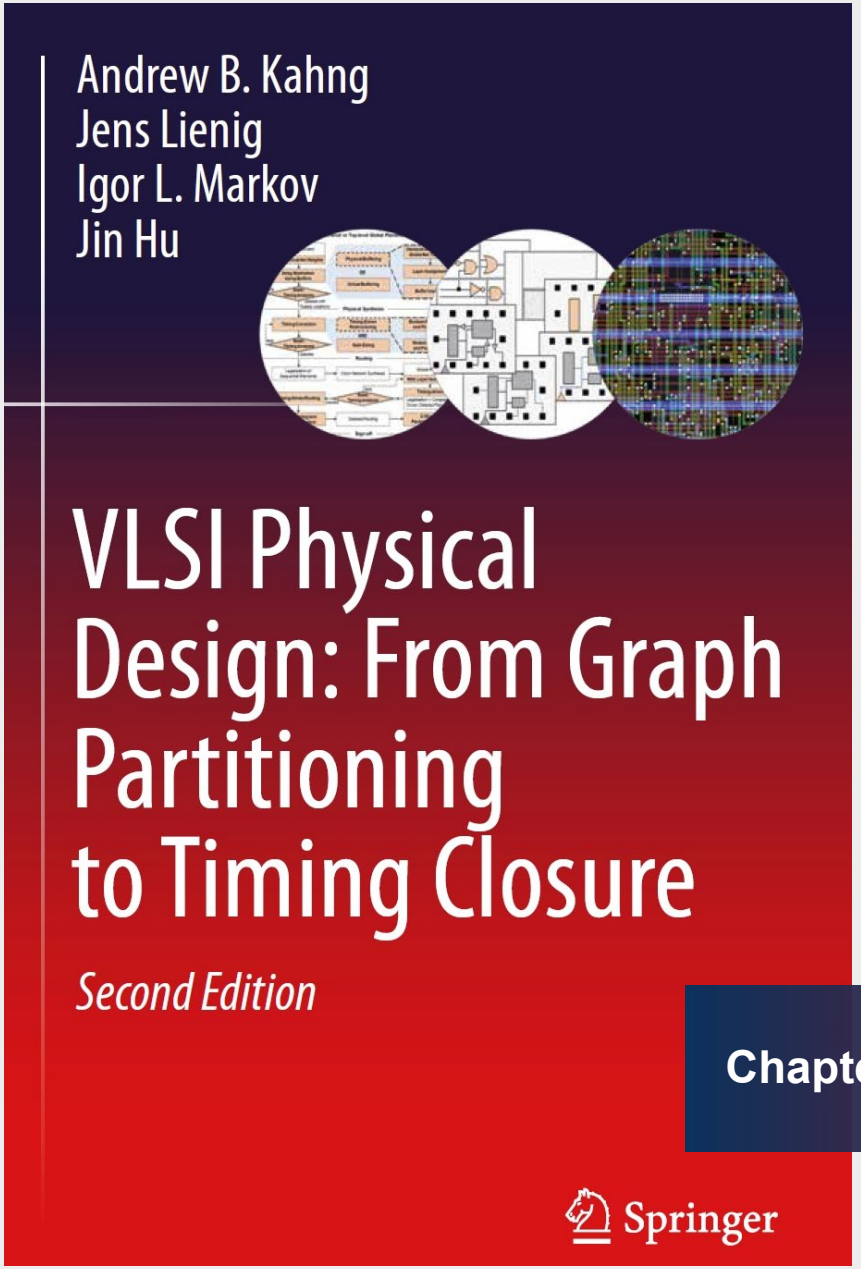
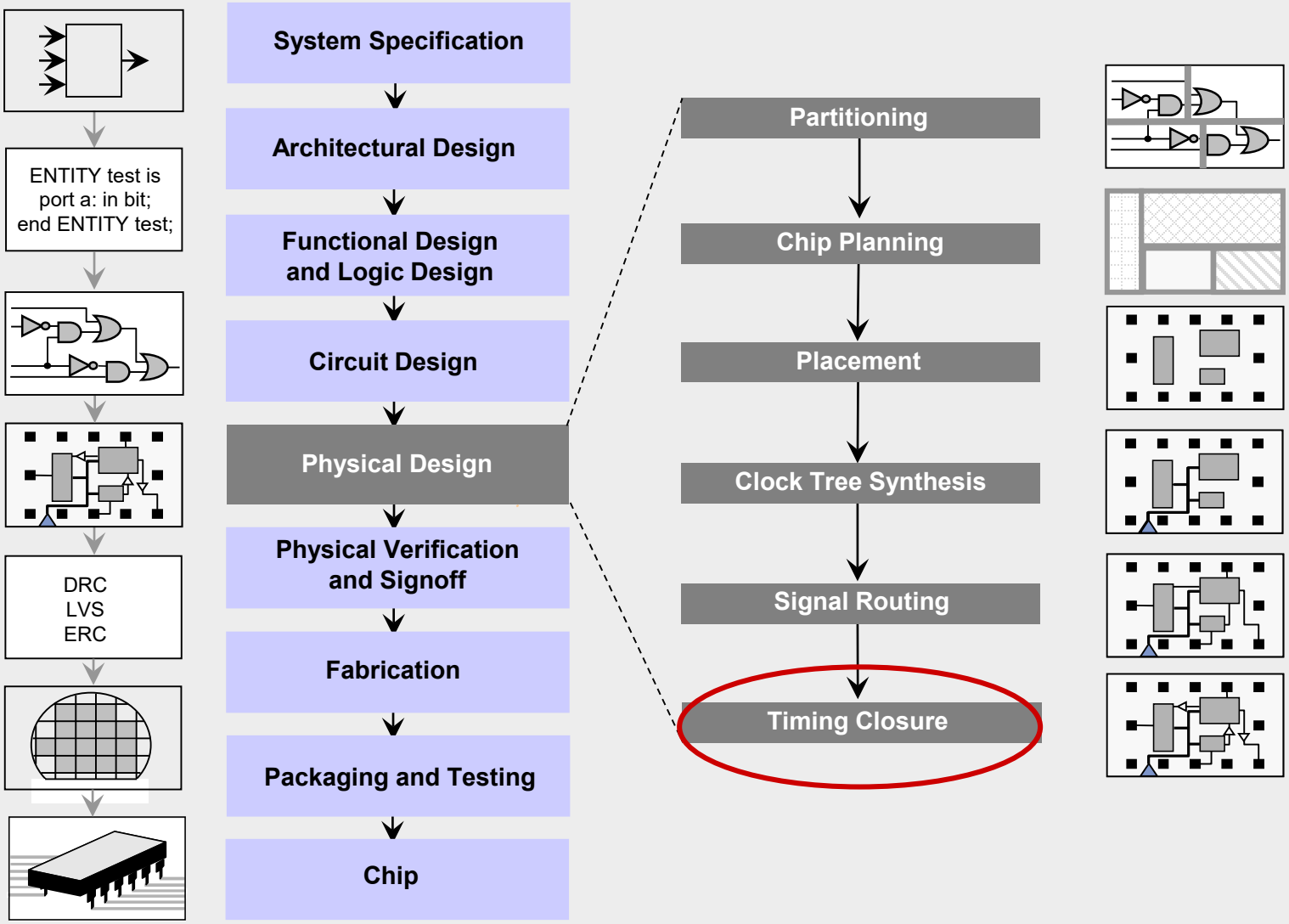


Chapter 8 – Timing Closure

Chapter 8 – Timing Closure

8.1 Introduction



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

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)

- 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

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

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

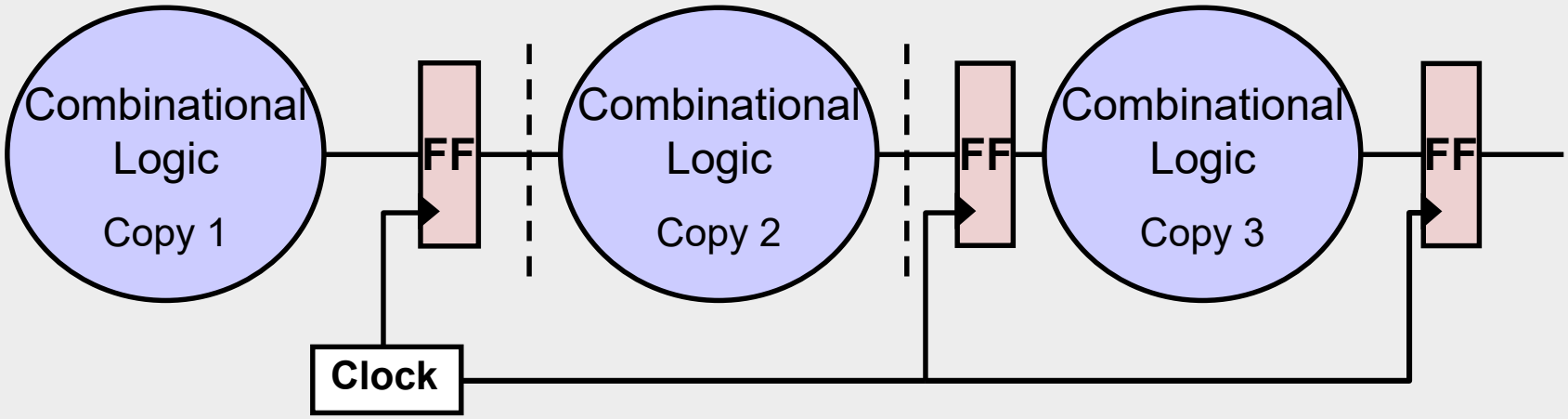
8.1 Introduction

- 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

Sequential circuit, “unrolled” in time



Storage elements

Combinational logic

- 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

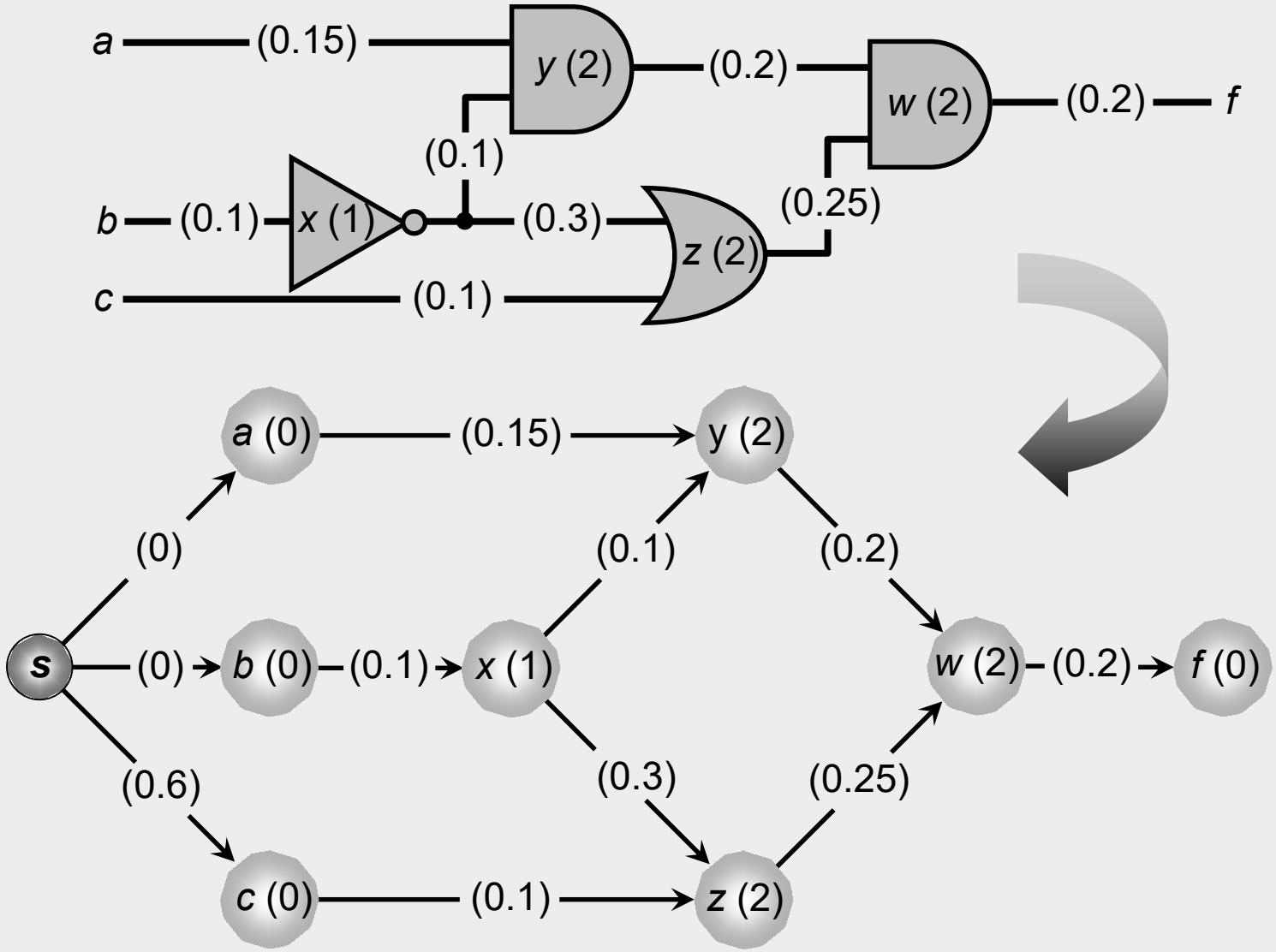
- Given combinational circuit, represent as directed acyclic graph (DAG)
  - Every edge (node) has weight = wire (gate) delay

- 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

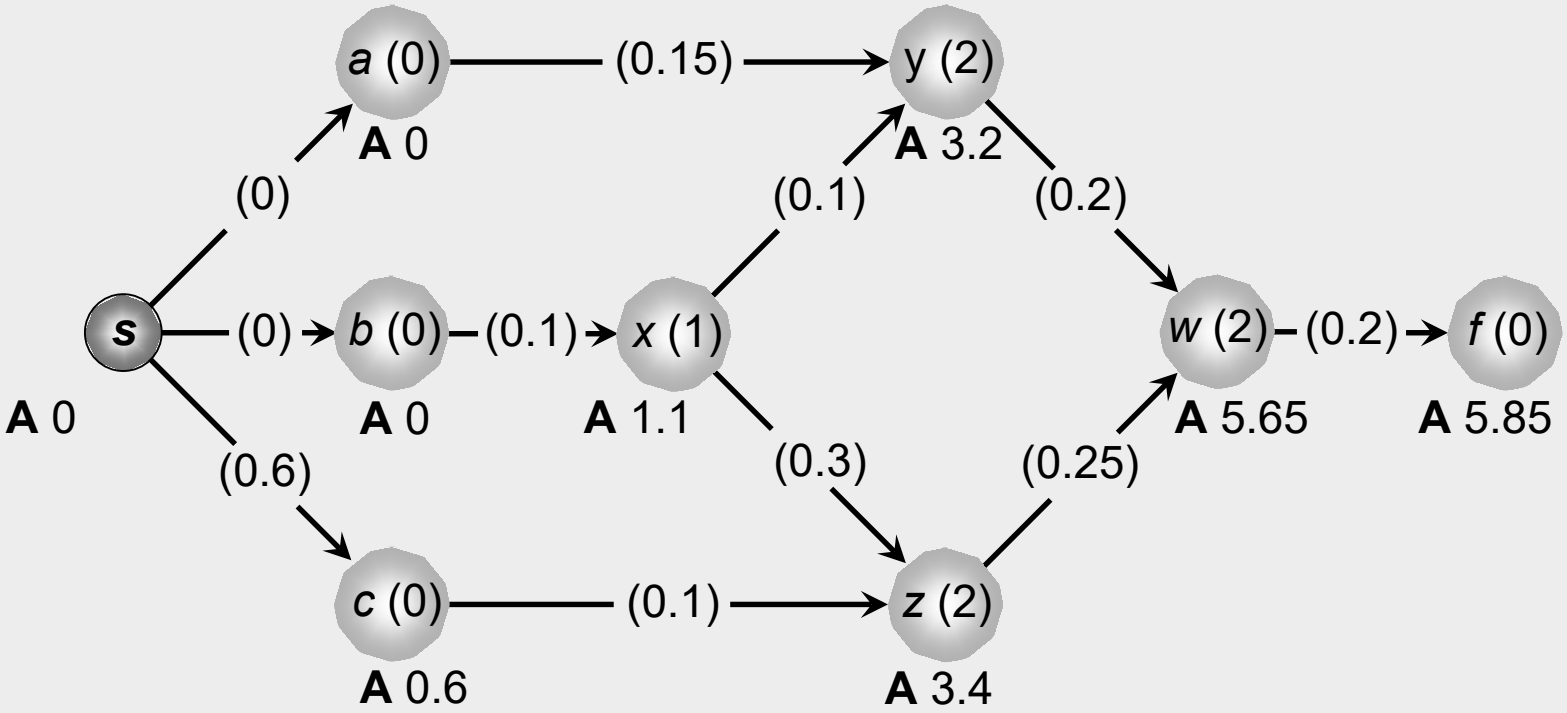


Compute AATs at each node:

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

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

(AATs of inputs are given)

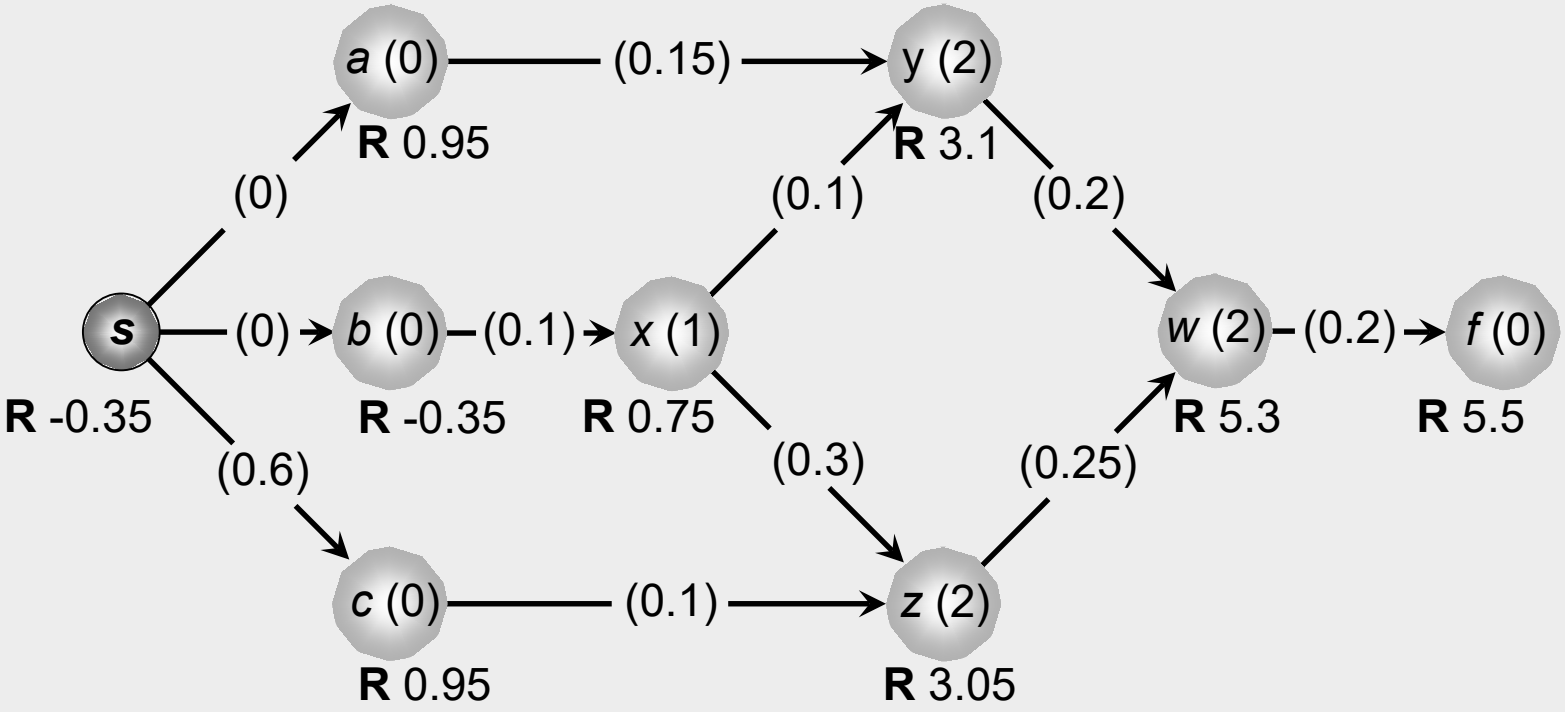


Compute RATs at each node:

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

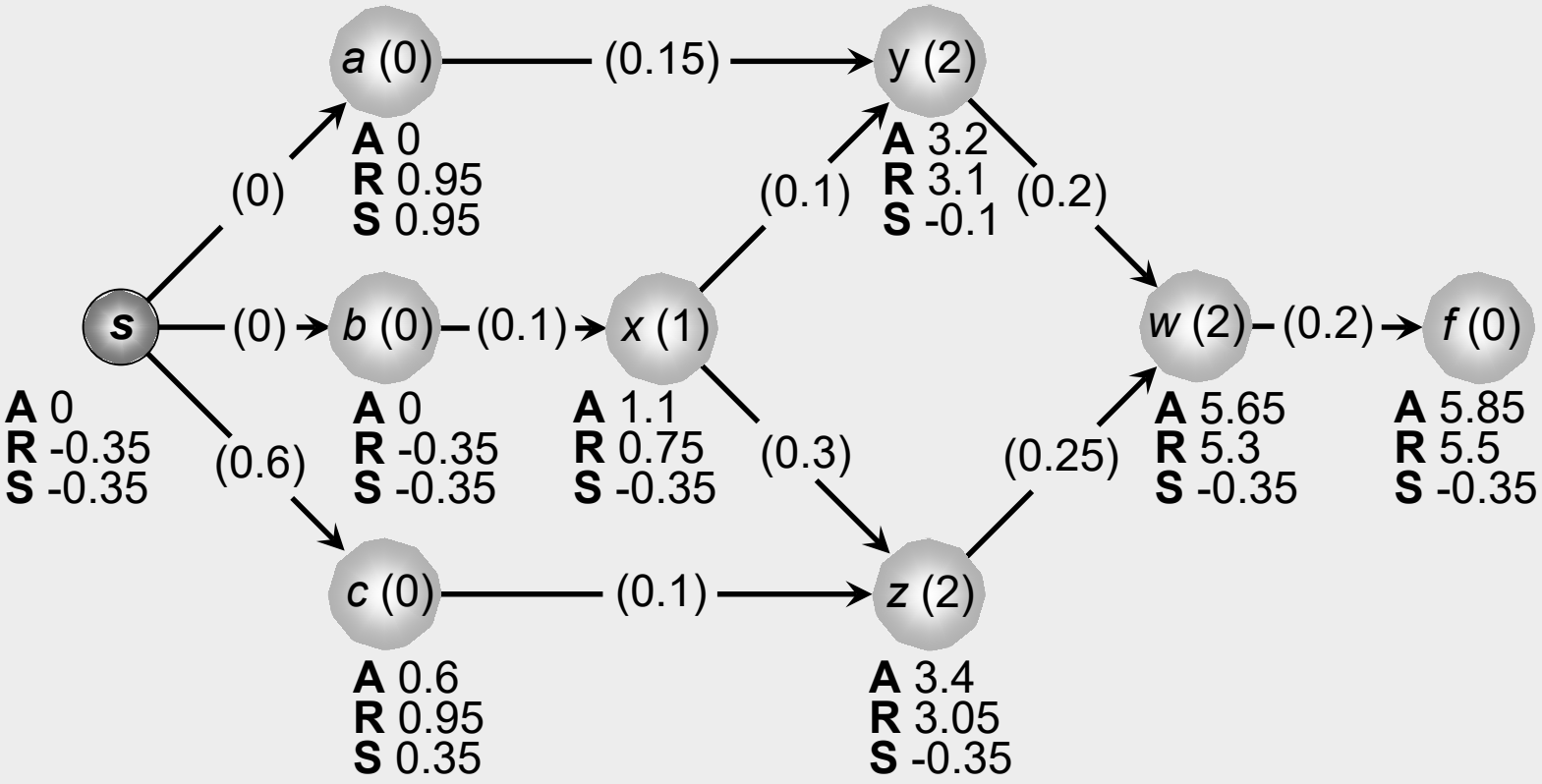
where $FO(v)$ are the fanout nodes, and $t(u,v)$ is the delay between u and v

(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 $v \in V$

1. **do**
2. $(AAT,RAT,slack) = \text{STA}(G)$
3. **foreach** $(v_i \in V)$
4. $TB[v_i] = \text{DELAY}(v_i) + \text{DELAY}(e_i)$
5. $slack_{min} = \infty$
6. **foreach** $(v \in V)$
7. **if** $((slack[v] < slack_{min})$ **and** $(slack[v] > 0))$
8. $slack_{min} = slack[v]$
9. $v_{min} = v$
10. **if** $(slack_{min} \neq \infty)$
11. $path = v_{min}$
12. ADD_TO_FRONT($path$,BACKWARD_PATH(v_{min},G))
13. ADD_TO_BACK($path$,FORWARD_PATH(v_{min},G))
14. $s = slack_{min} / |path|$
15. **for** $(i = 1$ **to** $|path|)$
16. $node = path[i]$ // evenly distribute
17. $TB[node] = TB[node] + s$ // slack along *path*
18. **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*

```
1. path = v_min
2. do
3.    flag = false
4.    node = LAST_ELEMENT(path)
5.    foreach (fanout node fo of node)
6.       if ((RAT[fo] == RAT[node] + TB[fo]) and (AAT[fo] == AAT[node] + TB[fo]))
7.          ADD_TO_BACK(path,fo)
8.          flag = true
9.          break
10. while (flag == true)
11. REMOVE_FIRST_ELEMENT(path)                          // remove v_min
```

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

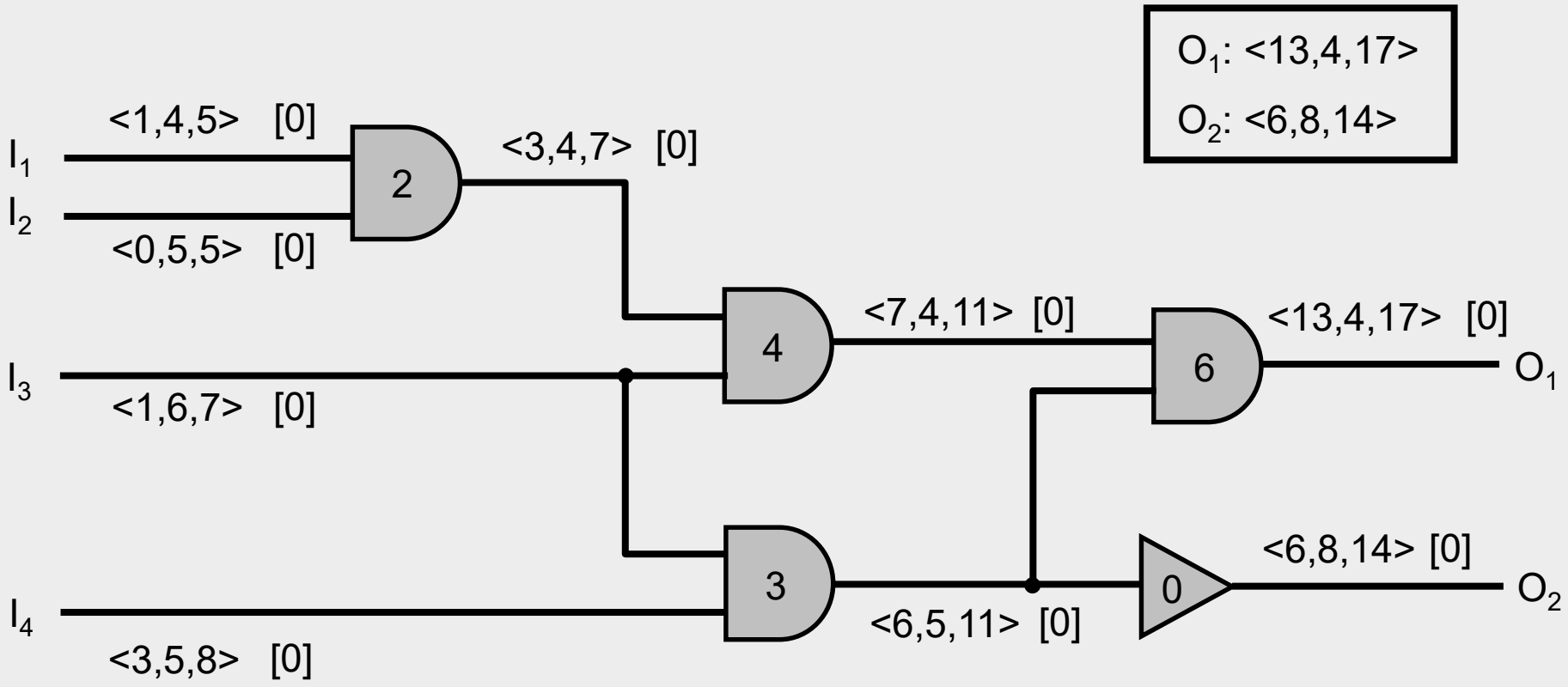
Input: node v_{min} with minimum slack $slack_{min}$, timing graph G

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

```
1.  path = v_min
2.  do
3.      flag = false
4.      node = FIRST_ELEMENT(path)
5.      foreach (fanin node fi of node)
6.          if ((RAT[fi] == RAT[node] – TB[fi]) and (AAT[fi] == AAT[node] – TB[fi]))
7.              ADD_TO_FRONT(path,fi)
8.              flag = true
9.              break
10. while (flag == true)
11. REMOVE_LAST_ELEMENT(path)                                   // remove v_min
```

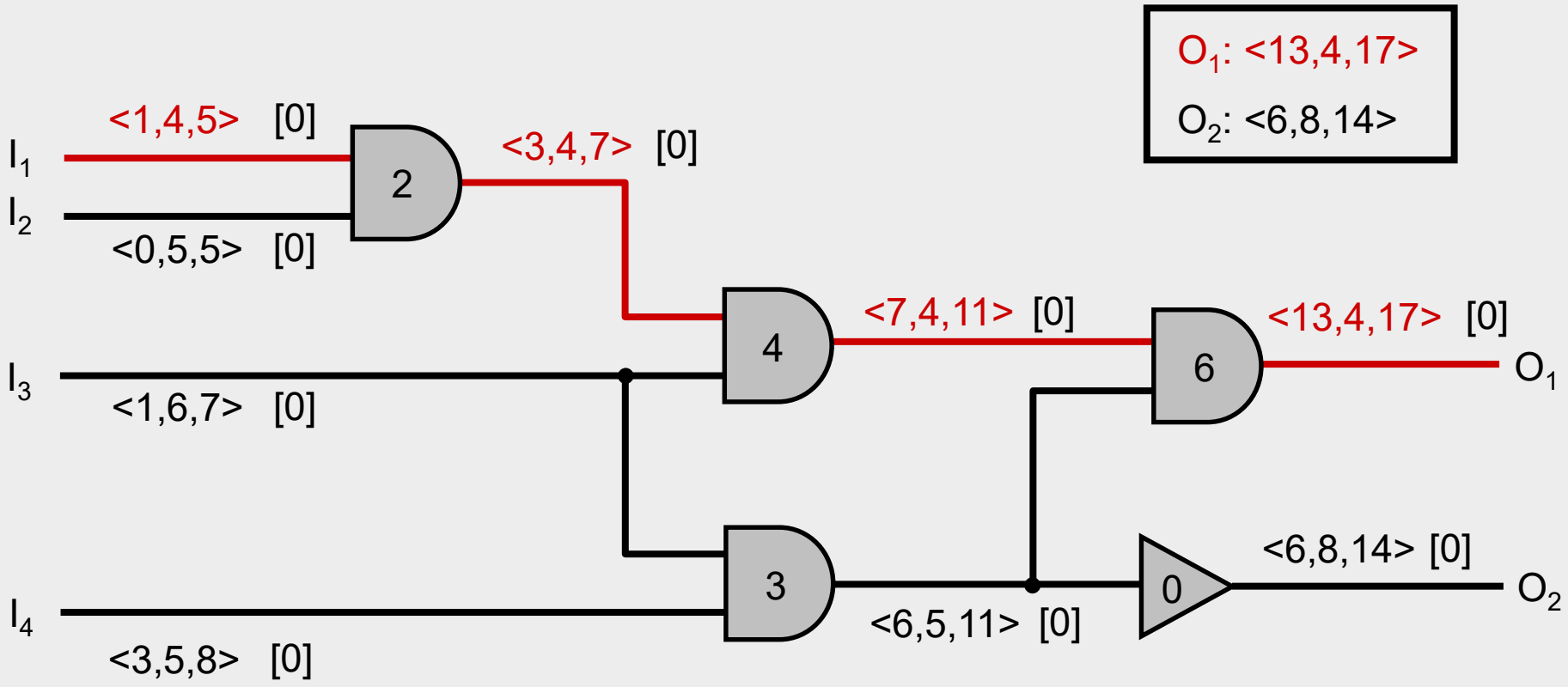
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

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



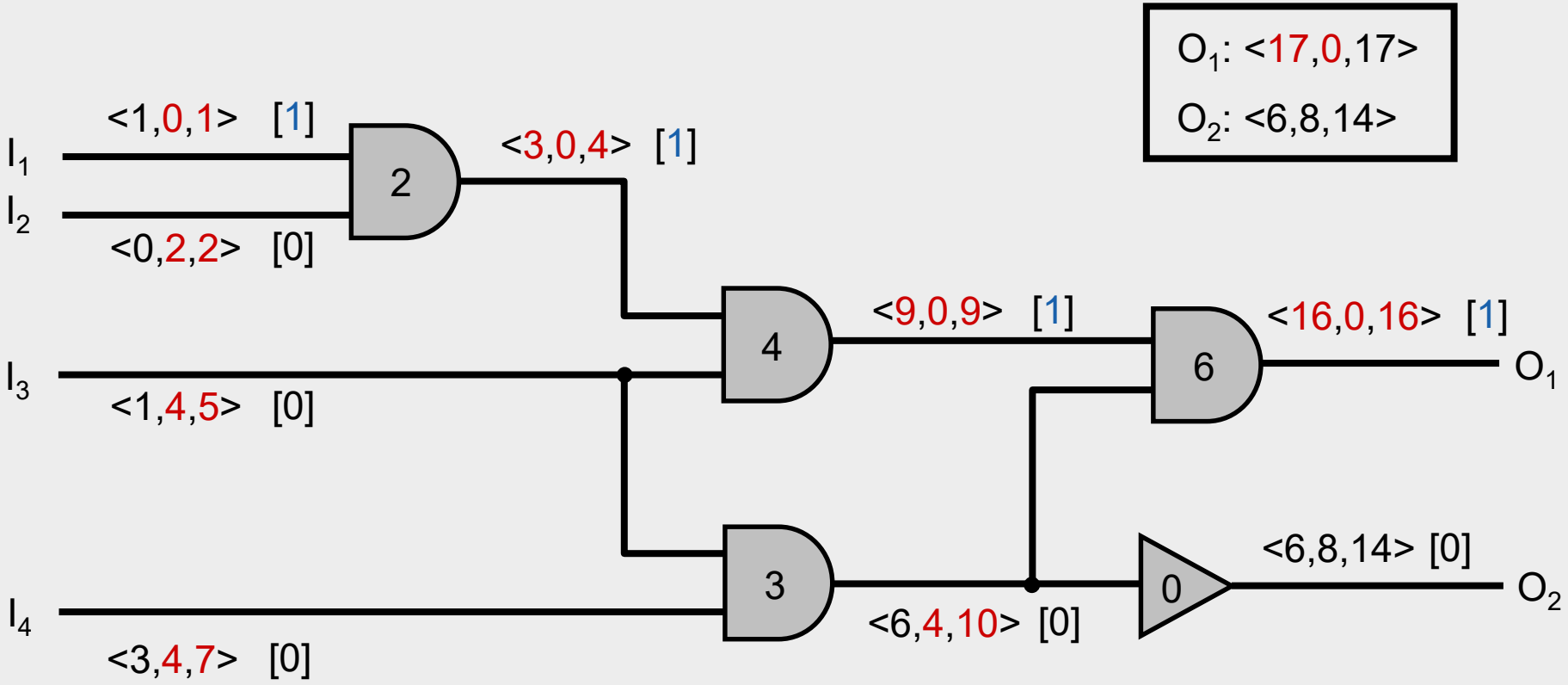
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

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



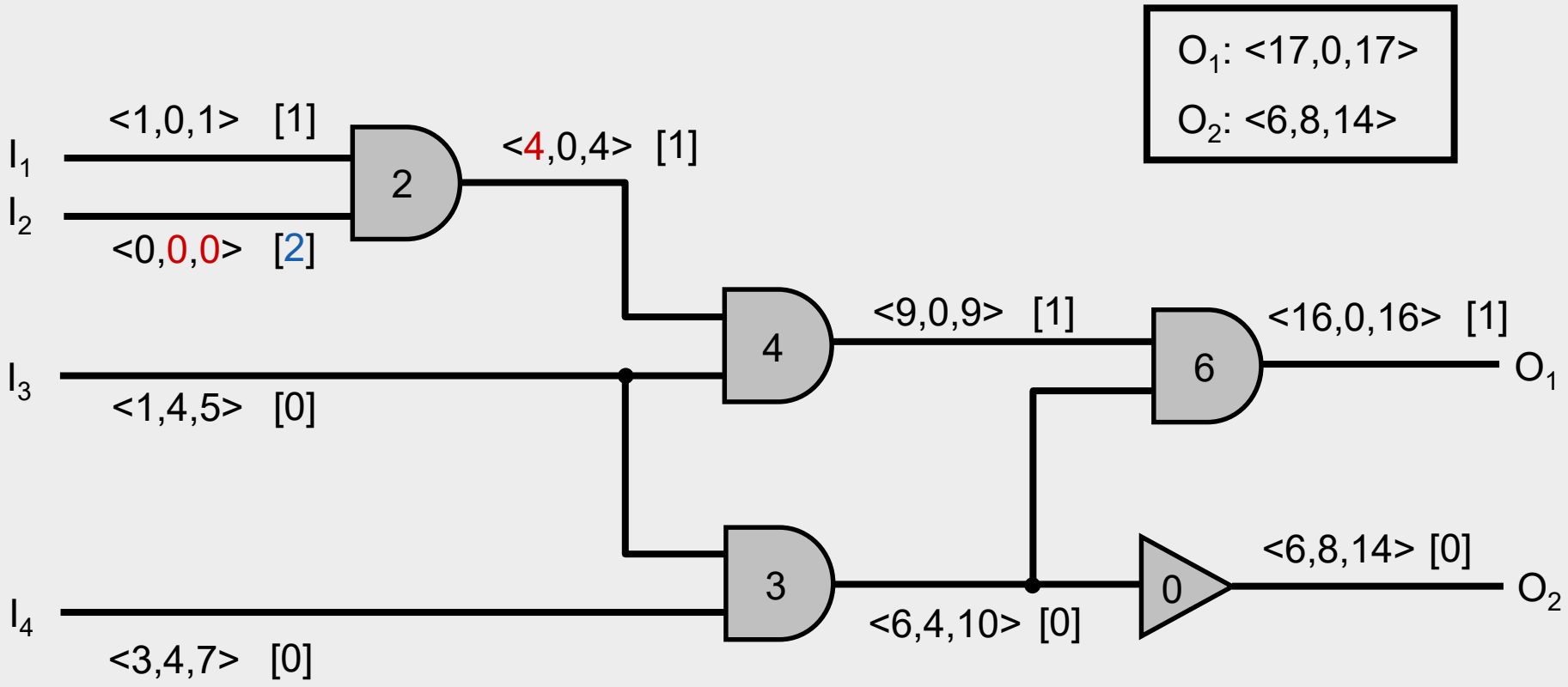
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- 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



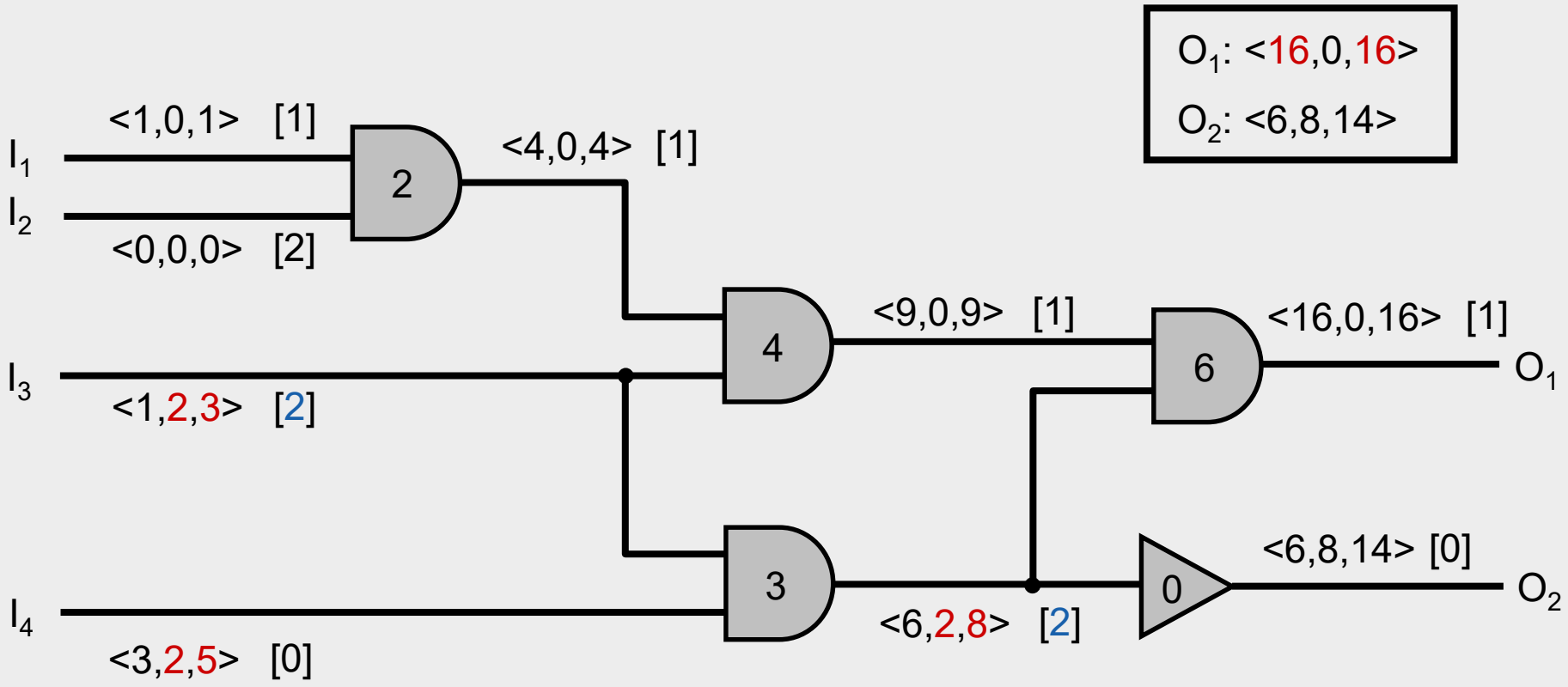
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- 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



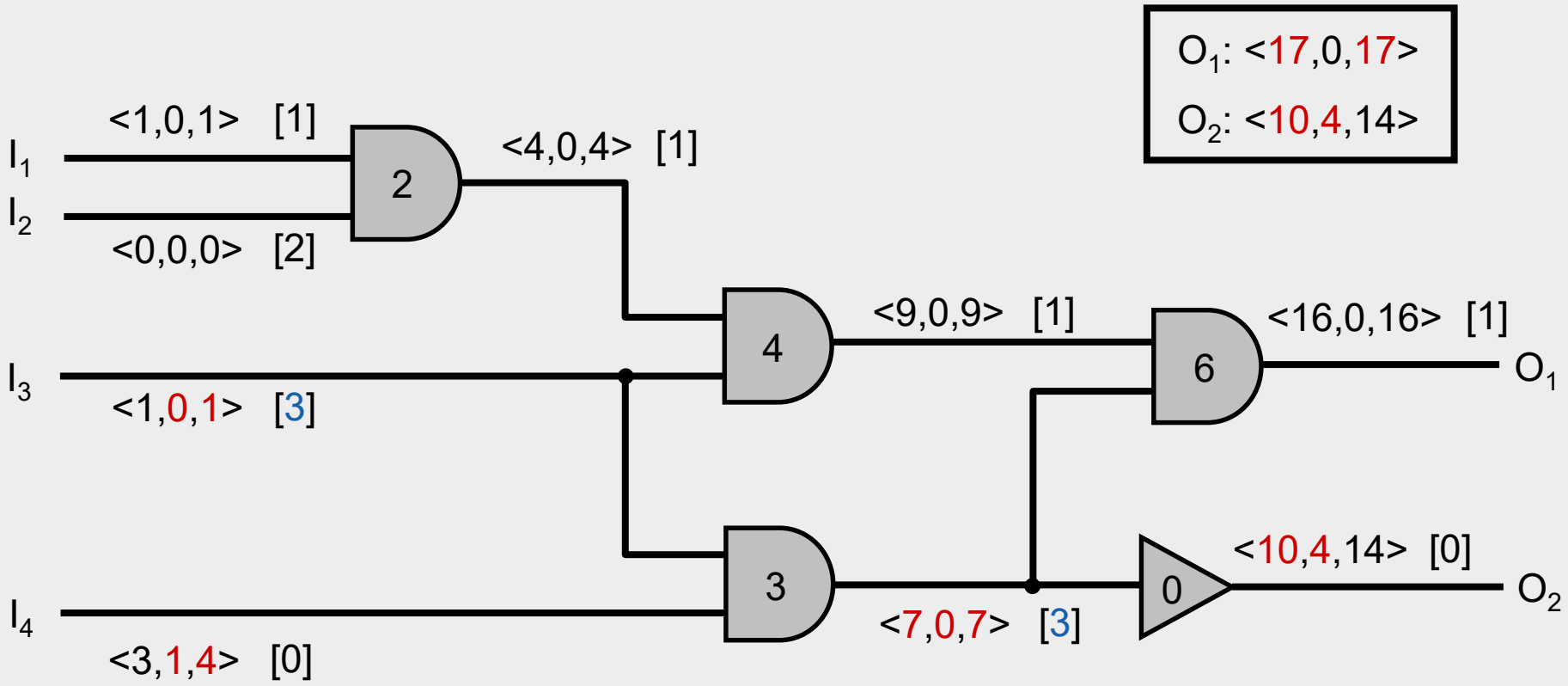
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- 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



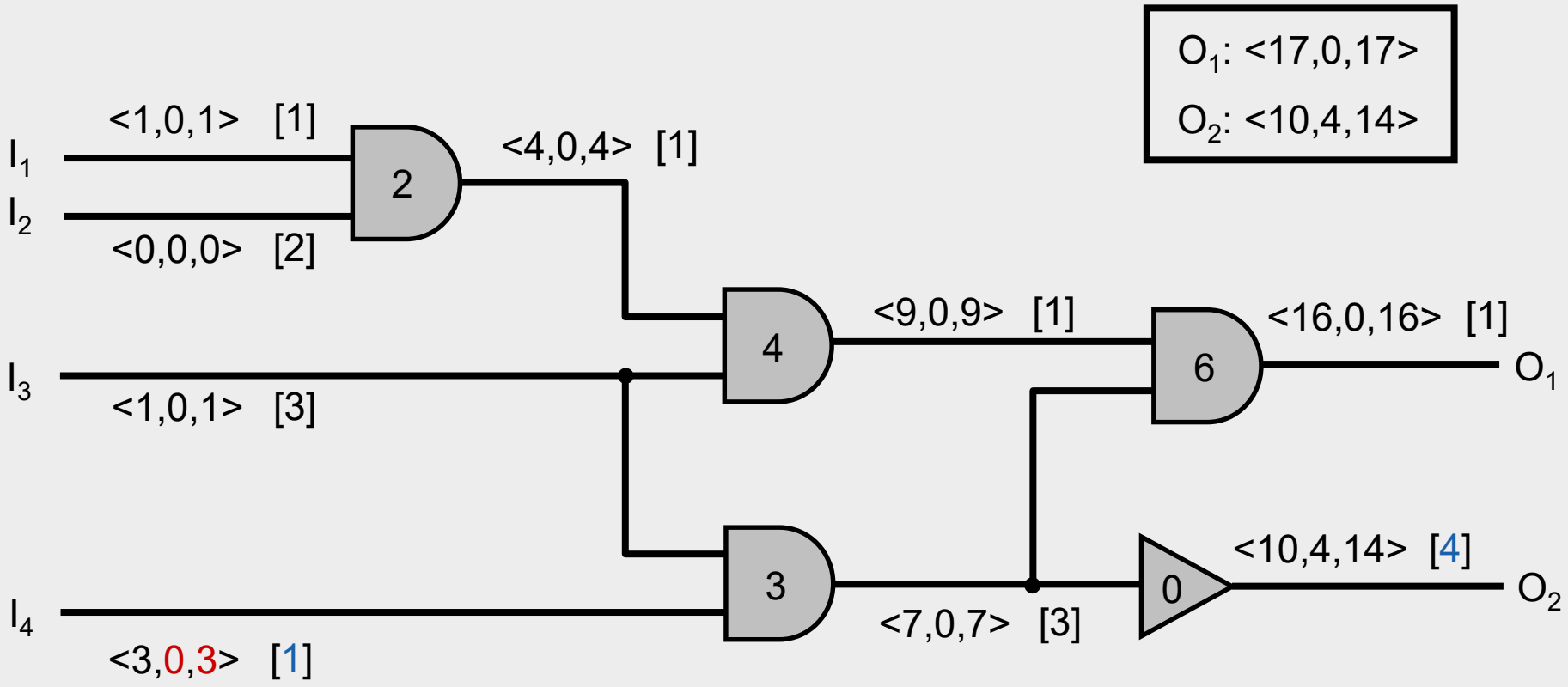
8.2.2 Delay Budgeting with the Zero-Slack Algorithm Example

- 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

- 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)

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

- Or total negative slack (TNS)

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

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

- Net weights are added to each net – placer optimizes weighted wirelength
- Static net weights: computed before placement (never changes)
  - Discrete net weights: $w = \begin{cases} \omega_1 & \text{if } slack > 0 \\ \omega_2 & \text{if } slack \leq 0 \end{cases}$ 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

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

- 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

- 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

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

$$left(e) \le x_v + \delta_x(v,e)$$
$$right(e) \ge x_v + \delta_x(v,e)$$
$$bottom(e) \le y_v + \delta_y(v,e)$$
$$top(e) \ge y_v + \delta_y(v,e)$$

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

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

- 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

- 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 $slack(\tau_p) \leq 0$

- Optimize for total negative slack:

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

- Optimize for worst negative slack:

$$\max : WNS$$

- Optimize a linear combination of multiple parameters:

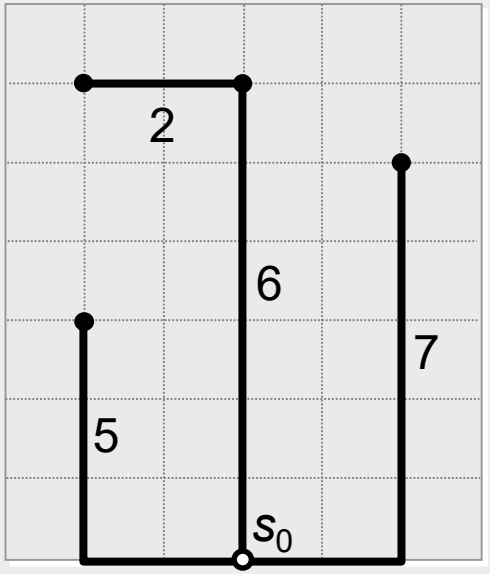
$$\min : \sum_{e \in E} L(e) - \alpha \cdot WNS$$

8.4 Timing-Driven Routing

- 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

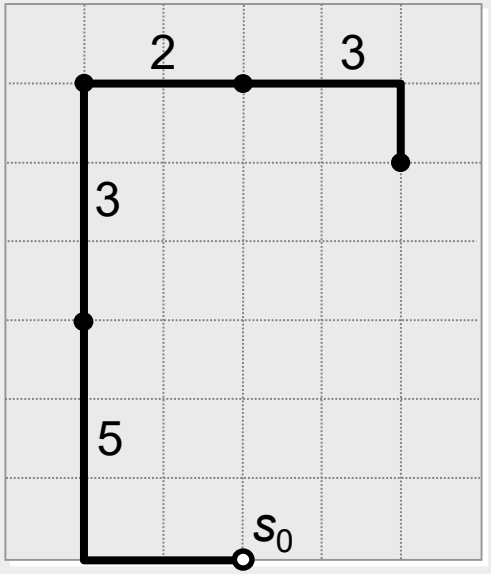
- For a signal net *net*, let
  - s_0 be the source node
  - $sinks = \{s_1, \ldots, s_n\}$ be the sinks
  - $G = (V,E)$ be a corresponding weighted graph where:
  - $V = \{v_0, v_1, \ldots, v_n\}$ represents the source and sink nodes of *net*, and
  - the weight of an edge $e(v_i,v_j) \in E$ represents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

- 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



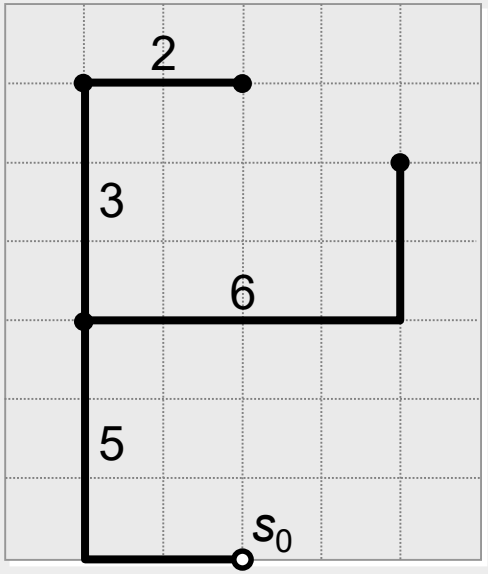
radius(*T*) = 8
cost(*T*) = 20

"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

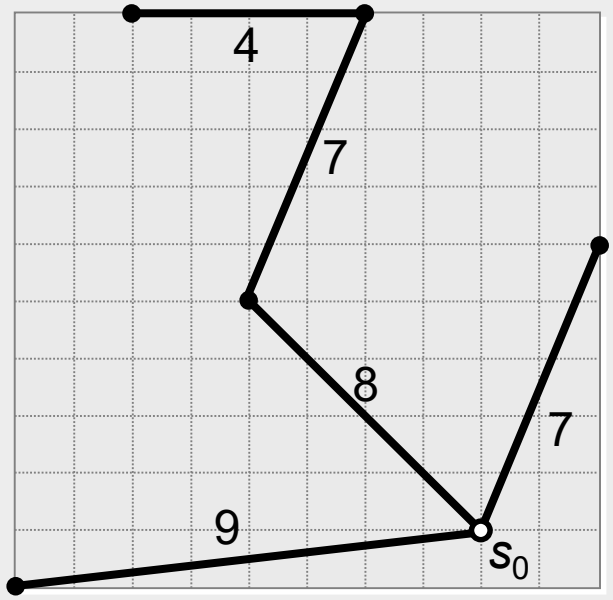
$$cost\ (T_{BRBC}) \leq \left(1+\frac{2}{\varepsilon}\right) \cdot cost\ (T_M)$$

- When ε = 0, T_{BRBC} has minimum radius
- When ε = ∞, T_{BRBC} has minimum cost

8.4.2 Prim-Dijkstra Tradeoff

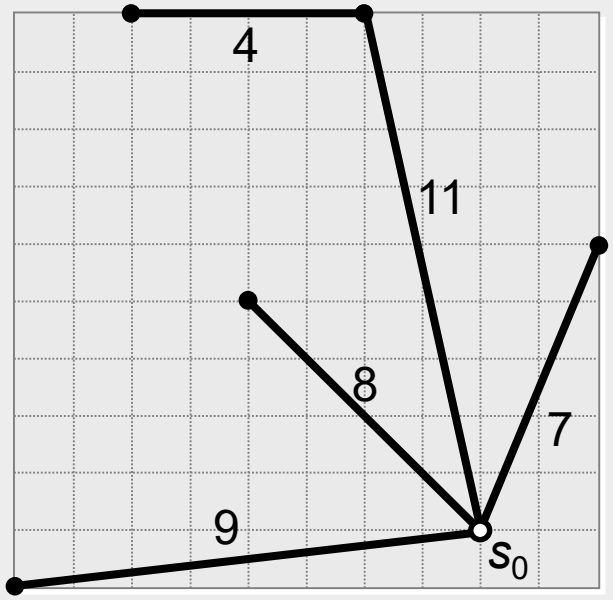
- Prim-Dijkstra Tradeoff based on Prim's algorithm and Dijkstra's algorithm
- From the set of sinks *S*, iteratively add sink *s* based on different cost function
  - Prim's algorithm cost function: $cost(s_i, s_j)$
  - Dijkstra's algorithm cost function: $cost(s_0, s_i) + cost(s_i, s_j)$
  - Prim-Dijkstra 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 $\alpha(i)$ 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

8.5 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

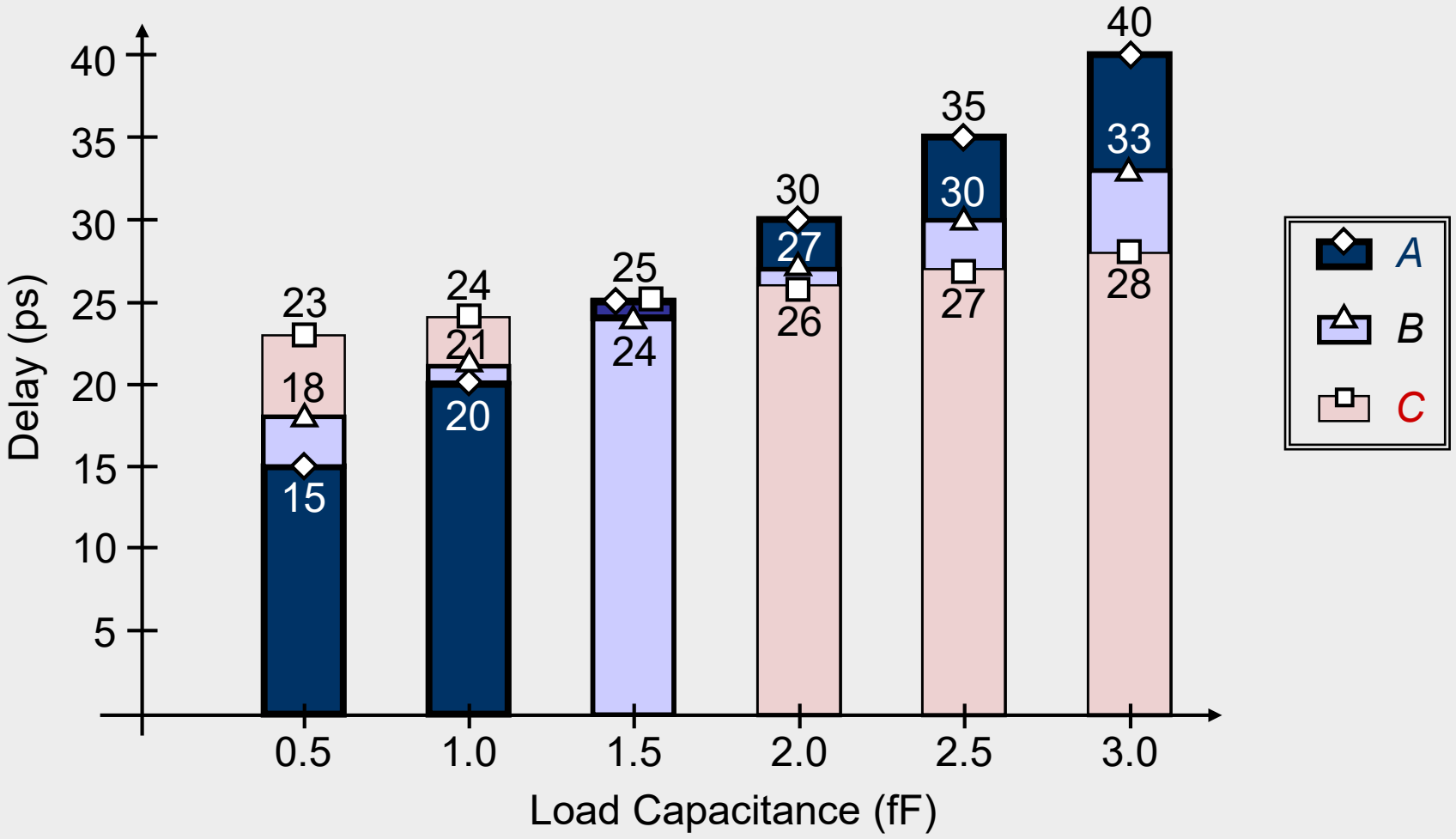
- Let a gate *v* have 3 sizes *A*, *B*, *C*, where: $size\ (v_C) > size\ (v_B) > size\ (v_A)$
- Gate with a larger size has lower output resistance
- When load capacitances are large:

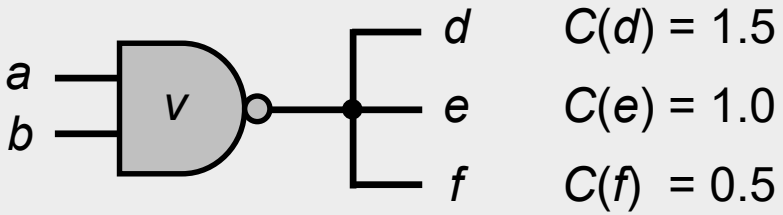
$$t(v_C) < t(v_B) < t(v_A)$$

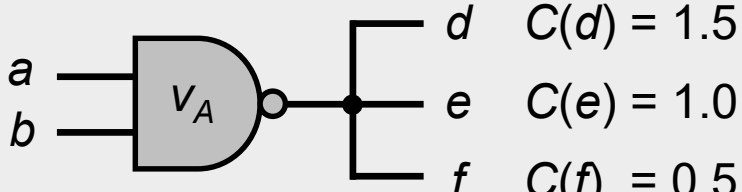
- Gate with a smaller size has higher output resistance
- When load capacitances are small:

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

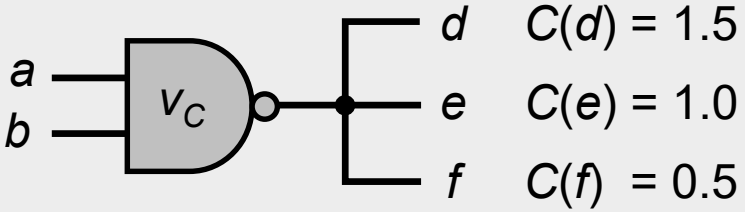
- Let a gate *v* have 3 sizes *A*, *B*, *C*, where: $size\ (v_C) > size\ (v_B) > size\ (v_A)$




a
b
v
d
e
f
C(d) = 1.5
C(e) = 1.0
C(f) = 0.5


a
b
v_A
d
e
f
C(d) = 1.5
C(e) = 1.0
C(f) = 0.5

$t(v_A) = 40$


a
b
v_C
d
e
f
C(d) = 1.5
C(e) = 1.0
C(f) = 0.5

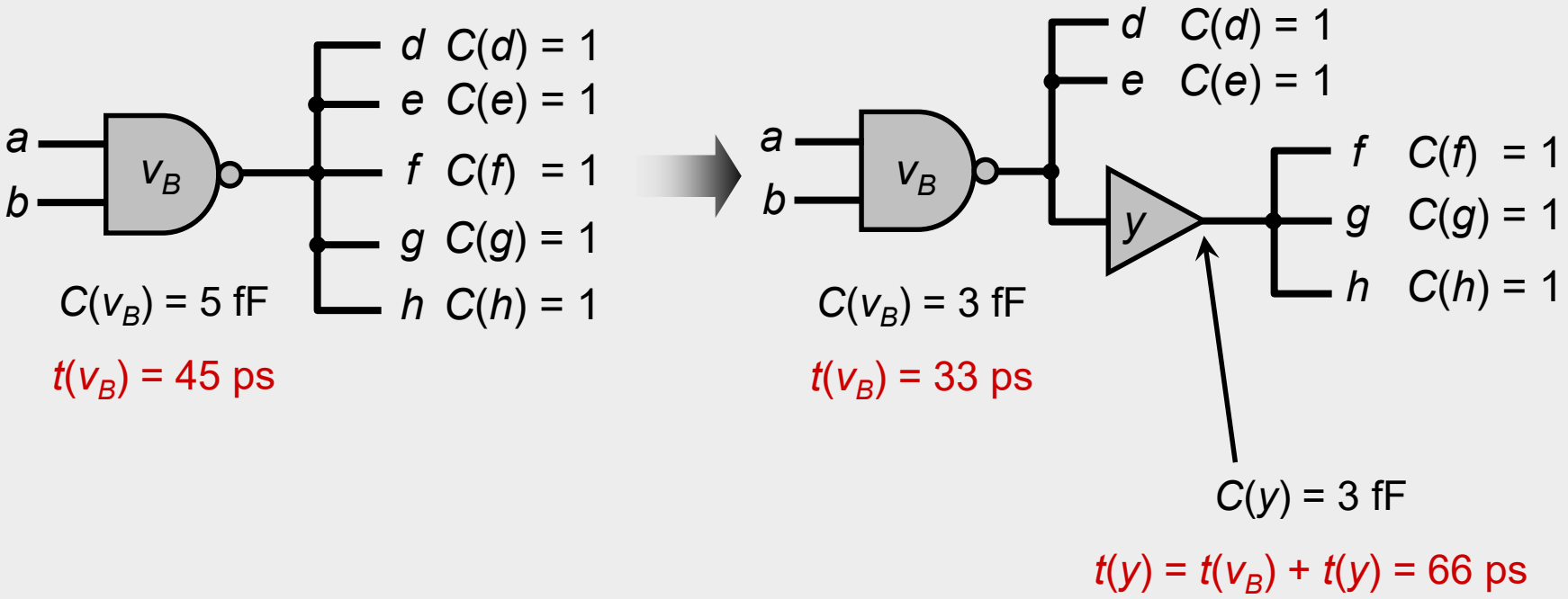
$t(v_C) = 28$

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


a
b
v_B
d C(d) = 1
e C(e) = 1
f C(f) = 1
g C(g) = 1
h C(h) = 1
$C(v_B)$ = 5 fF
$t(v_B)$ = 45 ps
a
b
v_B
d C(d) = 1
e C(e) = 1
y
f C(f) = 1
g C(g) = 1
h 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

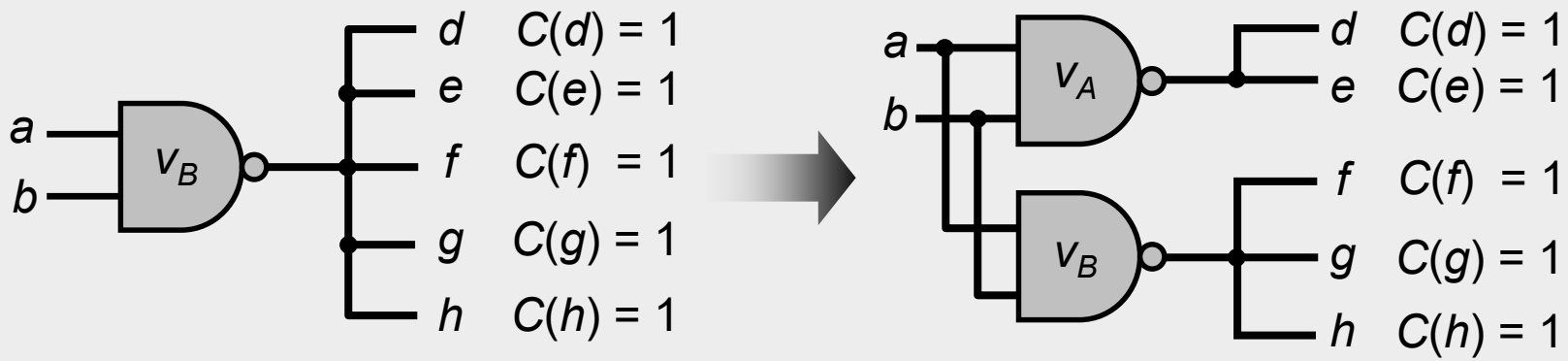
8.5.3 Netlist Restructuring

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

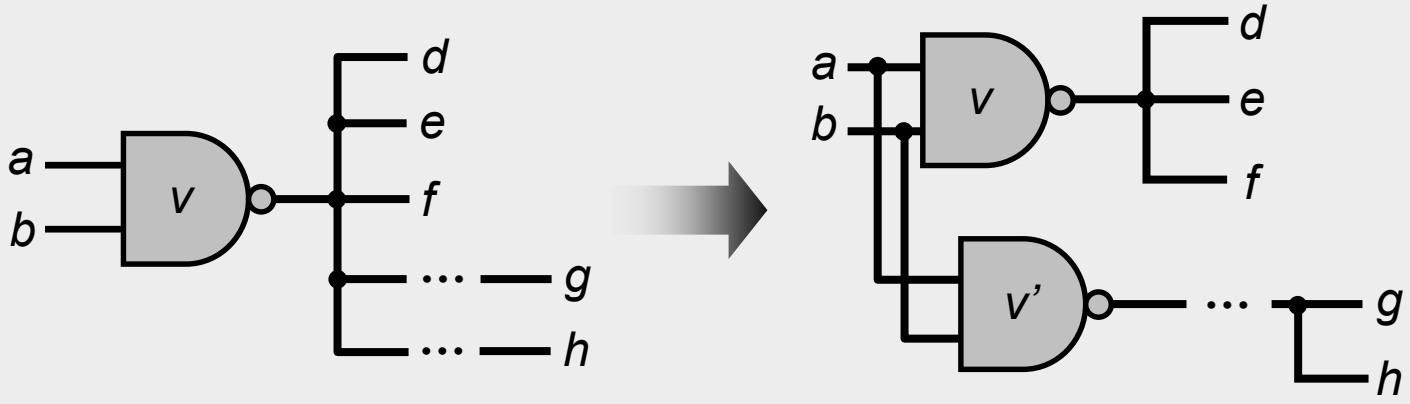
- Changes include
  - Cloning: duplicating gates
  - Redesign of fanin or fanout tree: changing the topology of gates
  - Swapping communicative pins: changing the connections
  - Gate decomposition: e.g., changing AND-OR to NAND-NAND
  - Boolean restructuring: e.g., applying Boolean laws to change circuit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

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

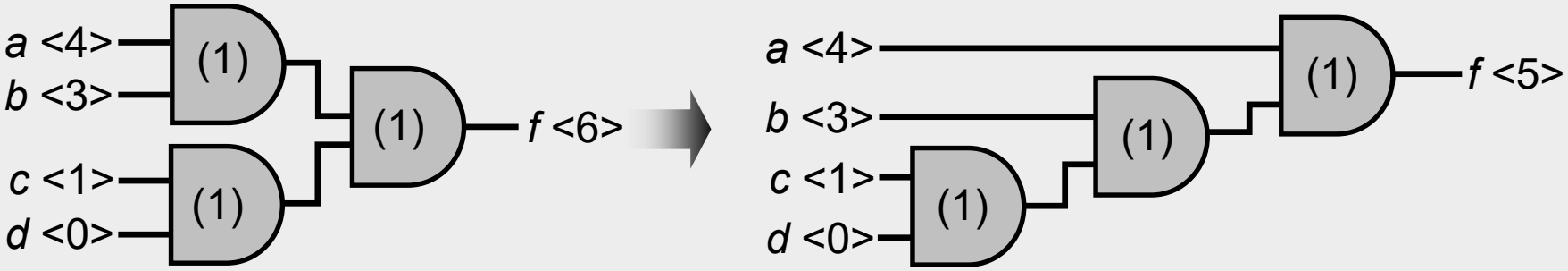
- Cloning can reduce fanout capacitance



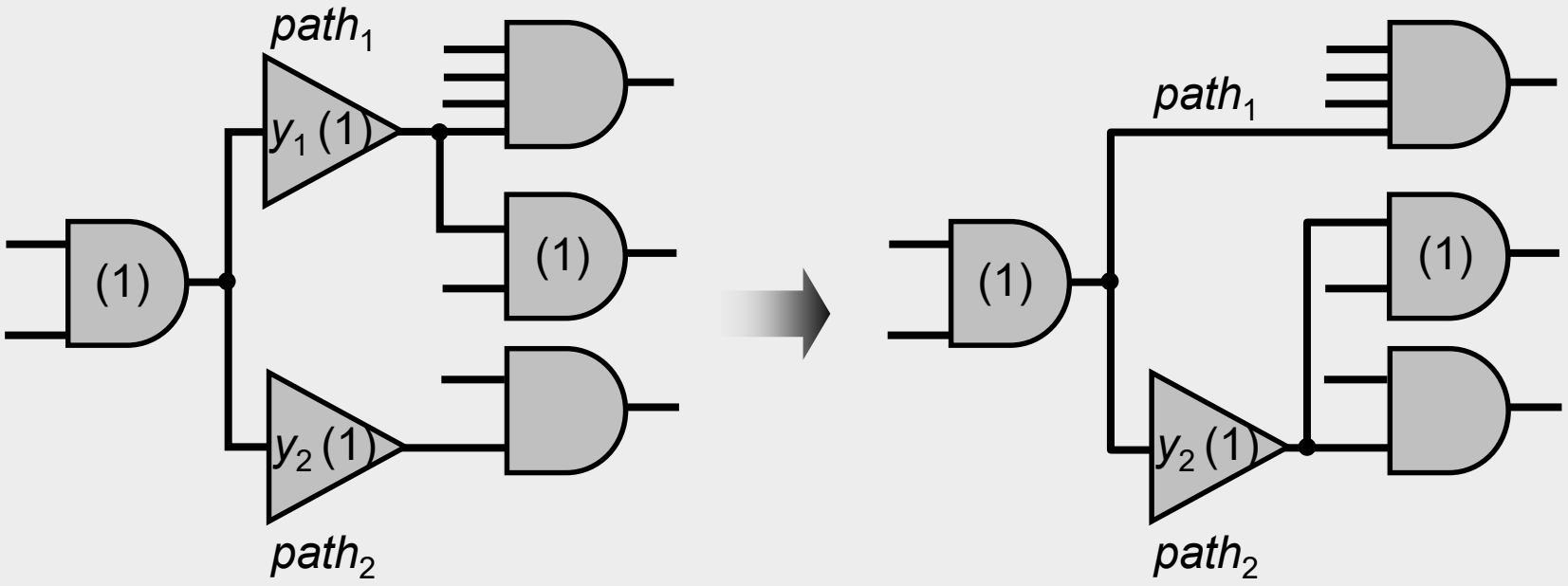
- and reduce downstream capacitance



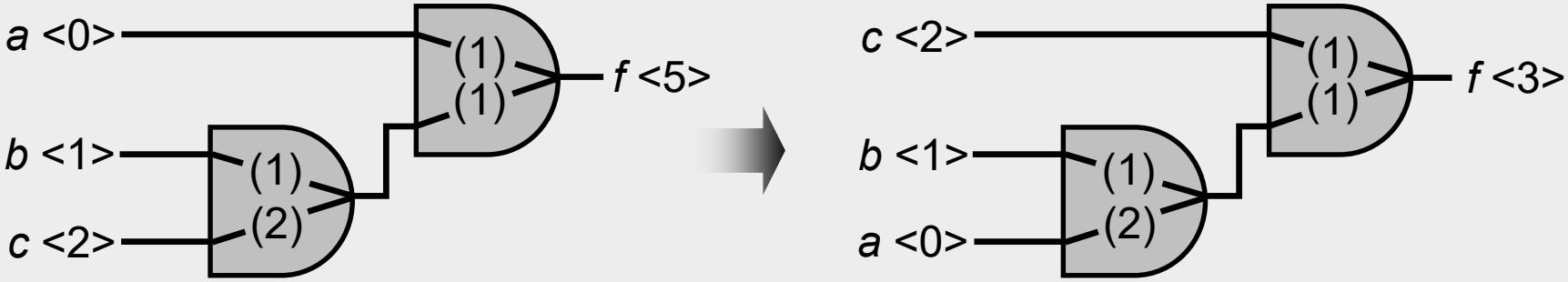
Redesigning the fanin tree can change AATs



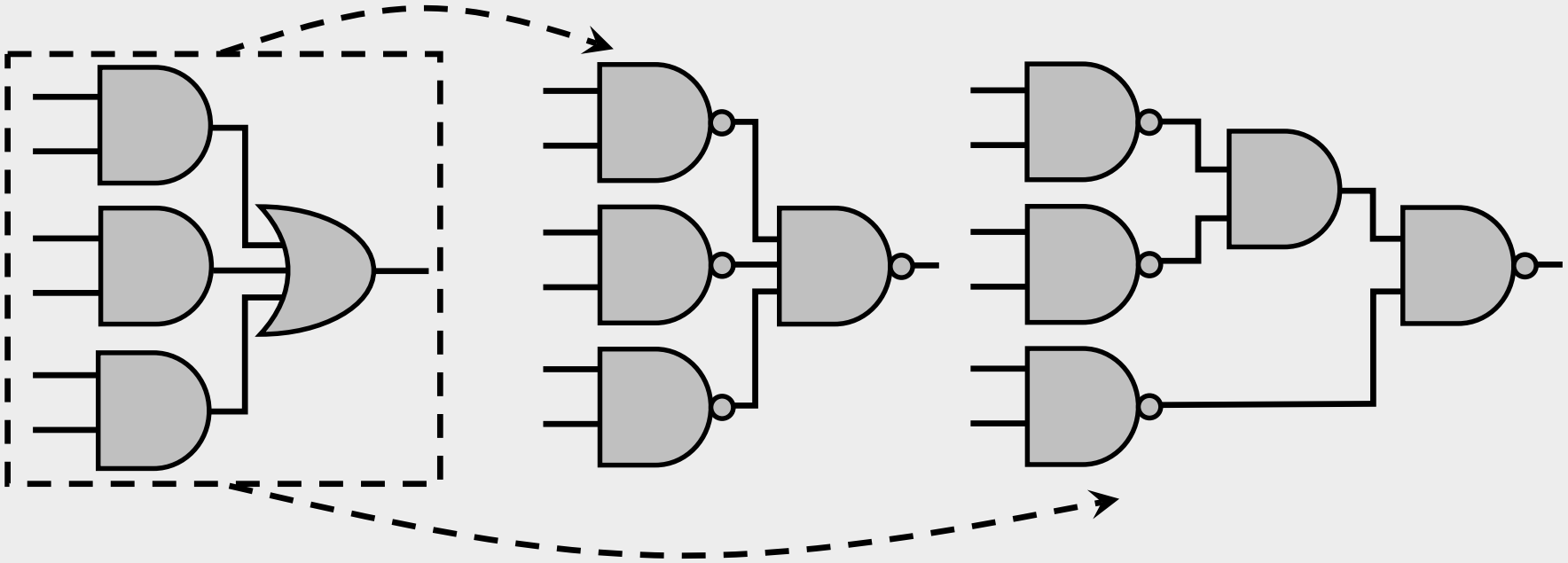
Redesigning fanout trees can change delays on specific paths



Swapping commutative pins can change the final delay

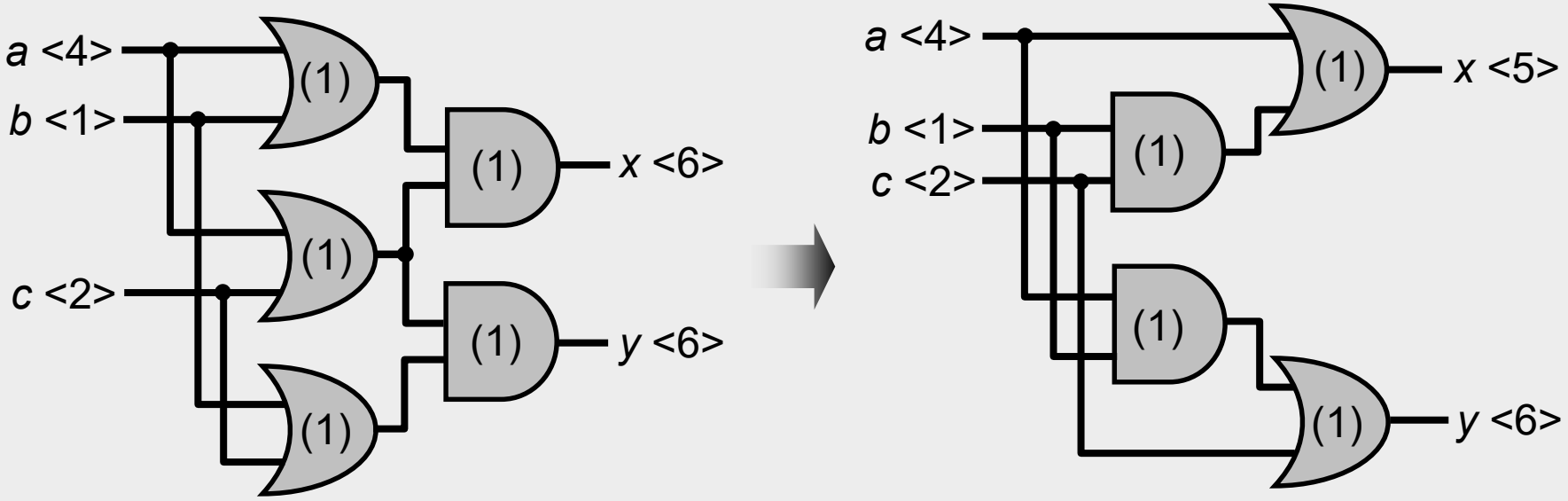


Gate decomposition can change the general structure of the circuit



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

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



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

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

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

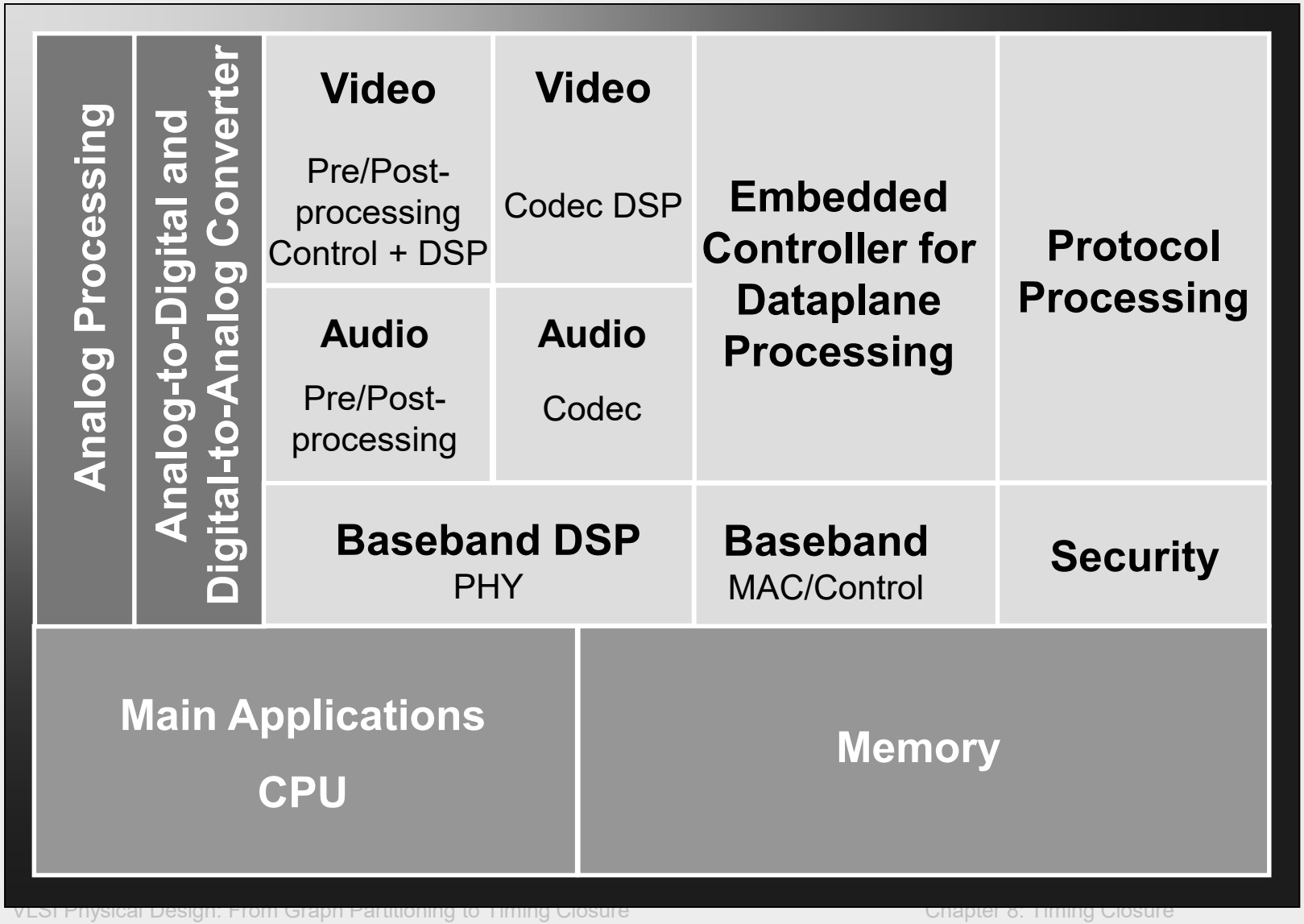
$y(a,b,c) = ab + c$

8.6 Performance-Driven Design Flow

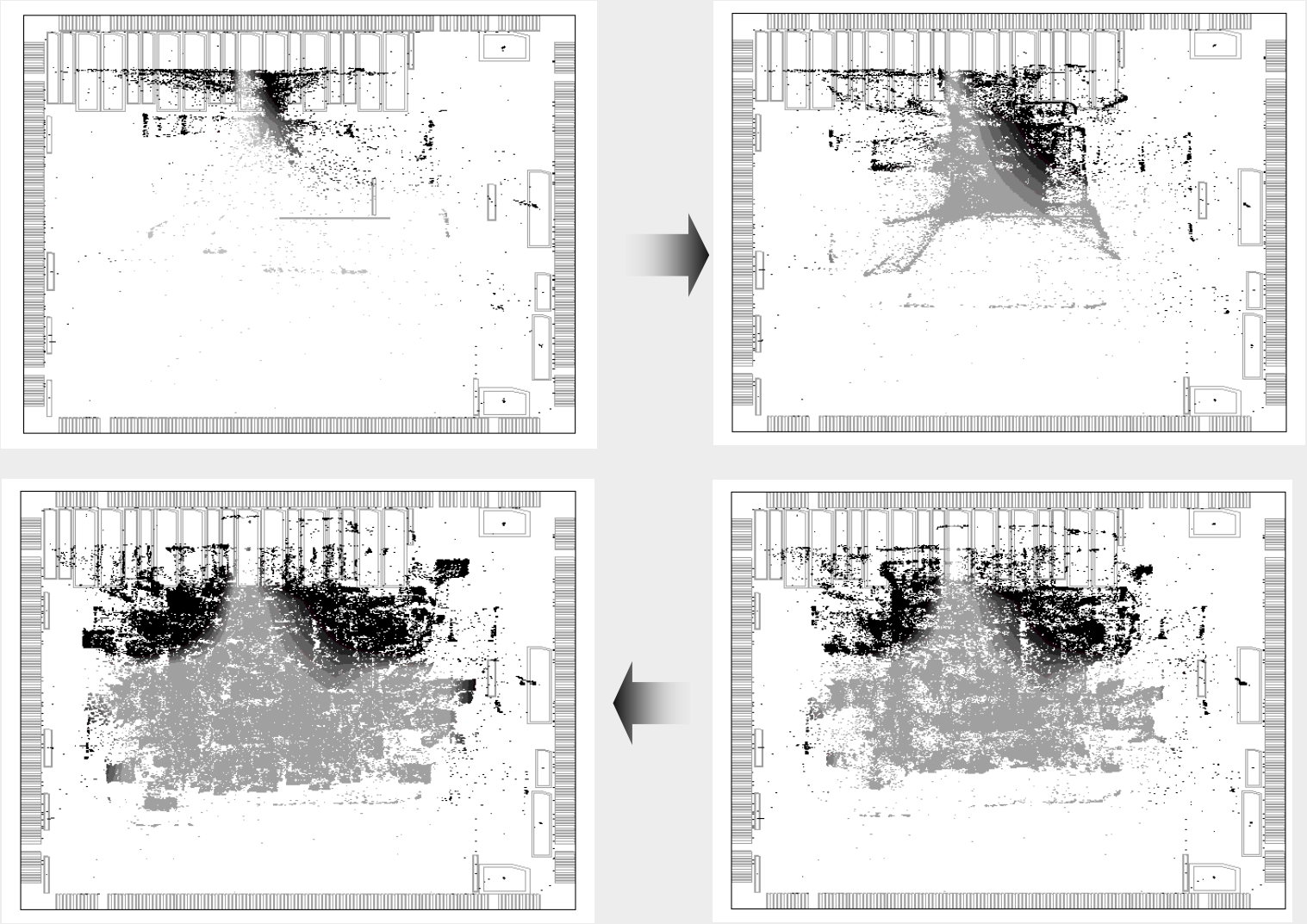
Baseline Physical Design Flow

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

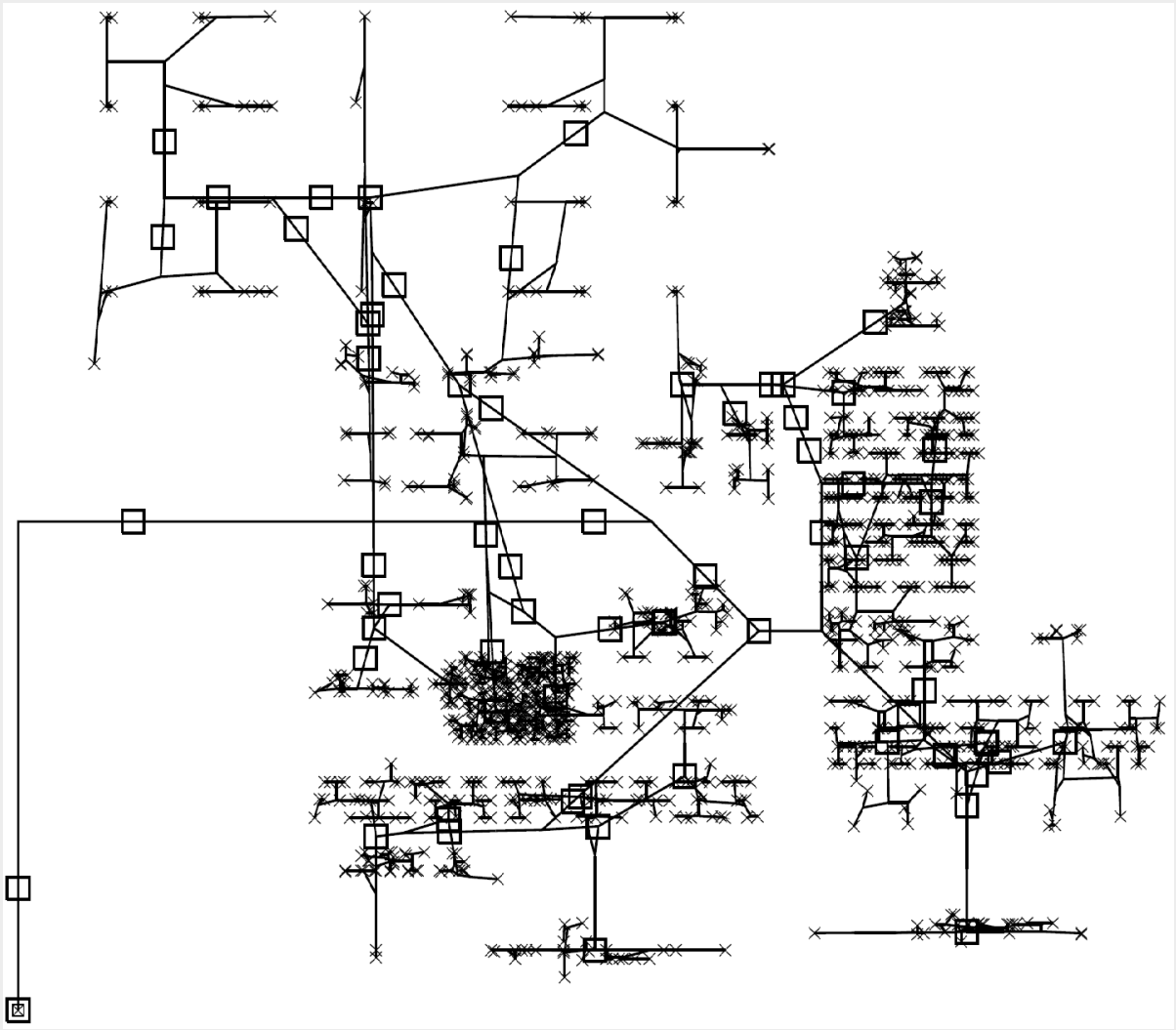
Floorplanning Example



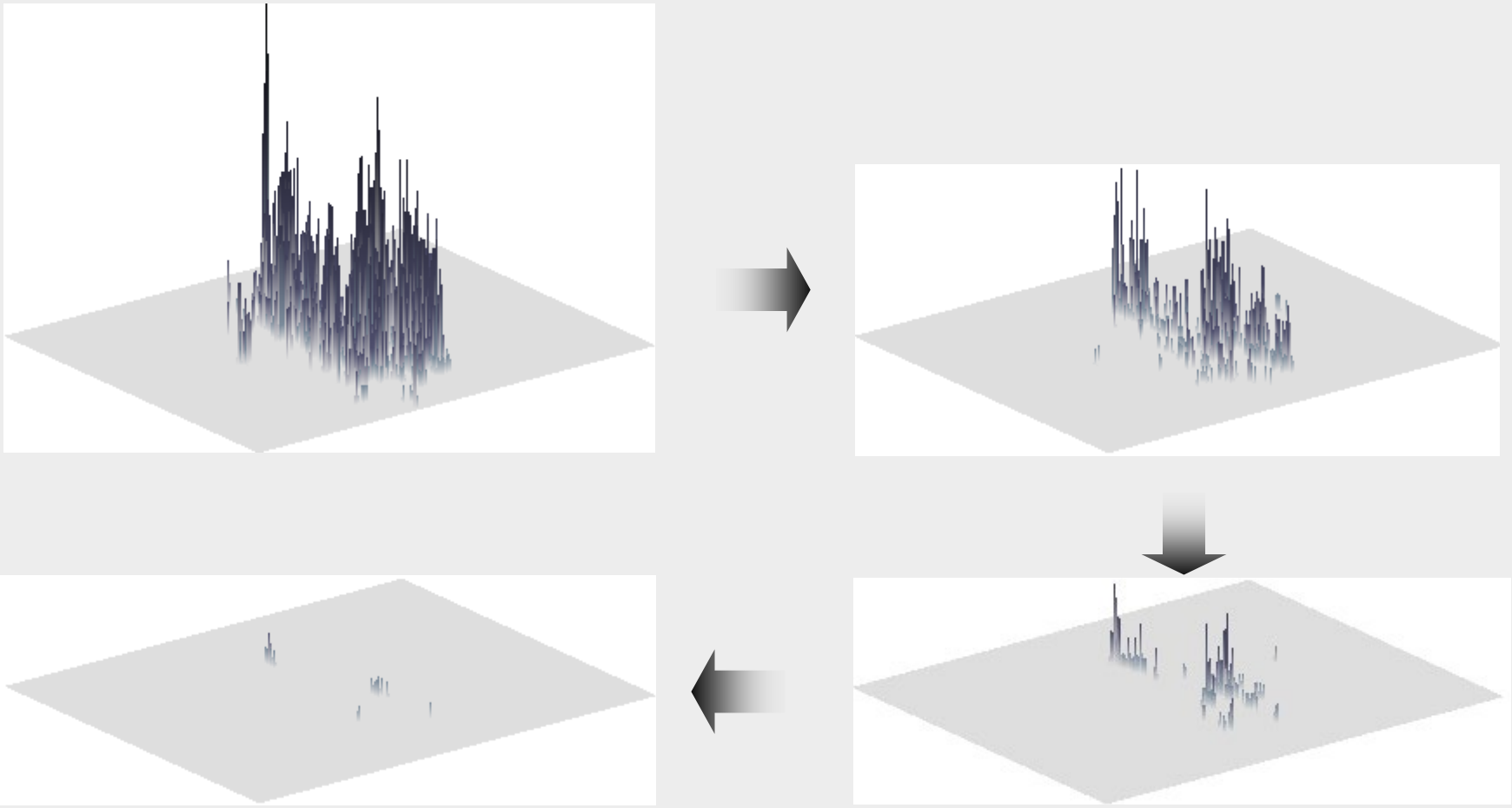
Global Placement Example

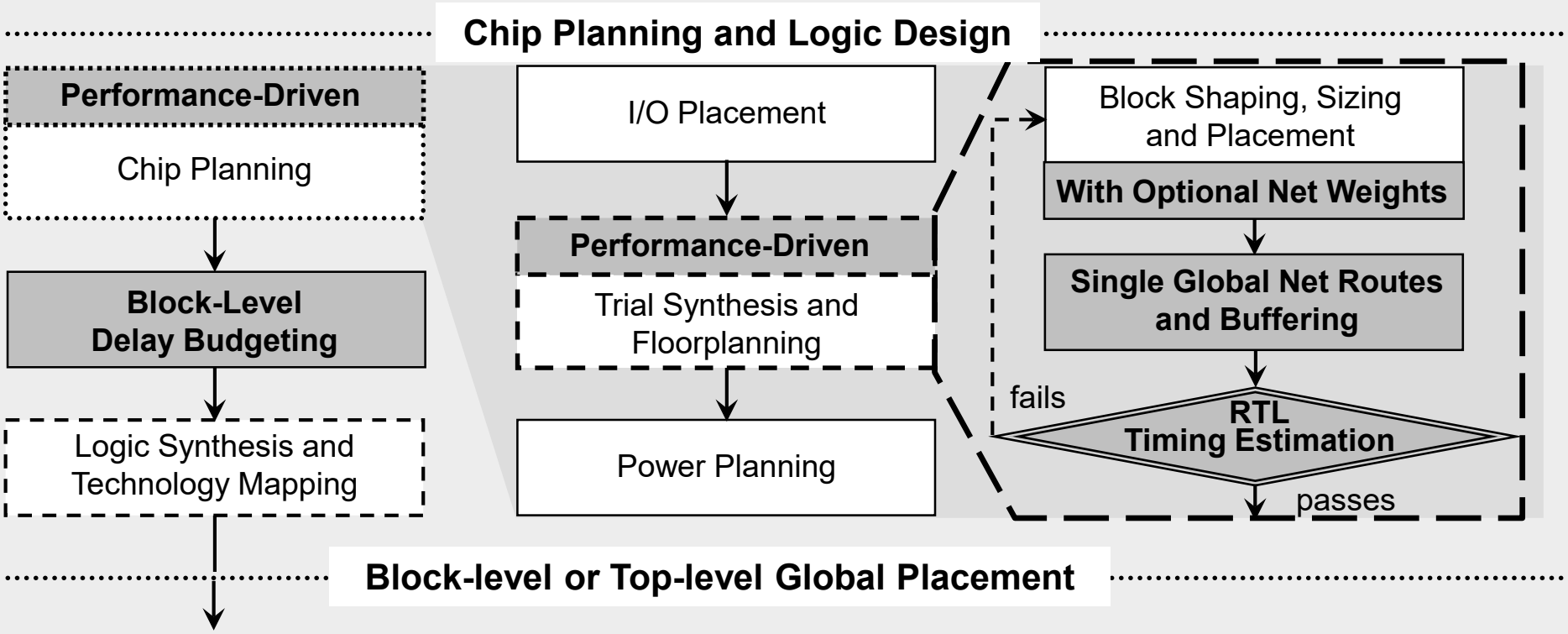


Clock Network Synthesis Example

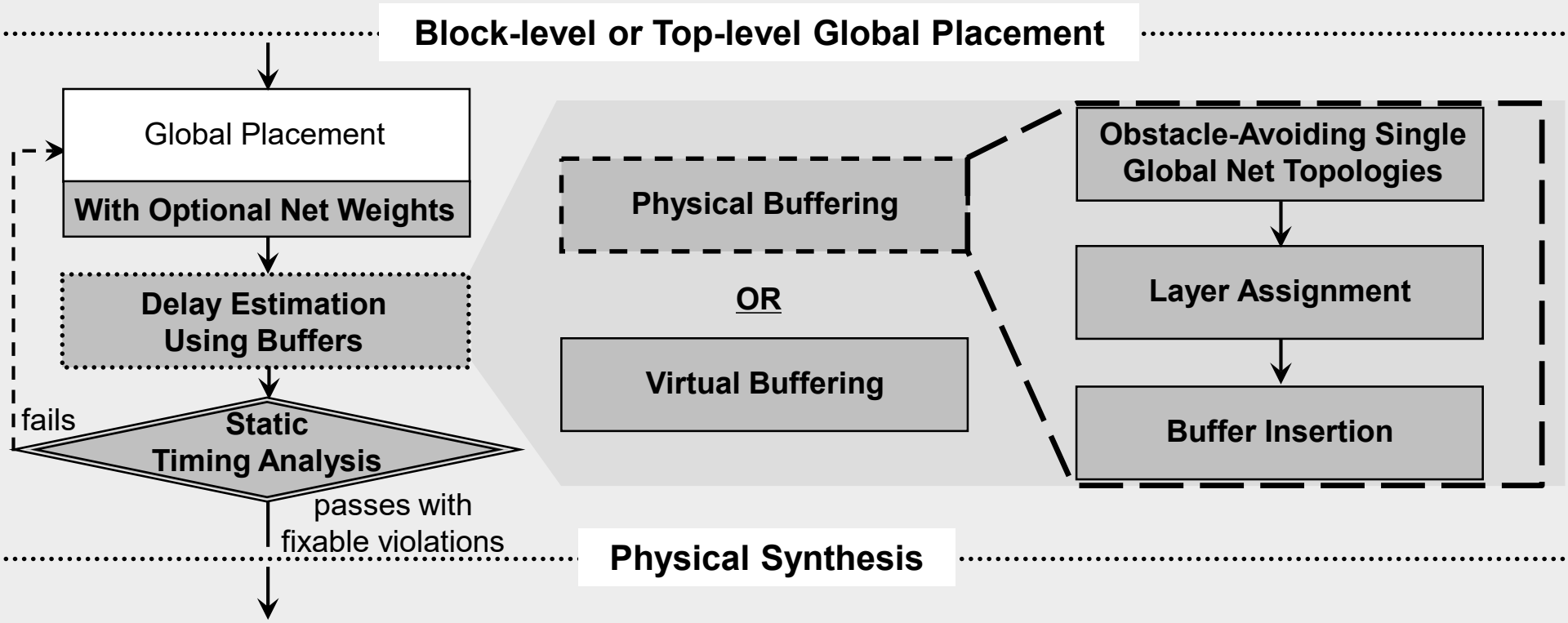


Global Routing Congestion Example

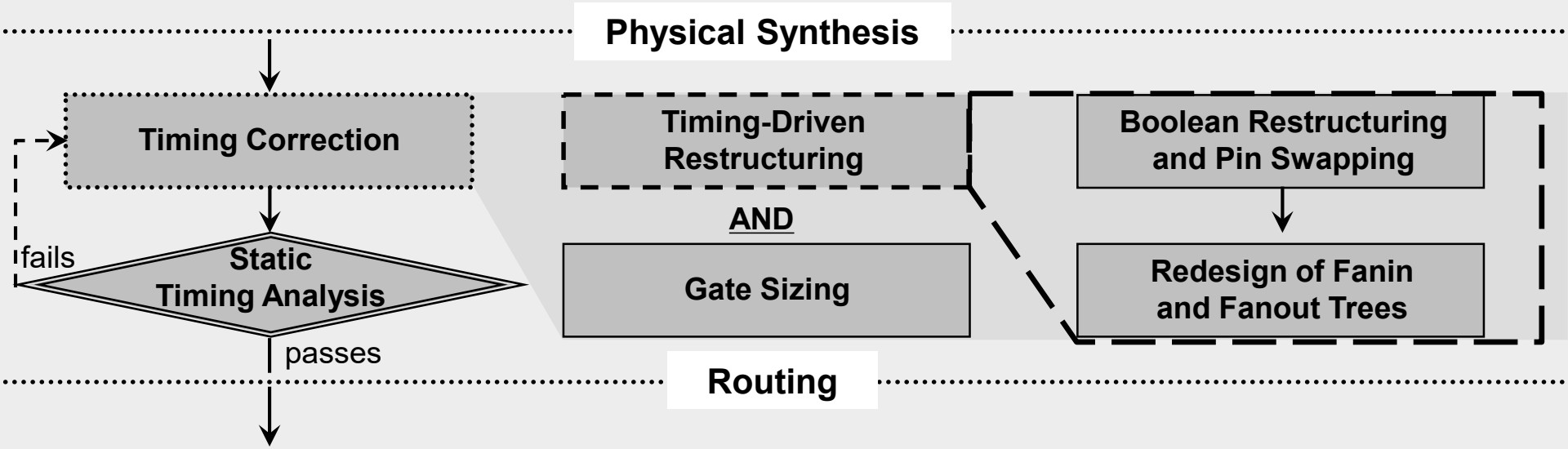




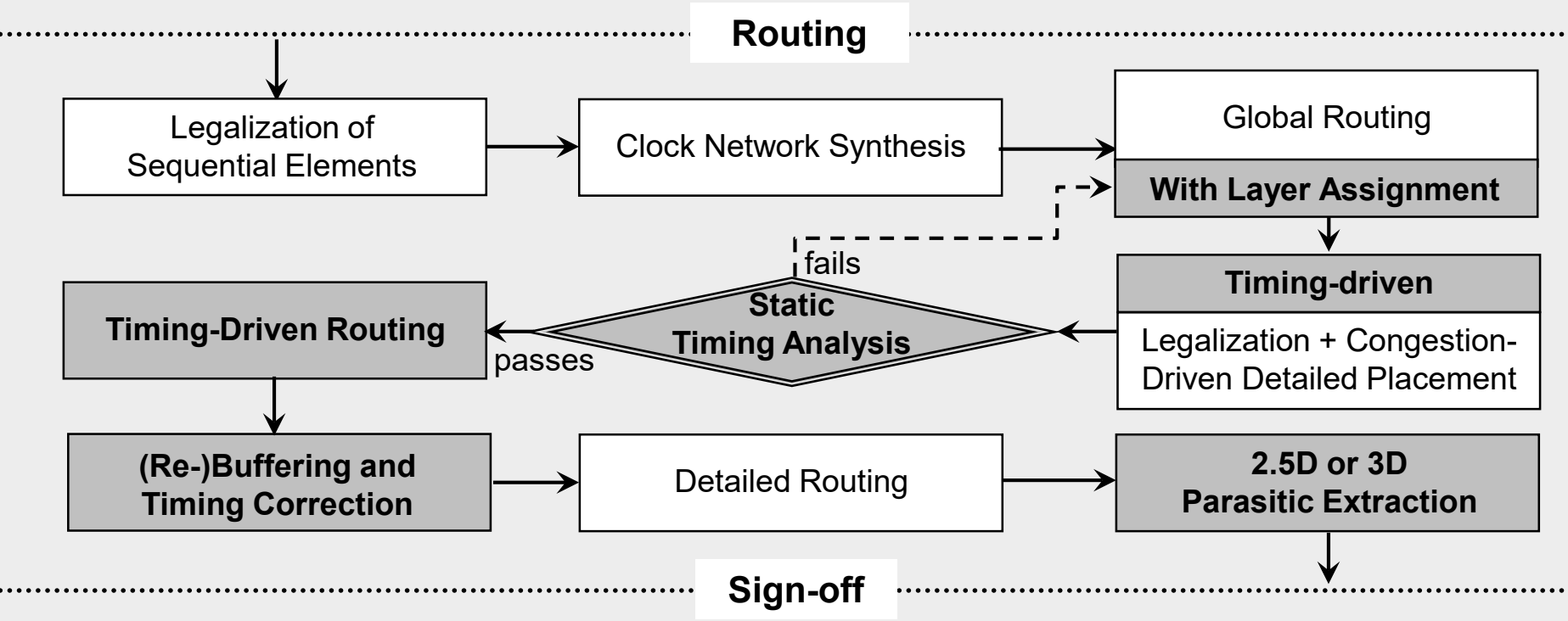
(see full flow chart in Figure 8.26)



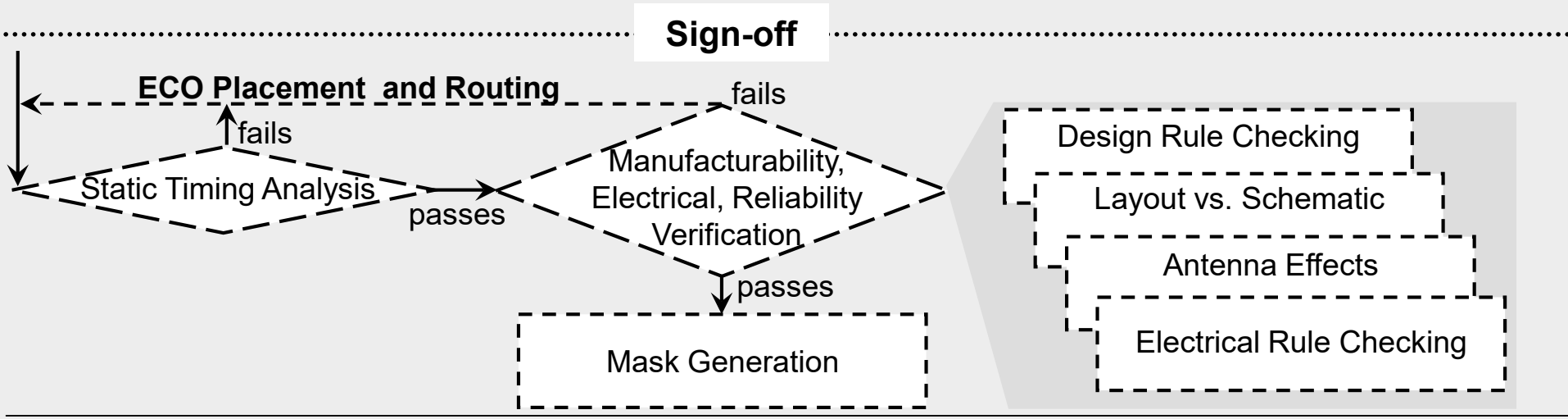
(see full flow chart in Figure 8.26)



(see full flow chart in Figure 8.26)



(see full flow chart in Figure 8.26)



(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