ECE 260C, Spring 2025

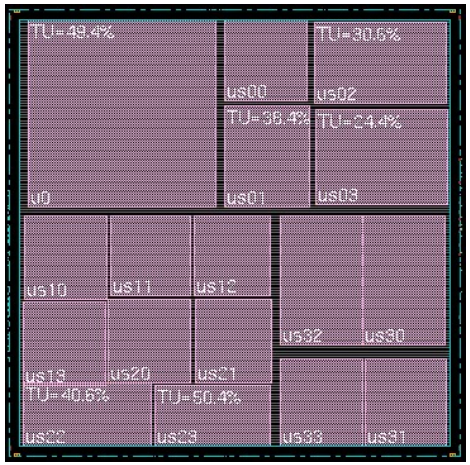
Floorplan and PDN

Andrew B. Kahng

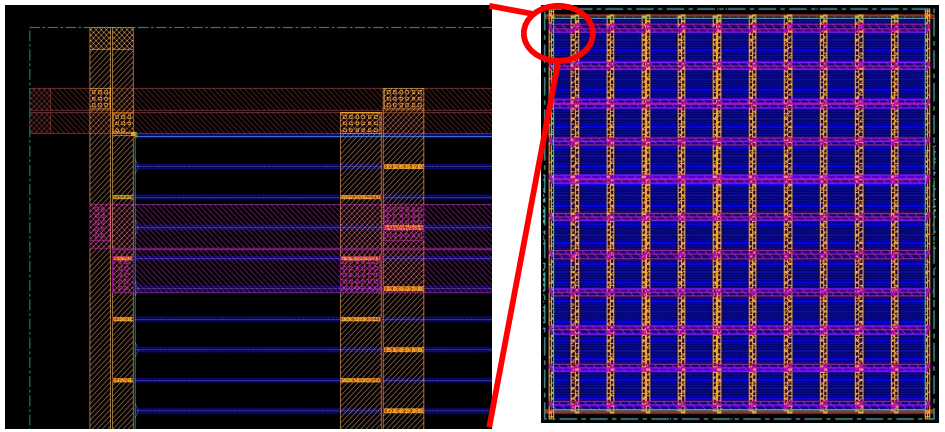
Thanks to: Zhiang Wang, Peter Gadfort

Physical Design Flow Pictures (old ECE 260B slide)

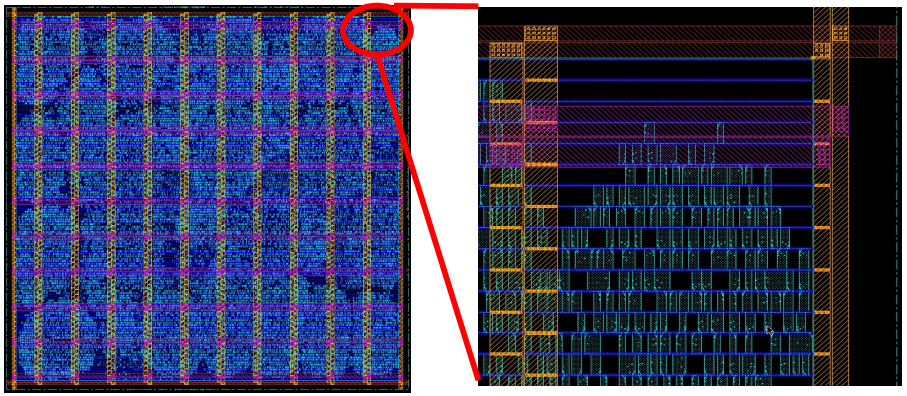
- Floorplanning



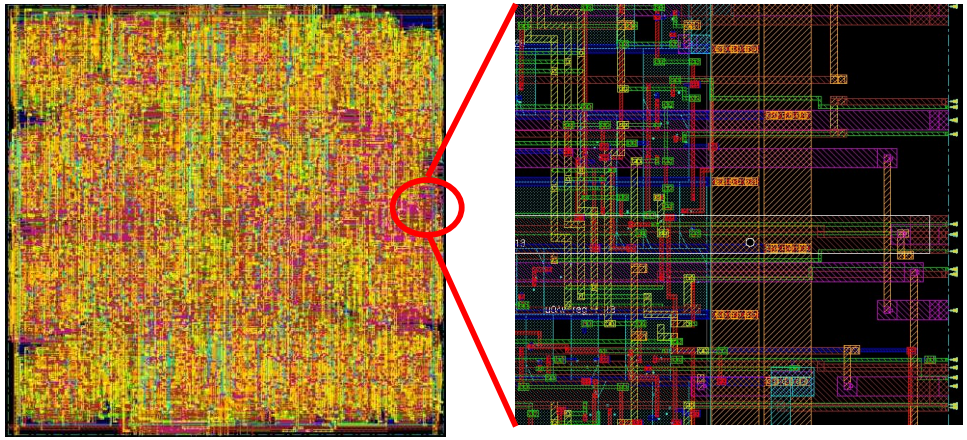
- Powerplanning



- Placement

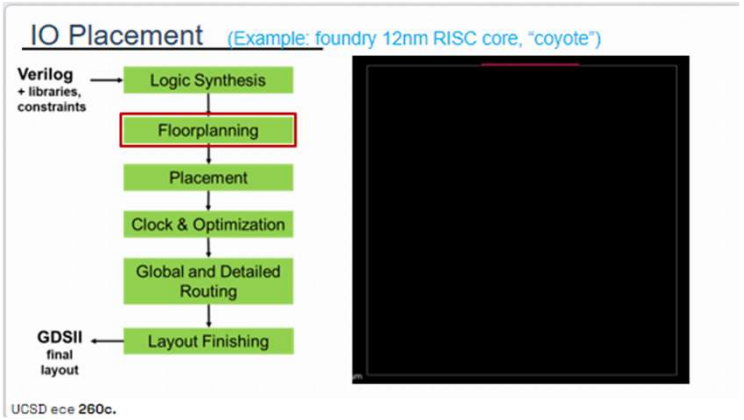


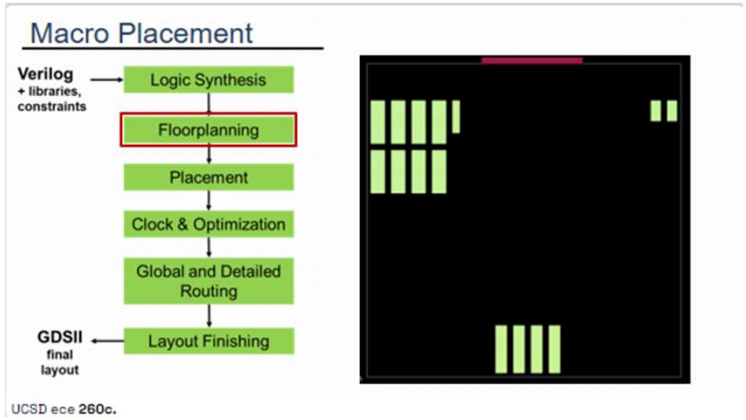
- Routing

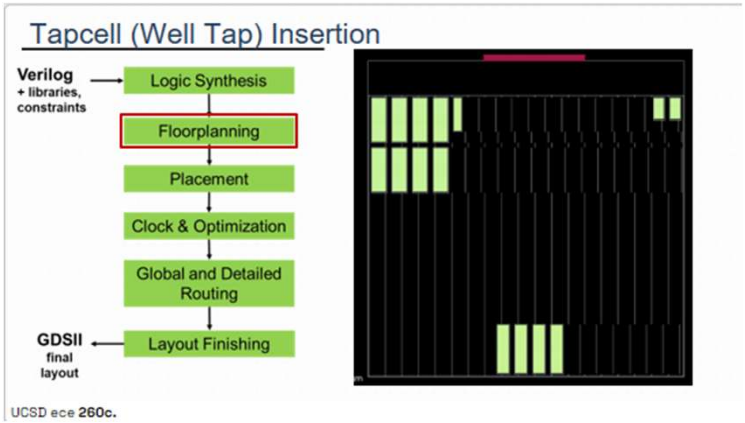


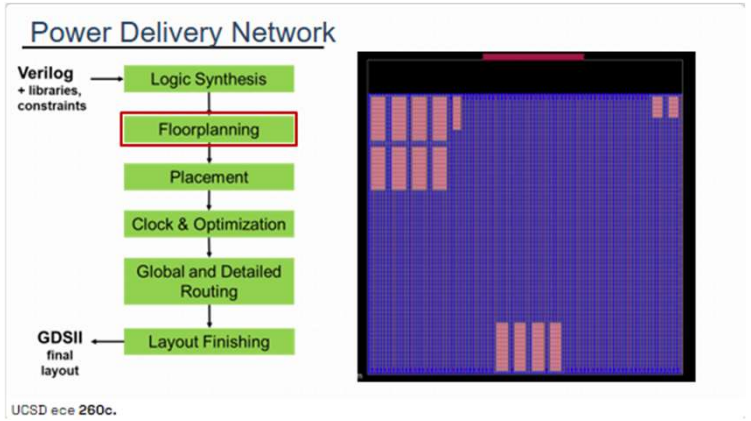
Floorplanning

Recall: "Floorplanning" Performs Many Tasks !

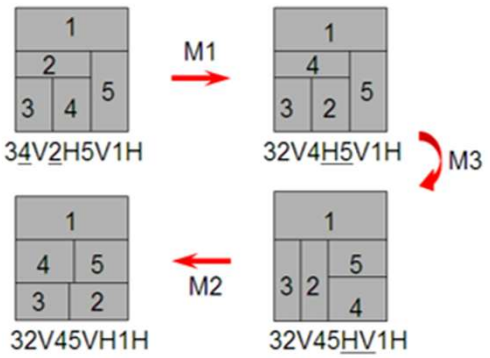








- https://openroad.readthedocs.io/en/latest/main/src/ifp/README.html ++
- IO placement = ppl
- Macro placement = mpl
  - Objectives? Constraints?
  - **Representation**? (Realization?) **Optimization**?
- Tapcell insertion = tap
- PDN generation = pdn



Conferences > 2020 18th IEEE International ...

Fast and Scalable I/O Pin Assignment with Divide-and-Conquer and Hungarian Matching

Publisher: IEEE Cite This PDF

Vitor Bandeira ; Mateus Fogaça ; Eder Matheus Monteiro ; Isadora Oliveira ; Mingyu Woo ; Ricardo Reis

All A

Pin Placement (ppo)

Fast and Scalable I/O Pin Assignment with Divide-and-Conquer and Hungarian Matching

Vitor Bandeira[1,3], Mateus Fogaça[1,3], Eder Matheus Monteiro[3], Isadora Oliveira[1,3], Mingyu Woo[4] and Ricardo Reis[1,2,3]
[1]PGMicro/[2]PPGC, [3]Instituto de Informática, Universidade Federal do Rio Grande do Sul
[4]ECE Department, UC San Diego, La Jolla, CA, USA
{vvbandeira, mpfogaca, emrmonteiro, isoliveira, reis}@inf.ufrgs.br, mwoo@eng.ucsd.edu

***Abstract*—I/O pin assignment is a crucial task in the floorplanning stage of IC implementation. However, this task has not received much attention in the physical design automation literature. Notably, floorplanning is a highly manual stage of the design flow. Nevertheless, it is known that the impact of I/O pin assignment in total routed wirelength (WL) is in the order of 5%. In advanced nodes, density, power and timing become very crucial and WL impacts of 5% are highly significant. We are therefore motivated to revisit the I/O pin assignment problem in this work. We present a fast and scalable Hungarian matching-based heuristic for I/O pin assignment. We present background scalability studies and a divide-and-conquer strategy that significantly reduces runtime without harm to the quality of results. Our algorithm converges in fewer iterations than previous works and presents superior performance according to criteria from the literature.**

pins are implemented using vertical/horizontal metal layers whose pitches are equal to $0.2\mu m$. Then, a core with a perimeter of $6mm$ would have 30,000 candidate locations. Many existing heuristics will face runtime/scalability issues with such high number of candidate locations. Thus, heuristic methods can benefit from criteria to filter candidate locations as well as the divide-and-conquer paradigm.

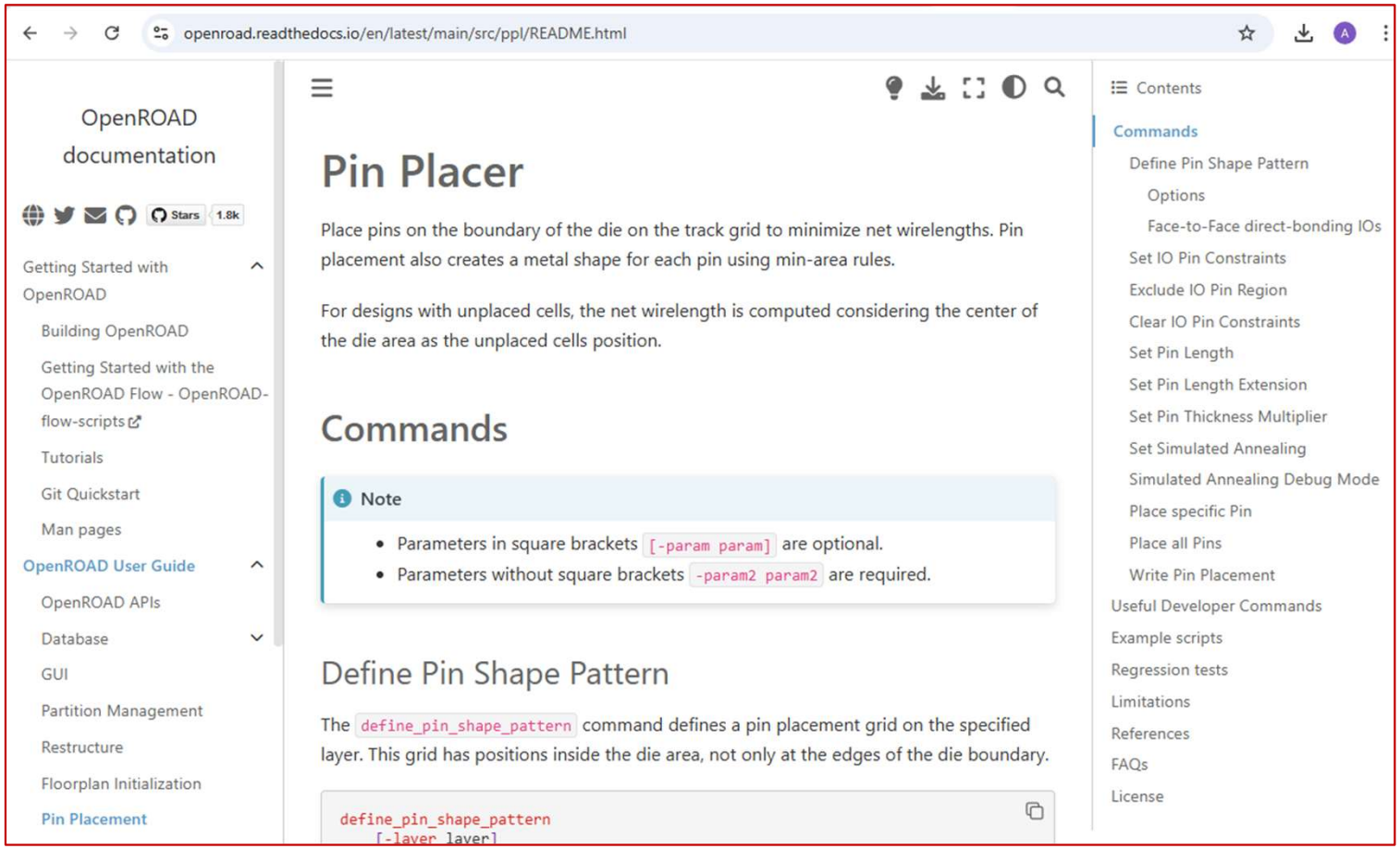
Our contributions are summarized as follows.

1) We devise a scalable Hungarian matching-based [6] approach to I/O pin assignment. Motivated by background scalability studies, we propose a divide-and-conquer strategy to reduce runtime. Our divide-and-conquer strategy can perform I/O pin assignment for a design with 2K I/O pins and 14K candidate positions in less than one second.
2) We integrate our I/O pin assigner tool with a global placement

https://ieeexplore.ieee.org/document/9159791

ppo = pin placement optimizer

- Where should pins be placed?
  - Chicken-egg: core determines periphery, and vice-versa!
    - Basic approach: “iterate until convergence …”
  - Sometimes, designer wants to provide guidance/constraints
    - Sometimes, the guidance/constraints are wrong (!) e.g., infeasible



Macro Placement: Hier-RTLMP (mpl)

Thanks to Dr. Zhiang Wang zhw033@ucsd.edu

Motivation

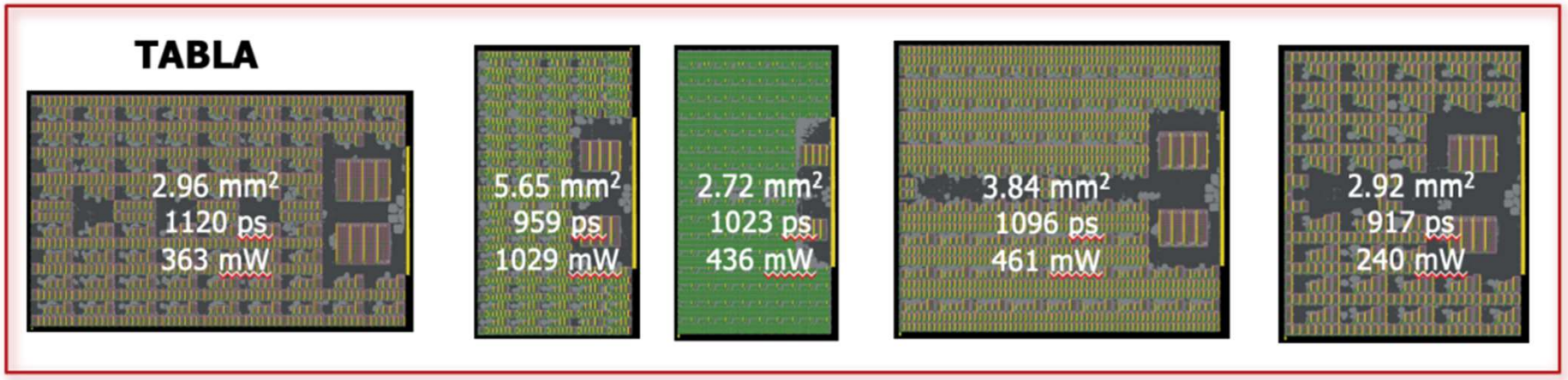
- Macro placement is a critical component of the “handoff” from frontend to the backend of a physical synthesis flow.

- Human experts usually need to consider multiple factors when they do macro placement:
  - Dataflow
  - Logical hierarchy
  - Connectivity between macros and input-output (IO) pins
  - …

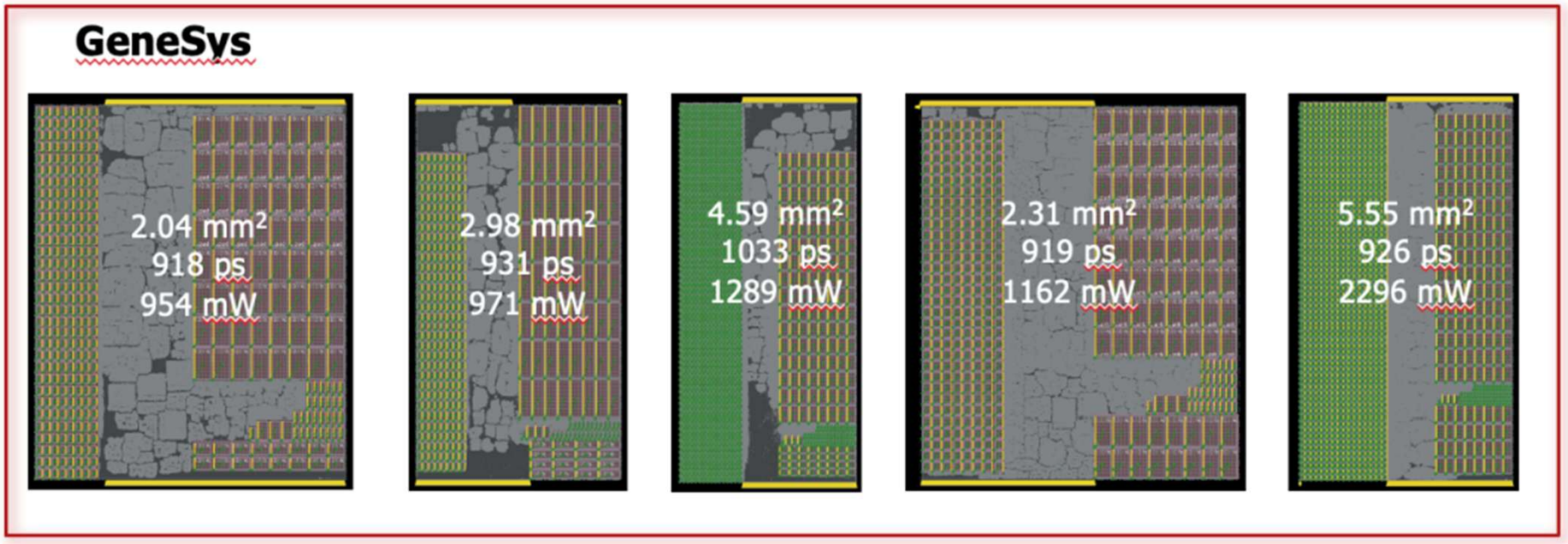
- ML hardware accelerators make the macro placement become more challengeable:
  - Hundreds of macros (examples will be given in the next slide)
  - Complex RTL structure with long, inscrutable autogenerated module names

Macro-Dominated ML Accelerators (paper: c397)

TABLA: machine learning accelerators for non-DNN algorithms



GeneSys: machine learning accelerators for DNN algorithms



Dataflow Analysis by Humans

- **Dataflow** refers to the movement of data between different functional units.

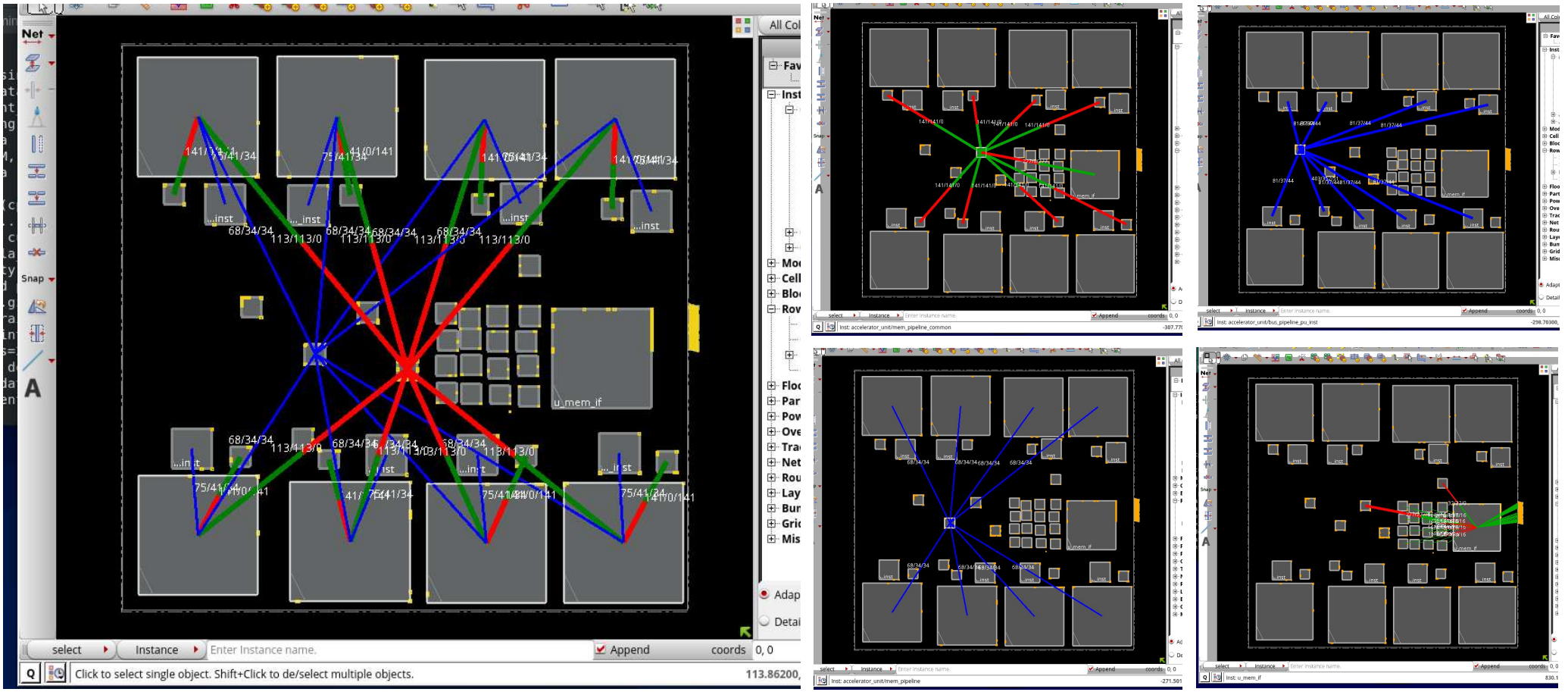
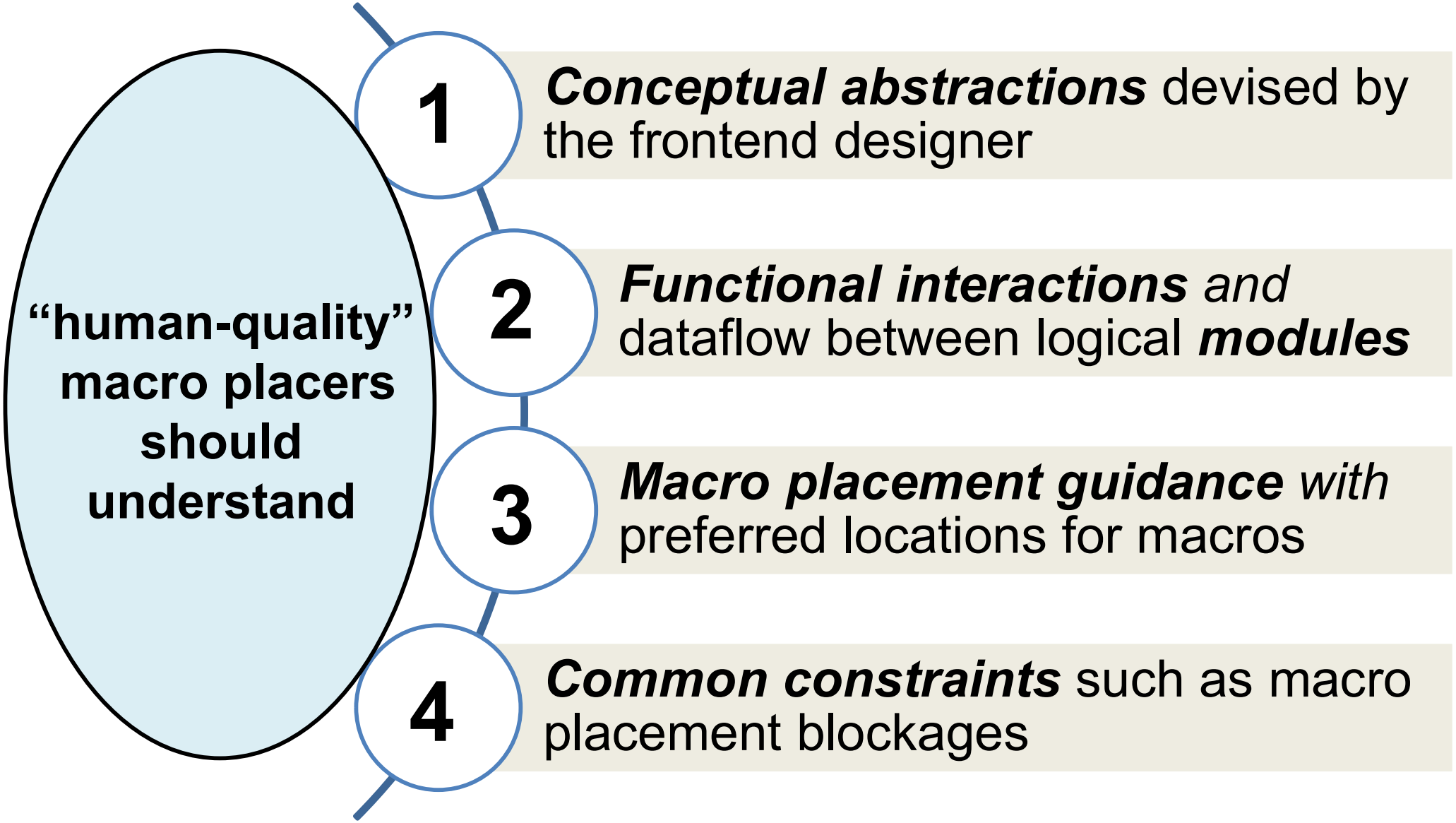


TABLA: systolic array structures
(Processing Units + Ctrl Logic + Mem Blocks)

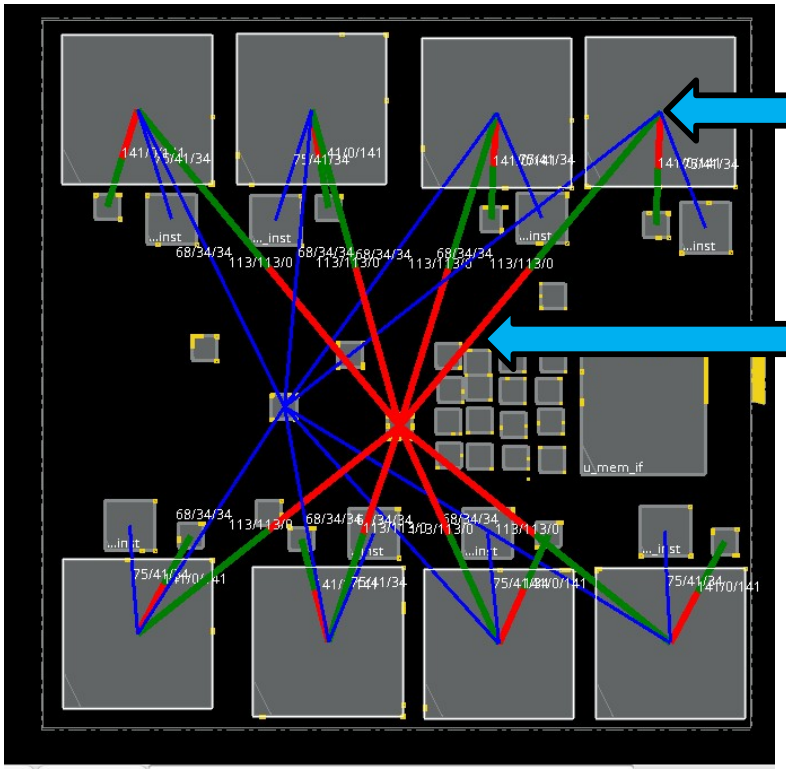
What Should a Human-quality Macro Placer Do ?



Macro Placer in OpenROAD: Hier-RTLMP

Hier-RTLMP

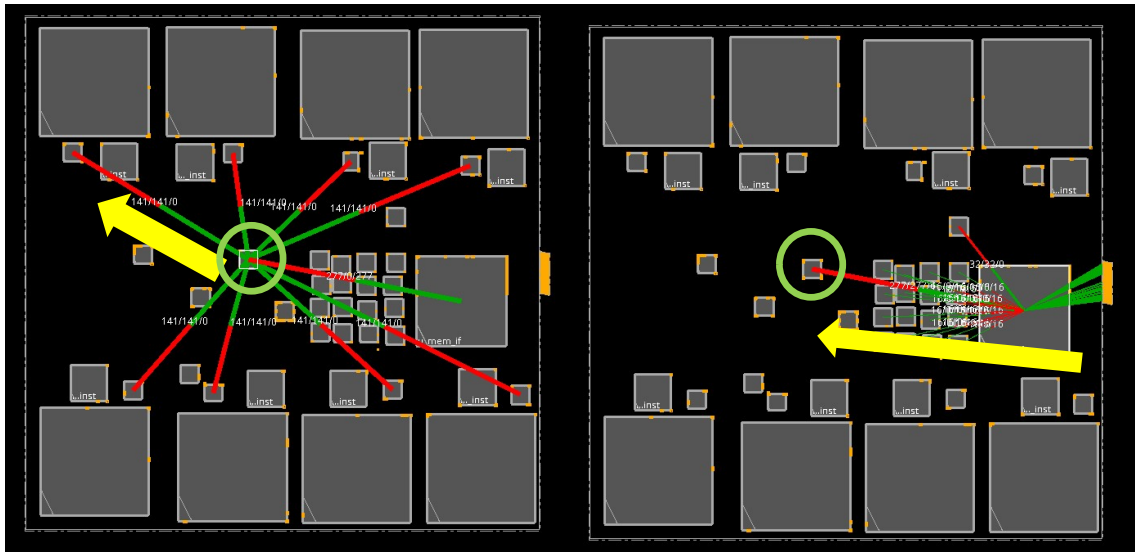
- *Conceptual abstractions devised by the frontend designer*
  - Through converting the structural netlist representation of the RTL design into a clustered netlist



Macro Placer in OpenROAD: Hier-RTLMP

Hier-RTLMP

- *Functional interactions*
  - Through analyzing the dataflow between logical modules (including Input-Output pins)

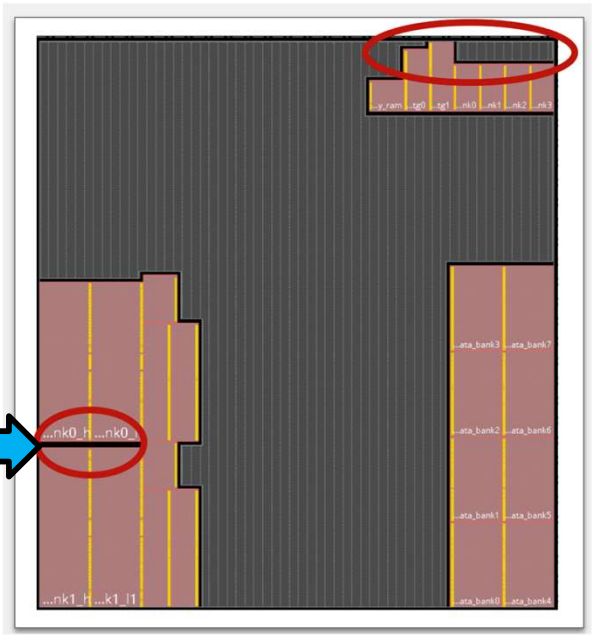


Macro Placer in OpenROAD: Hier-RTLMP

Hier-RTLMP

- *Macro placement guidance*
  - Through pushing macros to their preferred locations
  - Macro placement blockages
  - Small dead space (notch) avoidance
  - Pushing macros to peripheries

notch

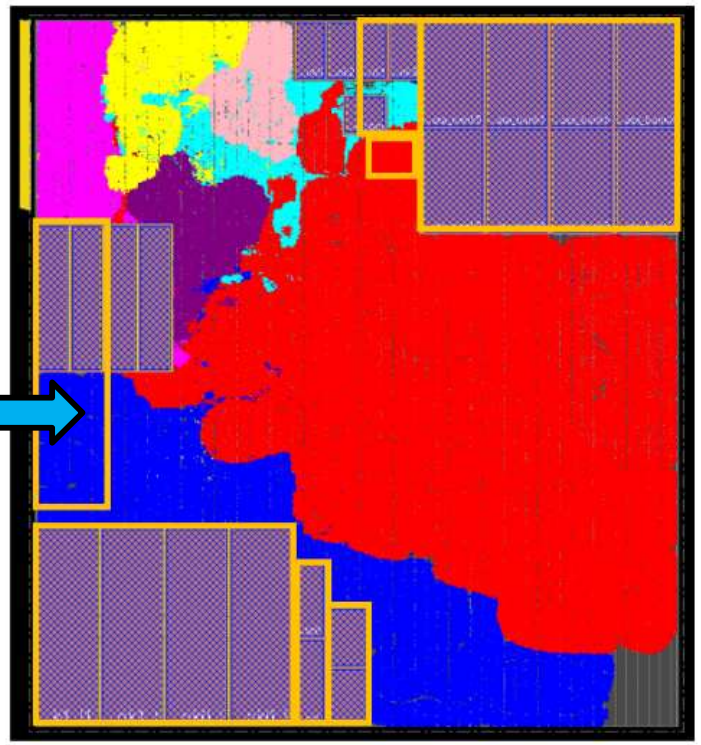


Macro Placer in OpenROAD: Hier-RTLMP

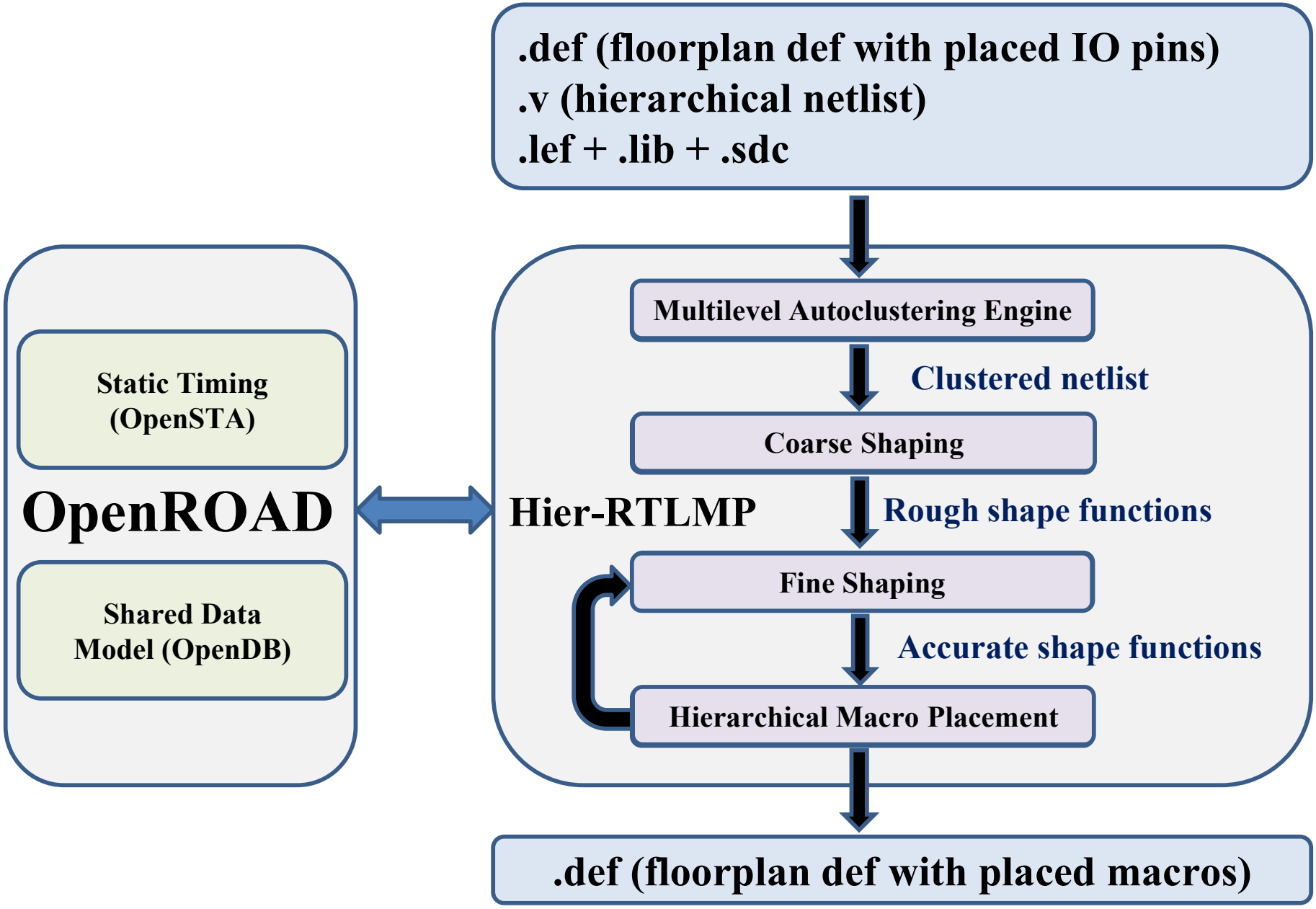
Hier-RTLMP

- *Common constraints*
  - Fixed macro constraints
  - Through pushing macros to their preferred locations

macro guidance



Hier-RTLMP flow (paper: link)



Autoclustering Engine

- Converts the structural netlist representation of the RTL design into a clustered netlist (clusterEngine.h)
  - Logical hierarchy (code)
  - Connection topology between clusters (code)
  - Macro regularity = grouping macros based on footprints ((code)
  - Separation between macros and corresponding standard cell logic (code)
  - Dataflow between logical modules (including Input-output pins) (code)

```
283   // Transform the logical hierarchy into a physical hierarchy.
284   void HierRTLMP::runMultilevelAutoclustering()
285   {
286     clustering_engine_ = std::make_unique<ClusteringEngine>(
287         block_, network_, logger_, tritonpart_);
288
289     // Set target structure
290     clustering_engine_->setTree(tree_.get());
291     clustering_engine_->run();
292
293     if (!tree_->has_unfixed_macros) {
294       skip_macro_placement_ = true;
295       return;
296     }
297
298     if (graphics_) {
299       graphics_->finishedClustering(tree_.get());
300     }
301   }
```

Line 284 in hier_rtlmp.cpp (link)

```
57   void ClusteringEngine::run()
58   {
59     design_metrics_ = computeModuleMetrics(block_->getTopModule());
60     init();
61
62     if (!tree_->has_unfixed_macros) {
63       return;
64     }
65
66     createRoot();
67     setBaseThresholds();
68
69     mapIOPinsAndPads();
70     createDataFlow();
71
72     createIOClusters();
73     classifyBoundariesStateForIOs();
74
75     if (design_metrics_->getNumStdCell() == 0) {
76       logger_->warn(MPL, 25, "Design has no standard cells!");
77       tree_->has_std_cells = false;
78       treatEachMacroAsSingleCluster();
79     } else {
80       multilevelAutocluster(tree_->root.get());
```

Line 57 in hier_rtlmp.cpp (link)

Coarse Shaping

Wait, what? → SA = how we search over solution representations;
Sequence Pair = how we represent (macro placement) solutions.

- Determine possible rough shapes (area and aspect ratio) for clusters
  - Simulated Annealing + Sequence Pair
  - Cost function: minimize the area

```
329  void HierRTLMP::runCoarseShaping()
330  {
331    setRootShapes();
332
333    if (tree_->has_only_macros) {
334      logger_->warn(MPL, 27, "Design has only macros!");
335      tree_->root->setClusterType(HardMacroCluster);
336      return;
337    }
338
339    if (graphics_) {
340      graphics_->startCoarse();
341    }
342
343    calculateChildrenTilings(tree_->root.get());
344
345    setPinAccessBlockages();
346    setPlacementBlockages();
347  }
348
```

Line 329 in hier_rtlmp.cpp (link)

```
469    // we vary the outline of parent cluster to generate different tilings
470    // we first vary the outline width while keeping outline height fixed
471    // Then we vary the outline height while keeping outline width fixed
472    // Vary the outline width
473    std::vector<float> vary_factor_list{1.0};
474    float vary_step = 1.0 / num_runs_;  // change the outline by based on num_runs
475    for (int i = 1; i < num_runs_; i++) {
476      vary_factor_list.push_back(1.0 - i * vary_step);
477    }
478    int remaining_runs = num_runs_;
479    int run_id = 0;
480
481    while (remaining_runs > 0) {
482      SoftSAVector sa_batch;
483      const int run_thread
484          = graphics_ ? 1 : std::min(remaining_runs, num_threads_);
485      for (int i = 0; i < run_thread; i++) {
486        const Rect new_outline(0,
487                               0,
488                               outline.getWidth() * vary_factor_list[run_id++],
489                               outline.getHeight());
490        if (graphics_) {
491          graphics_->setOutline(micronsToDbu(new_outline));
492        }
```

Simulated Annealing
Line 469 in hier_rtlmp.cpp (link)
(SACoreSoftMacro.h)

```
519          // multi threads
520          std::vector<std::thread> threads;
521          threads.reserve(sa_batch.size());
522          for (auto& sa : sa_batch) {
523            threads.emplace_back(runSA<SACoreSoftMacro>, sa.get());
524          }
525          for (auto& th : threads) {
526            th.join();
527          }
528        }
```

Multi-threading implementation
Line 519 in hier_rtlmp.cpp (link)

Fine Shaping + Hierarchical Macro Placement

- Fine Shaping: determine the accurate shape and aspect ratio for each cluster
  - Pick legal shapes (must fit into the outline of current cluster) for each child clusters

```
1734        const float target_util = target_util_list[run_id];
1735        const float target_dead_space = target_dead_space_list[run_id++];
1736        debugPrint(logger_,
1737                   MPL,
1738                   "fine_shaping",
1739                   1,
1740                   "Starting adjusting shapes for children of {}. target_util = "
1741                   "{}, target_dead_space = {}",
1742                   parent->getName(),
1743                   target_util,
1744                   target_dead_space);
1745        if (!runFineShaping(parent,
1746                            shaped_macros,
1747                            soft_macro_id_map,
1748                            target_util,
1749                            target_dead_space)) {
1750          debugPrint(logger_,
1751                     MPL,
1752                     "fine_shaping",
1753                     1,
1754                     "Cannot generate feasible shapes for children of {}, sa_id: "
1755                     "{}, target_util: {}, target_dead_space: {}",
1756                     parent->getName(),
1757                     run_id,
1758                     target_util,
1759                     target_dead_space);
1760          continue;
1761        }
```

Line 1731 in hier_rtlmp.cpp (link)

```
1987    // Determine the shape of each cluster based on target utilization
1988    // and target dead space.  In constrast to all previous works, we
1989    // use two parameters: target utilization, target_dead_space.
1990    // This is the trick part.  During our experiements, we found that keeping
1991    // the same utilization of standard-cell clusters and mixed cluster will make
1992    // SA very difficult to find a feasible solution.  With different utilization,
1993    // SA can more easily find the solution. In our method, the target_utilization
1994    // is used to determine the bloating ratio for mixed cluster, the
1995    // target_dead_space is used to determine the bloating ratio for standard-cell
1996    // cluster. The target utilization is based on tiling results we calculated
1997    // before. The tiling results (which only consider the contribution of hard
1998    // macros) will give us very close starting point.
1999    bool HierRTLMP::runFineShaping(Cluster* parent,
2000                                   std::vector<SoftMacro>& macros,
2001                                   std::map<std::string, int>& soft_macro_id_map,
2002                                   float target_util,
2003                                   float target_dead_space)
2004    {
2005      const float outline_width = parent->getWidth();
2006      const float outline_height = parent->getHeight();
2007      float pin_access_area = 0.0;
2008      float std_cell_cluster_area = 0.0;
2009      float std_cell_mixed_cluster_area = 0.0;
2010      float macro_cluster_area = 0.0;
2011      float macro_mixed_cluster_area = 0.0;
2012      // add the macro area for blockages, pin access and so on
2013      for (auto& macro : macros) {
2014        if (macro.getCluster() == nullptr) {
2015          pin_access_area += macro.getArea();  // get the physical-only area
2016        }
2017      }
```

Line 1999 in hier_rtlmp.cpp (link)

Fine Shaping + Hierarchical Macro Placement

- Hierarchical Macro Placement
  - Clusters are placement level by level in a pre-order DFS manner
  - Simulated Annealing + Sequence Pair
  - Cost function: minimize area, wirelength, penalty functions for different constraints (code)

```
303  void HierRTLMP::runHierarchicalMacroPlacement()
304  {
305    if (graphics_) {
306      graphics_->startFine();
307    }
308
309    adjustMacroBlockageWeight();
310    placeChildren(tree_->root.get());
311  }
```

```
1101 void HierRTLMP::placeChildren(Cluster* parent)
1102 {
1103   if (parent->getClusterType() == HardMacroCluster) {
1104     placeMacros(parent);
1105     return;
1106   }
1107
1108   if (parent->isLeaf()) {  // Cover IO Clusters && Leaf Std Cells
1109     return;
1110   }
1111
1112   debugPrint(logger_,
1113              MPL,
1114              "hierarchical_macro_placement",
1115              1,
1116              "Placing children of cluster {}",
1117              parent->getName());
1118
1119   if (graphics_) {
1120     graphics_->setCurrentCluster(parent);
1121   }
1122
```

Line 303 in hier_rtlmp.cpp (link)

```
1379      // Note that all the probabilities are normalized to the summation of 1.0.
1380      // Note that the weight are not necessaries summarized to 1.0, i.e., not
1381      // normalized.
1382      std::unique_ptr<SACoreSoftMacro> sa
1383          = std::make_unique<SACoreSoftMacro>(tree_.get(),
1384                                              outline,
1385                                              shaped_macros,
1386                                              placement_core_weights_,
1387                                              boundary_weight_,
1388                                              macro_blockage_weight_,
1389                                              notch_weight_,
1390                                              notch_h_th_,
1391                                              notch_v_th_,
1392                                              pos_swap_prob_ / action_sum,
1393                                              neg_swap_prob_ / action_sum,
1394                                              double_swap_prob_ / action_sum,
1395                                              exchange_swap_prob_ / action_sum,
1396                                              resize_prob_ / action_sum,
1397                                              init_prob_,
1398                                              max_num_step_,
1399                                              num_perturb_per_step,
1400                                              random_seed_,
1401                                              graphics_.get(),
1402                                              logger_);
1403      sa->setNumberOfMacrosToPlace(num_of_macros_to_place);
1404      sa->setCentralizationAttemptOn(true);
1405      sa->setFences(fences);
1406      sa->setGuides(guides);
1407      sa->setNets(nets);
1408      sa->addBlockages(placement_blockages);
1409      sa->addBlockages(macro_blockages);
1410      sa_batch.push_back(std::move(sa));
1411    }
```

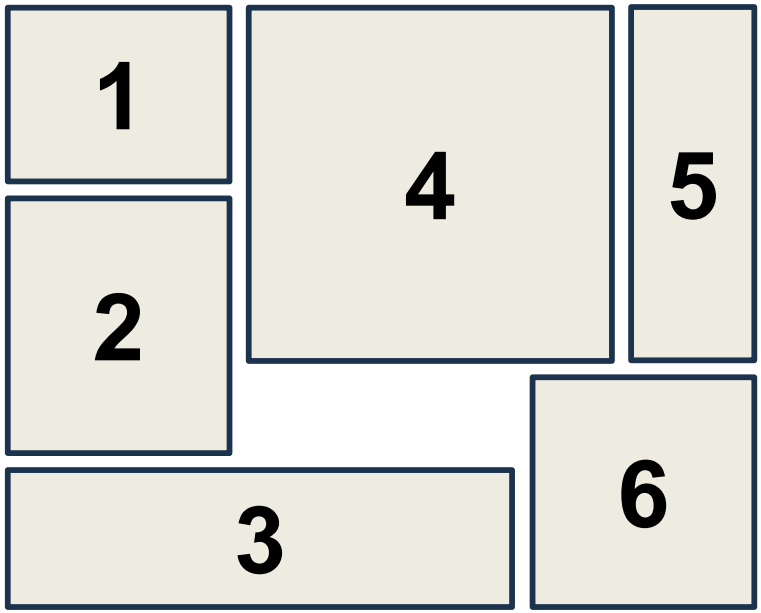
Line 1379 in hier_rtlmp.cpp (link)

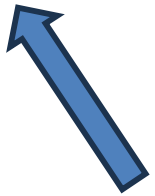
Sequence Pair and Simulated Annealing

Sequence Pair: How we represent the floorplan solution
Simulated Annealing: How we (heuristically) optimize the floorplan

Sequence Pair (SP) (paper: link)

- SP consists of an ordered pair (Γ^+, Γ^-) of cluster sequences.
- Given a sequence pair (Γ^+, Γ^-), the location of clusters can be determined as follows:
  - (**Horizontal constraint**): cluster i is left to cluster j if i appears before j in both Γ^+ and $\Gamma^- \rightarrow (\ldots\ i\ \ldots\ j \ldots, \ldots\ j \ldots\ i\ \ldots)$
  - (**Vertical constraint**): cluster i is below cluster j if i appears after j in Γ^+ and i appears before j $\Gamma^- \rightarrow (\ldots\ j\ \ldots\ i \ldots, \ldots\ j \ldots\ i\ \ldots)$



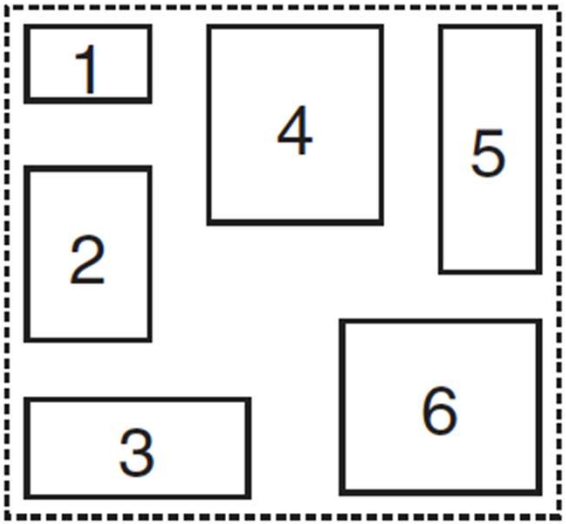


Longest Common Subsequence !
(paper: link)

$(\Gamma^+, \Gamma^-) = (124536, 326145)$

From Floorplan to Sequence Pair

- A sequence pair consists of two module name sequences. For example, the floorplan shown at right can be represented as the sequence pair $(\Gamma_+, \Gamma_-) = (124536, 326145)$.



- Construction of Γ_+
  - For each module, draw a **right-up locus and a left-down locus** as shown in Figure (a). Then we order these loci from the left to right. Γ_+ is resulting order of module names.
- Construction of Γ_-
  - For each module, we draw a **up-left locus and a down-right locus** as shown in figure (b). Then we order these loci from the left to right. Γ_- is the resulting order of module names.

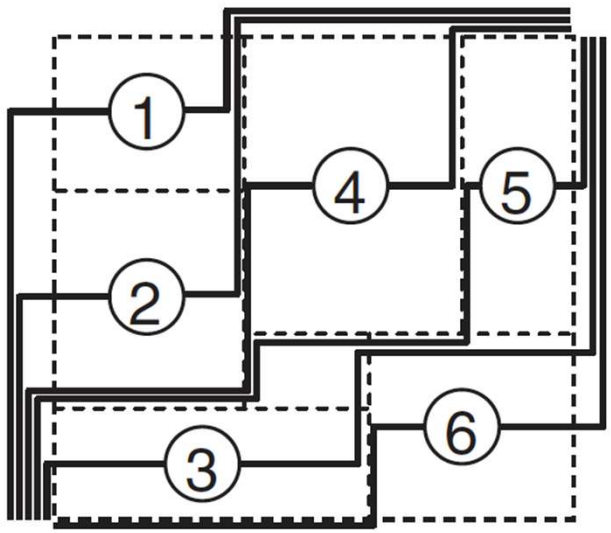


Figure (a)

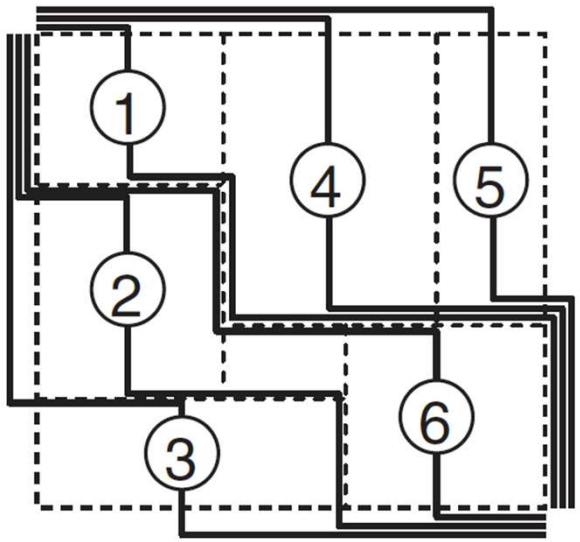
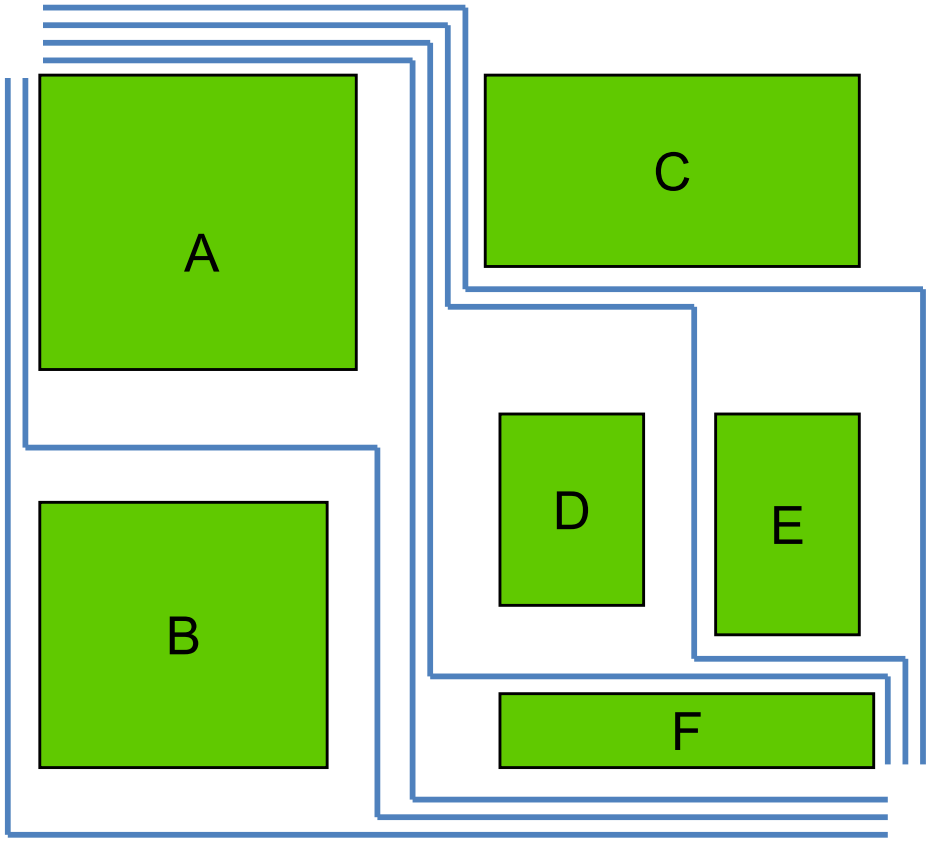


Figure (b)

Think: "non-crossing staircases"

Another example of (up-left, down-right) ordering Γ_-

Γ_- = CEDFAB (Exercise: What is Γ_+ ?)



From Sequence Pair to Floorplan

- Given an SP (Γ_+, Γ_-), the geometric relation between modules (clusters) is derived from the SP:
  - **Horizontal constraint**: module i is to the left of module j if i appears **before** j in **both** Γ_+ and Γ_- (... i ... j ..., ... i ... j ...)
  - **Vertical constraint**: module i is below module j if i appears **after** j in Γ_+ and i appears **before** j in Γ_-. (..... j ... i ..., ... i j...)

- From horizontal constraints: Create a **horizontal constraint graph** with a source and a sink, and a node-weighted directed acyclic graph G_H (V,E), where V is the set of nodes, and E is the set of edges as follows:
  - V source s, sink t, and n nodes labeled with module names
  - E: (s, t) and (i, t) for each module i, and (i, j) if and only if module i is on the left of module j (horizontal constraint).
  - Node weight : zero for s and t, width of module i for node i.

- From vertical constraints: create a vertical constraint graph G_V (V,E).

From Sequence Pair to Floorplan

- For sequence pair (Γ_+, Γ_-) = (124536, 326145), we construct the HCG (a) and the VCG (b)
- A "longest common subsequences" algorithm (see AdyaM03) is used in calculating this final floorplan (c)

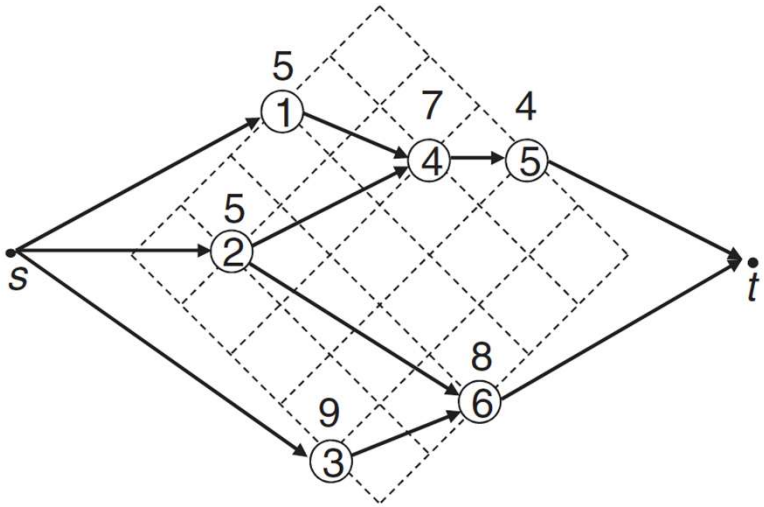


Figure (a)

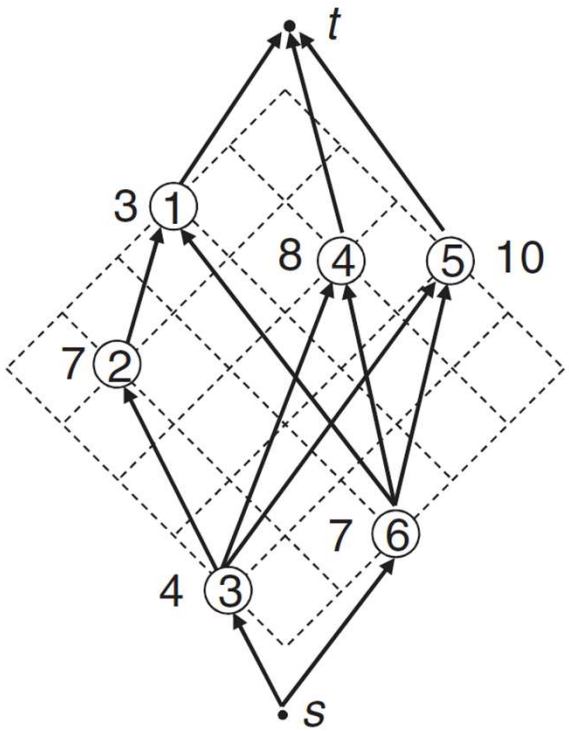


Figure (b)

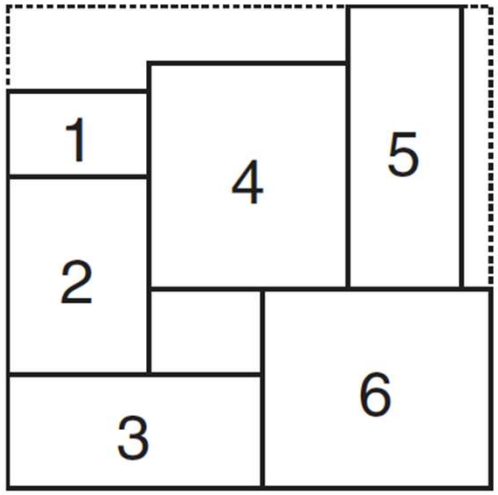


Figure (c)

Concepts from CSE 21 or CSE 101 – discrete math, algorithms

Sequence Pair (SP) (code)

```cpp
// Determine the positions of macros based on sequence pair
template <class T>
void SimulatedAnnealingCore<T>::packFloorplan()
{
  for (auto& macro_id : pos_seq_) {
    macros_[macro_id].setX(0.0);
    macros_[macro_id].setY(0.0);
  }

  // Each index corresponds to a macro id whose pair is:
  // <Position in Positive Sequence , Position in Negative Sequence>
  std::vector<std::pair<int, int>> sequence_pair_pos(pos_seq_.size());

  // calculate X position
  for (int i = 0; i < pos_seq_.size(); i++) {
    sequence_pair_pos[pos_seq_[i]].first = i;
    sequence_pair_pos[neg_seq_[i]].second = i;
  }

  std::vector<float> accumulated_length(pos_seq_.size(), 0.0);
  for (int i = 0; i < pos_seq_.size(); i++) {
    const int macro_id = pos_seq_[i];
    const int neg_seq_pos = sequence_pair_pos[macro_id].second;

    macros_[macro_id].setX(accumulated_length[neg_seq_pos]);

    const float current_length
        = macros_[macro_id].getX() + macros_[macro_id].getWidth();

    for (int j = neg_seq_pos; j < neg_seq_.size(); j++) {
      if (current_length > accumulated_length[j]) {
        accumulated_length[j] = current_length;
      } else {
        break;
      }
    }
  }

  width_ = accumulated_length[pos_seq_.size() - 1];
```

```cpp
  // calulate Y position
  std::vector<int> reversed_pos_seq(pos_seq_.size());
  for (int i = 0; i < reversed_pos_seq.size(); i++) {
    reversed_pos_seq[i] = pos_seq_[reversed_pos_seq.size() - 1 - i];
  }

  for (int i = 0; i < pos_seq_.size(); i++) {
    sequence_pair_pos[reversed_pos_seq[i]].first = i;
    sequence_pair_pos[neg_seq_[i]].second = i;

    // This is actually the accumulated height, but we use the same vector
    // to avoid more allocation.
    accumulated_length[i] = 0.0;
  }

  for (int i = 0; i < pos_seq_.size(); i++) {
    const int macro_id = reversed_pos_seq[i];
    const int neg_seq_pos = sequence_pair_pos[macro_id].second;

    macros_[macro_id].setY(accumulated_length[neg_seq_pos]);

    const float current_height
        = macros_[macro_id].getY() + macros_[macro_id].getHeight();

    for (int j = neg_seq_pos; j < neg_seq_.size(); j++) {
      if (current_height > accumulated_length[j]) {
        accumulated_length[j] = current_height;
      } else {
        break;
      }
    }
  }

  height_ = accumulated_length[pos_seq_.size() - 1];

  if (graphics_) {
    graphics_->saStep(macros_);
  }
}
```

Line 479 in SimulatedAnnealingCore.cpp (link)

Constraints in Hierarchical Macro Placement

- ***Fixed outline***: All clusters should be placed within the fixed outline specified by users. (code)
- ***Macro peripheral bias:*** All macros should be pushed to peripheries as much as possible. (code)
- ***Macro blockage:*** All macros should not overlap with macro placement blockages. (code)
- ***Pin access***: All macros should be kept from blocking access of input-output pins. (code)
- ***Macro guidance:*** All macros should be placed near specified regions if users provide such constraints. (code)
- ***Notch avoidance:*** A decent floorplan should avoid "dead space" which cannot be used effectively by P&R tools. (code)

Simulated Annealing (SA)

- **Kirkpatrick, Gelatt, Vecchi, *Science (1983):*** *One of the most cited scientific papers ever*
- SA is one of many "metaheuristics" that are used to deal with instances of intractable (NP-hard) combinatorial problems
  - Genetic algorithms (Holland, U. Michigan)
  - Tabu search (Glover, U. Colorado)
  - …
- Combinatorial optimization has a physical analogy to the annealing (slow cooling) of metals to produce a perfectly-ordered, minimum-energy state: a "state" is a "solution", "energy" is "cost", etc.

- ***Basic idea***
  - ***Initialize*** – Start with a random initial solution. Initialize high "temperature".
  - ***Step 2:*** *"Move Gen"* – **Perturb** current solution to obtain a **'neighbor' solution**
  - ***Step 3:*** *Calculate cost change* – calculate the change in solution cost due to the move (minimization: negative change is better, positive change is worse)
  - ***Step 4:*** *Accept/Reject* – Depending on the cost change, accept or reject the move. Probability of acceptance depends on current "temperature".
  - ***Step 5:*** *Update* – Update temperature, current solution. **Go to *Step 2*.**
  - **Continue until termination** condition ('freezing' or 'quenching') is satisfied

Neighborhood Structure

- **Topology (i.e., "graph") over the space of solutions**
- **Adjacency in this graph is induced by the definition of "perturbation" (aka "move")**
- **Traveling Salesperson Problem:** switch positions of two cities in the tour
- **Graph Bisection Problem:** swap a pair of vertices between the partitions
- ***TSP:*** *"optimally rearrange the positions of four cities in the tour" → more neighbors, more powerful move, more work to generate a move*

SA Pseudocode

http://www.ecs.umass.edu/ece/labs/vlsicad/ece665/slides/SimulatedAnnealing.ppt

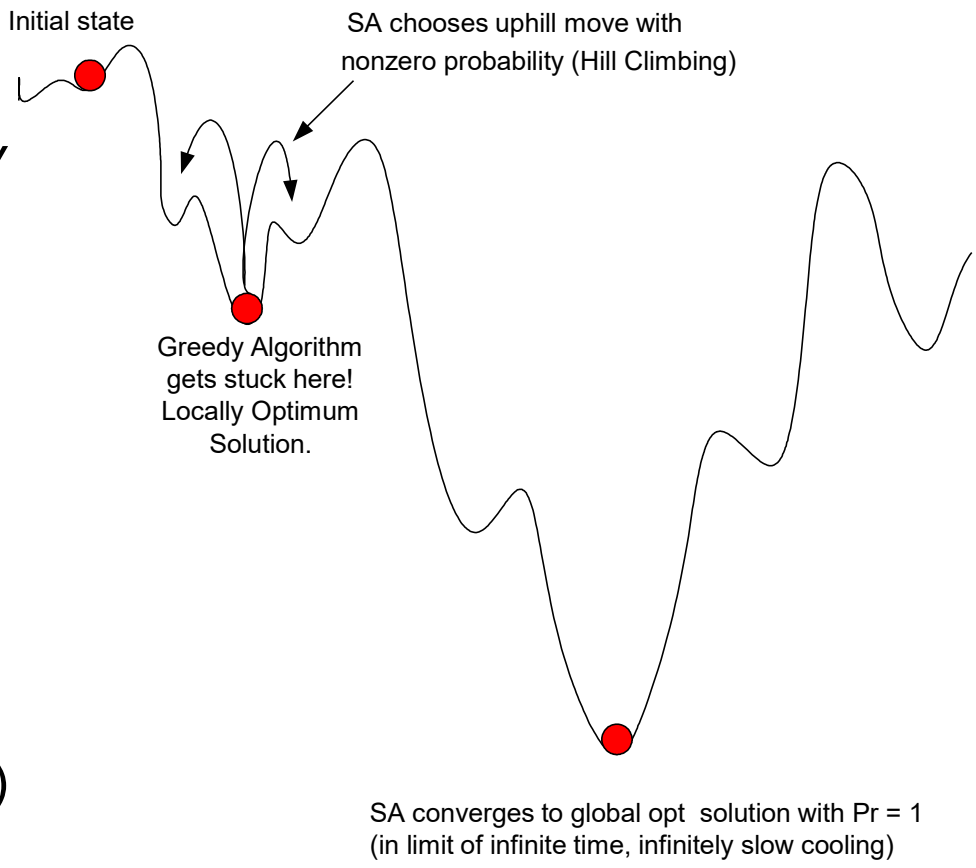
Algorithm SIMULATED-ANNEALING
Begin
    temp = INIT-TEMP;
    currentSol = INIT-SOLUTION;
    for i = 1 to M
        candidateSol = NEIGHBOR(*currentSol*);
            ΔC = COST(*candidateSol*) – COST(*currentSol*);
            if (ΔC < 0) **then**
                currentSol = *candidateSol*;
            else with Pr = $e^{-(\Delta C/temp)}$)
                currentSol = *candidateSol;*
        temp = SCHEDULE(*temp*);
End

What happens when *temp* = +∞ ?
What happens when *temp* = 0 ?

Simulated Annealing Facts

Question: Why is SA so widely applied?
Answer: (1) Easy to implement. (2) Can "handle" almost any discrete optimization (search in a solution space) task. [But: (3) "discrete", "almost any" are two-edged swords. And (4) SA needs help to scale …]

- NEIGHBOR(solution) defines a **topology** over all solutions in the solution space
- At a **fixed** value of ***temp***, SA behavior corresponds to a *homogeneous Markov chain*
  - Matrix of transition probabilities between states
- Steady-state (= equilibrium) probability of the Markov chain being in state A is proportional to $e^{(-\text{cost}(A)/temp)}$
  - When temp → 0, exponentially more likely to be in the global optimum state
  - "SA is optimal" (in the limit of 'infinite time')
  - Of course, we spend only a finite amount of time (#moves) at any temperature value
  - ***Is cooling the best strategy with finite time?*** See Boese/Kahng, 1993 [link] [link]



Neighborhood Structure of Sequence Pair

- Solution space
  - For n modules, the lengths of Γ_+ and Γ_- are both n, and thus each of Γ_+ and Γ_- have n! permutations. There are $(n!)^2$ permutations for a sequence pair with n modules.

- Neighborhood structure
  - M1 Rotate a module
  - M2 Swap two modules in only one sequence
  - M3: Swap two modules in both sequences

Simulated Annealing (SimulatedAnnealingCore.h)

```
759   void SimulatedAnnealingCore<T>::fastSA()
760   {
761     float cost = calNormCost();
762     float pre_cost = cost;
763     float delta_cost = 0.0;
764     int step = 1;
765     float temperature = init_temperature_;
766     const float min_t = 1e-10;
767     const float t_factor
768         = std::exp(std::log(min_t / init_temperature_) / max_num_step_);
769
770     // Used to ensure notch penalty is used only in the latter steps
771     // as it is too expensive
772     notch_weight_ = 0.0;
773
774     int num_restart = 1;
775     const int max_num_restart = 2;
776
777     if (isValid()) {
778       updateBestValidResult();
779     }
780
781     while (step <= max_num_step_) {
782       for (int i = 0; i < num_perturb_per_step_; i++) {
783         perturb();
784         cost = calNormCost();
785
786         const bool keep_result
787             = cost < pre_cost
788               || best_valid_result_.sequence_pair.pos_sequence.empty();
789         if (isValid() && keep_result) {
790           updateBestValidResult();
791         }
```

Line 759 in SimulatedAnnealingCore.cpp (link)

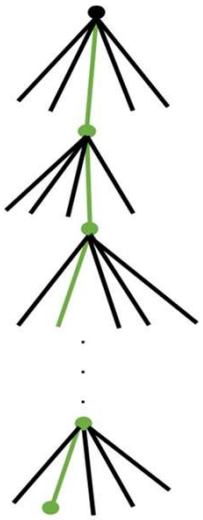
```
793         delta_cost = cost - pre_cost;
794         const float num = distribution_(generator_);
795         const float prob
796             = (delta_cost > 0.0) ? std::exp((-1) * delta_cost / temperature) : 1;
797         if (num < prob) {
798           pre_cost = cost;
799         } else {
800           restore();
801         }
802       }
803
804       temperature *= t_factor;
805       step++;
806
807       cost_list_.push_back(pre_cost);
808       T_list_.push_back(temperature);
809
810       if ((num_restart <= max_num_restart)
811           && (step == std::floor(max_num_step_ / max_num_restart)
812               && (outline_penalty_ > 0.0))) {
813         shrink();
814         packFloorplan();
815         calPenalty();
816         pre_cost = calNormCost();
817         num_restart++;
818         step = 1;
819         num_perturb_per_step_ *= 2;
820         temperature = init_temperature_;
821       }
822
823       if (step == max_num_step_ - macros_.size() * 2) {
824         notch_weight_ = original_notch_weight_;
825         packFloorplan();
826         calPenalty();
827         pre_cost = calNormCost();
828       }
829     }
830
831     packFloorplan();
832     if (graphics_) {
833       graphics_->doNotSkip();
834     }
835     calPenalty();
836
837     if (!isValid() && !best_valid_result_.sequence_pair.pos_sequence.empty()) {
838       useBestValidResult();
839     }
840   }
```

Accept or Reject ?

Temperature update

Go-With-The-Winners

- D. Aldous and U. Vazirani, "Go With the Winners Algorithms", Proc. FOCS, 1994, pp. 492-501. [link]
  - Launch multiple optimization threads
  - Periodically identify the most promising thread(s)
  - Clone the promising thread(s) and terminate others
  - *(continue with the optimization) (cf. "evolutionary", "PPSN")*
- Implementation: here
  - Initialize all SA workers in parallel
  - Main Loop:
    - Run each SA worker for *sync_iter iterations*, in parallel
    - Stop if *Iter* iterations have been performed; otherwise, select top *k* workers and replicate their solutions to remaining workers
    - Write out best macro placement solution of each worker



```
iter_count ← 0;
sync_iter ← Iters × sync_freq;
while true do
    end_iter ← min(Iters, iter_count + sync_iter);
    for i ← 0 to W − 1 in parallel do
        Each worker performs (end_iter − iter_count) SA
          iterations; applying N × #macro moves per
          iteration and updating temperature;
    iter_count ← end_iter;
    if iter_count = Iters then
        break;
    candidate_solutions ← extractTopK(workers, k);
    Evenly distribute these top-k solutions across all the
      workers;
Write out the best solution of each worker.
```

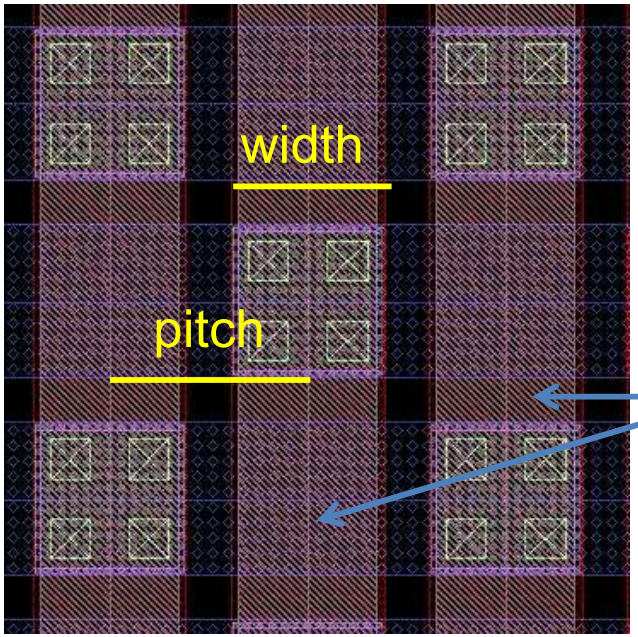
Power Grid Generation

Thanks to Dr. Peter Gadfort (LinkedIn)

Director of Silicon Engineering, Zero ASIC

Power Grid Design and Analysis

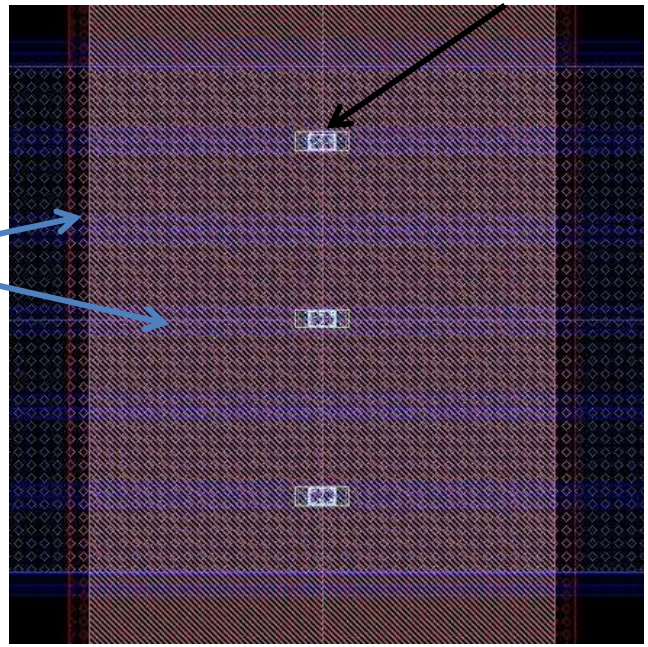
- Typically, power grids are constructed on higher metal layers due to their lower sheet resistance and higher current density specification. Often top metal layers are thick.
- Unless upper metal layers are used for signal routing, the usual approach is to maximize the power grid, so long as the maximum metal density rules are satisfied.
- We need to determine the following parameters.
  - Width and pitch of the power mesh stripes (depends on peak current)
  - How frequently to tap down from primary mesh to M1 rails
  - Via stack (e.g. – 1x4, 2x2, or 2x3), should be chosen carefully to avoid blocking extra routing tracks and give low resistance
- The two primary concerns are IR drop and electromigration.



Power Rails

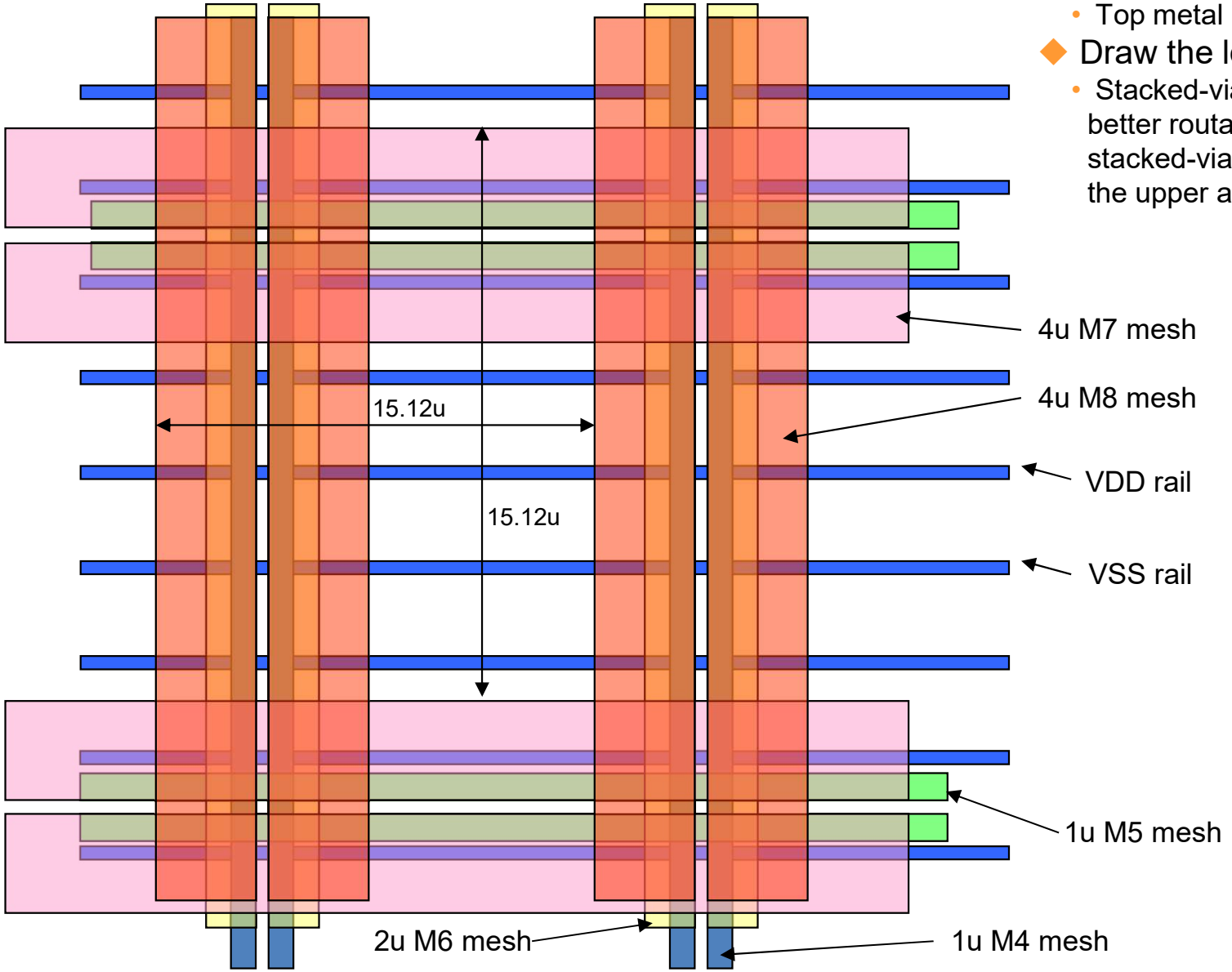
Power Stripes

Via Stack (2x3)



Power Grid Design and Analysis

- Structure of the multi-layer power mesh (You can see detail process using slide-show)

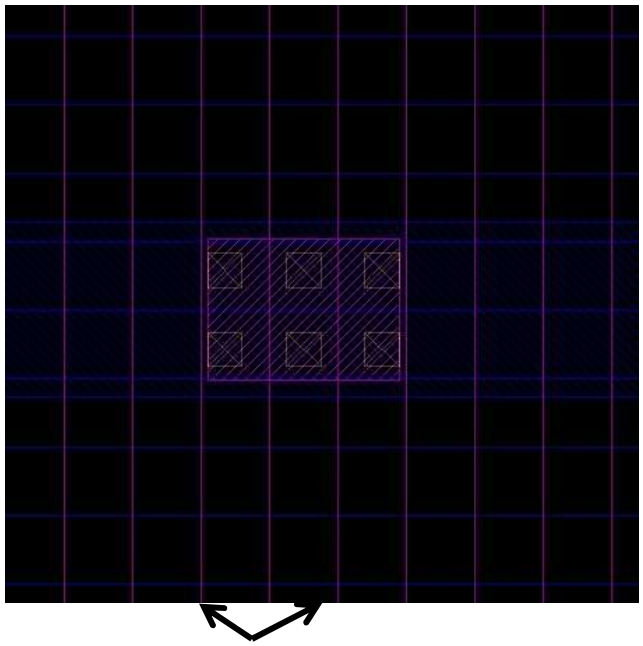


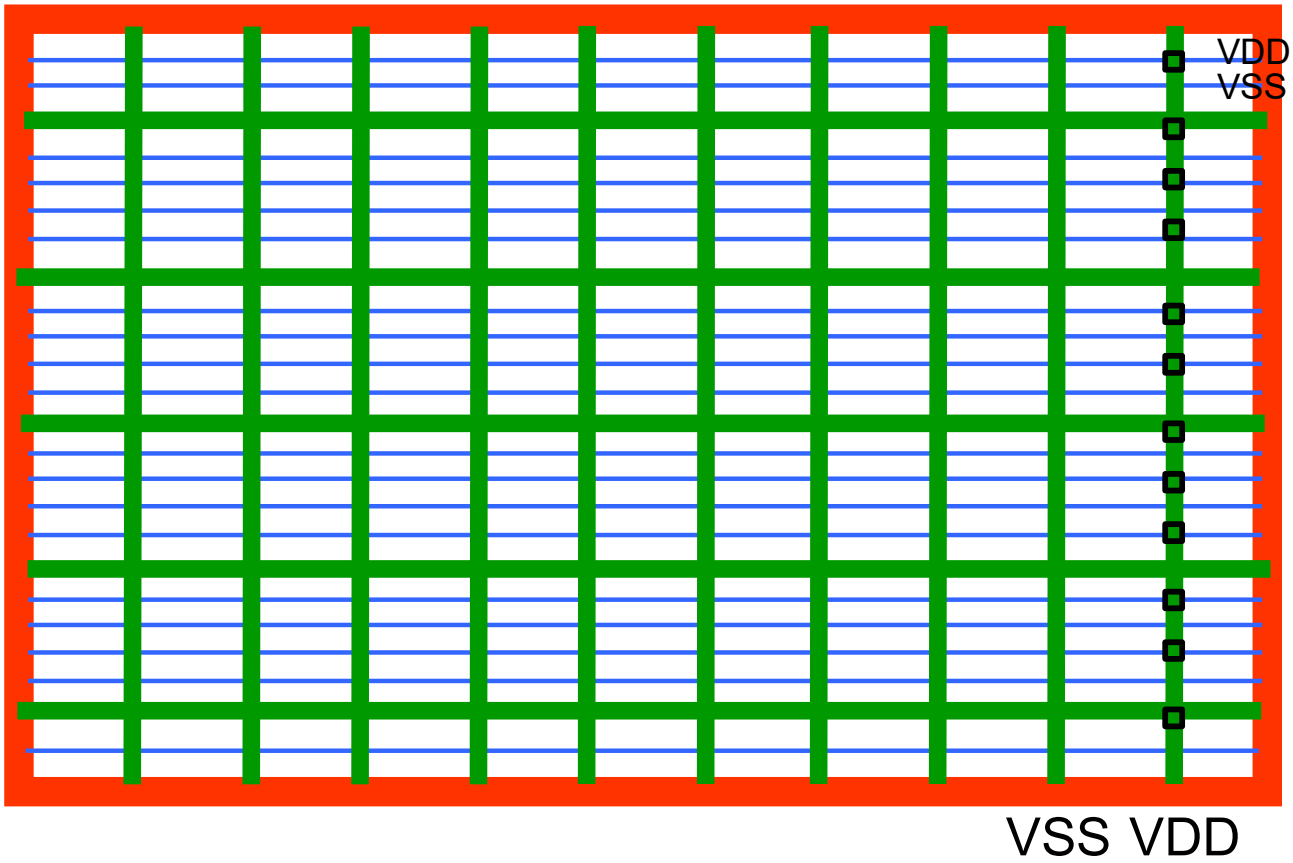
- Space of VDD mesh and VSS mesh
  - Top metal layer's minimum space rule
- Draw the lowest mesh first
  - Stacked-via range must be changed to get better routability. It is recommended that the stacked-via size is the cross sectional area of the upper and the lower metal of the via.

M1 rail:
M4 mesh:
M5 mesh:
M6 mesh:
M1_2B mesh:
M2_2B mesh:

Power Grid Design and Analysis

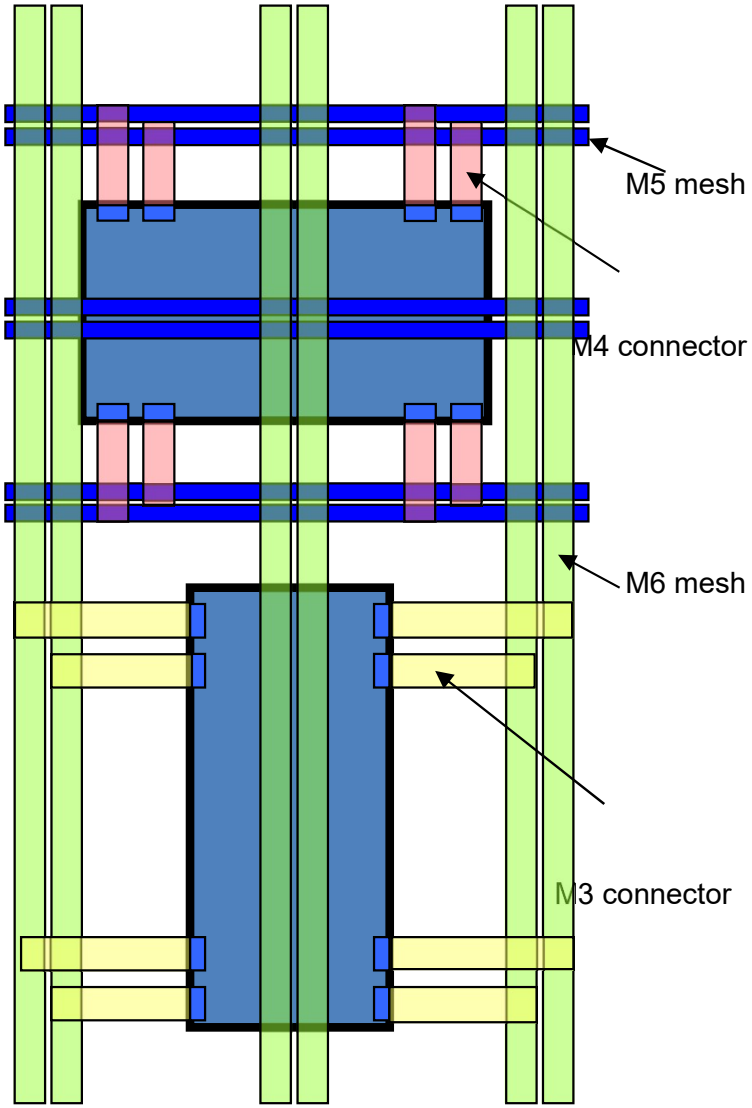
- Via Stacks
  - Typically we drop vias every time an upper layer power stripe crosses an M1 rail.
  - We choose the via stack so that the M1 rail is the limiting factor for IR drop – not the via stack.
  - The via stack can influence routability, depending on the number of tracks that it blocks.
    - Below example has 2x3 via array and blocks 4 vertical + 3 horizontal tracks.
    - A 2x2 array would block 3 vertical + 3 horizontal tracks.
    - A 1x4 array would block 5 vertical + 1 horizontal tracks.



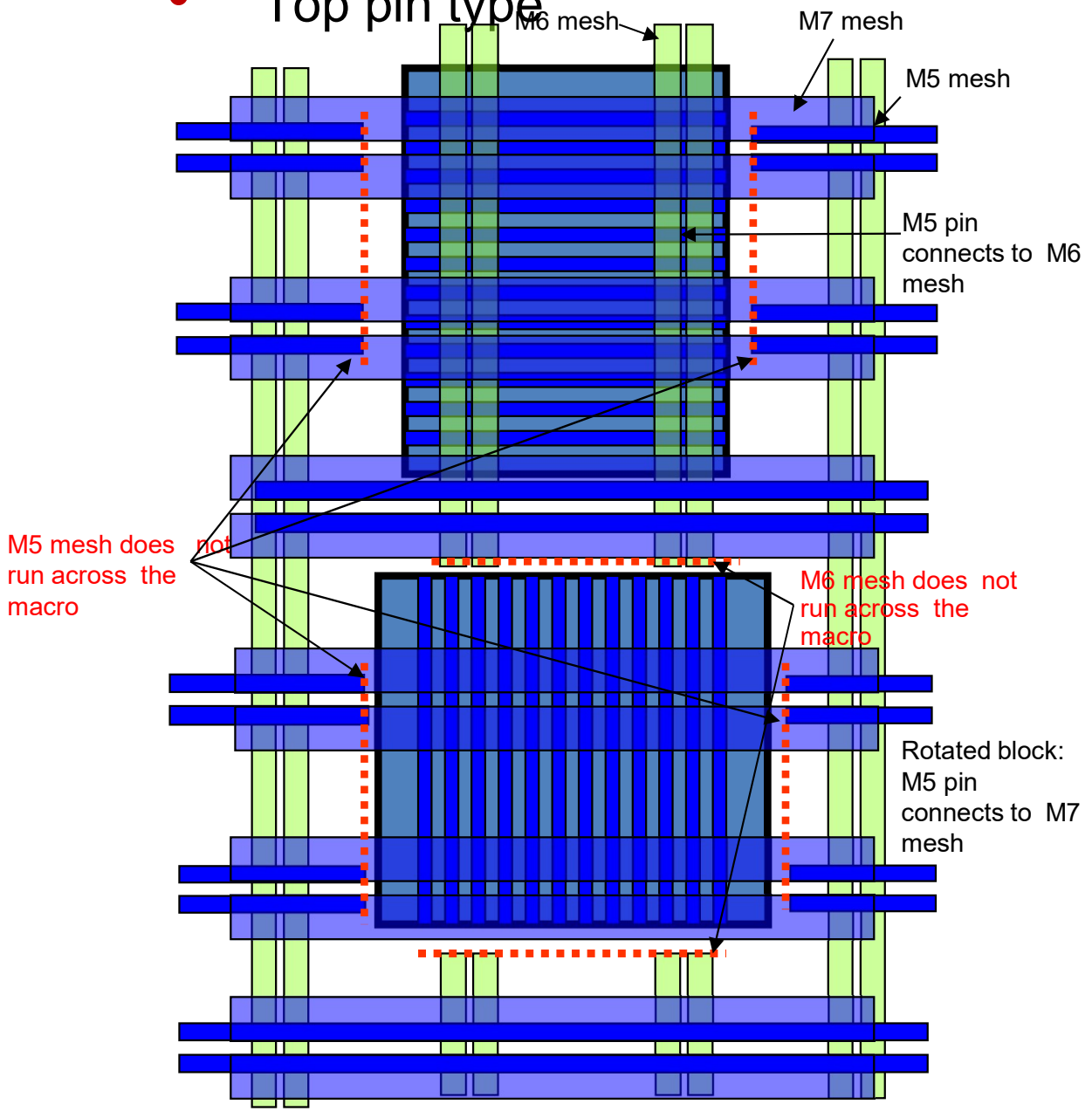


Macro Connection

- Side pin type



- Top pin type

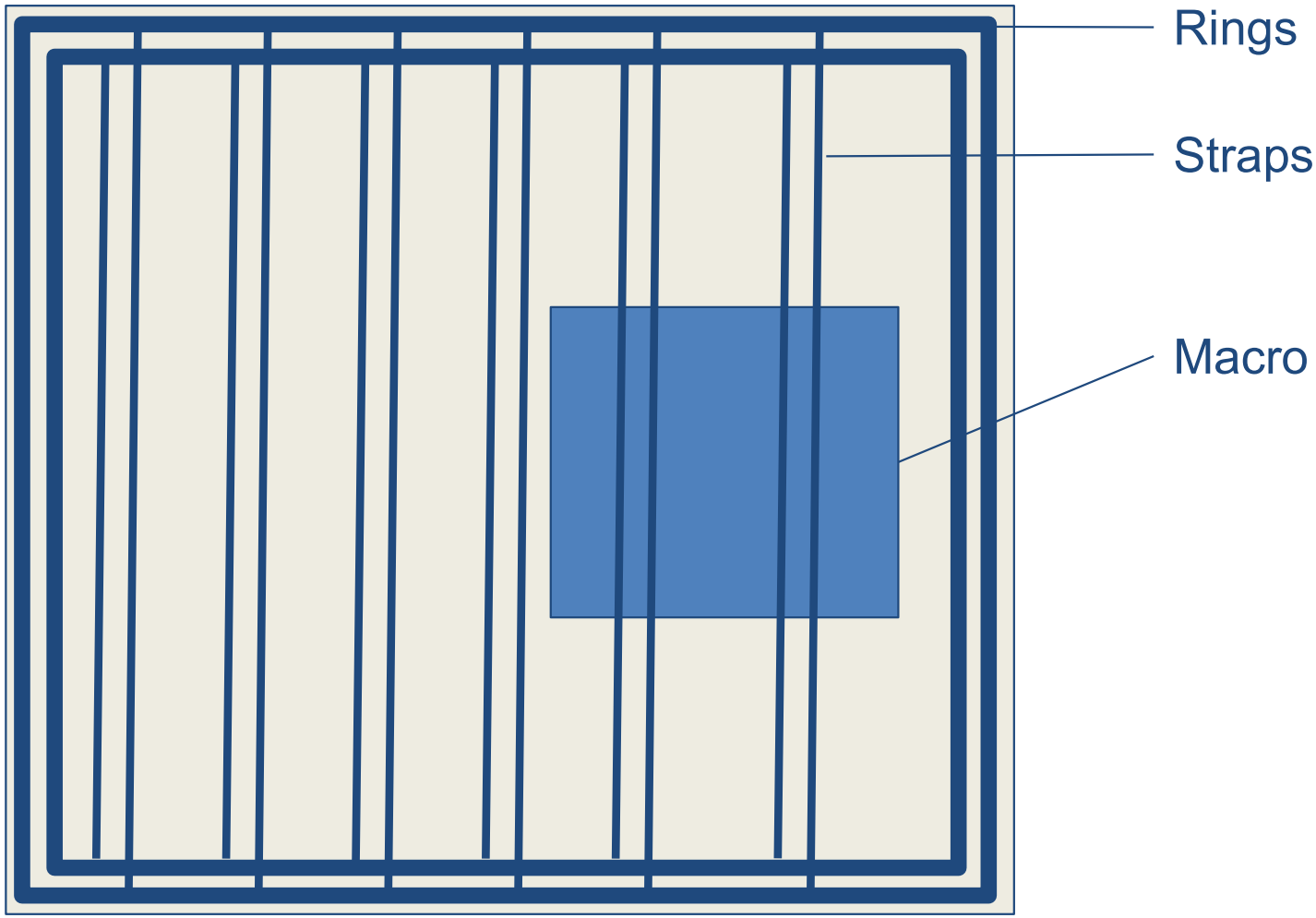


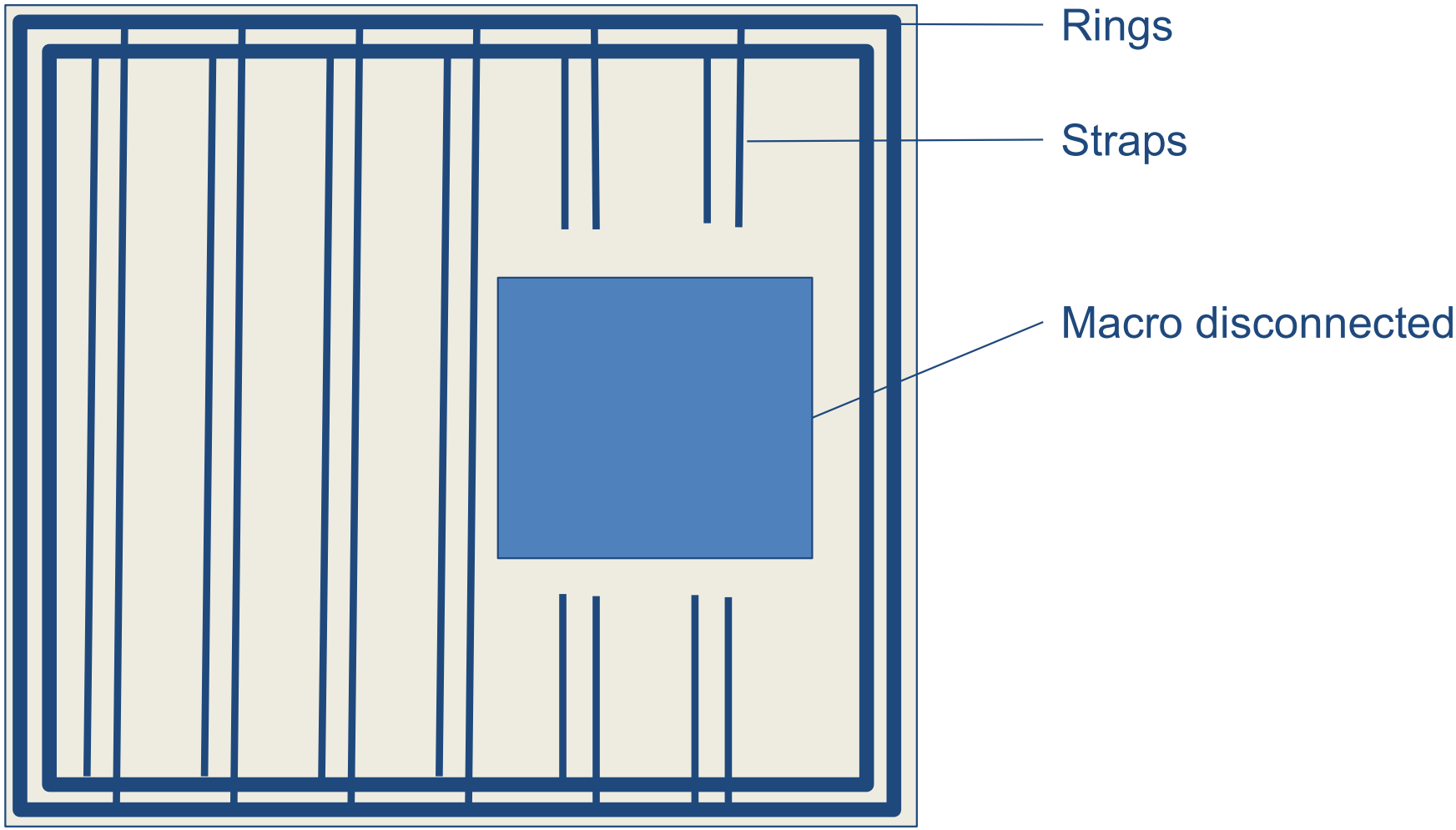
pdn (aka *pdngen*) User Interface

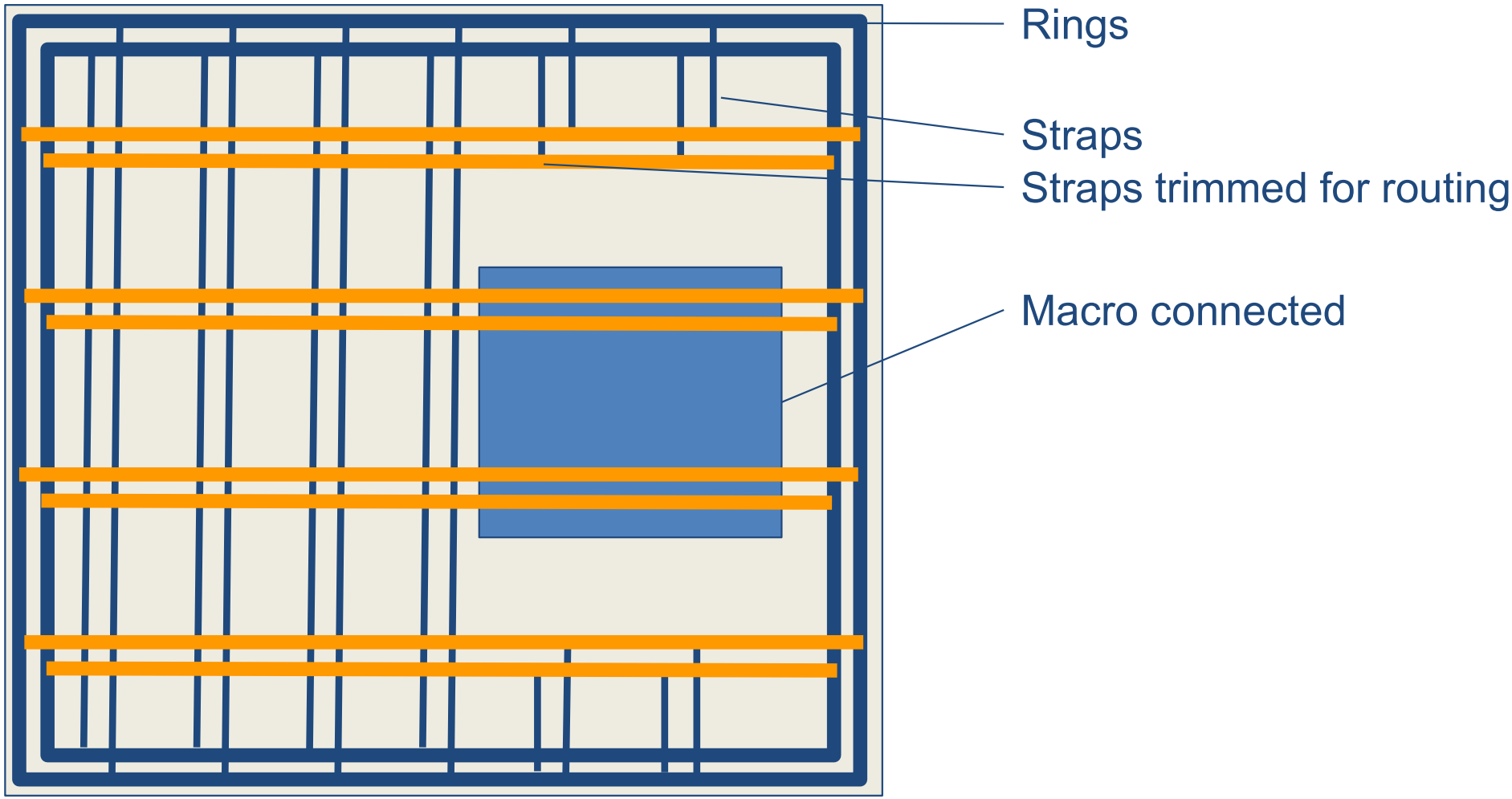
- set_voltage_domain - define a new power domain
- define_pdn_grid - defines a new grid
- add_pdn_stripe - add straps and followpins
- add_pdn_ring - add rings
- add_pdn_connect - add layer connectivity
- pdngen - create grids

Main PDN Build Steps

1. Extract existing routing and blockages
https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/PdnGen.cc#L112
2. For each grid:
https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/PdnGen.cc#L136
    a. Create rings, if requested
    https://github.com/The-OpenROAD-Project/OpenROAD/blob/c545b771cfd5a7a9c1f4c71e869899ac978ffb5f/src/pdn/src/rings.cpp#L223
    b. Create straps
    https://github.com/The-OpenROAD-Project/OpenROAD/blob/c545b771cfd5a7a9c1f4c71e869899ac978ffb5f/src/pdn/src/straps.cpp#L170
    c. Check connectivity and do channel repair (slide)
3. Estimate vias
https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/PdnGen.cc#L148
4. Trim excess
https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/PdnGen.cc#L151
5. Commit to database
https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/PdnGen.cc#L731
    a. Create all vias DRC free
    https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/via.cpp
    b. Write wires to database


Rings
Straps
Macro


Rings
Straps
Macro disconnected


Rings
Straps
Straps trimmed for routing
Macro connected

Channel Repair

https://github.com/The-OpenROAD-Project/OpenROAD/blob/c34a30e0045eb68ad30c98f653e57f9b7f12d83c/src/pdn/src/straps.cpp#L1532

1. Find disconnected channels (floating power grid shapes)
2. For each channel:
    a. Find appropriate connection layers
    b. Attempt repair in the center of the channels, if success, stop
    c. Bisect channel and attempt repair in on left and right side of channel, if success, stop, else keep trying
        i. Stop when bisection no longer can make progress

pdngen Improvements / Problem Statements

- Automatic connectivity rules (add_pdn_connect)
  - Create rules automatically for defining grid connectivity based on layers used, instances present, etc.
- Edge connectivity
  - Enable the power grid to connect to edge ports for macros, right now this is only possible on pad cells and standard cells, for macros we only connect from the top which requires macros use lower metals for power and routing. This is not a huge problem in nodes with a large number of metal layers, but for sky130, ihp130, etc, this imposes a large penalty.
- Power grid reinforcement (eco power grid)
  - Later in most flows, after detailed placement, it may be possible to determine if additional wires are needed to ensure IR drop stays within limits based on the power requirements of those areas. On the opposite side, it maybe possible to prune the grid if IR drop is not an issue and free up routing resources. [cf. https://arxiv.org/abs/2110.14184]
- Automatic power grid definition (hard)
  - Given an IR drop goal, create a power grid that meets this requirement based on the parasitic resistances and estimated power (either from placement or good guesses)

Paper References

- H. Murata, K. Fujiyoshi, S. Nakatake and Y. Kajitani, “VLSI module placement based on rectangle-packing by the sequence-pair”, IEEE Trans. on CAD, 15(12), 1996, pp. 1518-1524.
- X. Tang and D. F. Wong, “FAST-SP: a fast algorithm for block placement based on sequence pair”, Proc. ASP-DAC, 2001, pp. 521-526.
- S. N. Adya and I. L. Markov, “Fixed-outline Floorplanning : Enabling Hierarchical Design”, IEEE Trans. on VLSI Systems, 11(6), 2003, pp. 1120-1135.
- S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, “Optimization by Simulated Annealing”, Science 220(4598) 1983, pp. 671-680.

BACKUP

B-Tree* Floorplan Representation

- Ordered binary tree (cf. “Packing Tree”, “O-Tree”)
- Root represents the block on the left-bottom corner
- Left child of node n_i represents the lowest unvisited block that belongs to the set of blocks located on the right-hand side of and adjacent to b_i

$$x_j = x_i + w_i$$

- Right child of the node n_i represents the lowest block located above and with x-coordinate equal to that of b_i

$$x_j = x_i$$