ChatGPT

Full Adder ASIC Flow with OpenLane

This guide explains how to implement a simple 1-bit full adder using the OpenLane ASIC flow, mirroring the example SPM project structure. We cover the role of each design file, provide example contents (fully commented), and walk through all OpenLane commands from mounting the environment to obtaining a GDS layout. We also show how to view the final GDS in KLayout and provide a Python script (via KLayout's Python API) to extract layout features and compute a metal-density grid [1] [2]. Each VLSI flow step is summarized with its purpose and how the files and commands work together from Verilog to GDS [3] [4].

Design Folder Files and Their Roles

An OpenLane **design directory** typically contains the following key files:

- `config.json` – *Top-level design configuration.* This JSON file sets design-wide variables: `DESIGN_NAME` (unique project name), list of Verilog source files (`VERILOG_FILES`), technology (PDK) and standard cell libraries, clock settings, floorplan options, etc. All OpenLane steps use these settings. For example, a config.json might specify the clock period or disable clock-tree synthesis. Minimal required keys include the design name and Verilog sources [5].

- `run_config.json` – *(Optional) Run configuration.* In some flows, a `run_config.json` can override or extend `config.json` for a particular run (for example, choosing which steps to execute). In modern OpenLane, this is often not needed, as one can directly invoke steps via the flow commands.

- **Top-level Verilog (e.g. `full_adder.v`)** – *RTL source.* This file contains the Verilog description of the full adder module (with inputs, outputs, and the logic equations). It is the functional specification that OpenLane will synthesize. In the SPM example this was `spm.v`; here we use `full_adder.v`. It must match `DESIGN_NAME` in the config if that Verilog module is the top.

- `.sdc` **file (e.g. `full_adder.sdc`)** – *Static timing constraints.* This text file (using Synopsys Design Constraints format) specifies clock definitions and I/O timing constraints for STA (Static Timing Analysis) and implementation tools. For a purely combinational full adder, you may not have a clock, so the SDC can be empty or just contain comments. If you did have sequential elements, you would use `create_clock` commands here. OpenLane's synthesis and PnR steps will source this file if present [6].

- `pin_order.cfg` – *I/O pin placement (floorplan) configuration.* This text file tells the floorplanner which side of the chip each I/O pin (port) should be placed on. Its format uses sections like `#N`, `#E`, `#S`, `#W` (north, east, south, west) followed by pin name patterns. For example, to force pins `A, B, Cin` on the west side and `Sum, Cout` on the east side, you could write:

```
#W
A
B
Cin

#E
Sum
Cout
```

During floorplanning, OpenLane will fix I/O locations accordingly. (See example in [Newcomers Guide][55] with `#E`, `#S`, `#N` markers [2].) The `FP_PIN_ORDER_CFG` setting in `config.json` points to this file.

- **Other files (optional):** The SPM project also used things like a default `config.tcl` (legacy from OpenLane v1). In OpenLane v2 we use JSON instead. No other files are strictly required for a basic full adder.

Together, these files tell OpenLane *what* the design is (Verilog) and *how* to process it (config.json, constraints, pin placement).

Example File Contents

Below are example contents for a 1-bit full adder design. Comments (`//` or `#`) explain each setting:

```
// config.json: Design configuration for full_adder.
{
  "DESIGN_NAME": "full_adder",          // Unique design name (matches top-level
Verilog module name)
  "VERILOG_FILES": "dir::src/*.v",      // All Verilog source files in src/
  "PDK": "sky130A",                     // Use SkyWater 130nm PDK (adjust if
different)
  "STD_CELL_LIBRARY": "sky130_fd_sc_hd",// Standard cell library for logic
(high-density cell library)
  "CLOCK_TREE_SYNTH": false,            // Disable clock-tree (no registers in
combinational design)
  "CLOCK_PORT": null,                   // No clock port for combinational logic
  "FP_SIZING": "absolute",              // Use absolute core area sizing
  "DIE_AREA": "0 0 100 100",            // Core area (um). Adjust as needed for
utilization.
  "FP_IO_MODE": 1,                      // Random I/O spacing mode (1 =
equidistant)
  "FP_PIN_ORDER_CFG": "dir::pin_order.cfg"  // Use pin_order.cfg for I/O
placement
}
```

This `config.json` tells OpenLane the design name, where to find Verilog files, and how to treat clock and floorplanning. We disable clock synthesis since a combinational full adder has no clock. We set an arbitrary die area and point to `pin_order.cfg` for custom I/O pin placement [5].

```
// run_config.json: (Optional) per-run overrides. Not needed for basic flows.
// For example, you could override some settings here if desired.
{
  // Example: no overrides; leave empty or customize run steps.
}
```

```
// src/full_adder.v: 1-bit Full Adder
// Module computes Sum and Cout from inputs A, B, Cin.
`default_nettype none
module full_adder (
    input  wire A,   // First operand
    input  wire B,   // Second operand
    input  wire Cin, // Carry-in
    output wire Sum, // Sum output
    output wire Cout // Carry-out
);
    // Sum = A XOR B XOR Cin
    assign Sum  = A ^ B ^ Cin;
    // Cout = majority(A,B,Cin)
    assign Cout = (A & B) | (A & Cin) | (B & Cin);
endmodule
`default_nettype wire
```

The Verilog defines the full adder logic. Note the module name `full_adder` matches `DESIGN_NAME` (OpenLane will look for that). There are no flip-flops, so it is purely combinational.

```
# full_adder.sdc: Timing constraints (none needed here since no clock).
# If this were sequential, you'd put create_clock here.
# For example:
# create_clock -name clk -period 10 [get_ports clk]
```

The SDC file is empty (only comments) because there is no clock to constrain. For sequential designs, you would add clock definitions and input/output delays here. OpenLane will source this SDC during timing analysis if present.

```
# pin_order.cfg: Floorplan I/O placement
# Place inputs (A, B, Cin) on the west side, outputs (Sum, Cout) on the east.
#W
A
```

```
B
Cin

#E
Sum
Cout
```

This `pin_order.cfg` forces the inputs to appear on the west edge of the chip and outputs on the east, matching typical I/O connectivity. Each `#<dir>` line groups the following pin names to that side. (See [Newcomers Guide][55] for the format.) This helps the floorplanner place I/O pins in a controlled way, which is important for chip integration [2].

OpenLane Flow Commands

To run the flow and generate the layout, use the following commands (assuming a Linux host with Docker or Nix setup as per OpenLane instructions):

1. **Mount/Open the OpenLane container:**

   ```
   cd ~/OpenLane
   make mount
   ```

   This starts the OpenLane Docker/Nix environment and drops you into the container shell [7].

2. **Prepare the design:**
   Inside the container, create a new design directory and copy the files (config.json, src/full_adder.v, etc.) there. Then run:

   ```
   ./flow.tcl -interactive
   prep -design full_adder
   ```

   In interactive mode, `prep -design full_adder` initializes the design using `config.json`. You should see log messages like `[INFO]: Using configuration in 'designs/full_adder/config.json'` [8].

3. **Run synthesis:**

   ```
   run_synthesis
   ```

   This invokes Yosys to synthesize the Verilog into a gate-level netlist. (It uses clock settings or the lack thereof from the config.) After this step you get synthesis reports.

4. **Run floorplan:**

   ```
   run_floorplan
   ```

   This defines the core area and places I/O pins randomly or according to `pin_order.cfg`. It also places power rails. (Our `pin_order.cfg` will force the I/Os to the specified sides.)

5. **Run placement:**

   ```
   run_placement
   ```

   Cells from the netlist are placed inside the core. This also generates a placement report and prepares for routing.

6. **Run CTS (Clock Tree Synthesis):**

   ```
   run_cts
   ```

   Since we disabled clocks, this will effectively do nothing. In a sequential design, this would insert a clock tree. It must be run to proceed.

7. **Run routing:**

   ```
   run_routing
   ```

   This step uses OpenROAD to route all interconnects and generate the final circuit layout. After routing, the flow typically runs signoff steps (DRC, LVS).

8. **Run Magic & GDS export:**

   ```
   run_magic
   ```

   This invokes Magic to perform DRC and create the final GDSII. (OpenLane names this step "magic signoff"). The final GDS file will appear under `runs/RUN_*/results/signoff/full_adder.gds`.

Alternatively, you can run the entire flow non-interactively with one command (outside or inside Docker/Nix shell):

```
./flow.tcl -design full_adder
```

This executes all enabled steps in sequence. However, stepping through them interactively is useful for debugging. The commands above (`run_synthesis`, etc.) match those listed in the OpenLane user guide [8] [9].

Each step logs status. For example, during synthesis you'll see `[STEP 1] Running Synthesis`, and during routing `[STEP 4] Running Routing` etc. After each step the intermediate results and logs are saved in `designs/full_adder/runs/RUN_*/`.

Viewing the GDS Layout in KLayout

Once routing and signoff have completed, you can view the GDS layout in KLayout (a free GDSII viewer):

1. **Exit the container:** If you ran OpenLane inside Docker/Nix, exit back to your host shell.

2. **Locate the GDS file:** The final layout will be in the `results/signoff` folder of the last run. For example:

   ```
   ls ~/OpenLane/designs/full_adder/runs/*/results/signoff/*.gds
   ```

   You should see something like `full_adder.gds` and `full_adder.klayout.gds`.

3. **Open in KLayout:** Launch KLayout on your host and open the GDS:

   ```
   klayout ~/OpenLane/designs/full_adder/runs/RUN_*/results/signoff/
   full_adder.gds
   ```

   This displays the layout graphically. You can zoom and inspect the placed cells and routing. (You may need to select the correct PDK technology in KLayout's layer palette if it asks.) The layout you see corresponds to the Skywater 130nm layers of the standard cells and routing.

(*Tip:* KLayout can also open the Magic signoff GDS or a KLayout-custom GDS if provided. But the `full_adder.gds` here should be ready-to-view.)

Extracting Layout Features with KLayout Scripting

We can programmatically extract layout features (bounding boxes of transistors, wires, etc.) and compute a metal-density grid using KLayout's Python API (Py4K). Below is an example script adapted from the GDS Feature Extraction guide [10] [11]. It reads the GDS, iterates over specified layers, and writes out a CSV of object bounding boxes, as well as a CSV of metal density per grid cell:

```
# export_gds_features.py: Extract geometry and density from full_adder.gds using
KLayout Python API.
```

```python
import pya
import csv

# Open the GDS layout (change path to your full_adder.gds)
layout = pya.Layout()
layout.read("full_adder.gds")

cell = layout.top_cell()

# Define layers of interest (layer number, datatype) for N, P transistors and
metals
layers = {
    "N_active": layout.layer(1, 0),  # example layer numbers; use correct ones
for the PDK
    "P_active": layout.layer(2, 0),
    "metal1"  : layout.layer(3, 0),
    "metal2"  : layout.layer(4, 0)
    # (Adjust layer IDs based on the technology; 1,2 might be diff for Sky130)
}

# Prepare CSV for geometry features
with open("layout_features.csv", "w", newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["layer", "datatype", "area", "xmin", "ymin", "xmax",
"ymax"])
    for layer_name, layer_index in layers.items():
        # Get all shapes (polygons) on this layer in the top cell
        shapes = cell.each_shape(layer_index)
        for shape in shapes:
            if shape.is_box():
                box = shape.box()
                xmin, ymin = box.left, box.bottom
                xmax, ymax = box.right, box.top
                area = box.area()
            else:
                # For simplicity, use the bounding box of complex shapes
                bbox = shape.bbox()
                xmin, ymin = bbox.left, bbox.bottom
                xmax, ymax = bbox.right, bbox.top
                area = bbox.width() * bbox.height()
            writer.writerow([layer_index[0], layer_index[1], int(area), xmin,
ymin, xmax, ymax])

# Compute a simple metal density grid on metal layers
grid_size = 10_000  # 10 µm grid (units depend on technology units)
density = {}
# Initialize grid cells
for x in range(0, int(layout.dbu * 1000 * 1000), grid_size):  # example extents
```

```
    for y in range(0, int(layout.dbu * 1000 * 1000), grid_size):
        density[(x//grid_size, y//grid_size)] = 0

# Accumulate metal area per cell for metal1 and metal2 layers
for layer_name in ["metal1", "metal2"]:
    layer_index = layers[layer_name]
    shapes = cell.each_shape(layer_index)
    for shape in shapes:
        pts = shape.to_itype(layout.dbu).polygon[:]
# polygon points in integer DBU
        # Rasterize the polygon into grid cells (this is simplified)
        bbox = shape.bbox()
        xmin_cell = int(bbox.left / grid_size)
        xmax_cell = int(bbox.right / grid_size)
        ymin_cell = int(bbox.bottom / grid_size)
        ymax_cell = int(bbox.top / grid_size)
        # Add full area of box to each intersecting cell (overestimate)
        box_area = (bbox.right - bbox.left) * (bbox.top - bbox.bottom)
        for i in range(xmin_cell, xmax_cell+1):
            for j in range(ymin_cell, ymax_cell+1):
                density[(i,j)] += box_area

# Write density grid to CSV
with open("density_grid.csv", "w", newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["cell_x", "cell_y", "metal1_area", "metal2_area"])
    for (i,j), area in density.items():
        writer.writerow([i, j, area if layer_name=="metal1" else "", area if
layer_name=="metal2" else ""])
```

This script (saved as, e.g., `export_gds_features.py` inside the design run folder) does the following:

- Uses KLayout's `pya.Layout()` to read the GDS.
- Defines layer indices for N/P active regions and metal layers. *(Note: actual layer numbers/types depend on the Sky130 tech file; adjust as needed.)*
- Iterates over all shapes on each layer, recording each shape's bounding box and area into `layout_features.csv` [12] [11].
- Creates a uniform grid (10 µm cells here) and sums the metal area within each cell to estimate density. The (over)simplified grid write-out is saved to `density_grid.csv`.

To run this script after the OpenLane flow completes, use KLayout in batch mode (inside the container or on the host with KLayout installed):

```
klayout -b -r export_gds_features.py
```

This generates `layout_features.csv` and `density_grid.csv` in the current directory. If you ran KLayout inside Docker, you can copy them out (as shown in the reference) with Docker cp. For example:

```
docker ps
docker cp <container_id>:/openlane/designs/full_adder/runs/RUN_*/
layout_features.csv ~/Downloads/
```

(Here `<container_id>` is the ID of the running OpenLane container.) These CSV files can then be opened in Python or Excel for analysis [13].

Summary of Flow Steps

Each step of the OpenLane ASIC flow has a specific purpose in turning RTL into a layout [3]:

- **Synthesis:** Converts the Verilog RTL into a gate-level **netlist** of standard cells. Tools like Yosys map high-level logic into library gates. The output is a synthesized netlist that will be physically implemented [3].
- **Floorplanning:** Defines the chip **core area** and I/O regions. The core area (`DIE_AREA`) is set (e.g. 100×100 µm above) and I/O pins are (by default randomly or by `pin_order.cfg`) placed around the periphery. Power rails and well taps are also inserted. Floorplanning sets up the canvas for placement.
- **Placement:** The netlist cells are **placed** inside the core area. The placer positions cells to minimize wire length and meet any constraints (like distance between certain cells). At this point, a rough placement is obtained.
- **Clock-Tree Synthesis (CTS):** If the design had registers and a clock, CTS would build a balanced tree of clock buffers. In our full adder (no registers), CTS is effectively skipped. In general, CTS ensures the clock reaches all flip-flops with minimal skew.
- **Routing:** All nets are **routed** using routing layers. OpenROAD's router connects pin-to-pin, creates metal wires, vias, etc. The output of routing is a complete layout (with all metal shapes) and final parasitic extraction for timing.
- **Signoff (DRC/LVS/GDS Export):** Magic (and OpenROAD checks) verify design rules (DRC) and perform layout-vs-schematic checks (LVS). Finally, a GDSII file is generated (`full_adder.gds`), which is the layout "blueprint" for fabrication. This GDS reflects the final placement and routing of the full adder.

In summary, **config.json** and related files describe the design and constraints, and each `run_*` command in OpenLane executes one stage of the flow using those settings. Together, they automate the path from your Verilog code to a finalized GDS layout. For example, synthesis uses the HDL from `full_adder.v`, placement respects `pin_order.cfg`, and routing ensures timing (though unimportant here) as per any

SDC constraints. Each step outputs reports and intermediate files, culminating in the GDS which can be viewed with tools like KLayout.

Sources: OpenLane documentation and examples [5] [3] (for design file structure and flow overview), the OpenLane tutorial and Newcomers guide [4] [2] (for config and pin file formats), and the GDS feature-extraction example [1] [12] (for viewing GDS and Python scripting). These sources illustrate how the config files and commands work together in the RTL-to-GDS flow.

---

[1] [7] [8] [9] [10] [11] [12] [13] Gds Feature Extraction.pdf
file://file-GYJSxSUH1QTpkgSnrH3GXC

[2] [3] Newcomers - OpenLane Documentation
https://openlane2.readthedocs.io/en/latest/getting_started/newcomers/index.html

[4] [5] Design Configuration Files - OpenLane Documentation
https://openlane2.readthedocs.io/en/latest/reference/configuration.html

[6] Configuration Variables - OpenLane Documentation
https://armleo-openlane.readthedocs.io/en/merge-window-4/reference/configuration.html