NOTE: Please watch for Chipshub-related admin notes in Piazza !

ECE 260C, Spring 2026

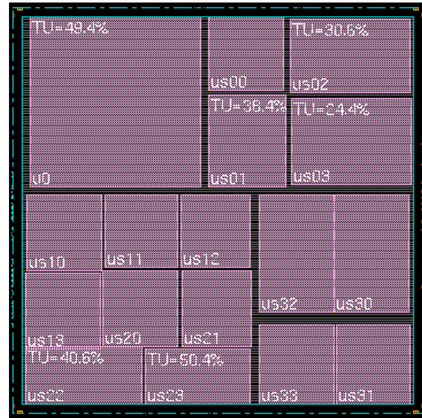
Floorplan and PDN

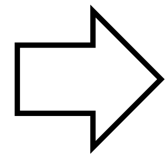
Andrew B. Kahng

Thanks to: Zhiang Wang, Peter Gadfort

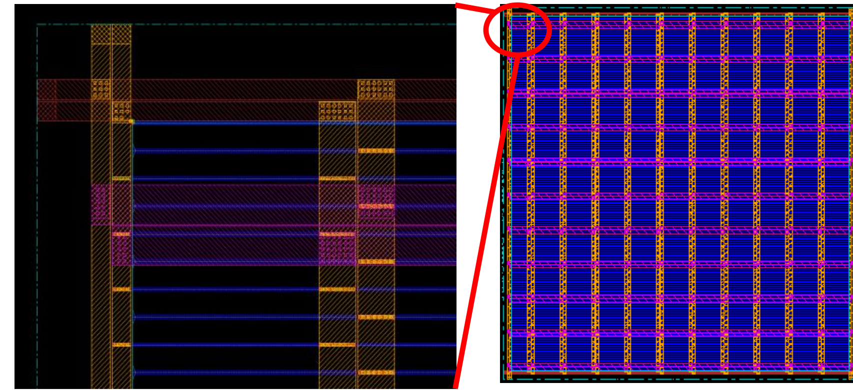
Physical Design Flow Pictures (old ECE 260B slide)

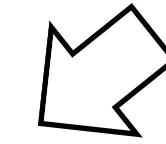
- Floorplanning



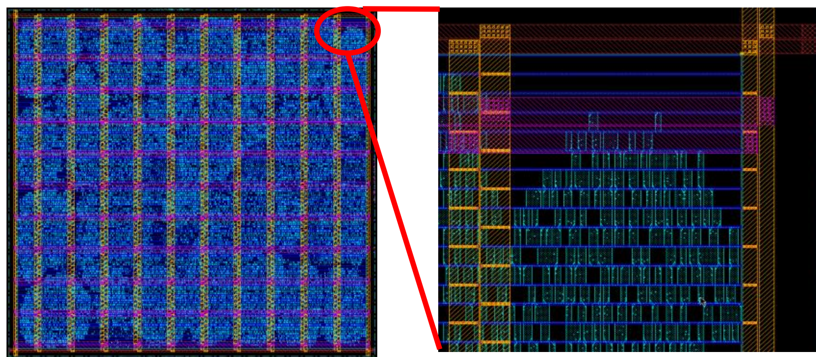


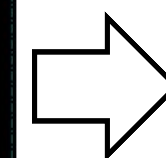
- Powerplanning



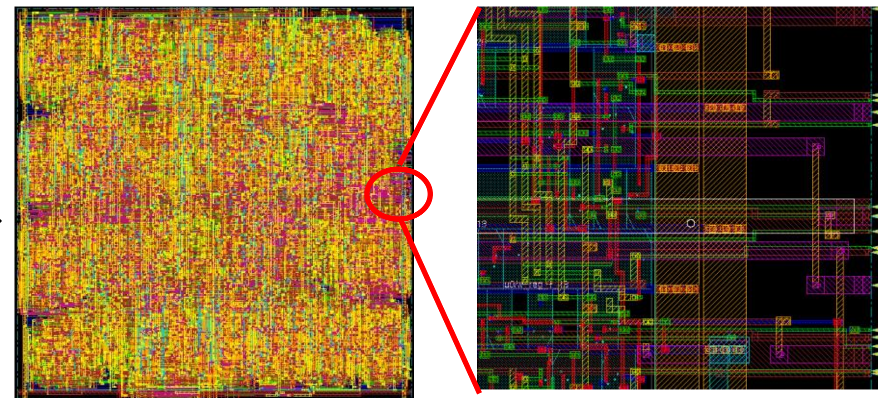


- Placement



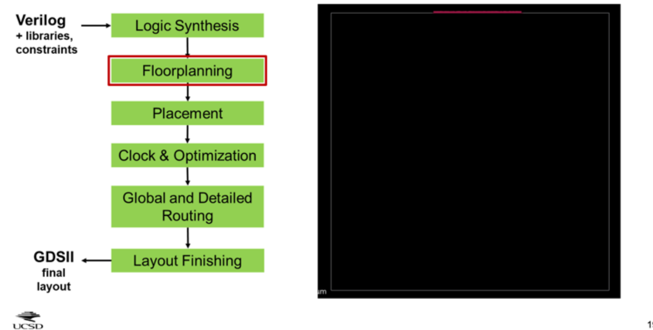


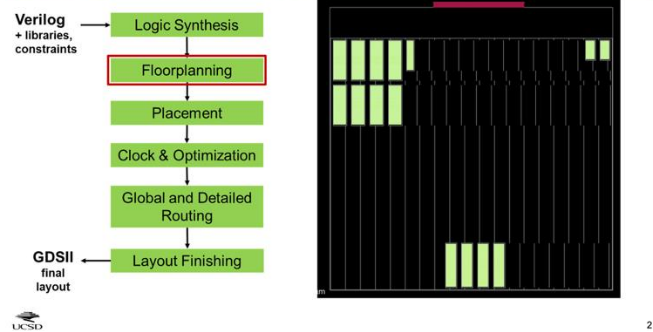
- Routing

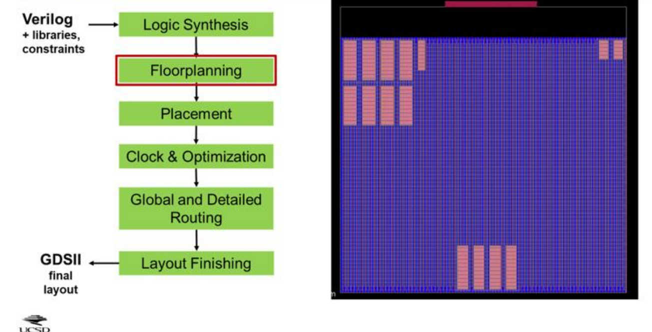


Floorplanning

Recall: "Floorplanning" Performs Many Tasks !







https://openroad.readthedocs.io/en/latest/main/src/ifp/README.html ++

IO placement = ppl

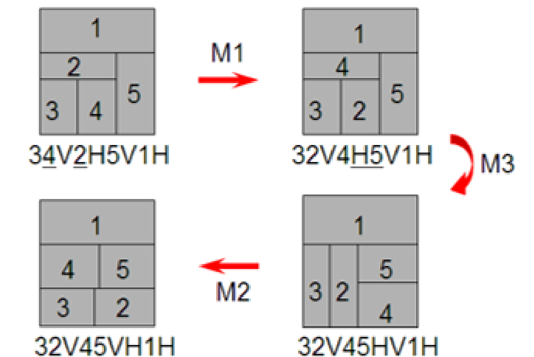
Macro placement = mpl

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 (ppl)

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 tool [17] to create an I/O pin assignment flow. In doing so,

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

ppl = 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

openroad.readthedocs.io/en/latest/main/src/ppl/README.html

OpenROAD documentation

Stars 1.8k

Getting Started with OpenROAD
Building OpenROAD
Getting Started with the OpenROAD Flow - OpenROAD-flow-scripts
Tutorials
Git Quickstart
Man pages
OpenROAD User Guide
OpenROAD APIs
Database
GUI
Partition Management
Restructure
Floorplan Initialization
Pin Placement

Pin Placer

Place pins on the boundary of the die on the track grid to minimize net wirelengths. Pin placement also creates a metal shape for each pin using min-area rules.

For designs with unplaced cells, the net wirelength is computed considering the center of the die area as the unplaced cells position.

Commands

Note

- Parameters in square brackets `[-param param]` are optional.
- Parameters without square brackets `-param2 param2` are required.

Define Pin Shape Pattern

The `define_pin_shape_pattern` command defines a pin placement grid on the specified layer. This grid has positions inside the die area, not only at the edges of the die boundary.

```
define_pin_shape_pattern
    [-layer layer]
```

Contents

Commands
Define Pin Shape Pattern
Options
Face-to-Face direct-bonding IOs
Set IO Pin Constraints
Exclude IO Pin Region
Clear IO Pin Constraints
Set Pin Length
Set Pin Length Extension
Set Pin Thickness Multiplier
Set Simulated Annealing
Simulated Annealing Debug Mode
Place specific Pin
Place all Pins
Write Pin Placement
Useful Developer Commands
Example scripts
Regression tests
Limitations
References
FAQs
License

Macro Placement: Hier-RTLMP (mpl)

Thanks to Prof. Zhiang Wang

Motivation

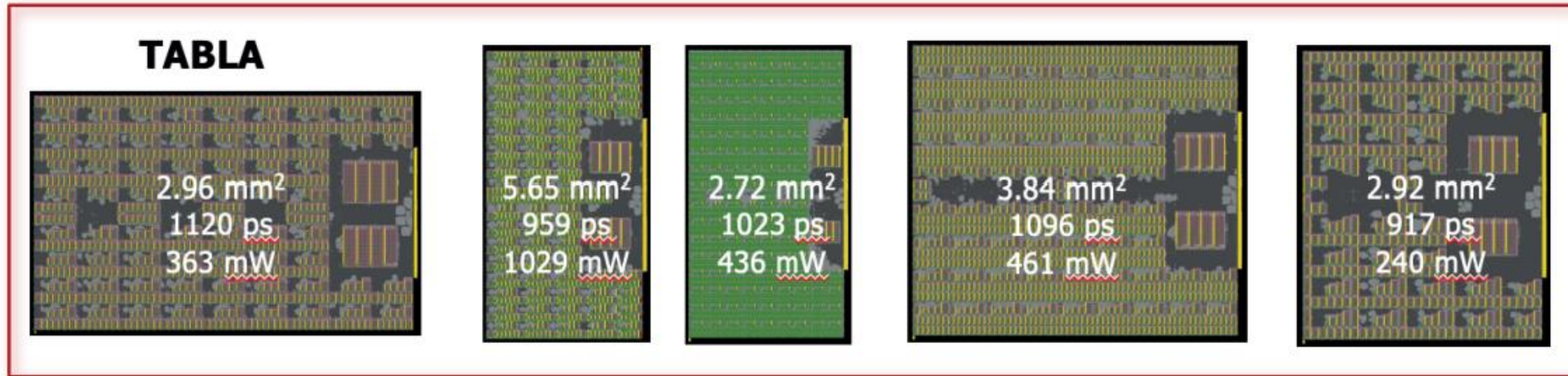
- Macro placement is a critical component of the "handoff" from the frontend to the backend of a physical synthesis flow.

- Human experts usually need to consider multiple factors when they do macro placement: (see, e.g., works by Vidal-Obiols et al. and by J.-M. Lin et al. among references in this paper)
  - Dataflow
  - Logical hierarchy
  - Connectivity between macros and input-output (IO) pins
  - …

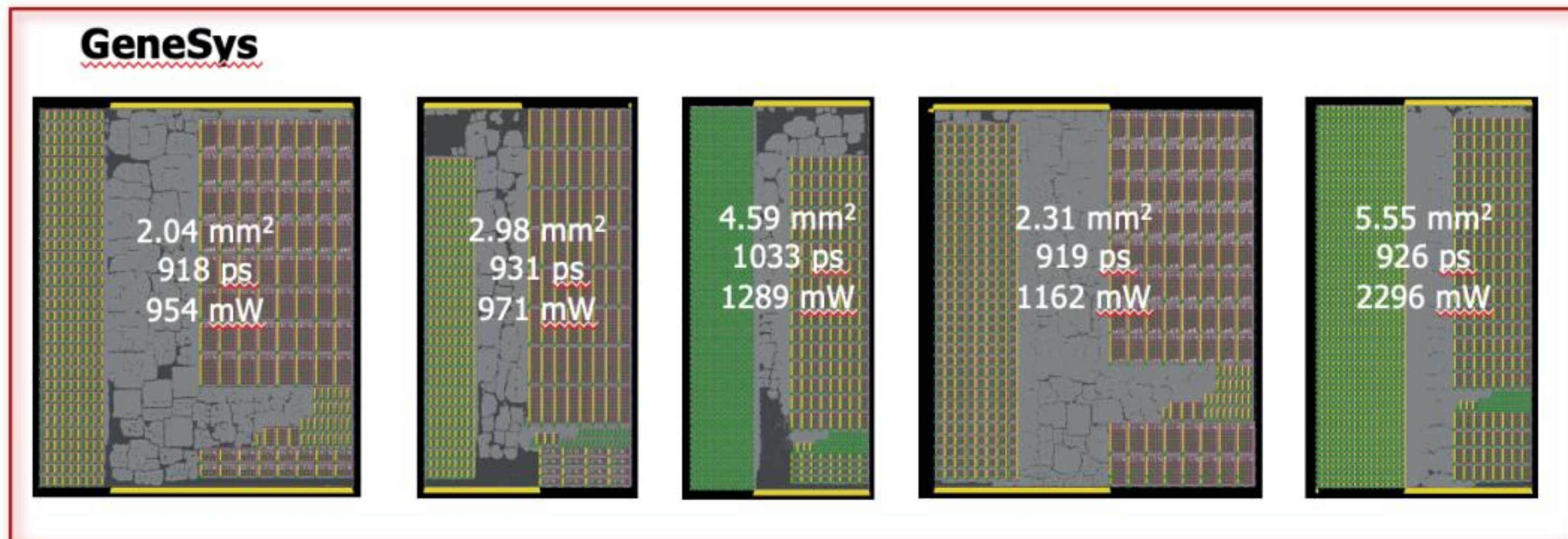
- ML hardware accelerators make macro placement even more challenging
  - Hundreds of macros (examples will be given in the next slide)
  - Complex RTL structure with long, inscrutable autogenerated module names

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.

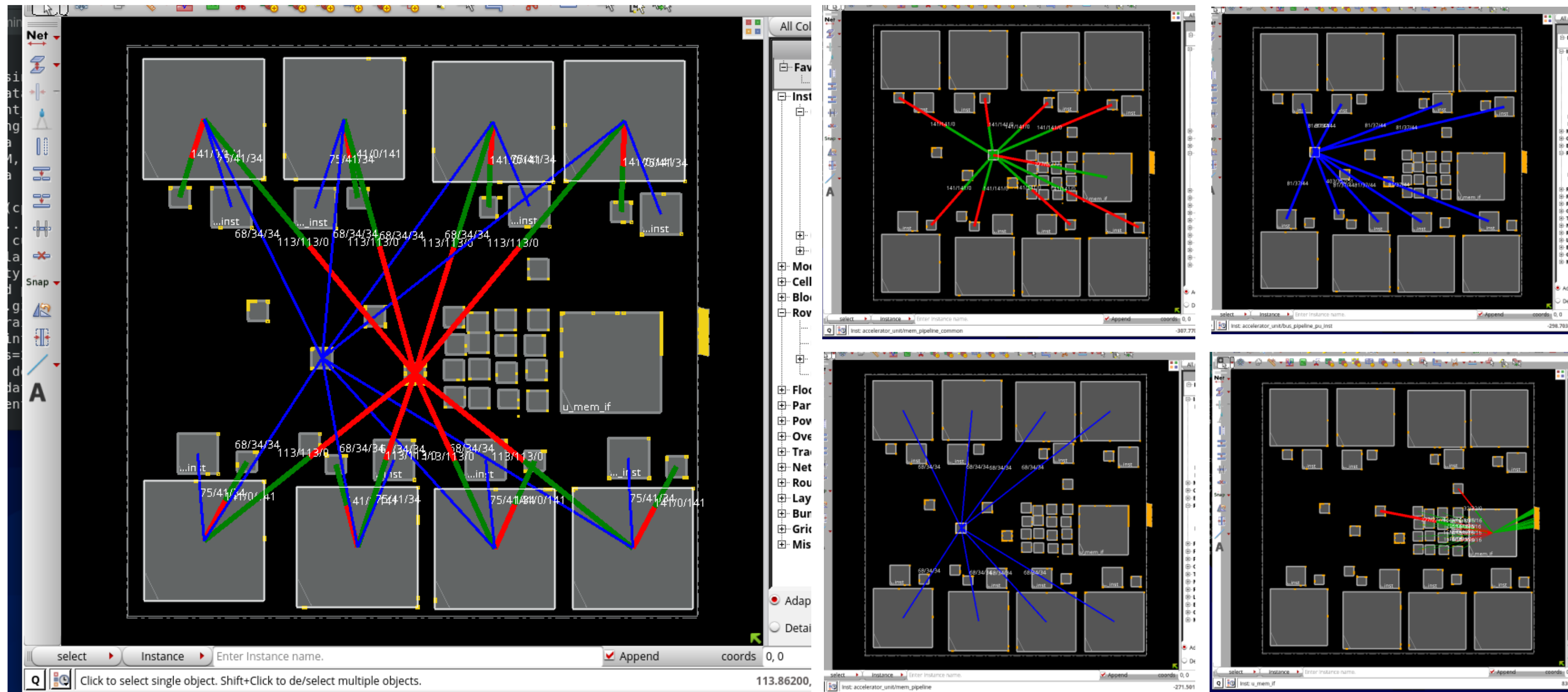
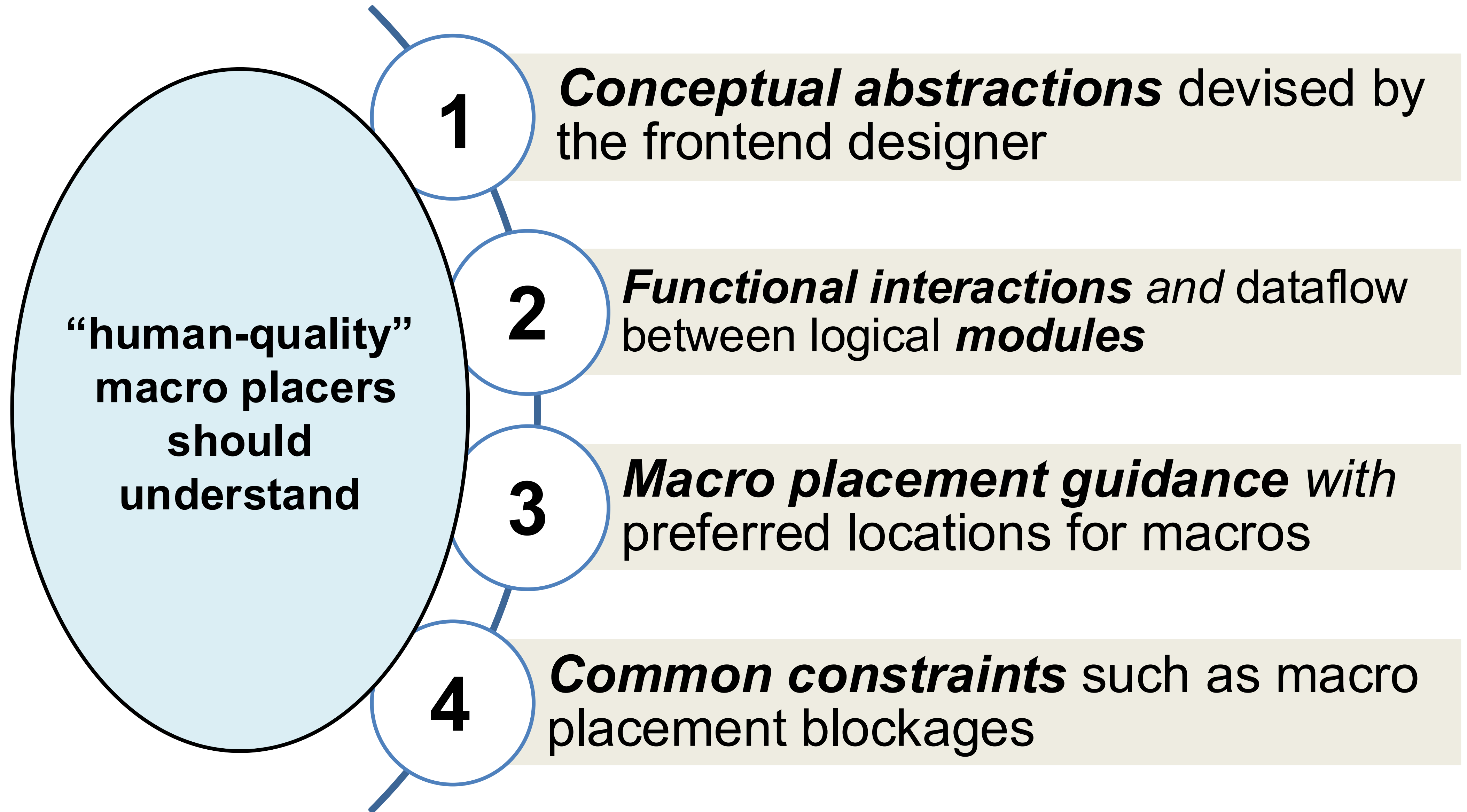
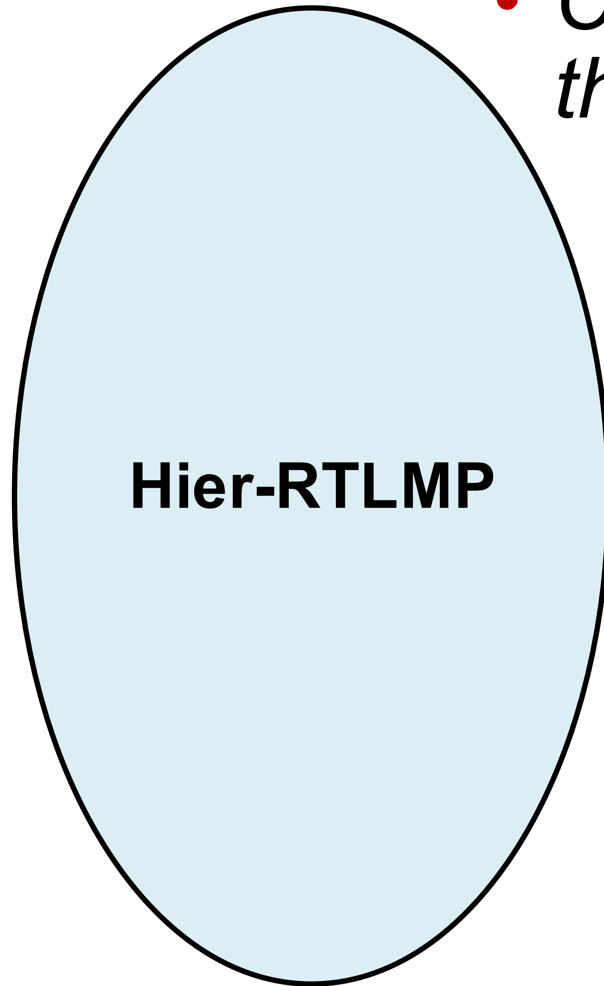


TABLA: systolic array structures
(Processing Units + Ctrl Logic + Mem Blocks)

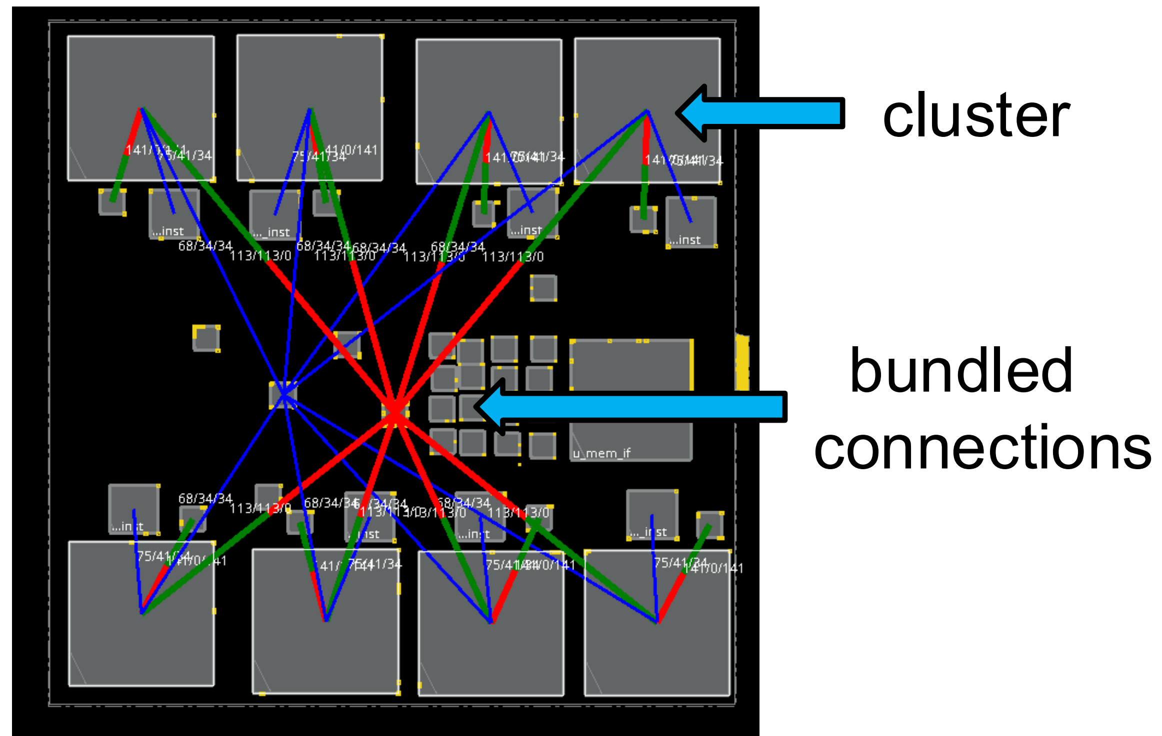
What Should a Human-quality Macro Placer Do ?



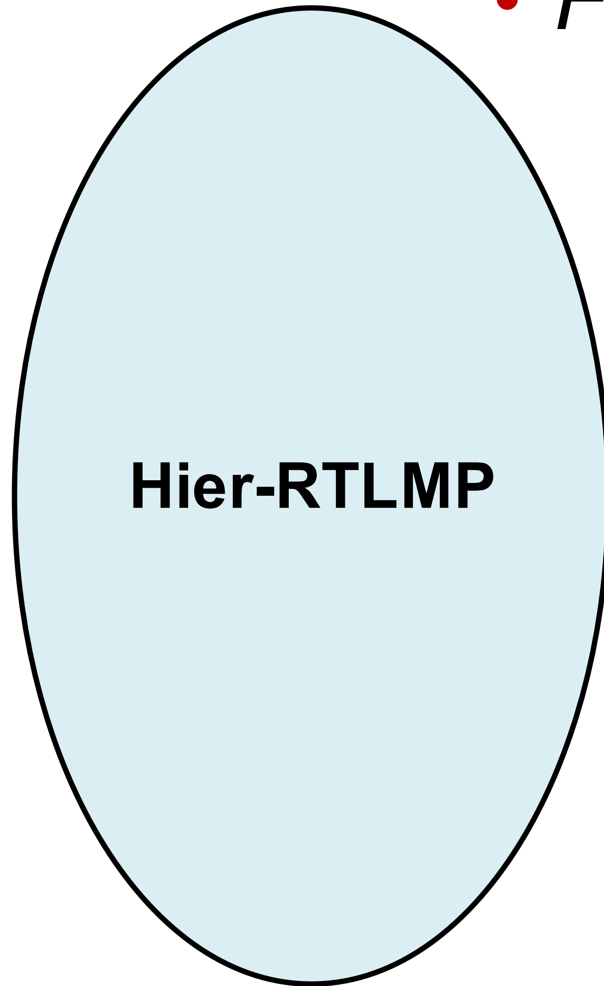
Macro Placer in OpenROAD: Hier-RTLMP



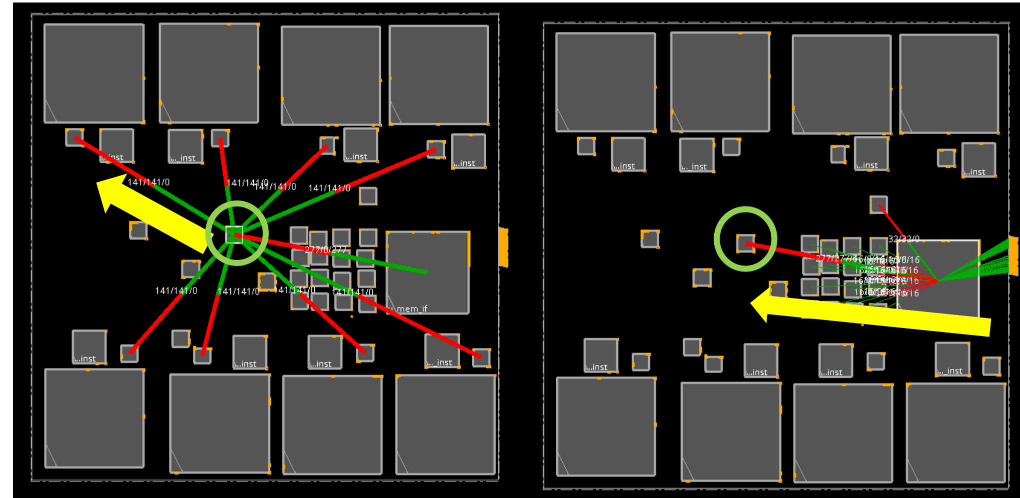
- *Conceptual abstractions devised by the frontend designer*
  - Through converting the structural netlist representation of the RTL design into a clustered netlist



Macro Placer in OpenROAD: Hier-RTLMP

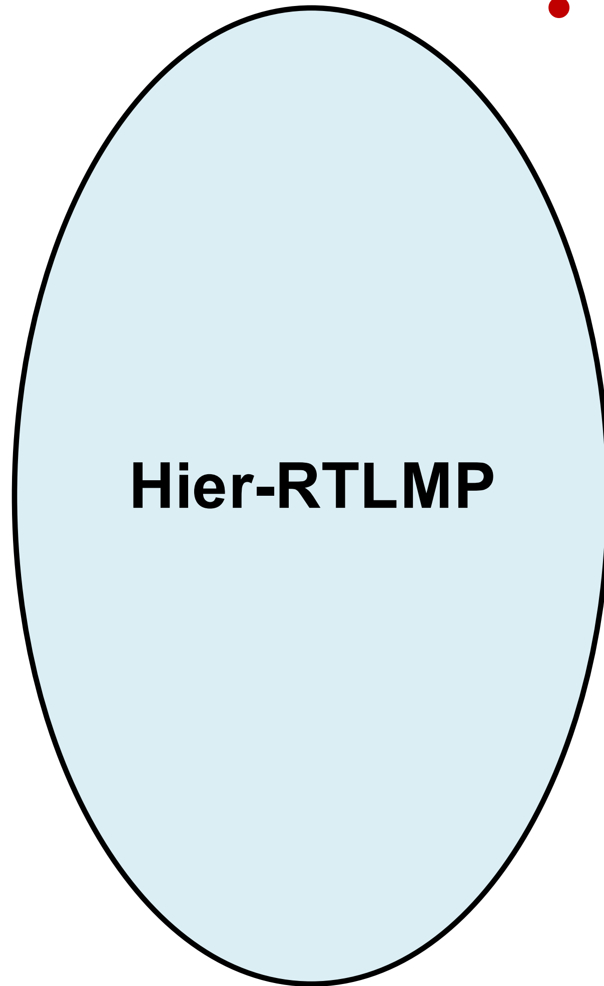


- *Functional interactions*
  - Through analyzing the dataflow between logical modules (including Input-Output pins)

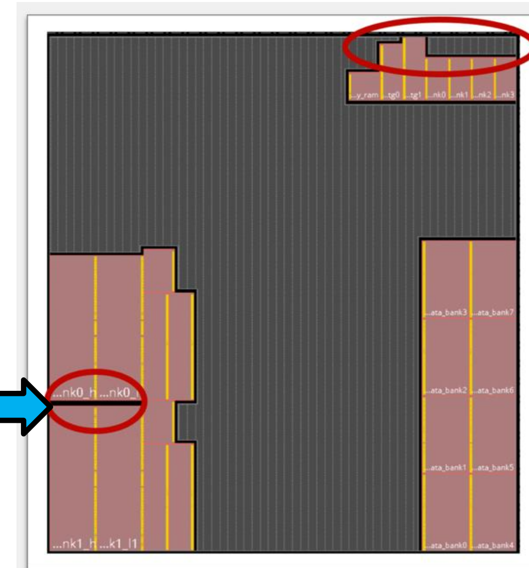


See https://github.com/The-OpenROAD-Project/OpenROAD/pull/9328 -- this part of mpl was removed on February 5, 2026

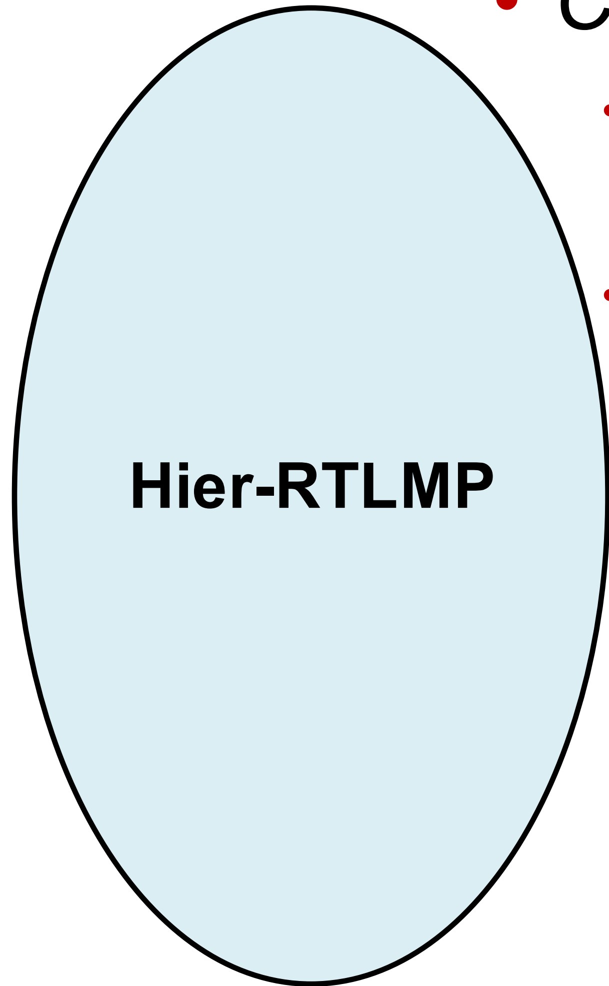
Macro Placer in OpenROAD: Hier-RTLMP



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

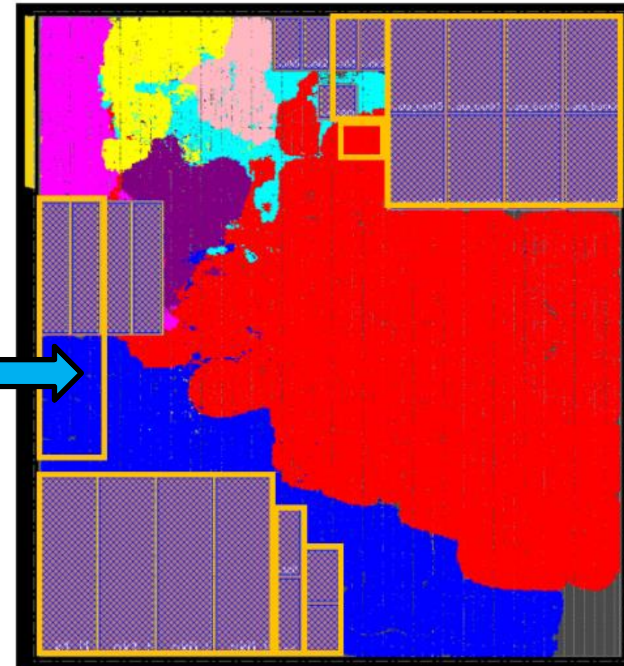


Macro Placer in OpenROAD: Hier-RTLMP

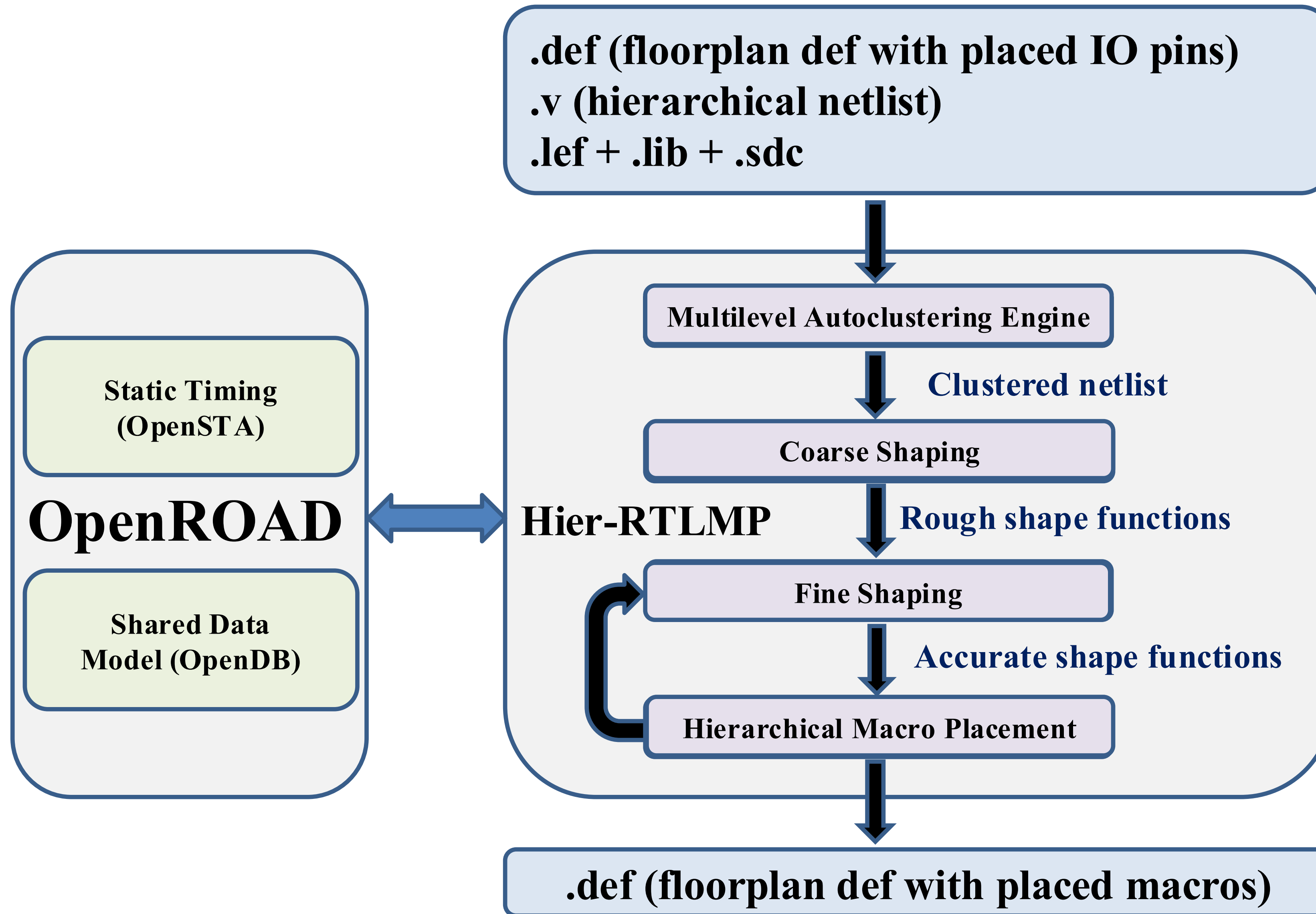


- *Common constraints*
  - Fixed macro constraints
  - Through pushing macros to their preferred locations

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)

```cpp
266   // Transform the logical hierarchy into a physical hierarchy.
267   void HierRTLMP::runMultilevelAutoclustering()
268   {
269     clustering_engine_ = std::make_unique<ClusteringEngine>(
270         block_, network_, logger_, tritonpart_, graphics_.get());
271
272     // Set target structure
273     clustering_engine_->setTree(tree_.get());
274     clustering_engine_->setHalos(macro_to_halo_);
275     clustering_engine_->run();
276
277     if (!tree_->has_unfixed_macros) {
278       skip_macro_placement_ = true;
279       return;
280     }
281
282     if (graphics_) {
283       graphics_->finishedClustering(tree_.get());
284     }
285   }
```

Line 267 in hier_rtlmp.cpp (link)

```cpp
void ClusteringEngine::run()
{
  init();

  if (!tree_->has_unfixed_macros) {
    return;
  }

  createRoot();
  setBaseThresholds();
  createIOClusters();

  if (design_metrics_->getNumStdCell() == 0) {
    logger_->warn(MPL, 25, "Design has no standard cells!");
    tree_->has_std_cells = false;
    treatEachMacroAsSingleCluster();
  } else {
    multilevelAutocluster(tree_->root.get());

    std::vector<std::vector<Cluster*>> mixed_leaves;
    fetchMixedLeaves(tree_->root.get(), mixed_leaves);
    breakMixedLeaves(mixed_leaves);
  }

  if (logger_->debugCheck(MPL, "multilevel_autoclustering", 1)) {
    logger_->report("\nPrint Physical Hierarchy\n");
    printPhysicalHierarchyTree(tree_->root.get(), 0);
  }

  // Map the macros in each cluster to their HardMacro objects
  for (auto& [cluster_id, cluster] : tree_->maps.id_to_cluster) {
    mapMacroInCluster2HardMacro(cluster);
  }
}
```

Line 47 in clusterEngine.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

```
331  void HierRTLMP::runCoarseShaping()
332  {
333    setRootShapes();
334
335    if (tree_->has_only_macros) {
336      logger_->warn(MPL, 27, "Design has only macros!");
337      tree_->root->setClusterType(HardMacroCluster);
338      return;
339    }
340
341    if (graphics_) {
342      graphics_->startCoarse();
343    }
344
345    calculateChildrenTilings(tree_->root.get());
346
347    searchAvailableRegionsForUnconstrainedPins();
348    createPinAccessBlockages();
349    setPlacementBlockages();
350  }
```

Line 331 in hier_rtlmp.cpp (link)

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

Simulated Annealing
Line 468 in hier_rtlmp.cpp (link)
(SACoreSoftMacro.h)

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

Multi-threading implementation
Line 517 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

```
1345  while (remaining_runs > 0) {
1346    SoftSAVector sa_batch;
1347    const int number_of_attempts
1348        = graphics_ ? 1 : std::min(remaining_runs, num_threads_);
1349
1350    for (int i = 0; i < number_of_attempts; i++) {
1351      const float utilization = utilization_list[run_id++];
1352      if (!validUtilization(utilization, outline, macros)) {
1353        continue;
1354      }
1355
1356      std::vector<SoftMacro> inflated_macros
1357          = applyUtilization(utilization, outline, macros);
1358
1359      const bool single_array_single_std_cell_cluster
1360          = singleArraySingleStdCellCluster(macros);
1361
1362      std::unique_ptr<SACoreSoftMacro> sa
1363          = std::make_unique<SACoreSoftMacro>(tree_.get(),
1364                                              outline,
1365                                              inflated_macros,
1366                                              placement_core_weights_,
1367                                              cluster_placement_weights_,
1368                                              notch_h_th_,
1369                                              notch_v_th_,
1370                                              pos_swap_prob_ / action_sum,
1371                                              neg_swap_prob_ / action_sum,
1372                                              double_swap_prob_ / action_sum,
1373                                              exchange_swap_prob_ / action_sum,
1374                                              resize_prob_ / action_sum,
1375                                              init_prob_,
1376                                              max_num_step_,
1377                                              num_perturb_per_step,
1378                                              random_seed_,
1379                                              graphics_.get(),
1380                                              logger_,
1381                                              block_);
1382      sa->setNumberOfSequencePairMacros(number_of_sequence_pair_macros);
1383      sa->enableEnhancements();
1384      sa->setFences(fences);
1385      sa->setGuides(guides);
1386      sa->setNets(nets);
1387      if (single_array_single_std_cell_cluster) {
1388        sa->forceCentralization();
1389      }
1390      sa_batch.push_back(std::move(sa));
1391    }
1392
```

Line 1345 in hier_rtlmp.cpp (link)

```
1675  std::vector<SoftMacro> HierRTLMP::applyUtilization(
1676      const float utilization,
1677      const odb::Rect& outline,
1678      const std::vector<SoftMacro>& original_soft_macros) const
1679  {
1680    std::vector<SoftMacro> new_soft_macros = original_soft_macros;
1681    const bool single_array_single_std_cell_cluster
1682        = singleArraySingleStdCellCluster(original_soft_macros);
1683
1684    for (SoftMacro& new_soft_macro : new_soft_macros) {
1685      Cluster* cluster = new_soft_macro.getCluster();
1686
1687      if (!cluster || cluster->isIOCluster() || cluster->isFixedMacro()) {
1688        continue;
1689      }
1690
1691      if (new_soft_macro.isStdCellCluster()) {
1692        int64_t area = cluster->getArea();
1693        int width = std::sqrt(area);
1694        int height = width;
1695
1696        if (cluster->getNumStdCell() <= tiny_cluster_max_number_of_std_cells_
1697            || single_array_single_std_cell_cluster) {
1698          const int negligible_width = 1;
1699          width = negligible_width;
1700          height = width;
1701          area = width * static_cast<int64_t>(height);
1702        } else {
1703          area = cluster->getArea() / utilization;
1704          width = std::sqrt(area / min_ar_);
1705        }
1706
1707        const int minimum_width = area / width;
1708        const int maximum_width = width;
1709        Interval width_interval(minimum_width, maximum_width);
1710
1711        new_soft_macro.setShapes({width_interval}, area);
1712      }
1713
1714      if (new_soft_macro.isMixedCluster()) {
1715        const TilingList& tilings = cluster->getTilings();
1716        int64_t macro_area = tilings.back().area();
1717        int64_t inflated_area
1718            = macro_area + (cluster->getStdCellArea() / utilization);
```

Line 1675 in hier_rtlmp.cpp (link)

Change since SP25: dedicated "runFineShaping" function no longer exists.

(But, similar code still used in UCLA-UCSD "ChipletPart" to find floorplan-feasible system partitioning into chiplets (arXiv).)

Fine Shaping + Hierarchical Macro Placement

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

```
287  void HierRTLMP::runHierarchicalMacroPlacement()
288  {
289    if (graphics_) {
290      graphics_->startFine();
291    }
292
293    adjustMacroBlockageWeight();
294    tiny_cluster_max_number_of_std_cells_
295        = computeTinyClusterMaxNumberOfStdCells();
296    placeChildren(tree_->root.get());
297  }
```

```
1189  void HierRTLMP::placeChildren(Cluster* parent)
1190  {
1191    if (parent->getClusterType() == HardMacroCluster) {
1192      placeMacros(parent);
1193      return;
1194    }
1195
1196    // Cover IO Clusters, Leaf Std Cells and Fixed Macros.
1197    if (parent->isLeaf()) {
1198      return;
1199    }
1200
1201    debugPrint(logger_,
1202               MPL,
1203               "hierarchical_macro_placement",
1204               1,
1205               "Placing children of cluster {}",
1206               parent->getName());
1207
1208    for (auto& cluster : parent->getChildren()) {
1209      clustering_engine_->updateInstancesAssociation(cluster.get());
1210    }
1211
1212    if (graphics_) {
1213      graphics_->setCurrentCluster(parent);
1214    }
```

Line 287 in hier_rtlmp.cpp (link)

```
1362      std::unique_ptr<SACoreSoftMacro> sa
1363          = std::make_unique<SACoreSoftMacro>(tree_.get(),
1364                                              outline,
1365                                              inflated_macros,
1366                                              placement_core_weights_,
1367                                              cluster_placement_weights_,
1368                                              notch_h_th_,
1369                                              notch_v_th_,
1370                                              pos_swap_prob_ / action_sum,
1371                                              neg_swap_prob_ / action_sum,
1372                                              double_swap_prob_ / action_sum,
1373                                              exchange_swap_prob_ / action_sum,
1374                                              resize_prob_ / action_sum,
1375                                              init_prob_,
1376                                              max_num_step_,
1377                                              num_perturb_per_step,
1378                                              random_seed_,
1379                                              graphics_.get(),
1380                                              logger_,
1381                                              block_);
1382      sa->setNumberOfSequencePairMacros(number_of_sequence_pair_macros);
1383      sa->enableEnhancements();
1384      sa->setFences(fences);
1385      sa->setGuides(guides);
1386      sa->setNets(nets);
1387      if (single_array_single_std_cell_cluster) {
1388        sa->forceCentralization();
1389      }
1390      sa_batch.push_back(std::move(sa));
1391    }
```

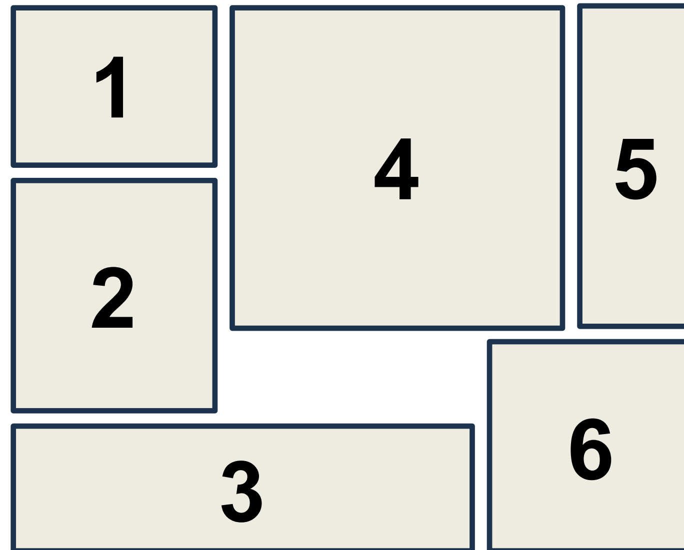
Line 1362 in hier_rtlmp.cpp (link)

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 block sequences.
- Given a sequence pair (Γ^+, Γ^-), the location of blocks can be determined as follows:
  - (Horizontal constraint): block i is **left of** block j if i appears before j in both Γ^+ and Γ^- → $(\ldots\ i\ \ldots\ j \ldots,\ \ldots\ j \ldots\ i\ \ldots)$
  - (Vertical constraint): block i is **below** block j if i appears after j in Γ^+ and i appears before j Γ^- → $(\ldots\ j\ \ldots\ i \ldots,\ \ldots\ j \ldots\ i\ \ldots)$

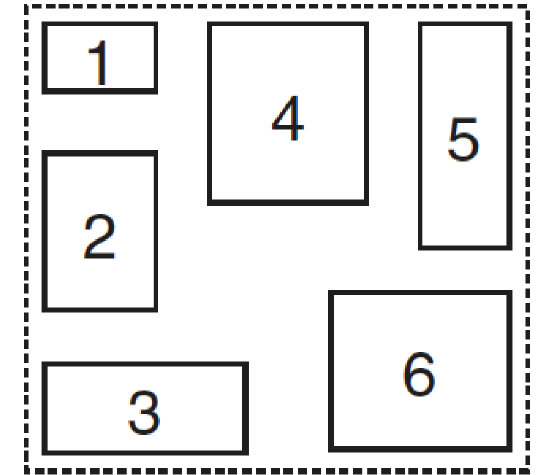


$(\Gamma^+, \Gamma^-) = (124536, 326145)$

Note: The “left of / left of / left of / …” **Longest Common Subsequence** determines the **minimum** possible width of the floorplan! (paper: link)

From Floorplan to Sequence Pair

- A sequence pair (SP) consists of two module (block) name sequences. Example: The floorplan shown at right can be represented as the SP (Γ_+, Γ_-) = (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.



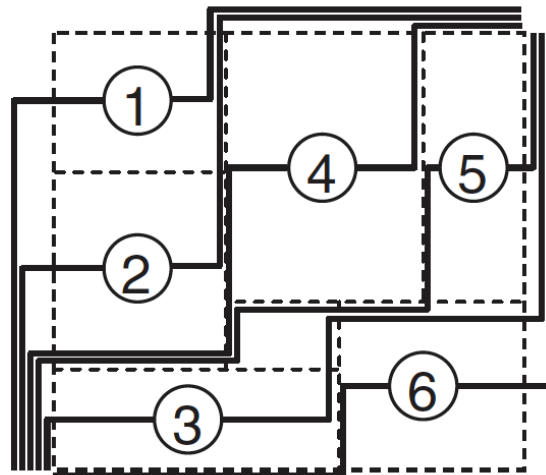


Figure (a)

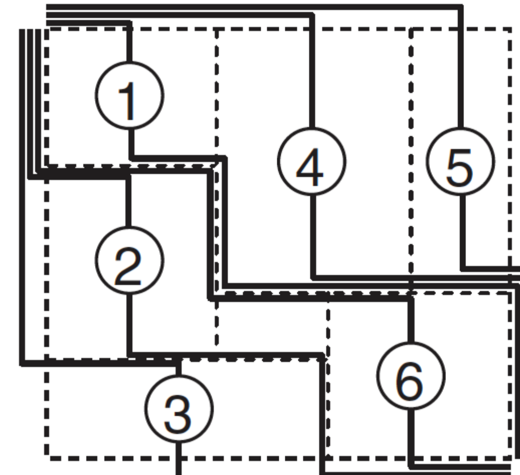
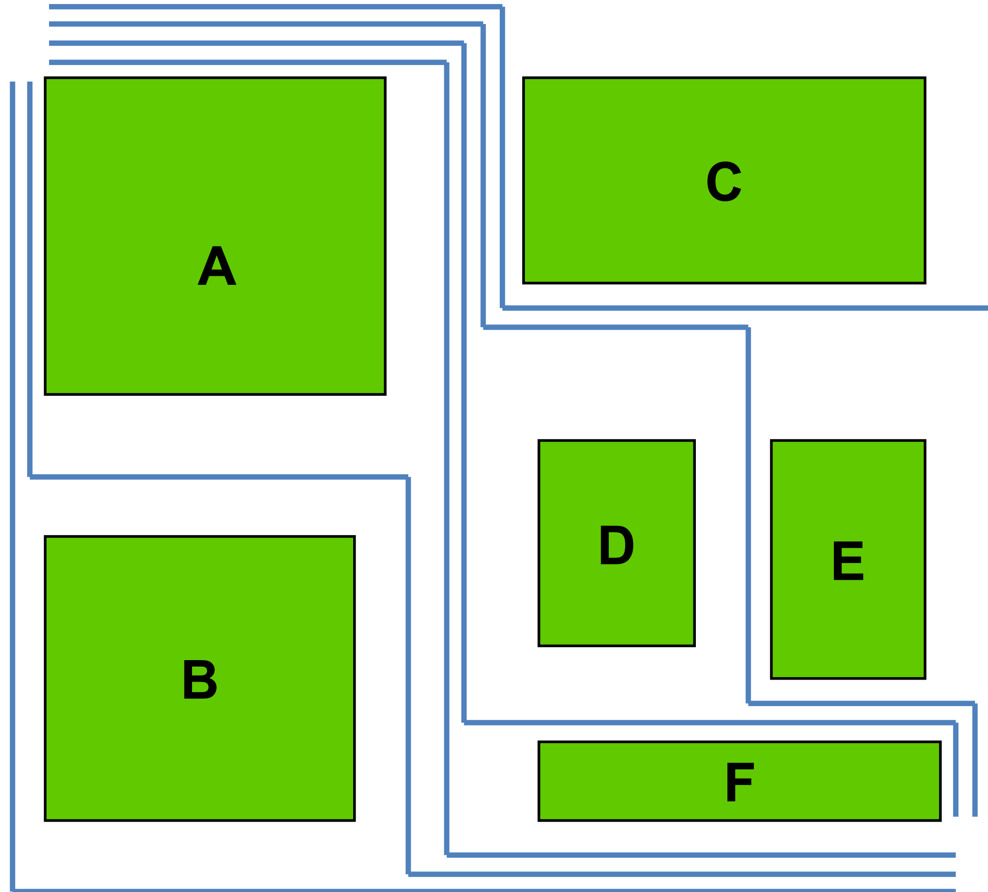


Figure (b)

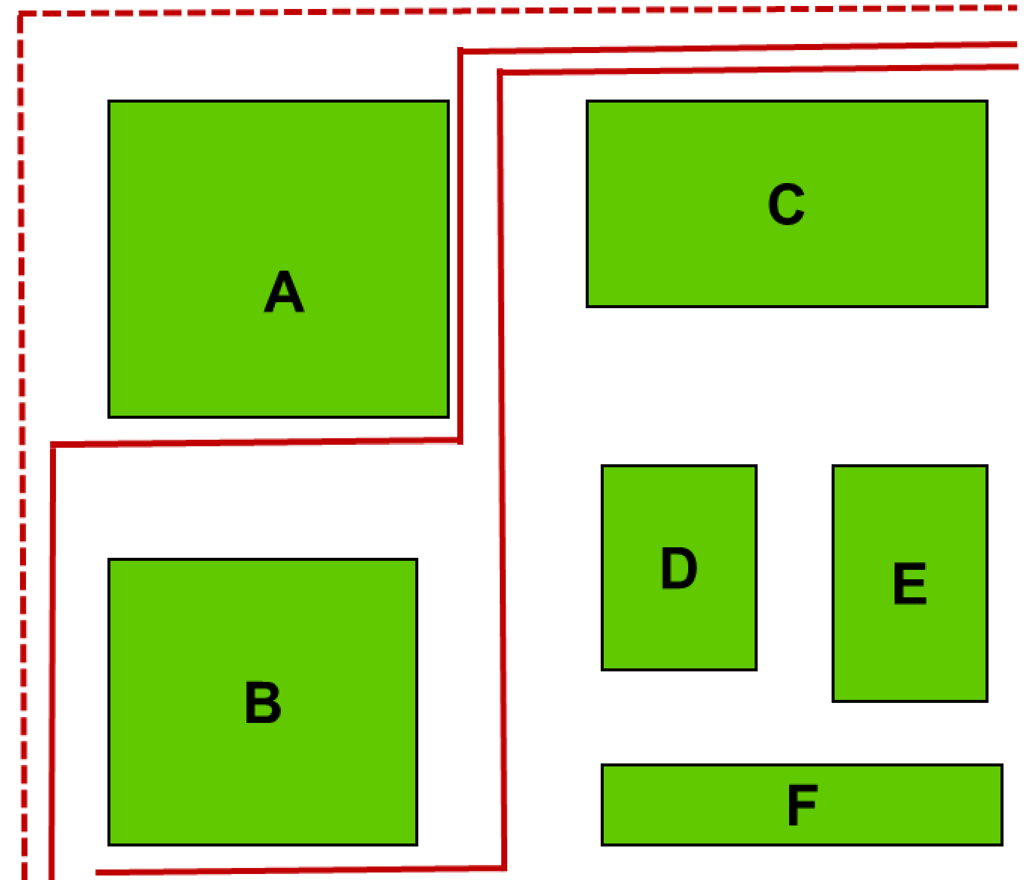
Think: "non-crossing staircases" !

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 (blocks) 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), first construct the HCG (a) and the VCG (b)
- Then, a "longest common subsequence" algorithm (see AdyaM03) is used in calculating the final floorplan (c)

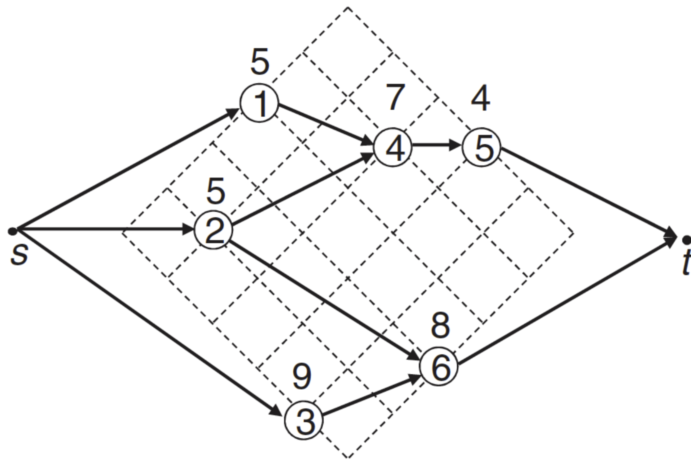


Figure (a)

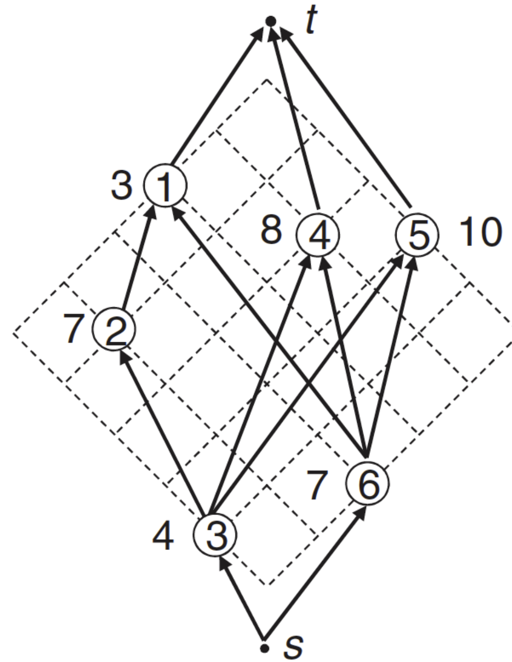


Figure (b)

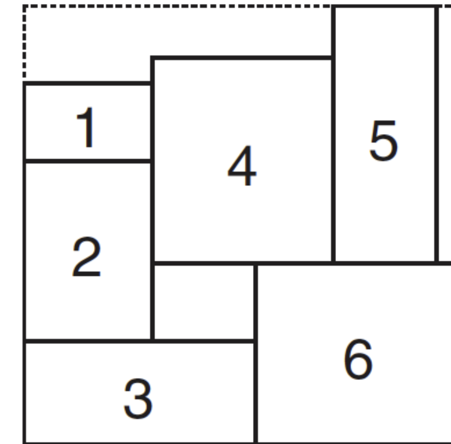


Figure (c)

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()
{
  // 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<int> accumulated_length(pos_seq_.size(), 0);
  for (int macro_id : pos_seq_) {
    const int neg_seq_pos = sequence_pair_pos[macro_id].second;

    T& macro = macros_[macro_id];

    if (!macro.isFixed()) {
      macro.setX(accumulated_length[neg_seq_pos]);
    }

    const int current_length = macro.getX() + macro.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];

  // 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;
  }

  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;
    T& macro = macros_[macro_id];

    if (!macro.isFixed()) {
      macro.setY(accumulated_length[neg_seq_pos]);
    }

    const int current_height = macro.getY() + macro.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 447 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 ecs.umass.edu/ece/labs/vlsicad/ece665/slides/SimulatedAnnealing.ppt

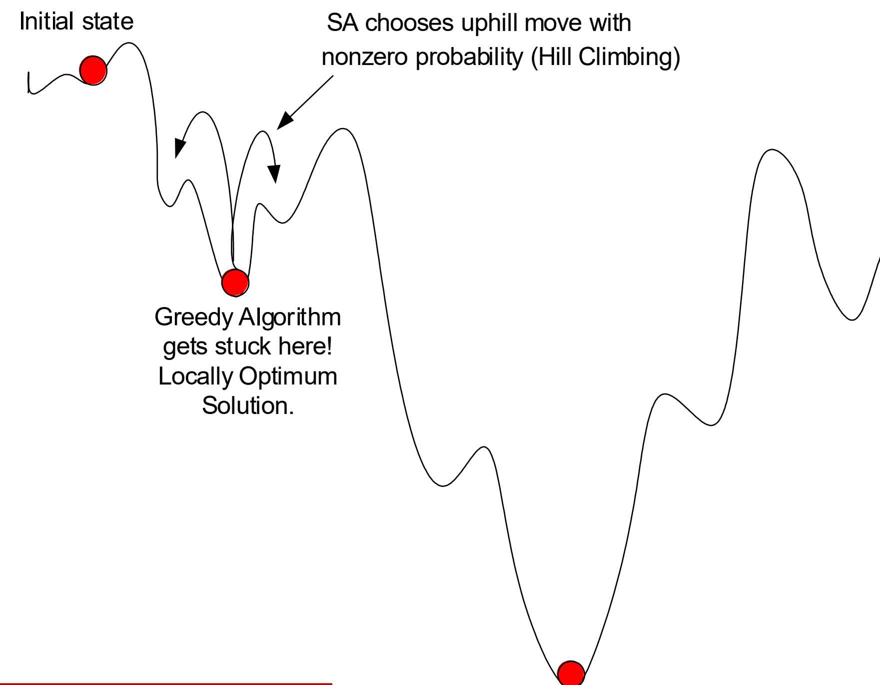
Algorithm SIMULATED-ANNEALING
Begin
    temp = INIT-TEMP;
    currentSol = INIT-SOLUTION;
    for i = 1 to M
        candidateSol = NEIGHBOR(*currentSol*);
            ΔC = COST(*candidateSol*) – COST(*currentSol*);
            if ($\Delta C < 0$) **then**
                currentSol = *candidateSol*;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

What happens when *temp* = $+\infty$?
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 … (e.g., make problem smaller and/or sparser – how?)]

- 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 $e^{(-cost(A)/temp)}$
  - 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]



SA converges to global opt solution with Pr = 1 (in limit of infinite time, infinitely slow cooling)

Best-so-far vs. where-you-are: implications for optimal finite-time annealing*

Kenneth D. Boese and Andrew B. Kahng
UCLA Computer Science Department, Los Angeles, CA 90024-1596, USA

Received 2 December 1992
Revised 25 April 1993

1. Introduction

Given a finite

Neighborhood Structure of Sequence Pair (to apply SA)

- 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)]

```
713  void SimulatedAnnealingCore<T>::fastSA()
714  {
715    float cost = calNormCost();
716    float pre_cost = cost;
717    int step = 1;
718    float temperature = init_temperature_;
719    const float min_t = 1e-10;
720    const float t_factor
721        = std::exp(std::log(min_t / init_temperature_) / max_num_step_);
722
723    updateBestResult(cost);
724
725    while (step <= max_num_step_) {
726      for (int i = 0; i < num_perturb_per_step_; i++) {
727        saveState();
728        perturb();
729        cost = calNormCost();
730
731        const bool is_valid = isValid();
732        if (!invalid_states_allowed_ && !is_valid) {
733          restoreState();
734          continue;
735        }
736
737        const bool found_new_best_result = cost < best_result_.cost;
738        if ((!is_best_result_valid_ || is_valid) && found_new_best_result) {
739          updateBestResult(cost);
740          is_best_result_valid_ = is_valid;
741        }
```

Line 713 in SimulatedAnnealingCore.cpp [(link)]

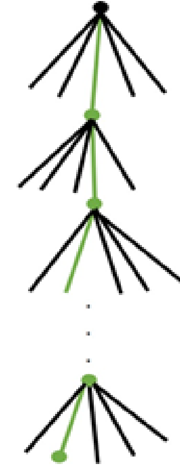
```
743        const float delta_cost = cost - pre_cost;
744        if (delta_cost <= 0) {
745          // always accept improvements
746          pre_cost = cost;
747        } else {
748          // probabilistically accept degradations for hill climbing
749          const float num = distribution_(generator_);
750          const float prob = std::exp(-delta_cost / temperature);
751          if (num < prob) {
752            pre_cost = cost;
753          } else {
754            restoreState();
755          }
756        }
757      }
758
759      temperature *= t_factor;
760      step++;
761
762      cost_list_.push_back(pre_cost);
763      T_list_.push_back(temperature);
764    }
765
766    packFloorplan();
767    if (graphics_) {
768      graphics_->doNotSkip();
769    }
770    calPenalty();
771    cost = calNormCost();
772
773    const bool found_new_best_result = cost < best_result_.cost;
774    if ((is_best_result_valid_ && !isValid()) || !found_new_best_result) {
775      useBestResult();
776    }
777  }
```

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.
```

Simulated Annealing for Macro Placement

- Simulated Annealing (SA) has recently received attention as a “strong baseline” for macro placers [that operate in a discrete, gridded ‘canvas’ of macro locations]
- For an example SA implementation for macro placement: see the MacroPlacement GitHub repository
  - Uses SA wrapped with **“go-with-the-winners” (GWTW)** metaheuristic
  - **SA actions**: (i) move, (ii) shuffle, (iii) shift, (iv) flip and (v) swap
- SA objective: **minimize** $\boldsymbol{proxy\ cost} = WL + 0.5 \times (cong. + den.)$
- **SA hyperparameters:**
  - Action probabilities
  - Initial and final temperature
  - Number of iterations ($\#iter$) and number of moves per iteration
  - Placement initialization: spiral and greedy
- **GWTW hyperparameters:**
  - #workers: number of SA workers
  - $sync_freq$: After $sync_freq \times \#iter$ iterations, SA workers synchronize
  - Top K solutions (proxy cost) are evenly replicated among SA workers

Reproducibility of SA Results

- SA is implemented using C++
- **Deterministic** outcome:
  - Use of seed ensures deterministic SA result
  - Assign different seed for each SA worker ensure variation
- Deterministic code yields **different outcomes** across C++ **versions/environments**
  - Reproducible only on same compute machine and same environment
- **Floating point** operations: varying results across CPUs
  - Lookup table for computing exponents ensures determinism
    → Lookup table helps speed up SA code
- **Reproducibility** across different platforms
  - Need to ensure same environment, same lookup table and same input
  - **Containerization**: Docker/Singularity images ensure same environment
  - Providing **run script** with lookup table ensures exact same SA input

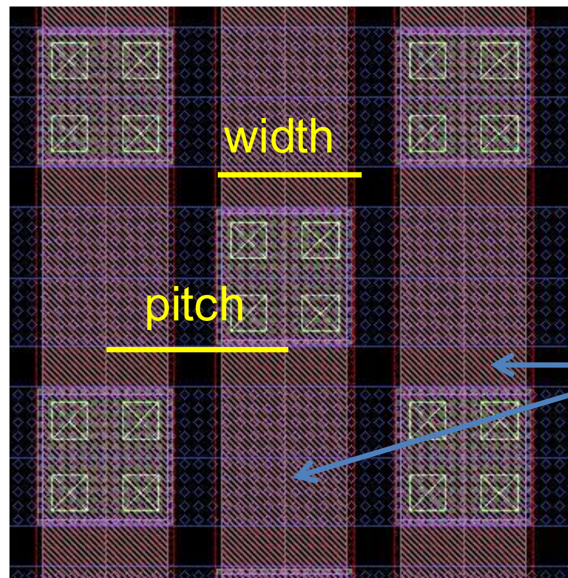
Power Grid Generation

Thanks to Dr. Peter Gadfort (LinkedIn)

Director of Silicon Engineering, Zero ASIC

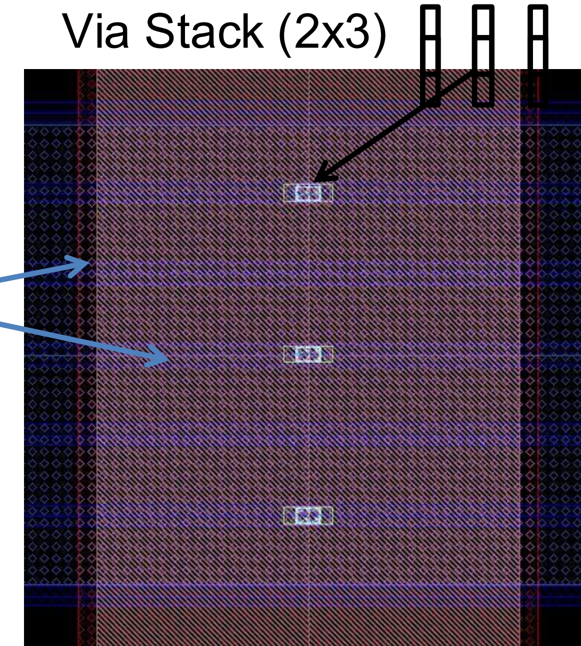
Power Grid Design and Analysis

- 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.



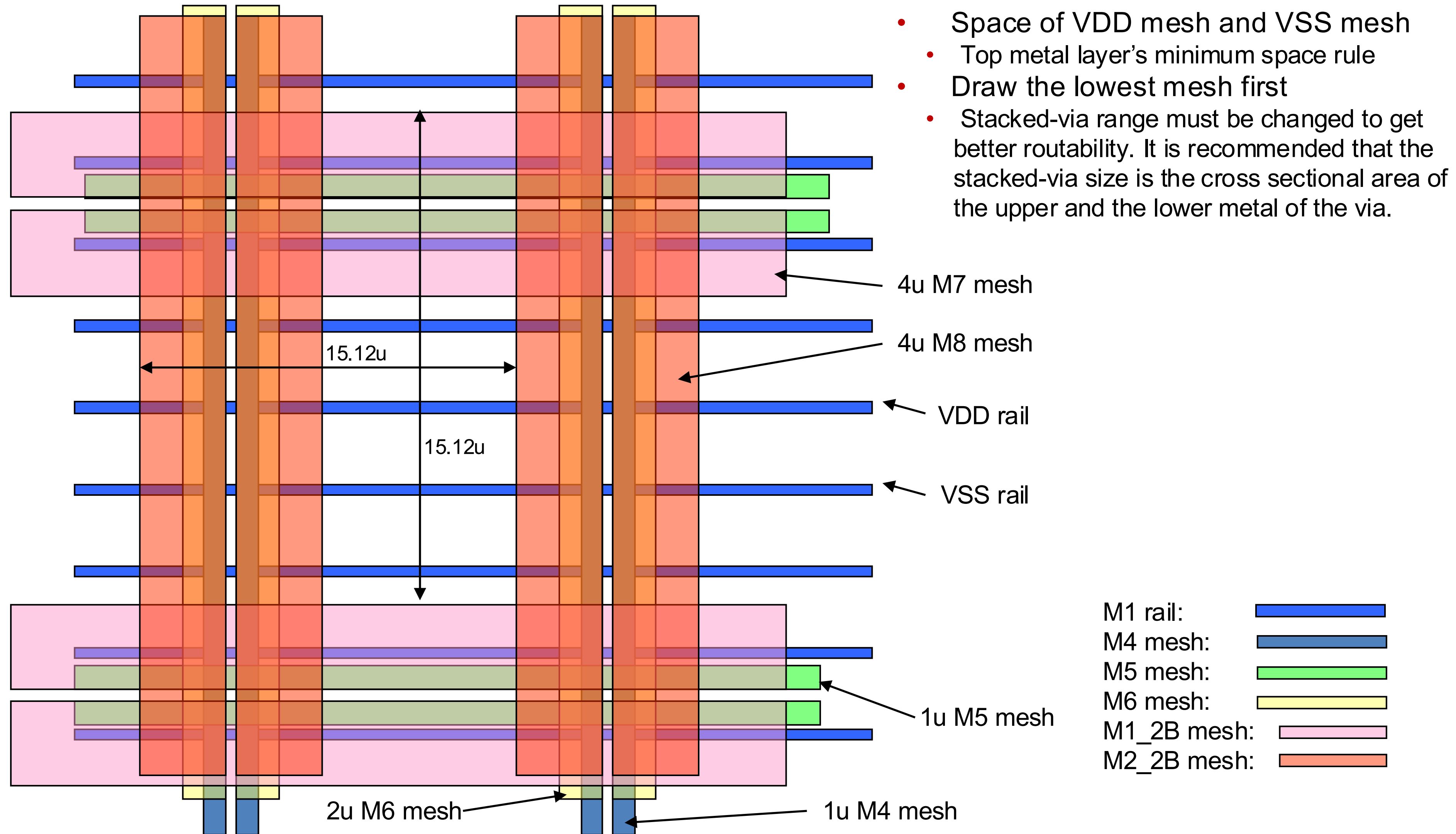
Power Rails

Power Stripes



Power Grid Design and Analysis

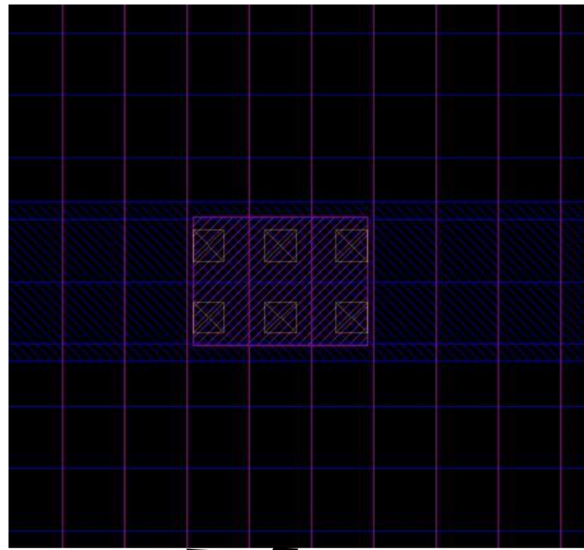
- Structure of the multi-layer power mesh (You can see detail process using slide-show)



- 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.

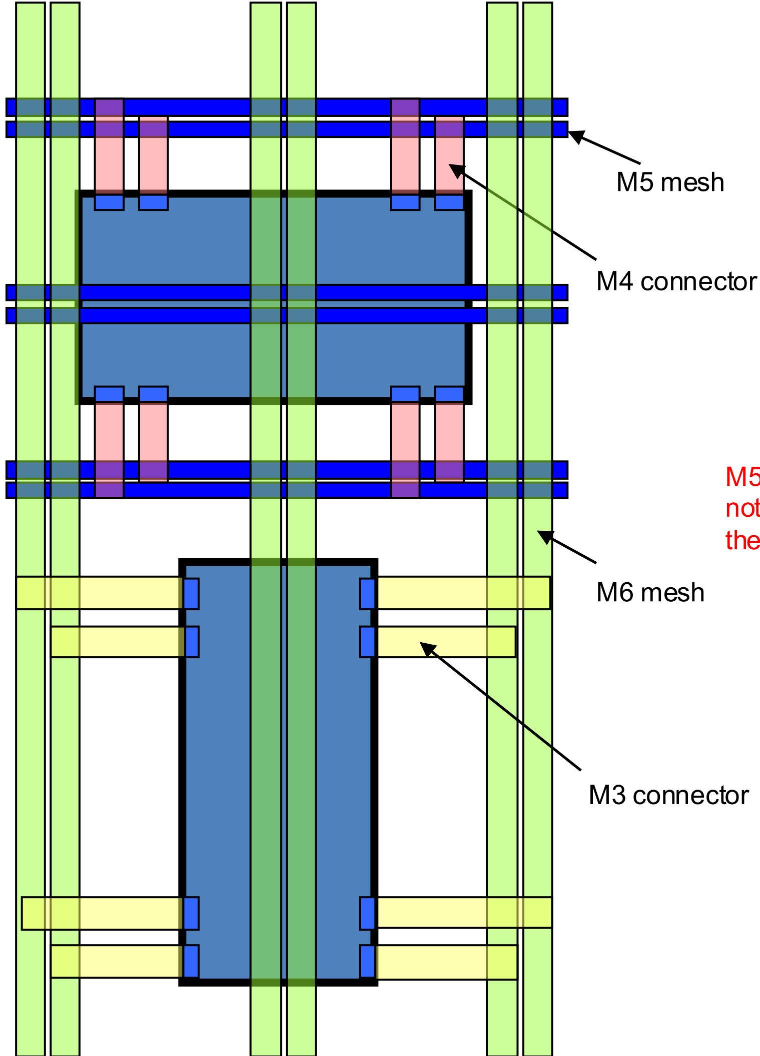
Power Grid Design and Analysis

- 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.

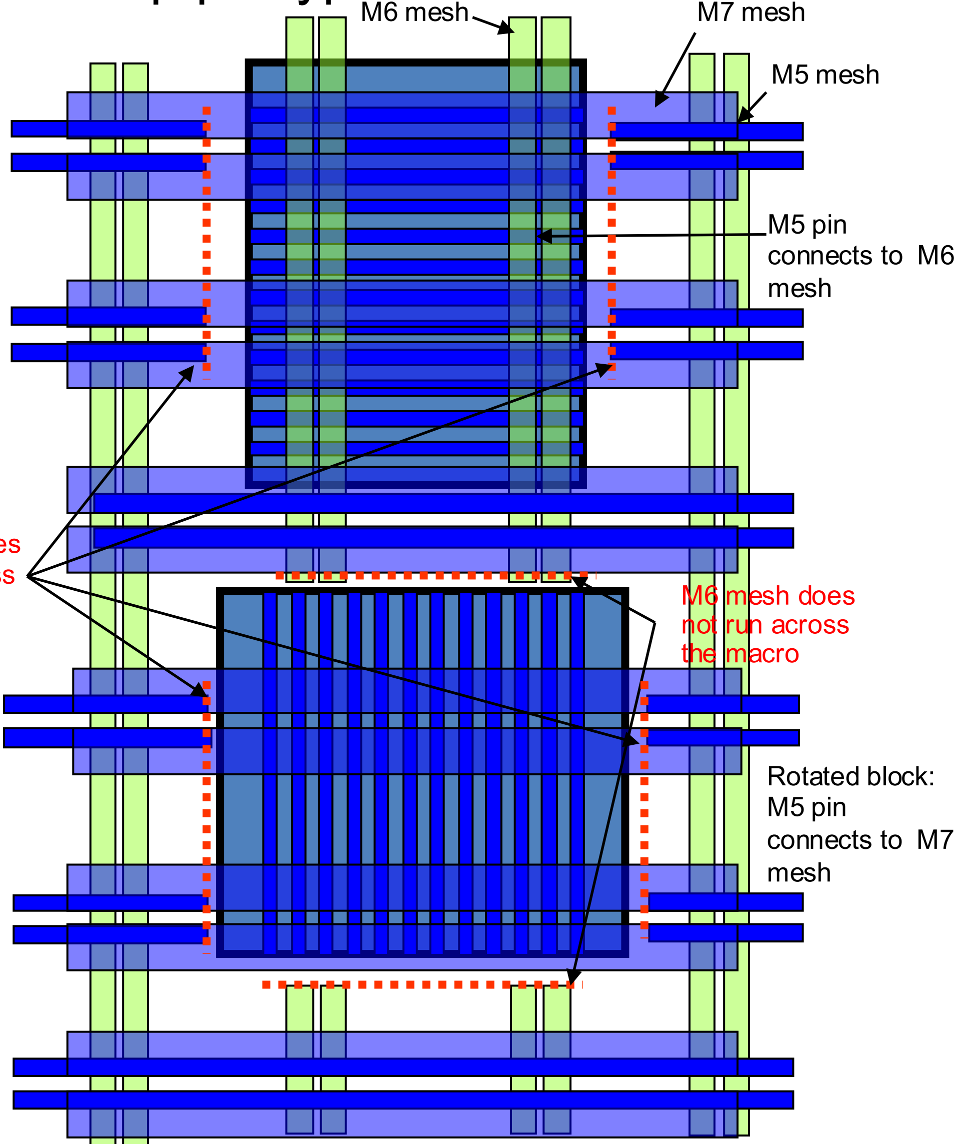


Macro Connection Cases: Side, Top, Rotated…

- Side pin type



- Top pin type

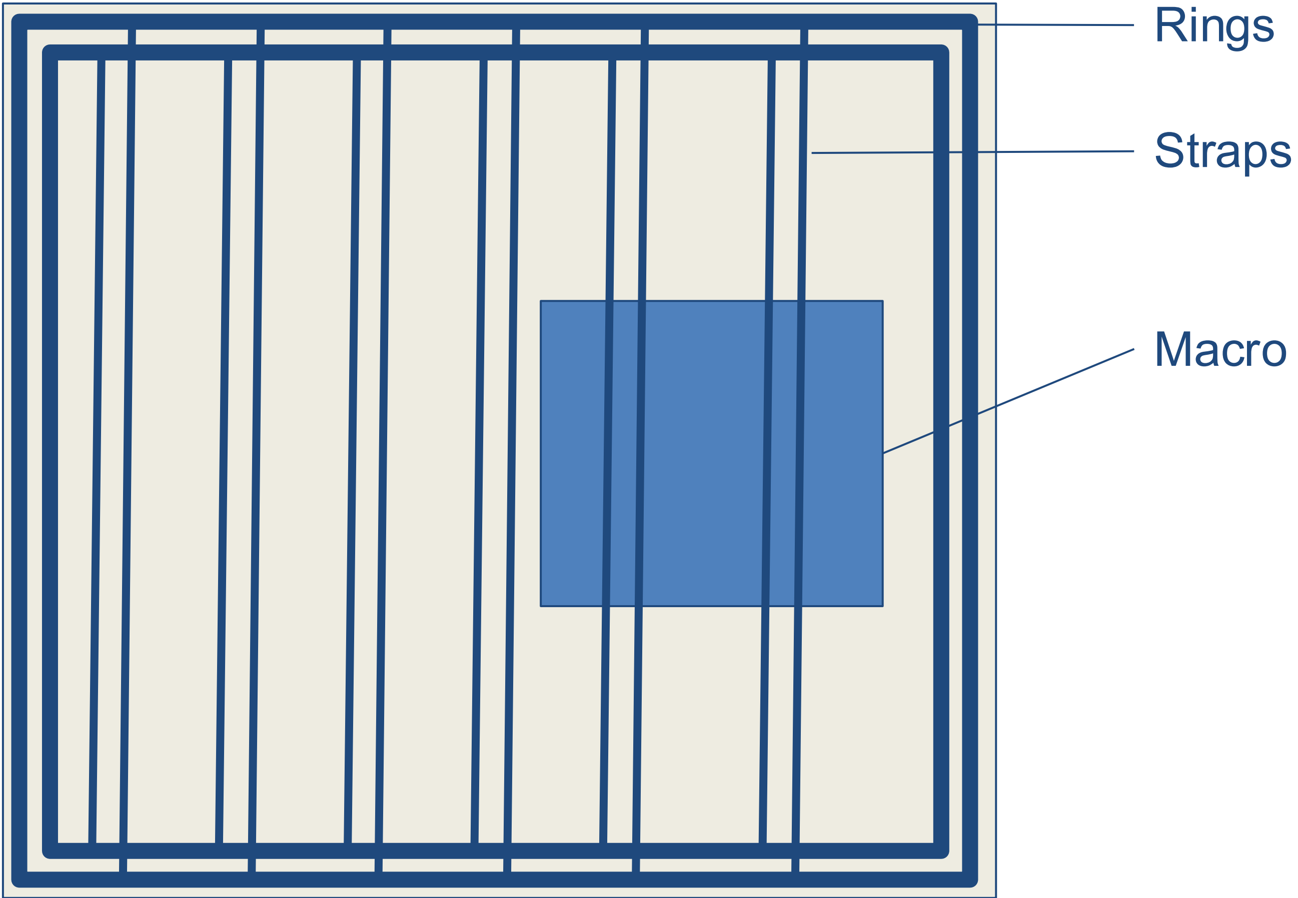


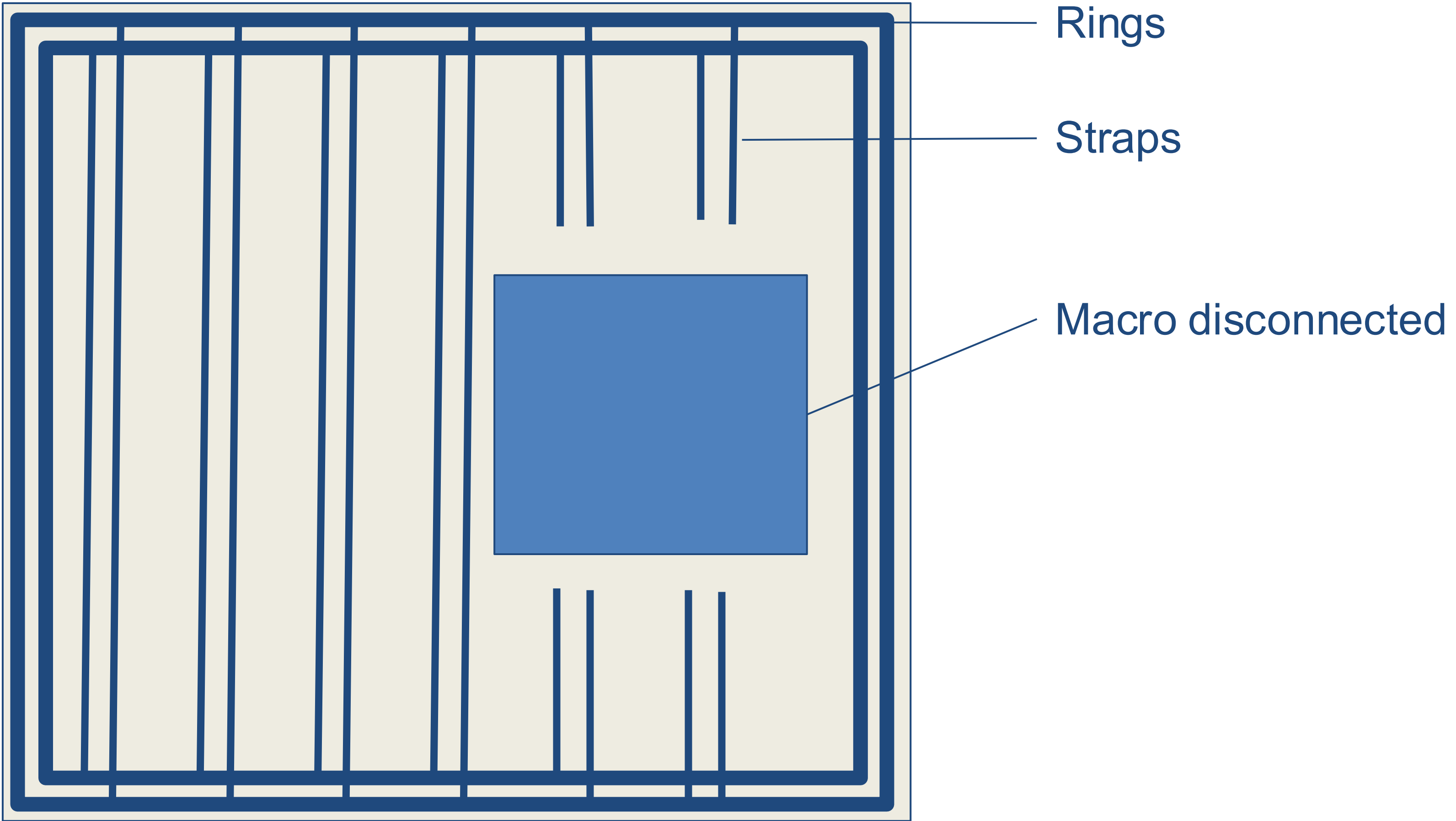
pdn (aka *pdngen*) User Interface

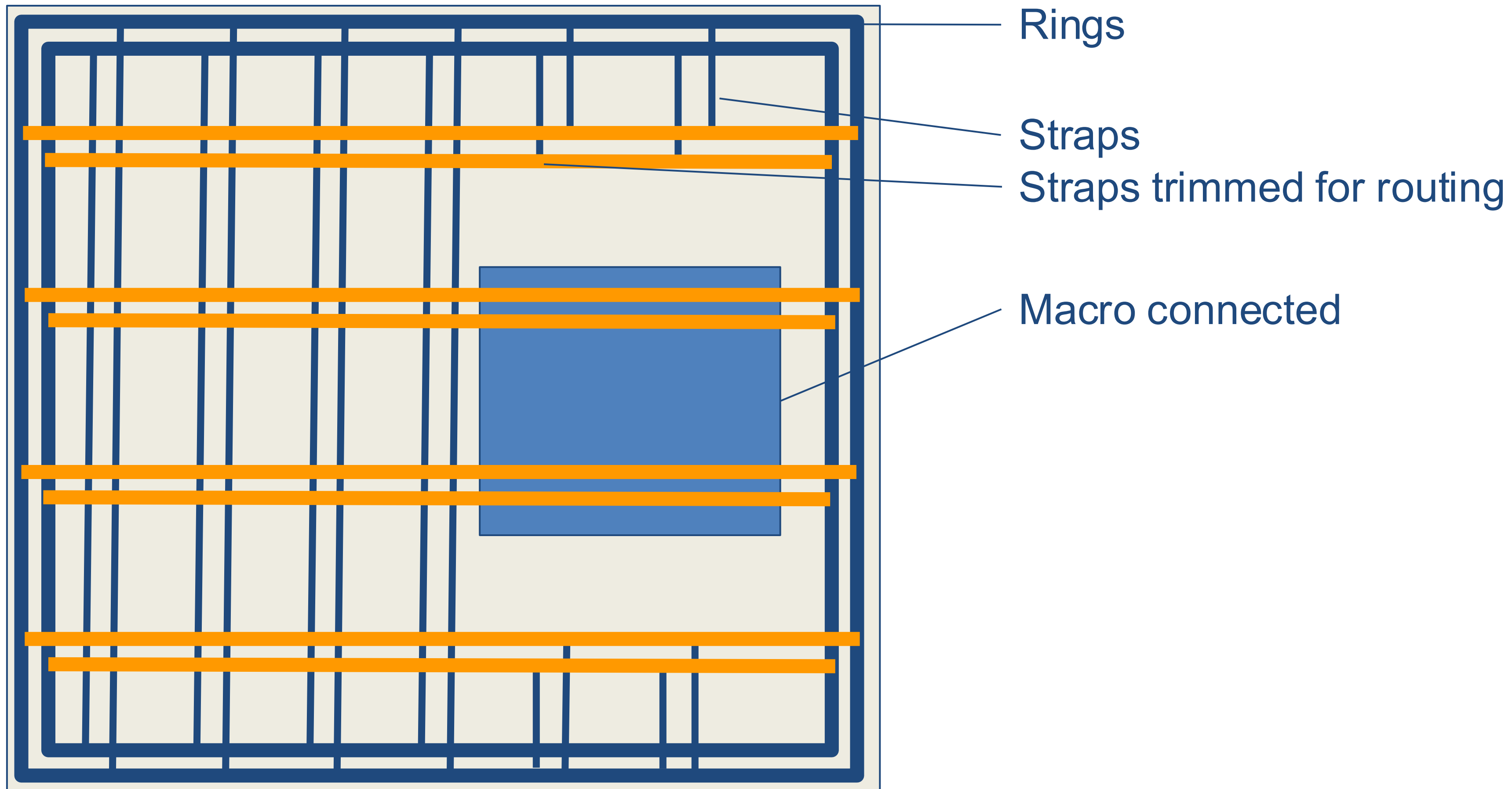
- `set_voltage_domain` - define a new power domain
- `define_pdn_grid` - defines a new grid
- `add_pdn_stripe` - add straps and followpins
- `add_pdn_ring` - add rings
- `add_pdn_connect` - add layer connectivity
- `pdngen` - create grids

Main PDN Build Steps

1. Extract existing routing and blockages
https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/PdnGen.cc#L91
2. For each grid:
https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/PdnGen.cc#L118
    a. Create rings, if requested
    https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/rings.cpp#L196
    b. Create straps
    https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/straps.cpp#L157
    c. Check connectivity and do channel repair (slide)
3. Estimate vias
https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/PdnGen.cc#L130
4. Trim excess
https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/PdnGen.cc#L133
5. Commit to database
https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/PdnGen.cc#L742
    a. Create all vias DRC free
    https://github.com/The-OpenROAD-Project/OpenROAD/blob/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/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/52ff2a5ea5814dc671c1cf7c4b950f840b6a4e88/src/pdn/src/straps.cpp#L1623
- 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 (SP25)

- Automatic connectivity rules (add_pdn_connect) *AI/ML opportunity!*
  - 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) *or sparsification* *AI/ML opportunity!*
  - 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) ***AI/ML opportunity !***
  - 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)

Food for thought: Why is a dense (sparse) power mesh helpful? And why is it harmful? What are some implications (good and bad) of "backside power delivery" at 2nm and below?

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

$$x_j = x_i$$