An Expert Report on Advanced VLSI Synthesis and Physical Design Methodologies

The Synthesis Foundation: From RTL to Gate-Level Netlist

Logic synthesis is the foundational process in the Application-Specific Integrated Circuit (ASIC) design flow that transforms a high-level, abstract description of a circuit's behavior into a concrete, physically implementable gate-level netlist. This automated process, pioneered by Electronic Design Automation (EDA) companies like Synopsys and Cadence, bridges the gap between the designer's intent, captured in a Hardware Description Language (HDL), and the physical standard cells provided by a semiconductor foundry. The quality of the final silicon is heavily dependent on the quality of this initial transformation, which is governed by a critical triad of inputs: the design's functional description, the characteristics of the target technology, and the performance goals set by the architect.

Core Inputs for Synthesis: The Triad of Design Intent

The synthesis engine's ability to produce an optimal netlist is fundamentally constrained by the quality and consistency of its inputs. A deficiency in any one of these core components can compromise the final Quality of Results (QoR), regardless of the sophistication of the synthesis algorithms.

Register-Transfer Level (RTL) Code (.v,.vhd,.sv)

The Register-Transfer Level (RTL) code is the primary input that describes the functional behavior of the digital circuit.[1] Written in HDLs such as Verilog, VHDL, or SystemVerilog, the RTL defines the flow of data between registers and the logical operations performed on that data. The coding style adopted by the designer has a profound and direct impact on the synthesis outcome. For instance, the way an arithmetic operation is described can lead the tool to infer either a fast but large parallel-prefix adder or a slower but smaller ripple-carry adder. Similarly, poorly structured case statements or nested if-else blocks can result in complex priority-encoded logic, creating long timing paths that are difficult for the tool to optimize.[2]

Technology Libraries (.lib,.db)

Technology libraries are the cornerstone of the technology mapping phase, providing the synthesis tool with the palette of building blocks it can use to implement the design's logic.[2] These files, provided by the foundry or a library vendor, contain meticulously characterized data for every standard cell (e.g., AND, OR, NAND, Flip-Flops, Adders) in a given process technology.

- **Content Deep Dive:** The library file is far more than a simple list of functions. It contains comprehensive models that describe the behavior of each cell across various Process, Voltage, and Temperature (PVT) corners.[3] Key information includes:
    - **Timing Arcs:** Detailed lookup tables (Non-Linear Delay Models or NLDM) that define a cell's propagation delay and output transition time as a function of its input signal transition and total output capacitive load.[4]
    - **Power Information:** Characterization of both dynamic power (internal switching power and power consumed charging output capacitance) and static power (leakage current).[4]
    - **Physical Area:** The physical footprint of the cell, which is used for area estimation and optimization.[2]
    - **Functionality:** A Boolean logic description of the cell's function.[6]
    - **Design Rule Constraints:** Intrinsic limits of the cell, such as maximum input transition time and maximum output capacitance, that must not be violated for the cell to function correctly.[2]
- **Tool-Specific Formats:** While the content is standardized by the Liberty format (.lib), EDA tools often use proprietary, optimized versions. Synopsys tools, such as Design Compiler and Fusion Compiler, predominantly use a compiled binary format (.db) for faster loading and processing, while other tools, including Cadence Genus, often work directly with the ASCII .lib files.[5]

Design Constraints (SDC - Synopsys Design Constraints)

The Synopsys Design Constraints (SDC) file is the mechanism by which the designer communicates the performance, area, and power goals to the synthesis tool.[2] Without this file, the tool would only perform minimal area optimization. The SDC file guides every decision made during the logic optimization and technology mapping phases.

- **Key Timing Constraints:** These define the performance targets.
  - create_clock: Defines the clock sources, their periods, and waveforms.[2]
  - set_input_delay / set_output_delay: Specifies the timing of signals at the design's primary inputs and the timing requirements for signals at the primary outputs, accounting for external logic delays.[2]
  - set_clock_uncertainty / set_clock_latency: Models the effects of the clock network, such as skew and insertion delay, before the actual clock tree is built.[2]
- **Timing Exceptions:** These constraints are used to relax or remove timing analysis on paths that are not critical or functionally relevant.
  - set_false_path: Instructs the tool to ignore timing on paths that are structurally present but can never be logically sensitized, such as paths between asynchronous clock domains.[2]
  - set_multicycle_path: Informs the tool that a particular path has more than one clock cycle to propagate its signal, which is common in pipelined datapaths.[2]
- **Design Rule Constraints:** These enforce the physical limits of the technology.
  - set_max_transition, set_max_fanout, set_max_capacitance: These commands set global limits on signal transition times, the number of gates a net can drive, and the total capacitive load on a net, respectively, to ensure signal integrity and prevent cell performance degradation.[2]

Power Intent (UPF - Unified Power Format)

For modern low-power designs, a separate file describing the power architecture is essential. The IEEE 1801 Unified Power Format (UPF) is the industry standard for specifying power intent.[2] This file defines elements such as:

- **Power Domains:** Regions of the design that can be powered by different voltage levels or can be powered down independently.
- **Voltage Levels:** The operating voltages for each power domain.
- **Isolation Strategies:** The logic (isolation cells) required to prevent signal corruption

when a signal crosses from a powered-on domain to a powered-off domain.
- **Level Shifters:** Cells required to safely transmit signals between domains operating at different voltage levels.
- **State Retention Policies:** Defines which registers must retain their state when their power domain is shut down, guiding the tool to insert special retention flip-flops.

The synthesis tool uses the UPF to automatically insert the necessary power management cells (isolation, level shifters, retention registers) into the netlist, ensuring the design is "power-aware" from the very beginning.[1]

The quality and consistency of these inputs are deeply intertwined. A well-written RTL can be rendered un-routable or unable to meet timing if the constraints are unrealistic or the library characterization is inaccurate. Conversely, even with perfect libraries and constraints, a poorly architected RTL (e.g., one with an excessively long combinational path) presents a fundamental limitation that the synthesis tool cannot overcome through simple gate sizing or logic optimization. The final QoR is therefore not merely a product of the synthesis tool's algorithmic prowess but is fundamentally bounded by the weakest link in this input triad, underscoring the necessity for close collaboration between RTL, library, and physical design teams.

The Synthesis Workflow: A Three-Stage Process

The core synthesis process, as executed by tools like Synopsys Design Compiler or Cadence Genus, can be broken down into three distinct stages: translation, logic optimization, and technology mapping.[2]

Translation (Elaboration)

The first stage involves reading the HDL source files and translating them into an internal, technology-independent format.[2] In Synopsys tools, this representation is famously known as GTECH (Generic Technology), which consists of basic logic gates (AND, OR, XOR) and sequential elements (D-flip-flops) that have no specific technology attributes.[2] During this phase, known as elaboration, the tool performs several key tasks:

- **Syntax and Rule Checking:** The HDL code is parsed and checked for syntactical correctness and adherence to synthesis-friendly coding rules.[2]
- **Hardware Inference:** The tool infers hardware structures from the HDL constructs. For

example, an always @(posedge clk) block implies a register, while arithmetic operators (+, *) are mapped to components from a generic library like Synopsys's DesignWare.[2]
- **Design Hierarchy Elaboration:** The tool builds an in-memory representation of the design's hierarchy, instantiating sub-modules and connecting their ports.[2]

Logic Optimization

Once the design is in a generic format, the tool performs a series of technology-independent logic optimizations. The goal is to simplify the Boolean equations representing the design's combinational logic, which can lead to improvements in area, power, and timing.[2] This stage is often described as a manipulation of Boolean equations and includes two primary processes:

- **Structuring:** This process introduces intermediate logic structures and variables to break down complex, multi-level logic cones into simpler, more manageable forms. This can help reduce fanout on critical nets and create more opportunities for efficient mapping.[2]
- **Flattening:** This is the opposite of structuring. It collapses logic into a two-level sum-of-products (SOP) or product-of-sums (POS) representation. While this can sometimes expose more optimization opportunities, excessive flattening can lead to very high-fan-in gates that are not available in the target technology library.[2]

  Modern tools dynamically apply a mix of these techniques to achieve the best overall result.

Technology Mapping

This is the final and most crucial stage of synthesis, where the optimized, generic netlist is mapped to the physical standard cells available in the target technology library.[2] This is a constraint-driven process. The tool analyzes the timing paths in the generic netlist, considers the timing, power, and area characteristics of the cells in the

.lib file, and selects the specific cells (e.g., AND2X1, NAND3X4, DFFX2) that best meet the design's goals as defined in the SDC file.[2] This stage involves a complex series of sub-steps:

- **Delay Optimization:** The tool focuses on the critical paths (those with the worst timing slack) and attempts to speed them up by selecting faster, higher-drive-strength cells, restructuring logic, or inserting buffers.
- **Design Rule Fixing:** The tool ensures that no design rule constraints (max transition, max capacitance, max fanout) are violated. It will insert buffers or resize gates as needed to

fix these violations, even if it comes at the cost of timing or area, because design rules are functional requirements.[2]
- **Area Optimization:** After timing and design rules are met, the tool attempts to recover area by swapping large, fast cells on non-critical paths with smaller, slower equivalents, and by sharing common logic where possible.

Key Outputs and Initial Quality of Results (QoR) Assessment

Upon completion of the synthesis process, the tool generates several critical output files that form the input to the next stage of the design flow (physical design).

- **Gate-Level Netlist (.v,.vg):** This is the primary output. It is a structural Verilog file that describes the design as an interconnection of standard cells from the target technology library. This netlist is functionally equivalent to the input RTL but now has a physical correspondence.[1]
- **Updated SDC:** The synthesis tool may perform optimizations like clock gating or clock propagation, which can require modifications to the original constraints. The tool outputs an updated SDC file reflecting these changes for use in downstream tools.[1]
- **Reports (QoR, Area, Timing):** The tool generates a suite of reports that provide the first comprehensive assessment of the design's Quality of Results (QoR). These reports are indispensable for the design team to gauge whether the design is on a viable path to meeting its goals.[2] Key reports include:
    - **Timing Report (report_timing):** Details the timing slack on the most critical paths for each clock group.[2]
    - **Area Report (report_area):** Breaks down the total cell area by hierarchy and cell type (combinational, sequential, etc.).[2]
    - **QoR Report (report_qor):** Provides a high-level summary of the design's status, including timing violations (WNS, TNS), total area, and design rule violations.[2]
    - **Constraint Report (report_constraint):** Checks which constraints have been met and which have been violated.[2]

These initial reports are a critical checkpoint. If significant timing or design rule violations exist at this stage, it often indicates a fundamental issue with the RTL, constraints, or choice of technology library that must be addressed before proceeding to the time-consuming physical design stage.

Mastering Timing Closure: WNS and TNS Optimization

Timing closure is the iterative process of modifying a design to meet all of its timing constraints. In the context of synthesis, it is the primary optimization objective. The success of this process is measured by two key metrics: Worst Negative Slack (WNS) and Total Negative Slack (TNS). Understanding these metrics and the tools available to influence them, such as path grouping, is essential for guiding the synthesis engine to a successful outcome.

The Anatomy of a Timing Path and the Concept of Path Groups

A timing path is the fundamental unit of analysis in Static Timing Analysis (STA). Each path has a defined startpoint and endpoint.[2]

- **Startpoint:** An input port of the design or the clock pin of a sequential element (e.g., a flip-flop).
- **Endpoint:** An output port of the design or the data input pin of a sequential element.

Based on these definitions, there are four canonical types of timing paths [2]:

1. **Input-to-Register:** From a primary input port to a sequential element's data pin.
2. **Register-to-Register:** From one sequential element's clock pin to another's data pin. This is often the most numerous and critical path type in a synchronous design.
3. **Register-to-Output:** From a sequential element's clock pin to a primary output port.
4. **Input-to-Output:** A purely combinational path from a primary input port to a primary output port.

To manage the complexity of analyzing millions of such paths, synthesis tools organize them into **path groups**. By default, paths are grouped according to the clock associated with the endpoint of the path.[17] For example, all paths that end at flip-flops captured by

CLK_A belong to the CLK_A path group. This allows the tool to optimize all paths related to a specific clock domain as a coherent set. A special default path group exists for paths that are not associated with any clock, such as purely combinational input-to-output paths.[17]

Defining and Analyzing WNS and TNS

Slack is the quantitative measure of timing margin for a given path. For setup timing, it is calculated as the difference between the time the data is required to be stable at the

endpoint and the time it actually arrives.[2]

$$Slack_{setup} = T_{required} - T_{arrival}$$

A positive slack value indicates that the timing constraint is met, while a negative value signifies a timing violation. From this fundamental calculation, two critical high-level metrics are derived:

- **WNS (Worst Negative Slack):** This represents the slack of the single most timing-critical path in a given path group or across the entire design. It is the largest negative slack value (i.e., the value closest to negative infinity). WNS is the primary indicator of whether a design can meet its target frequency; the clock period cannot be reduced further until the WNS is non-negative.[18]
- **TNS (Total Negative Slack):** This is the arithmetic sum of the negative slacks of all paths that fail to meet timing. TNS provides a measure of the overall extent of timing violations. A design might have a small WNS, indicating the most critical path is only failing by a small amount, but a very large TNS, indicating that thousands of paths are failing. This distinction is crucial for diagnosis.[18]

The relationship between WNS and TNS serves as a powerful diagnostic tool. A design with a large WNS but a small TNS points to a localized "critical path" problem, likely caused by a single, deep cone of logic. The solution here is targeted, focusing on restructuring that specific path, perhaps by adding pipeline stages or using manual path grouping to apply higher optimization effort. In contrast, a design with a small WNS but a large TNS suggests a more systemic issue. This scenario often arises from overly aggressive global constraints, excessive gate density leading to congestion, or an inadequate floorplan. The solution is not to fix individual paths but to address the global problem, for example, by relaxing the clock constraint, reducing the target utilization, or refining the physical design strategy. Analyzing the magnitude and ratio of these two metrics allows engineers to correctly diagnose the root cause of timing failures and select the most effective optimization strategy.

Strategic Optimization via Path Grouping: The group_path Command

By default, a synthesis tool's optimization effort is often monopolized by the single worst violating path in the design. This can be inefficient if that path is architecturally difficult to fix, leaving many other, more easily correctable violations unaddressed. The group_path command is a powerful mechanism for designers to manually override the default grouping and redirect the tool's optimization focus.[2]

By creating custom path groups, engineers can isolate specific sets of paths—such as all paths from the inputs to a critical ALU, or all paths originating from a specific memory

interface—and apply unique optimization pressure to them.[24] This ensures that important interfaces receive dedicated optimization effort, preventing them from being starved by a single, unrelated critical path elsewhere in the design.

- **Tool-Specific Syntax:**
  - **Synopsys (Design Compiler / Fusion Compiler):** The command directly creates and populates the group.
    Tcl
    # Create a high-priority group for paths from inputs to the 'control_regs' module
    group_path -name control_path_group -from [all_inputs] -to

    2
  - **Cadence (Genus Synthesis Solution):** The concept is similar but involves associating path groups with "cost groups," which are the entities that optimization settings are applied to.
    Tcl
    # Define a cost group with a specific weight
    define_cost_group -name core_logic_paths -weight 2.0
    # Assign paths to this cost group
    path_group -from [all_inputs] -to [find / -inst core_logic*] -group core_logic_paths

    26

Leveraging weight and critical_range for Prioritized Optimization

Within the group_path command, two options provide fine-grained control over the optimization algorithm's behavior: -weight and -critical_range.

- **-weight:** This option assigns a multiplicative cost to any timing violations within the specified group. The default weight is 1.0. By setting a weight of 2.0, for example, the designer instructs the tool to treat a -0.1ns violation in this group as being as "costly" as a -0.2ns violation in a default-weight group.[2] This forces the tool to expend more effort to fix paths in the higher-weighted group, even if it means sacrificing some slack on less critical paths. The valid range for the weight is typically from 0.0 (no optimization) to 100.0.[17]
- **-critical_range:** By default, the optimization engine focuses solely on the path with the worst slack (WNS) within each group. The -critical_range option broadens this focus. It defines a slack window relative to the WNS path. All violating paths whose slack falls within this window are also considered "critical" and are optimized concurrently.[2] For instance, if a group has a WNS of -0.8ns and a

-critical_range of 0.3ns is set, the tool will work on all paths with slacks between -0.8ns and -0.5ns. This is vital for preventing a "whack-a-mole" scenario, where fixing the single worst path causes a slightly less critical path to become the new WNS, leading to little overall progress.[28]

Example Usage:
Consider a design with a critical data processing block (datapath_core) and a less critical control block (config_regs). The goal is to ensure the datapath meets its aggressive timing goals, even if it costs some area or slightly degrades the timing of the control logic.

Tcl

```
# Synopsys Example
# Group all paths ending in the datapath with a high weight and a critical range
group_path -name DP_PATHS -to -weight 2.5 -critical_range 0.2

# Group control paths with a lower weight to de-prioritize them
group_path -name CTRL_PATHS -to -weight 0.8
```

In this scenario, the tool will prioritize fixing violations in the DP_PATHS group, treating them as 2.5 times more critical than default paths. It will also optimize all datapath paths that are within 0.2ns of the worst datapath path. Conversely, it will deprioritize the CTRL_PATHS, potentially sacrificing their slack to achieve the goals for the datapath.

Optimization Phases: Initial High-Effort vs. Incremental Compiles

The synthesis process is not a single-pass execution but an iterative flow involving different optimization strategies to balance QoR with runtime.

- **Initial High-Effort Compile:** The first synthesis run on a new or significantly modified RTL is typically a full, high-effort compilation. In Synopsys tools, this is invoked with compile_ultra, while in Cadence Genus, it involves running the full flow (syn_generic, syn_map, syn_opt) with high-effort settings.[2] This phase performs a comprehensive, from-scratch optimization, including architectural choices, extensive logic restructuring, and detailed technology mapping. It establishes a baseline QoR but can be time-consuming.[30]
- **Incremental Compile:** After the initial compile, designers analyze the results and often refine constraints (e.g., add path groups, adjust I/O delays). For these subsequent runs, a

full high-effort compile is inefficient. An incremental compile re-optimizes only the portions of the design affected by the changes.[30] This dramatically reduces runtime, enabling rapid design iterations. A common methodology is to follow a high-effort initial compile with one or more low-effort incremental compiles to fine-tune the design based on analysis.[30]

Advanced Logic and Sequential Optimization Techniques

Beyond standard technology mapping, modern synthesis tools employ a portfolio of sophisticated algorithms to restructure the netlist for superior Power, Performance, and Area (PPA). These techniques often involve manipulating the design's hierarchy and the placement of sequential elements to unlock optimization potential that is otherwise inaccessible.

Boundary Optimization: Seeing Beyond the Hierarchy

In a hierarchical design, module boundaries typically act as opaque barriers to optimization. Boundary optimization is a collection of techniques that allows the synthesis tool to peer across these boundaries and exploit logical relationships between a parent module and its instances.[2] This enables a more global optimization context without fully flattening the design.

- **Mechanisms of Boundary Optimization:**
  - **Constant Propagation:** If an input port of a sub-module is connected to a constant logic '0' or '1' in the parent design, the tool propagates this constant value into the sub-module. This can lead to a cascade of logic simplification, as entire sections of the sub-module's logic may become redundant and can be removed.[2]
  - **Unconnected Port Removal:** If an output port of a sub-module is not connected to any logic in the parent design (i.e., it is unloaded), the tool can trace back from this port and remove all the driver logic within the sub-module, saving area and power.[2]
  - **Inverter Pushing (Phase Inversion):** An inverter located at a module boundary can often be "pushed" into or out of the module. For example, if a sub-module's output is inverted by the parent, it may be more efficient to push that inversion inside the sub-module, where it might be absorbed into an existing logic gate or enable the use of a cell with an inverted output, reducing overall delay and area.[2]
  - **Equal/Opposite Pin Merging:** If a sub-module has two pins that are driven by the

same signal (or one by the signal and the other by its inverse), the tool can recognize this redundancy and merge the internal logic, sourcing it from a single input pin.[29]

Ungrouping Strategies: The Trade-off Between Hierarchy and Global Optimization

Ungrouping, also known as flattening, is a more aggressive form of optimization that completely dissolves the hierarchical boundary of a sub-design, merging its logic into the parent module.[2]

- **Benefits:** By creating a larger, unified logic cone, ungrouping provides the synthesis tool with maximum flexibility. It can perform logic restructuring and resource sharing across what were previously separate hierarchical domains. This is particularly effective for improving timing on critical paths that traverse multiple small modules and for reducing area by sharing common logic (e.g., adders or comparators) that existed in separate instances.[2]
- **Drawbacks and Trade-offs:**
  - **Hierarchy and Debug:** The primary drawback is the loss of the original design hierarchy in the netlist. A flattened netlist is significantly more difficult for engineers to read, navigate, and debug. It also complicates the process of applying Engineering Change Orders (ECOs) to specific functional blocks.[30]
  - **Wire Load Model (WLM) Impact:** In traditional logical synthesis flows, ungrouping can have a counterintuitive negative impact on timing. A small, compact sub-module might be assigned an optimistic WLM, assuming short interconnects. When ungrouped into a large parent module that uses a more pessimistic WLM, the estimated wire delays for the sub-module's internal nets can increase dramatically, potentially negating any gains from logic optimization.[30] This trade-off is less severe in modern physical-aware synthesis flows that do not rely on WLMs.
- **Tool Commands and Control:**
  - **Synopsys Tools (Design Compiler/Fusion Compiler):** The ungroup command allows for manual flattening of specified instances. Modern tools also feature "auto-ungrouping," which can be controlled via switches. For example, compile_ultra -auto_ ungroup area will automatically ungroup small hierarchies to optimize for area, while -no_autoungroup disables this behavior.[2] The set_ungroup false attribute can be applied to specific instances to protect them from being ungrouped.[31]
  - **Cadence Genus:** Genus also provides an ungroup command. Automatic ungrouping can be prevented by setting attributes like set_db ungroup_ok false on hierarchical instances that must be preserved.[40]

The decision to ungroup a module is a strategic trade-off. A seemingly beneficial local optimization, like ungrouping a small module to fix an internal critical path, can have unintended global consequences. For instance, an internal high-fanout net, once exposed to the parent module, might require a large buffer tree that introduces significant routing congestion in the physical layout, leading to new and more challenging timing violations. This illustrates the critical link between logical hierarchy decisions and their physical ramifications, reinforcing the value of physical-aware synthesis which can better predict and manage these cross-domain effects.

Register Retiming: Balancing Combinational Logic Delays

Register retiming is a powerful sequential optimization technique that repositions registers across combinational logic gates to balance path delays and improve the overall clock frequency of the design.[2] Unlike combinational optimization, retiming alters the sequential structure of the netlist while preserving its functional behavior and latency.

- **Mechanism:** The tool analyzes the design as a timing graph where nodes are registers and edges are the combinational logic paths between them. It then solves an optimization problem to find a new placement for the registers that minimizes the longest delay between any two registers (the critical path).
    - **Forward Retiming:** Moves a register from the input(s) of a logic gate to its output. This is possible only if all inputs to the gate are fed by registers with identical control signals (clock, enable, reset).[43]
    - **Backward Retiming:** Moves a register from the output of a logic gate to all of its inputs. This requires that the gate's output is the sole fanout of the register being moved.[43]
    - **Algorithms:** The underlying optimization is often solved using graph-based algorithms like the Bellman-Ford algorithm, which can efficiently find the optimal register movements to achieve a target clock period.[42]
- **Application and Limitations:** Retiming is most effective in designs with regular, pipelined datapaths, such as those found in DSP or processor cores. However, its application can be restricted. Asynchronous resets, for example, can block retiming because moving a register might disconnect it from its reset source. Similarly, complex control logic, timing exceptions (set_false_path), or explicit dont_touch attributes on registers will prevent the tool from moving them, as their positions are considered critical to the design's intended function.[42]

Concurrent Clock and Data (CCD) Optimization: A Holistic Approach to Timing

Concurrent Clock and Data (CCD) optimization represents a paradigm shift from traditional timing closure methods. Instead of treating the clock network as an ideal entity with zero skew and focusing solely on fixing data paths, CCD holistically optimizes both simultaneously.[45]

- **Concept:** The core principle of CCD is to use clock skew as a resource. If a data path is too slow and violates a setup constraint, the tool has two choices: speed up the data path (the traditional approach) or slow down the clock path by intentionally delaying the clock signal's arrival at the capturing flip-flop. This "useful skew" effectively "borrows" time from the subsequent timing path, providing more margin for the critical path to meet its timing.[45]
- **Mechanism:** CCD is not a single command but an optimization philosophy integrated throughout the modern physical design flow. During placement and routing optimization, the tool constantly evaluates the trade-off between a data path fix (e.g., upsizing a gate, which increases area and power) and a clock path fix (e.g., inserting a clock buffer, which also has PPA costs). By considering both options concurrently, the tool can find a more globally optimal solution that balances timing, power, and area.
- **Implementation in Tools:** Advanced RTL-to-GDSII platforms like Synopsys Fusion Compiler and IC Compiler II are built around this concept. They feature unified optimization engines that perform CCD at every stage, from initial placement through clock tree synthesis and post-route optimization.[49] This continuous, co-optimized approach ensures that decisions made early in the flow are consistent with the final timing closure goals, leading to a highly convergent and predictable design process.

Integrating Design for Testability (DFT)

Design for Testability (DFT) refers to a set of design techniques that add testability features to a hardware design. The goal is to make it easier and more efficient to test the manufactured device for physical defects. The most common DFT methodology for logic testing is scan design, which is typically implemented during or immediately after logic synthesis.

Scan Insertion: Replacing Standard Flip-Flops with Scan-Equivalents

The fundamental challenge in testing sequential circuits is the difficulty of controlling and observing the state of internal flip-flops. Scan insertion solves this problem by modifying each flip-flop to be part of a shift register during a special test mode.[52]

- **Purpose:** The primary goal is to provide direct access to all state elements in the design, effectively converting a difficult sequential testing problem into a much simpler combinational one.[52]
- **Mechanism:** The synthesis tool, when invoked with a scan-aware option (e.g., compile_ultra -scan in Synopsys tools), replaces the standard flip-flops in the netlist with their scan-equivalent versions from the technology library.[2] A scan flip-flop contains an additional 2-to-1 multiplexer on its data input. In normal (functional) mode, the multiplexer selects the functional data input (D). In test mode, a global scan_enable (SE) signal switches the multiplexer to select a dedicated scan_in (SI) port.[55]
- **Inputs:** The process requires a synthesized netlist and a technology library that includes scan-enabled flip-flops (e.g., SDFF instead of DFF).[55]
- **Outputs:** The output is a modified gate-level netlist where nearly all sequential cells are now scan-capable. Some flip-flops may be excluded by the designer for specific reasons (e.g., those in sensitive analog-mixed-signal interfaces) and are designated as non-scan cells.[52]

Scan Chain Stitching: Constructing the Test Infrastructure

After scan insertion, the individual scan-ready flip-flops are logically isolated. Scan chain stitching is the process of connecting them together to form one or more long shift registers, known as scan chains.[52]

- **Purpose:** To create a serial path through which test patterns can be shifted in to control the state of the flip-flops and captured responses can be shifted out for observation.[55]
- **Mechanism:** The DFT tool connects the functional output (Q) of one scan flip-flop to the scan input (SI) of the next flip-flop in the chain. The SI of the first flip-flop in a chain is connected to a primary input pin of the chip (ScanIn), and the Q of the last flip-flop is connected to a primary output pin (ScanOut).[52]
- **Configuration and Challenges:**
  - **Chain Count:** A design is typically partitioned into multiple scan chains to reduce the test application time (since chains can be loaded in parallel) and to manage power consumption during test.[52]
  - **Clock Domain Crossing:** When a scan chain crosses between flip-flops clocked by different or skewed clocks, there is a risk of hold time violations during the shift operation. To prevent this, DFT tools automatically insert "lock-up latches"—simple latches that hold the data stable during the clock skew window—at the domain

boundaries.[53]
    - **Physical Awareness:** In modern flows, scan chain stitching is often performed after placement. This allows the tool to order the flip-flops in the chain based on their physical location, minimizing the total wirelength of the scan connections and reducing routing congestion.[53]
- **Outputs:** The primary outputs are the scan-stitched netlist and a Scan DEF (Design Exchange Format) file. The Scan DEF contains the logical ordering of the cells in each scan chain, which is passed to the place-and-route tool to guide the physical routing of the scan connections.[1]

The Role of the Test Protocol File

The test protocol file is a critical output of the DFT insertion process that serves as the bridge to the next stage: Automatic Test Pattern Generation (ATPG). This file describes the operational procedures and timing for using the newly inserted scan test structures.[57]

- **Purpose:** It provides a formal description to the ATPG tool (e.g., Synopsys TestMAX ATPG, Mentor Tessent) of how to manipulate the chip's primary inputs to perform scan operations.
- **Format and Content:** The format is often STIL (Standard Test Interface Language) or a similar proprietary format. The file contains two key components:
    - **TimePlate Definitions:** These define the timing of events within a single tester cycle. It specifies when primary inputs should be driven (force_pi), when outputs should be measured (measure_po), and when clocks should be pulsed (pulse).[57]
    - **Procedure Definitions:** These define multi-cycle operations. For example, a load_unload procedure describes the sequence of events needed to put the chip into test mode (e.g., asserting scan_enable) and then apply a specified number of shift cycles to load or unload the scan chains.[57]

Without the test protocol file, the ATPG tool would have no knowledge of the design-specific scan architecture, clocking scheme, or timing, and would be unable to generate valid test patterns.

The Transition to Physical Implementation

In the era of deep-submicron process technologies, the distinction between logical design

and physical design has blurred. The parasitic effects of interconnects (wires) now dominate the total delay of timing paths, making it impossible to achieve timing closure without considering the physical layout during synthesis. This has driven the evolution from traditional logical synthesis to modern physical-aware synthesis flows.

Logical vs. Physical-Aware Synthesis: Bridging the Correlation Gap

The fundamental difference between these two methodologies lies in how they estimate interconnect delay, which directly impacts the accuracy of their timing calculations and the quality of their optimizations.

- **Logical Synthesis:** This is the conventional approach, which operates in a "physical vacuum." It has no information about where cells will be placed on the die. To estimate wire delays, it relies on statistical **Wire Load Models (WLMs)**.[59] A WLM is essentially a lookup table that provides an estimated wire length, capacitance, and resistance based on a net's fanout (the number of pins it connects to) and the size of the block it is in.[2] This statistical guess is highly inaccurate for advanced nodes, leading to a significant "correlation gap" where a design that appears to meet timing in synthesis reports massive violations after place-and-route. This poor correlation results in numerous, time-consuming iterations between the front-end (synthesis) and back-end (P&R) teams.[59]
- **Physical-Aware Synthesis (Physical Synthesis):** This modern methodology embodies a "shift-left" philosophy, incorporating physical layout information much earlier in the design flow.[59] By using an initial floorplan and performing a trial placement of cells, the tool gains a realistic understanding of the design's physical topology. This allows it to perform a much more accurate estimation of interconnect parasitics, leading to optimizations that are highly correlated with the final post-layout timing.[59] The primary benefit is a predictable, convergent design flow that drastically reduces the number of iterations required to achieve timing closure, accelerating the overall time-to-market.[59]

Essential Inputs for Physical Awareness: LEF, DEF, and Technology Files

To become "physically aware," the synthesis tool requires additional input files that describe the physical characteristics of the technology and the initial layout of the design.

- **LEF (Library Exchange Format):** This file provides the physical abstract view of all

library cells and the technology itself.[3] It is typically split into two parts:
    - **Technology LEF (.tech.lef):** Contains process-specific information provided by the foundry. This includes definitions of all metal and via layers, their electrical properties (e.g., sheet resistance), and physical design rules like minimum width and spacing.[3]
    - **Cell LEF:** Describes the physical layout of each standard cell and macro. For each cell, it defines its physical boundary (dimensions), the location and layer of all its pins, and any internal metal blockages that would obstruct routing.[3]
- **DEF (Design Exchange Format):** This file contains the design-specific physical information.[1] For physical-aware synthesis, an initial floorplan DEF is provided as input. This file specifies:
    - The overall die area and core dimensions.
    - The placement and orientation of large macros (memories, IP blocks).
    - The locations of the design's I/O pins/ports.
    - Any pre-placed cells or placement blockages (keep-out regions).
- **RC Coefficient File (e.g., TLU+, QRC Tech File):** This file provides detailed parasitic information for the interconnect layers. It contains a matrix of resistance (R) and capacitance (C) values per unit length for each metal layer, including coupling capacitance values for different wire spacing scenarios. This file is critical for the tool's internal parasitic extractor to accurately model wire delays.[1]

From Wire Load Models (WLMs) to Global Route-Based RC Estimation

The key innovation that enables physical-aware synthesis is the move away from WLMs towards a more deterministic delay estimation method.

- **WLM Limitations:** As established, WLMs are statistical and placement-agnostic. Their estimations are based on averages from past designs and do not account for the actual placement of cells in the current design. Two nets with the same fanout could have vastly different physical lengths—one connecting adjacent cells and the other spanning the entire die. A WLM would assign them the same delay, leading to gross inaccuracies.[60]
- **Global Route for RC Estimation:** Physical-aware synthesis tools perform a process called "virtual routing" or "global routing" for delay estimation.[59] After an initial coarse placement of the standard cells, the tool overlays a coarse grid (composed of "g-cells") on the design. For each net, it finds an approximate path through these g-cells from the driver to the loads. While this is not a detailed, track-by-track route, it provides a highly accurate estimate of the net's topology and length. Using this estimated length and the parasitic data from the RC coefficient file, the tool can perform an on-the-fly RC extraction and calculate a much more realistic interconnect delay. This global route-based estimation is the core mechanism that provides the high correlation

between pre- and post-layout timing.[59]

The Physical Synthesis Flow and its Enhanced Outputs

The workflow for physical synthesis is an integrated process that combines traditional synthesis optimizations with physical layout tasks. In a tool like Synopsys Fusion Compiler, a single command such as compile_fusion can execute a flow that includes logic optimization, placement, and physical optimization stages in a unified environment.[39]

The outputs of this flow are more comprehensive than those of logical synthesis. In addition to the optimized gate-level netlist, updated SDC, and QoR reports, the key additional output is a **placed DEF file**.[1] This DEF file contains the optimized netlist along with the physical coordinates of every standard cell and macro. This serves as a high-quality, timing-aware starting point for the physical design team, significantly reducing the effort required in the subsequent detailed placement and routing stages.

Attribute	Logical Synthesis	Physical-Aware Synthesis
Primary Inputs	RTL, .lib, SDC	RTL, .lib, SDC, **LEF, DEF, RC Coeffs**
Delay Estimation	Wire Load Models (WLM)	Global Route-based RC Extraction
Physical Awareness	None (placement-agnostic)	High (placement-aware)
Correlation	Poor	High
Design Flow	Sequential, requires many iterations	Convergent, fewer iterations
Primary Outputs	Netlist, SDC, Reports	Netlist, SDC, Reports, **Placed DEF**
Best For	Older process nodes (>90nm), FPGAs	Advanced process nodes (<65nm), high-performance ASICs

Advanced Power Optimization Strategies

With power consumption becoming a first-order design constraint for nearly all modern electronic devices, synthesis and physical design tools have incorporated sophisticated techniques to minimize both dynamic (switching) and static (leakage) power. Two of the most impactful structural optimization techniques are register banking and activity-driven optimization using SAIF files.

Register Banking (Multi-Bit Flip-Flops) for Power and Area Reduction

Register banking, also known as multi-bit flip-flop (MBFF) implementation, is a physical optimization technique that merges multiple single-bit registers into a single, larger standard cell that contains multiple flip-flops.[12]

- **Concept:** Instead of instantiating eight individual 1-bit flip-flops, the tool can replace them with a single 8-bit MBFF cell. The key feature of an MBFF is that all the internal flip-flops share a common clock driver (inverter), and often common set/reset logic.[69]
- **Benefits:**
  - **Power Reduction:** The primary benefit is a significant reduction in dynamic clock power. The capacitive load of a single clock pin on an N-bit MBFF is substantially lower than the combined load of N individual flip-flop clock pins. This reduction in clock sinks leads to a smaller, less complex clock tree with fewer buffers, directly cutting down on the largest source of dynamic power in many designs.[69]
  - **Area and Congestion Reduction:** An N-bit MBFF cell is physically smaller than N individual 1-bit cells, leading to direct area savings. Furthermore, it reduces routing congestion by consolidating multiple clock, scan, and potentially reset nets into a smaller number of connections.[69]
- **Mechanism and Tool Commands:** Synthesis tools identify candidates for banking by finding groups of flip-flops that share common attributes, such as being driven by the same clock and reset signals, and having no complex or conflicting timing constraints.[69] In physical-aware flows, physical proximity is also a key criterion.
  - In **Cadence Genus**, MBFF optimization is enabled using the command set_db use_multibit_cells true. The tool can then automatically infer and map to MBFFs during the syn_map stage. Manual banking can be performed with the create_multibit_cells command.[74]

- In **Synopsys Fusion Compiler**, the application option compile.seqmap.enable_register_merging controls this behavior during synthesis.[24] The tool also supports advanced placement-aware banking to group physically adjacent registers during physical optimization.[75]

Activity-Driven Power Optimization with SAIF Files

Default power optimization techniques often assume a uniform or default switching activity across the design, which can lead to suboptimal results. Activity-driven optimization uses realistic simulation data to guide the tool's decisions, resulting in more effective power reduction.[77]

- **SAIF (Switching Activity Interchange Format):** A SAIF file is an ASCII file that contains information on the switching activity of nets in a design. For each net, it records the toggle rate (number of transitions per unit of time) and the static probability (percentage of time the signal is at logic '1' vs. '0').[77]
- **Generation Flow:** SAIF files are generated from simulation. The most accurate flow involves:
  1. Running a comprehensive functional simulation of the RTL design using a realistic testbench that represents typical, power-dominant use cases.
  2. The simulator (e.g., Synopsys VCS) is configured to dump the signal activity into a SAIF file.[77] For even higher accuracy, a gate-level simulation of the post-P&R netlist can be used to generate a SAIF file, though this occurs much later in the flow.[80]
- **Usage in Synthesis and Physical Design:** The synthesis tool reads the SAIF file using a command like read_saif and annotates the activity data onto the corresponding nets in the design database. This information is then used to drive a variety of power optimization techniques:
  - **Activity-Driven Clock Gating:** The tool can make more intelligent decisions about where to insert clock gates. It prioritizes gating the clocks of registers with very low toggle rates, as this provides the maximum power-saving benefit.[12]
  - **Low-Power Placement:** In physical-aware synthesis, the tool can identify nets with high switching activity. It then prioritizes placing the driver and load cells of these nets close together to minimize their wire length. Since dynamic power is directly proportional to capacitance (P_dyn \propto C \cdot V^2 \cdot f \cdot \alpha), reducing the capacitive load (C) of these high-activity nets yields significant power savings.[12]
  - **Logic Restructuring:** The tool can restructure logic cones to minimize the switching activity on high-capacitance internal nodes. For example, it might reorder the inputs to a multiplexer so that the input with the highest toggle rate controls the selection

of a stable data signal, rather than being propagated through the multiplexer unnecessarily.[77]

The techniques of register banking and clock gating are not independent; they have a symbiotic relationship. Clock gating saves power by disabling the clock to a group of registers when none of them are changing state. Register banking physically groups registers together, making them ideal candidates for a shared clock gate. By banking registers that have a common enable condition, the tool can implement a single, highly efficient clock gate that controls the shared clock pin of the MBFF. The power savings from gating this single, larger driver are often greater than gating multiple, smaller, distributed drivers. A truly power-aware synthesis flow co-optimizes these two techniques, using shared enable logic to guide banking decisions and using the physical banking information to create more efficient clock-gating structures.

Conclusion and Recommendations for a Convergent Flow

The journey from a high-level RTL description to a physically optimized gate-level netlist is a complex, multi-stage process governed by a delicate balance of competing objectives: performance, power, and area (PPA). The evolution of VLSI design, particularly at advanced technology nodes, has necessitated a paradigm shift from sequential, logically-focused methodologies to integrated, physically-aware flows. The key to success in modern ASIC design lies in understanding the deep interdependencies between these stages and leveraging the advanced capabilities of modern EDA tools to achieve a convergent flow.

Synthesizing the Flow: Interdependencies and Iterations

This report has detailed the critical stages and techniques within synthesis and physical design, revealing a web of interconnected dependencies:

- **Logical Choices Have Physical Consequences:** A decision made at the RTL or logical synthesis stage, such as ungrouping a module or choosing a specific datapath architecture, has direct and often non-obvious impacts on the physical layout, affecting routing congestion and final timing.
- **Physical Layout Informs Logical Optimization:** Conversely, physical information, such as an initial floorplan and placement, provides the crucial context needed for the

synthesis engine to make intelligent decisions about gate sizing, buffering, and logic restructuring. The failure of traditional Wire Load Models and the rise of global route-based RC estimation are testaments to this principle.
- **PPA Objectives are Interlocked:** Optimizing for one goal often impacts another. Aggressive timing optimization through the use of high-drive cells can increase both area and power consumption. Power optimization techniques like clock gating and register banking must be implemented in a timing-aware manner to avoid creating new critical paths.

The ultimate goal of a modern design flow is **convergence**. This means minimizing the number of costly, time-consuming iterations between the front-end (synthesis) and back-end (place-and-route) teams. By incorporating physical awareness, DFT constraints, and power intent early in the synthesis process, a convergent flow ensures that the initial netlist is already on a viable path to meeting all design goals, drastically reducing late-stage surprises and accelerating the overall time-to-market.

Best Practices for Achieving Optimal PPA in Complex Designs

Based on the advanced techniques and methodologies discussed, a set of best practices emerges for engineering teams seeking to achieve optimal PPA in today's complex SoC designs:

1. **Embrace a "Shift-Left" Mentality:** Do not treat synthesis as an isolated, purely logical step. Integrate physical information (floorplan DEF, LEF), power intent (UPF), and test requirements (scan configuration) at the earliest possible stage. This provides the synthesis tool with the complete context needed to produce a high-quality, physically-aware netlist.
2. **Prioritize High-Quality Constraints:** The SDC file is the single most important document guiding the optimization engine. Invest significant effort in creating a complete and accurate set of constraints. Define all clocks correctly, provide realistic I/O delays, and judiciously apply timing exceptions. Use advanced features like path groups with weights and critical ranges to strategically guide the tool's effort towards the most critical parts of the design, rather than relying on default behavior.
3. **Leverage Modern, Integrated Tools:** Fully utilize the capabilities of modern RTL-to-GDSII platforms like Synopsys Fusion Compiler or the Cadence integrated digital flow. These tools are built from the ground up to support physical-aware synthesis, concurrent clock and data optimization, and multi-objective PPA optimization. Avoid legacy scripts and flows based on outdated concepts like Wire Load Models for any design at 65nm or below.
4. **Analyze, Diagnose, and Iterate Intelligently:** The reports generated by synthesis tools

are a rich source of diagnostic information. Use them to understand the root cause of PPA issues before attempting to apply fixes. Analyze the relationship between WNS and TNS to determine if a timing problem is local or systemic. Use fast, incremental synthesis runs to test the impact of specific constraint changes or minor RTL modifications, enabling rapid exploration of the design space without the overhead of a full recompilation.

By adopting these principles, design teams can navigate the complexities of modern VLSI design, transforming the traditionally iterative and often unpredictable process of synthesis and physical design into a convergent, predictable, and efficient flow that delivers superior results.

Works cited

1. Synthesis Methodology & Netlist Qualification - Design And Reuse, accessed September 13, 2025, https://www.design-reuse.com/article/61358-synthesis-methodology-netlist-qualification/
2. logic_synthesis_html_1.pdf
3. LEF, DEF & LIB - SignOff Semiconductors, accessed September 13, 2025, https://signoffsemiconductors.com/lef-def-lib/
4. Contents in the LIB, LEF and DEF files - Learn VLSI, accessed September 13, 2025, https://learnvlsi.com/pd/liblef-and-def/222/
5. Inputs for Physical Design | VLSI - logicmadness.com, accessed September 13, 2025, https://logicmadness.com/inputs-for-physical-design/
6. Tutorial on Cadence Genus Synthesis Solution, accessed September 13, 2025, http://public2.yuantsy.com/Test/ECE201A/GenusTutorial.pdf
7. ECE 5745 Tutorial 5: Synopsys/Cadence ASIC Tools - GitHub Pages, accessed September 13, 2025, https://cornell-ece5745.github.io/ece5745-tut5-asic-tools/
8. What is difference between logical synthesis and physical synthesis? - VLSI Interview Community - Quora, accessed September 13, 2025, https://vlsiqaforstudents.quora.com/What-is-difference-between-logical-synthesis-and-physical-synthesis
9. SYNTHESIS - VLSI TALKS, accessed September 13, 2025, https://vlsitalks.com/physical-design/synthesis/
10. Introduction to Netlist (.v), LEF, and DEF Files - Wix.com, accessed September 13, 2025, https://vorasaumil.wixsite.com/pdinsight/post/introduction-to-netlist-lef-def-files
11. (PDF) Optimization of Physically-Aware Synthesis for Digital Implementation Flow, accessed September 13, 2025, https://www.researchgate.net/publication/324526833_Optimization_of_Physically-Aware_Synthesis_for_Digital_Implementation_Flow
12. Power Optimization in Design Compiler Datasheet - Synopsys, accessed September 13, 2025, https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datas

heets/power-compiler-ds.pdf
13. synthesis: Synopsys DC - maaldaar, accessed September 13, 2025, http://www.maaldaar.com/index.php/vlsi-cad-design-flow/synthesis/synthesis-dc
14. Under the Hood of Genus - Breakfast Bytes - Cadence Blogs, accessed September 13, 2025, https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/under-the-hood-of-genus
15. Physical Synthesis | iVLSI Technologies, accessed September 13, 2025, https://ivlsi.com/physical-synthesis/
16. Synopsys Fusion Compiler-Comprehensive RTL-to-GDSII Implementation System | PPTX, accessed September 13, 2025, https://www.slideshare.net/slideshow/synopsys-fusion-compilercomprehensive-rtltogdsii-implementation-system/239633312
17. Timing Analysis: Path Group | PDF | Electrical Circuits | Electronic ..., accessed September 13, 2025, https://www.scribd.com/document/380892093/synopsys-class-3-6-1
18. Total Negative Slack vs Worst Negative Slack - Adaptive Support - AMD, accessed September 13, 2025, https://adaptivesupport.amd.com/s/question/0D52E00006iHoJ8SAK/total-negative-slack-vs-worst-negative-slack?language=en_US
19. What are TNS, WNS, THS, NVP and WHS 原创 - CSDN博客, accessed September 13, 2025, https://blog.csdn.net/cy413026/article/details/86098466
20. [Glean] Two terms for timing analysis: WNS and TNS | SingularityKChen, accessed September 13, 2025, https://singularitykchen.github.io/blog/2022/10/04/Glean-WNS-and-TNS/
21. Reduce TNS/WNS in synthesis with individual path algorithm - EDN, accessed September 13, 2025, https://www.edn.com/reduce-tns-wns-in-synthesis-with-individual-path-algorithm/
22. How to Analyze Timing Reports in Vivado? : r/FPGA - Reddit, accessed September 13, 2025, https://www.reddit.com/r/FPGA/comments/vubr1e/how_to_analyze_timing_reports_in_vivado/
23. dc_shell.html - VLSI IP, accessed September 13, 2025, http://www.vlsiip.com/dc_shell/
24. Synthesis Optimization Techniques 7 | PDF | Program Optimization ..., accessed September 13, 2025, https://www.scribd.com/presentation/725405098/Synthesis-Optimization-Techniques-7
25. Optimization Synthesis | PDF - Scribd, accessed September 13, 2025, https://www.scribd.com/presentation/724197864/Optimization-Synthesis-1
26. How to Use the Path Group for Timing Analysis in Genus Synthesis Solution - YouTube, accessed September 13, 2025, https://www.youtube.com/watch?v=ZmZ28wDVDgQ
27. Creating Path Groups | PDF | Computing - Scribd, accessed September 13, 2025,

https://www.scribd.com/document/838908256/Creating-Path-Groups
28. Fusion Compiler Videos - Synopsys, accessed September 13, 2025, https://www.synopsys.com/implementation-and-signoff/physical-implementation/fusion-compiler/videos.html
29. Design Compiler Tutorial, accessed September 13, 2025, https://picture.iczhiku.com/resource/eetop/SHiWUjlOrZkFLxnc.pdf
30. High Performance Synthesis using Design Compiler, accessed September 13, 2025, https://picture.iczhiku.com/resource/convert/8017d17bd56b4d4f831eca9d0d2f2822.pdf
31. PowerPoint Presentation - UTK-EECS, accessed September 13, 2025, https://web.eecs.utk.edu/~dbouldin/protected/chip-synthesis/lecture3_OptimizationDCUltra.ppt
32. Saving Compile Time Series 3: Using Incremental Synthesis - Adaptive Support - AMD, accessed September 13, 2025, https://adaptivesupport.amd.com/s/article/991841
33. Interactive incremental synthesis flow for integrated circuit design - Google Patents, accessed September 13, 2025, https://patents.google.com/patent/US10614188B2/en
34. How does Synthesis Tool do Boundary Optimization? – Chipress, accessed September 13, 2025, https://chipress.online/2025/03/09/how-does-synthesis-tool-do-boundary-optimization/
35. What Is Boundary Optimization - YouTube, accessed September 13, 2025, https://www.youtube.com/watch?v=6yqJUvCl25c
36. Different Types of Boundary Optimization in VLSI Synthesis - SuccessBridge, accessed September 13, 2025, https://www.successbridge.co.in/types-of-boundary-optimization-in-vlsi-synthesis/
37. DC Ultra: Concurrent Timing, Area, Power, and Test ... - Synopsys, accessed September 13, 2025, https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/dc-ultra-ds.pdf
38. Tips for Area Reduction using Synopsys DC and Cadence RTL Compiler Tools, accessed September 13, 2025, https://daffy1108.wordpress.com/2011/01/11/tips-for-area-reduction-using-synopsys-dc-and-cadence-rtl-compiler-tools/
39. FC - Flow 1 | PDF - Scribd, accessed September 13, 2025, https://www.scribd.com/document/711675828/FC-FLOW-1
40. Cadence Commands | PDF | Hardware Description Language | Digital Technology - Scribd, accessed September 13, 2025, https://www.scribd.com/document/268928504/Cadence-Commands
41. Genus - Hierarchy - Logic Design - Cadence Technology Forums, accessed September 13, 2025, https://community.cadence.com/cadence_technology_forums/f/logic-design/4671

5/genus---hierarchy
42. Retiming and Register Balancing in Logic Synthesis | by Rana Umar ..., accessed September 13, 2025, https://medium.com/@ranaumarnadeem/retiming-and-register-balancing-in-logic-synthesis-4a25c8ac96b6
43. Retiming in Vivado Synthesis - Adaptive Support - AMD, accessed September 13, 2025, https://adaptivesupport.amd.com/s/article/934201
44. Retiming - Wikipedia, accessed September 13, 2025, https://en.wikipedia.org/wiki/Retiming
45. RL-CCD: Concurrent Clock and Data Optimization using Attention ..., accessed September 13, 2025, https://gtcad.gatech.edu/www/papers/yi-chen_DAC23_RL-CCD.pdf
46. Presentation – 60 DAC - conference program, accessed September 13, 2025, https://60dac.conference-program.com/presentation/?id=RESEARCH1166&sess=sess134
47. Concurrent Clock and Data Optimization with IC Compiler II Design - Synopsys, accessed September 13, 2025, https://www.synopsys.com/implementation-and-signoff/resources/whitepapers/iccii-ccd.html
48. CCD Everywhere throughout the RTL-to-GDSII Design Flow with Synopsys' Fusion Compiler - YouTube, accessed September 13, 2025, https://www.youtube.com/watch?v=W1Lurff-eis
49. IC Compiler II: Industry Leading Place and Route System - Synopsys, accessed September 13, 2025, https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/ic-compiler-ii-ds.pdf
50. Synopsys Unveils Fusion Compiler, Enabling 20 Percent Higher Quality-of-Results and 2X Faster Time-to-Results, accessed September 13, 2025, https://news.synopsys.com/2018-11-06-Synopsys-Unveils-Fusion-Compiler-Enabling-20-Percent-Higher-Quality-of-Results-and-2X-Faster-Time-to-Results
51. Fusion Compiler | Synopsys, accessed September 13, 2025, https://www.synopsys.com/implementation-and-signoff/resources/datasheets/fusion-compiler-ds.html
52. DFT, Scan and ATPG - VLSI Tutorials, accessed September 13, 2025, https://vlsitutorials.com/dft-scan-and-atpg/
53. Scan Insertion for better ATPG - Tessent Solutions, accessed September 13, 2025, https://blogs.sw.siemens.com/tessent/2017/04/24/scan-insertion-for-better-atpg/
54. DFT Scan Types And Their Mechanism - ChipEdge, accessed September 13, 2025, https://chipedge.com/resources/dft-scan-types-and-their-mechanism/
55. Synthesis: DFT-Aware Design Essentials | by Rana Umar Nadeem ..., accessed September 13, 2025, https://medium.com/@ranaumarnadeem/synthesis-dft-aware-design-essentials-56bf3662bb52
56. Scan Chains: PnR Outlook - Design And Reuse, accessed September 13, 2025, https://www.design-reuse.com/article/61206-scan-chains-pnr-outlook/

57. Writing a Test Procedure file | Prasad Addagarla, accessed September 13, 2025, https://prasadaddagarla.com/2008/09/11/writing-a-test-procedure-file/
58. Computer-Aided VLSI System Design DFT Compiler Lab: Insert Scan Chain, accessed September 13, 2025, https://cc.ee.ntu.edu.tw/~jhjiang/instruction/courses/fall11-cvsd/Lab4-Testing_DFT.pdf
59. What is Physical Synthesis? – How Does it Work? - Synopsys, accessed September 13, 2025, https://www.synopsys.com/glossary/what-is-physical-synthesis.html
60. SYNTHESIS (LOGIC/HDL VERSUS PHYSICALLY AWARE) - FPGAs: World Class Designs, accessed September 13, 2025, https://www.fpgakey.com/tutorial/section389
61. pd. Differentiate between Logical and... | by Medha Kadam - Medium, accessed September 13, 2025, https://medium.com/@medhakadam21/pd-f524d5363a59
62. LEF, DEF & LIB - Physical Design, STA & Synthesis, DFT, Automation & Flow Dev, Verification Services. Turnkey Projects - Scribd, accessed September 13, 2025, https://www.scribd.com/document/445862831/LEF-DEF-LIB-Physical-design-STA-Synthesis-DFT-Automation-Flow-Dev-Verification-services-Turnkey-Projects
63. Wire Load Model in VLSI: A Comprehensive Guide - SuccessBridge, accessed September 13, 2025, https://www.successbridge.co.in/wireload-model-in-vlsi/
64. On the Relevance of Wire Load Models - UCSD VLSI CAD Laboratory, accessed September 13, 2025, https://vlsicad.ucsd.edu/Publications/Conferences/124/c124.pdf
65. Wire Load Model (WLM) - VLSI- Physical Design For Freshers, accessed September 13, 2025, https://www.physicaldesign4u.com/2020/05/wire-load-model-wlm.html
66. When might a standard cell optimized for zero wireload capacitance be used?, accessed September 13, 2025, https://electronics.stackexchange.com/questions/68972/when-might-a-standard-cell-optimized-for-zero-wireload-capacitance-be-used
67. Global and detailed routing, accessed September 13, 2025, http://cc.ee.ntu.edu.tw/~ywchang/Courses/PD_Source/EDA_routing.pdf
68. Routing Tasks - VLSI Master, accessed September 13, 2025, https://verificationmaster.com/routing-tasks/
69. Power reduction in SoC platform - IOSR Journal, accessed September 13, 2025, https://www.iosrjournals.org/iosr-jvlsi/papers/vol10-issue2/Series-1/A10020107.pdf
70. Synthesis of Multi-Bit Flip-Flops for Clock Power Reduction - J-Stage, accessed September 13, 2025, https://www.jstage.jst.go.jp/article/iis/18/2/18_IIS180211/_article/-char/en
71. Low Power Multi-Bit Flip-Flops Design for VLSI Applications - IJERA, accessed September 13, 2025, https://www.ijera.com/special_issue/NCDATES/ECE/PART-1/ECE%20126-2631.pdf
72. A New Multi-Bit Flip-Flop Merging Mechanism for Power Consumption Reduction in the Physical Implementation Stage of ICs Conception - MDPI, accessed

September 13, 2025, https://www.mdpi.com/2079-9268/9/1/3
73. Clock Network Optimization with Multi-bit Flip-flop Generation Considering Multi-corner Multi-mode Timing Constraint - Computer Engineering Research Center (CERC), accessed September 13, 2025, https://www.cerc.utexas.edu/utda/publications/J74.pdf
74. MBIT Genus Flow | PDF | Legal Liability | Damages - Scribd, accessed September 13, 2025, https://www.scribd.com/document/629307681/MBIT-Genus-flow
75. Advanced Fusion Compiler Synthesis and P& R Technologies To Drive Performance and - Scribd, accessed September 13, 2025, https://www.scribd.com/document/647037420/Advanced-Fusion-Compiler-Synthesis-and-P-amp-R-Technologies-to-Drive-Performance-and
76. Multibit Register Synthesis and Physical Implementation Application Note, accessed September 13, 2025, https://picture.iczhiku.com/resource/eetop/WHIdDtIGAhwftxnC.pdf
77. Understanding Your Power Profile from RTL to Gate-level Implementation - Synopsys, accessed September 13, 2025, https://www.synopsys.com/articles/understanding-power-profile.html
78. Power estimation in OpenROAD using SAIF in Verilator - Antmicro, accessed September 13, 2025, https://antmicro.com/blog/2025/07/power-estimation-in-openroad-using-saif-in-verilator/
79. Project POW: Power Estimation and Design for Low Power - Sabih Gerez, accessed September 13, 2025, https://sabihgerez.com/ut/vlsisys/pow06.html
80. Generating SAIF file - Adaptive Support - AMD, accessed September 13, 2025, https://adaptivesupport.amd.com/s/question/0D52E00006hpkFTSAY/generating-saif-file?language=en_US
81. Low Power Design - UTK-EECS, accessed September 13, 2025, https://web.eecs.utk.edu/~dbouldin/protected/chip-synthesis/lecture5_DFTpoweropt.ppt