

AI-Assisted Automation for Advanced Packaging

Using LLM-Generated Scripts to Streamline UCle Routing and Design Revisions

Table of Contents

Executive Summary3

Background and Motivation4

The Automation Journey: Designing UCle Advanced Package Layout..... 7

From Prompt to Advanced Package Layout 10

Key Takeaways..... 23

Broader Applications 24

Conclusion25

Executive Summary

This application note demonstrates how a Large Language Model (LLM) can be combined with ADS Python automation to design a complex UCle advanced package layout. The case study illustrates how design requirements, physical constraints, implementation rules, and ADS Python API references can be provided to an LLM and iteratively refined to generate executable, parameterized automation scripts.

Using a conventional manual design approach, the initial layout implementation would typically require more than a week of engineering effort, with each subsequent revision taking approximately three additional days. By contrast, the AI-assisted methodology reduced the initial implementation time to about one day, while later design revisions could be completed in roughly 30 minutes by regenerating the layout with updated design parameters.

Beyond the immediate productivity gains, the generated scripts captured a proven implementation methodology that could be reused and adapted for future package configurations. Instead of recreating layouts from scratch for every new project, engineers can leverage and extend existing automation, significantly improving efficiency and consistency across designs.

The same approach can be applied to boost productivity in other ADS workflows, including schematic generation, design verification and evaluation, particularly when tasks involve repetitive operations, complex engineering rules, or frequent design iterations. As reusable automation continues to expand across these areas, it can also serve as a practical foundation for future agent-driven ADS workflows.

Background and Motivation

Project Context

I was working on a UCIe advanced package layout project in ADS that involved extensive die-to-die routing under a demanding schedule. Building the layout required creating and maintaining a large amount of geometry while meeting numerous design rules, connectivity requirements, and project-specific constraints.

What made the effort particularly time consuming was the number of repetitive tasks throughout the design. Many structures had to be built using the same design logic, but the work still had to be performed manually. These structures also required precise placement and execution to satisfy design constraints and avoid unintended performance degradation. As the design grew in complexity, so did the amount of manual effort required to design, verify, and maintain the layout.

Based on previous project records, designing the initial layout typically required more than a week of effort. However, the initial implementation was only part of the challenge. Many of the layout structures were highly inter-dependent. A change to a single design parameter could impact multiple areas of the layout, requiring manual updates and revalidation to keep everything consistent. As revisions accumulated, the amount of rework continued to grow, often extending the overall development effort beyond a month.

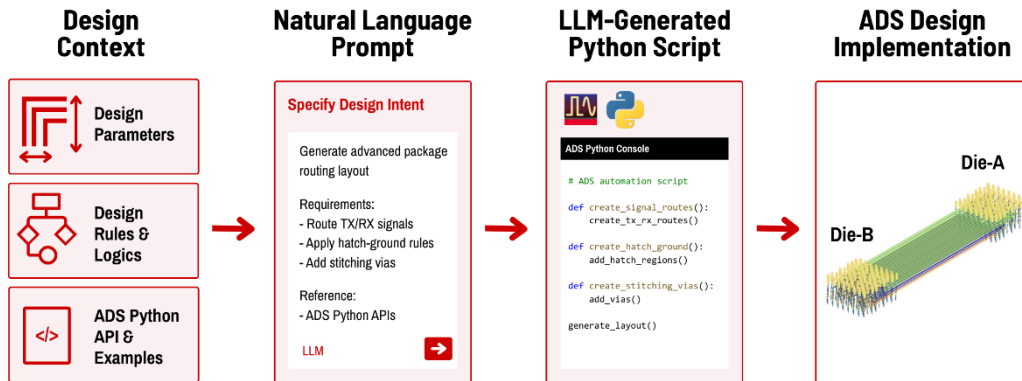
Exploring a Different Approach

Faced with the growing development effort and revision overhead, I began looking for alternatives to the traditional manual workflow. Automation seemed like a natural direction, and readers interested in learning more about Python-based automation in ADS can refer to the [ADS Python Automation for Developers](#) and [ADS Python Tutorials - YouTube](#) demonstration from Keysight. However, there was a problem. Capturing numerous design rules, connectivity constraints, and project-specific logic in automation would require significant scripting effort. The automation itself could easily become another major project, at least a year or two earlier.

By now, Large Language Models (LLMs) have matured to the point where they could generate functional scripts from a clear description of engineering requirements. I decided to try an LLM for much of the Python script-generation process. For each design task, I provided the LLM with the engineering context needed to create the automation, including:

- Design parameters
- Design rules and logic
- ADS Python APIs
- Reference design script

LLM-Assisted ADS Automation



Key Benefits

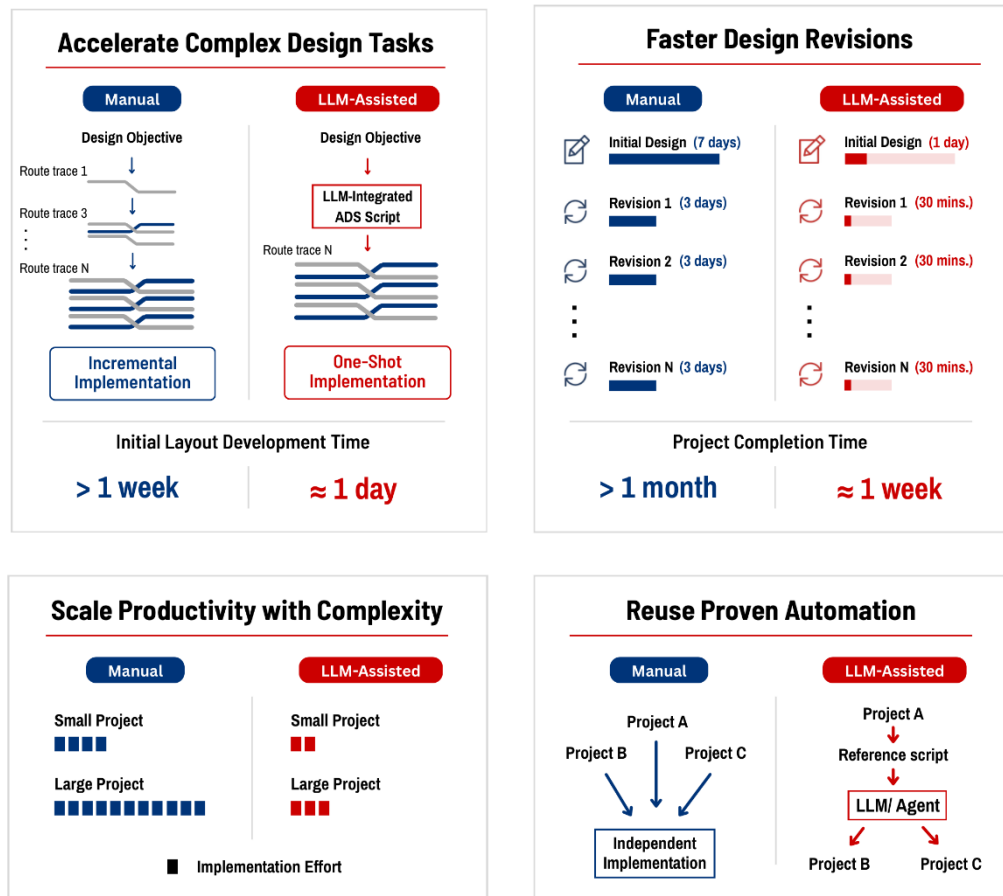


Figure 1. LLM-Assisted ADS Automation: Workflow and Benefits.

Using this information, the LLM generated parameterized ADS Python scripts that could be executed directly within ADS (Figure 1), automatically generating the targeted layout structures in a single execution step.

When the targeted structure did not produce the desired result, or when additional design rules and logic needed to be incorporated, I updated the information provided to the LLM and generated a revised automation script. Within minutes, I could evaluate a new version and continue refining the design logic. This rapid build-and-verify cycle made it possible to develop the initial automated layout flow in approximately one day.

As the project evolved, one of the things I appreciated most was how well the approach scaled with design complexity. With the automation in place, I could generate significantly more complex layout designs without a comparable increase in development time.

The benefits became particularly noticeable during design revisions. Using this approach, a design revision could be completed in about 30 minutes, compared to approximately three days using the previous manual process. As a result, the entire project was completed within a week, whereas similar development efforts had previously extended beyond a month.

The effort invested in developing the automation continued to provide value beyond the original project. When I later worked on a different package configuration, I was able to use the reference scripts and design knowledge from the original project as a foundation for the LLM, enabling the new project to be completed within only a few hours.

This application note shares the approach used to apply AI-assisted, prompt-driven automation to a chiplet advanced-package layout in ADS. The following sections describe the design challenges, the engineering logic identified across different layout stages, and how that information was translated into LLM prompts to generate reusable ADS automation. Through this case study, I demonstrate how AI-assisted automation can transform complex, time-consuming manual tasks into fast, repeatable workflows.

The Automation Journey: Designing UCle Advanced Package Layout

Design Overview

In this case study, I share how I automated the design of an advanced package layout in ADS using an LLM. In chiplet advanced packaging, multiple dies are integrated on a silicon interposer/bridge and connected through fine-pitch micro-bumps with dense routing to support die-to-die communication. One of the key characteristics of advanced package layouts is the extensive use of hatch-ground planes in place of solid ground planes, a choice often driven by process and manufacturing requirements. Hatch-ground planes introduce additional layout complexity. Signal traces must remain carefully aligned with the surrounding hatch-ground pattern to avoid signal reflections and impedance discontinuities. As a result, maintaining these constraints often requires significant manual effort, adding considerably to overall development time.

The design used in this project implements a UCle die-to-die interface between two chiplets. UCle defines the communication interface between the dies, while the advanced package layout provides the physical interconnects needed to realize that connectivity. Figure 3A shows the UCle bump-map configuration used in this project, consisting of 16 columns and 32 Tx/Rx data lanes. The bump map defines the signal, power, and ground bump locations on each die. These bump locations are connected through routing within the silicon interposer/bridge, establishing the electrical connections needed for die-to-die communication. Figure 3B shows the package stack-up used throughout the design process.

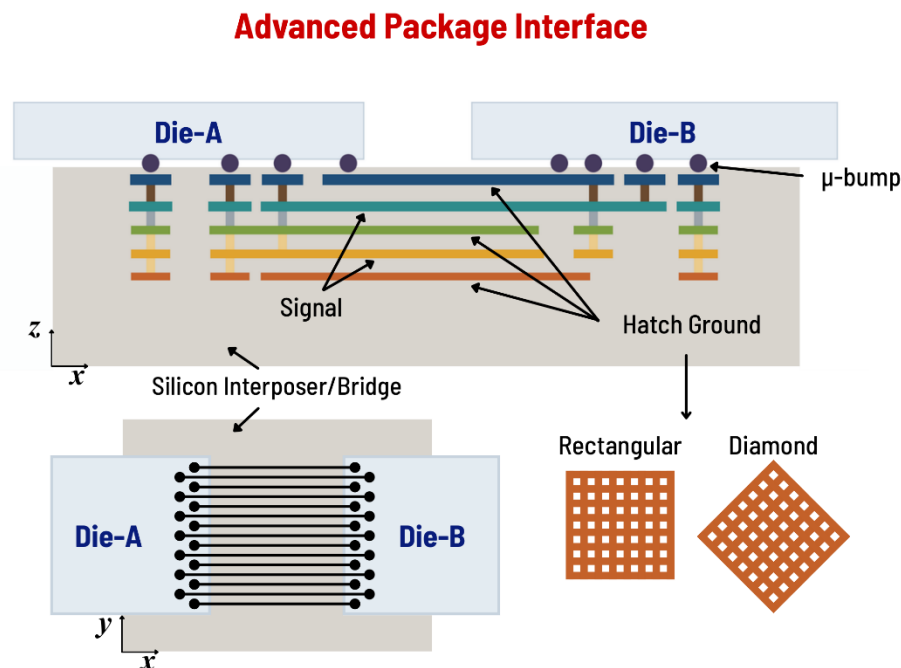


Figure 2. Advanced-package interface showing signal and hatch-ground layer stack-up.

A

UCIe 16-column x32 Advanced Package Module

32 Tx/Rx data lanes with sideband, clock, power, and ground bumps

Col. 0	Col. 1	Col. 2	Col. 3	Col. 4	Col. 5	Col. 6	Col. 7	Col. 8	Col. 9	Col. 10	Col. 11	Col. 12	Col. 13	Col. 14	Col. 15
0	vss	0	vss	0	vss	0	vss	0	vss	0	vss	0	vss	0	vss
vccio	0	vccio	0	vccio	0	vccio	0	vccio	0	vccio	0	vccio	0	vccio	0
0	txdatasb	0	rxcksbR	0	txdatasb	0	rxcksb	0	txcksb	0	rxdatasb	0	txcksbR	0	rxdatasb
vss	0	txdata5	0	vss	0	vss	0	vss	0	vss	0	vss	0	vss	0
0	txdata4	0	txdata20	0	txdata22	0	txtrk	0	rxdata30	0	rxdata26	0	rxdata13	0	rxdataRD
txdata2	0	vss	0	txdata21	0	txckp	0	rxdata31	0	rxdata27	0	rxdata14	0	rxdata11	0
0	txdata3	0	txdata17	0	txdata23	0	vss	0	rxdata29	0	vss	0	rxdata12	0	vss
txdata1	0	txdata6	0	txdata19	0	txckn	0	vss	0	rxdata28	0	rxdata15	0	rxdata10	0
0	txdata8	0	vss	0	vss	0	txvld	0	rxdataRD	0	rxdata25	0	vss	0	rxdata0
vss	0	txdata7	0	txdata18	0	txckRD	0	rxvldRD	0	rxdata24	0	rxdata16	0	rxdata9	0
0	txdata9	0	txdata16	0	txdata24	0	txvldRD	0	rxckRD	0	rxdata18	0	rxdata7	0	vss
txdata0	0	vss	0	txdata25	0	txdataRD	0	rxvld	0	vss	0	rxdata17	0	rxdata8	0
0	txdata10	0	txdata15	0	txdata28	0	vss	0	rxckn	0	rxdata19	0	rxdata6	0	rxdata1
vss	0	txdata12	0	vss	0	txdata29	0	vss	0	rxdata23	0	vss	0	rxdata3	0
0	txdata11	0	txdata14	0	txdata27	0	txdata31	0	rxckp	0	rxdata21	0	rxdata5	0	rxdata2
txdataRD	0	txdata13	0	txdata26	0	txdata30	0	rxtrk	0	rxdata22	0	rxdata20	0	rxdata4	0
0	vccio	0	vccio	0	vccio	0	vccio	0	vccfwdio	0	vccfwdio	0	vccfwdio	0	vccfwdio
vccio	0	vccio	0	vccio	0	vccio	0	vccfwdio	0	vccfwdio	0	vccfwdio	0	vccfwdio	0

Die Edge

B

Stackup Information

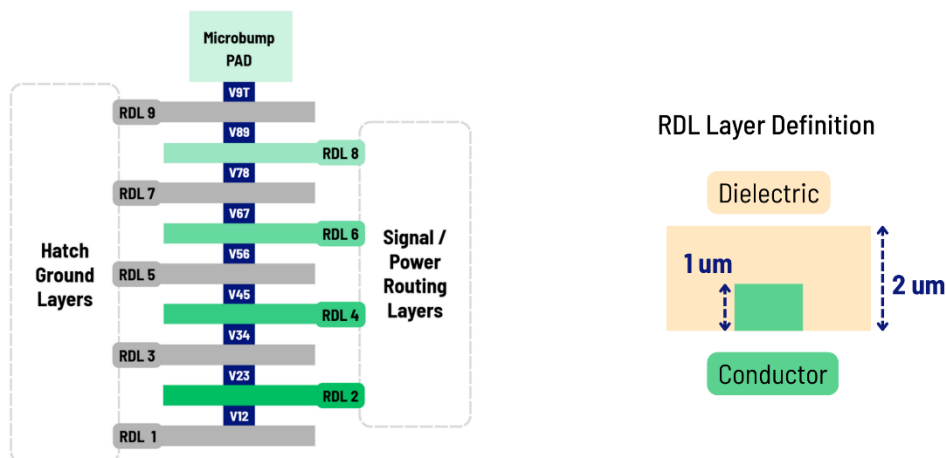


Figure 3. UCIe bump-map configuration (A) and package stack-up (B) used in the design project.

Structuring the Design

To make the design process more manageable, I divided the layout into five major stages based on the key structures required throughout the package. Each stage introduced its own design requirements, design logic, and automation challenges. Breaking the design into smaller building blocks also made it easier to develop, validate, and refine the automation throughout the project.

The five stages are summarized below:

1. Vertical Transitions at Bump Locations

Vertical transitions connect the die bumps to the routing layers, following the UCle 16-column, 32-data-lane bump-map configuration.

2. Hatched-Ground Structures

Hatched-ground structures form the ground reference across multiple routing layers, maintaining connectivity with ground bump transitions while providing clearance around signal and power interconnects.

3. Tx/Rx Signal Routing

Signal-routing establishes the die-to-die connections between corresponding transmitter (Tx) and receiver (Rx) locations defined by the UCle interface.

4. Coplanar Guard Grounds

Coplanar guard-ground structures are created alongside signal routes to provide local ground reference, helping maintain signal isolation and reduce crosstalk.

5. Stitching Vias

Stitching vias connect ground structures across different routing layers, creating a continuous ground reference throughout the package.

Design Parameters

Table 1 summarizes the representative design parameters used throughout the layout design.

Table 1 - Representative Design Parameters and values

Parameters	Value
Bump pitch	60 μm
Die center-to-center spacing	2000 μm
Hatch-ground conductor width	2.5 μm
Hatch-ground row spacing	2.5 μm
Hatch-ground column spacing	1.8 μm
Signal trace width	2 μm
Stitching via size (Length x Width)	4 μm x 1.5 μm
Stitching-via interval	50 μm

From Prompt to Advanced Package Layout

For clarity, the two dies in this design are referred to as die-A and die-B throughout this paper. The following sections demonstrate how each stage of the advanced package layout design was generated using AI-assisted automation. For every stage, I describe the design challenge, the engineering context provided to the LLM, and the instructions used to guide script generation. The resulting ADS automation was then refined and validated to produce the required layout structures.

Stage 1: Create Vertical Transitions from Bump Map Definitions

Design Complexity

The UCle bump map defines 144 bump locations per die, including signal, power, and ground connections, that must be translated into physical transition geometries. Each generated transition must be placed at the correct bump-map coordinate while preserving the intended signal connectivity. In addition, die-B is rotated 180° relative to die-A, requiring the bump locations and signal directions to remain correctly aligned between the two dies. Signal bumps also require directional net names to distinguish A-to-B and B-to-A data transfers, while power and ground bumps must use shared net names. Maintaining these relationships consistently across both dies can become a significant manual task and increase the risk of placement and net-assignment errors. Figure 4 illustrates the vertical transition generation flow, including top-die generation, bottom-die orientation, and net-naming rules.

Inputs to the LLM

- UCle bump map represented as a matrix-based dataset
- Bump pitch and die-to-die spacing
- Die rotation rule
- Net-naming rules
- ADS Python export of a reference transition geometry

Stage 1: Vertical Transition Generation

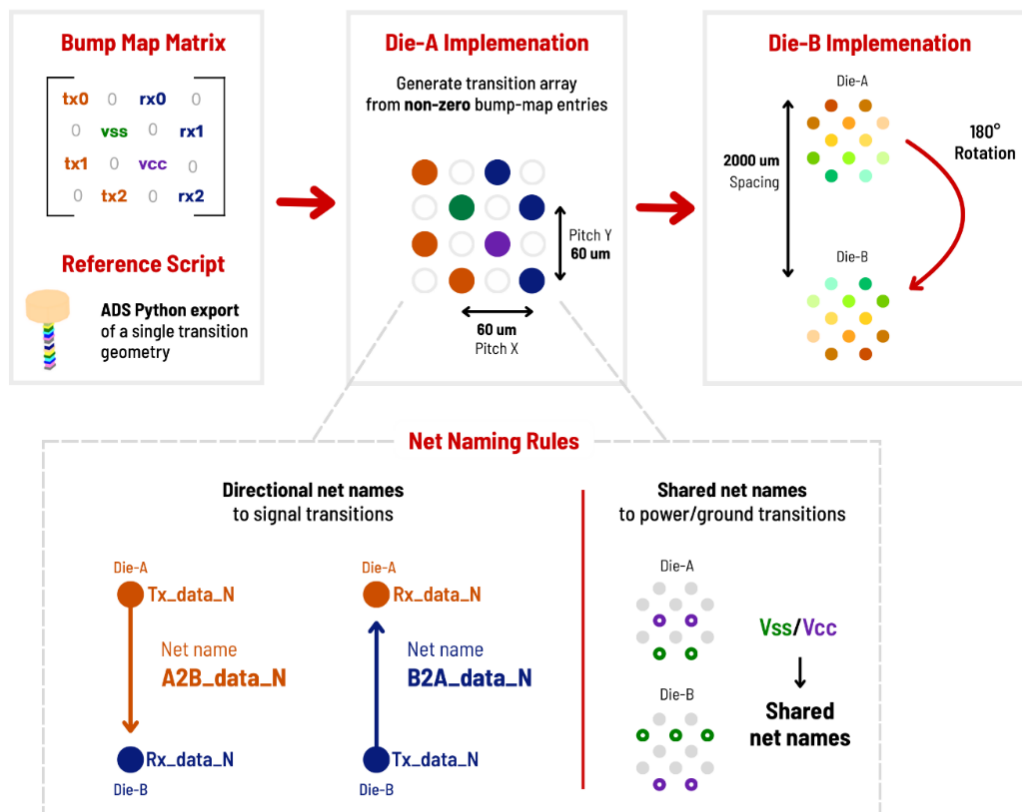


Figure 4. Vertical transition generation for die-A and die-B, including die rotation and net-naming rules.

```

# UCIE Bump Map Matrix
# -----
ucie_bump_map_matrix = [
["0", "vss", "0", "vss", "0", "vss", "0", "vss", "0", ...],
["vccio", "0", "vccio", "0", "vccio", "0", "vccio", "0", ...],
["0", "txdatasbRD", "0", "rxcksbRD", "0", "txdatasb", ...],
["vss", "0", "txdata5", "0", "vss", "0", ...],
...]

```

Note: "0" entries represent inactive bump locations and preserve the spacing and geometry of the diamond-pattern bump map.

```

# ADS-Python Reference Vertical Transition
# -----
# Create signal net
signal_net = "A2B_data0"

# Create top microbump pad
design.add_circle(
    layer = Microbump_Layer,
    center = (bump_x, bump_y),
    radius = 10um,
    net = signal_net)

# Create RDL routing structures
for layer in RDL_Layers:
    design.add_rectangle(
        layer = RDL_1_or_9,
        width = 3.0um,
        length = 5.5um,
        center = (bump_x, bump_y),
        net = signal_net)

# Create transition-via structures
for layer in Transition_Via_Layers:
    design.add_rectangle(
        layer = v_nm,
        width = 2.5um,
        length = 4.5um,
        center = (bump_x, bump_y),
        net = signal_net)

```

Note: This ADS-Python script is the exported reference for a single vertical transition, manually created in ADS layout. It defines the μ -bump pad, RDL routing, and transition-via geometries, providing the LLM with a concrete reference to generate the full transition array across the bump map.

Actual Prompt Provided to the LLM

1. Assign the bump map matrices of die-A and die-B using the following rules:
 - a. Use the provided matrix-based dataset to define die-A's bump location
 - b. Assign die-B's bump location by rotating die-A's matrix 180°
2. Generate the transition geometry at each non-zero matrix location (skip zero terms). Position the geometries according to the matrix grid, which uses 30 µm cell-to-cell spacing and yields a 60 µm bump pitch in both X and Y. Place die-B 2000 µm below die-A along Y, with no X offset.
3. Set net names of the transitions using the following rules:
 - a. vss, vccio, and vccfwdio each carry identical net names across both dies.
 - b. Die-A's Tx net that connects to die-B's corresponding Rx net is named A2B_<signal_name> (e.g., Txdata0_DieA → Rxdata0_DieB = A2B_data0).
 - c. Die-B's Tx net that connects to die-A's corresponding Rx net is named B2A_<signal_name> (e.g., Txdata0_DieB → Rxdata0_DieA = B2A_data0).

LLM-Generated Automation Script (Pseudocode)

```
# Design inputs
bump_map = ucie_bump_map_matrix
bump_pitch = 60 um
d2d_spacing = 2000 um

# Die-B is rotated 180° relative to die-A
die_b_map = reverse_rows_and_columns(bump_map)

# Apply net-naming rules
def net_name_rule(bump_name, die_name):
    # Shared power and ground nets
    if bump_name == "vss": return bump_name
    if bump_name.startswith("vcc"): return bump_name

    # Directional signal nets
    if die_name == "Die-A":
        if bump_name.startswith("tx"): return "A2B_" + signal_name
        if bump_name.startswith("rx"): return "B2A_" + signal_name
```

```

        if die_name == "Die-B":
            if bump_name.startswith("tx"): return "B2A_" + signal_name
            if bump_name.startswith("rx"): return "A2B_" + signal_name

# Generate vertical transitions for a die
def generate_die(die_map, die_name, y_offset):
    for row, column, bump_name in die_map:
        # Skip inactive locations
        if bump_name == "0": continue

        location = column * bump_pitch, y_offset - row * bump_pitch
        create_io_transition(
            location = location,
            net_name = net_name_rule (bump_name, die_name)
        )

# Generate die-A
generate_die(
    die_map = bump_map,
    die_name = "Die-A",
    y_offset = 0
)

# Generate die-B
generate_die(
    die_map = die_b_map,
    die_name = "Die-B",
    y_offset = d2d_spacing
)

```

Automated Execution

- Generate all vertical transitions for Die A and Die B directly from the UCle bump map
- Automatically apply die orientation, bump placement, and spacing rules
- Generate directional A2B and B2A signal nets while maintaining shared power and ground nets

Stage 2: Net-Aware Hatch-Ground Generation

Design Complexity

The hatch-ground spans multiple layers, surrounding all transition locations throughout the package. At each vertical transition, ground connections must remain electrically connected to the reference ground, while signal and power connections must stay isolated from it. Applying this behavior consistently across 288 bump locations across both dies required significant manual effort and frequently led to connectivity errors. Figure 5 shows the hatch-ground construction across the selected routing layers, including the connectivity conditions applied at ground, signal, and power transitions.

Inputs to the LLM

- Hatch width and spacings
- Conductor layers used for hatch-ground generation
- Net-aware ground connectivity rules
- Reference ADS python script

Automated Execution

- Generate multi-layer hatch-grounds across the specified routing layers
- Maintain hatch-ground connectivity at ground transitions
- Create clearance regions around signal and power transitions to prevent unintended grounding

Stage 2: Net-Aware Hatch Ground Generation

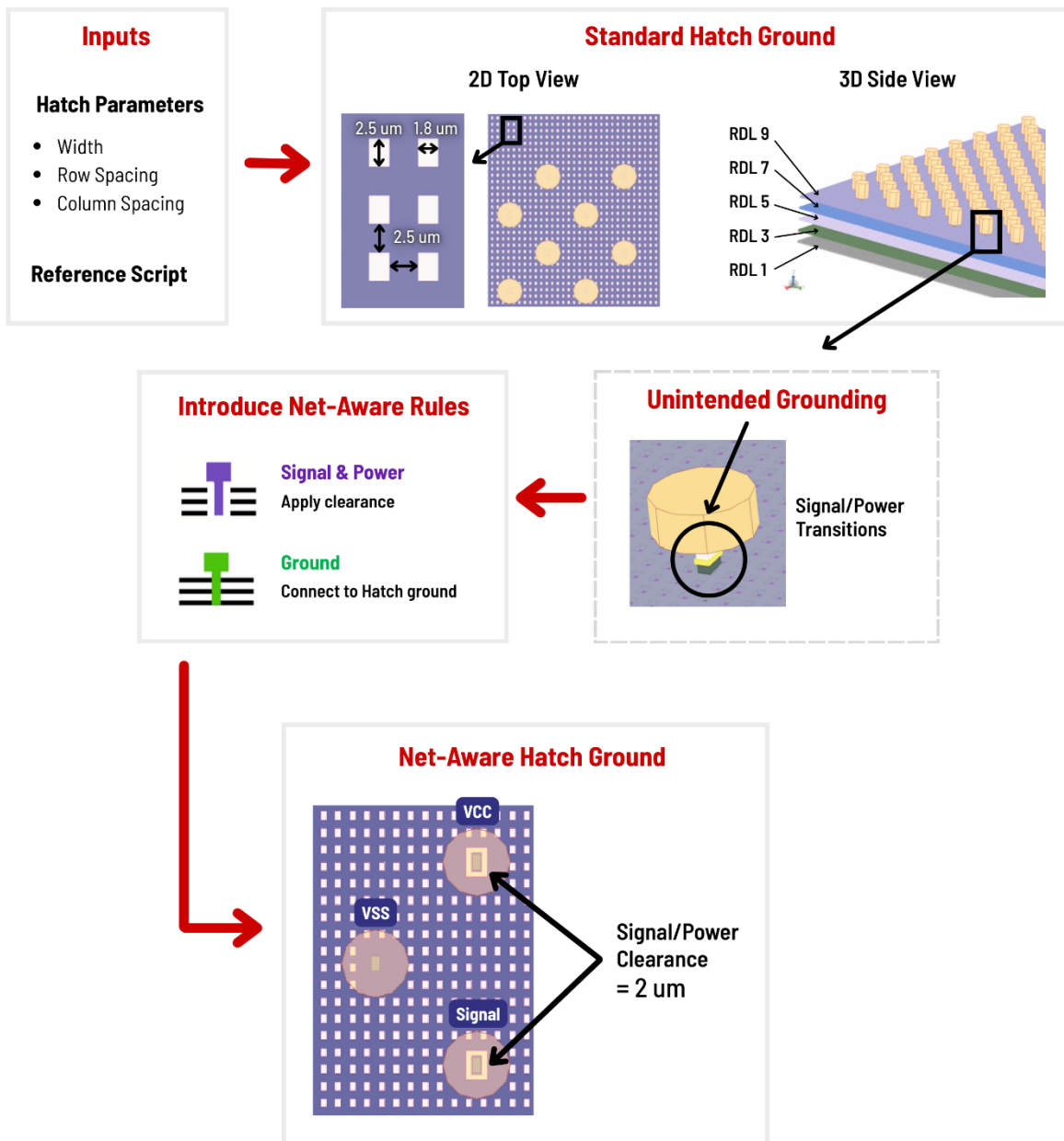


Figure 5. Net-aware hatch-ground generation with clearance applied at signal and power transitions.

Stage 3: Die-to-Die Signal Routing

Design Complexity

The 16-column × 32-lane UCle bump map introduces a high-density routing requirement between the die-A and die-B. Multiple signal connections must be routed through each bump-map column region and share the limited routing space available between adjacent bump columns. Once signal-width, spacing, and ground-reference requirements are considered, only a small number of routes can be accommodated within each routing corridor. In addition to these physical constraints, the signal lanes must satisfy signal-integrity requirements through consistent routing topology and ground-reference conditions across the data bus. Ensuring that all die-to-die connections satisfy these requirements can become a complex and repetitive design task.

Strategy

The routing strategy for the current design is based on several key routing considerations (see Figure 6A). Corresponding Tx and Rx signals are aligned by the die orientation. Within the 16-column bump-map organization, up to six signal nets may require routing through the same column region. However, the available routing corridor between adjacent bump columns can accommodate only two signal routes. To accommodate all required connections, I distributed the signals across three routing layers using predefined layer-allocation rules and a symmetric left/right signal-assignment strategy.

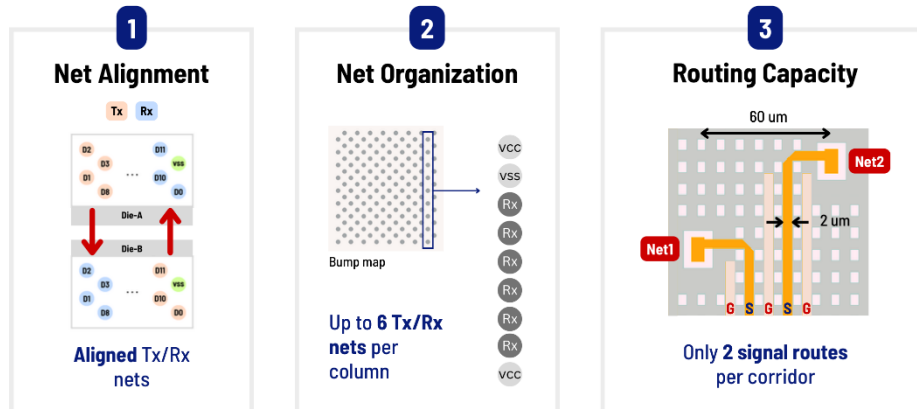
To maintain a consistent topology, each route breaks out of its starting transition horizontally, transitions to vertical routing between the dies, and then returns to horizontal routing before reaching the destination transition (see Figure 6B). During routing, the signal traces maintain alignment with the surrounding hatch-ground pattern and change direction only at designated routing turn points. This approach ensures that all signal lanes follow a consistent routing methodology across the data bus.

Inputs to the LLM

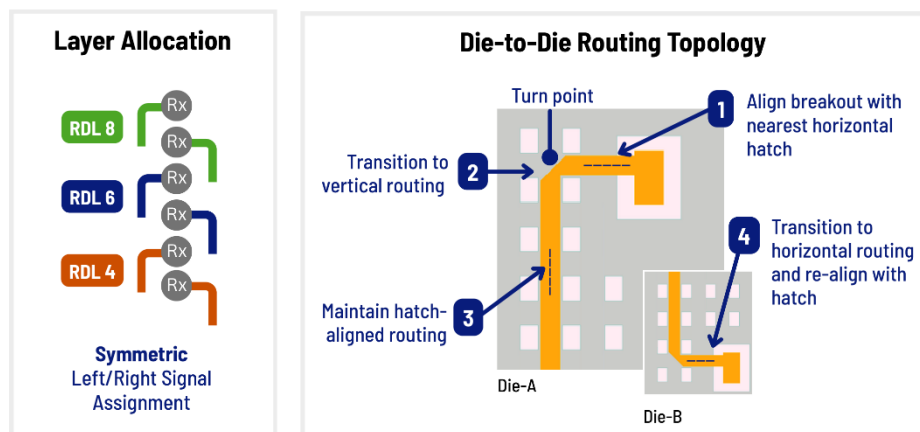
- Trace width
- Hatch-ground locations
- Bump transition locations
- Routing layer allocation rules
- Die-to-die routing topology

Stage 3: Die-to-Die Signal Routing

A Routing Considerations



B Routing Strategy



C Automated Routing Implementation

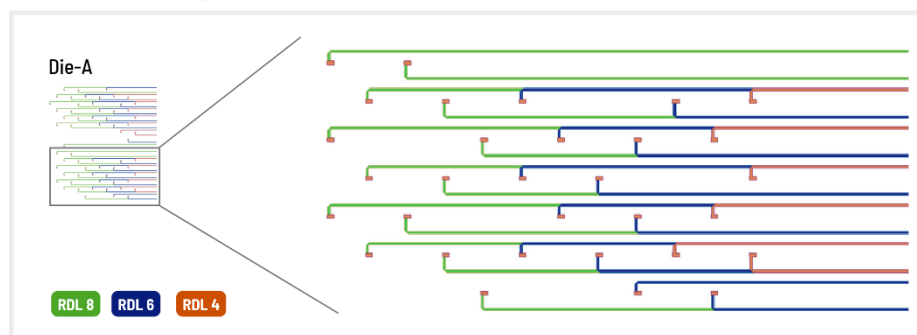


Figure 6. Die-to-die signal routing considerations, strategy, and Automated Execution.

Automated Execution

- Assign signal nets to routing layers according to the layer-allocation rules
- Apply symmetric left/right signal assignment within each bump-map column
- Generate hatch-aligned die-to-die routing between corresponding bump transitions

Stage 4: Coplanar Guard-Ground Generation

Design Complexity

The routing approach used in this project included coplanar guard-ground traces adjacent to each signal route. Because the guard-grounds were derived directly from the signal-routing paths generated in the previous stage, maintaining the required spacing relationships and alignment with the surrounding ground reference across all signal lanes became a significant design task when performed manually.

Strategy

Figure 7A illustrates the guard-ground routing approach used in this design. Two guard-ground traces were generated alongside each signal route, maintaining the required spacing relationship while following the signal-routing topology. The guard grounds also remained aligned with the surrounding hatch-ground and terminated at adjacent hatch rows near the signal endpoints.

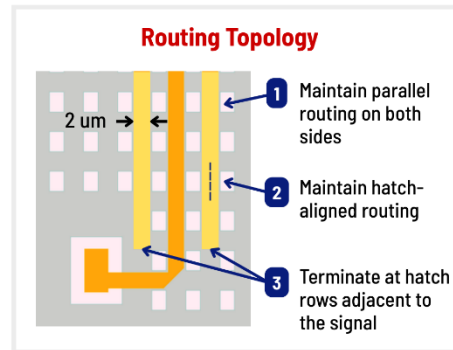
Inputs to the LLM

- Signal routing paths
- Hatch-ground locations
- Guard ground dimensions and spacing
- Guard ground routing topology

Automated Execution

- Generate coplanar guard-ground conductors alongside each routed signal

A Stage 4: Coplanar Guard Ground Generation



B Stage 5: Guard-Ground Stitching Via Generation

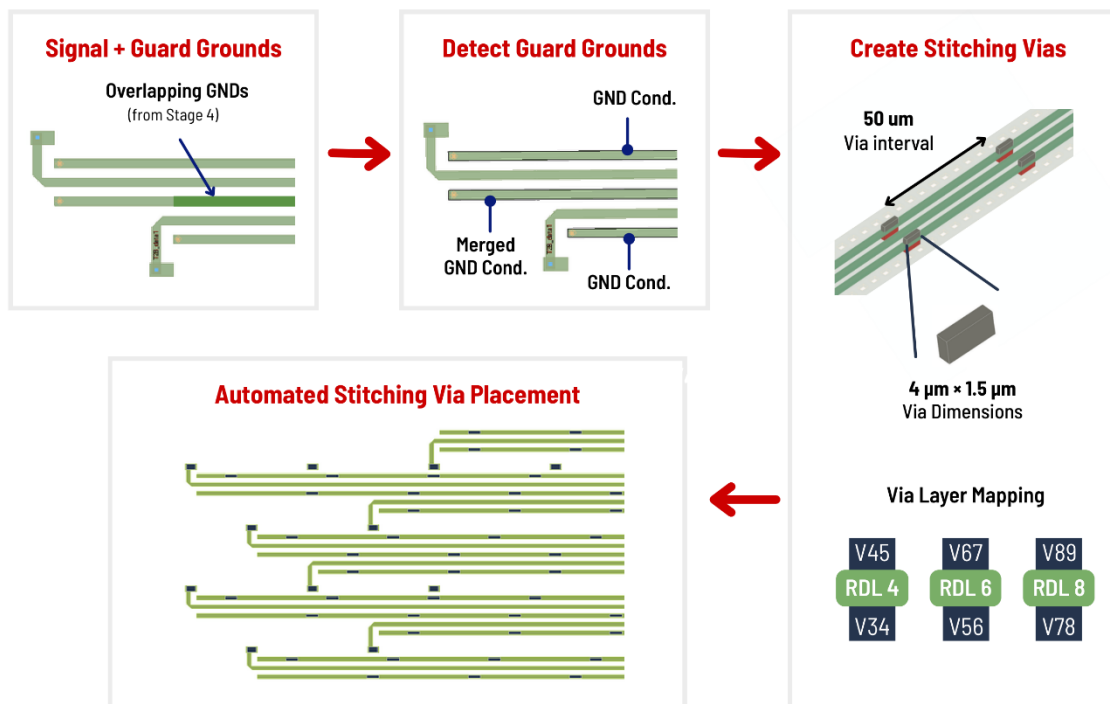


Figure 7. Coplanar guard-ground (A) and stitching via (B) generation.

Stage 5: Guard-Ground Stitching Via Generation

Design Complexity

Coplanar guard-grounds must be connected to neighboring ground-reference layers throughout the package. Because the guard grounds are generated as individual routing structures, overlapping and adjacent segments must first be consolidated into continuous ground regions before stitching vias can be added. Stitching vias must then be placed across multiple routing layers while satisfying layer-specific connection and spacing requirements. Applying these rules consistently throughout the layout can become a repetitive design task, particularly when design changes require widespread updates.

Strategy

Figure 7B illustrates the stitching via creation process. Guard-ground traces generated in the previous stage are first identified. If overlapping guard-ground regions exist because of the previous automation stage, they are merged into a single continuous ground structure. Stitching vias are then placed along the merged guard-ground structures according to predefined dimensions and spacing intervals. The generated vias follow layer-mapping rules to connect the guard grounds to neighboring ground-reference layers.

Inputs to the LLM

- Guard-ground detection and merging rules
- Stitching via dimensions and spacing requirements
- Stitching via layer-mapping rules

Automated Execution

- Detect guard-ground conductors
- Merge overlapping guard-ground regions
- Generate stitching vias according to the layer mapping rules

Together, these five automation stages transform a UCle bump-map definition into a complete package design. Starting from the bump-map description, the workflow automatically generates the required bump transitions, hatch-ground geometry, die-to-die signal routing, coplanar guard grounds, and stitching via connections. Figure 8 shows the resulting UCle die-to-die routing generated through LLM-assisted automation. The top panel shows the complete signal routing connecting die-A and die-B, while the bottom panel shows the resulting multi-layer ground across the package.

Automated UCle Die-to-Die Routing with LLM

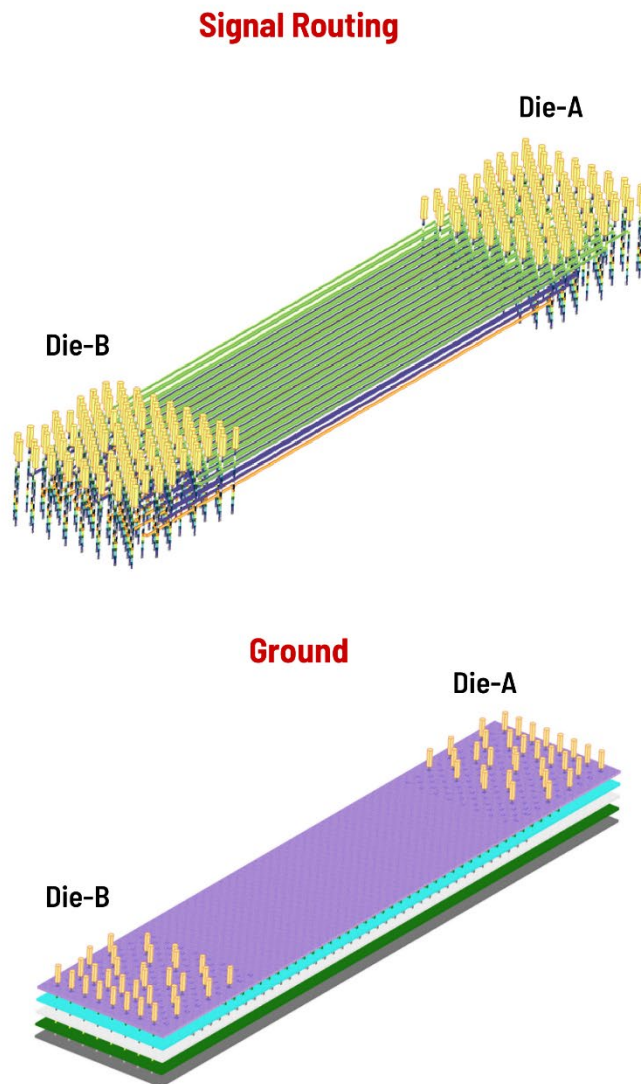


Figure 8. Automated UCle Die-to-Die signal and ground routing.

Key Takeaways

Accelerating Complex Routine Tasks

By using an LLM to generate automation scripts, complex, recurring design tasks that would otherwise take over a week to design manually can be completed in about a day.

Speeding Up Design Revisions

By capturing the design methodology in automation, design updates can be addressed through regeneration rather than repeated manual reconstruction, cutting revision time from days to minutes.

Reusing Proven Methodologies Across Future Projects

Once developed, the same automation framework can be adapted to new projects with similar requirements, allowing proven design methodologies to be reused rather than recreated.

Broader Applications

Although this paper focuses on advanced package layout design, similar opportunities exist across many engineering activities that rely on complex rules, constraints, and project-specific methodologies.

Design Execution

Generate schematics, connectivity structures, and other project-specific design content that requires large numbers of repetitive design decisions.

Design Validation

Use ADS Python automation to extract design information and evaluate executed design against intended design logic, helping identify errors and inconsistencies that would otherwise require extensive manual review.

Design Evaluation

Automate repetitive evaluation activities such as data processing, result analysis, and multi-step validation procedures while maintaining visibility and control over the evaluation process.

Cross-Tool Engineering Workflows

Automate engineering tasks that span multiple tools, data sources, and design environments, particularly workflows that require complex rule-based decisions and coordination between different stages of the design process.

Agent-Driven Engineering Workflows

The automation generated through this approach can serve as a foundation for agent-driven workflows, where AI systems use reusable engineering knowledge and automation assets to perform increasingly complex engineering tasks from high-level requirements.

Conclusion

My journey began with a challenging advanced-package layout implementation that required significant manual effort and repeated rework as the design evolved. Using AI-assisted automation, which combines ADS Python automation with LLM-generated implementation scripts, I was able to quickly generate the complete package layout from design requirements, constraints, and engineering rules.

The initial automation took about a day to develop, and I later reused it to implement a different package configuration in just a few hours. More importantly, the implementation methodology was captured in reusable automation rather than repeated manually with each design iteration.

Although I demonstrated this approach using a UCle advanced-package example, the same idea can apply to many engineering activities involving repetitive tasks, complex rules, and project-specific methodologies. As these automation assets grow, they can form the foundation for more efficient, connected, and increasingly intelligent engineering workflows, ultimately enabling agent-driven engineering workflows that can execute complex engineering tasks from high-level design objectives.

For more information or to book a free trial visit: [Advanced Design System \(ADS\) | Keysight](#)



As a mission-critical design enablement partner, Keysight connects design, emulation, and test so customers can solve the world's most complex engineering challenges. Accelerate innovation at www.keysight.com.

This information is subject to change without notice. © Keysight Technologies, 2026, Published in USA, October 23, 2026, 3126-1800.EN