Basics of Characterization files

Anil Kumar Gundu

November 2025

Contents

Chapter 1

Important Files and their Uses

Memory Characterization Files in VLSI

1. **Liberty (.lib) files**
   - **Purpose:** Contains timing, power, and functional characterization data of memory cells (read-/write delays, setup/hold, leakage, etc.).
   - **Generated by:** Tools like *Synopsys SiliconSmart*, *Cadence Liberate*, or *Altos Liberate*.
   - **Used in:**
     - Static Timing Analysis (STA) — e.g., *PrimeTime*, *Tempus*
     - Synthesis — e.g., *Design Compiler*
     - Power analysis — e.g., *PrimePower*
   - **Contents:**
     - Cell leakage and dynamic power
     - Read/write access times
     - Setup/hold margins
     - Constraints on address/data/control pins
2. **Verilog (.v) / VHDL behavioral model**
   - **Purpose:** Describes the functional behavior of the memory for RTL simulation.
   - **Generated by:** Memory compiler (e.g., *TSMC*, *GF*, or *Arm Artisan Memory Compiler*).
   - **Used in:**
     - Logic simulation (*VCS*, *NC-Sim*, *Xcelium*, *ModelSim*)
     - Functional verification (pre-layout)
3. **SPICE (.sp, .scs, .ckt) netlist**
   - **Purpose:** Describes the transistor-level circuit of the memory for analog-level simulation.
   - **Used in:**
     - Pre-characterization (*HSPICE*, *Spectre*)
     - Corner and Monte Carlo analysis
     - Sense amplifier and bitline analysis
   - **Generated by:** Memory design team or extracted using *Calibre xRC*, *StarRC*, etc.
4. **LEF (.lef) file (Layout Exchange Format)**
   - **Purpose:** Contains abstract layout information (cell boundary, pin positions, blockages).
   - **Used in:**
     - Place and Route (P&R) tools (*Innovus*, *ICC2*)
     - Floorplanning and DRC checks

- **Note:** Does not contain real geometry — only outlines and metadata for integration.

5. **GDSII (.gds) / OASIS (.oas) file**
   - **Purpose:** Full physical layout of the memory block.
   - **Used in:**
     - Tape-out and fabrication
     - LVS/DRC verification (*Calibre*, *Pegasus*)
   - **Generated by:** Layout team or memory compiler.
6. **CDL / LVS netlist (.cdl)**
   - **Purpose:** Device-level netlist for LVS (Layout vs. Schematic) verification.
   - **Used in:** *Calibre LVS*, *ICV LVS*, etc.
   - Matches extracted layout with schematic.
7. **.db file (Compiled Liberty)**
   - **Purpose:** Binary-compiled version of .lib used by Synopsys tools for fast read access.
   - **Used in:** *Design Compiler*, *PrimeTime*.
   - **Generated by:** *lc_shell* (read_lib, write_lib).
8. **.tf / .itf (Technology files)**
   - **Purpose:** Defines technology-specific parameters like parasitics, capacitances, and process corners used during characterization.
   - **Used in:** SPICE characterization and extraction tools.
9. **.sdf (Standard Delay Format)**
   - **Purpose:** Annotates post-layout timing delays into simulations.
   - **Used in:** Gate-level simulation (GLS).
   - **Generated by:** STA tools.
10. **Characterization scripts**
    - **File types:** .tcl, .py, .csv, .cfg
    - **Purpose:** Automate SPICE simulations, corner sweeps, and extraction of delay/power data for .lib generation.

1.0.1 Typical File Set from a Memory Compiler

A typical memory compiler (e.g., TSMC 22nm SRAM compiler) usually provides:

```
<mem_name>.lib     → timing/power model
<mem_name>.v       → behavioral Verilog
<mem_name>.lef     → physical abstract
<mem_name>.gds     → full layout
<mem_name>.cdl     → LVS netlist
<mem_name>.db      → compiled .lib
README.txt         → integration guide
```

1.1 Understanding the .lib File in Memory Characterization

1.1.1 What is a .lib file?

A **.lib file** (Liberty file) is a text-based standard that describes Timing, Power, Area, Functionality, and Operating conditions for cells or memory blocks in a technology library. For instande if it is SRAMs, it captures: Read/write access times, Setup/hold constraints, Power consumption, and Control pin functions across different PVT corners. It is essentially the *soul* of timing analysis and synthesis.

1.1.2 Basic Structure of a .lib File

A simplified example looks like:

```
library (sram_1rw_32x1024) {
    technology( cmos );
    delay_model : table_lookup;
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1mA";
    pulling_resistance_unit : "1kohm";

    operating_conditions (tt_1p0v_25c) {
        process : 1;
        voltage : 1.0;
        temperature : 25;
    }

    cell (SRAM_1RW_32x1024) {
        area : 50000;  /* in um^2 */
        cell_leakage_power : 1.2e-03;

        /* -------- Pin Definitions -------- */
        pin (CLK) {
            direction : input;
            clock : true;
            capacitance : 0.025;
        }

        pin (ADDR[9:0]) {
            direction : input;
            capacitance : 0.05;
        }

        pin (DIN[31:0]) {
            direction : input;
            capacitance : 0.08;
        }

        pin (DOUT[31:0]) {
            direction : output;
            max_capacitance : 0.12;
            function : "(!CEN & !WEN) ? memory_read_data : 'bz'";
        }

        /* -------- Timing Arcs -------- */
        timing() {
            related_pin : "CLK";
            timing_sense : positive_unate;
            cell_rise (delay_template) {
                index_1 ("0.01, 0.1, 0.2, 0.3");
                index_2 ("0.01, 0.1, 0.2, 0.3");
                values ( \
                    "0.3, 0.4, 0.45, 0.48", \
                    "0.32, 0.42, 0.47, 0.50", \
                    "0.35, 0.44, 0.49, 0.52", \
                    "0.36, 0.45, 0.50, 0.53");
            }
        }
```

```
        /* -------- Power -------- */
        internal_power() {
            related_pin : "CLK";
            rise_power (delay_template) {
                index_1 ("0.01, 0.1, 0.2");
                values ("0.08, 0.12, 0.14");
            }
        }
    }
}
```

1.1.3 Important Sections Explained

library(...) – Defines one technology library (one memory or cell set).Inside this block are the units, models, and operating conditions.
cell(...) – Each memory instance (e.g., `SRAM_1RW_32x1024`) has its own cell definition containing Area, Cell leakage power, Pin definitions, and Timing and internal power data.

1.1.4 Pin-Level Details

Each `pin(...)` defines its direction, capacitance, function, and timing relationships.

Pin	Type	Role
CLK	Input	Clock for read/write cycle
CEN	Input	Chip enable (active low)
WEN	Input	Write enable (active low)
A[9:0]	Input	Address bus
DIN[31:0]	Input	Data input
DOUT[31:0]	Output	Data output
VDD, VSS	Power	Supplies

1.1.5 Timing Arcs (Core Concept)

For every input–output path, a timing arc is defined:

- From CLK → DOUT (read access delay)
- From CLK → internal write (write delay)
- From A, WEN, CEN → CLK (setup/hold constraints)

Example:

```
timing() {
    related_pin : "CLK";
    timing_sense : positive_unate;
    cell_rise (delay_template) {
        index_1 ("input transition");
        index_2 ("output load");
        values ("delay matrix");
    }
}
```

1.1.6 Power Modeling

Three main power types are defined:

Type	Purpose
internal_power	Dynamic power per transition
leakage_power	Static leakage per cell
switching_power	Dynamic output switching cost

For SRAMs, separate models exist for read, write, and standby modes.

1.1.7 Operating Conditions and Corners

Typical operating conditions:

```
operating_conditions (ss_0p9v_125c)
operating_conditions (tt_1p0v_25c)
operating_conditions (ff_1p1v_0c)
```

Each corresponds to a separate file:

```
sram_1rw_32x1024_ss_0p9v_125c.lib
sram_1rw_32x1024_tt_1p0v_25c.lib
sram_1rw_32x1024_ff_1p1v_0c.lib
```

Used for **multi-corner STA**.

Chapter 2

How to Invoke Opensource Tools

2.1 Icarus Verilog + GTKWave Workflow

The basic workflow to compile, simulate, and view waveforms for a Verilog module using **Icarus Verilog** and **GTKWave** is as follows:

2.1.1 Compile the Verilog code

Command:

```
iverilog -o <output_executable> <source_files>
```

Explanation:

- `iverilog` : Icarus Verilog compiler
- `-o <output_executable>` : Name of the compiled simulation executable
- `<source_files>` : List of Verilog files (module + testbench)

Example:

```
iverilog -o dff_tb dff_sync_reset.v tb_dff_sync_reset.v
```

Here, `dff-sync-reset.v` is the DFF module and `tb-dff-sync-reset.v` is the testbench. The output executable is *dff_tb*.

2.1.2 Run the simulation

Command:

```
vvp <output_executable>
```

Explanation:

- `vvp` : Runs the compiled simulation
- Generates a `.vcd` waveform file if `$dumpfile` and `$dumpvars` are used in the testbench

Example:

```
vvp dff_tb.vcd
```

2.1.3 View the waveform

Command:

```
gtkwave <vcd_file>
```

Explanation:

- `gtkwave` : Opens GTKWave waveform viewer
- `<vcd_file>` : Waveform file generated by simulation

Example:

```
gtkwave dff_sync_reset.vcd
```

2.1.4 Full Example Workflow

```
# 1. Compile
iverilog -o dff_tb dff_sync_reset.v tb_dff_sync_reset.v

# 2. Run simulation
vvp dff_tb

# 3. View waveform
gtkwave dff_sync_reset.vcd
```

2.2 Yosys tool

2.2.1 Interactive Yosys Command

- Type **yosys**. Starts the Yosys interactive shell, allowing you to run commands step by step. Useful for debugging and inspecting messages during synthesis.
- Once in the shell, run your script with **script file_name.tcl** to generate the synthesized netlist.

2.2.2 Useful Yosys Commands

- **yosys -p "synth_techlib -top top_module"**
  Runs Yosys synthesis on the specified top module using the given technology library. Generates a gate-level netlist and reports basic statistics.
- **yosys -p "read_verilog design.v"**
  Reads the Verilog RTL file into Yosys. Prepares the design for synthesis and further processing.
- **yosys -p "hierarchy -check -top top_module"**
  Checks the module hierarchy and ensures the top module exists. Reports any missing modules or hierarchy issues.
- **yosys -p "proc; opt; clean"**
  Performs RTL-to-gate translation and optimizations. Simplifies logic, removes unused wires and cells, and cleans the netlist.
- **yosys -p "techmap"**
  Maps generic logic to technology-specific cells. Required before mapping to your target standard-cell library.
- **yosys -p "abc -liberty library.lib"**
  Maps the logic to a given standard-cell library using ABC. Performs technology mapping and basic timing-aware optimizations.
- **yosys -p "dfflibmap -liberty library.lib"**
  Maps flip-flops in the design to library-specific DFF cells. Ensures that registers are correctly implemented in the gate-level netlist.

- **yosys -p "write_verilog -noattr synth.v"**
  Writes the synthesized gate-level netlist to a Verilog file. Removes unnecessary Yosys attributes while preserving module names.

- **yosys -h**
  Displays all available Yosys command-line options. Useful to explore flags, help, and script modes.

- **yosys -v**
  Shows the installed Yosys version. Useful for confirming compatibility with scripts and libraries.

- **yosys -p "stat"**
  Reports design statistics such as number of cells, wires, and levels of logic. Helps analyze complexity and resource usage.

- **yosys script.tcl**
  Runs a series of Yosys commands from a Tcl script. Useful for automating synthesis and optimization flows.

2.3 OpenSTA tool

1. **Purpose of OpenSTA**
   - Performs Static Timing Analysis (STA) on synthesized or placed-and-routed designs.
   - Verifies setup and hold timing without running simulations.
   - Checks timing paths using constraints (SDC).
   - Ensures timing closure before tape-out.
2. **Input Files Required**
   - `.lib` – Standard cell timing library.
   - `.v` – Gate-level netlist.
   - `.sdc` – Timing constraints file.
   - `.spef` (optional) – Parasitic extraction after routing.
   - `.sdf` (optional) – Delays for back-annotation.
3. **Basic OpenSTA Flow**
   - Read libraries: `read_liberty`
   - Read Verilog netlist: `read_verilog`
   - Read timing constraints: `read_sdc`
   - Read parasitics: `read_spef` (optional)
   - Update timing graph: `update_timing`
   - Report timing: `report_checks`, `report_timing`
4. **Key Concepts in STA**
   - **Setup Timing** – Ensures data arrives before the clock edge.
   - **Hold Timing** – Ensures data does not arrive too early.
   - **Slack** – Difference between required time and arrival time.
   - **WNS** – Worst Negative Slack (setup metric).
   - **TNS** – Total Negative Slack (sum of all violations).
   - **Clock Skew** – Difference in clock arrival times.
   - **Critical Path** – Path with the worst slack.
5. **Important Commands**
   - `report_timing` – Shows detailed critical timing path.

- `report_checks` – Shows setup/hold violations.
- `report_clock_tree` – Info about clock propagation.
- `report_design` – Summary of design statistics.
- `report_collection` – Inspect groups of objects.
- `set_false_path` – Declare paths not to be timed.
- `set_max_delay` / `set_min_delay` – Override default timing.

6. **How STA Checks Timing**
   - Builds a directed timing graph of all paths.
   - Traverses paths from clock startpoints → endpoints.
   - Computes:
     - Data arrival time
     - Data required time
     - Slack = Required Time – Arrival Time
   - Reports the worst violating paths.
7. **When OpenSTA is Used in OpenROAD Flow**
   - After synthesis (with ideal clocks)
   - After placement (initial timing estimation)
   - After CTS (timing with clock tree)
   - After routing (final STA with parasitics)
   - Before tape-out (sign-off level check)

2.4 OpenROAD Tool

OpenROAD is a full digital physical design (PD) tool that performs almost every backend step after synthesis.

When OpenSTA is Used in OpenROAD Flow :

- After synthesis (with ideal clocks)
- After placement (initial timing estimation)
- After CTS (timing with clock tree)
- After routing (final STA with parasitics)
- Before tape-out (sign-off level check)

2.4.1 What OpenROAD does & Full Digital Flow Steps

1. **Floorplanning**
   - Set the chip/core size
   - Place IO pins
   - Create power grid (VDD/VSS)
   - Place macros (if any)
2. **Placement**
   - Place all standard cells roughly (global placement)
   - Adjust them precisely (detailed placement)
   - Legalize positions (no overlaps)
   - Fix any cell overlaps

3. **Clock Tree Synthesis (CTS)**
    - Insert clock buffers/inverters
    - Build the clock tree
    - Reduce clock skew
    - Fix transition/capacitance issues
4. **Routing**
    - Plan routing paths (global routing)
    - Draw exact wires on metal layers (detailed routing)
    - Connect everything with final interconnects
5. **Timing Analysis**
    - Uses OpenSTA inside OpenROAD
    - Check setup/hold timing
    - Insert buffers to improve timing
    - Resize cells for timing closure
6. **Optimization**
    - Fix DRC violations
    - Fix antenna violations
    - Reduce wire length
    - Add buffers or resize cells
    - Improve clock quality
7. **GDSII Generation**
    - Produce final layout (GDSII)
    - Output ready for fabrication (tape-out)

2.5 Useful OpenROAD Commands

- **openroad -gui -log file_name.log file_name.tcl**
  Opens the OpenROAD GUI, runs the given Tcl script automatically, and saves all logs into a file. Best for debugging design flow visually and through logs.
- **openroad -verbose -log detailed.log flow.tcl**
  This command is strongly recommended while doing floorplanning and placement. Runs OpenROAD with verbose output for detailed analysis. Here we get more details and more step-by-step procedure so that it will be helpful for easy debugging. Useful when debugging CTS, routing, or timing issues.
- **openroad -gui**
  Launches the OpenROAD graphical interface. Allows interactive inspection of floorplan, placement, routing, and timing.
- **openroad flow.tcl**
  Runs the specified Tcl script in terminal mode without GUI. Used for fast execution and automated flows.
- **openroad -gui flow.tcl**
  Opens the GUI and automatically executes the Tcl script. Useful for viewing each design stage while the script runs.

- **openroad -log run.log flow.tcl**
  Executes the flow script and saves all outputs to a log file. No GUI here. Helpful for debugging warnings and errors.

- **openroad flow.tcl > run.log 2>&1**
  Redirects all terminal output and errors to a single log file. Works on any Linux shell even without using the `-log` flag.

- **openroad -gui; source flow.tcl**
  Starts the GUI first and loads the script manually. Useful for executing commands step-by-step during debugging.

- **openroad -h**
  Displays all available OpenROAD command-line options. Useful to check supported flags and usage.

- **openroad -version**
  Shows the currently installed OpenROAD version. Helps verify compatibility with PDKs or tutorials.

Chapter 3

TCL Commands and Explanation

3.1 ABC tool in Yosys :

ABC is a tool. A separate software program developed at UC Berkeley. Full name: "Berkeley ABC". It is included inside Yosys, but it is actually an independent tool. ABC is used for logic optimization and technology mapping.

- ABC takes your logic (from Yosys).
- Optimizes it (reduces gates, improves speed).
- Converts it into real standard-cell gates from your library (.lib)

3.2 Role of Yosys and ABC in Technology Mapping :

Yosys reads the .lib file only to understand what kinds of cells, flip-flops, and reset types are available in the technology, so it can generate a valid generic netlist that ABC can later map correctly. Yosys itself does not perform technology mapping; it only produces a technology-independent netlist using generic cells like *and*, *or*, dff, etc. ABC is the tool that actually uses the .lib to map this generic logic into real standard-cell gates and optimize it. In short: Yosys reads the .lib to know what is allowed; ABC uses the .lib to implement the final mapped design. The following four commands form the minimum mandatory setup in almost every Yosys synthesis flow for ASIC/standard-cell design :

- **read_libraries**
  read_libraries is a Yosys command used in ABC (the logic optimization + mapping tool Yosys uses). It loads the cell library files (usually .lib files) so that ABC knows: what gates the technology has (NAND, NOR, INV, DFF, etc.), their timing, their area, their drive strengths, their delays etc...
- **read_liberty**
  read_liberty is a Yosys command used to load a .lib file (standard-cell library) into Yosys itself. Yosys reads the .lib file only to understand what kinds of cells, flip-flops, and reset types are available in the technology, so it can generate a valid generic netlist.
- **link_design**
  link_design in Yosys resolves all module and cell references across multiple RTL files. It ensures every module instantiation points to the correct definition and merges libraries and RTL into a single coherent design. This makes the design ready for synthesis or ABC mapping without errors due to missing or unlinked modules.
- **read_sd**
  read_sdc in Yosys reads a Synopsys Design Constraints (SDC) file, which contains timing and clock constraints for your design. This file allows Yosys (and ABC) to understand clock definitions, input/output delays, and setup/hold requirements so that synthesis and optimization can meet the desired timing.

Flow : Verilog → Yosys → generic logic → ABC → tech-mapped logic → GDS flow

3.3 Chip Planning, Placement, Routing, and Clock Synthesis

3.3.1 Floor Planning

Floorplanning decides where the core of your chip will be placed inside the die, how much space the core and IOs take, and setting up the grid/rows for standard cells.

- ```
  initialize_floorplan -site $site \
                       -die_area $die_area \
                       -core_area $core_area
  ```

- **initialize_floorplan** sets up the chip's initial layout by defining the die boundary (-die_area), the core region for standard cells (-core_area), and the placement grid (-site) based on the standard cell site. site is defined in the .var file in technology.

- A site defines the placement unit for standard cells in a row. For example site with height 10 μm and width 1.2 μm, the row height is 10 μm, and each horizontal slot is 1.2 μm wide. Standard cells snap to these sites: small cells occupy one site, while larger cells span multiple sites. The width of 1.2 μm is the placement unit, not spacing, and rows are laid out horizontally according to the row_type.

3.3.2 Buffer Removal

Buffers and inverters are usually inserted during synthesis or technology mapping for: Strength matching (driving large loads), Signal routing, Timing fixes. But sometimes, they are redundant — e.g., a buffer driving another buffer or an inverter whose effect cancels out.

- `remove_buffers`

- **remove_buffers :** This command automatically deletes unnecessary buffers and inverters to simplify the netlist, reduce area, and improve optimization, while keeping the design functionally correct.

3.3.3 I/O Placement

place_pins is an OpenROAD command used during physical design, specifically IO placement - deciding where input/output pins go on the die boundary.

```
place_pins -random -hor_layer $io_placer_hor_layer -ver_layers $io_placer_ver_layer
```

- **-random :** Pins (inputs/outputs) are placed randomly along the core boundary. Alternative options: you could use -spread (spread pins evenly) or -manual (user specifies positions).

- **-hor_layer $io_placer_hor_layer :** Specifies the horizontal metal layer to use for IO routing. Example: M2 → horizontal wires for connecting pins.

- **-ver_layers $io_placer_ver_layer :** Specifies the vertical metal layers to use for IO routing. Example: M3 M4 → vertical wires for connecting pins to the core.

3.3.4 Global Placement of Standard Cells

global_placement -density $global_place_density This performs global placement of standard cells (small logic cells like AND, OR, DFF). Global placement is a rough placement stage where cells are spread across the core to avoid congestion before detailed placement.Always used before macro placement and detailed placement.

```
global_placement -density $global_place_density
```

- **-density :** sets the target utilization of the placement rows.

- **global_place_density :** Defined in the .var file.The placer will try to fill with the percentage defined in global_place_density (say for exampel 80) of the area with cells. Remaining area will be kept empty for routing tracks, buffers, upsizing, etc.
```

3.3.5 Macro Placement

This step places macros such as: SRAM blocks, ROM, PLL, Hard IPs, or any big pre-designed blocks. Macros are large and can block routing, so they need special spacing. Macro placement must come after global placement because global placement first spreads all standard cells. Then macros are placed with halo and channel rules; if macros are placed first, they block the proper spreading of standard cells.

```
macro_placement -halo $macro_place_halo -channel $macro_place_channel
```

- **-halo :** gives extra space around each macro.
- **macro_place_halo :** Defined in the .var file. set macro_place_halo 2.0 leave 2 µm of empty space around all sides of the macro.
- **-channel :** Defined in the .var file. set macro_place_channel 1.5. Keep 1.5 µm space between two macros for routing wires

3.3.6 Tapcell Insertion

Tap cells tie wells to VDD/VSS to prevent latch-up, while endcap cells protect the boundaries of standard-cell rows. Some PDKs also require corner tap cells for proper well and boundary coverage.

- **eval tapcell $tapcell_args :** runs the tapcell insertion command using the arguments stored inside the variable $tapcell_args stored in .var file.

```
Example :
set tapcell_args "-distance 14 -tapcell_master TAP -endcap ENDCAP"
eval tapcell $tapcell_args
```

3.3.7 Power Delivery Network Generator

Power Delivery Network Generator is a tool/command in OpenROAD that automatically creates the power grid for the chip. It generates VDD and VSS (power/ground) straps, rails, rings, and vias across the core and around macros. Builds power rails for standard cells (usually Metal1). Creates power stripes/straps on higher metal layers (M3/M4/M5 etc.) for uniform power distribution. Adds power rings around the core or macros (if configured). Automatically inserts vias between metal layers so the grid is electrically connected. Ensures IR-drop safety and follows PDK design rules for spacing and widths.

```
pdngen
```

3.3.8 Set RC for SIGNAL nets

When doing placement, routing, CTS, and timing analysis, the tool must know the **resistance (R) and capacitance (C)** of wires on each metal layer. This is because **wire delay = R × C**, which directly affects timing, skew, clock insertion delay, setup/hold paths, buffer insertion, and routing congestion. By specifying which metal layers will be used for signal and clock wires, the tool can **accurately estimate delays, balance the clock tree, optimize placement, and ensure correct timing** across the design.

```
 Commands :
set_wire_rc -signal -layer M2
set_wire_rc -clock -layer M4
set_dont_use $dont_use  (you can define which layes to not use also through .var file)
```

3.3.9 Estimate paracitics

Tells the tool to calculate wire resistance (R) and capacitance (C) based on the current cell placement (not the final routed wires). It uses cell locations and approximate interconnect lengths between pins to calculate R and C. This gives an early timing estimate before detailed routing is done, helping guide placement and optimization. So basically: "use placement info to predict wire parasitics."

```
 Commands :
estimate_parasitics -placement
```

3.3.10 Design Repair and Tie Cell Optimization

repair_design fixes small timing/slew/capacitance issues automatically to meet margins.repair_tie_fanout ensures tie0 and tie1 cells are spaced properly, reducing congestion and DRC problems. These following commands automatically fix small design violations (timing, slew, capacitance, tie cell spacing) inside OpenROAD

```
 Commands :
repair_design -slew_margin $slew_margin -cap_margin $cap_margin
repair_tie_fanout -separation $tie_separation $tielo_port
repair_tie_fanout -separation $tie_separation $tiehi_port
```

Chapter 4

Key Points - Opensource Synthesis

4.1 Important Point: Memory Handling in Full Digital Flow

Important Reminder: After RTL synthesis with Yosys, the synthesized netlist must **not** contain any `reg` arrays. If you still see entries such as:

```
\centering
reg [7:0] mem [15:0];
```

in the synthesized netlist, it implies the following:

- The memory was **not** converted into flip-flops from the standard-cell library.
- The RAM is still behavioral RTL, not real hardware.
- OpenROAD, OpenSTA, and PnR tools will fail or throw syntax errors.

Common Issue: Your synthesized netlist contains behavioral memory like:

```
reg [7:0] rem_memory_temp [15:0];
```

This means Yosys did **not** convert the memory into real hardware structures (flip-flops + mux + logic).

Effect of This Problem

- OpenSTA reports syntax errors.
- PnR tools cannot place or route these constructs.
- Netlist is not gate-level.
- Timing analysis becomes impossible.
- Results are invalid and the flow breaks.

Solution: Mandatory Yosys Passes

Use the following three Yosys passes immediately after `proc`:

```
memory          // detect the memory
memory_dff      // convert memory into flip-flops
memory_map      // build address logic (decoders + muxes)
```

These are required for any RAM written as:

```
reg [WIDTH-1:0] mem [0:DEPTH-1];
```

Explanation of Each Command

1. **memory**
   - Detects reg arrays in RTL.
   - Converts them into Yosys `$mem` blocks.
   - Meaning: "A RAM was found in the code."
2. **memory_dff**
   - Converts `$mem` into `$dff` flip-flops.
   - Meaning: "RAM is mapped into real D-flip-flops."
3. **memory_map**
   - Builds address decode logic and multiplexers.
   - Generates real gate-level logic for reads/writes.
   - Meaning: "Flip-flops are wired like a real RAM."

Result After Running These Passes

- RAM is converted into standard-cell flip-flops.
- Sky130 logic cells are inserted.
- Proper decoders and mux logic are created.
- No behavioral memory remains.
- OpenSTA works cleanly.

Required Yosys Sequence

```
proc
memory
memory_dff
memory_map
opt
techmap
dfflibmap -liberty sky130_fd_sc_hd__tt_025C_1v80.lib
abc       -liberty sky130_fd_sc_hd__tt_025C_1v80.lib
clean
write_verilog -noattr synth_memory.v
```

One-Line Summary

If your synthesized netlist still contains a reg array, you did not run `memory` + `memory_dff` + `memory_map`**.**