ECE 260C, Spring 2025

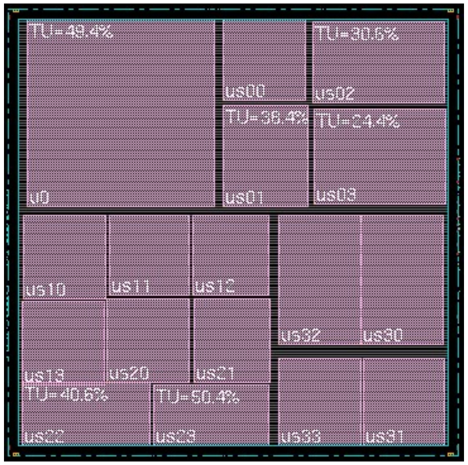
Routing

Andrew B. Kahng

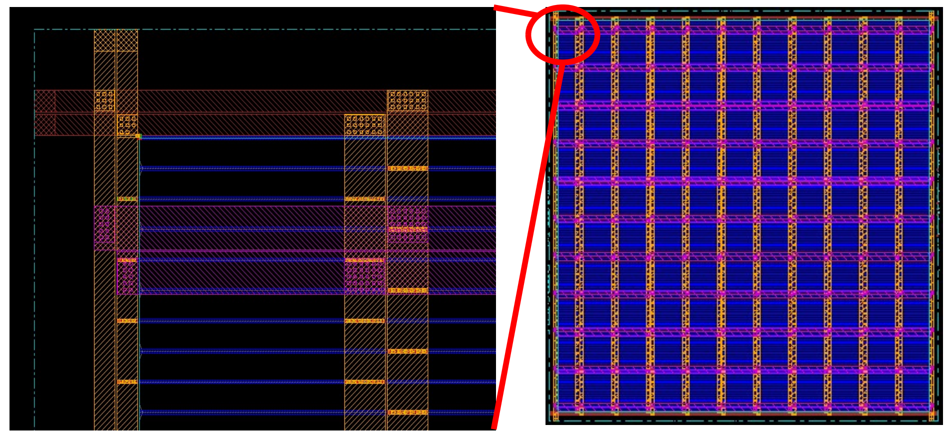
Thanks to: Bangqi Xu, Matt Liberty, Cho Moon, Eder Monteiro, Zhiang Wang, …

Physical Design Flow Pictures (old ECE 260B slide)

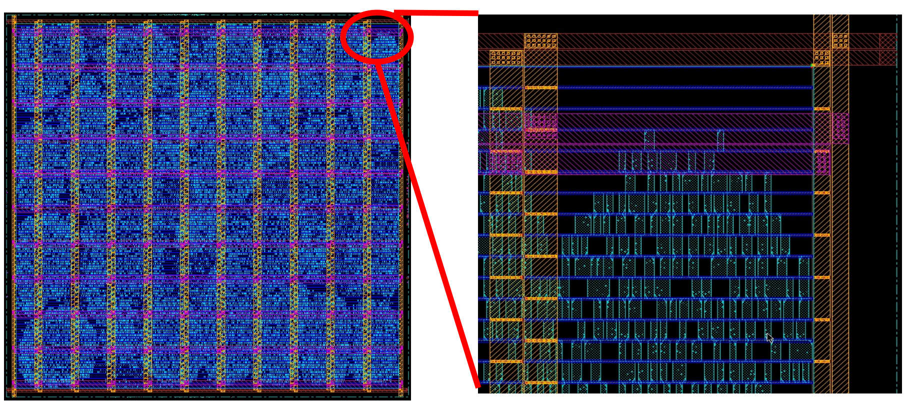
- Floorplanning



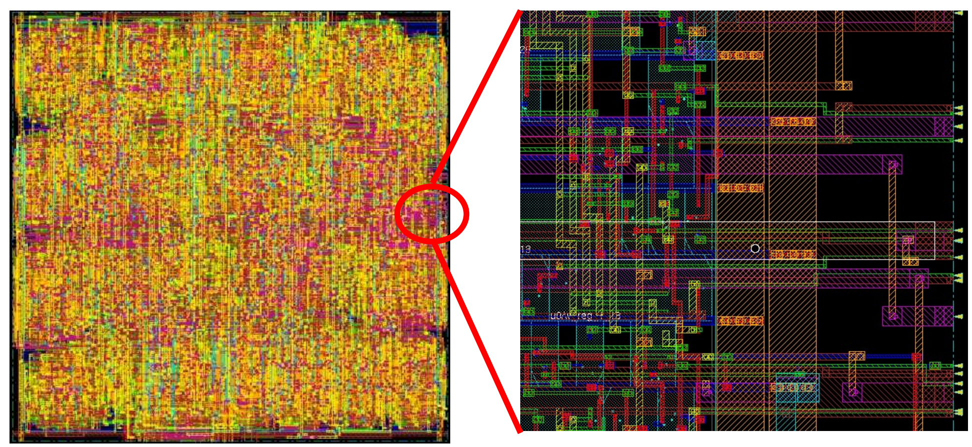
- Powerplanning



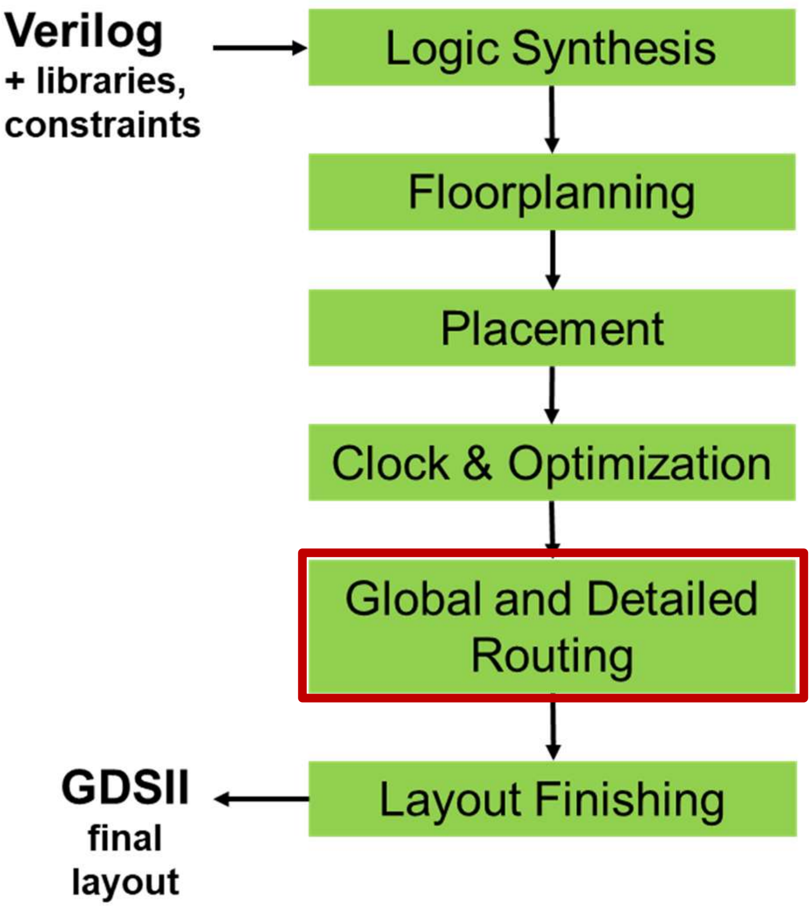
- Placement

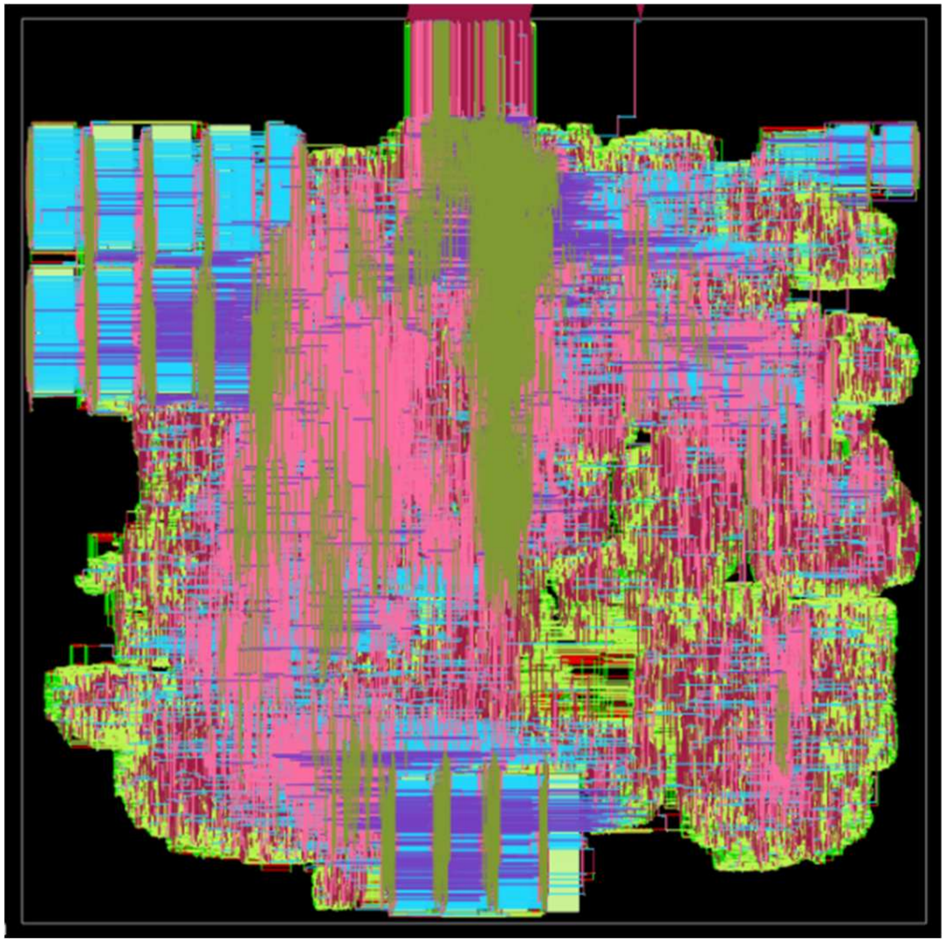


- Routing



Final (Detailed) Routing

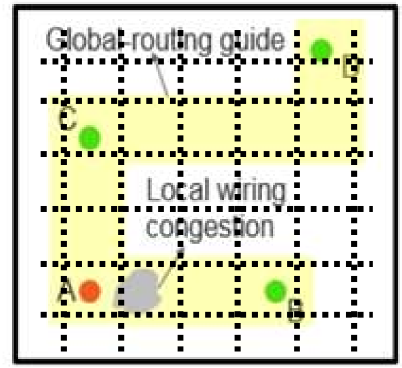




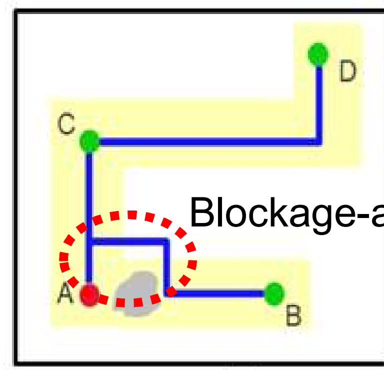
Background

[ICCAD18]

- Routing challenges
  - Complex design rules
  - Enormous solution space
  - Physical and circuit considerations
- Generic "area routing" flow
  - Global routing
    - Produces 3D "route guides"
  - Detailed routing
    - Input: route guides = **union of gcells**
    - Output: physical nets
    - Subject to: honoring route guides, honoring design rules



gcell



http://www.ispd.cc/contests/18/index.htm

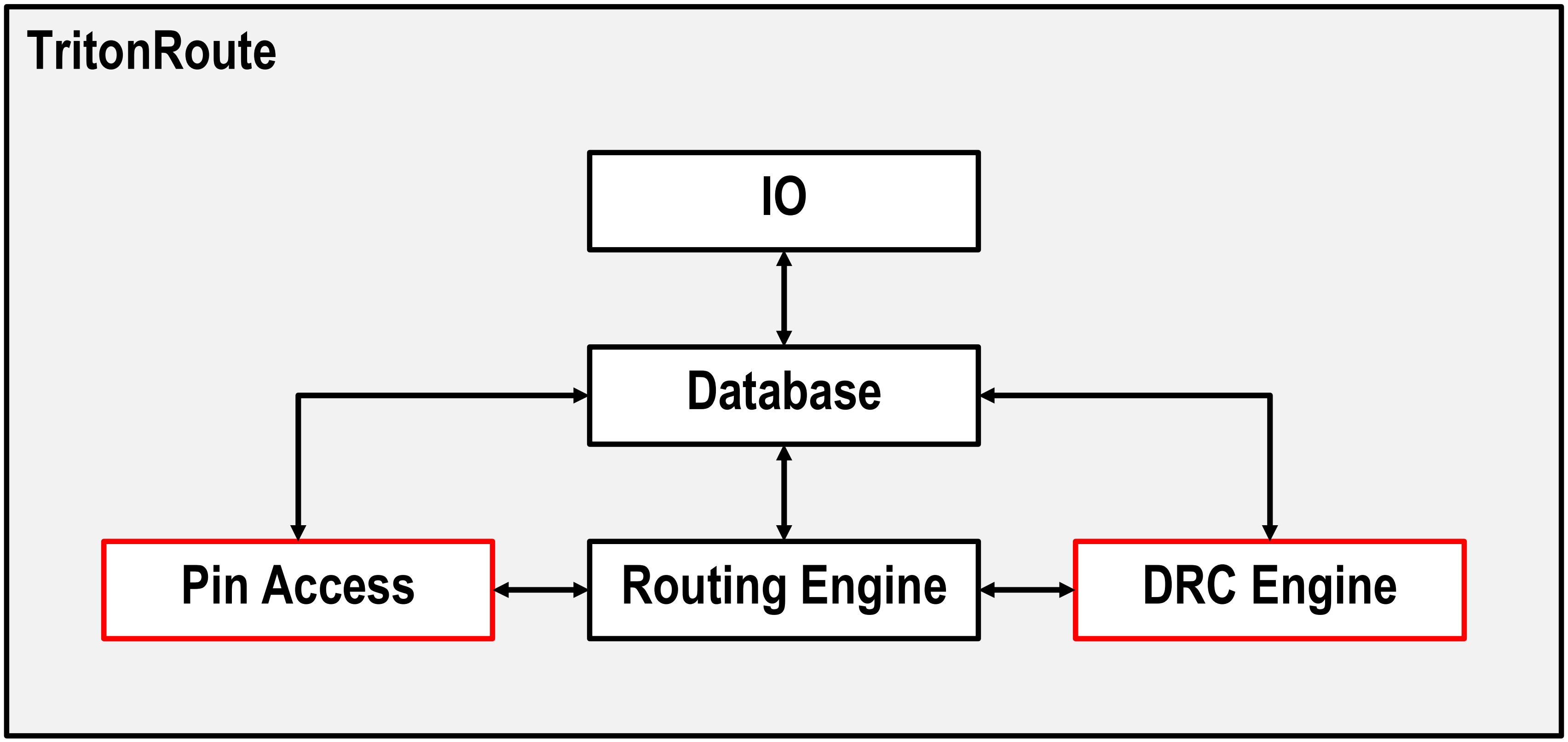
GCell = global routing grid;
Global router will only generate gcell-to-gcell connections

Critical Elements

- Must be able to clean up DRC (design rule check) violations !
  - Without a **DRC engine** → can't tell that violations exist !
  - Without violation filtering → have no clue what to ripup
- Major source of violations: naive **pin access**
  - On-track access assumption
  - No inter-cell pin access compatibility check
  - No accurate modeling of design rules
- **DRC engine** and **robust pin access** are "scuba tanks"
  - SCUBA: Self-Contained, Underwater, Breathing Apparatus

TritonRoute (2018-2022) – Overall Structure

- Best academic detailed router for contest benchmarks
- Only academic detailed router capable of delivering DRC-clean solution for commercial foundry nodes



Geometry-Based Design Rule Checking for Detailed Routing

Motivation

- Design rule checking is critical for EDA enablement
  - New technology has increasingly complex design rules
  - Mandatory physical verification for signoff
- **No** end-to-end framework for design rule checking in the open literature

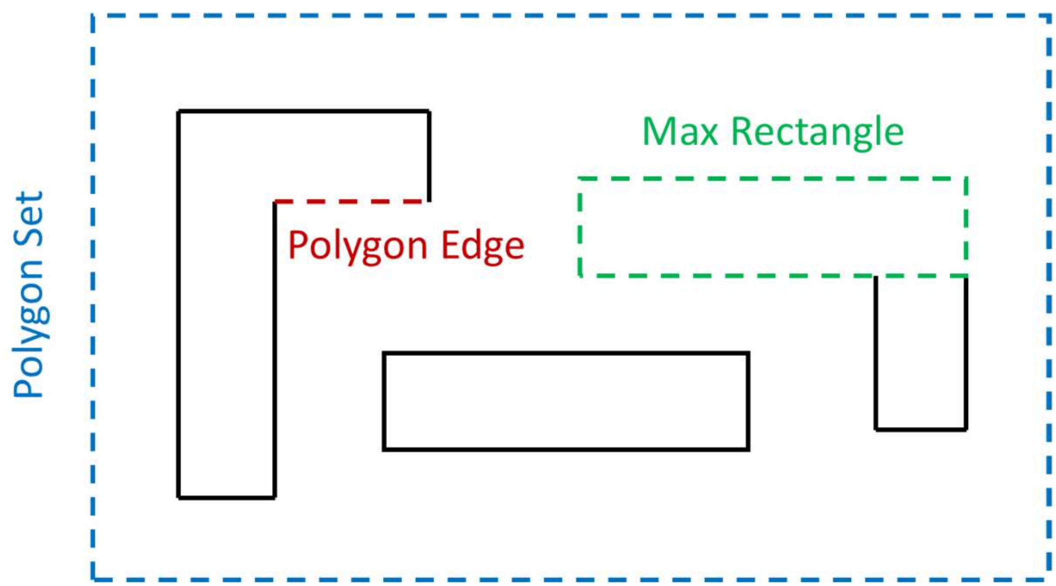
  → Missing key enablement for DRC convergence

Work at UCSD:

(i) Optimized data structures for design rule check **for detailed routing**
(ii) Industry-format (LEF) based design rule check methodology
(iii) Differentiation between fixable and non-fixable design rule violations **for detailed routing**
(iv) Foundry nodes: confirmed "clean" by commercial DRC tools

Preliminaries: Basic Geometry Objects

- Basic geometry objects in a DRC checking database
  - **Polygon edge**
    = Edge of a polygon
  - **Max rectangle**
    = Maximum rectangle inside a polygon
  - **Polygon set**
    = Union of disjoint polygons

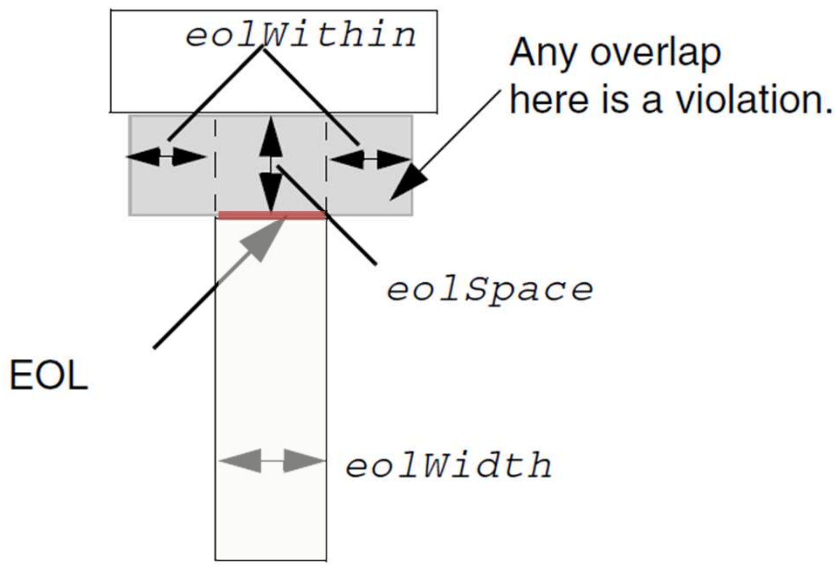


Preliminaries: Design Rule Syntax

- Typical design rule can have three components
  - Spacing value
  - Intrinsic property condition (optional trigger)
  - Extrinsic property condition (optional trigger)
- Example

```
SPACING eolSpacing ENDOFLINE eolWidth WITHIN eolWithin
```

Spacing value — Intrinsic property — Extrinsic property



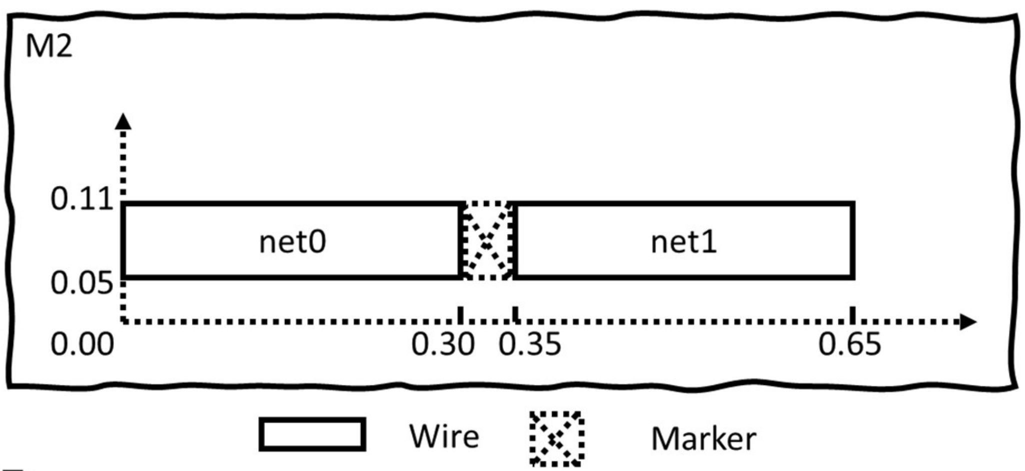
a) EOL width < *eolWidth* requires *eolSpace* beyond EOL to either side by < *eolWithin* distance.

EOL = END-OF-LINE

Preliminaries: Design Rule Violation Marker

- A design rule violation **marker** consists of
  - Bounding box
  - Layer
    } where
  - Violation net(s) } what
  - Design rule } why
- **Usage**: Give hints to DR where / what to ripup
- Example
  - Rule: SPACING 0.06

    → Object needs to be 0.06 unit away from each other

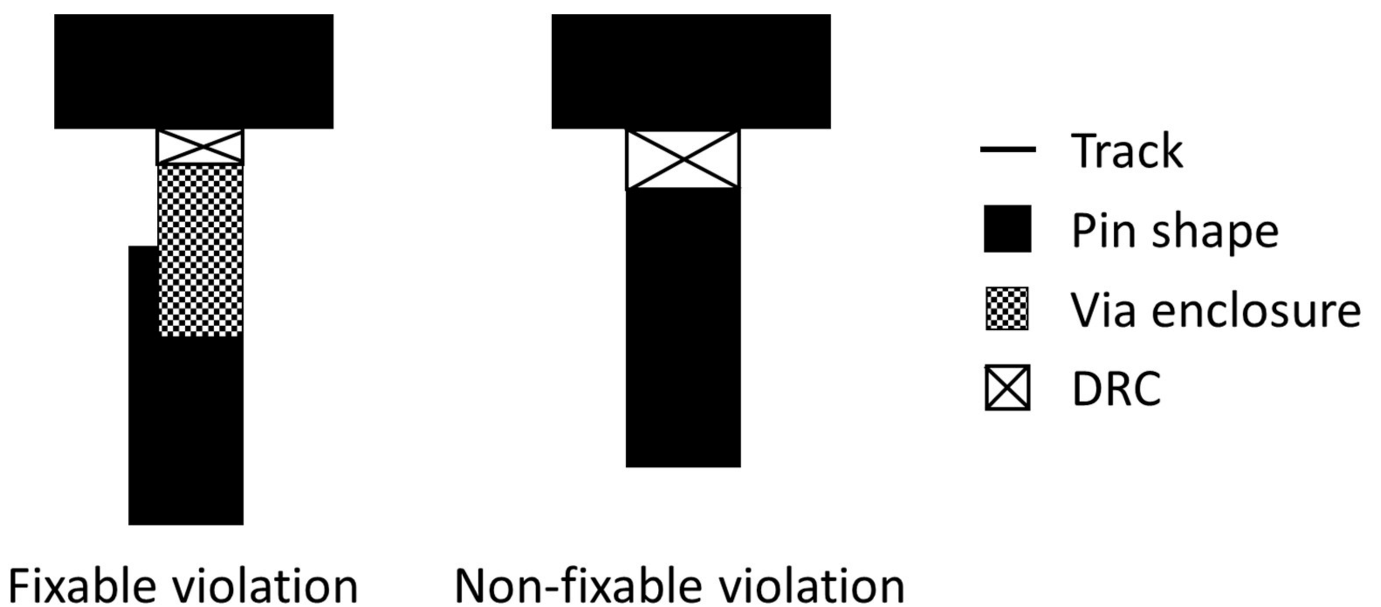


Marker
- **Bbox: (0.30, 0.05) – (0.35, 0.11)**
- **Layer: M2**
- **Violation nets: net0 & net1**
- **Design rule: SPACING**

Preliminaries: Fixable and Non-Fixable DRC

- Knowing locations of DRCs is not enough
  - DRC could happen inside standard cell itself
  - DRC could happen between PG stripes

    → DR cannot help resolve such **non-fixable** DRCs
- Need to filter DRCs before give them to DR
  - I.e., only provide DR with **fixable** DRCs
- Example



Problem Statement

- **Goal:** Given layout objects, find fixable design rule violations (if any)

- **Inputs**
  - Design layout database
  - Design rules
- **Constraints**
  - Design rule checking bounding box
  - Layer range (e.g., M2-M5)
- **Output**
  - Fixable design rule violation markers

Database Objects

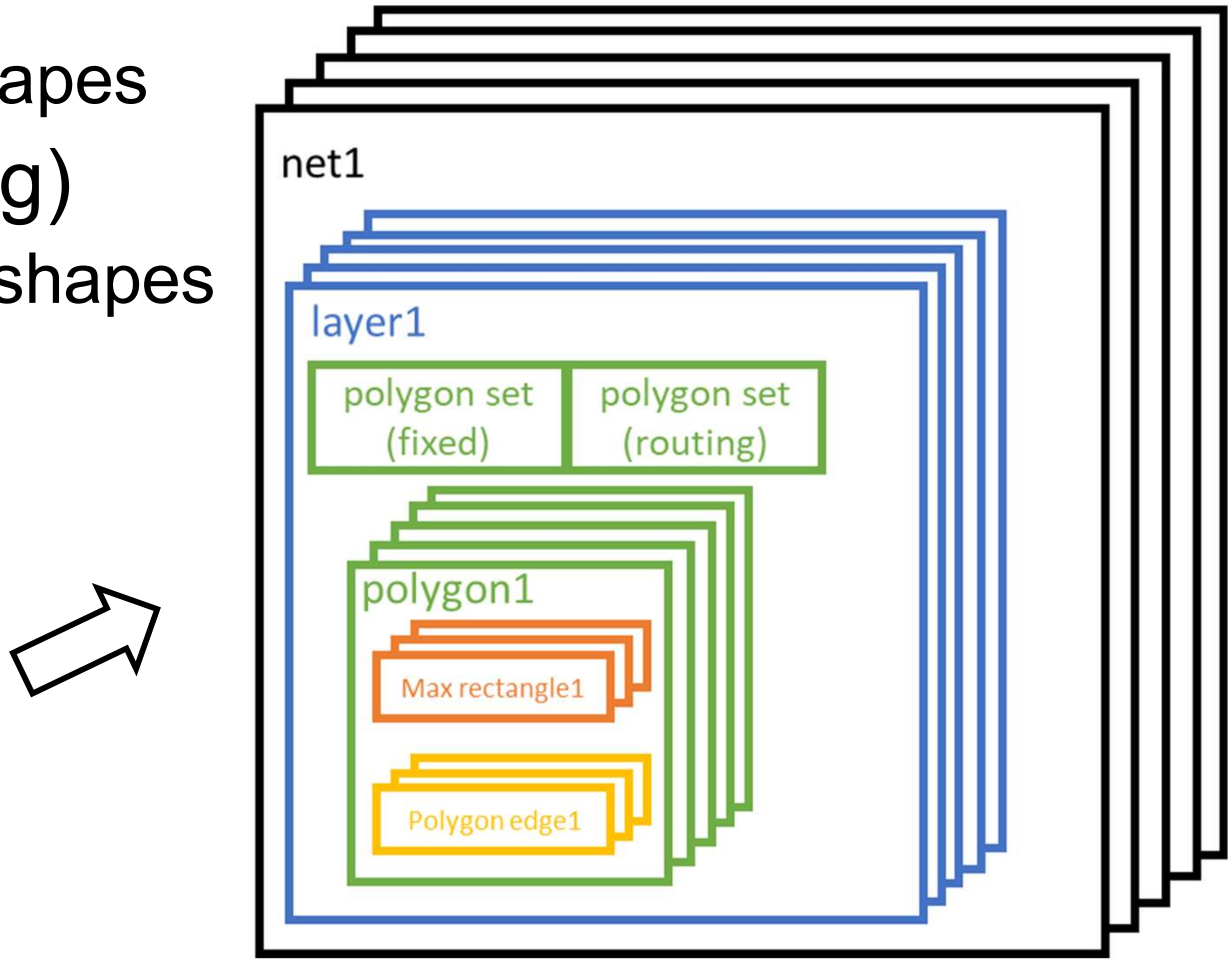
Object	Meaning	Status	Geometries
instTerm	cell pin	fixed	polygon(s)
term	block IO pin	fixed	polygon(s)
instBlockage	cell blockage	fixed	polygon(s)
blockage	block blockage	fixed	polygon(s)
pathSeg	regular net wire	routing	rectangle
pathSeg	special net wire	fixed	rectangle
via	regular net via	routing	rectangle(s)
via	special net via	fixed	rectangle(s)
patchMetal	regular net patch metal	routing	rectangle
patchMetal	special net patch metal	fixed	rectangle

Data Structure

- Object (fixed) & object (routing)

  → Key for fixable / non-fixable differentiation
- Polygon set (fixed)
  - Union of fixed obj. shapes
- Polygon set (routing)
  - Union of routing obj. shapes

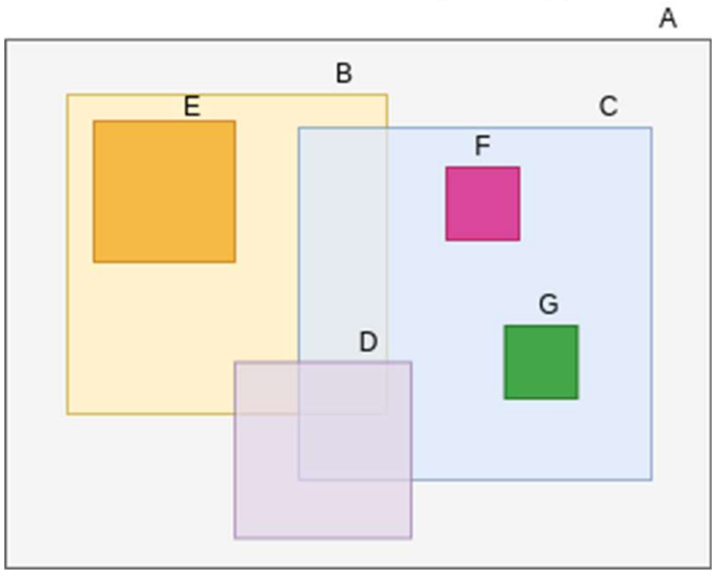
Object	Status
instTerm	fixed
term	fixed
instBlockage	fixed
blockage	fixed
pathSeg	routing
pathSeg	fixed
via	routing
via	fixed
patchMetal	routing
patchMetal	fixed



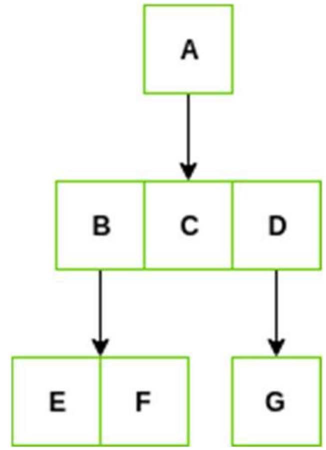
DRC checking database

Region Query

- RTree as underlying container



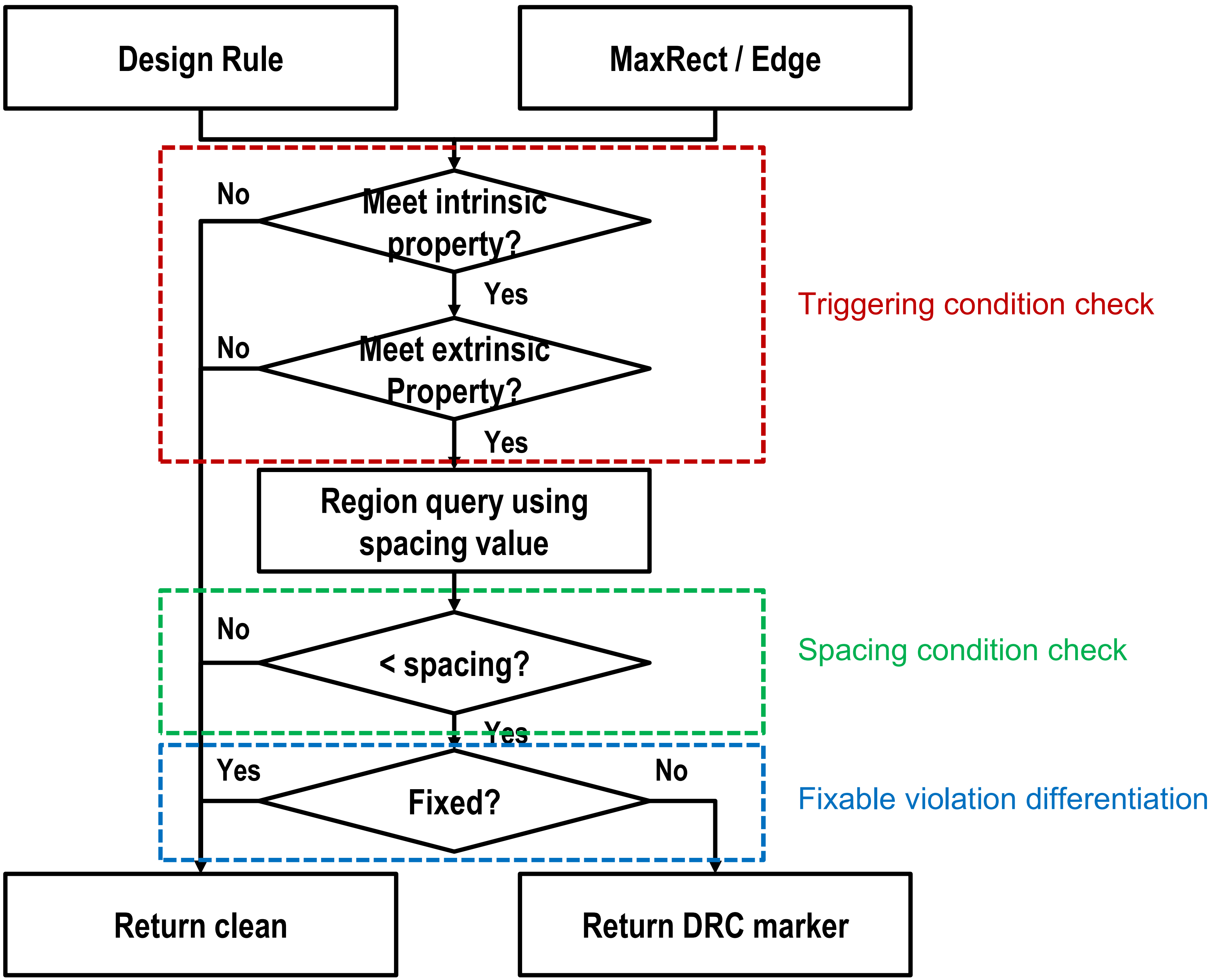
Layout



RTree Representation

- Two RTrees for each layer
  - MaxRect RTree
  - Edge Rtree
- Each maxRect / Edge has property indicating whether it is **fixed** or **routing**

High-Level Flow for DRC Check (Single Obj.)



Design Rules Types

- Design rules are divided into two categories
  → **Metal** layer and **Cut** layer
- **Metal layer rules**
  - Metal short
  - Non-sufficient-metal-overlap
  - Parallel run length (PRL) spacing
  - Minimum width
  - Minimum step
  - End-of-line (EOL) spacing
- **Cut layer rules**
  - Cut short
  - Cut spacing

Metal Spacing Rule (Metal short, Non-sufficient-metal-overlap, Parallel run length (PRL) spacing)

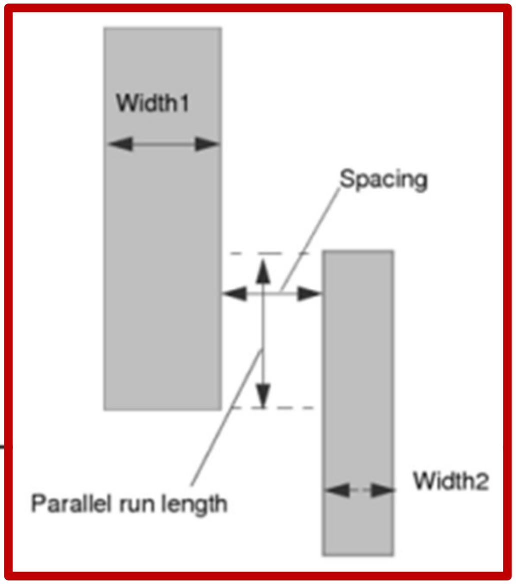
Metal Shape Rule (Minimum width, Minimum step)

Many more design rule types in advanced technologies! See the "LEF5.8" standard, e.g., here.

Metal Spacing Checking

Algorithm 1 Check metal spacing

```
Input: max rectangle m
N ← queryMaxRectangles(m, maxDist)
for all n ≠ m in N do
    if isOverlap(m, n) then
        if getNet(m) = getNet(n) then
            checkNSMetal(m, n)
        else
            checkMetalShort(m, n)
        end if
    else
        checkPRL(m, n)
    end if
end for
```



https://www.ispd.cc/contests/19/Introduction_of_ISPD19_Contest_Problem.pdf

Short Checking

Algorithm 2 Check metal short

```
Input: max rectangles m, n
shortRect ← getIntersection(m, n)
if isFixed(m) AND isFixed(n) then
    return
end if
if isCoveredByPin(shortRect) AND isBlockage(m, n) then
    return
end if
if not hasRouting(shortRect) then
    return
end if
addMarker(MetalShort)
```

NS-Metal Checking

Algorithm 3 Check non-sufficient metal overlap

```
Input: max rectangles m, n
nsRect ← getIntersection(m, n)
if diagLen(nsRect) ≥ minWidth then
    return
end if
if width(m) < minWidth OR width(n) < minWidth then
    return
end if
if hasValid3rdObj(nsRect) then
    return
end if
addMarker(NonSufficientMetalOverlap)
```

PRL Spacing Checking

Algorithm 4 Check parallel run length spacing

```
Input: max rectangles m, n
actVal ← getActualSpacing(m, n)
reqVal ← getRequiredSpacing(m, n)
if actVal ≥ reqVal then
    return
end if
if isFixed(m) AND isFixed(n) then
    return
end if
prlRect ← getIntersection(m, n)
if not hasPolyEdge(prlRect) then
    return
end if
maxWidth ← getMaxWidth(m, n)
if not hasExclusiveRoutingWithin(prlRect, maxWidth) then
    return
end if
addMarker(ParallelRunLengthSpacing)
```

Min Width Checking

Algorithm 5 Check minimum width

```
Input: polygon m
N ← slicePolygon(m, vertical)
for all n in N do
    if ySpan(n) ≥ minWidth then
        return
    end if
    if not hasRouting(n) then
        return
    end if
    addMarker(MinimumWidth)
end for
N ← slicePolygon(m, horizontal)
for all n in N do
    if xSpan(n) ≥ minWidth then
        return
    end if
    if not hasRouting(n) then
        return
    end if
    addMarker(MinimumWidth)
end for
```

Min Step Checking

Algorithm 6 Check minimum step

```
Input: polygon edge e
if length(e) < minStepLength then
    return
end if
initializeBBox(bbox, endPoint(e))
beginEdge ← e
numEdges ← 0
while beginEdge ≠ nextEdge(e) do
    e ← nextEdge(e)
    updateBBox(bbox, endPoint(e))
    if length(e) < minStepLength then
        numEdges ← numEdges +1
    else
        break
    end if
end while
if e = beginEdge then
    return
end if
if numEdges <= maxEdges then
    return
end if
if not hasRoute(bbox) then
    return
end if
addMarker(MinimumStep)
```

EOL Spacing Checking

Algorithm 7 Check end-of-line spacing

```
Input: polygon edge e
if len(e) ≥ eolWidth then
    return
end if
if not hasParallelEdge(e) then
    return
end if
E ← queryPolygonEdge(e, eolWithin, eolSpacing)
for all e' in E do
    eolRect ← getIntersection(e, e')
    if not isEmpty(eolRect) then
        return
    end if
    if not hasRoute(e) then
        return
    end if
    addMarker(EndOfLineSpacing)
end for
```

Cut Spacing Checking

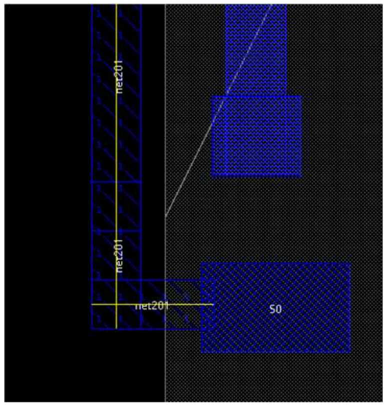
Algorithm 8 Check cut spacing

```
Input: cuts m, n
actVal ← getActualSpacing(m, n)
reqVal ← getRequiredSpacing(m, n)
if actVal ≥ reqVal then
    return
end if
if isFixed(m) AND isFixed(n) then
    return
end if
if not hasAdjCuts(m) then
    return
end if
if not hasParallelOverlap(m, n) then
    return
end if
if not hasArea(m, n) then
    return
end if
addMarker(CutSpacing)
```

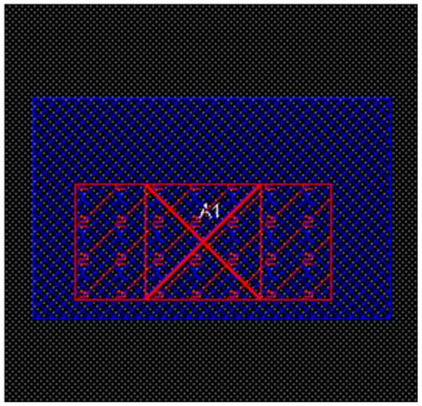
Dynamic Programming-Based Multi-Level Pin Access Analysis

Motivation

- Pin access = wire / via connection to access a pin



Wire access



Via access

- Critical to decrease DRCs in detailed routing
  → Failure results in repeating violation patterns

- **Need robust & scalable pin access analysis (!)**

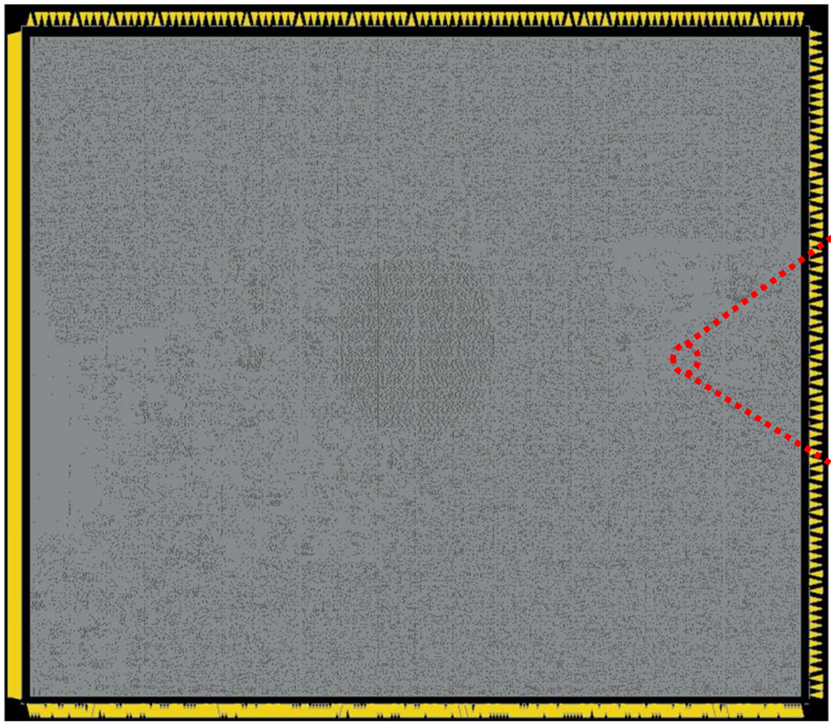
Previous Works / Our Work

- Existing work [Han15] assumes **on-track** access
- Usually assume alignment between routing track and placement site
  → Not always true (ISPD18/19 contests)
- LUT-based abutting cell pair analysis [Xu16]
  → Not scalable (> 10M combinations)

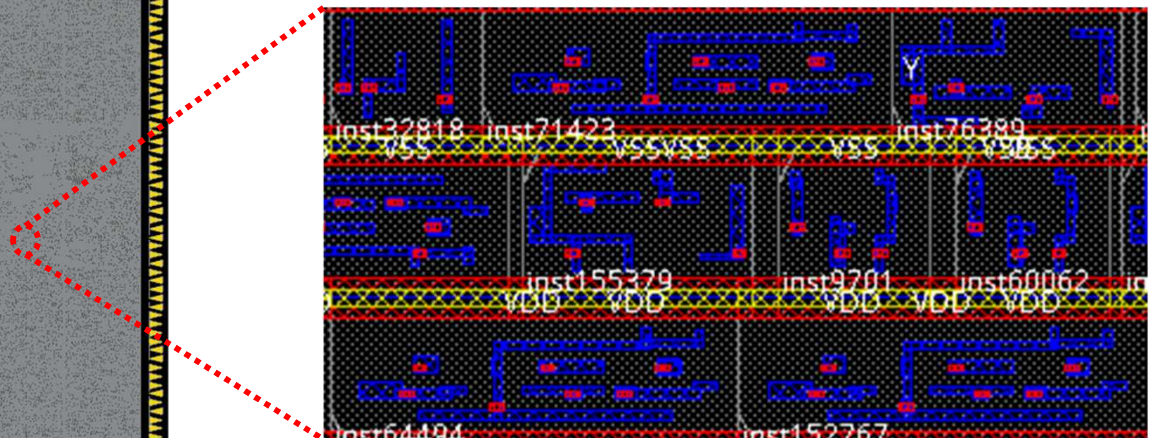
Our work:
(i) Robust pin access point enumeration
(ii) Boundary conflict-aware access pattern enumeration
(iii) Dynamic programming-based access pattern selection for standard cell instance cluster

What Does Pin Access Analysis Do?

- Testcase: ISPD18_test10
  - 290K standard cell instances
  - 992K nets
- DRC clean pin access pattern selection in 241s



Whole design layout

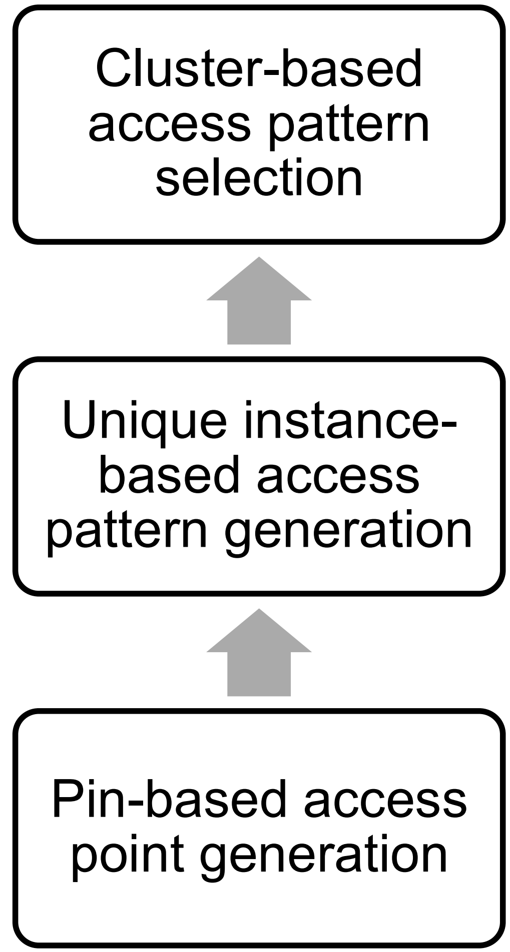


Zoomed-in region
(red dots are via access locations)

How to Find Such Access Points?

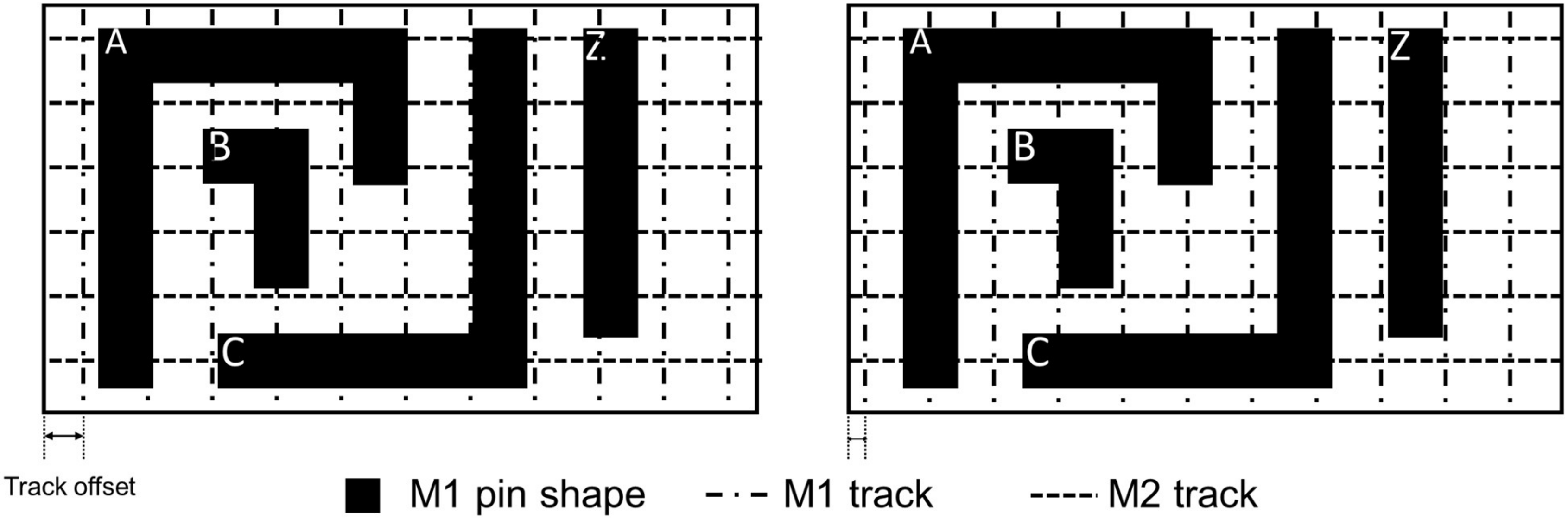
- **Multi-level hierarchical pin access analysis**
  - Unique instance pin level
  - Unique instance level
  - Instance cluster level
- **Scalable memory usage**
  - On-demand design-based analysis
- **Scalable runtime**
  - DRC check on the fly

    → More than 2M DRC engine calls in 8min with single thread



Unique Instance

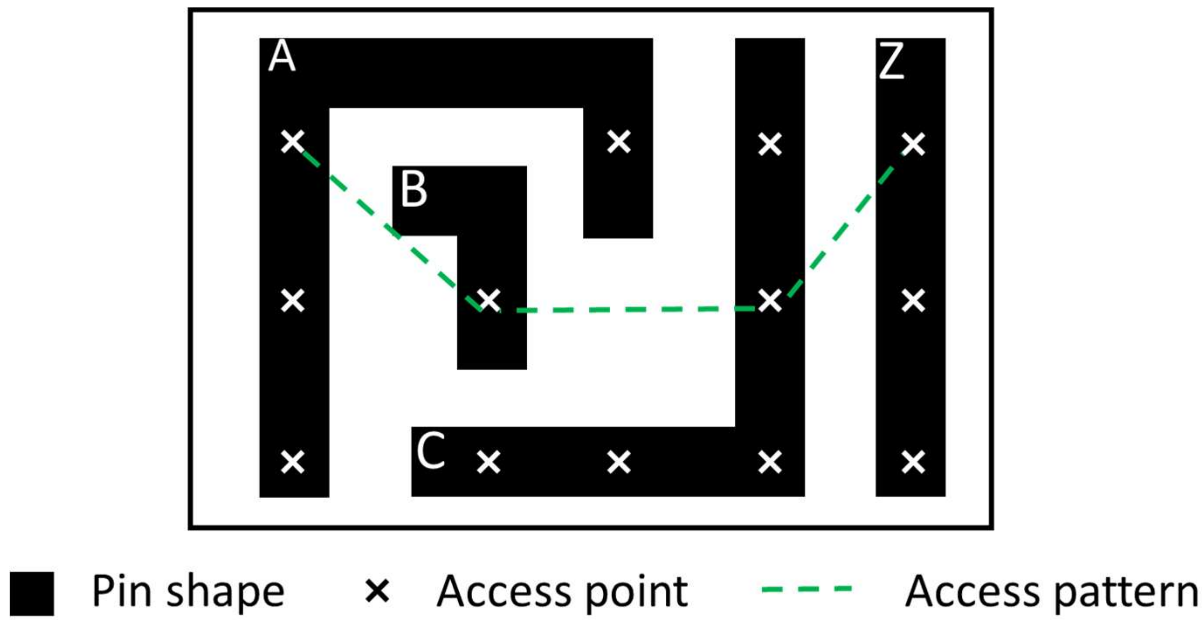
- To address track-placement site misalignment
- Defined by a **signature** consisting of
  - Cell master (e.g., BUFFX4)
  - Orientation
  - Offsets to all track patterns
- Two cell instances point to the same unique instance if they share the same **signature**



Two different unique instances due to M1 track offset

Definitions

- **Access point** (for a pin)
  - A location (x, y, layer) that detailed router (DR) can make route to
- **Valid access point**
  - An access point that allows DRC-clean routing
- **Valid access pattern**: combination of mutually DRC-clean access points (one access point per pin)



Access Point Quality Assessment

- Goal: an evaluation system compatible with a broad range of technology nodes

- Four coordinate types

Type	Cost
On-track	1
Half-track	2
Shape-center	3
Enclosure boundary	4

- Quality of an access point = sum of coordinate type costs for both x and y coordinates

Coordinate Types

1. **On-track**: On preferred and non-preferred routing track of upper layer
2. **Half-track**: At midpoint between two neighboring routing tracks
3. **Shape-center**: At midpoint between left and right (or top and bottom) coordinates of a rectangular pin shape
4. **Enclosure boundary**: Via enclosure aligns with pin shape boundary

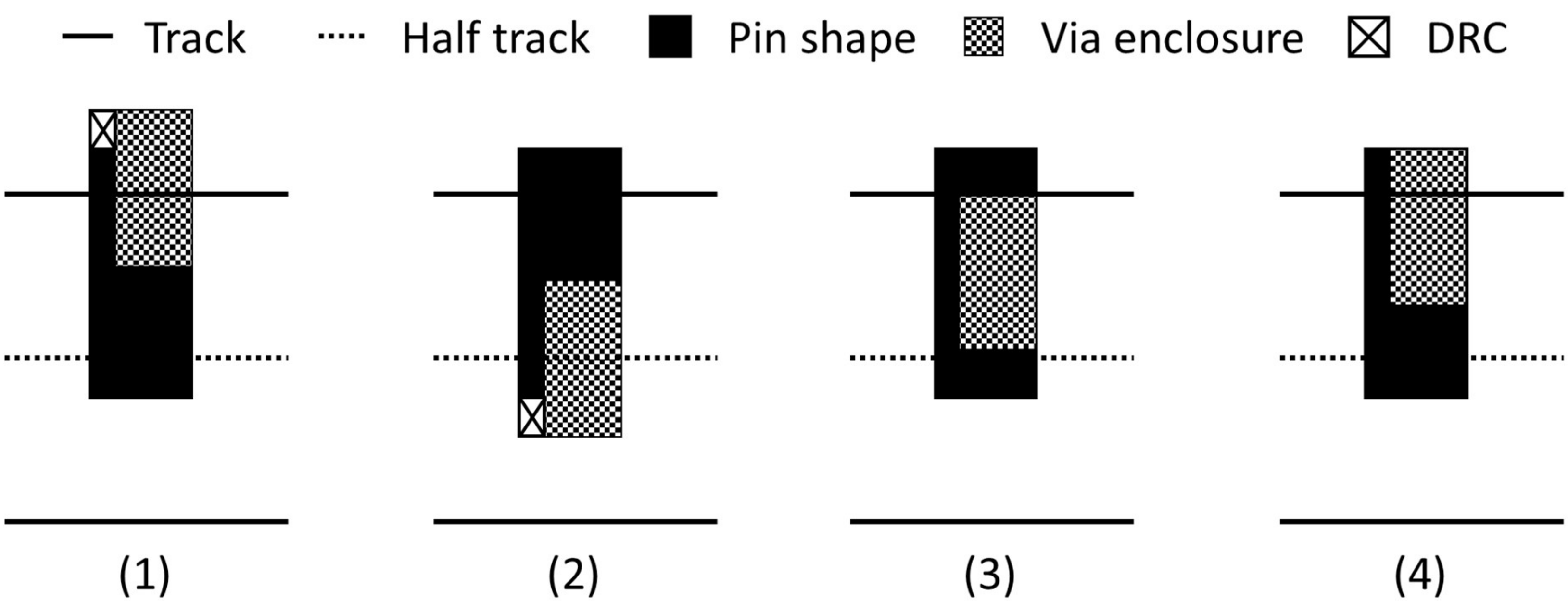


Illustration of y-coordinate types

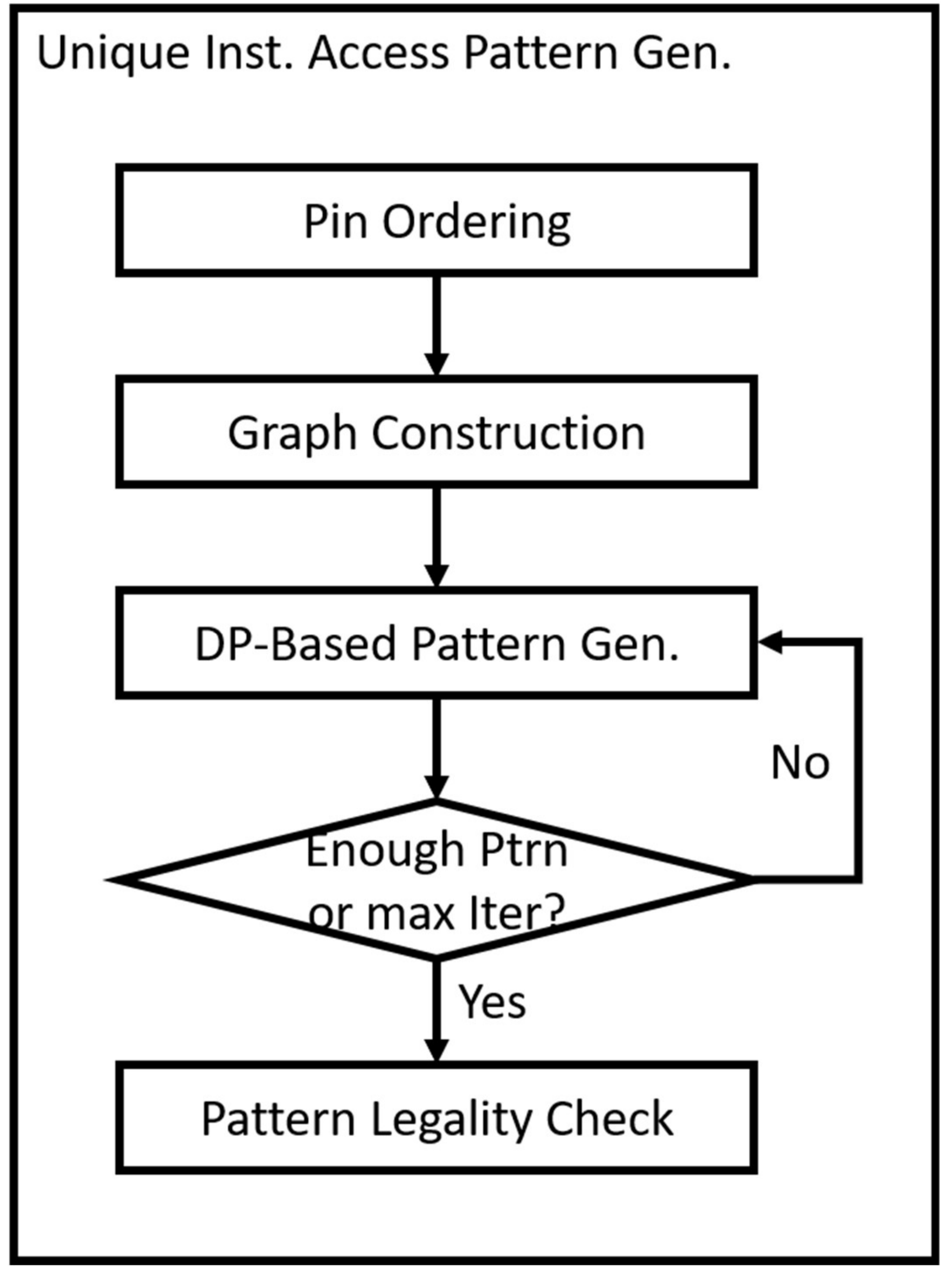
Pin-Based Access Point Generation

Algorithm 1 Pin-based access point generation

```
Inputs: pin, track patterns tps, viadefs vias
Output: valid access points aps
for all nonPreferredDirCoordType t1 ∈ {0, 1, 2} do
    for all preferredDirCoordType t0 ∈ {0, 1, 2, 3} do
        tmpAps ← genAccessPoint(pin, tps, vias, t0, t1)
        for all ap ∈ tmpAps do
            if isValid(ap) then
                aps += ap
            end if
        end for
        if |aps| ≥ k then
            return
        end if
    end for
end for
```

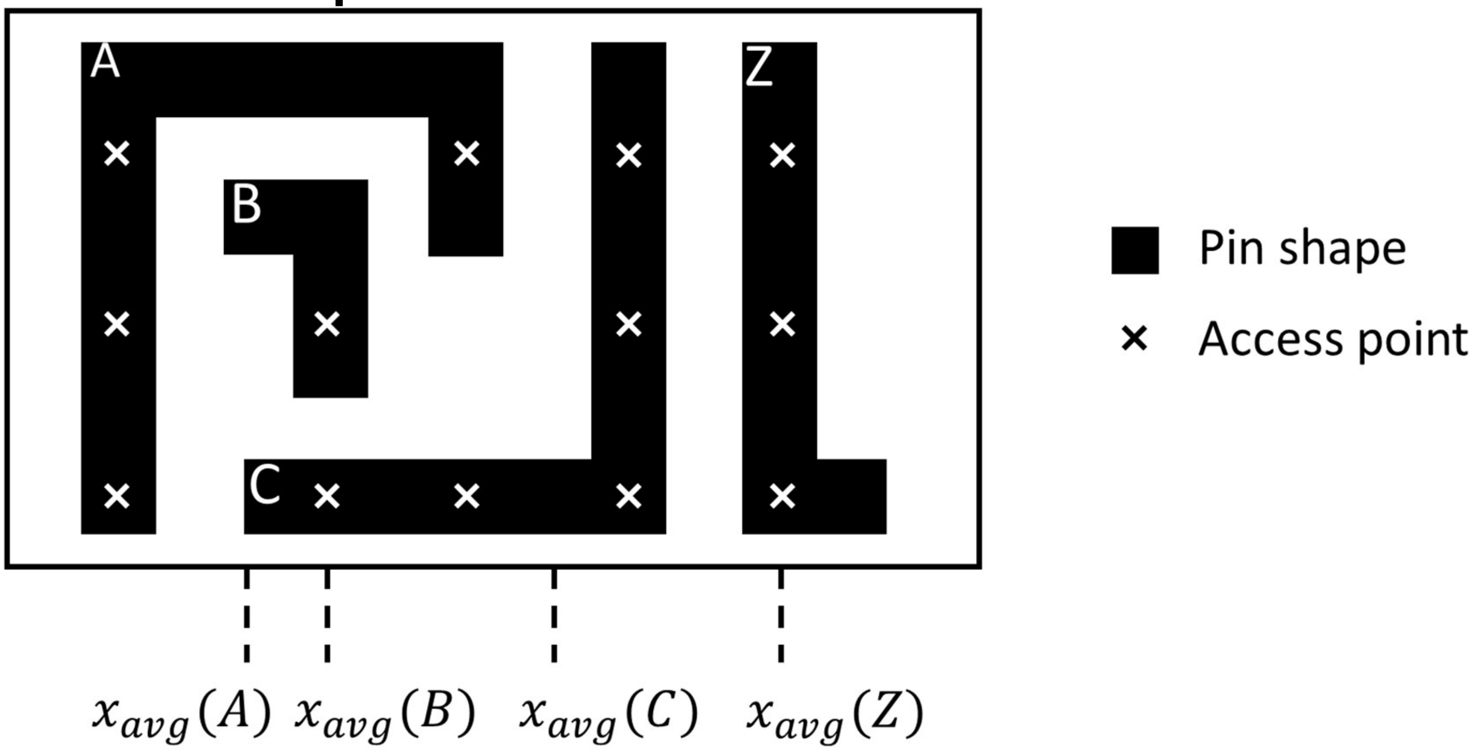
Unique Instance-Based Access Pattern Generation

- **Input**
  - Valid access points of pins in a unique instance
- **Output**
  - Valid access patterns



Pin Ordering

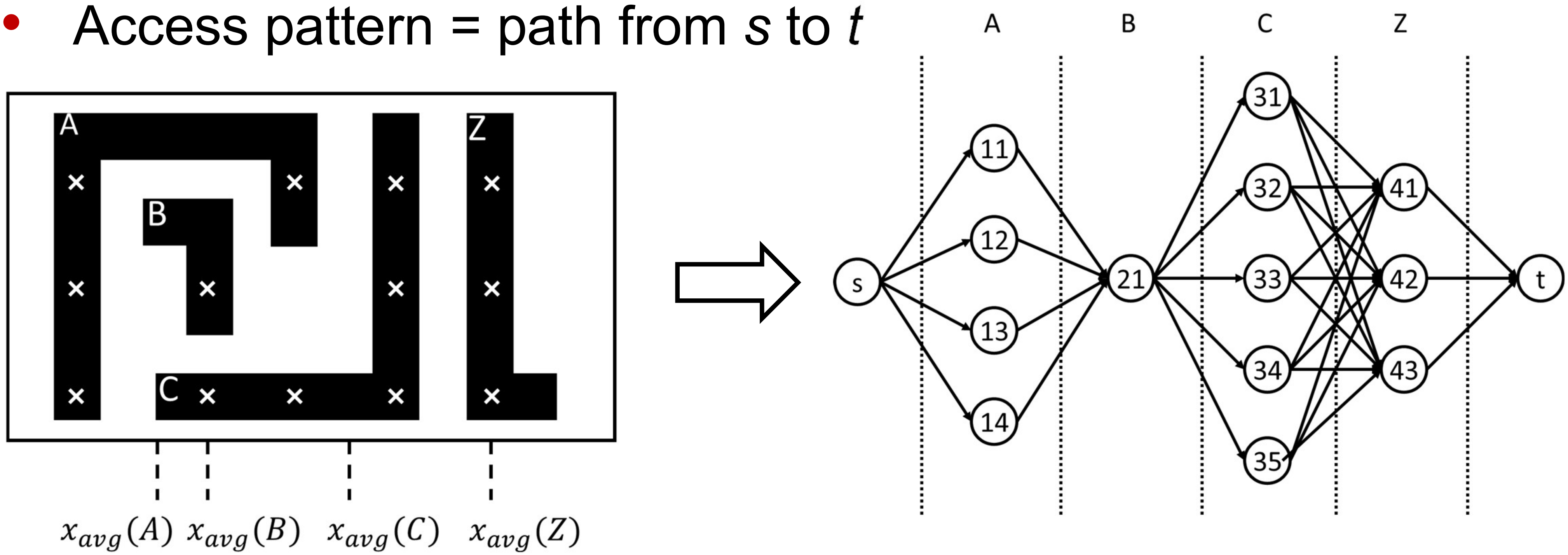
- Sort pins according to their average x coordinate of valid access points



- Idea: neighbors in sorted pin list are more likely to have conflicts due to DRC
  - I.e., pin A and pin B are more likely to have conflict compared to pin A and pin C

Graph Construction

- Vertex = access point
  - Marked with pin index and access point index
    - E.g., 23 means the third access point of the second pin
  - *s* and *t* are virtual start and end points
- Pin correspond to a “group” of vertices in graph
- Edge exists between pair of access points from neighboring groups, weighted by physical distance
- Access pattern = path from *s* to *t*



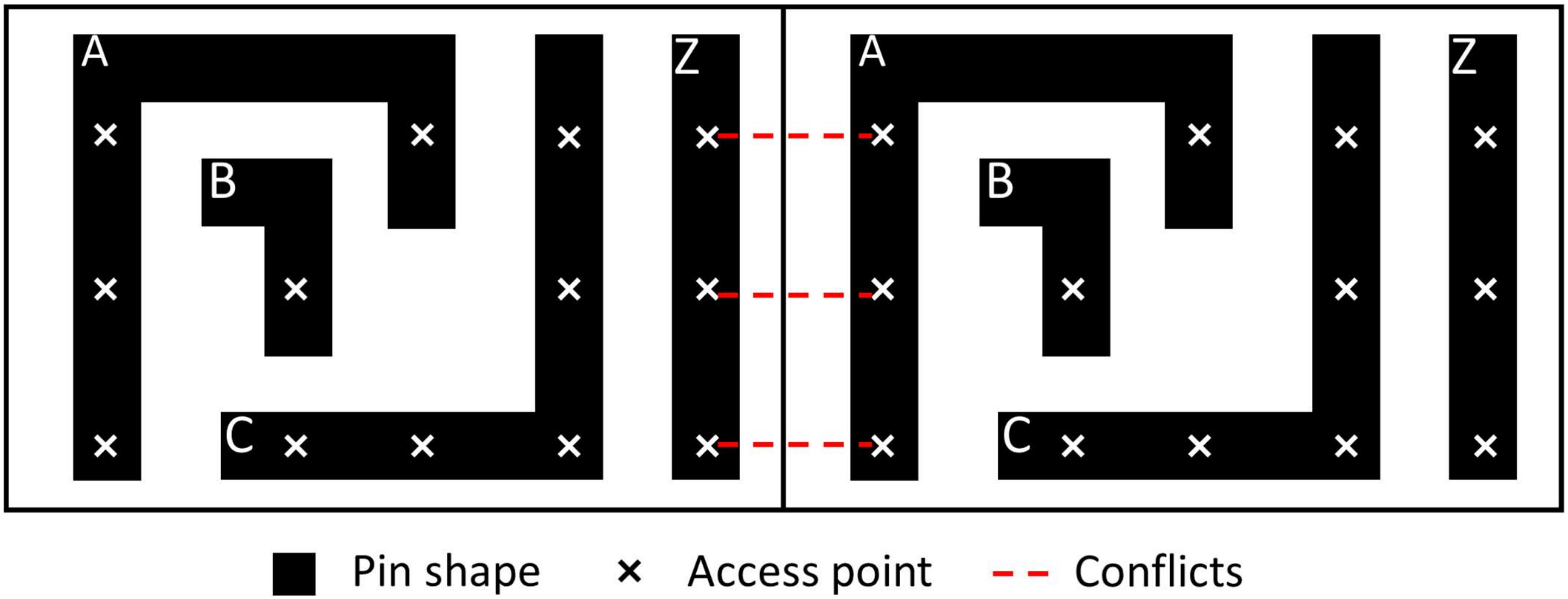
Dynamic Programming-Based Pattern Generation

Algorithm 2 Access pattern generation

```
Inputs: graph G(V, E)
Output: access patterns APs
Initialize array dp[m][n] G(V, E)
for all currPinIdx m do
    for all currApIdx n do
        for all prevApIdx n' do
            prev ← aps[m − 1][n']
            curr ← aps[m][n]
            edgeCost ← getEdgeCost(prev, curr)
            pathCost ← prev.cost + edgeCost
            if pathCost < curr.cost then
                curr.cost ← pathCost
                curr.prev ← prev
            end if
        end for
    end for
end for
APs += traceBack()
return APs
```

Iterative Edge Penalty Method

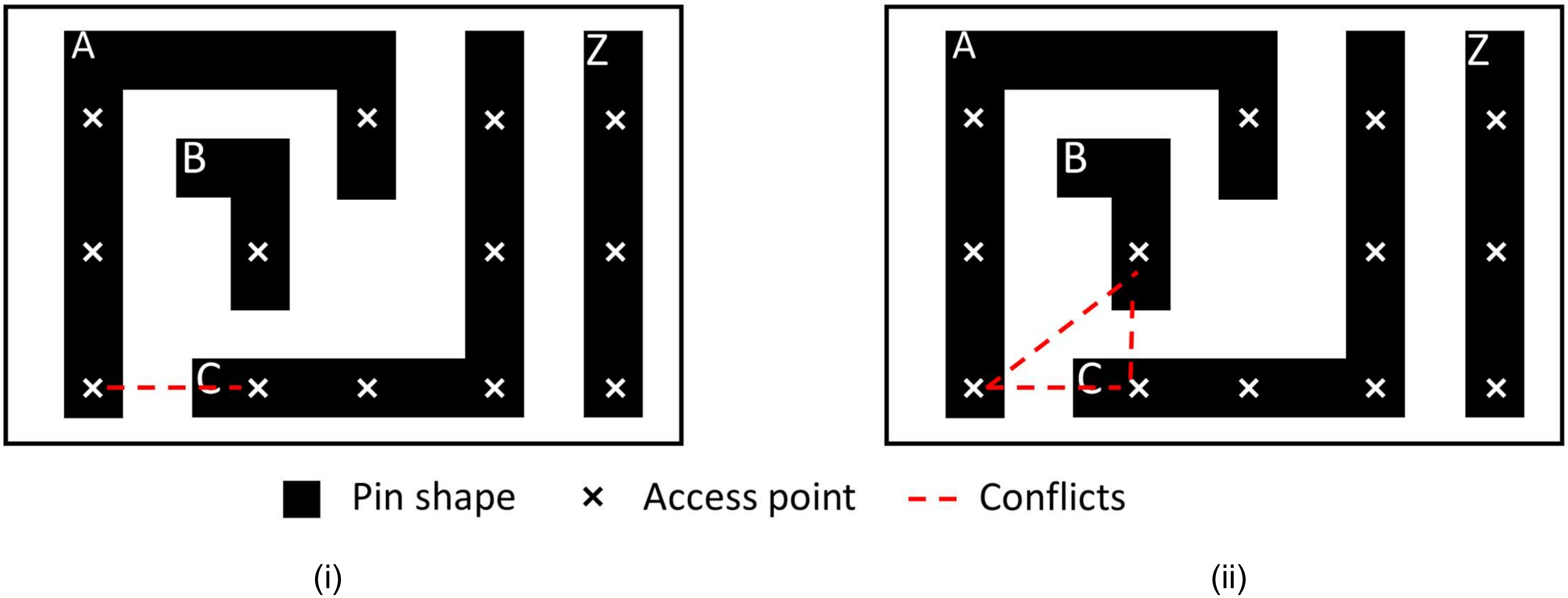
- Inter-cell pattern conflicts between cell-boundary pins
  → Need to encourage to choose different boundary pin access point



- Solution: add penalty cost to boundary pin access points if they have been selected in existing pattern

Pattern Legality Check

- Use DRC check engine to validate access pattern
  i. Violation can occur between non-neighboring pins
  ii. Some design rules check multiple objects (access points)

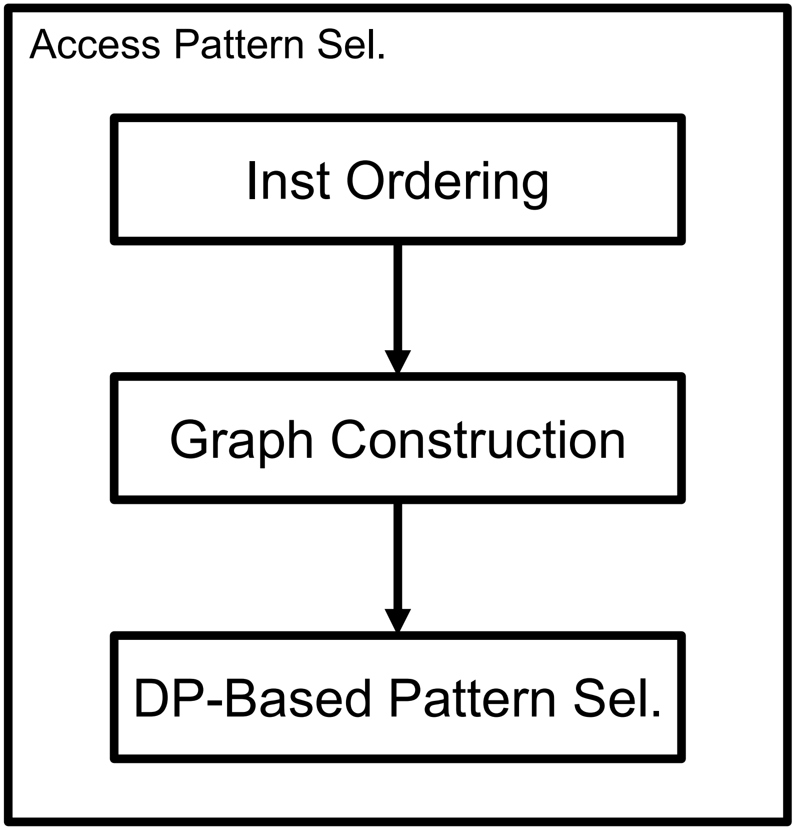


- Only **DRC-clean** patterns will be seen in next stage

Cluster-Based Access Pattern Selection

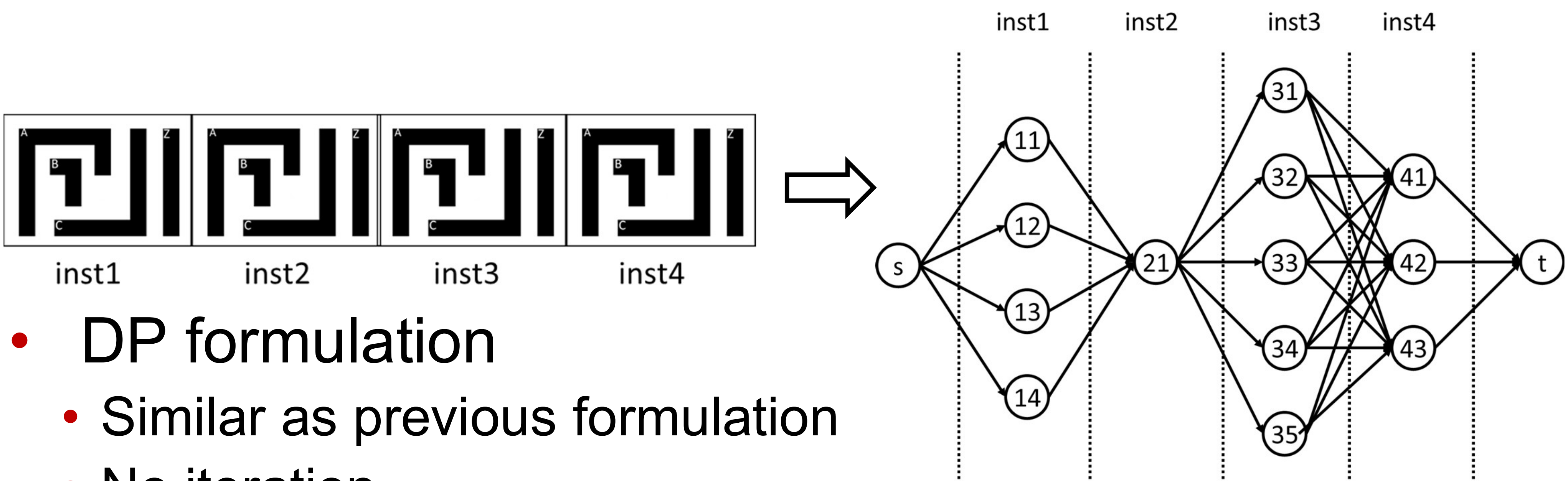
- **Inputs**
  - Instances in a cluster (of the same row)
  - Access patterns of each unique instance
  - Map from instance to corresponding unique instance

- **Output**
  - Access pattern for each instance in the cluster with minimized overall cost



Cluster-Based Access Pattern Selection

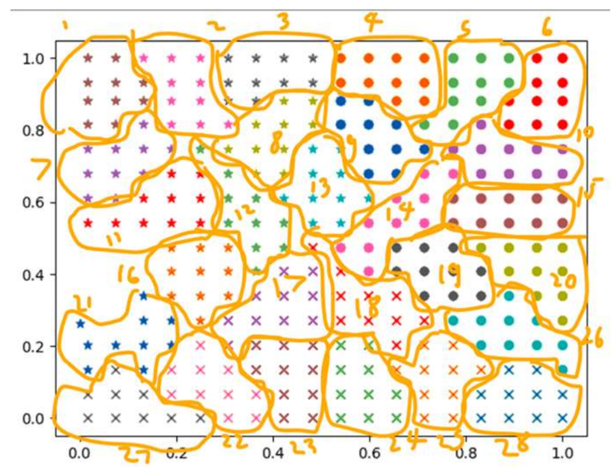
- Instance ordering
  - Sort instances in the cluster according to x coordinate of the lower-left corner
- Graph construction
  - Vertex = access pattern
  - Shortest path from *s* to *t* is the best pattern combination



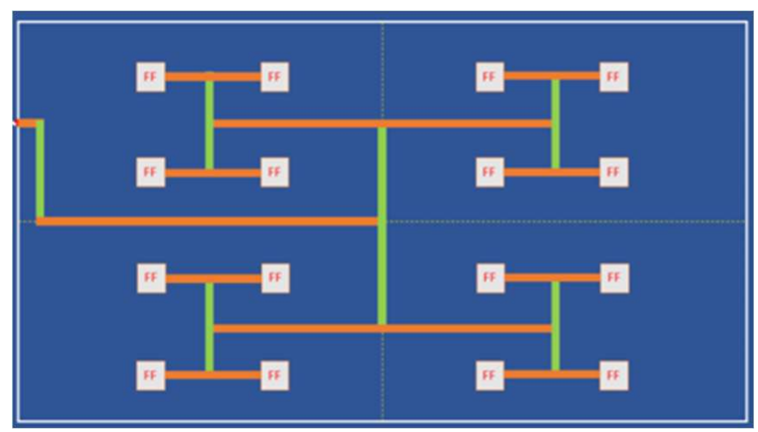
- DP formulation
  - Similar as previous formulation
  - No iteration

CTS

CTS Main Steps

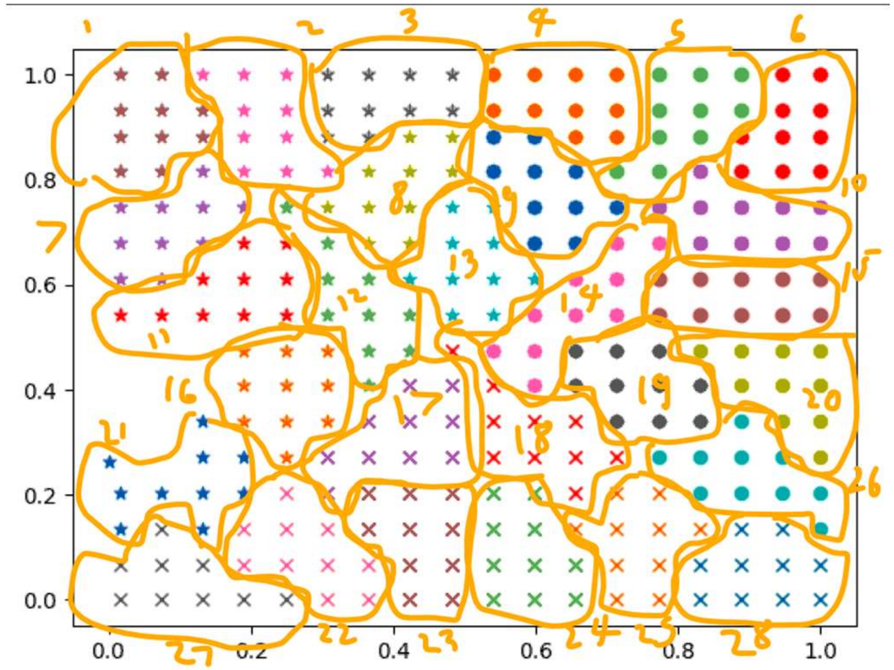


- Sink clustering
  - Sequential elements are grouped into a fixed number of clusters based on their locations
- Tree construction and balancing
  - Buffers are inserted based on some structure, e.g., hierarchical H-Tree
  - Tree lengths are balanced such that clock skews are minimized
- LDRC (electrical rules) repair
  - LDRC violations are repaired during or after CTS
    - Max transition, max capacitance, max wire length, etc.



Sink Clustering

- Group sequential elements based on their locations to produce the best results (e.g., minimum wire length)
  - These parameters can be specified manually or determined automatically
    - Cluster size
    - Cluster diameter
- All elements in the cluster will be driven by the same buffer



Early "TritonCTS" versions used spacefilling curves to perform sink clustering !

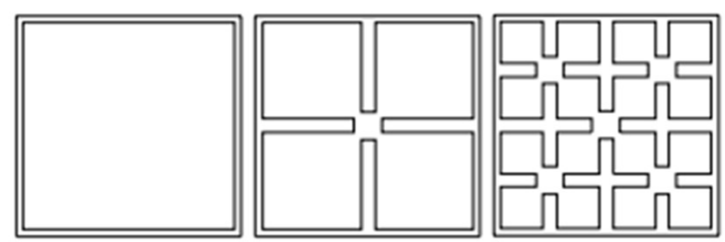


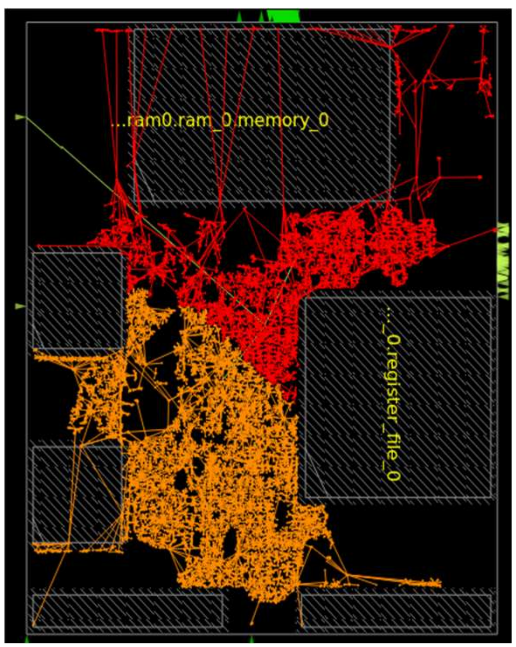
Figure 2: The planar Sierpinski spacefilling curve.

https://vlsicad.ucsd.edu/Publications/Conferences/32/c32.pdf

Obstruction-Aware CTS

Clock buffers should not be placed on top of macros, placement blockages or another clock buffers

- Detailed placement may displace “illegal” buffers and cause timing to change after CTS
- New “legal” buffer locations need to preserve balanced clock tree
- Obstruction-aware CTS can reduce legalizer displacement by up to 4X



sky130hd/microwatt **without** obstruction-aware CTS

sky130hd/microwatt **with** obstruction-aware CTS

Thanks: Dr. Cho Moon, Precision Innovations

OpenROAD CTS Commands

<table>
<tr><th>Command</th><th>Description</th><th>Example Output</th></tr>
<tr><td><code>clock_tree_synthesis</code></td><td>Build a balanced Htree by choosing appropriate clock buffers</td><td><pre>
[INFO CTS-0050] Root buffer is BUF_X4.
[INFO CTS-0051] Sink buffer is BUF_X4.
[INFO CTS-0052] The following clock buffers will be used for CTS:
                    BUF_X4
. . .
[INFO CTS-0017]   Max level of the clock tree: 5.
[INFO CTS-0098] Clock net "clk"
[INFO CTS-0099] Sinks 2537
[INFO CTS-0100] Leaf buffers 96
[INFO CTS-0101] Average sink wire length 9247.25 um
[INFO CTS-0102] Path depth 18 - 19
[INFO CTS-0207] Leaf load cells 62
[INFO RSZ-0058] Using max wire length 693um.
[INFO RSZ-0047] Found 33 long wires.
[INFO RSZ-0048] Inserted 94 buffers in 33 nets.
</pre></td></tr>
<tr><td><code>repair_clock_nets</code></td><td>Fixes LDRC violations including max wire length</td><td><pre>
[INFO RSZ-0058] Using max wire length 2154um.
</pre></td></tr>
<tr><td><code>report_clock_skew</code></td><td>Report worst clock skew for each clock signal in the design</td><td><pre>
Clock clk
   1.26 source latency inst_7_12/clk ^
  -1.13 target latency inst_8_12/clk ^
   0.00 CRPR

   0.13 setup skew
</pre></td></tr>
<tr><td><code>report_checks -format full_clock_expanded</code></td><td>Report timing violations including clock paths</td><td><pre>
Startpoint: dp.rf.rf[31][3]$_DFFE_PP_
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: aluout[0] (output port clocked by clk)
Path Group: clk
Path Type: max

Fanout   Cap   Slew   Delay   Time   Description

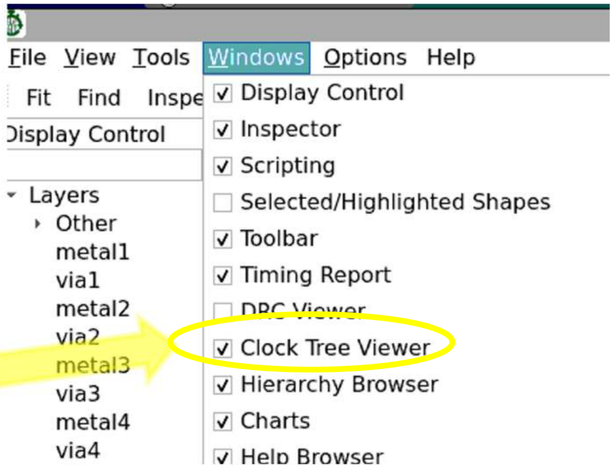
                          0.00   0.00   clock clk (rise edge)
                          0.00   0.00   clock source latency
     1   0.09   0.00   0.00   0.00 ^ clk (in)
                                         clk (net)
                  0.00   0.00   0.00 ^ clkbuf_0_clk/A (sky130_fd_sc_hd__clkbuf_16)
     8   0.21   0.22   0.25   0.25 ^ clkbuf_0_clk/X (sky130_fd_sc_hd__clkbuf_16)
                                         clknet_0_clk (net)
                  0.22   0.00   0.25 ^ clkbuf_3_3__f_clk/A (sky130_fd_sc_hd__clkbuf_16)
    17   0.23   0.24   0.34   0.59 ^ clkbuf_3_3__f_clk/X (sky130_fd_sc_hd__clkbuf_16)
                                         clknet_3_3__leaf_clk (net)
                  0.24   0.00   0.59 ^ clkbuf_leaf_47_clk/A (sky130_fd_sc_hd__clkbuf_16)
    11   0.04   0.06   0.20   0.79 ^ clkbuf_leaf_47_clk/X (sky130_fd_sc_hd__clkbuf_16)
                                         clknet_leaf_47_clk (net)
                  0.06   0.00   0.79 ^ dp.rf.rf[31][3]$_DFFE_PP_/CLK
(sky130_fd_sc_hd__dfxtp_2)
     3   0.01   0.03   0.32   1.11 v dp.rf.rf[31][3]$_DFFE_PP_/Q
(sky130_fd_sc_hd__dfxtp_2)
                                         dp.rf.rf[31][3] (net)
</pre></td></tr>
</table>

Clock Tree Viewer

Open GUI

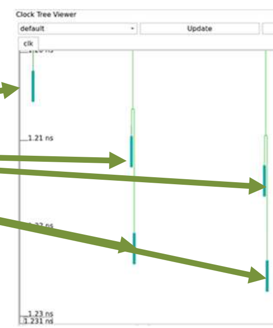
- gui::show

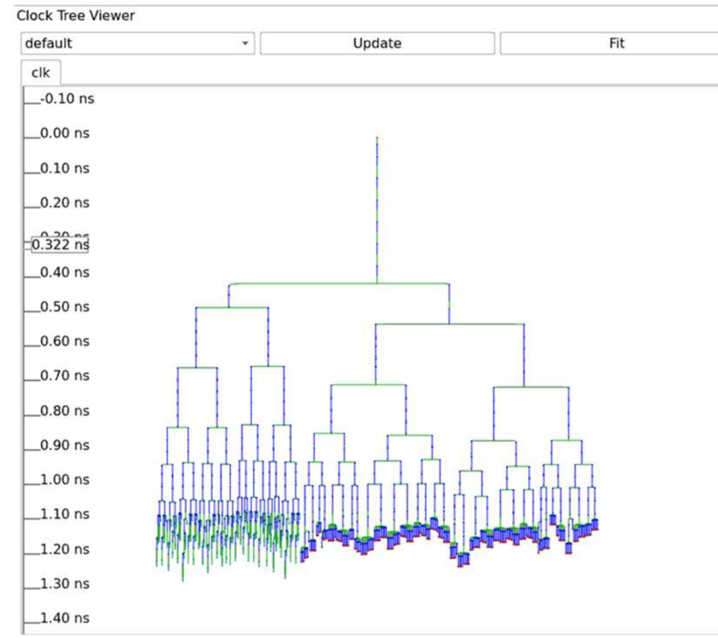
Enable “Clock Tree Viewer” if not enabled



Clock tree viewer shows latencies at all sinks

- Red sinks = FF/latches
- Green sinks = macros
  - Insertion delays are added to macro sinks

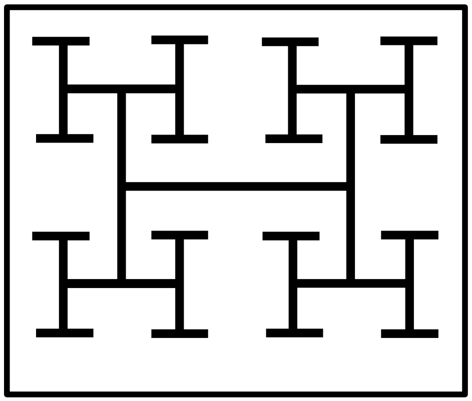




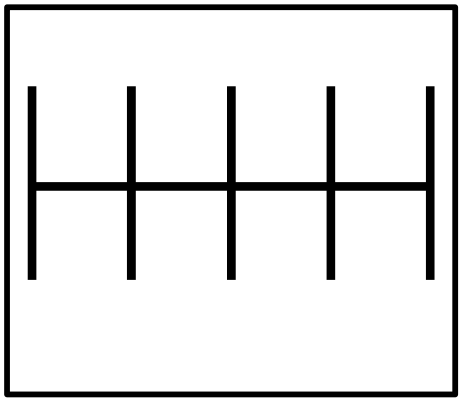
Generalized H-Tree Concept

- Structured clock trees

K. Han, A. B. Kahng and J. Li, "Optimal Generalized H-Tree Topology and Buffering for High-Performance and Low-Power Clock Distribution", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **https://vlsicad.ucsd.edu/Publications/Journals/j128.pdf**.



H-tree



"Fishbone"

	H-tree	Fishbone
Skew	<	
Wirelength	>	
Latency	>	
Power	>	

Generalized H-tree (GH-tree)

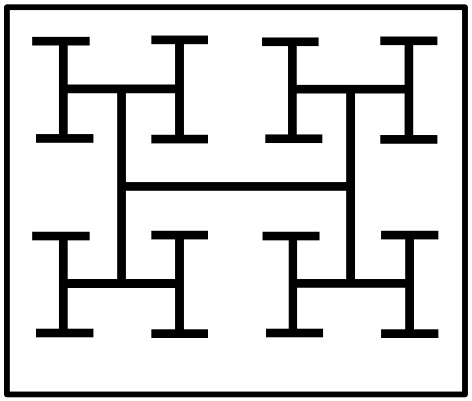
Can we mix two clock structures to have better tradeoff between clock power vs. skew or latency?

- History: (1) Bakoglu's 1988 book made H-tree approach well-known. Cadence CTGen (Dr. Lars Hagen), mid-1990s, started trend toward "fishbone" style – save capacitance!
- These days: on-chip variation (OCV) derates are costly, so goal is to reduce insertion delay (== "latency").

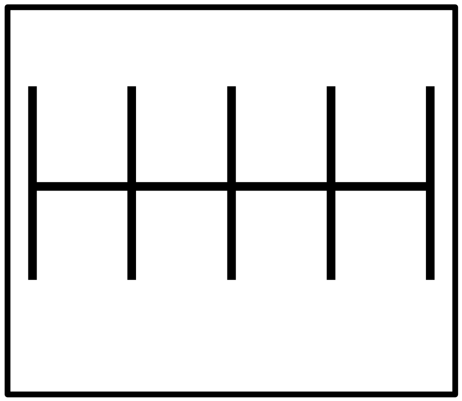
Generalized H-Tree Concept

- Structured clock trees

K. Han, A. B. Kahng and J. Li, "Optimal Generalized H-Tree Topology and Buffering for High-Performance and Low-Power Clock Distribution", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **https://vlsicad.ucsd.edu/Publications/Journals/j128.pdf**.



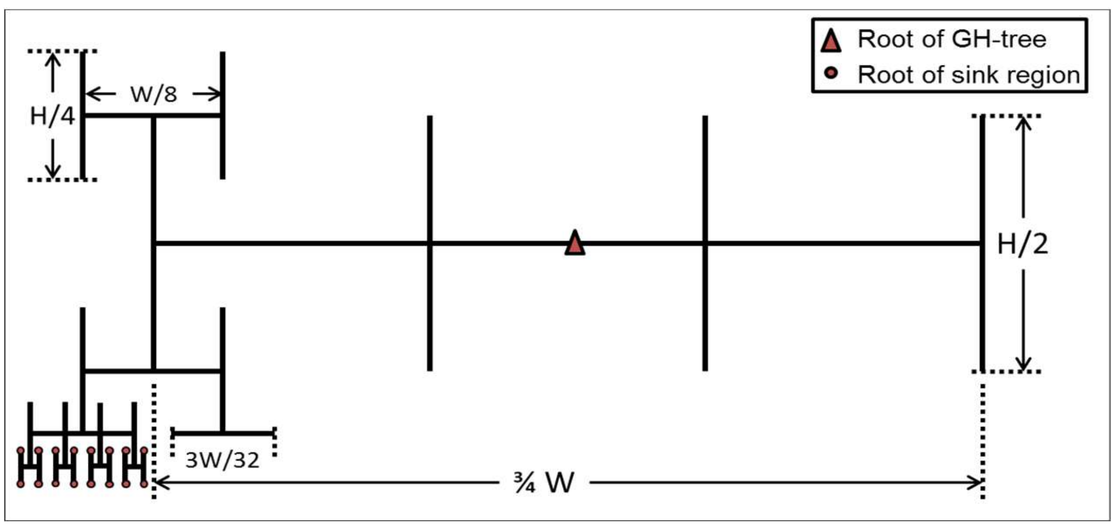
H-tree



"Fishbone"

	H-tree	Fishbone
Skew	<	
Wirelength	>	
Latency	>	
Power	>	

Generalized H-tree (GH-tree)

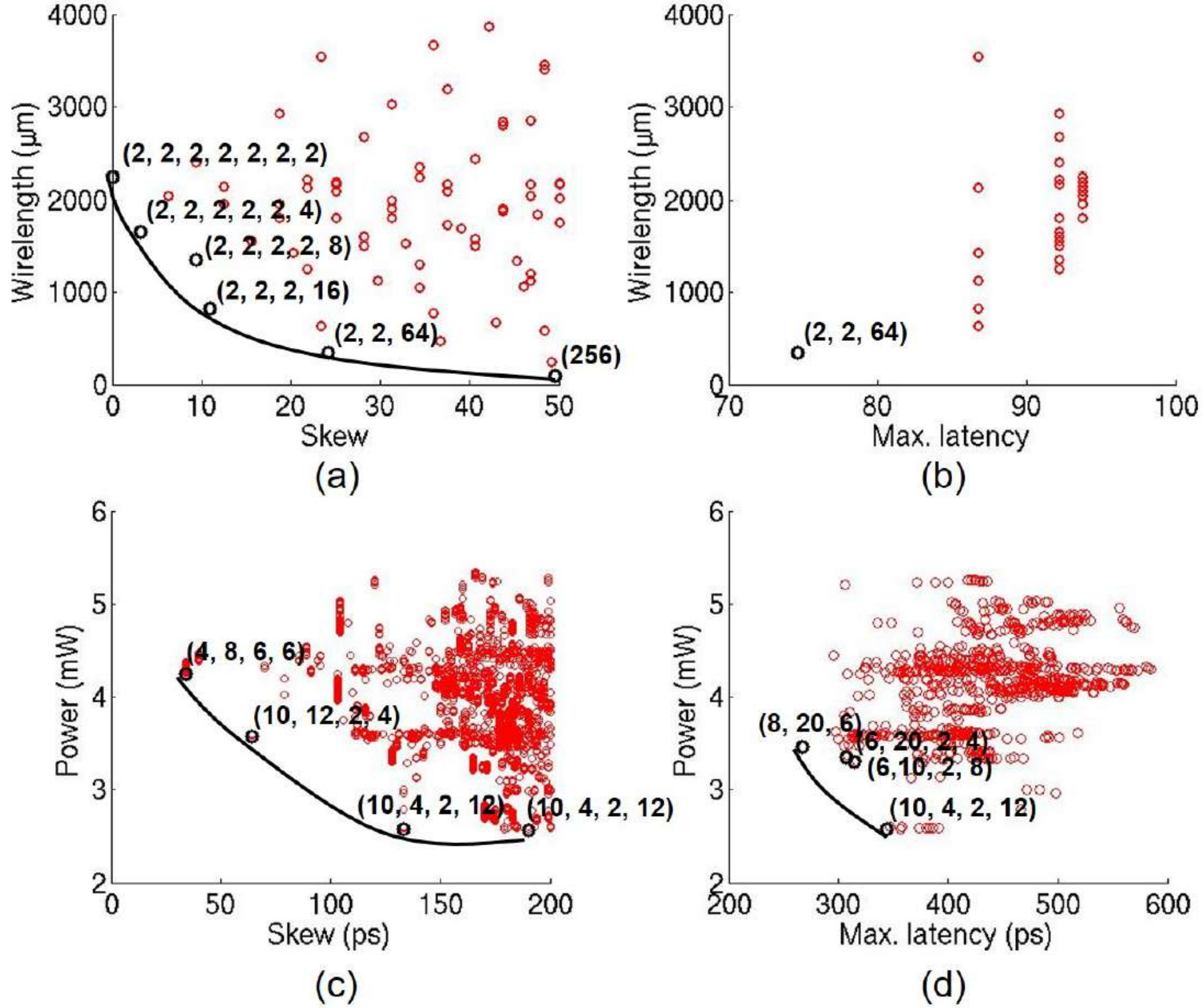


a balanced tree topology with arbitrary branching factor at each level

GH-tree with depth P = 8 and branching factors (4, 2, 2, 2, 4, 2, 2, 2)

Idea: Capture/Explore Tradeoff ("Pareto" Frontier)

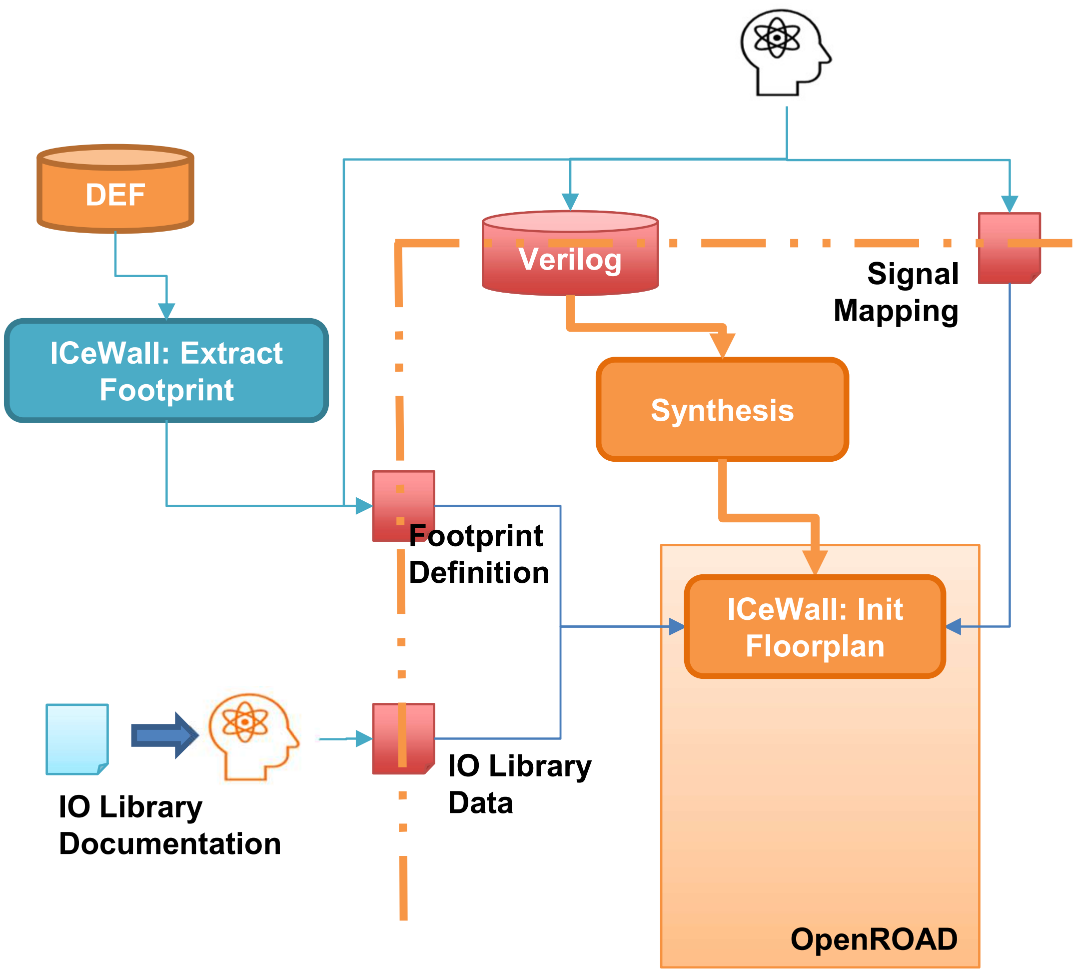
- **(Recall: floorplan shape functions ?)**



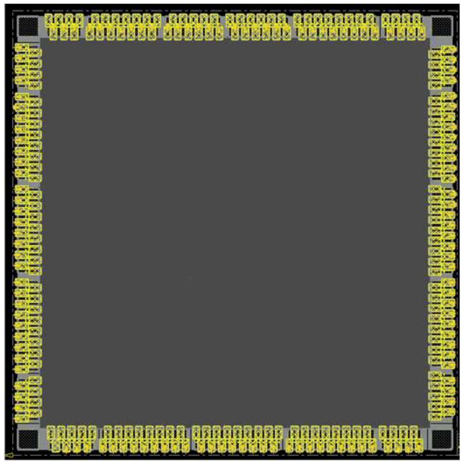
Also: Routing to IOs: pad (ICeWall)

SOC Integration and Planning: ICeWall Padring Gen

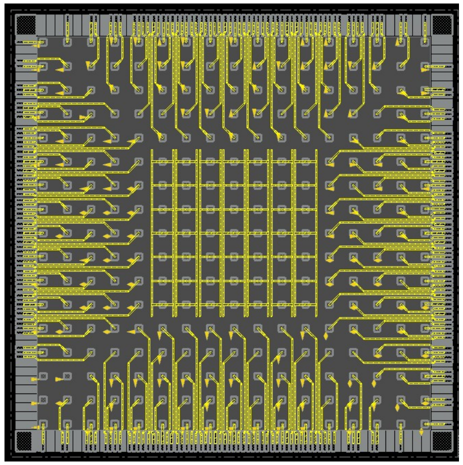
- **Starts with:**
  - Verilog netlist with signal IO pads for simulation and STA
  - Power/ground IO cells may be present
  - IO cell data (signal, P/G, fillers, …) from library documentation
- **Footprint file** defines where each padcell is to be placed in the padring – supports reuse of pre-existing padframes
- **Signal mapping file** defines which signal in the Verilog is to be associated with which padcell in the padring
  - + Auto-assignment capability in ICeWall
- **Decouples footprint and signal mapping** for padframe reuse



ICeWall Padring Examples

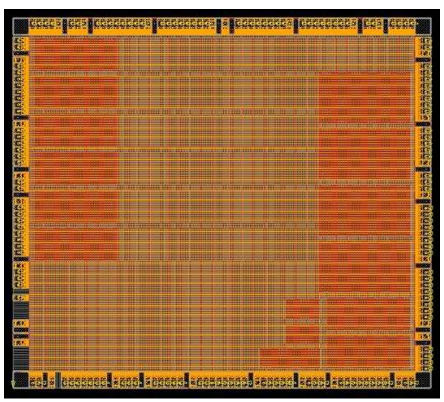


GF12LP BP-1,
staggered pads



GF12LP BP-1,
as a flipchip

SKY130
coyote, + pads



What designers ask for …

- Determining the number of required P/G pads to be provided as callback functions to allow to encapsulate specs from library documentation
- Definition of padring segments for analog signals, PHYs, different IO voltages, etc.
- Definition of control cells that are required on a per-IO cell basis

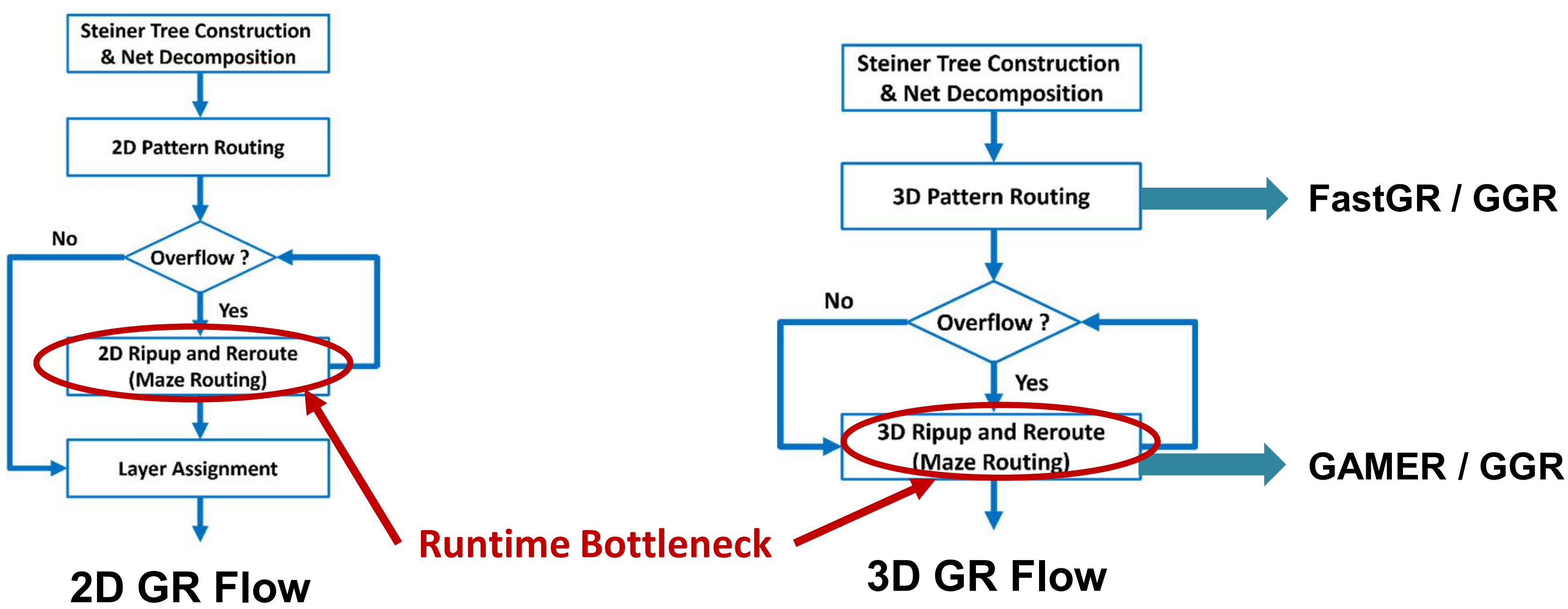
Thanks: Colin Holehouse, Arm

Also: GPU-Accelerated GRT

Summary of Previous Methods

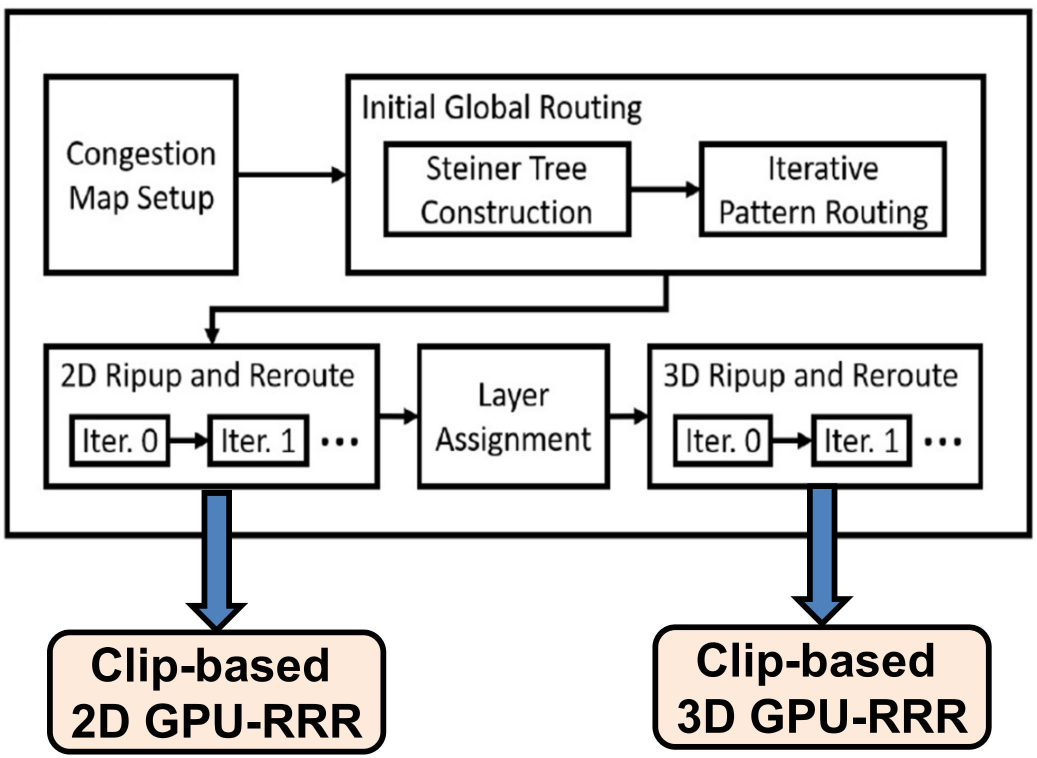
- 2D-GPU-accelerated GRs are based on FastRoute4.1
  - **SPRoute** and **SPRoute2.0**: implement parallel maze routing
- 3D-GPU-accelerated GRs are based on CUGR
  - **FastGR** and **GGR**: implement parallel L/Z-shape pattern routing
  - **GAMER** and **GGR**: implement parallel maze routing
    - Replace the A* search algorithm with the parallel n-bend pattern routing algorithm

Route a batch of non-overlapping nets concurrently !

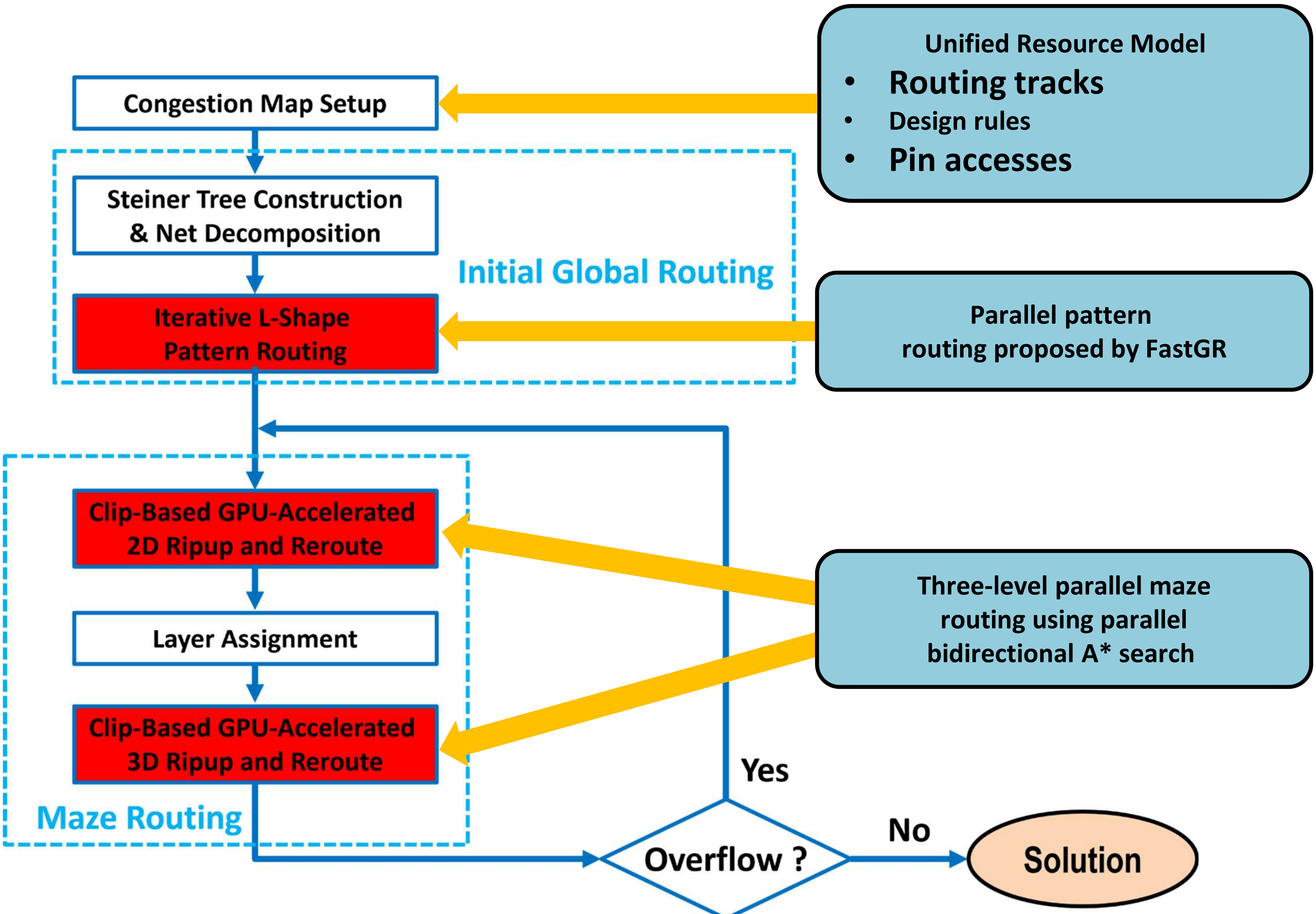


Proposed GPU-Accelerated TritonRoute-GR

- Our GPU-GR is based on TritonRoute-GR
  - TritonRoute-GR adopts a two-step approach (2D + 3D GR)
    - 2D global routing can effectively reduce the solution space
    - 3D global routing can further optimize the solution locally
  - Replace the original 2D-RRR and 3D-RRR with corresponding GPU-accelerated GPU-RRR

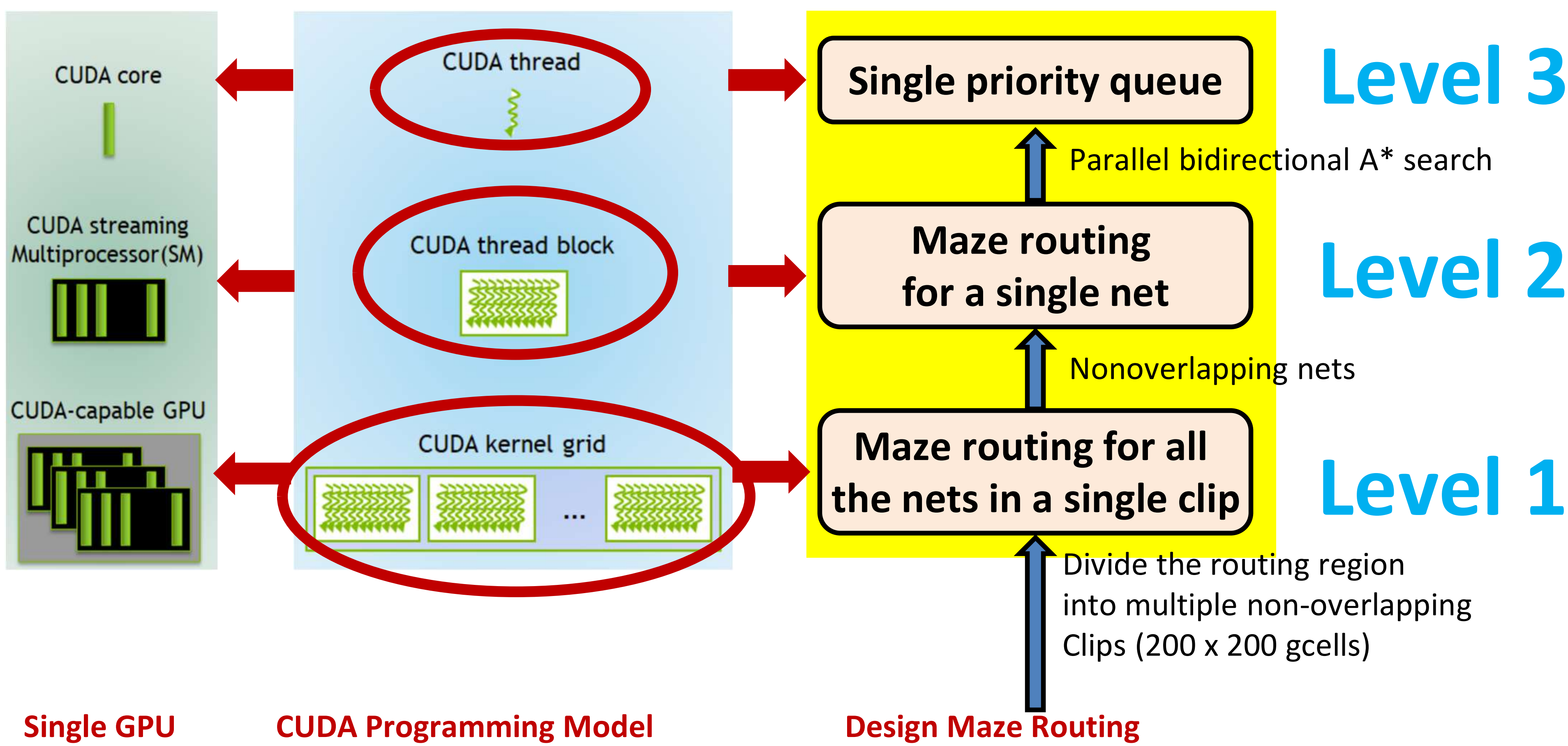


Current Approach



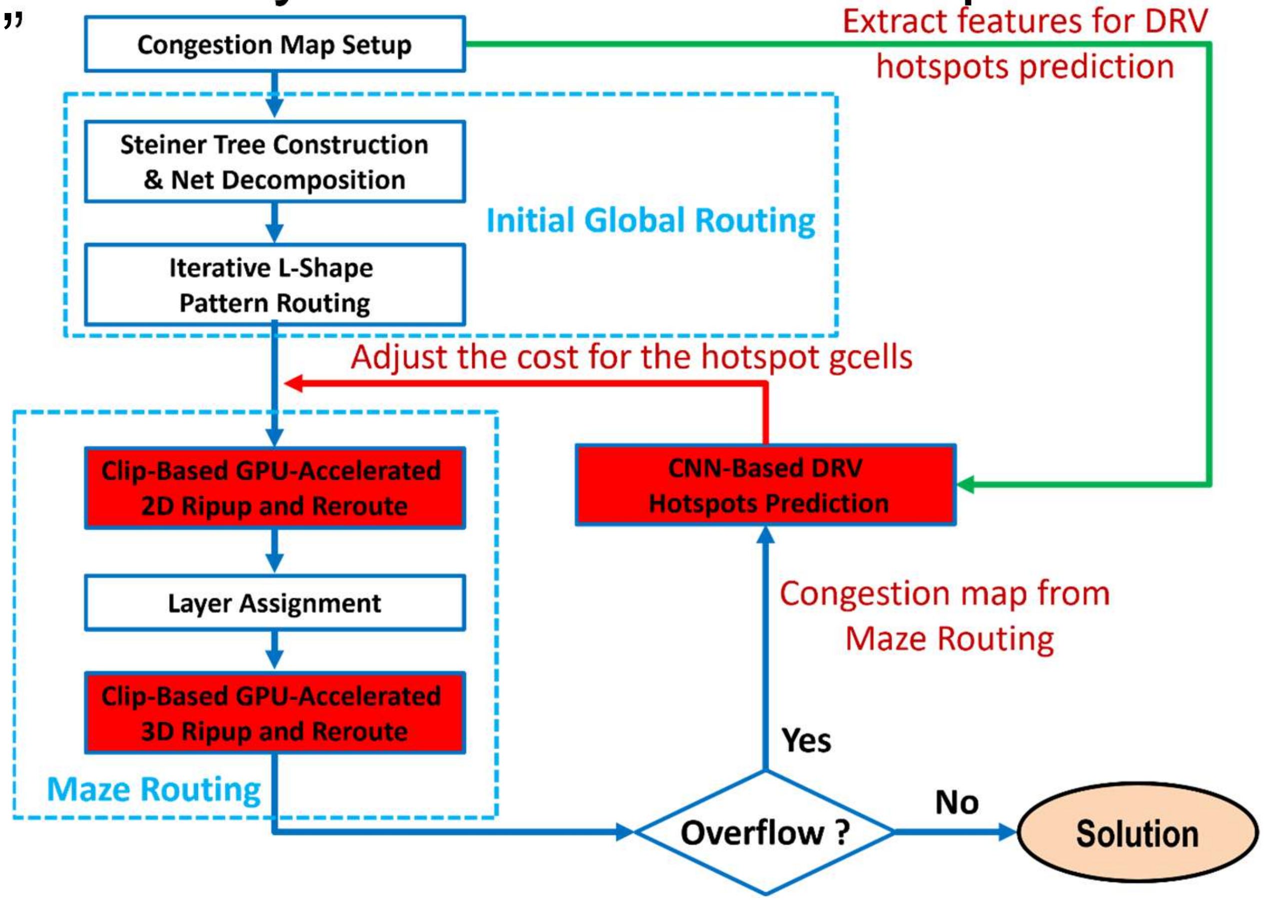
Three-Level Parallel Maze Routing

- We adopt three-level parallel maze routing
  - Level 1: maze routing for all the nets in a single clip ↔ Grid on GPU
  - Level 2: maze routing for a single net ↔ Block on GPU
    - Parallel A* search based on multiple priority queues [ZhouZ'15AAAI]
  - Level 3: expansion for a single priority queue ↔ Thread on GPU



Future Work

- "machine learning alongside optimization algorithms"
  - Combine the detailed-routability-driven GR with the ML-based DRV prediction models
  - Improve accuracy and robustness compared to "end to end learning"



BACKUP