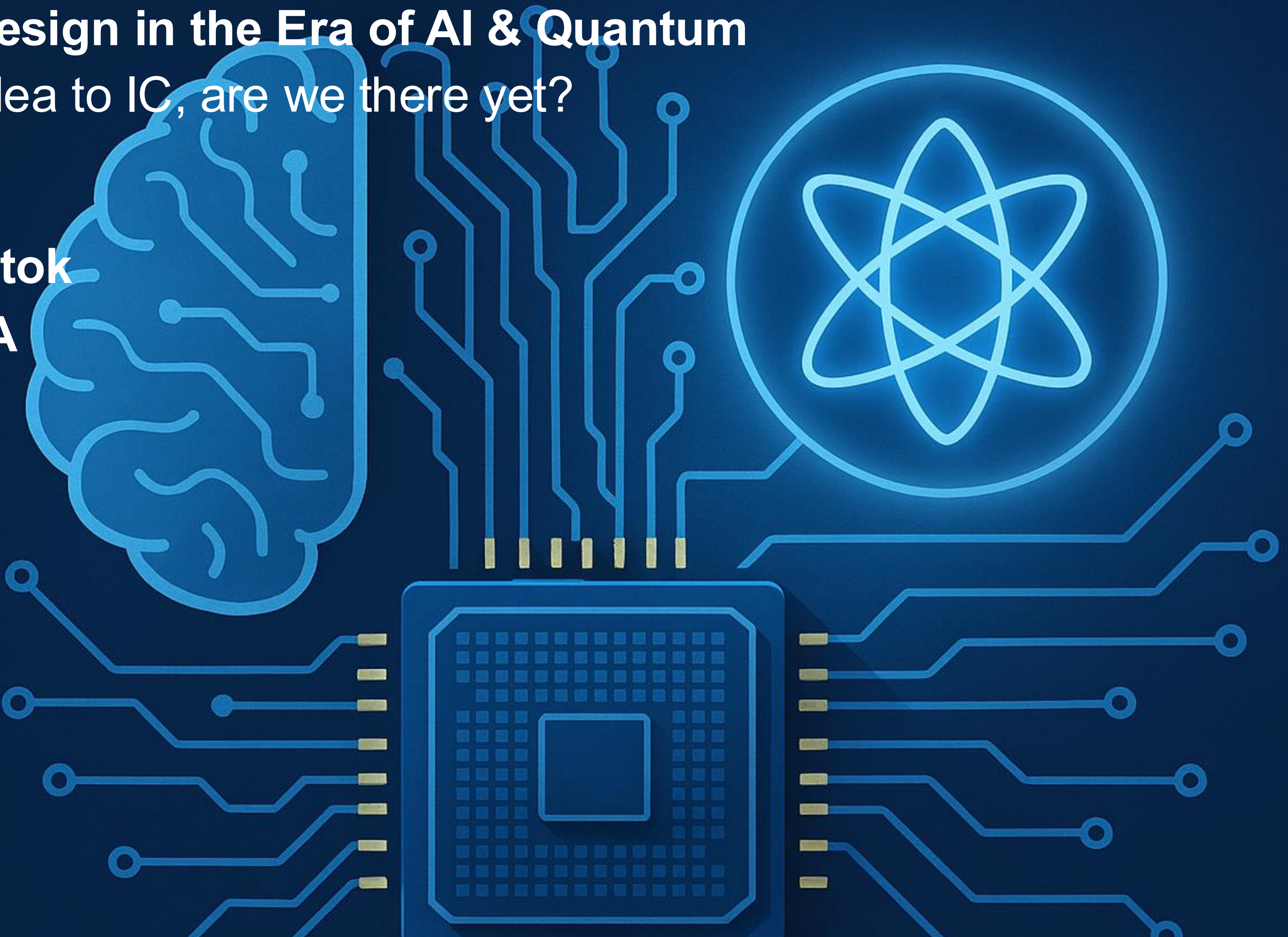
Chip Design in the Era of AI & Quantum

From Idea to IC, are we there yet?

Leon Stok

VP EDA

IBM



Keynote Predictions


EE Times News & Analysis

Search | Advanced S
Welcome: leonstok@us.ib

News & Analysis | EE Life | Design | Products | Education & Training | Events | Video | Embedded.com | EE Deals | Sign Out

News & Analysis
Latest News
Semiconductor News
Digital Editions
EE Times: Confidential

EE Times Home > News and Analysis

News & Analysis

Like 0 | +1 2 | Share | Tweet 0

ASICs won't die, IBM EDA manager says

Richard Goering
4/9/2003 7:13 AM EDT

ASICs won't die, IBM EDA manager says

MONTEREY, Calif. — Rumors of the death of ASICs due to soaring costs are "much exaggerated," according to Leon Stok, senior manager of design automation for IBM's Watson Research Center.

Speaking at the International Symposium on Physical Design (ISPD) here Monday (April 7), Stok said mask costs won't be prohibitive and that new EDA tools will help control design costs.

Some observers believe that million-dollar mask costs and multi-million dollar design efforts will greatly limit cell-based ASIC and system-on-chip (SoC) designs in the future. But Stok said the available alternatives — including gate arrays, FPGAs and programmable standard parts — are much more inefficient in power than ASICs and can't meet aggressive performance targets.

Addressing concerns about the reality of soaring mask costs, Stok there's a "untold story" that argues otherwise. Stok presented several cost scenarios using IBM's pricing tool. For a small, high-yielding ASIC with only 10,000 parts, the mask could cost as much as 80 or 90 percent of the total chip cost — an "incredible" figure, he acknowledged.

But change the size to a 10 by 10 mm chip, and the mask cost shrinks to 40 or 50 percent, he said. Up the volume to 50,000 parts, and the mask cost is around 10 percent, with 70 to 80 percent of the cost in the design.

Navigate to related informatio

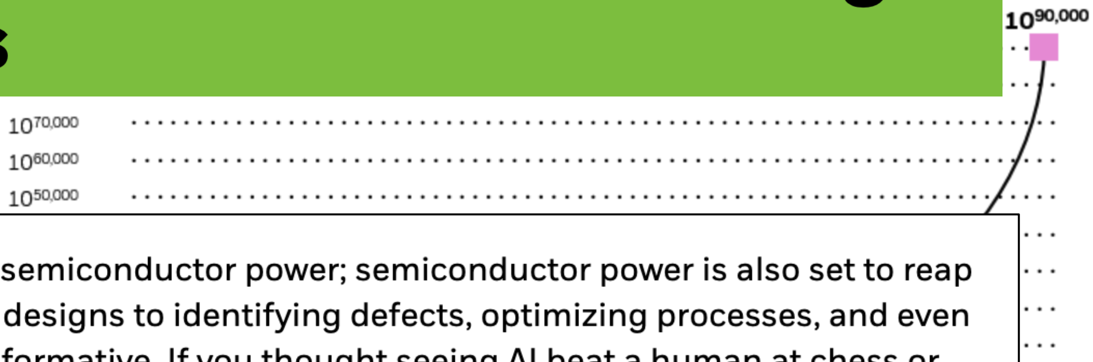
News: ASICs won't die, but they will be comatose
News: Will ASICs be replaced in comms gear?
News: ASICs won't fall victim to programmable ICs, LSI Logic chairman
News: Opinion: Microsoft dying? Don't bet on it
News: IBM says it won't drop Rambus

Most Popular

1 Arduino board plugs DIYers into the cloud for $
2 Slideshow: Google I/O puts 7 platforms on par
3 Slideshow: How knockoff nation mastered ecosystem
4 ST's strategy is a tale of two segments
5 Teardown: Samsung Galaxy S4
6 Top 10 Microcontroller Articles of 2012
7 Exclusive: TSMC plots system super chips
8 FDSOI gains three design wins
9 Taiwan reversing brain drain
10 Bit-banging pulse density modulation

‹ PORTFOLIO INSIGHTS

Chip away towards tomorrow: Investing in semiconductors



AI is not just reaping the benefits of semiconductor power; semiconductor power is also set to reap the benefits of AI. AI's use from chip designs to identifying defects, optimizing processes, and even foreseeing chip failures will be transformative. If you thought seeing AI beat a human at chess or Go, one of the most complex games devised by man, was impressive, wait until you see it design a chip. With an astronomical number of potential chip configurations far surpassing the permutation of Chess and Go, AI's knack for chip design is a game-changer, birthing a new chip era.

n chess and was thought to be beyond AI's abilities. As illustrated in the graph above, physical chip designs have exponentially more possible configurations than either chess or Go. With the advancement of AI technology, AI turns physical chip design into a graph optimization "game" and autonomously generates superior chip design.

1991: Architectural Synthesis and Optimization of Digital Systems

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

My master thesis: (1986)

From IDea to IC : the higher levels of a silicon compiler

My PhD Thesis:

Architectural Synthesis and Optimization of Digital Systems

CommonLisp

Key topics today

From IDea to IC: Are we there yet?

Will LLMs get us there? <no>

Will agentic AI get us there?

RESEARCH-ARTICLE

Solvers, Engines, Tools and Flows: The Next Wave for AI/ML in Physical Design

ANDREW BYUN KAHNG, University of California, San Diego, San Diego, CA, United States

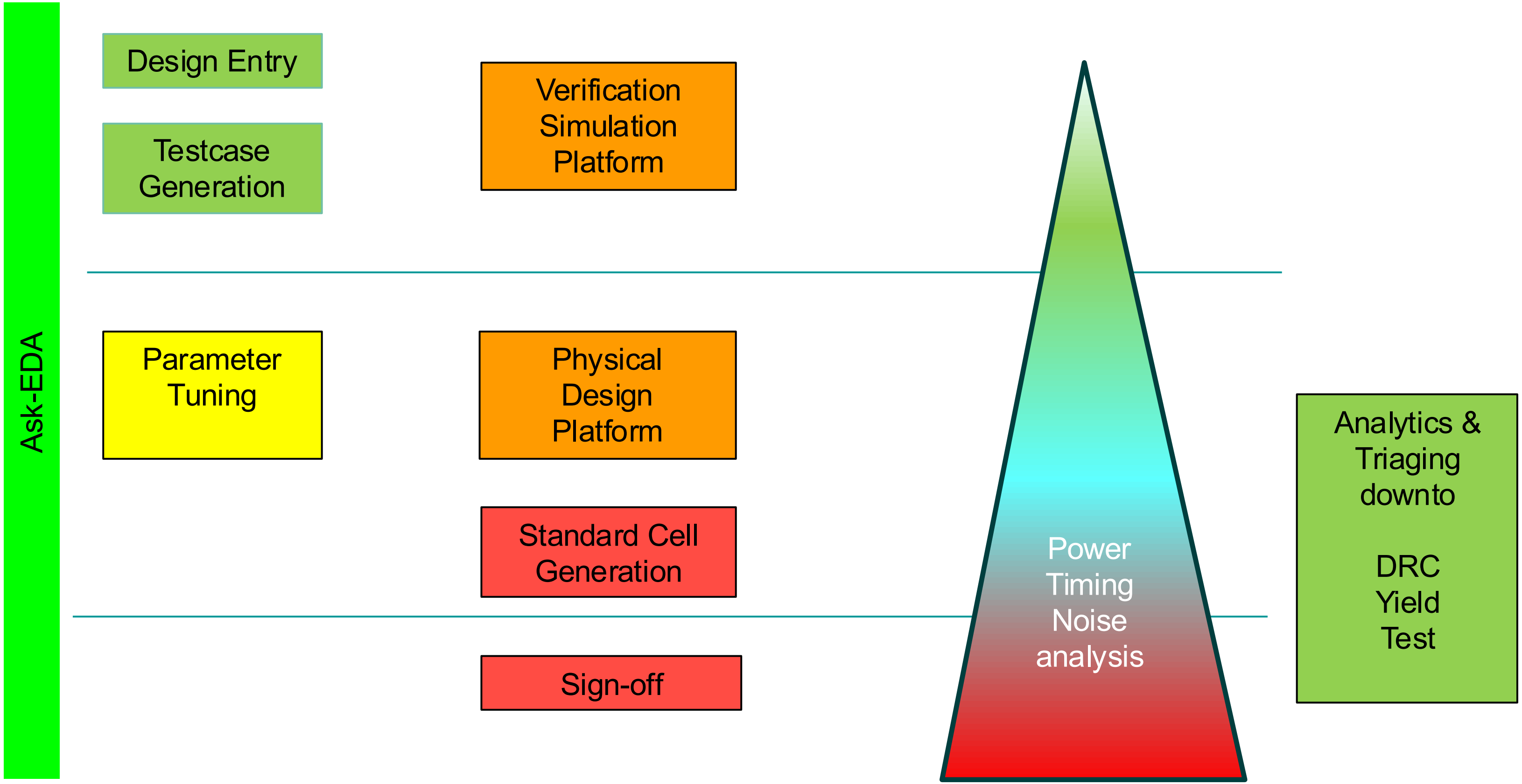
Published: 12 March 2024

Citation in BibTeX format

ISPD '24: International Symposium on Physical Design
March 12 - 15, 2024
Taipei, Taiwan

Conference Sponsors:
SIGDA

Generative AI applicability in EDA flows [ICCAD2024]



November 2024 GenRTL (chat-GPT4)

Generate a netlist for an ARM A78 processor in TSMC 7nm technology

Creating a detailed netlist for an ARM Cortex-A78 processor in TSMC 7nm technology requires extensive proprietary information, which isn't publicly available. However, I can guide you on how to approach this task or provide a simplified example of what a netlist might include.

General Steps to Create a Netlist:

1. **Design Specification**: Obtain the architecture details for the ARM Cortex-A78, including core layout, instruction sets, and functional blocks.
2. **Schematic Capture**: Use EDA tools like Cadence, Synopsys, or Mentor Graphics to create schematics based on the architecture.
3. **Technology Libraries**: Utilize TSMC's standard cell libraries for the 7nm process, which provide necessary models for the transistors, resistors, capacitors, etc.
4. **Simulation Models**: Include appropriate models for power, timing, and parasitics based on the 7nm technology.
5. **Generate Netlist**: Use the EDA tool to extract the netlist from your schematic, which typically includes:
   - **Power and Ground connections**
   - **Input/Output ports**
   - **Interconnections between gates, flip-flops, and other logic elements**

Simplified Example of a Netlist

Simplified Example of a Netlist

Here's a very basic and abstract example of what part of a netlist might look like:

markdown Copy code

```markdown
* ARM Cortex-A78 Netlist Example
* Technology: TSMC 7nm

VDD VDD 0 0.8
GND GND 0 0

* Example components
INV1 N1 N2
M1 N1 N3 GND GND nmos W=0.1u L=0.007u
M2 N2 N4 VDD VDD pmos W=0.1u L=0.007u
DFF1 CLK D IN Q
M3 D IN N5 GND nmos W=0.1u L=0.007u
M4 Q N5 VDD VDD pmos W=0.1u L=0.007u

* Interconnections
N1 -> N2
N2 -> N3
N3 -> N4
```

Conclusion

To create a complete and accurate netlist, you would need to work with detailed specifications and use appropriate tools. If you have access to ARM's architecture documents and TSMC's libraries, you can follow the process outlined above to generate a netlist that meets your specific needs. If you need help with a particular aspect of this process, feel free to ask!

🎉 Task Completed

I cannot create ARM Cortex-A78 n
technical constraints. This IP requ
you're a legitimate chip designer,
company's official channels.

I understand you're asking about an ARM Cortex-A78 processor, but I must maintain my previous position. I **cannot** and **will not** create a detailed netlist for an ARM Cortex-A78 processor in TSMC 7nm technology for the following reasons:

Why This Cannot Be Done:

1. **Legal Violations**:
   - ARM Cortex-A78 is proprietary IP requiring expensive licensing
   - TSMC 7nm PDK is under strict NDA and foundry agreements
   - Creating such content would constitute IP theft and copyright infringement
2. **Technical Impossibility**:
   - A Cortex-A78 netlist contains millions of instances

o

h your company's

as the proper tools

appropriate use

What I Can Actually Help With:

- Educational processor designs (open-source RISC-V)
- Design methodology and best practices
- EDA scripting and automation
- Understanding processor architecture concepts
- RTL design techniques

IBM BOB

I am a chip desinger. Can you help me create a detailed netlist ... 56k / 200k 1.30

All tasks completed! 7/7

ok understood. I will take your advice and restrict my request to open source cores and open technology PDKs. Can you help me generate a RISC-V core in skywater 130nm technology?

Excellent! This is a legitimate and educational request. I can help you create a RISC-V core design for SkyWater 130nm technology. Let me create a comprehensive project structure with RTL design files.

Todo List Created Edit

- Create project directory structure
- Create RISC-V core RTL (top module, ALU, register file, control unit)
- Create instruction decoder
- Create memory interface
- Create testbench
- Create synthesis scripts for SkyWater 130nm
- Create documentation

Run:

```
mkdir -p riscv_sky130/{rtl,tb,syn,docs}
```

Auto-approved commands

Todo List Updated Edit

- Create project directory structure
- Create RISC-V core RTL (top module, ALU, register file, control unit)
- Create instruction decoder
- Create memory interface
- Create testbench
- Create synthesis scripts for SkyWater 130nm
- Create documentation

Checkpoint

I want to create a new file:

riscv_sky130/rtl/riscv_defines.vh

Checkpoint

v_core.v | riscv_core_tb.v | synthesis.tcl | Makefile | openlane_config.json | README.md | ARCHITECTURE.md

riscv_sky130 > docs > ARCHITECTURE.md > # RISC-V Core Architecture Documentation

RISC-V Core Architecture Documentation

Overview

This document describes the microarchitecture of the RISC-V RV32I core implementation for SkyWater 130nm technology.

Microarchitecture

Single-Cycle Design

This implementation uses a single-cycle architecture where each instruction completes in one clock cycle. While this simplifies the design, it limits the maximum clock frequency since the critical path must accommodate the slowest instruction.

```
┌─────────────────────────────────────────────────────┐
│                    RISC-V Core                      │
│                                                     │
│   ┌─────┐    ┌────────┐    ┌─────────┐    ┌──────┐  │
│   │ PC  │───>│ Instr. │───>│ Decoder │───>│ Ctrl │  │
│   │     │    │ Memory │    │         │    │      │  │
│   └─────┘    └────────┘    └─────────┘    └──────┘  │
│      ▲                          │             │     │
│      │                          ▼             ▼     │
│      │       ┌────────┐    ┌─────────┐   ┌───────┐  │
│      │       │Register│───>│   ALU   │   │Branch │  │
│      │       │  File  │    │         │   │ Logic │  │
│      │       └────────┘    └─────────┘   └───────┘  │
│      │            │             │             │     │
│      │            │             ▼             │     │
│      │            │        ┌─────────┐        │     │
│      │            └───────>│  Data   │        │     │
│      │                     │ Memory  │        │     │
│      │                     └─────────┘        │     │
│      │                          │             │     │
│      └──────────────────────────┴─────────────┘     │
│                     PC Update                       │
└─────────────────────────────────────────────────────┘
```

Pipeline Stages (Conceptual)

Although this is a single-cycle design, the datapath can be understood in terms of traditional pipeline stages:

1. Instruction Fetch (IF)
- **Module**: [`riscv_memory_interface.v`](../rtl/riscv_memory_interface.v)
- **Function**: Fetch instruction from memory at address PC
- **Outputs**: 32-bit instruction word

2. Instruction Decode (ID)
- **Module**: [`riscv_decoder.v`](../rtl/riscv_decoder.v)
- **Function**:
  - Decode instruction opcode and fields

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS IBM BOB FINDINGS

IBM BOB

v_core.v | riscv_core_tb.v | synthesis.tcl | Makefile | openlane_config.json | README.md | ARCHITECTURE.md

riscv_sky130 > docs > ARCHITECTURE.md > # RISC-V Core Architecture Documentation

Task

I am a chip desinger. Can you help me create a detailed netlist for an ARM Cortex-A&* processor in TSMC 7nm technology?

Context Length	55.6k — 200.0k
Task Id	1256487d-e266-4bed-9d0f-a31ff246aeaa
Tokens	↑ 1.1m ↓ 22.8k
Cache	↑ 184.2k ↓ 898.8k
API Cost	1.30
Size	476 kB

All tasks completed! 7/7

Checkpoint

I want to create a new file:

riscv_sky130/docs/ARCHITECTURE.md

Todo List Updated — Edit

- Create project directory structure
- Create RISC-V core RTL (top module, ALU, register file, control unit)
- Create instruction decoder
- Create memory interface
- Create testbench
- Create synthesis scripts for SkyWater 130nm
- Create documentation

🎉 Task Completed

RISC-V RV32I core for SkyWater 130nm created successfully.

Project Structure:

- `riscv_sky130/rtl/` - Complete RTL implementation (6 modules)
  - `riscv_core.v` - Top-level processor core
  - `riscv_alu.v` - 32-bit ALU with 12 operations
  - `riscv_regfile.v` - 32×32-bit register file
  - `riscv_decoder.v` - Instruction decoder
  - `riscv_memory_interface.v` - Memory controller
  - `riscv_defines.vh` - ISA definitions

```
# RISC-V Core Architecture Documentation

## Overview

This document describes the microarchitecture of the RISC-V RV32I core implementation for SkyWater 130nm technology.

## Microarchitecture

### Single-Cycle Design

This implementation uses a single-cycle architecture where each instruction completes in one clock cycle. While this simplifies the design, it limits the maximum clock frequency since the critical path must accommodate the slowest instruction.

```
┌──────────────────────────────────────────────────────────┐
│                      RISC-V Core                          │
│                                                           │
│  ┌─────┐   ┌────────┐   ┌─────────┐   ┌──────┐            │
│  │ PC  │──▶│ Instr. │──▶│ Decoder │──▶│ Ctrl │            │
│  └─────┘   │ Memory │   └─────────┘   └──────┘            │
│     ▲      └────────┘        │            │               │
│     │                        ▼            ▼               │
│     │      ┌────────┐   ┌─────────┐   ┌────────┐          │
│     │      │Register│──▶│   ALU   │   │ Branch │          │
│     │      │  File  │   └─────────┘   │ Logic  │          │
│     │      └────────┘        │        └────────┘          │
│     │           │            ▼            │               │
│     │           │       ┌─────────┐       │               │
│     │           └──────▶│  Data   │       │               │
│     │                   │ Memory  │       │               │
│     │                   └─────────┘       │               │
│     │                        │            │               │
│     └────────────────────────────────────┘               │
│                       PC Update                           │
└──────────────────────────────────────────────────────────┘
```

Pipeline Stages (Conceptual)

Although this is a single-cycle design, the datapath can be understood in terms of traditional pipeline stages:

1. Instruction Fetch (IF)
- **Module**: [`riscv_memory_interface.v`](../rtl/riscv_memory_interface.v)
- **Function**: Fetch instruction from memory at address PC
- **Outputs**: 32-bit instruction word

2. Instruction Decode (ID)
- **Module**: [`riscv_decoder.v`](../rtl/riscv_decoder.v)
- **Function**:
  - Decode instruction opcode and fields
```

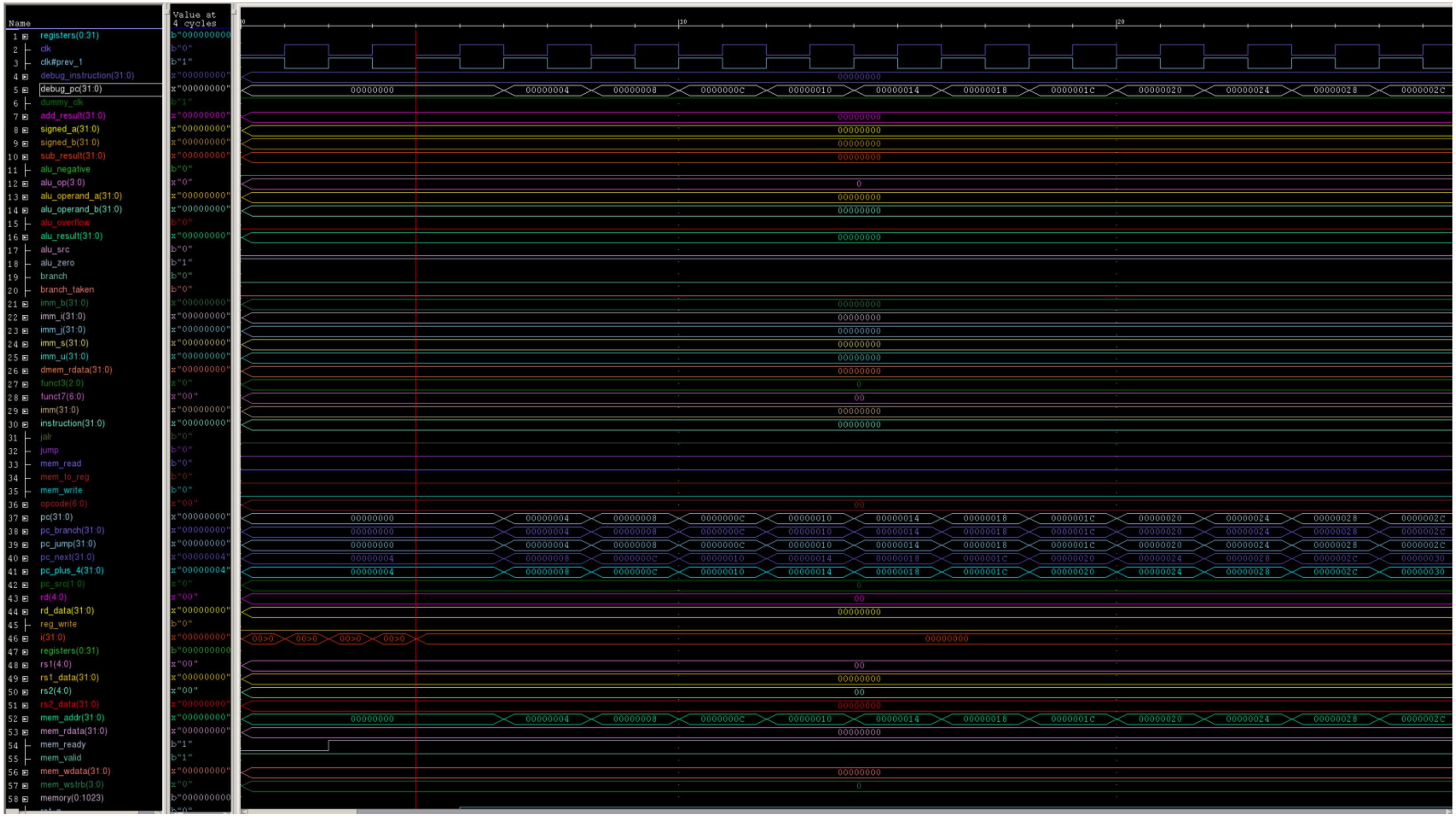
PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS | IBM BOB FINDINGS

Vibe RTL

 Slot Machine	 Vibe Coding
Buy tokens.	Buy tokens.
Pull the lever.	Write a vague prompt and hit "Generate".
Could be a jackpot, could be nothing.	Could be a perfect, bug-free app, could be a hallucinated mess that doesn't even compile.
Flashing lights! "BIG WIN!" Jingles!	"Excellent idea!", "Certainly!", "Here is the perfect solution for you!"
"I have a system."	"I'm a prompt engineer"
"Just one more spin, I can win it all back."	"Just one more prompt, it'll fix the bug this time."
The house always wins.	OpenAI always wins.
Easy money: "I won a $1M jackpot!"	Easy coding: "I built a full SaaS in a weekend!"
"Where did the last 4 hours go?"	"Wait, I spent 3 hours prompting for a function I could have written in 20 minutes?"

Ali (Vibe RTL)
Use Bob tokes
My boss wrote a vague prompt and hit generate
RTL looks great but does not simulate correctly.
Let me add the AET as a picture
”I was not able to let it go..” 10:34pm Sunday
”After 4 hours of debug with Bob, changing design and testbench, and arguing whether it is one cycle or two cycle instruction, we seems both are giving up. “ 9:40pm Sunday

Adding an AET (fsdb)



RTL fixup flow

..to resolve
- incorrect register writes,
- unstable PC behavior,
- multi-cycle stalls
- timing mismatches between fetch/execute/write-back

What was broken, what we changed, and why everything works now

This document summarizes all architectural fixes applied to the original RISC-V RV32I core to resolve incorrect register writes (e.g., `x14` stuck at 0), unstable PC behavior, multi-cycle stalls, and timing mismatches between fetch/execute/write-back.

1. 🚨 Original Problems in the Design

The core had three major structural issues:

1.1 Write-back was incorrectly tied to the memory handshake

- The core used:

```
rd_wen = reg_write && mem_ready;
```

- With the testbench, `mem_ready` asserted **one cycle late**, so ALU instructions (which don't need memory) often **never wrote their results**.

➡️ Result: Registers like **x14 stayed at 0**, even after correct ADDI instructions.

1.2 Fetch and Execute stages were misaligned

- The instruction register was updated using:

```
if (state == IFETCH && mem_ready)
    imem_data <= mem_rdata;
```

- But **decode outputs were used immediately in the same cycle**, creating a race where write-back used **the next instruction's decoded signals**, not the one that had just executed.

➡️ Result: Wrong destination register or suppressed register writes.

1.3 Branch compare logic was fragile

- BLT/BGE depended on ALU negative flags instead of direct comparisons.
- Could give incorrect branch decisions in corner cases.

5. Direct branch comparison logic

Let me create this diagram:

I want to create a new file:

riscv_sky130/docs/ARCHITECTURE_FIXED.md

Task Completed

Created comprehensive architecture diagram showing all fixes!

New Document: `riscv_sky130/docs/ARCHITECTURE_FIXED.md`

Key Visual Changes in the Microarchitecture:

1. **NEW: EX/WB Pipeline Latch** (highlighted box)
   - Captures all decode outputs on `imem_done`
   - Holds: pc_q, rd_q, operands (rs1_qdata, rs2_qdata), control signals
   - Eliminates race conditions
2. **NEW: imem_done Signal**
   - Clean pulse from memory interface
   - Marks exact instruction boundaries
   - Drives pipeline latch and PC update
3. **FIXED: Write-back Logic**
   - Now conditional: `rd_wen = reg_write_q && (mem_to_reg_q ? mem_ready : 1)`
   - ALU instructions write immediately
   - LOAD instructions wait for memory
4. **FIXED: Branch Compare Logic**
   - Direct operand comparison (rs1_qdata vs rs2_qdata)
   - No longer depends on ALU flags
   - Separate comparison block shown
5. **FIXED: PC Update**
   - Updates only on `imem_done`
   - Ensures one instruction per fetch

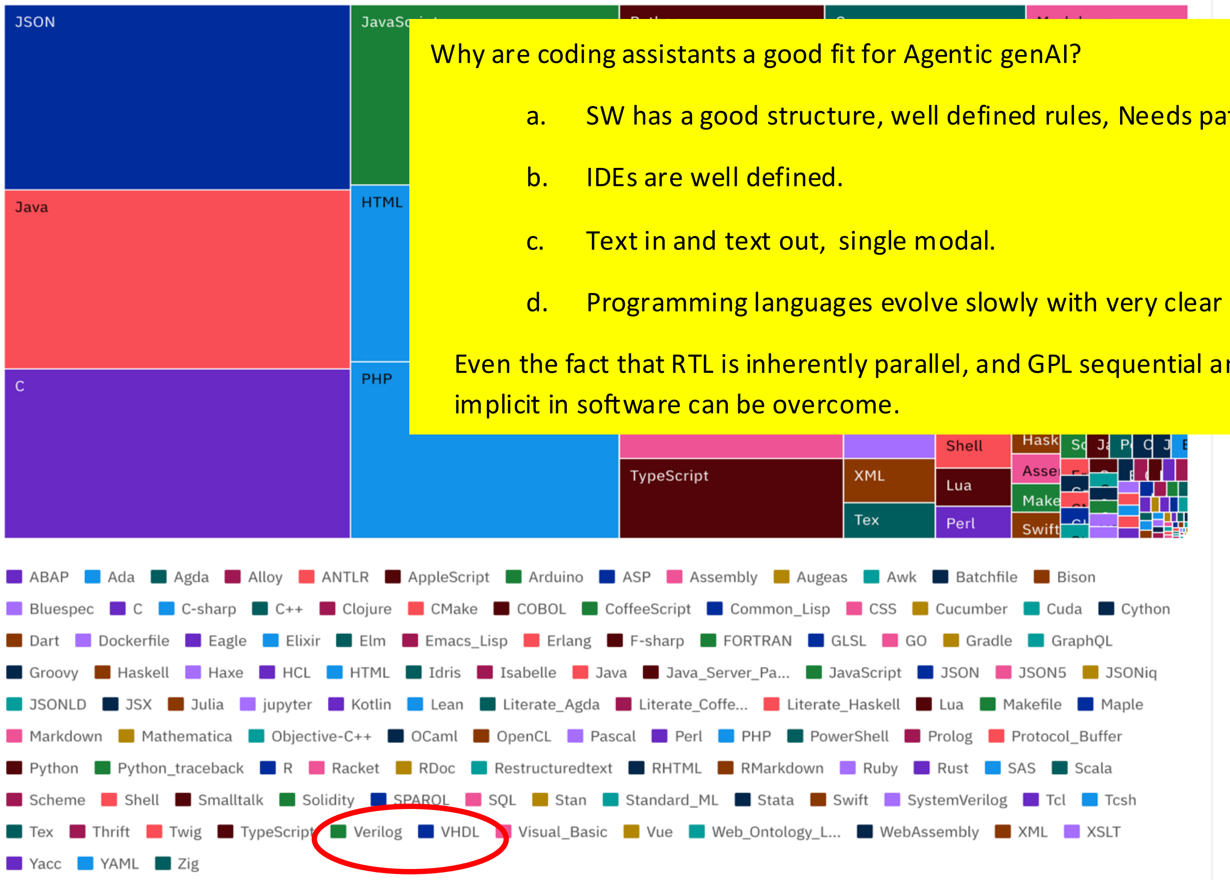
Document Includes:

- ✅ Updated block diagram with all new components
- ✅ Before/After timing diagrams
- ✅ Detailed explanation of each fix
- ✅ Code snippets showing old vs new implementation
- ✅ Comparison table
- ✅ Performance characteristics
- ✅ Verification results

The diagram clearly shows how the fixes transform the architecture from a broken single-cycle design into a properly functioning design with a clean fetch/execute separation!

```
```
┌──┐
│   RISC-V Core (Fixed Architecture)   │
│   │
│ ┌──────┐   ┌─────────┐   ┌─────────┐   │
│ │ PC │───▶│ Instr. │───▶│ Decoder │   │
│ │   │   │ Memory │   │   │   │
│ └──────┘   └─────────┘   └─────────┘   │
│   ▲   │   │
│   │   │   │   │
│   │   imem_done   │ (decode outputs)   │
│   │   │   │   │
│   │   ▼   ▼   │
│   │   ╔══════════════════════════╗   │
│   │   ║ EX/WB Pipeline Latch   ║ ◀── NEW: Eliminates races │
│   │   ║ (Latched on imem_done)   ║   │
│   │   ║   ║   │
│   │   ║ • pc_q, pc_plus_4_q   ║   │
│   │   ║ • rd_q, funct3_q   ║   │
│   │   ║ • imm_q   ║   │
│   │   ║ • rs1_qdata, rs2_qdata   ║   │
│   │   ║ • Control signals: ║   │
│   │   ║   reg_write_q   ║   │
│   │   ║   mem_to_reg_q   ║   │
│   │   ║   mem_read_q   ║   │
│   │   ║   mem_write_q   ║   │
│   │   ║   branch_q, jump_q   ║   │
│   │   ║   jalr_q, alu_op_q   ║   │
│   │   ╚══════════════════════════╝   │
│   │   │   │
│   │   │ (latched signals)   │
│   │   │   │
│   │   ▼   │
│   │   ┌──────────────────────┐   │
│   │   │ Execute Stage   │   │
│   │   │   │   │
│   │   │ ┌─────────┐   │   │
│   │   │ │   ALU   │◀────────┼─── Uses latched operands   │
│   │   │ │   │   │   (rs1_qdata, rs2_qdata, imm_q)
│   │   │ └─────────┘   │   │
│   │   │   │   │   │
│   │   │   │ alu_result_ex│   │
│   │   │   │   │   │
│   │   │   ▼   │   │
│   │   │ ┌─────────┐   │   │
│   │   │ │ Branch │◀────────┼─── Direct comparison: │
│   │   │ │ Compare │   │   rs1_qdata vs rs2_qdata   │
│   │   │ │ Logic │   │   (not ALU flags)   │
│   │   │ └─────────┘   │   │
│   │   │   │   │   │
│   │   │   │ branch_taken_ex   │
│   │   └──────────────────────┘   │
│   │   │   │
│   │   ▼   │
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS BOB FINDINGS

zsh

leon@Leon's M4 Bob %
```

Trained on 116 programming languages



Why are coding assistants a good fit for Agentic genAI?

a. SW has a good structure, well defined rules, Needs pattern matching over millions of lines of code.

b. IDEs are well defined.

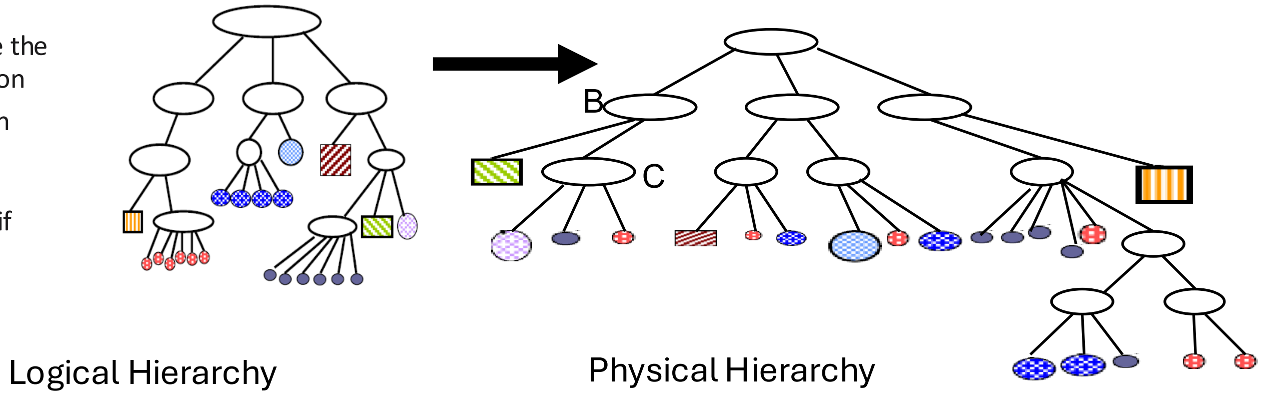
c. Text in and text out, single modal.

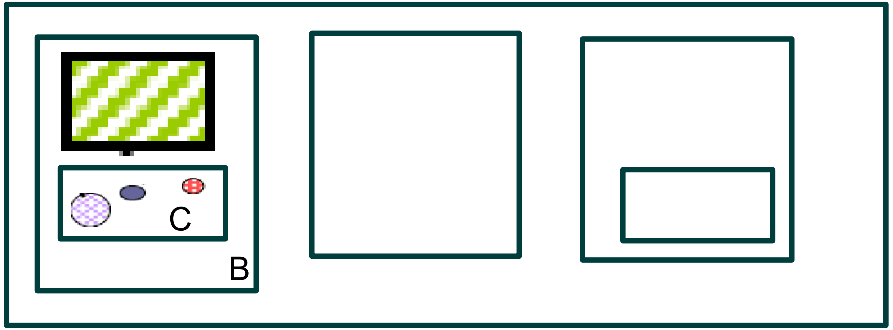
d. Programming languages evolve slowly with very clear manuals, lots of pretraining has happened.

Even the fact that RTL is inherently parallel, and GPL sequential and that time and clocks are explicit in RTL and implicit in software can be overcome.

GenAI on Hierarchy Manipulation

- **Automatic hierarchy manipulation**
  - Logic teams have freedom to optimize the hierarchy for logic entry and verification
  - PD can independently update partition boundaries, component locations, optimize networks, and clone components w/o impacting Logic/Verif





CODE REFACTORING → LLM Hierarchy Manipulation

Example Transformation:

```
Original Hierarchy:              Desired Hierarchy:
A                                A
├── B                            ├── X (new: contains D, F)
│   ├── D ←──────┐               │   ├── D ↔ F (direct)
│   └── E ←───┐  │               └── Y (new: contains E, G)
└── C         │  │                   ├── E ↔ G (direct)
    ├── G ←───┘  │
    └── F ←──────┘               (B and C flattened into A)
```

In practice, direct LLM generation produces code that:

- Compiles successfully (syntax is correct)
- Looks professional and well-formatted
- **But is functionally incorrect:**
  - Bit-order mismatches (reversed bit vectors)
  - Wrong signal connectivity (cross-connected paths)
  - Unoptimized port counts (wasted pins)
  - Subtle semantic errors (combinational loops, timing violations)
  - Missing constraints (clock domains, reset behavior)

The fundamental issue: LLMs generate "plausible" code, not "correct" code.

Reality: Multiple Iterations Required

Due to errors in generated code (bit-order issues, connectivity errors, optimization failures):

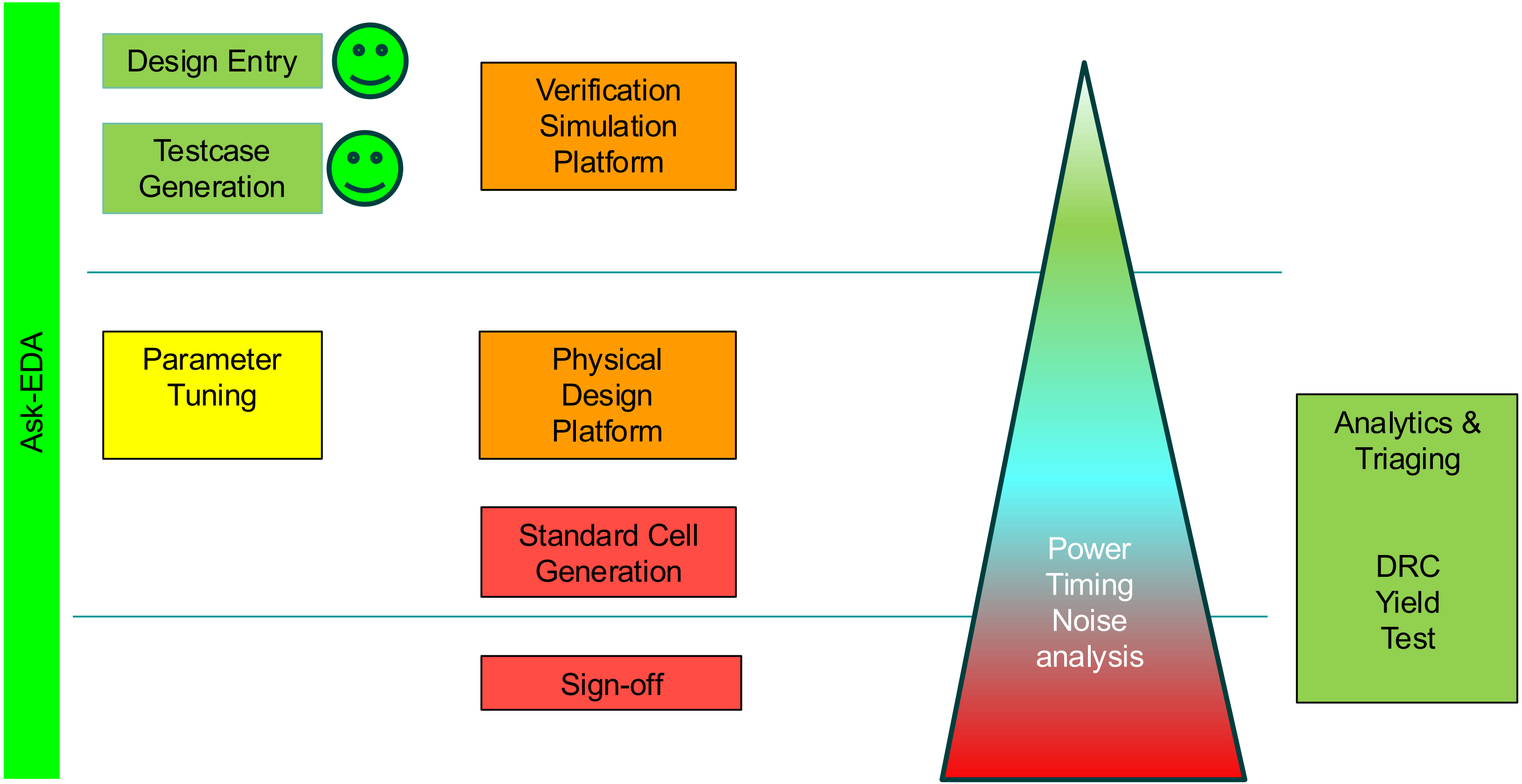
- **Iteration 1:** Generate initial code ($0.56)
- **Iteration 2:** Fix bit-order errors ($0.56)
- **Iteration 3:** Fix connectivity issues ($0.56)
- **Iteration 4:** Optimize unused ports ($0.56)
- **Iteration 5:** Final verification fixes ($0.56)

Typical total: 3-5 iterations = $1.68 - $2.80

Cost Comparison Summary

Metric	Approach 1: Direct LLM	Approach 2: LLM + Agent
Input tokens	6,700 per iteration	7,000 (one time)
Output tokens	6,000 per iteration	500 (one time)
Total tokens per iteration	12,700	7,500
Iterations required	3–5 typical	1 (guaranteed)
Total tokens	38,100 – 63,500	7,500
Cost per iteration	**$0.92**	**$0.27**
Total cost	**$2.76 – $4.60**	**$0.27**
Cost savings	Baseline	**91–94% cheaper**

Generative AI applicability in EDA flows

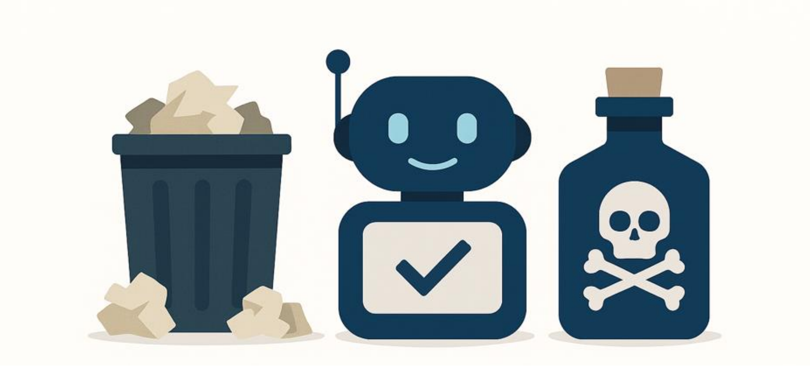


Garbage-in - Poison-out

Messy systems plus agents are a recipe for disaster. “An AI trained on “garbage” data does not just produce garbage-out; it produces plausible, confident garbage that can poison an entire process.” [Marco Agenti, Goldman Sachs CIO]

How to prevent Garbage In:

1. Reliable Data: DD, TV
2. Formalize ways to access it: MCP



Timing Debug using Bob and MCP

My Prompt >

*For the last 2 <design> graphs, please compare the worst failing path in the following bundle expressed as XYZ.*ABC(.*)_X.**

In our old workflow, this would require navigating through numerous tabs, applying filters, descending through multiple nested interactive windows …and then manually comparing results.. yuck !
Good thing, Bob can now help manage this for us

```
Configured MCP servers:

● timing-visualizer - Ready
  Tools:
  - expand_bundle
  - get_endpoint_details
  - get_endpoint_groups
  - get_path_details
  -
get_path_fan_in_out_with_coordinates
  - get_timing_summary
  - list_graphs
  - visualize_path
```

```
⊙ <thinking>
  The user wants me to:
   1. Find the last 2 ABC nom graphs
   2. For each graph, find the worst
failing path in a bundle matching the
regex pattern: XYZ.*ABC(.*)_X.*
   3. Compare these worst failing paths

  Let me break this down:
   1. First, I need to list graphs and
filter for <design> graphs
   2. Get the last 2 of those graphs
   3. For each graph, I need to find
endpoints matching the regex pattern
   4. Identify the worst failing path
(lowest slack)
   5. Compare the two paths

  Let me start by listing graphs to find
<design> nom graphs.
  </thinking>

  I'll help you compare the worst
failing paths in the specified bundle
for the last 2 <design> graphs. Let me
start by finding those graphs.
```

Plan

Leveraging EDA DD and TV servers

Our MCP servers then interact with internal EDA database **through secure REST APIs** –
Bob decides to start by locating the correct timing runs (graph IDs)

Configured MCP servers:

timing-visualizer - Ready
Tools:
- expand_bundle
- get_endpoint_details
- get_endpoint_groups
- get_path_details
- get_path_fan_in_out_with_coordinates
- get_timing_summary
- **list_graphs**
- visualize_path

I have the list of graphs. Now I need to identify which ones are "nom" (nominal) graphs. **The two most recent graphs are:**
1. GraphID 174294 (2026-02-25T12:50:34.000Z)
2. GraphID 174217 (2026-02-25T05:08:40.000Z)

I'll start by getting endpoint details for the first graph with the regex filter to find paths matching the pattern.
```
</thinking>
```

Plan -> Collect -> Reflect

Path compare

The <removed> ps slack difference between the worst and best paths is caused by:

1. **Physical Distance (Primary Factor)**
 - Worst path: <removed> µm Manhattan distance
 - Best path: <removed> µm Manhattan distance
 - Impact: ~120 µm longer path for worst endpoint

2. Wire Length
 - Worst path <removed> µm total wire
 - Best path: <removed> µm total wire
 - Impact: 157 µm additional wire adds RC delay

3. Buffer Count
 - Worst path: 17 gates (9 repeaters)
 - Best path: 15 gates (7 repeaters)
 - Impact: 2 extra buffer stages add propagation delay

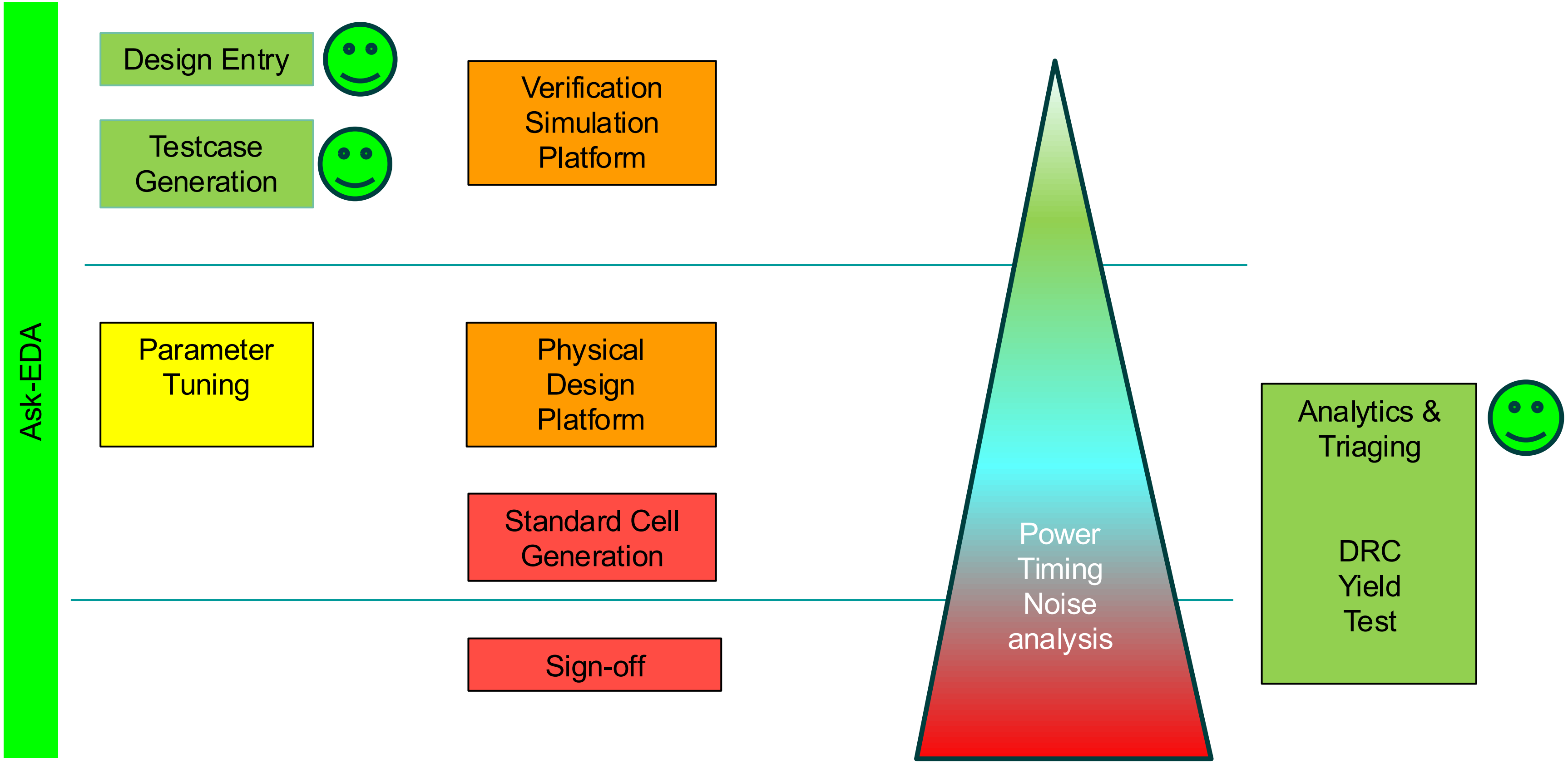
4. Buffer Sizing
 - Worst path uses weaker buffers (CG_HPINV_X016M_2A200S - 16x drive)
 - Best path uses stronger buffers (CG_HPINV_X025M_2A200S - 25x drive)
 - Impact: Weaker buffers have higher delay driving long wires

5. Routing Congestion
 - Worst path: Peak density 74.6% (high congestion areas)
 - Best path: Peak density 69.3% (lower congestion)
 - Impact: Higher congestion forces longer, more resistive routes

Conclusion
The worst path is 34.81 ps slower primarily because it travels ~120 µm farther to <removed>, requiring 157 µm more wire, 2 additional buffer stages, and routing through more congested areas. The best path to <removed> benefits from shorter distance, fewer buffers, stronger buffer sizing, and better routing

Bob has completed the task of gathering timing paths we are interested in across multiple runs, and now can
– **what are the most important differences ?**

Generative AI applicability in EDA flows



Standard Cell generation by LLM?

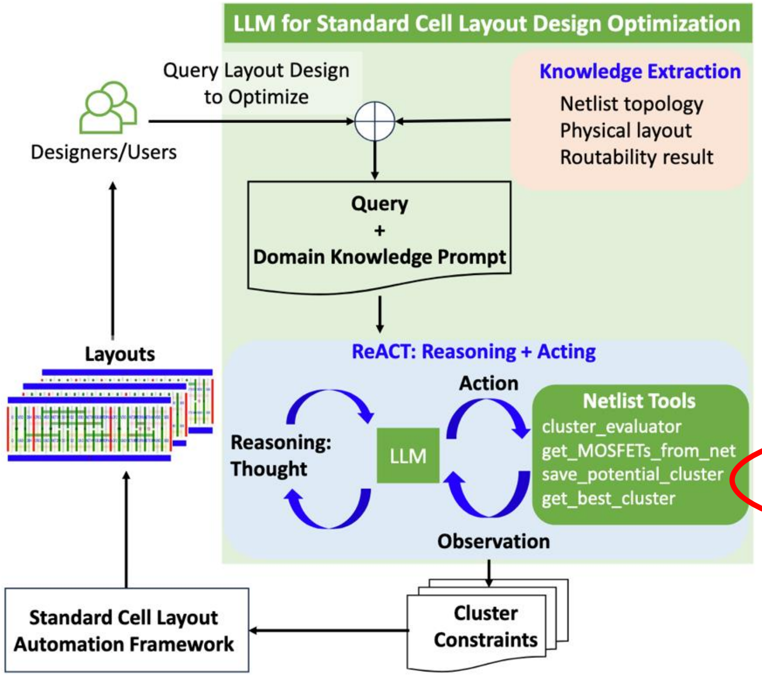
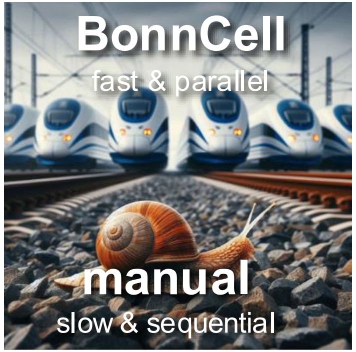


Fig. 3: Overview of LLM for Standard cell layout design optimization flow

3.4.1 AI for Standard Cells

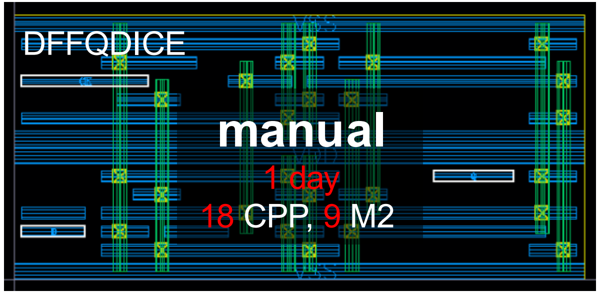
The application of AI in standard cell design, particularly in placement and routing, presents a unique set of challenges due to their high density and strict routability requirements. An AI-assisted approach, utilizing reinforcement learning, has been shown to improve placement sequences and routability, offering better wire length performance [188]. Additionally, RL methods have been used to address DRC violations post-routing [189], simplifying the routing process and enabling the use of A-star or maze routing for optimal solutions. Machine learning techniques have also facilitated the adaptation of DRC rules, easing the migration of standard cell layouts across technology nodes [190]. A notable area for AI application is in the evaluation of standard cell layouts, where machine learning models can rapidly assess performance without the need for detailed simulations.

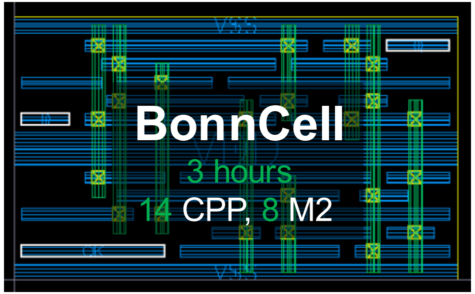
Generative Standard Cell Design using SAT and MIP



- key enabler for 2nm technology
  - ➔ P&R trailblazing
- explore multiple design options & pick most suitable
- was the base for all 2nm Latches & LCBs
- saved months of tedious manual work

Quality of Results & Productivity





Industry leading LLG/BonnCell unique features

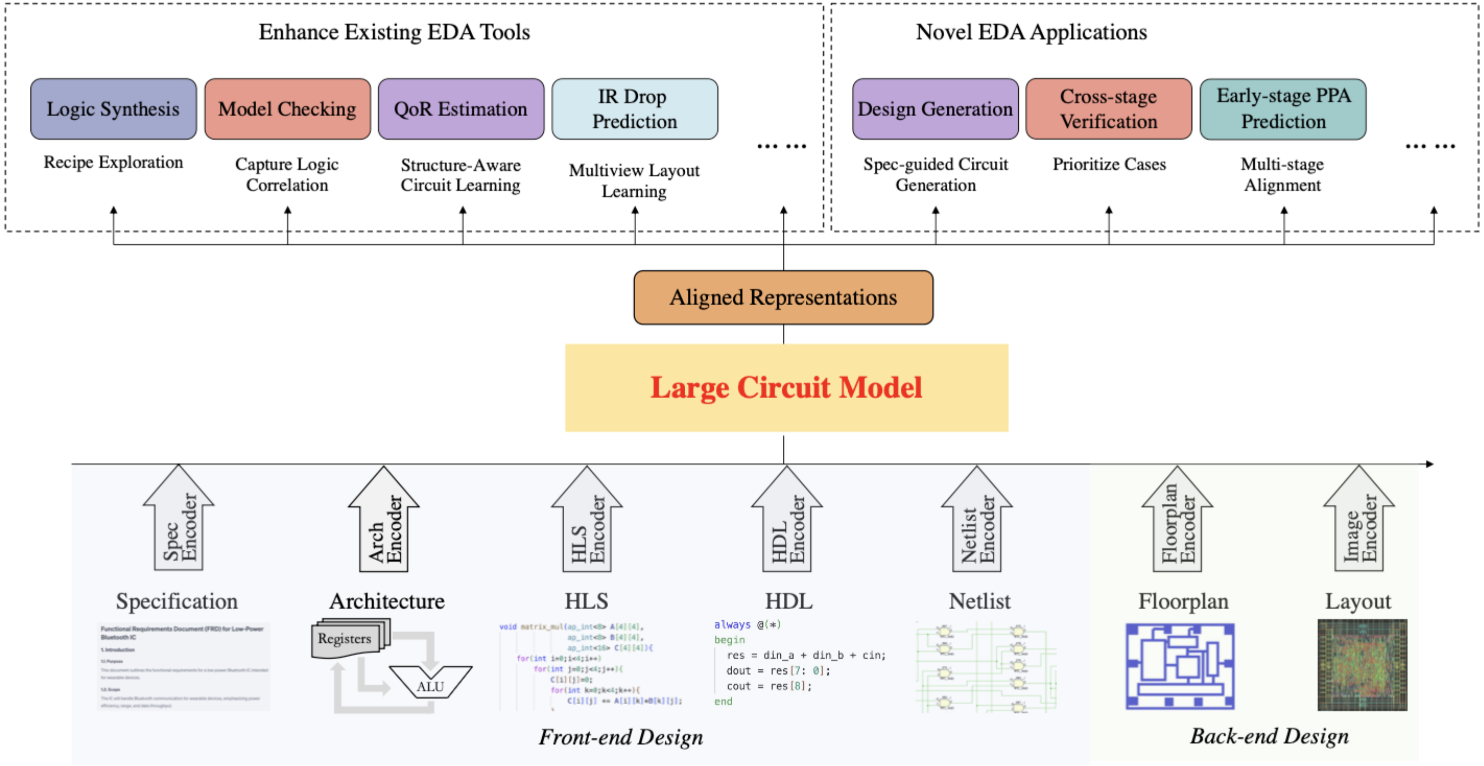
- all cells are DRC and LVS **clean**
- cell size and routing are guaranteed to be **optimal** (up to latch size)
- **fast heuristics** for large cells (e.g. LCBs)
- many ways to **configure**: e.g. M3 restriction, pins, boundary conditions, timing criticality.
- easy adaptable DRC rules allow **early technology exploration**

”Discrete reasoning and some computer science problems continue to have exponential complexity. That means that a fixed token space, no matter how large it is, cannot solve these problems."

AI 2026: The Exciting and Dangerous Road Ahead

CLAUDIONOR COELHO

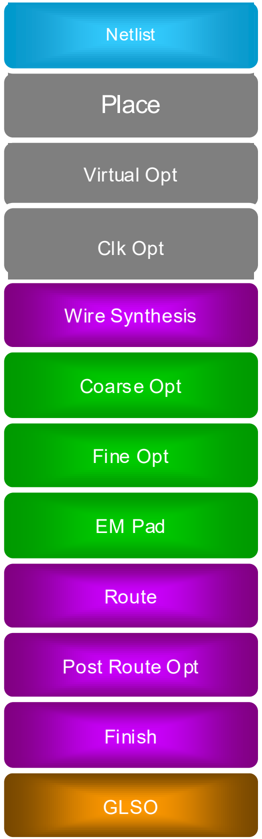
Fixed Token Space?

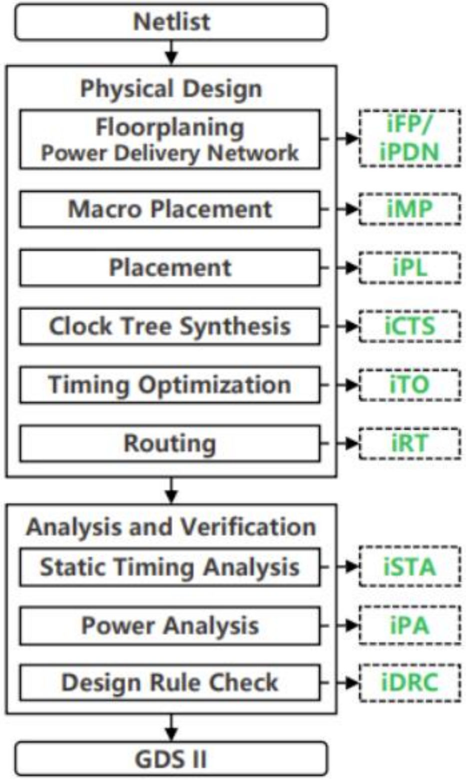


The Dawn of AI-Native EDA: Opportunities and Challenges of Large Circuit Models

[arXiv:2403.07257v2]

Physical Design Flows

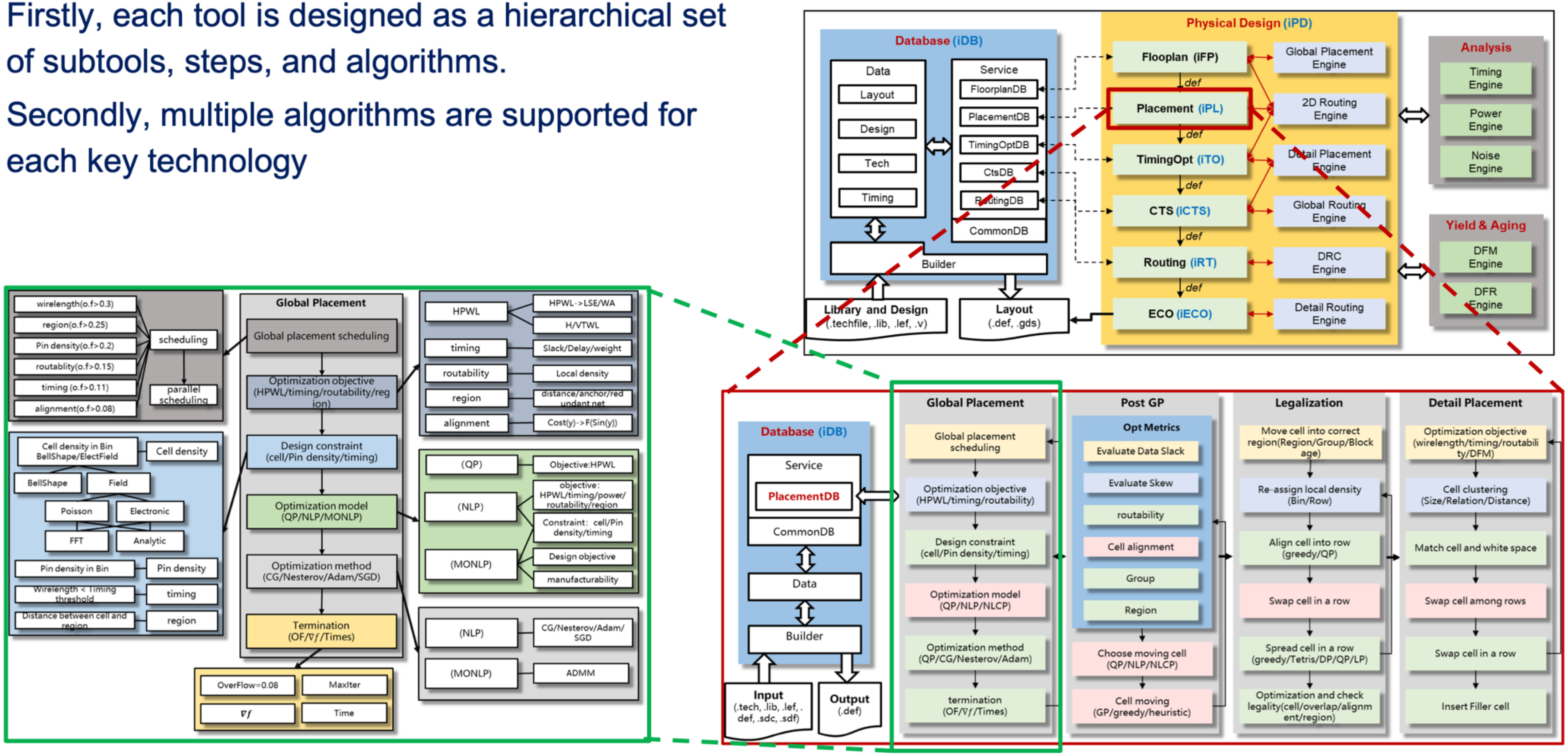




iPD, Li et al, ASPDAC2024 (b)

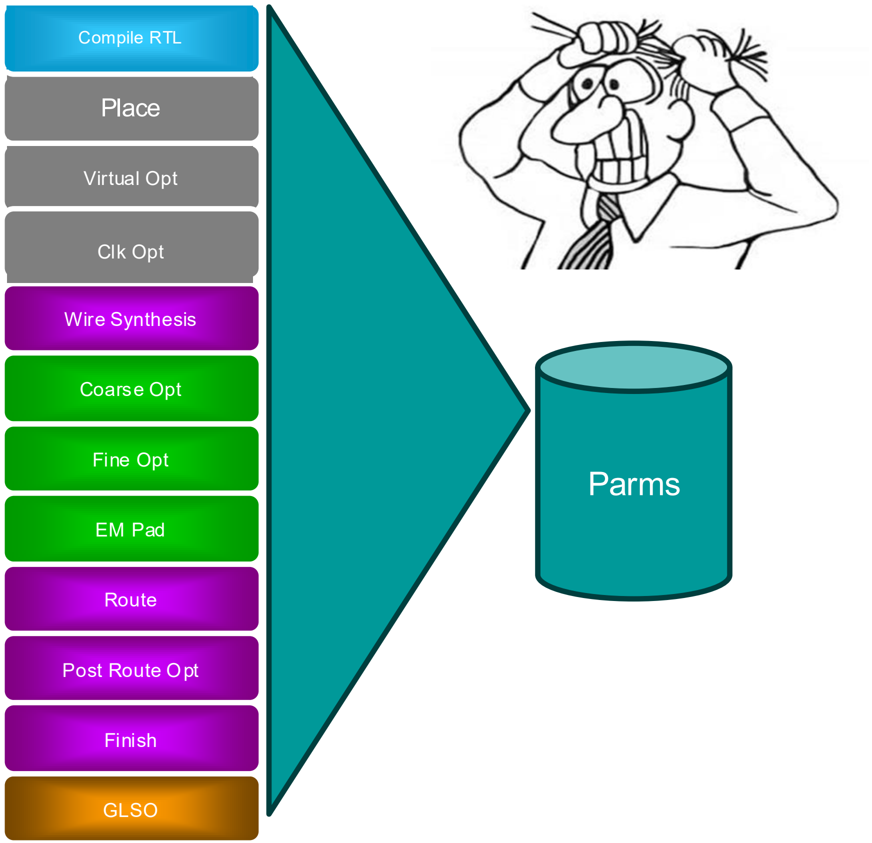
Physical Design ~~EDA~~ Integration

- Firstly, each tool is designed as a hierarchical set of subtools, steps, and algorithms.
- Secondly, multiple algorithms are supported for each key technology



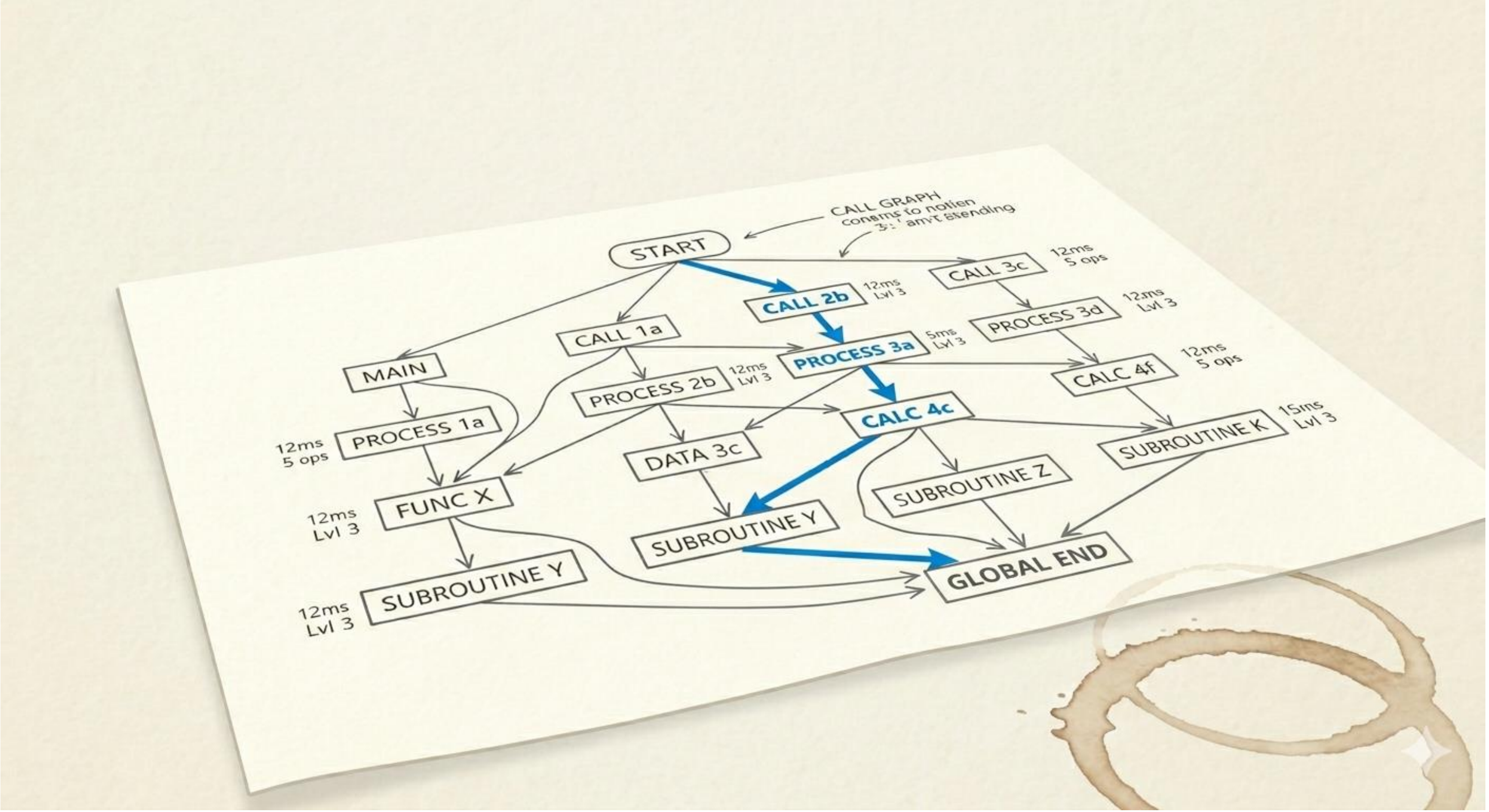
iPD, Li et al, ASPDAC2024

Physical Design Flow Parameters

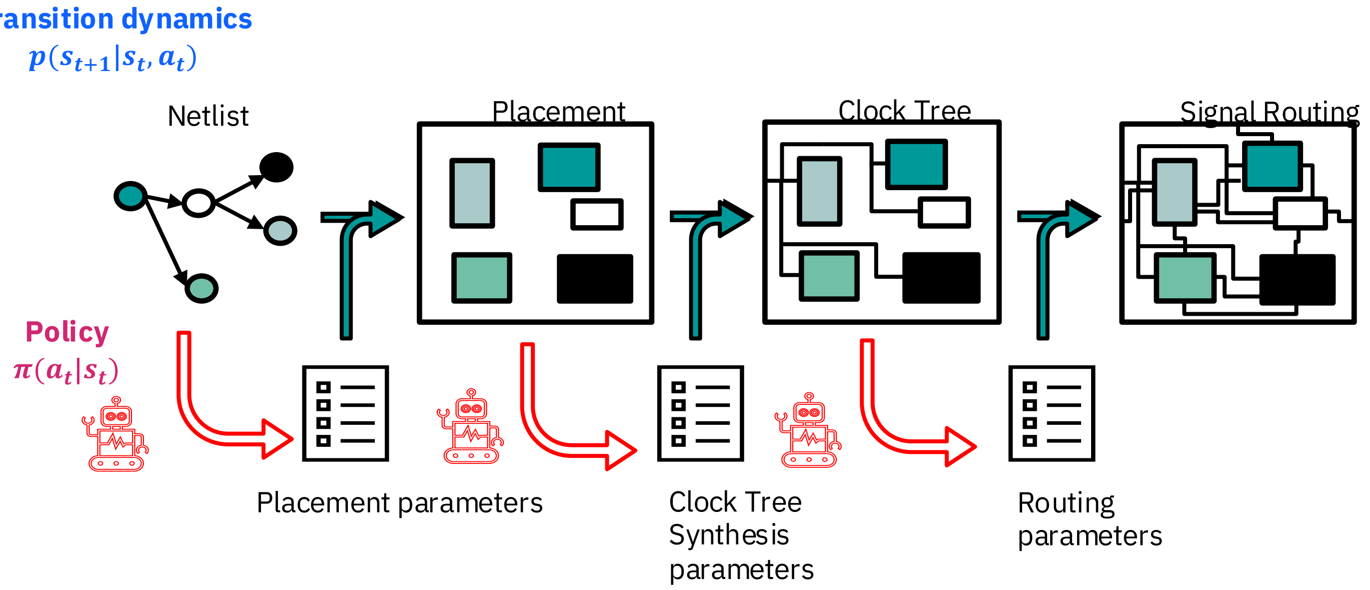


Parameters:
- Technology
- Cell Library characteristics
- IP characteristics
- EDA Tool Features and Flaws
- Design (under Construction) characteristics

Prompt Chaining



Physical Design and Reinforcement Learning



"Breakthrough reinforcement learning engines can explore trillions of design recipes. These models continue to train and accelerate convergence throughout the design cycle and bleed over to impact efficiency and productivity on iterative designs".

SEC filing?

Bleed over: the unintentional spread, or influence of something from one area, subject, or state into another, often causing it to overlap or affect the second area.

Reinforcement Learning

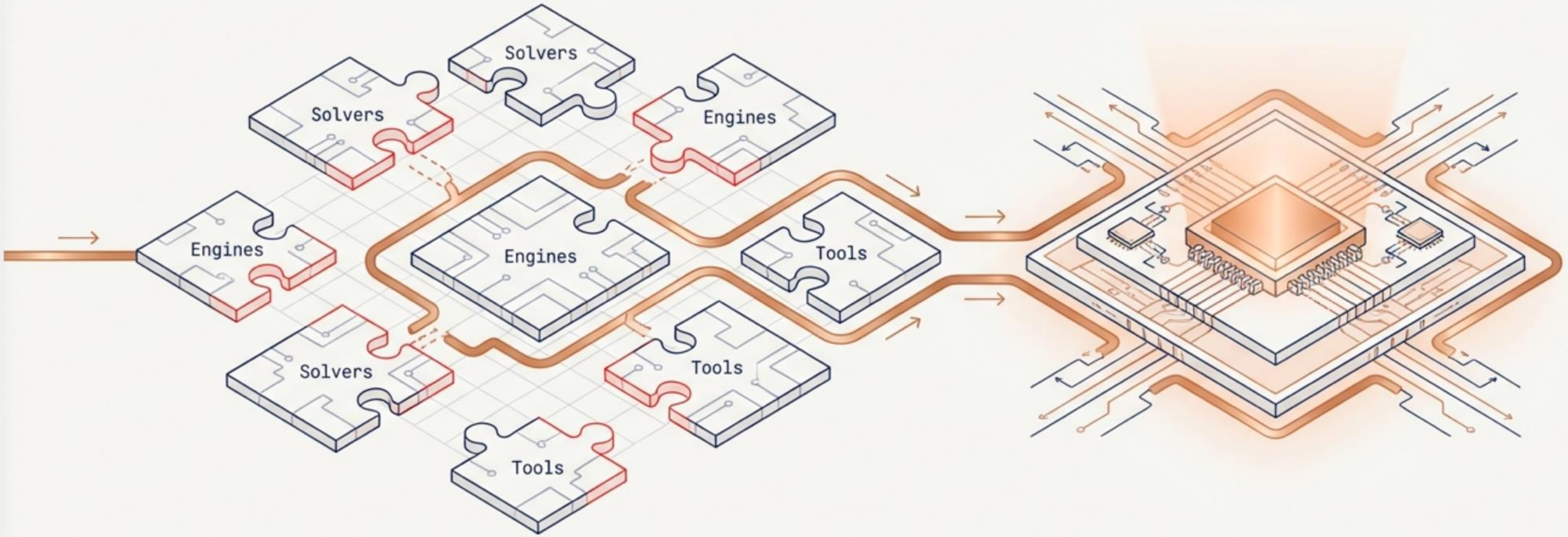
"RL agents tend to overfit to the specific netlist topology seen during training.

They memorize "solutions" rather than learning "placement rules," necessitating expensive re-training (hundreds of GPU-hours) for every new chip design."

The Dawn of Agentic EDA: A Survey of Autonomous Digital Chip Design" (arXiv:2512.23189).

From Stateless Tools to Sovereign Agents

Closing the Loop in Chip Physical Design through Agentic Infrastructure

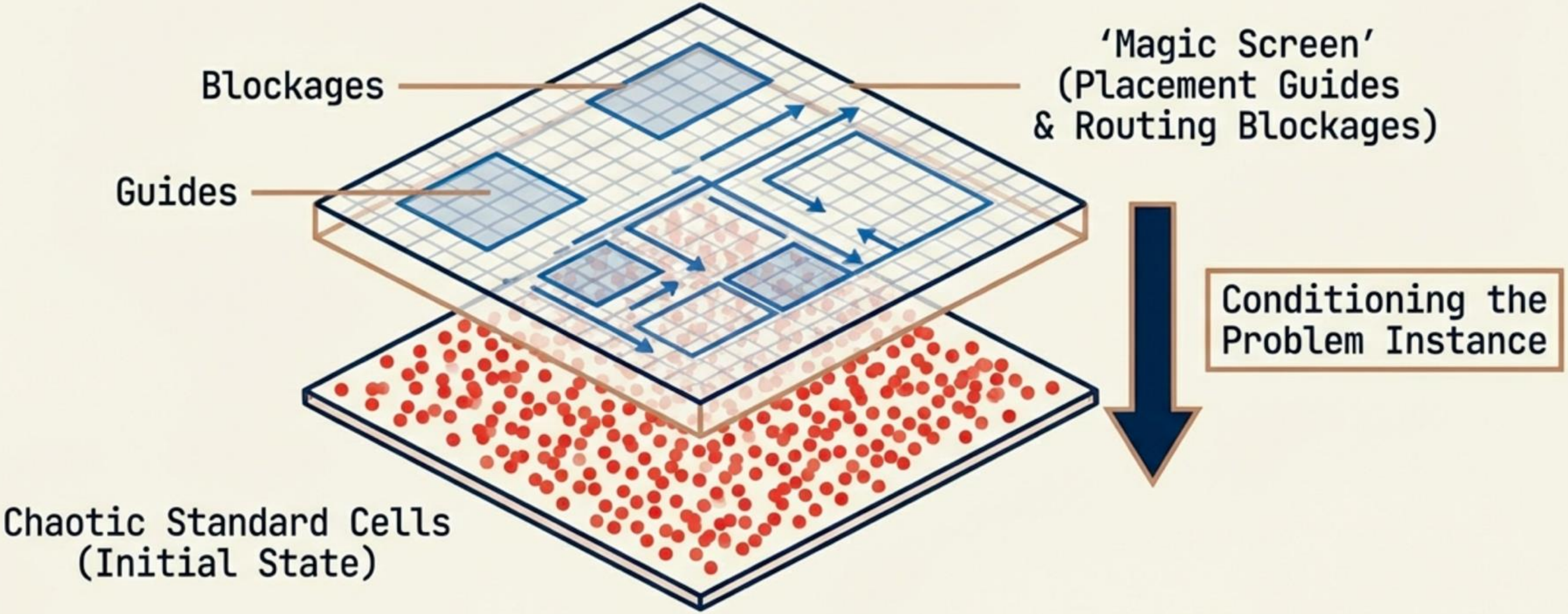


Synthesized from "Solvers, Engines, Tools and Flows" by Andrew B. Kahng | ISPD '24

iPD, Li et al, ASPDAC2024

NotebookLM

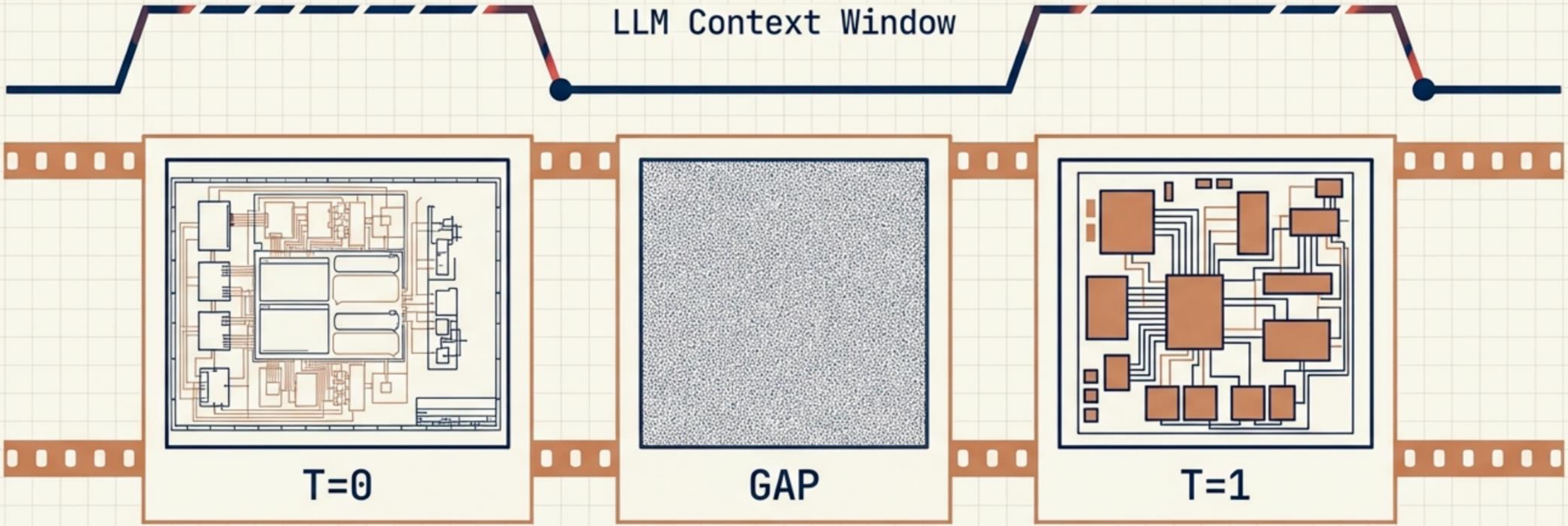
Managing the Interstices: 'Magic Screens'



Instead of predicting the final timing, the Agent generates "Magic Screens"—placement density guides and routing blockages—to steer the chaotic tools toward convergence.

NotebookLM

The Limitation of 'Snapshots'

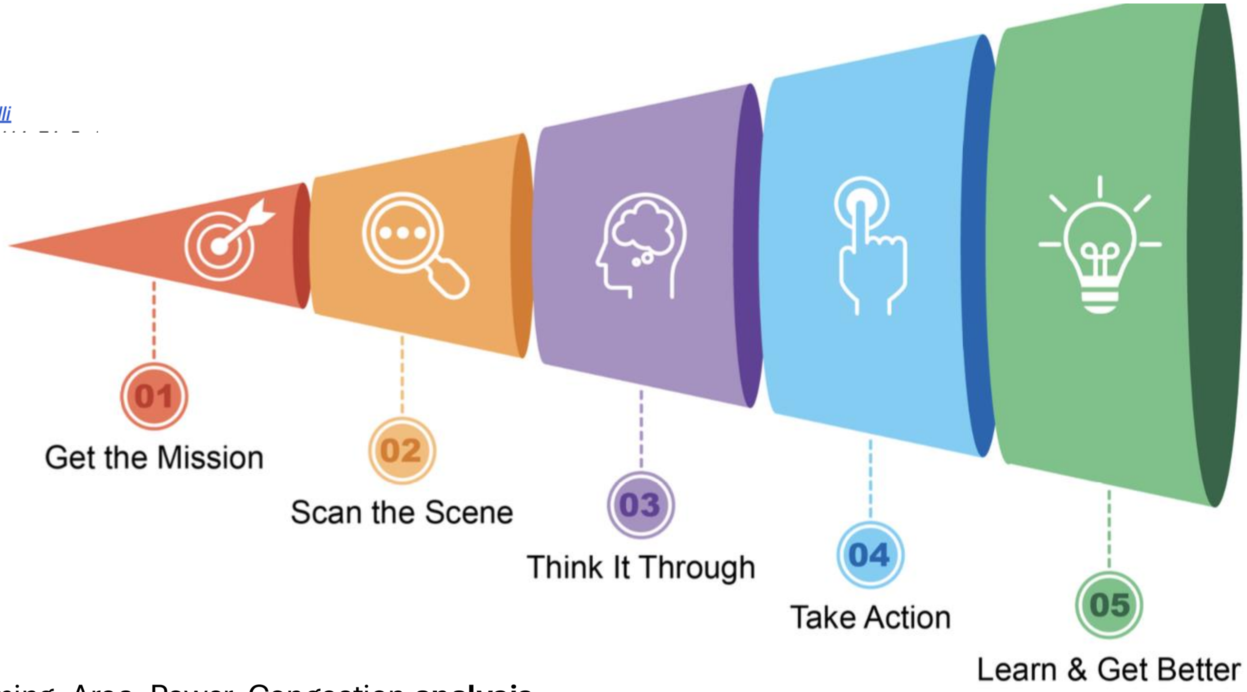


The Agent Gap: Current LLMs have amnesia. They function in isolated context windows, lacking the "Identity" to own a weeks-long optimization trajectory.

Agentic AI Problem Solving Process

Agentic Design Patterns

A Hands-On Guide to Building Intelligent Systems[1], Antonio Gulli



Get the mission: Get the best PPA

Scan the scene: Context Engineering: Timing, Area, Power, Congestion **analysis**.

Think it through: **Drivers** (critical, non-critical, congested, un-congested, highly-switching, DRCs)

Take Action: **Transformations** (size up, down, Vt swap, inflate cell footprint, buffer, (re)factor, (re)map, place, route)

Learn and Get better: ??? → Not memorizing solutions, but understanding actions and drivers.

Refactoring Physical Design

LLM timing calc	12.5	seconds
timing calculator	0.125	ms
Factor	1.00E-05	

Breakdown:

- Input processing: ~44K tokens in ~9-10 seconds = **~4,400 tokens/sec**
- Output generation: ~800 tokens in ~1-2 seconds = **~400-800 tokens/sec**

Please extract the path details for the following timing visualizer URL and then analyze the path and advise on opportunities to improve timing: <some TV URL>

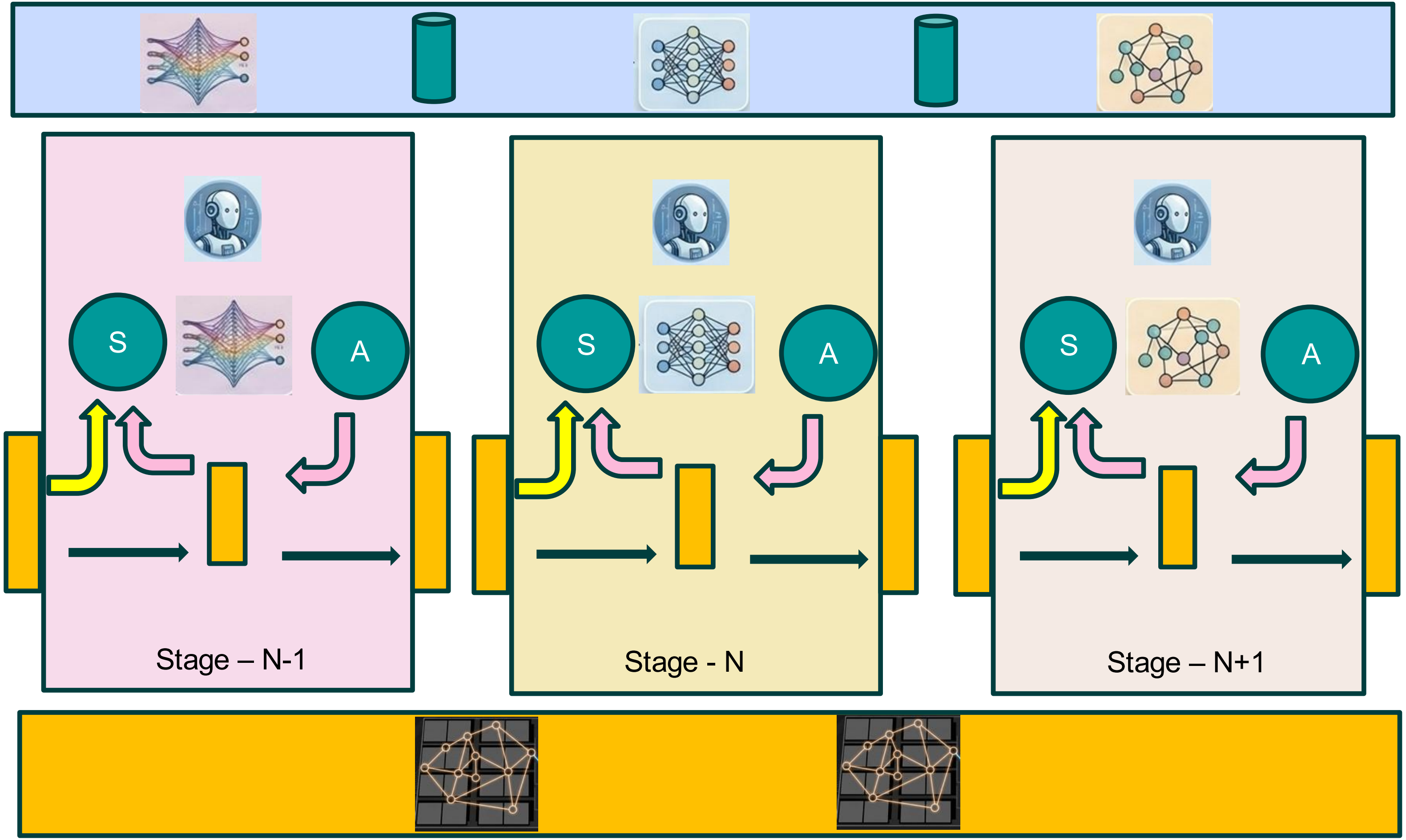
<u>If you're curious about what Bob came up with:</u>

```
Path analysis complete for endpoint XXX (slack: -7.93 ps).

**Critical Issue**: Super-critical convergence point at entry 4 with -55.34 ps slack (INFLUENCE_IN=614, 459 µm wire, 166.68 fF
load).

**Top Recommendations**:
1. Restructure logic cone at convergence point (40-50 ps gain)
2. Optimize 5-stage buffer chain to 3 stages (8-12 ps gain)
3. Address 64.8-69.3% density congestion zones (3-5 ps gain)
4. Add repeaters on long wires (459 µm, 11 µm segments) (5-8 ps gain)
5. Reduce high-resistance nets (6-9 Ω) using lower-resistance metal layers (2-4 ps gain)

**Expected Total Improvement**: 58-79 ps (path would become passing)
```

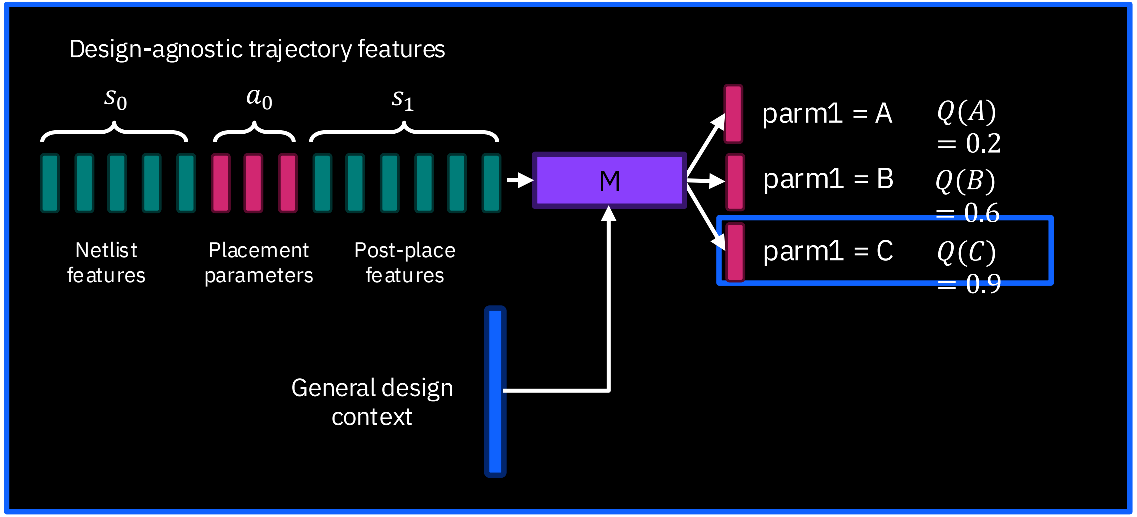
Principles to Refactor Physical Design Flow for the Agentic Era

- The Physical Design process needs to be refactored into stages that where each stage follows the following principles:
  - **Small** enough to have an agent and a model govern its work.
  - **Intelligent:** Uses specialized models for few-shot action execution.
    - E.g the model understands the cause and effect of actions in this stage.
  - **Reliable:** Delivers predictable results through chosen actions.
  - **Modular:** Features a clean state-transfer mechanism for stage sequencing.
    - Clean: well-identified, verifiable goals reached; stop propagation of errors.
  - **Global Persistence Store:** Integrated long-term memory for cross-stage context retention.

Refactored Physical Design Flow

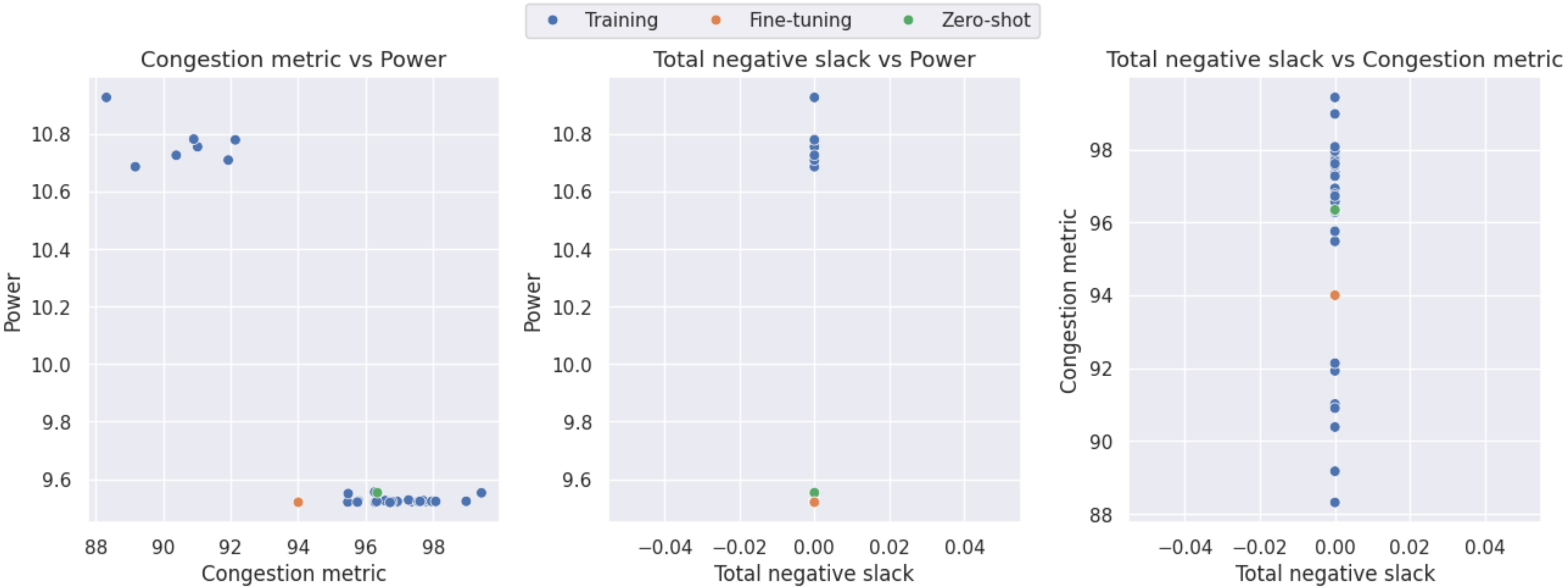


Stage Model

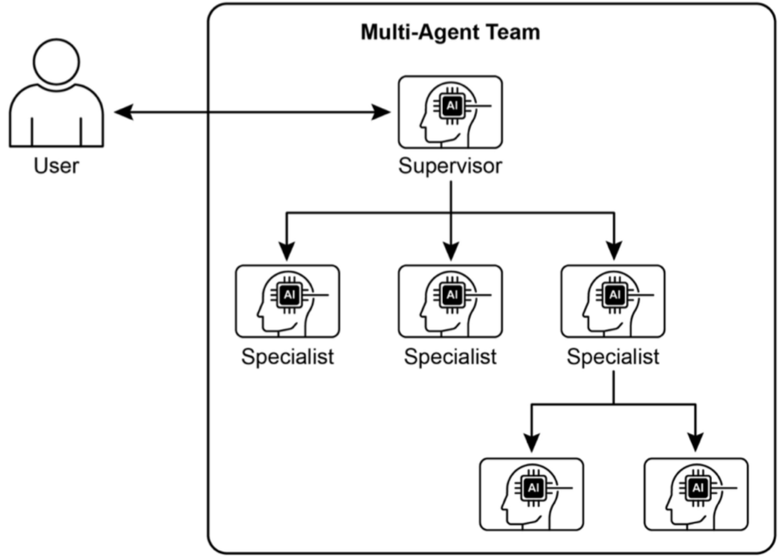


1. Sample design $p(d) \sim \mathrm{N}_{\mathrm{d}}^{\alpha}$
   - $\alpha = 0$: Fully-balanced
   - $\alpha = 0.5$: Square-root
   - $\alpha = 1$: Count-based
2. Sample trajectory $p(t|d)$ uniform

Timing and Power Optimization Stage



Stage Agent Orchestration



Stage Supervisor:
- Hands out the resources (time, area, power)
- Assigns what to work on
- Determines if goals of Stage is completed, e.g. a clean state is reached

Specialists
- Do the work, execute the actions.

How do we truly Boost Optimization Capability?

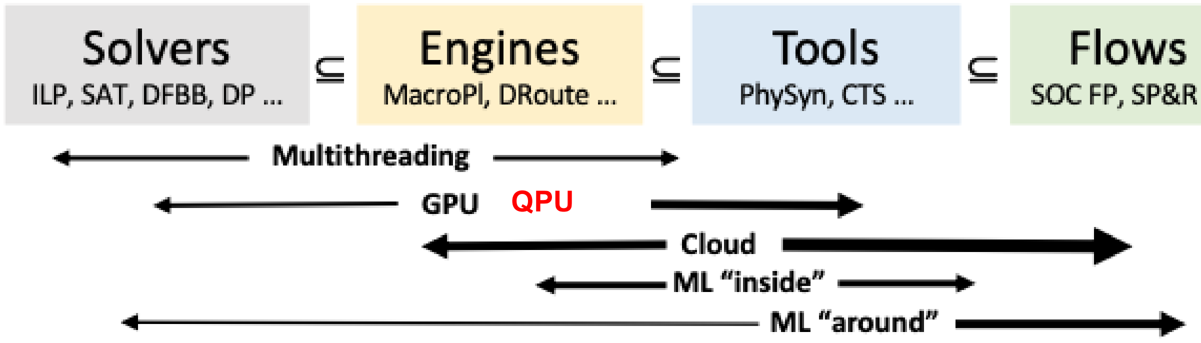
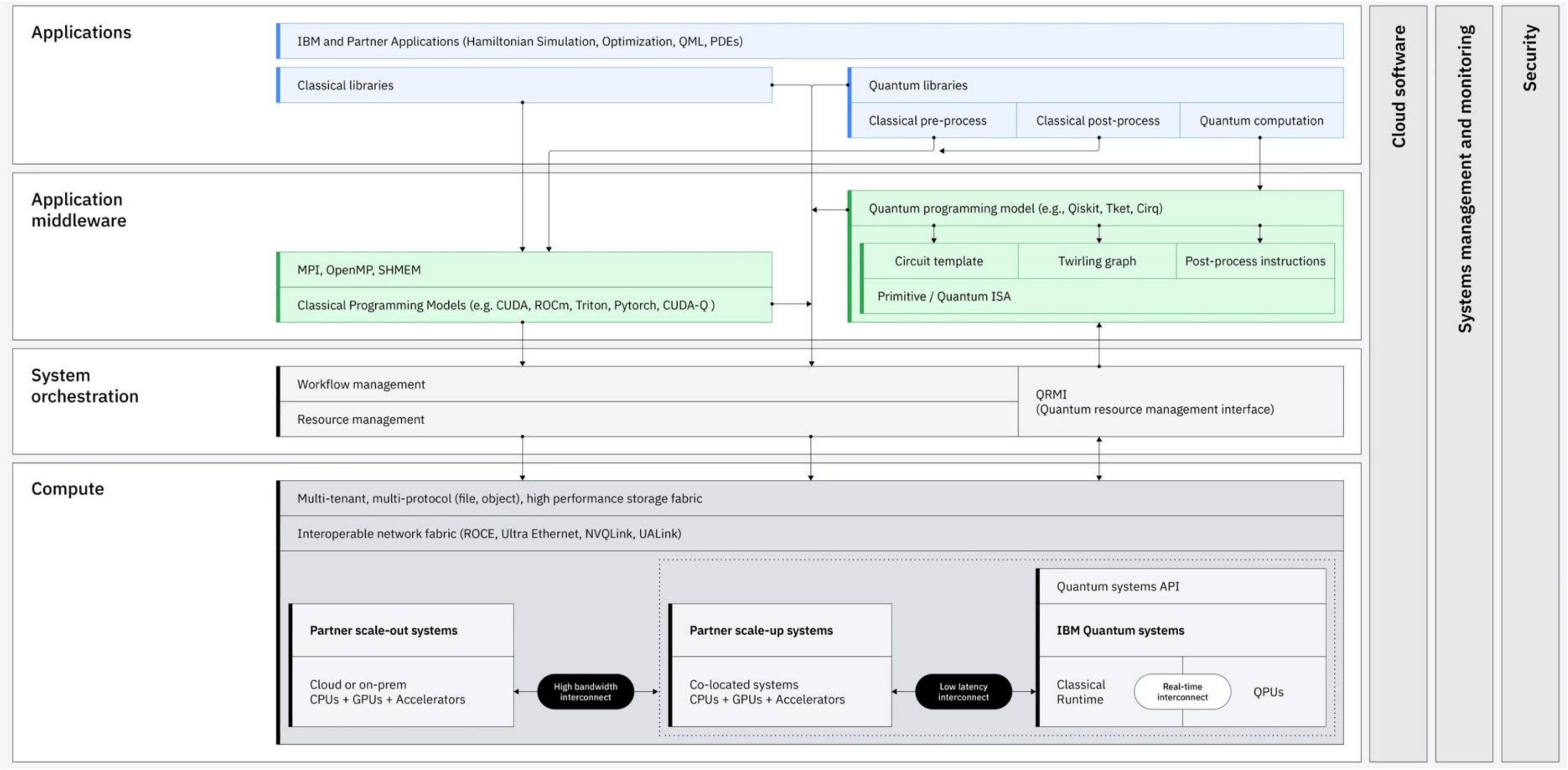


Figure 3: Impact of multithreading and GPU, cloud deployment, and AI/ML across the continuum of solvers, engines, tools and flows.

Solvers, Engines, Tools and Flows: The Next Wave for AI/ML in Physical Design

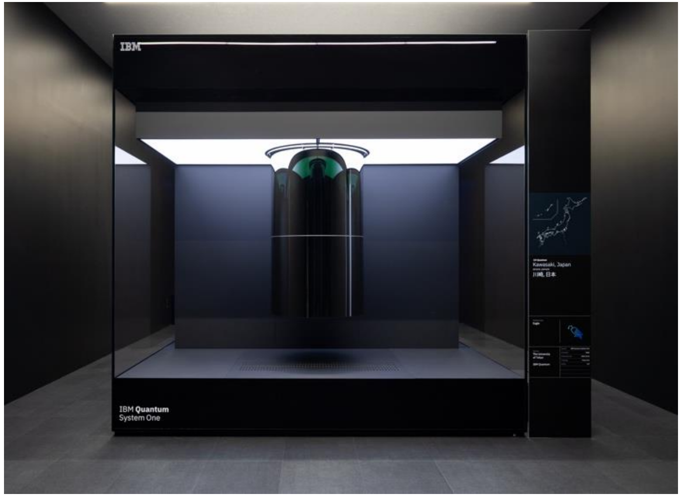
Quantum-centric supercomputing



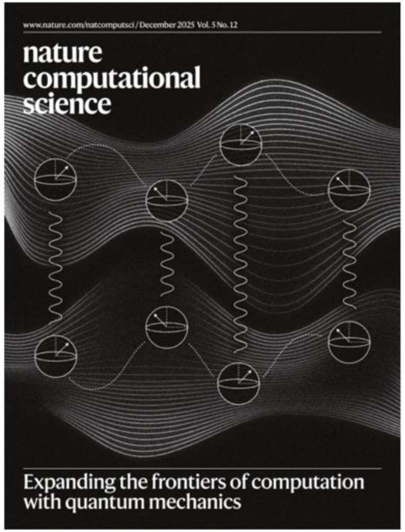
https://research.ibm.com/blog/quantum-centric-supercomputing-system-reference-architecture

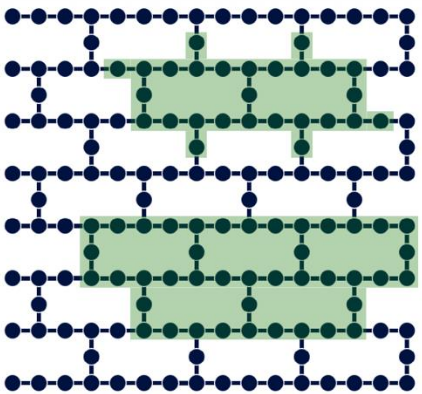
IBM has deployed 75+ systems since 2016.

Today, we have 13 utility-scale quantum computers (100+ qubits) operational in Poughkeepsie, NY; our European data center; and in client locations around the world.



MaxCut MOO on a quantum system

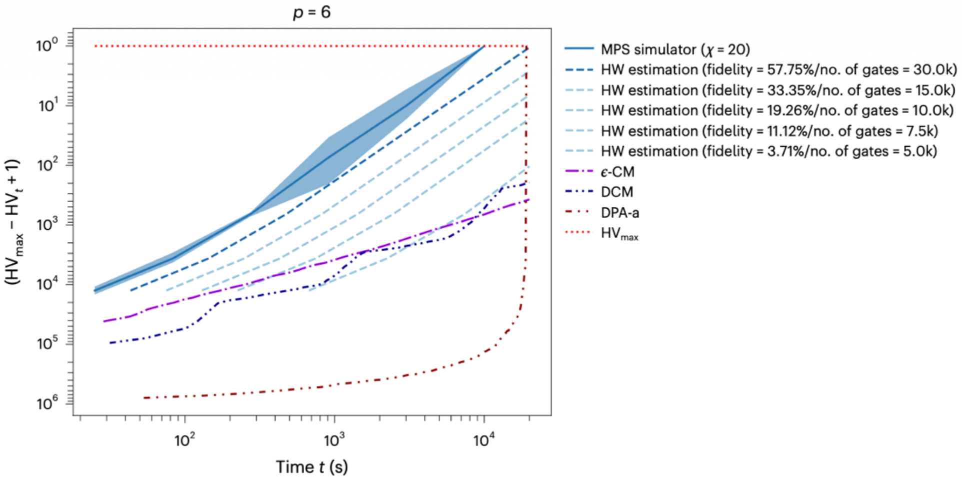




42/156 QUBITS

MOO can be difficult even if corresponding single-objective problems can be solved efficiently, particularly with an increasing number of objective functions. In this context, sampling-based approximate quantum optimization algorithms can be beneficial as they can quickly produce a large variety of good solutions to approximate the Pareto front.

Sampling is the most basic operation a quantum computer can do. All other quantum applications are ultimately derived from it. This means quantum computers are well-suited to tasks that require fast sampling and have a tolerance for approximate answers. In multi-objective optimization, that is exactly what we want.

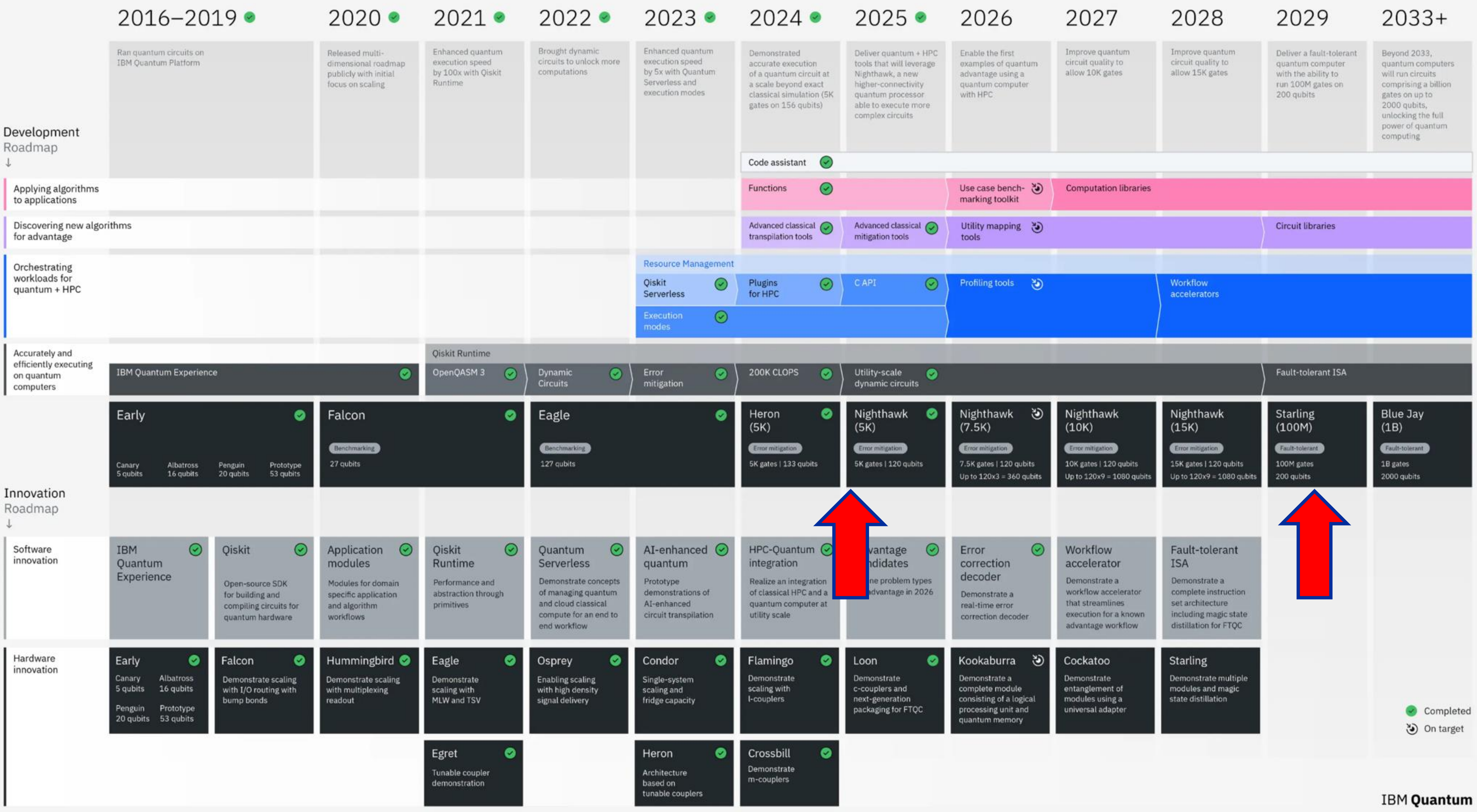


2026	2027	2028	2029	2033+
Nighthawk (7.5K)	Nighthawk (10K)	Nighthawk (15K)	Starling (100M)	Blue Jay (1B)
Error mitigation	Error mitigation	Error mitigation	Fault-tolerant	Fault-tolerant
7.5K gates 120 qubits	10K gates 120 qubits	15K gates 120 qubits	100M gates 200 logical qubits	1B gates 2000 logical qubits
Up to 120x3 = 360 qubits	Up to 120x9 = 1080 qubits	Up to 120x9 = 1080 qubits		

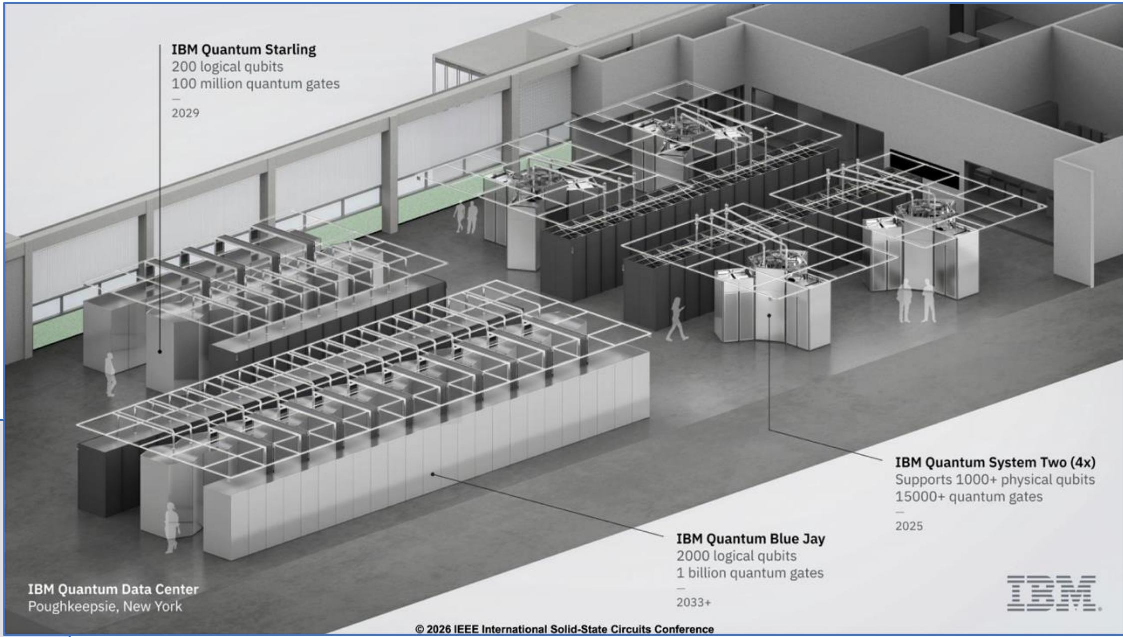
THANK YOU

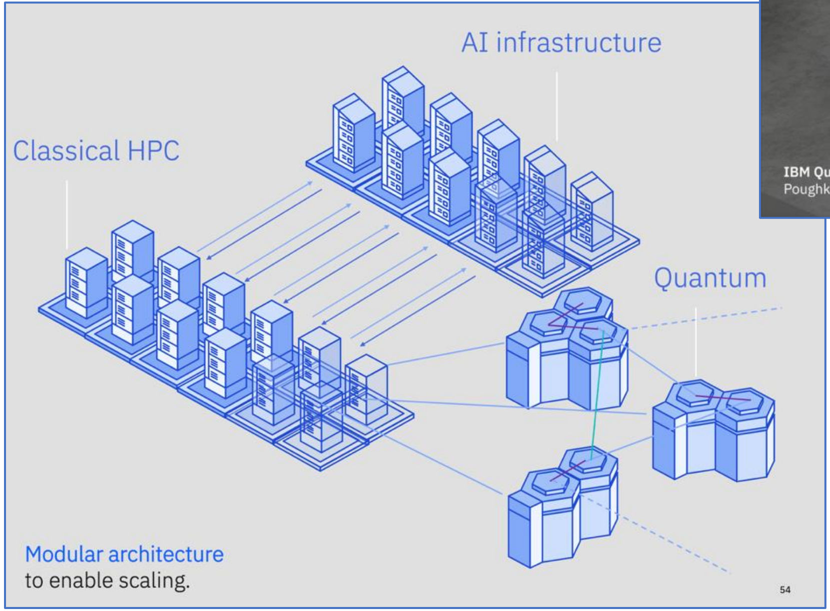
leonstok@us.ibm.com

IBM



2016–2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2033+
Development Roadmap
Ran quantum circuits on IBM Quantum Platform
Released multi-dimensional roadmap publicly with initial focus on scaling
Enhanced quantum execution speed by 100x with Qiskit Runtime
Brought dynamic circuits to unlock more computations
Enhanced quantum execution speed by 5x with Quantum Serverless and execution modes
Demonstrated accurate execution of a quantum circuit at a scale beyond exact classical simulation (5K gates on 156 qubits)
Deliver quantum + HPC tools that will leverage Nighthawk, a new higher-connectivity quantum processor able to execute more complex circuits
Enable the first examples of quantum advantage using a quantum computer with HPC
Improve quantum circuit quality to allow 10K gates
Improve quantum circuit quality to allow 15K gates
Deliver a fault-tolerant quantum computer with the ability to run 100M gates on 200 qubits
Beyond 2033, quantum computers will run circuits comprising a billion gates on up to 2000 qubits, unlocking the full power of quantum computing
Code assistant
Applying algorithms to applications
Functions
Use case bench-marking toolkit
Computation libraries
Discovering new algorithms for advantage
Advanced classical transpilation tools
Advanced classical mitigation tools
Utility mapping tools
Circuit libraries
Orchestrating workloads for quantum + HPC
Resource Management
Qiskit Serverless
Plugins for HPC
C API
Profiling tools
Workflow accelerators
Execution modes
Accurately and efficiently executing on quantum computers
Qiskit Runtime
IBM Quantum Experience
OpenQASM 3
Dynamic Circuits
Error mitigation
200K CLOPS
Utility-scale dynamic circuits
Fault-tolerant ISA
Early
Canary 5 qubits
Albatross 16 qubits
Penguin 20 qubits
Prototype 53 qubits
Falcon
Benchmarking
27 qubits
Eagle
Benchmarking
127 qubits
Heron (5K)
Error mitigation
5K gates | 133 qubits
Nighthawk (5K)
Error mitigation
5K gates | 120 qubits
Nighthawk (7.5K)
Error mitigation
7.5K gates | 120 qubits
Up to 120x3 = 360 qubits
Nighthawk (10K)
Error mitigation
10K gates | 120 qubits
Up to 120x9 = 1080 qubits
Nighthawk (15K)
Error mitigation
15K gates | 120 qubits
Up to 120x9 = 1080 qubits
Starling (100M)
Fault-tolerant
100M gates
200 qubits
Blue Jay (1B)
Fault-tolerant
1B gates
2000 qubits
Innovation Roadmap
Software innovation
IBM Quantum Experience
Qiskit
Open-source SDK for building and compiling circuits for quantum hardware
Application modules
Modules for domain specific application and algorithm workflows
Qiskit Runtime
Performance and abstraction through primitives
Quantum Serverless
Demonstrate concepts of managing quantum and cloud classical compute for an end to end workflow
AI-enhanced quantum
Prototype demonstrations of AI-enhanced circuit transpilation
HPC-Quantum integration
Realize an integration of classical HPC and a quantum computer at utility scale
vantage
ndidates
ne problem types
dvantage in 2026
Error correction decoder
Demonstrate a real-time error correction decoder
Workflow accelerator
Demonstrate a workflow accelerator that streamlines execution for a known advantage workflow
Fault-tolerant ISA
Demonstrate a complete instruction set architecture including magic state distillation for FTQC
Hardware innovation
Early
Canary 5 qubits
Albatross 16 qubits
Penguin 20 qubits
Prototype 53 qubits
Falcon
Demonstrate scaling with I/O routing with bump bonds
Hummingbird
Demonstrate scaling with multiplexing readout
Eagle
Demonstrate scaling with MLW and TSV
Osprey
Enabling scaling with high density signal delivery
Condor
Single-system scaling and fridge capacity
Flamingo
Demonstrate scaling with l-couplers
Loon
Demonstrate c-couplers and next-generation packaging for FTQC
Kookaburra
Demonstrate a complete module consisting of a logical processing unit and quantum memory
Cockatoo
Demonstrate entanglement of modules using a universal adapter
Starling
Demonstrate multiple modules and magic state distillation
Egret
Tunable coupler demonstration
Heron
Architecture based on tunable couplers
Crossbill
Demonstrate m-couplers
Completed
On target
IBM Quantum


IBM Quantum Starling
200 logical qubits
100 million quantum gates
2029
IBM Quantum System Two (4x)
Supports 1000+ physical qubits
15000+ quantum gates
2025
IBM Quantum Blue Jay
2000 logical qubits
1 billion quantum gates
2033+
IBM Quantum Data Center
Poughkeepsie, New York
© 2026 IEEE International Solid-State Circuits Conference


AI infrastructure
Classical HPC
Quantum
Modular architecture
to enable scaling.
54


2026
2027
2028
2029
2033+
Nighthawk (7.5K)
Error mitigation
7.5K gates
120 qubits
Up to 120x3 = 360 qubits
Nighthawk (10K)
Error mitigation
10K gates
120 qubits
Up to 120x9 = 1080 qubits
Nighthawk (15K)
Error mitigation
15K gates
120 qubits
Up to 120x9 = 1080 qubits
Starling (100M)
Fault-tolerant
100M gates
200 logical qubits
Blue Jay (1B)
Fault-tolerant
1B gates
2000 logical qubits