Chapter 8

Timing Closure

8

8 Timing Closure 221
8.1 Introduction 221
8.2 Timing Analysis and Performance Constraints 223
8.2.1 Static Timing Analysis 224
8.2.2 Delay Budgeting with the Zero-Slack Algorithm 229
8.3 Timing-Driven Placement 233
8.3.1 Net-Based Techniques 234
8.3.2 Embedding STA into Linear Programs for Placement . 237
8.4 Timing-Driven Routing 239
8.4.1 The Bounded-Radius, Bounded-Cost Algorithm 240
8.4.2 Prim-Dijkstra Tradeoff 241
8.4.3 Minimization of Source-to-Sink Delay 242
8.5 Physical Synthesis 244
8.5.1 Gate Sizing 244
8.5.2 Buffering 245
8.5.3 Netlist Restructuring 246
8.6 Performance-Driven Design Flow 250
8.7 Conclusions 258
Chapter 8 Exercises 260
Chapter 8 References 262

8 Timing Closure

The layout of an integrated circuit (IC) must not only satisfy geometric requirements, e.g., non-overlapping cells and routability, but also meet the design's *timing constraints*, e.g., *setup* (long-path) and *hold* (short-path) constraints. The optimization process that meets these requirements and constraints is often called *timing closure*. It integrates point optimizations discussed in previous chapters, such as placement (Chap. 4) and routing (Chaps. 5-7), with specialized methods to improve circuit performance. The following components of timing closure are covered in this chapter.

- *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 (Sec. 8.5.1).
  - Inserting buffers into nets to decrease propagation delays (Sec. 8.5.2).
  - Restructuring the circuit along its critical paths (Sec. 8.5.3).

Sec. 8.6 integrates these optimizations in a *performance-driven physical design flow*.

8.1 Introduction

For many years, *signal propagation delay* in *logic gates* was the main contributor to circuit delay, while wire delay was negligible. Therefore, cell placement and wire routing did not noticeably affect circuit performance. Starting in the mid-1990s, technology scaling significantly increased the relative impact of wiring-induced delays, making high-quality placement and routing critical for timing closure.

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.

A. B. Kahng et al., *VLSI Physical Design: From Graph Partitioning to Timing Closure,*
DOI 10.1007/978-90-481-9591-6_8, © Springer Science+Business Media B.V. 2011

Setup constraints ensure that no signal transition occurs too late. Initial phases of timing closure focus on these types of constraints, which are formulated as follows.

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

Here, t_{cycle} is the clock period, $t_{combDelay}$ is the longest path delay through combinational logic, t_{setup} is the setup time of the receiving storage element (e.g., flip-flop), and t_{skew} is the *clock skew* (Sec. 7.4). Checking whether a circuit meets setup constraints requires estimating how long signal transitions will take to propagate from one storage element to the next. Such delay estimation is typically based on *static timing analysis* (*STA*), which propagates *actual arrival times* (*AATs*) and *required arrival times* (*RATs*) to the pins of every gate or cell. STA quickly identifies *timing violations*, and diagnoses them by tracing out *critical paths* in the circuit that are responsible for these timing failures (Sec. 8.2.1).

Motivated by efficiency considerations, STA does not consider circuit functionality and specific signal transitions. Instead, STA assumes that every cell propagates every 0-1 (1-0) transition from its input(s) to its output, and that every such propagation occurs with the worst possible delay[1]. Therefore, STA results are often pessimistic for large circuits. This pessimism is generally acceptable during optimization because it affects competing layouts equally, without biasing the optimization toward a particular layout. It is also possible to evaluate the timing of several competing layouts with more accurate techniques in order to choose the best solution.

One approach to mitigate pessimism in STA is to analyze the most critical paths. Some of these can be *false paths* – those that cannot be sensitized by any input transition because of the logic functions implemented by the gates or cells. IC designers often enumerate false paths that are likely to become timing-critical to exclude them from STA results and ignore them during timing optimization.

STA results are used to estimate how important each cell and each net are in a particular layout. A key metric for a given timing point g – that is, a pin of a gate or cell – is *timing slack*, the difference between g's RAT and AAT: $slack(g) = RAT(g) - AAT(g)$. Positive slack indicates that timing is met – the signal arrives before it is required – while negative slack indicates that timing is violated – the signal arrives after its required time. Algorithms for timing-driven layout guide the placement and routing processes according to timing slack values.

Guided by slack values, *physical synthesis* restructures the netlist to make it more suitable for high-performance layout implementation. For instance, given an unbalanced tree of gates, (1) the gates that lie on many critical paths can be upsized to propagate signals faster, (2) buffers may be inserted into long critical wires, and (3) the tree can be restructured to decrease its overall depth.

[1] Path-based approaches for timing optimizations are discussed in Secs. 8.3-8.4.

Hold-time constraints ensure that signal transitions do not occur too early. Hold violations can occur when a signal path is too short, allowing a receiving flip-flop to capture the signal at the current cycle instead of the next cycle. The hold-time constraint is formulated as follows.

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

Here, $t_{combDelay}$ is the delay of the circuit's combinational logic, t_{hold} is the hold time required for the receiving storage element, and t_{skew} is the clock skew. As clock skew affects hold-time constraints significantly more than setup constraints, hold-time constraints are typically enforced after synthesizing the clock network (Sec. 7.4).

Timing closure is the process of satisfying timing constraints through layout optimizations and netlist modifications. It is common to use verbal expressions such as "the design has closed timing" when the design satisfies all timing constraints.

This chapter focuses on individual timing algorithms (Secs. 8.2-8.4) and optimizations (Sec. 8.5), but in practice, these must be applied in a carefully balanced design flow (Sec. 8.6). Timing closure may repeatedly invoke certain optimization and analysis steps in a loop until no further improvement is observed. In some cases, the choice of optimization steps depends on the success of previous steps as well as the distribution of timing slacks computed by STA.

8.2 Timing Analysis and Performance Constraints

8.2

Almost all digital ICs are *synchronous*, *finite state machines* (*FSM*), or *sequential machines*. In FSMs, transitions occur at a set clock frequency. Fig. 8.1 "unrolls" a sequential circuit in time, from one clock period to the next. The figure shows two types of circuit components: (1) clocked *storage elements*, e.g., flip-flops or latches, also referred to as *sequential elements*, and (2) *combinational logic*. During each clock period in the operation of the sequential machine, (1) present-state bits stored in the clocked storage elements flow from the storage elements' output pins, along with system inputs, into the combinational logic, (2) the network of combinational logic then generates values of next-state functions, along with system outputs, and (3) the next-state bits flow into the clocked elements' data input pins, and are stored at the next clock tick.

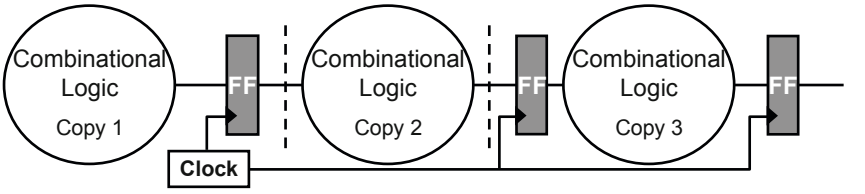


Fig. 8.1 A sequential circuit, consisting of flip-flops and combinational logic, "unrolled" in time.

The maximum clock frequency for a given design depends upon (1) *gate delays*, which are the signal delays due to gate transitions, (2) *wire delays*, which are the delays associated with signal propagation along wires, and (3) *clock skew* (Sec. 7.4). In practice, the predominant sources of delay in standard signals come from gate and wire delays. Therefore, when analyzing setup constraints, this section considers clock skew negligible. A lower bound on the design's clock period is given by the sum of gate and wire delays along any *timing path* through combinational logic – from the output of a storage element to the input of the next storage element. This lower bound on the clock period determines an upper bound on the clock frequency.

In earlier technologies, gate delays accounted for the majority of circuit delay, and the number of gates on a timing path provided a reasonable estimate of path delay. However, in recent technologies, wire delay, along with the component of gate delay that is dependent on capacitive loading, comprises a substantial portion of overall path delay. This adds complexity to the task of estimating path delays and, hence, achievable (maximum) clock frequency.

For a chip to function correctly, *path delay constraints* (Sec. 8.3.2) must be satisfied whenever a signal transition traverses a path through combinational logic. The most critical verification task faced by the designer is to confirm that all path delay constraints are satisfied. To do this *dynamically*, i.e., using circuit simulation is infeasible for two reasons. First, it is computationally intractable to enumerate all possible combinations of state and input variables that can cause a transition, i.e., *sensitize*, a given combinational logic path. Second, there can be an exponential number of paths through the combinational logic. Consequently, design teams often *signoff* on circuit timing *statically*, using a methodology that pessimistically assumes all combinational logic paths can be sensitized. This framework for timing closure is based on *static timing analysis* (*STA*) (Sec. 8.2.1), an efficient, linear-time verification process that identifies *critical paths*.

After critical paths have been identified, *delay budgeting*[2] (Secs. 8.2.2 and 8.3.1) sets upper bounds on the lengths or propagation delays for these paths, e.g., using the *zero-slack algorithm* [8.19], which is covered in Sec. 8.2.2. Other delay budgeting techniques are described in [8.29].

► 8.2.1 Static Timing Analysis

In STA, a combinational logic network is represented as a *directed acyclic graph* (*DAG*) (Sec. 1.7). Fig. 8.2 illustrates a network of four combinational logic gates *x*, *y*, *z* and *w*, three inputs *a*, *b* and *c*, and one output *f*. The inputs are annotated with times 0, 0 and 0.6 time units, respectively, at which signal transitions occur relative

[2] This methodology is intended for layout of circuits directly represented by graphs rather than circuits partitioned into high-level modules. However, this methodology can also be adapted to assign budgets to entire modules instead of circuit elements.

to the start of the clock cycle. Fig. 8.2 also shows gate and wire delays, e.g., the gate delay from the input to the output of inverter x is 1 unit, and the wire delay from input b to the input of inverter x is 0.1 units. For modern designs, gate and wire delays are typically on the order of *picoseconds* (*ps*).

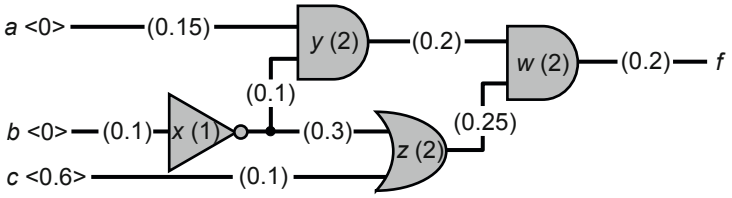


Fig. 8.2 Three inputs a, b and c are annotated with times at which signal transitions occur in angular brackets. Each edge and gate is annotated with its delay in parentheses.

Fig. 8.3 illustrates the corresponding DAG, which has one node for each input and output, as well as one node for each logic gate. For convenience, a source node is introduced with a directed edge to each input. Nodes corresponding to logic gates are labeled with the respective gate delays (e.g., node y has the label 2). Directed edges from the source to the inputs are labeled with transition times, and directed edges between gate nodes are labeled with wire delays. Because this DAG representation has one node per logic gate, it follows the *gate node convention*. The *pin node convention*, where the DAG has a node for each pin of each gate, is more detailed and will be used later in this section.

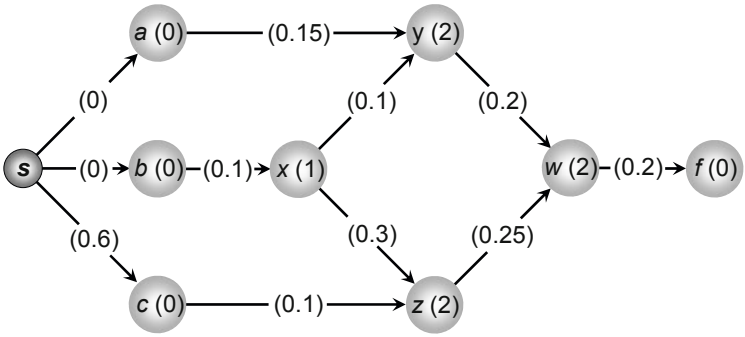


Fig. 8.3 DAG representation using gate node convention of the circuit in Fig. 8.2.

Actual arrival time. In a circuit, the latest transition time at a given node $v \in V$, measured from the beginning of the clock cycle, is the *actual arrival time* (*AAT*), denoted as $AAT(v)$. By convention, this is the arrival time at the *output side* of node v. For example, in Fig. 8.3, $AAT(x) = 1.1$ because of the wire delay from input b (0.1) and the gate delay of inverter x (1.0). For node y, although the signal transitions on the path through a will arrive at time 2.15, the arrival time is dominated by transitions along the path through x. Hence, $AAT(a) = 3.2$. Formally, the AAT of node v is

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

where $FI(v)$ is the set of all nodes from which there exists a directed edge to v, and $t(u,v)$ is the delay on the (u,v) edge. This recurrence enables all AAT values in the DAG to be computed in $O(|V| + |E|)$ time or $O(|\text{gates}| + |\text{edges}|)$. This linear scaling of runtime makes STA applicable to modern designs with hundreds of millions of gates. Fig 8.4 illustrates the AAT computation from the DAG in Fig. 8.3.

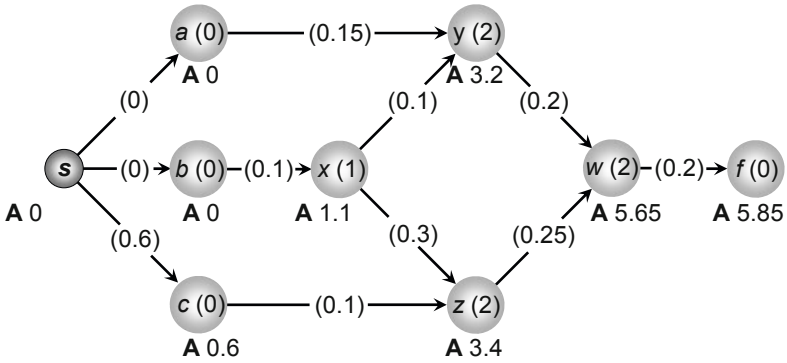


Fig. 8.4 Actual arrival times (AATs) of the DAG, denoted with '**A**', of Fig. 8.3.

Although STA is *pessimistic*, i.e., the longest path in the DAG of a circuit might not actually be sensitizable, the design team must satisfy all timing constraints. An algorithm to find all longest paths from the source in a DAG was proposed by *Kirkpatrick* in 1966 [8.18]. It uses a *topological ordering* of nodes – if there exists a (u,v) edge in the DAG, then u is ordered before v. This ordering can be determined in linear time by reversing the post-order labeling obtained by depth-first search.

Longest Paths Algorithm [8.18]
Input: directed graph $G(V,E)$
Output: AATs of all nodes $v \in V$ based on worst-case (longest) paths

```
1.  foreach (node v ∈ V)
2.    AAT[v] = -∞                          // all AATs are by default unknown
3.  AAT[source] = 0                        //     except source, which is 0
4.  Q = TOPO_SORT(V)                       // topological order
5.  while (Q != Ø)
6.    u = FIRST_ELEMENT(Q)                 // u is the first element in Q
7.    foreach (neighboring node v of u)
8.      AAT[v] = MAX(AAT[v],AAT[u] + t[u][v]) // t[u][v] is the (u,v) edge delay
9.    REMOVE(Q,u)                          // remove u from Q
```

Required arrival time. The *required arrival time* (*RAT*), denoted as $RAT(v)$, is the time by which the latest transition at a given node v must occur in order for the circuit to operate correctly within a given clock cycle. Unlike AATs, which are determined from multiple paths from *upstream* inputs and flip-flop outputs, RATs

are determined from multiple paths to *downstream* outputs and flip-flop inputs. For example, suppose that $RAT(f)$ for the circuit in Fig 8.2 is 5.5. This forces $RAT(w)$ to be 5.3, $RAT(y)$ to be 3.1, and so on (Fig. 8.5). Formally, the RAT of a node v is

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

where $FO(v)$ is the set of all nodes with a directed edge from v.

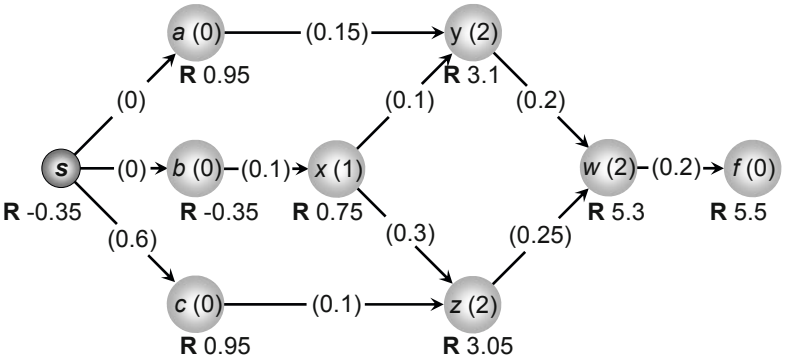


Fig. 8.5 Required arrival times (RATs) of the DAG, denoted with '**R**', of Fig. 8.3.

Slack. Correct operation of the chip with respect to *setup constraints*, e.g., maximum path delay, requires that the AAT at each node does not exceed the RAT. That is, for all nodes $v \in V$, $AAT(v) \leq RAT(v)$ must hold.

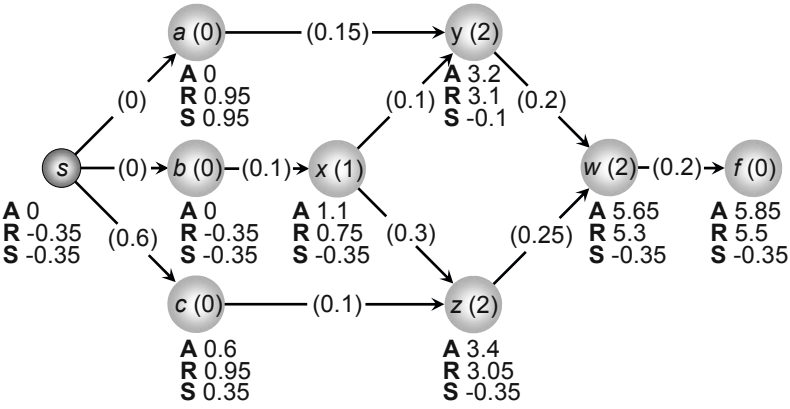


Fig. 8.6 The STA result of the DAG of Fig. 8.3, showing the actual (**A**) and required (**R**) arrival times, and the slack (**S**) of each node.

The *slack* of a node v, defined as

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

is an indicator of whether the timing constraint for v has been satisfied. *Critical paths* or *critical nets* are signals that have *negative slack*, while *non-critical paths* or *non-critical nets* have *positive slack*.

Timing optimization (Sec. 8.5) is the process by which (1) negative slack is increased to achieve design correctness, and (2) positive slack is reduced to minimize overdesign and recover power and area. Fig 8.6 illustrates the full STA computation, including slack, from Fig. 8.3.

A DAG labeled with the *pin node convention* facilitates a more detailed and accurate timing analysis, as the delay of a gate output depends on which input pin has switched. Fig. 8.7 shows the circuit of Fig. 8.2 annotated with the pin node convention, where v_i is the i^{th} input pin of gate v, and v_O is the output pin of v.

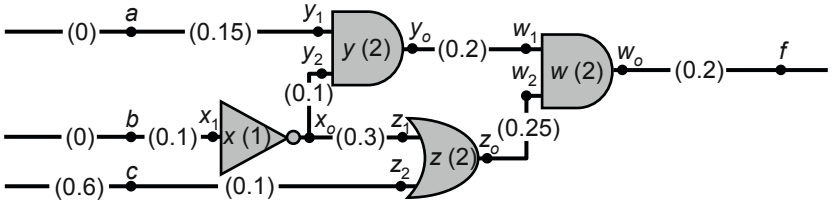


Fig. 8.7 Circuit of Fig. 8.2 annotated with the pin node convention. For a logic gate v, v_i denotes its i^{th} input pin, and v_O denotes its output pin.

Fig. 8.8 shows the result of STA constructed using the pin node convention.

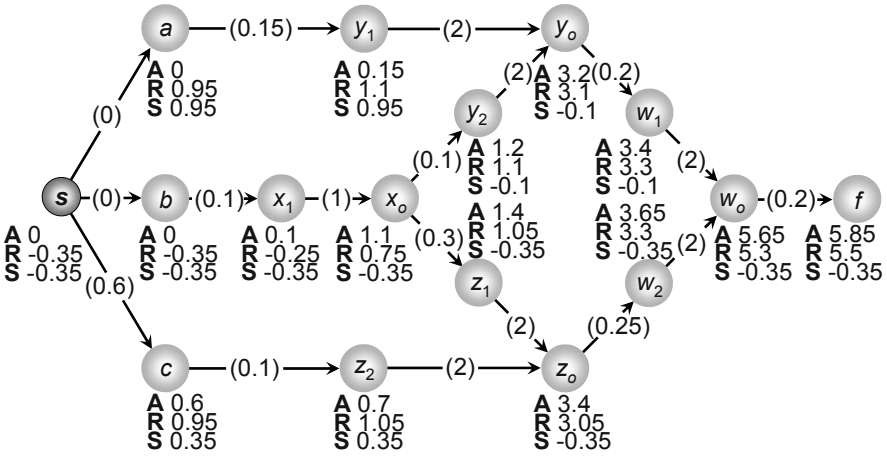


Fig. 8.8 STA result for the circuit of Fig. 8.2, with a DAG constructed using the pin node convention. Each node in the DAG is an input pin or an output pin of a logic gate.

Current practice. In modern designs, separate timing analyses are performed for the cases of *rise delay* (rising transitions) and *fall delay* (falling transitions).

Signal integrity extensions to STA consider changes in delay due to switching activity on neighboring wires of the path under analysis. For signal integrity analysis, the STA engine keeps track of *windows* (*intervals*) of AATs and RATs, and typically executes multiple timing analysis iterations before these timing windows stabilize to a clean and accurate result.

Statistical STA (*SSTA*) is a generalization of STA where gate and wire delays are modeled by random variables and represented by probability distributions [8.21]. Propagated AATs, RATs and timing slacks are also random variables. In this context, timing constraints can be satisfied with high probability (e.g., 95%). SSTA is an increasingly popular methodology choice for leading-edge designs, due to the increased manufacturing variability in advanced process nodes. Propagating statistical distributions instead of intervals avoids some of STA's inherent pessimism. This reduces the costly power, area and schedule impacts of overdesign.

The static verification approach is continually challenged by two fundamental weaknesses – (1) the assumption of a clock and (2) the assumption that all paths are sensitizable. First, STA is not applicable in *asynchronous* contexts, which are increasingly prevalent in modern designs, e.g., asynchronous interfaces in systems-on-chips (SoCs), asynchronous logic design styles to improve speed and power. Second, optimization tools waste considerable runtime and chip resources – e.g., power, area and speed – satisfying "phantom" constraints. In practice, designers can manually or semi-automatically specify *false* and *multicycle paths* – paths whose signal transitions do not need to finish within one clock cycle. Methodologies to fully exploit the availability of such *timing exceptions* are still under development.

► 8.2.2 Delay Budgeting with the Zero-Slack Algorithm

In timing-driven physical design, both gate and wire delays must be optimized to obtain a timing-correct layout. However, there is a chicken-and-egg dilemma: (1) timing optimization requires knowledge of capacitive loads and, hence, actual wirelength, but (2) wirelengths are unknown until placement and routing are completed. To help resolve this dilemma, *timing budgets* are used to establish delay and wirelength constraints for each net, thereby guiding placement and routing to a timing-correct result. The best-known approach to timing budgeting is the *zero-slack algorithm* (*ZSA*) [8.19], which is widely used in practice.

Algorithm. Consider a netlist consisting of logic gates $v_1, v_2, \ldots, v_n$ and nets $e_1, e_2, \ldots, e_n$, where e_i is the output net of gate v_i. Let $t(v)$ be the gate delay of v, and let $t(e)$ be the wire delay of e.[3] The ZSA takes the netlist as input, and seeks to decrease positive slacks of all nodes to zero by increasing $t(v)$ and $t(e)$ values. These increased delay values together constitute the *timing budget* $TB(v)$ of node v, which should not be exceeded during placement and routing.

[3] A multi-fanout net e_i has multiple source-sink delays, so ZSA must be adjusted accordingly.

$$TB(v) = t(v) + t(e)$$

If $TB(v)$ is exceeded, then the place-and-route tool typically (1) decreases the wirelength of e by replacement or rerouting, or (2) changes the size of gate v. The delay impact of a wire or gate size change can be estimated using the *Elmore delay* model [8.13]. If most *arcs* (branches) of a timing path are within budget, then the path may meet its timing constraints even if some arcs exceed their budgets. Thus, another approach to satisfying the timing budget is (3) *rebudgeting*. As in Sec. 8.2.1, let $AAT(v)$, $RAT(v)$ and $slack(v)$ denote respectively the AAT, RAT, and slack at node v in the timing graph G.

Zero-Slack Algorithm (Late-Mode Analysis)
Input: timing graph $G(V,E)$
Output: timing budgets TB for each $v \in V$

```
1.  do
2.     (AAT,RAT,slack) = STA(G)
3.     foreach (v_i ∈ V)
4.        TB[v_i] = DELAY(v_i) + DELAY(e_i)
5.     slack_min = ∞
6.     foreach (v ∈ V)
7.        if ((slack[v] < slack_min) and (slack[v] > 0))
8.           slack_min = slack[v]
9.           v_min = v
10.    if (slack_min ≠ ∞)
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 ≠ ∞)
```

The ZSA consists of three major steps. First, determine the initial slacks of all nodes (lines 2-4), and select a node v_{min} with minimum positive slack $slack_{min}$ (lines 5-9). Second, find a path *path* of nodes that *dominates* $slack(v_{min})$, i.e., any change in delays in *path*'s nodes will cause $slack(v_{min})$ to change. This is done by calling the two procedures *BACKWARD_PATH* and *FORWARD_PATH* (lines 12-13). Third, evenly distribute the slack by increasing $TB(v)$ for each v in *path* (lines 14-17). Each budget increment s will decrement a node slack $slack(v)$. By repeating the process (lines 1-18), the slack of each node in V will end up at zero. The resulting timing budgets at all nodes are the final output of ZSA. Further details of ZSA, including proofs of correctness and complexity analyses, are given in [8.19].

FORWARD_PATH(v_{min},G) constructs a path starting with node v_{min}, and iteratively adds a node v to the path from among the fanouts of the previously-added node in *path* (lines 2-10). Each node v satisfies the condition in line 6, where $RAT(v_{min})$ is

determined by $RAT(v)$, and $AAT(v)$ is determined by $AAT(v_{min})$ – changing the delay of either node affects the slack of both nodes.

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(v_{min},G) iteratively finds a fanin node of the current node so that both nodes' slacks will change if either node's delay is changed.

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
```

Early-mode analysis. ZSA uses *late-mode analysis* with respect to *setup constraints*, i.e., the latest times by which signal transitions can occur for the circuit to operate correctly. Correct operation also depends on satisfying *hold-time constraints* on the earliest signal transition times. *Early-mode analysis* considers these constraints. If the data input of a sequential element changes too soon after the triggering edge of the clock signal, the logic value at the output of that sequential element may become incorrect during the current clock cycle. As geometries shrink, early-mode violations have become an overriding concern. While setup violations can be avoided by lowering the chip's operating frequency, the chip's cycle time

does not affect hold-time constraints (Sec. 8.1). Violations of hold-time constraints typically result in chip failures.

To correctly analyze this timing constraint, the *earliest actual arrival time* of signal transitions at each node must be determined. The *required arrival time* of a sequential element *in early mode* is the time at which the earliest signal can arrive and still satisfy the library-cell hold-time requirement.

For each gate v, $AAT_{EM}(v) \geq RAT_{EM}(v)$ must be satisfied, where $AAT_{EM}(v)$ is the earliest actual arrival time of a signal transition, and $RAT_{EM}(v)$ is the required arrival time in early mode, at gate v. The early-mode slack is then defined as

$$slack_{EM}(v) = AAT_{EM}(v) - RAT_{EM}(v)$$

When adapted to early-mode analysis, ZSA is called the *near zero-slack algorithm*. The adapted algorithm seeks to *decrease* $TB(v)$ by decreasing $t(v)$ or $t(e)$, so that all nodes have minimum early-mode timing slacks. However, since $t(v)$ and $t(e)$ cannot be negative, node slacks may not necessarily all become zero.

Near Zero-Slack Algorithm (Early-Mode Analysis)
Input: timing graph $G(V,E)$
Output: timing budgets TB for each $v \in V$

```
1.  foreach (node v ∈ V)
2.    done[v] = false
3.  do
4.    (RAT_EM,AAT_EM,slack_EM) = STA_EM(G)          // early-mode STA
5.    slack_min = ∞
6.    foreach (node v_i ∈ V)
7.      TB[v_i] = DELAY(v_i) + DELAY(e_i)
8.    foreach (node v ∈ V)
9.      if ((done[v] == false) and (slack_EM[v] < slack_min) and (slack_EM[v] > 0))
10.       slack_min = slack_EM[v]
11.       v_min = v
12.   if (slack_min ≠ ∞)
13.     path = v_min
14.     ADD_TO_FRONT(path,BACKWARD_PATH_ EM(v_min,G))
15.     ADD_TO_BACK(path,FORWARD_PATH _EM(v_min,G))
16.     for (i = 1 to |path|)
17.       node = path[i]
18.       path_E[i] = FIND_EDGE(V[node],E)        // corresponding edge of v_i
19.     for (i = 1 to |path|)
20.       node = path[i]
21.       s = MIN(slack_min / |path|,DELAY(path_E[i]))
22.       TB[node] = TB[node] – s                 // decrease DELAY(node) or
                                                  // DELAY(path_E[node])
23.       if (DELAY(path_E[i]) == 0)
24.         done[node] = true
25. while (slack_min < ∞)
```

In relation to the ZSA pseudocode, the procedure *BACKWARD_PATH_EM*(v_{min},G) is equivalent to *BACKWARD_PATH*(v_{min},G), and *FORWARD_PATH_EM*(v_{min},G) is the equivalent to *FORWARD_PATH*(v_{min},G), except that early-mode analysis is used for all arrival and required times.

Compared to the original ZSA, the two differences are (1) the use of early-mode timing constraints and (2) the handling of the Boolean flag *done*(v). Lines 1-2 set *done*(v) to false for every node v in V. If $t(e_i)$ reaches zero for a node v_i, *done*(v_i) is set to true (lines 23-24). In subsequent iterations (lines 3-25), if v_i is the minimum slack node of G, it will be skipped (line 9) because $t(e_i)$ cannot be decreased further. After the algorithm completes, each node v will either have *slack*(v) = 0 or *done*(v) = true.

In practice, if the delay of a node does not satisfy its early-mode timing budget, the delay constraint can be satisfied by adding additional delay (padding) to appropriate components. However, there is always the danger that additional delay may cause violations of late-mode timing constraints. Thus, a circuit should be first designed with ZSA and late-mode analysis. Early-mode analysis may then be used to confirm that early-mode constraints are satisfied, or to guide circuit modifications to satisfy such constraints.

8.3 Timing-Driven Placement

8.3

Timing-driven placement (*TDP*) optimizes *circuit delay*, either to satisfy all *timing constraints* or to achieve the greatest possible clock frequency. It uses the results of STA (Sec. 8.2.1) to identify *critical nets* and attempts to improve signal propagation delay through those nets. Typically, TDP minimizes one or both of the following. (1) *worst negative slack* (*WNS*)

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

where T is the set of timing endpoints, e.g., primary outputs and inputs to flip-flops, and (2) *total negative slack* (*TNS*)

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

Algorithmic techniques for timing-driven placement can be categorized as *net-based* (Sec. 8.3.1), *path-based* or *integrated* (Sec. 8.3.2). There are two types of net-based techniques – (1) *delay budgeting* assigns upper bounds to the timing or length of individual nets, and (2) *net weighting* assigns higher priorities to *critical nets* during placement. Path-based placement seeks to shorten or speed up entire *timing-critical* paths rather than individual nets. While more accurate than net-based placement, path-based placement does not scale to large, modern designs because the number of

paths in some circuits, such as multipliers, can grow exponentially with the number of gates. Both path-based and net-based approaches (1) rely on support within the placement algorithm, and (2) require a dedicated infrastructure for (incremental) calculation of timing statistics and parameters. Some placement approaches facilitate integration with timing-driven techniques. For instance, net weighting is naturally supported by simulated annealing and all analytic algorithms. Netlist partitioning algorithms support small integer net weights, but can usually be extended to support non-integer weights, either by scaling or by replacing bucket-based data structures with more general priority queues.

Timing-driven placement algorithms often operate in multiple iterations, during which the delay budgets or net weights are adjusted based on the results of STA. Integrated algorithms typically use constraint-driven mathematical formulations in which STA results are incorporated as constraints and possibly in the objective function. Several TDP methods are discussed below, while more advanced algorithms can be found in [8.8], [8.17], [8.20], and Chap. 21 of [8.5].

In practice, some industrial flows do not incorporate timing-driven methods during initial placement because timing information can be very inaccurate until locations are available. Instead, subsequent placement iterations, especially during detailed placement, perform timing optimizations. Integrated methods are commonly used; for example, the linear programming formulation (Sec. 8.3.2) is generally more accurate than net-weighting or delay budgeting, at the cost of increased runtime. A practical design flow for timing closure is introduced in Sec. 8.6.

► 8.3.1 Net-Based Techniques

Net-based approaches impose either quantitative priorities that reflect timing criticality (net weights), or upper bounds on the timing of nets, in the form of *net constraints* (delay budgets). Net weights are more effective at the early design stages, while delay budgets are more meaningful if timing analysis is more accurate. More information on net weighting can be found in [8.12].

Net weighting. Recall that a traditional placer optimizes total wirelength and routability. To account for timing, a placer can minimize the total *weighted* wirelength, where each net is assigned a net weight (Chap. 4). Typically, the higher the net weight is, the more timing-critical the net is considered. In practice, net weights are assigned either *statically* or *dynamically* to improve timing.

Static net weights are computed before placement and do not change. They are usually based on slack – the more critical the net (the smaller the slack), the greater the weight. Static net weights can be either *discrete*, e.g.,

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

where $\omega_1 < \omega_2$ are constants greater than zero, or *continuous*, e.g.,

$$w = \left(1 - \frac{slack}{t}\right)^{\alpha}$$

where t is the longest path delay and α is a criticality exponent.

In addition to slack, various other parameters can be accounted for, such as net size and the number of critical paths traversing a given net. However, assigning too many higher weights may lead to increased total wirelength, routability difficulties, and the emergence of new critical paths. In other words, excessive net weighting may eventually lead to inferior timing. To this end, net weights can be assigned based on *sensitivity*, or how each net affects TNS. For example, the authors of [8.27] define the net weight of *net* as follows. Let

- $w_o(net)$ be the original net weight of *net*
- $slack(net)$ be the slack of *net*
- $slack_{target}$ be the target slack of the design
- $s_w^{SLACK}(net)$ be the slack sensitivity to the net weight of *net*
- $s_w^{TNS}(net)$ be the TNS sensitivity to the net weight of *net*
- α and β be constant bounds on the net weight change that control the tradeoff between WNS and TNS

Then, if $slack(net) \leq 0$,

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

Otherwise, if $slack(net) > 0$, then $w(net)$ remains the same, i.e., $w(net) = w_o(net)$.

Dynamic net weights are computed during placement iterations and keep an updated timing profile. This can be more effective than static net weights, since they are computed before placement, and can become outdated when net lengths change. An example method updates slack values based on efficient calculation of incremental slack for each net *net* [8.7]. For a given iteration k, let

- $slack_{k-1}(net)$ be the slack at iteration $k - 1$
- $s_L^{DELAY}(net)$ be the delay sensitivity to the wirelength of *net*
- $\Delta L(net)$ be the change in wirelength between iteration $k - 1$ and k for *net*

Then, the estimated slack of *net* at iteration k is

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

After the timing information has been updated, the net weights should be adjusted accordingly. In general, this incremental method of weight modification is based on previous iterations. For instance, for each net *net*, the authors of [8.14] first compute the *net criticality* υ at iteration k as

$$\upsilon_k(net) = \begin{cases} \frac{1}{2}(\upsilon_{k-1}(net) + 1) & \text{if } net \text{ is among the 3\% most critical nets} \\ \frac{1}{2}\upsilon_{k-1}(net) & \text{otherwise} \end{cases}$$

and then update the net weights as

$$w_k(net) = w_{k-1}(net) \cdot (1 + \upsilon_k(net))$$

Variants include using the previous j iterations and using different relations between the net weight and criticality.

In practice, dynamic methods can be more effective than using static net weights, but require careful net weight assignment. Unlike static net weights, which are relevant to any placer, dynamic net weights are typically tailored to each type of placer; their computation is integrated with the placement algorithm. To be scalable, the re-computation of timing information and net weights must be efficient [8.7].

Delay budgeting. An alternative to using net weights is to limit the delay, or the total length, of each net by using *net constraints*. This mitigates several drawbacks of net weighting. First, predicting the exact effect of a net weight on timing or total wirelength is difficult. For example, increasing weights of multiple nets may lead to the same (or very similar) placement. Second, there is no guarantee that a net's timing or length will decrease because of a higher net weight. Instead, net-constraint methods have better control and explicitly limit the length or slack of nets. However, to ensure scalability, net constraints must be generated such that they do not over-constrain the solution space or limit the total number of solutions, thereby hurting solution quality. In practice, these net constraints can be generated *statically*, before placement, or *dynamically*, when the net constraints are added or modified during each iteration of placement. A common method to calculate delay budgets is the *zero-slack algorithm* (*ZSA*), previously discussed in Sec. 8.2.2. Other advanced methods for delay budgeting can be found in [8.15].

The support for constraints in each type of placer must be implemented carefully so as to not sacrifice runtime or solution quality. For instance, min-cut placers must choose how to assign cells to partitions while meeting wirelength constraints. To meet these constraints, some cells may have to be assigned to certain partitions. Force-directed placers can adjust the attraction force on certain nets that exceed a certain length, but must ensure that these forces are in balance with those forces on other nets. More advanced algorithms for min-cut and force-directed placers on TDP can be found in [8.16] and [8.26], respectively.

► 8.3.2 Embedding STA into Linear Programs for Placement

Unlike net-based methods, where the timing requirements are mapped to net weights or net constraints, path-based methods for timing-driven placement *directly* optimize the design's timing. However, as the number of (critical) paths of concern can grow quickly, this method is much slower than net-based approaches. To improve scalability, timing analysis may be captured by a *set of constraints* and an *optimization objective* within a mathematical programming framework, such as *linear programming*. In the context of timing-driven placement, a linear program (LP) minimizes a function of slack, such as TNS, subject to two major types of constraints: (1) *physical*, which define the locations of the cells, and (2) *timing*, which define the slack requirements. Other constraints such as electrical constraints may also be incorporated.

Physical constraints. The physical constraints can be defined as follows. Given the set of cells V and the set of nets E, 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 $v \in V_e$,

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

That is, every pin of a given net e must be contained within e's bounding box. Then, e's *half-perimeter wirelength* (*HPWL*) (Sec. 4.2) is defined as

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

Timing constraints. The timing constraints can be defined as follows. 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

Then, define two types of timing constraints – those that account for input pins, and those that account for output pins.

For every input pin v_i of cell v, the arrival time at each v_i is the arrival time at the previous output pin u_o of cell u plus the net delay.

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

For every output pin v_o of cell v, the arrival time at v_o should be greater than or equal to the arrival time plus gate delay of each input v_i. That is, for each input v_i 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, the slack is computed as the difference between the required arrival time $RAT(\tau_p)$ and actual arrival time $AAT(\tau_p)$.

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

The required time $RAT(\tau_p)$ is specified at every input pin of a flip-flop and all primary outputs, and the arrival time $AAT(\tau_p)$ is specified at each output pin of a flip-flop and all primary inputs. To ensure that the program does not over-optimize, i.e., does not optimize beyond what is required to (safely) meet timing, upper bound all pin slacks by zero (or a small positive value).

$$slack(\tau_p) \leq 0$$

Objective functions. Using the above constraints and definitions, the LP can optimize (1) total negative slack (TNS)

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

where $Pins(\tau)$ is the set of pins of cell τ, and T is again the set of all sequential elements or endpoints, or (2) worst-negative slack (WNS)

$$\max : WNS$$

where $WNS \leq slack(\tau_p)$ for all pins, or (3) a combination of wirelength and slack

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

where E is the set of all nets, α is a constant between 0 and 1 that trades off WNS and wirelength, and $L(e)$ is the HPWL of net e.

8.4 Timing-Driven Routing

In modern ICs, interconnect can contribute substantially to total signal delay. Thus, interconnect delay is a concern during the routing stages. *Timing-driven routing* seeks to minimize one or both of (1) *maximum sink delay*, which is the maximum interconnect delay from the source node to any sink of a given net, and (2) *total wirelength*, which affects the load-dependent delay of the net's driving gate.

For a given signal net *net*, let s_0 be the source node and *sinks* = $\{s_1, \ldots, s_n\}$ be the sinks. Let $G = (V,E)$ 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 the terminals 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, and let *cost*(T) be the total edge weight of T.

Because source-sink wirelength reflects source-sink signal delay, i.e., the linear and Elmore delay [8.13] models are well-correlated, a routing tree ideally minimizes both radius and cost. However, for most signal nets, radius and cost cannot be minimized at the same time.

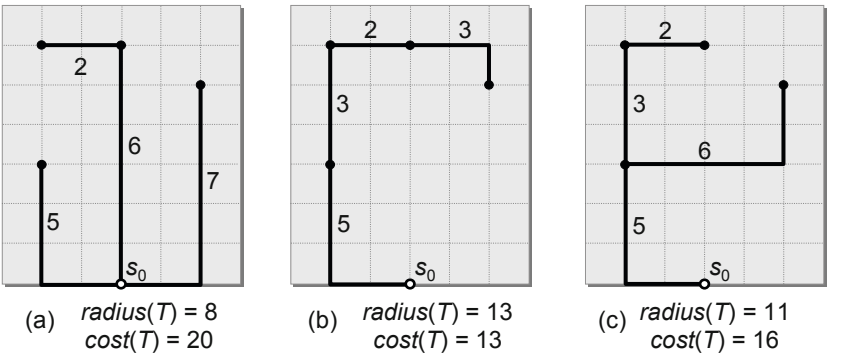


Fig. 8.9 Delay vs. cost, i.e., "shallow" vs. "light", tradeoff in tree construction. (a) A shortest-paths tree. (b) A minimum-cost tree. (c) A compromise with respect to radius (depth) and cost.

Fig. 8.9 illustrates this radius vs. cost ("shallow" vs. "light") tradeoff, where the labels represent edge costs. The tree in Fig. 8.9(a) has minimum radius, and the shortest possible path length from the source to every sink. It is therefore a *shortest-paths tree*, and can be constructed using Dijkstra's algorithm (Sec. 5.6.3). The tree in Fig. 8.9(b) has minimum cost and is a *minimum spanning tree* (*MST*), and can be constructed using Prim's algorithm (Sec. 5.6.1). Due to their respective large cost and large radius, neither of these trees may be desirable in practice. The tree in Fig. 8.9(c) is a compromise that has both shallow and light properties.

► 8.4.1 The Bounded-Radius, Bounded-Cost Algorithm

The *bounded-radius, bounded-cost* (*BRBC*) algorithm [8.11] finds a shallow-light spanning tree with provable bounds on both radius and cost. Each of these parameters is within a constant factor of its optimal value. From graph $G(V,E)$, the algorithm first constructs a *subgraph G'* that contains all $v \in V$, and has small cost and small radius. Then, the shortest-paths tree T_{BRBC} in *G'* will also have small radius and cost because it is a subgraph of *G'*. T_{BRBC} is determined by a parameter $\varepsilon \geq 0$, which trades off between radius and cost. When $\varepsilon = 0$, T_{BRBC} has minimum radius, and when $\varepsilon = \infty$, T_{BRBC} has minimum cost. More precisely, T_{BRBC} satisfies both

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

where T_S is a shortest-paths tree of *G*, and

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

where T_M is a minimum spanning tree of *G*.

BRBC Algorithm
Input: graph $G(V,E)$, parameter $\varepsilon \geq 0$
Output: spanning tree T_{BRBC}

```
1.   T_S = SHORTEST_PATHS_TREE(G)
2.   T_M = MINIMUM_COST_TREE(G)
3.   G' = T_M
4.   U = DEPTH_FIRST_TOUR(T_M)
5.   sum = 0
6.   for (i = 1 to |U| − 1)
7.     u_prev = U[i]
8.     u_curr = U[i + 1]
9.     sum = sum + cost_TM[u_prev][u_curr]      // sum of u_prev ~ u_curr costs in T_M
10.    if (sum > ε · cost_TS[v0][u_curr])       // shortest-path cost in T_S
                                                //    from source v0 to u_curr
11.      G' = ADD(G', PATH(T_S,v0,u_curr))      // add shortest-path edges to G'
12.      sum = 0                                //    and reset sum
13.  T_BRBC = SHORTEST_PATHS_TREE(G')
```

To execute the BRBC algorithm, compute a shortest-paths tree T_S of *G*, and a minimum spanning tree T_M of *G* (lines 1-2). Initialize the graph *G'* to T_M (line 3). Let *U* be the sequence of edges corresponding to any *depth-first tour* of T_M (line 4). This tour traverses each edge of T_M exactly twice (Fig. 8.10), so $cost(U) = 2 \cdot cost(T_M)$. Traverse *U* while keeping a running total *sum* of traversed edge costs (lines 6-9). As the traversal visits each node u_{curr}, check whether *sum* is strictly greater than the cost (distance) between v_0 and u_{curr} in T_S. If so, merge the edges of the $s_0 \sim u_{curr}$ path with

G' and reset *sum* to 0 (lines 11-12). Continue traversing *U* while repeating this process (lines 6-12). Return T_{BRBC}, a shortest-paths tree over *G'* (line 13).

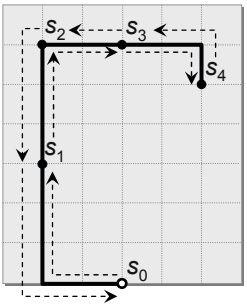


Fig. 8.10 Depth-first tour of the minimum spanning tree in Fig. 8.9 with traversal sequence: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_3 \rightarrow s_2 \rightarrow s_1 \rightarrow s_0$.

► 8.4.2 Prim-Dijkstra Tradeoff

Another method to generate a routing tree trades off radius and cost by using an explicit, quantitative metric. Typically, the minimum cost and minimum radius objectives are optimized by *Prim's minimum spanning tree algorithm* (Sec. 5.6.1) and *Dijkstra's shortest paths tree algorithm* (Sec. 5.6.3), respectively. Although these two algorithms target two different objectives, they construct spanning trees over the set of terminals in very similar ways. From the set of sinks *S*, each algorithm begins with tree *T* consisting only of s_0, and iteratively adds a sink s_j and the edge connecting a sink s_i in *T* to s_j. The algorithms differ only in the *cost function* by which the next sink and edge are chosen.

In Prim's algorithm, sink s_j and edge $e(s_i,s_j)$ are selected to minimize the edge cost between sinks s_i and s_j

$$cost(s_i,s_j)$$

where $s_i \in T$ and $s_j \in S - T$. In Dijkstra's algorithm, sink s_j and edge $e(s_i,s_j)$ are selected to minimize the path cost between source s_0 and sink s_j

$$cost(s_i) + cost(s_i,s_j)$$

where $s_i \in T$, $s_j \in S - T$, and $cost(s_i)$ is the total cost of the shortest path from s_0 to s_i.

To combine the two objectives, the authors of [8.1] proposed the *PD tradeoff*, an explicit compromise between Prim's and Dijkstra's algorithms. This algorithm iteratively adds the sink s_j and the edge $e(s_i,s_j)$ to *T* such that

$$\gamma \cdot cost(s_i) + cost(s_i,s_j)$$

is minimum over all $s_i \in T$ and $s_j \in S - T$ for a prescribed constant $0 \le \gamma \le 1$.

When $\gamma = 0$, the PD tradeoff is identical to Prim's algorithm, and T is a minimum spanning tree T_M. As γ increases, the PD tradeoff constructs spanning trees with progressively higher cost but lower radius. When $\gamma = 1$, the PD tradeoff is identical to Dijkstra's algorithm, and T is a shortest-paths tree T_S.

Fig. 8.11 shows the behavior of the PD tradeoff with different values of γ. The tree in Fig. 8.11(a) is the result of a smaller value of γ, and has smaller cost but larger radius than the tree in Fig. 8.11(b).

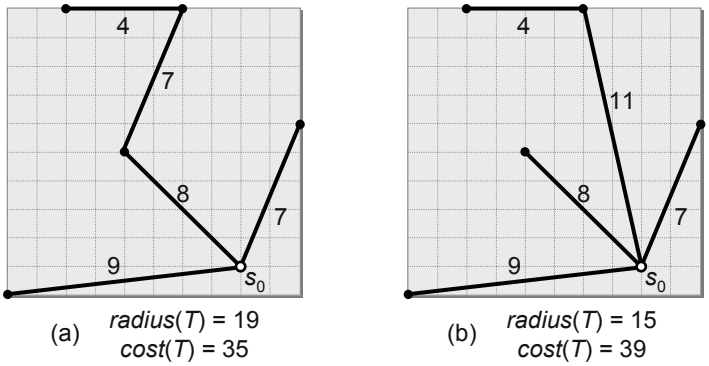


Fig. 8.11 Result of the Prim-Dijkstra (PD) tradeoff. Let the cost between any two terminals be their *Manhattan* distance. (a) A tree with $\gamma = 1/4$. (b) A tree with $\gamma = 3/4$, which has higher cost but smaller radius.

► 8.4.3 Minimization of Source-to-Sink Delay

The previous subsections described algorithms that seek radius-cost tradeoffs. Since the spanning tree radius reflects actual wire delay, these algorithms indirectly minimize sink delays. However, the wirelength-delay abstraction, as well as the parameters ε in BRBC and γ in the PD tradeoff, prevent direct control of delay. Instead, given a set of sinks S, the *Elmore routing tree* (*ERT*) *algorithm* [8.6] iteratively adds sink s_j and edge $e(s_i,s_j)$ to the growing tree T such that the Elmore delay from the source s_0 to the sink s_j, where $s_i \in T$ and $s_j \in S - T$, is minimized.

Since the ERT algorithm does not treat any particular sink differently from the others, it is classified as a *net-dependent* approach. However, during the actual design and timing optimization, different timing constraints and slacks are imposed for each sink of a multi-pin net.

The sink with the least timing slack is the *critical sink* of the net. A routing tree construction that is oblivious to critical-sink information may create avoidable negative slack, and degrade the overall timing performance of the design. Thus, several routing tree constructions, i.e., *path-dependent approaches*, have been developed that address the *critical-sink routing problem*.

Critical-sink routing tree (CSRT) problem. Given a signal net *net* with source s_0, sinks $S = \{s_1, \ldots, s_n\}$, and sink criticalities $\alpha(i) \geq 0$ for each $s_i \in S$, construct a routing tree T such that

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

is minimized, where $t(s_0,s_i)$ is the signal delay from source s_0 to sink s_i. The sink criticality $\alpha(i)$ reflects the timing criticality of the corresponding sink s_i. If a sink is on a critical path, then its timing criticality will be greater than that of other sinks.

A *critical-sink Steiner tree* heuristic [8.18] for the CSRT problem [8.6] first constructs a heuristic minimum-cost Steiner tree T_0 over all terminals of S *except* the *critical sink* s_c, the sink with the highest criticality. Then, to reduce $t(s_0,s_c)$, the heuristic adds s_c into T_0 by heuristic variants, e.g., such as the following approaches.

- H_0: introduce a single wire from s_c to s_0.
- H_1: introduce the shortest possible wire that can join s_c to T_0, so long as the path from s_0 to s_c is *monotone*, i.e., of shortest possible total length.
- H_{Best}: try all shortest connections from s_c to edges in T_0, as well as from s_c to s_0. Perform timing analysis on each of these trees and return the one with the lowest delay at s_c.

The time complexity of the critical-sink Steiner heuristic is dominated by the construction of T_0, or by the timing analysis in the H_{Best} variant. Though H_{Best} achieves the best routing solution in terms of timing slack, the other two variants may also provide acceptable combinations of runtime efficiency and solution quality. For high-performance designs, even more comprehensively timing-driven routing tree constructions are needed. Available slack along each source-sink timing arc is best reflected by the *required arrival time* (*RAT*) at each sink. In the following *RAT tree problem* formulation, each sink of the signal net has a required arrival time which should not be exceeded by the source-sink delay in the routing tree.

RAT tree problem. For a signal net with source s_0 and sink set S, find a minimum-cost routing tree T such that

$$\min_{s \in S} \big(RAT(s) - t(s_0, s) \big) \geq 0$$

Here, $RAT(s)$ is the required arrival time for sink s, and $t(s_0,s)$ is the signal delay in T from source s_0 to sink s. Effective algorithms to solve the RAT tree problem can be found in [8.19]. More information on timing-driven routing can be found in [8.3].

8.5 8.5 Physical Synthesis

Recall from Sec. 8.2 that the correct operation of a chip with respect to setup constraints requires that $AAT \leq RAT$ at all nodes. If any nodes violate this condition, i.e., exhibit negative slack, then *physical synthesis*, a collection of timing optimizations, is applied until all slacks are non-negative. There are two aspects to the optimization – *timing budgeting* and *timing correction*. During timing budgeting, target delays are allocated to arcs along timing paths to promote timing closure during the placement and routing stages (Secs. 8.2.2 and 8.3.1), as well as during timing correction (Secs. 8.5.1-8.5.3). During timing correction, the netlist is modified to meet timing constraints using such operations as changing the size of gates, inserting buffers, and netlist restructuring. In practice, a critical path of minimum-slack nodes between two sequential elements is identified, and timing optimizations are applied to improve slack without changing the logical function.

► 8.5.1 Gate Sizing

In the standard-cell methodology, each logic gate, e.g., NAND or NOR, is typically available in multiple *sizes* that correspond to different drive strengths. The drive strength is the amount of current that the gate can provide during switching.

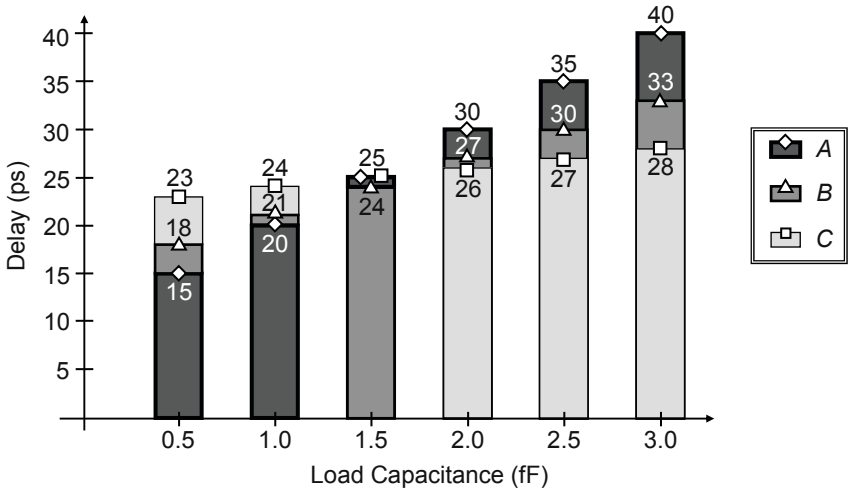


Fig. 8.12 Gate delay vs. load capacitance for three gate sizes *A*, *B*, and *C*, in increasing order.

Fig. 8.12 graphs load capacitance versus delay for versions *A*, *B* and *C* of a gate *v*, with different sizes (drive strengths), where

$$size(v_A) < size(v_B) < size(v_C)$$

A gate with larger size has lower output resistance and can drive a larger load capacitance with smaller *load-dependent delay*. However, a gate with larger size also has a larger *intrinsic delay* due to the parasitic output capacitance of the gate itself. Thus, when the load capacitance is large,

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

because the load-dependent delay dominates. When the load capacitance is small,

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

because the intrinsic delay dominates. Increasing *size*(v) also increases the gate capacitance of v, which, in turn, increases the load capacitance seen by fanin drivers. Although this relationship is not shown, the effects of gate capacitance on the delays of fanin gates will be considered below.

Resizing transformations adjust the size of v to achieve a lower delay (Fig. 8.13). Let $C(p)$ denote the load capacitance of pin p. In Fig. 8.13 (top), the total load capacitance drive by gate v is $C(d) + C(e) + C(f) = 3$ fF. Using gate size A (Fig. 8.13, lower left), the gate delay will be $t(v_A) = 40$ ps, assuming the load-delay relations in Fig. 8.12. However, using gate size C (Fig. 8.13, lower right), the gate delay is $t(v_C) = 28$ ps. Thus, for a load capacitance value of 3 fF, gate delay is improved by 12 ps if v_C is used instead of v_A. Recall that v_C has larger input capacitance at pins a and b, which increases delays of fanin gates. Details of resizing strategies can be found in [8.34]. More information on gate sizing can be found in [8.33].

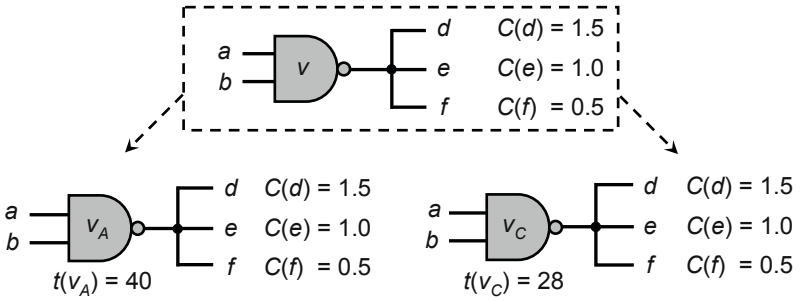


Fig. 8.13 Resizing gate v from gate size A to size C (Fig. 8.12) can achieve a lower gate delay.

► 8.5.2 Buffering

A *buffer* is a gate, typically two serially-connected inverters, that regenerates a signal without changing functionality. Buffers can (1) improve timing delays either by speeding up the circuit or by serving as delay elements, and (2) modify transition times to improve signal integrity and coupling-induced delay variation.

In Fig. 8.14 (left), the (actual) arrival time at fanout pins *d*-*h* for gate v_B is $t(v_B)$ = 45 ps. Let pins *d* and *e* be on the critical path with required arrival times below 35 ps, and let the input pin capacitance of buffer *y* be 1 fF. Then, adding *y* reduces the load capacitance of v_B from 5 to 3, and reduces the arrival times at *d* and *e* to $t(v_B)$ = 33 ps. That is, the delay of gate v_B is improved by using *y* to shield v_B from some portion of its initial load capacitance. In Fig. 8.14 (right), after *y* is inserted, the arrival time at pins *f, g* and *h* becomes $t(v_B) + t(y) = 33 + 33 = 66$ ps.

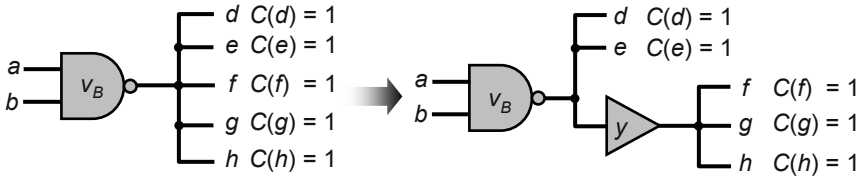


Fig. 8.14 Improving $t(v_B)$ by inserting buffer *y* to partially shield v_B's load capacitance.

A major drawback of buffering techniques is that they consume the available *area* and increase *power consumption*. Despite the judicious use of buffering by modern tools, the number of buffers has been steadily increasing in large designs due to technology scaling trends, where interconnect is becoming relatively slower compared to gates. In modern high-performance designs, buffers can comprise 10-20% of all standard cell instances, and up to 44% in some designs [8.31].

► 8.5.3 Netlist Restructuring

Often, the netlist itself can be modified to improve timing. Such changes should not alter the functionality of the circuit, but can use additional gates or modify (rewire) the connections between existing gates to improve driving strength and signal integrity. This section discusses common netlist modifications. More advanced methods for restructuring can be found in [8.25].

Cloning (Replication). Duplicating gates can reduce delay in two situations – (1) when a gate with significant fanout may be slow due to its fanout capacitance, and (2) when a gate's output fans out in two different directions, making it impossible to find a good placement for this gate. The effect on cloning (replication) is to split the driven capacitance between two equivalent gates, at the cost of increasing the fanout of upstream gates.

In Fig. 8.15 (left), using the same load-delay relations of Fig. 8.12, the gate delay $t(v_B)$ of gate v_B is 45 ps. However, In Fig. 8.15 (right), after cloning, $t(v_A)$ = 30 ps and $t(v_B)$ = 33 ps. Cloning also increases the input pin capacitance seen by the fanin gates that generate signals *a* and *b*. In general, cloning allows more freedom for local placement, e.g., the instance v_A can be placed close to sinks *d* and *e*, while the instance v_B can be placed close to sinks *f*, *g* and *h*, with the tradeoff of increased congestion and routing cost.

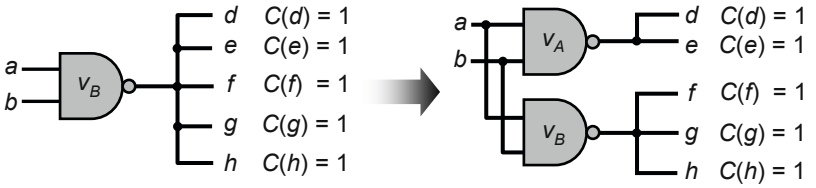


Fig. 8.15 Cloning or duplicating gates to reduce maximum local fanout.

When the downstream capacitance is large, buffering may be a better alternative than cloning because buffers do not increase the fanout capacitance of upstream gates. However, buffering cannot replace placement-driven cloning. An exercise at the end of this chapter expands further upon this concept.

The second application of cloning allows the designers to replicate gates and place each clone closer to its downstream logic. In Fig. 8.16, *v* drives five signals *d*-*h*, where signals *d*, *e* and *f* are close, and *g* and *h* are located much farther away. To mitigate the large fanout of *v* and the large interconnect delay caused by remote signals, gate *v* is cloned. The original gate *v* remains with only signals *d*, *e*, and *f*, and a new copy of *v* (*v'*) is placed closer to *g* and *h*.

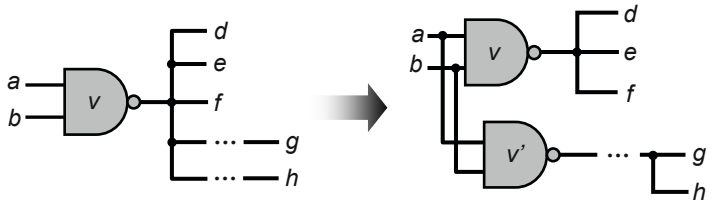


Fig. 8.16 Cloning transformation: a driving gate is duplicated to reduce remoteness of its fanouts.

Redesign of fanin tree. The logic design phase often provides a circuit with the minimum number of *logic levels*. Minimizing the maximum number of gates on a path between sequential elements tends to produce a *balanced* circuit with similar path delays from inputs to outputs. However, input signals may arrive at varied times, so the minimum-level circuit may not be timing-optimal. In Fig. 8.17, the arrival time $AAT(f)$ of pin *f* is 6 no matter how the input signals are mapped to gate input pins. However, the unbalanced network has a shorter input-output path which can be used by a later-arriving signal, where $AAT(f) = 5$.

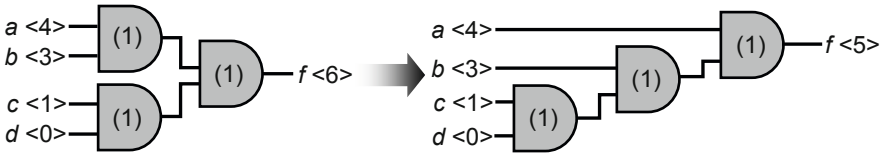


Fig. 8.17 Redesigning a fanin tree to have smaller input-to-output delay. The arrival times are denoted in angular brackets, and the delay are denoted in parentheses.

Redesign of fanout tree. In the same spirit as Fig. 8.17, it is possible to improve timing by rebalancing the output load capacitance in a fanout tree so as to reduce the delay of the longest path. In Fig. 8.18, buffer y_1 is needed because the load capacitance of critical path $path_1$ is large. However, by redesigning the fanout tree to reduce the load capacitance of $path_1$, use of the buffer y_1 can be avoided. Increased delay on $path_2$ may be acceptable if that path is not critical even after the load capacitance of buffer y_2 is increased.

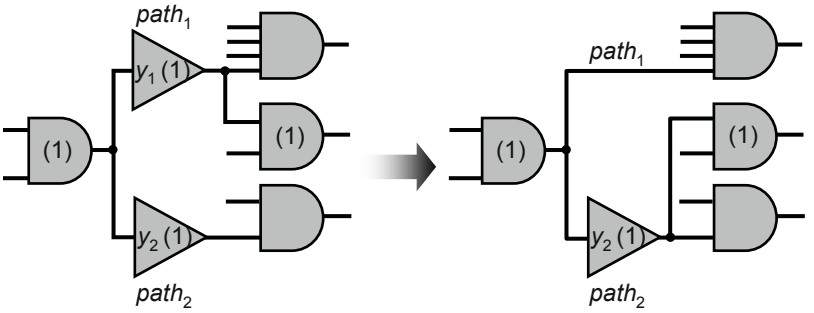


Fig. 8.18 Redesign of a fanout tree to reduce the load capacitance of $path_1$.

Swapping commutative pins. Although the input pins of, e.g., a two-input NAND gate are logically equivalent, in the actual transistor network they will have different delays to the output pin. When the *pin node convention* is used for STA (Sec. 8.2.1), the internal input-output arcs will have different delays. Hence, path delays can change when the input pin assignment is changed. The rule of thumb for pin assignment is to assign a later- (sooner-) arriving signal to an equivalent input pin with shorter (longer) input-output delay.

In Fig. 8.19, the internal timing arcs are labeled with corresponding delays in parentheses, and pins *a*, *b*, *c* and *f* are labeled with corresponding arrival times in angular brackets. In the circuit on the left, the arrival time at *f* can be improved from 5 to 3 by swapping pins *a* and *c*.

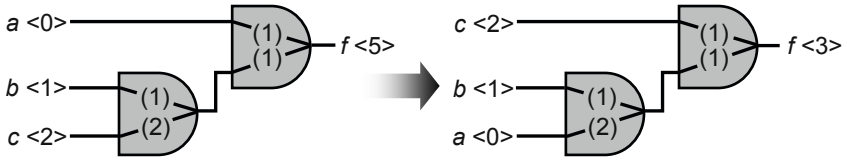


Fig. 8.19 Swapping commutative pins to reduce the arrival time at *f*.

More advanced techniques for pin assignment and swapping of commutative pins can be found in [8.9].

Gate decomposition. In CMOS designs, a gate with multiple inputs usually has larger size and capacitance, as well as a more complex transistor-level network topology that is less efficient with respect to speed metrics such as *logical effort* [8.32]. Decomposition of multiple-input gates into smaller, more efficient gates can decrease delay and capacitance while retaining the same Boolean functionality. Fig. 8.20 illustrates the decomposition of a multiple-input gate into equivalent networks of two- and three-input gates.

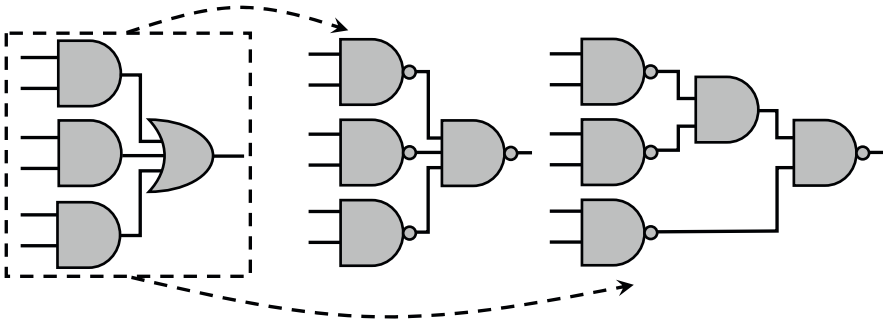


Fig. 8.20 Gate decomposition of a complex network into alternative networks.

Boolean restructuring. In digital circuits, Boolean logic can be implemented in multiple ways. In the example of Fig. 8.21, $f(a,b,c) = (a + b)(a + c) \equiv a + bc$ (distributive law) can be exploited to improve timing when two functions have overlapping logic or share logic nodes. The figure shows two functions $x = a + bc$ and $y = ab + c$ with arrival times $AAT(a) = 4$, $AAT(b) = 1$, and $AAT(c) = 2$. When implemented using a common node $a + c$, the arrival times of x and y are $AAT(x) = AAT(y) = 6$. However, implementing x and y separately achieves $AAT(x) = 5$ and $AAT(y) = 6$.

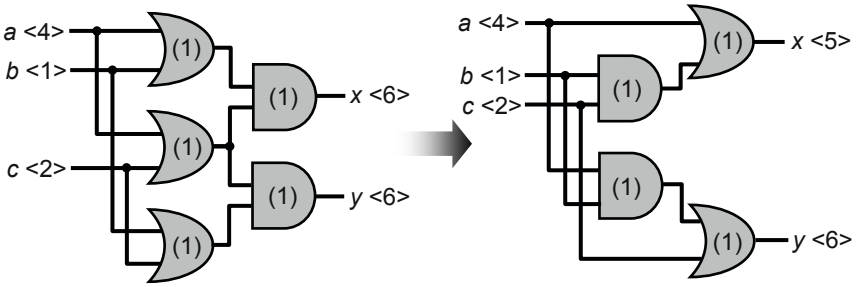


Fig. 8.21 Restructuring using logic properties, e.g., the distributive law, to improve timing.

Reverse transformations. Timing optimizations such as buffering, sizing, and cloning increase the original area of the design. This change can cause the design to be *illegal*, as some new cells can now overlap with others. To maintain legality, either (1) perform the respective reverse operations *unbuffering*, *downsizing*, and *merging*, or (2) perform *placement legalization* after all timing corrections.

8.6 8.6 Performance-Driven Design Flow

The previous sections have presented several algorithms and techniques to improve timing of digital circuit designs. This section combines all these optimizations in a consistent *performance-driven physical design flow*, which seeks to satisfy timing constraints, i.e., "close on timing". Due to the nature of performance optimizations, their ordering is important, and their interactions with conventional layout techniques are subject to a number of subtle limitations. Evaluation steps, particularly STA, must be invoked several times, and some optimizations, such as buffering, must be redone multiple times to facilitate a more accurate evaluation.

Baseline physical design flow. Recall that a typical design flow starts with *chip planning* (Chap. 3), which includes *I/O placement*, *floorplanning* (Fig. 8.22), and *power planning*. *Trial synthesis* provides the floorplanner with an estimate of the total area needed by modules. Besides logic area, additional whitespace must be allocated to account for buffers, routability, and gate sizing.

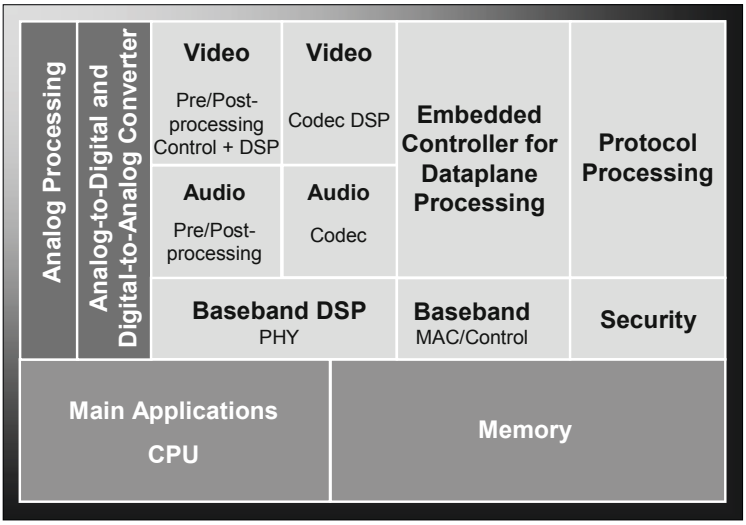


Fig. 8.22 A floorplan of a system-on-chip (SoC) design. Each major component is given dimensions based on area estimates. The audio and video components are adjacent to each other, given that their connections to other blocks and their performance constraints are similar.

Then, *logic synthesis* and *technology mapping* produce a gate- (cell)-level netlist from a high-level specification, which is tailored to a specific technology library. Next, *global placement* assigns locations to each movable object (Chap. 4). As illustrated in Fig. 8.23, most of the cells are clustered in highly concentrated regions (colored black). As the iterations progress, the cells are gradually spread across the chip, such that they no longer overlap (colored light gray).

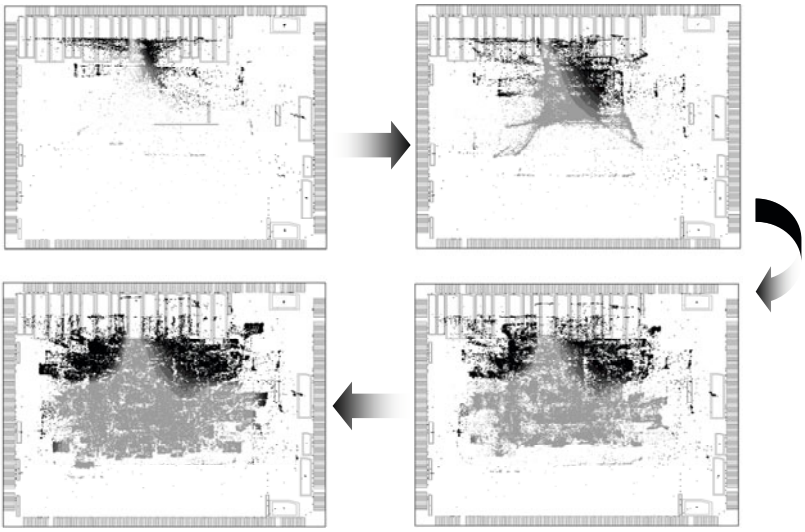


Fig. 8.23 The progression of cell spreading during global placement in a large, flat (non-floorplanned) ASIC design with fixed macro blocks. Darker shades indicate greater cell overlap while lighter shades indicate smaller cell overlap.

These locations, however, do not have to be aligned with cell rows or sites, and can allow slight cell overlap. To ensure that the overlap is small, it is common to (1) establish a uniform grid, (2) compute the total area of objects in each grid square, and (3) limit this total by the area available in the square.

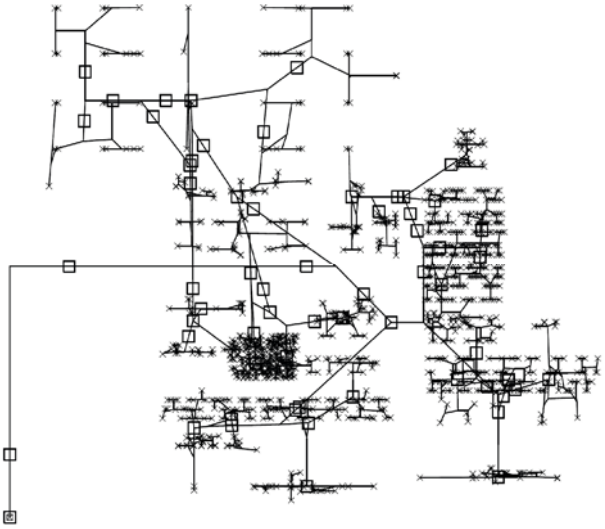


Fig. 8.24 Buffered clock tree in a small CPU design. The clock source is in the lower left corner. Crosses (×) indicate sinks, and boxes (□) indicate buffers. Each diagonal segment represents a horizontal plus a vertical wire (*L*-shape), the choice of which can be based on routing congestion.

After global placement, the sequential elements are legalized. Once the locations of sequential elements are known, a *clock network* (Chap. 7) is generated. ASICs, SoCs and low-power (mobile) CPUs commonly use clock trees (Fig. 8.24), while high-performance microprocessors incorporate structured and hand-optimized clock distribution networks that may combine trees and meshes [8.30][8.31].

The locations of globally placed cells are first temporarily rounded to a uniform grid, and then these rounded locations are connected during *global routing* (Chap. 5) and *layer assignment*, where each route is assigned to a specific metal layer. The routes indicate areas of wiring congestion (Fig. 8.25). This information is used to guide *congestion-driven detailed placement* and *legalization* of combinational elements (Chap. 4).

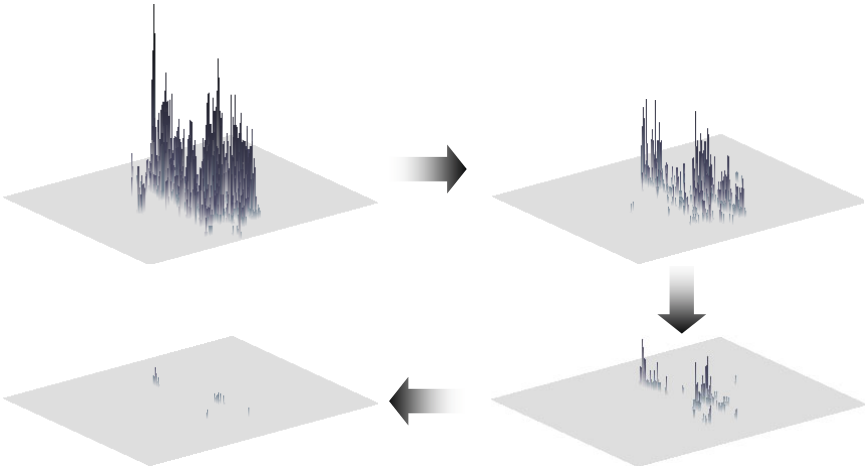


Fig. 8.25 Progression of congestion maps through iterations of global routing. The light-colored areas are those that do not have congestion; dark-colored peaks indicate congested regions. Initially, several dense clusters of wires create edges that are far over capacity. After iterations of rip-up and reroute, the route topologies are changed, alleviating the most congested areas. Though more regions can become congested, the maximum congestion is reduced.

While detailed placement conventionally comes before global routing, the reverse order can reduce overall congestion and wirelength [8.28]. Note that EDA flows require a legal placement before global routing. In this case, legalization will be performed after global placement. The global routes of signal nets are then assigned to physical routing tracks during *detailed routing* (Chap. 6).

The layout generated during the place-and-route stage is subjected to *reliability*, *manufacturability* and *electrical verification*. During *mask generation*, each standard cell and each route are represented by collections of rectangles in a format suitable for generating optical lithography masks for chip fabrication.

This baseline PD flow is illustrated in Fig. 8.26 with white boxes.

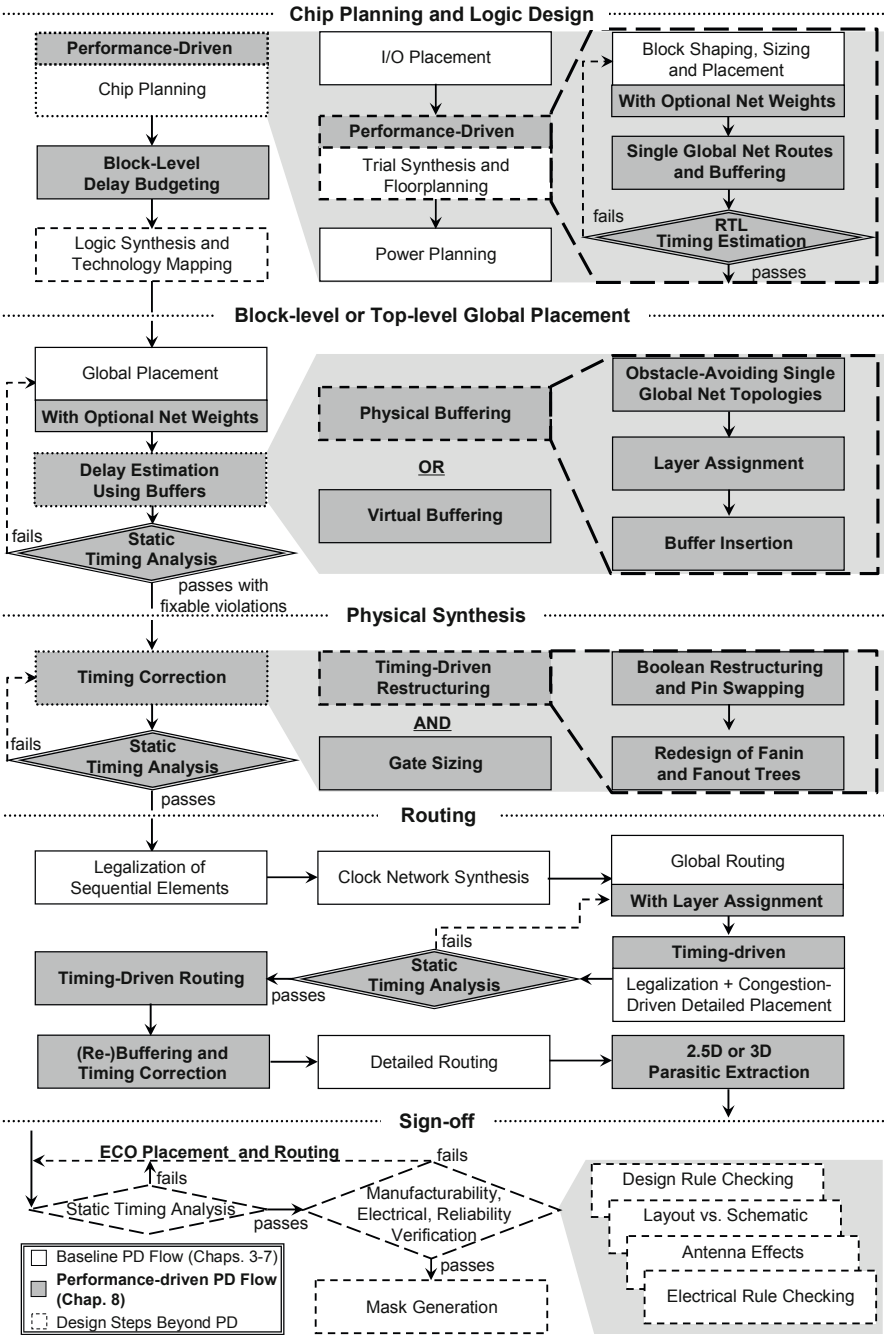


Fig. 8.26 Integrating optimizations covered in Chaps. 3-8 into a performance-driven design flow. Some tools bundle several optimization steps, which changes the appearance of the flow to users and often alters the user interface. Alternatives to this flow are discussed in this section.

Performance-driven physical design flow. Extending the baseline design flow, contemporary industrial flows are typically built around static timing analysis and seek to minimize the amount of change required to close on timing. Some flows start timing-driven optimizations as early as the chip planning stage, while others do not account for timing until detailed placement to ensure accuracy of timing results. This section discusses the timing-driven flow illustrated in Fig. 8.26 with gray boxes. Advanced methods for physical synthesis are found in [8.4].

Chip planning and logic design. Starting with a high-level design, performance-driven chip planning generates the I/O placement of the pins and rectangular blocks for each circuit module while accounting for block-level timing, and the power supply network. Then, logic synthesis and technology mapping produces a netlist based on delay budgets.

Performance-driven chip planning. Once the locations and shapes of the blocks are determined, global routes are generated for each top-level net, and buffers are inserted to better estimate timing [8.2]. Since chip planning occurs before global placement or global routing, there is no detailed knowledge of where the logic cells will be placed within each block or how they will be connected. Therefore, buffer insertion makes optimistic assumptions.

After buffering, STA checks the design for timing errors. If there are a sufficient number of violations, then the logic blocks must be re-floorplanned. In practice, modifications to existing floorplans to meet timing are performed by experienced designers with little to no automation. Once the design has satisfied or mostly met timing constraints, the I/O pins can be placed, and power (VDD) and ground (GND) supply rails can be routed around floorplan blocks.

Timing budgeting. After performance-driven floorplanning, delay budgeting sets upper bounds on *setup* (long path) timing for each block. These constraints guide logic synthesis and technology mapping to produce a performance-optimized gate-level netlist, using standard cells from a given library.

Block-level or top-level global placement. Starting at global placement, timing-driven optimizations can be performed at the *block level*, where each individual block is optimized, or *top level*, where transformations are global, i.e., cross block boundaries, and all movable objects are optimized.[4] Block-level approaches are useful for designs that have many macro blocks or *intellectual properties* (*IPs*) that have already been optimized and have specific shapes and sizes. Top-level approaches are useful for designs that have more freedom or do not reuse previously-designed logic; a hierarchical methodology offers more parallelism and is more common for large design teams.

[4] In hierarchical design flows, different designers concurrently perform top-level placement and block-level placement.

Buffer insertion. To better estimate and improve timing, buffers are inserted to break any extremely long or high fanout nets (Sec. 8.5.2). This can be done either *physically*, where buffers are directly added to the placement, or *virtually*, where the impact of buffering is included in delay models, but the netlist is not modified.

Physical buffering. Physical buffering [8.1] performs the full process of buffer insertion by (1) generating obstacle-avoiding global net topologies for each net, (2) estimating which metal layers the route uses, and (3) actually inserting buffers (Fig. 8.27).

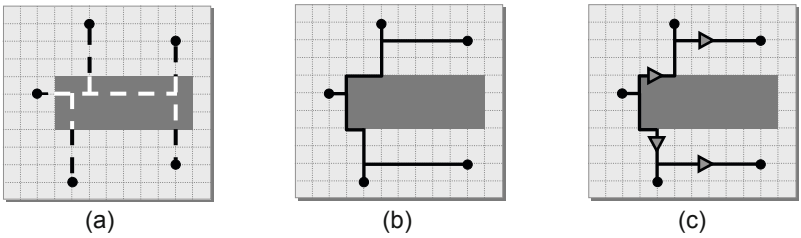


Fig. 8.27 Physical buffering for timing estimation. (a) A five-pin net is routed with a minimum Steiner tree topology that does not avoid a routing obstacle (shown in gray). (b) The net routed with an obstacle-avoiding Steiner tree topology. (c) The buffered topology offers a relatively accurate delay estimation.

Virtual buffering [8.24], on the other hand, estimates the delay by modeling every pin-to-pin connection as an optimally buffered line with linear delay [8.23] as

$$t_{LD}(net) = L(net) \cdot \left(R(B) \cdot C(w) + R(w) \cdot C(B) + \sqrt{2 \cdot R(B) \cdot C(B) \cdot R(w) \cdot C(w)} \right)$$

where *net* is the net, $L(net)$ is the total length of *net*, $R(B)$ and $C(B)$ are the respective intrinsic resistance and capacitance of the buffer, and $R(w)$ and $C(w)$ are the respective unit wire resistance and capacitance. Though no buffers are added to the netlist, they are assumed for timing purposes. When timing information becomes more accurate, subsequent re-buffering steps often remove any existing buffers and re-insert them from scratch. In this context, virtual buffering saves effort, while preserving the accuracy of timing analysis. Physical buffering can avoid unnecessary upsizing of drivers and is more accurate than virtual buffering, but also more time-consuming.

Once buffering is complete, the design is checked for timing violations using static timing analysis (Sec. 8.2.1). Unless timing is met, the design returns to buffering, global placement, or, in some cases, to logic synthesis. When timing constraints are mostly met, the design moves on to timing correction, which includes gate sizing (Sec. 8.5.1) and timing-driven netlist restructuring (Sec. 8.5.3). Subsequently, another timing check is performed using STA.

Physical synthesis. After buffer insertion, physical synthesis applies several timing correction techniques (Sec. 8.5) such as operations that modify the pin ordering or the netlist at the gate level, to improve delay on critical paths.

Timing correction. Methods such as *gate sizing* increase (decrease) the size of a physical gate to speed up (slow down) the circuit. Other techniques such as *redesign of fanin and fanout trees*, *cloning*, and *pin swapping* reduce timing by rebalancing existing logic to reduce load capacitance for timing-critical nets. Transformations such as *gate decomposition* and *Boolean restructuring* modify logic locally to improve timing by merging or splitting logic nodes from different signals. After physical synthesis, another timing check is performed. If it fails, another pass of timing correction attempts to fix timing violations.

Routing. After physical synthesis, all combinational and sequential elements in the design are connected during global and clock routing, respectively. First, the sequential elements of the design, e.g., flip-flop and latches, are legalized (Sec. 4.4). Then, clock network synthesis generates the clock tree or mesh to connect all sequential elements to the clock source. Modern clock networks require a number of large clock buffers;[5] performing clock-network design *before* detailed placement allows these buffers to be placed appropriately. Given the clock network, the design can be checked for *hold-time (short path) constraints*, since the clock skews are now known, whereas only *setup (long path) constraints* could be checked before.

Layer assignment. After clock-network synthesis, global routing assigns global route topologies to connect the combinational elements. Then, layer assignment matches each global route to a specific metal layer. This step improves the accuracy of delay estimation because it allows the use of appropriate *resistance-capacitance* (*RC*) parasitics for each net. Note that clock routing is performed before signal-net routing when the two share the same metal layers – clock routes take precedence and should not detour around signal nets.

Timing-driven detailed placement. The results of global routing and layer assignment provide accurate estimates of wire congestion, which is then used by a congestion-driven detailed placer [8.10][8.35]. The cells are (1) spread to remove overlap among objects and decrease routing congestion, (2) snapped to standard-cell rows and legal cell sites, and then (3) optimized by swaps, shifts and other local changes. To incorporate timing optimizations, *either* perform (1) non-timing-driven legalization followed by timing-driven detailed placement, *or* (2) perform timing-driven legalization followed by non-timing-driven detailed placement. After detailed placement, another timing check is performed. If timing fails, the design could be globally re-routed or, in severe cases, globally re-placed.

To give higher priority to the clock network, the sequential elements can be legalized first, and then followed by global and detailed routing. With this approach,

[5] These buffers are legalized immediately when added to the clock network.

signal nets must route around the clock network. This is advantageous for large-scale designs, as clock trees are increasingly becoming a performance bottleneck. A variant flow, such as the industrial flow described in [8.28], first fully legalizes the locations of all cells, and then performs detailed placement to recover wirelength.

Another variant performs detailed placement before clock network synthesis, and then is followed by legalization and several optimization steps.[6] After the clock network has been synthesized, another pass of setup optimization is performed. Hold violations may be addressed at this time or, optionally, after routing and initial STA.

Timing-driven routing. After detailed placement, clock network synthesis and post-clock network optimization, the *timing-driven routing* phase aims to fix the remaining timing violations. Algorithms discussed in Sec. 8.4 include generating *minimum-cost, minimum-radius trees* for critical nets (Secs. 8.4.1-8.4.2), and *minimizing the source-to-sink delay* of critical sinks (Sec. 8.4.3).

If there are still outstanding timing violations, further optimizations such as re-buffering and late timing corrections are applied. An alternative is to have designers manually tune or fix the design by relaxing some design constraints, using additional logic libraries, or exploiting design structure neglected by automated tools. After this time-consuming process, another timing check is performed. If timing is met, then the design is sent to detailed routing, where each signal net is assigned to specific routing tracks. Typically, incremental STA-driven *Engineering Change Orders* (*ECOs*) are applied to fix timing violations after detailed placement; this is followed by ECO placement and routing. Then, *2.5D* or *3D parasitic extraction* determines the electromagnetic impact on timing based on the routes' shapes and lengths, and other technology-dependent parameters.

Signoff. The last few steps of the design flow validate the layout and timing, as well as fix any outstanding errors. If a timing check fails, ECO minimally modifies the placement and routing such that the violation is fixed and no new errors are introduced. Since the changes made are very local, the algorithms for ECO placement and ECO routing differ from the traditional place and route techniques discussed in Chaps. 4-7.

After completing timing closure, *manufacturability*, *reliability* and *electrical verification* ensure that the design can be successfully fabricated and will function correctly under various environmental conditions. The four main components are equally important and can be performed in parallel to improve runtime.

– *Design Rule Checking* (*DRC*) ensures that the placed-and-routed layout meets all technology-specified design rules e.g., minimum wire spacing and width.

[6] These include post-clock-network-synthesis optimizations, post-global-routing optimizations, and post-detailed-routing optimizations.

- *Layout vs. Schematic* (*LVS*) checking ensures the placed-and-routed layout matches the original netlist.
- *Antenna Checks* seek to detect undesirable *antenna effects*, which may damage a transistor during plasma-etching steps of manufacturing by collecting excess charge on metal wires that are connected to PN-junction nodes. This can occur when a route consists of multiple metal layers and a charge is induced on a metal layer during fabrication.
- *Electric Rule Checking* (*ERC*) finds all potentially dangerous electric connections, such as floating inputs and shorted outputs.

Once the design has been physically verified, *optical-lithography masks* are generated for manufacturing.

8.7 8.7 Conclusions

This chapter explained how to combine timing optimizations into a comprehensive physical design flow. In practice, the flow described in Sec. 8.6 (Fig. 8.26) can be modified based on several factors, including

- Design type.
  - ASIC, microprocessor, IP, analog, mixed-mode.
  - Datapath-heavy specifications may require specialized tools for structured placement or manual placement. Datapaths typically have shorter wires and require fewer buffers for high-performance layout.
- Design objectives.
  - High-performance, low-power or low-cost.
  - Some high-performance optimizations, such as buffering and gate sizing, increase circuit area, thus increasing circuit power and chip cost.
- Additional optimizations.
  - *Retiming* shifts locations of registers among combinational gates to better balance delay.
  - *Useful skew* scheduling, where the clock signal arrives earlier at some flip-flops and later at others, to adjust timing.
  - *Adaptive body-biasing* can improve the leakage current of transistors.
- Additional analyses.
  - Multi-corner and multi-mode static timing analysis, as industrial ASICs and microprocessors are often optimized to operate under different temperatures and supply voltages.
  - Thermal analysis is required for high-performance CPUs.
- Technology node, typically specified by the *minimum feature size*.
  - Nodes < 180 nm require timing-driven placement and routing flows, as lumped-capacitance models are inadequate for performance estimation.

  - Nodes < 130 nm require timing analysis with signal integrity, i.e., interconnect coupling capacitances and the resulting delay increase (decrease) of a given *victim net* when a neighboring *aggressor* net switches simultaneously in the opposite (same) direction.
  - Nodes < 90 nm require additional *resolution enhancement techniques* (*RET*) for lithography.
  - Nodes < 65 nm require power-integrity (e.g., IR drop-aware timing, electromigration reliability) analysis flows.
  - Nodes < 45 nm require additional statistical power-performance tradeoffs at the transistor level.
  - Nodes < 32 nm impose significant limitations on detailed routing, known as *restricted design rules* (*RDRs*), to ensure manufacturability.
- Available tools.
  - In-house software, commercial EDA tools [8.34].
- Design size and the extent of design reuse.
  - Larger designs often include more global interconnect, which may become a performance bottleneck and typically requires buffering.
  - IP blocks are typically represented by hard blocks during floorplanning.
- Design team size, required time-to-market, available computing resources.
  - To shorten time-to-market, one can leverage a large design team by partitioning the design into blocks and assigning blocks to teams.
  - After floorplanning, each block can be laid out in parallel; however, *flat* optimization (no partitioning) sometimes produces better results.

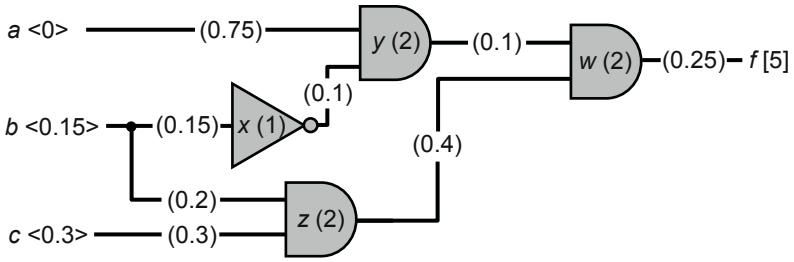
Reconfigurable fabrics such as FPGAs require less attention to buffering, due to already-buffered programmable interconnect. Wire congestion is often negligible for FPGAs because interconnect resources are overprovisioned. However, FPGA detailed placement must satisfy a greater number of constraints than placement for other circuit types, and global routing must select from a greater variety of interconnect types. Electrical and manufacturability checks are unnecessary for FPGAs, but technology mapping is more challenging, as it affects the area and timing to a greater extent, and can benefit more from the use of physical information. Therefore, modern physical-synthesis flows for FPGAs perform global placement, often in a trial mode, between logic synthesis and technology mapping.

Physical design flows will require additional sophistication to support increasing transistor densities in semiconductor chips. The advent of future technology nodes – 28 nm, 22 nm and 16 nm – will bring into consideration new electrical and manufacturing-related phenomena, while increasing uncertainty in device parameters [8.22]. Further increase in transistor counts may require integrating multiple chips into three-dimensional integrated circuits, thus changing the geometry of fundamental physical design optimizations [8.36]. Nevertheless, the core optimizations described in this chapter will remain vital in chip design.

Chapter 8 Exercises

Exercise 1: Static Timing Analysis

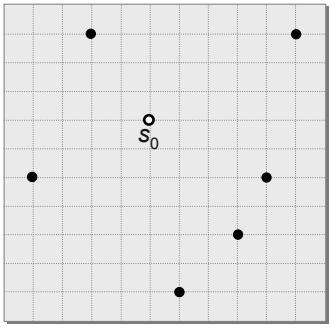
Given the logic circuit below, draw the timing graph (a), and determine the (b) AAT, (c) RAT, and (d) slack of each node. The AATs of the inputs are in angular brackets, the delays are in parentheses, and the RAT of the output is in square brackets.



Exercise 2: Timing-Driven Routing

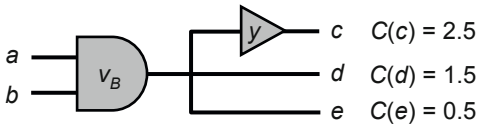
Given the terminal locations of a signal net, assume that all distances are Manhattan, and that no Steiner point is used when routing. Construct the spanning tree T, and calculate $radius(T)$ and $cost(T)$, for each of the following.

(a) Prim-Dijkstra tradeoff with $\gamma = 0$ (Prim's MST algorithm).
(b) Prim-Dijkstra tradeoff with $\gamma = 1$ (Dijkstra's algorithm).
(c) Prim-Dijkstra tradeoff with $\gamma = 0.5$.



Exercise 3: Buffer Insertion for Timing Improvement

For the logic circuit and load capacitances on the following page, assume that available gate sizes and timing performances are similar to those in Fig. 8.12. Assume that gate delay always increases linearly with load capacitance. Let the input capacitance of buffer y be 0.5 fF, 1 fF, and 2 fF with sizes A, B and C, respectively. Determine the size for buffer y that minimizes the AAT of sink c.



Exercise 4: Timing Optimization
List at least two timing optimizations covered *only* in this chapter (not mentioned beforehand). Describe these optimizations in your own words and discuss scenarios in which (1) they can be useful and (2) they can be harmful.

Exercise 5: Cloning vs. Buffering
List and explain scenarios where cloning results in better timing improvements than buffering, and vice-versa. Explain why both methods are necessary for timing-driven physical synthesis.

Exercise 6: Physical Synthesis
In terms of timing corrections such as buffering, gate sizing, and cloning, when are their reverse transformations useful? In what situations will a given timing correction cause the design to be illegal? Explain for each timing correction.

Chapter 8 References

[8.1] C. J. Alpert, M. Hrkic, J. Hu and S. T. Quay, "Fast and Flexible Buffer Trees that Navigate the Physical Layout Environment", *Proc. Design Autom. Conf.*, 2004, pp. 24-29.

[8.2] C. J. Alpert, J. Hu, S. S. Sapatnekar and C. N. Sze, "Accurate Estimation of Global Buffer Delay Within a Floorplan", *IEEE Trans. on CAD* 25(6) (2006), pp. 1140-1146.

[8.3] C. J. Alpert, T. C. Hu, J. H. Huang and A. B. Kahng, "A Direct Combination of the Prim and Dijkstra Constructions for Improved Performance-Driven Global Routing", *Proc. Intl. Conf. on Circuits and Sys.*, 1993, pp. 1869-1872.

[8.4] C. J. Alpert, S. K. Karandikar, Z. Li, G.-J. Nam, S. T. Quay, H. Ren, C. N. Sze, P. G. Villarrubia and M. C. Yildiz, "Techniques for Fast Physical Synthesis", *Proc. IEEE* 95(3) (2007), pp. 573-599.

[8.5] C. J. Alpert, D. P. Mehta and S. S. Sapatnekar, eds., *Handbook of Algorithms for Physical Design Automation*, CRC Press, 2009.

[8.6] K. D. Boese, A. B. Kahng and G. Robins, "High-Performance Routing Trees with Identified Critical Sinks", *Proc. Design Autom. Conf.*, 1993, pp. 182-187.

[8.7] M. Burstein and M. N. Youssef, "Timing Influenced Layout Design", *Proc. Design Autom. Conf.*, 1985, pp. 124-130.

[8.8] T. F. Chan, J. Cong and E. Radke, "A Rigorous Framework for Convergent Net Weighting Schemes in Timing-Driven Placement", *Proc. Intl Conf. on CAD*, 2009, pp. 288-294.

[8.9] K.-H. Chang, I. L. Markov and V. Bertacco, "Postplacement Rewiring by Exhaustive Search for Functional Symmetries", *ACM Trans. on Design Autom. of Electronic Sys.* 12(3) (2007), pp. 1-21.

[8.10] A. Chowdhary, K. Rajagopal, S. Venkatesan, T. Cao, V. Tiourin, Y. Parasuram and B. Halpin, "How Accurately Can We Model Timing in a Placement Engine?", *Proc. Design Autom. Conf.*, 2005, pp. 801-806.

[8.11] J. Cong, A. B. Kahng, G. Robins, M. Sarrafzadeh and C. K. Wong, "Provably Good Algorithms for Performance-Driven Global Routing", *Proc. Intl. Symp. on Circuits and Sys.*, 1992, pp. 2240-2243.

[8.12] A. E. Dunlop, V. D. Agrawal, D. N. Deutsch, M. F. Jukl, P. Kozak and M. Wiesel, "Chip Layout Optimization Using Critical Path Weighting", *Proc. Design Autom. Conf.*, 1984, pp. 133-136.

[8.13] W. C. Elmore, "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers", *J. Applied Physics* 19(1) (1948), pp. 55-63.

[8.14] H. Eisenmann and F. M. Johannes, "Generic Global Placement and Floorplanning", *Proc. Design Autom. Conf.*, 1998, pp. 269-274.

[8.15] S. Ghiasi, E. Bozorgzadeh, P.-K. Huang, R. Jafari and M. Sarrafzadeh, "A Unified Theory of Timing Budget Management", *IEEE Trans. on CAD* 25(11) (2006), pp. 2364-2375.

[8.16] B. Halpin, C. Y. R. Chen and N. Sehgal, "Timing Driven Placement Using Physical Net Constraints", *Proc. Design Autom. Conf.*, 2001, pp. 780-783.

[8.17] P. S. Hauge, R. Nair and E. J. Yoffa, "Circuit Placement for Predictable Performance", *Proc. Intl. Conf. on CAD*, 1987, pp. 88-91.

[8.18] T. I. Kirkpatrick and N. R. Clark, "PERT as an Aid to Logic Design", *IBM J. Research and Development* 10(2) (1966), pp. 135-141.

[8.19] J. Lillis, C.-K. Cheng, T.-T. Y. Lin and C.-Y. Ho, "New Performance Driven Routing Techniques With Explicit Area/Delay Tradeoff and Simultaneous Wire Sizing", *Proc. Design Autom. Conf.*, 1996, pp. 395-400.

[8.20] R. Nair, C. L. Berman, P. S. Hauge and E. J. Yoffa, "Generation of Performance Constraints for Layout", *IEEE Trans. on CAD* 8(8) (1989), pp. 860-874.

[8.21] M. Orshansky and K. Keutzer, "A General Probabilistic Framework for Worst Case Timing Analysis", *Proc. Design Autom. Conf.*, 2002, pp. 556-561.

[8.22] M. Orshansky, S. Nassif and D. Boning, *Design for Manufacturability and Statistical Design: A Constructive Approach*, Springer, 2008.

[8.23] R. H. J. M. Otten and R. K. Brayton, "Planning for Performance", *Proc. Design Autom. Conf.*, 1998, pp. 122-127.

[8.24] D. A. Papa, T. Luo, M. D. Moffitt, C. N. Sze, Z. Li, G.-J. Nam, C. J. Alpert and I. L. Markov, "RUMBLE: An Incremental, Timing-Driven, Physical-Synthesis Optimization Algorithm", *IEEE Trans. on CAD* 27(12) (2008), pp. 2156-2168.

[8.25] S. M. Plaza, I. L. Markov and V. Bertacco, "Optimizing Nonmonotonic Interconnect Using Functional Simulation and Logic Restructuring", *IEEE Trans. on CAD* 27(12) (2008), pp. 2107-2119.

[8.26] K. Rajagopal, T. Shaked, Y. Parasuram, T. Cao, A. Chowdhary and B. Halpin, "Timing Driven Force Directed Placement with Physical Net Constraints", *Proc. Intl. Symp. on Phys. Design*, 2003, pp. 60-66.

[8.27] H. Ren, D. Z. Pan and D. S. Kung, "Sensitivity Guided Net Weighting for Placement Driven Synthesis", *IEEE Trans. on CAD* 24(5) (2005) pp. 711-721.

[8.28] J. A. Roy, N. Viswanathan, G.-J. Nam, C. J. Alpert and I. L. Markov, "CRISP: Congestion Reduction by Iterated Spreading During Placement", *Proc. Intl. Conf. on CAD*, 2009, pp. 357-362.

[8.29] M. Sarrafzadeh, M. Wang and X. Yang, *Modern Placement Techniques*, Kluwer, 2003.

[8.30] R. S. Shelar, "Routing With Constraints for Post-Grid Clock Distribution in Microprocessors", *IEEE Trans. on CAD* 29(2) (2010), pp. 245-249.

[8.31] R. S. Shelar and M. Patyra, "Impact of Local Interconnects on Timing and Power in a High Performance Microprocessor", *Proc. Intl. Symp. on Phys. Design*, 2010, pp. 145-152.

[8.32] I. Sutherland, R. F. Sproull and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann, 1999.

[8.33] H. Tennakoon and C. Sechen, "Nonconvex Gate Delay Modeling and Delay Optimization", *IEEE Trans. on CAD* 27(9) (2008), pp. 1583-1594.

[8.34] M. Vujkovic, D. Wadkins, B. Swartz and C. Sechen, "Efficient Timing Closure Without Timing Driven Placement and Routing", *Proc. Design Autom. Conf.*, 2004, pp. 268-273.

[8.35] J. Westra and P. Groeneveld, "Is Probabilistic Congestion Estimation Worthwhile?", *Proc. Sys. Level Interconnect Prediction*, 2005, pp. 99-106.

[8.36] Y. Xie, J. Cong and S. Sapatnekar, eds., *Three-Dimensional Integrated Circuit Design: EDA, Design and Microarchitectures*, Springer, 2010.