BEC (IBM 1991 - 1996):
    - workstation based re-implementation of SAS
    - mainly used in BooleDozer synthesis environment
  - Verity (IBM 1992 - today):
    - originally developed for switch-level designs
    - today IBMs standard EC tool for any combination of switch-, gate-, and RTL designs
- 

- 75 -



## History of Equivalence Checking

---

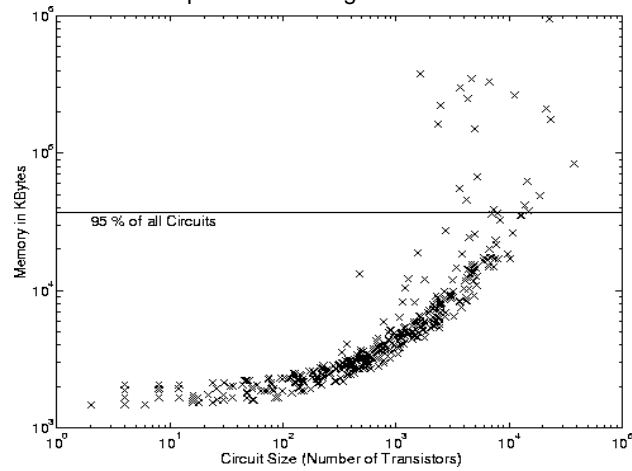
- Chrysalis (1994 - now Avanti):
    - based on ATPG technology and cutpoint exploitation
    - very weak if many cutpoints present
    - did not adopt BDDs for a long time
  - Formality (1997 - Synopsys)
    - multi-engine technology including strong structural matching techniques
  - Verplex (1998)
    - strong multi-engine based tool
    - to our knowledge heavy SAT-based
    - very fast front-end
- 

- 76 -



## Application of Pure BDDs

Statistics on a PowerPC processor design:

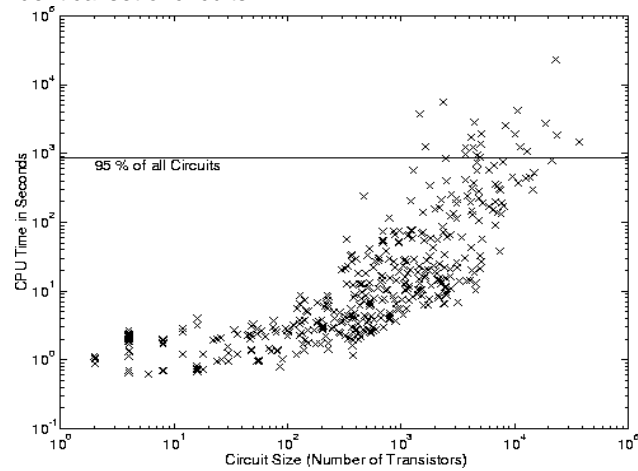


- 77 -



## Application of Pure BDDs

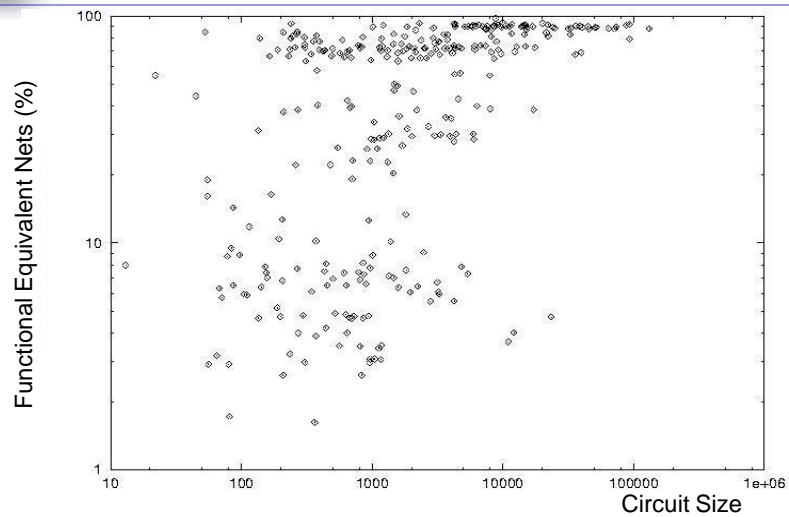
Time for identical set of circuits:



- 78 -



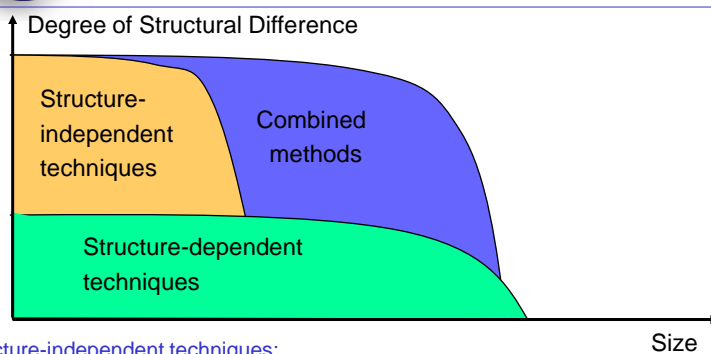
## Structural Similarity



- 79 -



## Methods



Structure-independent techniques:

- exhaustive simulation
- decision diagrams (\*DD\*)

Structure dependent techniques:

- graph hashing
- SAT solvers including learning techniques

- 80 -

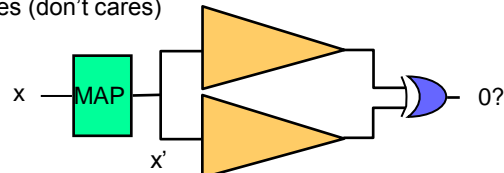


## Handling Constraints

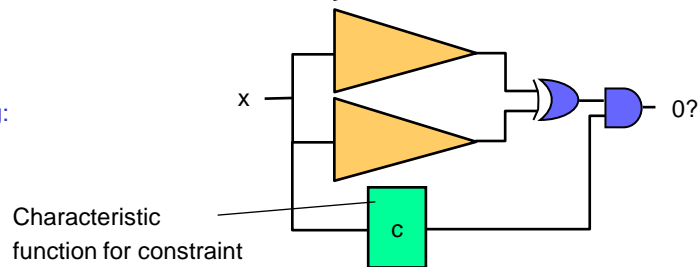
### Input constraints:

- non-occurring input values (don't cares)
- non-reachable states
- candidate for R

### 1. Input Mapping:



### 2. Output Masking:

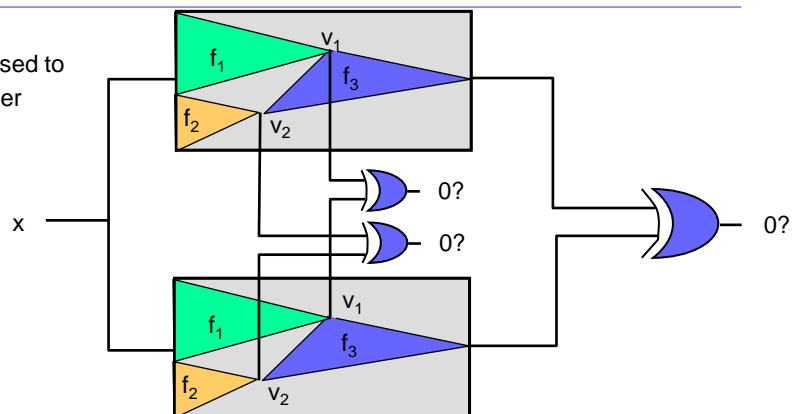


- 81 -



## Cutpoint-based EC

Cutpoints are used to partition the Miter



Cutpoint guessing:

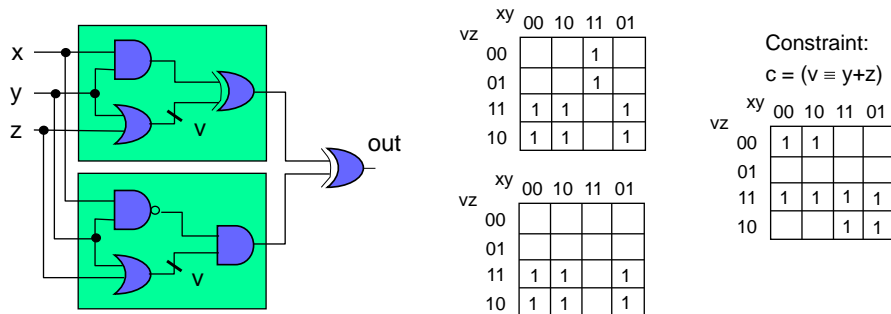
- Compute net signature with random simulator
- Sort signatures + select cutpoints
- Iteratively verify and refine cutpoints
- Verify outputs

- 82 -



## False Negatives

Outputs may miscompare for invalid cutpoint values:



What can we do about false negatives:

- constrain input space to  $c = (v \equiv y+z)$
- if  $(v \in \text{SUPPORT}(\text{out}))$  then  $\text{out} = \text{compose}(\text{out}, v, f_v)$

- 83 -

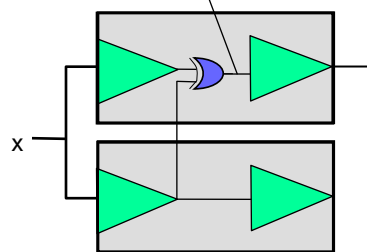


## Permissible Cutpoints

Testable for s-a-0 or s-a-1?

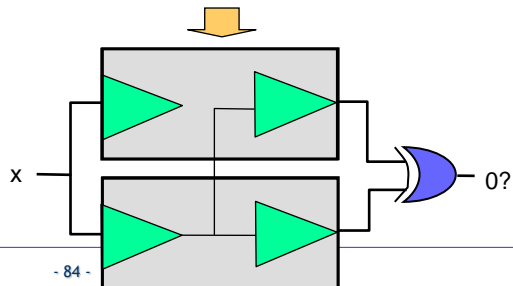
Based in ATPG:

- test for s-a-0 at output  
checks for permissible functions
- test for s-a-1 out output  
checks for inverse permissible functions



Permissible functions:

- successively merge circuits  
from input to outputs



- 84 -



## Sequential Equivalence Checking

---

- If combinational verification paradigm fails (e.g. we have no name matching)
  
- Two options:
  - run full sequential verification based on state traversal
    - very expensive but most general
  - try to match registers automatically
    - functional register correspondence
    - structural register correspondence

---

- 85 -



## Register Correspondence

---

- Find registers in product machine that implement identical or complemented function
  - these are matching registers in the two machines under comparison
  - BUT: might be more, we may have redundant registers

**Definition:** A register correspondence  $RC \subseteq \underline{s} \times \underline{s}$  is an equivalence relation in the set of registers  $\underline{s}$ .  
(This definition includes only identical functions, it can be extended to also include complemented functions)

A register correspondence can be used as a **candidate for R**:

$$r(s) = \prod_{\forall (s^i, s^j) \in RC} (s^i \equiv s^j)$$

---

- 86 -



## Functional Register Correspondence

```

Algorithm REGISTER_CORRESPONDENCE {
  RC' = {(si, sj) | si0 = sj0} // start with registers with
  do { // identical initial states
    RC = RC'
    r(s) = Pv (si, sj) ∈ RC (si ≡ sj)
    RC' = {(si, sj) | (si, sj) ∈ RC ∧ δi(x, s) = δj(x, s) ∧ r(s)}
  } while (RC' != RC)
  return RC
}

```

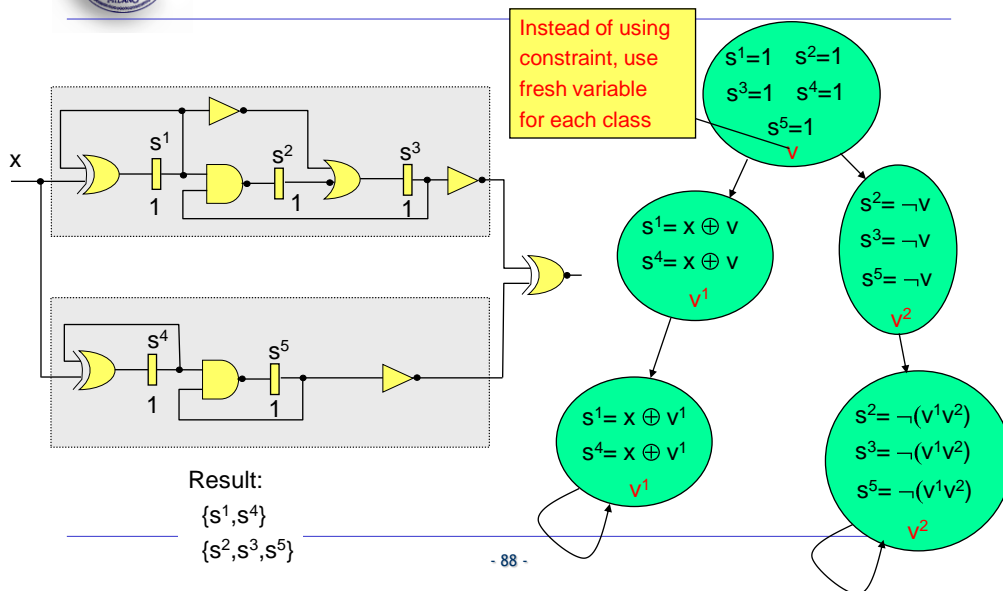
In essence:

- the algorithm starts with an initial partitioning with two equivalence classes, one for each initial value
- the algorithm computes iteratively the next state function, assuming that the RC is correct
  - if yes, fixed point is reached and RC returned
  - if no, split equivalence classes along the mismatches

- 87 -



## Example



- 88 -



## Problems with Functional Reg. Corr.

---

- In case of miscomparing designs
  - effect of miscomparing cone may ripple through entire algorithm and split all equivalence classes until they contain only single registers
  - difficult to debug since no hint of error location
  
- Solution:
  - relaxation of equivalence criteria
    - e.g. structural register correspondence algorithm based on support set of registers
    - combined techniques with name mapping, functional/structural criteria

---

- 89 -



## Sequential Equivalence Checking

---

- In case that combinational equivalence checking model fails:
  - use generalized register correspondence to also consider retiming
    - in essence, use all internal nets as candidates for possible matches
  
- Worst case: full sequential verification
  - Prove that the output of the product machine is not satisfiable (sequentially)
  - special case of general property checking


---

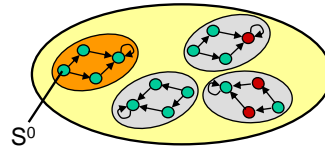
- 90 -




## State Traversal Techniques

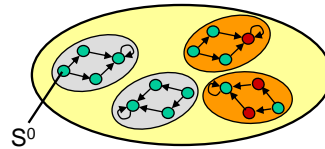
### Forward Traversal:

- start from initial state(s)
- traverse forward to check whether “bad” state(s) is reachable 




### Backward Traversal:

- start from bad state(s)
- traverse backward to check whether initial state(s) can reach them 



### Combines Forward/Backward traversal:

- compute over-approximation of reachable states by forward traversal 
- for all bad states in over-approximation, start backward traversal to see whether initial state can reach them 