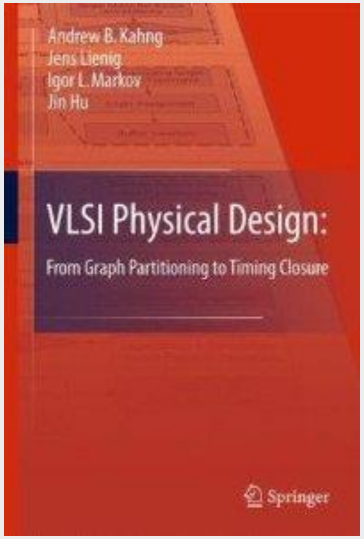
VLSI Physical Design: From Graph Partitioning to Timing Closure

Chapter 2 – Netlist and System Partitioning



Andrew B. Kahng
Jens Lienig
Igor L. Markov
Jin Hu

VLSI Physical Design:

From Graph Partitioning to Timing Closure

Springer

Original Authors:

Andrew B. Kahng, Jens Lienig, Igor L. Markov, Jin Hu

Chapter 2 – Netlist and System Partitioning

2.1 Introduction

2.2 Terminology

2.3 Optimization Goals

2.4 Partitioning Algorithms

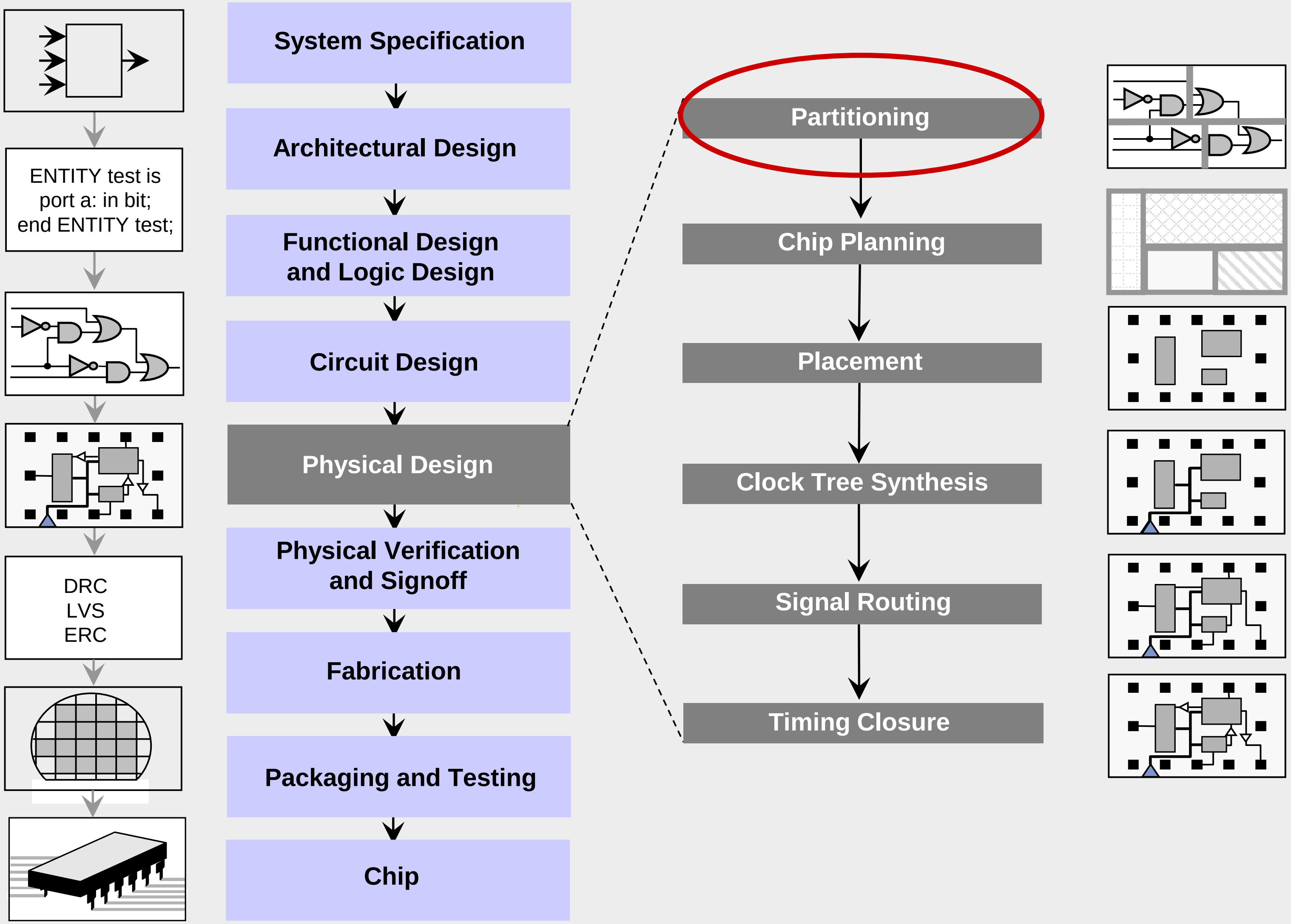
- 2.4.1 Kernighan-Lin (KL) Algorithm
- 2.4.2 Extensions of the Kernighan-Lin Algorithm
- 2.4.3 Fiduccia-Mattheyses (FM) Algorithm

2.5 Framework for Multilevel Partitioning

- 2.5.1 Clustering
- 2.5.2 Multilevel Partitioning

2.6 System Partitioning onto Multiple FPGAs

2.1 Introduction



ENTITY test is
port a: in bit;
end ENTITY test;

DRC
LVS
ERC

System Specification

Architectural Design

Functional Design and Logic Design

Circuit Design

Physical Design

Physical Verification and Signoff

Fabrication

Packaging and Testing

Chip

Partitioning

Chip Planning

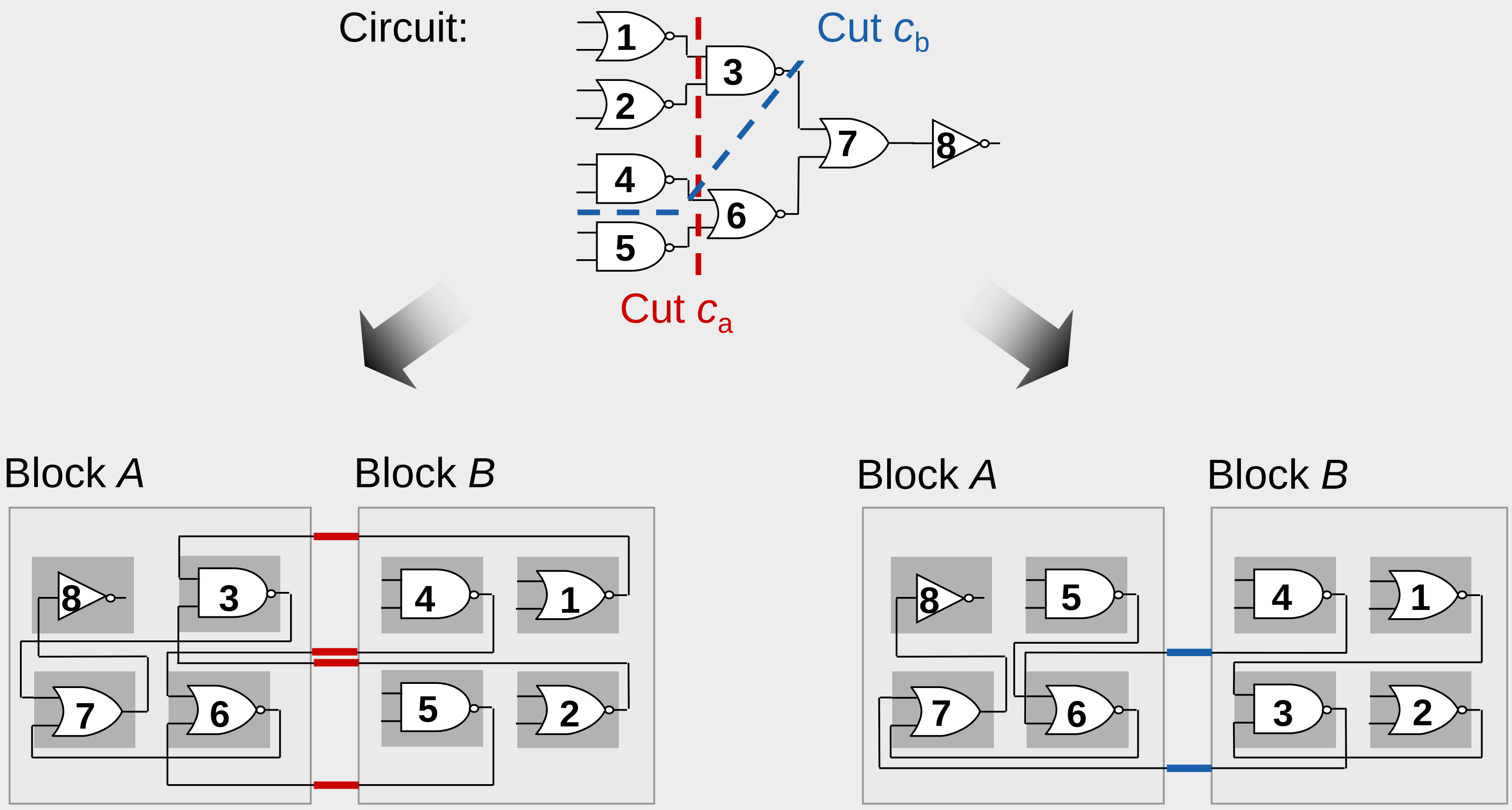
Placement

Clock Tree Synthesis

Signal Routing

Timing Closure

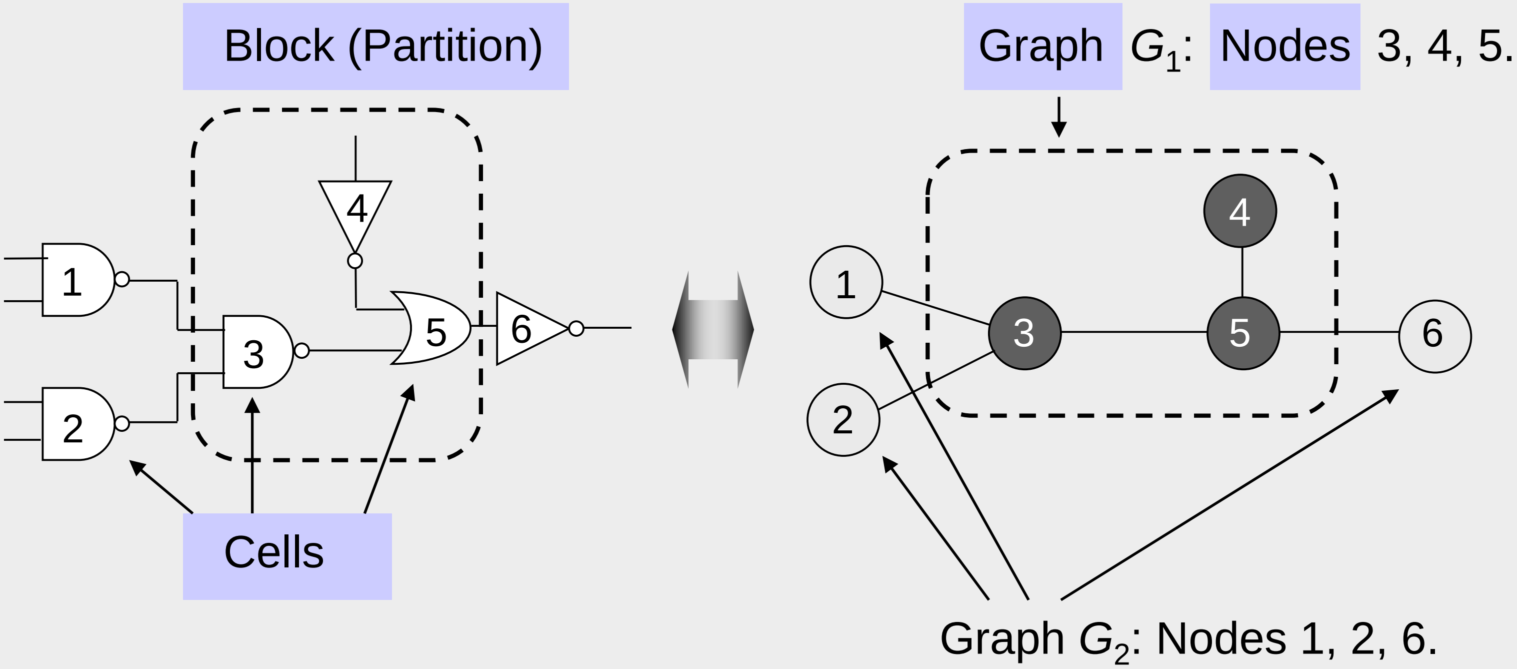
2.1 Introduction



Cut c_a: four external connections

Cut c_b: two external connections

2.2 Terminology



Block (Partition)

Cells

Graph G_1: Nodes 3, 4, 5.

Graph G_2: Nodes 1, 2, 6.

Collection of cut edges

Cut set: (1,3), (2,3), (5,6),

2.3 Optimization Goals

- Given a graph $G(V,E)$ with $|V|$ nodes and $|E|$ edges where each node $v \in V$ and each edge $e \in E$.

- Each node has area $s(v)$ and each edge has cost or weight $w(e)$.

- The objective is to divide the graph G into k disjoint subgraphs such that all optimization goals are achieved and all original edge relations are respected.

2.3 Optimization Goals

- In detail, what are the optimization goals?
  - Number of connections between partitions is minimized
  - Each partition meets all design constraints (size, number of external connections..)
  - Balance every partition as well as possible

- How can we meet these goals?
  - Unfortunately, this problem is NP-hard
  - Efficient heuristics are developed in the 1970s and 1980s.
    They are high quality and in low-order polynomial time.

Chapter 2 – Netlist and System Partitioning

2.1 Introduction

2.2 Terminology

2.3 Optimization Goals

2.4 Partitioning Algorithms

2.4.1 Kernighan-Lin (KL) Algorithm

2.4.2 Extensions of the Kernighan-Lin Algorithm

2.4.3 Fiduccia-Mattheyses (FM) Algorithm

2.5 Framework for Multilevel Partitioning

2.5.1 Clustering

2.5.2 Multilevel Partitioning

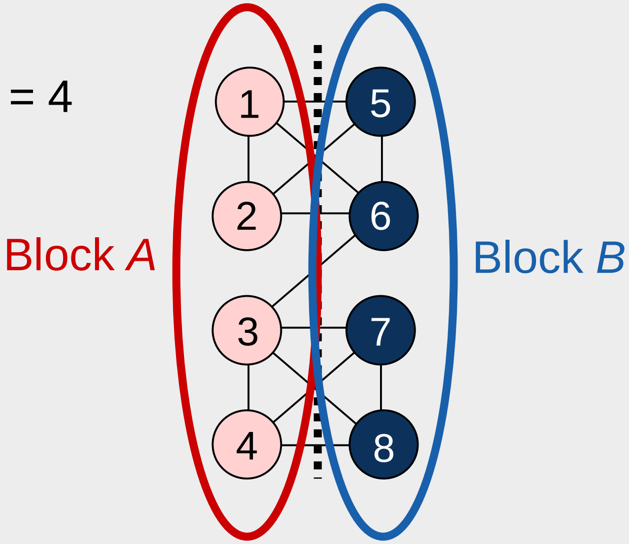
2.6 System Partitioning onto Multiple FPGAs

2.4.1 Kernighan-Lin (KL) Algorithm

Given: A graph with 2*n* nodes where each node has the same weight.

Goal: A partition (division) of the graph into two disjoint subsets *A* and *B* with minimum cut cost and |*A*| = |*B*| = *n*.

Example: *n* = 4



2.4.1 Kernighan-Lin (KL) Algorithm – Terminology

Cost $D(v)$ of moving a node v

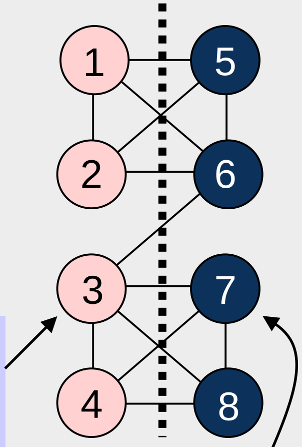
$$D(v) = |E_c(v)| - |E_{nc}(v)| \;,$$

where

$E_c(v)$ is the set of v's incident edges that are cut by the cut line, and

$E_{nc}(v)$ is the set of v's incident edges that are not cut by the cut line.

High costs ($D > 0$) indicate that the node should move, while low costs ($D < 0$) indicate that the node should stay within the same partition.

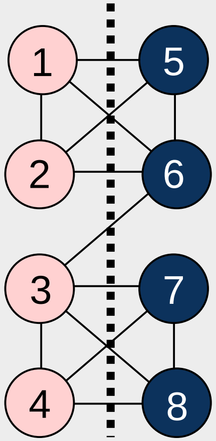


Node 3:
$D(3)$ = 3-1=2

Node 7:
$D(7)$ = 2-1=1

2.4.1 Kernighan-Lin (KL) Algorithm – Terminology

Gain of swapping a pair of nodes *a* und *b*



$\Delta g = D(a) + D(b) - 2 * c(a,b),$

where

- $D(a)$, $D(b)$ are the respective costs of nodes *a*, *b*
- $c(a,b)$ is the connection weight between *a* and *b*:
  If an edge exists between *a* and *b*,
  then $c(a,b)$ = edge weight (here 1),
  otherwise, $c(a,b) = 0$.

The gain Δg indicates how useful the swap between two nodes will be

The larger Δg, the more the total cut cost will be reduced

2.4.1 Kernighan-Lin (KL) Algorithm – Terminology

Gain of swapping a pair of nodes *a* und *b*

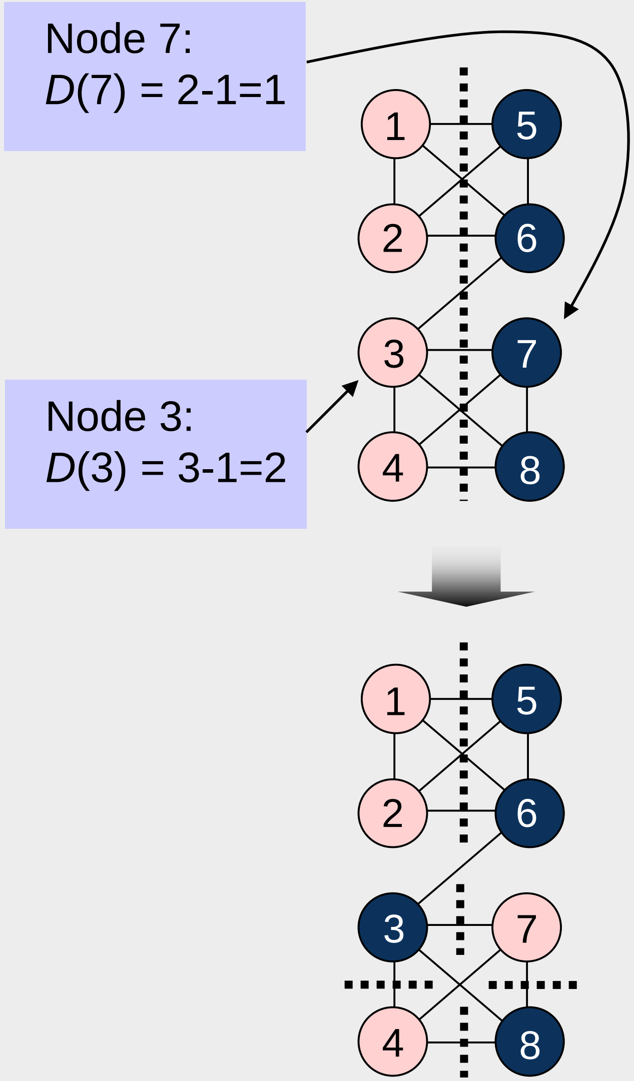
$$\Delta g = D(a) + D(b) - 2 * c(a,b),$$

where

- *D*(*a*), *D*(*b*) are the respective costs of nodes *a*, *b*
- *c*(*a*,*b*) is the connection weight between *a* and *b*:
  If an edge exists between *a* and *b*,
  then *c*(*a*,*b*) = edge weight (here 1),
  otherwise, *c*(*a*,*b*) = 0.

$$\Delta g\ (3,7) = D(3) + D(7) - 2 * c(a,b) = 2 + 1 – 2 = 1$$

=> Swapping nodes 3 and 7 would reduce the cut size by 1


Node 7:
D(7) = 2-1=1

Node 3:
D(3) = 3-1=2

2.4.1 Kernighan-Lin (KL) Algorithm – Terminology

Gain of swapping a pair of nodes *a* und *b*

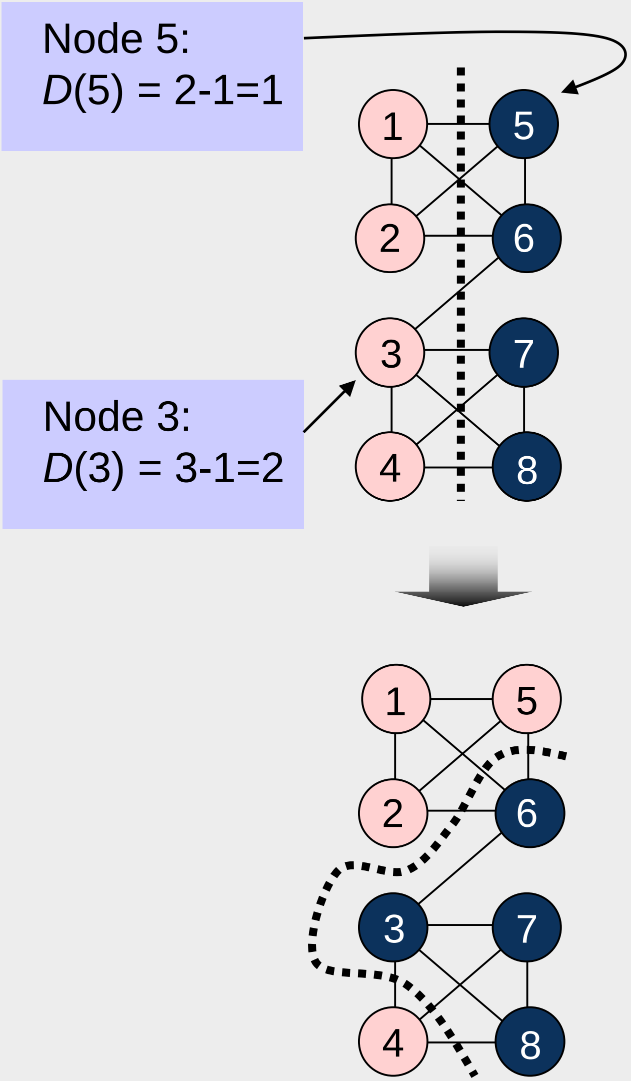
$$\Delta g = D(a) + D(b) - 2 * c(a,b),$$

where

- *D*(*a*), *D*(*b*) are the respective costs of nodes *a*, *b*
- *c*(*a*,*b*) is the connection weight between *a* and *b*:
  If an edge exists between *a* and *b*,
  then *c*(*a*,*b*) = edge weight (here 1),
  otherwise, *c*(*a*,*b*) = 0.

$$\Delta g\ (3,5) = D(3) + D(5) - 2 * c(a,b) = 2 + 1 – 0 = 3$$

=> Swapping nodes 3 and 5 would reduce the cut size by 3



Node 5:
D(5) = 2-1=1

Node 3:
D(3) = 3-1=2

2.4.1 Kernighan-Lin (KL) Algorithm – Terminology

Gain of swapping a pair of nodes *a* und *b*

The goal is to find a pair of nodes *a* and *b* to exchange such that Δg is maximized and swap them.

2.4.1 Kernighan-Lin (KL) Algorithm – Terminology

Maximum positive gain G_m of a pass

The maximum positive gain G_m corresponds to the best prefix of *m* swaps within the swap sequence of a given pass.

These *m* swaps lead to the partition with the minimum cut cost encountered during the pass.

G_m is computed as the sum of Δg values over the first *m* swaps of the pass, with *m* chosen such that G_m is maximized.

$$G_m = \sum_{i=1}^{m} \Delta g_i$$

2.4.1 Kernighan-Lin (KL) Algorithm – One pass

Step 0:

- $V = 2n$ nodes
- $\{A, B\}$ is an initial arbitrary partitioning

Step 1:

- $i = 1$
- Compute $D(v)$ for all nodes $v \in V$

Step 2:

- Choose a_i and b_i such that $\Delta g_i = D(a_i) + D(b_i) - 2 * c(a_i b_i)$ is maximized
- Swap and fix a_i and b_i

Step 3:

- If all nodes are fixed, go to Step 4. Otherwise
- Compute and update D values for all nodes that are connected to a_i and b_i and are not fixed.
- $i = i + 1$
- Go to Step 2

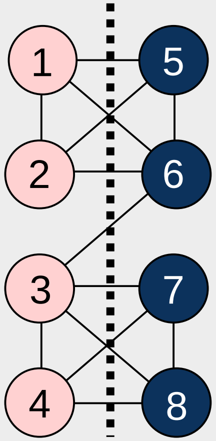
Step 4:

- Find the move sequence 1...m ($1 \leq m \leq i$), such that $G_m = \sum_{i=1}^{m} \Delta g_i$ is maximized
- If $G_m > 0$, go to Step 5. Otherwise, END

Step 5:

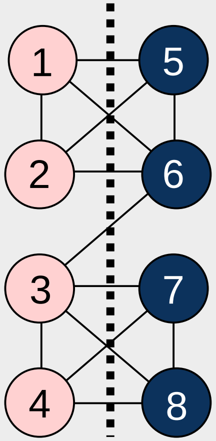
- Execute m swaps, reset remaining nodes
- Go to Step 1

2.4.1 Kernighan-Lin (KL) Algorithm – Example



Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

2.4.1 Kernighan-Lin (KL) Algorithm – Example



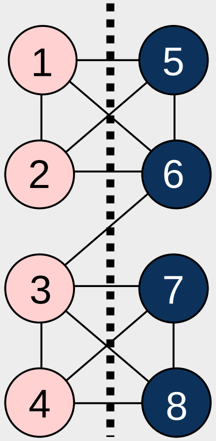
Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Costs *D*(*v*) of each node:

D(1) = 1	*D*(5) = 1
D(2) = 1	*D*(6) = 2
D(3) = 2	*D*(7) = 1
D(4) = 1	*D*(8) = 1

Nodes that lead to maximum gain

2.4.1 Kernighan-Lin (KL) Algorithm – Example



Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Costs $D(v)$ of each node:

$D(1) = 1$	$\boldsymbol{D(5) = 1}$
$D(2) = 1$	$D(6) = 2$
$\boldsymbol{D(3) = 2}$	$D(7) = 1$
$D(4) = 1$	$D(8) = 1$

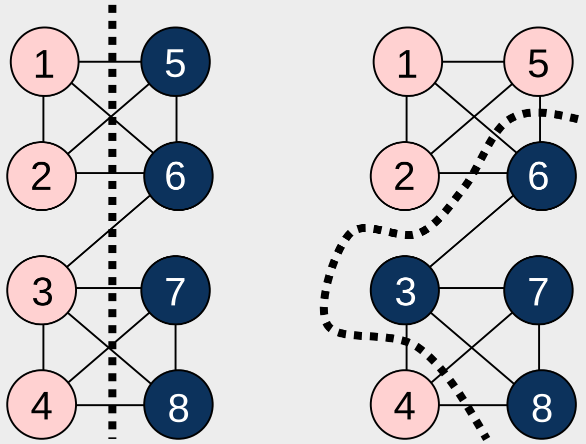
$\Delta g_1 = 2+1-0 = 3$
Swap (3,5)
$G_1 = \Delta g_1 = 3$

Nodes that lead to maximum gain

Gain after node swapping

Gain in the current pass

2.4.1 Kernighan-Lin (KL) Algorithm – Example



Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

$D(1) = 1$ **$D(5) = 1$**
$D(2) = 1$ $D(6) = 2$
$D(3) = 2$ $D(7) = 1$
$D(4) = 1$ $D(8) = 1$

Nodes that lead to maximum gain

$\Delta g_1 = 2+1-0 = 3$

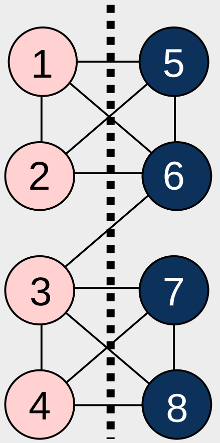
Gain after node swapping

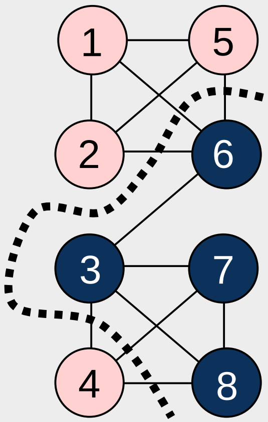
Swap (3,5)

$G_1 = \Delta g_1 = 3$

Gain in the current pass

2.4.1 Kernighan-Lin (KL) Algorithm – Example





Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Cut cost: 6
Not fixed:
1,2,4,6,7,8

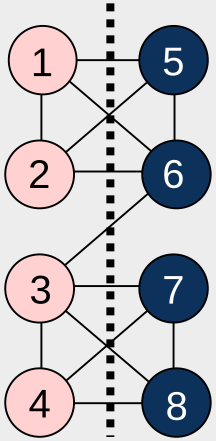
$D(1) = 1$	$\boldsymbol{D(5) = 1}$
$D(2) = 1$	$D(6) = 2$
$\boldsymbol{D(3) = 2}$	$D(7) = 1$
$D(4) = 1$	$D(8) = 1$

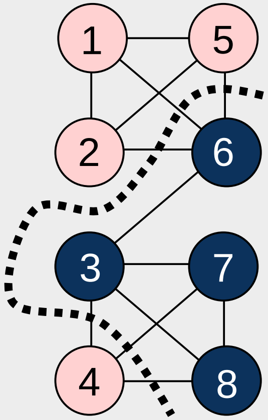
$\Delta g_1 = 2+1-0 = 3$

Swap (3,5)

$G_1 = \Delta g_1 = 3$

2.4.1 Kernighan-Lin (KL) Algorithm – Example





Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Cut cost: 6
Not fixed:
1,2,4,6,7,8

$D(1) = 1$	**$D(5) = 1$**
$D(2) = 1$	$D(6) = 2$
$D(3) = 2$	$D(7) = 1$
$D(4) = 1$	$D(8) = 1$

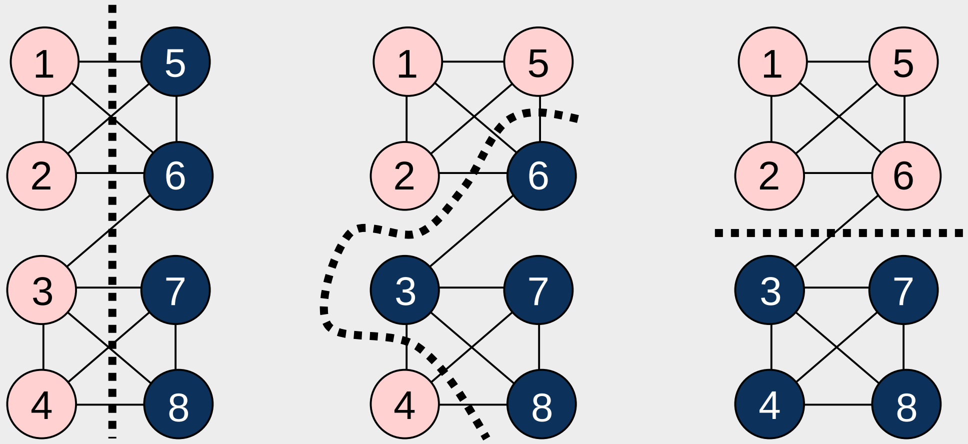
$\Delta g_1 = 2+1-0 = 3$

Swap (3,5)

$G_1 = \Delta g_1 = 3$

$D(1) = -1$	**$D(6) = 2$**
$D(2) = -1$	$D(7)=-1$
$D(4) = 3$	$D(8)=-1$

2.4.1 Kernighan-Lin (KL) Algorithm – Example



Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Cut cost: 6
Not fixed:
1,2,4,6,7,8

$D(1) = 1$	**$D(5) = 1$**
$D(2) = 1$	$D(6) = 2$
$D(3) = 2$	$D(7) = 1$
$D(4) = 1$	$D(8) = 1$

$\Delta g_1 = 2+1-0 = 3$
Swap (3,5)
$G_1 = \Delta g_1 = 3$

$D(1) = -1$	**$D(6) = 2$**
$D(2) = -1$	$D(7)=-1$
$D(4) = 3$	$D(8)=-1$

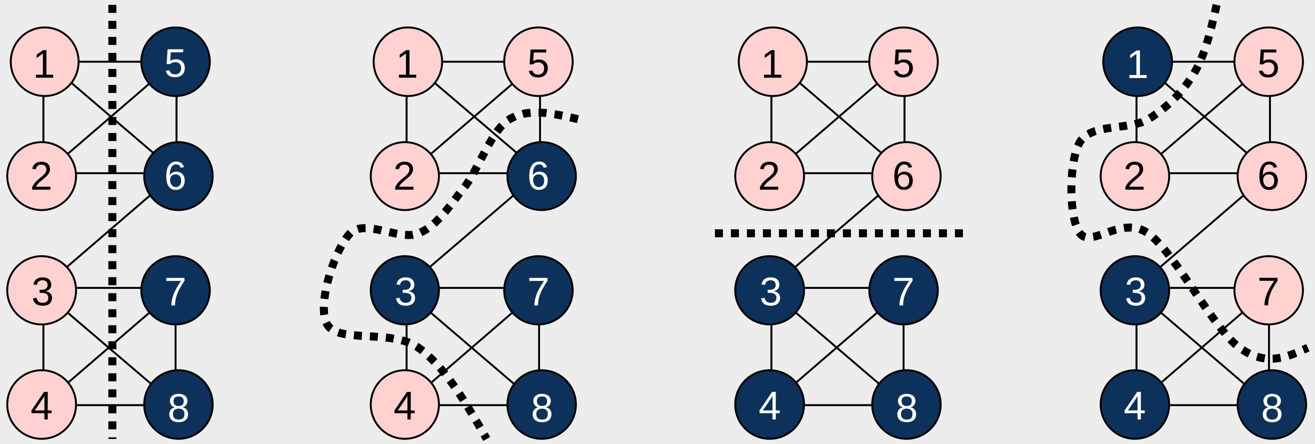
$\Delta g_2 = 3+2-0 = 5$
Swap (4,6)
$G_2 = G_1+\Delta g_2 = 8$

Nodes that lead to maximum gain

Gain after node swapping

Gain in the current pass

2.4.1 Kernighan-Lin (KL) Algorithm – Example



Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Cut cost: 6
Not fixed:
1,2,4,6,7,8

Cut cost: 1
Not fixed:
1,2,7,8

Cut cost: 7
Not fixed:
2,8

$D(1) = 1$ **$D(5) = 1$**
$D(2) = 1$ $D(6) = 2$
$D(3) = 2$ $D(7) = 1$
$D(4) = 1$ $D(8) = 1$

$\Delta g_1 = 2+1-0 = 3$
Swap (3,5)
$G_1 = \Delta g_1 = 3$

$D(1) = -1$ **$D(6) = 2$**
$D(2) = -1$ $D(7)=-1$
$D(4) = 3$ $D(8)=-1$

$\Delta g_2 = 3+2-0 = 5$
Swap (4,6)
$G_2 = G_1+\Delta g_2 = 8$

$D(1) = -3$ **$D(7)=-3$**
$D(2) = -3$ $D(8)=-3$

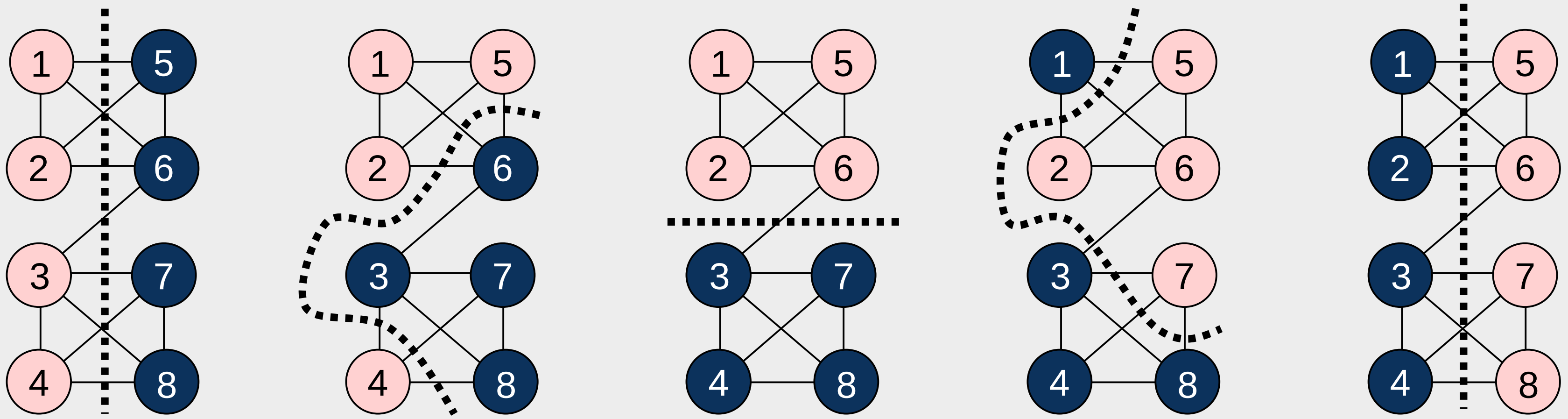
$\Delta g_3 = -3-3-0 = -6$
Swap (1,7)
$G_3 = G_2 + \Delta g_3 = 2$

Nodes that lead to maximum gain

Gain after node swapping

Gain in the current pass

2.4.1 Kernighan-Lin (KL) Algorithm – Example



Cut cost: 9
Not fixed:
1,2,3,4,5,6,7,8

Cut cost: 6
Not fixed:
1,2,4,6,7,8

Cut cost: 1
Not fixed:
1,2,7,8

Cut cost: 7
Not fixed:
2,8

Cut cost: 9
Not fixed:
–

$D(1) = 1$ **$D(5) = 1$**
$D(2) = 1$ $D(6) = 2$
$D(3) = 2$ $D(7) = 1$
$D(4) = 1$ $D(8) = 1$

$\Delta g_1 = 2+1-0 = 3$
Swap (3,5)
$G_1 = \Delta g_1 = 3$

$D(1) = -1$ **$D(6) = 2$**
$D(2) = -1$ $D(7)=-1$
$D(4) = 3$ $D(8)=-1$

$\Delta g_2 = 3+2-0 = 5$
Swap (4,6)
$G_2 = G_1+\Delta g_2 = 8$

$D(1) = -3$ **$D(7)=-3$**
$D(2) = -3$ $D(8)=-3$

$\Delta g_3 = -3-3-0 = -6$
Swap (1,7)
$G_3 = G_2 + \Delta g_3 = 2$

$D(2) = -1$ **$D(8)=-1$**

$\Delta g_4 = -1-1-0 = -2$
Swap (2,8)
$G_4 = G_3 + \Delta g_4 = 0$

2.4.1 Kernighan-Lin (KL) Algorithm – Example

$D(1) = 1$ **$D(5) = 1$**
$D(2) = 1$ $D(6) = 2$
$D(3) = 2$ $D(7) = 1$
$D(4) = 1$ $D(8) = 1$

$\Delta g_1 = 2+1-0 = 3$
Swap (3,5)
$G_1 = \Delta g_1 = 3$

$D(1) = -1$ **$D(6) = 2$**
$D(2) = -1$ $D(7)=-1$
$D(4) = 3$ $D(8)=-1$

$\Delta g_2 = 3+2-0 = 5$
Swap (4,6)
$G_2 = G_1+\Delta g_2 = 8$

$D(1) = -3$ **$D(7)=-3$**
$D(2) = -3$ $D(8)=-3$

$\Delta g_3 = -3-3-0 = -6$
Swap (1,7)
$G_3 = G_2 + \Delta g_3 = 2$

$D(2) = -1$ **$D(8)=-1$**

$\Delta g_4 = -1-1-0 = -2$
Swap (2,8)
$G_4 = G_3 + \Delta g_4 = 0$

Maximum positive gain $G_m = 8$ with $m = 2$.

2.4.1 Kernighan-Lin (KL) Algorithm – Example

$D(1) = 1$ **$D(5) = 1$**
$D(2) = 1$ $D(6) = 2$
$D(3) = 2$ $D(7) = 1$
$D(4) = 1$ $D(8) = 1$

$\Delta g_1 = 2+1-0 = 3$
Swap (3,5)
$G_1 = \Delta g_1 = 3$

$D(1) = -1$ **$D(6) = 2$**
$D(2) = -1$ $D(7)=-1$
$D(4) = 3$ $D(8)=-1$

$\Delta g_2 = 3+2-0 = 5$
Swap (4,6)
$G_2 = G_1+\Delta g_2 =8$

$D(1) = -3$ **$D(7)=-3$**
$D(2) = -3$ $D(8)=-3$

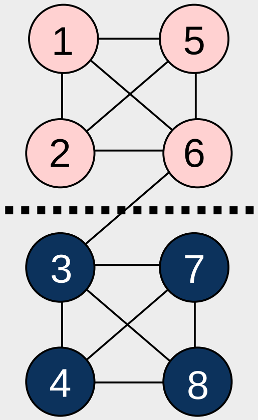
$\Delta g_3 = -3-3-0 = -6$
Swap (1,7)
$G_3= G_2 +\Delta g_3 = 2$

$D(2) = -1$ **$D(8)=-1$**

$\Delta g_4 = -1-1-0 = -2$
Swap (2,8)
$G_4 = G_3 +\Delta g_4 = 0$

Maximum positive gain $G_m = 8$ with $m = 2$.

Since $G_m > 0$, the first $m = 2$ swaps
(3,5) and (4,6) are executed.



Since $G_m > 0$, more passes are needed until
$G_m \leq 0$.

2.4.2 Extensions of the Kernighan-Lin (KL) Algorithm

- Unequal partition sizes
  - Apply the KL algorithm with only min(|*A*|,|*B*|) pairs swapped

- Unequal node weights
  - Try to rescale weights to integers, e.g., as multiples of *the greatest common divisor* of all node weights
  - Maintain area balance or allow a *one-move deviation* from balance

- *k*-way partitioning (generating *k* partitions)
  - Apply the KL two-way partitioning algorithm to all possible pairs of partitions
  - Recursive partitioning (convenient when k is a power of two)
  - Direct k-way extensions exist

2.4.3 Fiduccia-Mattheyses (FM) Algorithm

- Single cells are moved independently instead of swapping pairs of cells --- cannot and do not need to maintain exact partition balance
  - The area of each individual cell is taken into account
  - Applicable to partitions of unequal size
    and in the presence of initially fixed cells

- Cut costs are extended to include hypergraphs
  - nets with 2+ pins
- While the KL algorithm aims to minimize cut costs based on edges,
  the FM algorithm minimizes cut costs based on nets

- Nodes and subgraphs are referred to as *cells* and *blocks*, respectively

2.4.3 Fiduccia-Mattheyses (FM) Algorithm

Given: a hypergraph $G(V,H)$ with nodes and *weighted* hyperedges
partition size constraints

Goal: to assign all nodes to disjoint partitions,
so as to minimize the total cost (weight) of all cut nets
while satisfying *partition size constraints*

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Terminology

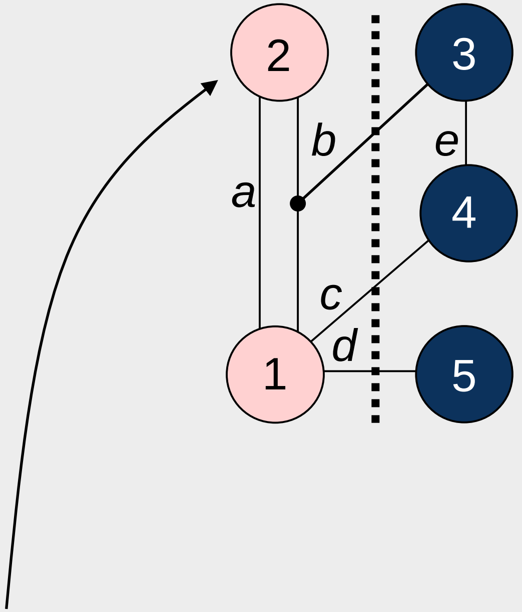
Gain $\Delta g(c)$ for cell *c*

$$\Delta g(c) \;= FS(c) - TE(c)\ ,$$

where

the “moving force“ $FS(c)$ is the number of nets connected to *c* but not connected to any other cells within *c*’s partition, i.e., cut nets that connect only to *c*, and

the “retention force“ $TE(c)$ is the number of *uncut* nets connected to *c*.



Cell 2: $FS(2) = 0$ $TE(2) = 1$ $\Delta g(2) = -1$

The higher the gain $\Delta g(c)$, the higher is the priority to move the cell *c* to the other partition.

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Terminology

Gain $\Delta g(c)$ for cell c

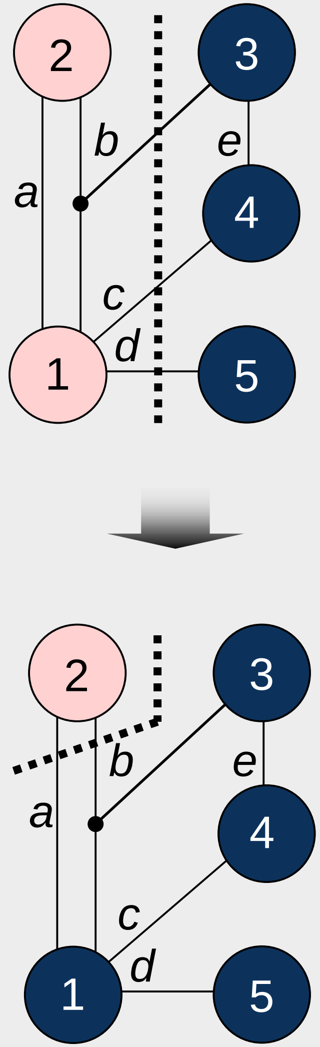
$$\Delta g(c) = FS(c) - TE(c) \,,$$

where

the “moving force“ $FS(c)$ is the number of nets connected to c but not connected to any other cells within c’s partition, i.e., cut nets that connect only to c, and

the “retention force“ $TE(c)$ is the number of *uncut* nets connected to c.

Cell 1:	$FS(1) = 2$	$TE(1) = 1$	$\Delta g(1) = 1$
Cell 2:	$FS(2) = 0$	$TE(2) = 1$	$\Delta g(2) = -1$
Cell 3:	$FS(3) = 1$	$TE(3) = 1$	$\Delta g(3) = 0$
Cell 4:	$FS(4) = 1$	$TE(4) = 1$	$\Delta g(4) = 0$
Cell 5:	$FS(5) = 1$	$TE(5) = 0$	$\Delta g(5) = 1$



2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Terminology

Maximum positive gain G_m of a pass

The maximum positive gain G_m is the cumulative cell gain of *m* moves that produce a minimum cut cost.

G_m is determined by the maximum sum of cell gains Δg over a prefix of *m* moves in a pass

$$G_m = \sum_{i=1}^{m} \Delta g_i$$

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Terminology

Ratio factor

The *ratio factor* is the relative balance between the two partitions with respect to cell area

It is used to prevent all cells from clustering into one partition.

The ratio factor *r* is defined as

$$r = \frac{area(A)}{area(A) + area(B)}$$

where *area*(*A*) and *area*(*B*) are the total respective areas of partitions *A* and *B*

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Terminology

Balance criterion

The balance criterion enforces the ratio factor.

To ensure feasibility, the maximum cell area $area_{max}(V)$ must be taken into account.

A partitioning of V into two partitions A and B is said to be balanced if

$$[\ r \cdot area(V) - area_{max}(V)\ ] \leq area(A) \leq [\ r \cdot area(V) + area_{max}(V)\ ]$$

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Terminology

Base cell

A base cell is a cell *c* that has the greatest cell gain Δ*g*(*c*) among all free cells, and whose move does not violate the balance criterion.

Base cell

Cell 1:	FS(1) = 2	TE(1) = 1	Δg(1) = 1
Cell 2:	*FS*(2) = 0	*TE*(2) = 1	Δ*g*(2) = -1
Cell 3:	*FS*(3) = 1	*TE*(3) = 1	Δ*g*(3) = 0
Cell 4:	*FS*(4) = 1	*TE*(4) = 1	Δ*g*(4) = 0

2.4.3 Fiduccia-Mattheyses (FM) Algorithm - One pass

Step 0: Compute the balance criterion

Step 1: Compute the cell gain Δg_1 of each cell

Step 2: $i = 1$

- Choose base cell c_1 that has maximal gain Δg_1 , move this cell

Step 3:

- Fix the base cell c_i
- Update all cells' gains that are connected to critical nets via the base cell c_i

Step 4:

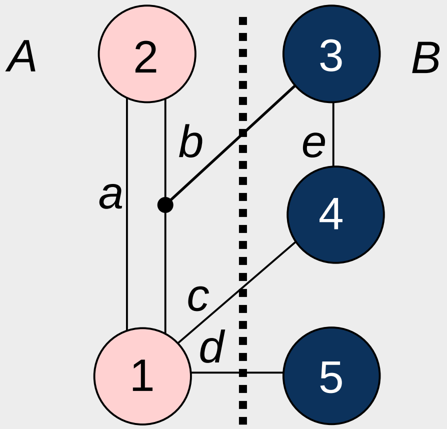
- If all cells are fixed, go to Step 5. If not:
- Choose next base cell c_i with maximal gain Δg_i and move this cell
- $i = i + 1$, go to Step 3

Step 5:

- Determine the best move sequence c_1, c_2, .., c_m $(1 \leq m \leq i)$, so that $G_m = \sum_{i=1}^{m} \Delta g_i$ is maximized
- If $G_m > 0$, go to Step 6. Otherwise, END

Step 6:

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example



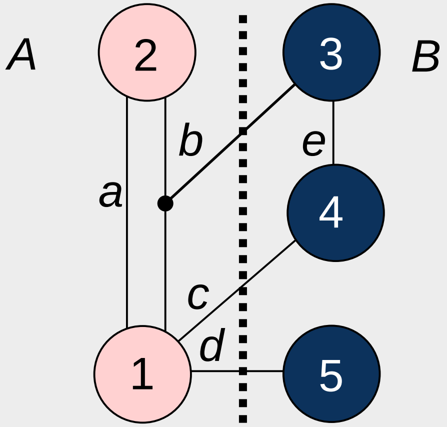
Given:
Ratio factor *r* = 0,375
area(Cell_1) = 2
area(Cell_2) = 4
area(Cell_3) = 1
area(Cell_4) = 4
area(Cell_5) = 5.

Step 0: Compute the balance criterion

$$[\, r \cdot area(V) - area_{max}(V) \,] \leq area(A) \leq [\, r \cdot area(V) + area_{max}(V) \,]$$

$$0{,}375 * 16 - 5 = 1 \leq area(A) \leq 11 = 0{,}375 * 16 + 5.$$

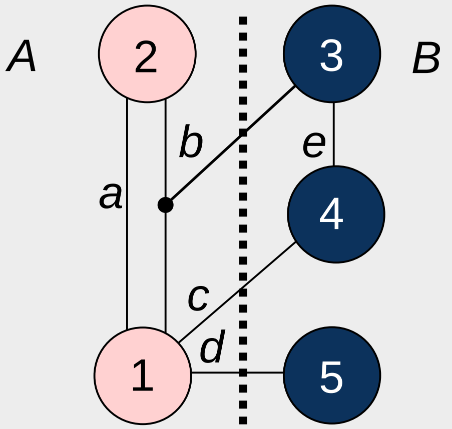
2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example



Step 1: Compute the gains of each cell

Cell 1:	*FS*(Cell_1) = 2	*TE*(Cell_1) = 1	Δ*g*(Cell_1) = 1
Cell 2:	*FS*(Cell_2) = 0	*TE*(Cell_2) = 1	Δ*g*(Cell_2) = -1
Cell 3:	*FS*(Cell_3) = 1	*TE*(Cell_3) = 1	Δ*g*(Cell_3) = 0
Cell 4:	*FS*(Cell_4) = 1	*TE*(Cell_4) = 1	Δ*g*(Cell_4) = 0
Cell 5:	*FS*(Cell_5) = 1	*TE*(Cell_5) = 0	Δ*g*(Cell_5) = 1

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example

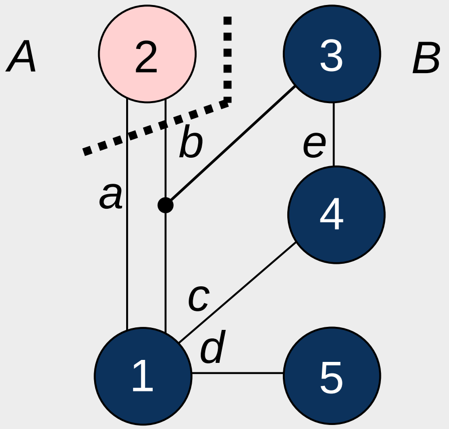


Cell1:	*FS*(Cell_1) = 2	*TE*(Cell_1) = 1	Δ*g*(Cell_1) = 1
Cell 2:	*FS*(Cell_2) = 0	*TE*(Cell_2) = 1	Δ*g*(Cell_2) = -1
Cell 3:	*FS*(Cell_3) = 1	*TE*(Cell_3) = 1	Δ*g*(Cell_3) = 0
Cell 4:	*FS*(Cell_4) = 1	*TE*(Cell_4) = 1	Δ*g*(Cell_4) = 0
Cell 5:	*FS*(Cell_5) = 1	*TE*(Cell_5) = 0	Δ*g*(Cell_5) = 1

Step 2: Select the base cell

Possible base cells are Cell 1 and Cell 5
Balance criterion after moving Cell 1: *area*(*A*) = *area*(Cell_2) = 4
Balance criterion after moving Cell 5: *area*(*A*) = *area*(*Cell_1*) + *area*(*Cell_2*) + *area*(*Cell_5*) = 11
Both moves respect the balance criterion, but Cell 1 is selected, moved,
and fixed as a result of the tie-breaking criterion.

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example

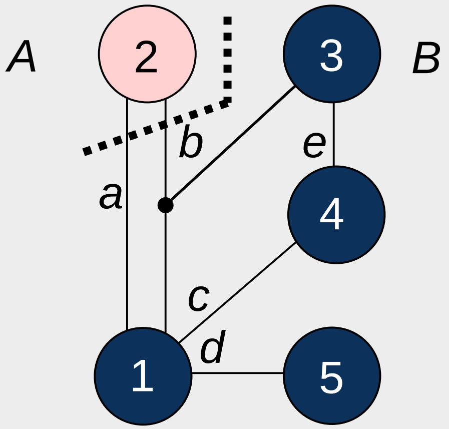


Step 3: Fix base cell, update Δg values

Cell 2:	FS(Cell_2) = 2	TE(Cell_2) = 0	Δg(Cell_2) = 2
Cell 3:	FS(Cell_3) = 0	TE(Cell_3) = 1	Δg(Cell_3) = -1
Cell 4:	FS(Cell_4) = 0	TE(Cell_4) = 2	Δg(Cell_4) = -2
Cell 5:	FS(Cell_5) = 0	TE(Cell_5) = 1	Δg(Cell_5) = -1

After Iteration i = 1: Partition A_1 = {2}, Partition B_1 = {1,3,4,5}, with fixed cell {1}.

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example



Iteration i = 1

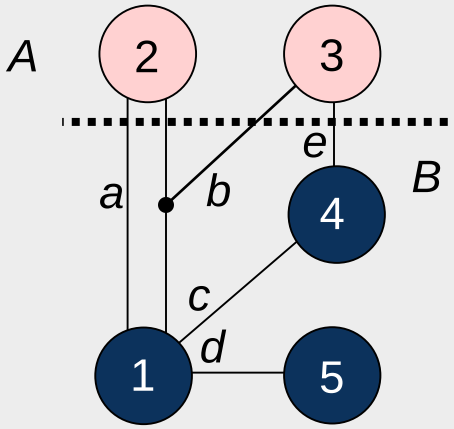
Cell 2:	FS(Cell_2) = 2	TE(Cell_2) = 0	Δg(Cell_2) = 2
Cell 3:	FS(Cell_3) = 0	TE(Cell_3) = 1	Δg(Cell_3) = -1
Cell 4:	FS(Cell_4) = 0	TE(Cell_4) = 2	Δg(Cell_4) = -2
Cell 5:	FS(Cell_5) = 0	TE(Cell_5) = 1	Δg(Cell_5) = -1

Iteration i = 2

Cell *2* has maximum gain Δg_2 = 2, $area(A)$ = 0, balance criterion is violated.
Cell *3* has next maximum gain Δg_2 = -1, $area(A)$ = 5, balance criterion is met.
Cell *5* has next maximum gain Δg_2= -1, $area(A)$ = 9, balance criterion is met.

Move cell *3*, updated partitions: A_2 = {2,3}, B_2 = {1,4,5}, with fixed cells {1,3}

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example



Iteration i = 2

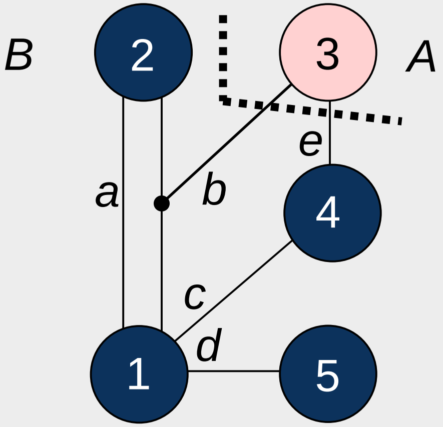
Cell 2: $\Delta g(\text{Cell\_2}) = 1$
Cell 4: $\Delta g(\text{Cell\_4}) = 0$
Cell 5: $\Delta g(\text{Cell\_5}) = -1$

Iteration i = 3

Cell *2* has maximum gain $\Delta g_3 = 1$, $area(A) = 1$, balance criterion is met.

Move cell *2*, updated partitions: $A_3 = \{3\}$, $B_3 = \{1,2,4,5\}$, with fixed cells {1,2,3}

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example



Iteration i = 3

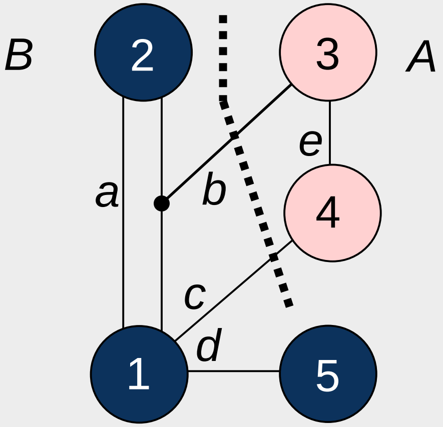
Cell 4: Δg(Cell_4) = 0
Cell 5: Δg(Cell_5) = -1

Iteration i = 4

Cell 4 has maximum gain $\Delta g_4 = 0$, *area*(*A*) = 5, balance criterion is met.

Move cell 4, updated partitions: A_4 = {3,4}, B_3 = {1,2,5}, with fixed cells {1,2,3,4}

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example



Iteration i = 4

Cell 5: Δg(Cell_5) = -1

Iteration i = 5

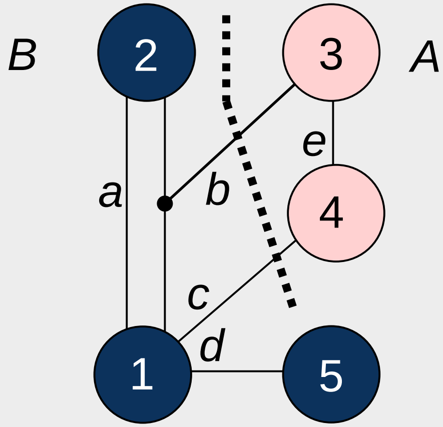
Cell 5 has maximum gain Δg_5 = -1, *area*(A) = 10, balance criterion is met.

Move cell 5, updated partitions: A_4 = {3,4,5}, B_3 = {1,2}, all cells {1,2,3,4,5} fixed.

2.4.3 Fiduccia-Mattheyses (FM) Algorithm – Example

Step 5: Find best move sequence $c_1 \ldots c_m$

$G_1 = \Delta g_1 = 1$
$G_2 = \Delta g_1 + \Delta g_2 = 0$
$G_3 = \Delta g_1 + \Delta g_2 + \Delta g_3 = 1$
$G_4 = \Delta g_1 + \Delta g_2 + \Delta g_3 + \Delta g_4 = 1$
$G_5 = \Delta g_1 + \Delta g_2 + \Delta g_3 + \Delta g_4 + \Delta g_5 = 0.$



Maximum positive cumulative gain $G_m = \sum_{i=1}^{m} \Delta g_i = 1$

found in iterations 1, 3 and 4.

The move prefix m = 4 is selected due to the better balance ratio (*area*(*A*) = 5); the four cells 1, 2, 3 and 4 are then moved.

Result of Pass 1: Current partitions: *A* = {3,4}, *B* = {*1,2,5*}, cut cost reduced from 3 to 2.

Runtime difference between KL & FM

- Runtime of partitioning algorithms
  - KL is sensitive to the number of nodes and edges
  - FM is sensitive to the number of nodes and nets (hyperedges)

- Asymptotic complexity of partitioning algorithms
  - KL has cubic time complexity *per pass*
  - FM has linear time complexity *per pass*

Chapter 2 – Netlist and System Partitioning

2.1 Introduction

2.2 Terminology

2.3 Optimization Goals

2.4 Partitioning Algorithms

- 2.4.1 Kernighan-Lin (KL) Algorithm
- 2.4.2 Extensions of the Kernighan-Lin Algorithm
- 2.4.3 Fiduccia-Mattheyses (FM) Algorithm

2.5 Framework for Multilevel Partitioning

- 2.5.1 Clustering
- 2.5.2 Multilevel Partitioning

2.6 System Partitioning onto Multiple FPGAs

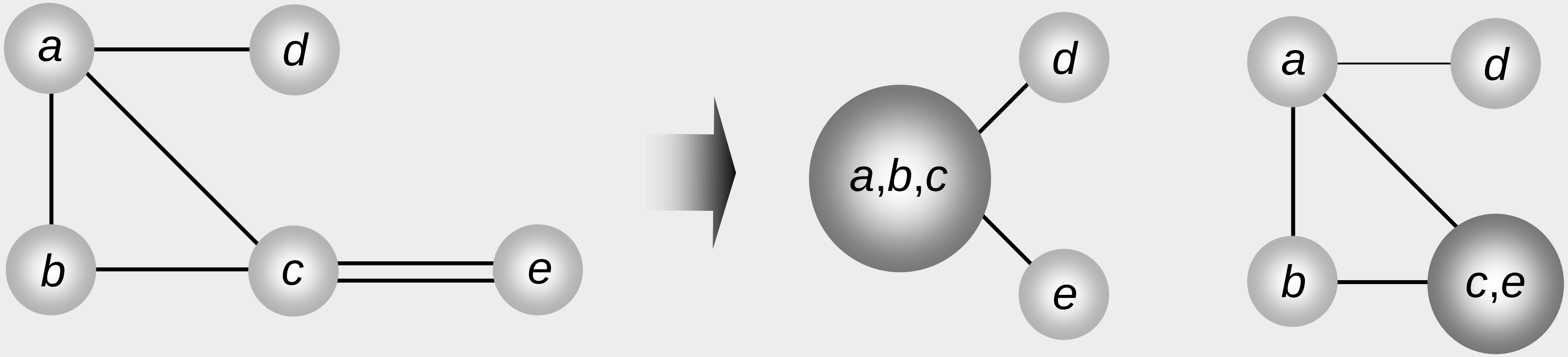
2.5.1 Clustering

- To simplify the problem, groups of tightly-connected nodes can be clustered, absorbing connections between these nodes

- Size of each cluster is often limited so as to prevent degenerate clustering, i.e. a single large cluster dominates other clusters

- Refinement should satisfy balance criteria

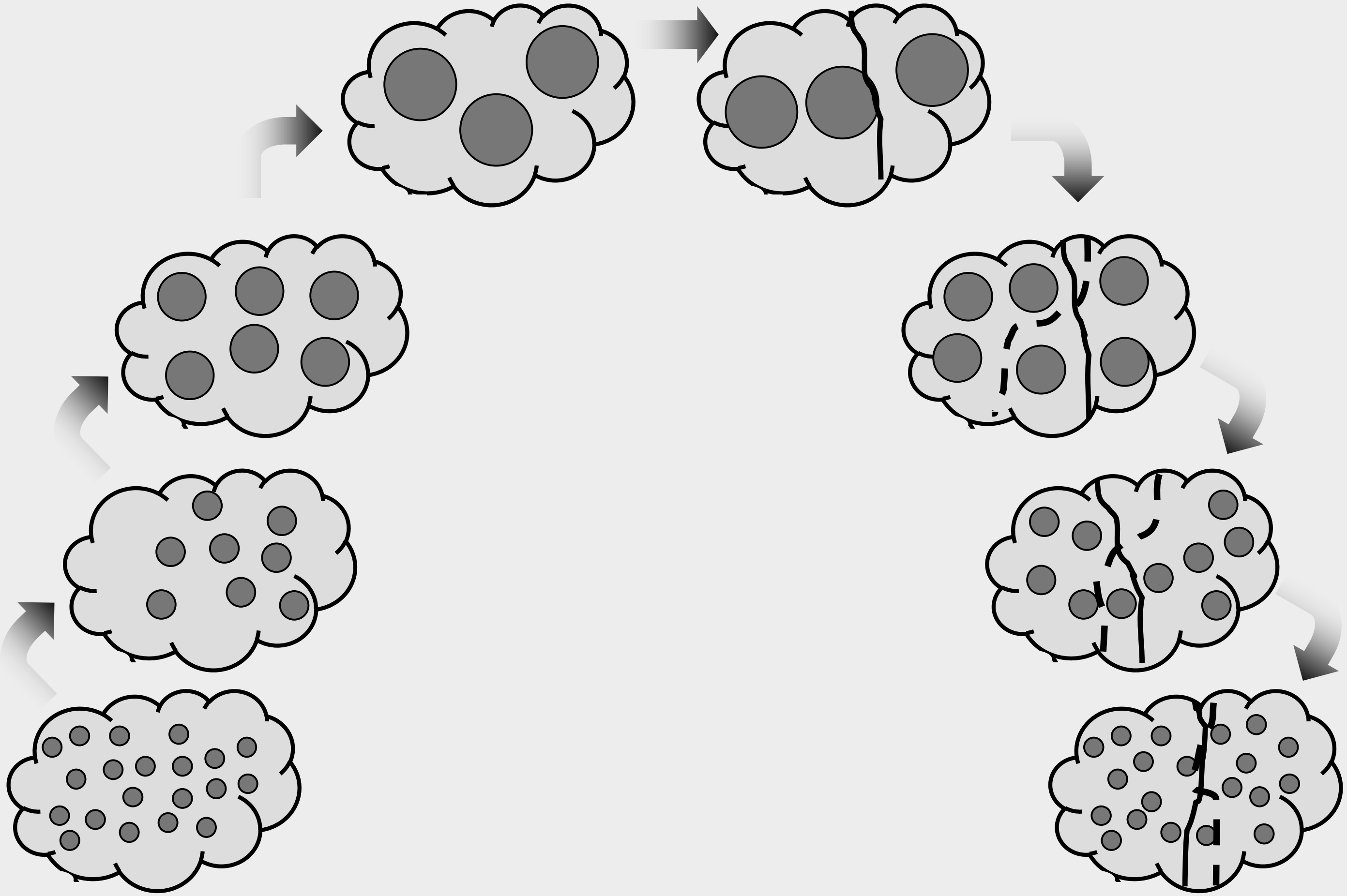
2.5.1 Clustering



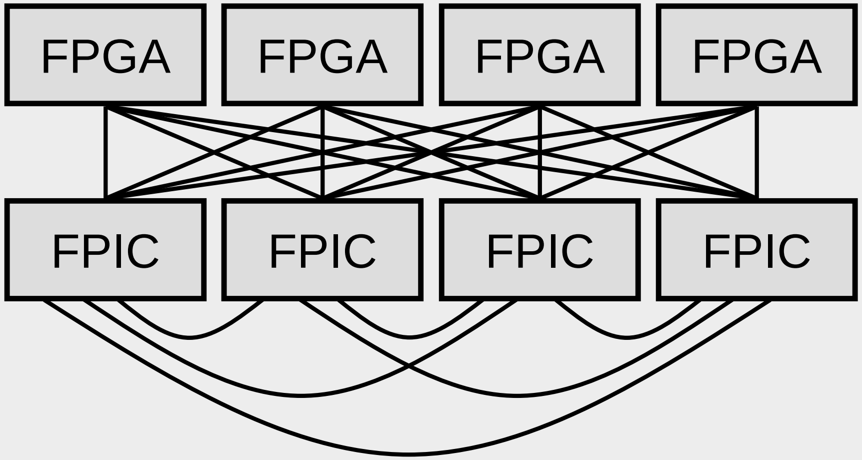
Initital graph

Possible clustering hierarchies of the graph

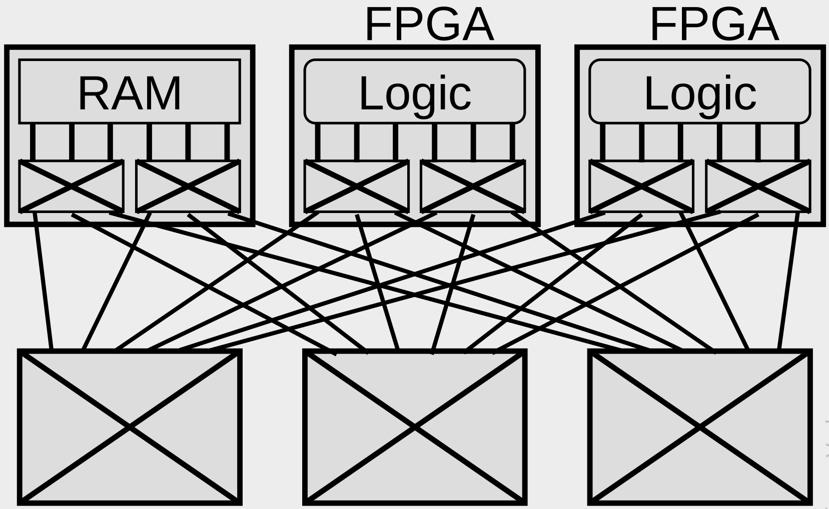
2.5.2 Multilevel Partitioning



2.6 System Partitioning onto Multiple FPGAs



Reconfigurable system with multiple FPGA and FPIC devices



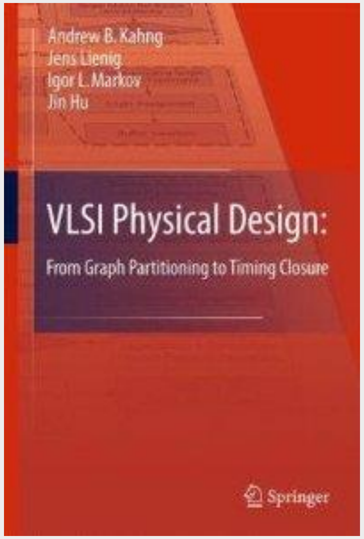
Mapping of a typical system architecture onto multiple FPGAs

Summary of Chapter 2

- Circuit netlists can be represented by graphs
- Partitioning a graph means assigning nodes to disjoint partitions
  - Total size of each partition (number/area of nodes) is limited
  - Objective: minimize the number connections between partitions
- Basic partitioning algorithms
  - Move-based, move are organized into passes
  - KL swaps pairs of nodes from different partitions
  - FM re-assigns one node at a time
  - FM is faster, usually more successful
- Multilevel partitioning
  - Clustering
  - FM partitioning
  - Refinement (also uses FM partitioning)
- Application: system partitioning into FPGAs
  - Each FPGA is represented by a partition

VLSI Physical Design: From Graph Partitioning to Timing Closure

Chapter 7 – Specialized Routing



Original Authors:
Andrew B. Kahng, Jens Lienig, Igor L. Markov, Jin Hu

Chapter 7 – Specialized Routing

7.1 Introduction to Area Routing

7.2 Net Ordering in Area Routing

7.3 Non-Manhattan Routing

- 7.3.1 Octilinear Steiner Trees
- 7.3.2 Octilinear Maze Search

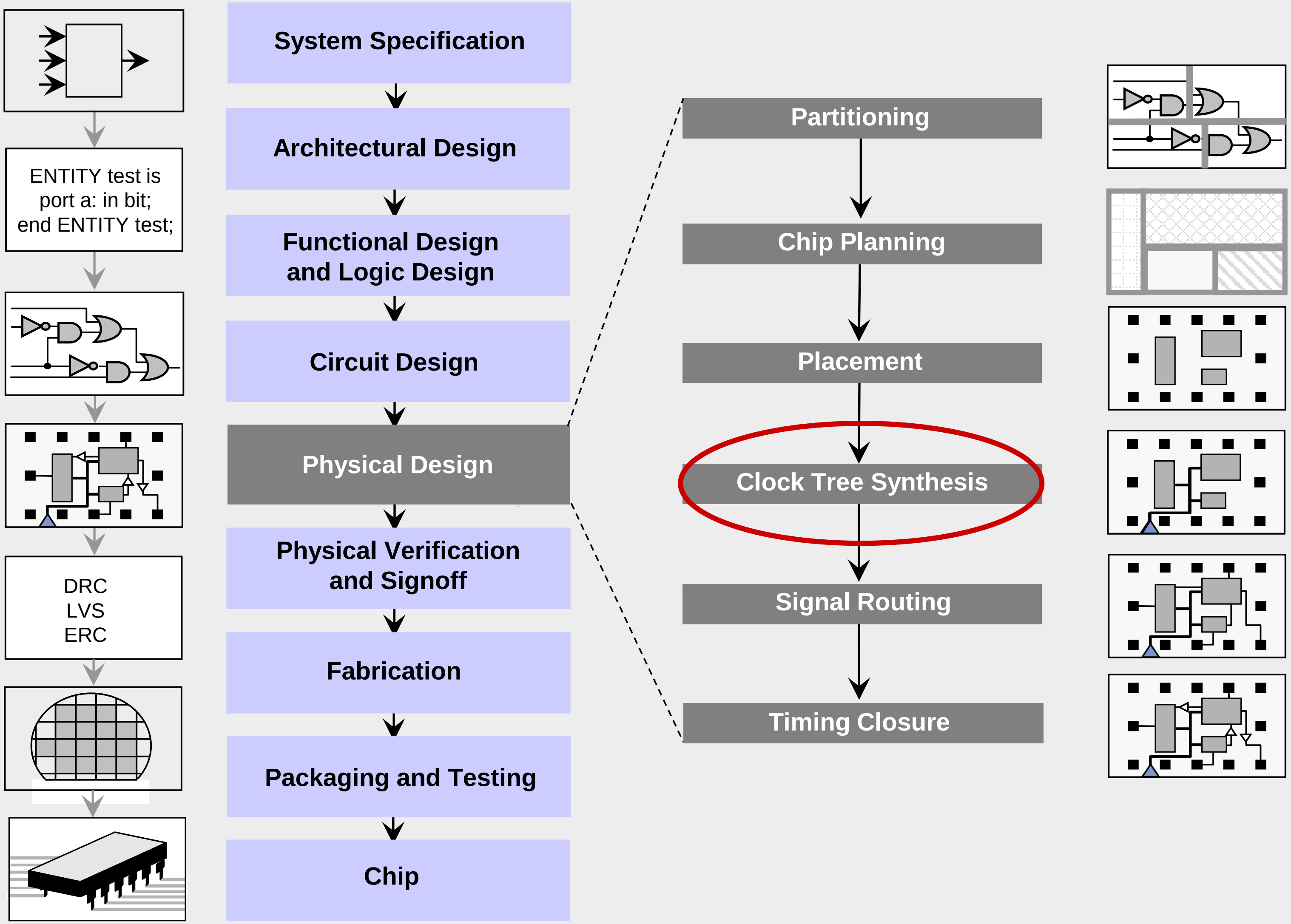
7.4 Basic Concepts in Clock Networks

- 7.4.1 Terminology
- 7.4.2 Problem Formulations for Clock-Tree Routing

7.5 Modern Clock Tree Synthesis

- 7.5.1 Constructing Trees with Zero Global Skew
- 7.5.2 Clock Tree Buffering in the Presence of Variation

7 Specialized Routing



```
ENTITY test is
port a: in bit;
end ENTITY test;
```

DRC
LVS
ERC

System Specification

Architectural Design

Functional Design and Logic Design

Circuit Design

Physical Design

Physical Verification and Signoff

Fabrication

Packaging and Testing

Chip

Partitioning

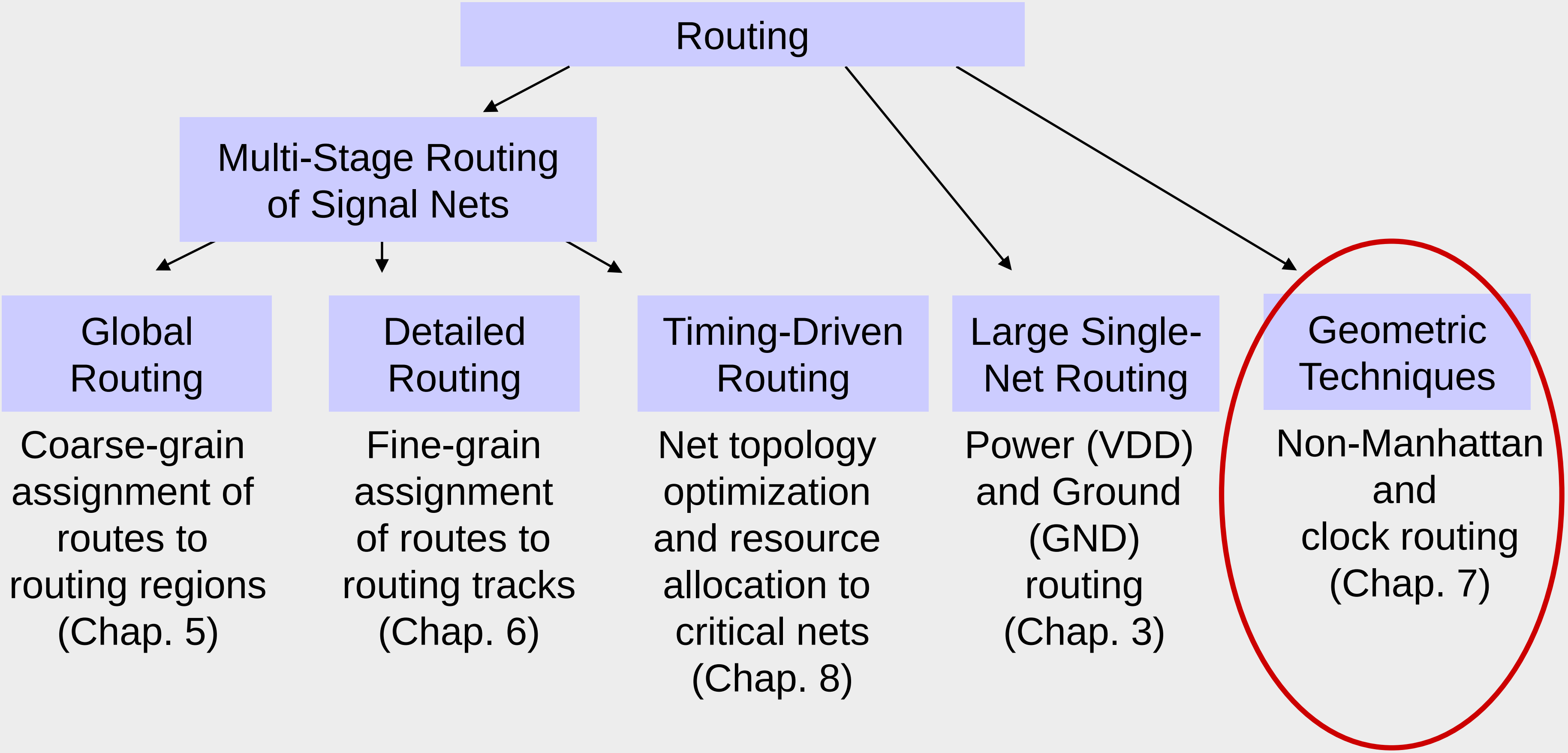
Chip Planning

Placement

Clock Tree Synthesis

Signal Routing

Timing Closure



Routing

Multi-Stage Routing of Signal Nets

Global Routing

Coarse-grain assignment of routes to routing regions (Chap. 5)

Detailed Routing

Fine-grain assignment of routes to routing tracks (Chap. 6)

Timing-Driven Routing

Net topology optimization and resource allocation to critical nets (Chap. 8)

Large Single-Net Routing

Power (VDD) and Ground (GND) routing (Chap. 3)

Geometric Techniques

Non-Manhattan and clock routing (Chap. 7)

7 Specialized Routing

- Area routing directly constructs metal routes for signal connections (no global and detailed routing, Secs. 7.1-7.2)
- Non-Manhattan routing is presented in Sec. 7.3
- Clock signals and other nets that require special treatment are discussed in Secs. 7.4-7.5

7.1 Introduction to Area Routing

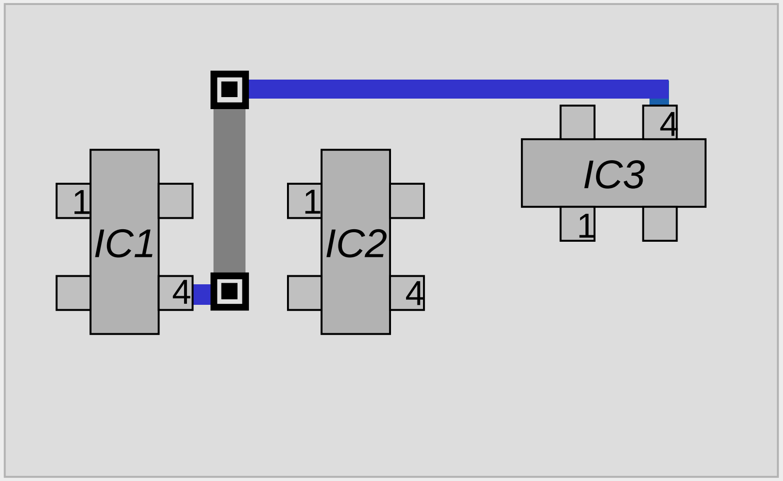
- The goal of area routing is to route all nets in the design
  - without global routing
  - within the given layout space
  - while meeting all geometric and electrical design rules

- Area routing performs the following optimizations
  - minimizing the total routed length and number of vias of all nets
  - minimizing the total area of wiring and the number of routing layers
  - minimizing the circuit delay and ensuring an even wire density
  - avoiding harmful capacitive coupling between neighboring routes

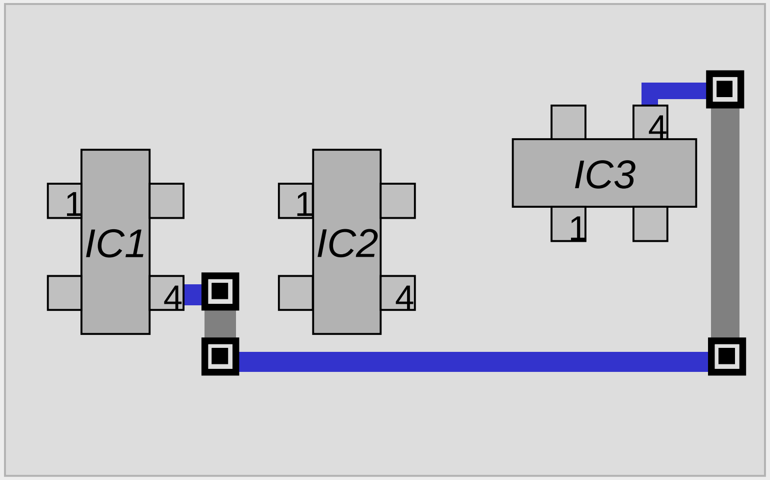
- Subject to
  - technology constraints (number of routing layers, minimal wire width, etc.)
  - electrical constraints (signal integrity, coupling, etc.)
  - geometry constraints (preferred routing directions, wire pitch, etc.)

7.1 Introduction to Area Routing

Minimal wirelength:



Alternative routing path:



▬ Metal1

▮ Metal2

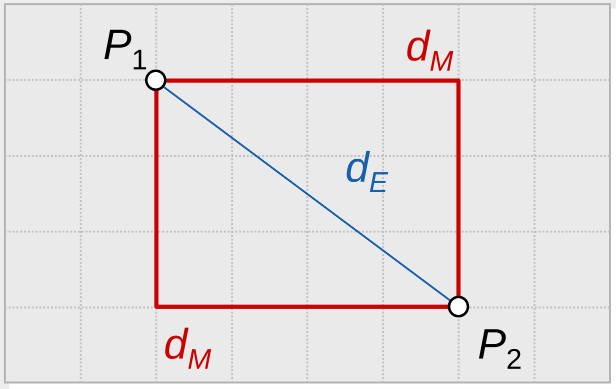
▣ Via

7.1 Introduction to Area Routing

Distance metric between two points P_1 (x_1,y_1) and P_2 (x_2,y_2)

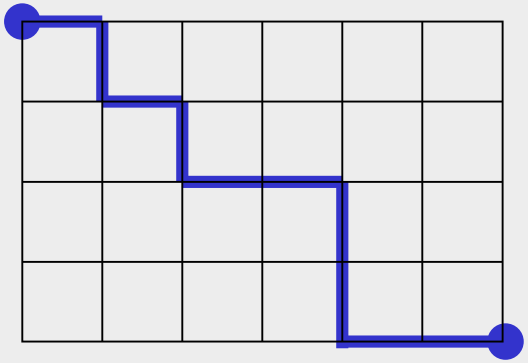
Euclidean distance $$d_E(P_1,P_2)=\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}=\sqrt{(\Delta x)^2+(\Delta y)^2}$$

Manhattan distance $$d_M(P_1,P_2)=|x_2-x_1|+|y_2-y_1|=|\Delta x|+|\Delta y|$$



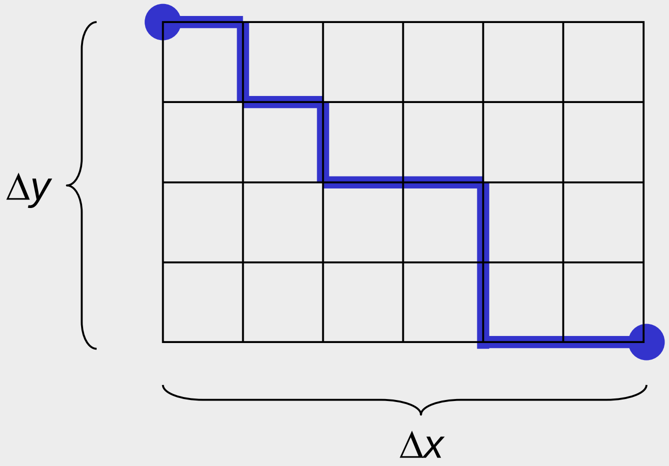
7.1 Introduction to Area Routing

- Multiple Manhattan shortest paths between two points



7.1 Introduction to Area Routing

- Multiple Manhattan shortest paths between two points



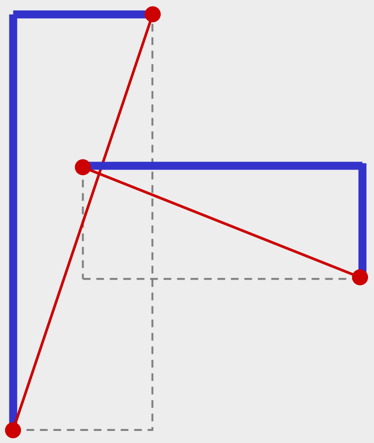
$m = 210$

With no obstacles, the number of Manhattan shortest paths in an $\Delta x \times \Delta y$ region is

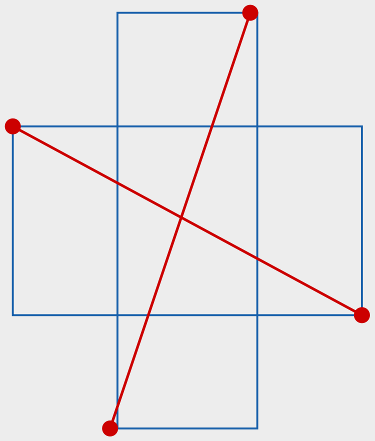
$$m = \binom{\Delta x + \Delta y}{\Delta x} = \binom{\Delta x + \Delta y}{\Delta y} = \frac{(\Delta x + \Delta y)!}{\Delta x! \Delta y!}$$

7.1 Introduction to Area Routing

- Two pairs of points may admit non-intersecting Manhattan shortest paths, while their Euclidean shortest paths intersect (but not vice versa).



- If all pairs of Manhattan shortest paths between two pairs of points intersect, then so do Euclidean shortest paths.



7.1 Introduction to Area Routing

- The Manhattan distance d_M is (slightly) larger than the Euclidean distance d_E:

$$\frac{d_{\mathrm{M}}}{d_{\mathrm{E}}} = \begin{cases} 1.41 & \text{worst case: a square where } \Delta x = \Delta y \\ 1.27 & \text{on average, without obstacles} \\ 1.15 & \text{on average, with obstacles} \end{cases}$$

7.2 Net Ordering in Area Routing

7.1 Introduction to Area Routing

7.2 Net Ordering in Area Routing

7.3 Non-Manhattan Routing

7.3.1 Octilinear Steiner Trees

7.3.2 Octilinear Maze Search

7.4 Basic Concepts in Clock Networks

7.4.1 Terminology

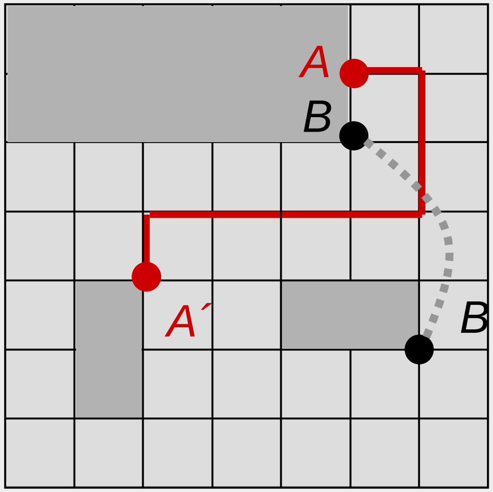
7.4.2 Problem Formulations for Clock-Tree Routing

7.5 Modern Clock Tree Synthesis

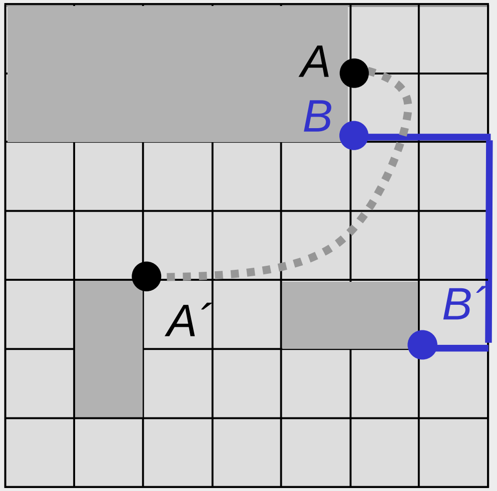
7.5.1 Constructing Trees with Zero Global Skew

7.5.2 Clock Tree Buffering in the Presence of Variation

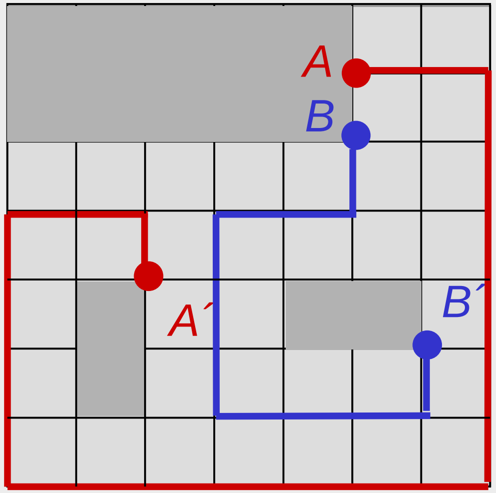
Effect of net ordering on routability



Optimal routing of net *A*



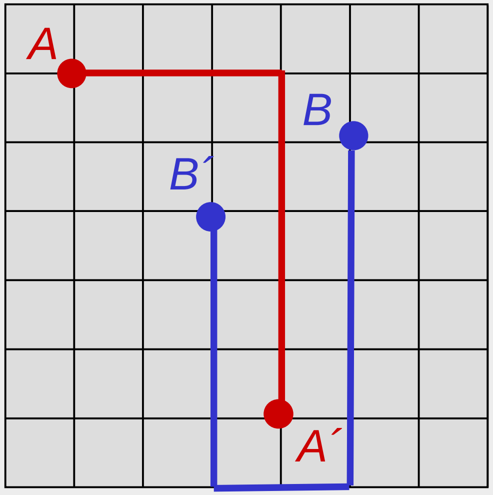
Optimal routing of net *B*



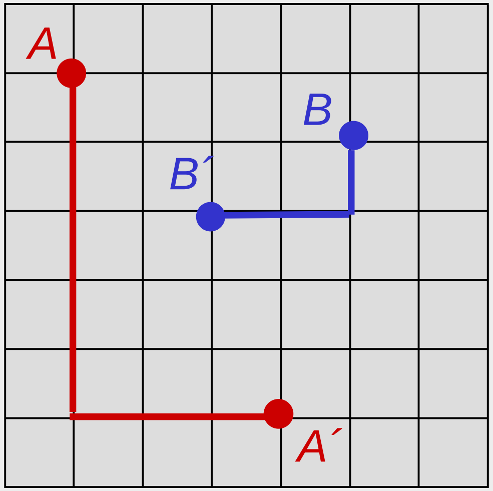
Nets *A* and *B* can be routed only with detours

7.2 Net Ordering in Area Routing

Effect of net ordering on total wirelength



Routing net *A* first



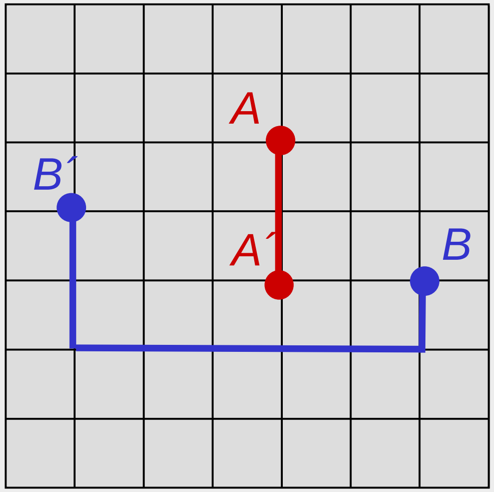
Routing net *B* first

7.2 Net Ordering in Area Routing

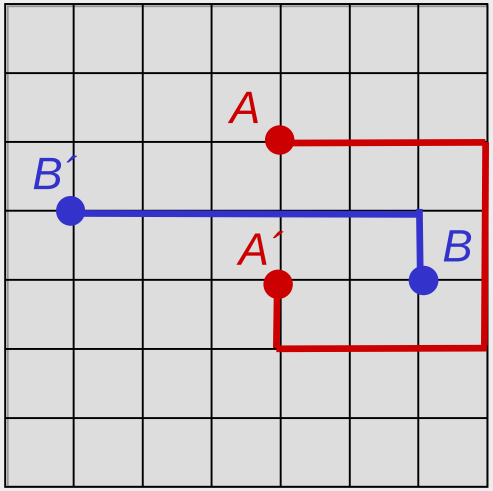
- For *n* nets, there are *n*! possible net orderings

⇒ Constructive heuristics are used

- **Rule 1:** For two nets *i* and *j*, if *aspect ratio* (*i*) > *aspect ratio* (*j*), then *i* is routed before *j*



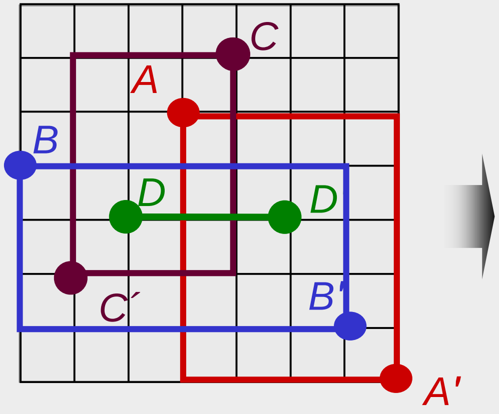
Net *A* has a higher aspect ratio of its bounding box; routing *A* first results in shorter total wirlength



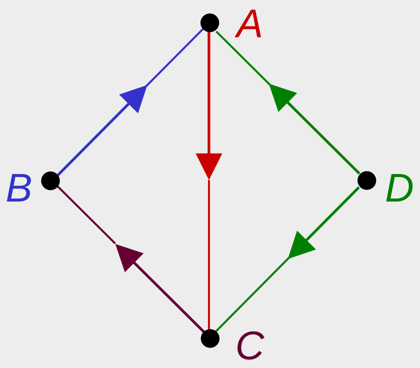
Routing net *B* first results in longer total wirelength

7.2 Net Ordering in Area Routing

- **Rule 2:** For two nets *i* and *j*, if the pins of *i* are contained within *MBB*(*j*), then *i* is routed before *j*

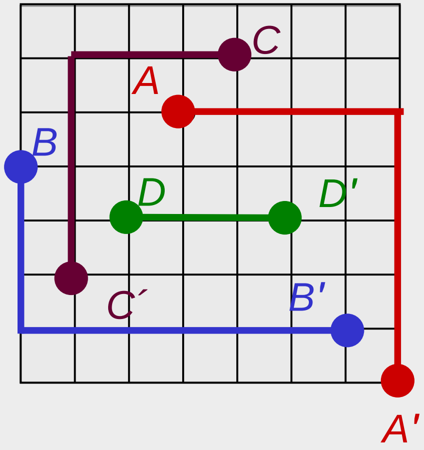


Constraint Graph



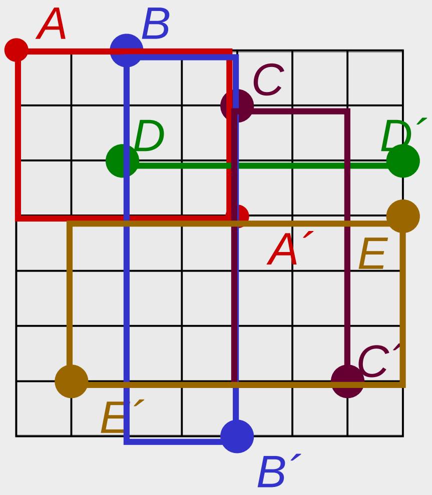
Net Ordering

Ordering *D*-*A*-*C*-*B*
or *D*-*C*-*B*-*A*
(not *D*-*B*-*A*-*C*)

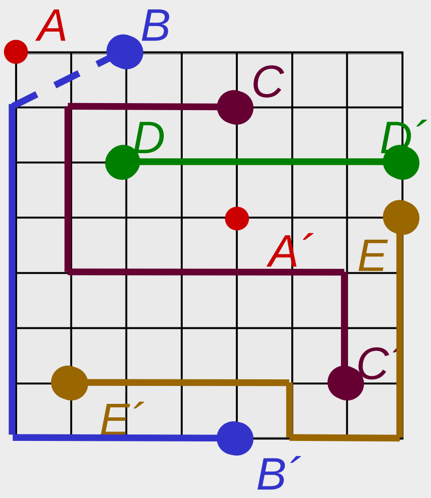


7.2 Net Ordering in Area Routing

- **Rule 3:** Let $\Pi(net)$ be the number of pins within *MBB*(*net*) for net *net*. For two nets *i* and *j*, if $\Pi(i) < \Pi(j)$, then *i* is routed before *j*.
  - For each net, consider the pins of other nets within its bounding box
  - The net with the smallest number of such pins is routed first
  - Ties are broken based on the number of pins that are contained within the bounding box and on its edge



	Pins Inside (Edge)	π (*net*)
MBB (*A*)	*D* (*B*,*C*,*D*)	3
B	- (*A*,*C*,*D*)	3
C	- (*A*)	1
D	- (-)	0
E	- (*A*,*C*)	2



7.3 Non-Manhattan Routing

7.1 Introduction to Area Routing

7.2 Net Ordering in Area Routing

7.3 Non-Manhattan Routing

7.3.1 Octilinear Steiner Trees

7.3.2 Octilinear Maze Search

7.4 Basic Concepts in Clock Networks

7.4.1 Terminology

7.4.2 Problem Formulations for Clock-Tree Routing

7.5 Modern Clock Tree Synthesis

7.5.1 Constructing Trees with Zero Global Skew

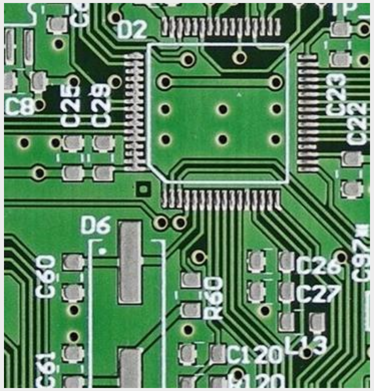
7.5.2 Clock Tree Buffering in the Presence of Variation

7.3 Non-Manhattan Routing

- Allow 45- or 60-degree segments in addition to horizontal and vertical segments

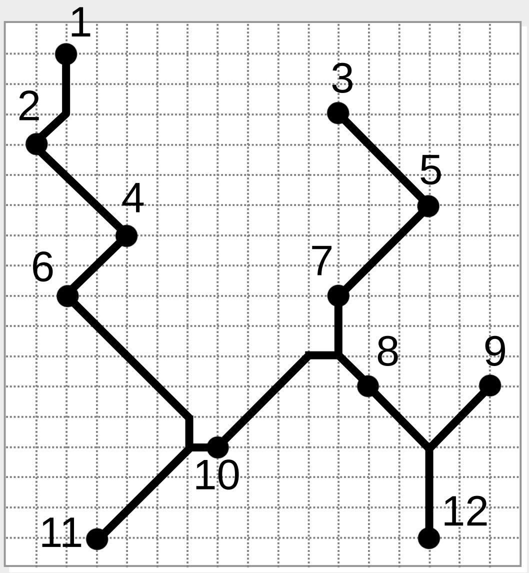
- λ-geometry, where λ represents the number of possible routing directions and the angles π / λ at which they can be oriented
  - λ = 2 (90 degrees): Manhattan routing (four routing directions)
  - λ = 3 (60 degrees): Y-routing (six routing directions)
  - λ = 4 (45 degrees): X-routing (eight routing directions)

- Non-Manhattan routing is primarily employed on printed circuit boards (PCBs)



7.3.1 Octilinear Steiner Trees

- Route planning using octilinear Steiner minimum trees (OSMT)
- Generalize rectilinear Steiner trees by allowing segments that extend in eight directions
- More freedom when placing Steiner points



7.3.1 Octilinear Steiner Trees

Octilinear Steiner Tree Algorithm

Input: set of all pins *P* and their coordinates

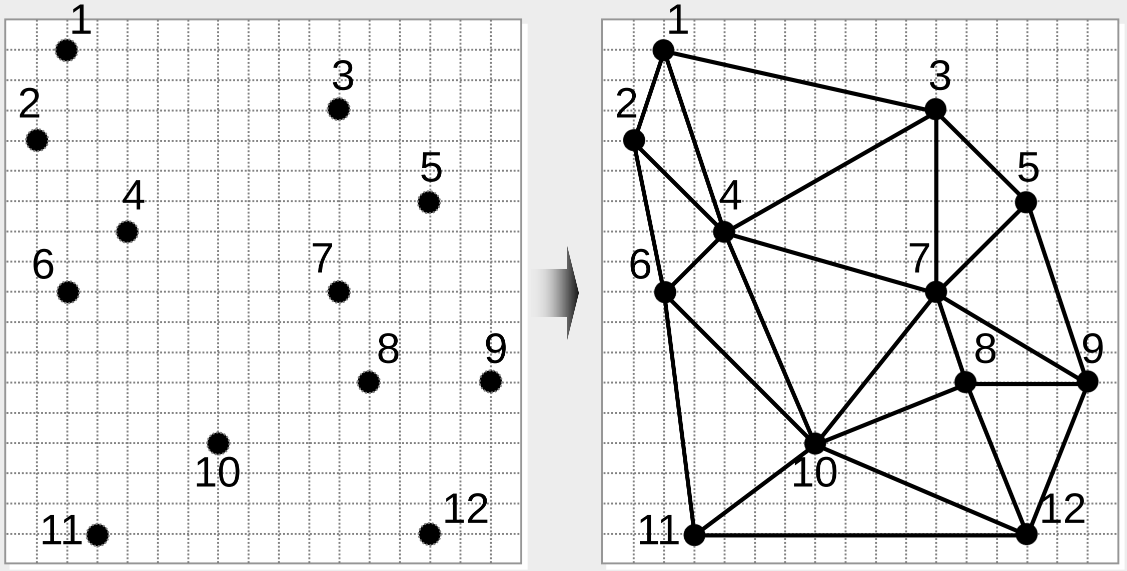
Output: heuristic octilinear minimum Steiner tree *OST*

```
$OST$ = Ø
$T$ = set of all three-pin nets of $P$ found by Delaunay triangulation
$sortedT$ = SORT($T$,minimum octilinear distance)
for ($i$ = 1 to |$sortedT$|)
    $subT$ = ROUTE($sortedT[i]$)        // route minimum tree over $subT$
    ADD($OST$,$subT$)                   // add route to existing tree
    IMPROVE($OST$,$subT$)               // locally improve $OST$ based on $subT$
```

T.-Y.; Chang, et. al.: Multilevel Full-Chip Routing for the X-Based Architecture

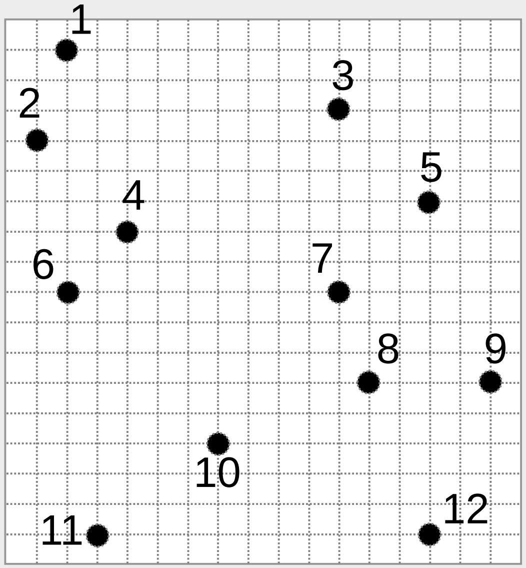
7.3.1 Octilinear Steiner Trees

(1) Triangulate

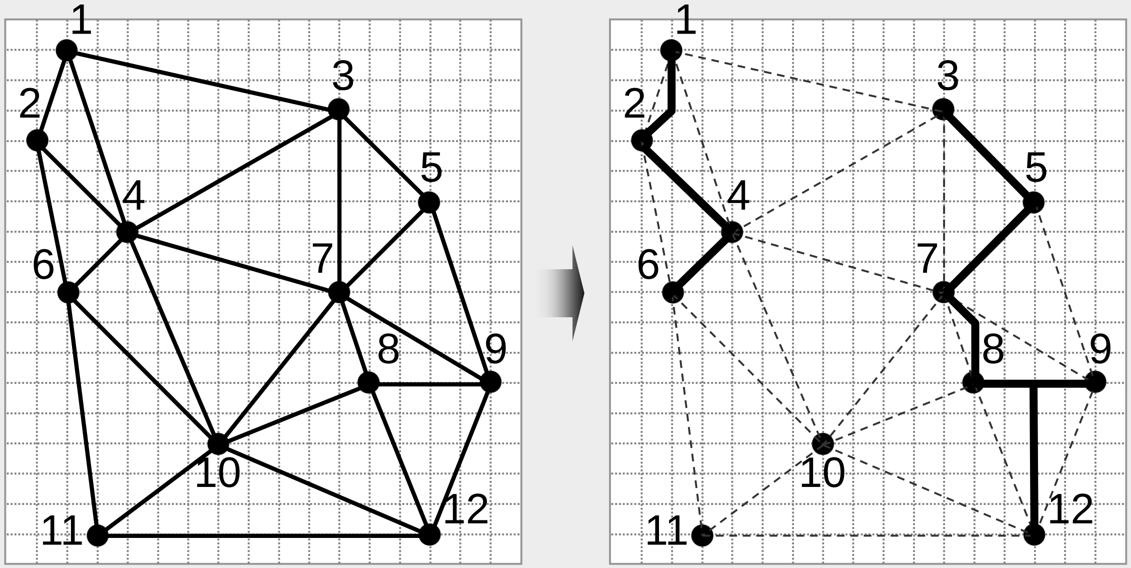


7.3.1 Octilinear Steiner Trees

(1) Triangulate



(2) Add route to existing tree

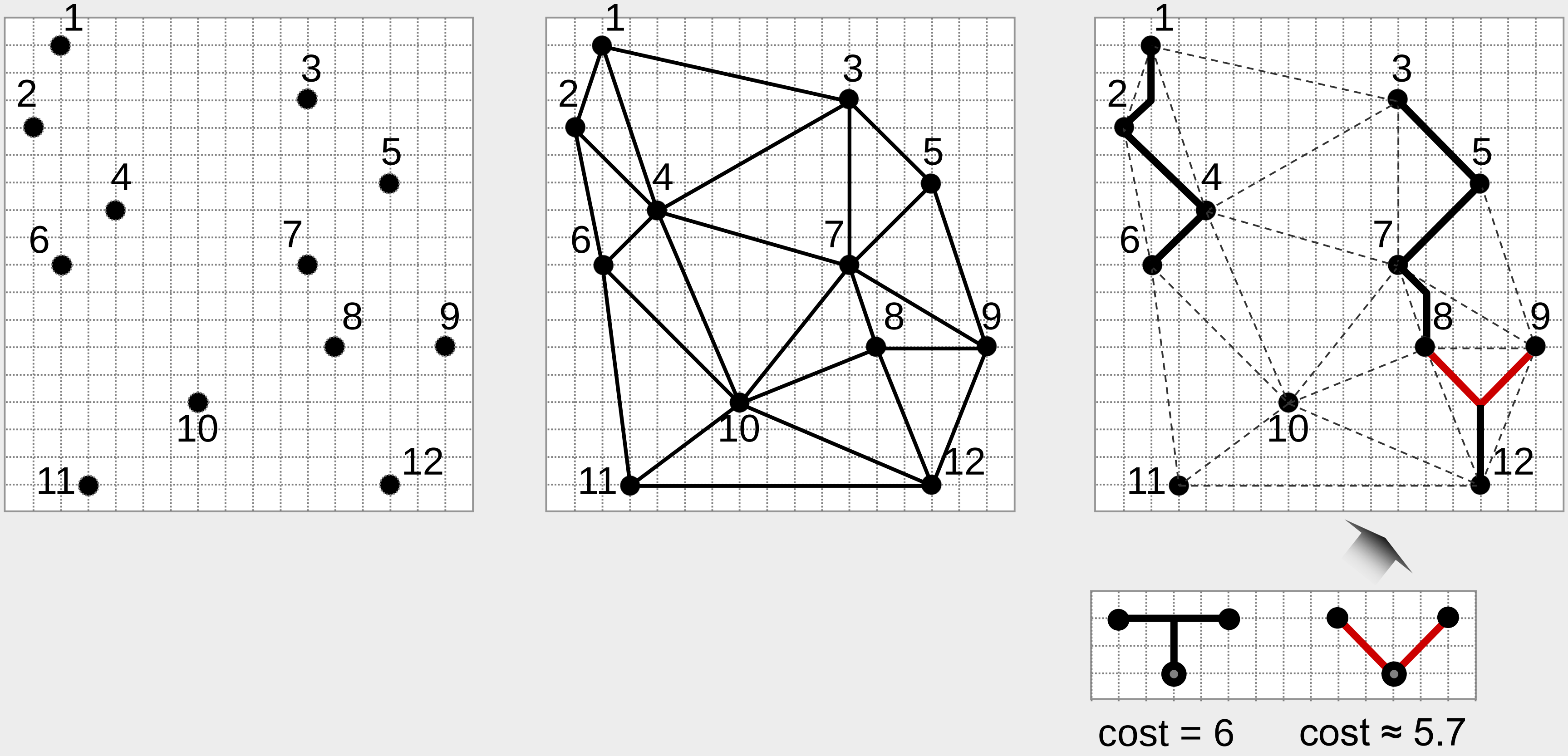


7.3.1 Octilinear Steiner Trees

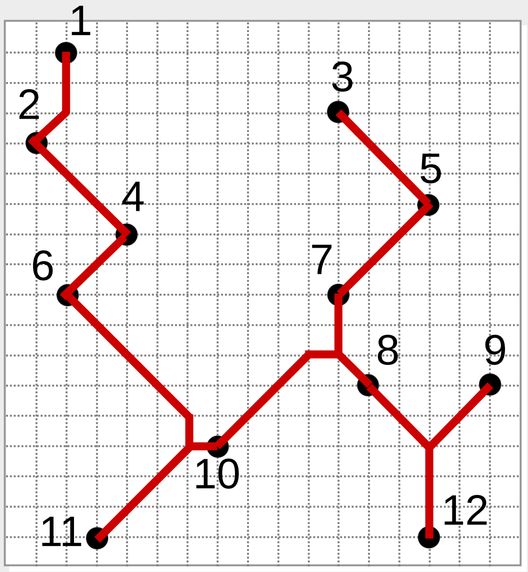
(1) Triangulate

(2) Add route to existing tree

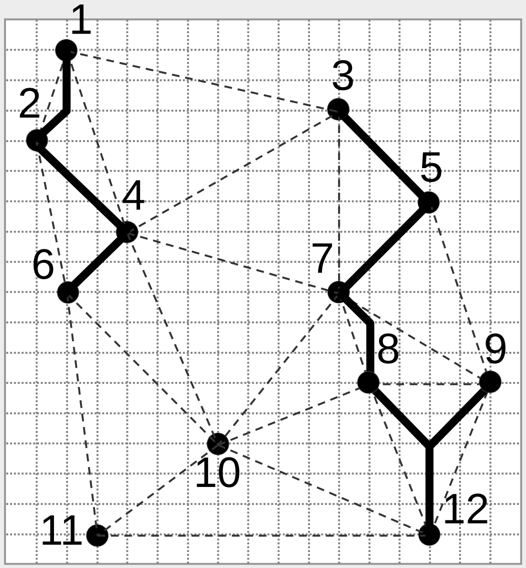
(3) Locally improve OST



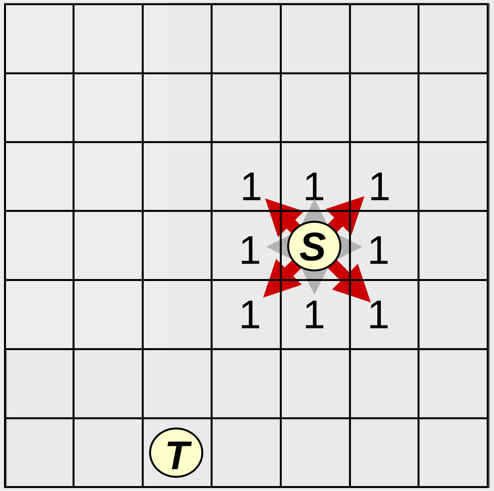
Final OST after merging all subtrees



(3) Locally improve OST

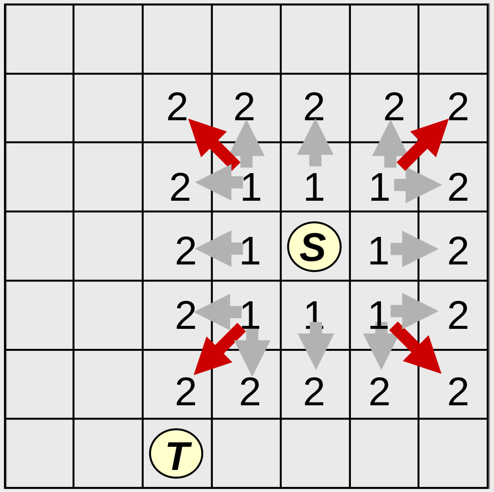


7.3.2 Octilinear Maze Search



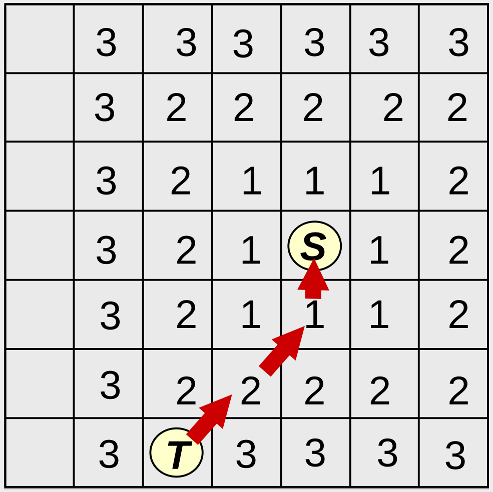
			1	1	1	
			1	S	1	
			1	1	1	
		T				

Expansion (1)



		2	2	2	2	2
		2	1	1	1	2
		2	1	S	1	2
		2	1	1	1	2
		2	2	2	2	2
		T				

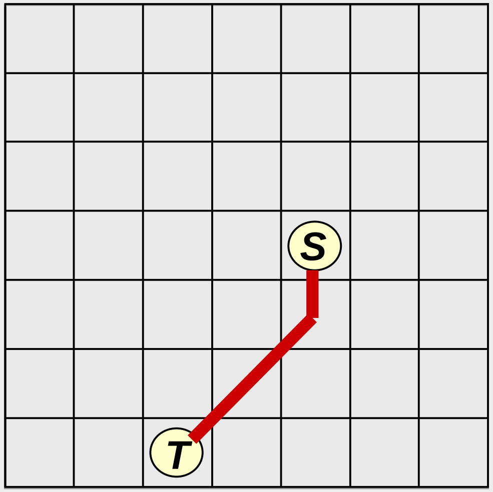
Expansion (2)



	3	3	3	3	3	3
	3	2	2	2	2	2
	3	2	1	1	1	2
	3	2	1	S	1	2
	3	2	1	1	1	2
	3	2	2	2	2	2
	3	T	3	3	3	3

Backtracing

7.3.2 Octilinear Maze Search



7.4 Basic Concepts in Clock Networks

7.1 Introduction to Area Routing

7.2 Net Ordering in Area Routing

7.3 Non-Manhattan Routing

- 7.3.1 Octilinear Steiner Trees
- 7.3.2 Octilinear Maze Search

7.4 Basic Concepts in Clock Networks

- 7.4.1 Terminology
- 7.4.2 Problem Formulations for Clock-Tree Routing

7.5 Modern Clock Tree Synthesis

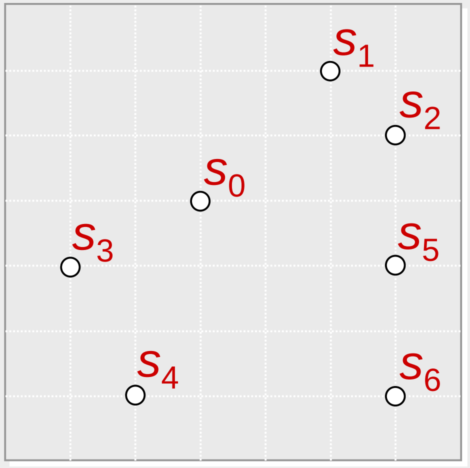
- 7.5.1 Constructing Trees with Zero Global Skew
- 7.5.2 Clock Tree Buffering in the Presence of Variation

7.4.1 Terminology

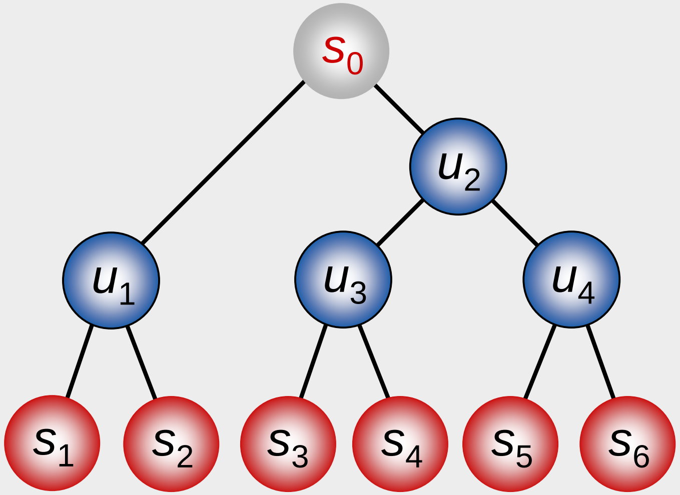
- A clock routing instance (clock net) is represented by n+1 terminals, where s_0 is designated as the source, and $S = \{s_1,s_2, \ldots ,s_n\}$ is designated as sinks
  - Let s_i, $0 \leq i \leq n$, denote both a terminal and its location
- A clock routing solution consists of a set of wire segments that connect all terminals of the clock net, so that a signal generated at the source propagates to all of the sinks
  - Two aspects of clock routing solution: topology and geometric embedding
- The clock-tree topology (clock tree) is a rooted binary tree G with n leaves corresponding to the set of sinks
  - Internal nodes = Steiner points

7.4.1 Terminology

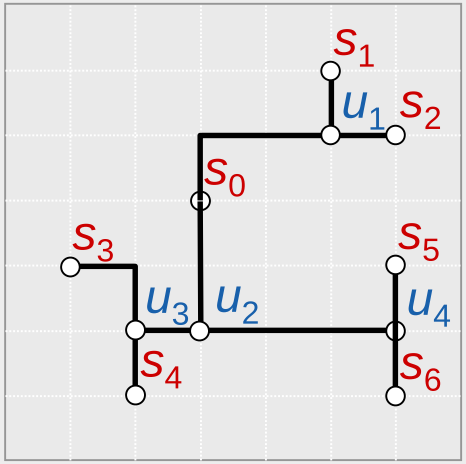
Clock routing problem instance



Connection topology



Embedding



7.4.1 Terminology

- Clock skew: (maximum) difference in clock signal arrival times between sinks

$$skew(T) = \max_{s_i, s_j \in S} | t(s_0, s_i) - t(s_0, s_j) |$$

- Local skew: maximum difference in arrival times of the clock signal at the clock pins of two or more related sinks
  - Sinks within distance *d* > 0
  - Flip-flops or latches connected by a directed signal path

- Global skew: maximum difference in arrival times of the clock signal at the clock pins of any two (related or unrelated) sinks
  - Difference between shortest and longest source-sink path delays in the clock distribution network
  - The term “skew” typically refers to “global skew”

7.4.2 Problem Formulations for Clock-Tree Routing

- Zero skew: zero-skew tree (ZST)
  - ZST problem

- Bounded skew: true ZST may not be necessary in practice
  - Signoff timing analysis is sufficient with a non-zero skew bound
  - In addition to final (signoff) timing, this relaxation can be useful with intermediate delay models when it facilitates reductions in the length of the tree
  - Bounded-Skew Tree (BST) problem

- Useful skew: correct chip timing only requires control of the local skews between pairs of interconnected flip-flops or latches
  - Useful skew formulation is based on analysis of local skew constraints

7.5 Modern Clock Tree Synthesis

7.1 Introduction to Area Routing

7.2 Net Ordering in Area Routing

7.3 Non-Manhattan Routing

- 7.3.1 Octilinear Steiner Trees
- 7.3.2 Octilinear Maze Search

7.4 Basic Concepts in Clock Networks

- 7.4.1 Terminology
- 7.4.2 Problem Formulations for Clock-Tree Routing

7.5 Modern Clock Tree Synthesis

- 7.5.1 Constructing Trees with Zero Global Skew
- 7.5.2 Clock Tree Buffering in the Presence of Variation

7.5 Modern Clock Tree Synthesis

- A clock tree should have low skew, while delivering the same signal to every sequential gate

- Clock tree synthesis is performed in two steps:

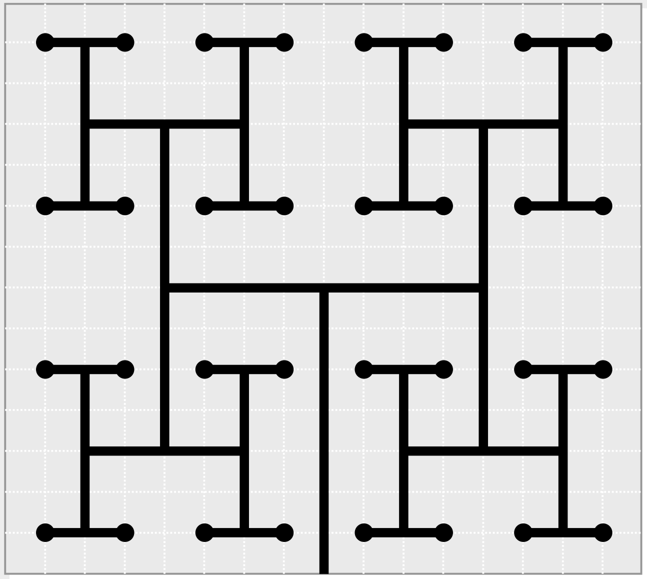
(1) Initial tree construction (Sec. 7.5.1) with one of these scenarios

- Construct a regular clock tree, largely independent of sink locations
- Simultaneously determine a topology and an embedding
- Construct only the embedding, given a clock-tree topology as input

(2) Clock buffer insertion and several subsequent skew optimizations (Sec. 7.5.2)

7.5.1 Constructing Trees with Zero Global Skew

H-tree



- Exact zero skew due to the symmetry of the H-tree
- Used for top-level clock distribution, not for the entire clock tree
  - Blockages can spoil the symmetry of an H-tree
  - Non-uniform sink locations and varying sink capacitances also complicate the design of H-trees

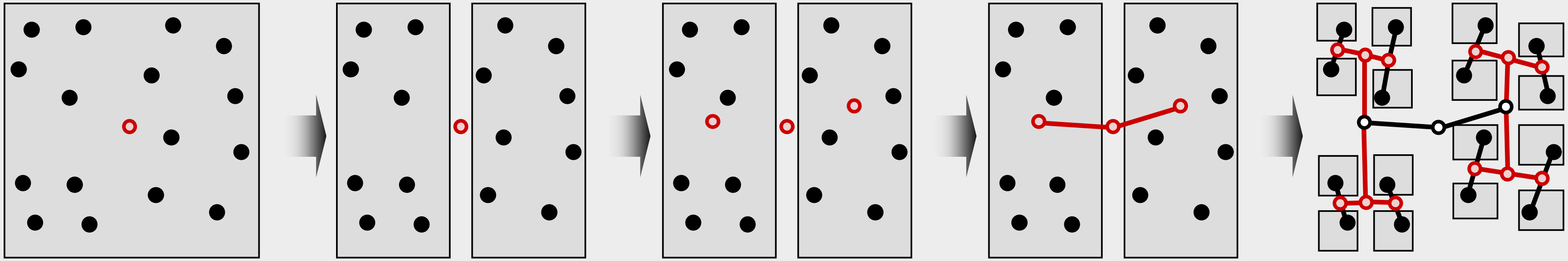
7.5.1 Constructing Trees with Zero Global Skew

Method of Means and Medians (MMM)

- Can deal with arbitrary locations of clock sinks
- Basic idea:
  - Recursively partition the set of terminals into two subsets of equal size (median)
  - Connect the center of gravity (COG) of the set to the centers of gravity of the two subsets (the mean)

7.5.1 Constructing Trees with Zero Global Skew

Method of Means and Medians (MMM)



Find the center of gravity○

Partition *S* by the median

Find the center of gravity for the left and right subsets of *S*

Connect the center of gravity of *S* with the centers of gravity of the left and right subsets

Final result after recursively performing MMM on each subset

7.5.1 Constructing Trees with Zero Global Skew

Method of Means and Medians (MMM)

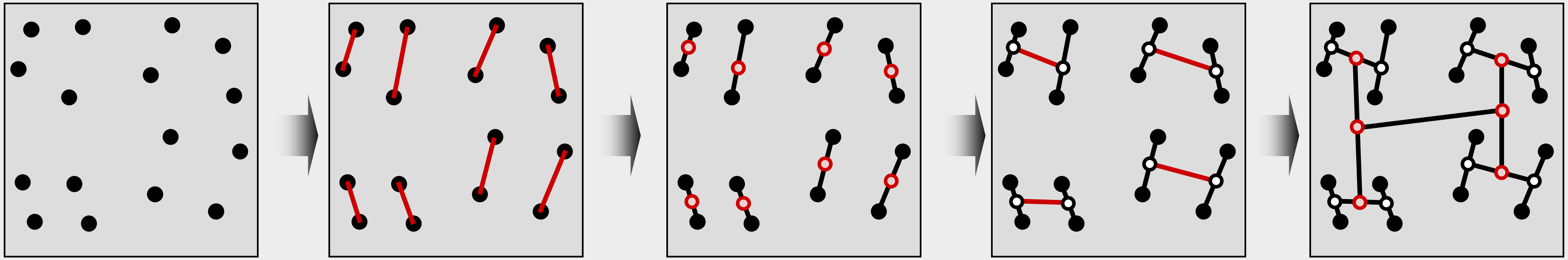
```
Input: set of sinks $S$, empty tree $T$
Output: clock tree $T$

if (|$S$| ≤ 1)
    return
$(x_0,y_0) = (x_c(S),y_c(S))$                        // center of mass for $S$
$(S_A,S_B)$ = PARTITION($S$)                         // median to determine $S_A$ and $S_B$
$(x_A,y_A) = (x_c(S_A),y_c(S_A))$                    // center of mass for $S_A$
$(x_B,y_B) = (x_c(S_B),y_c(S_B))$                    // center of mass for $S_B$
ROUTE($T,x_0,y_0,x_A,y_A$)                           // connect center of mass of $S$ to
ROUTE($T,x_0,y_0,x_B,y_B$)                           //   center of mass of $S_A$ and $S_B$
BASIC_MMM($S_A,T$)                                   // recursively route $S_A$
BASIC_MMM($S_B,T$)                                   // recursively route $S_B$
```

7.5.1 Constructing Trees with Zero Global Skew

Recursive Geometric Matching (RGM)

- RGM proceeds in a bottom-up fashion
  - Compare to MMM, which is a top-down algorithm
- Basic idea:
  - Recursively determine a minimum-cost geometric matching of *n* sinks
  - Find a set of *n* / 2 line segments that match *n* endpoints and minimize total length (subject to the matching constraint)
  - After each matching step, a balance or tapping point is found on each matching segment to preserve zero skew to the associated sinks
  - The set of *n* / 2 tapping points then forms the input to the next matching step

Recursive Geometric Matching (RGM)



Set of *n* sinks *S*

Min-cost geometric matching

Find balance or tapping points (point that achieves zero skew in the subtree, not always midpoint)

Min-cost geometric matching

Final result after recursively performing RGM on each subset

7.5.1 Constructing Trees with Zero Global Skew

Recursive Geometric Matching (RGM)

Input: set of sinks *S*, empty tree *T*

Output: clock tree *T*

```
if (|$S$| ≤ 1)
    return
$M$ = min-cost geometric matching over $S$
$S'$ = Ø
foreach (<$P_i$,$P_j$ > ∈ $M$)
    $TP_i$ = subtree of $T$ rooted at $P_i$
    $TP_j$ = subtree of $T$ rooted at $P_j$
    $tp$ = tapping point on ($P_i$,$P_j$)        // point that minimizes the skew of
                                                 //   the tree $T_{tp}$ = $T_{Pi}$ U $T_{Pj}$ U ($P_i$,$P_j$)
    ADD($S'$,$tp$)                               // add $tp$ to $S'$
    ADD($T$,($P_i$,$P_j$))                       // add matching segment ($P_i$,$P_j$) to $T$
if (|$S$| % 2 == 1)                              // if |$S$| is odd, add unmatched node
    ADD($S'$, unmatched node)
RGM($S'$,$T$)                                    // recursively call RGM
```

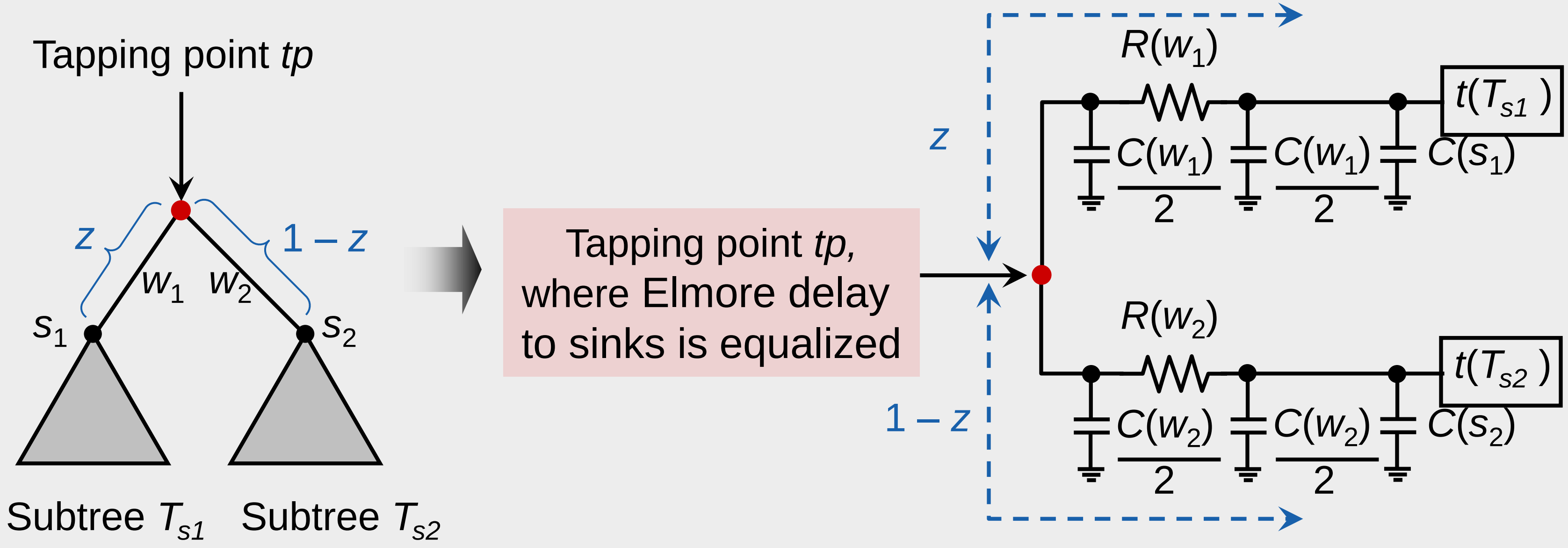
7.5.1 Constructing Trees with Zero Global Skew

Exact Zero Skew

- Adopts a bottom-up process of matching subtree roots and merging the corresponding subtrees, similar to RGM
- Two important improvements:
  - Finds exact zero-skew tapping points with respect to the Elmore delay model rather than the linear delay model
  - Maintains exact delay balance even when two subtrees with very different source-sink delays are matched (by wire elongation)

7.5.1 Constructing Trees with Zero Global Skew

Exact Zero Skew



Tapping point *tp*, where Elmore delay to sinks is equalized

7.5.1 Constructing Trees with Zero Global Skew

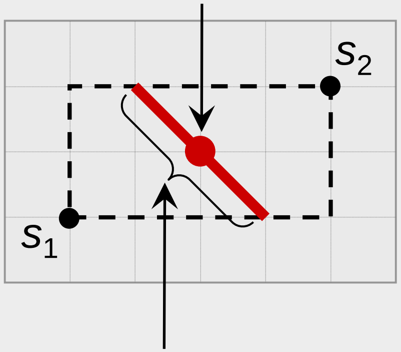
Deferred-Merge Embedding (DME)

- Defers the choice of merging (tapping) points for subtrees of the clock tree
- Needs a tree topology as input
- Weakness in earlier algorithms:
  - Determine locations of internal nodes of the clock tree too early; once a centroid is found, it is never changed
- Basic idea:
  - Two sinks in general position will have an infinite number of midpoints, creating a tilted line segment – Manhattan arc
  - Manhattan arc: same minimum wirelength and exact zero skew
  - Selection of embedding points for internal nodes on Manhattan arc will be delayed for as long as possible

7.5.1 Constructing Trees with Zero Global Skew

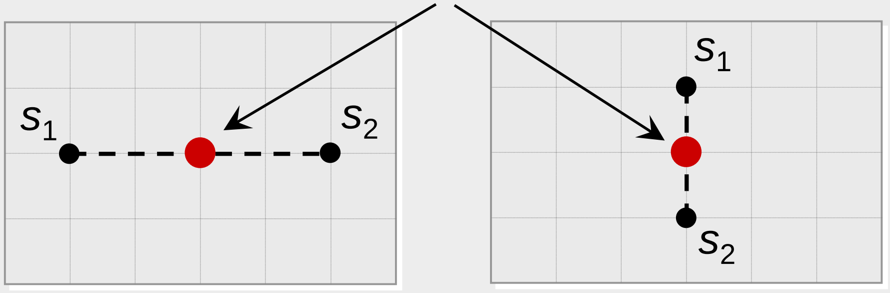
Deferred-Merge Embedding (DME)

Euclidean midpoint



Locus of all
Manhattan midpoints is
a Manhattan arc in the
Manhattan geometry

Euclidean midpoint



Sinks are aligned, hence, Manhattan arc
has zero length

7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

- Embeds internal nodes of the given topology *G* via a two-phase process

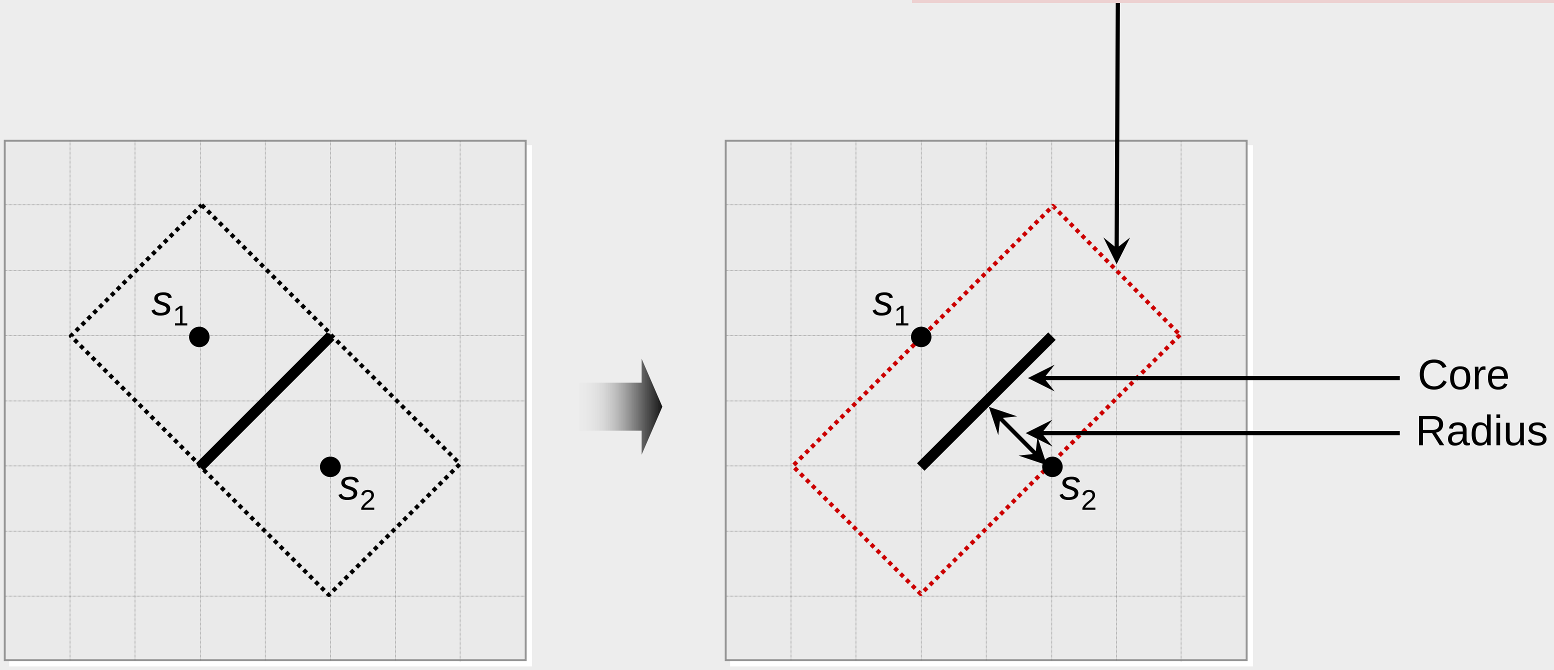
- First phase is bottom-up
  - Determines all possible locations of internal nodes of *G* consistent with a minimum-cost ZST *T*
  - Output: “tree of line segments”, with each line segment being the locus of possible placements of an internal node of *T*

- Second phase is top-down
  - Chooses the exact locations of all internal nodes in *T*
  - Output: fully embedded, minimum-cost ZST with topology *G*

7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Tilted Rectangular Region (TRR)
for the Manhattan arc of s_1 and s_2
with a radius of two units



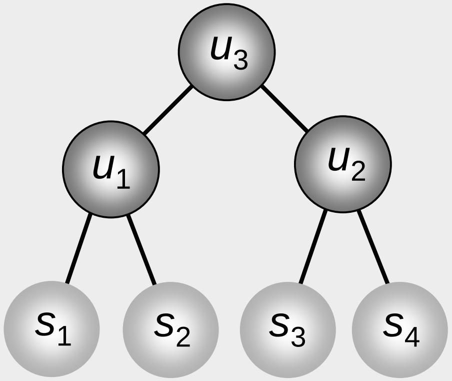
Core

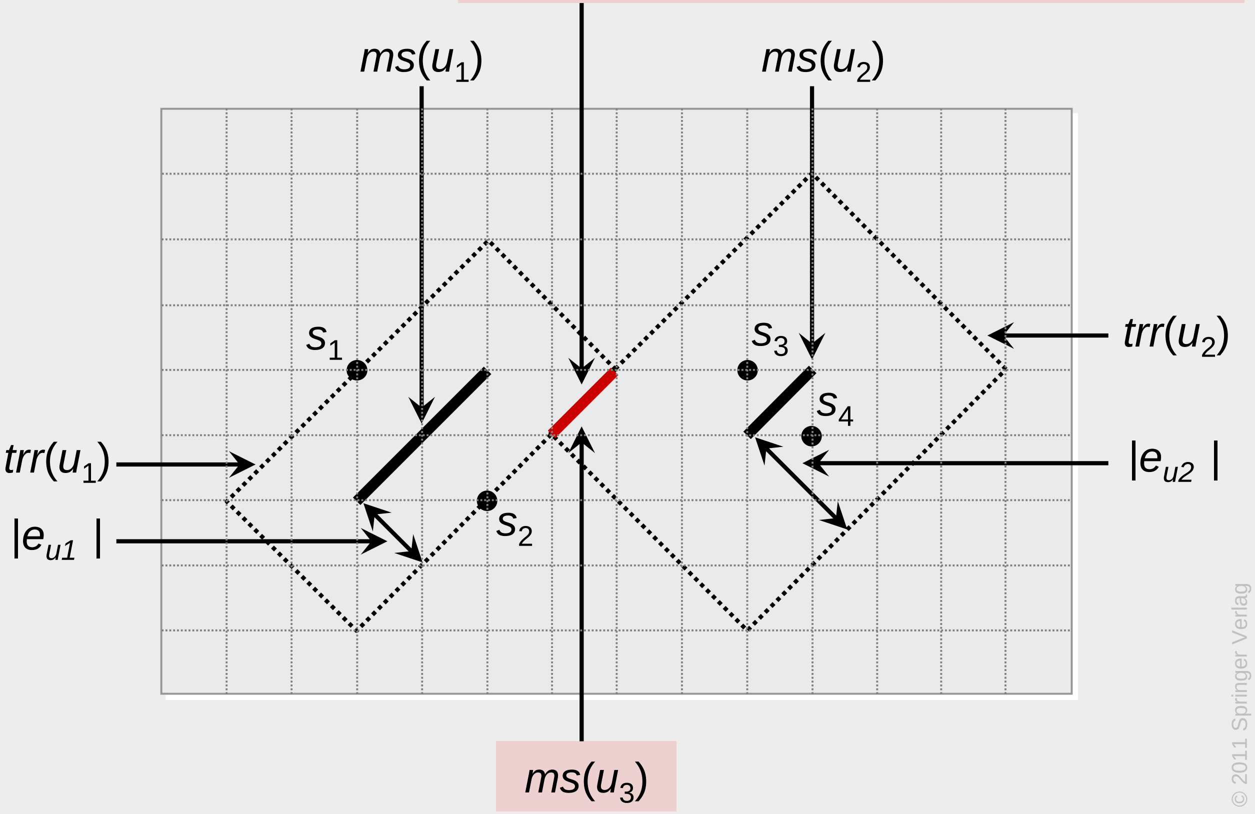
Radius

7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Merging segment for node u_3 (the parent of nodes u_1 and u_2) is the locus of feasible locations of u_3 with zero skew and minimum wirelength

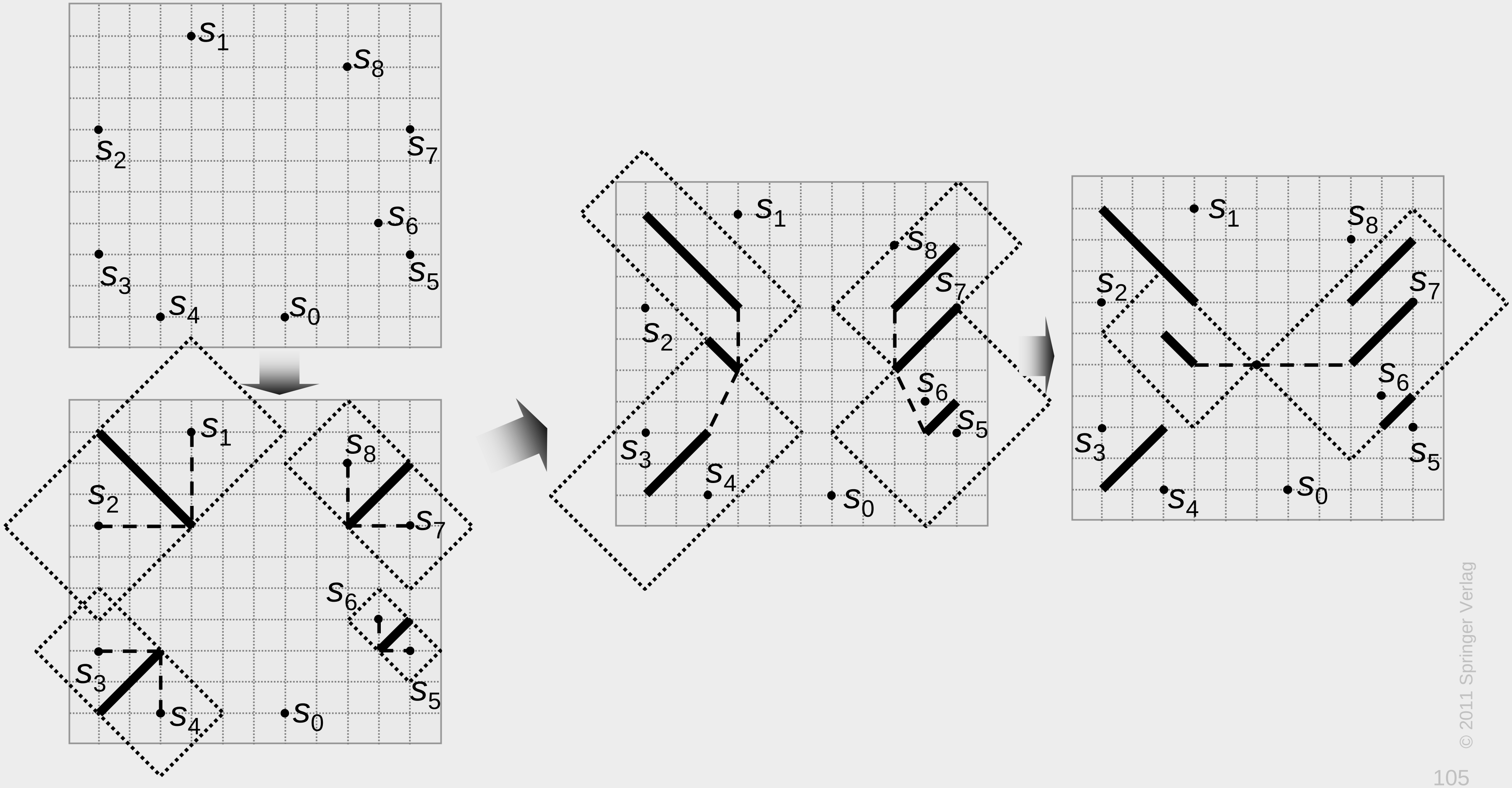




7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Build Tree of Segments Algorithm (DME Bottom-Up Phase)



7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Build Tree of Segments Algorithm (DME Bottom-Up Phase)

Input: set of sinks *S* and tree topology *G*(*S*,*Top*)

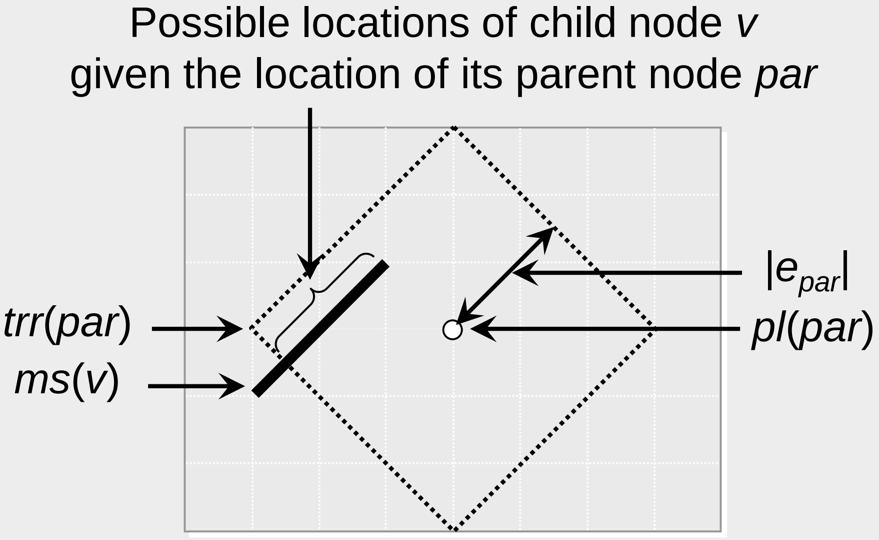
Output: merging segments *ms*(*v*) and edge lengths $|e_v|$, $v \in G$

```
foreach (node $v \in G$, in bottom-up order)
    if ($v$ is a sink node)                          // if $v$ is a terminal, then $ms(v)$ is a
        $ms[v] = PL(v)$                              //   zero-length Manhattan arc
    else                                             // otherwise, if $v$ is an internal node,
        $(a,b)$ = CHILDREN($v$)                      //   find $v$'s children and
        CALC_EDGE_LENGTH($e_a,e_b$)                  //   calculate the edge length
        $trr[a][core] = MS(a)$                       // create $trr(a)$ – find merging segment
        $trr[a][radius] = |e_a|$                     //   and radius of $a$
        $trr[b][core] = MS(b)$                       // create $trr(b)$ – find merging segment
        $trr[b][radius] = |e_b|$                     //   and radius of $b$
        $ms[v] = trr[a] \cap trr[b]$                 // merging segment of $v$
```

7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Find Exact Locations (DME Top-Down Phase)



Possible locations of child node *v* given the location of its parent node *par*

trr(*par*)

ms(*v*)

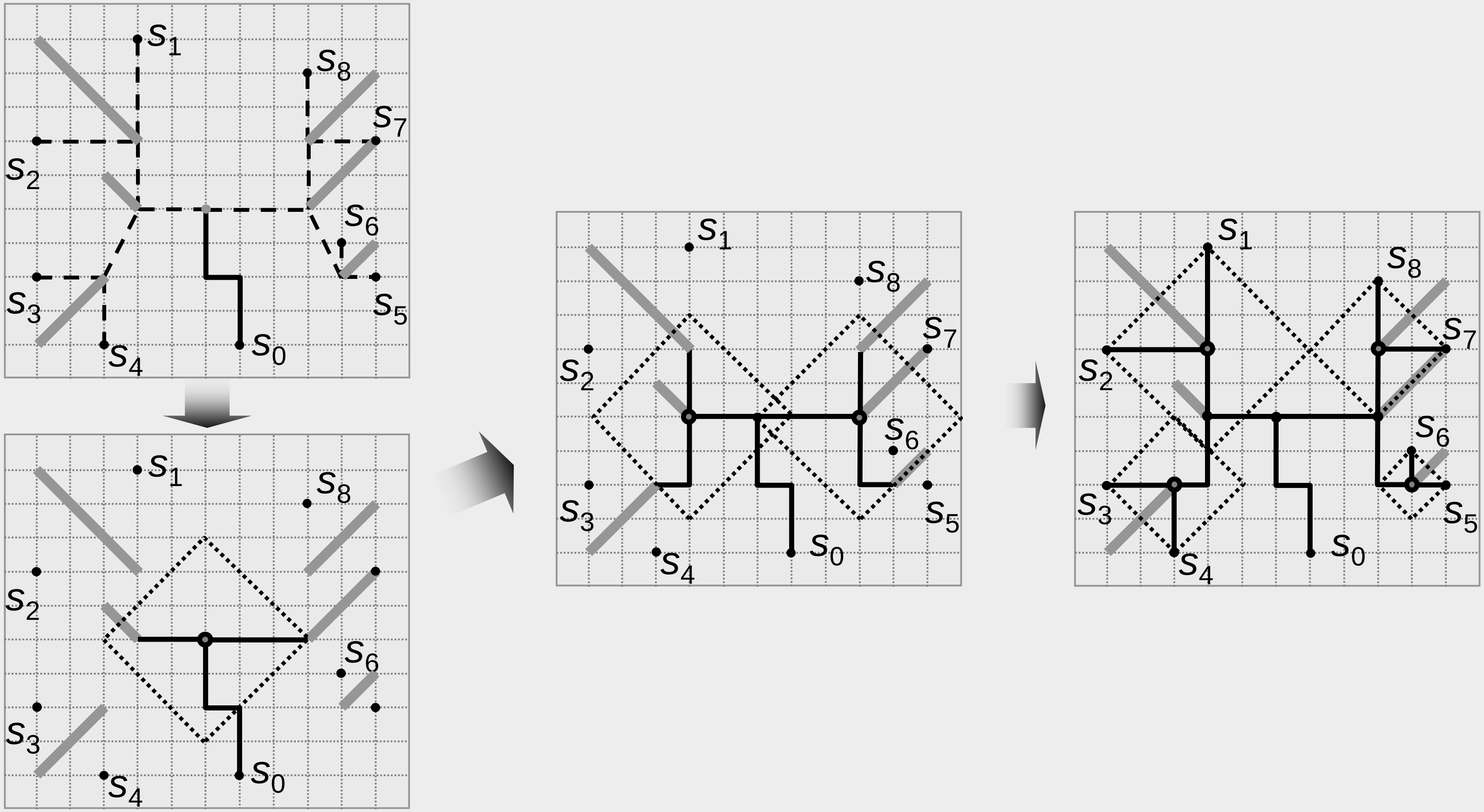
$|e_{par}|$

pl(*par*)

7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Find Exact Locations (DME Top-Down Phase)



7.5.1 Constructing Trees with Zero Global Skew

Deferred-Merge Embedding (DME)

Find Exact Locations (DME Top-Down Phase)

Input: set of sinks *S*, tree topology *G*, outputs of DME bottom-up phase

Output: minimum-cost zero-skew tree *T* with topology *G*

```
foreach (non-sink node $v \in G$ top-down order)
    if ($v$ is the root)
        loc = any point in ms(v)
    else
        par = PARENT(v)                 // par is the parent of v
        trr[par][core] = PL(par)        // create trr(par) – find merging segment
        trr[par][radius] = $|e_v|$      //   and radius of par
        loc = any point in ms[v] ∩ trr[par]
  pl[v] = loc
```

7.5.2 Clock Tree Buffering in the Presence of Variation

- To address challenging skew constraints, a clock tree undergoes several optimization steps, including
  - Geometric clock tree construction
  - Initial clock buffer insertion
  - Clock buffer sizing
  - Wire sizing
  - Wire snaking

- In the presence of process, voltage, and temperature variations, such optimizations require modeling the impact of variations
  - Variation model encapsulates the different parameters, such as width and thickness, of each library element as well-defined random variables

Summary of Chapter 7 – Area Routing

- Area routing: avoiding the division into global and detailed routing
  - Doing everything at once, subject to design rules
  - Small netlists with complicated constraints
  - Analog, MCM and PCB routing

- Manhattan vs Euclidean paths
  - Euclidean paths are no longer than Manhattan, usually shorter
  - Unique Euclidean shortest path
  - Multiple Manhattan paths
  - When Euclidean shortest paths intersect, there may exist Manhattan shortest paths that do not (not vice versa)

- Net ordering is important in area routing
  - Rule 1: nets with higher aspect ratio (less flexible) routed first
  - Rule 2: nets surrounded by other nets (more constrained) routed first
  - Rule 3: nets with more pins inside other net's bounding boxes routed first

Summary of Chapter 7 – Non-Manhattan Tree Routing

- Recall that Manhattan routing is dictated by the limitations of modern semiconductor manufacturing for thin wires

- PCB routing is not subject to those limitations
  - Can use shorter connections

- Non-Manhattan connections
  - Diagonal (45- or 60-degree) segments in addition to horizontal and vertical segments
  - Create more freedom to place Steiner points

- Octilinear Steiner Tree construction
  - Algorithms are generally adapted from the Manhattan case
  - Should produce results that are at least as good as the Manhattan case

Summary of Chapter 7 – Clock Network Routing

- Similar to signal-net routing, except for
  - Very large numbers of sinks
  - The need to equalize propagation delays from the root to sinks
  - Longer routes (to satisfy the equalization constraint)
  - Typical algorithms determine topology first, then geometric embedding

- Clock skew
  - Consider propagation delay from the root to each sink
  - Skew is the maximal pairwise difference between delays (over all pairs of sinks)
  - May be limited to sinks that are within distance $d > 0$ (local skew)

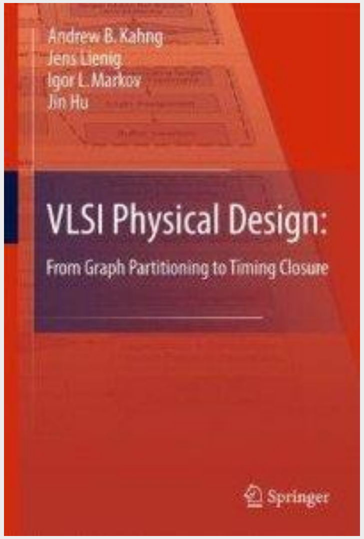
- For a specified wire delay model
  - ZST: Zero-Skew Tree routing requires that skew = *0*
  - BST: Bounded-Skew Tree routing requires that skew < *Bound*

Summary of Chapter 7 – Modern Clock Tree Synthesis

- Initial clock tree construction
  - Topology determination (MMM or RGM)
  - DME embedding (different flavors for ZST and BST)
  - Working with the Elmore delay model requires more effort than working with linear delay models
- Geometric obstacles (e.g., macros)
  - May require detours
  - Can be handled during DME (complicated) or during post-processing (often achieves as good results)
- Clock-tree optimization
  - Buffer insertion
  - Buffer sizing
  - Wire sizing
  - Wire snaking by small amounts
  - Decreasing the impact of process variability

VLSI Physical Design: From Graph Partitioning to Timing Closure

Chapter 8 – Timing Closure



Andrew B. Kahng
Jens Lienig
Igor L. Markov
Jin Hu

VLSI Physical Design:

From Graph Partitioning to Timing Closure

Springer

Original Authors:

Andrew B. Kahng, Jens Lienig, Igor L. Markov, Jin Hu

Chapter 8 – Timing Closure

8.1 Introduction

8.2 Timing Analysis and Performance Constraints

- 8.2.1 Static Timing Analysis
- 8.2.2 Delay Budgeting with the Zero-Slack Algorithm

8.3 Timing-Driven Placement

- 8.3.1 Net-Based Techniques
- 8.3.2 Embedding STA into Linear Programs for Placement

8.4 Timing-Driven Routing

- 8.4.1 The Bounded-Radius, Bounded-Cost Algorithm
- 8.4.2 Prim-Dijkstra Tradeoff
- 8.4.3 Minimization of Source-to-Sink Delay

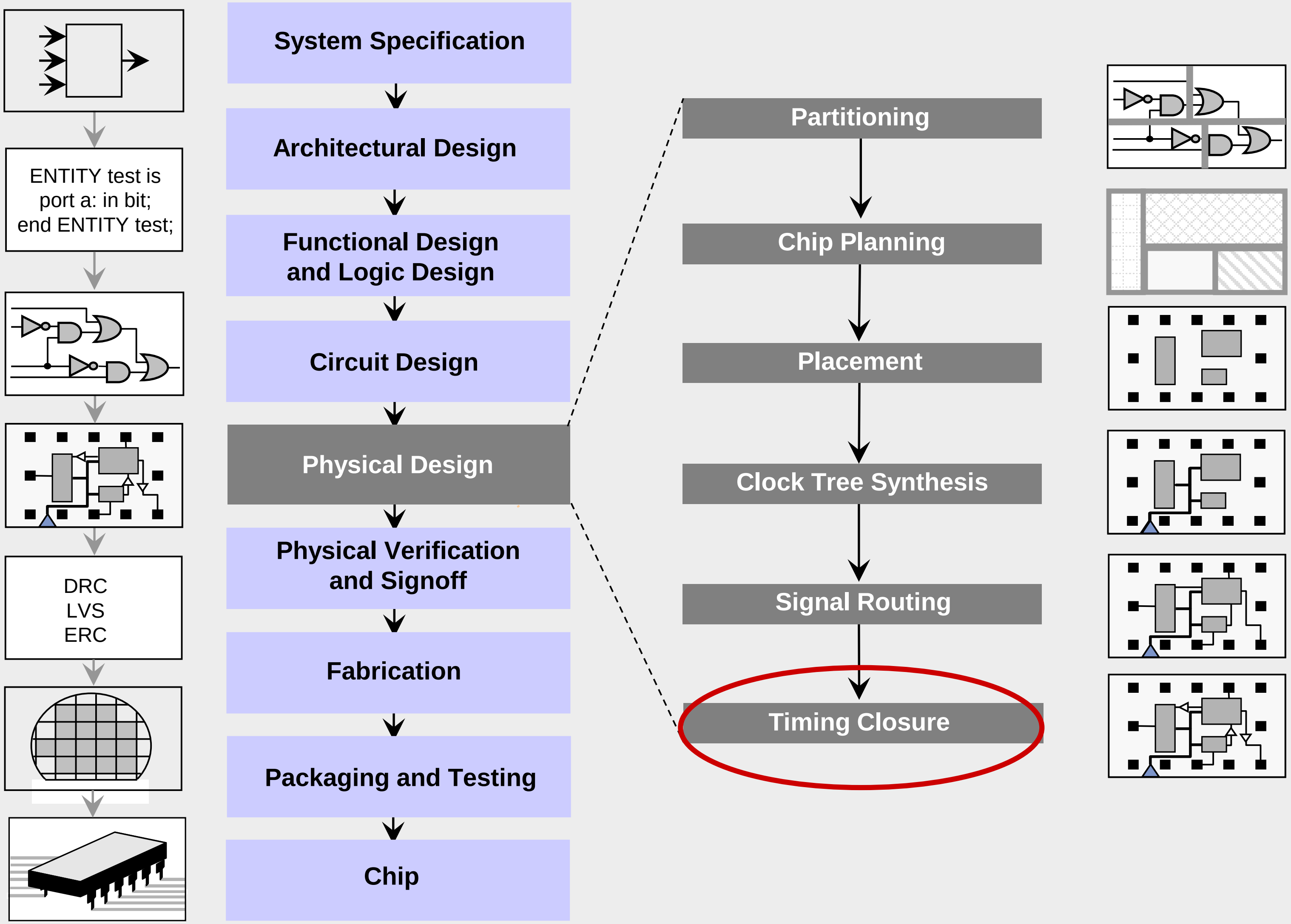
8.5 Physical Synthesis

- 8.5.1 Gate Sizing
- 8.5.2 Buffering
- 8.5.3 Netlist Restructuring

8.6 Performance-Driven Design Flow

8.7 Conclusions

8.1 Introduction



System Specification

Architectural Design

Functional Design and Logic Design

Circuit Design

Physical Design

Physical Verification and Signoff

Fabrication

Packaging and Testing

Chip

```
ENTITY test is
port a: in bit;
end ENTITY test;
```

DRC
LVS
ERC

Partitioning

Chip Planning

Placement

Clock Tree Synthesis

Signal Routing

Timing Closure

8.1 Introduction

- IC layout must satisfy geometric constraints, electrical constraints, power & thermal constraints as well as timing constraints
  - Setup (long-path) constraints
  - Hold (short-path) constraints

- Chip designers must complete timing closure
  - Optimization process that meets timing constraints
  - Integrates point optimizations discussed in previous chapters, e.g., placement and routing, with specialized methods to improve circuit performance

8.1 Introduction

Components of *timing closure* covered in this lecture:

- Timing-driven placement (Sec. 8.3) minimizes signal delays when assigning locations to circuit elements
- Timing-driven routing (Sec. 8.4) minimizes signal delays when selecting routing topologies and specific routes
- Physical synthesis (Sec. 8.5) improves timing by changing the netlist
  - Sizing transistors or gates: increasing the width:length ratio of transistors to decrease the delay or increase the drive strength of a gate
  - Inserting buffers into nets to decrease propagation delays
  - Restructuring the circuit along its critical paths
- Performance-driven physical design flow (Sec. 8.6)

8.1 Introduction

- Timing optimization engines must estimate circuit delays quickly and accurately to improve circuit timing

- Timing optimizers adjust propagation delays through circuit components, with the primary goal of satisfying timing constraints, including
  - Setup (long-path) constraints, which specify the amount of time a data input signal should be *stable* (steady) before the clock edge for each storage element (e.g., flip-flop or latch)
  - Hold-time (short-path) constraints, which specify the amount of time a data input signal should be stable after the clock edge at each storage element

$$t_{cycle} \geq t_{combDelay} + t_{setup} + t_{skew}$$

$$t_{combDelay} \geq t_{hold} + t_{skew}$$

8.1 Introduction

- Timing closure is the process of satisfying timing constraints through layout optimizations and netlist modifications
- Industry jargon: “the design has closed timing”

8.2 Timing Analysis and Performance Constraints

8.1 Introduction

8.2 Timing Analysis and Performance Constraints

8.2.1 Static Timing Analysis

8.2.2 Delay Budgeting with the Zero-Slack Algorithm

8.3 Timing-Driven Placement

8.3.1 Net-Based Techniques

8.3.2 Embedding STA into Linear Programs for Placement

8.4 Timing-Driven Routing

8.4.1 The Bounded-Radius, Bounded-Cost Algorithm

8.4.2 Prim-Dijkstra Tradeoff

8.4.3 Minimization of Source-to-Sink Delay

8.5 Physical Synthesis

8.5.1 Gate Sizing

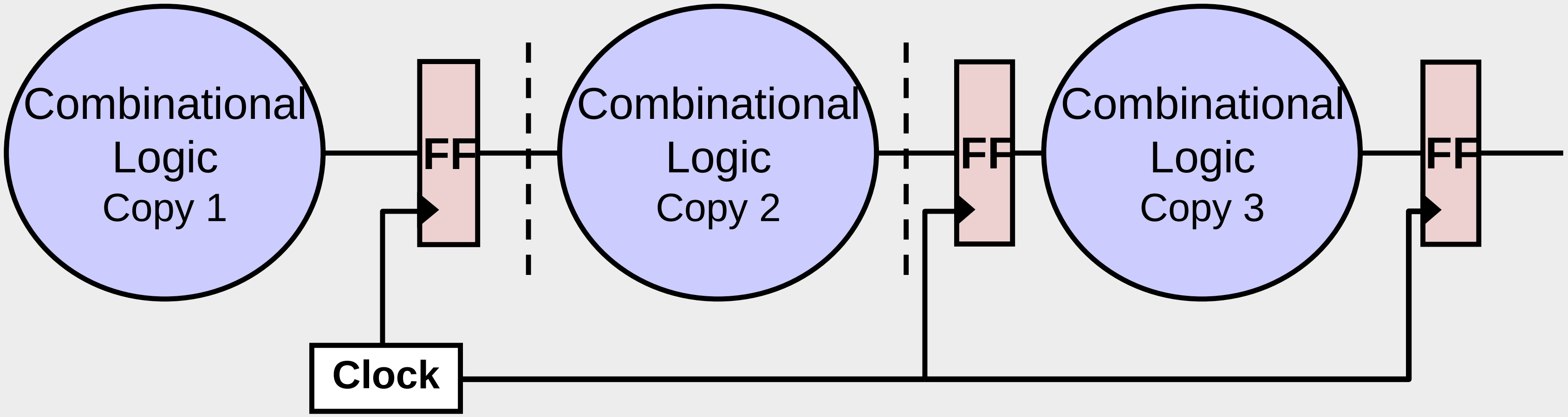
8.5.2 Buffering

8.5.3 Netlist Restructuring

8.6 Performance-Driven Design Flow

8.7 Conclusions

Sequential circuit, “unrolled” in time



Storage elements

Combinational logic

8.2 Timing Analysis and Performance Constraints

- Main delay concerns in sequential circuits
  - Gate delays are due to gate transitions
  - Wire delays are due to signal propagation along wires
  - Clock skew is due to the difference in time the sequential elements activate

- Need to quickly estimate sequential circuit timing
  - Perform static timing analysis (STA)
  - Assume clock skew is negligible, postpone until after clock network synthesis

8.2.1 Static Timing Analysis

- STA: assume worst-case scenario where every gate transitions

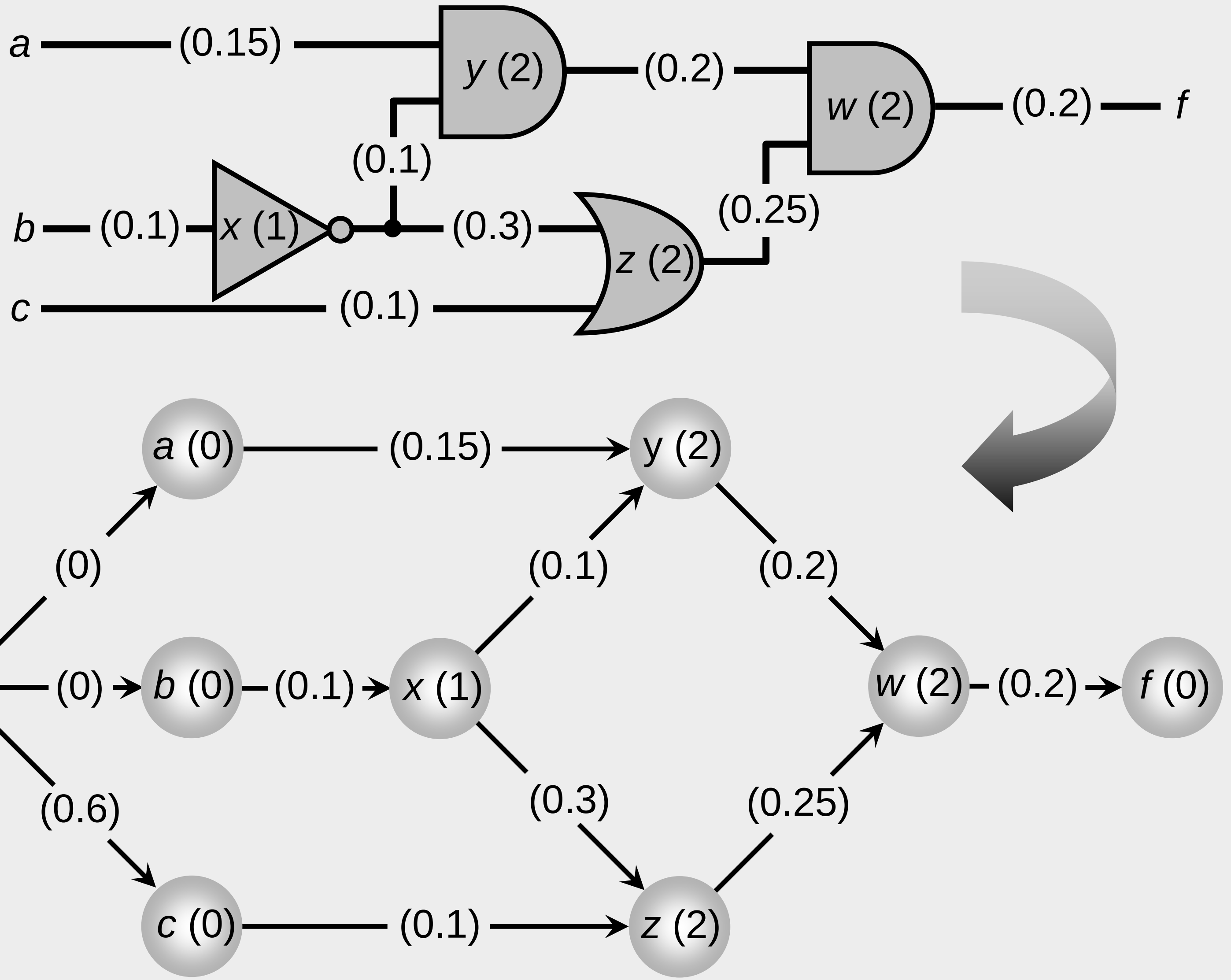
- Given combinational circuit, represent as directed acyclic graph (DAG)
  - Every edge (node) has weight = wire (gate) delay

- Compute the slack = RAT – AAT for each node
  - RAT is the required arrival time, latest time signal can transition
  - AAT is the actual arrival time
  - By convention, AAT is defined at the output of every node

⇒ Negative slack at any output means the circuit does not meet timing

⇒ Positive slack at all outputs means the circuit meets timing

Combinational circuit as DAG

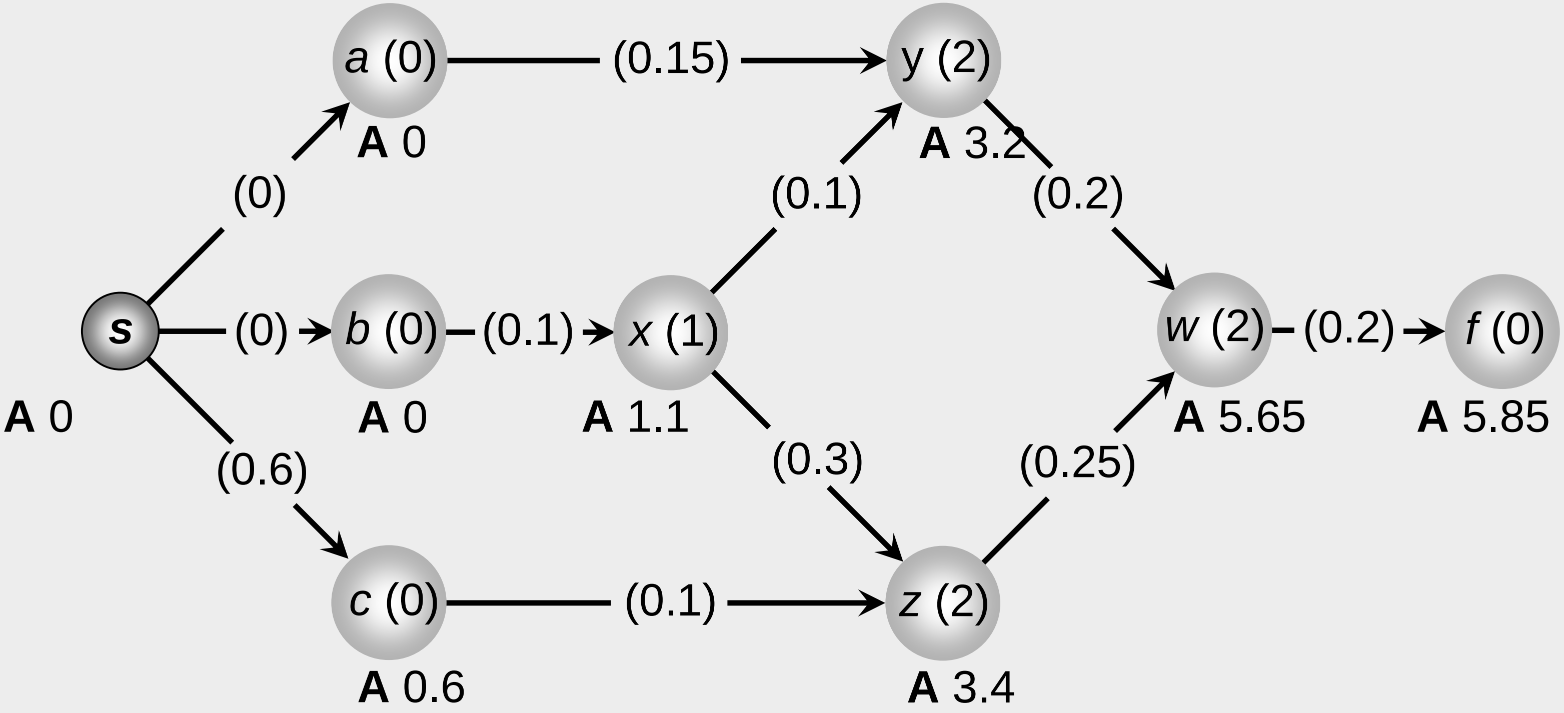


Compute AATs at each node:

$$AAT(v) = \max_{u \in FI(v)} \left(AAT(u) + t(u,v) \right)$$

where *FI*(*v*) is the fanin nodes, and *t*(*u*,*v*) is the delay between *u* and *v*

(AATs of inputs are given)



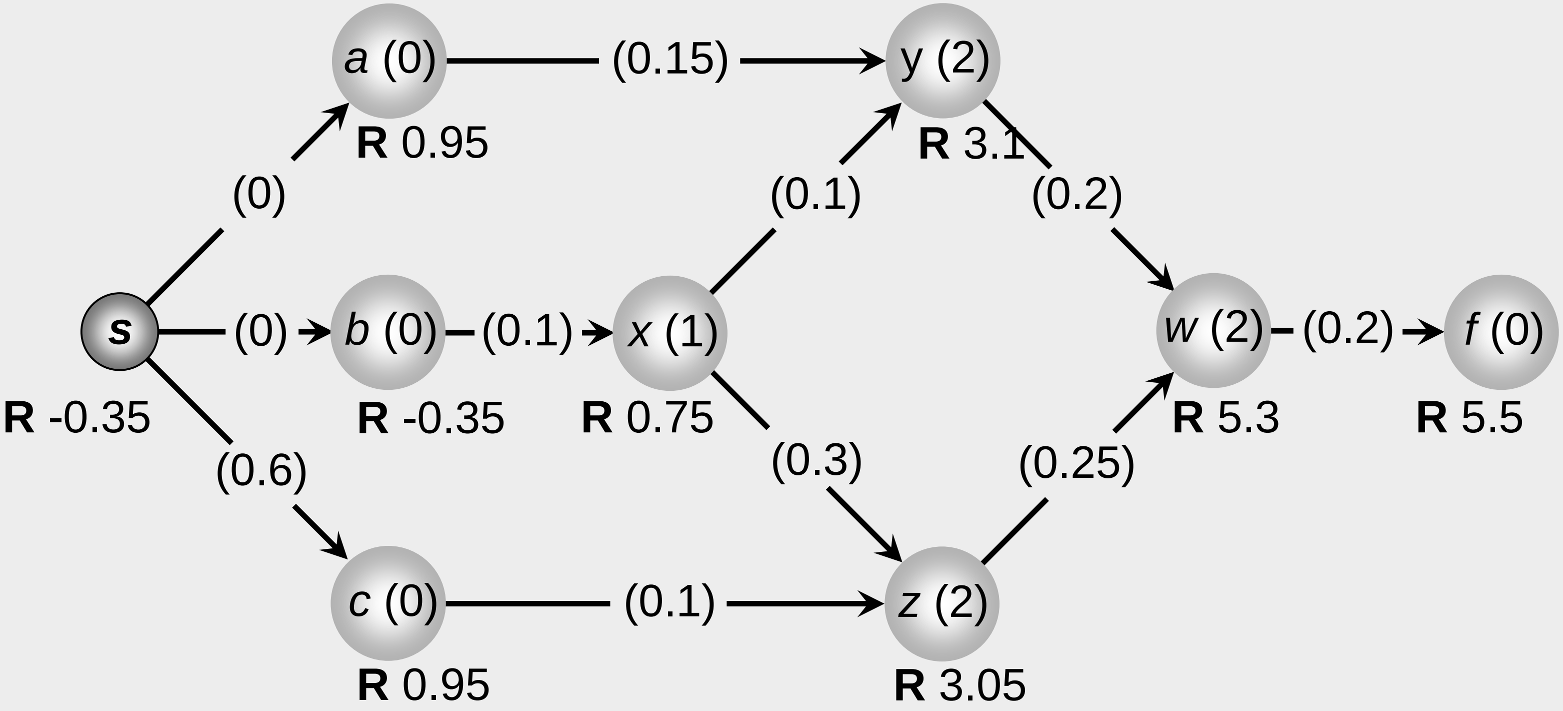
8.2.1 Static Timing Analysis

Compute RATs at each node:

$$RAT(v) = \min_{u \in FO(v)} \left(RAT(u) - t(u,v) \right)$$

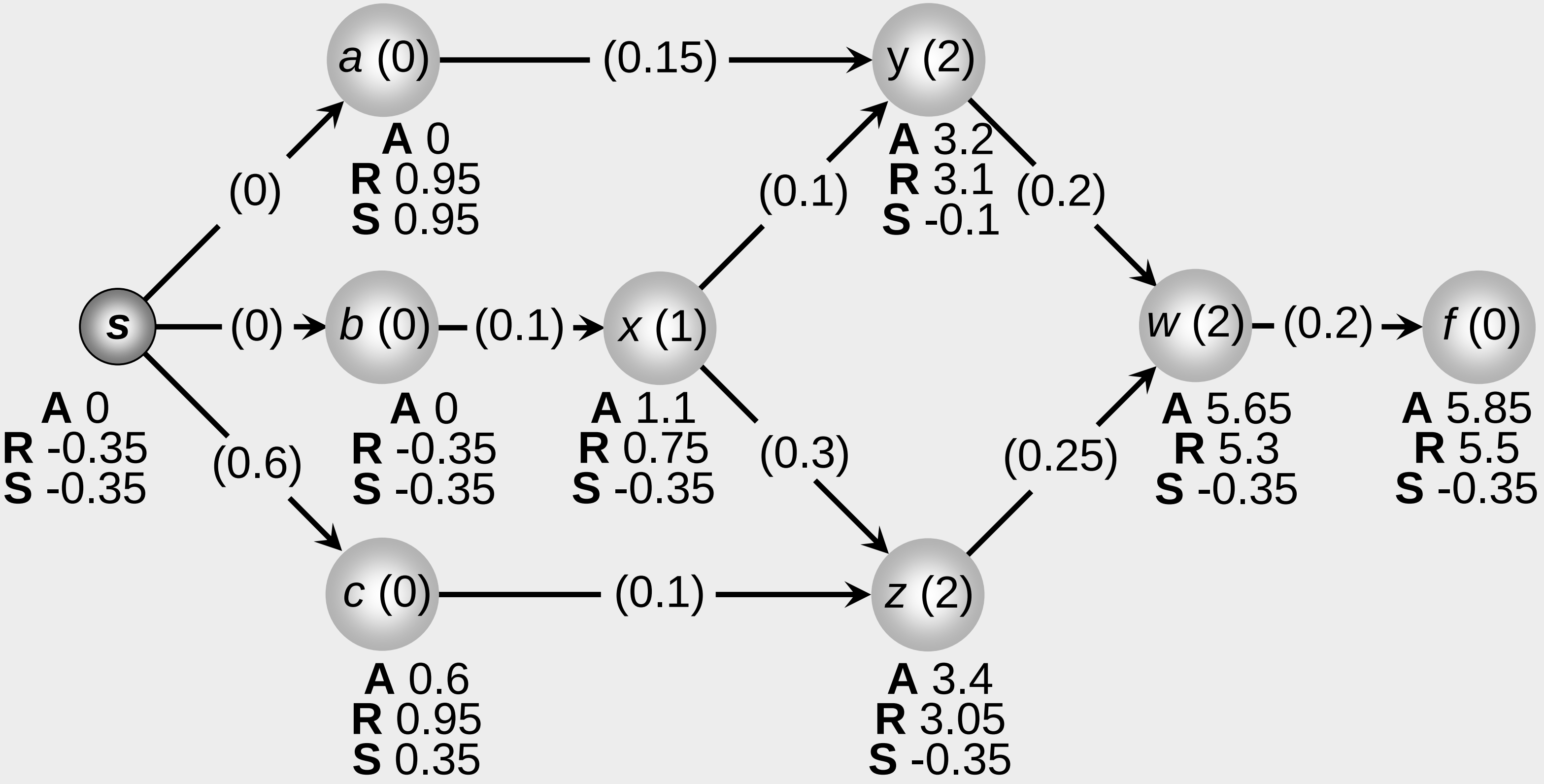
where *FO*(*v*) are the fanout nodes, and *t*(*u*,*v*) is the delay between *u* and *v*

(RATs of outputs are given)



Compute slacks at each node:

$$slack(v) = RAT(v) - AAT(v)$$



8.2.2 Delay Budgeting with the Zero-Slack Algorithm

- Establish timing budgets for nets
  - Gate and wire delays must be optimized during timing driven layout design
  - Wire delays depend on wire lengths
  - Wire lengths are not known until after placement and routing

- Delay budgeting with the zero-slack algorithm
  - Let v_i be the logic gates
  - Let e_i be the nets
  - Let DELAY(v) and DELAY(e) be the delay of the gate and net, respectively
  - Timing budget $TB(v)$ of a gate corresponds to DELAY(v) + DELAY(e)

8.2.2 Delay Budgeting with the Zero-Slack Algorithm

```
Input: timing graph $G(V,E)$
Output: timing budgets $TB$ for each $v \in V$
do
    $(AAT,RAT,slack)$ = STA($G$)
    foreach ($v_i \in V$)
        $TB[v_i]$ = DELAY($v_i$) + DELAY($e_i$)
    $slack_{min} = \infty$
    foreach ($v \in V$)
        if (($slack[v] < slack_{min}$) and ($slack[v] > 0$))
            $slack_{min} = slack[v]$
            $v_{min} = v$
    if ($slack_{min} \neq \infty$)
        $path = v_{min}$
        ADD_TO_FRONT($path$,BACKWARD_PATH($v_{min}$,$G$))
        ADD_TO_BACK($path$,FORWARD_PATH($v_{min}$,$G$))
        $s = slack_{min} / |path|$
        for ($i$ = 1 to $|path|$)
            $node = path[i]$                          // evenly distribute
            $TB[node] = TB[node] + s$                 // slack along path
while ($slack_{min} \neq \infty$)
```

8.2.2 Delay Budgeting with the Zero-Slack Algorithm

Forward Path Search (FORWARD_PATH(v_{min},*G*))

Input: node v_{min} with minimum slack $slack_{min}$, timing graph *G*

Output: maximal downstream path *path* from v_{min} such that no node $v \in V$ affects the slack of *path*

```
path = $v_{min}$
do
    flag = false
    node = LAST_ELEMENT(path)
    foreach (fanout node fo of node)
        if ((RAT[fo] == RAT[node] + TB[fo]) and (AAT[fo] == AAT[node] + TB[fo]))
            ADD_TO_BACK(path,fo)
            flag = true
            break
while (flag == true)
REMOVE_FIRST_ELEMENT(path)                                   // remove $v_{min}$
```

8.2.2 Delay Budgeting with the Zero-Slack Algorithm

Backward Path Search (BACKWARD_PATH(v_{min},G))

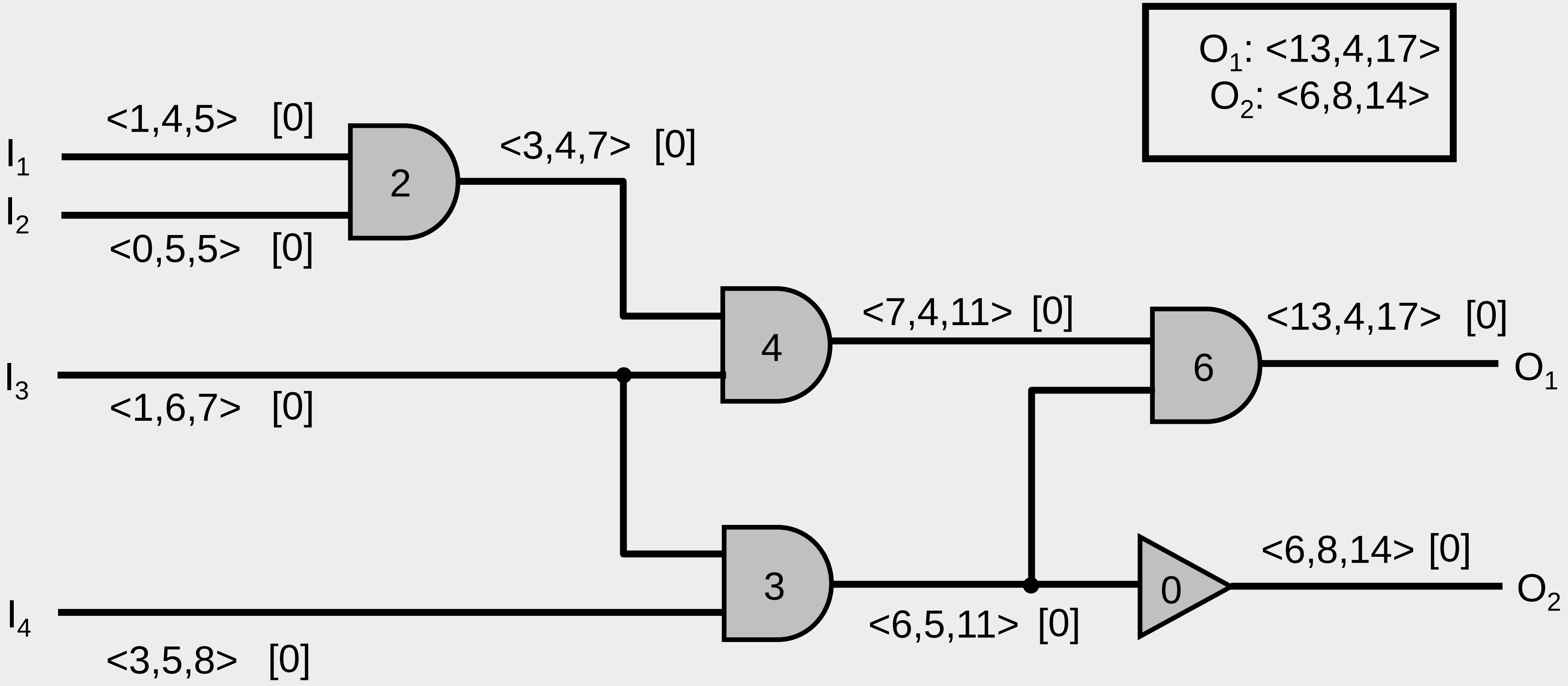
Input: node v_{min} with minimum slack $slack_{min}$, timing graph G

Output: maximal upstream path *path* from v_{min} such that no node $v \in V$ affects the slack of *path*

```
path = $v_{min}$
do
   flag = false
   node = FIRST_ELEMENT(path)
   foreach (fanin node fi of node)
      if ((RAT[fi] == RAT[node] – TB[fi]) and (AAT[fi] == AAT[node] – TB[fi]))
         ADD_TO_FRONT(path,fi)
         flag = true
         break
while (flag == true)
REMOVE_LAST_ELEMENT(path)                                  // remove $v_{min}$
```

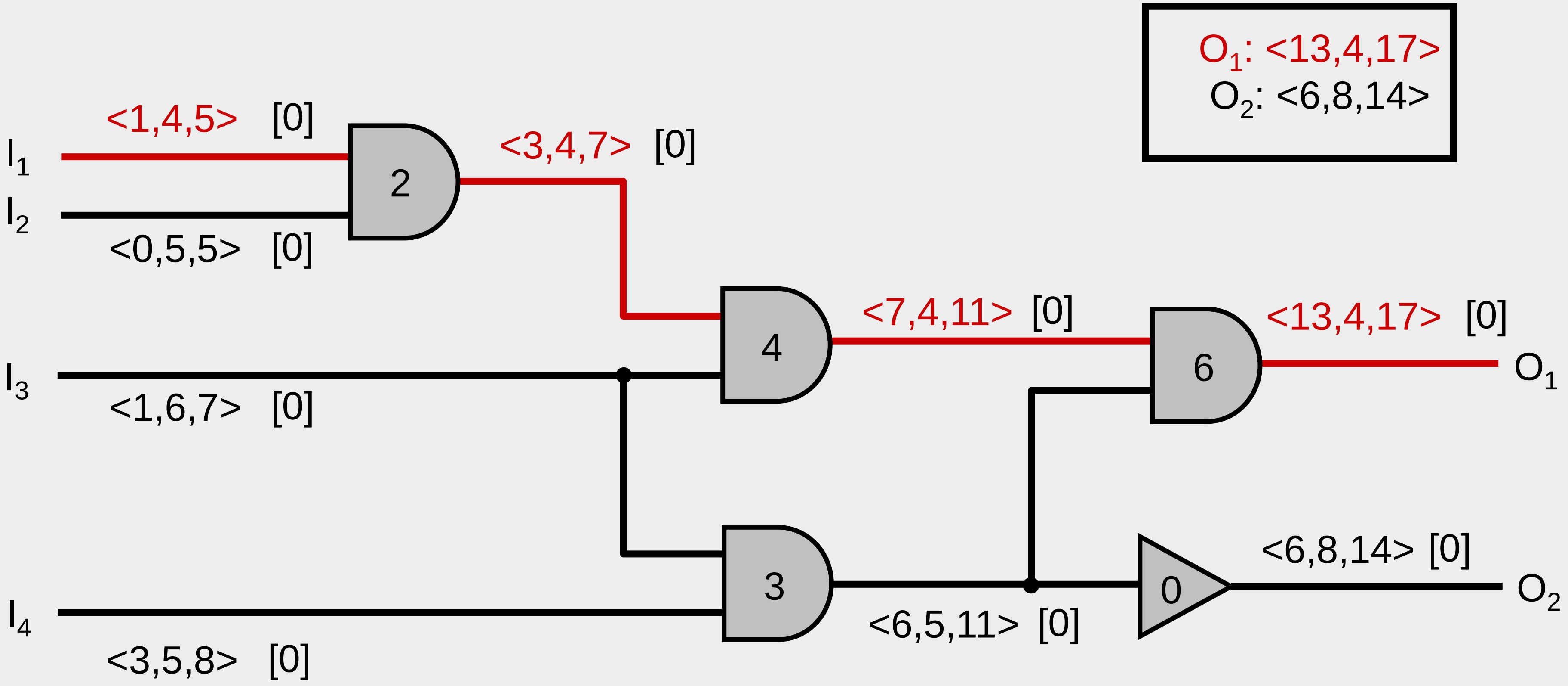
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
- Format: <AAT, Slack, RAT>, [timing budget]



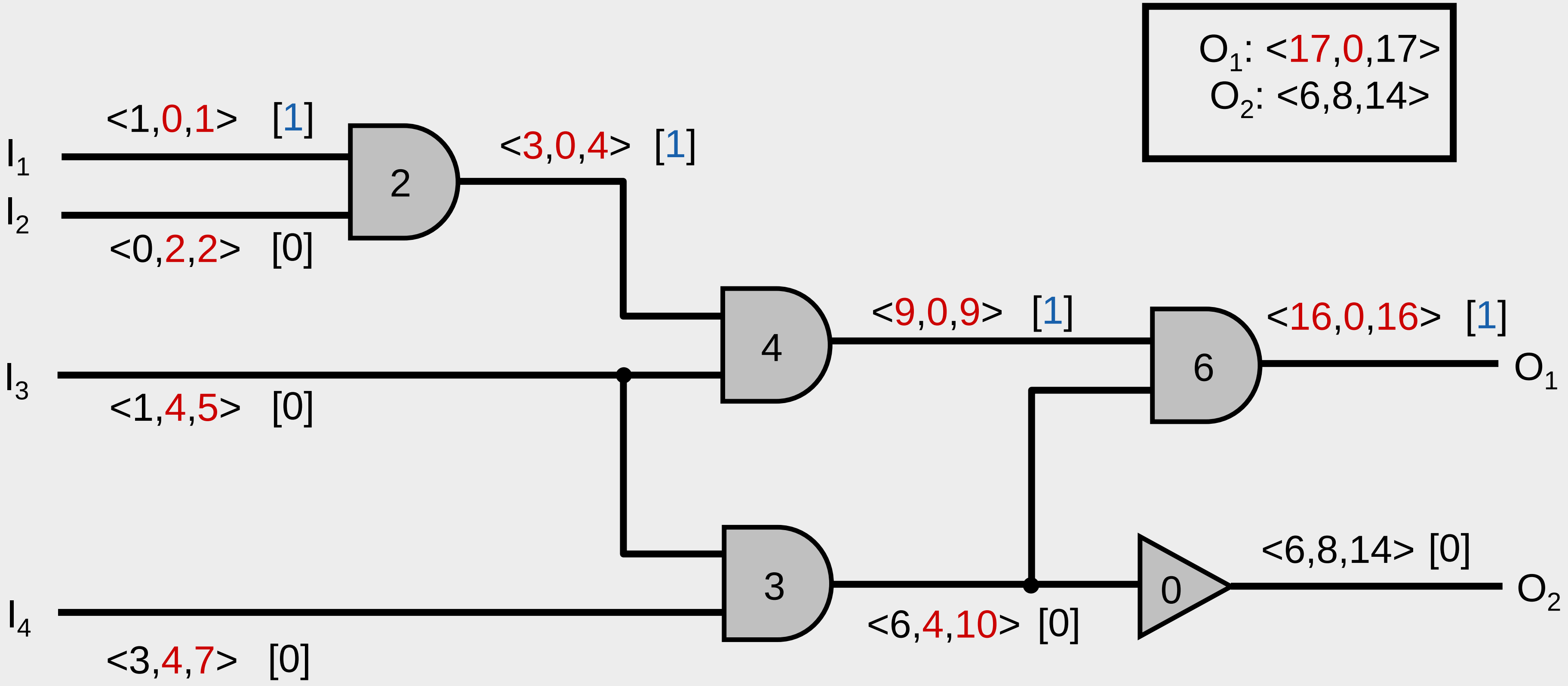
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
- Format: <AAT, Slack, RAT>, [timing budget]
- Find the path with the minimum nonzero slack



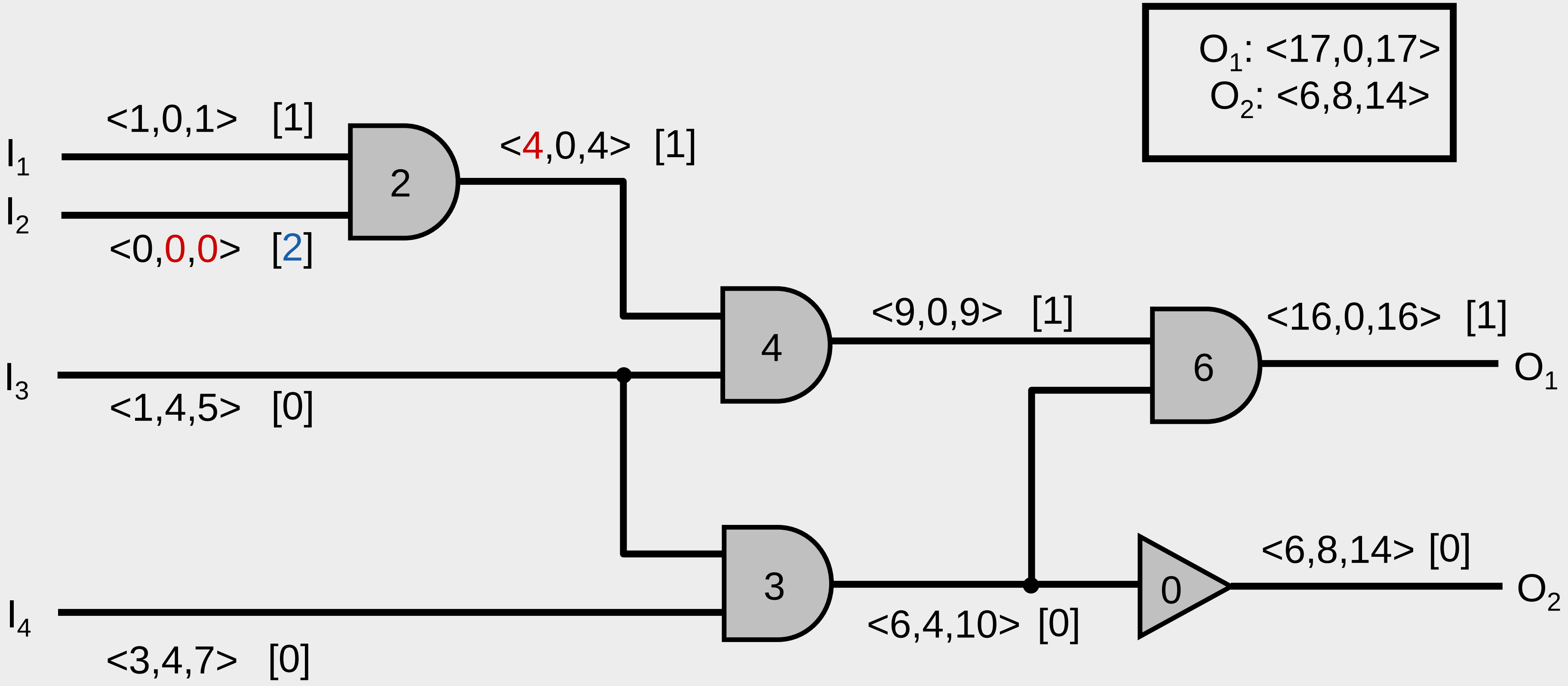
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
- Format: <AAT, Slack, RAT>, [timing budget]
- Find the path with the minimum slack
- Distribute the slacks and update the timing budgets



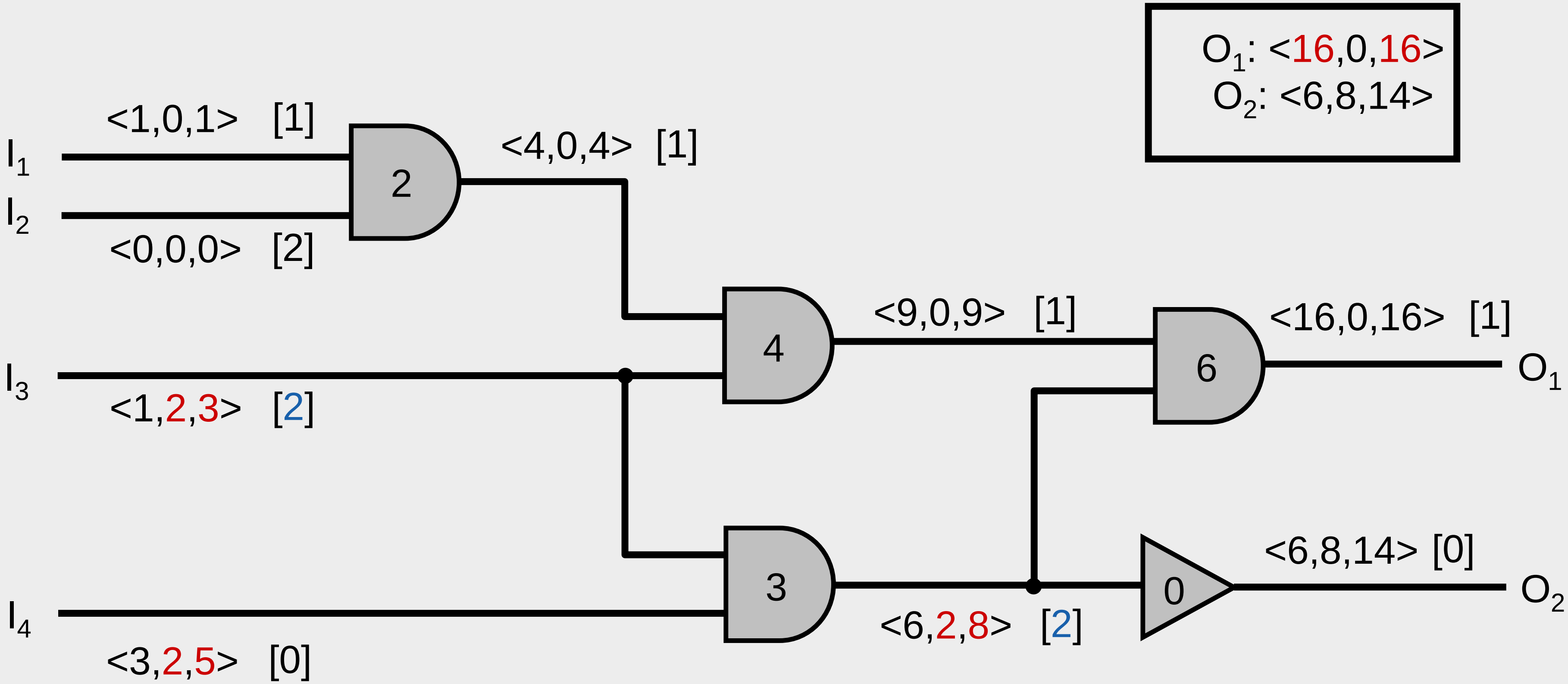
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
  - Format: <AAT, Slack, RAT>, [timing budget]
    - Find the path with the minimum slack
  - Distribute the slacks and update the timing budgets



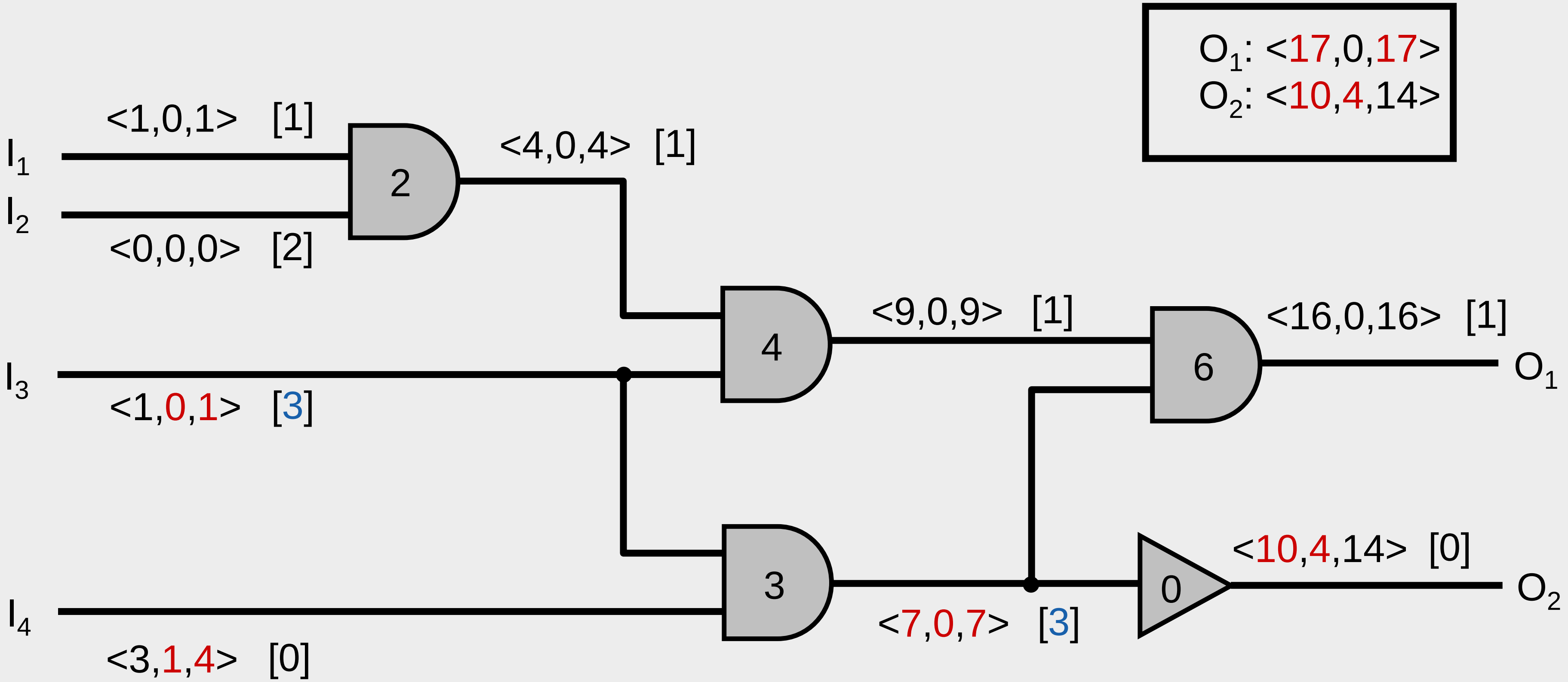
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
- Format: <AAT, Slack, RAT>, [timing budget]
- Find the path with the minimum slack
- Distribute the slacks and update the timing budgets



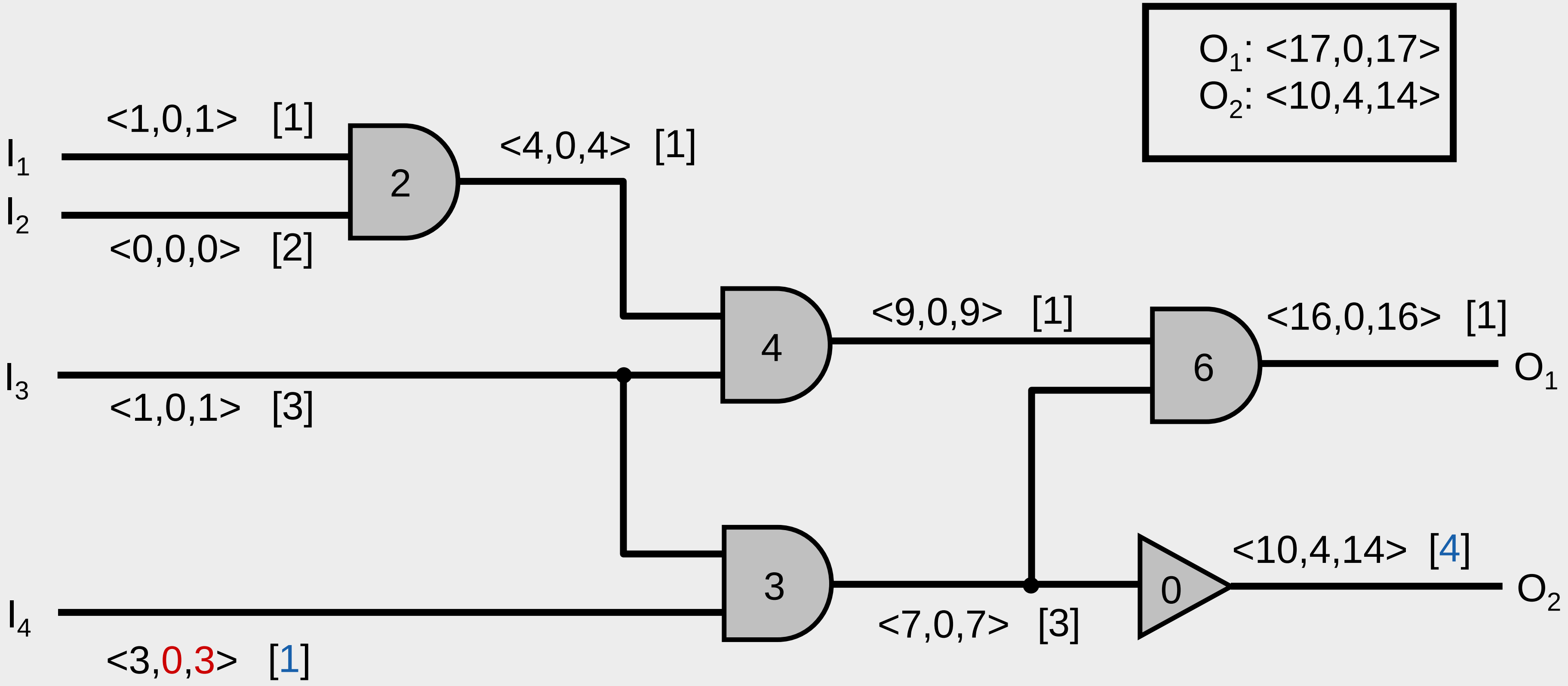
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
- Format: <AAT, Slack, RAT>, [timing budget]
- Find the path with the minimum slack
- Distribute the slacks and update the timing budgets



8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- Example: Use the zero-slack algorithm to distribute slack
- Format: <AAT, Slack, RAT>, [timing budget]
- Find the path with the minimum slack
- Distribute the slacks and update the timing budgets



8.3 Timing-Driven Placement

8.1 Introduction

8.2 Timing Analysis and Performance Constraints

8.2.1 Static Timing Analysis

8.2.2 Delay Budgeting with the Zero-Slack Algorithm

8.3 Timing-Driven Placement

8.3.1 Net-Based Techniques

8.3.2 Embedding STA into Linear Programs for Placement

8.4 Timing-Driven Routing

8.4.1 The Bounded-Radius, Bounded-Cost Algorithm

8.4.2 Prim-Dijkstra Tradeoff

8.4.3 Minimization of Source-to-Sink Delay

8.5 Physical Synthesis

8.5.1 Gate Sizing

8.5.2 Buffering

8.5.3 Netlist Restructuring

8.6 Performance-Driven Design Flow

8.7 Conclusions

8.3 Timing-Driven Placement

- Timing-driven placement optimizes circuit delay to satisfy timing constraints
- Let *T* be the set of all timing endpoints
- Constraint satisfaction is measured by worst negative slack (WNS)

$$WNS = \min_{\tau \in \mathrm{T}} \left(slack(\tau) \right)$$

- Or total negative slack (TNS)

$$TNS = \sum_{\tau \in \mathrm{T}, slack(\tau) < 0} slack(\tau)$$

- Classifications: net-based, path-based, integrated

8.3.1 Net-Based Techniques

- Net weights are added to each net – placer optimizes weighted wirelength

- Static net weights: computed before placement (never changes)

  - Discrete net weights: $w = \begin{cases} \omega_1 & \text{if } slack > 0 \\ \omega_2 & \text{if } slack \le 0 \end{cases}$ where $\omega_1 > 0$, $\omega_2 > 0$, and $\omega_2 > \omega_1$

  - Continuous net weights: $w = \left(1 - \frac{slack}{t}\right)^{\alpha}$ where t is the longest path delay and α is a criticality exponent

  - Based on net sensitivity to TNS and slack

$$w = w_o + \alpha(slack_{target} - slack) \cdot s_w^{SLACK} + \beta \cdot s_w^{TNS}$$

8.3.1 Net-Based Techniques

- Dynamic net weights: (re)computed during placement
  - Estimate slack at every iteration:

$$slack_k = slack_{k-1} - s_L^{DELAY} \cdot \Delta L$$

where ΔL is the change in wirelength

  - Update net criticality: $\upsilon_k = \begin{cases} \frac{1}{2}(\upsilon_{k-1} + 1) & \text{if among the top 3\% of critical nets} \\ \frac{1}{2}\upsilon_{k-1} & \text{otherwise} \end{cases}$
  - Update net weight: $w_k = w_{k-1} \cdot (1 + \upsilon_k)$

- Variations include updating every *j* iterations, different relations between criticality and net weight

8.3.2 Embedding STA into Linear Programs for Placement

- Construct a set of constraints for timing-driven placement
  - Physical constraints define locations of cells
  - Timing constraints define slack requirements

- Optimize an optimization objective
  - Improving worst negative slack (WNS)
  - Improving total negative slack (TNS)
  - Improving a combination of both WNS and TNS

8.3.2 Embedding STA into Linear Programs for Placement

- For physical constraints, let:
  - x_v and y_v be the center of cell $v \in V$
  - V_e be the set of cells connected to net $e \in E$
  - *left*(e), *right*(e), *bottom*(e), and *top*(e) respectively be the coordinates of the left, right, bottom, and top boundaries of e's bounding box
  - $\delta_x(v,e)$ and $\delta_y(v,e)$ be pin offsets from x_v and y_v for v's pin connected to e

8.3.2 Embedding STA into Linear Programs for Placement

- Then, for all $v \in V_e$:

$$\begin{aligned} left(e) &\le x_v + \delta_x(v,e) \\ right(e) &\ge x_v + \delta_x(v,e) \\ bottom(e) &\le y_v + \delta_y(v,e) \\ top(e) &\ge y_v + \delta_y(v,e) \end{aligned}$$

- Define *e*’s half-perimeter wirelength (HPWL):

$$L(e) = right(e) - left(e) + top(e) - bottom(e)$$

8.3.2 Embedding STA into Linear Programs for Placement

- For timing constraints, let
  - $t_{GATE}(v_i,v_o)$ be the gate delay from an input pin v_i to the output pin v_o for cell v
  - $t_{NET}(e,u_o,v_i)$ be net e's delay from cell u's output pin u_o to cell v's input pin v_i
  - $AAT(v_j)$ be the arrival time on pin j of cell v

8.3.2 Embedding STA into Linear Programs for Placement

- For every input pin v_i of cell v:

$$AAT(v_i) = AAT(u_o) + t_{NET}(u_o, v_i)$$

- For every output pin v_o of cell v:

$$AAT(v_o) \geq AAT(v_i) + t_{GATE}(v_i, v_o)$$

- For every pin τ_p in a sequential cell τ:

$$slack(\tau_p) \leq RAT(\tau_p) - AAT(\tau_p)$$

- Ensure that every $slack(\tau_p) \leq 0$

8.3.2 Embedding STA into Linear Programs for Placement

- Optimize for total negative slack:

$$\max : \sum_{\tau_p \in Pins(\tau), \tau \in \mathrm{T}} slack(\tau_p)$$

- Optimize for worst negative slack:

$$\max : WNS$$

- Optimize a linear combination of multiple parameters:

$$\min : \sum_{e \in E} L(e) - \alpha \cdot WNS$$

8.4 Timing-Driven Routing

8.1 Introduction

8.2 Timing Analysis and Performance Constraints

8.2.1 Static Timing Analysis

8.2.2 Delay Budgeting with the Zero-Slack Algorithm

8.3 Timing-Driven Placement

8.3.1 Net-Based Techniques

8.3.2 Embedding STA into Linear Programs for Placement

8.4 Timing-Driven Routing

8.4.1 The Bounded-Radius, Bounded-Cost Algorithm

8.4.2 Prim-Dijkstra Tradeoff

8.4.3 Minimization of Source-to-Sink Delay

8.5 Physical Synthesis

8.5.1 Gate Sizing

8.5.2 Buffering

8.5.3 Netlist Restructuring

8.6 Performance-Driven Design Flow

8.7 Conclusions

8.4 Timing-Driven Routing

- Timing-driven routing seeks to minimize:
  - Maximum sink delay: delay from the source to any sink in a net
  - Total wirelength: routed length of the net

- For a signal net *net*, let
  - s_0 be the source node
  - *sinks* = $\{s_1, \ldots, s_n\}$ be the sinks
  - $G = (V,E)$ be a corresponding weighted graph where:
  - $V = \{v_0, v_1, \ldots, v_n\}$ represents the source and sink nodes of *net*, and
  - the weight of an edge $e(v_i,v_j) \in E$ represents the routing cost between v_i and v_j

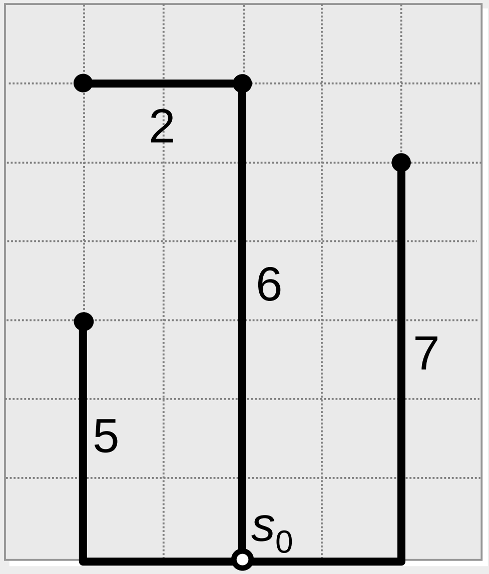
8.4 Timing-Driven Routing

- For any spanning tree *T* over *G*, let:
  - *radius*(*T*) be the length of the longest source-sink path in *T*
  - *cost*(*T*) be the total edge weight of *T*

- Trade off between "shallow" and "light" trees

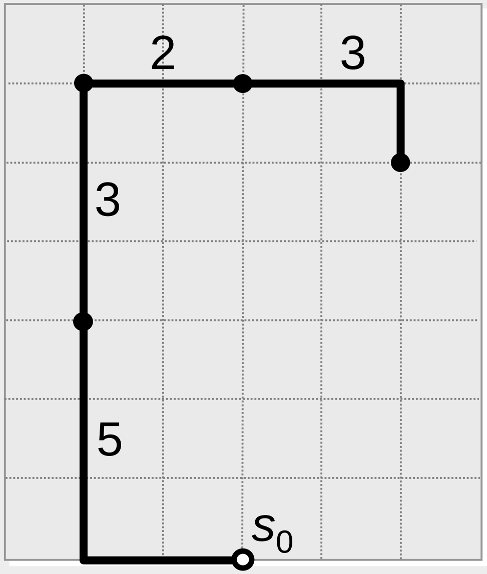
- "Shallow" trees have minimum radius
  - Shortest-paths tree
  - Constructed by Dijkstra's Algorithm

- "Light" trees have minimum cost
  - Minimum spanning tree (MST)
  - Constructed by Prim's Algorithm



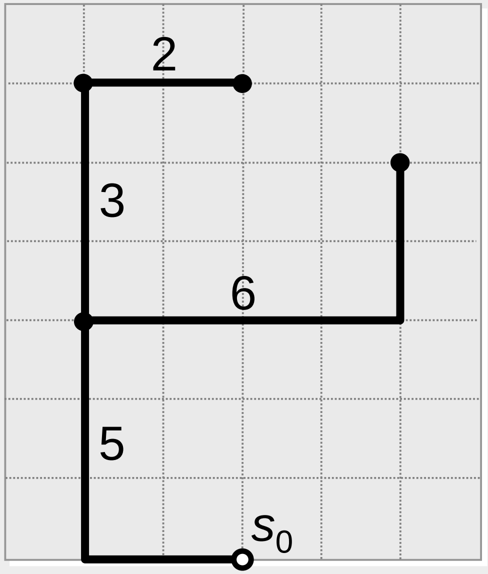
radius(*T*) = 8
cost(*T*) = 20

“Shallow”



radius(*T*) = 13
cost(*T*) = 13

“Light”



radius(*T*) = 11
cost(*T*) = 16

Tradeoff between shallow and light

8.4.1 The Bounded-Radius, Bounded-Cost Algorithm

- Trades off radius for cost by setting upper bounds on both

- In the bounded-radius, bounded-cost (BRBC) algorithm, let:
  - T_S be the shortest-paths tree
  - T_M be the minimum spanning tree

- T_{BRBC} is the tree constructed with parameter ε that satisfies:

$$radius\ (T_{BRBC}) \leq (1+\varepsilon) \cdot radius\ (T_S)$$

and

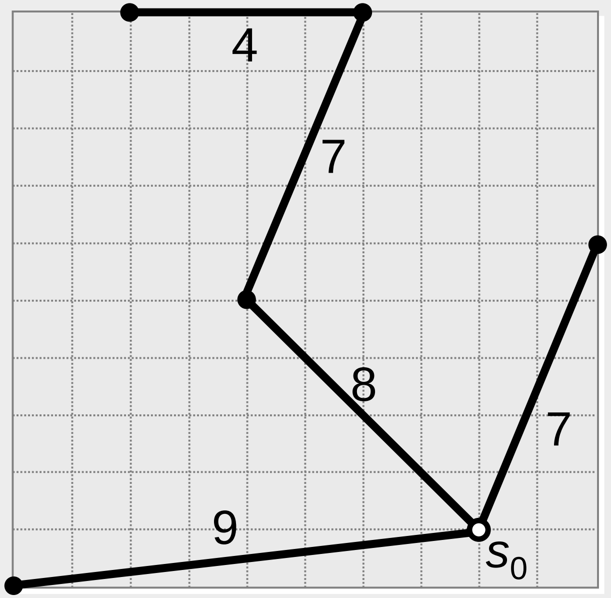
$$cost\ (T_{BRBC}) \leq \left(1+\frac{2}{\varepsilon}\right) \cdot cost\ (T_M)$$

- When $\varepsilon = 0$, T_{BRBC} has minimum radius
- When $\varepsilon = \infty$, T_{BRBC} has minimum cost

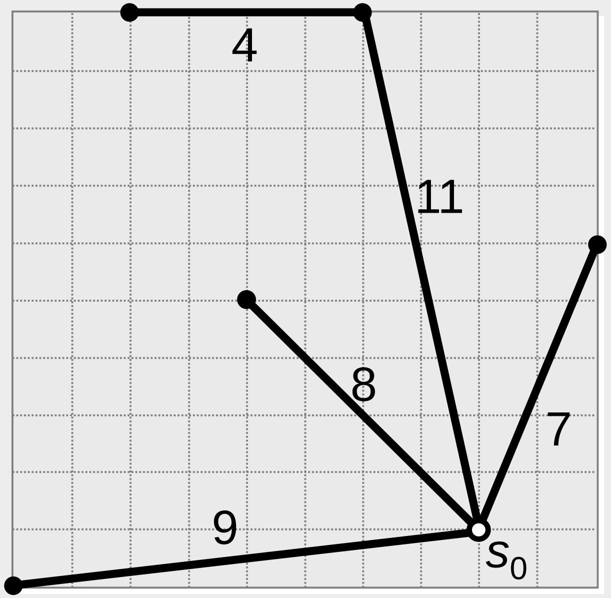
8.4.2 Prim-Dijkstra Tradeoff

- Prim-Dijkstra Tradeoff based on Prim's algorithm and Dijkstra's algorithm
- From the set of sinks *S*, iteratively add sink *s* based on different cost function
  - Prim's algorithm cost function: $cost(s_i, s_j)$
  - Dijkstra's algorithm cost function: $cost(s_0, s_i) + cost(s_i, s_j)$
  - Prim-Dijkstra Tradeoff cost function: $\gamma \cdot cost(s_0, s_i) + cost(s_i, s_j)$
- γ is a constant between 0 and 1

8.4.2 Prim-Dijkstra Tradeoff



radius(*T*) = 19
cost(*T*) = 35
γ = 0.25



radius(*T*) = 15
cost(*T*) = 39
γ = 0.75

8.4.3 Minimization of Source-to-Sink Delay

- Iteratively forms a tree by adding sinks, and optimizes for critical sink(s)

  - In the critical-sink routing tree (CSRT) problem, minimize:

$$\sum_{i=1}^{n} \alpha(i) \cdot t(s_0, s_i)$$

where α(*i*) are sink criticalities for sinks s_i, and $t(s_0,s_i)$ is the delay from s_0 to s_i

8.4.3 Minimization of Source-to-Sink Delay

- In the critical-sink Steiner tree problem, construct a minimum-cost Steiner tree T for all sinks except for the most critical sink s_c

- Add in the critical sink by:
  - H_0: a single wire from s_c to s_0
  - H_1: the shortest possible wire that can join s_c to T, so long as the path from s_0 to s_c is the shortest possible total length
  - H_{Best}: try all shortest connections from s_c to edges in T and from s_c to s_0. Perform timing analysis on each of these trees and pick the one with the lowest delay at s_c

8.5 Physical Synthesis

8.1 Introduction

8.2 Timing Analysis and Performance Constraints

- 8.2.1 Static Timing Analysis
- 8.2.2 Delay Budgeting with the Zero-Slack Algorithm

8.3 Timing-Driven Placement

- 8.3.1 Net-Based Techniques
- 8.3.2 Embedding STA into Linear Programs for Placement

8.4 Timing-Driven Routing

- 8.4.1 The Bounded-Radius, Bounded-Cost Algorithm
- 8.4.2 Prim-Dijkstra Tradeoff
- 8.4.3 Minimization of Source-to-Sink Delay

8.5 Physical Synthesis

- 8.5.1 Gate Sizing
- 8.5.2 Buffering
- 8.5.3 Netlist Restructuring

8.6 Performance-Driven Design Flow

8.7 Conclusions

8.5 Physical Synthesis

- Physical synthesis is a collection of timing optimizations to fix negative slack
- Consists of creating timing budgets and performing timing corrections
- Timing budgets include:
  - allocating target delays along paths or nets
  - often during placement and routing stages
  - can also be during timing correction operations
- Timing corrections include:
  - gate sizing
  - buffer insertion
  - netlist restructuring

8.5.1 Gate Sizing

- Let a gate *v* have 3 sizes *A*, *B*, *C*, where: $size\ (v_C) > size\ (v_B) > size\ (v_A)$
- Gate with a larger size has lower output resistance
- When load capacitances are large:

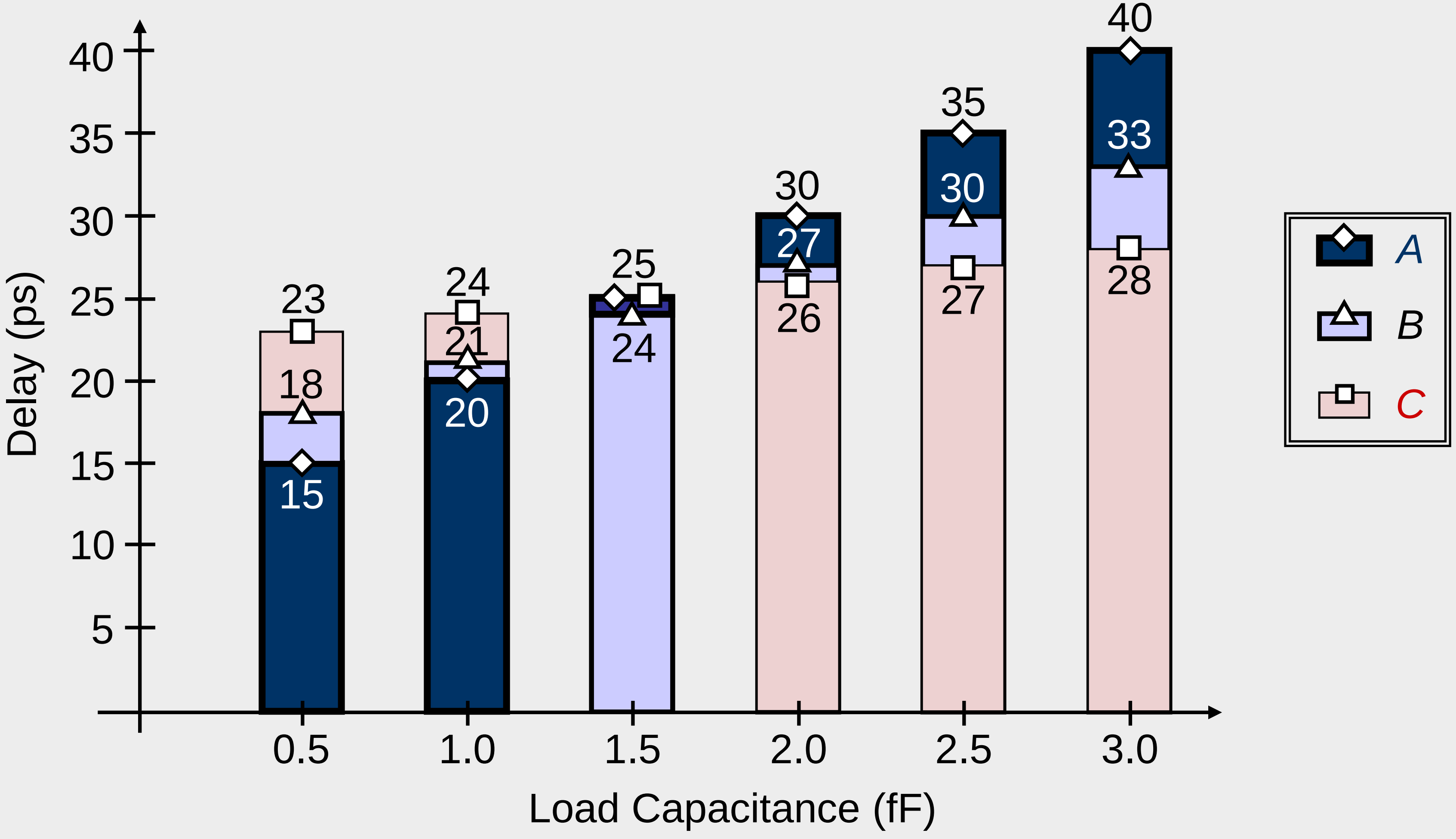
$$t(v_C) < t(v_B) < t(v_A)$$

- Gate with a smaller size has higher output resistance
- When load capacitances are small:

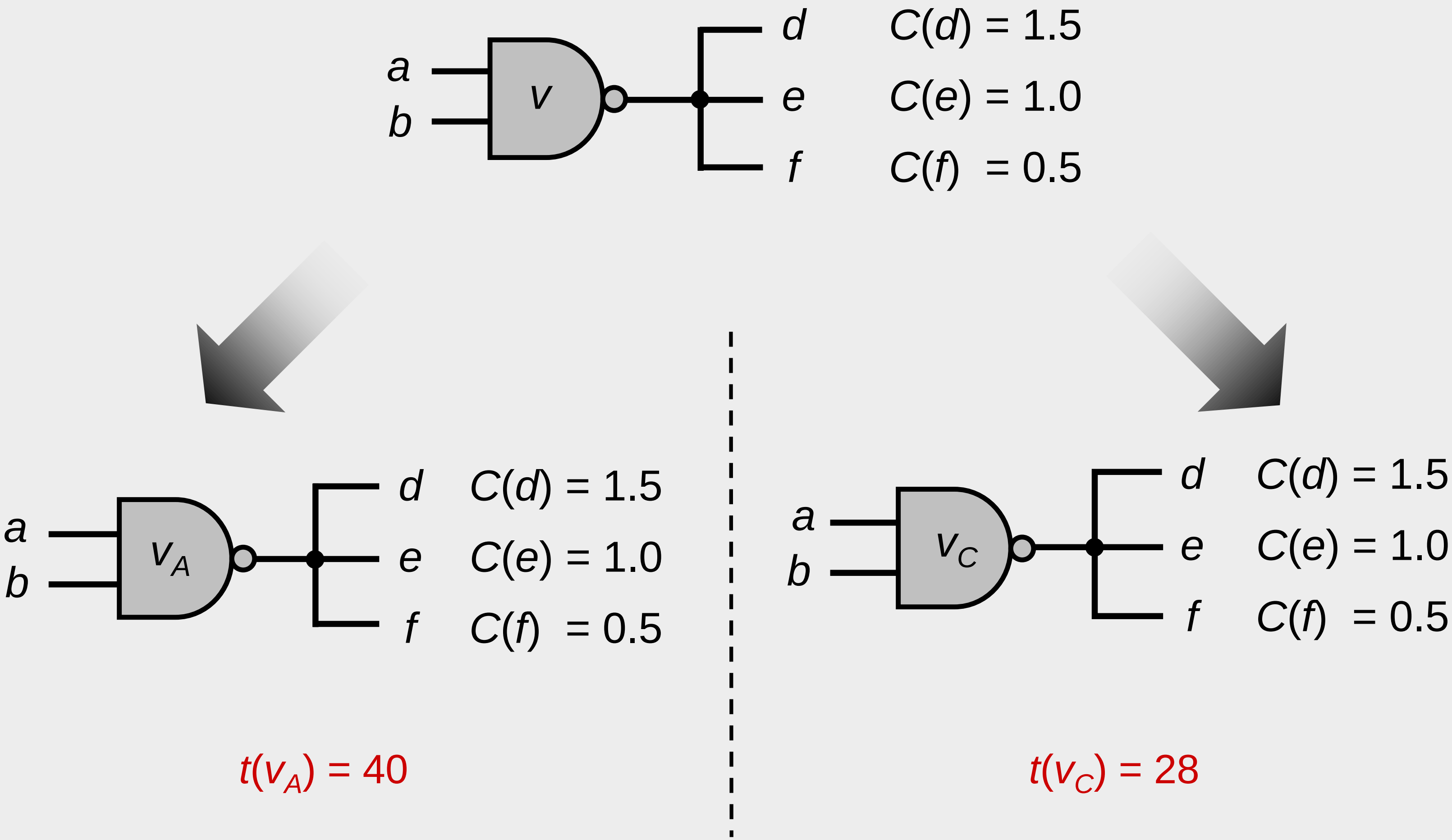
$$t(v_C) > t(v_B) > t(v_A)$$

8.5.1 Gate Sizing

- Let a gate *v* have 3 sizes *A*, *B*, *C*, where: $size(v_C) > size(v_B) > size(v_A)$



8.5.1 Gate Sizing



$C(d) = 1.5$

$C(e) = 1.0$

$C(f) = 0.5$

$C(d) = 1.5$

$C(e) = 1.0$

$C(f) = 0.5$

$t(v_A) = 40$

$C(d) = 1.5$

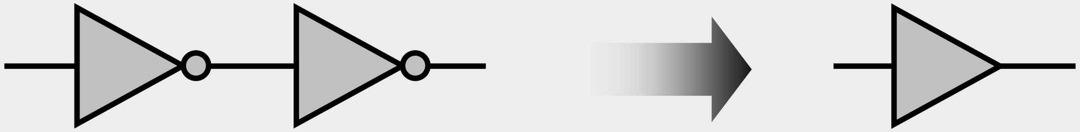
$C(e) = 1.0$

$C(f) = 0.5$

$t(v_C) = 28$

8.5.2 Buffering

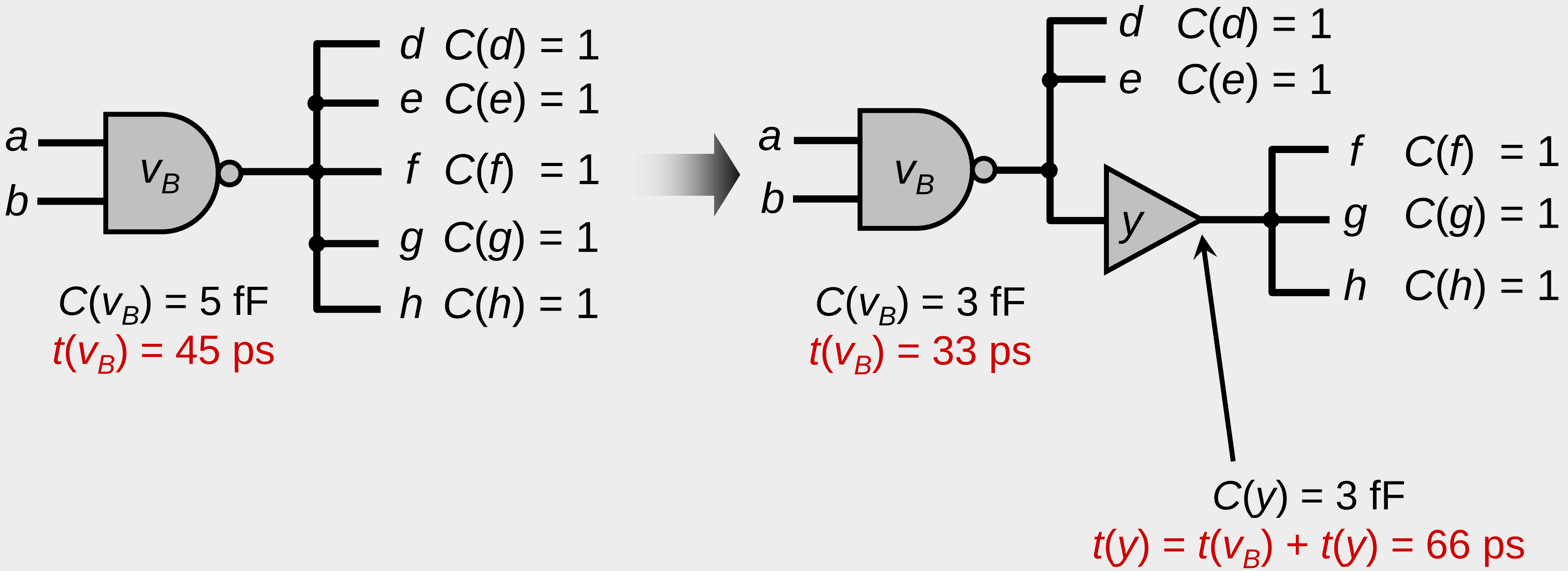
- Buffer: a series of two serially-connected inverters



- Improve delays by
  - speeding up the circuit or serving as delay elements
  - changing transition times
  - shielding capacitive load

- Drawbacks:
  - Increased area usage
  - Increased power consumption

8.5.2 Buffering



$C(d) = 1$

$C(e) = 1$

$C(f) = 1$

$C(g) = 1$

$C(h) = 1$

$C(v_B) = 5$ fF

$t(v_B) = 45$ ps

$C(d) = 1$

$C(e) = 1$

$C(f) = 1$

$C(g) = 1$

$C(h) = 1$

$C(v_B) = 3$ fF

$t(v_B) = 33$ ps

$C(y) = 3$ fF

$t(y) = t(v_B) + t(y) = 66$ ps

8.5.3 Netlist Restructuring

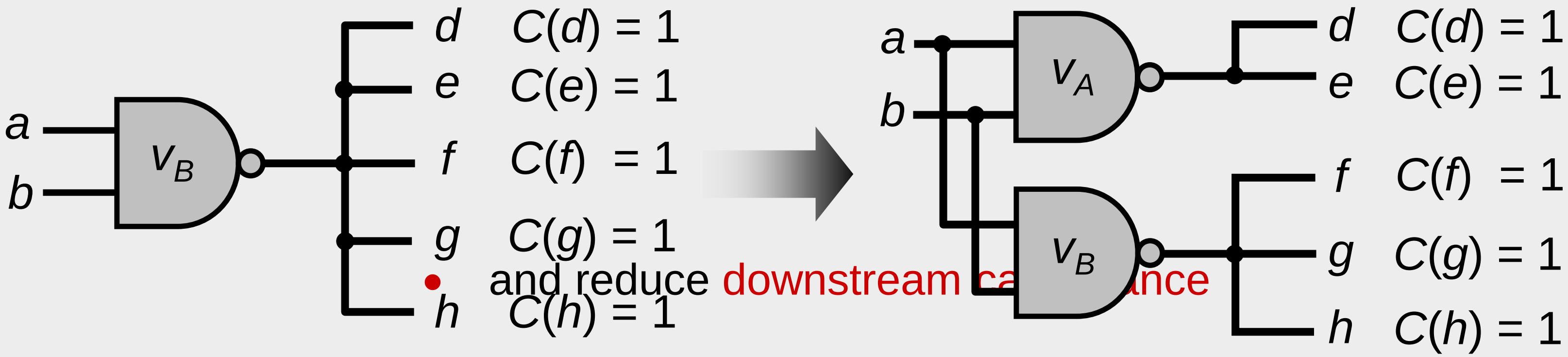
- Netlist restructuring only changes existing gates, does not change functionality

- Changes include
  - Cloning: duplicating gates
  - Redesign of fanin or fanout tree: changing the topology of gates
  - Swapping communicative pins: changing the connections
  - Gate decomposition: e.g., changing AND-OR to NAND-NAND
  - Boolean restructuring: e.g., applying Boolean laws to change circuit gates

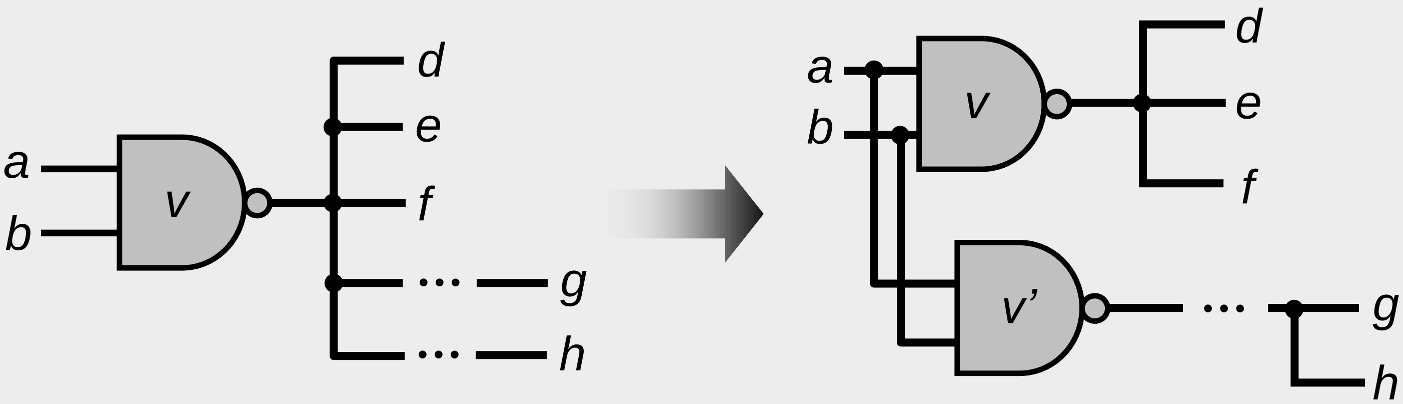
- Can also do reverse transformations of above, e.g., downsizing, merging

8.5.3 Netlist Restructuring

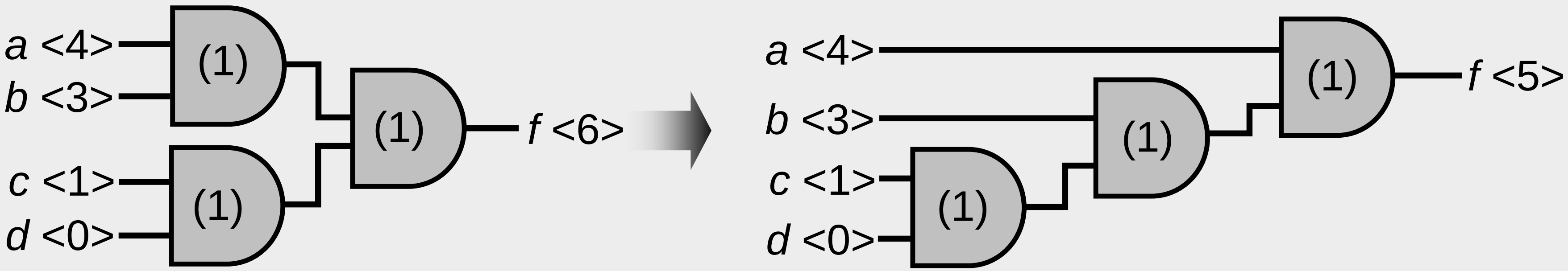
- Cloning can reduce fanout capacitance



- and reduce downstream capacitance

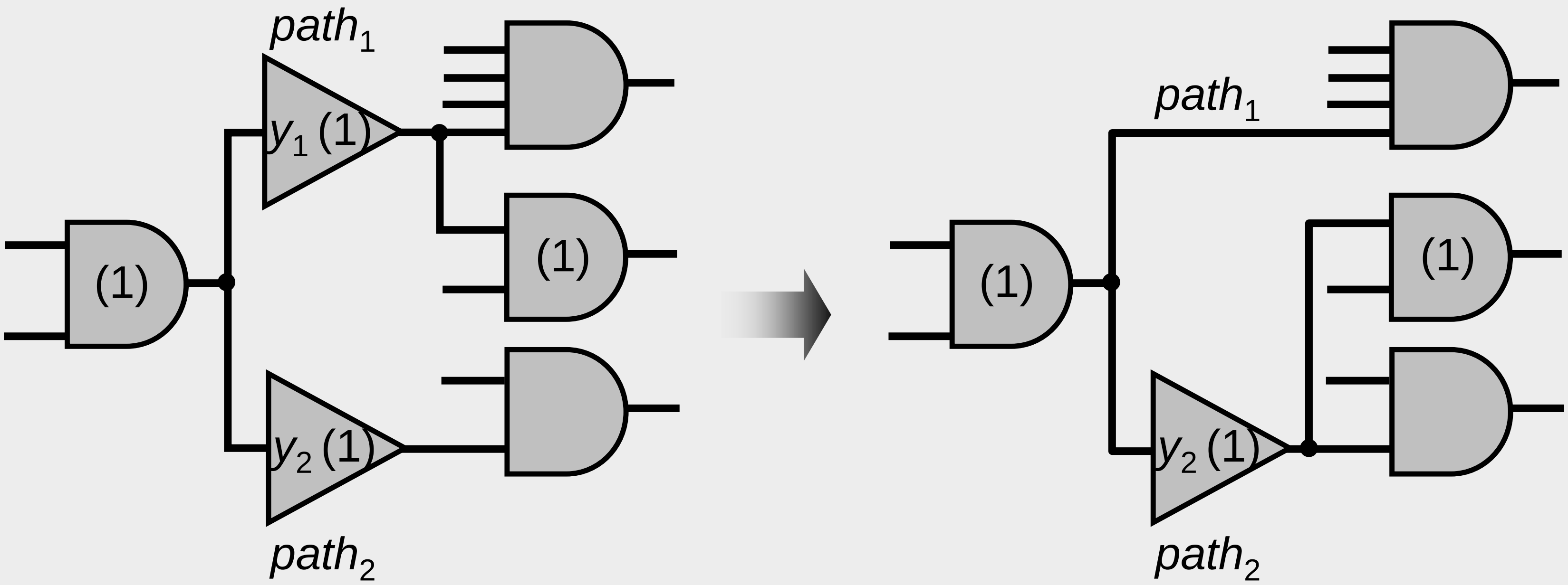


Redesigning the fanin tree can change AATs

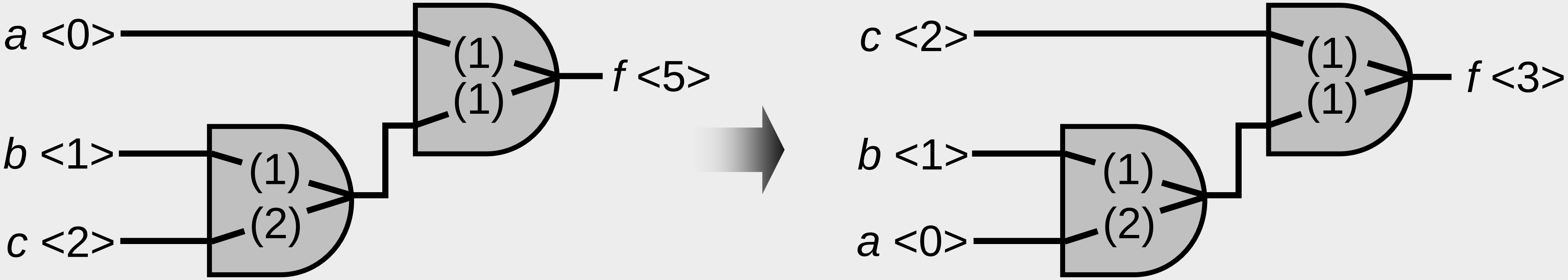


8.5.3 Netlist Restructuring

Redesigning fanout trees can change delays on specific paths

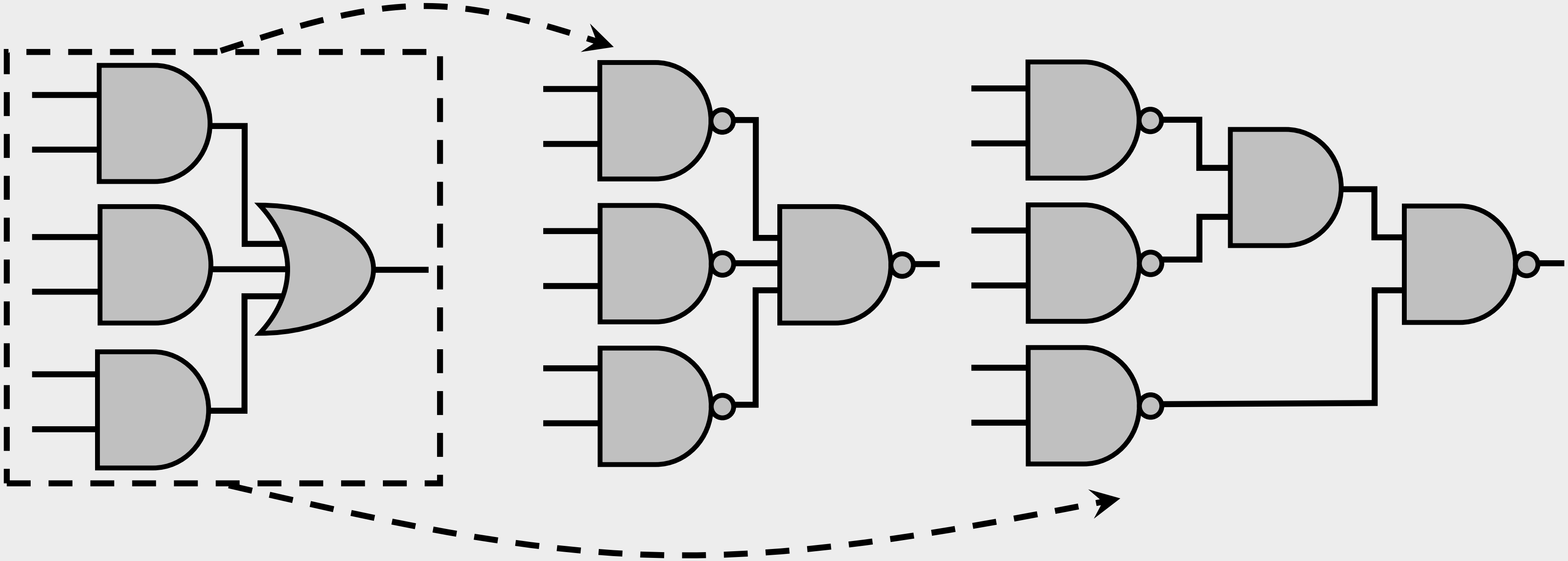


Swapping commutative pins can change the final delay



8.5.3 Netlist Restructuring

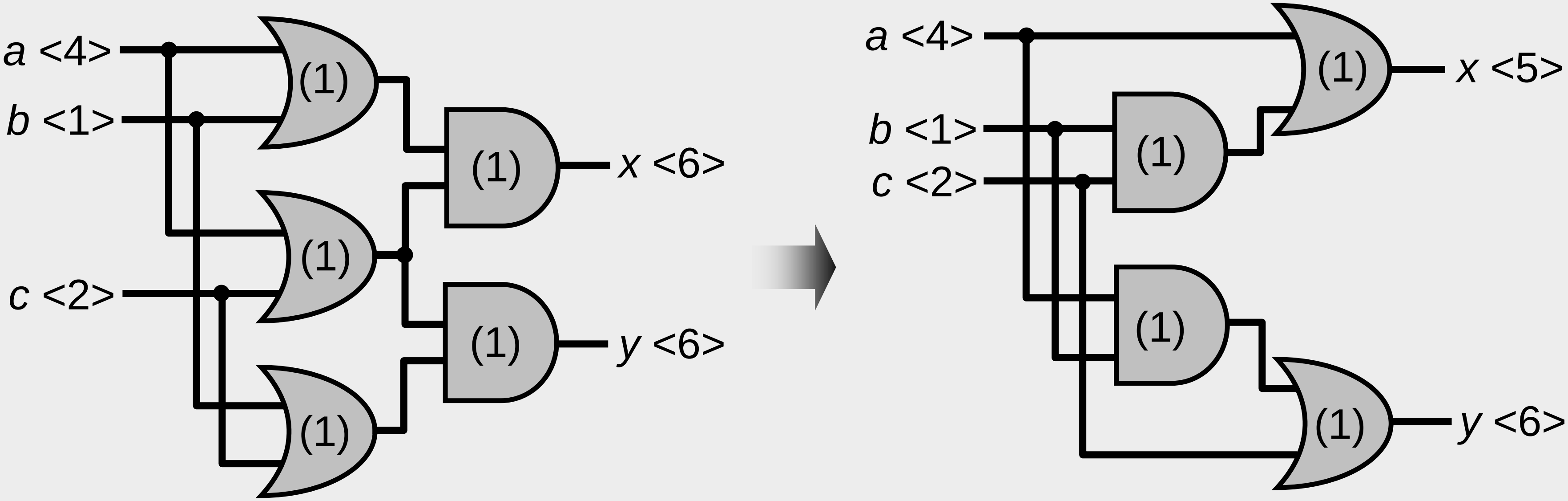
Gate decomposition can change the general structure of the circuit



8.5.3 Netlist Restructuring

Boolean restructuring uses laws or properties,
e.g., distributive law, to change circuit topology

$(a + b)(a + c) = a + bc$



$x(a,b,c) = (a + b)(a + c)$
$y(a,b,c) = (a + c)(b + c)$

$x(a,b,c) = a + bc$
$y(a,b,c) = ab + c$

8.6 Performance-Driven Design Flow

8.1 Introduction

8.2 Timing Analysis and Performance Constraints

- 8.2.1 Static Timing Analysis
- 8.2.2 Delay Budgeting with the Zero-Slack Algorithm

8.3 Timing-Driven Placement

- 8.3.1 Net-Based Techniques
- 8.3.2 Embedding STA into Linear Programs for Placement

8.4 Timing-Driven Routing

- 8.4.1 The Bounded-Radius, Bounded-Cost Algorithm
- 8.4.2 Prim-Dijkstra Tradeoff
- 8.4.3 Minimization of Source-to-Sink Delay

8.5 Physical Synthesis

- 8.5.1 Gate Sizing
- 8.5.2 Buffering
- 8.5.3 Netlist Restructuring

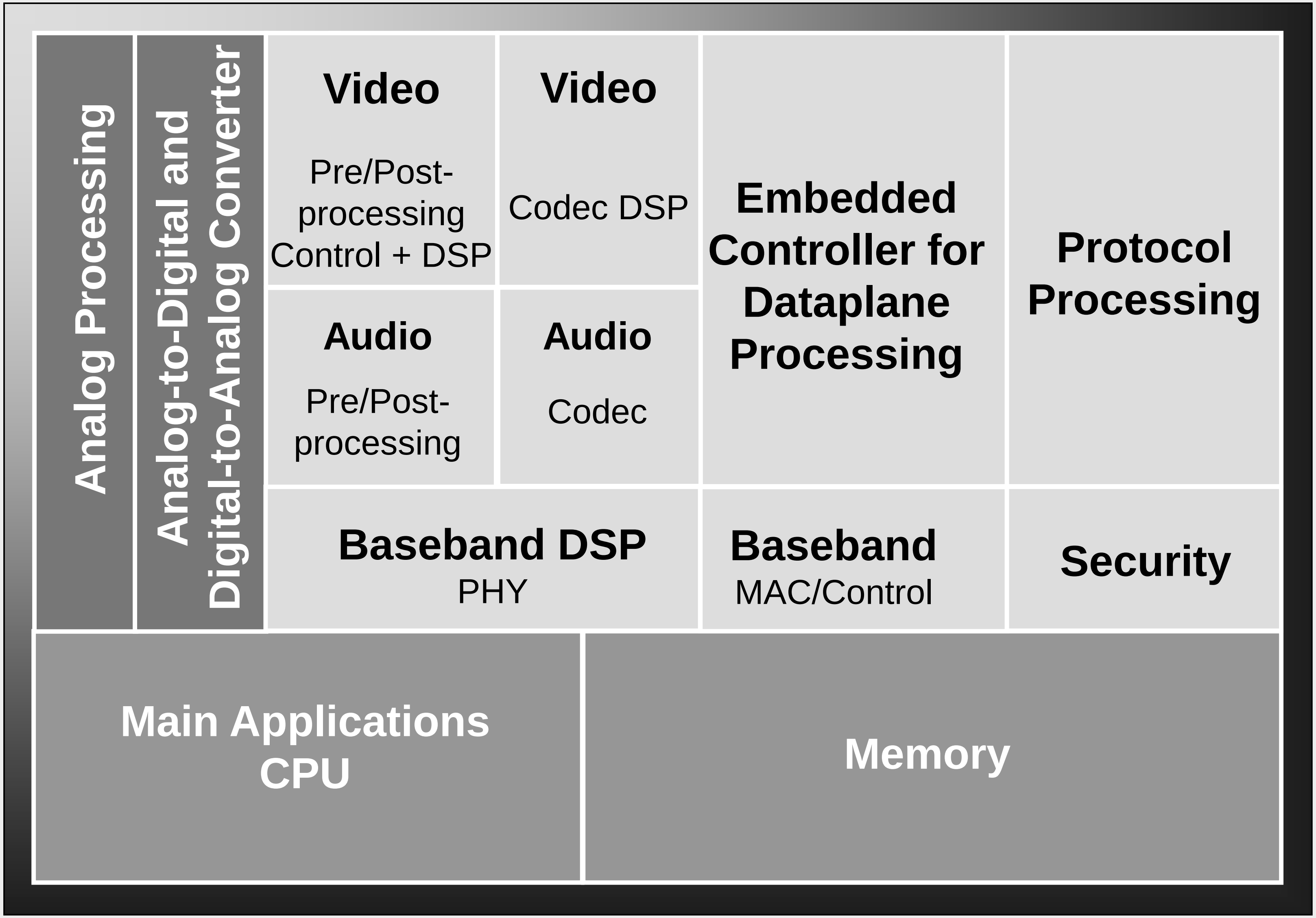
→ 8.6 Performance-Driven Design Flow

8.7 Conclusions

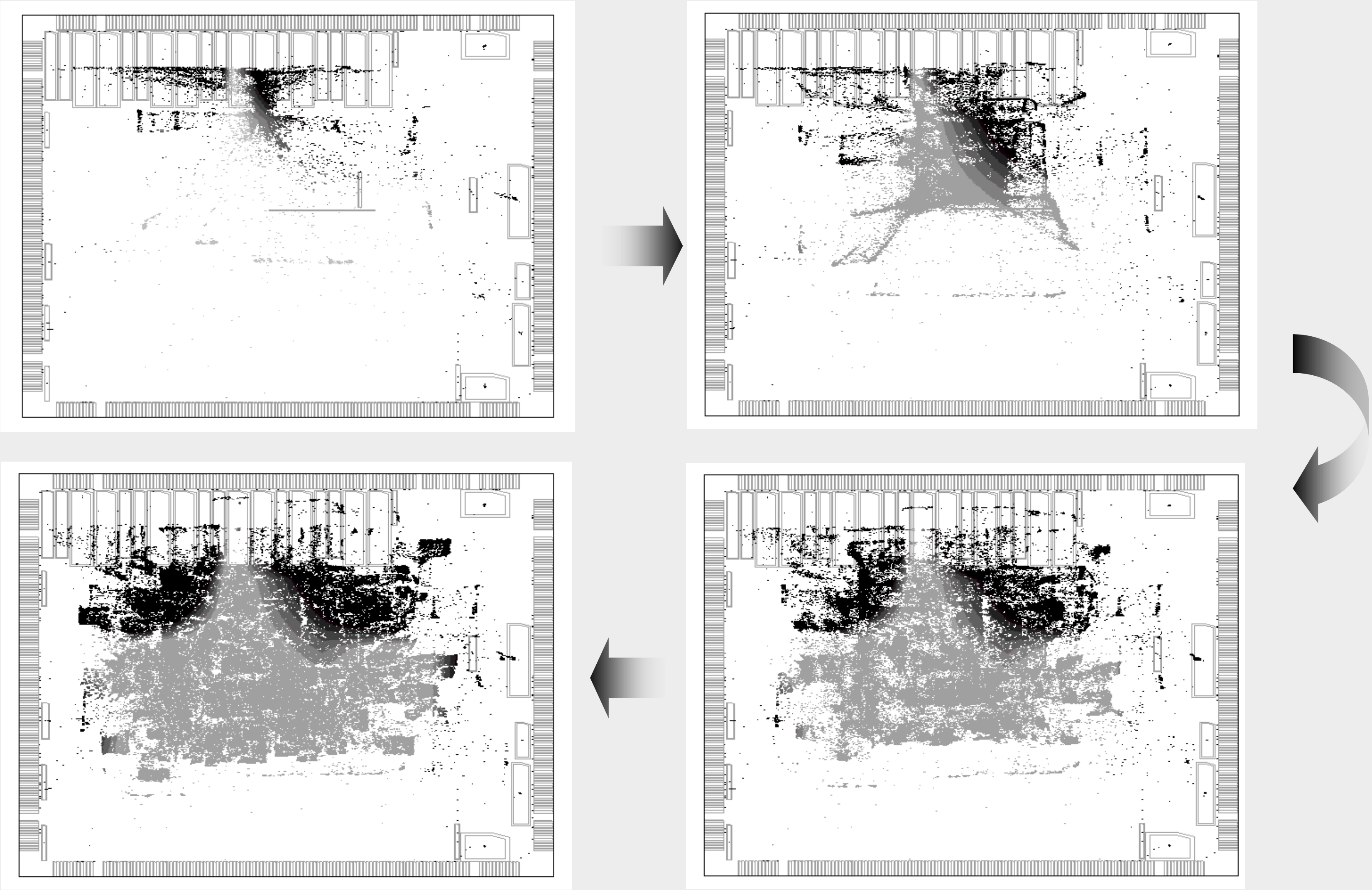
Baseline Physical Design Flow

1. Floorplanning, I/O placement, power planning
2. Logic synthesis and technology mapping
3. Global placement and sequential element legalization
4. Clock network synthesis
5. Global routing and layer assignment
6. Congestion-driven detailed placement and legalization
7. Detailed routing
8. Design for manufacturing
9. Physical verification
10. Mask optimization and generation

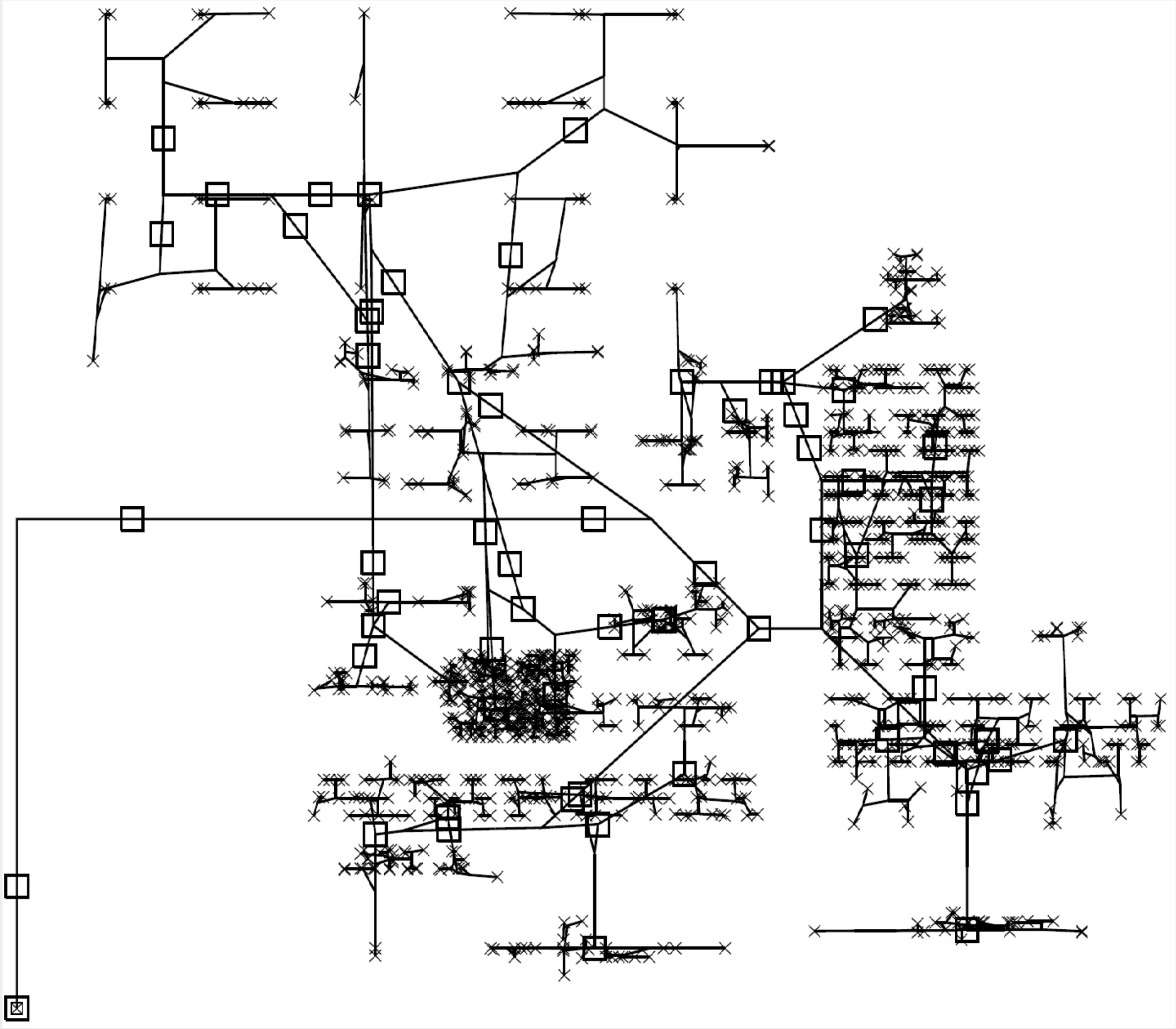
Floorplanning Example



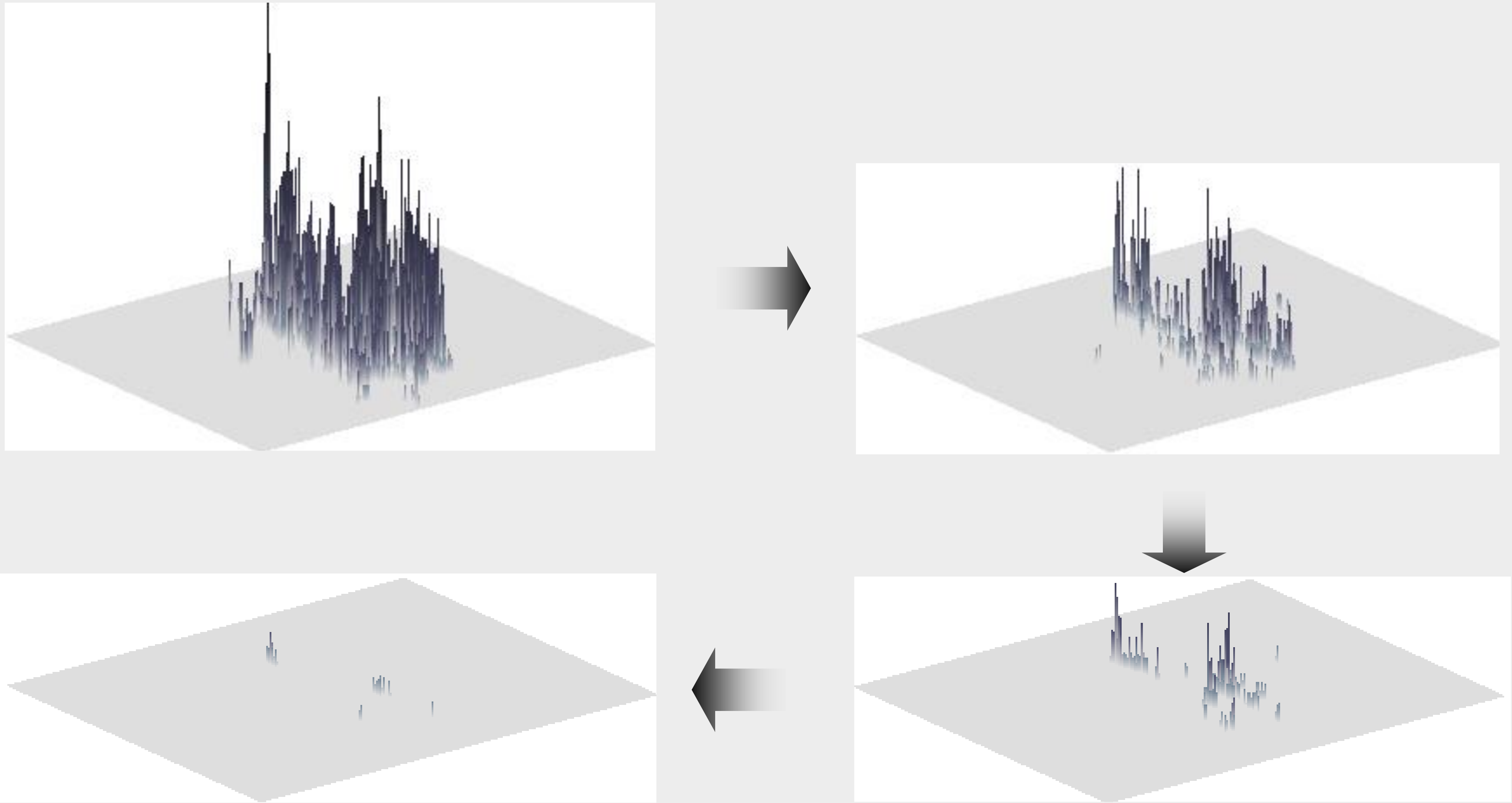
Global Placement Example



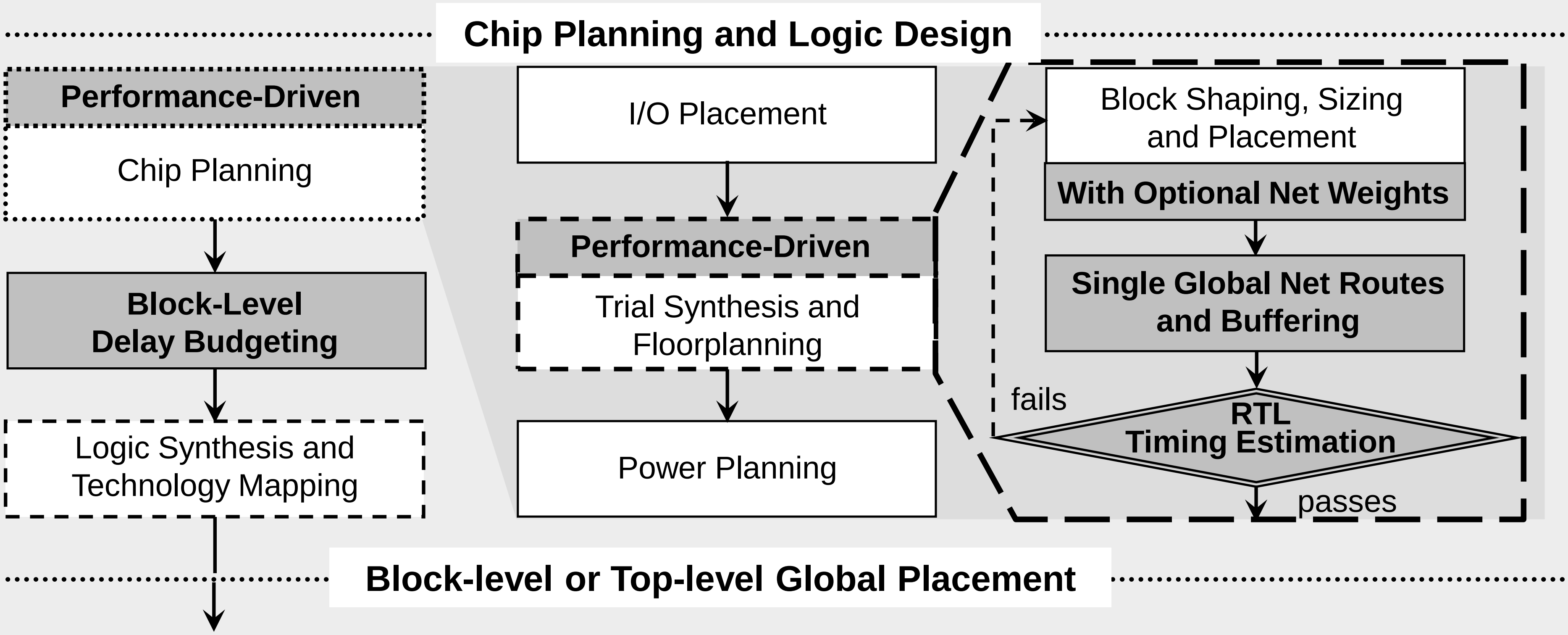
Clock Network Synthesis Example



Global Routing Congestion Example

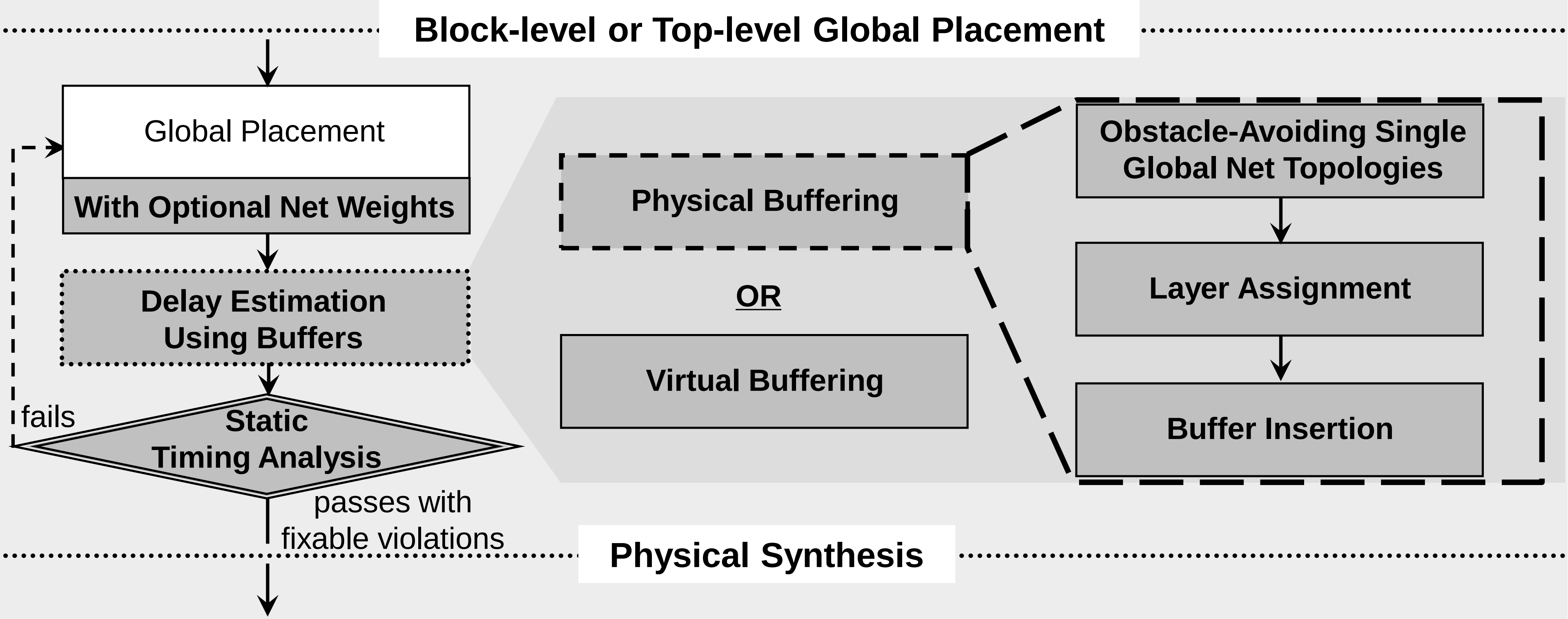


8.6 Performance-Driven Design Flow



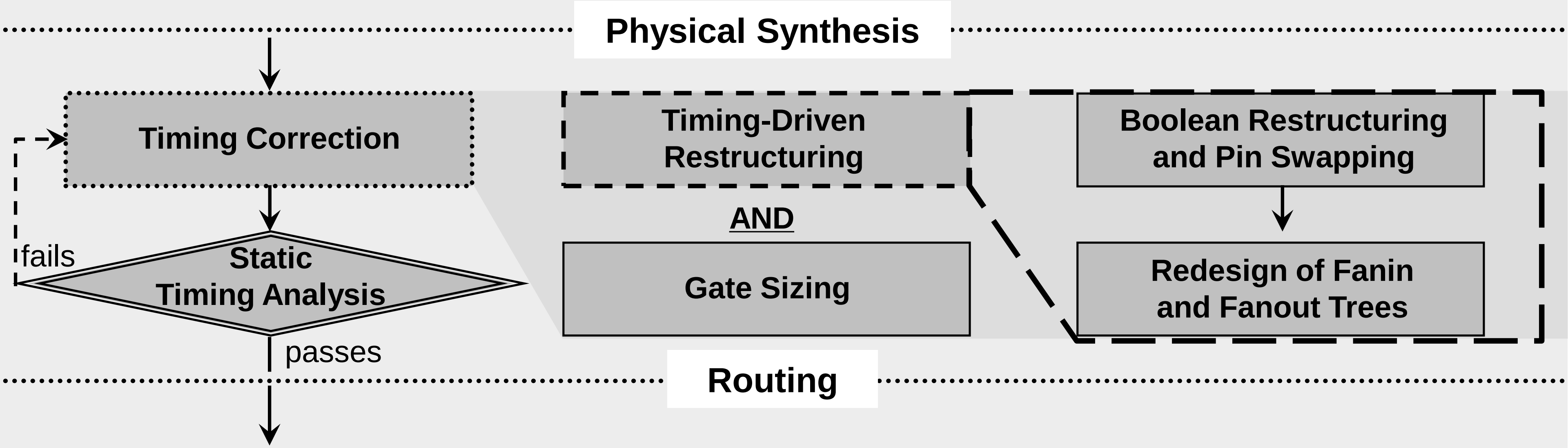
(see full flow chart in Figure 8.26)

8.6 Performance-Driven Design Flow


Block-level or Top-level Global Placement
Global Placement
With Optional Net Weights
Delay Estimation Using Buffers
Static Timing Analysis
fails
passes with fixable violations
Physical Buffering
OR
Virtual Buffering
Obstacle-Avoiding Single Global Net Topologies
Layer Assignment
Buffer Insertion
Physical Synthesis

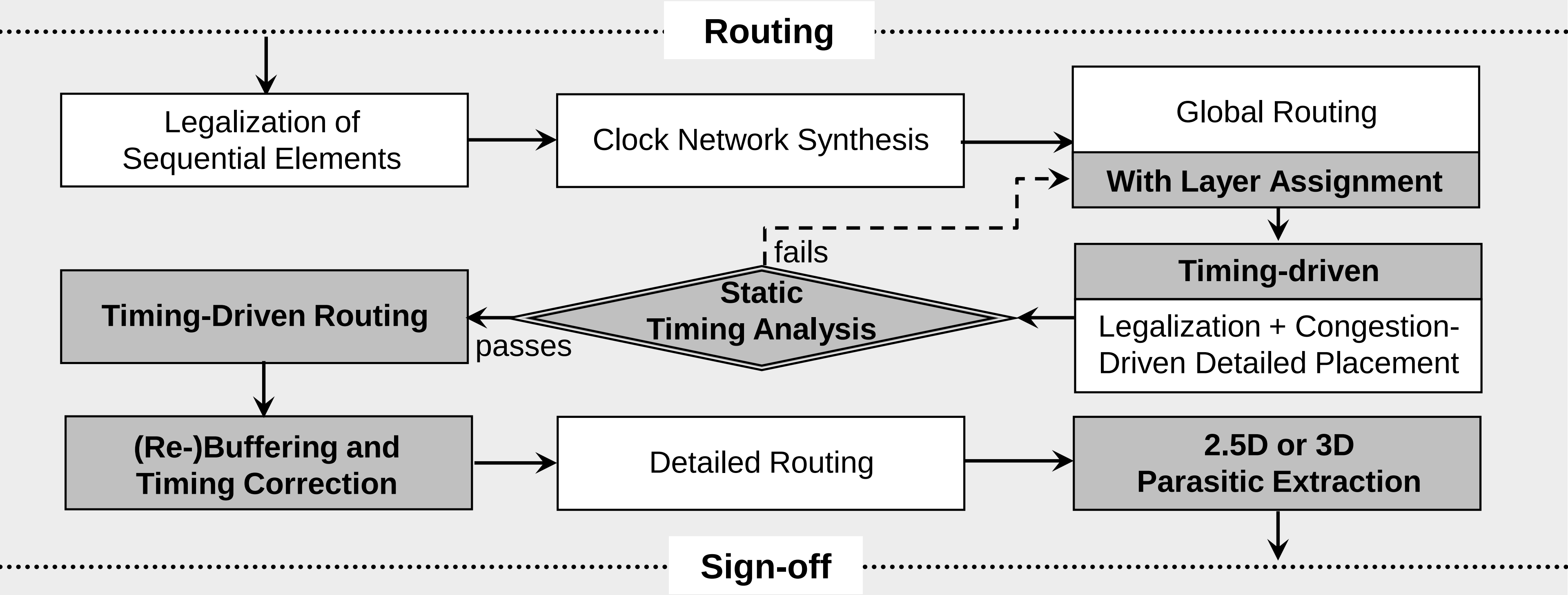
(see full flow chart in Figure 8.26)

Performance-Driven Design Flow


Physical Synthesis
Timing Correction
Timing-Driven Restructuring
Boolean Restructuring and Pin Swapping
AND
Static Timing Analysis
fails
Gate Sizing
Redesign of Fanin and Fanout Trees
passes
Routing

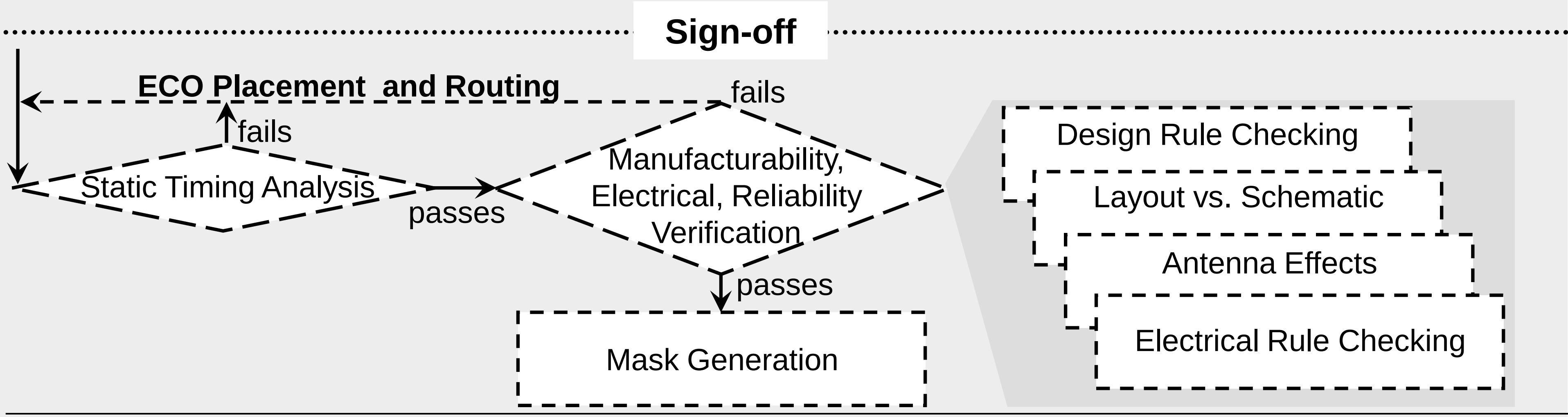
(see full flow chart in Figure 8.26)

8.6 Performance-Driven Design Flow



(see full flow chart in Figure 8.26)

8.6 Performance-Driven Design Flow



(see full flow chart in Figure 8.26)

Summary of Chapter 8 – Timing Constraints and Timing Analysis

- Circuit delay is measured on signal paths
  - From primary inputs to sequential elements; from sequentials to primary outputs
  - From sequentials to sequentials
- Components of path delay
  - Gate delays: over-estimated by worst-case transition per gate (to ensure fast Static Timing Analysis)
  - Wire delays: depend on wire length and (for nets with >2 pins) topology
- Timing constraints
  - Actual arrival times (AATs) at primary inputs and output pins of sequentials
  - Required arrival times (RATs) at primary outputs and input pins of sequentials
- Static timing analysis
  - Two linear-time traversals compute AATs and RATs for each gate (and net)
  - At each timing point: slack = RAT-AAT
  - Negative slack = timing violation; critical nets/gates are those with negative slack
- Time budgeting: divides prescribed circuit delay into net delay bounds

Summary of Chapter 8 – Timing-Driven Placement

- Gate/cell locations affect wire lengths, which affect net delays
- Timing-driven placement optimizes gate/cell locations to improve timing
  - Interacts with timing analysis to identify critical nets, then biases placement opt.
  - Must keep total wirelength low too, otherwise routing will fail
  - Timing optimization may increase routing congestion
- Placement by net weighting
  - The least invasive technique for timing-driven placement
  - Performs tentative placement, then changes net weights based on timing analysis
- Placement by net budgeting
  - Allocates delay bounds for each net; translates delay bounds into length bounds
  - Performs placement subject to length constraints for individual nets
- Placement based on linear programming
  - Placement is cast as a system of equations and inequalities
  - Timing analysis and optimization are incorporated using additional inequalities

Summary of Chapter 8 – Timing-Driven Routing

- Timing-driven routing has several aspects
  - Individual nets: trading longer wires for shorter source-to-sink paths
  - Coupling capacitance and signal integrity: parallel wires act as capacitors and can slow-down/speed-up signal transitions
  - Full-netlist optimization: prioritize the nets that should be optimized first
- Individual net optimization
  - One extreme: route each source-to-sink path independently (high wirelength)
  - Another extreme: use a Minimum Spanning Tree (low wirenegth, high delay)
  - Tunable tradeoff: a hybrid of Prim and Dijkstra algorithms
- Coupling capacitance and signal integrity
  - Parallel wires are only worth attention when they transition at the same time
  - Identify critical nets, push neighboring wires further away to limit crosstalk
- Full-netlist optimization
  - Run trial routing, then run timing analysis to identify critical nets
  - Then adjust accordingly, repeat until convergence

Summary of Chapter 8 – Physical Synthesis

- Traditionally, place-and-route have been performed after the netlist is known
- However, fixing gate sizes and net topologies early does not account for placement-aware timing analysis
  - Gate locations and net routes are not available
- Physical synthesis uses information from trial placement to modify the netlist
- Net buffering: splits a net into smaller (approx. equal length) segments
  - A long net has high capacitance, the driver may be too weak
- Gate/buffer sizing: increases driver strength & physical size of a gate
  - Large gates have higher input pin capacitance, but smaller driver resistance
  - Larger gates can drive larger fanouts, longer nets; faster transitions
  - Large gates require more space, larger upstream drivers
- Gate cloning: splits large fanouts
  - Cloned gates can be placed separately, unlike with a single larger gate