Leibniz Institute for high performance microelectronics

SRLatch Manual

Self-Paced Design of a StrongARM Comparator with Open-Source Tools

This document expands the workshop presentation into a self-paced manual for designing, simulating, and preparing the layout of a dynamic comparator. The focus is the StrongARM latch, using the same terminology, figures, and open-source workflow as the slide deck.

Author: Krzysztof Herman
Event: Analog Academy
Contact: herman@ihp-microelectronics.com
Date: June 21, 2026

Contents

1 Purpose and Scope

This manual is intended as a companion to the workshop presentation *Strong ARM Latch - Workshop*. The presentation provides the compact teaching path. This document provides the long-form path: what to build, why it is built that way, what to inspect in simulation, and what good intermediate results should look like.

The objective is not only to draw a comparator but to learn a repeatable analog design flow that starts from architecture and ends with a layout-ready implementation. The manual combines three sources of guidance:

- the existing workshop slide deck and its figure set,
- the StrongARM design reasoning summarized by Behzad Razavi in *The Design of a Comparator*,
- the practical tool flow already used in this project: xschem, ngspice, klayout, and magic.

The result is a self-paced path for a learner who wants to understand comparator behavior rather than only reproduce a netlist. By the end of the manual, the learner should be able to:

- explain how a StrongARM latch makes a decision,
- capture a clean comparator schematic in xschem,
- create and run transient, mismatch, process-variation, and kickback-oriented testbenches,
- understand the main trade-offs among offset, speed, metastability, and power,
- prepare a symmetry-aware layout that is suitable for DRC, LVS, and extraction.

This is a design manual, not a push-button recipe. The important outcome is understanding which device pair dominates which error mechanism and how each design or layout decision moves the comparator in speed, offset, noise, or robustness.

2 Design Targets and Project Constraints

Razavi frames comparator design around a small set of decisive parameters: input offset, speed, power, metastability, kickback noise, and input-referred noise. Those same parameters are the backbone of this manual.

For the self-paced design exercise, use the following targets as the conceptual reference point:

- input-referred offset in the low-millivolt range,
- hundreds-of-megahertz-class dynamic operation as the architectural ambition,
- no static current in the latch core during the reset state,
- rail-to-rail digital-friendly output after buffering or latching,
- low enough kickback and metastability risk to support system-level use.

The workshop-specific implementation constraints remain equally important because they define what a portable and verifiable cell means in this project:

Technology boundary

- Target both **SG13G2** and **CMOS5L** where practical.
- Use only low-voltage MOS devices.
- Keep routing inside `Metal1-Metal3`.
- Use local well and substrate ties such as `ptap1` and `ntap1`.
- Add guard-ring related protection elements such as `dantenna` and `dpantenna` where required by the process flow.

These restrictions are educationally useful because they prevent the design from hiding behind process-specific conveniences. The comparator must work because the architecture, sizing, symmetry, and verification flow are sound.

3 Open-Source Comparator Design Flow

The project uses a conventional analog loop: schematic capture, simulation, layout, verification, and iteration. The tools are open-source, but the engineering discipline is the same as in any industrial flow.

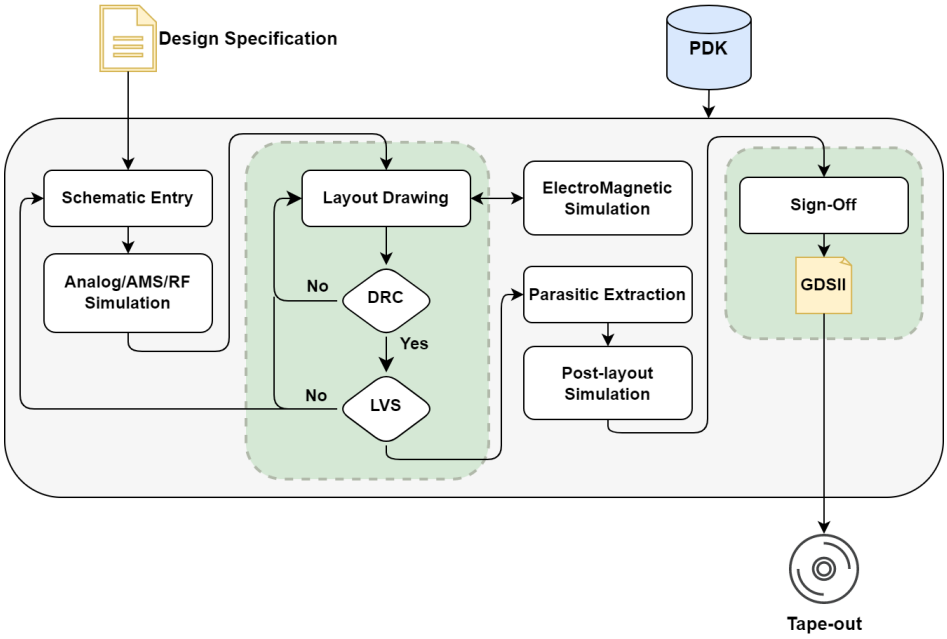

Fig. 1: Generic analog design loop used throughout the workshop and this manual.

Tool roles

- **xschem**: schematic capture, hierarchy definition, symbol-level organization, and testbench entry.
- **ngspice**: DC, transient, Monte Carlo style mismatch runs, process variation studies, and measurement extraction.
- **klayout**: custom layout editing, GDS inspection, and physical verification support.
- **magic**: extraction-oriented layout support and additional physical checks.

Keep the following working principle in mind: every design iteration should reduce uncertainty. If a simulation only confirms what was already obvious, it is less valuable than a simulation that separates two competing hypotheses.

4 Architecture of the StrongARM Comparator

The StrongARM latch is a dynamic comparator. It consumes no static current in the core during reset, evaluates on a clock edge, and converts a small differential input into a large logic-level difference through regeneration.

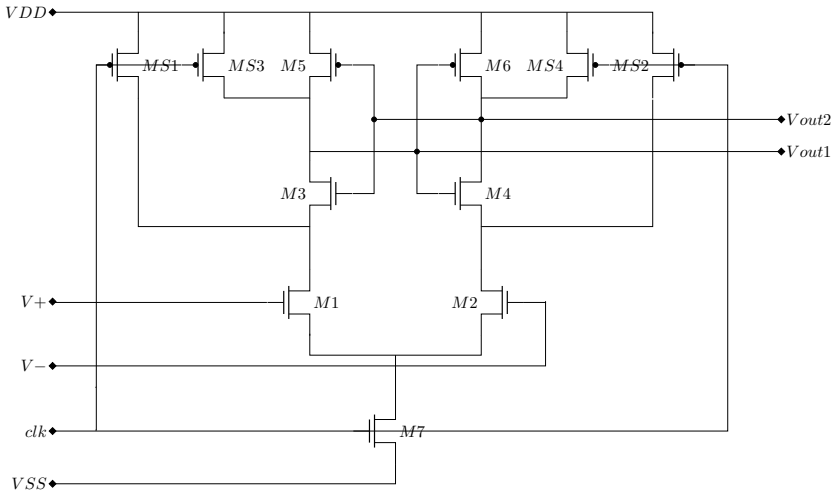

Fig. 2: StrongARM latch architecture used as the main comparator core in this project.

The main transistor groups play distinct roles:

- **M1/M2**: the input differential pair. This pair senses the initial input voltage difference and is usually the dominant source of input-referred offset.
- **M3/M4**: the NMOS cross-coupled contribution in the discharge path. Their mismatch matters, but their offset is partially suppressed because some gain has already developed before they dominate the regeneration.
- **M5/M6**: the PMOS cross-coupled regenerative pair. They provide positive feedback and quickly force one side high and the other side low.
- **M7**: the tail device. It controls the evaluation current and therefore has strong influence on speed, power, and kickback.

- **MS1–MS4**: reset or precharge switches. They restore internal nodes and suppress dynamic memory from the previous cycle.

4.1 Two Essential Phases

Reset or precharge phase. Before evaluation, the internal nodes are precharged so that the next comparison starts from a known and symmetric state. This is not just a convenience. It is what prevents one decision from contaminating the next decision through residual charge.

Evaluation or regeneration phase. Once the clock activates the tail path, the internal nodes begin to discharge. The side associated with the stronger input branch falls faster. That small difference is then amplified by the cross-coupled structure until a full logic decision is reached.

4.2 Why This Topology Is Attractive

Razavi highlights several reasons this topology remains widely used:

- single-phase clocking,
- zero static power in the latch core,
- rail-to-rail outputs after regeneration,
- a reasonably intuitive partition of offset contributors,
- high speed when device parasitics and the tail path are well managed.

The price paid for these advantages is that the circuit is strongly nonlinear, strongly time-variant, sensitive to mismatch near zero differential input, and capable of disturbing the preceding stage through kickback.

5 From Idea to Schematic in xschem

The schematic work should start with a simple architectural view and then move toward a layout-compatible final view. That sequence matters because it separates conceptual correctness from implementation detail.

5.1 Basic Architectural View

The first view should answer a simple question: *does the topology reflect the intended signal flow?* Use the most readable version first.

Typical launch command:

```
cd $HOME/IHP__CMP7345/CMP7345-main/schematic/xschem
xschem CMP7345-basic.sch
```

At this stage, do not worry about visual polish. Focus on:

- correct connectivity,
- clear differential symmetry,
- stable naming of critical nodes such as `P`, `Q`, `X`, `Y`, `clk`, and the outputs,
- a clean separation between the comparator core and any output buffer or output latch.

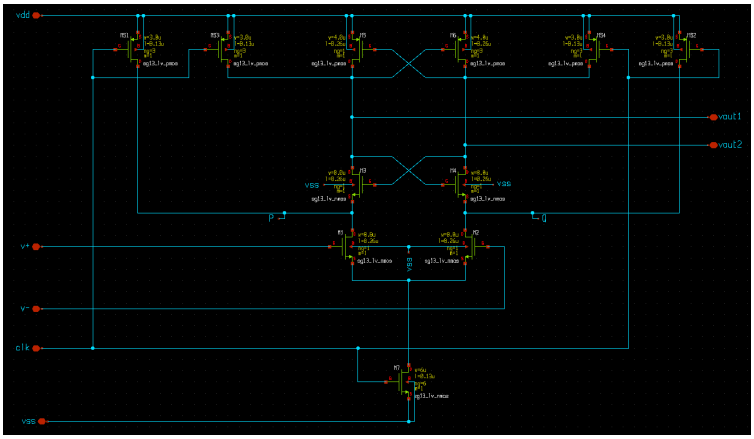

Fig. 3: Basic architectural schematic view for early reasoning and review.

5.2 Extended Layout-Compatible View

The final schematic should already anticipate layout and verification. This means the net names, device grouping, source and bulk handling, and hierarchy boundaries should all support later extraction and LVS.

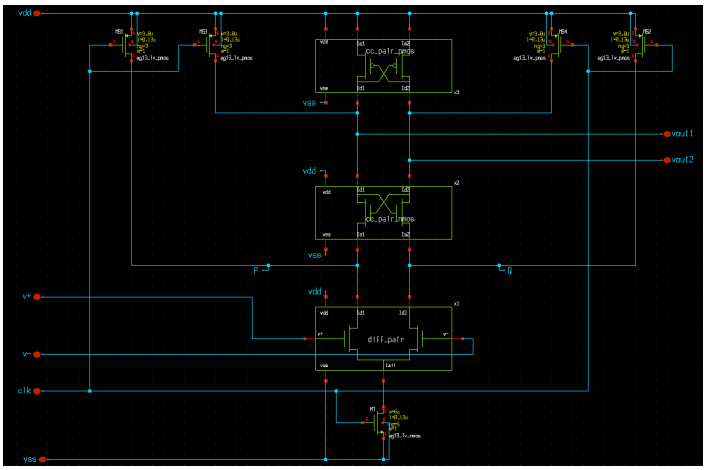

Fig. 4: Extended schematic view aligned with a layout-oriented implementation.

Typical launch command:

```
cd $HOME/IHP__CMP7345/CMP7345-main/schematic/xschem
xschem CMP7345-final.sch
```

Schematic quality rules

- Keep left and right signal paths visually mirrored where possible.
- Use explicit labels on the differential and regenerative nodes.
- Group device pairs that will later be matched in layout.
- Avoid accidental asymmetry in auxiliary circuitry, especially output loading.
- Keep a clean distinction between the comparator core and the surrounding testbench circuitry.

5.3 xschem Productivity Notes

For self-paced work, editing speed matters because it keeps the focus on design rather than tool friction.

Key	Action
`q`	Open the properties window of the selected object.
`m`	Move the selected object.
`c`	Copy the selected object.
`Alt+R`	Rotate the selected symbol.
`Alt+F`	Flip the selected symbol.
`w`	Start wiring.
`Delete`	Remove the selected object.
`Shift+A`	Inspect the generated netlist.

6 Sizing Philosophy Before Optimization

One of the most useful lessons from Razavi's article is that comparator design does not begin with brute-force optimization. It begins with sensible first guesses and an understanding of which devices deserve area.

The practical starting principles are:

- Start near minimum length unless a stronger reason exists to use longer channels.
- Give the input pair enough area that random threshold mismatch does not immediately dominate the offset budget.
- Remember that larger devices reduce mismatch but also increase capacitance and can slow the early gain development.
- Treat the tail device as a speed and kickback lever, not only as a current source.
- Treat reset devices as dynamic correctness devices: if they are too weak, the next cycle can inherit offset from the previous cycle.

For a first-order mismatch estimate, Pelgrom-style reasoning is still a useful mental tool:

$$\sigma(\Delta V_{TH}) \approx \frac{A_{VTH}}{\sqrt{WL}} \tag{1}$$

The immediate design implication is simple: the devices that most directly translate threshold mismatch into input-referred decision error should receive the greatest matching care. In the StrongARM latch, that is most often the input pair `M1/M2`.

7 Simulation Strategy

The comparator cannot be understood from one simulation view alone. Use a sequence of increasingly realistic analyses.

7.1 What Each Analysis Teaches

View	Typical Setup	What We Learn
DC	Sweep input or common-mode level at a fixed clock condition.	Bias behavior, trip tendency, headroom, and basic operating-point sanity.
Transient	Pulse `CLK` and apply a small differential input.	Regeneration, delay, output polarity, node sequencing, and failure modes near the decision boundary.
AC	Use only on biased linear sub-blocks, not on the full decision event.	Bandwidth and pole-zero intuition for front-end linear portions.
Monte Carlo mismatch	Re-run many transient cases with mismatch enabled.	Offset spread, delay spread, and statistical decision behavior.
Process variation	Sweep or randomize process conditions.	Corner robustness, timing spread, and sensitivity to global shifts.
Kickback-focused transient	Inspect input currents and sampled-node disturbance during clock transitions.	Charge injection back into the source network.

AC analysis is not a substitute for transient analysis in a StrongARM comparator. The actual decision event is nonlinear and clock-driven. Use AC only where the circuit is meaningfully linearized.

7.2 Transient Testbench Setup

The transient bench is the main learning environment because it reveals the actual decision process.

Typical launch command:

```
cd $HOME/IHP__CMP7345/CMP7345-main/testbenches/tran/xschem
xschem CMP7345_tb.sch
```

Model inclusion example:

```
.lib cornerMOSlv.lib mos_tt
.lib cornerDIO.lib dio_tt
.lib cornerRES.lib res_typ
.INCLUDE sg13g2_stdcell.spice
```

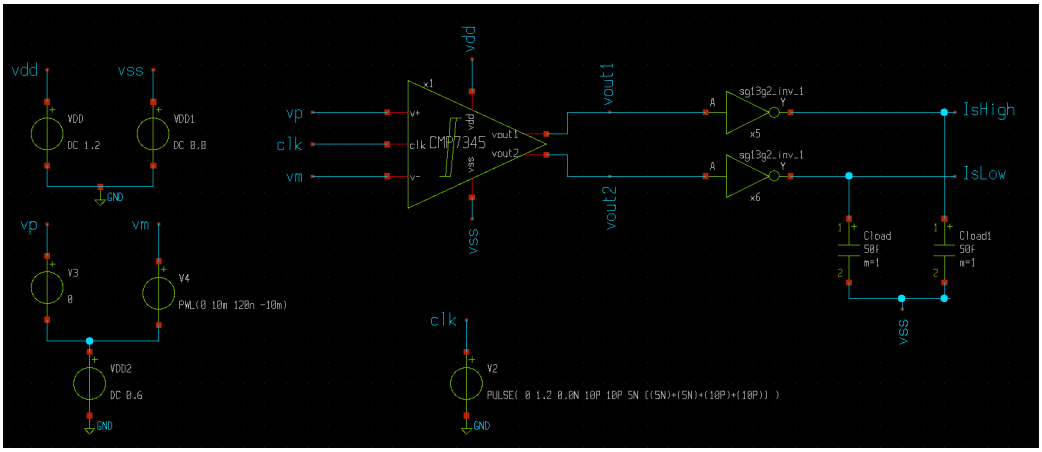

Fig. 5: Transient testbench used to drive the comparator and observe its time-domain response.

The distinction between `.lib` and `.INCLUDE` matters. A `.lib` statement selects one named model section from a library file. An `.INCLUDE` statement inserts the full contents of another SPICE file, which is useful when the testbench contains digital support cells such as output inverters.

Recommended control-block setup:

```
.options cshunt=20f
.options rseries=1
.control
save all
shell rm tran.raw
tran 10p 120n
meas tran tdly TRIG v(clk) VAL=0.6 RISE=11 \
  TARG v(IsHigh) VAL=0.6 RISE=5
write tran.raw
.endc
```

These commands are not decorative. They stabilize the numerical problem, preserve enough observability for debugging, and formalize at least one measurable performance metric.

7.3 Full Nominal Deck Listing

The repository contains a compact listing for the nominal transient deck. It is short, but it already shows the essential structure of a reusable ngspice experiment: load the right model sections, add numerical stabilization, run the transient, and write the raw database.

```
.lib cornerMOSlv.lib mos_tt
.lib cornerDIO.lib dio_tt
.lib cornerRES.lib res_typ
.INCLUDE sg13g2_stdcell.spice

.options cshunt=20f
.options rseries=1
.control
save all
tran  10p 120n
write tran.raw
.endc
```

Read this deck in four layers:

- **Model selection**: `mos_tt`, `dio_tt`, and `res_typ` define the nominal MOS, diode, and resistor conditions.
- **Digital support**: `.INCLUDE sg13g2_stdcell.spice` brings in the standard-cell devices used by the output logic or buffering stages.
- **Numerical conditioning**: `cshunt` and `rseries` make the dynamic latch easier to simulate robustly.
- **Data generation**: the `tran` command creates the waveform set and `write tran.raw` preserves it for later inspection.

Line by line, the deck means the following:

- `.lib cornerMOSlv.lib mos_tt`: use the nominal low-voltage MOS corner.
- `.lib cornerDIO.lib dio_tt`: use the nominal diode corner.
- `.lib cornerRES.lib res_typ`: use the typical resistor section.
- `.INCLUDE sg13g2_stdcell.spice`: include digital support cells used around the latch.
- `.options cshunt=20f`: add a tiny capacitance from each node to ground to suppress unrealistic numerical singularities.
- `.options rseries=1`: add small series resistance to idealized branches and switches.
- `.control`: enter ngspice control mode.
- `save all`: keep all node voltages and branch currents available for debug.
- `tran 10p 120n`: simulate 120 ns while sampling outputs every 10 ps.
- `write tran.raw`: export the raw waveform database.
- `.endc`: leave the control block.

This deck is the right starting point because it is deliberately minimal. Once it behaves correctly, the same structure can be extended into statistical and post-layout variants without changing the basic measurement philosophy.

7.4 What to Look for in the Waveforms

- Are internal nodes properly precharged before evaluation begins?
- Which of `P` or `Q` departs first after the clock edge?
- Is the latch clearly regenerating, or only weakly separating?
- Are the outputs reaching full logic levels in the intended time window?
- Does the comparator recover cleanly before the next cycle?

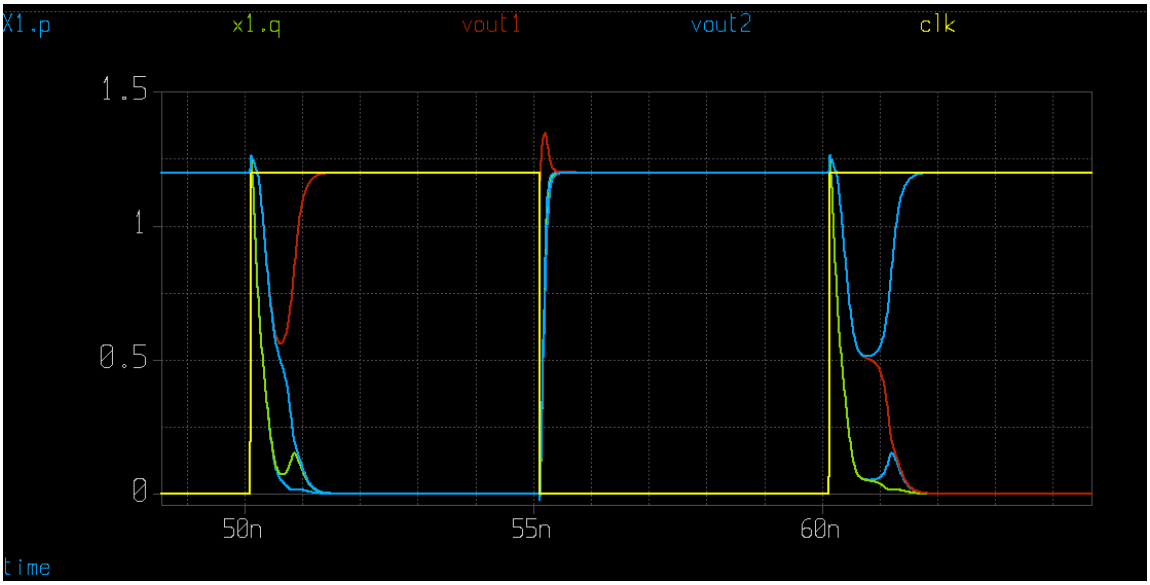

Fig. 6: Example transient result. The key observation is not only the final logic state but the sequence of internal-node separation and reset recovery.

8 Understanding the Core Trade-Offs

8.1 Regeneration and Delay

During evaluation, the StrongARM comparator passes through a short analog amplification phase before strong positive feedback takes over. That early gain matters because it reduces how much later mismatches are referred back to the input.

The practical timing metric for this project can be defined in several ways:

- clock-to-output crossing delay,
- time for the differential internal nodes to exceed a chosen value,
- time for a following latch or inverter to capture a valid logic state.

The exact metric is less important than consistency. Use the same metric while sweeping device sizes so that each change has a comparable interpretation.

8.2 Mismatch and Input Offset

Mismatch is the first statistical topic to study because it explains why a comparator can favor one side even when the applied differential input is nearly zero.

Most important conceptual point
Not every mismatched pair contributes equally to input-referred offset. In a StrongARM latch, the input pair usually dominates because it acts before significant regeneration has developed. Later pairs such as `M3/M4` and `M5/M6` still matter, but their offset contribution is partially attenuated by the gain already present in the signal path.

Practical mismatch guidance from the workshop flow:

- enable mismatch only on the pairs of interest first,
- use `mm_ok=1` on selected instances,

- choose the mismatch-enabled model section such as `mos_tt_mismatch`,
- examine not only nominal delay but the distribution of decisions for tiny input differences.

8.3 Mismatch Deck Listing

The stored mismatch listing is much more than a single transient run. It is a small statistical experiment written directly in the ngspice control language.

```
.lib cornerMOSlv.lib mos_tt_mismatch
.lib cornerRES.lib res_typ
.lib cornerDIO.lib dio_tt
.INCLUDE sg13g2_stdcell.spice



.options cshunt=20f
.options rseries=1
.control
shell rm tran.raw
  let mc_runs = 2
  let run = 0
  set curplot=new
  set scratch=$curplot
  set appendwrite
  let vh=unitvec(mc_runs)
  let vl=unitvec(mc_runs)
  let td=unitvec(mc_runs)
  let teval=66n
  let vtdly=unitvec(mc_runs)

  dowhile run < mc_runs
    reset
    tran 10p 120n
    meas tran vh1 find v(vout1) at=73n
    meas tran vl1 find v(vout2) at=73n
    meas tran tdly TRIG v(clk) VAL=0.6 RISE=2 TARG v(vout2) VAL=0.6
       FALL=2

    set run = $&run
    set plt = $curplot
    setplot $scratch
    let vvd{$run} = {$plt}.v(vout1)
    let vh[run]={$plt}.vh1
    let vl[run]={$plt}.vl1
    let vtdly[run]={$plt}.tdly
    setplot $plt
    let run = run + 1
    write tran.raw
    end
let vdv=vp-vm
meas tran vdiff find v(vdv) at=73n
print {$scratch}.vtdly
.endc
```

The purpose of this deck is to repeat the same comparator decision under different mismatch realizations and then capture the resulting spread in output levels and delay. The most important structural change relative to the nominal deck is the model section:

- `mos_tt_mismatch` enables device-level random mismatch in the MOS models.
- The circuit itself decides which instances participate through parameters such as `mm_ok=1` in the schematic netlist.

The control block is worth reading carefully:

- `shell rm tran.raw`: remove the old database so the new Monte Carlo-style run is easier to interpret.
- `let mc_runs = 2`: define how many mismatch realizations will be executed. The stored file uses two runs, which is enough to illustrate the mechanism but not enough for statistical signoff.
- `set curplot=new` and `set scratch=$curplot`: create a separate plot space to hold accumulated results across runs.
- `set appendwrite`: allow repeated writing into the same raw output stream.
- `unitvec(mc_runs)` allocations such as `vh`, `vl`, and `vtdly`: allocate vectors to store one scalar result per run.
- `dowhile run < mc_runs`: begin the loop over mismatch realizations.
- `reset`: reinitialize the simulator before each run so each realization starts cleanly.
- `tran 10p 120n`: execute one transient experiment for the current mismatch sample.
- `meas tran vh1 find v(vout1) at=73n`: sample the final high-side output at a chosen evaluation time.
- `meas tran vl1 find v(vout2) at=73n`: sample the low-side output at the same time.
- `meas tran tdly ...`: measure decision delay from the clock edge to the output threshold crossing.
- `set plt = $curplot` and `setplot $scratch`: move data from the transient run plot into the persistent scratch plot.
- `let vvd{$run} = {$plt}.v(vout1)`: preserve a waveform per run for later overlay and visual comparison.
- `let vh[run]`, `let vl[run]`, `let vtdly[run]`: store the scalar measurements into the result vectors.
- `write tran.raw`: append the current run to the raw database.
- `let vdv=vp-vm`: form a differential-input monitor after the loop.
- `meas tran vdiff find v(vdv) at=73n`: evaluate the differential stimulus at the chosen time point.
- `print {$scratch}.vtdly`: print the delay spread across mismatch realizations.

Conceptually, this deck answers three questions at once:

- does the comparator still decide consistently under random mismatch,
- how much does delay move from run to run,

- how close is the operating point to ambiguous or metastable behavior near zero differential input.

In a production-oriented flow, the next step would be to increase `mc_runs` substantially and export the resulting vectors into a histogram or cumulative-yield analysis. For a self-paced learning flow, the stored deck is already very valuable because it exposes the mechanics of how statistical comparator experiments are scripted.

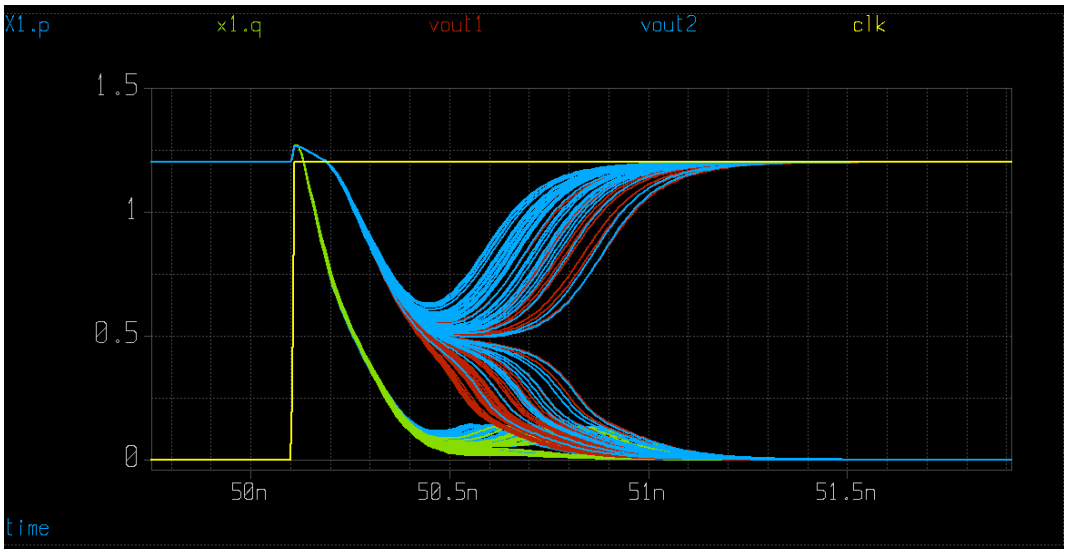

Fig. 7: Mismatch-oriented simulation result. The objective is to read spread, not only a single winner waveform.

8.4 Process Variation

Process variation differs from local mismatch in that it can shift many device parameters together. Threshold voltage, mobility, effective dimensions, resistance, and parasitic behavior can all move. A comparator that looks excellent at one nominal corner can become too slow or too offset-sensitive elsewhere.

Use process-variation studies to answer these questions:

- Does the decision remain correct across corners?
- Does delay remain inside the system timing budget?
- Does the comparator become more metastable or more kickback-prone in some corners?
- Does precharge remain strong enough under slow devices and low supply?

8.5 Process-Variation Deck Listing

The process-variation deck follows the same control structure as the mismatch deck, but it changes the enabled model sections and writes out additional result data for later processing.

```
.lib cornerMOSlv.lib mos_tt_stat
.INCLUDE sg13g2_stdcell.spice
.lib cornerDIO.lib dio_tt
.lib cornerRES.lib res_stat

.options cshunt=20f
```

```
.options rseries=1
.control
  shell rm tran.raw
  let mc_runs = 20
  let run = 0
  set curplot=new
  set scratch=$curplot
  set appendwrite
  let vh=unitvec(mc_runs)
  let vl=unitvec(mc_runs)
  let td=unitvec(mc_runs)
  let teval=66n
  let vtdly=unitvec(mc_runs)

  dowhile run < mc_runs
    reset
    tran 10p 120n
    meas tran vh1 find v(vout1) at=73n
    meas tran vl1 find v(vout2) at=73n
    meas tran tdly TRIG v(clk) VAL=0.6 RISE=2 TARG v(vout2) VAL=0.6
       FALL=2

    set run = $&run
    set plt = $curplot
    setplot $scratch
    let vvd{$run} = {$plt}.v(vout1)
    let vh[run]={$plt}.vh1
    let vl[run]={$plt}.vl1
    let vtdly[run]={$plt}.tdly
    setplot $plt
    let run = run + 1
    write tran.raw
    end
let vdv=vp-vm
meas tran vdiff find v(vdv) at=73n
print {$scratch}.vtdly
wrdata distribution.csv {$scratch}.vh {$scratch}.vl
.endc
```

The key model-level differences are:

- `mos_tt_stat`: enables statistical process variation for MOS devices rather than only local mismatch.
- `res_stat`: enables statistical resistor variation.
- The diode library remains at `dio_tt`, which is appropriate when the experiment is focused mainly on MOS and resistor spread in this comparator context.

The overall loop structure is the same, but a few details deserve emphasis:

- `mc_runs = 20`: this file already uses a larger number of runs than the mismatch example, which makes the resulting spread more meaningful.
- `vh1`, `vl1`, and `tdly`: the same measurement trio is used so that process spread can be compared directly against mismatch spread.
- `write tran.raw`: still preserves the waveform history across runs.

- `wrdata distribution.csv {$scratch}.vh {$scratch}.vl`: export selected scalar results into a CSV-style file that can be post-processed outside ngspice.

This last line is especially important from a workflow point of view. It marks the transition from waveform inspection to data analysis. Once results are written to a structured file, the designer can build histograms, estimate spread, compare tails, and track whether the comparator still meets the intended logic levels and timing margin under process variation.

The process deck therefore teaches a useful engineering habit: reuse the same measurement definitions across nominal, mismatch, and statistical runs so that every comparison is anchored to the same observables.

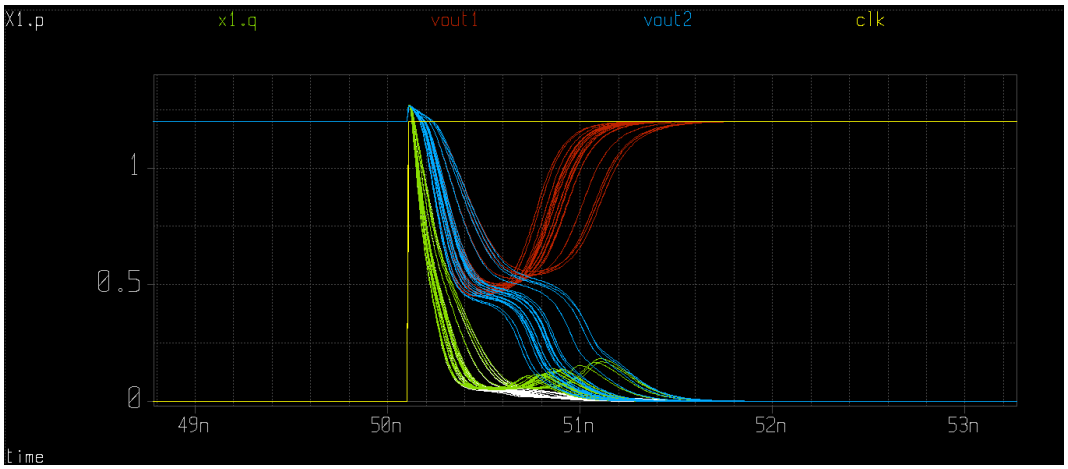

Fig. 8: Process-variation oriented result. Evaluate both timing spread and qualitative waveform health.

8.6 Metastability

Metastability is the failure mode in which the comparator takes too long to resolve a very small input difference. In a synchronous system, this is dangerous because the following digital stage may observe an undefined or late state.

The key mental model is exponential regeneration. Once the latch enters the positive-feedback-dominated region, smaller initial differential conditions shift the response later in time. This is why tiny input differences can transform a healthy-looking transient into a near-failure case.

For self-paced study, perform a sequence of transient simulations with progressively smaller differential inputs, for example from millivolts toward microvolts. Measure how much extra delay each decade of input reduction introduces. This gives direct intuition about regeneration strength and how much timing margin remains before a half-cycle or cycle boundary is violated.

When exploring metastability, remove or carefully control all avoidable asymmetries in the surrounding circuitry. Even a slightly unbalanced load can bias the result and hide the true behavior of the core comparator.

8.7 Input-Referred Noise

For a dynamic comparator, input-referred noise is best studied through repeated noisy transient decisions rather than through a simple linear output-noise-over-gain calculation. The practical interpretation is statistical: how much random input-equivalent disturbance is required to skew the balance between zeros and ones?

This manual does not prescribe a single simulator-specific noise script, but the self-paced design task is clear: run repeated transient decisions near zero differential input and estimate how much deterministic input skew is needed before the output probability distribution becomes noticeably unbalanced.

8.8 Kickback Noise

Kickback noise is the current or charge disturbance pushed from the internal switching nodes back to the comparator inputs when the clock transitions and regeneration begins.

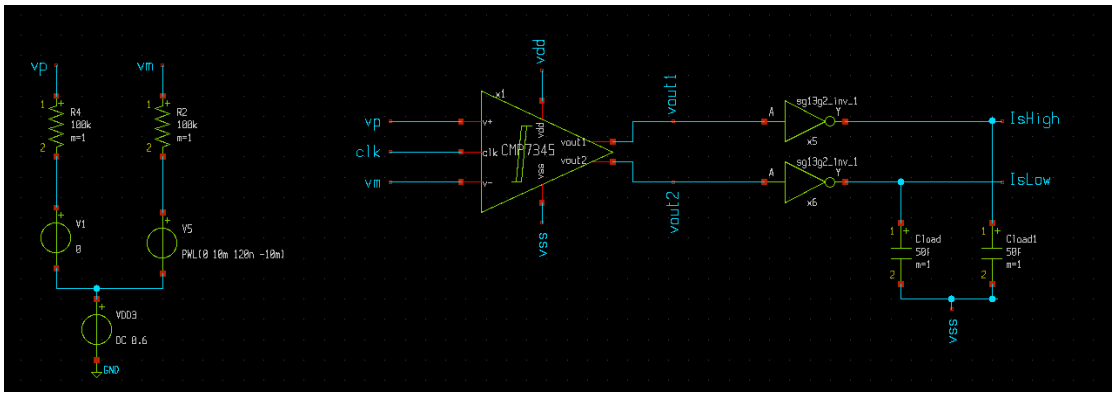

Fig. 9: Kickback-oriented testbench view. The objective is to inspect what the comparator injects back into the signal source.

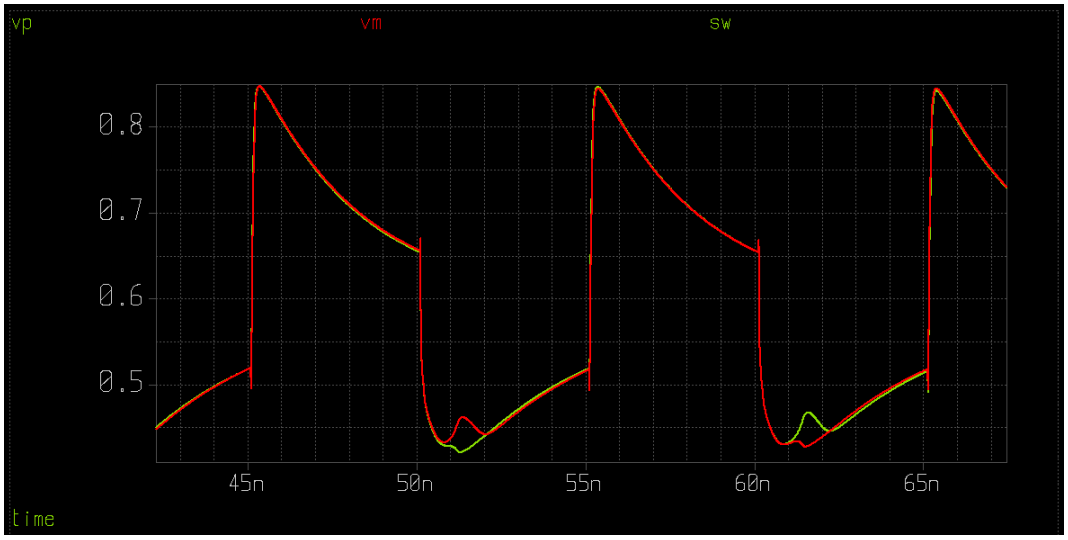

Fig. 10: Kickback result. Inspect both common-mode and differential disturbance components.

Kickback matters for two reasons:

- it can disturb the signal source or sample-and-hold network,
- it can become comparable to the actual signal when the intended differential input is very small.

The usual design tension is straightforward: stronger and faster switching often helps speed but can also increase charge injection and therefore worsen kickback.

9 Adding an Output Memory Element

Razavi emphasizes that the raw StrongARM core loses its decision during the reset phase because the internal nodes are precharged again. In many systems, this means the comparator alone does not hold a valid digital output between decisions.

The usual remedy is to add a reset-set latch or another dynamic-to-static conversion stage after the comparator core. The purpose is not to improve the analog decision itself but to preserve and present that decision to the following logic.

For self-paced work, treat this as a separate design block with its own requirements:

- input sensitivity compatible with the comparator's internal-node swing,
- small enough delay that the total timing budget is still met,
- balanced enough loading that the comparator core is not accidentally biased.

10 Layout-Oriented Thinking

Comparator layout is not a later cleanup activity. It is part of the offset and timing design from the beginning.

10.1 Implementation Rules

Portable cell recipe

- use only low-voltage MOS devices shared by the target technologies,
- keep local taps close and explicit,
- route compactly and symmetrically in `Metal1-Metal3`,
- plan the cell so extraction and LVS will see the same intent as the schematic.

Matching priorities

- mirror the left and right halves,
- consider common-centroid placement where beneficial,
- equalize parasitics on the regenerative nodes,
- keep the clock path compact and balanced,
- use dummies and uniform local environment for the most sensitive devices.

10.2 Training Layout Decomposition Views

It is often easier to lay out the StrongARM latch correctly if it is mentally decomposed into three matched substructures: the differential pair, the NMOS cross-coupled portion, and the PMOS cross-coupled portion.

Differential Pair **NMOS Cross-Coupled Pair**

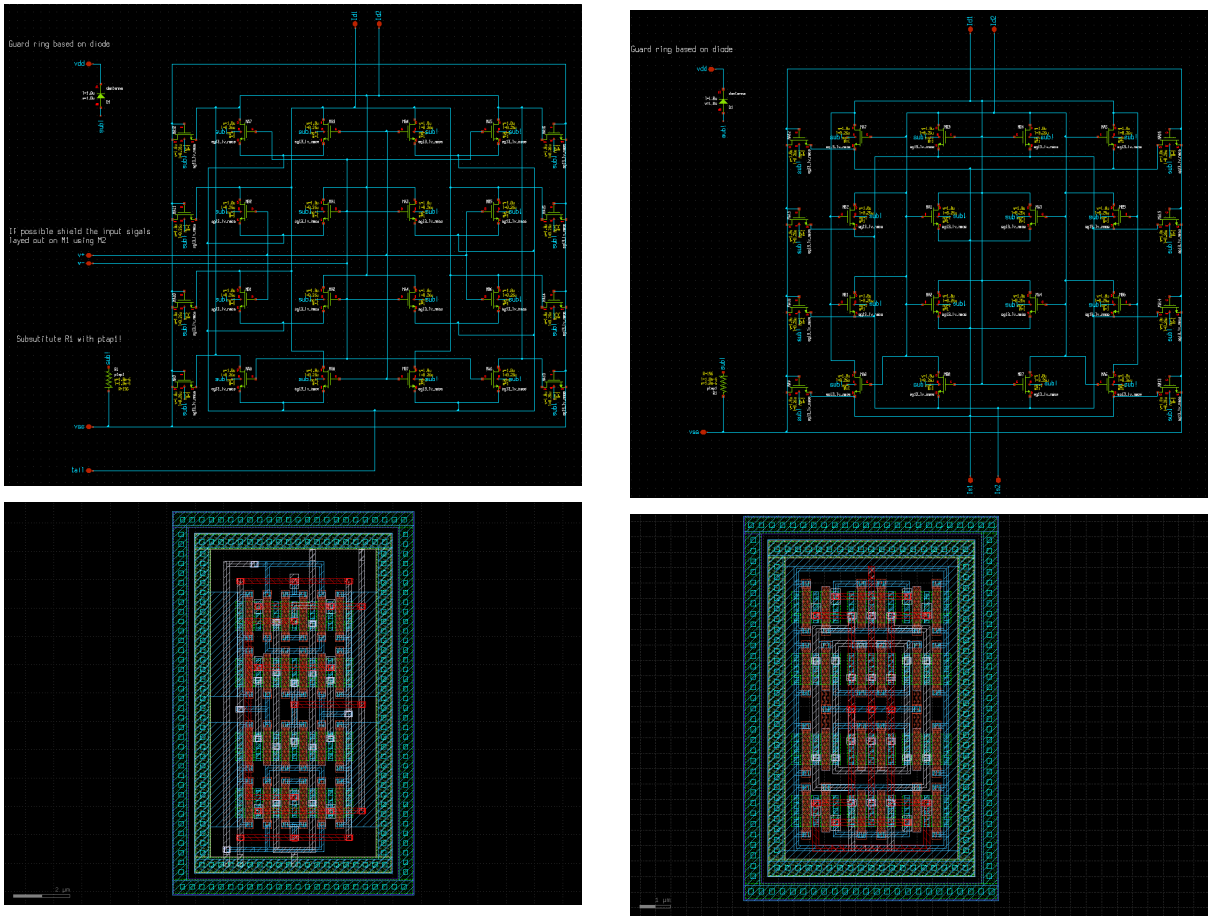

Fig. 11: Sub-block decomposition for learning symmetry and layout correspondence.

PMOS Cross-Coupled Pair **Final Comparator Cell**

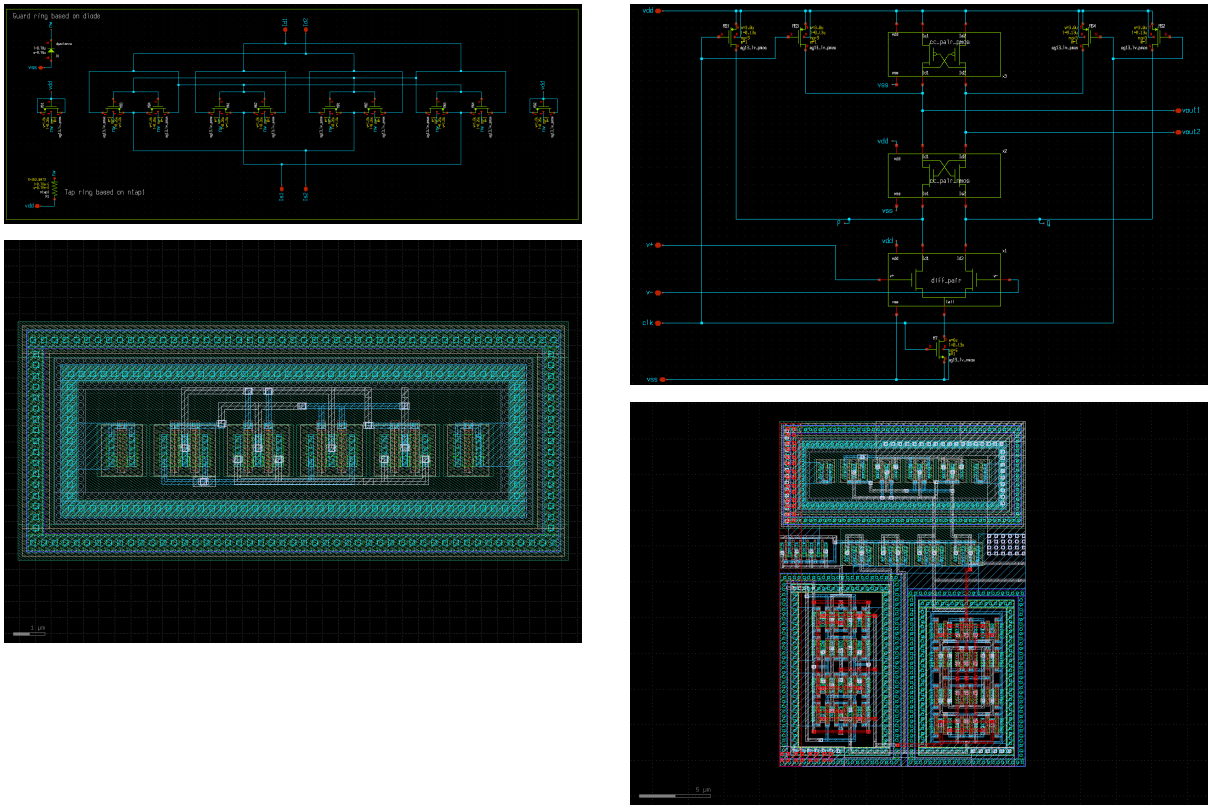

Fig. 12: From matched building blocks to the final comparator schematic and layout.

10.3 klayout Productivity Notes

Key	Action
`q`	Open the properties window.
`m`	Move the selected object.
`c`	Copy the selected object.
`a`	Align tool.
`o`	Generate via while using the path tool.

Typical launch command:

```
klayout -e
```

11 Parasitic Extraction and Post-Layout Simulation

Up to this point, the manual has treated the comparator mostly through its schematic intent. That is necessary, but it is not sufficient. A dynamic comparator is especially sensitive to parasitic capacitance because its decision is time-domain based, regenerative, and strongly dependent on the relative motion of internal nodes. For this reason, parasitic extraction is not a documentation step. It is part of the design validation loop.

11.1 Why PEX Matters for a StrongARM Latch

The extracted netlist differs from the schematic in two important ways:

- it preserves the connectivity as realized in layout rather than as intended in the hand-drawn schematic,
- it adds parasitic capacitances and extracted device segmentation that change speed, kickback, settling, and sometimes even functional behavior.

For a StrongARM latch, these effects are not secondary. They can directly shift:

- regeneration delay,
- dynamic offset behavior,
- metastability margin,
- kickback amplitude,
- output balance and reset quality.

Never assume that the extracted comparator is equivalent to the schematic-only comparator just because the layout looks symmetric. The extracted view must be simulated and checked as its own implementation artifact.

11.2 Extraction Command and Directory Structure

In the workshop flow, the extracted netlist is generated from the final GDS view. The command recorded in the project notes is:

```
cd $HOME/IHP__CMP7345/CMP7345-main/layout
kpex --pdk ihp-sg13g2 --magic --magic_cthresh=2.0 \
  --gds CMP7345.gds
```

The key paths used in the post-layout flow are:

- schematic netlist:
  `$HOME/IHP__CMP7345/CMP7345-main/schematic/CMP7345-final.spice`
- extracted netlist:
  `$HOME/IHP__CMP7345/CMP7345-main/netlist/pex/CMP7345.pex.spice`
- extracted-view symbol directory:
  `$HOME/IHP__CMP7345/CMP7345-main/netlist/pex/`
- post-layout transient bench:
  `$HOME/IHP__CMP7345/CMP7345-main/testbenches/tran/xschem/CMP7345_pex_tb.sch`

The additional xschem symbol in the PEX directory is useful because the extracted subcircuit often differs from the schematic cell in naming, pin order, or hierarchy flattening. It is usually better to instantiate a dedicated PEX symbol than to assume the original schematic symbol still maps correctly.

11.3 How to Use the Extracted View in Simulation

The practical flow is simple in principle:

1. Generate the extracted netlist from the final layout.
2. Add or use the dedicated xschem symbol that represents the extracted subcircuit.
3. Instantiate that symbol in `CMP7345_pex_tb.sch`.
4. Include the extracted netlist in the simulation deck instead of the schematic-only comparator view.
5. Re-run the same transient, kickback, and timing measurements used for the schematic version.

The most valuable comparison is not a brand-new testbench. It is the same testbench run twice: once with the schematic view and once with the extracted view. That isolates the impact of layout realization and parasitic loading.

11.4 What to Check First in the Extracted Netlist

From the comparison notes in this project, the extracted view should not be assumed correct without inspection. Several differences are especially important.

Observed high-priority checks

- top-level subcircuit pin order may differ between the schematic and PEX views,
- one PMOS cross-coupled source-side internal node should be verified carefully,
- the effective extracted width of the tail NMOS may differ from the schematic intent,
- parasitic capacitances are present only in the PEX view and will change transient behavior.

The top-level pin-order issue is particularly dangerous because it can produce a simulation that runs and looks plausible while actually connecting the comparator incorrectly.

11.5 Schematic Versus PEX Comparison Points

The following summary captures the main comparison items already identified in the project notes.

Check Item	Observation	Impact	Priority
Top-level pin order	Schematic and PEX subcircuits do not use the same pin ordering.	Wrong positional instantiation can miswire the testbench.	High
PMOS cross-coupled source connectivity	One internal PMOS-side source node should be verified in the extracted text.	Can alter bias and regeneration.	High
Tail NMOS effective width	Extracted multiplicity appears different from the schematic declaration.	Can change tail current, speed, and kickback.	High
Parasitic capacitors	Capacitors appear only in the PEX netlist.	Changes delay, settling, metastability, and kickback.	Medium
Device finger expansion	Many devices are flattened into unit or repeated devices.	Usually acceptable, but must be interpreted correctly.	Low

11.6 Pin-Mapping Example

The project comparison notes show that the top-level subcircuit interface does not match by position:

- schematic pin order: `v- v+ vss vdd clk vout2 vout1`
- PEX pin order: `vdd vout1 vout2 v+ v- clk vss`

This means that a positional subcircuit call copied from the schematic flow can silently connect the wrong nets when the extracted view is substituted. If the simulator deck uses positional instantiation, remap the pins explicitly or switch to a dedicated symbol that already encodes the PEX order.

Before trusting any post-layout waveform, verify that `VDD`, `VSS`, `V+`, `V-`, `clk`, and both outputs are mapped exactly as intended in the extracted-view instance.

11.7 How Parasitics Change the Interpretation of Results

The extracted netlist includes capacitors that do not exist in the ideal schematic description. In the project notes, examples include coupling between supply and outputs, between the differential inputs, and between the two outputs. These are exactly the types of parasitics that matter in a dynamic comparator.

Once PEX is included, interpret changed waveforms carefully:

- a longer delay does not automatically mean a connectivity error; it may be the expected effect of added capacitance,
- a different kickback waveform may be physically more realistic rather than worse in a design-quality sense,

- degraded reset symmetry can come from real layout imbalance or from incorrect extracted pin mapping,
- a changed metastability boundary often reflects the true post-layout regeneration time constant.

The correct question is therefore not, "Does PEX match the schematic waveform exactly?" The correct question is, "Does the extracted implementation still meet the functional, timing, and robustness goals of the comparator?"

11.8 Recommended Post-Layout Checks

After integrating the PEX netlist, run the following checks in order:

1. Verify the extracted subcircuit pin order and symbol mapping.
2. Compare nominal transient behavior between schematic and PEX views.
3. Re-measure clock-to-output delay with the same script used pre-layout.
4. Re-check kickback at the input network because interconnect parasitics can change it significantly.
5. Repeat at least a limited mismatch or corner study if the extracted setup supports it.
6. Inspect any large behavior difference before assuming either the schematic or layout is wrong.

A good post-layout result
A good PEX result does not need to reproduce the pre-layout waveform perfectly. It should preserve correct decision polarity, acceptable timing, reliable reset, and a believable change in analog behavior once parasitic capacitance is included.

12 Physical Verification: DRC and LVS

For the SG13G2 open PDK flow, physical verification is not a single checkbox. It is a sequence of consistency checks that answer two different questions [5]:

- **DRC**: does the layout obey the geometric and density rules of the process?
- **LVS**: does the realized layout represent the same circuit as the intended schematic netlist?

For a dynamic comparator, both are essential. A DRC-clean layout can still be electrically wrong, and an LVS-clean layout can still violate spacing, density, or antenna constraints.

12.1 Design Rule Checking (DRC)

The IHP documentation organizes DRC into multiple rule groups, including minimal precheck rules, the main DRC rule set, and extra checks. The KLayout-based DRC flow is the most directly useful one for this project because it is scriptable and integrates well with layout debug.

For the underlying numerical geometry requirements, layer definitions, and rule categories, the SG13G2 open-source layout rule manual remains the primary reference [7].

Practical meaning of the DRC groups

- **Precheck rules**: catch obvious structural or layer-usage issues early.
- **Main rules**: the default signoff-oriented rules, including density by default.
- **Extra rules**: additional checks that may be slower or more specialized.

12.2 Running KLayout DRC from the CLI

According to the SG13G2 verification documentation, the main command-line entry point is `run_drc.py` [5]. A typical invocation is:

```
python3 run_drc.py --path=CMP7345.gds \
  --run_mode=deep \
  --run_dir=drc_cmp7345 \
  --no_density
```

The most useful options for day-to-day comparator work are:

- `-path`: input GDS file,
- `-topcell`: explicit top-level cell selection,
- `-run_dir`: directory for logs and result databases,
- `-run_mode=deep|flat`: hierarchical versus flat execution,
- `-table`: run only a selected rule table when debugging,
- `-no_density`: useful for faster early debug,
- `-antenna`: enable antenna checks when needed.

The documentation notes that the main DRC rule set includes density checks by default. For early development iterations, it is reasonable to disable density temporarily and focus first on device, enclosure, spacing, and routing correctness.

12.3 DRC Outputs and Debug Workflow

The run directory typically contains a log file, the executed rule deck, and a KLayout marker database such as `<design>.lyrdb`. That marker database can be loaded directly into KLayout together with the GDS.

```
klayout CMP7345.gds -m CMP7345.lyrdb
```

This is a very productive debug loop because the marker browser points directly at the failing shapes. For a comparator, that usually means you should inspect:

- gate and active enclosure around the matched input pair,
- via enclosure and via-array symmetry on the two regenerative branches,
- guard-ring continuity and local taps,
- clock routing geometry and narrow jogs,
- accidental off-grid or forbidden-layer usage.

12.4 Running DRC from the KLayout GUI

The SG13G2 documentation also describes GUI-based DRC. The required menus are exposed by launching KLayout with the PDK tech paths added to `KLAYOUT_PATH`:

```
KLAYOUT_PATH=$PDKPATH/libs.tech/klayout:
$PDKPATH/libs.tech/klayout/tech/ klayout -e
```

Within the GUI, the same rule options can be configured from the SG13G2 DRC menus and the results appear as marker databases on top of the layout. The GUI mode is convenient for interactive debug; the CLI mode is better for repeatable regression runs.

12.5 Layout Versus Schematic (LVS)

LVS answers the second crucial question: does the layout extract to the same netlist that the schematic claims to implement? For this comparator, LVS is especially important because the topology is highly symmetric and flattened extraction can make wrong connectivity look visually plausible.

The IHP LVS documentation also emphasizes device recognition and derived layers. In practice, that means the layout must use process-recognized devices and layer combinations so the extractor can identify MOSFETs, taps, diodes, resistors, and other elements correctly. For this design, that is directly relevant to:

- `nmos` and `pmos` devices,
- local tap structures such as `ptap1` and `ntap1`,
- protection-related elements such as `dantenna` and `dpantenna` if present.

12.6 Running KLayout LVS from the CLI

The documented command-line entry point is `run_lvs.py` [5]. A practical example for this project is:

```
python3 run_lvs.py \
  --layout=CMP7345.gds \
  --netlist=CMP7345-final.spice \
  --run_dir=lvs_cmp7345 \
  --run_mode=flat
```

Important options include:

- `-layout`: GDS file to extract,
- `-netlist`: schematic netlist used as the reference,
- `-topcell`: explicit top-cell selection when needed,
- `-run_mode=flat|deep`: extraction mode,
- `-no_simplify`: disable default simplification if detailed debug is needed,
- `-no_series_res` and `-no_parallel_res`: useful when resistor reduction hides a problem,
- `-combine_devices`, `-purge`, and `-purge_nets`: simplification controls,
- `-verbose`: more detailed execution logs.

The documentation notes that simplification is enabled by default. That is often convenient, but if a comparison looks suspicious, disabling simplification can make the source of the mismatch much easier to understand.

12.7 LVS Outputs and Debug Workflow

The LVS run directory typically contains:

- an execution log,
- an extracted circuit file such as `<design>.cir`,
- an LVS results database such as `<design>.lvsdb`,
- and often an extracted netlist file such as `<layout_name>_extracted.cir`.

The results database can be loaded in KLayout with the netlist browser:

```
klayout CMP7345.gds -mn CMP7345.lvsdb
```

For this comparator, the most likely LVS pain points are:

- swapped or renamed output nodes in the symmetric latch,
- source and drain interpretation differences in extracted MOS instances,
- missing or misrecognized well and substrate ties,
- incorrect top-level pin naming or order,
- merged or split devices that obscure finger-count intent.

12.8 Running LVS from the KLayout GUI

As with DRC, the GUI flow is enabled by launching KLayout with the PDK paths in `KLAYOUT_PATH`:

```
KLAYOUT_PATH=$PDKPATH/libs.tech/klayout:
$PDKPATH/libs.tech/klayout/tech/ klayout -e
```

The SG13G2 LVS menus allow selection of the active cell and the schematic netlist path. The documentation notes that if no netlist path is specified, the tool will try to find a matching `.cdl`, `.spice`, or `.cir` file in the same directory as the layout file.

12.9 Comparator-Specific Verification Strategy

For a self-paced comparator project, the most effective verification order is:

1. run DRC early and often during layout construction,
2. reach a DRC-clean layout before trusting extraction results,
3. run LVS against the final schematic netlist,
4. investigate any pin, device-count, or tie-structure mismatch before moving to PEX,
5. only then generate and simulate the extracted netlist.

PEX is not a substitute for LVS. A post-layout simulation can still produce plausible waveforms even when the extracted layout is connected incorrectly. Always establish DRC and LVS confidence before interpreting PEX results.

13 Chip Finishing

After layout, DRC, LVS, and initial extraction, the design is still not fully ready for manufacturing-oriented handoff. The last physical-preparation step is chip finishing: adding the non-functional but process-critical structures that make the layout manufacturable and acceptable for submission. In the SG13G2 open PDK flow, the most important finishing topic for this project is filler generation [6].

13.1 Why Dummy Fill Is Required

Dummy metal fill is used to improve metal-density uniformity across the layout. Sparse metal regions and dense metal regions polish differently during CMP. Without density balancing, low-density regions can suffer over-polishing and dense regions can suffer under-polishing. The result is poorer layer planarity, degraded downstream process behavior, reduced yield, and possible reliability problems.

The key point is that the inserted fill is electrically inactive but physically important. It is not there to change the circuit function. It is there to make the layout manufacturable.

Why this matters even for a small analog block

- A comparator cell may be compact, but density rules are checked at the chip or region level, not only at the transistor core.
- Adding filler late can alter parasitic loading slightly, especially on upper metal layers and around sensitive routing.
- It is better to regard filler as part of the physical implementation rather than as a last-minute cosmetic step.

13.2 Mandatory Sealring

The IHP finishing documentation explicitly notes that a `sealring` is mandatory in the design [6]. In practice, that means the final chip-level layout must contain the process-provided sealring structure, which can be placed from the KLayout library menu.

For this manual, the comparator itself is only a core block, but the rule remains important because once the design is integrated into a larger top-level layout, the finishing flow must include the mandatory sealring before final verification and submission.

13.3 Filler Generation in KLayout

The SG13G2 PDK provides KLayout menu entries for filler generation. The documented menu flow includes:

- `Fill Activ/GatPoly`
- `Fill Metal`
- `Fill Top Metal`
- `Density Report`

These menu actions allow the designer to insert the corresponding filler shapes and then evaluate whether the resulting layout satisfies density expectations.

The documentation also provides a CLI-based KLayout flow using the filler script:

```
klayout -n sg13g2 -zz -r \
  $PDK_ROOT/$PDK/libs.tech/klayout/tech/scripts/filler.py \
  -rd output_file=./design_filled.gds \
  design_nofill.gds
```

This is useful when the filling step needs to be scripted or repeated as part of a batch flow.

13.4 Controlling Filler Density

For Metals 1 through 5, the filler density can be tuned by changing the spacing between filler shapes. The documentation notes that the minimal spacing between metal fillers is `0.42um`. For scripted flows, the relevant KLayout macro files can be edited to adjust parameters such as width, length, and distance.

The important engineering constraint is that density tuning must still remain DRC-clean. A more aggressive filler pattern is not helpful if it creates new spacing or enclosure violations. The finishing documentation points back to Sections 5.18 and 5.23 of the SG13G2 layout rule manual for filler sizing and separation constraints [7].

When filler parameters are customized, always re-run the default DRC flow afterward. Filler is part of the final physical implementation and must obey the same geometry rules as the rest of the design.

13.5 Verification After Filling

The documented verification sequence after fill insertion is straightforward [6]:

1. Run the `Density Report` tool to check whether the target densities are met.
2. Run DRC with the default rule options.
3. Make sure density checks are enabled, meaning the equivalent of `Disable Density Check` is *not* selected.

For large layouts, the documentation suggests an optimization path: run only density checks and increase the density-thread count to speed up the verification phase.

One subtle note from the documentation is also worth preserving here: during minimal DRC in KLayout, `AFil.g2` and `M1-5Fil.h` errors may appear outside the actual design area where density is computed, and these can be safely waived.

Submission implication
Density checks are part of the rejection criteria in the submission flow. If the design does not meet the required density values, it may be rejected even if the circuit itself is otherwise correct.

13.6 Alternative Flow with `gdsfill`

The IHP documentation also points to `gdsfill`, an open-source filler tool that can analyze, erase, and generate dummy fill patterns across supported layers [6]. It is useful when a more automated or reproducible command-line flow is needed.

Typical commands include:

```
gdsfill density <my-layout.gds>
gdsfill erase <my-layout.gds>
gdsfill fill <my-layout.gds>
```

Two optional modes are especially useful:

- `-keep-data`: keep intermediate files in `gdsfill-tmp`
- `-dry-run`: simulate the fill process without modifying the GDS

The documentation also supports custom configuration files so that only selected layers are filled or specific density targets are used, for example on upper metal layers.

13.7 Recommended Finishing Sequence for This Project

For this comparator-oriented learning flow, a good finishing sequence is:

1. complete the layout and make it DRC-clean before filler insertion,
2. ensure the intended top-level context includes the mandatory sealring,
3. insert filler using the KLayout menu flow or the scripted filler flow,
4. run a density report,
5. re-run full DRC with density enabled,
6. if parasitics are expected to change meaningfully, regenerate the extracted view for final post-fill confidence.

This last point is easy to forget. Even though filler is not part of the active circuit, it can still alter coupling and density-related parasitic context. For a precision dynamic comparator, that means finishing should occur before the very final post-layout signoff pass.

14 Verification Mindset

Even though the workshop presentation does not yet expand the DRC, LVS, extraction, and full-chip sections in detail, the manual should keep them visible as mandatory closure steps rather than optional polish.

Use the following checklist before calling the comparator ready:

- Schematic is final enough that naming and hierarchy no longer change every hour.
- Transient operation is correct for both obvious and near-threshold input cases.
- Mismatch runs identify a reasonable offset spread and no suspicious asymmetry.
- Process and supply variation do not break reset or regeneration.
- Kickback has been inspected at the input network, not only at the outputs.
- Layout preserves symmetry, short clock routing, and balanced regenerative nodes.
- DRC and LVS are planned as part of the normal iteration loop.
- Parasitic extraction is expected to shift delay and kickback, so post-layout verification is part of the design, not an afterthought.

15 Self-Paced Learning Path

The most effective way to use this manual is to work in modules. Each module should end with artifacts and observations, not just completed drawings.

15.1 Module 1: Read the Architecture

- Redraw the signal flow of the StrongARM latch from memory.
- Identify which devices belong to sensing, regeneration, reset, and current control.
- Explain in writing why incomplete precharge can create dynamic offset.

Exit criterion: you can explain the role of `M1/M2`, `M3/M4`, `M5/M6`, `M7`, and the reset devices without looking at the article.

15.2 Module 2: Capture the Schematic

- Build the basic architectural schematic.
- Convert it into the final layout-compatible view.
- Confirm that all differential and regenerative nodes are clearly named.

Exit criterion: the schematic is readable, symmetric, and netlist-clean.

15.3 Module 3: Build the Transient Bench

- Create the clock source and differential input stimulus.
- Add the required PDK library statements and any supporting standard-cell includes.
- Measure at least one timing metric automatically.

Exit criterion: the comparator produces a correct logic decision and the measurement script runs without manual edits.

15.4 Module 4: Study Regeneration

- Apply a moderate differential input and inspect the order in which internal nodes move.
- Reduce the input difference step by step and observe the delay growth.
- Record which waveform features mark the start of strong regeneration.

Exit criterion: you can point to the waveform region where the circuit changes from weak analog imbalance to strong positive feedback.

15.5 Module 5: Study Mismatch

- Enable mismatch on the core matched pairs.
- Run repeated trials near zero differential input.
- Separate random spread from obvious systematic asymmetry.

Exit criterion: you can explain why the input pair dominates offset and what the simulation suggests about statistical robustness.

15.6 Module 6: Study Process and Supply Variation

- Sweep corners or use the statistical model section if the flow supports it.
- Compare worst-case reset quality, delay, and output swing.
- Identify which corner appears most dangerous and why.

Exit criterion: you know which global variations stress the design most strongly.

15.7 Module 7: Study Kickback

- Measure or plot the input disturbance during switching.
- Distinguish common-mode from differential kickback behavior.
- Relate the result to the likely source impedance of the preceding stage.

Exit criterion: you can judge whether a fast comparator is still acceptable once its input disturbance is considered.

15.8 Module 8: Build the Layout

- Place the differential pair with symmetry as the first priority.
- Add the cross-coupled devices with balanced interconnect.
- Route the clock compactly and consistently.
- Compare the final layout against the final schematic and expected current paths.

Exit criterion: the layout clearly reflects the matching strategy rather than merely fitting inside an area box.

16 Common Failure Modes

The following issues are worth checking early because they can consume many hours if discovered too late:

- reset devices too weak, leaving residual charge and dynamic offset,
- output loads unintentionally asymmetric, masking the real offset behavior,
- tail device widened for speed without rechecking kickback,
- larger matched devices added for offset reduction without reevaluating capacitance and delay,
- beautiful schematic but poor node naming, causing trouble in LVS and debug,
- local layout asymmetry on the regenerative nodes, creating deterministic bias.

17 Headroom for improvements

The current comparator is a solid educational StrongARM implementation, but it still leaves clear room for architectural and circuit-level improvement. This is normal. The StrongARM latch is often the first useful dynamic comparator a designer learns, and it is also the first one that exposes the speed, offset, noise, and kickback trade-offs very directly.

An especially useful reference here is the work by Wicht, Nirschl, and Schmitt-Landsiedel [4]. That paper separates the operation into reset and integration phases very clearly and uses that split to reason about yield, speed, offset, and noise. It is highly recommended reading alongside the Razavi articles because it turns several intuitive observations into explicit optimization rules.

17.1 Input-Pair Sizing and Offset Margin

From the current implementation notes, the differential pair appears relatively small, on the order of `8 x (1um / 0.26um)` per side. That may be acceptable for a high-speed comparator, but it also suggests that the input-referred offset could be nontrivial.

The right question is not whether the pair is "small" in isolation. The right questions are:

- what is the simulated offset distribution,
- what is the estimated or simulated 3σ offset,
- is that offset acceptable for the intended application,
- if not, is a preamplifier needed to attenuate the latch-referred offset?

If the comparator is intended mainly for speed, the smaller input pair may be justified. If offset is a first-order requirement, then the present sizing should be challenged with explicit Monte Carlo data rather than accepted on intuition.

17.2 Kickback at the Tail-Release Boundary

Kickback is already discussed earlier in this manual, but the comments add an important nuance: the reset-to-evaluation transition at the tail path can itself become a dangerous disturbance source. A larger differential pair often helps mismatch, but it can also make the input network more susceptible to charge injection and false decisions coming out of reset.

In practice, that means the following mitigations deserve extra attention:

- keep the differential input routing short and balanced,
- match the input impedances seen by the two comparator inputs,
- avoid unnecessary asymmetry between the sampled source paths,
- consider using a preamplifier when the source is weak or the kickback budget is tight.

This is one of the recurring mixed-signal lessons of the StrongARM latch: the same design choice that improves one metric can worsen another.

17.3 How to Increase Early Gain

The latch noise and offset are still dominated mainly by the NMOS input pair. Their input-referred effect is reduced when the comparator develops more gain before full regeneration takes over. A useful design heuristic is therefore to maximize the early gain term, which is qualitatively proportional to:

$$\frac{g_{m,\text{diff pair}}}{I_{\text{tail}}} \tag{2}$$

This observation points to an important optimization target: keep the differential pair in saturation for as long as possible during the integration phase. That is one reason PMOS pull-up devices on the differential-pair drains are attractive. They help preserve operating headroom and can also reduce sensitivity to input common-mode variation.

17.4 Dynamic-Offset Improvement

One practical refinement proposed in the comments is the addition of a reset switch across the drains of the differential pair. The reason is subtle but important. Even if the PMOS pull-ups are nominally symmetric, Monte Carlo variation can make their reset action slightly unequal. That residual imbalance can appear as dynamic offset in the next evaluation cycle.

For a more thorough implementation, the designer should therefore consider:

- verifying how well the two drain nodes equalize during reset,
- checking whether mismatch in the PMOS pull-ups leaves a residual drain difference,
- evaluating whether a dedicated equalization or reset switch across the drains reduces dynamic offset without creating unacceptable parasitic cost.

This is exactly the kind of refinement that is easy to overlook in a first-pass design and very worthwhile in a more mature version.

17.5 Fundamental Architecture Limitation

The strongest architectural caution in the comments is also one of the most important. In the basic StrongARM latch, the sense amplifier action and the latch regeneration use the same current. This couples gain and speed very tightly:

- reducing tail current can improve gain and help noise or offset,
- increasing tail current helps speed,
- but the two goals fight each other because they are driven by the same current path.

This means the architecture cannot be pushed arbitrarily toward both very low noise or offset and very high speed at the same time. That does not make it a bad architecture. It simply defines where its performance frontier is.

Design interpretation
If the application needs a moderate-speed comparator with clean dynamic behavior and a compact implementation, the StrongARM latch remains an excellent choice. If the application needs both very high speed and aggressively low input-referred error, then more advanced latch families may become preferable.

The comments specifically point toward Schinkel-type and Elzakker-type latches as better candidates for higher-performance sigma-delta style applications. Even if those architectures are not implemented in this project, it is useful to understand *why* they exist: they relax the direct speed-versus-gain trade-off that is inherent in the classical StrongARM form.

17.6 Recommended Next Improvements

If this manual evolves into a second design iteration, the following upgrades would provide the highest learning value:

1. quantify input-referred offset explicitly, including a 3σ value,
2. compare small and enlarged input-pair options while tracking delay and kickback,
3. simulate a drain equalization switch and measure its impact on dynamic offset,
4. inspect the common-mode dependence of the comparator with and without stronger PMOS pull-up assistance,
5. benchmark the present StrongARM latch against a preamplified or alternative latch architecture.

18 Conclusion

The StrongARM latch is a compact circuit, but not a trivial one. It is a useful teaching vehicle because nearly every important analog mixed-signal concern appears inside one cell: mismatch, dynamic offset, time-domain gain, positive feedback, metastability, kickback, and layout-sensitive symmetry.

The best way to use this manual is iteratively. Read one section, modify the schematic or testbench, run a focused simulation, and write down what changed and why. If that loop is repeated carefully, the learner does not merely finish a comparator. The learner acquires a design method.

References

[1] B. Razavi, *The Design of a Comparator*, IEEE Solid-State Circuits Magazine, Fall 2020.

[2] B. Razavi, *The StrongARM Latch [A Circuit for All Seasons]*, IEEE Solid-State Circuits Magazine, vol. 7, no. 2, pp. 12–17, Spring 2015.

[3] M. J. M. Pelgrom, A. C. J. Duinmaijer, and A. P. G. Welbers, *Matching Properties of MOS Transistors*, IEEE Journal of Solid-State Circuits, vol. 24, no. 5, pp. 1433–1440, Oct. 1989.

[4] B. Wicht, T. Nirschl, and D. Schmitt-Landsiedel, *Yield and Speed Optimization of a Latch-Type Voltage Sense Amplifier*, IEEE Journal of Solid-State Circuits, vol. 39, no. 7, pp. 1148–1158, July 2004.

[5] IHP Open PDK Documentation, *Physical & Design Verification*, https://ihp-open-pdk-docs.readthedocs.io/en/latest/verification.html.

[6] IHP Open PDK Documentation, *Chip Finishing: Filler Generation*, https://ihp-open-pdk-docs.readthedocs.io/en/latest/finishing/filler.html.

[7] IHP Open PDK Documentation, *SG13G2 Open Source Layout Rules Manual*, https://github.com/IHP-GmbH/IHP-Open-PDK/blob/main/ihp-sg13g2/libs.doc/doc/SG13G2_os_layout_rules.pdf.