PnR and STA Flow for a Structured ASIC Platform

1. Introduction & Project Objectives

1.1. What is a Structured ASIC?

A Structured ASIC is a semiconductor device that bridges the gap between the full, field-programmability of an **FPGA (Field-Programmable Gate Array)** and the full, custom-mask-set of a **Standard Cell ASIC (Application-Specific Integrated Circuit)**.

It is based on a **pre-fabricated platform** or **"fabric"**. The foundry manufactures wafers with all base layers (diffusion, polysilicon, lower metals) already patterned into a fixed, 2D grid of logic cells. In this project, the fabric is an array of a repeating **tile** that contains a heterogeneous mixture of basic cells (NANDs, DFFs, buffers, etc.). A customer's design is then implemented by customizing *only* the top few metal layers to "wire up" the required cells.

1.2. Implication: "Placement" as "Assignment"

This pre-fabricated model fundamentally changes the physical design problem.

- **Standard Cell ASIC:** "Placement" means deciding the (x, y) coordinates for all cells on a blank slate.
- **Structured ASIC:** "Placement" is an **"Assignment" problem**. The (x, y) coordinates of all fabric cells are *fixed*. Your task is to solve the complex mapping problem: "Which of the 10,000 available NAND2 slots is the *optimal* one to assign to logical cell cpu.U_alu.1?"

1.3. Project Objectives

Your objective is to design, implement, and validate a complete, reusable physical design flow for this structured ASIC platform. Your flow must be generic and capable of processing multiple, arbitrary netlists. You will be provided with 4-5 test designs (including a 6502 CPU) to validate your flow.

Your flow must perform:

- Placement (Assignment): Map logical cells to physical fabric slots, optimizing for minimal total Half-Perimeter Wirelength (HPWL).
- **Clock Tree Synthesis (CTS):** Build a balanced clock tree using *only* the existing, un-used buffers in the fabric.
- **Netlist ECO (Engineering Change Order):** Modify the netlist to instantiate the CTS and to tie-off all unused logic inputs for power optimization.
- **Routing:** Integrate with the OpenROAD tool to perform global and detailed routing.
- **Signoff (STA):** Perform post-routing Static Timing Analysis (STA) to validate timing and analyze performance.

A strong emphasis is placed on **data visualization** at every stage to debug algorithms and analyze results.

2. Project Timeline & Milestones (6.5 Weeks Total)

Phase	Milestone	Duration	Due Date
Week 0	Groups, Repo, & Project Setup Form 3-person groups. Create private GitHub repo, add instructors. Set up GitHub Project board.	1 Week	Nov 2 (Sun)
Phase 1	Database, Validation, & Visualization Code due for Phase 1.	1 Week	Nov 9 (Sun)
Phase 2	Placement (Assignment) & Analysis Code due for Phase 2.	2 Weeks	Nov 23 (Sun)

CP 1	Checkpoint 1 Presentation 10-min deck: P1/P2 results, placer algorithm, visualizations (HPWL, density).	In class	Week of Nov 24
Phase 3	CTS & ECO Netlist Generation Code due for Phase 3.	1 Week	Nov 30 (Sun)
Phase 4-5	Routing, STA, & Full Flow Code due for Phases 4 & 5.	1 Week	Dec 7 (Sun)
Finals	Final Presentation & Submission: In-class 15-min final presentation (CP2). Final code, Makefile, and README.md due by Dec 10.	In class	Dec 10

3. Development Methodology & Workflow (Required)

This project will enforce a professional GitHub-based development workflow.

3.1. Repository & Project Board

- **Repository:** Each group must create a single **private** GitHub repository. The instructor must be added as a collaborator.
- **Project Board:** You are expected to use a GitHub Project (Kanban board) to manage your project.
- **Issues (Tickets):** All work should be tracked via GitHub Issues. Create issues for all tasks (e.g., "Implement fabric_cells.yaml parser," "Debug Greedy Placer HPWL calculation," "Write Phase 2 Report"). An issue defines a single, discrete task.

3.2. Simple GitHub Workflow

You are expected to follow this simple, protected-branch workflow:

1. The `main` branch is the source of truth. It should be **protected** (in repo settings) to require Pull Request (PR) reviews before merging.
2. To start a task:

   a. Assign yourself an Issue from the project board.

   b. Create a new feature branch from main: git checkout -b feature/my-new-task (e.g., feature/cts-htree).

3. **Do the work:** Make all commits on your feature branch. Commits should reference the issue number (e.g., `git commit -m "Add H-Tree partitioning logic (fixes #12)"`).
4. Finish the task:

   a. Push your branch: git push origin feature/cts-htree.

   b. Open a Pull Request (PR) on GitHub to merge your branch into main.

   c. Link the PR to the Issue it solves.

5. **Code Review:** At least **one other group member** must review the PR, test the changes (e.g., by running `make` and checking the output), and approve it.
6. **Merge:** Once approved, the PR can be merged into `main`. The linked issue will close automatically.

3.3. Policy on AI/LLM Tool Usage

- **Permitted:** The use of Large Language Models (LLMs) and AI coding assistants (e.g., ChatGPT, Gemini, GitHub Copilot) is permitted as a productivity tool. You may use them to brainstorm ideas, debug errors, understand new concepts, and generate boilerplate code.

- **Responsibility:** You are **solely responsible** for the correctness, originality, and functionality of all code you submit. LLM-generated code is a starting point, not a final product. It is often buggy, inefficient, or subtly incorrect.
- **Expectation:** You must be able to **fully explain, defend, and debug** every line of code in your submission. "The AI wrote it" is not an acceptable answer for any bugs, design flaws, or questions during grading.

3.4. Checkpoint Presentations

To ensure steady progress and provide a forum for feedback, there are two required project presentations. For each, your group will prepare and present a short slide deck.

- **Checkpoint 1 (Week of Nov 24):** A 10-minute presentation covering:
  - Project status and GitHub workflow.
  - Phase 1: Fabric validation results and fabric_layout.png visualization.
  - Phase 2: Placer algorithm (Greedy + SA) and key implementation details.
  - Key Results: Show and analyze your placement_density.png and net_length_histogram.png for the 6502 design.
- **Final Presentation (Checkpoint 2) (Dec 10):** This is your final project presentation. Prepare a 15-minute deck covering:
  - Full flow demonstration (e.g., make all DESIGN=...).
  - Phase 3: CTS tree visualization (_cts_tree.png) and analysis.
  - Phase 4: Congestion heatmap (_congestion.png) analysis.
  - Phase 5: Full STA results. Show and analyze the slack_histogram.png and critical_path.png.
  - Final README.md dashboard and conclusions for *all* designs.

4. Project Ecosystem (Input Files)

You will be provided with a set of platform files (static) and design files (variable).

A. Platform Files (Static - Same for all designs)
   - `fabric_cells.yaml`: The physical fabric database. A complete list of all physical cell slots, each with a name and coordinates.
   - `pins.yaml`: Coordinates of all top-level I/O pins.
   - `sky130_hd.lef:` (Provided) LEF file containing physical abstract models (macros, pins, blockages) for all fabric cells.
   - `sky130_hd_timing.lib:` (Provided) Liberty file containing cell timing models (delays, setup/hold checks) for all fabric cells.

B. Design-Specific Files (One set for each test case, 4-5 total)
   - `[design_name]_mapped.json`: (e.g., 6502_mapped.json). A Yosys-generated logical netlist containing all instances, their type, and their connectivity.
   - `[design_name].sdc`: (To be written by you). A Synopsys Design Constraints file defining the clock period and I/O delays for each design.

5. Project Phases & Deliverables (Detailed)

All scripts must be generic and accept a design_name as a parameter. All generated files should be placed in a `build/` directory and organized by design name (e.g., `build/6502/6502.map`).

<u>**Phase 1:** Database, Validation, & Visualization (Due: Nov 9)</u>

- **Objective:** Read all input data and validate if a given design is buildable on the fabric.
- **Tasks:**
  1. **Parse Platform:** Write parsers for fabric_cells.yaml and pins.yaml to create a master fabric_db.
  2. **Parse Design:** Write a parser for [design_name]_mapped.json to create a logical_db and a netlist_graph.
  3. **Validate Design (validator.py):** Compare the logical_db (required cells) against the fabric_db (available slots). The script must exit(1) with an error if len(logical_cells[type]) > len(fabric_slots[type]) for any cell type.
- **Visualization:**
  1. **Fabric Utilization Report:** The validator.py must print a console report (e.g., "NAND2: 4500/10000 used (45%)").

2. **Ground-Truth Plot (visualize.py init):** Generate a plot of the die, core, all pins, and a semi-transparent, color-coded rectangle for **every single fabric slot**.

- Deliverables:
    1. All parsing and validation scripts.
    2. fabric_layout.png (The Ground-Truth Plot).

Phase 2: Placement (Assignment) & Analysis (Due: Nov 23)

- **Objective:** Implement a high-quality, two-stage placement algorithm to map logical instances to physical slots, with the goal of **minimizing total HPWL**.
- **Tasks:**
    1. **Implement Placer (placer.py):** Your placer must be implemented in two distinct stages:
        - **Task 1.A (Required): Greedy Initial Placement:** Implement an "I/O-Driven Seed & Grow" algorithm. This will serve as the high-quality starting point (S<sub>0</sub>) for the annealer.
            1. **Seed:** First, place all cells connected directly to fixed I/O pins at the nearest valid fabric slot.
            2. **Grow:** Iteratively place the most-connected unplaced cell at the barycenter (center of gravity) of its already-placed neighbors.
        - **Task 1.B (Required): Simulated Annealing (SA) Optimization:** Implement an SA algorithm that takes the *output of the Greedy placer* as its initial state. This is critical to avoid the "scattering issue."
        - Task 1.C (Analysis): Simulated Annealing Knobs: Your SA algorithm has several "knobs" that you must tune for a good balance between runtime and solution quality (HPWL).
            - Annealing Schedule:
                - T_initial (Initial Temperature): How "hot" the annealer starts, allowing it to escape the greedy local minimum.
                - Cooling Rate (alpha): The multiplier for decreasing T (e.g., 0.95 for a slow cool, 0.85 for a fast cool).
                - Moves per Temp (N): The number of moves attempted in the inner loop at each temperature step.
            - Hybrid Move Set:
                - P_refine vs P_explore: The probability (e.g., 70%/30%) of choosing a "Refine" swap vs. a windowed "Explore" move.
            - Exploration Window:
                - W_initial: The starting size (e.g., 50% of die width) of the range-limiting window for "Explore" moves.
                - Window Cooling Rate (beta): The rate at which W shrinks (can be tied to alpha or be independent).
        - Task 1.D (Analysis): SA Placer Knob Exploration (10% of Grade): As a separate deliverable, you must explore how these knobs affect the placer's quality (final HPWL) and runtime.
            1. **Run Experiments:** Using one design (e.g., 6502), run your placer at least 5-10 times with different knob settings (e.g., vary the Cooling Rate (alpha) from 0.80 to 0.99, or vary Moves per Temp (N)).
            2. **Generate Plot:** Create a 2D scatter plot showing Runtime (seconds) on the X-axis vs. Final HPWL (um) on the Y-axis. Each point represents one run with a specific knob setting.
            3. **Analyze:** Add this plot to your README.md and write a short analysis on the "Pareto frontier" you found. What settings give the best HPWL? What settings give the fastest runtime? What do you recommend as the best "default" setting for your flow?
    2. **Calculate HPWL:** After placement, the script must report the **Final Total HPWL** to the console. This should be the final, SA-optimized HPWL.
- **Visualization**:
    1. **Placement Density Heatmap:** A 2D histogram (heatmap) of the chip, where color intensity represents the density of *placed* cells.

        2. **Net Length Histogram:** A 1D histogram of all net HPWLs.
- **Deliverables:**
    1. `placer.py` (containing your implemented algorithm(s)).
    2. `build/[design_name]/[design_name].map` (e.g., `build/6502/6502.map`): The output file mapping logical instance names to physical slot names.
    3. Console print-out of Final Total HPWL for each design.
    4. build/[design_name]/[design_name]_density.png (Heatmap).
    5. build/[design_name]/[design_name]_net_length.png (Histogram).
    6. `sa_knob_analysis.png` (The HPWL vs. Runtime plot from Task 1.D).

<u>**Phase 3:** CTS & ECO Netlist Generation (Due: Nov 30)</u>

- **Objective:** Modify the netlist to instantiate the clock tree and tie-off unused logic.
- **Tasks:**
    1. Implement eco_generator.py:
        - **CTS:** Implement a simple **H-Tree or X-Tree** algorithm. It must find all *placed* DFFs (sinks) and all *unused* buffer/inverter cells (resources). It recursively finds the geometric center of sinks, "claims" the nearest available buffer, and updates the placement.map and netlist connections.
        - **Power-Down ECO:** "Claims" one sky130_fd_sc_hd__conb_1 (tie-low) cell. It must find *all* other unused logic cells and modify the netlist to tie their inputs to the conb_1 output.
    2. **Output:** The script must generate a complete, valid Verilog file: build/[design_name]/[design_name]_final.v.
- **Visualization** (`visualize.py cts`):
    1. **CTS Tree Plot:** Generate a plot of the DFFs (leaves) and chosen buffers (nodes), with lines connecting them to show the synthesized tree structure overlaid on the chip layout.
- **Deliverables:**
    1. eco_generator.py script.
    2. build/[design_name]/[design_name]_final.v (The new, complete Verilog netlist).
    3. build/[design_name]/[design_name]_cts_tree.png (CTS visualization).

<u>**Phase 4 & 5:** Routing, STA, & Full Flow (Due: Dec 7)</u>

- **Objective:** Route the design, extract parasitics, and perform signoff timing analysis.
- Tasks (Phase 4):
    1. **Implement make_def.py:** Generates `build/[design_name]/[design_name]_fixed.def`. This file *must* contain the `DIEAREA`, all `PINS` (+ FIXED), and *all* COMPONENTS from `fabric_cells.yaml` (both used and unused) as `+ FIXED`.
    2. **Implement rename.py:** Parses _final.v and .map. Renames *every* Verilog instance to match its assigned physical slot name. Outputs `build/[design_name]/[design_name]_renamed.v`.
    3. **Implement route.tcl:** A generic Tcl script for OpenROAD (using env vars like $::env(DESIGN_NAME)). It must
       execute: read_lef, read_liberty, read_def, read_verilog, global_route, detailed_route, extract_parasitics, and report_congestion.
- Tasks (Phase 5):
    1. **Write SDC Files:** For *each* provided design, write a `[design_name].sdc` file. (e.g., 6502 @ 25MHz = 40ns period). Define `create_clock`, `set_input_delay`, and `set_output_delay`.
    2. **Implement sta.tcl:** A generic Tcl script that loads the LIB, _renamed.v, .spef, and .sdc. It must run report_timing -setup -n 100, report_timing -hold -n 100, and report_clock_skew.
- Visualization:
    1. **Final Layout Screenshot:** Take a high-resolution screenshot of `_routed.def` in `klayout` or `openroad –gui`.
    2. **Congestion Heatmap:** Parse the `_congestion.rpt` to generate a congestion heatmap.
    3. **Slack Histogram (visualize.py sta):** Parse `_setup.rpt` and plot a 1D histogram of all endpoint slacks.

4. **Critical Path Overlay (visualize.py sta):** Parse the *worst* path from _setup.rpt, get the cell locations, and draw a bright red line on the layout connecting them.

- Deliverables:
  1. All `*.py`, `*.tcl`, and `*.sdc` files.
  2. All generated `build/[design_name]/` files (`.def`, `.spef`, `.rpt`, etc.).
  3. All visualization PNGs (_layout.png, _congestion.png, _slack.png, _critical_path.png).

6. Final Submission & Analysis (Due: Dec 10)

- Task 1: Automation with Makefile (Makefile):
  - You are required to create a Makefile to automate your entire flow. A Python script (e.g., using subprocess) is also acceptable, but a Makefile is preferred.
  - The Makefile must be runnable by passing the design name as a variable:

    ```
    make all DESIGN=6502
    ```

  - It must define targets for each phase, using file dependencies:
    - `all`: Depends on the final STA report.
    - `sta`: Depends on .spef and .sdc files.
    - `route`: Depends on _renamed.v and _fixed.def.
    - `eco`: Depends on _final.v.
    - place: Depends on _mapped.json.
    - `validate`: Runs the validator.
    - `clean`: Removes all build/ files.
  - This dependency-based build is crucial. A change to placer.py should only trigger a re-build from Phase 2 onward.
- Task 2: Final Report (`README.md`):
  - This report must analyze the performance of your *flow* across the entire regression suite.
  - Include your **SA Knob Analysis** plot (sa_knob_analysis.png) and your recommendation for the best "default" knob settings.
  - Include a **Comparison Dashboard** (Markdown table):

Design Name	**Util %**	**Placer Alg.**	**HPWL (km)**	**WNS (ns)**	**TNS (ns)**
6502	45%	Greedy+SA	...	...	...
UART	15%	Greedy+SA	...	...	...
FPU	80%	Greedy+SA	...	...	...
Design4	...	...	...	...	...
Design5	...	...	...	...	...

- **Analysis:** Analyze this dashboard. Does your placer scale well with high utilization? Do high-congestion designs (from `_congestion.png`) correlate with WNS? Use your `_critical_path.png` plots to justify your analysis of *why* specific designs fail timing.
- **Final Visual:** Include your best `_layout.png` (e.g., the 6502) in your report.

7. Grading (Tentative)

- **Flow Functionality & Automation (25%):** Does the Makefile run end-to-end on all designs? Are all file dependencies correct?
- **Algorithm Quality (20%):** Quality and correctness of your placer.py (which must include both Greedy and SA stages) and `eco_generator.py` (CTS).
- **SA Knob Analysis (10%):** Quality of the experimental analysis (Task 1.D) and the resulting plot/conclusions.

- **Checkpoint 1 Presentation (10%):** Clarity, correctness, and analysis of P1/P2 results.
- **Final Presentation (CP2) (15%):** Quality of full-flow analysis, STA results, and conclusions.
- **Final Report & Visualizations (10%):** Quality of `README.md` dashboard, visualizations, and written analysis.
- **GitHub Workflow (10%):** Quality of issues, PRs, commits, and group collaboration.
- Total = 100%

8. Bonus Challenge: Timing Closure Loop (Up to 20% Extra Credit)

Your flow will likely fail timing on complex designs. For **up to 20% bonus credit**, propose, *implement*, and *validate* an iterative fix to improve timing.

This is a significant, open-ended challenge. A successful bonus submission requires:

1. **Analysis:** A clear, data-driven analysis of *why* the design fails timing (using your STA reports, congestion maps, and critical path visualizations).
2. **Hypothesis:** A specific, actionable change to your flow (e.g., to the placer, CTS, or router).
3. **Implementation:** The code for your fix (e.g., a "timing-aware" mode for your placer).
4. **Validation:** "Before" and "After" data (especially the comparison dashboard and slack histograms) proving your fix improved the WNS/TNS.

Example: "My flow fails timing on the FPU. The `FPU_critical_path.png` shows a long, winding path. I will modify my flow to be timing-aware. After running `make sta DESIGN=FPU`, I will parse `build/FPU/FPU_setup.rpt` to find the top 100 critical nets. I will then *re-run `make place DESIGN=FPU`*, but this time, my `placer.py` will read this critical net list and apply a 10x weighting factor to their HPWL calculation, forcing them to be placed closer together."

Deliverable: A new "Bonus Challenge" section in your `README.md` detailing your analysis, implementation, and a "before/after" dashboard proving your fix worked.