Northwestern University

Department of Electrical and Computer Engineering

Computer Engineering 493: Advanced Low Power Digital and Mixed-Signal Integrated Circuit Design

Drift and Noise Resilient Mixed-Signal Back End for On-Chip Magnetic Field Sensing

Can Afacan & Bemnet Tefera

Dec 13, 2025

Table of Contents

Abstract

This project presents a drift and noise resilient mixed-signal back-end architecture for on-chip magnetic field sensing. It puts together a Verilog-A Hall effect model, a low-noise analog front end, and a digitally calibrated signal processing chain. The analog subsystems, implemented in a GPDK 45nm technology, provide an 8-bit SAR ADC shared across the three sensor axes through a 3:1 analog multiplexer. To combat offset drift, temperature-dependent gain error, and flicker noise inherent in Hall and MTJ-based sensors, the digital backend performs startup calibration, real-time oversampling, exponential moving average drift tracking, and tri-axis vector assembly and single-pin serialized output. The result is a compact and fully integratable magnetic sensing solution that maintains long term accuracy and stability under environmental and device-level variations, allowing for robust operation in SoC-Embedded sensor platforms.

Introduction

Magnetic field sensors based on the Hall effect and Tunnelling magneto-resistance (TMR/MTJ) are powerful building blocks for position, current, and field sensing applications, but their accuracy is fundamentally limited by slow offset drift, temperature-dependent sensitivity, and low-frequency flicker noise. These effects cause gradual deviations in the measured field even when the true magnetic environment is constant, forcing traditional systems to rely on periodic recalibration or power-hungry analog filtering. As modern mixed-signal systems increasingly demand compact and long-life sensor interfaces, there is a growing need for an on-chip signal conditioning path that can continuously maintain measurement integrity over time. To address this challenge, our project assesses a drift and noise resilient mixed single back end for a 3-axis Hall Effect Magnetic sensor. The approach combines a physically inspired Verilog-A model of the Hall device with a fully custom mixed-signal chain implemented in the 45nm process. The analog front end provides high-resolution digitization using an 8-bit SAR ADC shared across the X, Y, and Z axes through an asynchronous 3:1 Analog Multiplexer. This structure allows for minimized power and area while supporting rapid per-axis sampling.

On the other hand, the digital subsystem ensures stability by operating on time-tagged ADC samples delivered sequentially from the external analog front end. Incoming samples are first demultiplexed into per-axis streams and converted into a signed fixed-point representation. Each axis then undergoes oversampling and moving-average decimation to improve signal-to-noise ratio, followed by a startup offset calibration stage that removes static bias introduced by the sensor and ADC interface. To address low-frequency drift and flicker-like behavior, the digital backend employs a slow exponential moving-average baseline tracker whose estimate is subtracted from the signal, suppressing long-term wander while preserving true magnetic field variations. The resulting drift-suppressed axis values are optionally gain-scaled and registered to form a coherent three-axis magnetic field vector. The final vector is serialized and streamed off-chip using a UART-style single-bit output, significantly reducing I/O complexity while maintaining full precision. Together, these digitally assisted operations form a compact, low-power, and self-stabilizing magnetic sensing backend that complements the precision of the analog front end and enables accurate tri-axis field readout under realistic noise and drift conditions.

Applications

Magnetic field sensors based on Hall-effect and magnetic tunnel junction (MTJ) devices have demonstrated utility across automotive, biomedical, and industrial domains. In automotive systems, Hall sensors are widely used for current monitoring, wheel speed detection, and rotor position tracking, where offset drift and thermal cycling pose significant challenges. Bilotti *et al.* showed that dynamic quadrature offset cancellation techniques can mitigate these issues in monolithic Hall sensor ICs, improving reproducibility and long-term stability [1]. Similarly, Crescentini *et al.* highlighted the role of Hall-effect current sensors in power electronics and smart building applications, emphasizing their integrability with CMOS technologies and low power consumption [2]. These findings align with the proposed mixed-signal backend's emphasis on startup calibration and drift tracking, ensuring reliable operation under environmental variations.

MTJ-based sensors extend these capabilities into biomedical and industrial contexts. Hao *et al.* reported on MgO-based MTJ devices for integrated current measurement, demonstrating their potential for high-sensitivity detection at the circuit level [3]. Reviews of MTJ applications further underscore their suitability for biomagnetic diagnostics such as magnetocardiography and magnetoencephalography, where weak biological signals demand low-noise architectures [4]. Industrial robotics and navigation also benefit from MTJ sensors' precision in magnetic field mapping and robustness against electromagnetic interference. By combining a low-noise analog front end with digitally calibrated signal processing, the proposed architecture reflects system-level advances in mixed-signal embedded sensing [5], positioning it as a versatile solution across automotive, biomedical, consumer, and industrial platforms.

Section I. Analog and Mixed Signal IC Design

In this section, we go through the front-end design of the Analog to Digital Conversion block alongside some additional blocks used to interact with the Magnetic Sensor.

The integrated circuit design flow of a Hall Sensor would typically involve Chopper Stabilization, Gain Amplification, and an Anti-Aliasing Filter. However, in this project, we will bypass that aspect with a Verilog-A module that provides those functionalities and then feeds into the physically realized aspect of the Mixed-Signal IC section. This is a conscious decision we made based on the fact that the Virtuoso Design Suite doesn't include a native magnetic field simulator. Therefore, we will have a three-axis buffered output with AC swing and phase difference to verify the functionality of our ADC. Please refer to the implementation of the Hall Effect Sensor in the Verilog-A environment.

1. 3:1 Analog Multiplexer

The Hall Effect Sensor has a 3 Channel output, each dedicated to one of the three axes analog outputs. With this in mind, it should be trivial that using one ADC block per output channel is very inefficient both in terms of power and area consumption. So to mitigate this, we use a 3:1 Analog Multiplexer with a 2-bit selector, which only allows the conduction of input signals from one channel while the other channels are set to their high impedance state. The digital select signals are decoded into three mutually exclusive enable lines to guarantee only one condition path is active at a time, which minimizes loading and crosstalk between channels. These selection bits are generated sequentially and fed to both the analog mux and the digital backend to request sampling of a specific axis, ensuring that only the desired channel is connected to the ADC during each conversion cycle.

1.1 Circuit Schematic and Testbench

We have constructed this block using three transmission gates, each receiving a selection signal from a logic that is realized as shown in the figure below (Figure 1.1). In the top left corner, we have two inverters, each to invert the selection signals. Following that, we have three AND2X1 gates, where each is receiving the logic S_0.'S_1', S_0.'S_1, and S_0.S_1' top to bottom, respectively. This ensures that no two channels are sampled simultaneously. The output from these three AND gates is then followed by a digital buffer (Figure 1.2) that outputs the buffered and inverted binary state of the AND gate output. This creates the complementary pulse needed to switch the transmission gate (Figure 1.2).

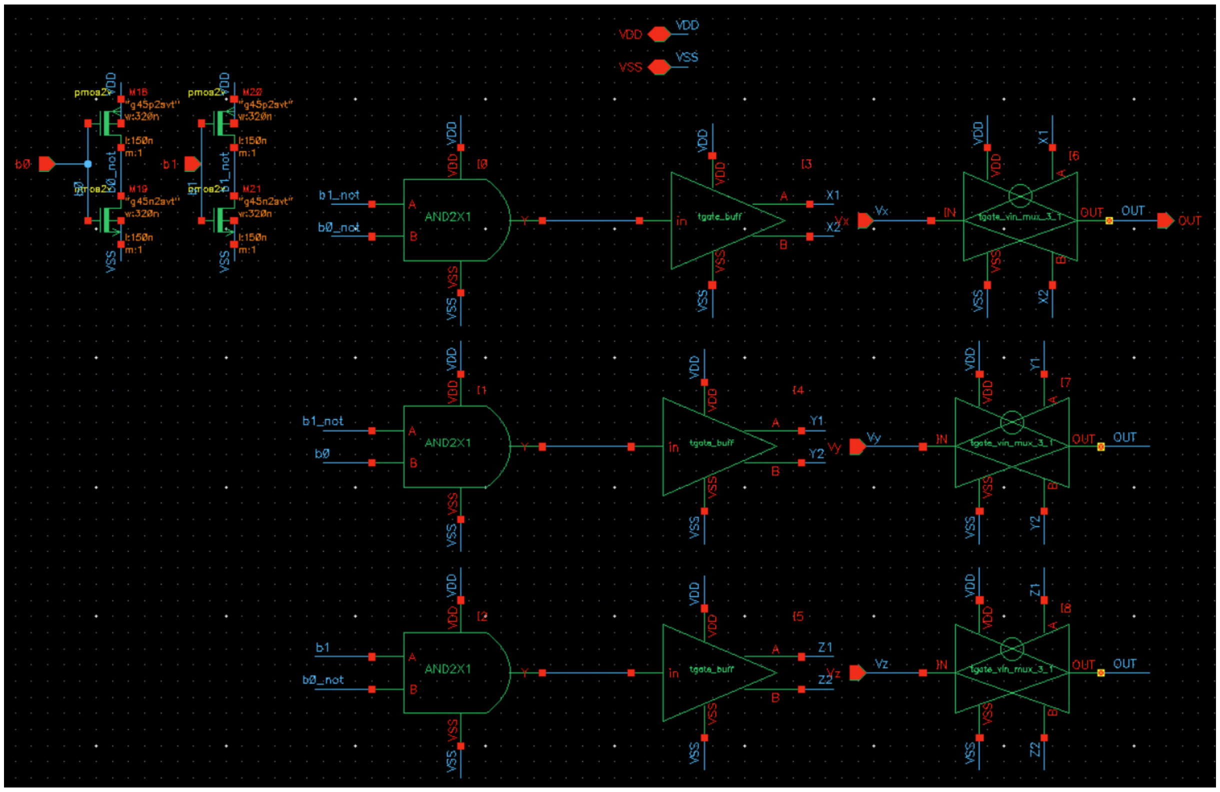
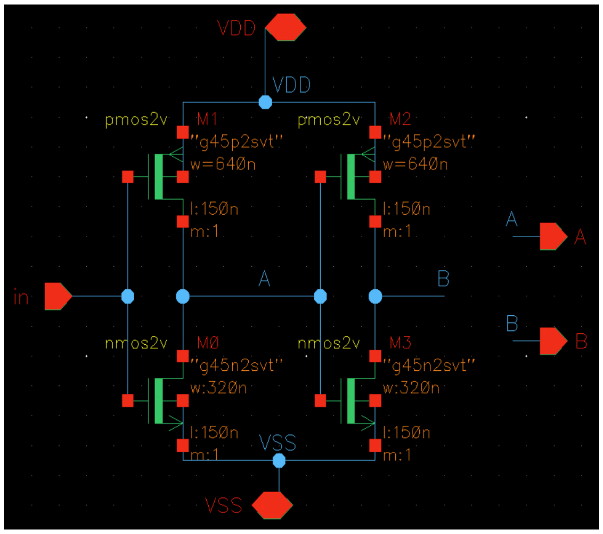


Figure 1.1*: Implementation of a 3:1 Analog Multiplexer for sensor output channel selection.*



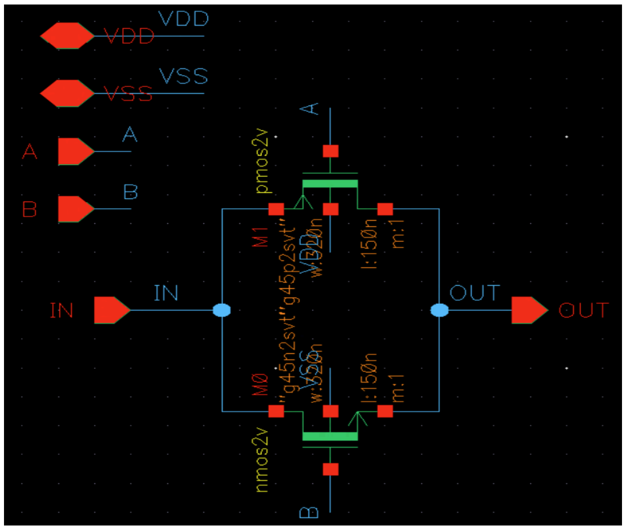


Figure 1.2: *Transmission Gate Driver for complementary differential gate control signal (Left).* *The transmission gate used in the 3:1 Analog Multiplexer shown in Figure 1.1. (Right)*

The transmission gate was sized 0.35µ/0.15µ for the CMOS devices such that charge injection and clock feedthrough are minimized by the resulting capacitance. On the other hand, the Transmission Gate driver is also sized such that the slew on the rising and falling edges is well balanced to avoid a mismatch at the transmission gate.

To verify the operation of this circuit, we use three sinusoidal signals, each with different frequencies, amplitudes, and dc offset, feeding into the input of the analog mux. The schematic testbench is shown in the figure below (Figure 1.3).

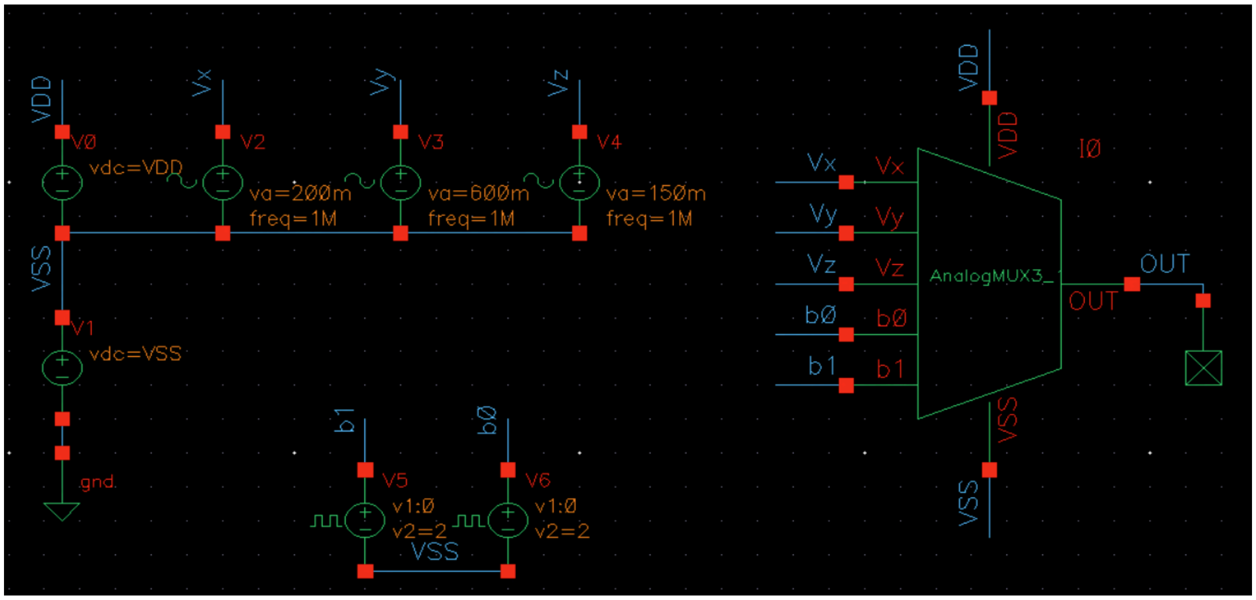


***Figure 1.3**: Schematic testbench organization of the 3:1 Analog multiplexer.*

The Vx, Vy, and Vz voltage sources output a sinusoidal signal, where:

Vx: DC_{offset} = 1.2V, Amplitude = 200mV, frequency = 3MHz

Vy: DC_{offset} = 1.4V, Amplitude = 600mV, frequency = 2MHz

Vz: DC_{offset} = 1.0V, Amplitude = 200mV, frequency = 1MHz

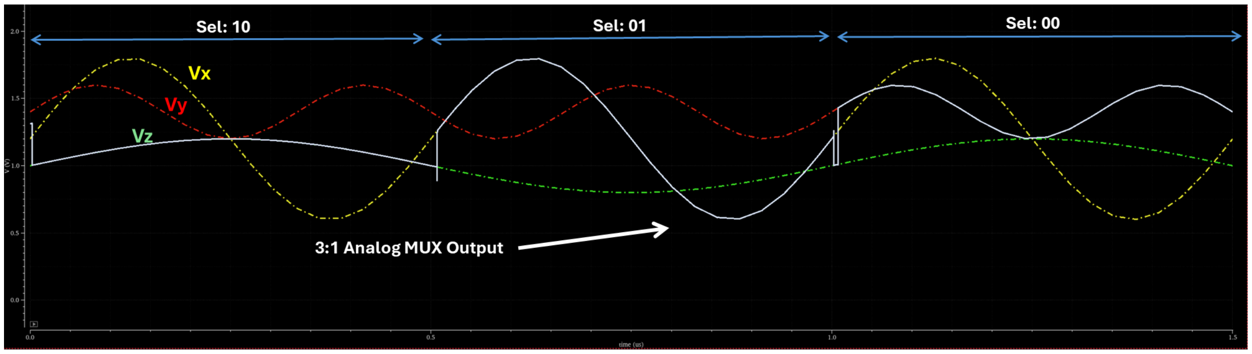


***Figure 1.4**: Output plot of the 3:1 Analog Mux testbench showing input tracking performance. This transient simulation was run for 1.5 microseconds with a conservative step size.*

The signal labeled '3:1 Analog Mux output' shows how well the input signal is tracked by the Analog Multiplexer. With this in hand, we design the layout with the proper layout techniques.

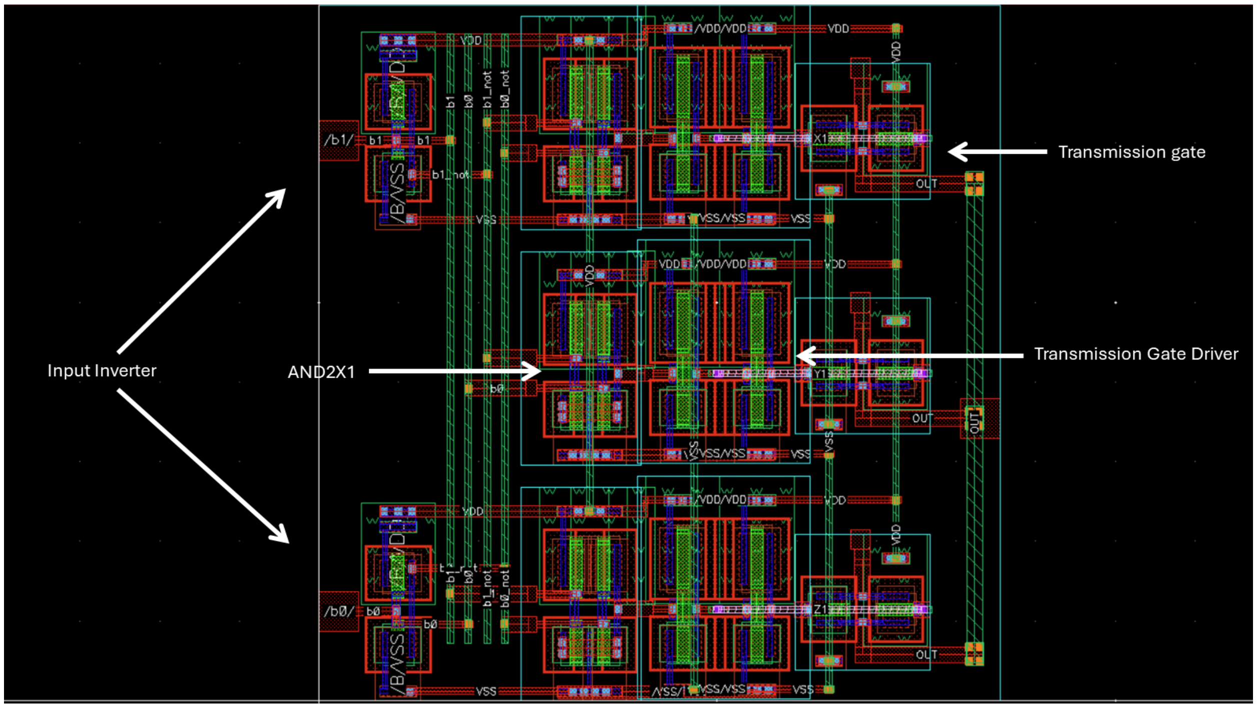


Figure 1.5: *Layout implementation of the 3:1 Analog Multiplexer.*

The RC-Extracted performance of this block shows no deviation from the schematic testbench, and this can be observed below in Figure 1.6. Transient, AC, and DC performance simulation results will be assessed thoroughly in later sections.

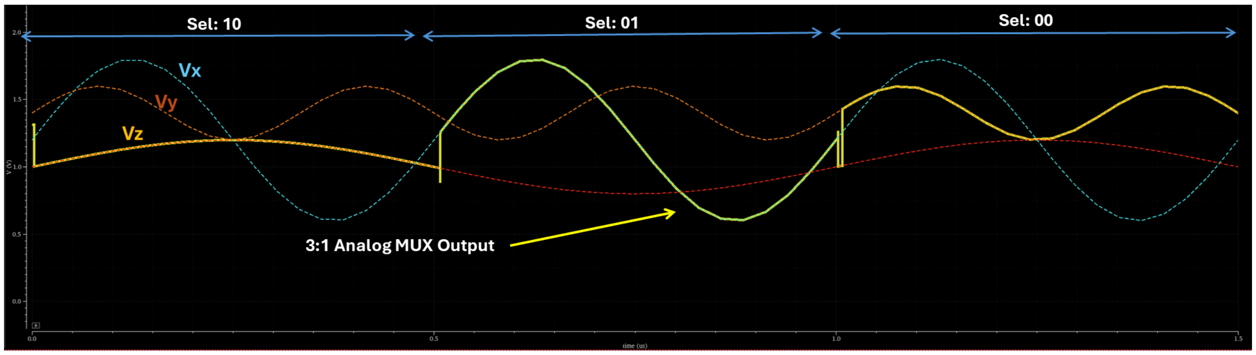


Figure 1.6: *Output plot of the 3:1 Analog Mux RC Extracted showing input tracking performance.*

2. Two-Stage Miller Compensated Op-Amp with Nulling Resistor

A typical low-power SAR ADC uses a capacitive DAC array in which the sampled input and the DAC trial voltages are integrated onto the same capacitor network. In this project, however, we intentionally depart from that approach. To gain deeper experience with fundamental analog building blocks, we chose to implement an Op-Amp-based Sample-and-Hold followed by an R-2R DAC and a two-stage Miller-compensated operational amplifier with a nulling resistor. This architecture allows us to more clearly explore and demonstrate key analog IC concepts such as settling behavior, linearity, compensation, and the interaction between analog blocks within a mixed-signal system.

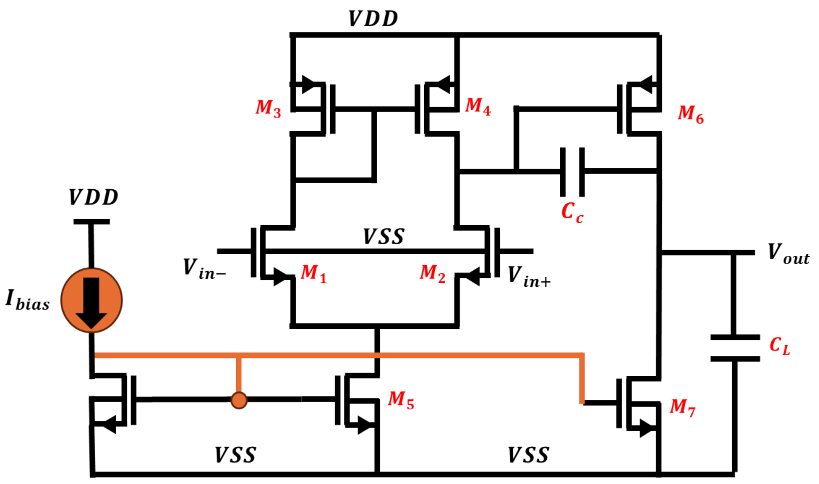


Figure 1.7: *Schematic of a two stage miller compensated Op Amp*

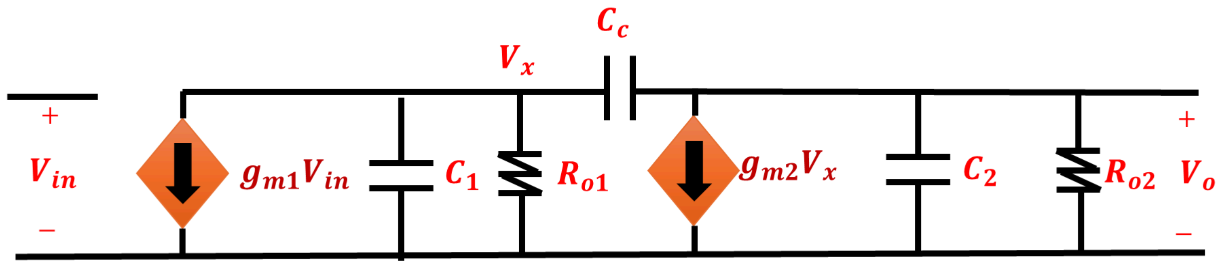


Figure 1.8: *Small signal analysis for a two-stage mille- compensated Op-Amp.*

To keep the mathematical derivation flow straightforward, the transfer function of the small signal representation shown above is given by:

$$\frac{V_o}{V_{in}} = \frac{gm_1 R_1 gm_2 R_2 (1 - SC_C/gm_2)}{S^2(R_1 R_2 (C_1 C_2 + C_1 C_C + C_2 C_C) + S(R_2(C_C + C_2) + R_1(C_C + C_1) + C_C gm_2 R_1 R_2)) + 1}$$

For a standard two-pole system, the typical transfer function is given as follows:

$$\frac{V_o}{V_{in}} = \frac{A_{DC}(1-S/2)}{(1+S/P_1)(1+S/P_2)} = \frac{A_{DC}(1-S/2)}{(1+S(\frac{1}{p1}+\frac{1}{p2})+S^2(\frac{1}{P_1P_2}))};\ A_{DC} = \text{DC gain}$$

We then apply concepts from control theory to obtain performance metrics such as pole-zero location, slew rate, DC gain, Phase and Gain Margin, and unity gain frequency. Refer to Table 1.1 below:

Design Metrics	**Relationship**
Slew Rate	$SR = I_5/C_C$
First-Stage Gain	$A_{V1} = gm1(ro1 \,\|\|\, ro3)$
Second-Stage Gain	$A_{V2} = gm2(ro6 \,\|\|\, ro7)$
DC Gain	$A_{dc} = gm1gm2(ro1 \,\|\|\, ro3)(ro6 \,\|\|\, ro7)$
Gain-Bandwidth	$GBW = gm_1/2\pi C_C$
Dominant Pole	$\omega_{p1} \approx GBW/A_{dc}$
Non-Dominant Pole	$\omega_{p2} \approx gm_2/C_1$
LHP Zero (Nulling Resistor)	$\omega_z = 1/R_n C_C$
Optimal Rn	$R_n \approx 1/gm_1$
Unity-Gain Frequency	$\omega_u = gm_1/C_C$
60° Phase Margin Condition	$gm_2 C_C \geq 2gm_1 C_1$
Phase Margin Expression	$PM = 180 - tan^{-1}(\frac{\omega_u}{\omega_{p2}}) + tan^{-1}(\frac{\omega_u}{\omega_z})$

***Table 1.1**: Related physical parameters that affect the overall performance of the OpAmp.*

Through the utilization of the parameters above, we implement an Op-Amp as shown in the figure below.

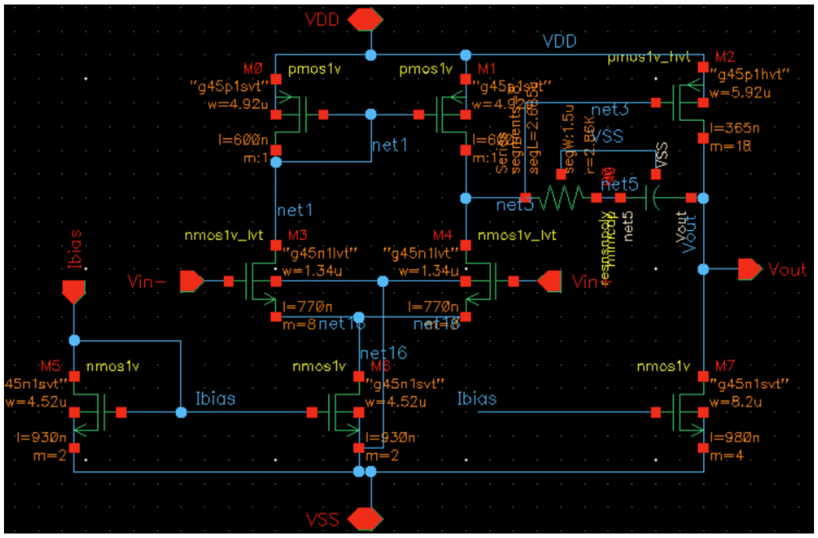


Figure 1.9: *Schematic design of the Two-Stage Miller Compensated Operational Amplifier.*

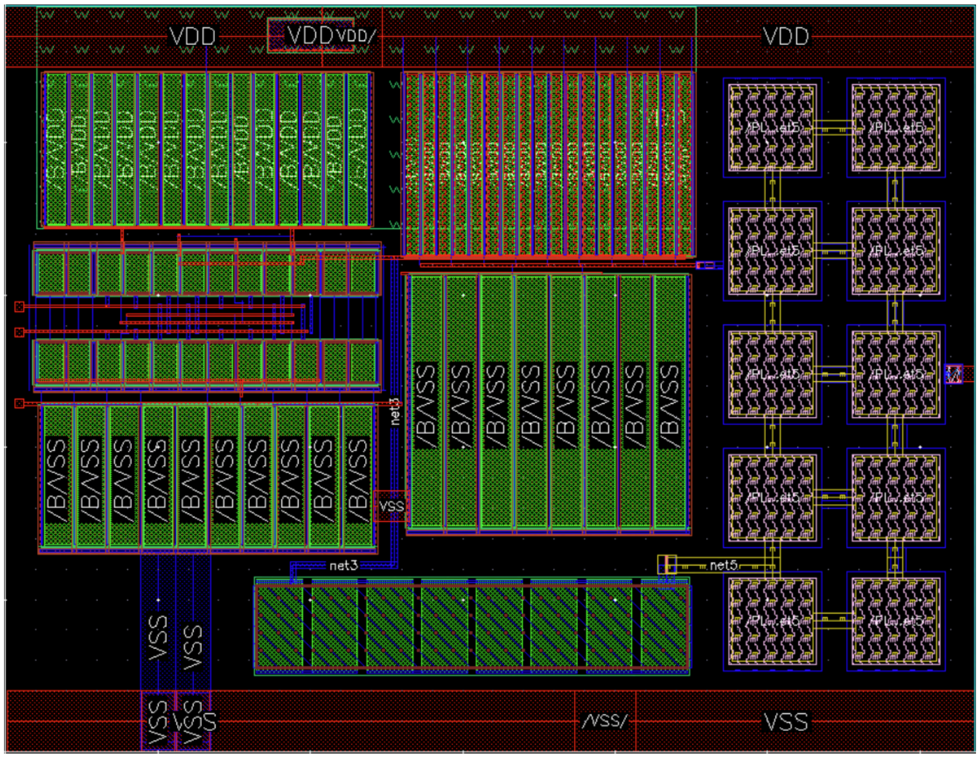


Figure 1.10: *Layout design of the Two-Stage Miller Compensated Operational Amplifier.*

The performance metrics we impose on this Op-Amp 12 include:

- ➔ **VDD**: 1.8V @ 125°C Slow-Slow Corner
- ➔ **Slew Rate**: 270V/µs
- ➔ **Unity Gain Frequency**: > 250MHz
- ➔ **DC Offset**: < 1.5mV (~ ½ LSB)
- ➔ **Stability Requirements**: >60° PM & >12dB GM
- ➔ **ICMR$_{min}$**: 0.8V
- ➔ **ICMR$_{max}$**: 1.6V
- ➔ **Bias Current: 50**µA

2.1 Verification of the Op-Amp

For the sake of presenting an uncluttered result, we report the RC-extracted transient and stability performance for the three corners.

2.1.1 Transient Analysis

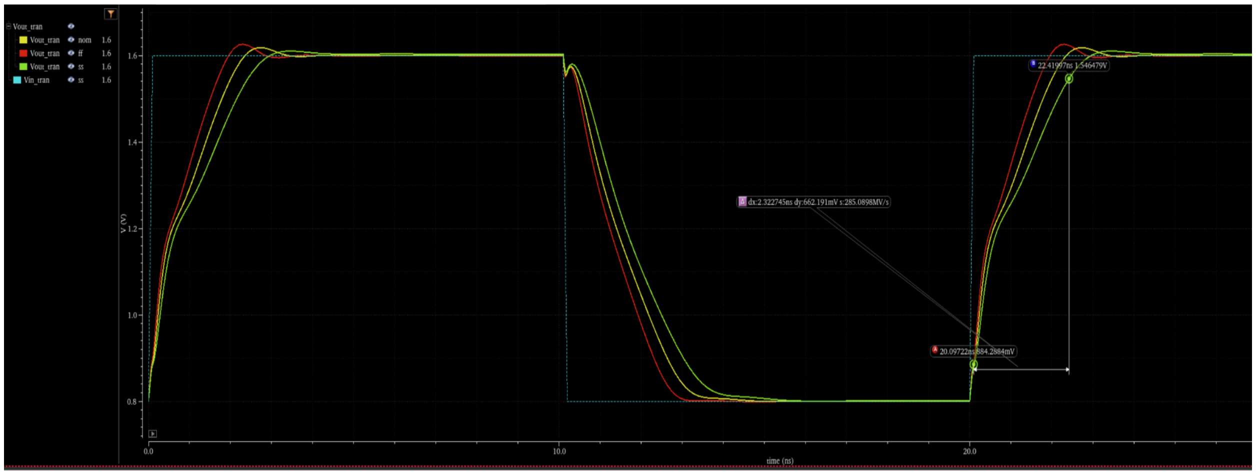


***Figure 2.1**: Transient Analysis of the Op-Amp over the three corners.*

The transient response above was run for 25ns with a square wave modulating between 0.8v and 1.6v at 50MHz, and the worst-case performances are extracted as follows:

- ➔ Slew Rate: 285 MV/s
- ➔ ICMRmin: 0.8V
- ➔ ICMRmax: 1.6V
- ➔ Settling time: 3.126ns
- ➔ Slew rate (rising): 293.2MV/s
- ➔ Slew rate (falling): 355.1.2MV/s
- ➔ Powerdiss, max: 521µW

2.1.2 Stability Analysis

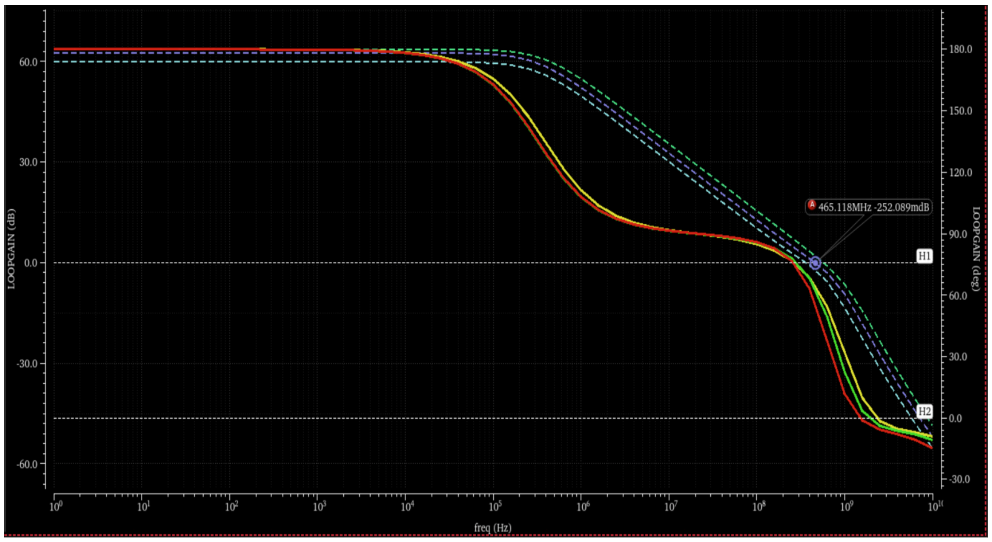


***Figure 2.2**: The RC-Extracted stability analysis result of the operational amplifier for a frequency sweep of 10Hz to 10GHz.*

Via the utilization of the calculator tool in the ADE, we extract and report the worst PVT corner case performance for this Op-Amp. Results are given below:

- ➔ Unity Gain Frequency: 290.5M
- ➔ Phase Margin: 57.46
- ➔ Gain Margin: 20.23
- ➔ DC Gain: 62.52dB

3. Sample and Hold (S/H)

The sample and hold is the first block in the Analog to Digital Conversion pipeline, as it is responsible for sampling the analog input voltage and holding it long enough for the successive approximation registry (SAR) block to carry out its approximation algorithm. It is made up of a CMOS transmission gate similar to the one implemented in the previous section, followed by a hold capacitor, which stores the sampled voltage. This node then feeds into a two-stage miller compensated Operational Amplifier in a unity gain configuration. The Op-Amp includes a dedicated feedback capacitor to stabilize the closed-loop response, charge redistribution transients, and ensure high linearity when we later drive the comparator input.

3.1 Circuit Operation

During the sample phase, the input transmission gate and the switch across the feedback capacitor are turned on. This allows the hold capacitor to directly track the input signal, and the Op-Amp behaves like a simple unity gain buffer. Both capacitors settle to the input voltage during this time.

When the sample signal falls, the circuit enters the hold phase. Both switches turn off, isolating the hold capacitor so it retains the sampled voltage. The Op-Amp then holds and buffers this voltage while the SAR block performs its binary search conversion. The key requirement during this phase is that the held voltage remains stable long enough for the entire SAR operation.

3.2 Verification of Circuit Operation

To test the overall performance of the sample and hold, we sweep through the overall input common mode range, that is, 0.8V to 1.6V with increments of 0.2V, and see the acquisition time, drop rate, and hold offset performance metrics.

The plot below is the output for a set ramp input that is incrementing from 0.8V to 1.6V for 200ns and is simulated over the nominal, best, and worst case corners. The sample frequency is 50MHz (refer to the green square wave)

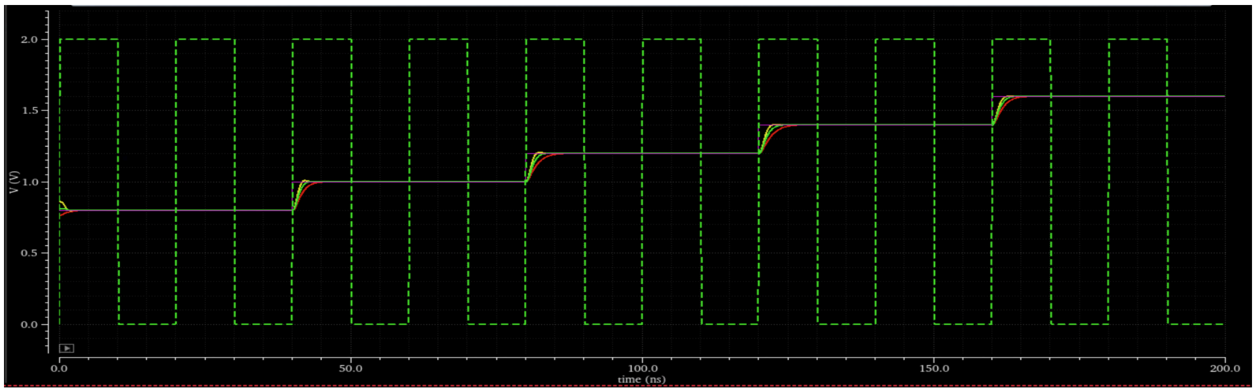
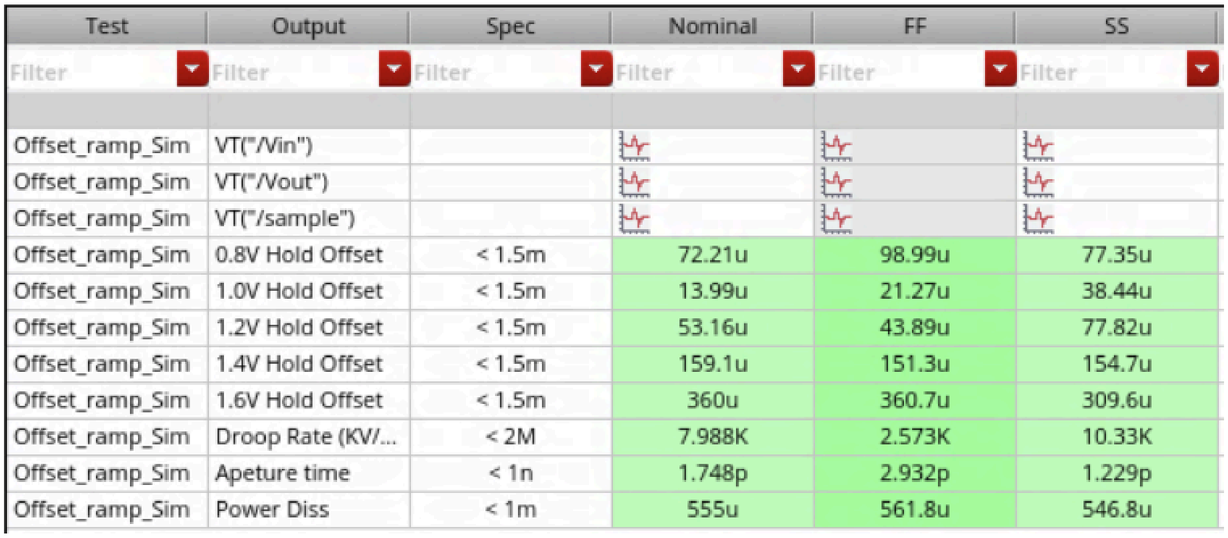


***Figure 3.1**: RC Extracted testbench result of the sample and hold block, where sampling happens on the rising edge.*

To further make the results we see above clearer, we utilize the ADE tool alongside the calculator to extract the worst-case performance parameters, as shown in the Maestro view screenshot below.



Test	Output	Spec	Nominal	FF	SS
Filter	Filter	Filter	Filter	Filter	Filter
Offset_ramp_Sim	VT("/Vin")				
Offset_ramp_Sim	VT("/Vout")				
Offset_ramp_Sim	VT("/sample")				
Offset_ramp_Sim	0.8V Hold Offset	< 1.5m	72.21u	98.99u	77.35u
Offset_ramp_Sim	1.0V Hold Offset	< 1.5m	13.99u	21.27u	38.44u
Offset_ramp_Sim	1.2V Hold Offset	< 1.5m	53.16u	43.89u	77.82u
Offset_ramp_Sim	1.4V Hold Offset	< 1.5m	159.1u	151.3u	154.7u
Offset_ramp_Sim	1.6V Hold Offset	< 1.5m	360u	360.7u	309.6u
Offset_ramp_Sim	Droop Rate (KV/...	< 2M	7.988K	2.573K	10.33K
Offset_ramp_Sim	Apeture time	< 1n	1.748p	2.932p	1.229p
Offset_ramp_Sim	Power Diss	< 1m	555u	561.8u	546.8u

Figure 3.2: *Maestro view simulation result record from the testbench setup in Figure 2.1.*

From this result, we learn that the worst-case hold offset happens when holding a 1.6V input signal, and even then, the weight is given by 0.115 LSB. The droop rate, on the one hand, shows us that we have a 10.33KV/s loss; however, that is not a challenge at all since it would take much longer than the sample and hold period to even lose 1 LSB.

4. Digital to Analog Converter (DAC)

The 8-bit DAC used in this SAR ADC architecture is implemented using an R-2R resistor ladder followed by a two-stage Miller compensated Op-Amp configured as a unity gain buffer. Its role is to generate precise reference voltages corresponding to each digital code during the binary search process. Because the DAC directly affects ADC resolution, monotonicity, and linearity, the resistor ladder and switching network are laid out with careful matching and symmetry. Refer to the figures below.

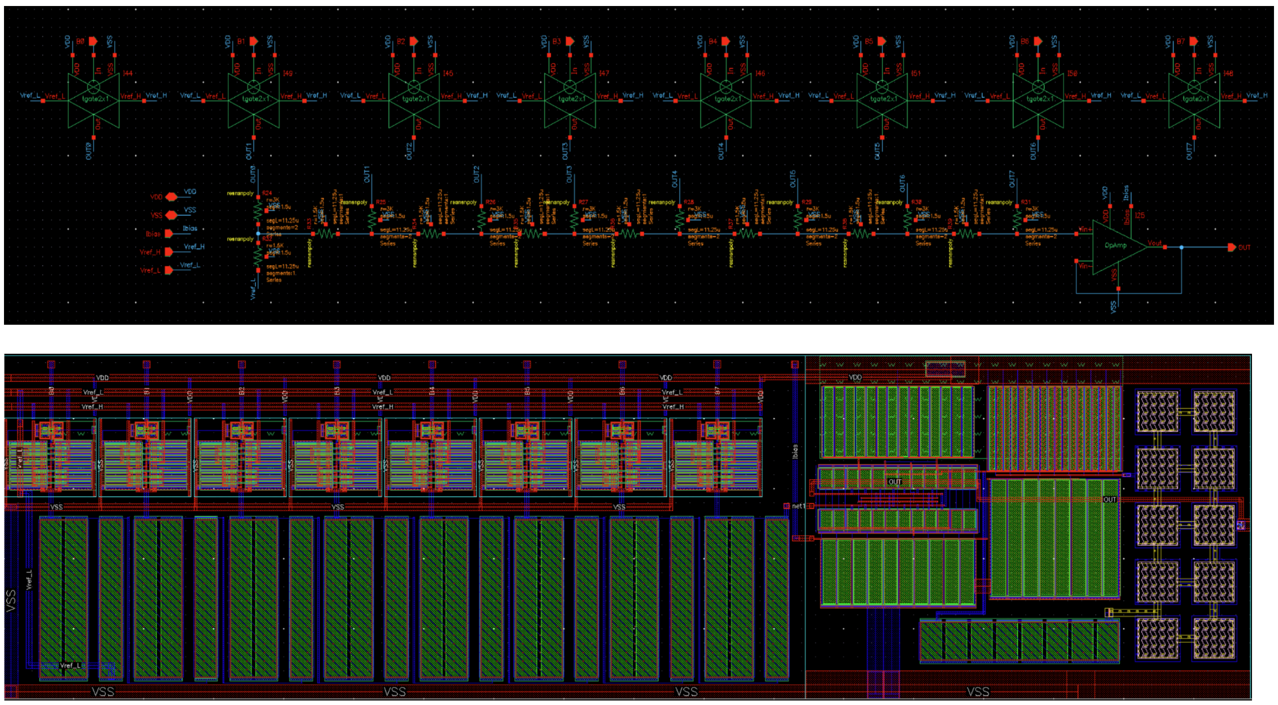


***Figure 4.1**: Schematic and Layout Implementation of 8-bit R-2R DAC with level shifting transmission gates: VDD = 2V, ICMRmin= 0.8V, ICMRmax= 1.6V.*

4.1 Circuit Operation

It is trivial that the linear operation range input of the DAC is limited by the ICMR of the Op-Amp. Therefore, we set the reference voltages for this block to $V_{ref,\ high} = 1.6V$ and $V_{ref,\ low} = 0.8V$. Now, we notice an issue with the way the DAC interacts with the Successive Approximation Registry block (SAR), as it is a digital block that can only output a 0V to 2V. To fix this issue, we have the level-shifting transmission gate in place. So now the 2V input coming into the DAC is leveled down to 1.6V, and the 0V is leveled up to 0.8V.

4.2 Verification of Circuit Operation

To verify the correct operation of this circuit, we use an 8-bit counter that increments from a decimal value of 0 to 265. This is done for the Nominal and the two extreme corners.

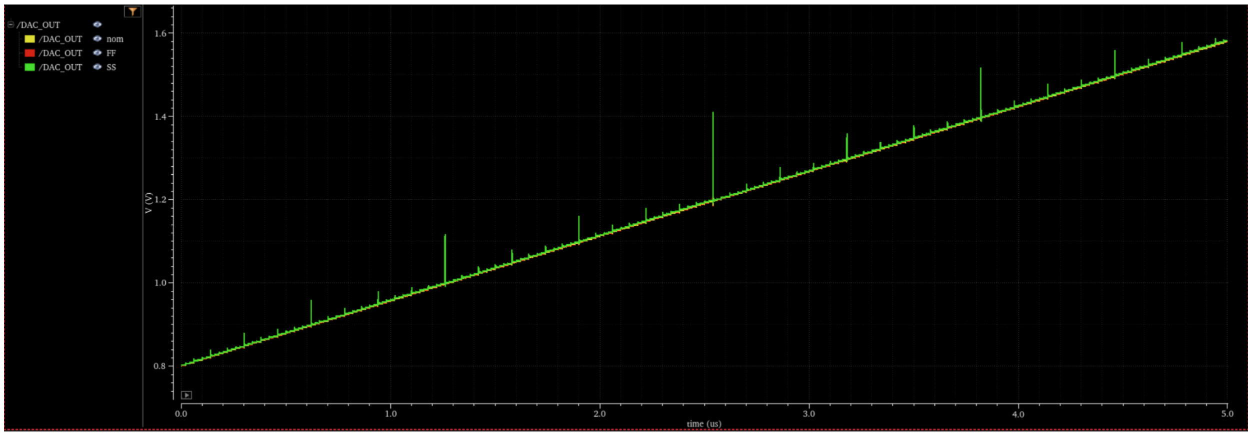


***Figure 4.2**: DAC simulation output for an 8-bit digital counter running on a 50MHz clock.*

We then extract this data and use the matplot.lib tool in the Python module to present the integral nonlinearity (INL) alongside the differential nonlinearity (DNL).

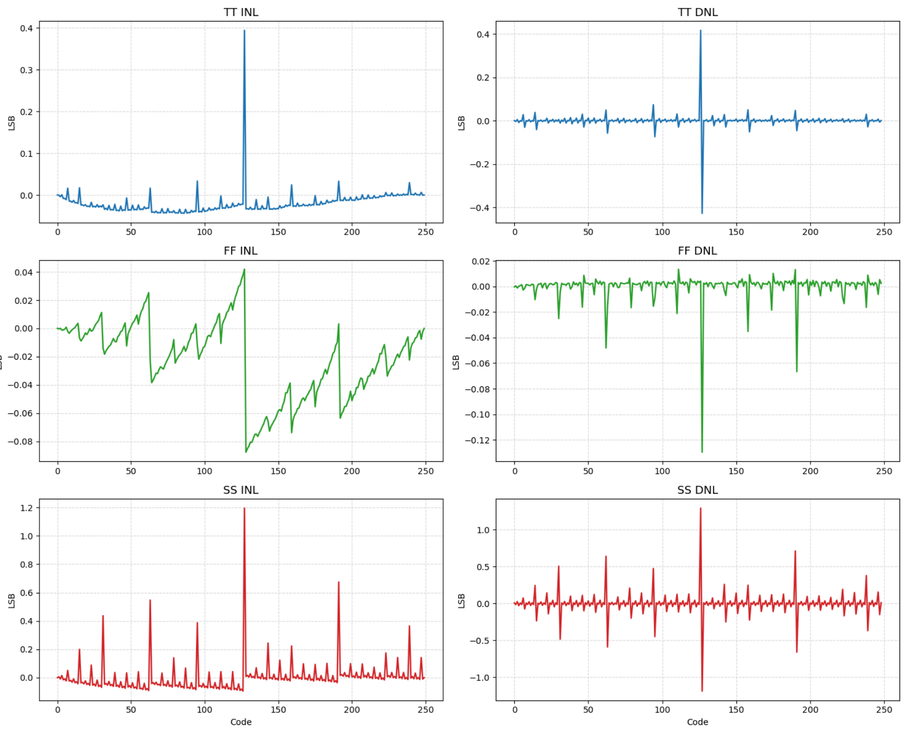


Figure 4.3*: Integral and Differential Nonlinearity measured across 255 code transitions for Fast-Fast, Typical-Typical & Slow-Slow Corners. Refer to Appendix A.1 for Application.*

Looking at the results from Figure 4.3, we can see that the DAC respects its linear constraints over the Nominal and Best case corners. However, in the worst case (ss), corners are shown to perform with a ~1.2 LSB affected only when the DAC switches its MSB (i.e when code transitions from 127 to 128).

This can later be fine-tuned with better matching and sizing of the R-2R ladder.

5. Strong-Arm Latch Comparator

The Comparator block consists of a dynamic Strong Arm Latch followed by a pair of digital output buffers and an SR latch. Its purpose is to resolve the difference between the sampled input voltage and the DAC output during each SAR bit decision and generate a clean, stable digital logic level for the SAR controller. Because the comparator directly determines each bit's accuracy, its speed, input-referred offset, and kickback behaviour are critical to overall ADC performance.

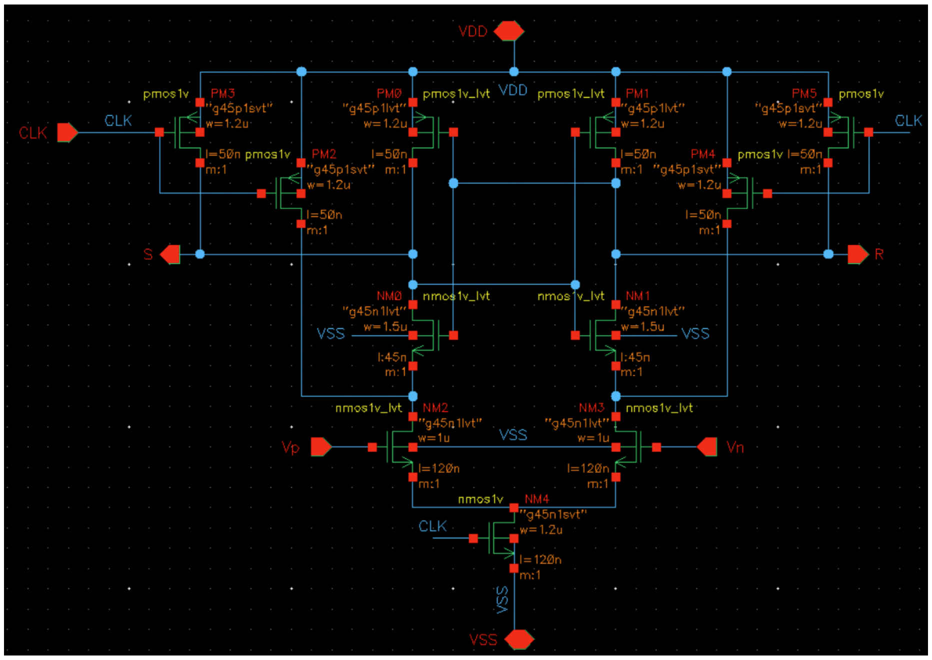


***Figure 5.1**: Circuit Implementation of the StrongArm Latch Comparator. The output of the strong-arm is buffered and sent to an SR-Latch to hold the decision after regeneration.*

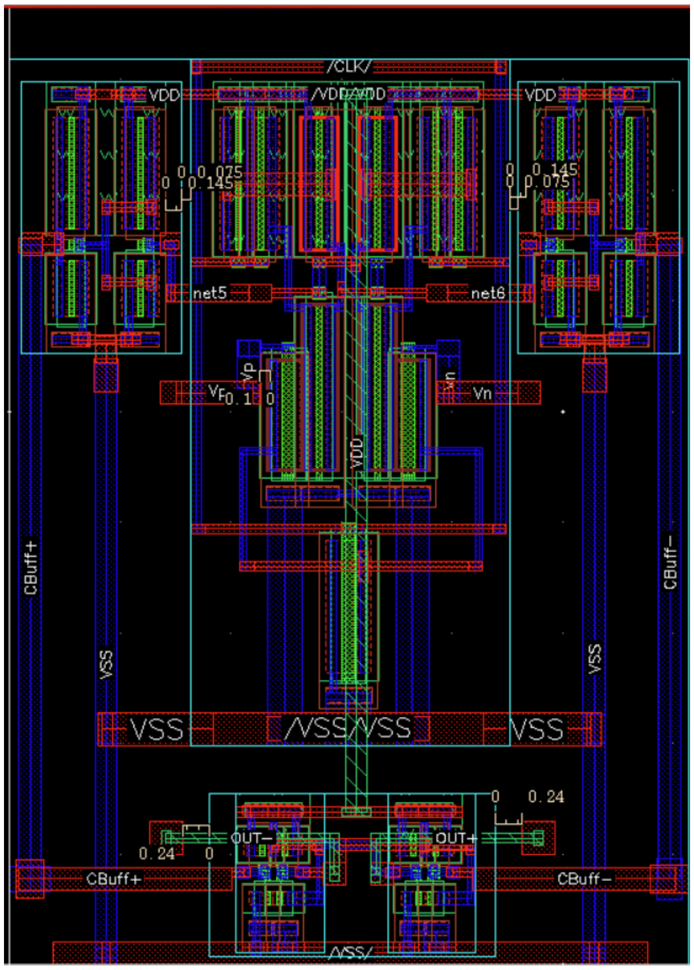


Figure 5.2*: Layout implementation of the StrongArm Latch Comparator. Proper matching and layout symmetry techniques are used at this stage to minimize mismatch challenges.*

5.1 Circuit Operation

During the evaluation cycle, the Strong-Arm Latch compares the differential input coming from the S/H output and the buffered DAC output, and regeneratively resolves their difference into full swing complementary digital outputs. Before evaluation, the reset phase ensures all internal nodes are precharged, eliminating memory effects and enabling predictable decisions each cycle.

After the latch resolves, the outputs drive a pair of digital buffers, which:

1. Restore full logic levels (0-2 V domain)
2. Isolate the latch from downstream capacitive loading, and

3. Reduce kickback from the SR latch or routing parasitics.

These buffered outputs then set or reset the SR latch, which holds the comparator decision stable for the entire SAR clock period until the next comparison cycle. This ensures that the SAR logic sees a clean, static bit value during the binary search, regardless of internal comparator transients or regeneration switching.

5.2 Verification of the Circuit Operation

We measure the performance of the Op-Amp by keeping one of the input pairs at a steady DC value while modulating the other in a sinusoidal manner. To do this, we set the comparator clock to 50MHz and have the modulating input swing at a 5MHz frequency. The following simulations are post-extraction results.

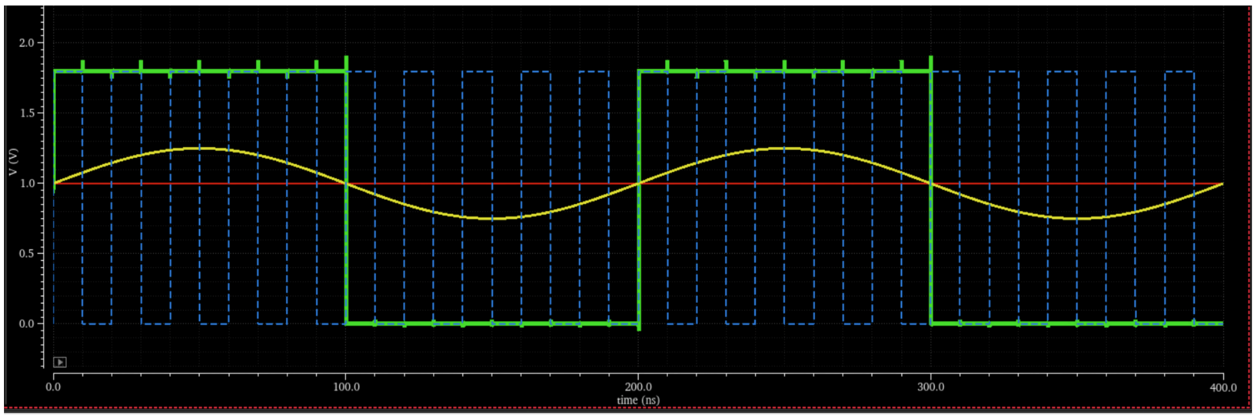


Figure 5.3: *DC input of 1V (red) is tied to the negative input of the comparator, while the positive input has an input that swings about that DC value with an amplitude of 200mV at 5MHz. The blue line shows the comparator clock.*

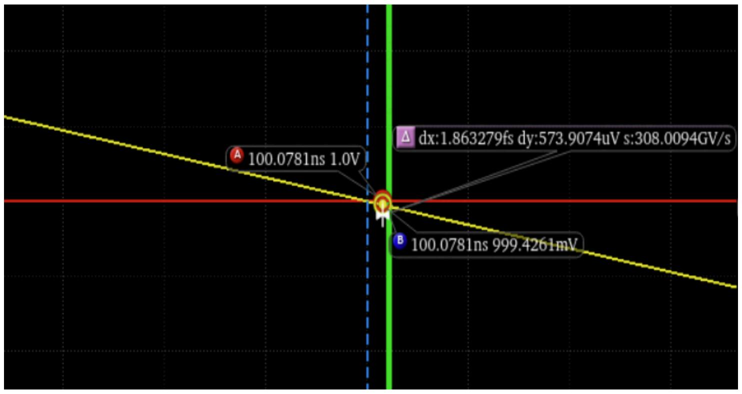


***Figure 5.4**: The dy value shown in the screenshot is the recorded value of the input referred offset (i.e., **573.907µV**). This is the minimum voltage difference between the differential inputs that the comparator is able to resolve.*

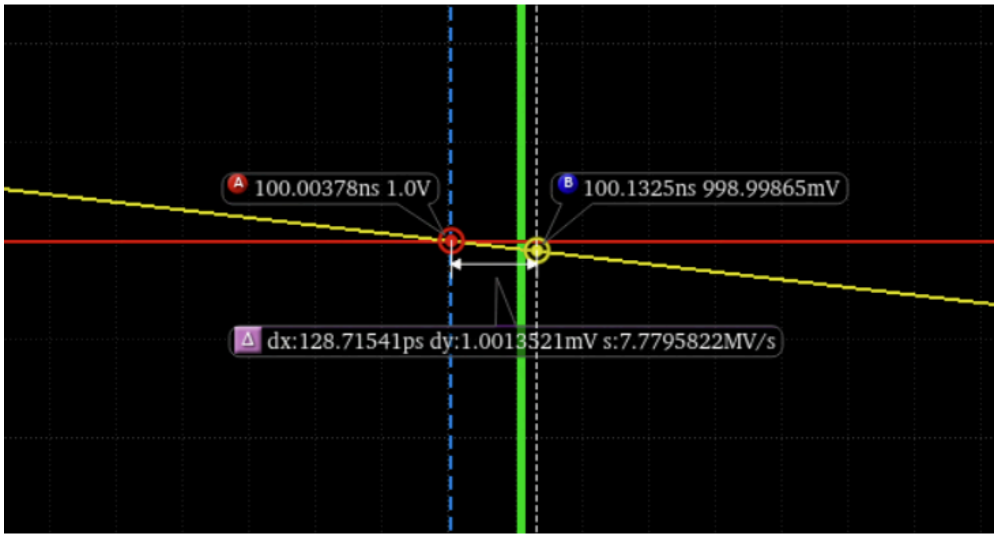


***Figure 5.5**: The dx value shown in the screenshot is the recorded value of the decision time (**128.715ps**). This is the minimum voltage difference between the differential inputs that the comparator is able to resolve.*

We finally report the worst-case performance for the Strong Arm latch as follows.

➔ **Input Referred Offset**: 573.907µV (SS)

➔ **Decision Time**: 128.715ps (SS)

➔ **Power**$_{\textbf{diss, ave}}$: 59.99µW (FF)

6. Successive Approximation Registry (SAR)

Unlike the other block in this Mixed-Signal IC design section, the SAR provides the digital control logic that drives the binary search process of the ADC. It determines each output bit by iteratively comparing the sampled input voltage against the DAC-generated trial voltages. The SAR block was initially developed in a Verilog-A behavioral model to enable co-simulation with the Analog front-end. Please refer to the Appendix section A.3 for the Verilog-A implementation of this block.

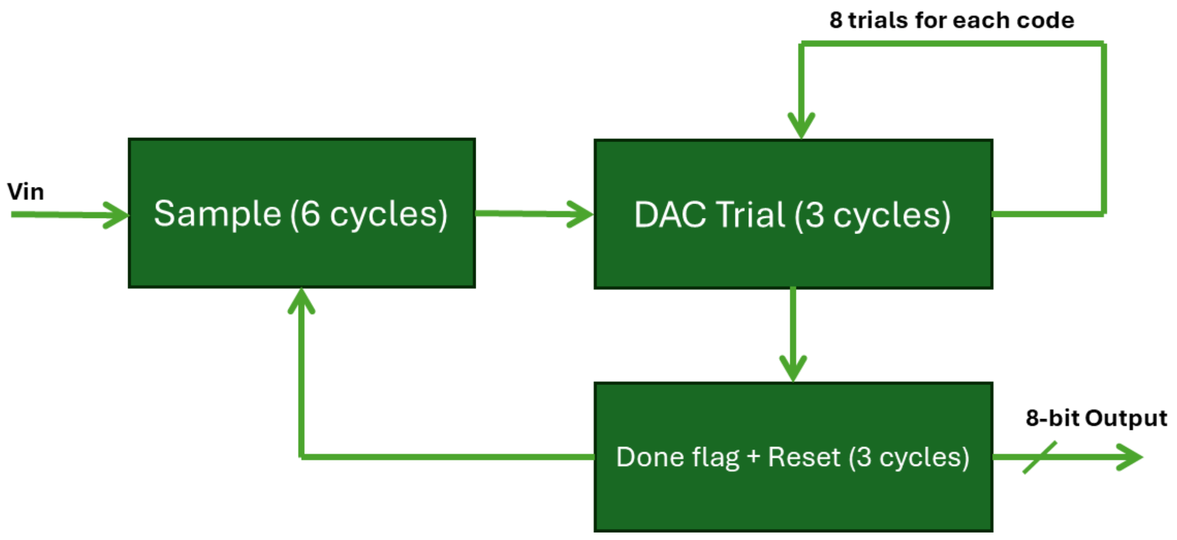


Figure 6.1: *FSM flow chart of the 8-bit Successive approximation registry.*

6.1 Circuit Operation

The SAR ADC operates synchronously at an 88.105 MHz core clock, executing a 36-cycle conversion sequence composed of sampling, eight successive DAC-comparator trials, and a final reset stage.

Each bit decision is completed in 3 cycles, providing sufficient settling time for both the capacitive DAC and the strong-arm comparator. The architecture achieves a 2.5 MHz effective sample rate. This block is verified in conjunction with the rest of the ADC; therefore, simulation results will be reported in the "*Verification of the ADC operation*" section. The Verilog-A Implementation of this block can be found in Appendix A.3.

7. Putting Everything Together

Before we put the ADC alongside the multiplexer, we need to verify its operation as a standalone block. To do this, we first move up one hierarchy and assemble the blocks we designed in previous sections. Refer to the figure below to see the schematic and layout implementation of the ADC.

The working directory for this project can be found:

"/home/bht4880/CE393_Hall_Effect_Sensor"

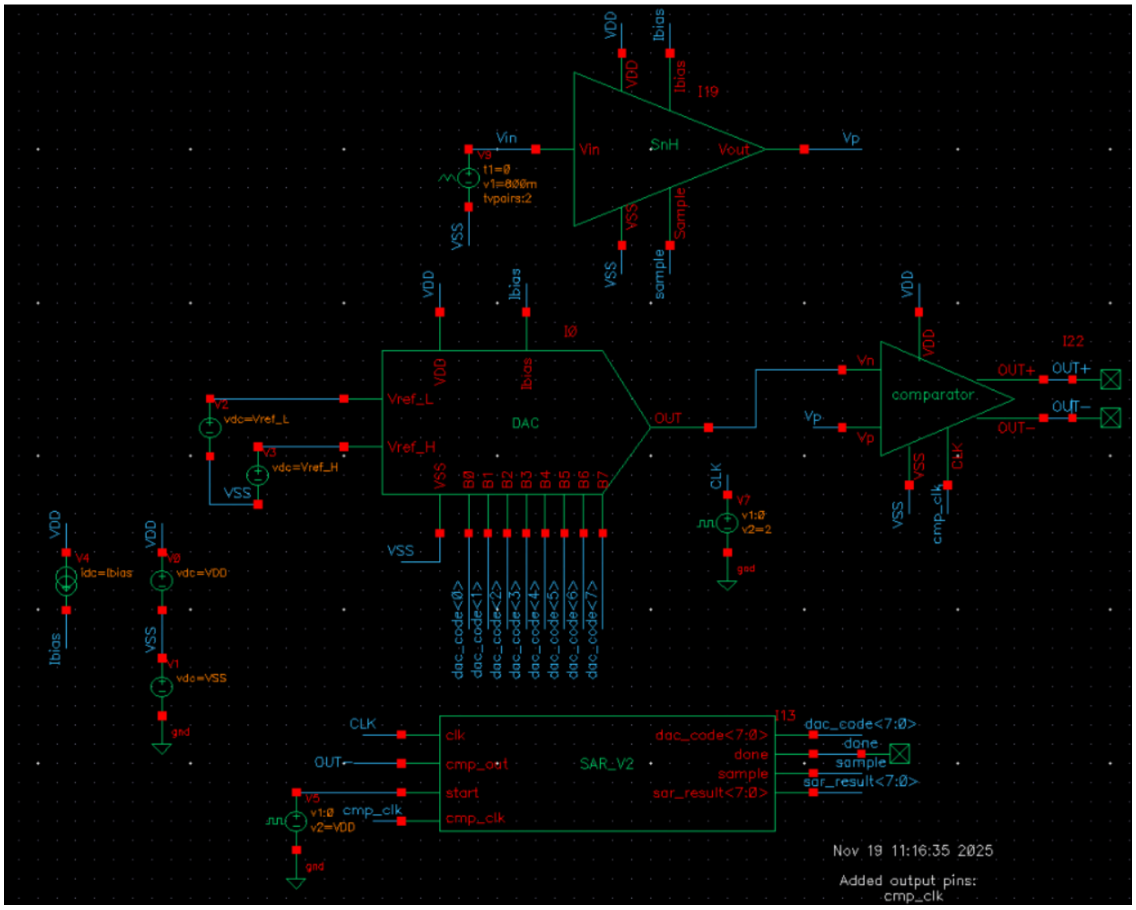


***Figure 7.1**: Schematic Implementation of the SAR-ADC*

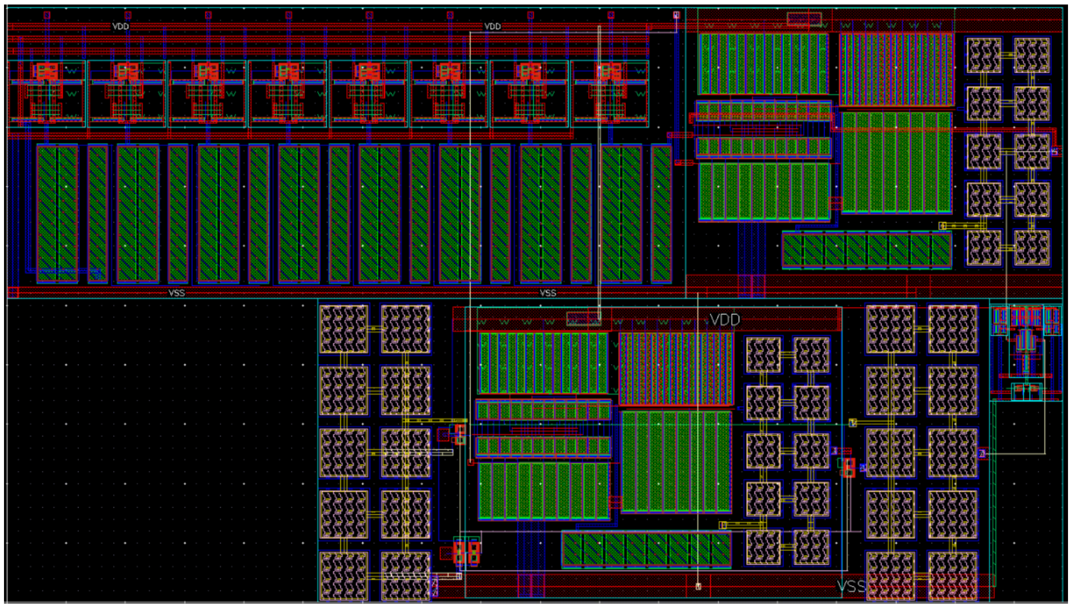


Figure 7.2**: Layout Implementation of the SAR-ADC with a **reported area of <u>50µm x 90µm</u>

7.1 Verification of ADC Operation

To verify the operation of the ADC, we ran a simulation with a ramp input being swept from *0.8v to 1.6v in 224.88µsec.* This will give the SAR ADC enough time to sample each step enough time as the sample rate is limited to 2.5MHz. The result of this simulation is then exported in a similar fashion to the DAC and INL/DNL parameters, which are then extracted using the python matplot.lib module. Refer to the figures below to see the worst-case performance of this block.

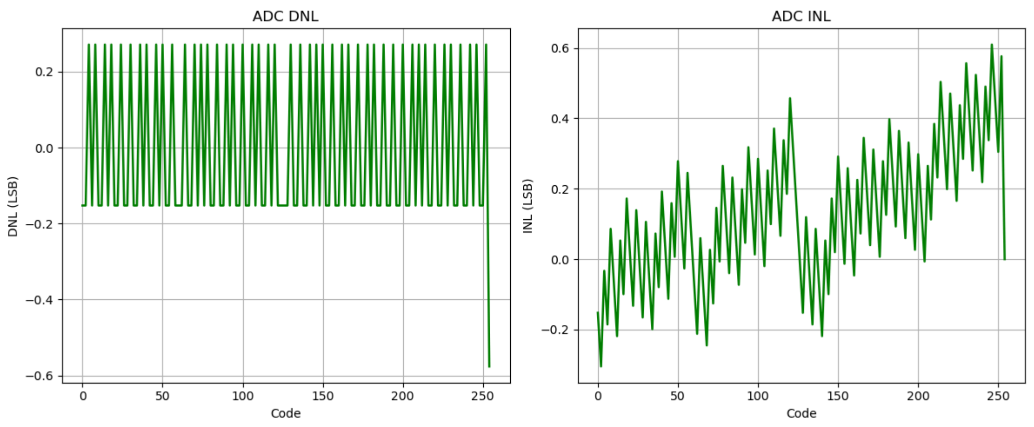


***Figure 7.3**: INL and DNL performance of the ADC over the worst-PVT corner case. Refer to Appendix A.2 for the application.*

For further verification, we reconstructed the ADC output by putting it through a DAC, and the result is shown below. We can see that the reconstructed output is very much tracking the ramp input; therefore, this gives confidence to the design.

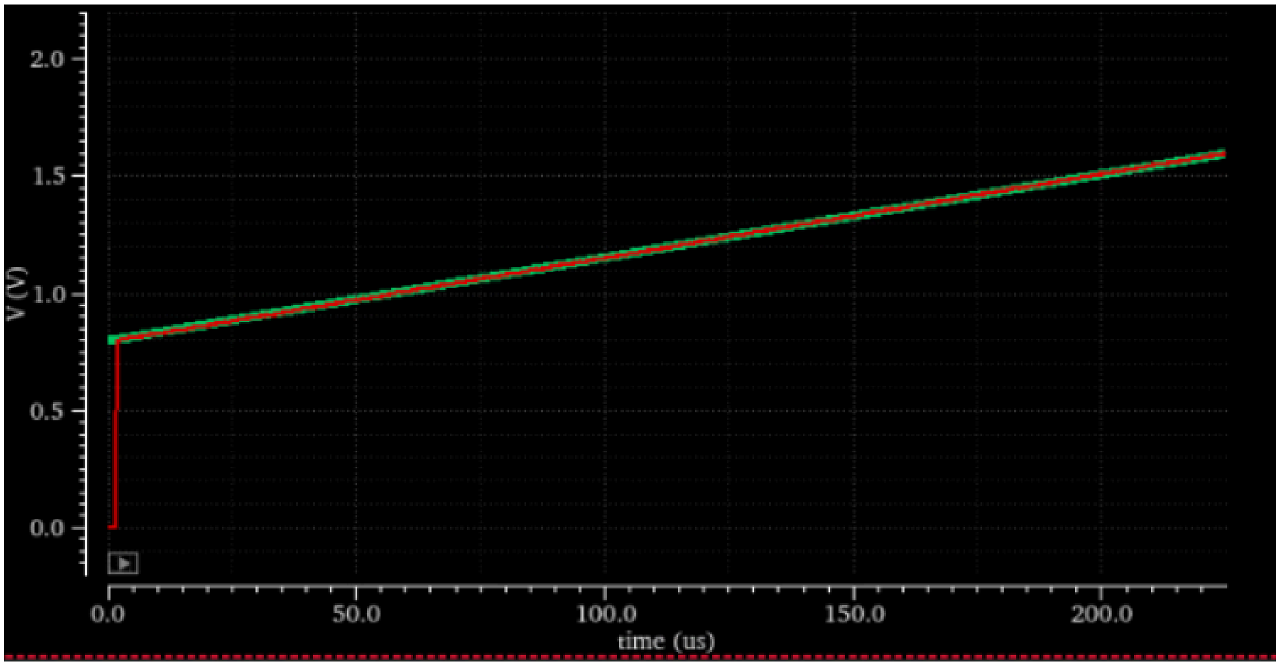


***Figure 7.4**: Reconstructed Analog Equivalent output of the ADC (red) vs Ideal ramp input (Green)*

7.2 Extracted Specifications of the 8-bit SAR ADC

7.2.1. Timing and throughput report

Specification	**Measured / Extracted**
Base Clock Frequency	88.105 MHz
Sampling Window	6 cycles
DAC + Comparator Trial Window	3 cycles per bit × 8 bits = 24 cycles
Finalization / Reset	6 cycles
Total Conversion Time	≈ 36 cycles
Effective Sample Rate	≈ 2.50 MHz
Sample Period	≈ 420 ns

7.2.2. Analog Performance

Metric	**Extracted Value**
Input Valid Range (ICMR)	0.8 V – 1.6 V
Full-Scale Input Span	0.8 V (0.8 to 1.6 V)
S/H Accuracy	<360.52 µV error
S/H Droop Rate	≈ 10.33 kV/s
Time Before Losing 1 LSB	≈ 300 ns (LSB ≈ 3.1 mV)
Comparator Offset	< 1 mV (≈ 0.25 LSB)
Comparator Decision Time	< 1 ns after DAC settles
Average Power Consumption	842.382µW

7.2.3. Linearity Performance

Metric	**Typical**	**Worst Case**
Integral Non-linearity INL	**±0.6 – 0.8 LSB**	**≈ ±1.1 LSB**
Differential Non-linearity DNL	**±0.4 LSB**	**≈ +0.8 LSB spikes**

7.3 Mux-ADC Verification

Now that we have confirmed the operation of our ADC, we implement it alongside the 3:1 analog mux and test the overall operation with three different inputs that mimic the Hall sensor.

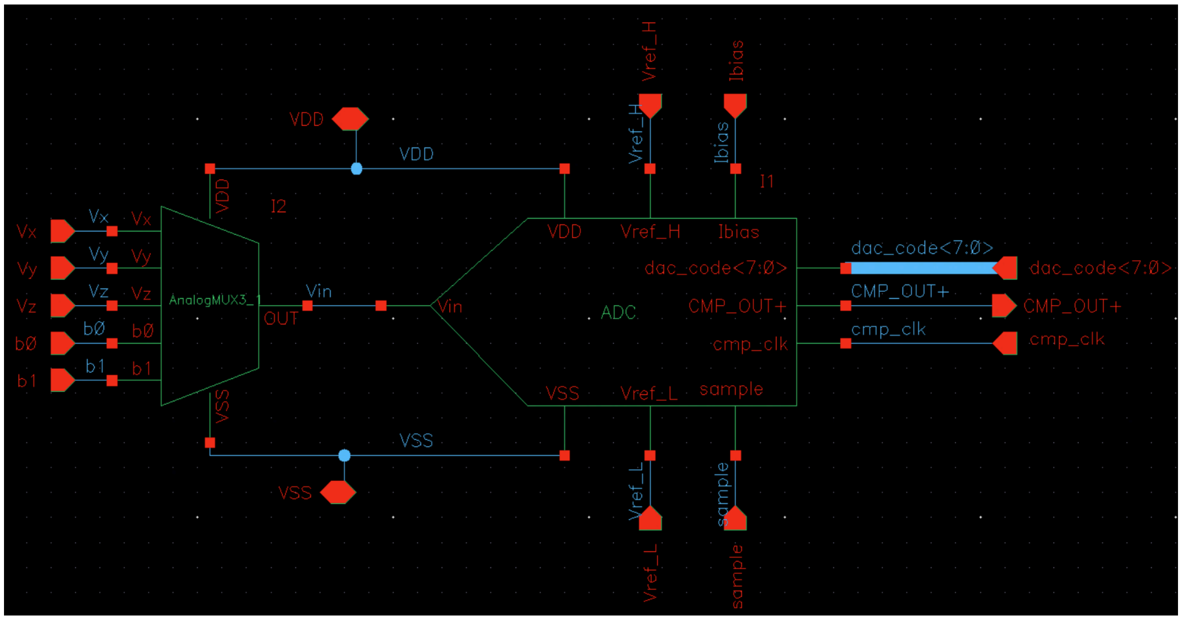


Figure 7.5: *Schematic implementation of the MUX-ADC Design*

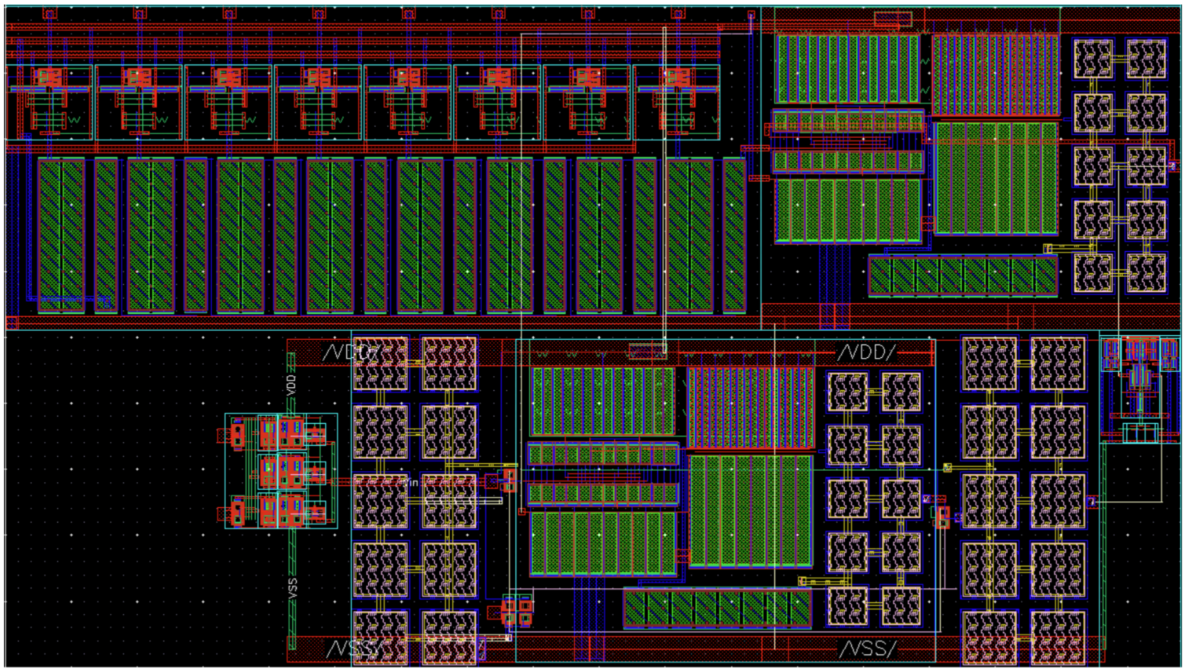


Figure 7.6*: Layout implementation of the MUX-ADC Design with a* ***reported area of*** ***<u>50µm x 90µm</u>***

To verify the operation of this circuit, we input three sinusoidal signals that are of different frequencies and observe if the ADC properly samples each signal as requested by the digital block. It is important to note that the ADC output is ready in the next sample period. Therefore, we compare the previously sampled input to the ADC output in the next sample period. The figure below confirms exactly what we expect.

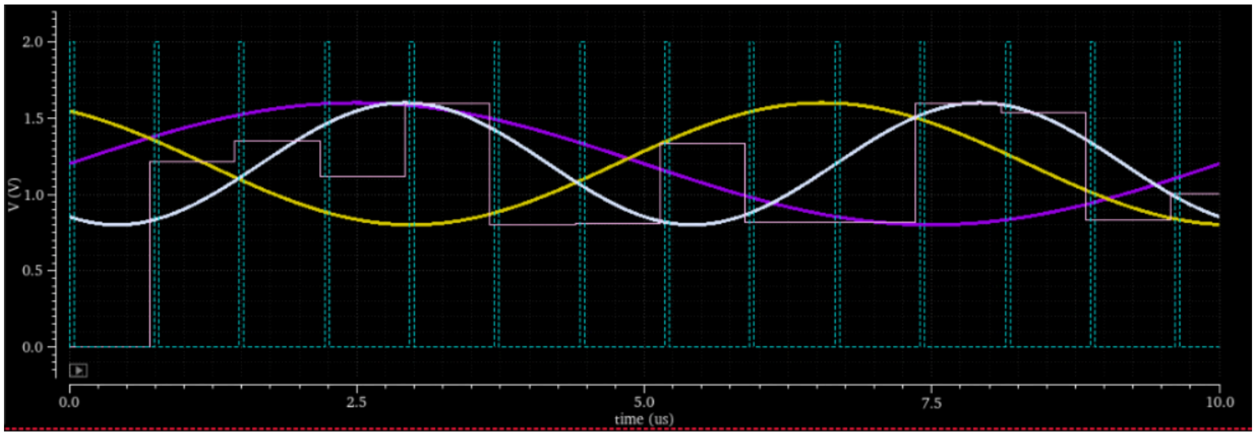


Figure 7.7*: Verification of the three multiplexed sampling of the ADC for the three input signals labeled in white, yellow, and purple. The light green dots show the time of the sample, whereas the light red line shows the output of the ADC.*

8. Challenges and Solutions to Designing the Mixed-Signal Front End

Designing the mixed-signal readout began with stabilizing the sample-and-hold (S/H) stage, which initially suffered from droop, long settling, and sensitivity to input slope. Because the topology used an op-amp in unity-gain with a hold capacitor, small variations in bias current and open-loop gain caused inconsistent sample accuracy across PVT corners. Similar issues have been documented in precision S/H circuits, where charge injection and clock feedthrough degrade linearity [6]. We resolved this by strengthening the op-amp biasing, refining the switch sizing to reduce charge injection, and giving the S/H block a well-defined 6-cycle sampling window. These adjustments produced a predictable held value that the SAR logic could reliably process.

The R-2R DAC and its transmission-gate switches introduced another set of challenges. The ladder settled differently in TT, FF, and SS corners, especially when multiple bits toggled simultaneously. Early simulations showed incomplete settling during the SAR trial window, which led to inconsistent comparator decisions. Settling-time limitations in R-2R DACs are well known, with resistor mismatch and parasitic capacitances often cited as dominant error sources [7]. To address this, we adjusted the resistor matching strategy, optimized switch sizing to reduce parasitics, and selected a 3-cycle DAC trial window that guaranteed adequate settling time before the comparator was clocked. Once these changes were in place, the DAC produced clean, monotonic output steps suitable for accurate binary search.

The strong-arm comparator required careful timing alignment with the SAR controller. Early versions produced incorrect decisions because the comparator was clocked before the R–2R DAC had fully settled, and polarity mistakes in the wiring caused reversed outputs during certain codes. These challenges mirror those described in strong-arm latch design studies, where offset calibration and timing synchronization are critical for reliable SAR operation [8]. Fixing the polarity, spacing the comparator strobe later in the cycle, and ensuring the SAR FSM only advanced on rising edges resulted in stable operation. The final integration challenge was synchronizing the entire pipeline in Verilog-A. Defining a fixed 36-cycle conversion window gave every block enough time to settle. Afterward, we built a Python tool to detect DONE edges, extract codes, and compute INL/DNL from both analog and digital outputs, which validated end-to-end functionality and highlighted remaining edge-case behaviors. Similar mixed-signal verification approaches have been emphasized in recent SAR ADC design methodologies [9].

Section II. Digital Backend Overview (Tri‑Axis Magnetic Field Readout)

1. Scope and Purpose

In this section we discuss the digital backend of a tri‑axis magnetic-field sensing system. The system receives digitized samples from an external ADC + analog MUX, processes the samples for three axes (X, Y, Z) in parallel digital pipelines, and outputs the final processed tri‑axis vector periodically.

The digital backend is responsible for:

- Demultiplexing ADC samples into per-axis streams based on a 2‑bit axis indicator.
- Oversampling + decimation using a moving-average (boxcar/CIC-like) filter to reduce quantization noise.
- Startup offset calibration to remove static bias (DC offset) caused by sensor front-end and ADC midscale mismatch.
- Drift and flicker suppression using a slow-tracking baseline estimator and a high-pass subtraction (EMA baseline removal).
- Gain scaling (unity gain in our delivered configuration, but parameterized).
- UART-like serialization of the output vector.

This section of the report focuses on the digital architecture, signal formats, filtering mathematics, throughput, and how each RTL module contributes to the overall function. Figure 1.1 below illustrates the overall architecture of the digital backend of the developed system. This section of the project is implemented using the GPDK45 standard cell library.

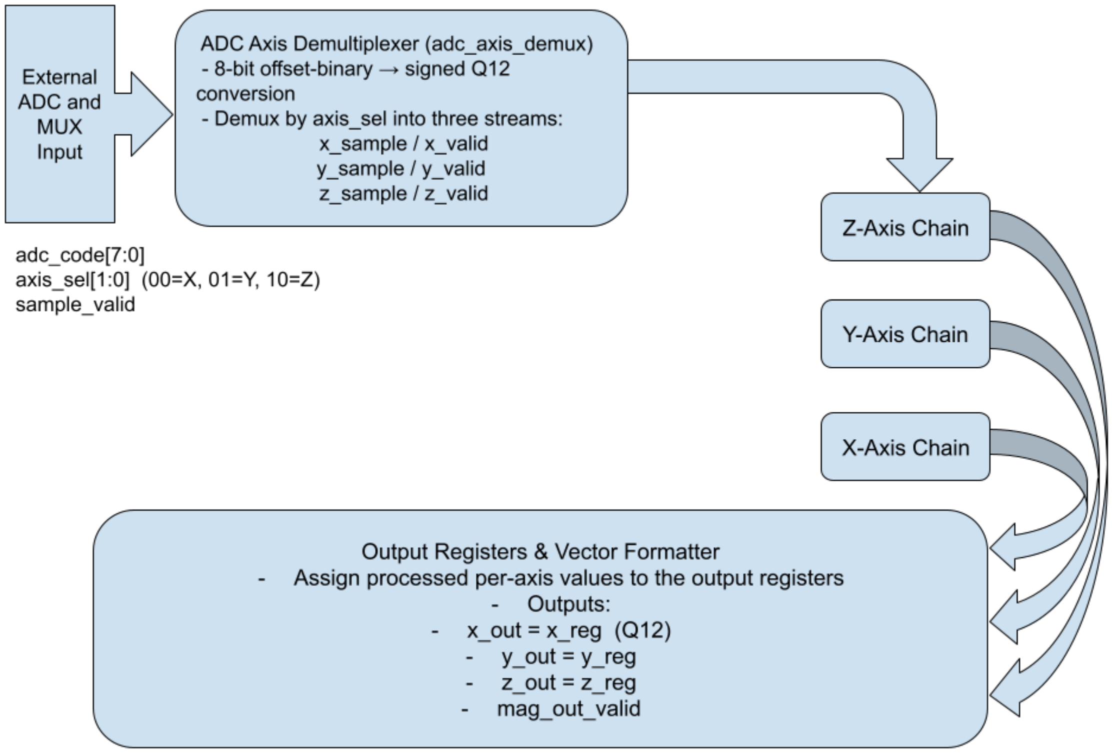

External ADC and MUX Input
adc_code[7:0]
axis_sel[1:0] (00=X, 01=Y, 10=Z)
sample_valid
ADC Axis Demultiplexer (adc_axis_demux)
- 8-bit offset-binary → signed Q12 conversion
- Demux by axis_sel into three streams:
x_sample / x_valid
y_sample / y_valid
z_sample / z_valid
Z-Axis Chain
Y-Axis Chain
X-Axis Chain
Output Registers & Vector Formatter
- Assign processed per-axis values to the output registers
- Outputs:
- x_out = x_reg (Q12)
- y_out = y_reg
- z_out = z_reg
- mag_out_valid

Figure 1.1: *Block Diagram of the System Architecture*

The system receives three signals from the ADC domain: an 8-bit offset-binary conversion result, a single-cycle sample_valid strobe, and a 2-bit axis_sel signal originating from the analog multiplexer. The analog front end sequences axis_sel through the values 00, 01, and 10, corresponding to the X, Y, and Z axes, respectively. As a result, consecutive ADC conversions produce samples for the X, Y, and Z axes in a repeating three-sample cycle.

The first functional block in the architecture is the ADC axis demultiplexer. This block performs two primary functions. First, it converts the 8-bit offset-binary ADC output into a signed fixed-point representation in Q12 format, with mid-scale corresponding to zero magnetic field. Second, it uses the axis_sel signal to demultiplex the incoming samples into three independent data paths: x_sample, y_sample, and z_sample. Each data path is accompanied by its own valid strobe, indicating the presence of a new sample for the corresponding axis.

Following the demultiplexer are three identical, per-axis processing chains (Figure 1.2). These chains operate in parallel on a common core clock. Each chain accepts the sample stream for one axis and performs signal conditioning and calibration, producing a cleaned, drift-suppressed output value for that axis.

At the output stage, the processed X, Y, and Z values are captured in output registers and combined by a vector formatting block. This block concatenates the three axis values into a single internal output bus. In addition, a mag_out_valid signal is generated when the Z-axis processing chain updates, indicating that a complete and coherent XYZ magnetic field vector is available. From an external interface perspective, the macro therefore behaves as a synchronous digital block: each assertion of mag_out_valid denotes the availability of a new three-dimensional magnetic field vector on the output bus.

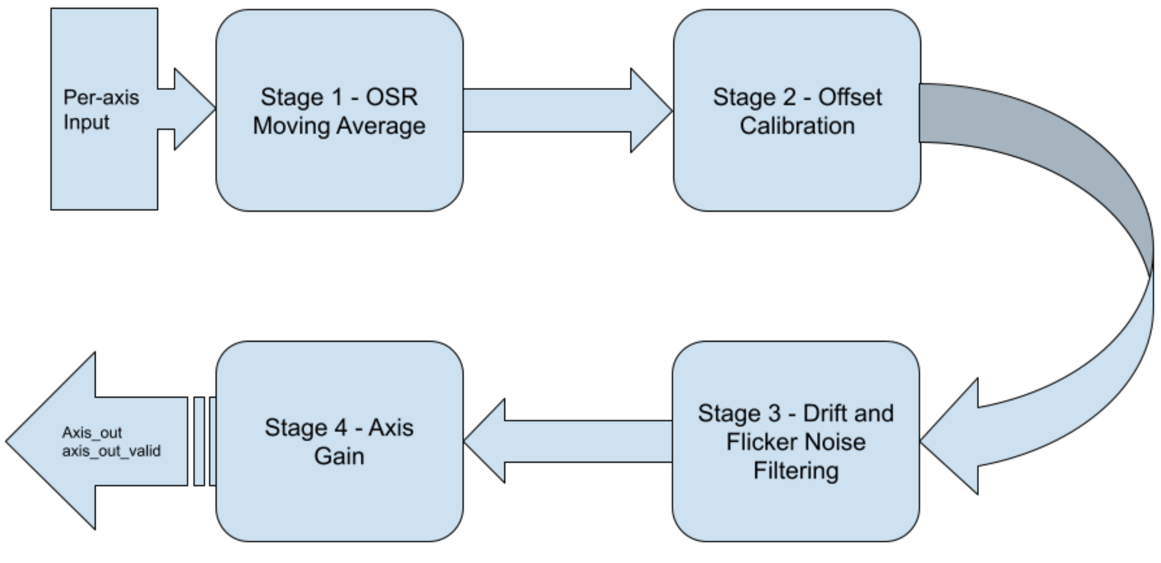

Per-axis Input
Stage 1 - OSR Moving Average
Stage 2 - Offset Calibration
Stage 3 - Drift and Flicker Noise Filtering
Stage 4 - Axis Gain
Axis_out
axis_out_valid

Figure 1.2: *Per-axis processing chain diagram*

Figure 1.2 illustrates the per-axis chains from Figure 1.1 and is composed of the following compartments, described in the following paragraphs.

Processing begins with Stage 1: Oversampling Ratio (OSR) Moving Average. Each axis is oversampled by a fixed factor of 64. Upon assertion of the axis-specific sample_valid signal, the incoming Q12-formatted sample is accumulated. After 64 samples have been collected, the accumulated sum is divided by 64 via a right-shift operation, producing a single decimated output sample. Functionally, this stage behaves as a simple cascaded integrator–comb (CIC) filter. It reduces signal bandwidth and improves in-band signal-to-noise ratio by approximately 18 dB for white quantization noise originating from the SAR ADC.

The decimated output is then passed to Stage 2: Offset Calibration. During device startup, 256 decimated samples are collected under the assumption that the true magnetic field is nominally zero. These samples are averaged to estimate the DC offset for the corresponding axis. The resulting offset value is stored and subsequently subtracted from all future samples. Throughout this calibration interval, the valid signal is suppressed to ensure that downstream logic does not observe uncalibrated data.

Next, the signal enters Stage 3: Drift and Flicker Noise Filtering. Although static offset is removed in the previous stage, Hall sensors and associated amplification circuitry exhibit slow drift over temperature and time, manifesting as low-frequency noise. To mitigate this effect, a very slow exponential moving average is applied to the calibrated signal to track the baseline drift. This estimated baseline is then subtracted from the signal. In the frequency domain, this operation is equivalent to a high-pass filter with an extremely low cutoff frequency, allowing legitimate low-frequency magnetic signals to pass with minimal attenuation while significantly suppressing ultra-slow drift and flicker noise components [10].

The final processing block is Stage 4: Axis Gain Adjustment. This stage consists of a fixed Q12 multiplier. In the current implementation, the gain is set to unity (1.0). However, the inclusion of this stage provides architectural flexibility to correct residual gain mismatches between axes in future revisions, without requiring modifications to the analog front end.

The output of the per-axis chain consists of axis_out and the corresponding axis_out_valid signal. These outputs are forwarded to the top-level logic, where they are combined with the other axes and formatted into the three-dimensional magnetic field vector, as shown in Figure 1.1.

2. System Interfaces and Operating Rates

2.1. Inputs from External ADC and MUX

The backend receives one ADC conversion at a time. An external analog multiplexer selects one of three magnetic axes (X/Y/Z), and the ADC digitizes the selected channel.

Digital inputs:

- clk: backend clock (targeted for high-speed implementation; the main digital blocks were designed to run up to the project's high-frequency timing target).
- rst_n: active-low reset.
- adc_code: ADC output word.
- sample_valid: pulse indicating adc_code is valid this cycle.
- axis_sel[1:0]: indicates which axis the current sample corresponds to:
  - 2'b00 = X
  - 2'b01 = Y
  - 2'b10 = Z
  - 2'b11 unused/ignored

Sampling cadence assumption (system-level): We assume one axis value is presented every 400 ns (i.e., sample_valid pulses every 400 ns). If we cycle X→Y→Z, then each axis receives a sample every 1200 ns.

2.2. ADC Resolution and Practical Handling

At the project integration level, the ADC is treated as producing 8-bit effective output in the final design. However, the backend interface can be carried as a wider bus (e.g., 10-bit) for convenience and future expansion.

In our testbenches, we state that the ADC bus may be represented as 10 bits for compatibility. The two LSBs are padded with 0, and the effective resolution is 8 bits for the current ADC implementation. The datapath is scalable to the full bitwidth if the ADC is later extended.

3. Fixed Point Representation and Output Width

Processed axis values are represented using a signed fixed-point Q12 format. In this representation, the binary point is positioned such that there are 12 fractional bits. Consequently, an integer value V_{q12} corresponds to a real-valued quantity given by $V_{real} = V_{q12}/2^{12}$. The Q12 format is well-suited to this application because it enables fractional arithmetic while relying exclusively on integer hardware operations, such as shifts and additions, thereby avoiding the complexity of floating-point logic.

Although the ADC provides only 8 effective bits of resolution, subsequent digital signal processing stages require substantially greater internal precision. Upon conversion, ADC codes are first transformed into signed values and left-shifted by 12 bits to align with the Q12 format, immediately increasing word width. The oversampling ratio (OSR) moving-average stage further increases bit requirements, as multiple samples are accumulated; this introduces additional bit growth on the order of $log_2(OSR)$. Similarly, offset calibration involves the accumulation and averaging of multiple samples, necessitating extra guard bits to prevent overflow and loss of precision. The exponential moving average (EMA) used for baseline and drift tracking retains internal state over time, and insufficient word width in this stage would lead to progressive loss of resolution due to repeated truncation or rounding.

For these reasons, a 24-bit word width (W = 24) is used throughout the internal processing chain and at the output. It provides sufficient headroom to preserve numerical precision across all filtering and calibration stages, prevents overflow under worst-case accumulation scenarios, and minimizes quantization error from intermediate truncations. Additionally, maintaining a fixed word width across all stages simplifies the hardware implementation and control logic.

Conceptually, the system maintains three simultaneous internal axis values (X, Y, and Z), each represented with 24 bits, resulting in a total internal vector width of 72 bits. This vector, however, is not exported as parallel output signals. Instead, the three-axis data are serialized and transmitted through a single UART-like output stream, reducing pin count and simplifying the external interface.

4. Digital Architecture Top-Level View

At a high level (see Figure 1.1), the backend architecture consists of the following functional blocks.

ADC Axis Demultiplexer (adc_axis_demux)

- Converts the ADC output code from offset-binary format to a signed representation.
- Translates the signed value into the Q12 fixed-point domain.
- Routes each incoming sample into one of three data streams (X, Y, or Z) based on the axis_sel signal.

Per-Axis Processing Pipelines (axis_proc_simple)

- Three identical processing pipelines are instantiated, one per axis.
- All pipelines operate in parallel and execute the same sequence of processing stages:
    - Oversampling-ratio (OSR) moving average with decimation
    - Offset calibration
    - Drift and flicker noise suppression using an exponential moving average (EMA) baseline and subtraction
    - Gain scaling

Output Registers and Vector Formatting

- The processed X, Y, and Z axis values are captured in output registers.
- When a complete three-axis vector is updated (typically aligned with the Z-axis valid event), an output-valid event is generated.

UART-Like Serializer

- On each output-valid event, the current XYZ vector is latched.
- The latched vector is serialized and transmitted over a single output pin.

This architecture exhibits two key properties:

- Functional determinism: Each processing stage is simple, parameterized, and fully synchronous to the system clock (clk), resulting in predictable and repeatable behavior.
- Throughput isolation: Input sampling occurs at a relatively low rate compared to the maximum digital clock frequency (591 MHz), providing some processing margin for filtering, registration, and serialized output transmission.

5. Per-Axis Pipeline Architecture and Mathematics

Each axis processing pipeline is identical. For clarity, the X-axis pipeline is described below; the Y- and Z-axis pipelines behave identically. Refer to Figure 1.2 to view the top-level explanation for this block.

5.1. Stage 0: ADC Code to Signed Q12 (in adc_axis_demux)

The ADC output is treated as offset-binary, where the midscale code represents zero magnetic field. Let:

- code is the ADC output code for a given sample
- CODE_ZERO be the midscale code (e.g., 128 for an 8-bit ADC, 512 for a 10-bit representation)

The signed integer difference is computed as: $d = code - CODE_{ZERO}$

This value is then converted to the Q12 fixed-point domain: $x_{q12} = d.\, 2^{12}$

This operation is implemented as a left shift by 12 bits. The result is a signed Q12 sample stream suitable for subsequent digital filtering stages.

5.2. Stage 1: OSR Moving Average and Decimation (axis_osr_mavg)

Oversampling and averaging are used to reduce noise [11]. With an oversampling ratio defined as:$OSR = 2^{OSR_{LG2}}$. Then the decimated output is computed as:

$$x_{dec}[k] = \frac{1}{OSR} \sum_{i=o}^{OSR-1} x_{q12}[k \,.\, OSR + i].$$

Hardware implementation:

- An accumulator sums incoming samples.
- A counter tracks samples from 0 to OSR−1.
- When the counter reaches OSR−1:
    - decim_out = acc >> OSR_LG2 (arithmetic right shift)
    - decim_valid is asserted
    The accumulator and counter are reset for the next block

This stage implements a boxcar finite impulse response (FIR) decimator. It is equivalent to a single-stage cascaded integrator–comb (CIC) integrate-and-dump structure.

Noise and SNR implications:

Averaging OSR samples reduces white noise power by a factor of OSR, resulting in an SNR improvement of: $10\ log_{10}(OSR)\ dB$.

Examples can be given as:

- OSR = 8 → approximately 9 dB improvement
- OSR = 64 → approximately 18 dB improvement

Since OSR is parameterized, the noise-reduction benefit is tunable.

5.3. Stage 2: Startup Offset Calibration (axis_offset_cal)

Static offsets are introduced by real sensors and interface circuitry. A startup calibration is performed by averaging the first:

$$N_{cal} = 2^{CAL_{LG2}}$$

Decimated samples:

$$offset = \frac{1}{N_{cal}} \sum_{k=0}^{N_{cal}-1} x_{dec}[k].$$

After calibration, the offset-corrected signal is: $x_{cal}[k] = x_{dec}[k] - offset.$

Hardware implementation:

- During calibration:
  - Accumulate acc += decim_in
  - Increment a sample counter

- After N_{cal} samples:
  - Store offset = acc >> CAL_LG2
  - Assert cal_done

- After calibration completes:
  - cal_out = decim_in - offset
  - cal_valid is asserted

Valid output is deliberately suppressed during the calibration window so that downstream logic never observes partially calibrated data.

5.4. Stage 3: Drift and Flicker Noise Suppression (axis_drift_hp)

This stage mitigates slow drift and low-frequency (1/f) noise. A baseline estimate is maintained using an exponential moving average (EMA):

$$b[k + 1] = b[k] + \alpha\cdot(x_{cal}[k] - b[k]),$$

where: $\alpha = \frac{1}{2^{ALPHASHIFT}}$.

The high-pass output is then computed as: $y[k] = x_{cal}[k] - b[k]$.

Signal interpretation:

- b[k] tracks the slow-varying baseline (drift).
- y[k] contains the residual signal with the baseline removed, emphasizing faster variations corresponding to the magnetic signal of interest.

This approach does not explicitly classify drift. Instead, the baseline updates slowly due to the small value of α, preventing it from following rapid signal changes. Subtraction of this slowly varying baseline yields high-pass behavior.

Cutoff-frequency intuition:

The EMA is a first-order IIR low-pass filter. Its approximate cutoff frequency is:

$$f_c \approx \frac{\alpha}{2\pi}\cdot f_s,$$

where f_s is the decimated per-axis sample rate.

In this system:

- Raw sampling: one sample every 400 ns, round-robin across axes
- Per-axis raw rate ≈ 0.833 MS/s

- OSR = 64 → per-axis decimated rate ≈ 13 kS/s

  With α = 1/256: $f_c \approx \frac{1}{256 \, . \, 2\pi} \, . \, 13000 \approx 8\, Hz.$

This cutoff aligns with the design intent: baseline drift and flicker-dominated components below approximately 8 Hz are tracked and removed, while higher-frequency magnetic signals are preserved.

5.5. Stage 4: Gain Scaling (axis_gain)

A fixed per-axis gain is applied $y_g[k] = \frac{GAIN_{q12} \cdot y[k]}{2^{12}}$.

Hardware implementation:

- Multiply: prod = in_sample × GAIN_Q12
- Arithmetic right shift by the fractional bit count

For unity gain: $GAIN_{q12} = 2^{12} = 4096.$ In this case, the output equals the input, aside from quantization or truncation effects.

This stage is retained to provide a convenient location for future sensor scaling or per-axis gain correction and to maintain consistency with a complete digital calibration architecture.

6. Top-Level Vector Assembly and Output Timing

6.1. Output Registering Strategy

Each axis pipeline produces axis_out_valid at the decimated rate after calibration. The top module registers X, Y, and Z outputs.

A simple, top-level strategy is:

- update x_reg when X-valid pulses,
- update y_reg when Y-valid pulses,
- update z_reg when Z-valid pulses,
- assert mag_out_valid when Z updates, meaning a full new XYZ vector is ready.

This aligns with a system where input axis conversions are interleaved and the Z-valid moment naturally marks the end of a full X/Y/Z cycle.

6.2. Internal Vectorization

Internally, the vector is: $V = [X_{REG}, Y_{REG}, Z_{REG}]$. This is conceptually 72 bits, but we do not expose it as 72 pins in the final version.

7. UART-Like Single-Pin Output Protocol

7.1. Motivation

A 72-bit parallel output is not desirable for physical I/O, because it consumes too many pads/pins, complicates routing and packaging, and is not needed if the output bandwidth is moderate.

Therefore, we use a UART-style single-pin serialized output: one data wire carries the full XYZ vector each time mag_out_valid indicates that a new vector is ready.

7.2. Framing and Content

A typical UART-like frame can be structured as:

- Optional preamble byte(s) to allow robust synchronization (e.g., 0xA5, 0x5A)
- Payload:

    - X value (24-bit signed, 3 bytes)
    - Y value (24-bit signed, 3 bytes)
    Z value (24-bit signed, 3 bytes)

- Checksum/CRC (not included in our scope)

This makes the output easy to read with a microcontroller or USB-UART bridge.

7.3. Throughput Constraint

Because the backend produces vectors at the decimated rate, the UART baud must be high enough so a full XYZ frame is transmitted before the next vector arrives (or a small buffer must be used).

Given the decimated per-axis rate is on the order of 10 kHz (depending on OSR and input cadence), the required UART baud is typically in the Mbps range to serialize 9+ bytes per vector comfortably. This is achievable, especially given the high backend clock frequency and the large cycle budget between vector updates.

8. Summary of the Digital Backend Behavior

Overall, the digital backend performs the following transformation:

1. **ADC code** → signed Q12 samples
2. **Noise reduction** via OSR moving average + decimation
3. **Offset removal** via initial calibration mean subtraction
4. **Drift/flicker suppression** via slow EMA baseline tracking + subtraction
5. **Gain scaling** (unity gain now; parameterized)
6. **Vector formation** (XYZ) and **serial streaming** via UART-like transmitter

This yields stable, drift-suppressed, smoothed tri-axis magnetic field outputs suitable for system-level integration.

9. File-by-File Description

1. **adc_axis_demux.v**
    - Converts ADC code from offset-binary to signed.
    - Converts signed code into Q12 (shift by FRAC).
    - Demultiplexes samples into X/Y/Z streams based on axis_sel.
    - Generates x_valid/y_valid/z_valid pulses.
2. **axis_osr_mavg.v**
    - Implements OSR moving-average decimator.
    - Accumulates OSR samples and outputs the average (shift right by OSR_LG2).
    - Produces decim_valid at the decimated rate.
3. **axis_offset_cal.v**
    - Averages the first $2^{CAL_{LG2}}$ decimated samples to estimate the offset.
    - After calibration completes, subtracts offset from each subsequent sample.
    - Suppresses output-valid during calibration (cal_valid only asserted after done).
4. **axis_drift_hp.v**
    - Implements EMA baseline tracking.
    - Outputs a drift-suppressed sample (y = x - b).
    - This is the main drift/flicker (low-frequency) suppression mechanism.
5. **axis_gain.v**
    Multiplies the drift-suppressed sample by a fixed Q12 gain.
    - Defaults to unity gain (4096 in Q12 for FRAC=12).
6. **axis_proc_simple.v**
    - Wraps the entire per-axis chain:
        - OSR → offset calibration → drift HP → gain
    - Exposes axis_out and axis_out_valid as the final processed stream for that axis.
7. **mag_backend_simple.v**
    - Top-level integration:
        - Instantiates adc_axis_demux
        - Instantiates three axis_proc_simple pipelines (X, Y, Z)
        - Registers X/Y/Z outputs and defines vector-ready timing
        - Integrates the UART-like serializer to stream XYZ over one output pin when a new vector is ready

10. Implementation Flow

We took the RTL for the mag backend design and carried it through a complete digital implementation flow: constraint definition (SDC), logic synthesis in Genus (mapped to the GPDK45 standard-cell library), physical implementation in Innovus (floorplan, power, placement, routing), and signoff checks (post-route timing, full connectivity, and antenna).

The end state is a fully routed database with physical verification at the level required for this project: no connectivity violations, no antenna violations, and positive timing slack under the applied constraints. The main directory is "/home/hdi3084/CE493".

11. Timing constraints (SDC)

To constrain synthesis and physical implementation consistently, we created an SDC file named top_vol6.sdc in the vol6 directory. This SDC defined the primary clock and basic timing intent. Given the ADC characteristics, we decided to test our digital section using 100 MHz clock frequency for lower power consumption. However, it would be functional up to around 591MHz.

top_vol6.sdc contents (as used in final simulations):

Main clock on port clk (100 MHz):

```
create_clock -name CORE -period 10.0 [get_ports clk]
```

Clock grouping (single clock captured as its own group):

```
set_clock_groups -asynchronous -group {CORE}
```

Reset handling (avoid timing checks through async reset deassert paths if rst_n exists):

```
set_false_path -from [get_ports rst_n] -to [get_pins */SN]
```

12. Logic synthesis in Genus (GPDK45)

12.1. Genus synthesis script

We captured the synthesis flow in a TCL script placed in scripts/:

scripts/genus_vol6_gpdk45.tcl contents (as used):

Genus synthesis for mag_backend_simple using GPDK045 (vol6)

Read RTL:
```
 read_hdl ../adc_axis_demux.v
 read_hdl ../axis_osr_mavg.v
 read_hdl ../axis_offset_cal.v
 read_hdl ../axis_drift_hp.v
 read_hdl ../axis_gain.v
 read_hdl ../axis_proc_simple.v
 read_hdl ../mag_backend_simple.v
```

Technology libraries:
```
 set_db library
/vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/timing/fast_vdd1v0_basicCells.lib

set_db lef_library {
 /vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/lef/gsclib045_tech.lef
 /vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/lef/gsclib045_macro.lef
 /vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/lef/gsclib045_multibitsDFF.lef
 }
```

Elaboration and top selection:
```
 elaborate
 current_design mag_backend_simple
```

Read constraints:
```
 read_sdc ../top_vol6.sdc
```

Synthesis stages:
```
 syn_generic
 syn_map
 syn_opt
```

Reports:
```
 report_timing > timing.rpt
 report_area > area.rpt
```

Write gate-level netlist:
```
 write_hdl > mag_backend_simple_syn.v
```

Exit

12.2. Running Genus

We ran synthesis in batch mode from the Synthesis folder and logged the run:

```
cd ~/CE493/vol6/Synthesis
 genus -batch -files ../scripts/genus_vol6_gpdk45.tcl | tee ../logs/genus_vol6_gpdk45.log
```

Key synthesis outputs included:
 Synthesis/mag_backend_simple_syn.v (gate-level netlist)
 Synthesis/timing.rpt and Synthesis/area.rpt (Genus reports)

13. Innovus MMMC setup (.view file)

Innovus timing/RC context was provided via a MMMC view file in backend/. We created it by copying the tutorial view file and editing it for GPDK45 and our SDC:

```
cd ~/CE493/vol6/backend
 cp /vol/ece303/genus_tutorial/alu_conv.view mag_backend_vol6.view
 vim mag_backend_vol6.view
```

Key edits made inside mag_backend_vol6.view were:

Operating condition:
```
 create_op_cond -name op -library_file
{/vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/timing/fast_vdd1v0_basicCells.lib} -P
{1.0} -V {1.0} -T {25.0}
```

Library set:
```
 create_library_set -name lib -timing
{/vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/timing/fast_vdd1v0_basicCells.lib}
```

Constraint mode:

```
 create_constraint_mode -name sdc -sdc_files {../top_vol6.sdc}
```

14. Physical implementation in Innovus

14.1. Launching Innovus and importing the design

From the backend directory, we launched Innovus:

```
cd ~/CE493/vol6/backend
 source /vol/ece303/genus_tutorial/cadence.env
 Innovus
```

In the GUI (File → Design Import), we imported:

- Verilog netlist: ../Synthesis/mag_backend_simple_syn.v
- LEF files:
  /vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/lef/gsclib045_tech.lef
  /vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/lef/gsclib045_macro.lef
  /vol/eecs391/GPDK045/gsclib045_all_v4.8/gsclib045/lef/gsclib045_multibitsDFF.lef
- MMMC view file: mag_backend_vol6.view
- Power net: VDD, Ground net: VSS
- Top cell: mag_backend_simple

14.2. Floorplanning and global net setup

In the Innovus console, we set the process and created a basic floorplan. To ensure power/ground pins were globally recognized, we defined and connected global nets. The commands used during this stage were:

```
setDesignMode -process 45
 floorPlan -r 1.0 0.63 2 2 2 2

set init_pwr_net {VDD}
 set init_gnd_net {VSS}
 globalNetConnect VDD -type pgpin -pin VDD -inst * -override
 globalNetConnect VSS -type pgpin -pin VSS -inst * -override
 globalNetConnect VDD -type tiehi
 globalNetConnect VSS -type tielo
```

14.3. IO pin assignment

IO pins were assigned using the Pin Editor (Edit → Pin Editor), where we placed pins on M3 with a left edge assignment for inputs and outputs. This provided predictable routing access and matched the project convention.

14.4. Placement

We placed standard cells using:

```
setPlaceMode -reset
 setPlaceMode -congEffort auto -timingDriven 1
 setPlaceMode -fp false
 placeDesign
```

14.5. Power distribution network (PDN): ring and stitching

We created a core power ring and stitched it into the design:

```
addRing -nets {VSS VDD} -type core_rings -follow io -layer {top M5 bottom M5 left M4 right M4} -width {top 2 bottom 2 left 2 right 2} -spacing {top 1 bottom 1 left 1 right 1}

sroute -connect { padPin padRing corePin blockPin } -layerChangeRange {1 10} -nets {VDD VSS} -allowJogging 1 -allowLayerChange 1
```

During PDN bring-up, the default power routing behavior resulted in the top and bottom VDD segments not being fully connected, which initially produced connectivity violations. We manually connected the missing VDD segments so that the power ring and rails formed a continuous network. After this manual correction, the design passed connectivity checks cleanly.

14.6. Filler insertion and routing

We inserted filler cells and then routed the design:

```
addFillerGap 0.6
 addFiller -cell FILL1 FILL2 FILL4 FILL8 FILL16 FILL32 FILL64 -prefix F -markFixed

setAnalysisMode -analysisType onChipVariation
 setNanoRouteMode -quiet -drouteFixAntenna false
 setNanoRouteMode -quiet -routeTopRoutingLayer 6
```

routeDesign -globalDetail

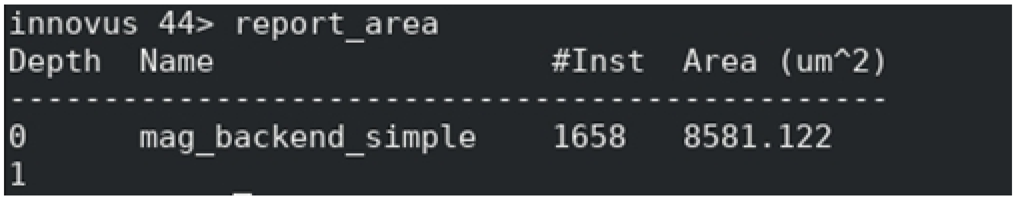
15. Reporting and signoff checks

15.1. Area report

To report the placed design area inside Innovus, we ran: report_area

mag_backend_simple had 1658 instances and a total cell area of 8581.122 µm².

Screenshot from the terminal showing the area report is given in Figure 15.1.



```
innovus 44> report_area
Depth  Name                #Inst  Area (um^2)
----------------------------------------------
0      mag_backend_simple   1658   8581.122
1
```

Figure 15.1: *Area Report Summary from Innovus Terminal*

15.2 Post-route timing (timeDesign)

We ran post-route timing with path and DRV reporting:

timeDesign -postRoute -pathReports -drvReports -slackReports -numPaths 200 -prefix mag_backend_vol6_postRoute -outDir timingReports_vol6

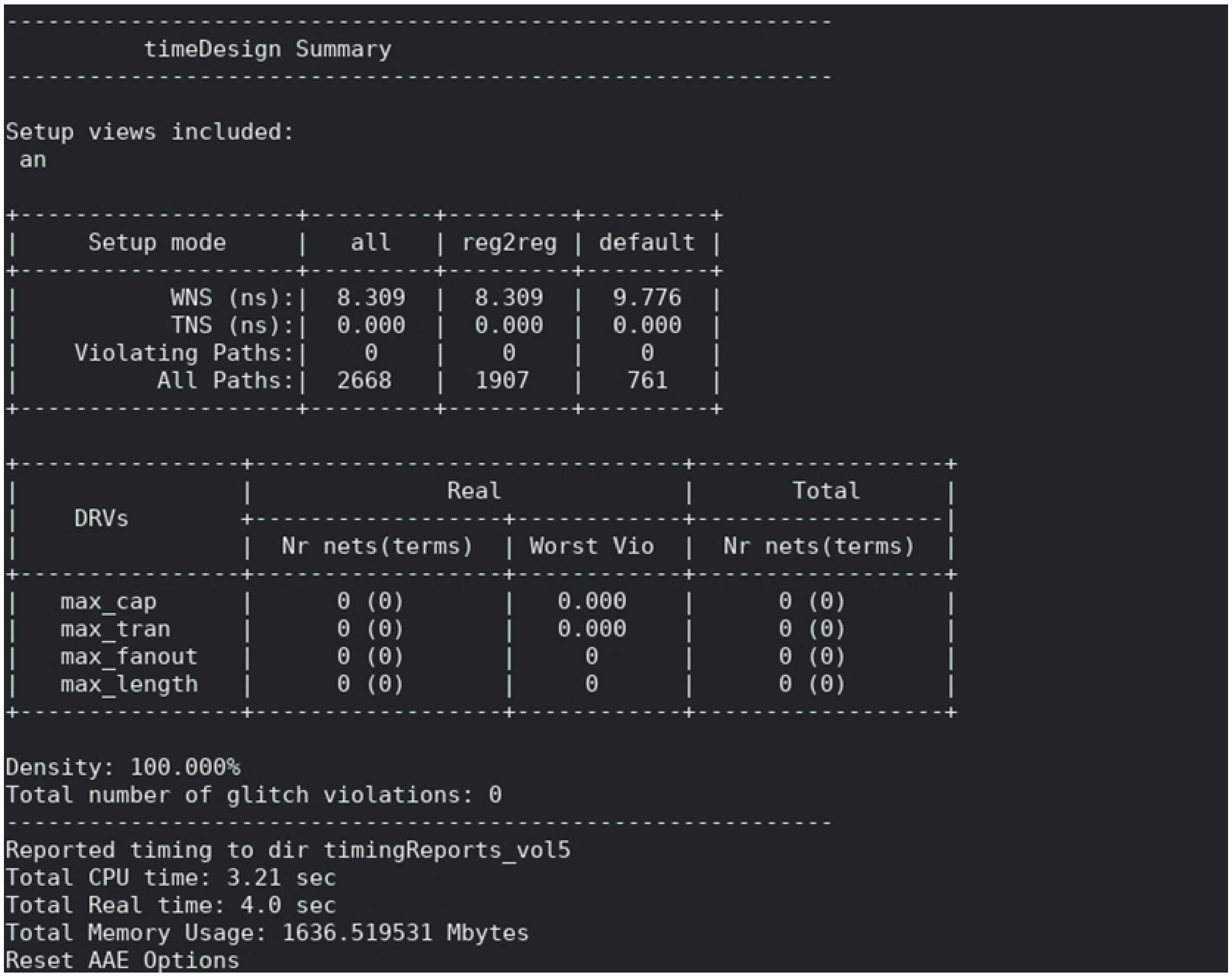
Interpretation of the warnings:

- Innovus reported that tQuantus technology was not specified for lower-node extraction and therefore performed postRoute extraction at effortLevel low. This reduces parasitic fidelity compared to a fully calibrated signoff flow.
- Innovus also reported that a cap table file was not specified; an internal cap table was generated. This again reduces absolute parasitic accuracy but remains sufficient for methodology demonstration.

Timing outcome (shown in Figure 15.2):

- WNS was positive (8.309 ns), TNS was 0, and there were 0 violating paths.

- No DRV violations were reported (max_cap, max_tran, max_fanout, max_length all zero).



```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

Setup views included:
 an

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):|  8.309  |  8.309  |  9.776  |
|           TNS (ns):|  0.000  |  0.000  |  0.000  |
|    Violating Paths:|    0    |    0    |    0    |
|          All Paths:|  2668   |  1907   |   761   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |              Real             |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |      0 (0)       |   0.000    |      0 (0)       |
|   max_tran     |      0 (0)       |   0.000    |      0 (0)       |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 100.000%
Total number of glitch violations: 0
------------------------------------------------------------
Reported timing to dir timingReports_vol5
Total CPU time: 3.21 sec
Total Real time: 4.0 sec
Total Memory Usage: 1636.519531 Mbytes
Reset AAE Options
```

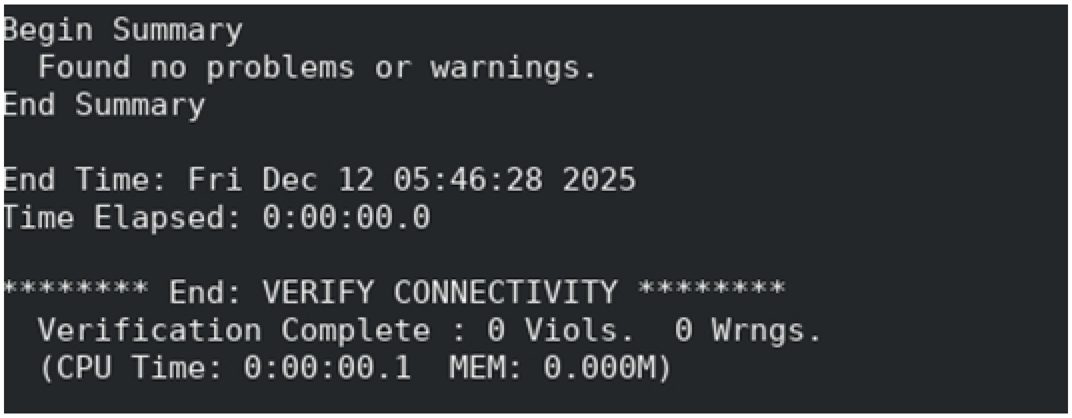
Figure 15.2: *Time Design Summary at 100 MHz clock from Innovus Terminal*

15.3. Connectivity verification

To confirm the physical database contained no opens/shorts and that all terminals were connected, we ran:

verifyConnectivity -type all -error 1000 -warning 50

Result: 0 violations and 0 warnings, confirming the manual VDD continuity fix and overall routing correctness. Screenshot from the terminal is shown in Figure 15.3.



```
Begin Summary
  Found no problems or warnings.
End Summary

End Time: Fri Dec 12 05:46:28 2025
Time Elapsed: 0:00:00.0

******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.
  (CPU Time: 0:00:00.1  MEM: 0.000M)
```

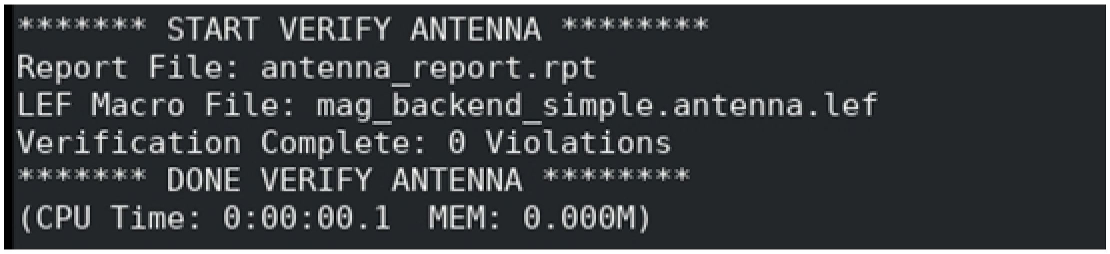
Figure 15.3: *Connectivity Report from Innovus Terminal*

15.4. Antenna verification

To verify that routed nets do not violate antenna rules, we ran:

verifyProcessAntenna -reportfile antenna_report.rpt -error 1000

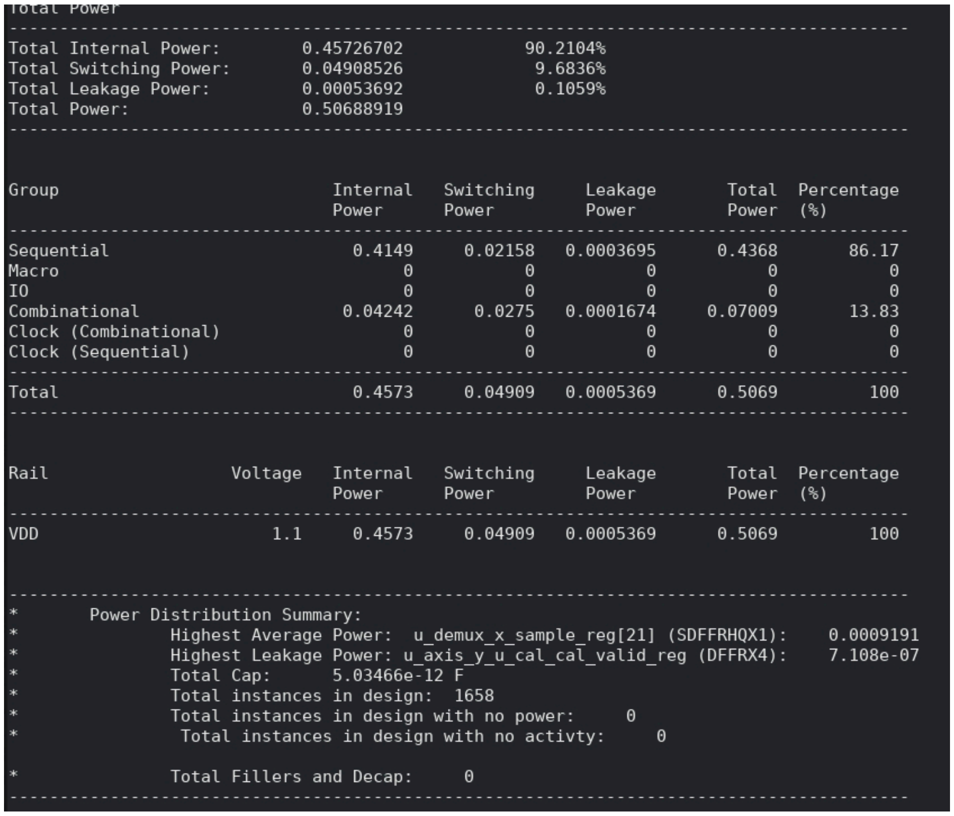
The result in Figure 15.4 shows that the verification is complete: 0 antenna violations.



```
******* START VERIFY ANTENNA ********
Report File: antenna_report.rpt
LEF Macro File: mag_backend_simple.antenna.lef
Verification Complete: 0 Violations
******* DONE VERIFY ANTENNA ********
(CPU Time: 0:00:00.1  MEM: 0.000M)
```

Figure 15.4: *Antenna Report from Innovus Terminal*

15.5. Power Specs



```
Total Power
-----------------------------------------------------------------------------------------
Total Internal Power:        0.45726702           90.2104%
Total Switching Power:       0.04908526            9.6836%
Total Leakage Power:         0.00053692            0.1059%
Total Power:                 0.50688919
-----------------------------------------------------------------------------------------


Group                           Internal   Switching     Leakage       Total  Percentage
                                Power      Power         Power         Power  (%)
-----------------------------------------------------------------------------------------
Sequential                        0.4149     0.02158   0.0003695      0.4368       86.17
Macro                                  0           0           0           0           0
IO                                     0           0           0           0           0
Combinational                    0.04242      0.0275   0.0001674     0.07009       13.83
Clock (Combinational)                  0           0           0           0           0
Clock (Sequential)                     0           0           0           0           0
-----------------------------------------------------------------------------------------
Total                             0.4573     0.04909   0.0005369      0.5069         100
-----------------------------------------------------------------------------------------


Rail                  Voltage   Internal   Switching     Leakage       Total  Percentage
                                Power      Power         Power         Power  (%)
-----------------------------------------------------------------------------------------
VDD                       1.1     0.4573     0.04909   0.0005369      0.5069         100


-----------------------------------------------------------------------------------------
*       Power Distribution Summary:
*               Highest Average Power:  u_demux_x_sample_reg[21] (SDFFRHQX1):    0.0009191
*               Highest Leakage Power: u_axis_y_u_cal_cal_valid_reg (DFFRX4):    7.108e-07
*               Total Cap:      5.03466e-12 F
*               Total instances in design:  1658
*               Total instances in design with no power:     0
*                Total instances in design with no activty:     0

*               Total Fillers and Decap:     0
-----------------------------------------------------------------------------------------
```

Figure 15.5: *Power Report from Innovus Terminal*

Post-layout power analysis (default primary input activity) was performed in Innovus using the finalized placed-and-routed netlist of the digital backend, operating at a 1.1 V supply, and a 100 MHz clock frequency. The total estimated power consumption of the design is approximately 0.507 mW, with the dominant contribution coming from internal dynamic power (≈90%), followed by net switching power (≈9.7%), and a negligible leakage component (≈0.1%). As expected for a heavily pipelined and sequentially dominated signal-processing design, sequential elements account for the majority of power consumption (≈86%), while combinational logic contributes the remaining ≈14%. The low leakage power confirms that static power is not a limiting factor for this architecture in the selected technology node. Overall, the results indicate that the digital backend achieves its signal-processing objectives with modest power consumption, validating the suitability of the chosen oversampling, calibration, and drift-suppression architecture for low-power magnetic sensing applications.

16. Final saved deliverables

At the end of implementation, we saved the database and exported implementation artifacts (netlists and SDF) for downstream verification and documentation. The save/export commands used were:

```
saveDesign mag_backend_vol6_final.enc (and/or saveDesign vol6_final.enc)
 saveNetlist -phys -includePowerGround mag_backend_vol6_phy.v -excludeLeafCell
 saveNetlist mag_backend_vol6_nophy.v -excludeLeafCell
 write_sdf mag_backend_vol6.sdf
```

Overall, the design was synthesized and physically implemented successfully in GPDK45. Post-route timing showed large positive slack with no DRVs, full connectivity verification passed with zero violations, and antenna checking completed with zero violations. The only notable manual intervention was the correction of an initial VDD discontinuity between default top/bottom power segments, after which the PDN and full design connectivity were clean.

17. Final Layout

The final layout with the area annotated can be seen in Figure 17.1 below.

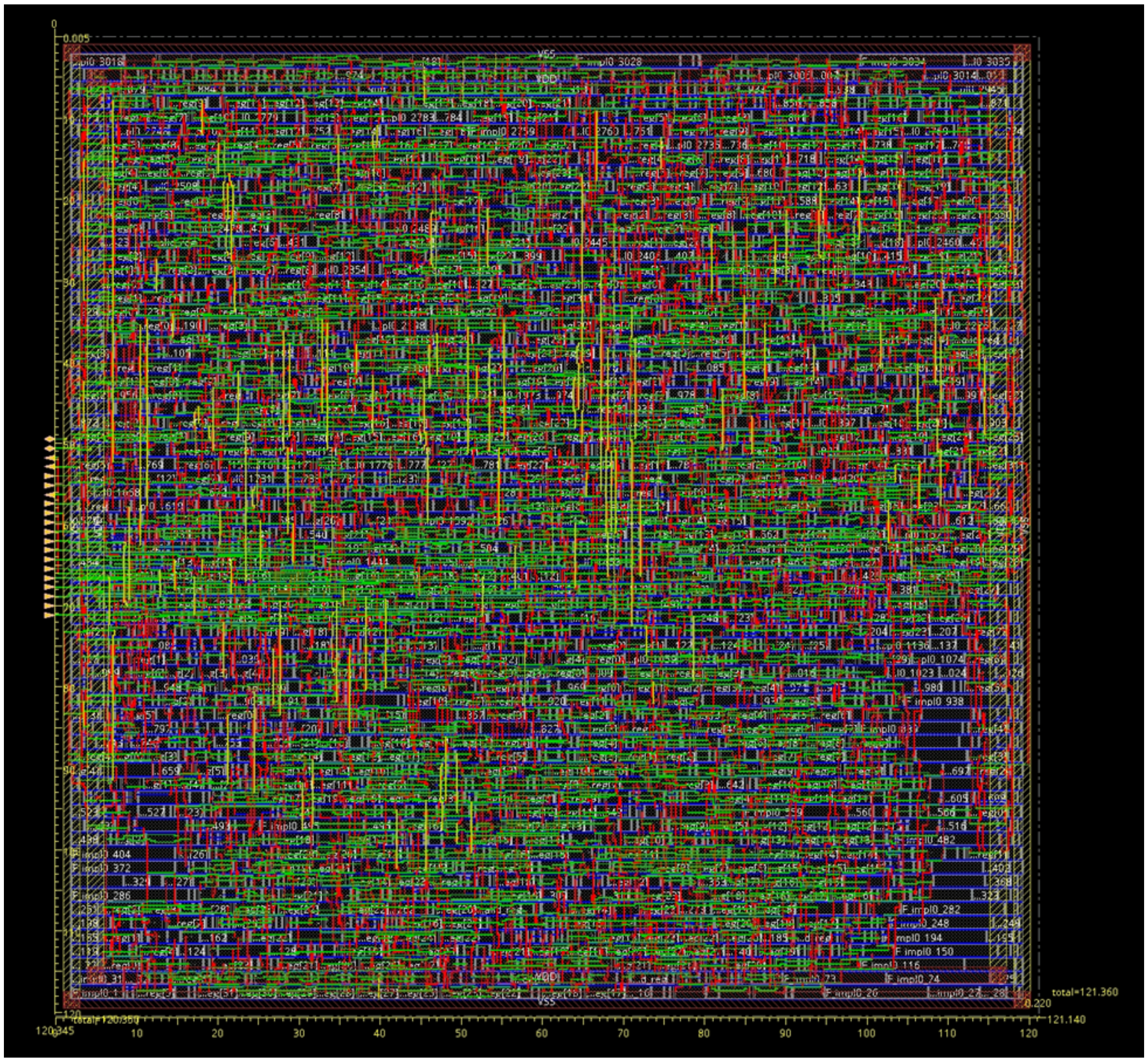


Figure 17.1: *Final Layout (Physical View) with Area Annotated*

The floorplan view can be seen in Figure 17.2.

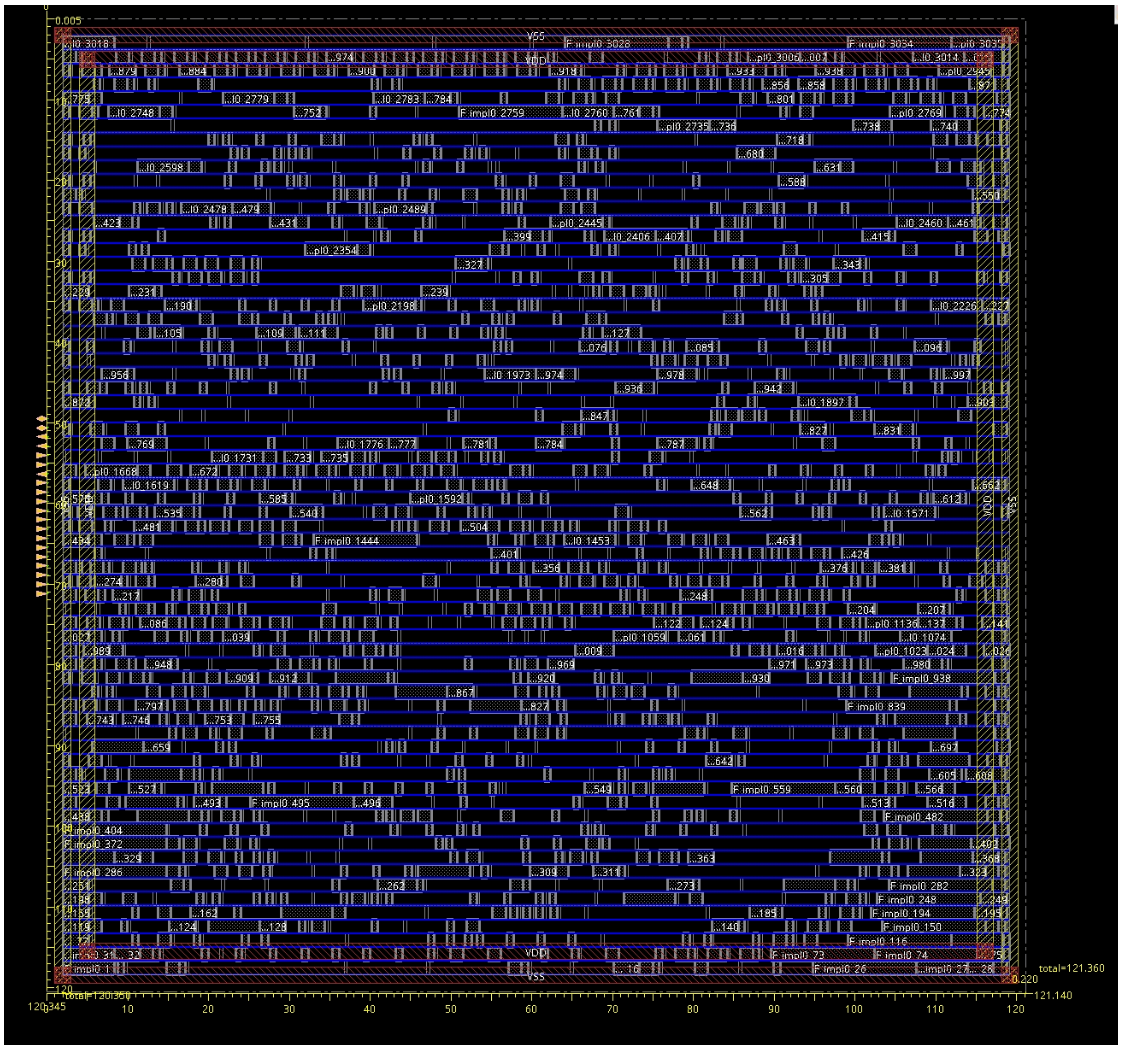


Figure 17.2: *Final Layout (Floorplan View) with Area Annotated*

18. Verification of the Digital Block

18.1. Verification flow and methodology (RTL → synthesis → PNR)

Our verification flow was incremental: we verified each submodule in isolation at RTL, then progressively integrated and re-verified at the next hierarchy level, and finally repeated the same end-to-end simulations on the synthesized and post-PNR netlists. This approach lets us catch interface/format issues early (e.g., fixed-point scaling, valid timing, OSR counters, calibration sequencing), before those issues become harder to debug in a flattened gate-level environment.

Concretely, each processing block (ADC demux, OSR moving-average decimator, offset calibration, drift/flicker suppression high-pass, and gain) was first simulated to confirm its standalone functionality, expected latency, and valid-pulse behavior. After module-level closure, we synthesized the modules as we moved upward and ensured that the synthesized versions preserved the intended sequential behavior. Once the full top-level RTL was stable, we ran top-level simulations using the same testbenches and generated CSV logs. After place-and-route (PNR) in Innovus, we ran the same testbenches again on the post-PNR netlist (with minor wrapper changes such as power pins/instance naming), producing parallel CSV outputs (RTL vs PNR). This repeated pattern was central to building confidence that physical implementation did not change functionality.

Across all three top-level tests described below, the Verilog testbenches produce deterministic CSV files containing (1) the ideal/reference model values computed inside the testbench, (2) the DUT outputs (RTL or PNR), and (3) additional internal reference signals (e.g., the drift baseline estimate). Those CSV logs were then imported into Python locally, and the nine plots in this section were generated from that data.

The top-level simulation plots shown in this section come from three primary testbenches:

1. **`tb_mag_backend_uart.v`**
   Validates the UART-like serial streaming output path. The backend produces XYZ updates internally, and whenever an output-valid event occurs, the design serializes one complete XYZ frame. The testbench decodes the serial bitstream back into signed fixed-point values and compares them against the reference XYZ values computed by the ideal model in the testbench.

2. **`tb_mag_backend_flicker.v`**
   Validates flicker/drift suppression behavior using an injected slow “random-walk” drift component on the X-axis input, combined with a faster sinusoidal signal component. The testbench logs the injected drift, the internal reference baseline estimate (EMA), the total input (drift+signal), and the drift-suppressed high-pass output.

3. **`tb_mag_backend_3d.v`**
   Validates the full backend pipeline under a smooth, highly wavy 3D trajectory stimulus. The testbench produces a time-varying XYZ input trajectory (continuous and multi-tone), runs it through the same ideal reference model used for checking, and compares ideal vs RTL vs post-PNR across all axes. It also generates a 3D trajectory plot and an aggregate error plot.

Each testbench was also slightly modified to run on the post-PNR netlist, producing corresponding “*_pnr.csv” outputs. The Python plotting scripts then overlay RTL and PNR results on the same figures. The mentioned Python Scripts can be found in Appendix Section B.1.

18.2. UART streaming output validation

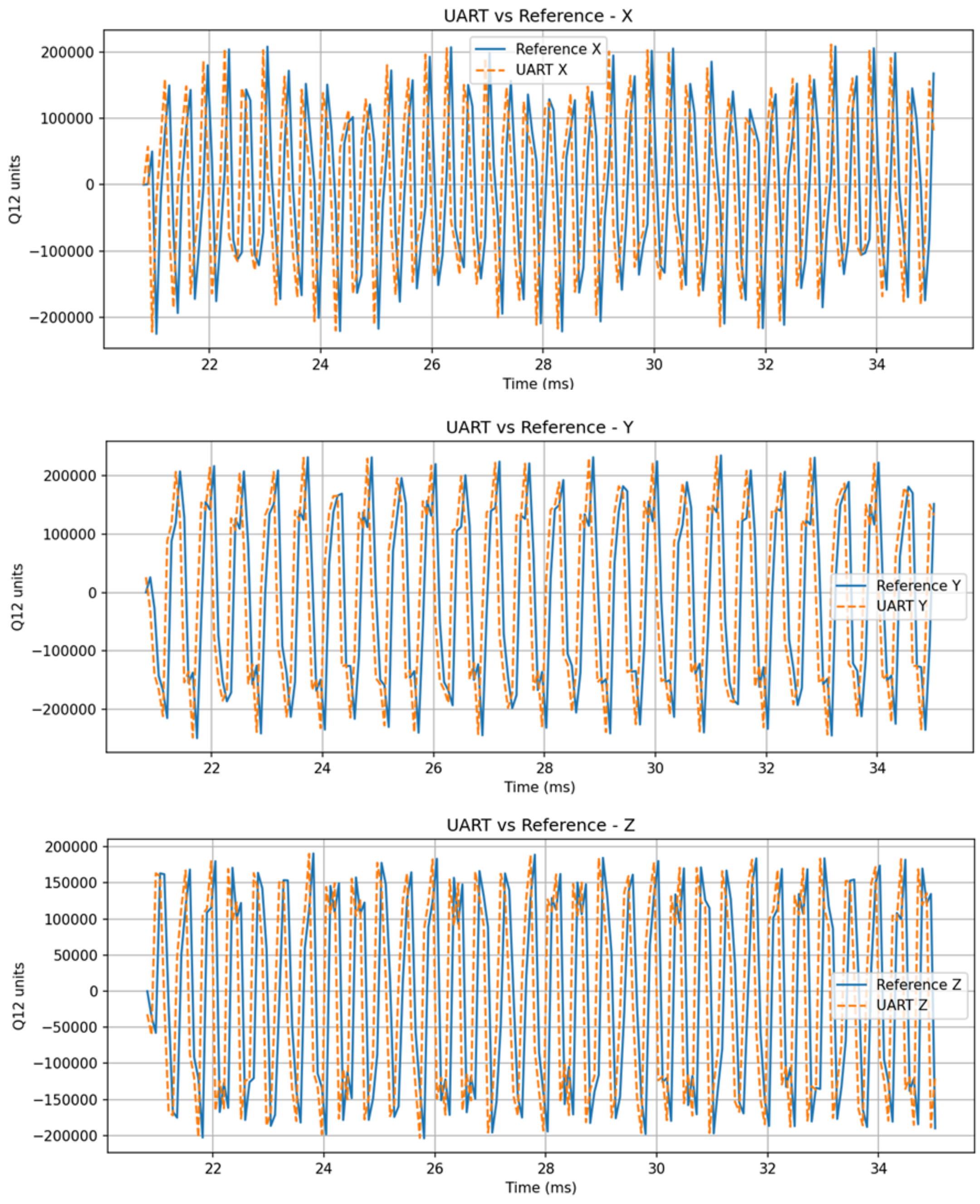


Figure 18.1: *UART vs Reference overlays for Z, X, and Y*

These three plots in Figure 18.1 (Z, X, Y) show the decoded UART frame data plotted against the reference axis values over time. The key result is that the UART-decoded waveforms sit essentially on top of the reference curves, indicating:

- The serial framing is correct (start/stop framing and bit ordering are consistent).
- The payload packing/unpacking is correct (the decoded values reconstruct the same signed fixed-point samples).
- There is no obvious evidence of dropped frames or bit-slips (which would appear as abrupt discontinuities, sign inversions, or large step errors).

Thus, this figure supports the claim that the backend can export processed magnetic field values over a single-bit serial pin without requiring a wide parallel output bus.

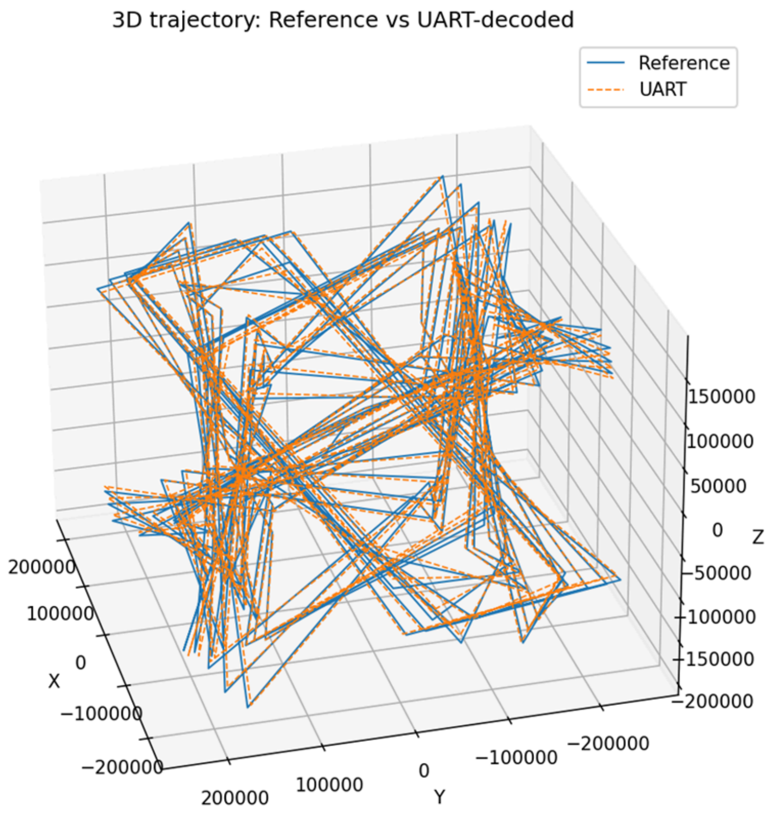


Figure 18.2: *3D trajectory: Reference vs UART-decoded*

This 3D plot reconstructs the spatial trajectory using the decoded UART output and compares it directly to the reference trajectory. When the UART protocol is correct, the reconstructed trajectory should trace the same path (within quantization and filtering lag limits).

The visual overlap here shows two important properties:

1. Correct axis association and ordering: if X/Y/Z were swapped (or sign-inverted), the 3D path would appear rotated, mirrored, or otherwise misaligned.
2. Temporal coherence of frames: a framing/timing issue would scramble the path, producing discontinuous teleporting segments instead of a continuous trace.

18.3. Flicker/drift suppression validation

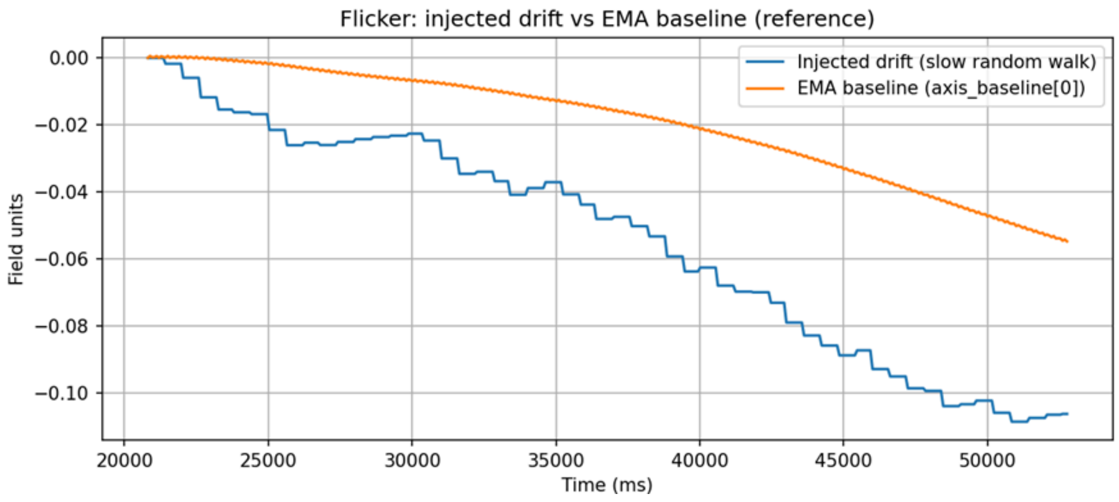


Figure 18.3: *Injected drift vs EMA baseline*

This figure plots two signals over time:

- **Injected drift**: a slow random-walk component added to the X-axis input to emulate low-frequency drift / flicker-like behavior.
- **EMA baseline**: the backend's baseline estimate produced by an exponential moving average (EMA) update.

The key point is that the EMA baseline changes smoothly and tracks the long-term drift trend, but it does not chase rapid fluctuations. This is the intended behavior: the baseline is designed to represent the slowly varying DC component (drift), not the actual signal content. Visually, that appears as a smoother curve that follows the general direction of the drift but with a lag (because it is a low-pass estimate).

This plot demonstrates the mechanism that enables drift suppression: we are explicitly estimating the “slow component” that will later be subtracted.

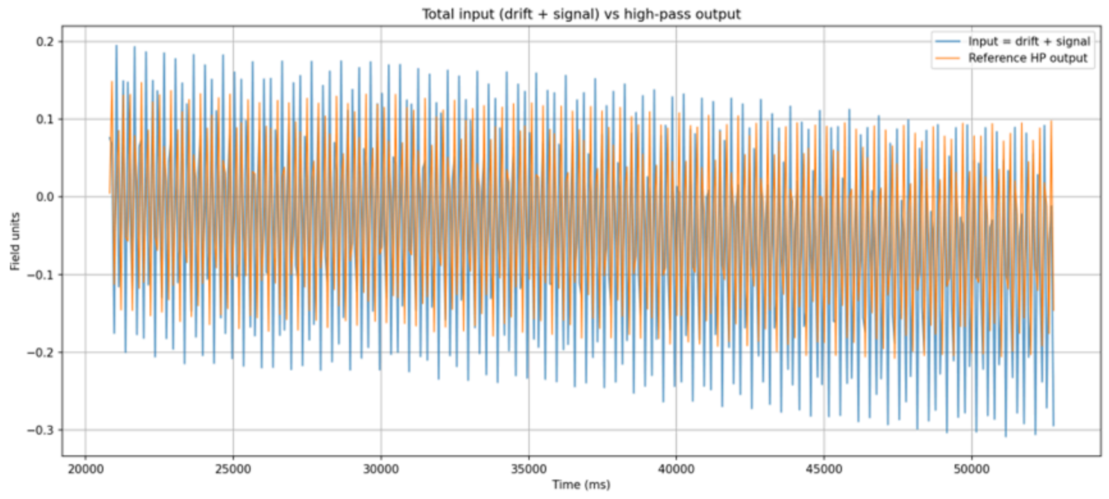


Figure 18.4: *Total input (drift + signal) vs high-pass output*

This plot compares:

- **Input = drift + signal** (what the ADC effectively sees),
- **High-pass output** (after baseline subtraction).

Even though the total input gradually shifts due to drift, the high-pass output remains centered around zero and primarily reflects the faster-varying signal component. In other words, the drift is largely removed in the output domain. This is the primary functional evidence that the backend is suppressing low-frequency drift: the slow bias does not propagate into the reported magnetic field values after filtering.

18.4. 3D backend behavior and RTL vs post-PNR correlation

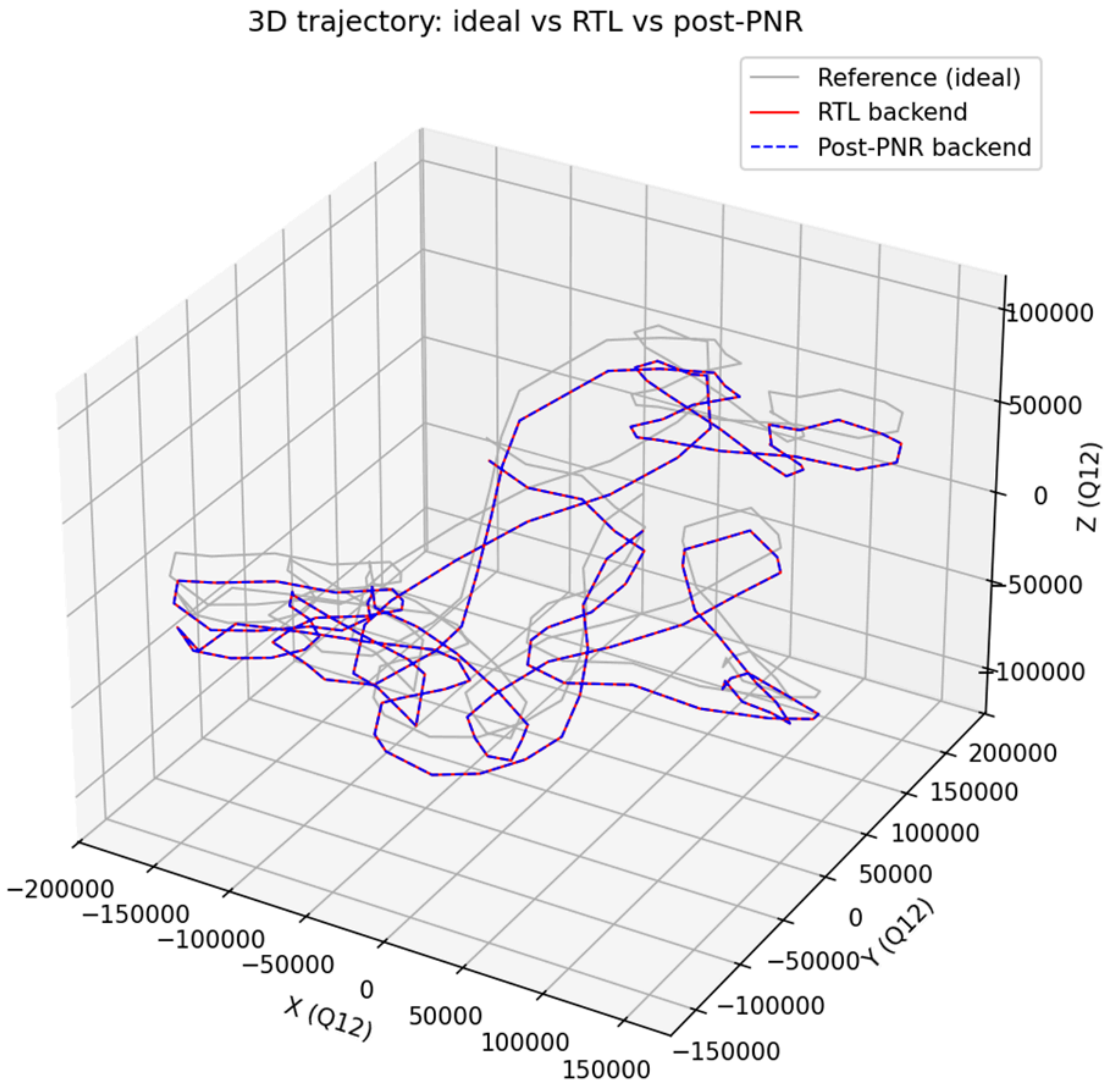


Figure 18.5: *3D trajectory: ideal vs RTL vs post-PNR*

This 3D plot overlays three trajectories:

- Ideal reference (stimulus/model),
- RTL backend output,
- Post-PNR backend output.

The most important observation is that RTL and post-PNR nearly overlap, indicating that synthesis and place-and-route preserved the intended sequential behavior. This is expected from the digital implementation flow: the physically implemented netlist is functionally equivalent to RTL for the exercised operating conditions.

Any residual deviation relative to the ideal trajectory is expected because the backend intentionally filters and decimates the signal; filtering introduces latency and can introduce small amplitude/phase differences depending on the trajectory frequency content. What matters is that RTL and PNR are consistent with each other and follow the same filtered interpretation of the input.

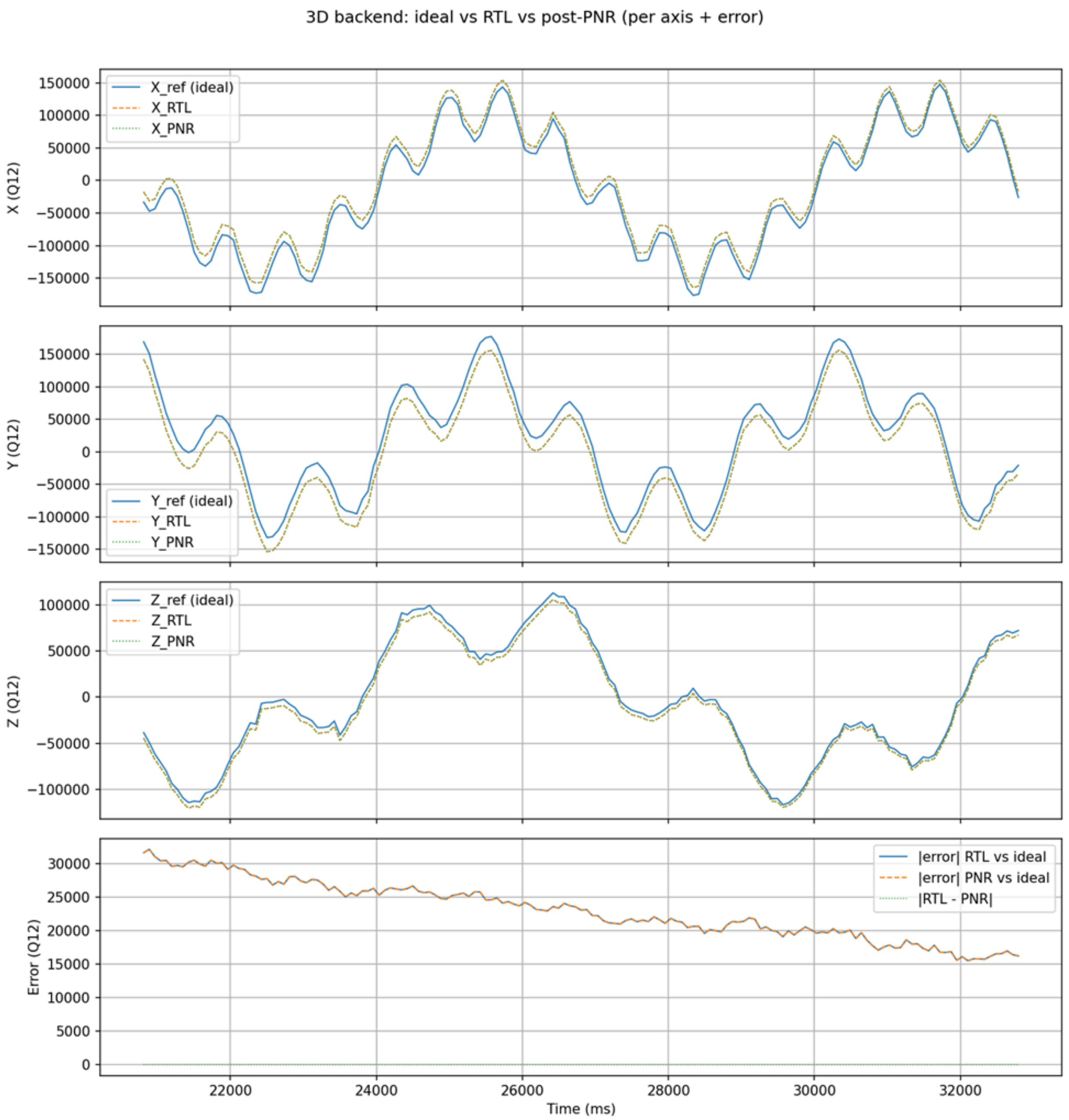


Figure 18.6: *3-D Backend, Ideal vs RTL vs PNR (Per-axis Overlay and Error)*

This combined Figure 18.6 shows:

- **X axis vs time**: ideal, RTL, and PNR overlaid
- **Y axis vs time**: ideal, RTL, and PNR overlaid
- **Z axis vs time**: ideal, RTL, and PNR overlaid
- **Error plot**: magnitude of error for RTL vs ideal, PNR vs ideal, and RTL vs PNR

The axis plots demonstrate that the backend outputs are smooth, band-limited, and track the expected trajectory over time, which is consistent with the design intent of oversampling + decimation + drift suppression. The error plot provides a compact quantitative summary; the RTL vs PNR error remaining near zero indicates that post-PNR does not introduce functional differences. The (RTL/PNR) vs ideal error reflects the combined effect of filtering, decimation, and the inherent latency of the chain, and it is expected, given that the backend output is not a raw ADC code but a processed signal.

Taken together, this figure supports both functional correctness (the waveforms behave as expected) and implementation correctness (RTL and PNR align).

18.5. Summary statement for synthesis/implementation closure

We synthesized the design incrementally as we built upward through the hierarchy, and then we repeated the same top-level verification on the post-PNR netlist. The close agreement between RTL and post-PNR outputs across the UART, flicker/drift, and 3D trajectory tests provides evidence that the design is reasonable across the full digital implementation flow, from behavioral RTL through gates and physical layout.

All files mentioned above can be accessed through the directory: "/home/hdi3084/CE493". "/home/hdi3084/CE493/vol6" includes the final version of the digital side of the project; however, most testbenches are under the directory "/home/hdi3084/CE493/vol5".

References

[1] A. Bilotti, G. Monreal and R. Vig, "Monolithic magnetic Hall sensor using dynamic quadrature offset cancellation," in IEEE Journal of Solid-State Circuits, vol. 32, no. 6, pp. 829-836, June 1997, doi: 10.1109/4.585275.

[2] M. Crescentini, S. F. Syeda, and G. P. Gibiino, “Hall-effect current sensors: Principles of operation and implementation techniques,” *IEEE Sensors Journal*, vol. 22, no. 11, pp. 10137–10151, Jun. 2022.

[3] X. Hao, X. Wang, and Y. Li, “Design and optimization of a RF mixer for electromagnetic sensor backend,” *Eng*, vol. 6, no. 11, p. 286, Oct. 2025.

[4] N. Maciel, E. Marques, L. Naviner, Y. Zhou, and H. Cai, “Magnetic tunnel junction applications,” *Sensors*, vol. 20, no. 1, p. 121, Dec. 2019.

[5] R. Holyaka, T. Marusenkova, and O. Shpur, “Intelligent mixed-signal embedded systems for electromagnetic tracking applications,” in *Digital Ecosystems: Interconnecting Advanced Networks with AI Applications*, Springer, 2024, pp. 543–565.

[6] S. K. Lim and B. A. Wooley, “A high-speed sample-and-hold technique using a Miller hold capacitance,” *IEEE J. Solid-State Circuits*, vol. 26, no. 4, pp. 643–651, Apr. 1991.

[7] M. H. Mahantesh *et al.*, “Design of R-2R digital-to-analog converter using CMOS technology,” *Int. J. Innovative Res. Electr. Electron. Instrum. Control Eng.*, vol. 3, no. 9, pp. 1–5, Sep. 2025.

[8] S. Dutta, “Design of a strong-arm dynamic-latch comparator with high speed, low power and low offset for SAR-ADC,” *arXiv preprint arXiv:2209.07259*, Sep. 2022.

[9] A. K. Singh *et al.*, “Understanding synthesizable design methodologies for mixed-signal SAR ADCs,” *IEEE Int. Symp. Circuits Syst.*, pp. 1–5, 2025.

[10] C. Simard, “Noise Characteristics and Measurement Methodology for TMR and Hall-Effect Current Sensors,” Application Information AN296360, Allegro MicroSystems, Manchester, NH, USA, May 28, 2025. [Online]. Available: https://www.allegromicro.com/-/media/files/application-notes/an296360-tmr-noise.pdf

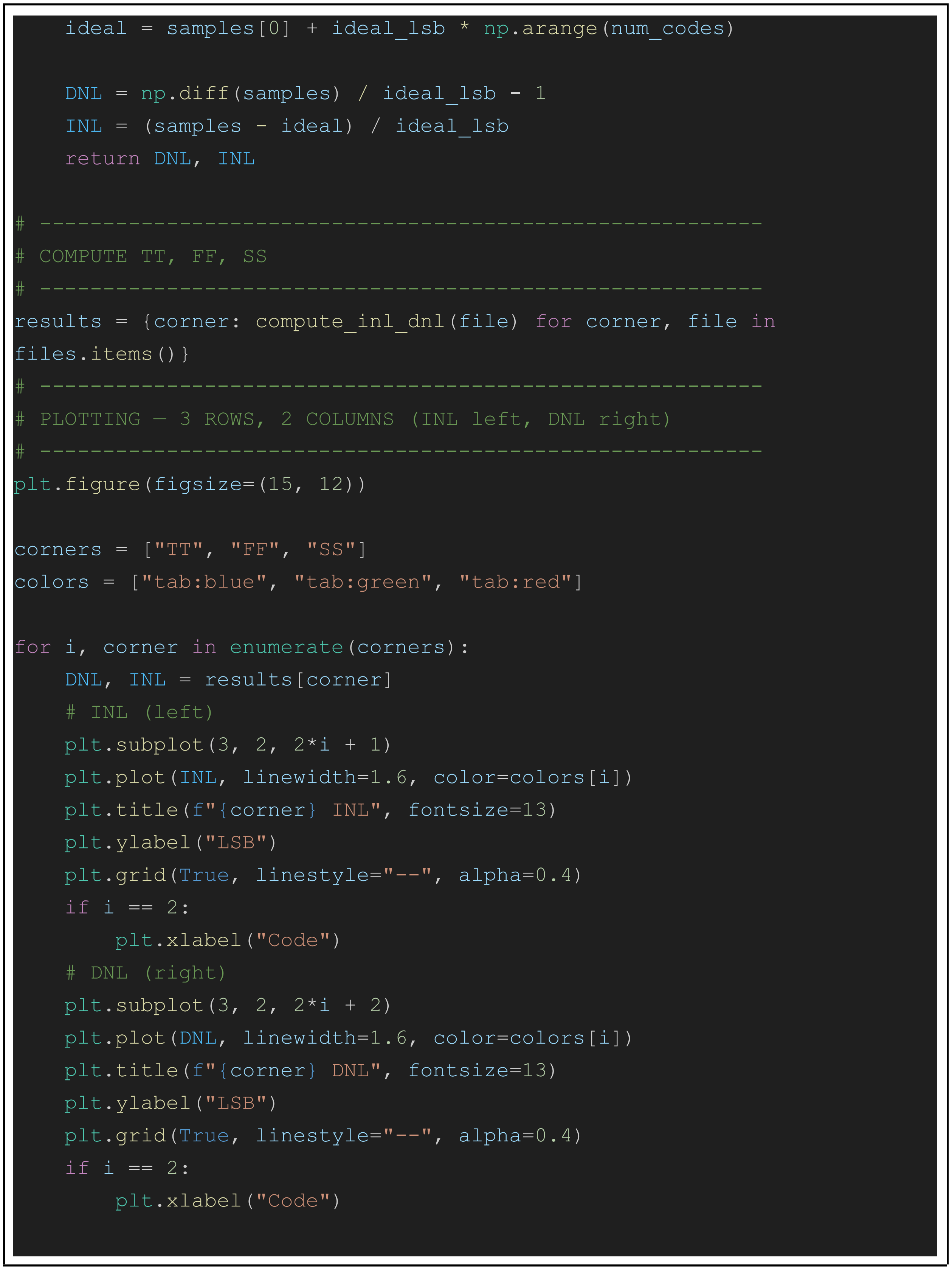
[11] Silicon Labs, “Improving ADC Resolution by Oversampling and Averaging,” Application Note AN118, Rev. 1.3, Jul. 2013. [Online]. Available: https://www.silabs.com/documents/public/application-notes/an118.pdf.

Appendix

A.1 DAC INL and DNL calculator implementation in Python

```python
import numpy as np
import matplotlib.pyplot as plt
# ---------------------------------------------------------
# USER SETTINGS
# ---------------------------------------------------------
N = 8
code_time = 20e-9
settle_frac = 0.75

files = {
    "TT": "dac_output.csv",
    "FF": "dac_output_ff.csv",
    "SS": "dac_output_ss.csv"
}
# ---------------------------------------------------------
# FUNCTION TO COMPUTE INL/DNL
# ---------------------------------------------------------
def compute_inl_dnl(file):
    data = np.loadtxt(file, delimiter=",", skiprows=1)
    t = data[:, 0]
    v = data[:, 1]
    sim_time = t[-1] - t[0]
    num_codes = int(np.floor(sim_time / code_time))
    samples = np.zeros(num_codes)
    for k in range(num_codes):
        t_start = t[0] + k * code_time
        t_end   = t_start + code_time
        idx = np.where((t >= t_start) & (t < t_end))[0]
        if len(idx) == 0:
            samples[k] = samples[k-1] if k > 0 else v[0]
            continue
        s_idx = idx[int(settle_frac * len(idx)):]
        samples[k] = np.mean(v[s_idx])
    ideal_lsb = (samples[-1] - samples[0]) / (num_codes - 1)
```



```
    ideal = samples[0] + ideal_lsb * np.arange(num_codes)

    DNL = np.diff(samples) / ideal_lsb - 1
    INL = (samples - ideal) / ideal_lsb
    return DNL, INL

# ----------------------------------------------------------
# COMPUTE TT, FF, SS
# ----------------------------------------------------------
results = {corner: compute_inl_dnl(file) for corner, file in
files.items()}
# ----------------------------------------------------------
# PLOTTING — 3 ROWS, 2 COLUMNS (INL left, DNL right)
# ----------------------------------------------------------
plt.figure(figsize=(15, 12))

corners = ["TT", "FF", "SS"]
colors = ["tab:blue", "tab:green", "tab:red"]

for i, corner in enumerate(corners):
    DNL, INL = results[corner]
    # INL (left)
    plt.subplot(3, 2, 2*i + 1)
    plt.plot(INL, linewidth=1.6, color=colors[i])
    plt.title(f"{corner} INL", fontsize=13)
    plt.ylabel("LSB")
    plt.grid(True, linestyle="--", alpha=0.4)
    if i == 2:
        plt.xlabel("Code")
    # DNL (right)
    plt.subplot(3, 2, 2*i + 2)
    plt.plot(DNL, linewidth=1.6, color=colors[i])
    plt.title(f"{corner} DNL", fontsize=13)
    plt.ylabel("LSB")
    plt.grid(True, linestyle="--", alpha=0.4)
    if i == 2:
        plt.xlabel("Code")
```

```python
plt.tight_layout()
plt.show()
```

A.2 ADC INL and DNL calculator implementation in Python

```python
import numpy as np
import matplotlib.pyplot as plt
# =================================
# Load CSV
# =================================
data = np.loadtxt("adc_digital_v2.csv", delimiter=",", skiprows=1)
done = data[:,1]      # DONE Y column
bits = data[:,3::2]   # take every Y column of the bits (skip X columns)
# bits shape should be (N_samples, 8)
if bits.shape[1] != 8:
    raise ValueError("Expected 8 SAR bit columns, found
{}".format(bits.shape[1]))
# Convert 0/2 V → 0/1
bits = (bits > 1.0).astype(int)

# =================================
# Detect DONE rising edges
# =================================
done_rise_idx = np.where((done[:-1] < 1.0) & (done[1:] > 1.0))[0]

codes = []
for idx in done_rise_idx:
    b7,b6,b5,b4,b3,b2,b1,b0 = bits[idx]
    code = (b7<<7)|(b6<<6)|(b5<<5)|(b4<<4)|(b3<<3)|(b2<<2)|(b1<<1)|(b0)
    codes.append(code)

codes = np.array(codes)
# =================================
# Compute DNL and INL
# =================================
unique_codes, counts = np.unique(codes, return_counts=True)
```

```
ideal = np.mean(counts)

DNL = (counts / ideal) - 1
INL = np.cumsum(DNL)

# ================================
# Plot results (green continuous lines)
# ================================
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.plot(unique_codes, DNL, color='green', linewidth=1.8)
plt.title("ADC DNL")
plt.xlabel("Code")
plt.ylabel("DNL (LSB)")
plt.grid(True)
plt.subplot(1,2,2)
plt.plot(unique_codes, INL, color='green', linewidth=1.8)
plt.title("ADC INL")
plt.xlabel("Code")
plt.ylabel("INL (LSB)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

A.3 Successive Approximation Registry implementation in Verilog-A

```
`include "constants.vams"
`include "disciplines.vams"

module sar_8bit_val(
    clk,
    start,
    cmp_out,
    sample,
    cmp_clk,
    done,
    dac_code,
    sar_result
);
    // Inputs
    input clk;       electrical clk;
    input start;     electrical start;
    input cmp_out;  electrical cmp_out;
    // Outputs
    output sample;           electrical sample;
    output cmp_clk;          electrical cmp_clk;
    output done;             electrical done;
    output [7:0] dac_code;  electrical [7:0] dac_code;
    output [7:0] sar_result; electrical [7:0] sar_result;
    // Threshold + logic levels
    parameter real clk_th = 1.0;
    parameter real vh = 2;
    parameter real vl = 0;
    // SETTLE time
    parameter integer SETTLE_CYCLES = 1;
    // FSM
    integer state;
    parameter IDLE=0, SAMPLE=1, SAMPLE_WAIT=2, HOLD=3, TRIAL=4, SETTLE=5,
DECIDE=6, FINISH=7;
    // Indices
    integer bit_index;
    integer settle_cnt;
    // bit storage
    integer dac_bits[7:0];
    integer result_bits[7:0];
    // analog driver vars
    real sample_val, cmp_clk_val, done_val;
    real dac_val0, dac_val1, dac_val2, dac_val3;
    real dac_val4, dac_val5, dac_val6, dac_val7;

    real sar_val0, sar_val1, sar_val2, sar_val3;
    real sar_val4, sar_val5, sar_val6, sar_val7;

    integer k; // loop index allowed at top-level only

    analog begin
        // ============================================================
        // INITIALIZATION (NO DECLARATIONS HERE)
        // ============================================================
        @(initial_step) begin
            state = IDLE;
            bit_index = 7;
            settle_cnt = 0;

            // legal: loop for integer array assignment
            for (k = 0; k < 8; k = k + 1) begin
                dac_bits[k] = 0;
                result_bits[k] = 0;
            end
        end
        // ==========================================================
        // MAIN FSM ON CLOCK EDGE (NO DECLARATIONS INSIDE!)
        // ==========================================================
        @(cross(V(clk)-clk_th, +1)) begin
            case(state)
                IDLE: begin
                    if (V(start) > clk_th) begin
                        bit_index = 7;
                        for (k = 0; k < 8; k = k + 1) begin
                            dac_bits[k] = 0;
                            result_bits[k] = 0;
                        end
                        state = SAMPLE;
                    end
                end

                SAMPLE:      state = SAMPLE_WAIT;
                SAMPLE_WAIT: state = HOLD;
                HOLD:        state = TRIAL;

                TRIAL: begin
                    dac_bits[bit_index] = 1;
                                        settle_cnt = SETTLE_CYCLES;
                    state = SETTLE;
                end

                SETTLE: begin
                    if (settle_cnt > 0)
                        settle_cnt = settle_cnt - 1;
                    else
                        state = DECIDE;
                end

                DECIDE: begin
                    if (V(cmp_out) > clk_th)
                        result_bits[bit_index] = 1;
                    else begin
                        result_bits[bit_index] = 0;
                        dac_bits[bit_index] = 0;
                    end

                    if (bit_index == 0)
                        state = FINISH;
                    else begin
                        bit_index = bit_index - 1;
                        state = TRIAL;
```

```
          end
        end

        FINISH: begin
          state = IDLE;
        end

    endcase
  end


  // =========================================================
  // OUTPUT CALCULATIONS ? no illegal transitions
  // =========================================================
  sample_val  = (state == SAMPLE || state == SAMPLE_WAIT) ? vh : vl;
  cmp_clk_val = (state == DECIDE) ? vh : vl;
  done_val    = (state == FINISH) ? vh : vl;
  dac_val0 = dac_bits[0] ? vh : vl;
  dac_val1 = dac_bits[1] ? vh : vl;
  dac_val2 = dac_bits[2] ? vh : vl;
  dac_val3 = dac_bits[3] ? vh : vl;
  dac_val4 = dac_bits[4] ? vh : vl;
  dac_val5 = dac_bits[5] ? vh : vl;
  dac_val6 = dac_bits[6] ? vh : vl;
  dac_val7 = dac_bits[7] ? vh : vl;
  sar_val0 = result_bits[0] ? vh : vl;
  sar_val1 = result_bits[1] ? vh : vl;
  sar_val2 = result_bits[2] ? vh : vl;
  sar_val3 = result_bits[3] ? vh : vl;
  sar_val4 = result_bits[4] ? vh : vl;
  sar_val5 = result_bits[5] ? vh : vl;
  sar_val6 = result_bits[6] ? vh : vl;
  sar_val7 = result_bits[7] ? vh : vl;

  // =========================================================
  // ANALOG OUTPUT DRIVING ? NO LOOPS
  // =========================================================
  V(sample)  <+ transition(sample_val, 1p, 1p);
  V(cmp_clk) <+ transition(cmp_clk_val, 1p, 1p);
  V(done)    <+ transition(done_val,   1p, 1p);

  V(dac_code[0]) <+ transition(dac_val0, 1p, 1p);
  V(dac_code[1]) <+ transition(dac_val1, 1p, 1p);
  V(dac_code[2]) <+ transition(dac_val2, 1p, 1p);
  V(dac_code[3]) <+ transition(dac_val3, 1p, 1p);
  V(dac_code[4]) <+ transition(dac_val4, 1p, 1p);
  V(dac_code[5]) <+ transition(dac_val5, 1p, 1p);
  V(dac_code[6]) <+ transition(dac_val6, 1p, 1p);
```

```
        V(dac_code[7]) <+ transition(dac_val7, 1p, 1p);

        V(sar_result[0]) <+ transition(sar_val0, 1p, 1p);
        V(sar_result[1]) <+ transition(sar_val1, 1p, 1p);
        V(sar_result[2]) <+ transition(sar_val2, 1p, 1p);
        V(sar_result[3]) <+ transition(sar_val3, 1p, 1p);
        V(sar_result[4]) <+ transition(sar_val4, 1p, 1p);
        V(sar_result[5]) <+ transition(sar_val5, 1p, 1p);
        V(sar_result[6]) <+ transition(sar_val6, 1p, 1p);
        V(sar_result[7]) <+ transition(sar_val7, 1p, 1p);

    end
endmodule
```

B.1. Top-Level Python Scripts for Digital Verification Visualization

(1- UART Verification, 2- 3D Verification (Ideal vs RTL vs PNR), 3- Drift Offset Verification)

B.1.1.

```
import argparse
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--csv", default="backend_uart_frames.csv")
    ap.add_argument("--phase", type=int, default=2, help="which phase to
plot (default: 2)")
    ap.add_argument("--q12_to_field", action="store_true", help="divide by
4096 to show 'field units'")
    args = ap.parse_args()

    df = pd.read_csv(args.csv)
    if df.empty:
        print("Empty CSV:", args.csv)
        return

    dfp = df[df["phase"] == args.phase].copy()
    if dfp.empty:
        print(f"No rows with phase=={args.phase} in {args.csv}")
        return

    t_ms = dfp["time_ns"].to_numpy(dtype=np.float64) / 1e6

    x_u = dfp["x_uart"].to_numpy(dtype=np.float64)
    y_u = dfp["y_uart"].to_numpy(dtype=np.float64)
    z_u = dfp["z_uart"].to_numpy(dtype=np.float64)

    y_r = dfp["x_ref"].to_numpy(dtype=np.float64)
    z_r = dfp["y_ref"].to_numpy(dtype=np.float64)
    x_r = dfp["z_ref"].to_numpy(dtype=np.float64)

    if args.q12_to_field:
        scale = 4096.0
        x_u /= scale; y_u /= scale; z_u /= scale
        x_r /= scale; y_r /= scale; z_r /= scale
        ylab = "Field units (Q12/4096)"
    else:
        ylab = "Q12 units"

    ex = x_u - x_r
    ey = y_u - y_r
    ez = z_u - z_r
    err_mag = np.sqrt(ex*ex + ey*ey + ez*ez)

    # Health stats
    stop_ok_rate = dfp["stop_ok"].mean()
    start_len_counts = dfp["start_len"].value_counts().to_dict()

    mae_x = np.mean(np.abs(ex))
    mae_y = np.mean(np.abs(ey))
    mae_z = np.mean(np.abs(ez))
    rmse  = np.sqrt(np.mean(ex*ex + ey*ey + ez*ez))

    print("=== UART Frame Decode Summary ===")
    print(f"Rows plotted: {len(dfp)}  (phase={args.phase})")
    print(f"Stop bit ok rate: {stop_ok_rate*100:.2f}%")
    print(f"start_len distribution: {start_len_counts}")
    print(f"MAE X={mae_x:.3f}, Y={mae_y:.3f}, Z={mae_z:.3f}  ({ylab})")
    print(f"RMSE vector magnitude = {rmse:.3f}  ({ylab})")

    # X/Y/Z comparisons
    plt.figure(figsize=(11, 4))
    plt.title("UART vs Reference - X")
    plt.plot(t_ms, x_r, label="Reference X")
    plt.plot(t_ms, x_u, label="UART X", linestyle="--")
    plt.xlabel("Time (ms)")
    plt.ylabel(ylab)
    plt.grid(True)
    plt.legend()

    plt.figure(figsize=(11, 4))
    plt.title("UART vs Reference - Y")
    plt.plot(t_ms, y_r, label="Reference Y")
    plt.plot(t_ms, y_u, label="UART Y", linestyle="--")
    plt.xlabel("Time (ms)")
    plt.ylabel(ylab)
    plt.grid(True)
    plt.legend()

    plt.figure(figsize=(11, 4))
    plt.title("UART vs Reference - Z")
    plt.plot(t_ms, z_r, label="Reference Z")
    plt.plot(t_ms, z_u, label="UART Z", linestyle="--")
    plt.xlabel("Time (ms)")
    plt.ylabel(ylab)
    plt.grid(True)
    plt.legend()

    # Error magnitude
    plt.figure(figsize=(11, 3.5))
    plt.title("UART decode error magnitude vs time")
    plt.plot(t_ms, err_mag, label="|UART - ref|")
    plt.xlabel("Time (ms)")
    plt.ylabel(ylab)
    plt.grid(True)
    plt.legend()

    # 3D trajectory
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection="3d")
    ax.set_title("3D trajectory: Reference vs UART-decoded")
    ax.plot(x_r, y_r, z_r, label="Reference", linewidth=1.0)
    ax.plot(x_u, y_u, z_u, label="UART", linewidth=0.9, linestyle="--")
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("Z")
    ax.legend()

    # Summary text
    plt.figure(figsize=(11, 1.8))
    plt.axis("off")
    plt.text(0.01, 0.65, f"stop_ok_rate={stop_ok_rate*100:.2f}%
start_len={start_len_counts}", fontsize=11)
    plt.text(0.01, 0.25,
f"MAE(X,Y,Z)=({mae_x:.3f},{mae_y:.3f},{mae_z:.3f})   RMSE={rmse:.3f}
({ylab})", fontsize=11)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

B.1.2.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D  # registers 3D projection


def main():
    rtl_csv = "backend_out_3d.csv"
```

```python
pnr_csv = "backend_out_3d_pnr (2).csv"

try:
    df_rtl = pd.read_csv(rtl_csv)
except FileNotFoundError:
    print("could not open", rtl_csv)
    return

try:
    df_pnr = pd.read_csv(pnr_csv)
except FileNotFoundError:
    print("could not open", pnr_csv)
    return
if "phase" in df_rtl.columns:
    df_rtl = df_rtl[df_rtl["phase"] == 2].copy()
if "phase" in df_pnr.columns:
    df_pnr = df_pnr[df_pnr["phase"] == 2].copy()

if df_rtl.empty or df_pnr.empty:
    print("empty")
    return
df_merged = pd.merge(
    df_rtl,
    df_pnr,
    on="idx",
    suffixes=("_rtl", "_pnr")
)

if df_merged.empty:
    print("Merged RTL/PNR dataframe is empty. idx alignment failed.")
    return

# ms
if "time_ns_rtl" in df_merged.columns:
    t_ms = df_merged["time_ns_rtl"].values / 1.0e6
else:
    # Fallback
    t_ms = df_merged["time_ns"].values / 1.0e6
```

```
# AXES EXTRACTION
# Reference (ideal model)
if "x_ref_rtl" in df_merged.columns:
    x_ref = df_merged["x_ref_rtl"].values.astype(float)
    y_ref = df_merged["y_ref_rtl"].values.astype(float)
    z_ref = df_merged["z_ref_rtl"].values.astype(float)
else:
    x_ref = df_merged["x_ref"].values.astype(float)
    y_ref = df_merged["y_ref"].values.astype(float)
    z_ref = df_merged["z_ref"].values.astype(float)

# DUT RTL
x_rtl = df_merged["x_dut_rtl"].values.astype(float)
y_rtl = df_merged["y_dut_rtl"].values.astype(float)
z_rtl = df_merged["z_dut_rtl"].values.astype(float)

# DUT PNR
x_pnr = df_merged["x_dut_pnr"].values.astype(float)
y_pnr = df_merged["y_dut_pnr"].values.astype(float)
z_pnr = df_merged["z_dut_pnr"].values.astype(float)

# Computes error vs ideal (per axis and magnitude) for RTL and PNR
ex_rtl = x_rtl - x_ref
ey_rtl = y_rtl - y_ref
ez_rtl = z_rtl - z_ref
err_mag_rtl = np.sqrt(ex_rtl**2 + ey_rtl**2 + ez_rtl**2)

ex_pnr = x_pnr - x_ref
ey_pnr = y_pnr - y_ref
ez_pnr = z_pnr - z_ref
err_mag_pnr = np.sqrt(ex_pnr**2 + ey_pnr**2 + ez_pnr**2)

#  diff between RTL and PNR themselves
d_rtl_pnr = np.sqrt((x_pnr - x_rtl)**2 +
                    (y_pnr - y_rtl)**2 +
                    (z_pnr - z_rtl)**2)

# Figure 1: per-axis overlay and error magnitude
fig1, axes = plt.subplots(4, 1, figsize=(11, 12), sharex=True)
```

```
    ax_x = axes[0]
    ax_y = axes[1]
    ax_z = axes[2]
    ax_e = axes[3]

    # X
    ax_x.plot(t_ms, x_ref, label="X_ref (ideal)", linewidth=1.0)
    ax_x.plot(t_ms, x_rtl, label="X_RTL", linewidth=0.9, linestyle="--")
    ax_x.plot(t_ms, x_pnr, label="X_PNR", linewidth=0.9, linestyle=":")
    ax_x.set_ylabel("X (Q12)")
    ax_x.legend(loc="best")
    ax_x.grid(True)

    # Y
    ax_y.plot(t_ms, y_ref, label="Y_ref (ideal)", linewidth=1.0)
    ax_y.plot(t_ms, y_rtl, label="Y_RTL", linewidth=0.9, linestyle="--")
    ax_y.plot(t_ms, y_pnr, label="Y_PNR", linewidth=0.9, linestyle=":")
    ax_y.set_ylabel("Y (Q12)")
    ax_y.legend(loc="best")
    ax_y.grid(True)

    # Z
    ax_z.plot(t_ms, z_ref, label="Z_ref (ideal)", linewidth=1.0)
    ax_z.plot(t_ms, z_rtl, label="Z_RTL", linewidth=0.9, linestyle="--")
    ax_z.plot(t_ms, z_pnr, label="Z_PNR", linewidth=0.9, linestyle=":")
    ax_z.set_ylabel("Z (Q12)")
    ax_z.legend(loc="best")
    ax_z.grid(True)

    # Error magnitude
    ax_e.plot(t_ms, err_mag_rtl, label="|error| RTL vs ideal",
linewidth=1.0)
    ax_e.plot(t_ms, err_mag_pnr, label="|error| PNR vs ideal",
              linewidth=1.0, linestyle="--")
    ax_e.plot(t_ms, d_rtl_pnr, label="|RTL - PNR|", linewidth=0.8,
linestyle=":")
    ax_e.set_xlabel("Time (ms)")
    ax_e.set_ylabel("Error (Q12)")
    ax_e.legend(loc="best")
    ax_e.grid(True)

    fig1.suptitle("3D backend: ideal vs RTL vs post-PNR (per axis +
error)")
    fig1.tight_layout(rect=[0, 0.03, 1, 0.97])

    # Figure 2: 3D trajectories (ideal vs RTL vs PNR)
    fig2 = plt.figure(figsize=(9, 8))
    ax3d = fig2.add_subplot(111, projection="3d")

    # Reference as light gray path
    ax3d.plot(x_ref, y_ref, z_ref, color="0.7", linewidth=1.0,
              label="Reference (ideal)")

    # RTL trajectory
    ax3d.plot(x_rtl, y_rtl, z_rtl, color="red", linewidth=1.0,
              label="RTL backend")

    # PNR trajectory
    ax3d.plot(x_pnr, y_pnr, z_pnr, color="blue", linewidth=1.0,
linestyle="--",
              label="Post-PNR backend")

    ax3d.set_xlabel("X (Q12)")
    ax3d.set_ylabel("Y (Q12)")
    ax3d.set_zlabel("Z (Q12)")
    ax3d.set_title("3D trajectory: ideal vs RTL vs post-PNR")
    ax3d.legend(loc="best")

    plt.show()


if __name__ == "__main__":
    main()
```

B.1.3.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def main():
    csv_file = "backend_flicker.csv"
    df = pd.read_csv(csv_file)

    # only the flicker phase (phase == 2)
    df = df[df["phase"] == 2].copy()
    if df.empty:
        print("No phase==2 samples found in", csv_file)
        return

    # Time axis in ms
    t_ms = df["time_ns"].to_numpy(dtype=float) / 1.0e6

    # Raw columns from TB
    x_dut_q12    = df["x_dut"].to_numpy(dtype=float)         # Q12
    x_ref_q12    = df["x_ref"].to_numpy(dtype=float)         # Q12
(reference HP output)
    baseline_q12 = df["baseline_ref"].to_numpy(dtype=float) # Q12 EMA
baseline

    drift_field  = df["drift_ref"].to_numpy(dtype=float)     # "field
units"
    signal_field = df["signal_ref"].to_numpy(dtype=float)    # "field
units"
    input_field  = drift_field + signal_field               # what went
into the ADC on X

    # Converts Q12 quantities back to field units.
    # TB used:
    #   CODE_POS1 = 612, CODE_ZERO = 512 -> ADC_SCALE = 100 codes / unit
    #   FRAC = 12 -> Q_SCALE = 2^12 = 4096
    # So Q12_value = field * (ADC_SCALE * Q_SCALE)  => field = Q12 /
(ADC_SCALE*Q_SCALE)
    ADC_SCALE   = 100.0
    FRAC        = 12
    Q_PER_FIELD = ADC_SCALE * (1 << FRAC)  # 409600

    x_dut_field    = x_dut_q12    / Q_PER_FIELD
    x_ref_field    = x_ref_q12    / Q_PER_FIELD
    baseline_field = baseline_q12 / Q_PER_FIELD

    # 1) Drift vs EMA baseline: Does the baseline follow the slow drift?
    plt.figure(figsize=(10, 4))
    plt.title("Flicker: injected drift vs EMA baseline (reference)")
    plt.plot(t_ms, drift_field,    label="Injected drift (slow random
walk)")
    plt.plot(t_ms, baseline_field, label="EMA baseline
(axis_baseline[0])")
    plt.xlabel("Time (ms)")
    plt.ylabel("Field units")
    plt.grid(True)
    plt.legend()

    # 2) Total input vs high-pass output: Does drift disappear?
    plt.figure(figsize=(10, 4))
    plt.title("Total input (drift + signal) vs high-pass output")
    plt.plot(t_ms, input_field,  label="Input = drift + signal",
alpha=0.7)
    plt.plot(t_ms, x_ref_field,  label="Reference HP output",
linewidth=1.0)
    plt.xlabel("Time (ms)")
    plt.ylabel("Field units")
    plt.grid(True)
    plt.legend()

    # 3) AC signal vs outputs + error
    # a) Signal vs HP outputs in field units
    plt.figure(figsize=(10, 4))
    plt.title("AC signal vs high-pass outputs (drift-suppressed)")
    plt.plot(t_ms, signal_field,  label="Injected AC signal",
linewidth=1.0)
    plt.plot(t_ms, x_ref_field,   label="Reference HP output",
linewidth=0.8, linestyle="--")
```

```
    plt.plot(t_ms, x_dut_field,   label="DUT HP output",
linewidth=0.8, linestyle=":")
    plt.xlabel("Time (ms)")
    plt.ylabel("Field units")
    plt.grid(True)
    plt.legend()

    # b) Error between DUT and reference (Q12 and field units)
    err_q12   = x_dut_q12 - x_ref_q12
    err_field = x_dut_field - x_ref_field

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 6), sharex=True)
    fig.suptitle("DUT vs reference error")

    ax1.plot(t_ms, err_q12, linewidth=0.8)
    ax1.set_ylabel("Error (Q12)")
    ax1.grid(True)

    ax2.plot(t_ms, err_field, linewidth=0.8)
    ax2.set_xlabel("Time (ms)")
    ax2.set_ylabel("Error (field units)")
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```