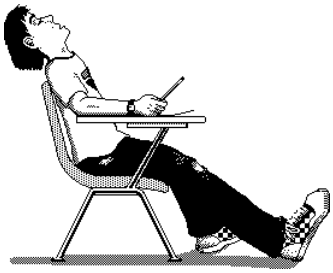
Basic Geometry Processing and LVS

Andreas Kuehlmann
Course 244
Fall 2005



Basics in Geometry Processing

Literature

- Ulrich Lauther: An O(n log n) algorithm for Boolean Mask Operations, 18th Design Automation Conference, 1981
- Thomas Szymanski, Christopher Van Wyk: Space Efficient Algorithms for VLSI Artwork Analysis, 20th Design Automation Conference, 1983
- Thomas Szymanski, Christopher Van Wyk: Goalie: A Space Efficient System for VLSI Artwork Analysis, IEEE Design and Test of Computers, June 1985
- Kuang-Wei Chiang, Surendra Naharm Chi-Yuan Lo: Time-Efficient VLSI Artwork Analysis Algorithms in GOALIE2, IEEE Trans. on CAD, June 1989

Introduction

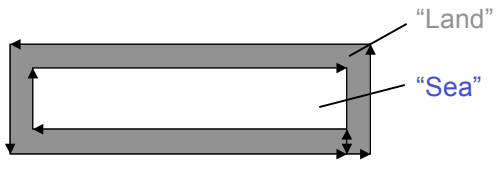
- Efficient geometry processing key for multiple layout tasks, including
  - Design rule checking (DRC)
  - Circuit extraction
    - Devices
    - Parasitics
  - Layout manipulation for yield and other issues
- Key operations:
  - Boolean mask operations (AND, OR, XOR) for
    - Auxiliary layers (e.g. gates of MOS transistors)
    - Secondary mask layers that can be derived (implantation)
  - Measuring areas and edge lengths of regions
  - Checking distance of point from regions

Historical Approaches

- Bitmaps
  - Represent each layer as matrix of individual bits
    - Bit = 1 ⇔ represents layer is present (land)
  - Highly efficient for Boolean operations
    - Implemented as Boolean operations on bits
  - Multiple problems:
    - Does not scale for large layouts
    - Difficult for non-Manhattan geometries
    - Difficult for measuring tasks
  - Use of special hardware proposed
- Representations of Boolean functions (e.g. BDDs, cube lists)
  - Similar to bitmap, mask is considered as Karnaugh map
  - Gives erratic behavior as size of Boolean function representation is unpredictable

Assumptions

- Geometries of individual layers are given as polygons
  - Set of directed edges
    - Not necessarily Manhattan geometry -> angled edges
  - Counter-clockwise orientation defines “inside” and “outside” of region
  - Overlaps are permitted and are interpreted as “OR”
  - “Holes” (also referred to as “sea”-areas versus “land”-areas) are described by overlaps/connecting regions
- Example:

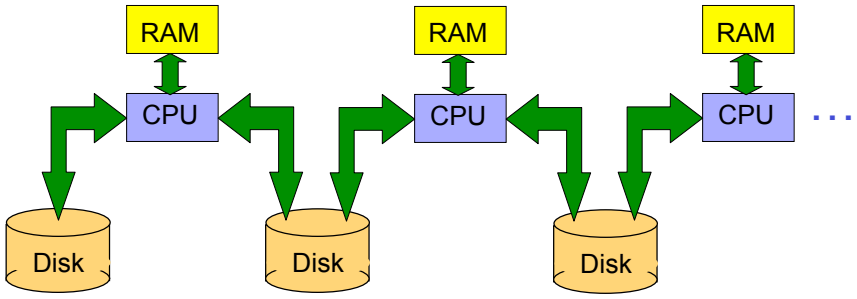


Assumptions

- Huge number of polygons to be processed
  - Billions per mask layer
  - Scales roughly with integration density
  - Thus “next generation” computers don’t help as they barely catch up with increasing integration density
- In-memory processing is hopeless with this data amount
- Solution:
  - Algorithms that exploits given computer architecture
    - Small amount of fast accessible random access memory (RAM)
    - Large amount of slower sequentially accessible memory (hard disk, tape)

Flow of Computation

- Sequential processing from disk to disk
  - Virtual memory management can mimic same behavior

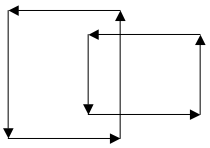


Basic Scanline Algorithm

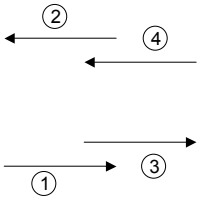
- Given:
  - Geometry information of set of layers in terms of edges
    - $((x_{left},y_{left}),(x_{right},y_{right}))$
    - $x_{left} < x_{right}$, omit vertical edges as they can be inferred during processing
    - Edge attribute indicated region (TOP, BOTTOM)
  - Stored in "edge file" in **canonical order**, i.e.,
    - First criteria: non-decreasing x_{left}
    - Second criteria: non-decreasing y_{left}
    - Third criteria: non-decreasing slope
- Processing
  - Sweep vertical "scanline" from left to right over layout stopping at positions x_{curr}
  - Read edges from input file and insert into scanline for processing
  - Drop edges from scanline as soon as $x_{right} < x_{curr}$

Simple Example for OR of Two Masks

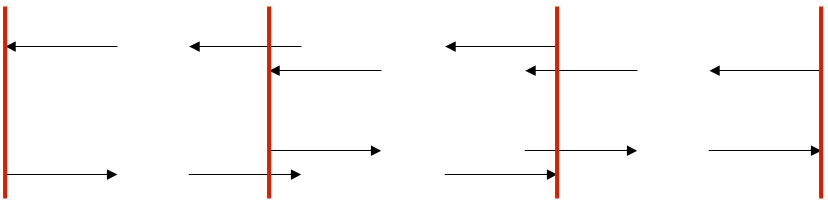
Polygons:



Edge file:

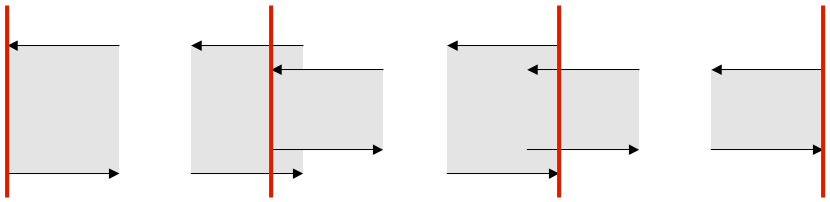


Scanline Stops:



Read 1&2 Read 3&4 Dump 1&2 Dump 3&4

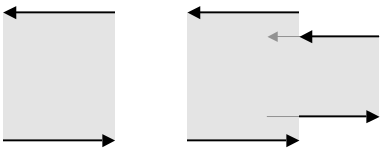
Simple Example for OR of Two Masks



Read 1&2 Read 3&4 Dump 1&2 Dump 3&4

Output Processing:

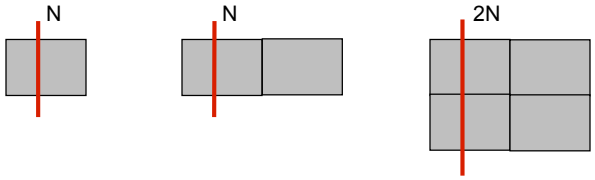
Chop edges 3 and 4
because part fully inside region



Write 1'&2' Write 3'&4'

Complexity of In-Core Memory

- Assumption that chip density and thus edge density is equally distributes
  - Assume processing area of size X x Y requires max N edges to be stored simultaneously on scanline
  - Doubling area to 2X x Y still required max N edges to be stored simultaneously on scanline
  - Doubling area again to 2X x 2Y requires max 2N edges to be stored simultaneously on scanline



- Hence: in-core memory complexity: $O\left(\sqrt{N}\right)$

Output Region Classification

- Overlapping areas need to be identified
- Simple counting algorithm at each scanline stop using one counter per layer
  - Set $count_{layer1} = count_{layer2} = 0$
  - Start from -∞ and enumerate edges of layers along scanline
    - Crossing bottom edge of layer1 : $count_{layer1}$++
    - Crossing top edge of layer1 : $count_{layer1}$--
    - Crossing bottom edge of layer2 : $count_{layer2}$++
    - Crossing top edge of layer2 : $count_{layer2}$--
  - For operation op area is part of $layer_1$ op $layer_2$ if BLACK = 1
    - BLACK = ($count_{layer1}$ > 0) op ($count_{layer2}$ > 0)
    - op ∈ {AND, OR, XOR, XNOR, NOT)
    - Whenever BLACK changes value, we cross true output edge

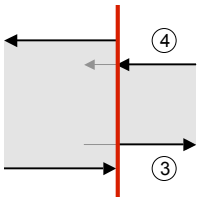
Lauther's Scanline Algorithm

```
Algorithm SCANLINE {  // QUEUE = sorted stream from edge files
  x_curr=x_left(MIN(QUEUE));  OLDSCANLINE = ∅
  do {
    NEWSCANLINE = ∅
    while (x_left (MIN(QUEUE)) == x_curr) // fill new scanline
      NEWSCANLINE = NEWSCANLINE ∪ MIN(QUEUE)
      REMOVE MIN(QUEUE)
    }
    OLDSCANLINE = MERGE(OLDSCANLINE, NEWSCANLINE) // see next
      OUTPUT all true edges with x_right=x_curr
      DELETE from OLDSCANLINE all edges ending at x_curr
    x_curr = ∞
    forall edge ∈ OLDSCANLINE {  // find next position
       x_curr=MIN(x_right(edge),x_left(MIN(QUEUE))
    }
  } until (x_curr = ∞)
}
```

Merge Operator

- Merges edges from `OLDSCANLINE` and `NEWSCANLINE` into `OLDSCANLINE`
  - For each adjacent pair of edges, if one of them is from `NEWSCANLINE` then check for intersection
    - If intersection occurs, edges are split
      - Left parts remains in `OLDSCANLINE`
      - Right parts are added back to `QUEUE`
  - Same check and split is performed for implicitly given vertical edges
- Back to example:

Edges 3 & 4 are split
at given scanline position

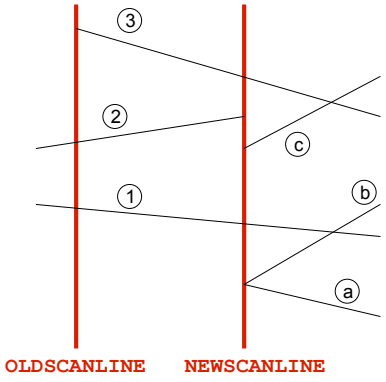


Example for Non-Manhatten Merging

```
OLDSCANLINE =
{1,2,3}
NEWSCANLINE =
{a,b,c}
```

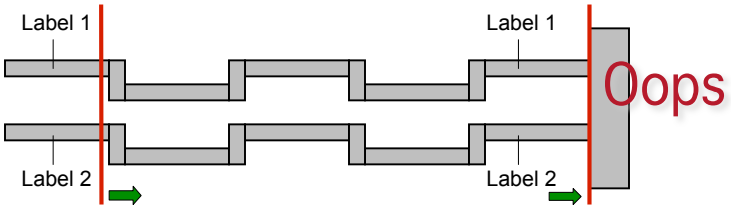
Merge steps:

```
 check (a,b)
 check (b,1)
   split at intersection
 check (1,c)
 delete 2
 check (c,3)
   split at intersection
```



Connectivity Analysis

- Connectivity analysis is critical for circuit extraction
  - Identifies connected components
    - Adjacent regions in same layer
    - Regions connected through via's
    - Terminal connections for MOS transistors
- Connections can be identified during scanline processing
  - However, connections on far right side cannot be predicted
    - Problem especially for power/ground connections



Three-Step Approach

- Forward "scanning phase" to assign temporary labels to regions
  - Result written in temporary file
  - Including information
    - New label is started
    - Two labels are merged
    - Label Ends
- Backward "renaming phase"
  - Processing temporary file in reverse order (from right to left)
  - Final assignments of labels to regions
- "Resorting phase" to make output file edges canonical in reverse order
  - Similar to scanline algorithm
    - Scanline "buffers" edges until they can be output

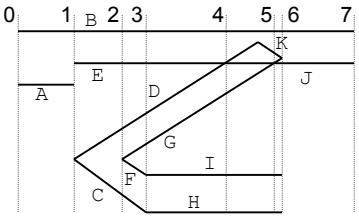
Forward Scanning Phase

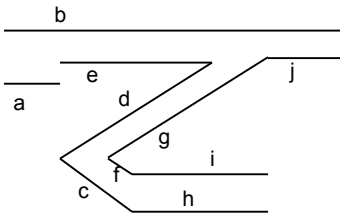
- Maintain
  - L: Set of edges at current scanline position x_{curr}
    - Count field for each edge records how many top and bottom edges on each side -> used for identifying regions
  - S: Set of sets of edges that form a partition of L
    - Given unique label (integer between 1…m)
    - Each set $s \in S$ contains edges that are the boundary of a already identified common output region
    - Sets $s_1 s_2 \in S$ may be merged if during scanline move
    - S is maintained using UNION-FIND structure
- During processing produce temporary output file containing
  - Edge with label
  - Merging of two labels (merged label can be reused)
  - End of a label when region is processed

Processing of S

- When edge e becomes boundary (potential output edge)
  - Add e to corresponding set in S
  - Create new $s \in S$ for e if new region with new label
- When two boundary edges e_1 and e_2 intersect on scanline or cross scanline at the endpoint of a vertical boundary edge,
  - Write record for Merge e_1 and e_2
- When edge ceases to be boundary (results in output edge e with $e\text{->}x_{right}=x_{curr}$
  - Write e(label) to temp file
- When set $s \in S$ becomes empty
  - Write special record end(s)

Example



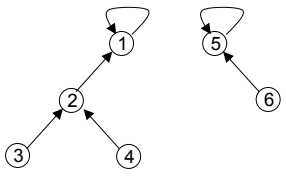


X_{curr}	S	Output written
0	{A,B}	
1	{B,E},{C,D}	a(1)
2	{B,E},{C,D,F,G}	
3	{B,E},{D,G,H,I}	c(2), f(2)
4	{B,G,H,I}	merge(2->1), e(1), d(1)
5	{B,H,I,J}	g(1)
6	{B,J}	i(1),h(1)
7	∅	b(1), j(1), end(1)

Union-Find Structure

- Three essential operations on elements x and y
  - **`Create-Set(x)`**: Create a set containing a single element x
  - **`Find-Set(x)`**: Find the set that contains x
  - **`Union(x,y)`**: Merges the set containing x and another set containing y to a single set.
    After that operation:
    - **`Find-Set(x) = Find-Set(y)`**

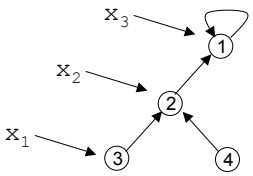
- Simple idea:
  - Put all elements in a tree data structure
  - Root of tree is representative of set
  - Root element points to itself



Union-Find Structure

```
Algorithm Create-Set(x) {
  create new vertex v
  vertex->parent = vertex
}
```

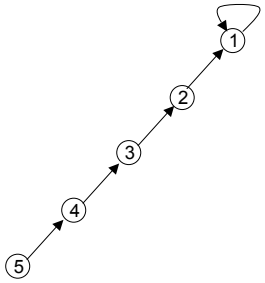

```
Algorithm Find-Set(x) {
  while (x->parent != x) {
    x = x->parent;
  }
  return x
}
```



Implementation of Merge

```
Algorithm Merge(x,y) {
  x->parent = y->parent
}
```

- Problem:
  - May result in linear linked list, e.g.
    - Merge(2,1)
    - Merge(3,2)
    - Merge(4,3)
    - Merge(5,4)
    - …



Better Implementation of Merge

- Make always the taller tree the parent of the shorter one

```
Algorithm Merge(x,y) {
  x=Find-Set(x)  // get representatives
  y=Find-Set(y)
  if(x->height <= y->height) {
    if(x->height == y->height) y->height++
    x->parent=y
  }
  else {
    y->parent=x
  }
}
```

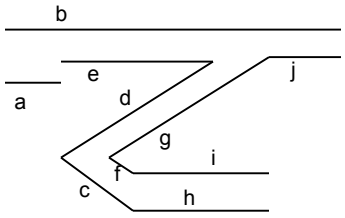
- Second trick
  - Relink tree structure during `Find-Set`

Backward Renaming Phase

```
// Temp file contains reverse tree structure of merged
// region labels
Algorithm RENAME {
  region_count = 0
  while (records left in temp file) {
    record = READ_NEXT   // READ temp file in reverse order
    if(record is of form "end(i)") {  // start new label
      region_count++
      final_label[i] = region_count  // static array
    elseif (record is of form "merge(i->j)") {
      final_label[i] = final_label[j]
    else {   // record is of form "e(i)"
      OUTPUT e tagged with final_label[i]
    }
  }
}
```

Resorting Phase

- Note that edges in file written by `RENAME` are sorted in non-increasing x_{right} order



- Written in following order
  - `j,b,h,i,g,d,e,f,c,a`
- Idea of resorting is to use another scanline to sort edges in reverse canonical order (next process needs to read file backwards)

Resorting Phase

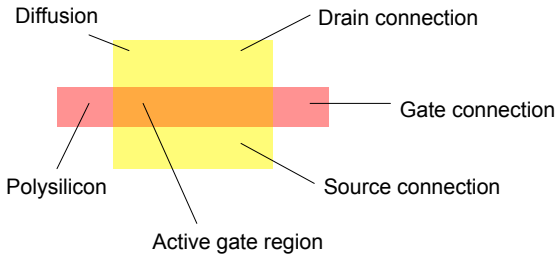
```
Algorithm RESORT {
  QUEUE = ∅ // queue sorted in reverse canonical order
  while (records left in temp file) {
    next_edge = READ_NEXT  // READ temp file
    while (x_right(next_edge) <= x_left(MAX(QUEUE)) {
      OUTPUT MAX(QUEUE)
      REMOVE MAX(QUEUE)
    }
    INSERT next_edge into QUEUE
  }
  while (QUEUE != empty) {
    OUTPUT MAX(QUEUE)
    REMOVE MAX(QUEUE)
  }
}
```

Notes on Connectivity Analysis

- Algorithm is very general
  - Can handle multi-layer connectivities
    - Process one scanline per layer in synchronous manner
    - Maintain one set S for all layers
    - Whenever via occurs between two layers
      - Merge sets in S
    - Use one temporary file and tag edge with layer number to separate them during resorting phase
- Renaming and resorting phase can be pipelined

Transistor Identification

- MOS transistors are formed by intersection of diffusion and polysilicon layer



- Generate three edge files for transistor identification:
  - Active gate area (Poly & Diff) (trans_file)
  - Polysilicon outside of active gate area (Poly & ^ Diff) (poly_file)
  - Diffusion outside of active gate area (^Poly & Diff) (diff_file)

Transistor Identification

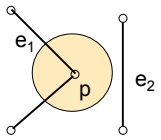
- Sweep scanline from left to right and process all transistor edges from trans file
  - For each edge processing:
    - Maintain transistor record by transistor label
      - Handles snaked and other weird transistors
    - Read all diff edges and poly edge touching edge
    - Build transistor record with using poly label, and diff label as connection identifier for Gate, Drain and Source
  - Check for illegal situations
    - Transistor edge not adjacent to poly or diff edge

Improvements

- For large chips updating region counters become bottleneck
  - Requires a complete scanline sweep from bottom to top for each x_{curr}
    - Done after all edges starting at x_{curr} read in
- However regions change only locally
  - Processing of entire scanline is overkill
  - Only required because of sea areas
- Solution: Convert layout into trapezoids
  - Reading of edges can be done in pairs
  - Change area is now predictable
    - Changes start as bottom of trapezoid
    - Ends at top of trapezoid

Design Rule Checking

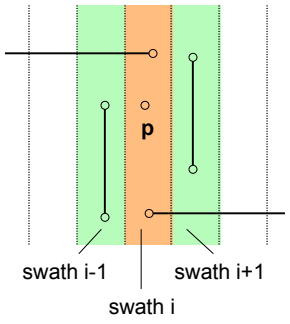
- Most rules can be reduced to checking the maximum tolerance of an edge e_1 from all other edges e_2 with difference label
  - Checking can be reduced to checking both endpoints p or e_1



- Search space can be reduced by excluding all edges e_2 which are father away than maximum design rule
  - W.l.o.g., assume we check for tolerance 1
  - Split the horizontal chip into “swaths” of width 1
  - Check for each endpoint p of edge e_1 whether edge of some different net lies within tolerance of 1

Swath Structure

- Key of checking algorithm is that we need to check p only against edges with endpoints in swath i-1, i, or i+1



- Extend “lifetime” of edges during scanline algorithm to stay during swath i+1, i, and i-1
  - Check endpoints in swath i

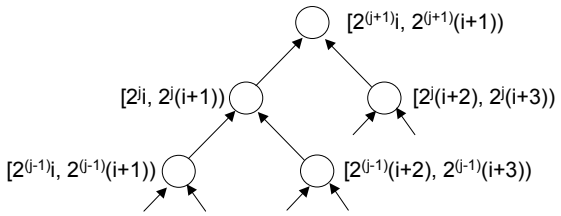
Tolerance Checking Algorithm

- Effectively "scanbar" algorithm

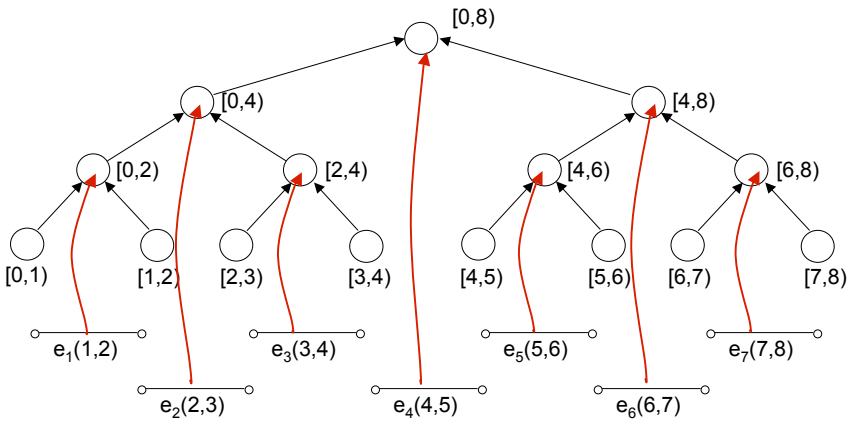
```
// W width of chip
// H height of chip
Algorithm CHECK {
  for (i = 0; i <= W; i++) {
    INSERT into S all edges with x_left in swath i+1
    foreach edge e with an endpoint in swath i do {
      CHECK_SEGMENT endpoint p of e against edges in S
    }
    REMOVE from S all edges with x_right is in swath i-1
  }
}
```


Vertical Segment Tree Structure

- To represent S, split vertical range 1..H into subintervals of length 1
  - Binary tree structure for effective searching
    - Nodes on level j cover range of 2^j units, intervals closed on left, open on right
      - Leaf is level 0, root level $\log_2 H-1$
  - Edges are inserted such that their y-range is completely covered by node range



Vertical Segment Tree Structure



Checking of Segment Structure

- Check interval that contains y(p) and its left and right neighbor

```
Algorithm CHECK_SEGMENT (p) {
  l = leaf in S that contains y(p)
  while (l is not root(S)) {
    TEST p against all edges in l
    TEST p against all edges in l's left neighbor
    TEST p against all edges in l's right neighbor
    l = parent(l)
  }
}
```

Improvements

- Each endpoint is contained in two edges of polygons
  - If test is independent of slope, one can eliminate half of the tests, e.d. test only
    - bottom endpoint of left edge
    - left endpoint of top edge
    - top endpoint of right edge
    - right endpoint of bottom edge
- On non-leaf segments, skip test of neighbors if y(p) is father than interval border
- Edges in higher parts of S are checked more often
  - Use asymmetric segment tree with redundant entries for edges

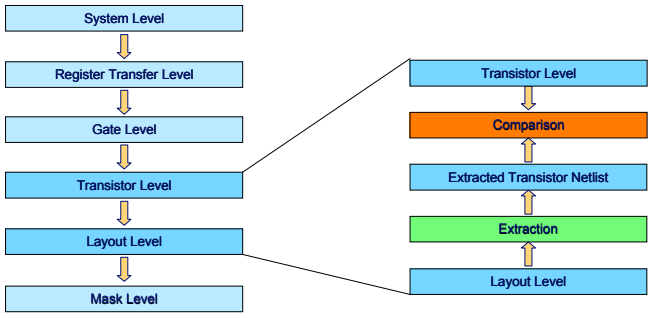
Basic LVS

Literature

- C. Ebeling and O. Zajicek: Validating VLSI Circuit Layout by Wirelist Comparison, International Conference on Computer Aided Design (ICCAD-83) pp. 172-173, September 1983.
- C. Ebeling: Gemini II: A Second Generation Layout Validation Tool, IEEE International Conference on Computer Aided Design (ICCAD-88), pp. 322-325, November 1988
- M. Ohlrich, C. Ebeling, Eka Ginting and Lisa Sather: SubGemini: Identifying Subcircuits Using a Fast Subgraph Isomorphism Algorithm, Design Automation Conference, June 1993. pp. 31-37

Problem

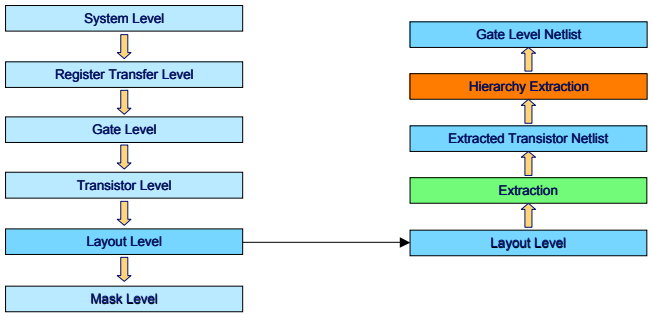
1. Compare two netlists on CMOS transistor level



- Applications:
  - Layout verification
  - Switch-level circuit simulation
  - Functional equivalence checking

Problem

2. Extracting a hierarchical netlist



- Applications:
  - Gate-level comparison
  - Gate-level simulation (faster than switch-level)
  - Technology migration
  - Technology mapping

Introduction

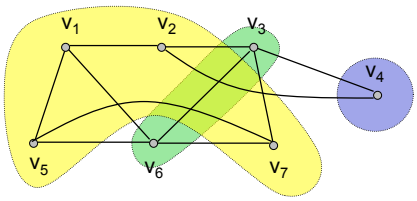
- Practical problems involve comparison of netlists with millions of transistors
  - Hierarchically or flat
- Circuit comparison is equivalent to testing of graph-isomorphism
  - No efficient deterministic algorithm
  - Naïve sol
- Fortunately many properties of practical netlists can be exploited to make solution feasible
  - Limited degree of graph nodes except few, e.g. VDD, GND, Clock
  - Limited stack height of transistors in practical circuits
    - < 5 for static CMOS

Graph Partitioning

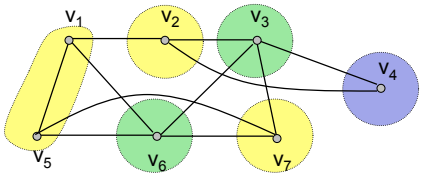
- Use vertex invariants to classify vertices into equivalence classes
  - Based on attributes and degree of vertex
  - Apply Hopcroft's equivalence refinement until fixpoint reached
  - Corresponding vertices in both graphs end up in one EC

```
Algorithm Partition {
  l(v) = initial label for vertex v
  E' = {(v_i,v_j) | l(v_i)=l(v_j)}
  do {
    E = E'
    E'= {(v_i,v_j) | l(v_i)=l(v_j) ∧
                     there is a pairwise mapping of all
                            adjacent vertices u_i, u_j such
   that
                     (u_i,u_j) ∈ E} }
   } while (E != E')
}
```

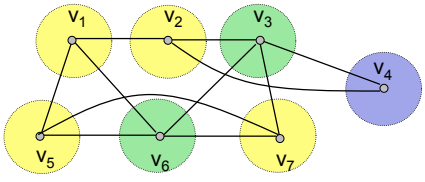

Example



E_1= { {v_1,v_2,v_5,v_7}, {v_3,v_6} ,{v_4} }



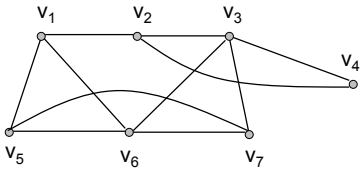
E_2= {{v_1,v_5},{v_2},{v_7},{v_3},
{v_6},{v_4}}



E_3= {{v_1},{v_5},{v_2},{v_7},{v_3},
{v_6},{v_4}}

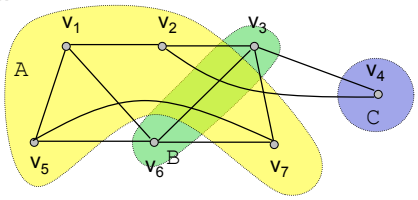
Using Neighbor Signatures

Initialization:



Sig(v_1) = 3
Sig(v_2) = 3
Sig(v_3) = 4
Sig(v_4) = 2
Sig(v_5) = 3
Sig(v_6) = 4
Sig(v_7) = 3

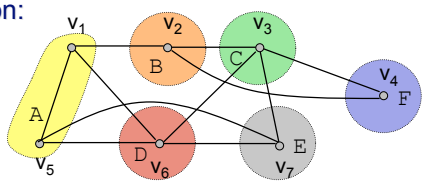
1. iteration:



Sig(v_1) = AAB
Sig(v_2) = ABC
Sig(v_3) = AABC
Sig(v_4) = AB
Sig(v_5) = AAB
Sig(v_6) = AAAB
Sig(v_7) = ABB

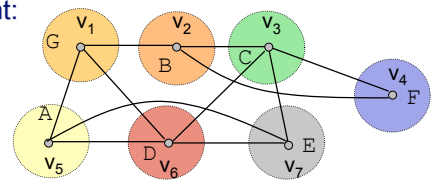
Using Neighborhood Signatures

2. iteration:



Sig(v_1) = ABD
Sig(v_2) = ACF
Sig(v_3) = BDEF
Sig(v_4) = BC
Sig(v_5) = ADE
Sig(v_6) = AACE
Sig(v_7) = ADC

Fixpoint:



Sig(v_1) = G
Sig(v_2) = B
Sig(v_3) = C
Sig(v_4) = F
Sig(v_5) = A
Sig(v_6) = D
Sig(v_7) = E

Implementation

- Hash signatures into computer word and store in linear array
  - May result in hash collisions, however, can be handled in post-processing
- Sort array according to signature
- Single sweep over array to identify new boundaries of equivalence classes
  - Refines old boundaries
- Refinements:
  - Second sorting criteria: size of the adjacent equivalence classes
  - Refinement only on first top classes
    - Idea: small equivalence classes lead to quick refinement of their neighbors

Implementation (2)

1. Sort array by signatures
   - neighbor class size as second criteria
2. Linear sweep to mark class boundaries

Hash value for neighbors	Size of largest neighbor class
0x575467	4
0x575467	19
0x605566	2
0x605566	4
0x605566	34
0x946284	567
0x787734	7
0x787734	67
....	
0x794383	424
0x805566	7

Implementation (3)

3. Sort classes by smallest neighbor class

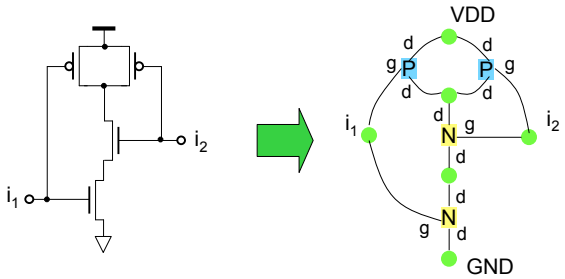
4. Update signatures for first n classes
   - e.g. classes of neighbor size 1

5. Goto 1

Hash value for neighbors	Size of largest neighbor class
0x605566	2
0x605566	4
0x605566	34
0x575476	4
0x575476	19
0x787734	7
0x787734	67
0x805566	7
....	
0x794383	424
0x946284	567

LVS Algorithm

- Both circuits to be compared as bipartite graph
  - First set of vertices models transistors/gates
  - Second set of vertices models nets
  - Initial vertex invariants used for initial labeling
    - Type of device for transistor/gate, number of terminals for net
    - Terminal types (gate or drain/source)
    - Length of smallest cycle containing vertex



LVS Algorithm

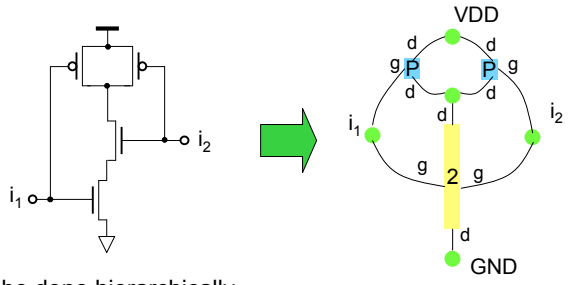
- First phases based on graph partitioning
  - Partitions both graphs simultaneously
  - Use terminal type during relabeling, e.g.
    - `new_device_sig = Σ(terminal_type · net_sig)`
    - `new_net_sig = Σ(terminal_type · device_sig)`
  - If final partitions contains only one element done
- Second phase
  - Exhaustive search through all combinations
  - Try match of pair vertices of both graphs
    - Recur to first phase
  - Possible use intervention of ambiguity cannot be resolved

End Result

- If Partitioning results in only singletons, graphs are matched
  - Sort labels in both graphs and match them up in linear order
- In case of miscomparing graphs
  - Even for simple, local miscompares partitioning process can ripple through the entire graph and make both parts incomparable
- Solution in Gemini:
  - At each level during partitioning
    - Match classes between both graphs
    - If mismatch, stop refining mismatches class and report error
  - Continue refining all other classes

Improvements

- Dealing with series transistors
  - Serial transistors are exchangeable since no change of function
  - Often done during late timing correction to accommodate for late arriving signals
  - Solution: Keep them as one vertex and resolve later



- Can be done hierarchically

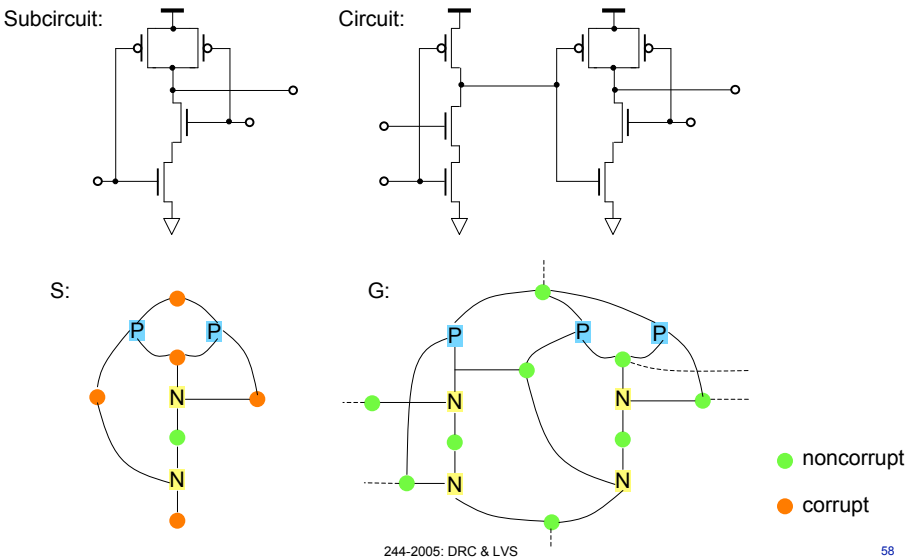
Improvements (2)

- Symmetric circuit structures cause many iterations during refinements
  - E.g. ring oscillators, rotator circuits, highly regular data-paths
- Local matching
  - Checks pairs of already matched vertices and matches them based on local information
    - Partitioning done only on neighbor vertices
    - Continues until no new matches found
    - Interleaved with global repartitioning
- Miscompares can be refined by local matches
  - One circuit declared as golden
  - Local matches applied on mismatched equivalence class

Subcircuit Identification

- Used in building hierarchy
  - Problem is now subgraph isomorphism
- Basic algorithm
  - Phase 1:
    - Use graph partitioning to find candidate vector for each subcircuit S in circuit G
      - Ensure that terminal net invariants are not used as connectivity not known yet (corrupt vertices)
      - Candidate vector stored as set of key vertices which are potential matches
  - Phase 2:
    - Continue relabeling phase of partitioning algorithm to confirm matches in circuit
  - Done in parallel for all subcircuits (e.g. library)

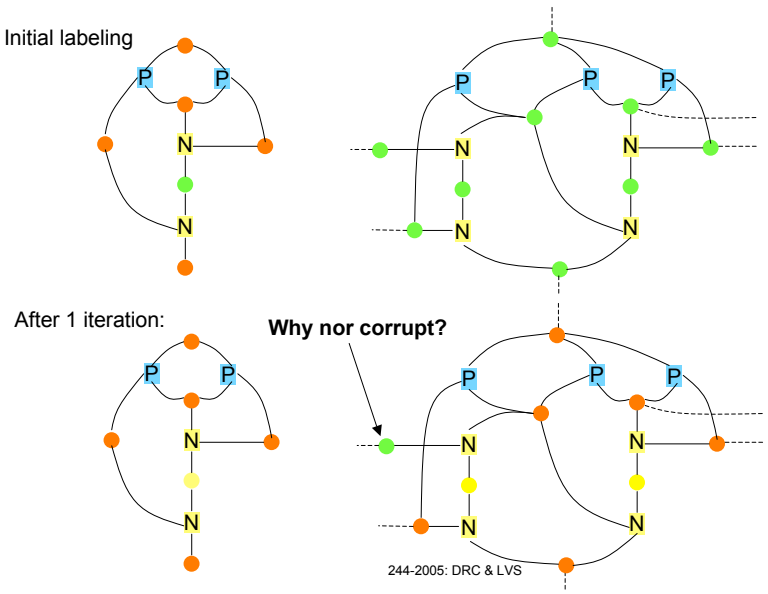
Example



Phase 1

- Iterative refinement process for graph partitioning
- All vertices with corrupt neighbors are also marked corrupt
  - Corrupt property used to indicate if vertex might be connected differently in S and G
  - Corruption propagates from external terminals into the circuit until all vertices are corrupt
- Stop propagation one level before circuit fully corrupt
  - Select key vertex from matching that is used for candidate vector
  - If multiple choices – pick arbitrary candidate

Example



Phase 2

- Process candidate vector in G to confirm matches
- Modified partitioning algorithm
  - Starts from key vertices and matches them
  - Propagate through both graphs S and G and update labels according to modified relabeling function
  - Ensure that other partitions do not spoil label function